Suricata User Guide

Release 9.0.0-dev

OISF

Nov 07, 2026

CONTENTS

1 What is Suricata **3**
- 1.1 About the Open Information Security Foundation . . . 3

2 Quickstart guide **5**
- 2.1 Installation . . . 5
- 2.2 Basic setup . . . 5
- 2.3 Signatures . . . 6
- 2.4 Running Suricata . . . 6
- 2.5 Alerting . . . 7
- 2.6 EVE Json . . . 7

3 Installation **9**
- 3.1 Source . . . 9
- 3.2 Binary packages . . . 11
- 3.3 Advanced Installation . . . 16

4 Upgrading **17**
- 4.1 General instructions . . . 17
- 4.2 Upgrading to 9.0.0 . . . 17
- 4.3 Upgrading to 8.0.1 . . . 20
- 4.4 Upgrading 7.0 to 8.0 . . . 20
- 4.5 Upgrading 6.0 to 7.0 . . . 23
- 4.6 Upgrading 5.0 to 6.0 . . . 25
- 4.7 Upgrading 4.1 to 5.0 . . . 26

5 Security Considerations **27**
- 5.1 Running as a User Other Than Root . . . 27
- 5.2 Containers . . . 29

6 Support Status **31**
- 6.1 Levels of Support . . . 31
- 6.2 Distributions . . . 32
- 6.3 Architecture Support . . . 33

7 Command Line Options **35**
- 7.1 Unit Tests . . . 39

8 Suricata Rules **41**
- 8.1 Rules Format . . . 41
- 8.2 Meta Keywords . . . 49
- 8.3 Ethernet Keywords . . . 53

8.4 IP Keywords . . . 53
8.5 TCP keywords . . . 58
8.6 UDP keywords . . . 61
8.7 ICMP keywords . . . 62
8.8 IGMP keywords . . . 66
8.9 Payload Keywords . . . 66
8.10 Integer Keywords . . . 90
8.11 Transformations . . . 92
8.12 Prefiltering Keywords . . . 102
8.13 Flow Keywords . . . 104
8.14 Bypass Keyword . . . 113
8.15 HTTP Keywords . . . 113
8.16 File Keywords . . . 134
8.17 DNS Keywords . . . 138
8.18 mDNS Keywords . . . 142
8.19 LLMNR Keywords . . . 143
8.20 SSL/TLS Keywords . . . 144
8.21 SSH Keywords . . . 149
8.22 JA3/JA4 Keywords . . . 152
8.23 Modbus Keyword . . . 153
8.24 DCERPC Keywords . . . 156
8.25 DHCP keywords . . . 157
8.26 DNP3 Keywords . . . 158
8.27 ENIP/CIP Keywords . . . 161
8.28 FTP/FTP-DATA Keywords . . . 165
8.29 Kerberos Keywords . . . 169
8.30 SMB Keywords . . . 171
8.31 SNMP keywords . . . 173
8.32 NTP Keywords . . . 176
8.33 Base64 keywords . . . 177
8.34 SIP Keywords . . . 179
8.35 SDP Keywords . . . 183
8.36 SCTP Keywords . . . 188
8.37 RFB Keywords . . . 191
8.38 MQTT Keywords . . . 192
8.39 IKE Keywords . . . 196
8.40 HTTP2 Keywords . . . 199
8.41 Quic Keywords . . . 201
8.42 NFS Keywords . . . 202
8.43 SMTP Keywords . . . 203
8.44 WebSocket Keywords . . . 204
8.45 Generic App Layer Keywords . . . 205
8.46 Generic Decode Layer Keywords . . . 209
8.47 Xbits Keyword . . . 209
8.48 Alert Keywords . . . 211
8.49 Thresholding Keywords . . . 212
8.50 IP Reputation Keyword . . . 216
8.51 IP Addresses Match . . . 217
8.52 Config Rules . . . 217
8.53 Datasets . . . 218
8.54 Lua Scripting for Detection . . . 226
8.55 Differences From Snort . . . 229
8.56 Multiple Buffer Matching . . . 238
8.57 Tag . . . 241

8.58 VLAN Keywords . . . 243
8.59 LDAP Keywords . . . 245
8.60 PGSQL Keywords . . . 254
8.61 Email Keywords . . . 255
8.62 Rule Types and Categorization . . . 258
8.63 Rule Processing . . . 280

9 Rule Management **285**
9.1 Rule Management with Suricata-Update . . . 285
9.2 Adding Your Own Rules . . . 287
9.3 Rule Reloads . . . 288
9.4 Rules Profiling . . . 290

10 Making sense out of Alerts **291**

11 Performance **293**
11.1 Runmodes . . . 293
11.2 Packet Capture . . . 298
11.3 Tuning Considerations . . . 299
11.4 Hyperscan . . . 301
11.5 High Performance Configuration . . . 303
11.6 Statistics . . . 309
11.7 Ignoring Traffic . . . 311
11.8 Packet Profiling . . . 313
11.9 Rule Profiling . . . 313
11.10 Tcmalloc . . . 315
11.11 Performance Analysis . . . 315

12 Configuration **321**
12.1 Suricata.yaml . . . 321
12.2 Global-Thresholds . . . 386
12.3 Exception Policies . . . 389
12.4 Snort.conf to Suricata.yaml . . . 394
12.5 Multi Tenancy . . . 398
12.6 Dropping Privileges After Startup . . . 403
12.7 Using Landlock LSM . . . 404
12.8 systemd notification . . . 404
12.9 Includes . . . 405

13 Reputation **407**
13.1 IP Reputation . . . 407

14 Init Scripts **411**

15 Output **413**
15.1 EVE . . . 413
15.2 Lua Output . . . 498
15.3 Syslog Alerting Compatibility . . . 500
15.4 Custom tls logging . . . 501
15.5 Log Rotation . . . 502

16 Lua support **505**
16.1 Lua usage in Suricata . . . 505
16.2 Lua functions . . . 505
16.3 Lua Libraries . . . 508

17 File Extraction **547**
17.1 Architecture . . . 547
17.2 Settings . . . 547
17.3 Output . . . 548
17.4 Rules . . . 549
17.5 MD5 . . . 549
17.6 Updating Filestore Configuration . . . 552

18 Protocols **553**
18.1 App-Layer . . . 553
18.2 Further Reading . . . 554

19 Public Datasets (PCAPs) **555**

20 Using Capture Hardware **557**
20.1 AF_PACKET . . . 557
20.2 Endace DAG . . . 557
20.3 Napatech . . . 558
20.4 Myricom . . . 566
20.5 eBPF and XDP . . . 568
20.6 Netmap . . . 578
20.7 AF_XDP . . . 582
20.8 DPDK . . . 586
20.9 PCAP File Reading . . . 589

21 Interacting via Unix Socket **593**
21.1 Introduction . . . 593
21.2 Commands in standard running mode . . . 593
21.3 Commands on the cmd prompt . . . 595
21.4 PCAP processing mode . . . 595
21.5 Build your own client . . . 596

22 Plugins **599**
22.1 nDPI . . . 599

23 IPS Mode **601**
23.1 IPS Concept . . . 601
23.2 Setting up IPS/inline for Linux . . . 602
23.3 Setting up IPS/inline for Windows . . . 610

24 Firewall Mode **613**
24.1 Firewall Mode Design . . . 613
24.2 Firewall Ruleset Examples . . . 620
24.3 Firewall Mode Stats . . . 624
24.4 Migrating from the old TLS state names . . . 624
24.5 Migrating from the old SSH state names . . . 627
24.6 Firewall Mode Unsupported Keywords . . . 628

25 3rd Party Integration **631**
25.1 Symantec SSL Visibility (BlueCoat) . . . 631

26 Man Pages **633**
26.1 Suricata . . . 633
26.2 Suricata Socket Control . . . 639
26.3 Suricata Control . . . 642

26.4 Suricata Control Filestore . . . 642

27 Acknowledgements **645**

28 Licenses **649**
28.1 GNU General Public License . . . 649
28.2 Creative Commons Attribution-NonCommercial 4.0 International Public License . . . 653
28.3 Suricata Source Code . . . 657
28.4 Suricata Documentation . . . 657

29 Suricata Developer Guide **659**
29.1 Working with the Codebase . . . 659
29.2 Contributing . . . 681
29.3 Suricata Internals . . . 692
29.4 Extending Suricata . . . 697
29.5 LibSuricata and Plugins . . . 736
29.6 Upgrading . . . 737

30 Verifying Suricata Source Distribution Files **739**
30.1 Verification Steps . . . 739

31 Appendix **741**
31.1 EVE JSON Schema . . . 741
31.2 EVE Index . . . 742

Bibliography **863**

Index **865**

This is the documentation for Suricata 9.0.0-dev.

CHAPTER

ONE

WHAT IS SURICATA

Suricata is a high performance Network IDS, IPS and Network Security Monitoring engine. It is open source and owned by a community-run non-profit foundation, the Open Information Security Foundation (OISF). Suricata is developed by the OISF.

1.1 About the Open Information Security Foundation

The Open Information Security Foundation is a non-profit foundation organized to build community and to support open-source security technologies like Suricata, the world-class IDS/IPS engine.

1.1.1 License

The Suricata source code is licensed under version 2 of the *GNU General Public License*.

This documentation is licensed under the *Creative Commons Attribution-NonCommercial 4.0 International Public License*.

CHAPTER

TWO

QUICKSTART GUIDE

This guide will give you a quick start to run Suricata and will focus only on the basics. For more details, read through the more specific chapters.

2.1 Installation

It's assumed that you run a recent Ubuntu release as the official PPA can then be used for the installation. To install the latest stable Suricata version, follow the steps:

```
sudo apt-get install software-properties-common
sudo add-apt-repository ppa:oisf/suricata-stable
sudo apt update
sudo apt install suricata jq
```

The dedicated PPA repository is added, and after updating the index, Suricata can be installed. We recommend installing the `jq` tool at this time as it will help with displaying information from Suricata's EVE JSON output (described later in this guide).

For the installation on other systems or to use specific compile options see *Installation*.

After installing Suricata, you can check which version of Suricata you have running and with what options, as well as the service state:

```
sudo suricata --build-info
sudo systemctl status suricata
```

2.2 Basic setup

First, determine the interface(s) and IP address(es) on which Suricata should be inspecting network packets:

```
$ ip addr

2: enp1s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group␣
↪default qlen 1000
link/ether 00:11:22:33:44:55 brd ff:ff:ff:ff:ff:ff
inet 10.0.0.23/24 brd 10.23.0.255 scope global noprefixroute enp1s0
```

Use that information to configure Suricata:

```
sudo vim /etc/suricata/suricata.yaml
```

> **Note**
>
> The location of `suricata.yaml` depends on how Suricata was installed. Distribution packages (such as the PPA used above) install it to `/etc/suricata/suricata.yaml`. A build from source installs it under the prefix passed to `configure`, which defaults to `/usr/local`, so the file is at `/usr/local/etc/suricata/suricata.yaml` instead. See *Installation* for details.

There are many possible configuration options, we focus on the setup of the `HOME_NET` variable and the network interface configuration. The `HOME_NET` variable should include, in most scenarios, the IP address of the monitored interface and all the local networks in use. The default already includes the RFC 1918 networks. In this example `10.0.0.23` is already included within `10.0.0.0/8`. If no other networks are used the other predefined values can be removed.

In this example the interface name is `enp1s0` so the interface name in the `af-packet` section needs to match. An example interface config might look like this:

Capture settings:

```
af-packet:
    - interface: enp1s0
      cluster-id: 99
      cluster-type: cluster_flow
      defrag: yes
      tpacket-v3: yes
```

This configuration uses the most recent recommended settings for the IDS runmode for basic setups. There are many of possible configuration options which are described in dedicated chapters and are especially relevant for high performance setups.

2.3 Signatures

Suricata uses Signatures to trigger alerts so it's necessary to install those and keep them updated. Signatures are also called rules, thus the name *rule-files*. With the tool `suricata-update` rules can be fetched, updated and managed to be provided for Suricata.

In this guide we just run the default mode which fetches the ET Open ruleset:

```
sudo suricata-update
```

Afterwards the rules are installed at `/var/lib/suricata/rules` which is also the default at the config and uses the sole `suricata.rules` file.

2.4 Running Suricata

With the rules installed, Suricata can run properly and thus we restart it:

```
sudo systemctl restart suricata
```

To make sure Suricata is running check the Suricata log:

```
sudo tail /var/log/suricata/suricata.log
```

The last line will be similar to this:

```
<Notice> - all 4 packet processing threads, 4 management threads initialized, engine␣
↪started.
```

The actual thread count will depend on the system and the configuration.

To see statistics, check the `stats.log` file:

```
sudo tail -f /var/log/suricata/stats.log
```

By default, it is updated every 8 seconds to show updated values with the current state, like how many packets have been processed and what type of traffic was decoded.

2.5 Alerting

To test the IDS functionality of Suricata it's best to test with a signature. The signature with ID `2100498` from the ET Open ruleset is written specific for such test cases.

alert ip any any -> any any (msg:"GPL ATTACK_RESPONSE id check returned root"; content:"uid=0|28|root|29|"; classtype:bad-unknown; sid:2100498; rev:7; metadata:created_at 2010_09_23, updated_at 2010_09_23;)

The syntax and logic behind those signatures is covered in other chapters. This will alert on any IP traffic that has the content within its payload. This rule can be triggered quite easy. Before we trigger it, start `tail` to see updates to `fast.log`.

Rule trigger:

```
sudo tail -f /var/log/suricata/fast.log
curl http://testmynids.org/uid/index.html
```

The following output should now be seen in the log:

```
[1:2100498:7] GPL ATTACK_RESPONSE id check returned root [**] [Classification:␣
↪Potentially Bad Traffic] [Priority: 2] {TCP} 217.160.0.187:80 -> 10.0.0.23:41618
```

This should include the timestamp and the IP of your system.

2.6 EVE Json

The more advanced output is the EVE JSON output which is explained in detail in *Eve JSON Output*. To see what this looks like it's recommended to use `jq` to parse the JSON output.

Alerts:

```
sudo tail -f /var/log/suricata/eve.json | jq 'select(.event_type=="alert")'
```

This will display more detail about each alert, including meta-data.

Stats:

```
sudo tail -f /var/log/suricata/eve.json | jq 'select(.event_type=="stats")|.stats.
↪capture.kernel_packets'
sudo tail -f /var/log/suricata/eve.json | jq 'select(.event_type=="stats")'
```

The first example displays the number of packets captured by the kernel; the second examples shows all of the statistics.

CHAPTER

THREE

INSTALLATION

Before Suricata can be used it has to be installed. Suricata can be installed on various distributions using binary packages: *Binary packages*.

For people familiar with compiling their own software, the *Source method* is recommended.

Advanced users can check the advanced guides, see *Advanced Installation*.

3.1 Source

Installing from the source distribution files gives the most control over the Suricata installation.

The Suricata source distribution files should be verified before building the source, see *Verifying Suricata Source Distribution Files*.

Basic steps using the Suricata 8.0.3 release as example:

```
tar xzvf suricata-8.0.3.tar.gz
cd suricata-8.0.3
./configure
make
make install
```

This will install Suricata into `/usr/local/bin/`, use the default configuration in `/usr/local/etc/suricata/` and will output to `/usr/local/var/log/suricata`

3.1.1 Common configure options

`--disable-gccmarch-native`

Do not optimize the binary for the hardware it is built on. Add this flag if the binary is meant to be portable or if Suricata is to be used in a VM.

`--prefix``=/usr/`

Installs the Suricata binary into /usr/bin/. Default `/usr/local/`

`--sysconfdir``=/etc`

Installs the Suricata configuration files into /etc/suricata/. Default `/usr/local/etc/`

`--localstatedir``=/var`

Setups Suricata for logging into /var/log/suricata/. Default `/usr/local/var/log/suricata`

`--enable-geoip`

Enables GeoIP support for detection.

`--enable-dpdk`
: Enables DPDK packet capture method.

3.1.2 Dependencies and compilation

Debian 13+

> **Note**
>
> The following instructions require sudo to be installed.

Listing 1: Minimal dependencies for Debian

```
sudo apt -y install autoconf automake build-essential cargo \
    libjansson-dev libpcap-dev libpcre2-dev libtool \
    libyaml-dev make pkg-config rustc zlib1g-dev
```

Ubuntu

> **Note**
>
> The following instructions require sudo to be installed.

Listing 2: Minimal dependencies for Ubuntu

```
sudo apt -y install autoconf automake build-essential cargo-1.89 \
    libjansson-dev libpcap-dev libpcre2-dev libtool \
    libyaml-dev make pkg-config zlib1g-dev
export PATH=/usr/lib/rust-1.89/bin:$PATH
```

CentOS, AlmaLinux, RockyLinux, Fedora, etc

> **Note**
>
> The following instructions require sudo to be installed.

To install all minimal dependencies, it is required to enable extra package repository in most distros. You can enable it possibly by one of the following ways:

```
sudo dnf -y update
sudo dnf -y install epel-release dnf-plugins-core
# AlmaLinux 8 / RockyLinux 8
sudo dnf config-manager --set-enabled powertools
# AlmaLinux 9 / RockyLinux 9
sudo dnf config-manager --set-enable crb
# Oracle Linux 8
sudo dnf config-manager --set-enable ol8_codeready_builder
# Oracle Linux 9
sudo dnf config-manager --set-enable ol9_codeready_builder
```

Listing 3: Minimal dependencies for RPM-based distributions

```
sudo dnf install -y cargo gcc jansson-devel libpcap-devel \
    libyaml-devel make pcre2-devel zlib-devel
```

Windows

For building and installing from source on Windows, see install/windows.

Compilation

Follow these steps from your Suricata directory:

```
./configure # you may want to add additional parameters here
# ./configure --help to get all available parameters
# j is for adding concurrency to make; the number indicates how much
# concurrency so choose a number that is suitable for your build system
make -j8
make install # to install your Suricata compiled binary
# make install-full - installs configuration and rulesets as well
```

Rust support

Rust packages can be found in package managers but some distributions don't provide Rust or provide outdated Rust packages. In case of insufficient version you can install Rust directly from the Rust project itself at https://rust-lang.org/tools/install/.

3.1.3 Auto-Setup

You can also use the available auto-setup features of Suricata:

```
./configure && make && sudo make install-conf
```

make install-conf would do the regular "make install" and then it would automatically create/setup all the necessary directories and `suricata.yaml` for you.

```
./configure && make && sudo make install-rules
```

make install-rules would do the regular "make install" and then it would automatically download and set up the latest ruleset from Emerging Threats available for Suricata.

```
./configure && make && sudo make install-full
```

make install-full would combine everything mentioned above (install-conf and install-rules) and will present you with a ready-to-run (configured and set-up) Suricata.

3.2 Binary packages

3.2.1 Ubuntu Package Installation

For Ubuntu, the OISF maintains a Personal Package Archive (PPA) `suricata-stable` that always contains the latest stable release.

> **Note**
>
> The following instructions require sudo to be installed.

Setup to install the latest stable Suricata:

```
sudo apt-get install software-properties-common
sudo add-apt-repository ppa:oisf/suricata-stable
sudo apt-get update
```

Then, you can install the latest stable with:

```
sudo apt-get install suricata
```

After installing you can proceed to the *Basic setup*.

OISF launchpad: suricata-stable.

Upgrading

To upgrade:

```
sudo apt-get update
sudo apt-get upgrade suricata
```

Remove

To remove Suricata from your system:

```
sudo apt-get remove suricata
```

Getting Debug or Pre-release Versions

> **Note**
>
> The following instructions require sudo to be installed.

If you want Suricata with built-in (enabled) debugging, you can install the debug package:

```
sudo apt-get install suricata-dbg
```

If you would like to help test the Release Candidate (RC) packages, the same procedures apply, just using another PPA: `suricata-beta`:

```
sudo add-apt-repository ppa:oisf/suricata-beta
sudo apt-get update
sudo apt-get upgrade
```

You can use both the suricata-stable and suricata-beta repositories together. Suricata will then always be the latest release, stable or beta.

OISF launchpad: suricata-beta.

Daily Releases

> **Note**
>
> The following instructions require `sudo` to be installed.

If you would like to help test the daily build packages from our latest git(dev) repository, the same procedures as above apply, just using another PPA, `suricata-daily`:

```
sudo add-apt-repository ppa:oisf/suricata-daily-allarch
sudo apt-get update
sudo apt-get upgrade
```

> **Note**
>
> Please have in mind that this is packaged from our latest development git main and is therefore potentially unstable.
>
> We do our best to make others aware of continuing development and items within the engine that are not yet complete or optimal. With this in mind, please refer to Suricata's issue tracker on Redmine for an up-to-date list of what we are working on, planned roadmap, and to report issues.

OISF launchpad: suricata-daily.

After Installation

After installing you can proceed to the *Basic setup*.

3.2.2 Debian Package Installation

Suricata is available in the official Debian repositories for Debian 9 (stretch) and later versions.

> **Note**
>
> The following instructions require `sudo` to be installed.

In Debian 9 (stretch) and later do:

```
sudo apt-get install suricata
```

In the "stable" version of Debian, Suricata is usually not available in the latest version. A more recent version is often available from Debian backports, if it can be built there.

To use backports, the backports repository for the current stable distribution needs to be added to the system-wide sources list. For Debian 10 (buster), for instance, run the following as `root`:

```
echo "deb http://http.debian.net/debian buster-backports main" > \
    /etc/apt/sources.list.d/backports.list
apt-get update
apt-get install suricata -t buster-backports
```

After Installation

After installing you can proceed to the *Basic setup*.

3.2.3 RPM Installation

Using the Fedora COPR system, the OISF provides Suricata packages for Fedora, Red Hat Enterprise Linux, and Enterprise Linux rebuilds.

The benefit of using the OISF maintained COPR package repositories is that the OISF maintains packages for all non-EOL Suricata versions for each distribution version. For example, the OISF maintains Suricata 7 and Suricata 8 packages for RHEL 9 and 10.

Installing From Package Repositories

> **Note**
>
> Instructions in the following sections require `sudo` to be installed.

Enterprise Linux and Rebuilds

```
sudo dnf install epel-release dnf-plugins-core
sudo dnf copr enable @oisf/suricata-8.0
sudo dnf install suricata
```

Fedora

```
sudo dnf install dnf-plugins-core
sudo dnf copr enable @oisf/suricata-8.0
sudo dnf install suricata
```

Additional Notes for RPM Installations

- Suricata is pre-configured to run as the `suricata` user.
- Command line parameters such as providing the interface names can be configured in `/etc/sysconfig/suricata`.
- Users can run `suricata-update` without being root provided they are added to the `suricata` group.
- Directories:
  - `/etc/suricata`: Configuration directory
  - `/var/log/suricata`: Log directory
  - `/var/lib/suricata`: State directory rules, datasets.

Starting Suricata On-Boot

The Suricata RPMs are configured to run from Systemd.

> **Note**
>
> The following instructions require sudo to be installed.

To start Suricata:

```
sudo systemctl start suricata
```

To stop Suricata:

```
sudo systemctl stop suricata
```

To have Suricata start on-boot:

```
sudo systemctl enable suricata
```

To reload rules:

```
sudo systemctl reload suricata
```

After Installation

After installing you can proceed to the *Basic setup*.

3.2.4 Other Package Installations

Suricata can be found in the package managers for many other operating systems and distributions, but it is important to note that these are not created or supported by the OISF and the Suricata development team.

Arch Based

The ArchLinux AUR contains Suricata and suricata-nfqueue packages, with commonly used configurations for compilation (may also be edited to your liking). You may use makepkg, yay (sample below), or other AUR helpers to compile and build Suricata packages.

```
yay -S suricata
```

After Installation

After installing you can proceed to the *Basic setup*.

Suricata is available on various distributions as binary packages. These offer a convenient way to install and manage Suricata without compiling from source.

For Ubuntu systems:

See *Ubuntu Package Installation* for detailed instructions on installing from PPA repositories.

For Debian systems:

See *Debian Package Installation* for detailed instructions on installing from official repositories and backports.

For RPM-based distributions (CentOS, AlmaLinux, RockyLinux, Fedora, etc):

See *RPM Installation* for detailed instructions on installing from COPR repositories.

For other distributions:

See *Other Package Installations* for installation instructions for Arch Linux and other distributions.

3.3 Advanced Installation

If you are using Ubuntu, you can follow *Installation from GIT*.

For other various installation guides for installing from GIT and for other operating systems, please check (bear in mind that those may be somewhat outdated): https://redmine.openinfosecfoundation.org/projects/suricata/wiki/Suricata_Installation

CHAPTER

FOUR

UPGRADING

4.1 General instructions

Suricata can be upgraded by simply installing the new version to the same locations as the already installed version. When installing from source, this means passing the same `--prefix`, `--sysconfdir`, `--localstatedir` and `--datadir` options to `configure`.

```
$ suricata --build-info|grep -A 3 '\-\-prefix'
    --prefix                                    /usr
    --sysconfdir                                /etc
    --localstatedir                             /var
    --datarootdir                               /usr/share
```

4.1.1 Configuration Updates

New versions of Suricata will occasionally include updated config files: `classification.config` and `reference.config`. Since the Suricata installation will not overwrite these if they exist, they must be manually updated. If there are no local modifications they can simply be overwritten by the ones Suricata supplies.

Major updates include new features, new default settings and often also remove features. This upgrade guide covers the changes that might have an impact of migrating from an older version and keeping the config. We encourage you to also check all the new features that have been added but are not covered by this guide. Those features are either not enabled by default or require dedicated new configuration.

4.2 Upgrading to 9.0.0

4.2.1 Major Changes

- Encrypted quic traffic bypass is now independently controlled through `app-layer.protocols.quic.encryption-handling` setting. The setting can either be `bypass`, `track-only` or `full`.
- Default value for `stream.reassembly.depth` when the value is not specified in suricata.yaml is now 1 MiB instead of 0/unlimited.
- Default value for `app-layer.protocols.ftp.memcap` when the value is not specified in suricata.yaml is now 64 MiB instead of 0/unlimited. The limit also covers FTP-DATA file buffering. An invalid value also keeps the 64 MiB default. Set it to `0` to remove the limit.
- LLMNR protocol parser, logger and sticky buffers are implemented.
- `pgsql` is no longer enabled by default. Previously, if the `app-layer.protocols.pgsql` section was absent from suricata.yaml, the parser would be enabled. It's now disabled by default. Simply enabling its EVE output will no longer suffice, either.

- The TLS state names used in firewall rules (`accept:hook tls:<state>`, `alert tls:<state>`) and in the `firewall.policies.app.tls` config keys changed from completion milestones to active phases (e.g. `client_hello_done` becomes `client_hello`, `client_in_progress` becomes `client_started`). A rule with an old state name fails to load and is **not enforced**, and a config key that matches a renamed state is reported at load and aborts init when `engine.init-failure-fatal` is set (a warning otherwise). The certificate phase states (`client_cert`, `server_cert`) are entered when the hello completes, before any certificate data exists: migrated certificate content rules must be paired with a plain accept at the cert state (or a drop-based cert check plus that accept). See *Migrating from the old TLS state names* for the old-to-new mapping.
- The firewall's per-packet evaluation of app-layer states is now bounded by the state the packet's parse progress reaches. This applies to every app-layer protocol, not only the renamed TLS states: a state above the packet's current progress can no longer decide the packet (for example, a default policy configured at a later state no longer drops a packet that stops at an earlier state), and on a rule no-match the states between the last rule and the packet's progress are now evaluated. Existing rulesets that relied on a policy at a later state to drop packets in an earlier state need to move that policy to the state the packet actually reaches.
- A ClientHello that carries no SNI extension is decided at the certificate state (the state the hello-completing record reaches) instead of at the hello-completion milestone: it matches no `tls.sni` rule (the buffer only exists once the message is fully parsed) and, without an accept at the certificate state, falls on the implicit default policy (drop:flow) - the same drop the base release gave it at the milestone. A plain accept at the certificate state (the remedy for migrated certificate rules) changes that: it also decides the hello-completing record, so a SNI-less hello is accepted by it. If your ruleset accepts SNI-less hellos, that is a relaxation over the base behaviour - pair the certificate-state accept with an explicit rule that catches the SNI-less case if you want to keep the drop (the reference SNI example in *Firewall Ruleset Examples* keeps the drop by having no certificate-state accept).

4.2.2 Logging Changes

- The ssh eve record now carries the per-direction fields `ssh.client.error` / `ssh.client.state` and `ssh.server.error` / `ssh.server.state` (`invalid_banner` or `invalid_record`, and the state the failure occurred in - the state field is present only alongside the error) for flows that hit an unrecoverable ssh parse error; a failed direction is logged even when it parsed no banner. The object is logged at the failing delivery, carrying the error and the state; successful flows are logged at the next EOF-flush delivery.
- The format of IKEv1 proposal attributes has been changed to handle duplicate attribute types. See IKE logging changes
- Ethertype values (`ether.ether_type`) are now logged matching the network order value. E.g., previously, `ether_type` values were logged in host order; an ethertype value of `0xfbb7` (network order) was logged as *47099* ` (`0xb7fb`). This ethertype value will be logged as `64439`.
- Alert verdict key is changed from to `reject-target` to `reject_target`
- App-layer stats protocols names replace dash by underscore, meaning `stats.app_layer.*.ftp-data` becomes `stats.app_layer.*.ftp_data`, and same for bittorrent_dht
- MQTT user properties are now logged as a new object called `user_properties` as an array of key-value pairs like `[{"key":"mykey", "value":"myvalue"}]` under `properties` object, instead of as `{key: value}` pairs as a part of the `properties` object itself.
- The `ts_progress`/`tc_progress` values in alerts for the renamed TLS firewall states now use the new phase names (e.g. `client_hello_done` is now `client_hello`). See *Migrating from the old TLS state names* for the old-to-new mapping.
- NFS eve records are now logged in the flow direction, like SMB, HTTP and DNS records: a record completed on a reply carries the client as `src_ip`/`src_port` where 8.x logged the packet's addresses. Filters that matched responses on `src_port 2049` need `dest_port 2049` now.

4.2.3 Removals

- The deprecated `http-log` output has been removed. Use `eve-log` with the `http` event type for HTTP logging.
- flowbits `toggle` command has been deprecated.

4.2.4 Keyword Changes

- HTTP2 keywords have now better progress defined, with the http2 transaction progress being split per direction. This means that some rules should match sooner, some rules will have less false negatives, and some rules will trigger once per transaction instead of twice (one time for each direction)
- The SSH app-layer state hooks have been reworked and renamed: the hooks are now `request_banner`, `request_kex` and `request_session` (and the `response_` equivalents); rules using the old names fail to load. The completion state is never reported, so `ssh:request_complete` and the `request-complete` policy key match nothing for ssh except disrupted flows (depth truncation or async reassembly) - session-admit rules or policies no longer fire, review them before upgrading. Failed key exchanges are reported through the `ssh.invalid_banner` / `ssh.invalid_record` app-layer events and the new `error` / `state` fields of the ssh eve record. See *SSH Keywords* for the full state and failure semantics.

4.2.5 Default Changes

- An absent configuration value for `stream.reassembly.urgent.policy` is now treated as `oob` instead of `inline`.
- An absent configuration value for `stream.reassembly.urgent.oob-limit-policy` is now treated as `drop` instead of `inline`.

4.2.6 Other Changes

- `engine-analysis` output has been changed for keywords that now use the generic integers framework
  - icmp_id
  - tcp.ack
  - tcp.seq
  - tcp.window
- `dnp3` has reduced the maximum number of open transactions from 500 down to 32, the maximum number of points per message from unbounded to 16384, and the maximum number of objects per message from unbounded to 2048. Configuration options, `max-tx`, `max-points`, and `max-objects` have been added for users who may need to change these defaults.
- Hyperscan caching (*detect.sgh-mpm-caching*), when enabled, prunes cache files that have not been used in the last 7 days by default. See *Hyperscan caching configuration* for more information.
- The `no` option in `app-layer.protocols.ssh.hassh` is now really enforced and there will be no hassh computation even if rules try to use it.
- Any inconsistent protocol enable/disable settings will issue a warning and would favor ipproto specific settings over protocol's global enable/disable settings. e.g. `app-layer.protocols.sip.tcp.enabled` would be read and preferred over `app-layer.protocols.sip.enabled`.
- *alert pkthdr* is now only available for decoder event rules. Previously it acted like *alert ip*.
- `ldap` has bound the maximum number of responses per transaction to 1024 by default.
- `http2` default value for max-streams is now 256 (instead of 4096).

- `ftp` keeps at most 16 preliminary (`1xx`) responses per transaction by default, set with `max-preliminary-responses`. Past the limit, the `ftp.too_many_preliminary_responses` event is raised. See *FTP*.

4.2.7 Changes for Library Users and Plugin Developers

- The built-in library run-mode has been removed. If your application is relying on this run-mode, it should implement its own application specific run-mode. See the example, `examples/lib/custom` for an example. For more details, see https://redmine.openinfosecfoundation.org/issues/8259.

4.3 Upgrading to 8.0.1

4.3.1 Major changes

- Various expected PPP packet types will no longer be marked as Unsupported Protocol when in a PPPOE packet.
- Added Cisco Discovery Protocol Control Protocol as a valid PPP packet.

4.3.2 Keyword changes

- Usage of multiple `tls.cert_subject` in a rule will print a warning as this keyword was not and is not implemented as a multi-buffer.

4.4 Upgrading 7.0 to 8.0

> **Note**
>
> `stats.whitelist` has been renamed to `stats.score` in `eve.json`

4.4.1 Major changes

- SIP parser has been updated to inspect traffic carried by TCP as well. SIP keywords can still match on their respective fields in addition to these improvements. Transactions are logged with the same schema regardless of which transport protocol is carrying the payload. Also, SIP protocol is detected using pattern matching and not only probing parser.
- `SIP_PORTS` variable has been introduced in suricata.yaml
- Application layer's `sip` counter has been split into `sip_tcp` and `sip_udp` for the `stats` event.
- Stats counters that are 0 can now be hidden from EVE logs. Default behavior still logs those (see *EVE Output - Stats* for configuration setting).
- SDP parser, logger and sticky buffers have been introduced. Due to SDP being encapsulated within other protocols, such as SIP, they cannot be directly enabled or disabled. Instead, both the SDP parser and logger depend on being invoked by another parser (or logger).
- ARP decoder and logger have been introduced. Since ARP can be quite verbose and produce many events, the logger is disabled by default.
- It is possible to see an increase of alerts, for the same rule-sets, if you use many stream/payload rules, due to Suricata triggering TCP stream reassembly earlier.
- New transform `from_base64` that base64 decodes a buffer and passes the decoded buffer. It's recommended that `from_base64` be used instead of `base64_decode`

- Datasets of type String now include the length of the strings to determine if the memcap value is reached. This may lead to memcaps being hit for older setups that didn't take that into account. For more details, check https://redmine.openinfosecfoundation.org/issues/3910
- DNS logging has been modified to be more consistent across requests, responses and alerts. See DNS Logging Changes for 8.0.
- PF_RING support has been moved to a plugin. See PF_RING plugin.
- LDAP parser and logger have been introduced.
- The following sticky buffers for matching SIP headers have been implemented:
  - sip.via
  - sip.from
  - sip.to
  - sip.content_type
  - sip.content_length
- Napatech support has been moved to a capture plugin. See Napatech plugin.
- Unknown requirements in the `requires` keyword will now be treated as unmet requirements, causing the rule to not be loaded. See *requires*.
- The configuration setting controlling stream checksum checks no longer affects checksum keyword validation. In Suricata 7.0, when `stream.checksum-validation` was set to `no`, the checksum keywords (e.g., `ipv4-csum`, `tcpv4-csum`, etc) will always consider it valid; e.g., `tcpv4-csum: invalid` will never match. In Suricata 8.0, `stream.checksum-validation` no longer affects the checksum rule keywords. E.g., `ipv4-csum: valid` will only match if the check sum is valid, even when engine checksum validations are disabled.
- Lua detection scripts (rules) now run in a sandboxed environment. See *Lua Scripting for Detection*. Lua rules are now also enabled by default.
- Lua output scripts have no default module search path, a search path will need to be set before external modules can be loaded. See the new default configuration file or *YAML* for more details.
- If the configuration value `ftp.memcap` is invalid, Suricata will set it to `0` which means no limit will be placed. In previous Suricata releases, Suricata would terminate execution. A warning message will be displayed *Invalid value <value> for ftp.memcap* when this occurs.
- The utility applications `suricatasc` and `suricatactl` have been rewritten in Rust. For most end-users this is a transparent change, however if you run these tools from the source directory, patch them or use them as Python modules your workflows may need to be adapted.
- Datasets now have a default max limit for hashsize of 65536. This is configurable via the `datasets.limits` options.
- For detect inspection recursion limits, if no value is provided, the default is now set to 3000.
- AF_PACKET now has better defaults:
  - AF_PACKET will now default to defrag off for inline mode with `cluster_flow` as its not recommended for inline use. However it can still be enabled with the `defrag` configuration parameter.
  - AF_PACKET will now default to tpacket-v3 for non-inline modes, it remains disabled for inline modes. To keep tpacket-v2 for non-inline modes, the existing `tpacket-v3` configuration parameter can be set to `false`.
  - The AF_PACKET default block size for both TPACKET_V2 and TPACKET_V3 has been increased from 32k to 128k. This is to allow for full size defragmented packets. For TPACKET_V3 the existing `block-size` parameter can be used to change this back to the old default of 32768 if needed. For

TPACKET_V2 a new configuration parameter has been added, `v2-block-size` which can be used to tune this value for TPACKET_V2. Due to the increased block size, memory usage has been increased, but should not be an issue in most cases.

- DPDK interface settings can now be configured automatically by setting `auto` to `mempool-size`, `mempool-cache-size`, `rx-descriptors`, `tx-descriptors`. See *Automatic interface configuration*.
- DPDK interface mempools are now allocated per thread instead of per port. This change improves performance and should not be visible from the user configuration perspective.
- DPDK supports link state check, allowing Suricata to start only when the link is up. This is especially useful for Intel E810 (ice) NICs as they need a few seconds before they are ready to receive packets. With this check disabled, Suricata reports as started but only begins processing packets after the previously mentioned interval. Other cards were not observed to have this issue. This feature is disabled by default. See *Link State Change timeout*.
- Encrypted traffic bypass has been decoupled from stream.bypass setting. This means that encrypted traffic can be bypassed while tracking/fully inspecting other traffic as well.
- Encrypted SSH traffic bypass is now independently controlled through `app-layer.protocols.ssh.encryption-handling` setting. The setting can either be `bypass`, `track-only` or `full`. To retain the previous behavior of encrypted traffic bypass combined with stream depth bypass, set `app-layer.protocols.ssh.encryption-handling` to `bypass` (while also setting `app-layer.protocols.tls.encryption-handling` to `bypass` and `stream.bypass` to `true`).
- Spaces are accepted in HTTP1 URIs instead of in the protocol version. That is: *GET /a b HTTP/1.1* gets now URI as */a b* and protocol as *HTTP/1.1* when it used to be URI as */a* and protocol as *b HTTP/1.1*
- The configuration structure of `threading.cpu-affinity` has been changed from a list format to a dictionary format. Additionally, member properties of **-cpu-set* nodes have been moved one level up. The support for list items such as - *worker-cpu-set*, - *management-cpu-set*, etc. is still supported. To convert to the new configuration format follow the example below or the description in *Threading*.

```
  threading:
    cpu-affinity:
-     - worker-cpu-set:
-         cpu: [0, 1]
+     worker-cpu-set:
+       cpu: [0, 1]
```

- All applayer protocols except FTP and HTTP now trigger inspection upon completion of a request/response in the respective direction. This means that earlier a content that matched just because it fell in the inspection chunk without wholly belonging to any one request/response may not match any longer.

4.4.2 Removals

- The ssh keywords `ssh.protoversion` and `ssh.softwareversion` have been removed.
- The detect engine stats counters for non-mpm-prefiltered rules `fnonmpm_list` and `nonmpm_list` were not in use since Suricata 8.0.0 and **were thus removed in 8.0.1.**

4.4.3 Deprecations

- The `http-log` output is now deprecated and will be removed in Suricata 9.0.
- The `tls-log` output is now deprecated and will be removed in Suricata 9.0.
- The `syslog` output is now deprecated and will be removed in Suricata 9.0. Note that this is the standalone `syslog` output and does **not** affect the `eve` outputs ability to send to syslog.

- The `default` option in `app-layer.protocols.tls.encryption-handling` is now deprecated and will be removed in Suricata 9.0. The `track-only` option should be used instead.

4.4.4 Keyword changes

- `ja3.hash` and `ja3s.hash` no longer accept contents with non hexadecimal characters, as they will never match.

4.4.5 Logging changes

- RFB security result is now consistently logged as `security_result` when it was sometimes logged with a dash instead of an underscore.
- Application layer metadata is logged with alerts by default **only for rules that use application layer keywords**. For other rules, the configuration parameter `detect.guess-applayer-tx` can be used to force the detect engine to guess a transaction, which is not guaranteed to be the one you expect. **In this case, the engine will NOT log any transaction metadata if there is more than one live transaction, to reduce the chances of logging unrelated data.** This may lead to what looks like a regression in behavior, but it is a considered choice.

4.4.6 Other Changes

- libhtp has been replaced with a rust version. This means libhtp is no longer built and linked as a shared library, and the libhtp dependency is now built directly into suricata.

4.5 Upgrading 6.0 to 7.0

4.5.1 Major changes

- Upgrade of PCRE1 to PCRE2. See *Changes from PCRE1 to PCRE2* for more details.
- IPS users: by default various new "exception policies" are set to DROP traffic. Please see *Exception Policies* for details on the settings and their scope. For trouble shooting, please check My traffic gets blocked after upgrading to Suricata 7.
- New protocols enabled by default: bittorrent-dht, quic, http2.
- The telnet protocol is also enabled by default, but only for the `app-layer`.

4.5.2 Security changes

- suricata.yaml now prevents process creation by Suricata by default with *security.limit-noproc*. The suricata.yaml configuration file needs to be updated to enable this feature. For more info, see *Configuration hardening*.
- Absolute filenames and filenames containing parent directory traversal are no longer allowed by default for datasets when the filename is specified as part of a rule. See *Datasets Security* and *Datasets File Locations* for more information.
- Lua rules are now disabled by default (change also introduced in 6.0.13), see *Lua Scripting for Detection*.

4.5.3 Removals

- The libprelude output plugin has been removed.
- EVE DNS v1 logging support has been removed. If still using EVE DNS v1 logging, see the manual section on DNS logging configuration for the current configuration options: *DNS EVE Configuration*

4.5.4 Logging changes

- IKEv2 Eve logging changed, the event_type has become `ike` which covers both protocol versions. The fields `errors` and `notify` have moved to `ike.ikev2.errors` and `ike.ikev2.notify`.
- FTP DATA metadata for alerts are now logged in `ftp_data` instead of root.
- Alert `xff` field is now logged as `alert.xff` for alerts instead of at the root.
- Protocol values and their names are built into Suricata instead of using the system's `/etc/protocols` file. Some names and casing may have changed in the values `proto` in `eve.json` log entries and other logs containing protocol names and values. See https://redmine.openinfosecfoundation.org/issues/4267 for more information.
- Logging of additional HTTP headers configured through the EVE `http.custom` option will now be logged in the `request_headers` and/or `response_headers` respectively instead of merged into the existing `http` object. In Suricata 6.0, a configuration like:

```
http:
  custom: [Server]
```

  would result in a log entry like:

```
"http": {
  "hostname": "suricata.io",
  "http_method": "GET",
  "protocol": "HTTP/1/1",
  "server": "nginx",
  ...
}
```

  This merging of custom headers in the `http` object could result in custom headers overwriting standard fields in the `http` object, or a response header overwriting request header.

  To prevent the possibility of fields being overwritten, **all** custom headers are now logged into the `request_headers` and `response_headers` arrays to avoid any chance of collision. This also facilitates the logging of headers that may appear multiple times, with each occurrence being logged in future releases (see note below).

  While these arrays are not new in Suricata 7.0, they had previously been used exclusively for the `dump-all-headers` option.

  As of Suricata 7.0, the above configuration example will now be logged like:

```
"http": {
  "hostname": "suricata.io",
  "http_method": "GET",
  "protocol": "HTTP/1/1",
  "response_headers": [
    { "name": "Server", "value": "nginx" }
  ]
}
```

  Effectively making the `custom` option a subset of the `dump-all-headers` option.

  If you've been using the `custom` option, this may represent a breaking change. However, if you haven't used it, there will be no change in the output.

> **Note**
>
> Currently, if the same HTTP header is seen multiple times, the values are concatenated into a comma-separated value.
>
> For more information, refer to: https://redmine.openinfosecfoundation.org/issues/1275.

- Engine logging/output now uses separate defaults for `console` and `file`, to provide a cleaner output on the console.

  Defaults are:

  - `console`: `%D: %S: %M`
  - `file`: `[%i - %m] %z %d:  %S: %M`

  The `console` output also changes based on verbosity level.

4.5.5 Deprecations

- Multiple "include" fields in the configuration file will now issue a warning and in Suricata 8.0 will not be supported. See *Includes* for documentation on including multiple files.
- For AF-Packet, the *cluster_rollover* setting is no longer supported. Configuration settings using `cluster_rollover` will cause a warning message and act as though *cluster_flow* ` was specified. Please update your configuration settings.

4.5.6 Other changes

- Experimental keyword *http2.header* is removed. *http.header*, *http.request_header*, and *http.response_header* are to be used.
- NSS is no longer required. File hashing and JA3 can now be used without the NSS compile time dependency.
- If installing Suricata without the bundled Suricata-Update, the `default-rule-path` has been changed from `/etc/suricata/rules` to `/var/lib/suricata/rules` to be consistent with Suricata when installed with Suricata-Update.
- FTP has been updated with a maximum command request and response line length of 4096 bytes. To change the default see *FTP*.
- SWF decompression in http has been disabled by default. To change the default see *Configure HTTP (libhtp)*. Users with configurations from previous releases may want to modify their config to match the new default. See https://redmine.openinfosecfoundation.org/issues/5632 for more information.
- The new option *livedev* is enabled by default with *use-for-tracking* being set to *true*. This should be disabled if multiple live devices are used to capture traffic from the same network.

4.6 Upgrading 5.0 to 6.0

- SIP now enabled by default
- RDP now enabled by default
- ERSPAN Type I enabled by default.

4.6.1 Major changes

- New protocols enabled by default: mqtt, rfb
- SSH Client fingerprinting for SSH clients
- Conditional logging
- Initial HTTP/2 support
- DCERPC logging
- Improved EVE logging performance

4.6.2 Removals

- File-store v1 has been removed. If using file extraction, the file-store configuration will need to be updated to version 2. See *Update File-store v1 Configuration to V2*.
- Individual Eve (JSON) loggers have been removed. For example, `stats-json`, `dns-json`, etc. Use multiple Eve logger instances if this behavior is still required. See *Multiple Logger Instances*.
- Unified2 has been removed. See unified2-removed.

4.6.3 Performance

- In YAML files w/o a *flow-timeouts.tcp.closed* setting, the default went from 0 to 10 seconds. This may lead to higher than expected TCP memory use: https://redmine.openinfosecfoundation.org/issues/6552

4.7 Upgrading 4.1 to 5.0

4.7.1 Major changes

- New protocols enabled by default: snmp (new config only)
- New protocols disabled by default: rdp, sip
- New defaults for protocols: nfs, smb, tftp, krb5 ntp are all enabled by default (new config only)
- VXLAN decoder enabled by default. To disable, set `decoder.vxlan.enabled` to `false`.
- HTTP LZMA support enabled by default. To disable, set `lzma-enabled` to `false` in each of the `libhtp` configurations in use.
- classification.config updated. ET 5.0 ruleset will use this.
- decoder event counters use 'decoder.event' as prefix now. This can be controlled using the `stats.decoder-events-prefix` setting.

4.7.2 Removals

- `dns-log`, the text dns log. Use EVE.dns instead.
- `file-log`, the non-EVE JSON file log. Use EVE.files instead.
- `drop-log`, the non-EVE JSON drop log.

See https://suricata.io/about/deprecation-policy/

CHAPTER

FIVE

SECURITY CONSIDERATIONS

Suricata is a security tool that processes untrusted network data, as well as requiring elevated system privileges to acquire that data. This combination deserves extra security precautions that we discuss below.

Additionally, supply chain attacks, particularly around rule distribution, could potentially target Suricata installations.

5.1 Running as a User Other Than Root

Note

If using the Suricata RPMs, either from the OISF COPR repo, or the EPEL repo, the following is already configured for you. The only thing you might want to do is add your management user to the `suricata` group.

Many Suricata examples and guides will show Suricata running as the *root* user, particularly when running on live traffic. As Suricata generally needs low level read (and in IPS write) access to network traffic, it is required that Suricata starts as root, however Suricata does have the ability to drop down to a non-root user after startup, which could limit the impact of a security vulnerability in Suricata itself.

Note

Currently the ability to drop root privileges after startup is only available on Linux systems.

5.1.1 Create User

Before running as a non-root user, you need to choose and possibly create the user and group that will Suricata will run as. Typically this user would be a sytem user with the name `suricata`. Such a user can be created with the following command:

```
useradd --no-create-home --system --shell /sbin/nologin suricata
```

This will create a user and group with the name `suricata`.

5.1.2 File System Permissions

Before running Suricata as the user `suricata`, some directory permissions will need to be updated to allow the `suricata` read and write access.

Assuming your Suricata was installed from source using the recommended configuration of:

```
./configure --prefix=/usr/ --sysconfdir=/etc/ --localstatedir=/var/
```

the following directories will need their permissions updated:

Directory	Permissions
/etc/suricata	Read
/var/log/suricata	Read, Write
/var/lib/suricata	Read, Write
/var/run/suricata	Read, Write

The following commands will setup the correct permissions:

- `/etc/suricata`:

```
chgrp -R suricata /etc/suricata
chmod -R g+r /etc/suricata
```

- `/var/log/suricata`:

```
chgrp -R suricata /var/log/suricata
chmod -R g+rw /var/log/suricata
```

- `/var/lib/suricata`:

```
chgrp -R suricata /var/lib/suricata
chmod -R g+srw /var/lib/suricata
```

- `/var/run/suricata`:

```
chgrp -R suricata /var/run/suricata
chmod -R g+srw /var/run/suricata
```

5.1.3 Configure Suricata to Run as `Suricata`

Suricata can be configured to run as an alternate user by updating the configuration file or using command line arguments.

- Using the configuration file, update the `run-as` section to look like:

```
run-as:
  user: suricata
  group: suricata
```

- Or if using command line arguments, add the following to your command:

```
--user suricata --group suricata
```

5.1.4 Starting Suricata

It is important to note that Suricata still needs to be started with **root** permissions in most cases. Starting as *root* allows Suricata to get access to the network interfaces and set the *capabilities* required during runtime before it switches down to the configured user.

5.1.5 Other Commands: Suricata-Update, SuricataSC

With the previous permissions setup, `suricata-update` and `suricatasc` can also be run without root or sudo. To allow a user to access these commands, add them to the `suricata` group.

5.2 Containers

Containers such as Docker and Podman are other methods to provide isolation between Suricata and the host machine running Suricata. However, we still recommend running as a non-root user, even in containers.

5.2.1 Capabilities

For both Docker and Podman the following capabilities should be provided to the container running Suricata for proper operation:

```
--cap-add=net_admin --cap-add=net_raw --cap-add=sys_nice
```

5.2.2 Podman

Unfortunately Suricata will not work with *rootless* Podman, this is due to Suricata's requirement to start with root privileges to gain access to the network interfaces. However, if started with the above capabilities, and configured to run as a non-root user, it will drop root privileges before processing network data.

CHAPTER

SIX

SUPPORT STATUS

6.1 Levels of Support

The support tiers detailed below do not represent a binding commitment. Instead, they serve as a framework that the OISF employs to prioritize features and functionality.

6.1.1 Tier 1

Tier 1 supported items are developed and supported by the Suricata team. These items receive full CI (continuous integration) coverage, and functional failures block git merges and releases. Tier 1 features are enabled by default on platforms that support the feature.

6.1.2 Tier 2

Tier 2 supported items are developed and supported by the Suricata team, sometimes with help from community members. Major functional failures block git merges and releases, however less major issues may be documented as "known issues" and may go into a release. Tier 2 features and functionality may be disabled by default.

6.1.3 Community

When a feature of Suricata is community supported, it means the OISF/Suricata development team won't directly support it. This is to avoid overloading the team.

When accepting a feature into the code base anyway, it will come with a number of limits and conditions:

- submitter must commit to maintaining it:
  - make sure code compiles and correctly functions after Suricata and/or external (e.g. library) changes.
  - support users when they encounter problems on forum and redmine tickets.
- the code will be disabled by default and will not become part of the QA setup. This means it will be enabled only by an `--enable` configure flag.
- the code may not have CI coverage by the OISF infrastructure.

If the feature gets lots of traction, and/or if the team just considers it very useful, it may get 'promoted' to being officially supported.

On the other hand, the feature will be removed if the submitter stops maintaining it and no-one steps up to take over.

6.1.4 Vendor

Vendor supported features are features specific to a certain vendor and usually require software and/or hardware from that vendor. While these features may exist in the main Suricata code, they rely on support from the vendor to keep the feature in a functional state.

Vendor supported functionality will generally not have CI or QA coverage by the OISF.

6.1.5 Unmaintained

When a feature is unmaintained it is very likely broken and may be (partially) removed during cleanups and code refactoring. No end-user support is done by the core team. If someone wants to help maintain and support such a feature, we recommend talking to the core team before spending a lot of time on it.

Please see *Contributing to Suricata* for more information if you wish to contribute.

6.2 Distributions

6.2.1 Tier 1

These tier 1 supported Linux distributions and operating systems receive full CI and QA, as well as documentation.

Distribution	Version	Support	QA	Notes
RHEL/CentOS	7	OISF		
RHEL/Alma/Rocky	8	OISF		
RHEL/Alma/Rocky	9	OISF		
Ubuntu	20.04	OISF		
Ubuntu	22.04	OISF		
Debian	10 (Buster)	OISF		
Debian	11 (Bullseye)	OISF		Foundation of SELKS
Debian	12 (Bookworm)	OISF		
FreeBSD	12	OISF		Foundation of OPNsense, pfSense
FreeBSD	13	OISF		Foundation of OPNSense

6.2.2 Tier 2

These tier 2 supported Linux distributions and operating systems receive CI but not full QA (functional testing).

Distribution	Version	Support	QA	Notes
CentOS	Stream	OISF		
Fedora	Active	OISF		
OpenBSD	7.2	OISF		
OpenBSD	7.1	OISF		
OSX/macOS	??	OISF		
Windows/MinGW64		OISF		

6.3 Architecture Support

6.3.1 Tier 1

Architecture	Support	QA	Notes
x86_64	OISF		
ARM8-64bit	OISF		

6.3.2 Tier 2

Architecture	Support	QA	Notes
ARM7-32bit	OISF		
i386	OISF		

6.3.3 Community

Architecture	Support	QA	Notes
PPC64el		Part of Fedora automated QA	Access can be arranged through IBM dev cloud
PPC64			No access to working hardware
PPC32			No access to working hardware
RISC-V			

6.3.4 High Level Features

Capture support

Tier 1

Capture Type	Maintainer	QA	Notes
AF_PACKET	OISF		Used by Security Onion, SELKS
NETMAP (FreeBSD)	OISF		Used by OPNsense, PFsense
NFQUEUE	OISF		
libpcap	OISF		

Tier 2

Capture Type	Maintainer	QA	Notes
PF_RING	OISF		
NETMAP (Linux)	OISF		
DPDK	OISF		
AF_PACKET (eBPF/XDP)	OISF		

Community

Capture Type	Maintainer	QA	Notes
NFLOG	Community		
AF_XDP	Community		

Vendor

Capture Type	Maintainer	QA	Notes
Napatech	Napatech / Community		

Unmaintained

Capture Type	Maintainer	QA	Notes
IPFW			
Endace/DAG			

Operation modes

Tier 1

Mode	Maintainer	QA	Notes
IDS (passive)	OISF		
IPS (active)	OISF		
Offline pcap file	OISF		

Tier 2

Mode	Maintainer	QA	Notes
Unix socket mode	OISF		
IDS (active)	OISF		Active responses, reject keyword

CHAPTER

SEVEN

COMMAND LINE OPTIONS

Suricata's command line options:

-h

Display a brief usage overview.

-V

Displays the version of Suricata.

-c <path>

Path to configuration file.

--include <path>

Additional configuration files to include. Multiple additional configuration files can be provided and will be included in the order specified on the command line. These additional configuration files are loaded as if they existed at the end of the main configuration file.

Example including one additional file:

```
--include /etc/suricata/other.yaml
```

Example including more than one additional file:

```
--include /etc/suricata/other.yaml --include /etc/suricata/extra.yaml
```

-T

Test configuration.

-v

Increase the verbosity of the Suricata application logging by increasing the log level from the default. This option can be passed multiple times to further increase the verbosity.

- -v: INFO
- -vv: PERF
- -vvv: CONFIG
- -vvvv: DEBUG

This option will not decrease the log level set in the configuration file if it is already more verbose than the level requested with this option.

-r <path>

Run in pcap offline mode (replay mode) reading files from pcap file. If <path> specifies a directory, all files in that directory will be processed in order of modified time maintaining flow state between files.

--pcap-file-continuous

Used with the -r option to indicate that the mode should stay alive until interrupted. This is useful with directories to add new files and not reset flow state between files.

--pcap-file-recursive

Used with the -r option when the path provided is a directory. This option enables recursive traversal into sub-directories to a maximum depth of 255. This option cannot be combined with --pcap-file-continuous. Symlinks are ignored.

--pcap-file-delete

Used with the -r option to force deletion of pcap files after they have been processed. This is useful with pcap-file-continuous to continuously feed files to a directory and have them cleaned up when done.

command-line vs Configuration: This command-line option overrides the `pcap-file.delete-when-done` configuration option in `suricata.yaml` and forces "always delete" mode (equivalent to `delete-when-done: true`).

For more control, use the `pcap-file.delete-when-done` configuration option instead, which supports three values:

- `false` (default): No files are deleted
- `true`: All files are deleted after processing
- `"non-alerts"`: Only files that generated no alerts are deleted

If neither `--pcap-file-delete` nor `delete-when-done` is configured, pcap files will not be deleted after processing.

--pcap-file-buffer-size <value>

Set read buffer size using `setvbuf` to speed up pcap reading. Valid values are 0, which disables `setvbuf` buffering, or 4 KiB to 64 MiB. Default value is 128 KiB. Supported on Linux only.

-i <interface>

After the -i option you can enter the interface card you would like to use to sniff packets from. This option will try to use the best capture method available. Can be used several times to sniff packets from several interfaces.

--pcap[=<device>]

Run in PCAP mode. If no device is provided the interfaces provided in the *pcap* section of the configuration file will be used.

--af-packet[=<device>]

Enable capture of packet using AF_PACKET on Linux. If no device is supplied, the list of devices from the af-packet section in the yaml is used.

--af-xdp[=<device>]

Enable capture of packet using AF_XDP on Linux. If no device is supplied, the list of devices from the af-xdp section in the yaml is used.

-q <queue id>

Run inline of the NFQUEUE queue ID provided. May be provided multiple times.

-s <filename.rules>

With the -s option you can set a file with signatures, which will be loaded together with the rules set in the yaml.

It is possible to use globbing when specifying rules files. For example, `-s '/path/to/rules/*.rules'`

-S <filename.rules>

With the -S option you can set a file with signatures, which will be loaded exclusively, regardless of the rules set in the yaml.

It is possible to use globbing when specifying rules files. For example, `-S '/path/to/rules/*.rules'`

-l <directory>

With the -l option you can set the default log directory. If you already have the default-log-dir set in yaml, it will not be used by Suricata if you use the -l option. It will use the log dir that is set with the -l option. If you do not set a directory with the -l option, Suricata will use the directory that is set in yaml.

-D

Normally if you run Suricata on your console, it keeps your console occupied. You can not use it for other purposes, and when you close the window, Suricata stops running. If you run Suricata as daemon (using the -D option), it runs at the background and you will be able to use the console for other tasks without disturbing the engine running.

--runmode <runmode>

With the *--runmode* option you can set the runmode that you would like to use. This command line option can override the yaml runmode option.

Runmodes are: *workers*, *autofp* and *single*.

For more information about runmodes see *Runmodes* in the user guide.

-F <bpf filter file>

Use BPF filter from file.

-k [all|none]

Force (all) the checksum check or disable (none) all checksum checks.

--user=<user>

Set the process user after initialization. Overrides the user provided in the *run-as* section of the configuration file.

--group=<group>

Set the process group to group after initialization. Overrides the group provided in the *run-as* section of the configuration file.

--pidfile <file>

Write the process ID to file. Overrides the *pid-file* option in the configuration file and forces the file to be written when not running as a daemon.

--init-errors-fatal

Exit with a failure when errors are encountered loading signatures.

--strict-rule-keywords[=all|<keyword>|<keywords(csv)]

Applies to: classtype, reference and app-layer-event.

By default missing reference or classtype values are warnings and not errors. Additionally, loading outdated app-layer-event events are also not treated as errors, but as warnings instead.

If this option is enabled these warnings are considered errors.

If no value, or the value 'all', is specified, the option applies to all of the keywords above. Alternatively, a comma separated list can be supplied with the keyword names it should apply to.

--disable-detection

Disable the detection engine.

--disable-hashing

Disable support for hash algorithms such as md5, sha1 and sha256.

By default hashing is enabled. Disabling hashing will also disable some Suricata features such as the filestore, ja3, and rule keywords that use hash algorithms.

--dump-config

Dump the configuration loaded from the configuration file to the terminal and exit.

--dump-features

Dump the features provided by Suricata modules and exit. Features list (a subset of) the configuration values and are intended to assist with comparing provided features with those required by one or more rules.

--build-info

Display the build information the Suricata was built with.

--list-app-layer-protos

List all supported application layer protocols.

--list-keywords=[all|json|csv|<kword>]

List all supported rule keywords.

--list-runmodes

List all supported run modes.

--plugin <path>

Load a plugin from *path* in addition to the plugins listed in the configuration file. This option can be specified multiple times.

If *path* is a directory, Suricata will attempt to load each `.so` file in that directory.

--set <key>=<value>

Set a configuration value. Useful for overriding basic configuration parameters. For example, to change the default log directory:

```
--set default-log-dir=/var/tmp
```

This option cannot be used to add new entries to a list in the configuration file, such as a new output. It can only be used to modify a value in a list that already exists.

For example, to disable the `eve-log` in the default configuration file:

```
--set outputs.1.eve-log.enabled=no
```

Also note that the index values may change as the `suricata.yaml` is updated.

See the output of `--dump-config` for existing values that could be modified with their index.

--engine-analysis

Print reports on analysis of different sections in the engine and exit. Please have a look at the conf parameter engine-analysis on what reports can be printed

--unix-socket=<file>

Use file as the Suricata unix control socket. Overrides the *filename* provided in the *unix-command* section of the configuration file.

--reject-dev=<device>

Use *device* to send out RST / ICMP error packets with the *reject* keyword.

--pcap-buffer-size=<size>

Set the size of the PCAP buffer (0 - 2147483647).

--netmap[=<device>]

Enable capture of packet using NETMAP on FreeBSD or Linux. If no device is supplied, the list of devices from the netmap section in the yaml is used.

--pfring[=<device>]

Enable PF_RING packet capture. If no device provided, the devices in the Suricata configuration will be used.

--pfring-cluster-id <id>

Set the PF_RING cluster ID.

--pfring-cluster-type <type>

Set the PF_RING cluster type (cluster_round_robin, cluster_flow).

-d <divert-port>

Run inline using IPFW divert mode.

--dag <device>

Enable packet capture off a DAG card. If capturing off a specific stream the stream can be select using a device name like "dag0:4". This option may be provided multiple times read off multiple devices and/or streams.

--napatech

Enable packet capture using the Napatech Streams API.

--erf-in=<file>

Run in offline mode reading the specific ERF file (Endace extensible record format).

--simulate-ips

Simulate IPS mode when running in a non-IPS mode.

7.1 Unit Tests

The builtin unittests are only available when Suricata has been configured and built with `--enable-unittests`.

Running unittests does not require a configuration file. Use -l to supply an output directory.:

```
sudo suricata -u
```

-u

Run the unit tests and exit. Requires that Suricata be configured with *--enable-unittests*.

-U, --unittest-filter=REGEX

With the -U option you can select which of the unit tests you want to run. This option uses REGEX. Example of use: suricata -u -U http

--list-unittests

Lists available unit tests.

--fatal-unittests

Enables fatal failure on a unit test error. Suricata will exit instead of continuing more tests.

--unittests-coverage

Display unit test coverage report.

CHAPTER

EIGHT

SURICATA RULES

8.1 Rules Format

Signatures play a very important role in Suricata. In most occasions people are using existing rulesets.

The official way to install rulesets is described in *Rule Management with Suricata-Update*.

There are a number of free rulesets that can be used via suricata-update. To aid in learning about writing rules, the Emerging Threats Open ruleset is free and a good reference that has a wide range of signature examples.

This Suricata Rules document explains all about signatures; how to read, adjust and create them.

A rule/signature consists of the following:

- The **action**, determining what happens when the rule matches.
- The **header**, defining the protocol, IP addresses, ports and direction of the rule.
- The **rule options**, defining the specifics of the rule.

An example of a rule is as follows:

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP GET Request Containing Rule in URI"; flow:established,to_server; http.method; content:"GET"; http.uri; content:"rule"; fast_pattern; classtype:bad-unknown; sid:123; rev:1;)

In this example, red is the action, green is the header and blue are the options.

We will be using the above signature as an example throughout this section, highlighting the different parts of the signature.

8.1.1 Action

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP GET Request Containing Rule in URI"; flow:established,to_server; http.method; content:"GET"; http.uri; content:"rule"; fast_pattern; classtype:bad-unknown; sid:123; rev:1;)

Valid actions are:

- alert - generate an alert.
- pass - stop further inspection of the packet.
- drop - drop packet and generate alert.
- reject - send RST/ICMP unreach error to the sender of the matching packet.
- rejectsrc - same as just *reject*.
- rejectdst - send RST/ICMP error packet to receiver of the matching packet.

- rejectboth - send RST/ICMP error packets to both sides of the conversation.

> **Note**
>
> In IPS mode, using any of the *reject* actions also enables *drop*.

For more information see *Action-order*.

8.1.2 Protocol

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP GET Request Containing Rule in URI"; flow:established,to_server; http.method; content:"GET"; http.uri; content:"rule"; fast_pattern; classtype:bad-unknown; sid:123; rev:1;)

The protocol value will limit what protocol(s) the signature will be applied to:

- ip (ip stands for 'all IP packets' or 'any IP packet')
- tcp (for TCP traffic)
- udp
- icmp (both ICMPv4 and ICMPv6)
- icmpv4
- icmpv6
- ipv4/ip4 - just IPv4
- ipv6/ip6 - just IPv6
- pkthdr (for matching on packets with decoder events)
- ether - Ethernet packets
- arp - ARP packets specifically

There are a couple of additional TCP related protocol options:

- tcp-pkt (for matching content in individual tcp packets)
- tcp-stream (for matching content only in a reassembled tcp stream)

There are also a few so-called application layer protocols, or layer 7 protocols you can pick from. These are:

- http (either HTTP1 or HTTP2)
- http1
- http2
- tls (this includes ssl)
- quic
- ftp (ftp command channel)
- ftp-data (ftp data channel)
- smb
- dns
- doh2 (dns over http/2)
- mdns

- dcerpc
- ldap
- dhcp
- ssh
- smtp
- imap
- pop3
- modbus (disabled by default)
- dnp3 (disabled by default)
- enip (disabled by default)
- nfs
- ike
- krb5
- bittorrent-dht
- mqtt
- ntp
- dhcp
- rfb
- rdp
- snmp
- tftp
- sip
- telnet
- websocket
- pgsql

This list of protocols can be obtained via `suricata --list-rule-protos`.

The availability of these protocols depends on whether the protocol is enabled in the configuration file, suricata.yaml.

If you have a signature with the protocol declared as 'http', Suricata makes sure the signature will only match if the TCP stream contains http traffic.

Matching on non-IP packets

Traditionally the rule language was only about matching on IP packets. For packets that caused decoder events in the layers before IP a special protocol *pkthdr* was added.

alert pkthdr any any -> any any (msg:"SURICATA IPv4 packet too small"; decode-event:ipv4.pkt_too_small; classtype:protocol-command-decode; sid:2200000; rev:2;)

Up until Suricata 8 this protocol was an alias for *alert ip*, but in Suricata 9 it is only to be used in decoder event rules.

Explicit rule hooks

In Suricata 8 the protocol field can be used to force evaluation of a rule at a specific explicit protocol state. This takes the format of:

```
<proto>:<hook>
```

Where each application protocol comes with a default set of hooks, as well as per protocol specific hooks.

More details can be found in *Explicit rule hook (states)*.

Note

A < hook at the protocol's first state (progress 0) is accepted and is equivalent to the plain hook form: there are no prior states to auto-accept (`http1:<request_started` equals `http1:request_started`).

Note

While developed for the firewall usecase, these hooks can be used in IDS/IPS rules as well.

8.1.3 Source and destination

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP GET Request Containing Rule in URI"; flow:established,to_server; http.method; content:"GET"; http.uri; content:"rule"; fast_pattern; classtype:bad-unknown; sid:123; rev:1;)

The first emphasized part is the traffic source, the second is the traffic destination (note the direction of the directional arrow).

With the source and destination, you specify the source of the traffic and the destination of the traffic, respectively. You can assign IP addresses, (both IPv4 and IPv6 are supported) and IP ranges. These can be combined with operators:

Operator	Description
../..	IP ranges (CIDR notation)
!	exception/negation
[.., ..]	grouping

Normally, you would also make use of variables, such as `$HOME_NET` and `$EXTERNAL_NET`. The suricata.yaml configuration file specifies the IP addresses these concern. The respective `$HOME_NET` and `$EXTERNAL_NET` settings will be used in place of the variables in your rules.

See *Rule-vars* for more information.

Rule usage examples:

Example	Meaning
!1.1.1.1	Every IP address but 1.1.1.1
![1.1.1.1, 1.1.1.2]	Every IP address but 1.1.1.1 and 1.1.1.2
$HOME_NET	Your setting of HOME_NET in yaml
[$EXTERNAL_NET, !$HOME_NET]	EXTERNAL_NET and not HOME_NET
[10.0.0.0/24, !10.0.0.5]	10.0.0.0/24 except for 10.0.0.5
[..., [....]]	
[..., ![.....]]	

Warning

If you set your configuration to something like this:

```
HOME_NET: any
EXTERNAL_NET: !$HOME_NET
```

You cannot write a signature using `$EXTERNAL_NET` because it evaluates to 'not any', which is an invalid value.

Note

Please note that the source and destination address can also be matched via the `ip.src` and `ip.dst` keywords (See *IP Addresses Match*). These keywords are mostly used in conjunction with the dataset feature (*Datasets*).

8.1.4 Ports (source and destination)

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP GET Request Containing Rule in URI"; flow:established,to_server; http.method; content:"GET"; http.uri; content:"rule"; fast_pattern; classtype:bad-unknown; sid:123; rev:1;)

The first emphasized part is the source port, the second is the destination port (note the direction of the directional arrow).

Traffic comes in and goes out through ports. Different protocols have different port numbers. For example, the default port for HTTP is 80 while 443 is typically the port for HTTPS. Note, however, that the port does not dictate which protocol is used in the communication. Rather, it determines which application is receiving the data.

The ports mentioned above are typically the destination ports. Source ports, i.e. the application that sent the packet, typically get assigned a random port by the operating system. When writing a rule for your own HTTP service, you would typically write `any -> 80`, since that would mean any packet from any source port to your HTTP application (running on port 80) is matched.

In setting ports you can make use of special operators as well. Operators such as:

Operator	Description
:	port ranges
!	exception/negation
[.., ..]	grouping

Rule usage examples:

Example	Meaning
[80, 81, 82]	port 80, 81 and 82
[80: 82]	Range from 80 till 82
[1024:]	From 1024 till the highest port-number
!80	Every port but 80
[80:100,!99]	Range from 80 till 100 but 99 excluded
[1:80,![2,4]]	Range from 1-80, except ports 2 and 4
[.., [.,..,..]]	

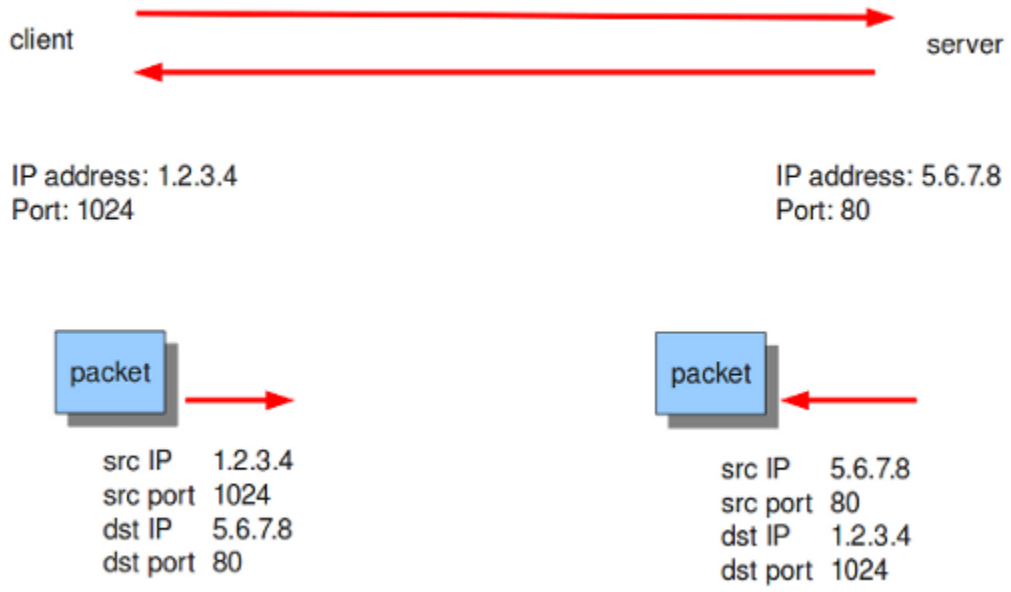
8.1.5 Direction

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP GET Request Containing Rule in URI"; flow:established,to_server; http.method; content:"GET"; http.uri; content:"rule"; fast_pattern; classtype:bad-unknown; sid:123; rev:1;)

The directional arrow indicates which way the signature will be evaluated. In most signatures an arrow to the right (->) is used. This means that only packets with the same direction can match. There is also the double arrow (=>), which respects the directionality as ->, but allows matching on bidirectional transactions, used with keywords matching each direction. Finally, it is also possible to have a rule match either directions (<>):

```
source -> destination
source => destination
source <> destination  (either directions)
```

The following example illustrates direction. In this example there is a client with IP address 1.2.3.4 using port 1024. A server with IP address 5.6.7.8, listening on port 80 (typically HTTP). The client sends a message to the server and the server replies with its answer.



Now, let's say we have a rule with the following header:

```
alert tcp 1.2.3.4 1024 -> 5.6.7.8 80
```

Only the traffic from the client to the server will be matched by this rule, as the direction specifies that we do not want to evaluate the response packet.

Now, if we have a rule with the following header:

```
alert tcp 1.2.3.4 any <> 5.6.7.8 80
```

Suricata will duplicate it and use the same rule with headers in both directions :

alert tcp 1.2.3.4 any -> 5.6.7.8 80 alert tcp 5.6.7.8 80 -> 1.2.3.4 any

Warning

There is no 'reverse' style direction, i.e. there is no <-.

Transactional rules

Here is an example of a transactional rule:

alert http any any => 5.6.7.8 80 (msg:"matching both uri and status"; sid: 1; http.uri; content: "/download"; http.stat_code; content: "200";)

It will match on flows to 5.6.7.8 and port 80. And it will match on a full transaction, using both the uri from the request, and the stat_code from the response. As such, it will match only when Suricata got both request and response.

Transactional rules can use direction-ambiguous keywords, by specifying the direction.

alert http any any => 5.6.7.8 80 (msg:"matching json to server and xml to client"; sid: 1; http.content_type: to_server; content: "json"; http.content_type: to_client; content: "xml";)

Transactional rules have some limitations :

- They are only meant to work on transactions with first a request to the server, and then a response to the client, and not the other way around (not tested).
- They cannot have `fast_pattern` or `prefilter` the direction to client if they also have a streaming buffer on the direction to server, see example below.
- They will refuse to load if a single directional rule is enough.

This rule cannot have the `fast_pattern` to client, as `file.data` is a streaming buffer and will refuse to load.

```
alert http any any => any any (file.data: to_server; content: "123"; http.stat_code;
content: "500"; fast_pattern; sid: 1;)
```

If not explicit, a transactional rule will choose a fast_pattern to server by default.

Transactional rules behavior depends on how transactions are implemented per protocol, see *Protocols* for more details.

8.1.6 Rule options

The rest of the rule consists of options. These are enclosed by parenthesis and separated by semicolons. Some options have settings (such as `msg`), which are specified by the keyword of the option, followed by a colon, followed by the settings. Others have no settings; they are simply the keyword (such as `nocase`):

```
<keyword>: <settings>;
<keyword>;
```

Rule options have a specific ordering and changing their order would change the meaning of the rule.

Note

The characters ; and " have special meaning in the Suricata rule language and must be escaped when used in a rule option value. For example:

```
msg:"Message with semicolon\;";
```

As a consequence, you must also escape the backslash, as it functions as an escape character.

The rest of this chapter in the documentation documents the use of the various keywords.

Some generic details about keywords follow.

Disabling Alerts

There is a way to disable alert generation for a rule using the keyword `noalert`. When this keyword is part of a rule, no alert is generated if the other portions of the rule match. That is, the other rule actions will *still be applied.* Using `noalert` can be helpful when a rule is collecting or setting state using *flowbits*, *datasets* or other state maintenance constructs of the rule language. See *Thresholding Keywords* for other ways to control alert frequency.

The following rules demonstrate `noalert` with a familiar pattern:

- The first rule marks state without generating an alert.
- The second rule generates an alert if the state is set and additional qualifications are met.

alert http any any -> $HOME_NET any (msg:"noalert example: set state"; flow:established,to_server; xbits:set,SC.EXAMPLE,track ip_dst, expire 10; noalert; http.method; content:"GET"; sid:1;)

alert http any any -> $HOME_NET any (msg:"noalert example: state use"; flow:established,to_server; xbits:isset,SC.EXAMPLE,track ip_dst; http.method; content:"POST"; sid: 2;)

In IPS mode, `noalert` is commonly used in when Suricata should *drop* network packets without generating alerts (example below). The following rule is a simplified example showing how `noalert` could be used with IPS deployments to drop inbound SSH requests.

drop tcp any any -> any 22 (msg:"Drop inbound SSH traffic"; noalert; sid: 3;)

Modifier Keywords

Some keywords function act as modifiers. There are two types of modifiers.

- The older style **'content modifiers'** look back in the rule, e.g.:

  ```
  alert http any any -> any any (content:"index.php"; http_uri; sid:1;)
  ```

  In the above example the pattern 'index.php' is modified to inspect the HTTP uri buffer.

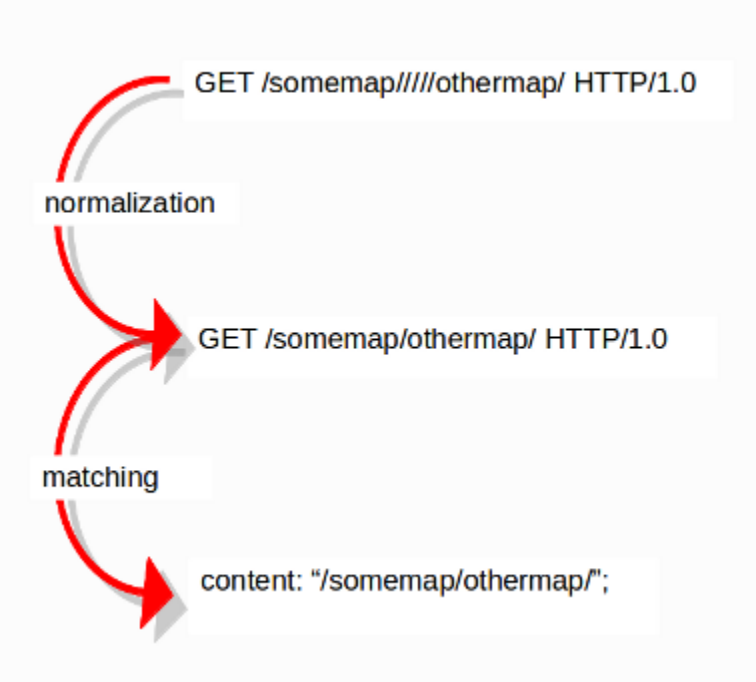
- The more recent type is called the **'sticky buffer'**. It places the buffer name first and all keywords following it apply to that buffer, for instance:

  ```
  alert http any any -> any any (http_response_line; content:"403 Forbidden"; sid:1;)
  ```

  In the above example the pattern '403 Forbidden' is inspected against the HTTP response line because it follows the `http_response_line` keyword.

Normalized Buffers

A packet consists of raw data. HTTP and reassembly make a copy of those kinds of packets data. They erase anomalous content, combine packets etcetera. What remains is a called the 'normalized buffer':



Because the data is being normalized, it is not what it used to be; it is an interpretation. Normalized buffers are: all HTTP-keywords, reassembled streams, TLS-, SSL-, SSH-, FTP- and dcerpc-buffers.

Note that there are some exceptions, e.g. the `http_raw_uri` keyword. See *http.uri* for more information.

8.2 Meta Keywords

Meta keywords have no effect on Suricata's inspection of network traffic; they do have an effect on the way Suricata reports events/alerts.

8.2.1 msg (message)

The keyword msg gives contextual information about the signature and the possible alert.

The format of msg is:

```
msg: "some description";
```

Examples:

```
msg:"ET MALWARE Win32/RecordBreaker CnC Checkin";
msg:"ET EXPLOIT SMB-DS DCERPC PnP bind attempt";
```

To continue the example from the previous chapter, the msg component of the signature is emphasized below:

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP GET Request Containing Rule in URI"; flow:established,to_server; http.method; content:"GET"; http.uri; content:"rule"; fast_pattern; classtype:bad-unknown; sid:123; rev:1;)

Tip

It is a standard practice in rule writing to make the first part of the signature msg uppercase and to indicate the class of the signature.

It is also standard practice that `msg` is the first keyword in the signature.

Note

The following characters must be escaped inside the msg: `;` `\` `"`

8.2.2 sid (signature ID)

The keyword sid gives every signature its own id. This id is stated with a number greater than zero. The format of sid is:

```
sid:123;
```

Example of sid in a signature:

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP GET Request Containing Rule in URI"; flow:established,to_server; http.method; content:"GET"; http.uri; content:"rule"; fast_pattern; classtype:bad-unknown; sid:123; rev:1;)

Tip

It is a standard practice in rule writing that the signature `sid` is provided as the last keyword (or second-to-last if there is a `rev`) of the signature.

There are reserved ranges of sids, the reservations are recorded at https://sidallocation.org/ .

Note

This value must be unique for all rules within the same *rule group* (`gid`).

As Suricata-update currently considers the rule's `sid` only (cf. Bug#5447), it is advisable to opt for a completely unique `sid` altogether.

Suricata Engine Events Rules SID range

Suricata provides signatures for engine events, bundled in the *rules* folder, in the *suricata* directory/repository. The full signature ID range for such rules is: 2200000 - 2299999.

These sid ranges can be seen in the rules README file).

8.2.3 rev (revision)

The sid keyword is commonly accompanied by the rev keyword. Rev represents the version of the signature. If a signature is modified, the number of rev will be incremented by the signature writers. The format of rev is:

```
rev:123;
```

Example of rev in a signature:

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP GET Request Containing Rule in URI"; flow:established,to_server; http.method; content:"GET"; http.uri; content:"rule"; fast_pattern; classtype:bad-unknown; sid:123; rev:1;)

Tip

It is a standard practice in rule writing that the rev keyword is expressed after the sid keyword. The sid and rev keywords are commonly put as the last two keywords in a signature.

8.2.4 gid (group ID)

The gid keyword can be used to give different groups of signatures another id value (like in sid). Suricata by default uses gid 1. It is possible to modify the default value. In most cases, it will be unnecessary to change the default gid value. Changing the gid value has no technical implications, the value is only noted in alert data.

Example of the gid value in an alert entry in the fast.log file. In the part [1:123], the first 1 is the gid (123 is the sid and 1 is the rev).

Note

07/12/2022-21:59:26.713297 [**] [1:123:1] HTTP GET Request Containing Rule in URI [**] [Classification: Potentially Bad Traffic] [Priority: 2] {TCP} 192.168.225.121:12407 -> 172.16.105.84:80

8.2.5 classtype

The classtype keyword gives information about the classification of rules and alerts. It consists of a short name, a long name and a priority. It can tell for example whether a rule is just informational or is about a CVE. For each classtype, the classification.config has a priority that will be used in the rule.

Example classtype definition:

```
config classification: web-application-attack,Web Application Attack,1
config classification: not-suspicious,Not Suspicious Traffic,3
```

Once we have defined the classification in the configuration file, we can use the classtypes in our rules. A rule with classtype web-application-attack will be assigned a priority of 1 and the alert will contain 'Web Application Attack' in the Suricata logs:

classtype	Alert	Priority
web-application-attack	Web Application Attack	1
not-suspicious	Not Suspicious Traffic	3

Our continuing example also has a classtype: bad-unknown:

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP GET Request Containing Rule in URI"; flow:established,to_server; http.method; content:"GET"; http.uri; content:"rule"; fast_pattern; classtype:bad-unknown; sid:123; rev:1;)

> **Tip**
>
> It is a standard practice in rule writing that the classtype keyword comes before the sid and rev keywords (as shown in the example rule).

8.2.6 reference

The reference keyword is used to document where information about the signature and about the problem the signature tries to address can be found. The reference keyword can appear multiple times in a signature. This keyword is meant for signature-writers and analysts who investigate why a signature has matched. It has the following format:

```
reference:type,reference
```

A typical reference to www.info.com would be:

```
reference:url,www.info.com
```

There are several systems that can be used as a reference. A commonly known example is the CVE-database, which assigns numbers to vulnerabilities, to prevent having to type the same URL over and over again. An example reference of a CVE:

```
reference:cve,CVE-2014-1234
```

This would make a reference to http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1234.

All the reference types are defined in the reference.config configuration file.

8.2.7 priority

The priority keyword comes with a mandatory numeric value which can range from 1 to 255. The values 1 through 4 are commonly used. The highest priority is 1. Signatures with a higher priority will be examined first. Normally signatures have a priority determined through a classtype definition. The classtype definition can be overridden by defining the priority keyword in the signature. The format of priority is:

```
priority:1;
```

8.2.8 metadata

The metadata keyword allows additional, non-functional, information to be added to the signature. While the format is free-form, it is recommended to stick to *[key, value]* pairs as Suricata can include these in eve alerts. The format is:

```
metadata: key value;
metadata: key value, key value;
```

8.2.9 target

The target keyword allows the rules writer to specify which side of the alert is the target of the attack. If specified, the alert event is enhanced to contain information about source and target.

The format is:

```
target:[src_ip|dest_ip]
```

If the value is src_ip then the source IP in the generated event (src_ip field in JSON) is the target of the attack. If target is set to dest_ip then the target is the destination IP in the generated event.

8.2.10 requires

The `requires` keyword allows a rule to require specific Suricata features to be enabled, specific keywords to be available, or the Suricata version to match an expression. Rules that do not meet the requirements will be ignored, and Suricata will not treat them as errors.

Requirements that follow the valid format of `<keyword> <expression>` but are not known to Suricata are allowed for future compatiblity, however unknown requirement expressions will lead to the requirement not being met, skipping the rule.

When parsing rules, the parser attempts to process the `requires` keywords before others. This allows it to occur after keywords that may only be present in specific versions of Suricata, as specified by the `requires` statement. However, the keywords preceding it must still adhere to the basic known formats of Suricata rules.

The format is:

```
requires: feature geoip, version >= 7.0.0, keyword foobar
```

To require multiple features, the feature sub-keyword must be specified multiple times:

```
requires: feature geoip, feature lua
```

Alternatively, *and* expressions may be expressed like:

```
requires: version >= 7.0.4 < 8
```

and *or* expressions may expressed with `|` like:

```
requires: version >= 7.0.4 < 8 | >= 8.0.3
```

to express that a rule requires version 7.0.4 or greater, but less than 8, **OR** greater than or equal to 8.0.3. Which could be useful if a keyword wasn't added until 7.0.4 and the 8.0.3 patch releases, as it would not exist in 8.0.1.

This can be extended to multiple release branches:

```
requires: version >= 7.0.10 < 8 | >= 8.0.5 < 9 | >= 9.0.3
```

If no *minor* or *patch* version component is provided, it will default to 0.

The `version` may only be specified once, if specified more than once the rule will log an error and not be loaded.

The `requires` keyword was introduced in Suricata 7.0.3 and 8.0.0.

8.3 Ethernet Keywords

8.3.1 ether.hdr

Sticky buffer to match on the whole Ethernet header.

Example rule:

alert ether any any -> any any (msg:"ARP packet"; ether.hdr; content:"|08 06|"; offset:12; depth:2; sid:1234; rev:5;)

8.4 IP Keywords

8.4.1 ttl

The ttl keyword is used to check for a specific IP time-to-live value in the header of a packet. The format is:

```
ttl:<number>;
```

For example:

```
ttl:10;
```

ttl uses an *unsigned 8-bit integer*.

At the end of the ttl keyword you can enter the value on which you want to match. The Time-to-live value determines the maximal amount of time a packet can be in the Internet-system. If this field is set to 0, then the packet has to be destroyed. The time-to-live is based on hop count. Each hop/router the packet passes subtracts one from the packet TTL counter. The purpose of this mechanism is to limit the existence of packets so that packets can not end up in infinite routing loops.

Example of the ttl keyword in a rule:

alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"IP Packet With TTL 0"; ttl:0; classtype:misc-activity; sid:1; rev:1;)

8.4.2 ipopts

With the ipopts keyword you can check if a specific IP option is set. Ipopts has to be used at the beginning of a rule. You can only match on one option per rule. There are several options on which can be matched. These are:

IP Option	Description
rr	Record Route
eol	End of List
nop	No Op
ts	Time Stamp
sec	IP Security
esec	IP Extended Security
lsrr	Loose Source Routing
ssrr	Strict Source Routing
satid	Stream Identifier
qs	Quick-Start
rtralt	Router Alert
cipso	Commercial Security
any	any IP options are set

Format of the ipopts keyword:

```
ipopts: <name>;
```

For example:

```
ipopts: ts;
```

Example of ipopts in a rule:

alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"IP Packet with timestamp option"; ipopts:ts; classtype:misc-activity; sid:2; rev:1;)

8.4.3 sameip

Every packet has a source IP-address and a destination IP-address. It can be that the source IP is the same as the destination IP. With the sameip keyword you can check if the IP address of the source is the same as the IP address of the destination. The format of the sameip keyword is:

```
sameip;
```

Example of sameip in a rule:

alert ip any any -> any any (msg:"IP Packet with the same source and destination IP"; sameip; classtype:bad-unknown; sid:3; rev:1;)

8.4.4 ip_proto

With the ip_proto keyword you can match on the IP protocol in the packet-header. You can use the name or the number of the protocol. You can match for example on the following protocols:

```
 1     ICMP        Internet Control Message
 6     TCP         Transmission Control Protocol
17     UDP         User Datagram
47     GRE         General Routing Encapsulation
50     ESP         Encap Security Payload for IPv6
51     AH          Authentication Header for Ipv6
58     IPv6-ICMP   ICMP for Ipv6
```

For the complete list of protocols and their numbers see http://en.wikipedia.org/wiki/List_of_IP_protocol_numbers

Example of ip_proto in a rule:

alert ip any any -> any any (msg:"IP Packet with protocol 1"; ip_proto:1; classtype:bad-unknown; sid:5; rev:1;)

The named variant of that example would be:

```
ip_proto:ICMP;
```

8.4.5 ipv4.hdr

Sticky buffer to match on content contained within an IPv4 header.

Example rule:

alert ip any any -> any any (msg:"IPv4 header keyword example"; ipv4.hdr; content:"|06|"; offset:9; depth:1; sid:1; rev:1;)

This example looks if byte 10 of IPv4 header has value 06, which indicates that the IPv4 protocol is TCP.

8.4.6 ipv6.hdr

Sticky buffer to match on content contained within an IPv6 header.

Example rule:

alert ip any any -> any any (msg:"IPv6 header keyword example"; ipv6.hdr; content:"|06|"; offset:6; depth:1; sid:1; rev:1;)

This example looks if byte 7 of IP64 header has value 06, which indicates that the IPv6 protocol is TCP.

8.4.7 id

With the id keyword, you can match on a specific IP ID value. The ID identifies each packet sent by a host and increments usually with one with each packet that is being send. The IP ID is used as a fragment identification number. Each packet has an IP ID, and when the packet becomes fragmented, all fragments of this packet have the same ID. In this way, the receiver of the packet knows which fragments belong to the same packet. (IP ID does not take care of the order, in that case offset is used. It clarifies the order of the fragments.)

id uses an *unsigned 16-bits integer*.

Format of id:

```
id:<number>;
```

Example of id in a rule:

alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"id keyword example"; id:1; content:"content|3a 20|"; fast_pattern; classtype:misc-activity; sid:12; rev:1;)

8.4.8 geoip

The geoip keyword enables matching on the source, destination or source and destination IPv4 addresses of network traffic, and to see to which country it belongs. To be able to do this, Suricata uses the GeoIP2 API of MaxMind.

The syntax of geoip:

```
geoip: src,RU;
geoip: both,CN,RU;
geoip: dst,CN,RU,IR;
geoip: both,US,CA,UK;
geoip: any,CN,IR;
```

Option	Description
both	Both source and destination have to match with the given geoip(s)
any	Either the source or the destination has to match with the given geoip(s).
dest	The destination matches with the given geoip.
src	The source matches with the given geoip.

geoip currently only supports IPv4. As it uses the GeoIP2 API of MaxMind, libmaxminddb must be compiled in. You must download and install the GeoIP2 or GeoLite2 database editions desired. Visit the MaxMind site at https://dev.maxmind.com/geoip/geolite2-free-geolocation-data for details.

You must also supply the location of the GeoIP2 or GeoLite2 database file on the local system in the YAML-file configuration (for example):

```
geoip-database: /usr/local/share/GeoIP/GeoLite2-Country.mmdb
```

8.4.9 fragbits (IP fragmentation)

With the fragbits keyword, you can check if the fragmentation and reserved bits are set in the IP header. The fragbits keyword should be placed at the beginning of a rule. Fragbits is used to modify the fragmentation mechanism. During routing of messages from one Internet module to the other, it can occur that a packet is bigger than the maximal packet size a network can process. In that case, a packet can be send in fragments. This maximum of the packet size is called Maximal Transmit Unit (MTU).

You can match on the following bits:

```
M - More Fragments
D - Do not Fragment
R - Reserved Bit
```

Matching on this bits can be more specified with the following modifiers:

```
+         match on all the specified bits (plus any others)
*         match if any of the specified bits are set
! or -    match if not all the specified bits are set
=         match on all the specified bits and only them
```

fragbits uses an *unsigned 16-bits integer* with bitmasks.

Format:

```
fragbits:[*+!=]<[MDR]>;
```

Example of fragbits in a rule:

alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"fragbits keyword example non-fragmented packet with fragment offset>0"; fragbits:M; fragoffset:>0; classtype:bad-unknown; sid:123; rev:1;)

8.4.10 fragoffset

With the fragoffset keyword you can match on specific decimal values of the IP fragment offset field. If you would like to check the first fragments of a session, you have to combine fragoffset 0 with the More Fragment option. The fragmentation offset field is convenient for reassembly. The id is used to determine which fragments belong to which packet and the fragmentation offset field clarifies the order of the fragments.

fragoffset uses an *unsigned 16-bits integer*.

You can use the following modifiers:

```
<       match if the value is smaller than the specified value
>       match if the value is greater than the specified value
!       match if the specified value is not present
```

Format of fragoffset:

```
fragoffset:[!|<|>]<number>;
```

Example of fragoffset in a rule:

alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"fragoffset keyword example invalid non-fragmented packet with fragment offset>0"; fragbits:M; fragoffset:>0; classtype:bad-unknown; sid:13; rev:1;)

8.4.11 tos

The tos keyword can match on specific decimal values of the IP header TOS field. The tos keyword can have a value from 0 - 255. This field of the IP header has been updated by rfc2474 to include functionality for Differentiated services. Note that the value of the field has been defined with the right-most 2 bits having the value 0. When specifying a value for tos, ensure that the value follows this.

E.g, instead of specifying the decimal value 34 (hex 22), right shift twice and use decimal 136 (hex 88).

You can specify hexadecimal values with a leading *x*, e.g, *x88*.

Format of tos:

```
tos:[!]<number>;
```

Example of tos in a rule:

alert ip any any -> any any (msg:"tos keyword example tos value 8"; flow:established; tos:8; classtype:not-suspicious; sid:123; rev:1;)

Example of tos with a negated value:

alert ip any any -> any any (msg:"tos keyword example with negated content"; flow:established,to_server; tos:!8; classtype:bad-unknown; sid:14; rev:1;)

8.5 TCP keywords

8.5.1 tcp.flags

The tcp.flags keyword checks for specific TCP flag bits.

The following flag bits may be checked:

Flag	Description
F	FIN - Finish
S	SYN - Synchronize sequence numbers
R	RST - Reset
P	PSH - Push
A	ACK - Acknowledgment
U	URG - Urgent
C	CWR - Congestion Window Reduced
E	ECE - ECN-Echo
0	No TCP Flags Set

The following modifiers can be set to change the match criteria:

Modifier	Description
+	match on all the bits, plus any others
*	match if any of the bits are set
!	match if the bits are not set (or -)
=	match on all the bits, and only them

To handle writing rules for session initiation packets such as ECN where a SYN packet is sent with CWR and ECE flags set, an option mask may be used by appending a comma and masked values. For example, a rule that checks for a SYN flag, regardless of the values of the reserved bits is `tcp.flags:S,CE;`

tcp.flags uses an *unsigned 8-bits integer* with bitmasks.

Format of tcp.flags:

```
tcp.flags:[modifier]<test flags>[,<ignore flags>];
tcp.flags:[!|*|+]<FSRPAUCE0>[,<FSRPAUCE>];
```

Example:

alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"Example tcp.flags sig"; tcp.flags:FPU,CE; classtype:misc-activity; sid:1; rev:1;)

It is also possible to use the *tcp.flags* content as a fast_pattern by using the *prefilter* keyword. For more information on *prefilter* usage see *Prefiltering Keywords*. Example:

alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"Example tcp.flags sig"; tcp.flags:FPU,CE; prefilter; classtype:misc-activity; sid:1; rev:1;)

8.5.2 seq

The `seq` keyword can be used in a signature to check for a specific TCP sequence number. A sequence number is a number that is generated practically at random by both endpoints of a TCP-connection. The client and the server both create a sequence number, which increases by one with every byte that they send. So this sequence number is different for both sides. This sequence number has to be acknowledged by both sides of the connection.

Through sequence numbers, TCP handles acknowledgement, order and retransmission. Its number increases with every data-byte the sender has sent. The seq helps keeping track of to what place in a data-stream a byte belongs. If the SYN flag is set at 1, then the sequence number of the first byte of the data is this number plus 1 (so, 2).

seq uses an *unsigned 32-bits integer*.

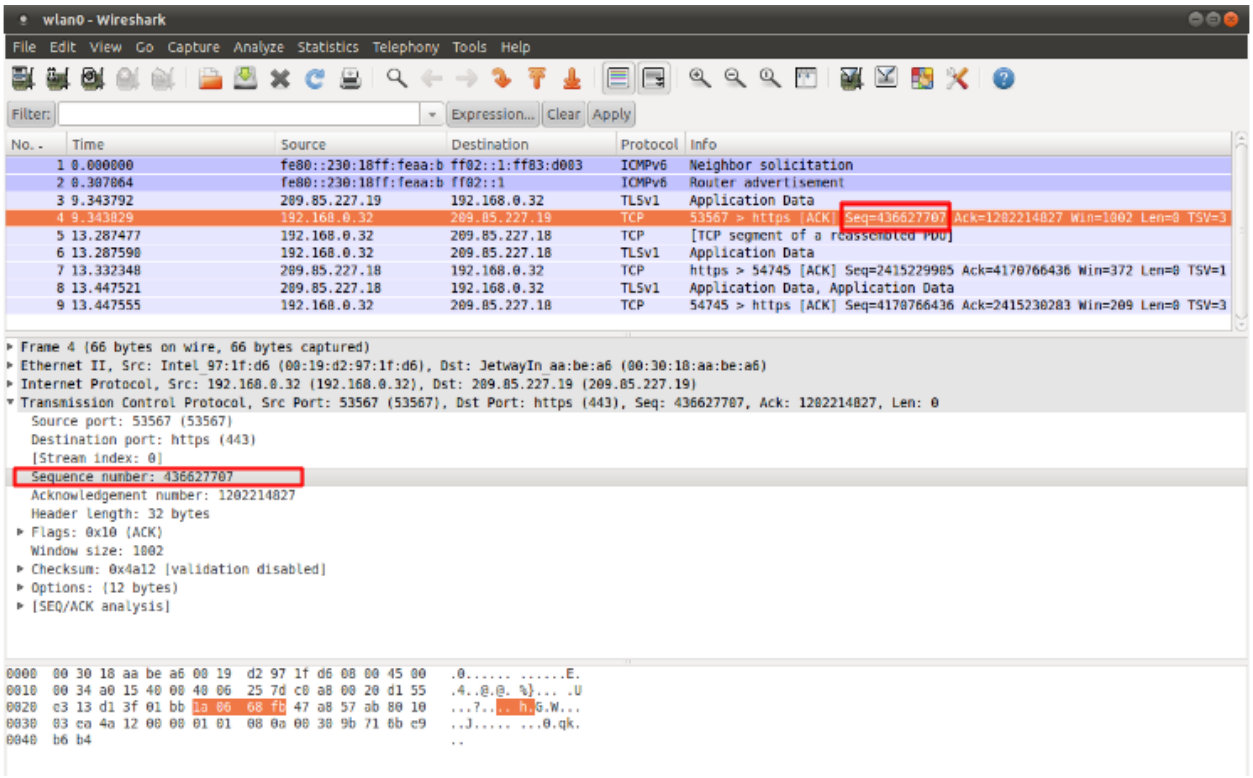
Example:

```
seq:0;
```

Example of seq in a signature:

alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"GPL SCAN NULL"; flow:stateless; ack:0; flags:0; seq:0; reference:arachnids,4; classtype:attempted-recon; sid:2100623; rev:7;)

Example of seq in a packet (Wireshark):



8.5.3 ack

The `ack` keyword can be used in a signature to check for a specific TCP acknowledgement number.

The `ack` is the acknowledgement of the receipt of all previous (data)-bytes send by the other side of the TCP-connection. In most occasions every packet of a TCP connection has an ACK flag after the first SYN and a ack-number which increases with the receipt of every new data-byte.

ack uses an *unsigned 32-bits integer*.

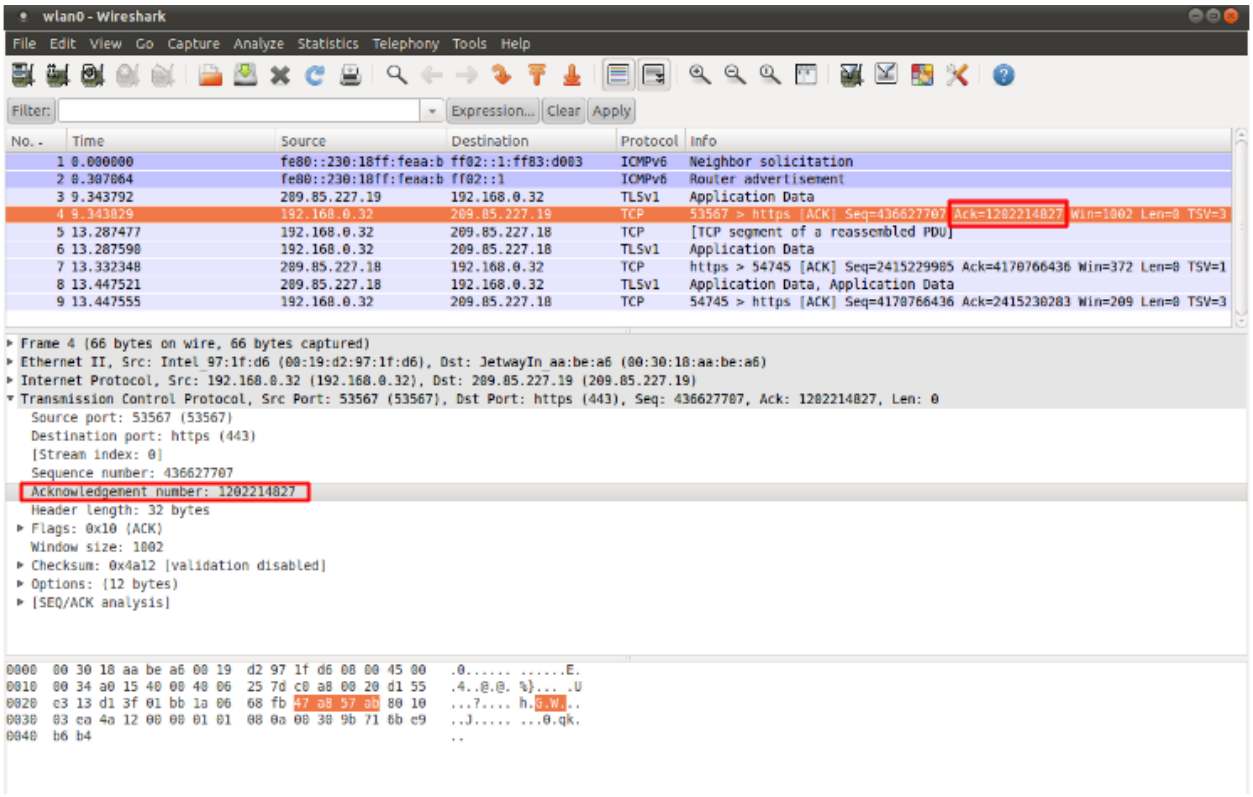
Format of `ack`:

```
ack:1;
```

Example of `ack` in a signature:

alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"GPL SCAN NULL"; flow:stateless; ack:0; flags:0; seq:0; reference:arachnids,4; classtype:attempted-recon; sid:2100623; rev:7;)

Example of `ack` in a packet (Wireshark):



8.5.4 window

The `window` keyword is used to check for a specific TCP window size.

The TCP window size is a mechanism that has control of the data-flow. The window is set by the receiver (receiver advertised window size) and indicates the amount of bytes that can be received. This amount of data has to be acknowledged by the receiver first, before the sender can send the same amount of new data.

This mechanism is used to prevent the receiver from being overflowed by data. The value of the window size is limited and can be 2 to 65.535 bytes. To make more use of your bandwidth you can use a bigger TCP-window.

window uses an *unsigned 16-bits integer*.

The format of the window keyword is:

```
window:[!]<number>;
```

Example of window in a rule:

alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"GPL DELETED typot trojan traffic"; flow:stateless; flags:S,12; window:55808; reference:mcafee,100406; classtype:trojan-activity; sid:2182; rev:8;)

8.5.5 tcp.mss

Match on the TCP MSS option value. Will not match if the option is not present.

`tcp.mss` uses an *unsigned 16-bit integer*.

The format of the keyword is:

```
tcp.mss:<min>-<max>;
tcp.mss:[<|>]<number>;
tcp.mss:<value>;
```

Example rule:

alert tcp $EXTERNAL_NET any -> $HOME_NET any (flow:stateless; flags:S,12; tcp.mss:<536; sid:1234; rev:5;)

8.5.6 tcp.wscale

Match on the TCP window scaling option value. Will not match if the option is not present.

`tcp.wscale` uses an *unsigned 8-bit integer*.

The format of the keyword is:

```
tcp.wscale:<min>-<max>;
tcp.wscale:[<|>]<number>;
tcp.wscale:<value>;
```

Example rule:

alert tcp $EXTERNAL_NET any -> $HOME_NET any (flow:stateless; flags:S,12; tcp.wscale:>10; sid:1234; rev:5;)

8.5.7 tcp.hdr

Sticky buffer to match on the whole TCP header.

Example rule:

alert tcp $EXTERNAL_NET any -> $HOME_NET any (flags:S,12; tcp.hdr; content:"|02 04|"; offset:20; byte_test:2,<,536,0,big,relative; sid:1234; rev:5;)

This example starts looking after the fixed portion of the header, so into the variable sized options. There it will look for the MSS option (type 2, option len 4) and using a byte_test determine if the value of the option is lower than 536. The *tcp.mss* option will be more efficient, so this keyword is meant to be used in cases where no specific keyword is available.

8.6 UDP keywords

8.6.1 udp.hdr

Sticky buffer to match on the whole UDP header.

Example rule:

alert udp any any -> any any (udp.hdr; content:"|00 08|"; offset:4; depth:2; sid:1234; rev:5;)

This example matches on the length field of the UDP header. In this case the length of 8 means that there is no payload. This can also be matched using *dsize:0;*.

8.7 ICMP keywords

ICMP (Internet Control Message Protocol) is a part of IP. IP at itself is not reliable when it comes to delivering data (datagram). ICMP gives feedback in case problems occur. It does not prevent problems from happening, but helps in understanding what went wrong and where. If reliability is necessary, protocols that use IP have to take care of reliability themselves. In different situations ICMP messages will be send. For instance when the destination is unreachable, if there is not enough buffer-capacity to forward the data, or when a datagram is send fragmented when it should not be, etcetera. More can be found in the list with message-types.

There are four important contents of a ICMP message on which can be matched with corresponding ICMP-keywords. These are: the type, the code, the id and the sequence of a message.

8.7.1 itype

The itype keyword is for matching on a specific ICMP type (number). ICMP has several kinds of messages and uses codes to clarify those messages. The different messages are distinct by different names, but more important by numeric values. For more information see the table with message-types and codes.

itype uses an *unsigned 8-bit integer*.

The format of the itype keyword:

```
itype:min<>max;
itype:[<|>]<number>;
```

Example This example looks for an ICMP type greater than 10:

```
itype:>10;
```

Example of the itype keyword in a signature:

alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"GPL SCAN Broadscan Smurf Scanner"; dsize:4; icmp_id:0; icmp_seq:0; itype:8; classtype:attempted-recon; sid:2100478; rev:4;)

The following lists all ICMP types known at the time of writing. A recent table can be found at the website of IANA

ICMP Type	Name
0	Echo Reply
3	Destination Unreachable
4	Source Quench
5	Redirect
6	Alternate Host Address
8	Echo
9	Router Advertisement
10	Router Solicitation
11	Time Exceeded
12	Parameter Problem
13	Timestamp
14	Timestamp Reply
15	Information Request
16	Information Reply
17	Address Mask Request
18	Address Mask Reply
30	Traceroute
31	Datagram Conversion Error
32	Mobile Host Redirect
33	IPv6 Where-Are-You
34	IPv6 I-Am-Here
35	Mobile Registration Request
36	Mobile Registration Reply
37	Domain Name Request
38	Domain Name Reply
39	SKIP
40	Photuris
41	Experimental mobility protocols such as Seamoby

8.7.2 icode

With the icode keyword you can match on a specific ICMP code. The code of a ICMP message clarifies the message. Together with the ICMP-type it indicates with what kind of problem you are dealing with. A code has a different purpose with every ICMP-type.

icode uses an *unsigned 8-bit integer*.

The format of the icode keyword:

```
icode:min<>max;
icode:[<|>]<number>;
```

Example: This example looks for an ICMP code greater than 5:

```
icode:>5;
```

Example of the icode keyword in a rule:

alert icmp $HOME_NET any -> $EXTERNAL_NET any (msg:"GPL MISC Time-To-Live Exceeded in Transit"; icode:0; itype:11; classtype:misc-activity; sid:2100449; rev:7;)

The following lists the meaning of all ICMP types. When a code is not listed, only type 0 is defined and has the meaning of the ICMP code, in the table above. A recent table can be found at the website of IANA

ICMP Code	ICMP Type	Description
3	0	Net Unreachable
	1	Host Unreachable
	2	Protocol Unreachable
	3	Port Unreachable
	4	Fragmentation Needed and Don't Fragment was Set
	5	Source Route Failed
	6	Destination Network Unknown
	7	Destination Host Unknown
	8	Source Host Isolated
	9	Communication with Destination Network is Administratively Prohibited
	10	Communication with Destination Host is Administratively Prohibited
	11	Destination Network Unreachable for Type of Service
	12	Destination Host Unreachable for Type of Service
	13	Communication Administratively Prohibited
	14	Host Precedence Violation
	15	Precedence cutoff in effect
5	0	Redirect Datagram for the Network (or subnet)
	1	Redirect Datagram for the Host
	2	Redirect Datagram for the Type of Service and Network
	3	Redirect Datagram for the Type of Service and Host
9	0	Normal router advertisement
	16	Doesn't route common traffic
11	0	Time to Live exceeded in Transit
	1	Fragment Reassembly Time Exceeded
12	0	Pointer indicates the error
	1	Missing a Required Option
	2	Bad Length
40	0	Bad SPI
	1	Authentication Failed
	2	Decompression Failed
	3	Decryption Failed
	4	Need Authentication
	5	Need Authorization

8.7.3 icmp_id

With the icmp_id keyword you can match on specific ICMP id-values. Every ICMP-packet gets an id when it is being send. At the moment the receiver has received the packet, it will send a reply using the same id so the sender will recognize it and connects it with the correct ICMP-request.

icmp_id uses an *unsigned 16-bits integer*.

Format of the icmp_id keyword:

```
icmp_id:<number>;
```

Example: This example looks for an ICMP ID of 0:

```
icmp_id:0;
```

Example of the icmp_id keyword in a rule:

alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"GPL SCAN Broadscan Smurf Scanner"; dsize:4; icmp_id:0; icmp_seq:0; itype:8; classtype:attempted-recon; sid:2100478; rev:4;)

8.7.4 icmp_seq

You can use the icmp_seq keyword to check for a ICMP sequence number. ICMP messages all have sequence numbers. This can be useful (together with the id) for checking which reply message belongs to which request message.

icmp_seq uses an *unsigned 16-bits integer*.

Format of the icmp_seq keyword:

```
icmp_seq:<number>;
```

Example: This example looks for an ICMP Sequence of 0:

```
icmp_seq:0;
```

Example of icmp_seq in a rule:

alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"GPL SCAN Broadscan Smurf Scanner"; dsize:4; icmp_id:0; icmp_seq:0; itype:8; classtype:attempted-recon; sid:2100478; rev:4;)

Note

Some pcap analysis tools, like wireshark, may give both a little endian and big endian value for `icmp_seq`. The `icmp_seq` keyword matches on the big endian value, this is due to Suricata using the network byte order (big endian) to perform the match comparison.

8.7.5 icmpv4.hdr

Sticky buffer to match on the whole ICMPv4 header.

8.7.6 icmpv6.hdr

Sticky buffer to match on the whole ICMPv6 header.

8.7.7 icmpv6.mtu

Match on the ICMPv6 MTU optional value. Will not match if the MTU is not present.

icmpv6.mtu uses an *unsigned 32-bit integer*.

The format of the keyword:

```
icmpv6.mtu:<min>-<max>;
icmpv6.mtu:[<|>]<number>;
icmpv6.mtu:<value>;
```

Example rule:

alert ip $EXTERNAL_NET any -> $HOME_NET any (icmpv6.mtu:<1280; sid:1234; rev:5;)

8.8 IGMP keywords

The Internet Group Management Protocol (IGMP) is the protocol used by IPv4 systems to report their IP multicast group memberships to neighboring multicast routers [RFC 9776].

Additionally, the RGMP protocol is a dialect of IGMP. The keywords below also apply to RGMP. RGMP is defined in RFC 3488.

8.8.1 igmp.hdr

Sticky buffer to match on the whole IGMP header.

Example rule:

alert igmp any any -> any any (igmp.hdr; content:"|22|"; startswith; sid:1234; rev:5;)

8.8.2 igmp.type

Match on the IGMP type field.

`igmp.type` uses an *unsigned 8-bit integer*.

Format:

```
igmp.type:0x11;
```

Example rule:

alert igmp any any -> any any (igmp.type:0x22; sid:1234; rev:5;)

8.8.3 igmp-csum

Match on the validity of the checksum field.

Format:

```
igmp-csum:valid;
igmp-csum:invalid;
```

alert igmp any any -> any any (igmp-csum:invalid; sid:1234; rev:5;)

8.9 Payload Keywords

Payload keywords inspect the content of the payload of a packet or stream.

8.9.1 content

The content keyword is very important in signatures. Between the quotation marks you can write on what you would like the signature to match. The most simple format of content is:

```
content: ".....";
```

It is possible to use several contents in a signature.

Contents match on bytes. There are 256 different values of a byte (0-255). You can match on all characters; from a till z, upper case and lower case and also on all special signs. But not all of the bytes are printable characters. For these bytes heximal notations are used. Many programming languages use 0x00 as a notation, where 0x means it concerns a binary value, however the rule language uses `|00|` as a notation. This kind of notation can also be used for printable characters.

Example:

```
|61| is a
|61 61| is aa
|41| is A
|21| is !
|0D| is carriage return
|0A| is line feed
```

There are characters you can not use in the content because they are already important in the signature. For matching on these characters you should use the heximal notation. These are:

```
"     |22|
;     |3B|
:     |3A|
|     |7C|
```

It is a convention to write the heximal notation in upper case characters.

To write for instance `http://` in the content of a signature, you should write it like this: `content: "http|3A|//";` If you use a heximal notation in a signature, make sure you always place it between pipes. Otherwise the notation will be taken literally as part of the content.

A few examples:

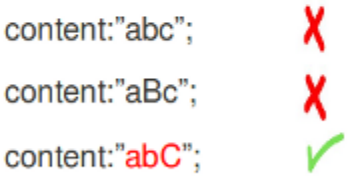
```
content:"a|0D|bc";
content:"|61 0D 62 63|";
content:"a|0D|b|63|";
```

It is possible to let a signature check the whole payload for a match with the content or to let it check specific parts of the payload. We come to that later. If you add nothing special to the signature, it will try to find a match in all the bytes of the payload.

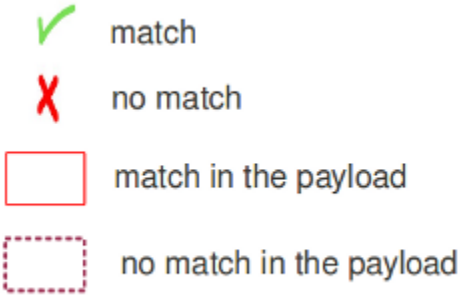
drop tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"ET TROJAN Likely Bot Nick in IRC (USA +..)"; flow:established,to_server; flowbits:isset,is_proto_irc; content:"NICK "; pcre:"/NICK .*USA.*[0-9]{3,}/i"; reference:url,doc.emergingthreats.net/2008124; classtype:trojan-activity; sid:2008124; rev:2;)

By default the pattern-matching is case sensitive. The content has to be accurate, otherwise there will not be a match.





Legend:



It is possible to use the ! for exceptions in contents as well.

For example:

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"Outdated Firefox on Windows"; content:"User-Agent|3A| Mozilla/5.0 |28|Windows|3B| "; content:"Firefox/3."; distance:0; content:!"Firefox/3.6.13"; distance:-10; sid:9000000; rev:1;)

You see `content:!"Firefox/3.6.13";`. This means an alert will be generated if the used version of Firefox is not 3.6.13.

Note

The following characters must be escaped inside the content: `;` `\` `"`

8.9.2 nocase

If you do not want to make a distinction between uppercase and lowercase characters, you can use nocase. The keyword nocase is a content modifier.

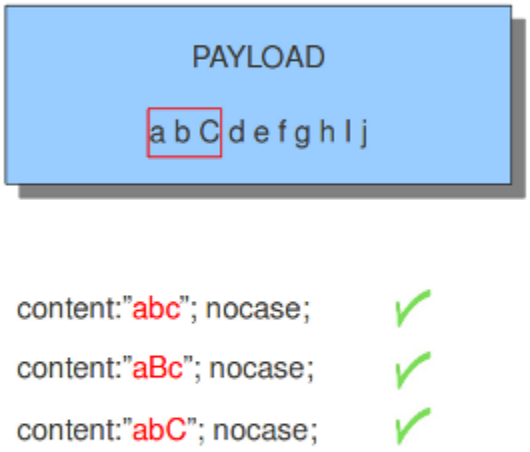
The format of this keyword is:

```
nocase;
```

You have to place it after the content you want to modify, like:

```
content: "abc"; nocase;
```

Example nocase:



It has no influence on other contents in the signature.

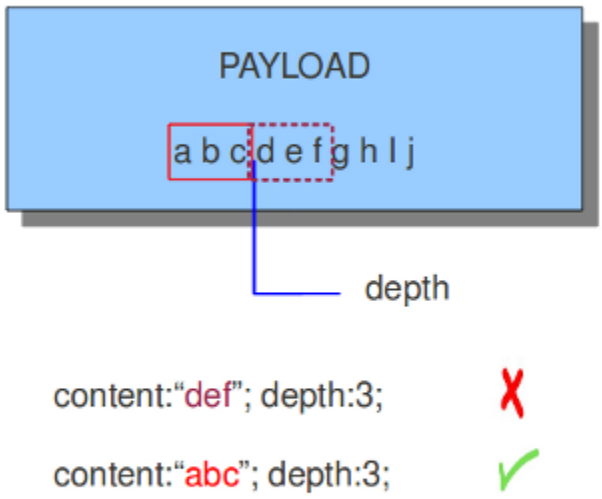
8.9.3 depth

The depth keyword is a absolute content modifier. It comes after the content. The depth content modifier comes with a mandatory numeric value, like:

```
depth:12;
```

The number after depth designates how many bytes from the beginning of the payload will be checked.

Example:



8.9.4 startswith

The `startswith` keyword is similar to `depth`. It takes no arguments and must follow a `content` keyword. It modifies the `content` to match exactly at the start of a buffer.

Example:

```
content:"GET|20|"; startswith;
```

`startswith` is a short hand notation for:

```
content:"GET|20|"; depth:4; offset:0;
```

`startswith` cannot be mixed with `depth`, `offset`, `within` or `distance` for the same pattern.

8.9.5 endswith

The `endswith` keyword is similar to `isdataat:!1,relative;`. It takes no arguments and must follow a `content` keyword. It modifies the `content` to match exactly at the end of a buffer.

Example:

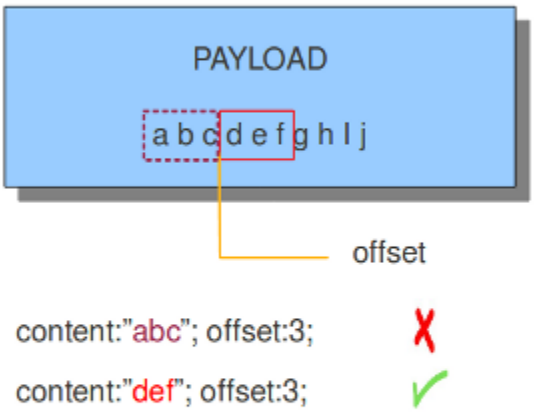
```
content:".php"; endswith;
```

`endswith` is a short hand notation for:

```
content:".php"; isdataat:!1,relative;
```

8.9.6 offset

The offset keyword designates from which byte in the payload will be checked to find a match. For instance offset:3; checks the fourth byte and further.

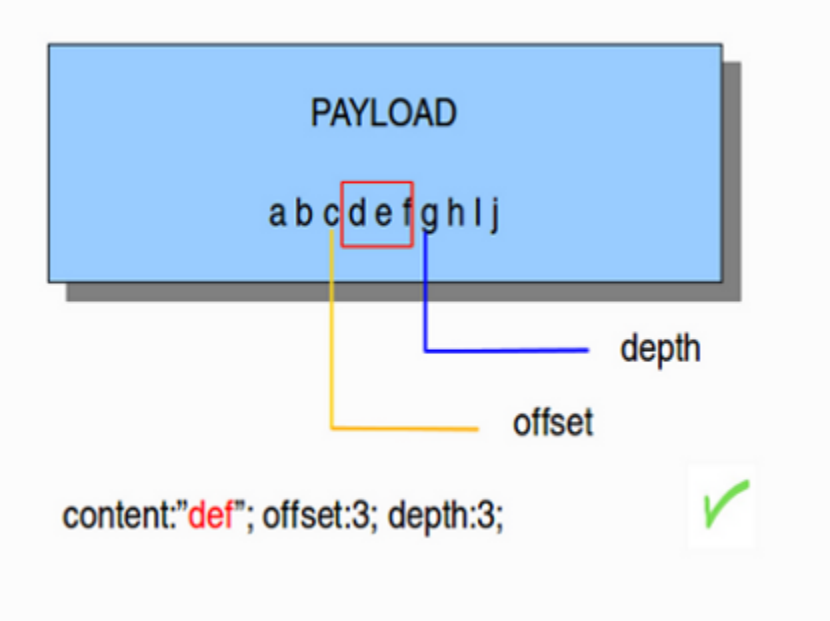


The keywords offset and depth can be combined and are often used together.

For example:

```
content:"def"; offset:3; depth:3;
```

If this was used in a signature, it would check the payload from the third byte till the sixth byte.

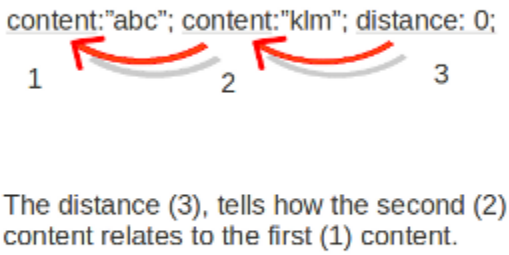


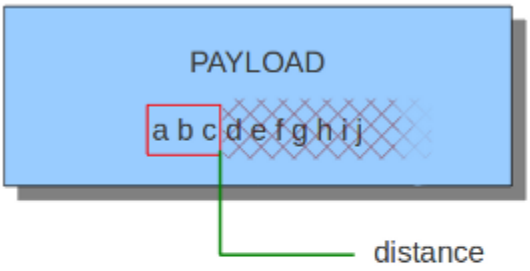
8.9.7 distance

The keyword distance is a relative content modifier. This means it indicates a relation between this content keyword and the content preceding it. Distance has its influence after the preceding match. The keyword distance comes with a mandatory numeric value. The value you give distance, determines the byte in the payload from which will be checked for a match relative to the previous match. Distance only determines where Suricata will start looking for a pattern. So, distance:5; means the pattern can be anywhere after the previous match + 5 bytes. For limiting how far after the last match Suricata needs to look, use 'within'.

The absolute value for distance must be less than or equal to 1MB (1048576).

Examples of distance:

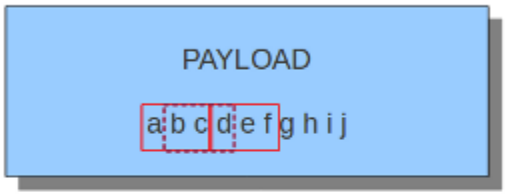




content:"abc"; content:"klm"; distance: 0;

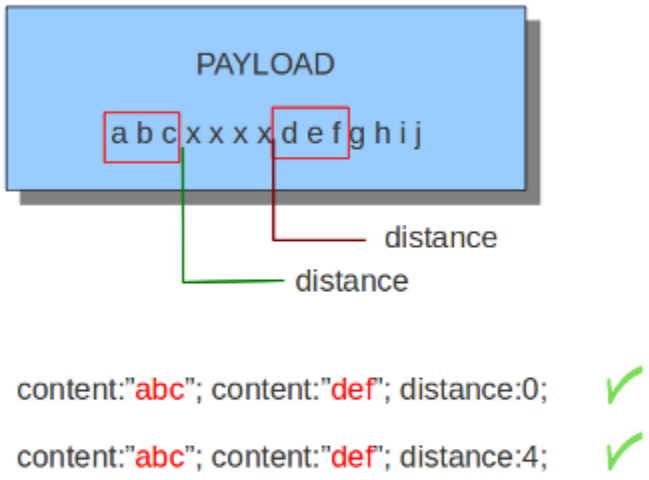


checked area using 'distance'



content:"abc"; content:"def"; distance:0;

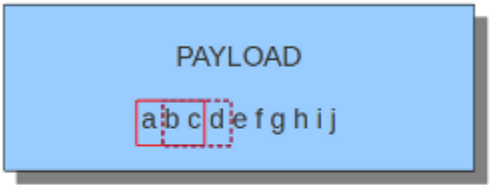
content:"abc"; content:"bcd"; distance:0;



content:"abc"; content:"def"; distance:0;

content:"abc"; content:"def"; distance:4;

Distance can also be a negative number. It can be used to check for matches with partly the same content (see example) or for a content even completely before it. This is not very often used though. It is possible to attain the same results with other keywords.



content:"abc"; content:"bcd"; distance:-2;

8.9.8 within

The keyword within is relative to the preceding match. The keyword within comes with a mandatory numeric value. Using within makes sure there will only be a match if the content matches with the payload within the set amount of bytes. Within can not be 0 (zero)

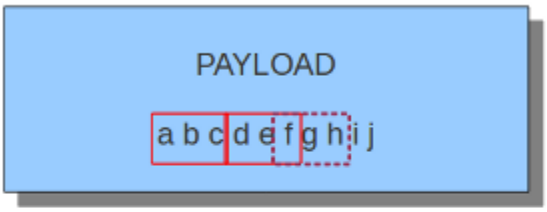
The absolute value for within must be less than or equal to 1MB (1048576).

Example:



The keyword within (3), tells how the second (2) content relates to the first (1) content.

Example of matching with within:

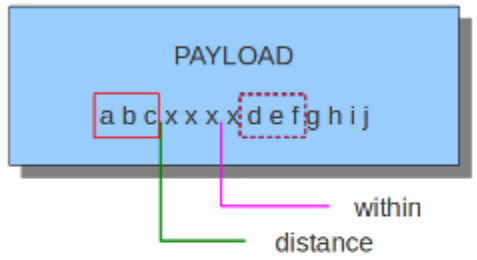


content:"abc"; content:"def"; within:3; ✓

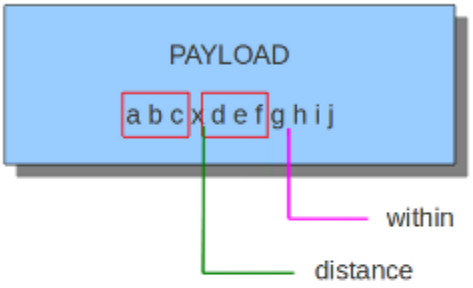
content:"abc"; content:"fgh"; within:3; ✗

The second content has to fall/come 'within 3 ' from the first content.

As mentioned before, distance and within can be very well combined in a signature. If you want Suricata to check a specific part of the payload for a match, use within.



content:"abc"; content:"def"; distance:0; within:3;



content:"abc"; content:"def"; distance:1; within:4;

8.9.9 rawbytes

The rawbytes keyword has no effect but is included to be compatible with signatures that use it, for example signatures used with Snort.

8.9.10 isdataat

The purpose of the isdataat keyword is to look if there is still data at a specific part of the payload. The keyword starts with a number (the position) and then optional followed by 'relative' separated by a comma and the option rawbytes. You use the word 'relative' to know if there is still data at a specific part of the payload relative to the last match.

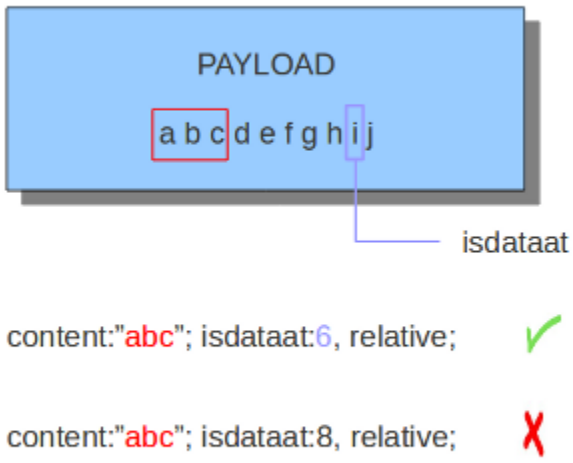
So you can use both examples:

```
isdataat:512;

isdataat:50, relative;
```

The first example illustrates a signature which searches for byte 512 of the payload. The second example illustrates a signature searching for byte 50 after the last match.

You can also use the negation (!) before isdataat.



absolute vs relative values

The absolute `isdataat` checks will succeed if the offset used is **less than** the size of the inspection buffer.

For *relative* `isdataat` checks, there is a **1 byte difference** vs the absolute handling.

Matching will succeed if the relative offset is **less than or equal to** the size of the inspection buffer. This is different from absolute `isdataat` checks.

As an example, consider a 32 byte payload:

rule statement	Match?
`isdataat:31;`	Yes
`isdataat:32;`	No
`isdataat:31,relative;`	Yes
`isdataat:32,relative;`	Yes
`isdataat:33,relative;`	No

Another example, consider the following payload:

Index	0	1	2	3	4	5	6	7
Value	a	b	c	d	e	f	g	h

Then the following rules match the payload as follows:

Rule statement	Match?
`isdataat:7;`	Yes
`isdataat:8;`	No
`isdataat:7,relative;`	Yes
`isdataat:8,relative;`	Yes
`isdataat:9,relative;`	No
`payload:"c"; isdataat:4,relative;`	Yes
`payload:"c"; isdataat:5,relative;`	Yes
`payload:"c"; isdataat:6,relative;`	No

These differences are also discussed in *Differences From Snort*.

A discussion of this difference can be found at https://redmine.openinfosecfoundation.org/issues/8031

8.9.11 absent

The keyword `absent` checks that a sticky buffer does not exist. It can be used without any argument to match only on absent buffer :

Example of `absent` in a rule:

alert http any any -> any any (msg:"HTTP request without referer"; http.referer; absent; sid:1; rev:1;)

It can take an argument "or_else" to match on absent buffer or on what comes next such as negated content, for instance :

alert http any any -> any any (msg:"HTTP request without referer"; http.referer; absent: or_else; content: !"abc"; sid:1; rev:1;)

For files (i.e `file.data`), absent means there are no files in the transaction.

8.9.12 bsize

With the `bsize` keyword, you can match on the length of the buffer. This adds precision to the content match, previously this could have been done with `isdataat`.

bsize uses an *unsigned 64-bit integer*.

An optional operator can be specified; if no operator is present, the operator will default to '='. When a relational operator is used, e.g., '<', '>' or '<>' (range), the bsize value will be compared using the relational operator. Ranges are exclusive.

If one or more `content` keywords precedes `bsize`, each occurrence of `content` will be inspected and an error will be raised if the content length and the bsize value prevent a match.

Format:

```
bsize:<number>;
bsize:=<number>;
bsize:<<number>;
bsize:><number>;
bsize:<lo-number><><hi-number>;
```

Examples of `bsize` in a rule:

alert dns any any -> any any (msg:"bsize exact buffer size"; dns.query; content:"google.com"; bsize:10; sid:1; rev:1;)

alert dns any any -> any any (msg:"bsize less than value"; dns.query; content:"google.com"; bsize:<25; sid:2; rev:1;)

alert dns any any -> any any (msg:"bsize buffer less than or equal value"; dns.query; content:"google.com"; bsize:<=20; sid:3; rev:1;)

alert dns any any -> any any (msg:"bsize buffer greater than value"; dns.query; content:"google.com"; bsize:>8; sid:4; rev:1;)

alert dns any any -> any any (msg:"bsize buffer greater than or equal value"; dns.query; content:"google.com"; bsize:>=8; sid:5; rev:1;)

alert dns any any -> any any (msg:"bsize buffer range value"; dns.query; content:"google.com"; bsize:8<>20; sid:6; rev:1;)

alert dns any any -> any any (msg:"test bsize rule"; dns.query; content:"short"; bsize:<10; sid:124; rev:1;)

alert dns any any -> any any (msg:"test bsize rule"; dns.query; content:"longer string"; bsize:>10; sid:125; rev:1;)

alert dns any any -> any any (msg:"test bsize rule"; dns.query; content:"middle"; bsize:6<>15; sid:126; rev:1;)

To emphasize how range works: in the example above, a match will occur if `bsize` is greater than 6 and less than 15.

8.9.13 dsize

With the dsize keyword, you can match on the size of the packet payload/data. You can use the keyword for example to look for abnormal sizes of payloads which are equal to some n i.e. 'dsize:n' not equal 'dsize:!n' less than 'dsize:<n' or greater than 'dsize:>n' This may be convenient in detecting buffer overflows.

dsize cannot be used when using app/streamlayer protocol keywords (i.e. http.uri)

dsize uses an *unsigned 16-bit integer*.

Format:

```
dsize:[<>!]number; || dsize:min<>max;
```

Examples of dsize values:

alert tcp any any -> any any (msg:"dsize exact size"; dsize:10; sid:1; rev:1;)

alert tcp any any -> any any (msg:"dsize less than value"; dsize:<10; sid:2; rev:1;)

alert tcp any any -> any any (msg:"dsize less than or equal value"; dsize:<=10; sid:3; rev:1;)

alert tcp any any -> any any (msg:"dsize greater than value"; dsize:>8; sid:4; rev:1;)

alert tcp any any -> any any (msg:"dsize greater than or equal value"; dsize:>=10; sid:5; rev:1;)

alert tcp any any -> any any (msg:"dsize range value"; dsize:8<>20; sid:6; rev:1;)

alert tcp any any -> any any (msg:"dsize not equal value"; dsize:!9; sid:7; rev:1;)

8.9.14 byte_test

The `byte_test` keyword extracts <`num of bytes`> and performs an operation selected with <`operator`> against the value in <`test value`> at a particular <`offset`>. The <`bitmask value`> is applied to the extracted bytes (before the operator is applied), and the final result will be right shifted one bit for each trailing `0` in the <`bitmask value`>.

Format:

```
byte_test:<num of bytes> | <variable_name>, [!]<operator>, <test value>, <offset> [,
↪relative] \
[,<endian>][, string, <num type>][, dce][, bitmask <bitmask value>];
```

<num of bytes>	The number of bytes selected from the packet to be converted or the name of a byte_extract/byte_math variable.
<operator>	• [!] Negation can prefix other operators • < less than • > greater than • = equal • <= less than or equal • >= greater than or equal • & bitwise AND • ^ bitwise OR
<value>	Value to test the converted value against [hex or decimal accepted]
<offset>	Number of bytes into the payload
[relative]	Offset relative to last content match
[endian]	Type of number being read: - big (Most significant byte at lowest address) - little (Most significant byte at the highest address)
[string] <num>	• hex - Converted string represented in hex • dec - Converted string represented in decimal • oct - Converted string represented in octal
[dce]	Allow the DCE module to determine the byte order
[bitmask]	Applies the AND operator on the bytes converted

alert tcp any any -> any any
(msg:"Byte_Test Example - Num = Value"; content:"|00 01 00 02|"; byte_test:2,=,0x01,0;)

alert tcp any any -> any any
(msg:"Byte_Test Example - Num = Value relative to content"; content:"|00 01 00 02|"; byte_test:2,=,0x03,2,relative;)

alert tcp any any -> any any
(msg:"Byte_Test Example - Num != Value"; content:"|00 01 00 02|"; byte_test:2,!=,0x06,0;)

alert tcp any any -> any any
(msg:"Byte_Test Example - Detect Large Values"; content:"|00 01 00 02|"; byte_test:2,>,1000,1,relative; sid:1;)

alert tcp any any -> any any
(msg:"Byte_Test Example - Lowest bit is set"; content:"|00 01 00 02|"; byte_test:2,&,0x01,12,relative;)

alert tcp any any -> any any (msg:"Byte_Test Example - Compare to String";
content:"foobar"; byte_test:4,=,1337,1,relative,string,dec; sid:1;)

8.9.15 byte_math

The `byte_math` keyword adds the capability to perform mathematical operations on extracted values with an existing variable or a specified value.

When `relative` is included, there must be a previous `content` or `pcre` match.

Note: if `oper` is `/` and the divisor is 0, there will never be a match on the `byte_math` keyword.

Note: if `oper` is << or >> and `rvalue` is 64 or greater, the result is 0.

The result can be stored in a result variable and referenced by other rule options later in the rule.

Keyword	Modifier
content	offset,depth,distance,within
byte_test	offset,value
byte_jump	offset
isdataat	offset

Format:

```
byte_math:bytes <num of bytes> | <variable-name> , offset <offset>, oper <operator>,␣
↪rvalue <rvalue>, \
      result <result_var> [, relative] [, endian <endian>] [, string <number-type>] \
      [, dce] [, bitmask <value>];
```

<num of bytes>	The number of bytes selected from the packet or the name of a byte_extract variable.
<offset>	Number of bytes into the payload
oper <operator>	Mathematical operation to perform: +, -, *, /, <<, >>
rvalue <rvalue>	Value to perform the math operation with
result <result-var>	Where to store the computed value
[relative]	Offset relative to last content match
[endian <type>]	• big (Most significant byte at lowest address) • little (Most significant byte at the highest address) • dce (Allow the DCE module to determine the byte order)
[string <num_type>]	• hex Converted data is represented in hex • dec Converted data is represented in decimal • oct Converted data is represented as octal
[dce]	Allow the DCE module to determine the byte order
[bitmask] <value>	The AND operator will be applied to the extracted value The result will be right shifted by the number of bits equal to the number of trailing zeros in the mask

alert tcp any any -> any any
(msg:"Testing bytemath_body"; content:"|00 04 93 F3|"; content:"|00 00 00 07|"; distance:4; within:4; byte_math:bytes 4, offset 0, oper +, rvalue 248, result var, relative; sid: 1;)

alert udp any any -> any any
(byte_extract: 1, 0, extracted_val, relative; byte_math: bytes 1, offset 1, oper +, rvalue extracted_val, result var; byte_test: 2, =, var, 13; msg:"Byte extract and byte math with byte test verification"; sid: 1;)

8.9.16 byte_jump

The `byte_jump` keyword allows for the ability to select a `<num of bytes>` from an `<offset>` and moves the detection pointer to that position. Content matches will then be based off the new position. The `bitmask` value is applied to the extracted value before the multiplier. Additionally, the result of the bitmask operation is right-shifted by the number of trailing zeroes in the bitmask value.

Format:

```
byte_jump:<num of bytes> | <variable-name>, <offset> [, relative][, multiplier <mult_
↪value>] \
```

(continues on next page)

(continued from previous page)

```
      [, <endian>][, string, <num_type>][, align][, from_beginning][, from_end] \
      [, post_offset <value>][, dce][, bitmask <value>];
```

<num of bytes>	The number of bytes selected from the packet to be converted or the name of a byte_extract/byte_math variable.
<offset>	Number of bytes into the payload
[relative]	Offset relative to last content match
[multiplier] <value>	Multiple the converted byte by the <value>
[endian]	• big (Most significant byte at lowest address) • little (Most significant byte at the highest address)
[string] <num_type>	• hex Converted data is represented in hex • dec Converted data is represented in decimal • oct Converted data is represented as octal
[align]	Rounds the number up to the next 32bit boundary
[from_beginning]	Jumps forward from the beginning of the packet, instead of where the detection pointer is set
[from_end]	Jump will begin at the end of the payload, instead of where the detection point is set
[post_offset] <value>	After the jump operation has been performed, it will jump an additional number of bytes specified by <value>
[dce]	Allow the DCE module to determine the byte order
[bitmask] <value>	The AND operator will be applied by <value> and the converted bytes, then jump operation is performed

alert tcp any any -> any any
(msg:"Byte_Jump Example"; content:"Alice"; byte_jump:2,0; content:"Bob";)

alert tcp any any -> any any
(msg:"Byte_Jump Multiple Jumps"; byte_jump:2,0; byte_jump:2,0,relative; content:"foobar"; distance:0; within:6;)

alert tcp any any -> any any
(msg:"Byte_Jump From the End -8 Bytes"; byte_jump:0,0, from_end, post_offset -8; content:"|6c 33 33 74|"; distance:0 within:4;)

8.9.17 byte_extract

The `byte_extract` keyword extracts <`num of bytes`> at a particular <`offset`> and stores it in <`var_name`>. The value in <`var_name`> can be used in any modifier that takes a number as an option and in the case of `byte_test` it can be used as a value.

Format:

```
byte_extract:<num of bytes>, <offset>, <var_name>, [,relative] [,multiplier <mult-value>
↪] \
      [,<endian>] [, dce] [, string [, <num_type>] [, align <align-value];
```

<num of bytes>	The number of bytes selected from the packet to be extracted
<offset>	Number of bytes into the payload
<var_name>	The name of the variable in which to store the value
[relative]	Offset relative to last content match
multiplier <value>	multiply the extracted bytes by <mult-value> before storing
[endian]	Type of number being read: - big (Most significant byte at lowest address) - little (Most significant byte at the highest address)
[string] <num>	• hex - Converted string represented in hex • dec - Converted string represented in decimal • oct - Converted string represented in octal
[dce]	Allow the DCE module to determine the byte order
align <align-value>	Round the extracted value up to the next <align-value> byte boundary post-multiplication (if any) ; <align-value> may be 2 or 4

Keyword	Modifier
content	offset,depth,distance,within
byte_test	offset,value
byte_math	rvalue
byte_jump	offset
isdataat	offset

alert tcp any any -> any any
(msg:"Byte_Extract Example Using distance"; content:"Alice"; byte_extract:2,0,size; content:"Bob"; distance:size; within:3; sid:1;)

alert tcp any any -> any any
(msg:"Byte_Extract Example Using within"; flow:established,to_server; content:"|00 FF|"; byte_extract:1,0,len,relative; content:"|5c 00|"; distance:2; within:len; sid:2;)

alert tcp any any -> any any
(msg:"Byte_Extract Example Comparing Bytes"; flow:established,to_server; content:"|00 FF|"; byte_extract:2,0,cmp_ver,relative; content:"FooBar"; distance:0; byte_test:2,=,cmp_ver,0; sid:3;)

8.9.18 entropy

The `entropy` keyword calculates the Shannon entropy value for content and compares it with an entropy value. When there is a match, rule processing will continue. Entropy values are between 0.0 and 8.0, inclusive. Internally, entropy is represented as a 64-bit floating point value.

The `entropy` keyword syntax is the keyword entropy followed by options and the entropy value and operator used to determine if the values agree.

The minimum entropy keyword specification is:

```
entropy: value <entropy-spec>
```

This results in the calculated entropy value being compared with *entropy-spec* using the (default) equality operator.

Example:

```
entropy: value 7.01
```

A match occurs when the calculated entropy and specified entropy values agree. This is determined by calculating the entropy value and comparing it with the value from the rule using the specified operator.

Example:

```
entropy: value <7.01
```

Options have default values: - bytes is equal to the current content length - offset is 0 - equality comparison

When entropy keyword options are specified, all options and "value" must be comma-separated. Options and value may be specified in any order.

The complete format for the `entropy` keyword is:

```
entropy: [bytes <byteval>] [offset <offsetval>] value <operator><entropy-value>
```

This example shows all possible options with default values and an entropy value of *4.037*:

```
entropy: bytes 0, offset 0,  value = 4.037
```

The following operators are available:

```
* = (default): Match when calculated value equals entropy value
* < Match when calculated value is strictly less than entropy value
* <=  Match when calculated value is less than or equal to entropy value
* > Match when calculated value is strictly greater than entropy value
* >= Match when calculated value is greater than or equal to entropy value
* !=  Match when calculated value is not equal to entropy value
* x-y Match when calculated value is within the exclusive range
* !x-y Match when calculated value is not within the exclusive range
```

This example matches if the *file.data* content for an HTTP transaction has a Shannon entropy value of 4 or higher:

```
alert http any any -> any any (msg:"entropy simple test"; file.data; entropy: value >= 4;
↪ sid:1;)
```

Logging

When the `entropy` rule keyword is provided and the rule is evaluated, the *calculated entropy* value is associated with the flow even if the calculated entropy value didn't result in a match or alert. Subsequent logging of event types that include the flow, including alerts, will contain the `entropy` value in the `metadata` section of an output log. The follow is an example that shows the calculated entropy value with the buffer on which the value was computed:

```
"metadata": {
   "entropy": {
     "file_data": 4.265743301617466
   }
 }
```

The events where entropy is logged will depend largely on how it's used within a rule and the rule's protocol.

For example -- this rule -- when evaluated by Suricata -- will result in the *calculated entropy* being included in the `alert, flow` and `http` events. Depending on the traffic and Suricata configuration, other event types may include the entropy value:

```
alert http any any -> any any (flow:established; file.data; entropy: value > 4.4; sid: 1;
↪)
```

8.9.19 rpc

The rpc keyword can be used to match in the SUNRPC CALL on the RPC procedure numbers and the RPC version.

You can modify the keyword by using a wild-card, defined with * With this wild-card you can match on all version and/or procedure numbers.

RPC (Remote Procedure Call) is an application that allows a computer program to execute a procedure on another computer (or address space). It is used for inter-process communication. See http://en.wikipedia.org/wiki/Inter-process_communication

Format:

```
rpc:<application number>, [<version number>|*], [<procedure number>|*]>;
```

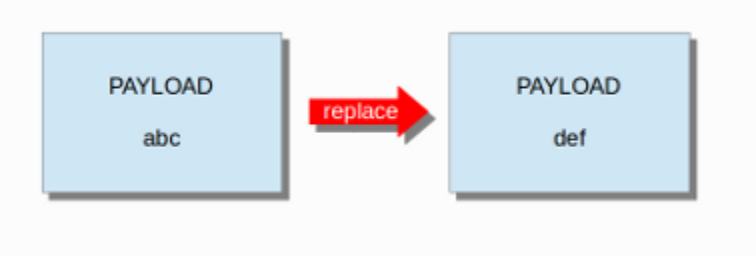
Example of the rpc keyword in a rule:

alert udp $EXTERNAL_NET any -> $HOME_NET 111 (msg:"RPC portmap request yppasswdd"; rpc:100009,*,*; reference:bugtraq,2763; classtype:rpc-portmap-decode; sid:1296; rev:4;)

8.9.20 replace

The replace content modifier can only be used in IPS. It adjusts network traffic. It changes the content it follows ('abc') into another ('def'), see example:





The replace modifier has to contain as many characters as the content it replaces. It can only be used with individual packets. It will not work for *Normalized Buffers* like HTTP uri or a content match in the reassembled stream.

The checksums will be recalculated by Suricata and changed after the replace keyword is being used.

> **Note**
>
> `replace` cannot be used in firewall mode, even if only in Threat Detection rules.

8.9.21 pcre (Perl Compatible Regular Expressions)

The keyword pcre matches specific on regular expressions. More information about regular expressions can be found here http://en.wikipedia.org/wiki/Regular_expression.

The complexity of pcre comes with a high price though: it has a negative influence on performance. So, to mitigate Suricata from having to check pcre often, pcre is mostly combined with 'content'. In that case, the content has to match first, before pcre will be checked.

Format of pcre:

```
pcre:"/<regex>/opts";
```

Example of pcre. In this example there will be a match if the payload contains six numbers following:

```
pcre:"/[0-9]{6}/";
```

Example of pcre in a signature:

drop tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"ET TROJAN Likely Bot Nick in IRC (USA +..)"; flow:established,to_server; flowbits:isset,is_proto_irc; content:"NICK "; pcre:"/NICK .*USA.*[0-9]{3,}/i"; reference:url,doc.emergingthreats.net/2008124; classtype:trojan-activity; sid:2008124; rev:2;)

There are a few qualities of pcre which can be modified:

- By default pcre is case-sensitive.
- The . (dot) is a part of regex. It matches on every byte except for newline characters.
- By default the payload will be inspected as one line.

These qualities can be modified with the following characters:

```
i    pcre is case insensitive
s    pcre does check newline characters
m    can make one line (of the payload) count as two lines
```

These options are perl compatible modifiers. To use these modifiers, you should add them to pcre, behind regex. Like this:

```
pcre: "/<regex>/i";
```

Pcre compatible modifiers

There are a few pcre compatible modifiers which can change the qualities of pcre as well. These are:

- `A`: A pattern has to match at the beginning of a buffer. (In pcre ^ is similar to A.)
- `E`: Ignores newline characters at the end of the buffer/payload.
- `G`: Inverts the greediness.

Note

The following characters must be escaped inside the content: `;` `\` `"`

PCRE extraction

It is possible to capture groups from the regular expression and log them into the alert events.

There are 3 capabilities:

- pkt: the extracted group is logged as pkt variable in `metadata.pktvars`
- alert: the extracted group is logged to the `alert.context` subobject
- flow: the extracted group is stored in a flow variable and end up in the `metadata.flowvars`

To use the feature, parameters of pcre keyword need to be updated. After the regular pcre regex and options, a comma-separated list of variable names. The prefix here is `flow:`, `pkt:` or `alert:` and the names can contain special characters now. The names map to the capturing substring expressions in order

```
pcre:"/([a-z]+)\/[a-z]+\/(.+)\/(.+)\/changelog$/GUR, \
    flow:ua/ubuntu/repo,flow:ua/ubuntu/pkg/base,     \
    flow:ua/ubuntu/pkg/version";
```

This would result in the alert event having something like

```
"metadata": {
  "flowvars": [
     {"ua/ubuntu/repo": "fr"},
     {"ua/ubuntu/pkg/base": "curl"},
     {"ua/ubuntu/pkg/version": "2.2.1"}
  ]
}
```

The other events on the same flow such as the `flow` one will also have the flow vars.

If this is not wanted, you can use the `alert:` construct to only get the event in the alert

```
pcre:"/([a-z]+)\/[a-z]+\/(.+)\/(.+)\/changelog$/GUR, \
    alert:ua/ubuntu/repo,alert:ua/ubuntu/pkg/base,     \
    alert:ua/ubuntu/pkg/version";
```

With that syntax, the result of the extraction will appear like

```
"alert": {
  "context": {
     "ua/ubuntu/repo": "fr",
     "ua/ubuntu/pkg/base": "curl",
     "ua/ubuntu/pkg/version": "2.2.1"
  ]
}
```

A combination of the extraction scopes can be combined.

> **Note**
>
> A `flow:` capture needs a flow to store into. Some packets are inspected without one -- ICMPv4 error messages, for example, or a packet that could not be given a flow because the flow memcap was reached. Those packets can still alert, but their `flow:` captures are dropped.

It is also possible to extract key/value pair in the `pkt` scope. One capture would be the key, the second the value. The notation is similar to the last

```
pcre:"^/([A-Z]+) (.*)\r\n/, pkt:key,pkt:value";
```

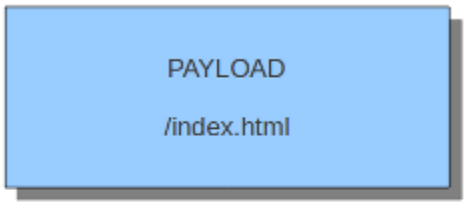
`key` and `value` are simply hardcoded names to trigger the key/value extraction. As a consequence, they can't be used as name for the variables.

Suricata's modifiers

Suricata has its own specific pcre modifiers. These are:

- R: Match relative to the last pattern match. It is similar to distance:0;

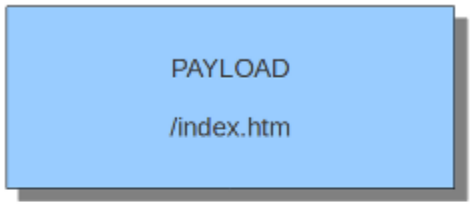
- U: Makes pcre match on the normalized uri. It matches on the uri_buffer just like uricontent and content combined with http_uri.U can be combined with /R. Note that R is relative to the previous match so both matches have to be in the HTTP-uri buffer. Read more about *HTTP URI Normalization*.



content:"/index."; http_uri; content:"htm"; http_uri; distance:0; ✓

content:"index."; http_uri; pcre:"/html?$/UR"; ✓

content:"index."; http_uri; pcre:"/^/index\.html?/$/U"; ✓



content:"/index."; http_uri; content:"htm"; http_uri; distance:0; ✓

content:"index."; http_uri; pcre:"/html?$/UR"; ✓

content:"index."; http_uri; pcre:"/^/index\.html?/$/U"; ✓



content:"/index."; http_uri; content:"htm"; http_uri; distance:0; ✓

content:"index."; http_uri; pcre:"/html?$/UR"; ✗

content:"index."; http_uri; pcre:"/^/index\.html?/$/U"; ✗



content:"/index."; http_uri; content:"htm"; http_uri; distance:0; ✗

content:"index."; http_uri; pcre:"/html?$/UR"; ✓

content:"index."; http_uri; pcre:"/^/index\.html?/$/U"; ✗

- `I`: Makes pcre match on the HTTP-raw-uri. It matches on the same buffer as http_raw_uri. I can be combined with /R. Note that R is relative to the previous match so both matches have to be in the HTTP-raw-uri buffer. Read more about *HTTP URI Normalization*.
- `P`: Makes pcre match on the HTTP- request-body. So, it matches on the same buffer as http_client_body. P can be combined with /R. Note that R is relative to the previous match so both matches have to be in the HTTP-request body.
- `Q`: Makes pcre match on the HTTP- response-body. So, it matches on the same buffer as http_server_body. Q can be combined with /R. Note that R is relative to the previous match so both matches have to be in the HTTP-response body.
- `H`: Makes pcre match on the HTTP-header. H can be combined with /R. Note that R is relative to the previous match so both matches have to be in the HTTP-header body.
- `D`: Makes pcre match on the unnormalized header. So, it matches on the same buffer as http_raw_header. D can be combined with /R. Note that R is relative to the previous match so both matches have to be in the HTTP-raw-header.
- `M`: Makes pcre match on the request-method. So, it matches on the same buffer as http_method. M can be combined with /R. Note that R is relative to the previous match so both matches have to be in the HTTP-method buffer.
- `C`: Makes pcre match on the HTTP-cookie. So, it matches on the same buffer as http_cookie. C can be combined with /R. Note that R is relative to the previous match so both matches have to be in the HTTP-cookie buffer.
- `S`: Makes pcre match on the HTTP-stat-code. So, it matches on the same buffer as http_stat_code. S can be combined with /R. Note that R is relative to the previous match so both matches have to be in the HTTP-stat-code buffer.
- `Y`: Makes pcre match on the HTTP-stat-msg. So, it matches on the same buffer as http_stat_msg. Y can be combined with /R. Note that R is relative to the previous match so both matches have to be in the HTTP-stat-msg buffer.
- `B`: You can encounter B in signatures but this is just for compatibility. So, Suricata does not use B but supports it so it does not cause errors.
- `O`: Overrides the configures pcre match limit.
- `V`: Makes pcre match on the HTTP-User-Agent. So, it matches on the same buffer as http_user_agent. V can be combined with /R. Note that R is relative to the previous match so both matches have to be in the HTTP-User-Agent buffer.

- `W`: Makes pcre match on the HTTP-Host. So, it matches on the same buffer as http_host. W can be combined with /R. Note that R is relative to the previous match so both matches have to be in the HTTP-Host buffer.

Changes from PCRE1 to PCRE2

The upgrade from PCRE1 to PCRE2 changes the behavior for some PCRE expressions.

- `\I` is a valid pcre in PCRE1, with a useless escape, so equivalent to `I`, but it is no longer the case in PCRE2. There are other characters than I exhibiting this pattern
- `[\d-a]` is a valid pcre in PCRE1, with either a digit, a dash or the character `a`, but the dash must now be escaped with PCRE2 as `[\d\-a]` to get the same behavior
- `pcre2_substring_copy_bynumber` now returns an error `PCRE2_ERROR_UNSET` instead of `pcre_copy_substring` returning no error and giving an empty string. If the behavior of some use case is no longer the expected one, please let us know.

8.10 Integer Keywords

Many keywords will match on an integer value on the network traffic. These are unsigned integers that can be 8, 16, 32 or 64 bits.

Simple example:

```
bsize:integer value;
```

The integer value can be written as base-10 like `100` or as an hexadecimal value like `0x64`.

The integer value can also have a unit/multiplier as a case-insensitive suffix: * kb/kib : 1024 * mb/mib : 1048576 * gb/gib : 1073741824

The most direct example is to match for equality, but there are different modes.

The list of integer keywords can be found by command `suricata --list-keywords=csv | grep "uint"`

Some other keywords may use unsigned integers as part of their logic: * iprep * stream_size * flow.pkts * flow.bytes

8.10.1 Comparison modes

Integers can be matched for

- Equality
- Inequality
- Greater than
- Less than
- Range
- Negated range
- Bitmask
- Negated Bitmask

Note

Comparisons are strict by default. Ranges are thus exclusive. That means a range between 1 and 4 will match 2 and 3, but neither 1 nor 4. Negated range !1-4 will match for 1 or below and for 4 or above.

Examples:

```
bsize:19; # equality
bsize:=0x13; # equality
bsize:!0x14; # inequality
bsize:!=20; # inequality
bsize:>21; # greater than
bsize:>=21; # greater than or equal
bsize:<22; # lesser than
bsize:<=22; # lesser than or equal
bsize:19-22; # range between value1 and value2
bsize:!19-22; # negated range between value1 and value2
bsize:&0xc0=0x80; # bitmask mask is compared to value for equality
bsize:&0xc0!=0; # bitmask mask is compared to value for inequality
```

8.10.2 Enumerations

Some integers on the wire represent an enumeration, that is, some values have a string/meaning associated to it. Rules can be written using one of these strings to check for equality or inequality. This is meant to make rules more human-readable and equivalent for matching.

Possible string values for a keyword can be found by running `suricata --list-keywords=json | jq '."keyword".enum_values'`

Examples:

```
websocket.opcode:text;
websocket.opcode:1; # behaves the same

websocket.opcode:!ping;
websocket.opcode:!9; # behaves the same
```

8.10.3 Bitmasks

Some integers on the wire represent multiple bits. Some of these bits have a string/meaning associated to it. Rules can be written using a list (comma-separated) of these strings, where each item can be negated. This list can be prefixed by a modifier:

Modifier	Description
+	match on all the bits, plus any others
*	match if any of the bits are set
-	match if not all the bits are set
=	match on all the bits, and only them

There is no right shift for trailing zeros applied here (even if there is one for `byte_test` and `byte_math`). That means a rule with `websocket.flags:&0xc0=2` will be rejected as invalid as it can never match.

Examples:

```
websocket.flags:fin,!comp;
websocket.flags:&0xc0=0x80; # behaves the same
```

8.10.4 Multi-integers

As *multi-buffers* and sticky buffers, some integer keywords are also multi-integer.

They expand the syntax of a single integer::
keyword: operation and value[,index,subslice];

Table 2: **Index values for multi-integers keyword**

Value	Description
[default]	Match with any index
any	Match with any index
all	Match only if all and at least one indexes match
all_or_absent	Match only if all indexes match or matches on an empty list
nb x	Matches a number of times
or_absent	Match with any index or matches on an empty list
0>=	Match specific index
0<	Match specific index with back to front indexing
oob_or x	Match with specific index or index out of bounds

Please note that:

The index `all` will not match if there is no value.

The index `all_or_absent` will match if there is no value and behaves like `all` if there is at least one value.

These keywords will wait for transaction completion to run, to be sure to have the final number of elements.

The index `nb` accepts all comparison modes as integer keywords. For example `nb>3` will match only if more than 3 integers in the array match the value.

The subslice may use positive or negative indexing. For the array [1,2,3,4,5,6], here are some examples: * 2:4 will have subslice [3,4] * -4:-1 will have subslice [3,4,5] * 3:-1 will have subslice [4,5] * -4:4 will have subslice [3,4]

If one index is out of bounds, an empty subslice is used.

8.10.5 Count

Multi-integer can also just count the number of occurences without matching to a specific value.

The syntax is::
keyword: count [mode] value;

Examples:

```
http2.window:count >5;
```

8.11 Transformations

Transformation keywords turn the data at a sticky buffer into something else. Some transformations support options for greater control over the transformation process

alert http any any -> any any (file_data; strip_whitespace;
content:"window.navigate("; sid:1;)

This example will match on traffic even if there are one or more spaces between the `navigate` and `(`.

The transforms can be chained. They are processed in the order in which they appear in a rule. Each transform's output acts as input for the next one.

alert http any any -> any any (http_request_line; compress_whitespace; to_sha256;
content:"|54A9 7A8A B09C 1B81 3725 2214 51D3 F997 F015 9DD7 049E E5AD CED3 945A FC79 7401|";
sid:1;)

Note

not all sticky buffers support transformations yet

8.11.1 dotprefix

Takes the buffer, and prepends a `.` character to help facilitate concise domain checks. For example, an input string of `hello.google.com` would be modified and become `.hello.google.com`. Additionally, adding the dot allows `google.com` to match against `content:".google.com"`

alert dns any any -> any any (dns.query; dotprefix;
content:".microsoft.com"; sid:1;)

This example will match on `windows.update.microsoft.com` and `maps.microsoft.com.au` but not `windows.update.fakemicrosoft.com`.

This rule can be used to match on the domain only; example:

alert dns any any -> any any (dns.query; dotprefix;
content:".microsoft.com"; endswith; sid:1;)

This example will match on `windows.update.microsoft.com` but not `windows.update.microsoft.com.au`.

Finally, this rule can be used to match on the TLD only; example:

alert dns any any -> any any (dns.query; dotprefix;
content:".co.uk"; endswith; sid:1;)

This example will match on `maps.google.co.uk` but not `maps.google.co.nl`.

8.11.2 domain

Takes the buffer and extracts the domain name from it. The domain name is defined using Mozilla's Public Suffix List. This implies that it is using traditional top level domain such as `.com` but also some specific domain like `airport.aero` or `execute-api.cn-north-1.amazonaws.com.cn` where declaration of sub domain by users below the domain is possible.

Example:

```
alert tls any any -> any any (tls.sni; domain; \
    dataset:isset,domains,type string,load domains.lst; sid:1;)
```

This example will match on all domains contained in the file `domains.lst`. For example, if `domains.lst` contains `oisf.net` then `webshop.oisf.net` will match.

8.11.3 tld

Takes the buffer and extracts the Top Level Domain (TLD) name from it. The TLD name is defined using Mozilla's Public Suffix List. This implies that it is will have traditional TLD such as `com` but also some specific domain like `airport.aero` or `execute-api.cn-north-1.amazonaws.com.cn` where declaration of sub domain by users below the domain is possible.

Example:

```
alert tls any any -> any any (tls.sni; tld; \
    dataset:isset,tlds,type string,load tlds.lst; sid:1;)
```

This example will match on all TLDs contained in the file `tlds.lst`. For example, if `tlds.lst` contains `net` then `oisf.net` will match.

8.11.4 strip_whitespace

Strips all whitespace as considered by the `isspace()` call in C.

alert http any any -> any any (file_data; strip_whitespace;
content:"window.navigate("; sid:1;)

8.11.5 compress_whitespace

Compresses all consecutive whitespace into a single space.

8.11.6 to_lowercase

Converts the buffer to lowercase and passes the value on.

This example alerts if `http.uri` contains `this text has been converted to lowercase`

alert http any any -> any any (http.uri; to_lowercase;
content:"this text has been converted to lowercase"; sid:1;)

8.11.7 to_md5

Takes the buffer, calculates the MD5 hash and passes the raw hash value on.

alert http any any -> any any (http_request_line; to_md5;
content:"|54 A9 7A 8A B0 9C 1B 81 37 25 22 14 51 D3 F9 97|"; sid:1;)

8.11.8 to_uppercase

Converts the buffer to uppercase and passes the value on.

This example alerts if `http.uri` contains `THIS TEXT HAS BEEN CONVERTED TO UPPERCASE`

alert http any any -> any any (http.uri; to_uppercase;
content:"THIS TEXT HAS BEEN CONVERTED TO UPPERCASE"; sid:1;)

8.11.9 to_sha1

Takes the buffer, calculates the SHA-1 hash and passes the raw hash value on.

alert http any any -> any any (http_request_line; to_sha1;
content:"|54A9 7A8A B09C 1B81 3725 2214 51D3 F997 F015 9DD7|"; sid:1;)

8.11.10 to_sha256

Takes the buffer, calculates the SHA-256 hash and passes the raw hash value on.

alert http any any -> any any (http_request_line; to_sha256;
content:"|54A9 7A8A B09C 1B81 3725 2214 51D3 F997 F015 9DD7 049E E5AD CED3 945A FC79 7401|"; sid:1;)

8.11.11 pcrexform

Takes the buffer, applies the required regular expression, and outputs the *first captured expression*.

> **Note**
>
> this transform requires a mandatory option string containing a regular expression.

This example alerts if `http.request_line` contains `/dropper.php`:

alert http any any -> any any (msg:"HTTP with pcrexform"; http.request_line;
pcrexform:"[a-zA-Z]+s+(.*)s+HTTP"; content:"/dropper.php"; sid:1;)

8.11.12 url_decode

Decodes url-encoded data, ie replacing '+' with space and '%HH' with its value. This does not decode unicode '%uZZZZ' encoding

8.11.13 xor

Takes the buffer, applies xor decoding.

The key can be a hexadecimal string or a variable specified inline using `extract <nbytes> <offset>`. When a variable key is used, the engine reads `<nbytes>` bytes starting at `<offset>` in the raw inspection buffer at transform time.

An optional `offset` parameter specifies the byte position in the buffer where XOR decoding starts. Bytes before this position are left as-is. For example, if the first byte of the buffer is the XOR key, use `offset 1` so decoding begins after that key byte.

Syntax:

```
xor:"<hex_key>"
xor:extract <nbytes> <offset>
xor:offset <N>,"<hex_key>"
xor:offset <N>,extract <nbytes> <offset>
```

Quotes around a hex key are optional; `xor:0d0ac8ff` and `xor:"0d0ac8ff"` are equivalent.

This example alerts if `http.uri` contains `password=` xored with 4-bytes key `0d0ac8ff`:

alert http any any -> any any (msg:"HTTP with xor"; http.uri;
xor:"0d0ac8ff"; content:"password="; sid:1;)

This example reads a 1-byte XOR key from offset 0 of the request body, then decodes the buffer starting at offset 1 (skipping the key byte) and matches `infected` in the decoded data:

alert http any any -> any any (msg:"XOR with variable key";
http.request_body; xor:offset 1,extract 1 0; content:"infected"; sid:2;)

8.11.14 header_lowercase

This transform is meant for HTTP/1 HTTP/2 header names normalization. It lowercases the header names, while keeping untouched the header values.

The implementation uses a state machine : - it lowercases until it finds `:` - it does not change until it finds a new line and switch back to first state

This example alerts for both HTTP/1 and HTTP/2 with a authorization header:

alert http any any -> any any (msg:"HTTP authorization"; http.header_names;
header_lowercase; content:"authorization:"; sid:1;)

8.11.15 strip_pseudo_headers

This transform is meant for HTTP/1 HTTP/2 header names normalization. It strips HTTP2 pseudo-headers (names and values).

The implementation just strips every line beginning by `:`.

This example alerts for both HTTP/1 and HTTP/2 with only a user agent:

alert http any any -> any any (msg:"HTTP ua only"; http.header_names;
bsize:16; content:"|0d 0a|User-Agent|0d 0a 0d 0a|"; nocase; sid:1;)

8.11.16 from_base64

This transform is similar to the keyword `base64_decode`: the buffer is decoded using the optional values for `mode`, `offset` and `bytes` and is available for matching on the decoded data.

After this transform completes, the buffer will contain only bytes that could be bases64-decoded. If the decoding process encountered invalid bytes, those will not be included in the buffer.

The option values must be `,` separated and can appear in any order.

> **Note**
>
> `from_base64` follows RFC 4648 by default i.e. encounter with any character that is not found in the base64 alphabet leads to rejection of that character and the rest of the string.

Format:

```
from_base64: [[bytes <value>] [, offset <offset_value> [, mode: strict|rfc4648|rfc2045]]]
from_base64
```

There are defaults for each of the options: - `bytes` defaults to the length of the input buffer - `offset` defaults to `0` and must be less than `65536` - `mode` defaults to `rfc4648`

The second example shows the rule keyword only which will cause the default values for each option to be used.

Note that both `bytes` and `offset` may be variables from *byte_extract* and/or *byte_math* in later versions of Suricata. They are not supported yet.

Mode `rfc4648` applies RFC 4648 decoding logic which is suitable for encoding binary data that can be safely sent by email, used in a URL, or included with HTTP POST requests.

Mode `rfc2045` applies RFC 2045 decoding logic which supports strings, including those with embedded spaces, line breaks, and any non base64 alphabet.

Mode `strict` will fail if an invalid character is found in the encoded bytes.

The following examples will alert when the buffer contents match (see the last `content` value for the expected strings).

This example uses the defaults and transforms *"VGhpcyBpcyBTdXJpY2F0YQ=="* to *"This is Suricata"*:

```
content: "VGhpcyBpcyBTdXJpY2F0YQ=="; from_base64; content:"This is Suricata";
```

This example transforms *"dGhpc2lzYXRlc3QK"* to *"thisisatest"*:

```
content:"/?arg=dGhpc2lzYXRlc3QK"; from_base64: offset 6, mode rfc4648; \
content:"thisisatest";
```

This example transforms *"Zm 9v Ym Fy"* to *"foobar"*:

```
content:"/?arg=Zm 9v Ym Fy"; from_base64: offset 6, mode rfc2045; \
content:"foobar";
```

8.11.17 luaxform

This transform allows a Lua script to apply a transformation to a buffer.

Lua scripts that are used for transformations *must* contain a function named `transform`. The function signature is:

```
-- transform function arguments:
--   input - the buffer to be transformed. Use #input to determine byte count
--   args - The arguments from the rule. Access each as args[0], args[1], ...
function transform(input, args)
```

Lua scripts can have other functions; Suricata only invokes the `transform` function.

Lua transforms can be passed optional arguments -- see the examples below -- but they are not required to do so. Specify comma-separated arguments in the rule, e.g:

```
luaxform:transform.lua, bytes 0, offset 2
```

A Lua transform function is not invoked if the buffer is empty or the Lua framework is not accessible (rare).

Lua transform functions must return two values (see below) or the buffer is not modified. An empty return ("" with a length of 0) is accepted and also leaves the buffer unchanged; `luaxform` cannot shorten a buffer to zero.

Note that the arguments and values are passed without validation nor interpretation. There is a maximum of 10 arguments.

The Lua transform function is invoked with these parameters:

- *input* The buffer provided to the transform
- *arguments* The list of arguments.

Lua transform functions must return two values [Lua datatypes shown]:

- *buffer* [Lua string] The return buffer containing the original input buffer or buffer modified by the transform.
- *bytes* [Lua integer] Number of bytes in return buffer.

This example supplies the HTTP data to a Lua transform and the transform results are checked with *content*.

Example:

```
alert http any any -> any any (msg:"Lua Xform example"; flow:established;  \
        file.data; luaxform:./lua/lua-transform.lua; content: "abc"; sid: 2;)
```

This example supplies the HTTP data to a Lua transform with with arguments that specify the offset and byte count for the transform. The resulting buffer is then checked with a *content* match.

Example:

```
alert http any any -> any any (msg:"Lua Xform example"; flow:established; \
        file.data; luaxform:./lua/lua-transform.lua, bytes 12, offset 13; content: "abc";
↪ sid: 1;)
```

The following Lua script shows a transform that handles arguments: *bytes* and *offset* and uses those values (or defaults, if there are no arguments) for applying the uppercase transform to the buffer.

```
local function get_value(item, key)
    if string.find(item, key) then
        local _, value = string.match(item, "(%a+)%s*(%d*)")
        if value ~= "" then
            return tonumber(value)
        end
    end

    return nil
end

-- Arguments supported
local bytes_key = "bytes"
local offset_key = "offset"

function transform(input, args)
    local bytes = #input
    local offset = 0

    -- Look for optional bytes and offset arguments
    for i, item in ipairs(args) do
        local value = get_value(item, bytes_key)
        if value ~= nil then
            bytes = value
        else
            local value = get_value(item, offset_key)
            if value ~= nil then
                offset = value
            end
        end
    end
    local str_len = #input
    if offset < 0 or str_len < offset then
        print("offset is out of bounds: " .. offset)
        return nil
    end
    str_len = str_len - offset
    if bytes < 0 or str_len < bytes then
        print("invalid bytes " ..  bytes .. " or bytes > length " .. bytes .. " length "␣
↪.. str_len)
        return nil
    end
```

(continues on next page)

(continued from previous page)

```
    local sub = string.sub(input, offset + 1, offset + bytes)
    return string.upper(sub), bytes
end
```

8.11.18 gunzip

Takes the buffer, applies gunzip decompression.

This transform takes an optional argument which is a comma-separated list of key-values. Key and value are separated by a space. The only key being interperted is `max-size`, which is the max output size. Default for max-size is 1024. If the decompressed data were to be larger than max-size, the transform will decompress data up to max-size. Value 0 is forbidden for max-size (there is no unlimited value).

This example alerts if `http.uri` contains base64-encoded gzipped value:

alert http any any -> any any (msg:"from_base64 + gunzip";
http.uri; content:"/gzb64?value="; fast_pattern; from_base64: offset 13 ; gunzip: max-size 1024; content:"This is compressed then base64-encoded"; startswith; endswith; sid:2; rev:1;)

8.11.19 zlib_deflate

Takes the buffer, applies zlib decompression.

This transform takes an optional argument which is a comma-separated list of key-values. Key and value are separated by a space. The only key being interperted is `max-size`, which is the max output size. Default for max-size is 1024. If the decompressed data were to be larger than max-size, the transform will decompress data up to max-size. Value 0 is forbidden for max-size (there is no unlimited value).

This example alerts if `http.uri` contains base64-encoded zlib-compressed value:

alert http any any -> any any (msg:"from_base64 + gunzip";
http.uri; content:"/zb64?value="; fast_pattern; from_base64: offset 12 ; zlib_deflate; content:"This is compressed then base64-encoded"; startswith; endswith; sid:2; rev:1;)

8.11.20 subslice

This transform creates a slice of the input buffer.

The subslice transform requires parameters:

- *offset* Specifies the starting offset at which to create the subslice. When negative, expresses how far from the end of the input buffer to begin. If the absolute value of a negative offset exceeds the buffer length and `truncate` is not present, the transform will produce an empty buffer. When `truncate` is present, the starting position is reset to the beginning of the buffer. [REQUIRED]
- *nbytes* Specifies the size of the subslice. When negative, specifies that the subslice ends *nbytes* bytes from the end of the input buffer. If the absolute value of a negative `nbytes` exceeds the buffer length and `truncate` is not present, the transform will produce an empty buffer. When `truncate` is present, the endpoint is limited to the beginning of the buffer. The default value is the size of the input buffer minus the value of `offset`. [OPTIONAL]
- *truncate* Specifies behavior when `offset + nbytes` is larger than the input buffer size, or when the absolute value of a negative offset or negative `nbytes` exceeds the buffer length. When present, the result is trimmed as though `offset + nbytes == buffer_length` and excessive negative values are reset to the buffer boundaries. When not present (DEFAULT), a subslice larger than `offset + nbytes` produces an empty buffer, on which `bsize:0` will match. [OPTIONAL]

Usage

Specify the subslice desired -- *nbytes* and *truncate* are optional:

Format:

```
subslice: offset <, nbytes>, <, truncate>;
```

When *nbytes* is not present, the size of the subslice will be the size of the input buffer minus the *offset* value.

When `truncate` is not present and the value of `offset + nbytes` exceeds the buffer length, an empty buffer will be produced such that `bsize: 0` will match.

Examples

The following examples use an input buffer of `This is Suricata`.

The subslice will be a copy of the input buffer but omit the input buffer's first byte. The subslice is `his is Suricata`:

```
subslice: 1;
```

This example creates the subslice `This is Suric`:

```
subslice: 0, 13;
```

This example starts at offset `10` and ends at 5 bytes from the end of the buffer which creates a subslice from offset `10` to offset `12`. The length of the input buffer is `17` bytes; `5` bytes from the end is `12`. The subslice is "r"

```
subslice: 10, -5;
```

This example will create a subslice from the last 3 bytes of the input buffer and create `ata`:

```
subslice: -3;
```

Summary of Truncate Behavior: Edge Cases

The following table summarizes how `truncate` handles edge cases with the input buffer `curl/7.64.1` (11 bytes):

Transform	No truncate	With truncate
`subslice: 5;`	**7.64.1** (6 bytes)	**7.64.1** (6 bytes)
`subslice: -20;`	Empty buffer	Full buffer (start at 0)
`subslice: -20, 5;`	Empty buffer	**curl/** (5 bytes)
`subslice: 0, -30;`	Empty buffer	Empty buffer (end at 0)
`subslice: 0, -8;`	**cur** (3 bytes)	**cur** (3 bytes)
`subslice: -20, -30;`	Empty buffer	Empty buffer
`subslice: 0, 30;`	Empty buffer	Full buffer (11 bytes)

Truncation Behavior

When the buffer has fewer bytes than `offset + nbytes`, the transform will either trim the resulting buffer as though `offset + nbytes == buffer_length` or produce an empty buffer on which `bsize:0` would match. The behavior is determined by the inclusion of `truncate` with the keyword.

The following examples use an input buffer with the value `curl/7.64.1`.

Without `truncate` (the default), when `offset + nbytes` exceeds the buffer length, the transform produces an empty buffer on which `bsize:0` would match:

```
subslice: 0, 30;
```

When `truncate` is present, `nbytes + offset` is reduced to equal the input buffer length. The transform produces `curl/7.64.1`:

```
subslice: 0, 30, truncate;
```

`truncate` does not require `nbytes` to be present. The transform produces `curl/7.64.1`:

```
subslice: 0, truncate;
```

Negative Offset Handling

When a negative offset's absolute value exceeds the buffer length, the behavior depends on whether `truncate` is present:

Without `truncate`, the transform produces an empty buffer. For example, with input buffer `This is Suricata` (16 bytes), using `subslice: -17;` produces an empty string and `bsize:0` would match:

```
subslice: -17;
```

With `truncate` present, an excessive negative offset is reset to the buffer length, effectively starting at offset 0. Using the same input buffer `This is Suricata` (16 bytes), `subslice: -17, truncate;` is treated as `subslice: -16, truncate;` and produces the full buffer `This is Suricata`:

```
subslice: -17, truncate;
```

This also works with `nbytes`. For example, `subslice: -20, 5, truncate;` with input buffer `This is Suricata` starts at offset 0 and takes 5 bytes, producing `This` (with a trailing space):

```
subslice: -20, 5, truncate;
```

Similarly, when `truncate` is present, negative `nbytes` values that would place the endpoint before the beginning of the buffer are reset to the beginning of the buffer. For example, `subslice: 0, -30, truncate;` with input buffer `This is Suricata` (16 bytes) resets the endpoint to the beginning of the buffer, producing an empty buffer:

```
subslice: 0, -30, truncate;
```

However, a moderate negative `nbytes` works normally. For example, `subslice: 0, -8, truncate;` ends 8 bytes from the end (position 8), producing `This is` (with a trailing space):

```
subslice: 0, -8, truncate;
```

Full Rule Examples

The following rule inspects the first 4 bytes of the HTTP User-Agent header to match traffic with a User-Agent beginning with `curl`:

alert http any any -> any any (msg:"curl User-Agent"; http.user_agent; subslice: 0, 4, truncate; content:"curl"; bsize:4; sid:1;)

The following rule inspects the last 4 bytes of the HTTP URI to match requests whose URI ends with `.php`:

alert http any any -> any any (msg:"PHP URI"; http.uri; subslice: -4; content:".php"; bsize:4; sid:2;)

8.12 Prefiltering Keywords

8.12.1 fast_pattern

Suricata Fast Pattern Determination Explained

If the 'fast_pattern' keyword is explicitly set in a rule, Suricata will use that as the fast pattern match. The 'fast_pattern' keyword can only be set once per rule. If 'fast_pattern' is not set, Suricata automatically determines the content to use as the fast pattern match.

The following explains the logic Suricata uses to automatically determine the fast pattern match to use.

Be aware that if there are positive (i.e. non-negated) content matches, then negated content matches are ignored for fast pattern determination. Otherwise, negated content matches are considered.

The fast_pattern selection criteria are as follows:

1. Suricata first identifies all content matches that have the highest "priority" that are used in the signature. The priority is based off of the buffer being matched on and generally application layer buffers have a higher priority (lower number is higher priority). The buffer *http_method* is an exception and has lower priority than the general *content* buffer.
2. Within the content matches identified in step 1 (the highest priority content matches), the longest (in terms of character/byte length) content match is used as the fast pattern match.
3. If multiple content matches have the same highest priority and qualify for the longest length, the one with the highest character/byte diversity score ("Pattern Strength") is used as the fast pattern match. See *Appendix A* for details on the algorithm used to determine Pattern Strength.
4. If multiple content matches have the same highest priority, qualify for the longest length, and the same highest Pattern Strength, the buffer ("list_id") that was *registered last* is used as the fast pattern match.
5. If multiple content matches have the same highest priority, qualify for the longest length, the same highest Pattern Strength, and have the same list_id (i.e. are looking in the same buffer), then the one that comes first (from left-to-right) in the rule is used as the fast pattern match.

It is worth noting that for content matches that have the same priority, length, and Pattern Strength, 'http_stat_msg', 'http_stat_code', and 'http_method' take precedence over regular 'content' matches.

Appendices

Appendix A - Pattern Strength Algorithm

From detect-engine-mpm.c. Basically the Pattern Strength "score" starts at zero and looks at each character/byte in the passed in byte array from left to right. If the character/byte has not been seen before in the array, it adds 3 to the score if it is an alpha character; else it adds 4 to the score if it is a printable character, 0x00, 0x01, or 0xFF; else it adds 6 to the score. If the character/byte has been seen before it adds 1 to the score. The final score is returned.

```
/** \brief Predict a strength value for patterns
 *
 *  Patterns with high character diversity score higher.
 *  Alpha chars score not so high
 *  Other printable + a few common codes a little higher
 *  Everything else highest.
 *  Longer patterns score better than short patters.
 *
 *  \param pat pattern
 *  \param patlen length of the pattern
```

(continues on next page)

(continued from previous page)

```
 *
 *  \retval s pattern score
 */
uint32_t PatternStrength(uint8_t *pat, uint16_t patlen) {
    uint8_t a[256];
    memset(&a, 0 ,sizeof(a));
    uint32_t s = 0;
    uint16_t u = 0;
    for (u = 0; u < patlen; u++) {
        if (a[pat[u]] == 0) {
            if (isalpha(pat[u]))
                s += 3;
            else if (isprint(pat[u]) || pat[u] == 0x00 || pat[u] == 0x01 || pat[u] ==␣
↪0xFF)
                s += 4;
            else
                s += 6;
            a[pat[u]] = 1;
        } else {
            s++;
        }
    }
    return s;
}
```

Only one content of a signature will be used in the Multi Pattern Matcher (MPM). If there are multiple contents, then Suricata uses the 'strongest' content. This means a combination of length, how varied a content is, and what buffer it is looking in. Generally, the longer and more varied the better. For full details on how Suricata determines the fast pattern match, see *Suricata Fast Pattern Determination Explained*.

Sometimes a signature writer concludes he wants Suricata to use another content than it does by default.

For instance:

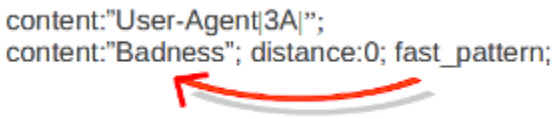
```
User-agent: Mozilla/5.0 Badness;

content:"User-Agent|3A|";
content:"Badness"; distance:0;
```

In this example you see the first content is longer and more varied than the second one, so you know Suricata will use this content for the MPM. Because 'User-Agent:' will be a match very often, and 'Badness' appears less often in network traffic, you can make Suricata use the second content by using 'fast_pattern'.

```
content:"User-Agent|3A|";
content:"Badness"; distance:0; fast_pattern;
```

The keyword fast_pattern modifies the content previous to it.



Fast-pattern can also be combined with all previous mentioned keywords, and all mentioned HTTP-modifiers.

fast_pattern:only

Sometimes a signature contains only one content. In that case it is not necessary Suricata will check it any further after a match has been found in MPM. If there is only one content, the whole signature matches. Suricata notices this automatically. In some signatures this is still indicated with 'fast_pattern:only;'. Although Suricata does not need fast_pattern:only, it does support it.

fast_pattern:'chop'

If you do not want the MPM to use the whole content, you can use fast_pattern 'chop'.

For example:

```
content: "aaaaaaaaabc"; fast_pattern:8,4;
```

This way, MPM uses only the last four characters.

8.12.2 prefilter

The prefilter engines for other non-MPM keywords can be enabled in specific rules by using the 'prefilter' keyword.

In the following rule the TTL test will be used in prefiltering instead of the single byte pattern:

alert ip any any -> any any (ttl:123; prefilter; content:"a"; sid:1;)

For more information on how to configure the prefilter engines, see *Prefilter Engines*

8.13 Flow Keywords

8.13.1 flowbits

Flowbits consists of two parts. The first part describes the action it is going to perform, the second part is the name of the flowbit.

There are multiple packets that belong to one flow. Suricata keeps those flows in memory. For more information see *Flow Settings*.

Flowbits can make sure an alert will be generated when for example two different packets match. An alert will only be generated when both packets match. So, when the second packet matches, Suricata has to know if the first packet was a match too. Flowbits mark the flow if a packet matches so Suricata 'knows' it should generate an alert when the second packet matches as well.

Flowbits have different actions. These are:

flowbits: set, name
: Will set the condition/'name', if present, in the flow.

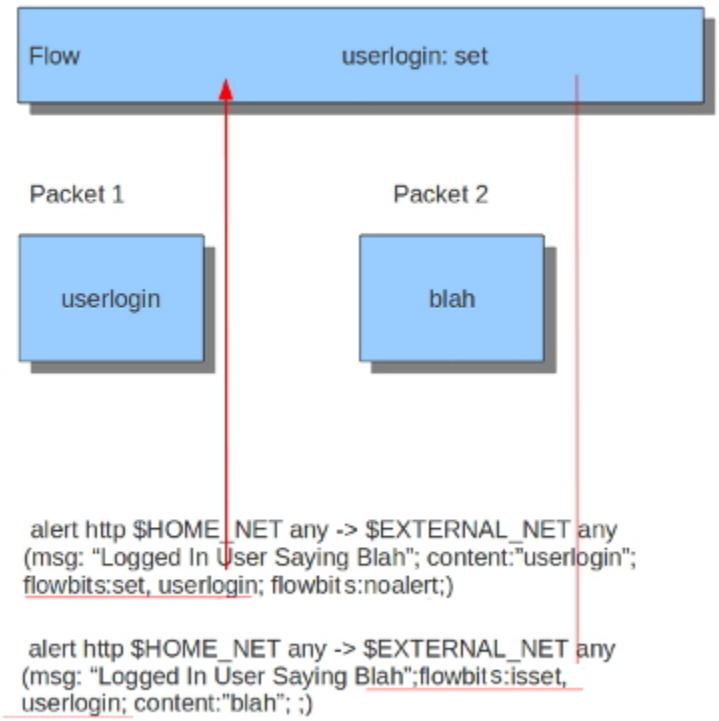
flowbits: isset, name
: Can be used in the rule to make sure it generates an alert when the rule matches and the condition is set in the flow.

flowbits: unset, name
: Can be used to unset the condition in the flow.

flowbits: isnotset, name
: Can be used in the rule to make sure it generates an alert when it matches and the condition is not set in the flow.

flowbits: noalert
: No alert will be generated by this rule.

Example:



When you take a look at the first rule you will notice it would generate an alert if it would match, if it were not for the 'flowbits: noalert' at the end of that rule.

The purpose of this rule is to check for a match on 'userlogin' and mark that in the flow. So, there is no need to generate an alert. The second rule has no effect without the first rule. If the first rule matches, the flowbit sets that specific condition to be present in the flow. Now the second rule can be checked whether or not the previous packet fulfills the first condition. If the second rule matches now, an alert will be generated.

Note

flowbit names are case-sensitive.

Note

It is possible to use flowbits several times in a rule and combine the different functions.

Note

It is possible to perform an *OR* operation with flowbits using the | (pipe).

alert http any any -> any any (msg:"User1 or User2 logged in"; content:"login"; flowbits:isset,user1|user2; sid:1;)

8.13.2 flow

The flow keyword can be used to match on direction of the flow, so to/from client or to/from server. It can also match if the flow is established or not. The flow keyword can also be used to say the signature has to match on stream only (only_stream) or on packet only (no_stream).

So with the flow keyword you can match on:

to_client
Match on packets from server to client.

to_server
Match on packets from client to server.

from_client
Match on packets from client to server (same as to_server).

from_server
Match on packets from server to client (same as to_client).

established
Match on established connections.

not_established
Match on packets that are not part of an established connection.

stateless
Match on packets that are part of a flow, regardless of connection state. (This means that packets that are not seen as part of a flow won't match).

only_stream
Match on packets that have been reassembled by the stream engine.

no_stream
Match on packets that have not been reassembled by the stream engine. Will not match packets that have been reassembled.

only_frag
Match packets that have been reassembled from fragments.

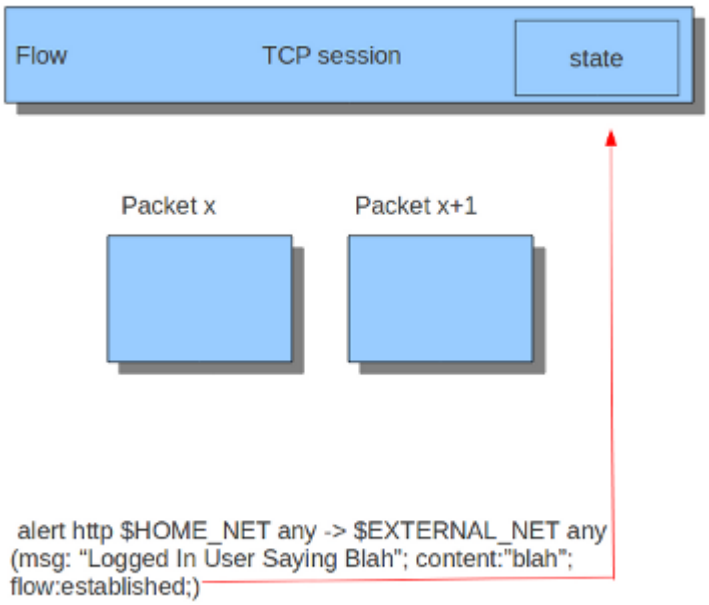
no_frag
Match packets that have not been reassembled from fragments.

Multiple flow options can be combined, for example:

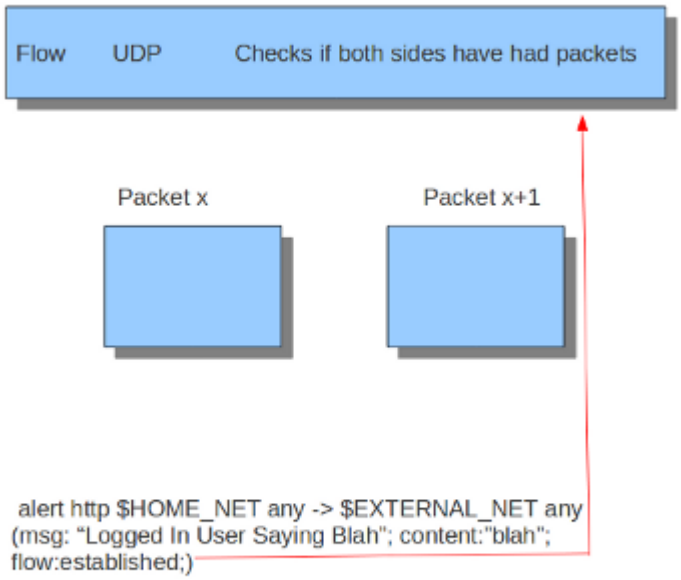
```
flow:to_client, established
flow:to_server, established, only_stream
flow:to_server, not_established, no_frag
```

The determination of *established* depends on the protocol:

- For TCP a connection will be established after a three way handshake.



- For other protocols (for example UDP), the connection will be considered established after seeing traffic from both sides of the connection.



8.13.3 tcp.session

In firewall mode, allowing a TCP connection typically requires covering both the handshake (`flow:not_established`) and the data transfer phase (`flow:established`). Without `tcp.session`, this means two separate rules. The `tcp.session` keyword lets a rule writer collapse both into a single rule, reducing boilerplate:

```
# Before: two rules needed
accept:hook tcp:all any any <> any any (flow:not_established; sid:1;)
accept:hook tcp:all any any <> any any (flow:established; sid:2;)

# After: one rule
accept:hook tcp:all any any <> any any (tcp.session:setup,established; sid:1;)
```

The `tcp.session` keyword matches on one or more phases of the TCP session lifecycle. It works in all engine modes (IDS, IPS, and firewall).

The keyword argument is a comma-separated subset of `setup`, `established` and `closing`. Tokens are case-sensitive (lowercase only), may appear in any order, and must not be duplicated. Multiple phases are evaluated as logical OR: a packet matches if it belongs to *any* of the named phases.

Format:

```
tcp.session:<phase>[,<phase>...];
```

The three phases map to the following TCP states:

setup
: `TCP_NONE`, `TCP_SYN_SENT`, `TCP_SYN_RECV`. Equivalent to `flow:not_established` on non-midstream flows.

established
: `TCP_ESTABLISHED`. Equivalent to `flow:established` on non-midstream flows.

closing
: `TCP_FIN_WAIT1`, `TCP_FIN_WAIT2`, `TCP_TIME_WAIT`, `TCP_LAST_ACK`, `TCP_CLOSE_WAIT`, `TCP_CLOSING`. The terminal `TCP_CLOSED` state is not part of any phase.

The state sets are disjoint, but a single packet may match more than one phase because `FLOW_PKT_ESTABLISHED` is sticky — it stays set during closing states. A packet in `TCP_FIN_WAIT1` matches both `tcp.session:established` and `tcp.session:closing`.

Examples:

```
tcp.session:setup
tcp.session:setup,established
tcp.session:established,closing
tcp.session:setup,established,closing
```

alert tcp any any -> any any (tcp.session:setup,established; sid:1;)

In firewall mode it is typically used in an `accept` rule:

```
accept:hook tcp:all any any <> any any (tcp.session:setup,established; sid:1;)
```

Note

When used in the `pre_stream` hook, the TCP session state has not yet been updated for the current packet. The keyword reflects the state *before* the stream engine processes the packet. For example, a SYN packet in the `pre_stream`

hook will see the state from the previous packet (`TCP_NONE` for the first packet of a flow).

> **Note**
>
> Using `tcp.session` in the `pre_flow` hook is not meaningful because no flow (and thus no TCP session) exists yet at that point. The keyword will never match in `pre_flow`.

Midstream pickup

When a flow is picked up midstream, `tcp.session` evaluates the flow's current TCP state directly. A midstream pickup in `TCP_ESTABLISHED` matches `tcp.session:established` and does **not** match `tcp.session:setup`.

Combining with the `flow` keyword

`tcp.session` may be combined with `flow:not_established` or `flow:established` in the same rule. Both keywords are evaluated independently (logical AND). On non-midstream flows, `tcp.session:setup` is equivalent to `flow:not_established` and `tcp.session:established` is equivalent to `flow:established`.

> **Note**
>
> `tcp.session` is TCP-specific; it never matches on non-TCP flows.

8.13.4 flowint

Flowint allows storage and mathematical operations using variables. It operates much like flowbits but with the addition of mathematical capabilities and the fact that an integer can be stored and manipulated, not just a flag set. We can use this for a number of very useful things, such as counting occurrences, adding or subtracting occurrences, or doing thresholding within a stream in relation to multiple factors. This will be expanded to a global context very soon, so users can perform these operations between streams.

The syntax is as follows:

```
flowint: name, modifier[, value];
```

Define a var (not required), or check that one is set or not set.

```
flowint: name, < +,-,=,>,<,>=,<=,==, != >, value;
flowint: name, (isset|notset|isnotset);
```

Compare or alter a var. Add, subtract, compare greater than or less than, greater than or equal to, and less than or equal to are available. The item to compare with can be an integer or another variable.

For example, if you want to count how many times a username is seen in a particular stream and alert if it is over 5.

```
alert tcp any any -> any any (msg:"Counting Usernames"; content:"jonkman"; \
      flowint: usernamecount, +, 1; noalert;)
```

This will count each occurrence and increment the var usernamecount and not generate an alert for each.

Now say we want to generate an alert if there are more than five hits in the stream.

```
alert tcp any any -> any any (msg:"More than Five Usernames!"; content:"jonkman"; \
      flowint: usernamecount, +, 1; flowint:usernamecount, >, 5;)
```

So we'll get an alert ONLY if usernamecount is over five.

So now let's say we want to get an alert as above but NOT if there have been more occurrences of that username logging out. Assuming this particular protocol indicates a log out with "jonkman logout", let's try:

```
alert tcp any any -> any any (msg:"Username Logged out"; content:"logout jonkman"; \
      flowint: usernamecount, -, 1; flowint:usernamecount, >, 5;)
```

So now we'll get an alert ONLY if there are more than five active logins for this particular username.

This is a rather simplistic example, but I believe it shows the power of what such a simple function can do for rule writing. I see a lot of applications in things like login tracking, IRC state machines, malware tracking, and brute force login detection.

Let's say we're tracking a protocol that normally allows five login fails per connection, but we have vulnerability where an attacker can continue to login after that five attempts and we need to know about it.

```
alert tcp any any -> any any (msg:"Start a login count"; content:"login failed"; \
      flowint:loginfail, notset; flowint:loginfail, =, 1; noalert;)
```

So we detect the initial fail if the variable is not yet set and set it to 1 if so. Our first hit.

```
alert tcp any any -> any any (msg:"Counting Logins"; content:"login failed"; \
      flowint:loginfail, isset; flowint:loginfail, +, 1; noalert;)
```

We are now incrementing the counter if it's set.

```
alert tcp any any -> any any (msg:"More than Five login fails in a Stream"; \
      content:"login failed"; flowint:loginfail, isset; flowint:loginfail, >, 5;)
```

Now we'll generate an alert if we cross five login fails in the same stream.

But let's also say we also need alert if there are two successful logins and a failed login after that.

```
alert tcp any any -> any any (msg:"Counting Good Logins";              \
      content:"login successful"; flowint:loginsuccess, +, 1; noalert;)
```

Here we're counting good logins, so now we'll count good logins relevant to fails:

```
alert tcp any any -> any any (msg:"Login fail after two successes";   \
      content:"login failed"; flowint:loginsuccess, isset;             \
      flowint:loginsuccess, =, 2;)
```

Here are some other general examples:

```
alert tcp any any -> any any (msg:"Setting a flowint counter"; content:"GET"; \
      flowint:myvar, notset; flowint:maxvar,notset;                            \
      flowint:myvar,=,1; flowint: maxvar,=,6;)
```

```
alert tcp any any -> any any (msg:"Adding to flowint counter";                 \
      content:"Unauthorized"; flowint:myvar,isset; flowint: myvar,+,2;)
```

```
alert tcp any any -> any any (msg:"when flowint counter is 3 create new counter"; \
      content:"Unauthorized"; flowint:myvar, isset; flowint:myvar,==,3; \
      flowint:cntpackets,notset; flowint:cntpackets, =, 0;)
```

```
alert tcp any any -> any any (msg:"count the rest without generating alerts"; \
      flowint:cntpackets,isset; flowint:cntpackets, +, 1; noalert;)
```

```
alert tcp any any -> any any (msg:"fire this when it reach 6";                \
      flowint: cntpackets, isset;                                             \
      flowint: maxvar,isset; flowint: cntpackets, ==, maxvar;)
```

8.13.5 stream_size

The stream size option matches on traffic according to the registered amount of bytes by the sequence numbers. There are several modifiers to this keyword:

```
>      greater than
<      less than
=      equal
!=     not equal
>=    greater than or equal
<=    less than or equal
```

Format

```
stream_size:<server|client|both|either>, <modifier>, <number>;
```

Example of the stream-size keyword in a rule:

```
alert tcp any any -> any any (stream_size:both, >, 5000; sid:1;)
```

8.13.6 flow.age

Flow age in seconds (integer) This keyword does not wait for the end of the flow, but will be checked at each packet.

flow.age uses an *unsigned 32-bit integer*.

Syntax:

```
flow.age: [op]<number>
```

The time can be matched exactly, or compared using the _op_ setting:

```
flow.age:3    # exactly 3
flow.age:<3   # smaller than 3 seconds
flow.age:>=2  # greater or equal than 2 seconds
```

alert tcp any any -> any any (msg:"Flow longer than one hour"; flow.age:>3600; flowbits: isnotset, onehourflow; flowbits: set, onehourflow; sid:1; rev:1;)

In this example, we combine *flow.age* and *flowbits* to get an alert on the first packet after the flow's age is older than one hour.

8.13.7 flow.pkts

Flow number of packets (integer) This keyword does not wait for the end of the flow, but will be checked at each packet.

flow.pkts uses an *unsigned 32-bit integer* and supports following directions:

- toclient
- toserver
- either
- both

Syntax:

```
flow.pkts:<direction>,[op]<number>
```

The number of packets can be matched exactly, or compared using the _op_ setting:

```
flow.pkts:toclient,3    # exactly 3
flow.pkts:toserver,<3   # smaller than 3
flow.pkts:either,>=2  # greater than or equal to 2
```

alert ip any any -> any any (msg:"Flow has 20 packets in toclient dir"; flow.pkts:toclient,20; sid:1;)

Note

Suricata also supports `flow.pkts_toclient` and `flow.pkts_toserver` keywords for `flow.pkts:toclient` and `flow.pkts:toserver` respectively but that is not the preferred syntax.

8.13.8 flow.bytes

Flow number of bytes (integer) This keyword does not wait for the end of the flow, but will be checked at each packet.

flow.bytes uses an *unsigned 64-bit integer* and supports following directions:

- toclient
- toserver
- either
- both

Syntax:

```
flow.bytes:<direction>,[op]<number>
```

The number of bytes can be matched exactly, or compared using the _op_ setting:

```
flow.bytes:toclient,3    # exactly 3
flow.bytes:toserver,<3   # smaller than 3
flow.bytes:either,>=2  # greater than or equal to 2
```

alert ip any any -> any any (msg:"Flow has less than 2000 bytes in toserver dir"; flow.bytes:toserver,<2000; sid:1;)

> **Note**
>
> Suricata also supports `flow.bytes_toclient` and `flow.bytes_toserver` keywords for `flow.bytes:toclient` and `flow.bytes:toserver` respectively but that is not the preferred syntax.

8.13.9 flow.elephant

Match an elephant flow. This keyword does not wait for the end of the flow, but will be checked at each packet as the state of a flow can change with any packet.

`flow.elephant` is tracked per direction internally which is why it is possible to give a direction that match should be made on:

- toclient
- toserver
- either
- both

Syntax:

```
flow.elephant:<direction>
```

alert tcp any any -> any any (msg:"Flow is elephant in toserver dir"; flow.elephant:toserver; sid:1;)

8.14 Bypass Keyword

Suricata has a `bypass` keyword that can be used in signatures to exclude traffic from further evaluation.

The `bypass` keyword is useful in cases where there is a large flow expected (e.g. Netflix, Spotify, YouTube).

The `bypass` keyword is considered a post-match keyword.

> **Note**
>
> `bypass` cannot be used in firewall mode, not even with Threat Detection rules, as this could lead to bypassing the firewall altogether.

8.14.1 bypass

Bypass a flow on matching http traffic.

alert http any any -> any any (http.host; content:"suricata.io"; bypass; sid:10001; rev:1;)

8.15 HTTP Keywords

Using the HTTP specific sticky buffers (see *Modifier Keywords*) provides a way to efficiently inspect the specific fields of HTTP protocol communications. After specifying a sticky buffer in a rule it should be followed by one or more *Payload Keywords* or using *pcre (Perl Compatible Regular Expressions)*.

8.15.1 HTTP Primer

HTTP is considered a client-server or request-response protocol. A client requests resources from a server and a server responds to the request.

In versions of HTTP prior to version 2 a client request could look like:

Example HTTP Request:

```
GET /index.html HTTP/1.1
User-Agent: Mozilla/5.0
Host: suricata.io
```

Example signature that would alert on the above request.

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP Request Example"; flow:established,to_server; http.method; content:"GET"; http.uri; content:"/index.html"; bsize:11; http.protocol; content:"HTTP/1.1"; bsize:8; http.user_agent; content:"Mozilla/5.0"; bsize:11; http.host; content:"suricata.io"; bsize:11; classtype:bad-unknown; sid:25; rev:1;)

In versions of HTTP prior to version 2 a server response could look like:

Example HTTP Response:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 258
Date: Thu, 14 Dec 2023 20:22:41 GMT
Server: nginx/0.8.54
Connection: Close
```

Example signature that would alert on the above response.

alert http $EXTERNAL_NET any -> $HOME_NET any (msg:"HTTP Stat Code Example"; flow:established,to_client; http.stat_code; content:"200"; bsize:8; http.content_type; content:"text/html"; bsize:9; classtype:bad-unknown; sid:30; rev:1;)

Request Keywords:

- *file.name*
- *http.accept*
- *http.accept_enc*
- *http.accept_lang*
- *http.host*
- *http.host.raw*
- *http.method*
- *http.referer*
- *http.request_body*
- *http.request_header*
- *http.request_line*
- *http.uri*
- *http.uri.raw*

- *http.user_agent*
- *urilen*

Response Keywords:

- *http.location*
- *http.response_body*
- *http.response_header*
- *http.response_line*
- *http.server*
- *http.stat_code*
- *http.stat_msg*

Request or Response Keywords:

- *file.data*
- *http.connection*
- *http.content_len*
- *http.content_type*
- *http.cookie*
- *http.header*
- *http.header.raw*
- *http.header_names*
- *http.protocol*
- *http.start*

8.15.2 Normalization

There are times when Suricata performs formatting/normalization changes to traffic that is seen.

Duplicate Header Names

If there are multiple values for the same header name, they are concatenated with a comma and space (", ") between each value. More information can be found in RFC 2616 https://www.rfc-editor.org/rfc/rfc2616.html#section-4.2

In the example below, notice that the User-Agent header, regardless of the letter casing is evaluated as the same header. The normalized header evaluation leads to the concatenated header values as described in the RFC above.

Example Duplicate HTTP Header:

```
GET / HTTP/1.1
Host: suricata.io
User-Agent: Mozilla/5.0
User-agent: Chrome/121.0.0
```

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"Example Duplicate Header"; flow:established,to_server; http.user_agent; content:"Mozilla/5.0, Chrome/121.0.0"; classtype:bad-unknown; sid:103; rev:1;)

8.15.3 file.name

The `file.name` keyword can be used with HTTP requests.

It is possible to use any of the *Payload Keywords* with the `file.name` keyword.

Example HTTP Request:

```
GET /picture.jpg HTTP/1.1
User-Agent: Mozilla/5.0
Host: suricata.io
```

alert http $EXTERNAL_NET any -> $HOME_NET any (msg:"HTTP file.name Example"; flow:established,to_client; file.name; content:"picture.jpg"; classtype:bad-unknown; sid:129; rev:1;)

> **Note**
>
> Additional information can be found at *File Keywords*

8.15.4 http.accept

The `http.accept` keyword is used to match on the Accept field that can be present in HTTP request headers.

It is possible to use any of the *Payload Keywords* with the `http.accept` keyword.

Example HTTP Request:

```
GET /index.html HTTP/1.1
User-Agent: Mozilla/5.0
Accept: */*
Host: suricata.io
```

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP Accept Example"; flow:established,to_server; http.accept; content:"*/*"; bsize:3; classtype:bad-unknown; sid:91; rev:1;)

> **Note**
>
> `http.accept` does not include the leading space or trailing \r\n

> **Note**
>
> `http.accept` can have additional formatting/normalization applied to buffer contents, see *Normalization* for additional details.

8.15.5 http.accept_enc

The `http.accept_enc` keyword is used to match on the Accept-Encoding field that can be present in HTTP request headers.

It is possible to use any of the *Payload Keywords* with the `http.accept_enc` keyword.

Example HTTP Request:

```
GET /index.html HTTP/1.1
User-Agent: Mozilla/5.0
Accept-Encoding: gzip, deflate
Host: suricata.io
```

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP Accept-Encoding Example"; flow:established,to_server; http.accept_enc; content:"gzip, deflate"; bsize:13; classtype:bad-unknown; sid:92; rev:1;)

Note

`http.accept_enc` does not include the leading space or trailing \r\n

Note

`http.accept_enc` can have additional formatting/normalization applied to buffer contents, see *Normalization* for additional details.

8.15.6 http.accept_lang

The `http.accept_lang` keyword is used to match on the Accept-Language field that can be present in HTTP request headers.

It is possible to use any of the *Payload Keywords* with the `http.accept_lang` keyword.

Example HTTP Request:

```
GET /index.html HTTP/1.1
User-Agent: Mozilla/5.0
Accept-Language: en-US
Host: suricata.io
```

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP Accept-Encoding Example"; flow:established,to_server; http.accept_lang; content:"en-US"; bsize:5; classtype:bad-unknown; sid:93; rev:1;)

Note

`http.accept_lang` does not include the leading space or trailing \r\n

Note

`http.accept_lang` can have additional formatting/normalization applied to buffer contents, see *Normalization* for additional details.

8.15.7 http.host

Matching on the HTTP host name has two options in Suricata, the `http.host` and the `http.host.raw` sticky buffers.

It is possible to use any of the *Payload Keywords* with both `http.host` keywords.

> **Note**
>
> The `http.host` keyword normalizes the host header contents. If a host name has uppercase characters, those would be changed to lowercase.

Normalization Example:

```
GET /index.html HTTP/1.1
User-Agent: Mozilla/5.0
Host: SuRiCaTa.Io
```

In the above example the host buffer would contain *suricata.io*.

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP Host Example"; flow:established,to_server; http.host; content:"suricata.io"; bsize:11; classtype:bad-unknown; sid:123; rev:1;)

> **Note**
>
> The `nocase` keyword is no longer allowed since the host names are normalized to contain only lowercase letters.

> **Note**
>
> `http.host` does not contain the port associated with the host (i.e. suricata.io:1234). To match on the host and port or negate a host and port use *http.host.raw*.

> **Note**
>
> `http.host` does not include the leading space or trailing \r\n

> **Note**
>
> The `http.host` and `http.host.raw` buffers are populated from either the URI (if the full URI is present in the request like in a proxy request) or the HTTP Host header. If both are present, the URI is used.

> **Note**
>
> `http.host` can have additional formatting/normalization applied to buffer contents, see *Normalization* for additional details.

8.15.8 http.host.raw

The `http.host.raw` buffer matches on HTTP host content but does not have any normalization performed on the buffer contents (see *http.host*)

Example HTTP Request:

```
GET /index.html HTTP/1.1
User-Agent: Mozilla/5.0
Host: SuRiCaTa.Io:8445
```

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP Host Raw Example"; flow:established,to_server; http.host.raw; content:"SuRiCaTa.Io|3a|8445"; bsize:16; classtype:bad-unknown; sid:124; rev:1;)

> **Note**
>
> `http.host.raw` does not include the leading space or trailing \r\n

> **Note**
>
> The `http.host` and `http.host.raw` buffers are populated from either the URI (if the full URI is present in the request like in a proxy request) or the HTTP Host header. If both are present, the URI is used.

> **Note**
>
> `http.host.raw` can have additional formatting/normalization applied to buffer contents, see *Normalization* for additional details.

8.15.9 http.method

The `http.method` keyword matches on the method/verb used in an HTTP request. HTTP request methods can be any of the following:

- GET
- POST
- HEAD
- OPTIONS
- PUT
- DELETE
- TRACE
- CONNECT
- PATCH

It is possible to use any of the *Payload Keywords* with the `http.method` keyword.

Example HTTP Request:

```
GET /index.html HTTP/1.1
User-Agent: Mozilla/5.0
Host: suricata.io
```

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP Request Example"; flow:established,to_server; http.method; content:"GET"; classtype:bad-unknown; sid:2; rev:1;)

8.15.10 http.referer

The `http.referer` keyword is used to match on the Referer field that can be present in HTTP request headers.

It is possible to use any of the *Payload Keywords* with the `http.referer` keyword.

Example HTTP Request:

```
GET / HTTP/1.1
Host: suricata.io
Referer: https://suricata.io
```

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP Referer Example"; flow:established,to_server; http.referer; content:"http|3a 2f 2f|suricata.io"; bsize:19; classtype:bad-unknown; sid:200; rev:1;)

Note

`http.referer` does not include the leading space or trailing \r\n

Note

`http.referer` can have additional formatting/normalization applied to buffer contents, see *Normalization* for additional details.

8.15.11 http.request_body

The `http.request_body` keyword is used to match on the HTTP request body that can be present in an HTTP request.

It is possible to use any of the *Payload Keywords* with the `http.request_body` keyword.

Example HTTP Request:

```
POST /suricata.php HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Host: suricata.io
Content-Length: 23
Connection: Keep-Alive

Suricata request body
```

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP Request Body Example"; flow:established,to_server; http.request_body; content:"Suricata request body"; classtype:bad-unknown; sid:115; rev:1;)

Note

How much of the request/client body is inspected is controlled in the *libhtp configuration section* via the `request-body-limit` setting.

Note

`http.request_body` replaces the previous keyword name, `http_client_body`. `http_client_body` can still be used but it is recommended that rules be converted to use `http.request_body`.

8.15.12 http.request_header

The `http.request_header` keyword is used to match on the name and value of a HTTP/1 or HTTP/2 request.

It is possible to use any of the *Payload Keywords* with the `http.request_header` keyword.

For HTTP/2, the header name and value get concatenated by ": " (colon and space). The colon and space are commonly noted with the hexadecimal format *|3a 20|* within signatures.

To detect if an HTTP/2 header name contains a ":" (colon), the keyword *http2.header_name* can be used.

Example HTTP/1 Request:

```
GET /index.html HTTP/1.1
User-Agent: Mozilla/5.0
Host: suricata.io
```

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP Request Example"; flow:established,to_server; http.request_header; content:"Host|3a 20|suricata.io"; classtype:bad-unknown; sid:126; rev:1;)

Note

`http.request_header` does not include the trailing \r\n

8.15.13 http.request_line

The `http.request_line` keyword is used to match on the entire contents of the HTTP request line.

Example HTTP Request:

```
GET /index.html HTTP/1.1
User-Agent: Mozilla/5.0
Host: suricata.io
```

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP Request Example"; flow:established,to_server; http.request_line; content:"GET /index.html HTTP/1.1"; bsize:24; classtype:bad-unknown; sid:60; rev:1;)

Note

`http.request_line` does not include the trailing \r\n

8.15.14 http.uri

Matching on the HTTP URI buffer has two options in Suricata, the `http.uri` and the `http.uri.raw` sticky buffers.

It is possible to use any of the *Payload Keywords* with both `http.uri` keywords.

The `http.uri` keyword normalizes the URI buffer. For example, if a URI has two leading `//`, Suricata will normalize the URI to a single leading `/`.

Normalization Example:

```
GET //index.html HTTP/1.1
User-Agent: Mozilla/5.0
Host: suricata.io
```

In this case //index.html would be normalized to /index.html.

Normalized HTTP Request Example:

```
GET /index.html HTTP/1.1
User-Agent: Mozilla/5.0
Host: suricata.io
```

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP URI Example"; flow:established,to_server; http.uri; content:"/index.html"; bsize:11; classtype:bad-unknown; sid:3; rev:1;)

8.15.15 http.uri.raw

The `http.uri.raw` buffer matches on HTTP URI content but does not have any normalization performed on the buffer contents. (see *http.uri*)

Abnormal HTTP Request Example:

```
GET //index.html HTTP/1.1
User-Agent: Mozilla/5.0
Host: suricata.io
```

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP URI Raw Example"; flow:established,to_server; http.uri.raw; content:"//index.html"; bsize:12; classtype:bad-unknown; sid:4; rev:1;)

> **Note**
>
> The `http.uri.raw` keyword/buffer does not allow for spaces.

Example Request:

```
GET /example spaces HTTP/1.1
User-Agent: Mozilla/5.0
Host: suricata.io
```

`http.uri.raw` would be populated with /example

http.protocol would be populated with spaces HTTP/1.1

Reference: https://redmine.openinfosecfoundation.org/issues/2881

8.15.16 http.user_agent

The `http.user_agent` keyword is used to match on the User-Agent field that can be present in HTTP request headers.

It is possible to use any of the *Payload Keywords* with the `http.user_agent` keyword.

Example HTTP Request:

```
GET /index.html HTTP/1.1
User-Agent: Mozilla/5.0
```

(continues on next page)

(continued from previous page)

```
Cookie: PHPSESSION=123
Host: suricata.io
```

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP User-Agent Example"; flow:established,to_server; http.user_agent; content:"Mozilla/5.0"; bsize:11; classtype:bad-unknown; sid:90; rev:1;)

> **Note**
>
> `http.user_agent` does not include the leading space or trailing \r\n

> **Note**
>
> Using the `http.user_agent` generally provides better performance than using *http.header*.

> **Note**
>
> `http.user_agent` can have additional formatting/normalization applied to buffer contents, see *Normalization* for additional details.

8.15.17 urilen

The `urilen` keyword is used to match on the length of the normalized request URI. It is possible to use the < and > operators, which indicate respectively *less than* and *larger than*.

urilen uses an *unsigned 64-bit integer*.

The `urilen` keyword does not require a content match on the *http.uri* buffer or the *http.uri.raw* buffer.

Example HTTP Request:

```
GET /index.html HTTP/1.1
User-Agent: Mozilla/5.0
Host: suricata.io
```

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP Request"; flow:established,to_server; urilen:11; http.method; content:"GET"; classtype:bad-unknown; sid:40; rev:1;)

The above signature would match on any HTTP GET request that has a URI length of 11, regardless of the content or structure of the URI.

The following signatures would all alert on the example request above as well and show the different `urilen` options.

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"urilen greater than 10"; flow:established,to_server; urilen:>10; classtype:bad-unknown; sid:41; rev:1;)

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"urilen less than 12"; flow:established,to_server; urilen:<12; classtype:bad-unknown; sid:42; rev:1;)

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"urilen greater/less than example"; flow:established,to_server; urilen:10<>12; classtype:bad-unknown; sid:43; rev:1;)

8.15.18 http.location

The `http.location` keyword is used to match on the HTTP response location header contents.

It is possible to use any of the *Payload Keywords* with the `http.location` keyword.

Example HTTP Response:

```
HTTP/1.1 200 OK
Content-Type: text/html
Server: nginx/0.8.54
Location: suricata.io
```

alert http $EXTERNAL_NET any -> $HOME_NET any (msg:"HTTP Location Example"; flow:established,to_client; http.location; content:"suricata.io"; bsize:11; classtype:bad-unknown; sid:122; rev:1;)

Note

`http.location` does not include the leading space or trailing \r\n

Note

`http.location` can have additional formatting/normalization applied to buffer contents, see *Normalization* for additional details.

8.15.19 http.response_body

The `http.response_body` keyword is used to match on the HTTP response body.

It is possible to use any of the *Payload Keywords* with the `http.response_body` keyword.

Example HTTP Response:

```
HTTP/1.1 200 OK
Content-Type: text/html
Server: nginx/0.8.54

Server response body
```

alert http $EXTERNAL_NET any -> $HOME_NET any (msg:"HTTP Response Body Example"; flow:established,to_client; http.response_body; content:"Server response body"; classtype:bad-unknown; sid:120; rev:1;)

Note

`http.response_body` will match on gzip decoded data just like *file.data* does.

Note

How much of the response/server body is inspected is controlled in your *libhtp configuration section* via the `response-body-limit` setting.

> **Note**
>
> `http.response_body` replaces the previous keyword name, `http_server_body`. `http_server_body` can still be used but it is recommended that rules be converted to use `http.response_body`.

8.15.20 http.response_header

The `http.response_header` keyword is used to match on the name and value of an HTTP/1 or HTTP/2 request.

It is possible to use any of the *Payload Keywords* with the `http.response_header` keyword.

For HTTP/2, the header name and value get concatenated by ": " (colon and space). The colon and space are commonly noted with the hexadecimal format *|3a 20|* within signatures.

To detect if an HTTP/2 header name contains a ":" (colon), the keyword *http2.header_name* can be used.

Example HTTP Response:

```
HTTP/1.1 200 OK
Content-Type: text/html
Server: nginx/0.8.54
Location: suricata.io
```

alert http $EXTERNAL_NET any -> $HOME_NET any (msg:"HTTP Response Example"; flow:established,to_client; http.response_header; content:"Location|3a 20|suricata.io"; classtype:bad-unknown; sid:127; rev:1;)

8.15.21 http.response_line

The `http.response_line` keyword is used to match on the entire HTTP response line.

It is possible to use any of the *Payload Keywords* with the `http.response_line` keyword.

Example HTTP Response:

```
HTTP/1.1 200 OK
Content-Type: text/html
Server: nginx/0.8.54
```

alert http $EXTERNAL_NET any -> $HOME_NET any (msg:"HTTP Response Line Example"; flow:established,to_client; http.response_line; content:"HTTP/1.1 200 OK"; classtype:bad-unknown; sid:119; rev:1;)

> **Note**
>
> `http.response_line` does not include the trailing \r\n

8.15.22 http.server

The `http.server` keyword is used to match on the HTTP response server header contents.

It is possible to use any of the *Payload Keywords* with the `http.server` keyword.

Example HTTP Response:

```
HTTP/1.1 200 OK
Content-Type: text/html
Server: nginx/0.8.54
```

alert http $EXTERNAL_NET any -> $HOME_NET any (msg:"HTTP Server Example"; flow:established,to_client; http.server; content:"nginx/0.8.54"; bsize:12; classtype:bad-unknown; sid:121; rev:1;)

> **Note**
>
> `http.server` does not include the leading space or trailing \r\n

> **Note**
>
> `http.server` can have additional formatting/normalization applied to buffer contents, see *Normalization* for additional details.

8.15.23 http.stat_code

The `http.stat_code` keyword is used to match on the HTTP status code that can be present in an HTTP response.

It is possible to use any of the *Payload Keywords* with the `http.stat_code` keyword.

Example HTTP Response:

```
HTTP/1.1 200 OK
Content-Type: text/html
Server: nginx/0.8.54
```

alert http $EXTERNAL_NET any -> $HOME_NET any (msg:"HTTP Stat Code Response Example"; flow:established,to_client; http.stat_code; content:"200"; classtype:bad-unknown; sid:117; rev:1;)

> **Note**
>
> `http.stat_code` does not include the leading or trailing space

8.15.24 http.stat_msg

The `http.stat_msg` keyword is used to match on the HTTP status message that can be present in an HTTP response.

For HTTP/2, an empty buffer is returned by Suricata. See rfc 7540 section 8.1.2.4. about Response Pseudo-Header Fields.

It is possible to use any of the *Payload Keywords* with the `http.stat_msg` keyword.

Example HTTP Response:

```
HTTP/1.1 200 OK
Content-Type: text/html
Server: nginx/0.8.54
```

alert http $EXTERNAL_NET any -> $HOME_NET any (msg:"HTTP Stat Message Response Example"; flow:established,to_client; http.stat_msg; content:"OK"; classtype:bad-unknown; sid:118; rev:1;)

> **Note**
>
> `http.stat_msg` does not include the leading space or trailing \r\n

Note

`http.stat_msg` will always be empty when used with HTTP/2

8.15.25 file.data

With `file.data`, the HTTP response body is inspected, just like with `http.response_body`. `file.data` also works for HTTP request body and can be used in protocols other than HTTP.

It is possible to use any of the *Payload Keywords* with the `file.data` keyword.

Example HTTP Response:

```
HTTP/1.1 200 OK
Content-Type: text/html
Server: nginx/0.8.54

Server response body
```

alert http $EXTERNAL_NET any -> $HOME_NET any (msg:"HTTP file.data Example"; flow:established,to_client; file.data; content:"Server response body"; classtype:bad-unknown; sid:128; rev:1;)

The body of an HTTP response can be very large, therefore the response body is inspected in definable chunks.

How much of the response/server body is inspected is controlled in the *libhtp configuration section* via the `response-body-limit` setting.

Note

If the HTTP body is a flash file compressed with 'deflate' or 'lzma', it can be decompressed and `file.data` can match on the decompressed data. Flash decompression must be enabled under 'libhtp' configuration:

```
# Decompress SWF files.
# 2 types: 'deflate', 'lzma', 'both' will decompress deflate and lzma
# compress-depth:
# Specifies the maximum amount of data to decompress,
# set 0 for unlimited. The maximum is 4294967282 bytes.
# decompress-depth:
# Specifies the maximum amount of decompressed data to obtain. The
# output buffer grows on demand up to this limit. Set 0 to use the
# default limit of 50 MB. The maximum is 2147483640 bytes.
swf-decompression:
  enabled: yes
  type: both
  compress-depth: 0
  decompress-depth: 0
```

Note

`file.data` replaces the previous keyword name, `file_data`. `file_data` can still be used but it is recommended that rules be converted to use `file.data`.

Note

If an HTTP body is using gzip or deflate, `file.data` will match on the decompressed data.

Note

Negated matching is affected by the chunked inspection. E.g. 'content:!"<html";' could not match on the first chunk, but would then possibly match on the 2nd. To avoid this, use a depth setting. The depth setting takes the body size into account. Assuming that the `response-body-minimal-inspect-size` is bigger than 1k, 'content:!"<html"; depth:1024;' can only match if the pattern '<html' is absent from the first inspected chunk.

Note

Additional information can be found at *File Keywords*

Note

`file.data` supports multiple buffer matching, see *Multiple Buffer Matching*.

8.15.26 http.connection

The `http.connection` keyword is used to match on the Connection field that can be present in HTTP request or response headers.

It is possible to use any of the *Payload Keywords* with the `http.connection` keyword.

Example HTTP Request:

```
GET /index.html HTTP/1.1
User-Agent: Mozilla/5.0
Accept-Language: en-US
Host: suricata.io
Connection: Keep-Alive
```

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP Connection Example"; flow:established,to_server; http.connection; content:"Keep-Alive"; bsize:10; classtype:bad-unknown; sid:94; rev:1;)

Note

`http.connection` does not include the leading space or trailing \r\n

Note

`http.connection` can have additional formatting/normalization applied to buffer contents, see *Normalization* for additional details.

8.15.27 http.content_len

The `http.content_len` keyword is used to match on the Content-Length field that can be present in HTTP request or response headers. Use `flow:to_server` or `flow:to_client` to force inspection of the request or response respectively.

It is possible to use any of the *Payload Keywords* with the `http.content_len` keyword.

Example HTTP Request:

```
POST /suricata.php HTTP/1.1
Content-Type: multipart/form-data; boundary=---------------123
Host: suricata.io
Content-Length: 100
Connection: Keep-Alive
```

Example HTTP Response:

```
HTTP/1.1 200 OK
Content-Type: text/html
Server: nginx/0.8.54
Connection: Close
Content-Length: 20
```

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP Content-Length Request Example"; flow:established,to_server; http.content_len; content:"100"; bsize:3; classtype:bad-unknown; sid:97; rev:1;)

alert http $EXTERNAL_NET any -> $HOME_NET any (msg:"HTTP Content-Length Response Example"; flow:established,to_client; http.content_len; content:"20"; bsize:2; classtype:bad-unknown; sid:98; rev:1;)

To do numeric evaluation of the content length, *byte_test* can be used.

If we want to match on an HTTP request content length equal to and greater than 100 we could use the following signature.

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP Content-Length Request Byte Test Example"; flow:established,to_server; http.content_len; byte_test:0,>=,100,0,string,dec; classtype:bad-unknown; sid:99; rev:1;)

> **Note**
>
> `http.content_len` does not include the leading space or trailing \r\n

8.15.28 http.content_type

The `http.content_type` keyword is used to match on the Content-Type field that can be present in HTTP request or response headers. Use `flow:to_server` or `flow:to_client` to force inspection of the request or response respectively.

It is possible to use any of the *Payload Keywords* with the `http.content_type` keyword.

Example HTTP Request:

```
POST /suricata.php HTTP/1.1
Content-Type: multipart/form-data; boundary=---------------123
Host: suricata.io
Content-Length: 100
Connection: Keep-Alive
```

Example HTTP Response:

```
HTTP/1.1 200 OK
Content-Type: text/html
Server: nginx/0.8.54
Connection: Close
```

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP Content-Type Request Example"; flow:established,to_server; http.content_type; content:"multipart/form-data|3b 20|"; startswith; classtype:bad-unknown; sid:95; rev:1;)

alert http $EXTERNAL_NET any -> $HOME_NET any (msg:"HTTP Content-Type Response Example"; flow:established,to_client; http.content_type; content:"text/html"; bsize:9; classtype:bad-unknown; sid:96; rev:1;)

Note

`http.content_type` does not include the leading space or trailing \r\n

Note

`http.content_type` can have additional formatting/normalization applied to buffer contents, see *Normalization* for additional details.

8.15.29 http.cookie

The `http.cookie` keyword is used to match on the cookie field that can be present in HTTP request (Cookie) or HTTP response (Set-Cookie) headers.

It is possible to use any of the *Payload Keywords* with both `http.header` keywords.

Example HTTP Request:

```
GET /index.html HTTP/1.1
User-Agent: Mozilla/5.0
Cookie: PHPSESSION=123
Host: suricata.io
```

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP Cookie Example"; flow:established,to_server; http.cookie; content:"PHPSESSIONID=123"; bsize:16; classtype:bad-unknown; sid:80; rev:1;)

Note

Cookies are passed in HTTP headers but Suricata extracts the cookie data to `http.cookie` and will not match cookie content put in the *http.header* sticky buffer.

Note

`http.cookie` does not include the leading space or trailing \r\n

Note

`http.cookie` can have additional formatting/normalization applied to buffer contents, see *Normalization* for additional details.

8.15.30 http.header

Matching on HTTP headers has two options in Suricata, the `http.header` and the `http.header.raw`.

It is possible to use any of the *Payload Keywords* with both `http.header` keywords.

The `http.header` keyword normalizes the header contents. For example if header contents contain trailing whitespace or tab characters, those would be removed.

To match on non-normalized header data, use the *http.header.raw* keyword.

Normalization Example:

```
GET /index.html HTTP/1.1
User-Agent: Mozilla/5.0     \r\n
Host: suricata.io
```

Would be normalized to Mozilla/5.0\r\n

Example HTTP Request:

```
GET /index.html HTTP/1.1
User-Agent: Mozilla/5.0
Host: suricata.io
```

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP Header Example 1"; flow:established,to_server; http.header; content:"User-Agent|3a 20|Mozilla/5.0|0d 0a|"; classtype:bad-unknown; sid:70; rev:1;)

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP Header Example 2"; flow:established,to_server; http.header; content:"Host|3a 20|suricata.io|0d 0a|"; classtype:bad-unknown; sid:71; rev:1;)

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP Header Example 3"; flow:established,to_server; http.header; content:"User-Agent|3a 20|Mozilla/5.0|0d 0a|"; startswith; content:"Host|3a 20|suricata.io|0d 0a|"; classtype:bad-unknown; sid:72; rev:1;)

Note

There are headers that will not be included in the `http.header` buffer, specifically the *http.cookie* buffer.

Note

`http.header` can have additional formatting/normalization applied to buffer contents, see *Normalization* for additional details.

8.15.31 http.header.raw

The `http.header.raw` buffer matches on HTTP header content but does not have any normalization performed on the buffer contents (see *http.header*)

Abnormal HTTP Header Example:

```
GET /index.html HTTP/1.1
User-Agent: Mozilla/5.0
User-Agent: Chrome
Host: suricata.io
```

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP Header Raw Example"; flow:established,to_server; http.header.raw; content:"User-Agent|3a 20|Mozilla/5.0|0d 0a|"; content:"User-Agent|3a 20|Chrome|0d 0a|"; classtype:bad-unknown; sid:73; rev:1;)

Note

`http.header.raw` can have additional formatting applied to buffer contents, see *Normalization* for additional details.

8.15.32 http.header_names

The `http.header_names` keyword is used to match on the names of the headers in an HTTP request or response. This is useful for checking for a header's presence, absence and/or header order. Use `flow:to_server` or `flow:to_client` to force inspection of the request or response respectively.

It is possible to use any of the *Payload Keywords* with the `http.header_names` keyword.

Example HTTP Request:

```
GET / HTTP/1.1
Host: suricata.io
Connection: Keep-Alive
```

Example HTTP Response:

```
HTTP/1.1 200 OK
Content-Type: text/html
Server: nginx/0.8.54
```

Examples to match exactly on header order:

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP Header Names Request Example"; flow:established,to_server; http.header_names; content:"|0d 0a|Host|0d 0a|Connection|0d 0a 0d 0a|"; bsize:22; classtype:bad-unknown; sid:110; rev:1;)

alert http $EXTERNAL_NET any -> $HOME_NET any (msg:"HTTP Header Names Response Example"; flow:established,to_client; http.header_names; content:"|0d 0a|Content-Type|0d 0a|Server|0d 0a 0d 0a|"; bsize:26; classtype:bad-unknown; sid:111; rev:1;)

Examples to match on header existence:

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP Header Names Request Example 2"; flow:established,to_server; http.header_names; content:"|0d 0a|Host|0d 0a|"; classtype:bad-unknown; sid:112; rev:1;)

alert http $EXTERNAL_NET any -> $HOME_NET any (msg:"HTTP Header Names Response Example 2"; flow:established,to_client; http.header_names; content:"|0d 0a|Content-Type|0d 0a|"; classtype:bad-unknown; sid:113; rev:1;)

Examples to match on header absence:

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP Header Names Request Example 3"; flow:established,to_server; http.header_names; content:!"|0d 0a|User-Agent|0d 0a|"; classtype:bad-unknown; sid:114; rev:1;)

alert http $EXTERNAL_NET any -> $HOME_NET any (msg:"HTTP Header Names Response Example 3"; flow:established,to_client; http.header_names; content:!"|0d 0a|Date|0d 0a|"; classtype:bad-unknown; sid:115; rev:1;)

Example to check for the `User-Agent` header and that the `Host` header is after `User-Agent` but not necessarily directly after.

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP Header Names Request Example 4"; flow:established,to_server; http.header_names; content:"|0d 0a|Host|0d 0a|"; content:"User-Agent|0d 0a|"; distance:-2; classtype:bad-unknown; sid:114; rev:1;)

Note

`http.header_names` starts with a \r\n and ends with an extra \r\n.

Note

`http.header_names` can have additional formatting/normalization applied to buffer contents, see *Normalization* for additional details.

8.15.33 http.protocol

The `http.protocol` keyword is used to match on the protocol field that is contained in HTTP requests and responses.

For HTTP/2, the constant string "HTTP/2" is used. See rfc 7540 section 8.1.2.4. about Response Pseudo-Header Fields.

It is possible to use any of the *Payload Keywords* with the `http.protocol` keyword.

Note

`http.protocol` does not include the leading space or trailing \r\n

Example HTTP Request:

```
GET /index.html HTTP/1.1
User-Agent: Mozilla/5.0
Host: suricata.io
```

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP Protocol Example"; flow:established,to_server; http.protocol; content:"HTTP/1.1"; bsize:9; classtype:bad-unknown; sid:50; rev:1;)

8.15.34 http.start

The `http.start` keyword is used to match on the start of an HTTP request or response. This will contain the request/response line plus the request/response headers. Use `flow:to_server` or `flow:to_client` to force inspection of the request or response respectively.

It is possible to use any of the *Payload Keywords* with the `http.start` keyword.

Example HTTP Request:

```
GET / HTTP/1.1
Host: suricata.io
Connection: Keep-Alive
```

Example HTTP Response:

```
HTTP/1.1 200 OK
Content-Type: text/html
Server: nginx/0.8.54
```

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP Start Request Example"; flow:established,to_server; http.start; content:"POST / HTTP/1.1|0d 0a|Host|0d 0a|Connection|0d 0a 0d 0a|"; classtype:bad-unknown; sid:101; rev:1;)

alert http $EXTERNAL_NET any -> $HOME_NET any (msg:"HTTP Start Response Example"; flow:established,to_client; http.start; content:"HTTP/1.1 200 OK|0d 0a|Content-Type|0d 0a|Server|0d 0a 0d 0a|"; classtype:bad-unknown; sid:102; rev:1;)

Note

`http.start` contains the normalized headers and is terminated by an extra \r\n to indicate the end of the headers.

8.16 File Keywords

Suricata comes with several rule keywords to match on various file properties. They depend on properly configured *File Extraction*.

8.16.1 file.data

The `file.data` sticky buffer matches on contents of files that are seen in flows that Suricata evaluates. The various payload keywords can be used (e.g. `startswith`, `nocase` and `bsize`) with `file.data`.

alert smtp any any -> any any (msg:"smtp app layer file.data example"; file.data; content:"example file content"; sid:1; rev:1;)

alert http any any -> any any (msg:"http app layer file.data example"; file.data; content:"example file content"; sid:2; rev:1;)

alert http2 any any -> any any (msg:"http2 app layer file.data example"; file.data; content:"example file content"; sid:3; rev:1;)

alert nfs any any -> any any (msg:"nfs app layer file.data example"; file.data; content:" "; sid:5; rev:1;)

alert ftp-data any any -> any any (msg:"ftp app layer file.data example"; file.data; content:"example file content"; sid:6; rev:1;)

alert tcp any any -> any any (msg:"tcp file.data example"; file.data; content:"example file content"; sid:4; rev:1;)

Note file_data is the legacy notation but can still be used.

8.16.2 file.name

`file.name` is a sticky buffer that is used to look at filenames that are seen in flows that Suricata evaluates. The various payload keywords can be used (e.g. `startswith`, `nocase` and `bsize`) with `file.name`.

Example:

```
file.name; content:"examplefilename";
```

`file.name` supports multiple buffer matching, see *Multiple Buffer Matching*.

Note `filename` can still be used. A notable difference between `file.name` and `filename` is that `filename` assumes `nocase` by default. In the example below the two signatures are considered the same.

Example:

```
filename:"examplefilename";

file.name; content:"examplefilename"; nocase;
```

8.16.3 fileext

`fileext` is used to look at individual file extensions that are seen in flows that Suricata evaluates.

Example:

```
fileext:"pdf";
```

Note: `fileext` does not allow partial matches. For example, if a PDF file (.pdf) is seen by a Suricata signature with fileext:"pd"; the signature will not produce an alert.

Note: `fileext` assumes `nocase` by default. This means that a file with the extension .PDF will be seen the same as if the file had an extension of .pdf.

Note: `fileext` and `file.name` can both be used to match on file extensions. In the example below the two signatures are considered the same.

Example:

```
fileext:"pdf";

file.name; content:".pdf"; nocase; endswith;
```

Note: While `fileeext` and `file.name` can both be used to match on file extensions, `file.name` allows for partial matching on file extensions. The following would match on a file with the extension of .pd as well as .pdf.

Example:

```
file.name; content:".pd";
```

8.16.4 file.magic

Matches on the information libmagic returns about a file.

Example:

```
file.magic; content:"executable for MS Windows";
```

Note `filemagic` can still be used. The only difference between `file.magic` and `file.magic` is that `filemagic` assumes `nocase` by default. In the example below the two signatures are considered the same.

Example:

```
filemagic:"executable for MS Windows";

file.magic; content:"executable for MS Windows"; nocase;
```

Note: Suricata currently uses its underlying operating systems version/implementation of libmagic. Different versions and implementations of libmagic do not return the same information. Additionally there are varying Suricata performance impacts based on the version and implementation of libmagic. Additional information about Suricata and libmagic can be found here: https://redmine.openinfosecfoundation.org/issues/437

`file.magic` supports multiple buffer matching, see *Multiple Buffer Matching*.

8.16.5 filestore

Stores files to disk if the signature matched.

Syntax:

```
filestore:<direction>,<scope>;
```

direction can be:

- request/to_server: store a file in the request / to_server direction
- response/to_client: store a file in the response / to_client direction
- both: store both directions

scope can be:

- file: only store the matching file (for filename,fileext,filemagic matches)
- tx: store all files from the matching HTTP transaction
- ssn/flow: store all files from the TCP session/flow.

If direction and scope are omitted, the direction will be the same as the rule and the scope will be per file.

8.16.6 filemd5

Match file *MD5* against list of MD5 checksums.

Syntax:

```
filemd5:[!]filename;
```

The filename is expanded to include the rule dir. In the default case it will become /etc/suricata/rules/filename. Use the exclamation mark to get a negated match. This allows for white listing.

Examples:

```
filemd5:md5-blacklist;
filemd5:!md5-whitelist;
```

File format

The file format is simple. It's a text file with a single md5 per line, at the start of the line, in hex notation. If there is extra info on the line it is ignored.

Output from md5sum is fine:

```
2f8d0355f0032c3e6311c6408d7c2dc2  util-path.c
b9cf5cf347a70e02fde975fc4e117760  util-pidfile.c
02aaa6c3f4dbae65f5889eeb8f2bbb8d  util-pool.c
dd5fc1ee7f2f96b5f12d1a854007a818  util-print.c
```

Just MD5's are good as well:

```
2f8d0355f0032c3e6311c6408d7c2dc2
b9cf5cf347a70e02fde975fc4e117760
02aaa6c3f4dbae65f5889eeb8f2bbb8d
dd5fc1ee7f2f96b5f12d1a854007a818
```

Memory requirements

Each MD5 uses 16 bytes of memory. 20 Million MD5's use about 310 MiB of memory.

See also: https://blog.inliniac.net/2012/06/09/suricata-md5-blacklisting/

8.16.7 filesha1

Match file SHA1 against list of SHA1 checksums.

Syntax:

```
filesha1:[!]filename;
```

The filename is expanded to include the rule dir. In the default case it will become /etc/suricata/rules/filename. Use the exclamation mark to get a negated match. This allows for white listing.

Examples:

```
filesha1:sha1-blacklist;
filesha1:!sha1-whitelist;
```

File format

Same as md5 file format.

8.16.8 filesha256

Match file SHA256 against list of SHA256 checksums.

Syntax:

```
filesha256:[!]filename;
```

The filename is expanded to include the rule dir. In the default case it will become /etc/suricata/rules/filename. Use the exclamation mark to get a negated match. This allows for white listing.

Examples:

```
filesha256:sha256-blacklist;
filesha256:!sha256-whitelist;
```

File format

Same as md5 file format.

8.16.9 filesize

Match on the size of the file as it is being transferred.

filesize uses an *unsigned 64-bit integer*.

Syntax:

```
filesize:<value>;
```

Possible units are KB, MB and GB, without any unit the default is bytes.

Examples:

```
filesize:100; # exactly 100 bytes
filesize:100<>200; # greater than 100 and smaller than 200
filesize:>100MB; # greater than 100 megabytes
filesize:<100MB; # smaller than 100 megabytes
```

Note: For files that are not completely tracked because of packet loss or stream.reassembly.depth being reached on the "greater than" is checked. This is because Suricata can know a file is bigger than a value (it has seen some of it already), but it can't know if the final size would have been within a range, an exact value or smaller than a value.

8.17 DNS Keywords

Suricata supports sticky buffers as well as keywords for efficiently matching on specific fields in DNS messages.

Note that sticky buffers are expected to be followed by one or more *Payload Keywords*.

8.17.1 dns.opcode

This keyword matches on the **opcode** found in the DNS header flags.

dns.opcode uses an *unsigned 8-bit integer*.

Syntax

```
dns.opcode:[!]<number>
dns.opcode:[!]<number1>-<number2>
```

Examples

Match on DNS requests and responses with **opcode** 4:

```
dns.opcode:4;
```

Match on DNS requests where the **opcode** is NOT 0:

```
dns.opcode:!0;
```

Match on DNS requests where the **opcode** is between 7 and 15, exclusively:

```
dns.opcode:7-15;
```

Match on DNS requests where the **opcode** is not between 7 and 15:

```
dns.opcode:!7-15;
```

8.17.2 dns.rcode

This keyword matches on the **rcode** field found in the DNS header flags.

dns.rcode uses an *unsigned 8-bit integer*. It can also be specified by text from the enumeration.

See `suricata --list-keywords=json | jq '."dns.rcode".enum_values'`

Currently, Suricata only supports rcode values in the range [0-15], while the current DNS version supports rcode values from [0-23] as specified in RFC 6895.

We plan to extend the rcode values supported by Suricata according to RFC 6895 as tracked by the ticket: https://redmine.openinfosecfoundation.org/issues/6650

Syntax

```
dns.rcode:[!]<number>
dns.rcode:[!]<number1>-<number2>
```

Examples

Match on DNS requests and responses with **rcode** 4:

```
dns.rcode:4;
```

Match on DNS requests and responses where the **rcode** is NOT 0:

```
dns.rcode:!0;
```

8.17.3 dns.rrtype

This keyword matches on the **rrtype** (integer) found in the DNS message.

dns.rrtype uses an *unsigned 16-bit integer*.

dns.rrtype is also a *multi-integer*.

See `suricata --list-keywords=json | jq '."dns.rrtype".enum_values'`

It can also be specified by text from the enumeration.

Syntax

```
dns.rrtype:[!]<number>
```

Examples

Match on DNS requests and responses with **rrtype** 4:

```
dns.rrtype:4;
```

Match on DNS requests and responses where the **rrtype** is NOT 0:

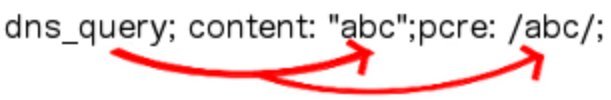
```
dns.rrtype:!0;
```

8.17.4 dns.query

`dns.query` is a sticky buffer that is used to inspect DNS query names in DNS request messages.

alert dns any any -> any any (msg:"Test dns.query option"; dns.query; content:"google"; nocase; sid:1;)

Being a sticky buffer, payload keywords such as content are to be used after `dns.query`:



The `dns.query` keyword affects all following contents, until pkt_data is used or it reaches the end of the rule.

> **Note**
>
> **dns.query** is equivalent to the older **dns_query**.

> **Note**
>
> **dns.query** will only match on DNS request messages, to also match on DNS response message, see *dns.queries.rrname*.

`dns.queries.rrname` supports *Multiple Buffer Matching*.

Normalized Buffer

Buffer contains literal domain name

- <length> values (as seen in a raw DNS request) are literal '.' characters
- no leading <length> value
- No terminating NULL (0x00) byte (use a negated relative `isdataat` to match the end)

Example DNS request for "mail.google.com" (for readability, hex values are encoded between pipes):

DNS query on the wire (snippet):

```
|04|mail|06|google|03|com|00|
```

`dns.query` buffer:

```
mail.google.com
```

8.17.5 dns.queries.rrname

`dns.queries.rrname` is a sticky buffer that is used to look at the name field in DNS query (question) resource records. It is nearly identical to `dns.query` but supports both DNS requests and responses.

`dns.queries.rrname` will look at both requests and responses, so `flow` is recommended to confine to a specific direction.

The buffer being matched on contains the complete re-assembled resource name, for example "www.suricata.io".

`dns.queries.rrname` supports *Multiple Buffer Matching*.

`dns.queries.rrname` was introduced in Suricata 8.0.0.

8.17.6 dns.answers.rrname

`dns.answers.rrname` is a sticky buffer that is used to look at the name field in DNS answer resource records.

`dns.answers.rrname` will look at both requests and responses, so `flow` is recommended to confine to a specific direction.

The buffer being matched on contains the complete re-assembled resource name, for example "www.suricata.io".

`dns.answers.rrname` supports *Multiple Buffer Matching*.

`dns.answers.rrname` was introduced in Suricata 8.0.0.

8.17.7 dns.authorities.rrname

`dns.authorities.rrname` is a sticky buffer that is used to look at the rrname field in DNS authority resource records.

`dns.authorities.rrname` will look at both requests and responses, so `flow` is recommended to confine to a specific direction.

The buffer being matched on contains the complete re-assembled resource name, for example "www.suricata.io".

`dns.authorities.rrname` supports *Multiple Buffer Matching*.

`dns.authorities.rrname` was introduced in Suricata 8.0.0.

8.17.8 dns.additionals.rrname

`dns.additionals.rrname` is a sticky buffer that is used to look at the rrname field in DNS additional resource records.

`dns.additionals.rrname` will look at both requests and responses, so `flow` is recommended to confine to a specific direction.

The buffer being matched on contains the complete re-assembled resource name, for example "www.suricata.io".

`dns.additionals.rrname` supports *Multiple Buffer Matching*.

`dns.additionals.rrname` was introduced in Suricata 8.0.0.

8.17.9 dns.response.rrname

`dns.response.rrname` is a sticky buffer that is used to look at all name and rdata fields of DNS response (answer) resource records that are represented as a resource name (hostname). It supports inspecting all DNS response sections.

alert dns any any -> any any (msg:"Test dns.response.rrname option";
dns.response.rrname; content:"google"; nocase; sid:1;)

`rdata` field matching supports a subset of types that contain domain name structured data, for example: "www.suricata.io". The list of types inspected is:

- CNAME
- PTR
- MX
- NS
- SOA (mname data: primary name server)

The buffer being matched on contains the complete re-assembled resource name, for example "www.suricata.io".

`dns.response.rrname` supports *Multiple Buffer Matching*.

`dns.response.rrname` was introduced in Suricata 8.0.0.

8.18 mDNS Keywords

Suricata supports sticky buffers for efficiently matching on specific fields in mDNS (Multicast DNS) messages.

Note that sticky buffers are expected to be followed by one or more *Payload Keywords*.

8.18.1 mdns.queries.rrname

`mdns.queries.rrname` is a sticky buffer that is used to look at the name field in mDNS query resource records.

The buffer being matched on contains the complete re-assembled resource name, for example "host.local".

`mdns.queries.rrname` supports *Multiple Buffer Matching*.

alert udp any any -> any 5353 (msg:"mDNS query for .local domain";
mdns.queries.rrname; content:".local"; sid:1;)

8.18.2 mdns.answers.rrname

`mdns.answers.rrname` is a sticky buffer that is used to look at the name field in mDNS answer resource records.

The buffer being matched on contains the complete re-assembled resource name, for example "printer.local".

`mdns.answers.rrname` supports *Multiple Buffer Matching*.

alert udp any 5353 -> any any (msg:"mDNS answer for printer.local";
mdns.answers.rrname; content:"printer.local"; sid:2;)

8.18.3 mdns.authorities.rrname

`mdns.authorities.rrname` is a sticky buffer that is used to look at the rrname field in mDNS authority resource records.

The buffer being matched on contains the complete re-assembled resource name, for example "device.local".

`mdns.authorities.rrname` supports *Multiple Buffer Matching*.

alert udp any 5353 -> any any (msg:"mDNS authority record check";
mdns.authorities.rrname; content:"auth.local"; sid:3;)

8.18.4 mdns.additionals.rrname

`mdns.additionals.rrname` is a sticky buffer that is used to look at the rrname field in mDNS additional resource records.

The buffer being matched on contains the complete re-assembled resource name, for example "service.local".

`mdns.additionals.rrname` supports *Multiple Buffer Matching*.

alert udp any any -> any 5353 (msg:"mDNS additional record check";
mdns.additionals.rrname; content:"_companion-link._tcp.local"; nocase; sid:4;)

8.18.5 mdns.response.rrname

`mdns.response.rrname` is a sticky buffer that is used to inspect all the rrname fields in a response, in the queries, answers, additionals and authorities. Additionally it will also inspect rdata fields that have the same format as an rrname (hostname).

`rdata` types that will be inspected are:

- CNAME

- PTR
- MX
- NS
- SOA

alert udp any 5353 -> any any (msg:"mDNS answer data match";
mdns.response.rrname; content:"Apple TV"; sid:5;)

8.19 LLMNR Keywords

Suricata supports sticky buffers for matching on specific fields in LLMNR (Link-Local Multicast Name Resolution) messages.

Note that sticky buffers are expected to be followed by one or more *Payload Keywords*.

8.19.1 llmnr.queries.rrname

`llmnr.queries.rrname` is a sticky buffer that is used to look at the name field in LLMNR query resource records.

`llmnr.queries.rrname` will look at both requests and responses, so `flow` is recommended to confine to a specific direction.

The buffer being matched on contains the complete re-assembled resource name, for example "workstation.local".

`llmnr.queries.rrname` supports *Multiple Buffer Matching*.

alert llmnr any any -> any 5355 (msg:"LLMNR query for workstation";
flow:to_server; llmnr.queries.rrname; content:"workstation"; nocase; sid:1;)

8.19.2 llmnr.answers.rrname

`llmnr.answers.rrname` is a sticky buffer that is used to look at the name field in LLMNR answer resource records.

`llmnr.answers.rrname` will look at both requests and responses, so `flow` is recommended to confine to a specific direction.

The buffer being matched on contains the complete re-assembled resource name, for example "server.local".

`llmnr.answers.rrname` supports *Multiple Buffer Matching*.

alert llmnr any 5355 -> any any (msg:"LLMNR answer for server.local";
flow:to_client; llmnr.answers.rrname; content:"server.local"; sid:2;)

8.19.3 llmnr.authorities.rrname

`llmnr.authorities.rrname` is a sticky buffer that is used to look at the rrname field in LLMNR authority resource records.

`llmnr.authorities.rrname` will look at both requests and responses, so `flow` is recommended to confine to a specific direction.

The buffer being matched on contains the complete re-assembled resource name, for example "local".

`llmnr.authorities.rrname` supports *Multiple Buffer Matching*.

alert llmnr any 5355 -> any any (msg:"LLMNR authority record check";
llmnr.authorities.rrname; content:"local"; sid:3;)

8.19.4 llmnr.additionals.rrname

`llmnr.additionals.rrname` is a sticky buffer that is used to look at the rrname field in LLMNR additional resource records.

`llmnr.additionals.rrname` will look at both requests and responses, so `flow` is recommended to confine to a specific direction.

The buffer being matched on contains the complete re-assembled resource name, for example "ns.local".

`llmnr.additionals.rrname` supports *Multiple Buffer Matching*.

alert llmnr any any -> any 5355 (msg:"LLMNR additional record check";
llmnr.additionals.rrname; content:"ns.local"; sid:4;)

8.19.5 llmnr.response.rrname

`llmnr.response.rrname` is a sticky buffer that is used to inspect all the rrname fields in a response, in the queries, answers, additionals and authorities. Additionally it will also inspect rdata fields that have the same format as an rrname (hostname).

`rdata` types that will be inspected are:

- CNAME
- PTR
- MX
- NS
- SOA

alert llmnr any 5355 -> any any (msg:"LLMNR response contains suspicious domain";
flow:to_client; llmnr.response.rrname; content:"malicious"; nocase; sid:5;)

8.20 SSL/TLS Keywords

Suricata comes with several rule keywords to match on various properties of TLS/SSL handshake. Matches are string inclusion matches.

8.20.1 tls.cert_subject

Match TLS/SSL certificate Subject field.

Examples:

```
tls.cert_subject; content:"CN=*.googleusercontent.com"; isdataat:!1,relative;
tls.cert_subject; content:"google.com"; nocase; pcre:"/google\.com$/";
```

`tls.cert_subject` is a 'sticky buffer'.

`tls.cert_subject` can be used as `fast_pattern`.

`tls.cert_subject` supports multiple buffer matching, see *Multiple Buffer Matching*.

tls.subject

Legacy keyword to match TLS/SSL certificate Subject field.

example:

```
tls.subject:"CN=*.googleusercontent.com"
```

Case sensitive, can't use 'nocase', or other modifiers.

Note: `tls.cert_subject` replaces the following legacy keywords: `tls_cert_subject` and `tls.subject`. It's recommended that rules be converted to use the new one.

8.20.2 tls.cert_issuer

Match TLS/SSL certificate Issuer field.

Examples:

```
tls.cert_issuer; content:"WoSign"; nocase; isdataat:!1,relative;
tls.cert_issuer; content:"StartCom"; nocase; pcre:"/StartCom$/";
```

`tls.cert_issuer` is a 'sticky buffer'.

`tls.cert_issuer` can be used as `fast_pattern`.

tls.issuerdn

Legacy keyword to match TLS/SSL certificate IssuerDN field

example:

```
tls.issuerdn:!"CN=Google-Internet-Authority"
```

Case sensitive, can't use 'nocase', or other modifiers.

Note: `tls.cert_issuer` replaces the following legacy keywords: `tls_cert_issuer` and `tls.issuerdn`. It's recommended that rules be converted to use the new one.

8.20.3 tls.cert_serial

Match on the serial number in a certificate.

alert tls any any -> any any (msg:"match cert serial"; tls.cert_serial; content:"5C:19:B7:B1:32:3B:1C:A1"; sid:200012;)

`tls.cert_serial` is a 'sticky buffer'.

`tls.cert_serial` can be used as `fast_pattern`.

`tls.cert_serial` replaces the previous keyword name: `tls_cert_serial`. You may continue to use the previous name, but it's recommended that rules be converted to use the new name.

8.20.4 tls.cert_fingerprint

Match on the SHA-1 fingerprint of the certificate.

alert tls any any -> any any (msg:"match cert fingerprint"; tls.cert_fingerprint; content:"4a:a3:66:76:82:cb:6b:23:bb:c3:58:47:23:a4:63:a7:78:a4:a1:18"; sid:200023;)

`tls.cert_fingerprint` is a 'sticky buffer'.

`tls.cert_fingerprint` can be used as `fast_pattern`.

`tls.cert_fingerprint` replaces the previous keyword name: `tls_cert_fingerprint` may continue to use the previous name, but it's recommended that rules be converted to use the new name.

8.20.5 tls.sni

Match TLS/SSL Server Name Indication field.

Examples:

```
tls.sni; content:"oisf.net"; nocase; isdataat:!1,relative;
tls.sni; content:"oisf.net"; nocase; pcre:"/oisf.net$/";
```

`tls.sni` is a 'sticky buffer'.

`tls.sni` can be used as `fast_pattern`.

`tls.sni` replaces the previous keyword name: `tls_sni`. You may continue to use the previous name, but it's recommended that rules be converted to use the new name.

8.20.6 tls.subjectaltname

Match TLS/SSL Subject Alternative Name field.

Examples:

```
tls.subjectaltname; content:"|73 75 72 69 63 61 74 61 2e 69 6f|";
```

`tls.subjectaltname` is a 'sticky buffer'.

`tls.subjectaltname` can be used as `fast_pattern`.

`tls.subjectaltname` supports multiple buffer matching, see *Multiple Buffer Matching*.

8.20.7 tls_cert_notbefore

Match on the NotBefore field in a certificate.

alert tls any any -> any any (msg:"match cert NotBefore";
tls_cert_notbefore:1998-05-01<>2008-05-01; sid:200005;)

8.20.8 tls_cert_notafter

Match on the NotAfter field in a certificate.

alert tls any any -> any any (msg:"match cert NotAfter";
tls_cert_notafter:>2015; sid:200006;)

For omitted formats, the first date is assumed.

`YYYY-MM` => `YYYY-MM-01`.

`YYYY` => `YYYY-01-01`.

8.20.9 tls_cert_expired

Match returns true if certificate is expired. It evaluates the validity date from the certificate.

Usage:

```
tls_cert_expired;
```

8.20.10 tls_cert_valid

Match returns true if certificate is not expired. It only evaluates the validity date. It does *not* do cert chain validation. It is the opposite of `tls_cert_expired`.

Usage:

```
tls_cert_valid;
```

8.20.11 tls.certs

Do a "raw" match on each of the certificates in the TLS certificate chain.

alert tls any any -> any any (msg:"match bytes in TLS cert"; tls.certs;
content:"|06 09 2a 86|"; sid:200070;)

`tls.certs` is a 'sticky buffer'.

`tls.certs` can be used as `fast_pattern`.

`tls.certs` supports multiple buffer matching, see *Multiple Buffer Matching*.

8.20.12 tls.version

Match on negotiated TLS/SSL version.

Supported values: "1.0", "1.1", "1.2", "1.3"

It is also possible to match versions using a hex string.

Examples:

```
tls.version:1.2;
tls.version:0x7f12;
```

The first example matches TLSv1.2, whilst the last example matches TLSv1.3 draft 16.

8.20.13 ssl_version

Match version of SSL/TLS record.

Supported values "sslv2", "sslv3", "tls1.0", "tls1.1", "tls1.2", "tls1.3"

alert tls any any -> any any (msg:"match TLSv1.2";
ssl_version:tls1.2; sid:200030;)

It is also possible to match on several versions at the same time.

alert tls any any -> any any (msg:"match SSLv2 and SSLv3";
ssl_version:sslv2,sslv3; sid:200031;)

The list can be prefixed with `!` to match if version is different than all the versions listed in the signature. Such a negation does not match on a yet undetermined version.

8.20.14 tls.fingerprint

match TLS/SSL certificate SHA1 fingerprint

example:

```
tls.fingerprint:!"f3:40:21:48:70:2c:31:bc:b5:aa:22:ad:63:d6:bc:2e:b3:46:e2:5a"
```

Case sensitive, can't use 'nocase'.

The tls.fingerprint buffer is lower case so you must use lower case letters for this to match.

8.20.15 tls.store

store TLS/SSL certificate on disk. The location can be specified in the *output.tls-store.certs-log-dir* parameter of the yaml configuration file, cf *TLS parameters and certificates logging (tls.log)*..

8.20.16 ssl_state

The `ssl_state` keyword matches the state of the SSL connection. The possible states are `client_hello`, `server_hello`, `client_keyx`, `server_keyx` and `unknown`. You can specify several states with `|` (OR) to check for any of the specified states.

8.20.17 tls.random

Matches on the 32 bytes of the TLS random field from the client hello or server hello records.

alert tls any any -> any any (msg:"TLS random test";
tls.random; content:"|9b ce 7a 5e 57 5d 77 02 07 c2 9d be 24 01 cc f0 5d cd e1 d2 a5 86 9c 4a 3e ee 38 db 55 1a d9 bc|"; sid: 200074;)

`tls.random` is a sticky buffer.

8.20.18 tls.random_time

Matches on the first 4 bytes of the TLS random field from the client hello or server hello records.

alert tls any any -> any any (msg:"TLS random_time test";
tls.random_time; content:"|9b ce 7a 5e|"; sid: 200075;)

`tls.random_time` is a sticky buffer.

8.20.19 tls.random_bytes

Matches on the last 28 bytes of the TLS random field from the client hello or server hello records.

alert tls any any -> any any (msg:"TLS random_bytes test";
tls.random_bytes; content:"|57 5d 77 02 07 c2 9d be 24 01 cc f0 5d cd e1 d2 a5 86 9c 4a 3e ee 38 db 55 1a d9 bc|"; sid: 200076;)

`tls.random_bytes` is a sticky buffer.

8.20.20 tls.cert_chain_len

Matches on the TLS certificate chain length.

tls.cert_chain_len uses an *unsigned 32-bit integer*.

tls.cert_chain_len supports *<*, *>*, *<>*, *!* and using an exact value.

alert tls any any -> any any (msg:"cert chain exact value"; tls.cert_chain_len:1; classtype:misc-activity; sid:1; rev:1;)

alert tls any any -> any any (msg:"cert chain less than value"; tls.cert_chain_len:<2; classtype:misc-activity; sid:2; rev:1;)

alert tls any any -> any any (msg:"cert chain greater than value"; tls.cert_chain_len:>0; classtype:misc-activity; sid:2; rev:1;)

alert tls any any -> any any (msg:"cert chain greater than less than value";tls.cert_chain_len:0<>2; classtype:misc-activity; sid:3; rev:1;)

alert tls any any -> any any (msg:"cert chain not value"; tls.cert_chain_len:!2; classtype:misc-activity; sid:4; rev:1;)

8.20.21 tls.alpn

Matches on the ALPN buffers.

alert tls any any -> any any (msg:"TLS ALPN test";
tls.alpn; content:"http/1.1"; sid:1;)

`tls.alpn` is a sticky buffer.

8.21 SSH Keywords

Suricata has several rule keywords to match on different elements of SSH connections.

8.21.1 Hooks

The SSH parser exposes a per-direction state machine. A rule that hooks one of the states below is evaluated once the transaction's progress in that direction has reached the state; for firewall rules the verdict applies while the transaction stays at the hooked state and stops applying once it advances past it (see *app-layer-state*).

Request (`to_server`) side:

State	Meaning
`reque`	Version exchange: the banner (version) phase; the first packets of the direction. A banner line of 256 bytes or more without an end-of-line keeps the direction here: the line is consumed greedily until the end-of-line, and the condition is observable through the `ssh.long_banner` event (the banner data is published when the long line first parses).
`reque`	A valid banner line was parsed; key exchange. `ssh.proto`, `ssh.software` and the hassh buffers are registered at this state. A long banner can publish them while the line is still open; if the line never completes, the values are not inspected at a clean flow end (the direction stays below the registered state); a flow that ends disrupted (depth truncation or async reassembly) reads as past this state, and the values are inspected there
`reque`	A `new keys` record was seen in this direction: the session is established for that direction; the two directions are tracked independently (a parse failure in either ends the flow's app-layer parsing, see Completion and failure). `new keys` is a transition rather than a phase with its own data: match the record itself with a frame rule (message code 21, see the example below).

Response (`to_client`) side:

State	Meaning
`response_banı`	Server banner (version) phase: the same behavior as `request_banner` above, in the server-to-client direction.
`response_kex`	Server key exchange: same as `request_kex` — the direction's `ssh.proto` / `ssh.software` and hassh buffers are registered at this state.
`response_ses:`	A `new keys` record was seen in the server-to-client direction: same as `request_session`.

Overly long banner lines

A banner line of 256 bytes or more without an end-of-line keeps the direction in the banner state. The line is consumed greedily until the end-of-line arrives, and the direction then advances to `kex`. The condition is observable through the `ssh.long_banner` app-layer event (the banner data is published when the long line first parses). If the line does not parse as a banner when it completes, the direction fails and is frozen in `banner`.

Completion and failure

`request_done` / `response_done` is the registered completion state. No direction ever reports it: a successful direction tops out at `session`, and a direction that hits an unrecoverable error — an invalid banner or an invalid record (header or payload) — is frozen in the state the error occurred in. It is the last progress the direction reports, and the reason is available through the `ssh.invalid_banner` / `ssh.invalid_record` app-layer events and the per-direction `error` / `state` fields of the eve `ssh` object (`invalid_banner` / `invalid_record` and the state the failure occurred in; a failed flow is logged even when no banner was parsed, so the error field may be its only content). A parse failure puts the flow into an error state: nothing after the failing packet is parsed or delivered, mid-flow or at the flow-end flush, and the frozen state is what the progress accessor - and thus the flow-end eve object - reports. The `ssh.invalid_banner` / `ssh.invalid_record` events are evaluated as soon as the tx is next inspected - the failing packet itself unless the error policy skips it - at the latest the flow-end flush. A keyword registered at the failed state gets no eof run (eof requires progress above the registered state), so its flow-end evaluation stays non-terminal; states before it were already terminalized when the direction left them. The `ssh.long_banner` and `ssh.long_kex_record` events are non-fatal: the direction continues (a long banner keeps it in `banner` until the line completes; an oversized key-exchange record keeps it in `kex` until the record has been buffered in full).

Consequences worth knowing:

- `ssh:request_complete` (and the firewall policy key `request-complete`) is bound to the completion state, which no direction reports: the hook matches nothing for well-formed flows. A disrupted flow (depth truncation or async reassembly) matches, because on a disrupted flow the engine reports the completion state as the transaction progress instead of querying the parser. Before this change it matched a flow whose session was established. Review existing `accept:* ssh:request_complete` rules and `request-complete` firewall policies: a session-admit rule like that no longer fires. Failures are reported through the `ssh.invalid_banner` / `ssh.invalid_record` events instead.
- Rules cannot reference `done` (`request_done` / `response_done` do not load): it is out of both name tables. The generic hook lists for the completion state are registered under the built-in completion names (`ssh:request_complete:generic` / `ssh:response_complete:generic`); the state-name-based lists are not registered for the completion id (the id-to-name lookup returns null). They run only at a disrupted flow end, where the engine reports the completion state as the transaction progress instead of querying the parser - so Lua hooks on ssh completion no longer fire for well-formed flows; before this change the accessor reported the terminal state once both directions had sent NewKeys.
- `app-layer-state:>request_session` matches only directions that were *disrupted past session (e.g. by depth truncation or async reassembly)* — a successful request direction stops at `request_session`, and a failed one is frozen at the state it failed in (use the `ssh.invalid_*` events for that).

For firewall rules and `app-layer-state` users, *Migrating from the old SSH state names* maps the old hook names to the new phases and describes the behaviour changes.

8.21.2 Frames

The SSH parser supports the following frames:

- ssh.record_hdr
- ssh.record_data
- ssh.record_pdu

These are header + data = pdu for SSH records, after the banner and before encryption. The SSH record header is 6 bytes long : 4 bytes length, 1 byte passing, 1 byte message code.

Example:

alert ssh any any -> any any (msg:"hdr frame new keys"; frame:ssh.record.hdr; content: "|15|"; endswith; bsize: 6; sid:2;)

This rule matches like Wireshark `ssh.message_code == 0x15`.

8.21.3 ssh.proto

Match on the version of the SSH protocol used. `ssh.proto` is a sticky buffer, and can be used as a fast pattern. `ssh.proto` replaces the previous buffer name: `ssh_proto`. You may continue to use the previous name, but it's recommended that existing rules be converted to use the new name.

Format:

```
ssh.proto;
```

Example:

alert ssh any any -> any any (msg:"match SSH protocol version"; ssh.proto; content:"2.0"; sid:1000010;)

The example above matches on SSH connections with SSH version 2.0.

8.21.4 ssh.software

Match on the software string from the SSH banner. `ssh.software` is a sticky buffer, and can be used as fast pattern.

Format:

```
ssh.software;
```

Example:

alert ssh any any -> any any (msg:"match SSH software string"; ssh.software; content:"openssh"; nocase; sid:1000020;)

The example above matches on SSH connections where the software string contains "openssh".

8.21.5 ssh.hassh

Match on hassh (md5 of hassh algorithms of client).

alert ssh any any -> any any (msg:"match hassh";
ssh.hassh; content:"ec7378c1a92f5a8dde7e8b7a1ddf33d1";sid:1000010;)

`ssh.hassh` is a 'sticky buffer'.

`ssh.hassh` can be used as `fast_pattern`.

8.21.6 ssh.hassh.string

Match on Hassh string (hassh algorithms of client).

alert ssh any any -> any any (msg:"match hassh-string";
ssh.hassh.string; content:"none,zlib@openssh.com,zlib"; sid:1000030;)

`ssh.hassh.string` is a 'sticky buffer'.

`ssh.hassh.string` can be used as `fast_pattern`.

8.21.7 ssh.hassh.server

Match on hassh (md5 of hassh algorithms of server).

alert ssh any any -> any any (msg:"match SSH hash-server";
ssh.hassh.server; content:"b12d2871a1189eff20364cf5333619ee"; sid:1000020;)

`ssh.hassh.server` is a 'sticky buffer'.

`ssh.hassh.server` can be used as `fast_pattern`.

8.21.8 ssh.hassh.server.string

Match on hassh string (hassh algorithms of server).

alert ssh any any -> any any (msg:"match SSH hash-server-string";
ssh.hassh.server.string; content:"umac-64-etm@openssh.com,umac-128-etm@openssh.com"; sid:1000040;)

`ssh.hassh.server.string` is a 'sticky buffer'.

`ssh.hassh.server.string` can be used as `fast_pattern`.

8.22 JA3/JA4 Keywords

Suricata comes with JA3 (https://github.com/salesforce/ja3) and JA4 (https://github.com/FoxIO-LLC/ja4) integration. JA3 and JA4 are used to fingerprint TLS and QUIC clients.

Support must be enabled in the Suricata config file (set `app-layer.protocols.tls.ja{3,4}-fingerprints` to `yes`). If it is not explicitly disabled (`no`) , it will be enabled if a loaded rule requires it. Note that JA3/JA4 support can also be disabled at compile time; it is possible to use the `requires: feature ja{3,4};` keyword to skip rules if no JA3/JA4 support is present.

8.22.1 ja3.hash

Match on JA3 hash (md5).

alert tls any any -> any any (msg:"match JA3 hash";
ja3.hash; content:"e7eca2baf4458d095b7f45da28c16c34"; sid:100001;)

`ja3.hash` is a 'sticky buffer'.

`ja3.hash` can be used as `fast_pattern`.

`ja3.hash` replaces the previous keyword name: `ja3_hash`. You may continue to use the previous name, but it's recommended that rules be converted to use the new name.

8.22.2 ja3.string

Match on JA3 string.

alert tls any any -> any any (msg:"match JA3 string";
ja3.string; content:"19-20-21-22"; sid:100002;)

`ja3.string` is a 'sticky buffer'.

`ja3.string` can be used as `fast_pattern`.

`ja3.string` replaces the previous keyword name: `ja3_string`. You may continue to use the previous name, but it's recommended that rules be converted to use the new name.

8.22.3 ja3s.hash

Match on JA3S hash (md5).

alert tls any any -> any any (msg:"match JA3S hash";
ja3s.hash; content:"b26c652e0a402a24b5ca2a660e84f9d5"; sid:100003;)

`ja3s.hash` is a 'sticky buffer'.

`ja3s.hash` can be used as `fast_pattern`.

8.22.4 ja3s.string

Match on JA3S string.

alert tls any any -> any any (msg:"match on JA3S string";
ja3s.string; content:"771,23-35"; sid:100004;)

`ja3s.string` is a 'sticky buffer'.

`ja3s.string` can be used as `fast_pattern`.

8.22.5 ja4.hash

Match on JA4 hash (e.g. `q13d0310h3_55b375c5d22e_cd85d2d88918`).

alert quic any any -> any any (msg:"match JA4 hash";
ja4.hash; content:"q13d0310h3_55b375c5d22e_cd85d2d88918"; sid:100001;)

`ja4.hash` is a 'sticky buffer'.

`ja4.hash` can be used as `fast_pattern`.

8.23 Modbus Keyword

The modbus keyword can be used for matching on various properties of Modbus requests.

There are three ways of using this keyword:

- matching on functions properties with the setting "function";
- matching on directly on data access with the setting "access";
- matching on unit identifier with the setting "unit" only or with the previous setting "function" or "access".

With the setting **function**, you can match on:

- an action based on a function code field and a sub-function code when applicable;
- one of three categories of Modbus functions;
- public functions that are publicly defined (setting "public")
- user-defined functions (setting "user")
- reserved functions that are dedicated to proprietary extensions of Modbus (keyword "reserved")
- one of the two sub-groups of public functions:
  - assigned functions whose definition is already given in the Modbus specification (keyword "assigned");
  - unassigned functions, which are reserved for future use (keyword "unassigned").

Syntax:

```
modbus: function <value>
modbus: function <value>, subfunction <value>
modbus: function [!] <assigned | unassigned | public | user | reserved | all>
```

Sign '!' is negation

Examples:

```
modbus: function 21                # Write File record function
modbus: function 4, subfunction 4  # Force Listen Only Mode (Diagnostics) function
modbus: function assigned          # defined by Modbus Application Protocol␣
↪Specification V1.1b3
modbus: function public            # validated by the Modbus.org community
modbus: function user              # internal use and not supported by the specification
modbus: function reserved          # used by some companies for legacy products and not␣
↪available for public use
modbus: function !reserved         # every function but reserved function
```

With the **access** setting, you can match on:

- a type of data access (read or write);
- one of primary tables access (Discretes Input, Coils, Input Registers and Holding Registers);
- a range of addresses access;
- a written value.

Syntax:

```
modbus: access <read | write>
modbus: access read <discretes | coils | input | holding>
modbus: access read <discretes | coils | input | holding>, address <value>
modbus: access write < coils | holding>
modbus: access write < coils | holding>, address <value>
modbus: access write < coils | holding>, address <value>, value <value>
```

With _<value>_ setting matches on the address or value as it is being accessed or written as follows:

```
address 100      # exactly address 100
address 100<>200 # greater than address 100 and smaller than address 200
address >100     # greater than address 100
address <100     # smaller than address 100
```

Examples:

```
modbus: access read                                  # Read access
modbus: access write                                 # Write access
modbus: access read input                            # Read access to Discretes Input␣
↪table
modbus: access write coils                           # Write access to Coils table
modbus: access read discretes, address <100          # Read access at address smaller␣
↪than 100 of Discretes Input table
modbus: access write holding, address 500, value >200  # Write value greater than 200 at␣
↪address 500 of Holding Registers table
```

With the setting **unit**, you can match on:

- a MODBUS slave address of a remote device connected on the sub-network behind a bridge or a gateway. The destination IP address identifies the bridge itself and the bridge uses the MODBUS unit identifier to forward the request to the right slave device.

Syntax:

```
modbus: unit <value>
modbus: unit <value>, function <value>
modbus: unit <value>, function <value>, subfunction <value>
modbus: unit <value>, function [!] <assigned | unassigned | public | user | reserved |␣
↪all>
modbus: unit <value>, access <read | write>
modbus: unit <value>, access read <discretes | coils | input | holding>
modbus: unit <value>, access read <discretes | coils | input | holding>, address <value>
modbus: unit <value>, access write < coils | holding>
modbus: unit <value>, access write < coils | holding>, address <value>
modbus: unit <value>, access write < coils | holding>, address <value>, value <value>
```

With _<value>_ setting matches on the address or value as it is being accessed or written as follows:

```
unit 10     # exactly unit identifier 10
unit 10<>20 # greater than unit identifier 10 and smaller than unit identifier 20
unit >10    # greater than unit identifier 10
unit <10    # smaller than unit identifier 10
```

Examples:

```
modbus: unit 10                                                    # Unit identifier␣
↪10
modbus: unit 10, function 21                                       # Unit identifier␣
↪10 and write File record function
modbus: unit 10, function 4, subfunction 4                         # Unit identifier␣
↪10 and force Listen Only Mode (Diagnostics) function
modbus: unit 10, function assigned                                 # Unit identifier␣
↪10 and assigned function
modbus: unit 10, function !reserved                                # Unit identifier␣
↪10 and every function but reserved function
modbus: unit 10, access read                                       # Unit identifier␣
↪10 and Read access
modbus: unit 10, access write coils                                # Unit identifier␣
↪10 and Write access to Coils table
modbus: unit >10, access read discretes, address <100              # Greater than␣
↪unit identifier 10 and Read access at address smaller than 100 of Discretes Input table
modbus: unit 10<>20, access write holding, address 500, value >200    # Greater than␣
↪unit identifier 10 and smaller than unit identifier 20 and Write value greater than␣
↪200 at address 500 of Holding Registers table
```

(cf. http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf)

Note: Address of read and write are starting at 1. So if your system is using a start at 0, you need to add 1 the address values.

Note: According to MODBUS Messaging on TCP/IP Implementation Guide V1.0b, it is recommended to keep the TCP connection opened with a remote device and not to open and close it for each MODBUS/TCP transaction. In that case, it is important to set the depth of the stream reassembling as unlimited (stream.reassembly.depth: 0)

Note: According to MODBUS Messaging on TCP/IP Implementation Guide V1.0b, the MODBUS slave device addresses on serial line are assigned from 1 to 247 (decimal). Address 0 is used as broadcast address.

(cf. http://www.modbus.org/docs/Modbus_Messaging_Implementation_Guide_V1_0b.pdf)

Paper and presentation (in french) on Modbus support are available : http://www.ssi.gouv.fr/agence/publication/detection-dintrusion-dans-les-systemes-industriels-suricata-et-le-cas-modbus/

8.24 DCERPC Keywords

Following keywords can be used for matching on fields in headers and payloads of DCERPC packets over UDP, TCP and SMB.

8.24.1 dcerpc.iface

Match on the value of the interface UUID in a DCERPC header. If *any_frag* option is given, the match shall be done on all fragments. If it's not, the match shall only happen on the first fragment.

The format of the keyword:

```
dcerpc.iface:<uuid>;
dcerpc.iface:<uuid>,[>,<,!,=]<iface_version>;
dcerpc.iface:<uuid>,any_frag;
dcerpc.iface:<uuid>,[>,<,!,=]<iface_version>,any_frag;
```

Examples:

```
dcerpc.iface:367abb81-9844-35f1-ad32-98f038001003;
dcerpc.iface:367abb81-9844-35f1-ad32-98f038001003,!10;
dcerpc.iface:367abb81-9844-35f1-ad32-98f038001003,any_frag;
dcerpc.iface:367abb81-9844-35f1-ad32-98f038001003,>1,any_frag;
```

ET Open rule example:

alert tcp any any -> $HOME_NET any (msg:"ET NETBIOS DCERPC WMI Remote Process Execution"; flow:to_server,established; dce_iface:00000143-0000-0000-c000-000000000046; classtype:bad-unknown; sid:2027167; rev:1; metadata:affected_product Windows_XP_Vista_7_8_10_Server_32_64_Bit, attack_target Client_Endpoint, created_at 2019_04_09, deployment Internal, former_category NETBIOS, signature_severity Informational, updated_at 2019_04_09;)

8.24.2 dcerpc.opnum

Match on one or many operation numbers and/or operation number range within the interface in a DCERPC header.

The format of the keyword:

```
dcerpc.opnum:<u16>;
dcerpc.opnum:[>,<,!,=]<u16>;
dcerpc.opnum:<u16>,<u16>,<u16>....;
dcerpc.opnum:<u16>-<u16>;
```

Examples:

```
dcerpc.opnum:15;
dcerpc.opnum:>10;
dcerpc.opnum:12,24,62,61;
```

(continues on next page)

(continued from previous page)

```
dcerpc.opnum:12,18-24,5;
dcerpc.opnum:12-14,12,121,62-78;
```

Note

earlier versions of the documentation incorrectly stated that greater than, less than, etc. notation was supported. This was added in suricata 9, see ticket #8179).

dcerpc.opnum can since suricata 9 use an *unsigned 16-bits integer*.

8.24.3 dcerpc.stub_data

Match on the stub data in a given DCERPC packet. It is a 'sticky buffer'.

Example:

```
dcerpc.stub_data; content:"123456";
```

8.24.4 Additional information

More information on the protocol can be found here:

- DCERPC: https://pubs.opengroup.org/onlinepubs/9629399/chap1.htm

8.25 DHCP keywords

8.25.1 dhcp.leasetime

DHCP lease time (integer).

dhcp.leasetime uses an *unsigned 64-bit integer*.

Syntax:

```
dhcp.leasetime:[op]<number>
```

The time can be matched exactly, or compared using the _op_ setting:

```
dhcp.leasetime:3    # exactly 3
dhcp.leasetime:<3   # smaller than 3
dhcp.leasetime:>=2  # greater or equal than 2
```

alert dhcp any any -> any any (msg:"small DHCP lease time (<3)"; dhcp.leasetime:<3; sid:1; rev:1;)

8.25.2 dhcp.rebinding_time

DHCP rebinding time (integer).

dhcp.rebinding_time uses an *unsigned 64-bit integer*.

Syntax:

```
dhcp.rebinding_time:[op]<number>
```

The time can be matched exactly, or compared using the _op_ setting:

```
dhcp.rebinding_time:3    # exactly 3
dhcp.rebinding_time:<3   # smaller than 3
dhcp.rebinding_time:>=2  # greater or equal than 2
```

alert dhcp any any -> any any (msg:"small DHCP rebinding time (<3)"; dhcp.rebinding_time:<3; sid:1; rev:1;)

8.25.3 dhcp.renewal_time

DHCP renewal time (integer).

dhcp.renewal_time uses an *unsigned 64-bit integer*.

Syntax:

```
dhcp.renewal_time:[op]<number>
```

The time can be matched exactly, or compared using the _op_ setting:

```
dhcp.renewal_time:3    # exactly 3
dhcp.renewal_time:<3   # smaller than 3
dhcp.renewal_time:>=2  # greater or equal than 2
```

alert dhcp any any -> any any (msg:"small DHCP renewal time (<3)"; dhcp.renewal_time:<3; sid:1; rev:1;)

8.26 DNP3 Keywords

The DNP3 keywords can be used to match on fields in decoded DNP3 messages. The keywords are based on Snort's DNP3 keywords and aim to be 100% compatible.

8.26.1 dnp3_func

This keyword will match on the application function code found in DNP3 request and responses. It can be specified as the integer value or the symbolic name of the function code.

dnp3_func uses an *unsigned 8-bits integer*.

Syntax

```
dnp3_func:<value>;
```

Where value is one of:

- An integer value between 0 and 255 inclusive.
- Function code name:
  - confirm
  - read
  - write
  - select
  - operate
  - direct_operate
  - direct_operate_nr

- immed_freeze
- immed_freeze_nr
- freeze_clear
- freeze_clear_nr
- freeze_at_time
- freeze_at_time_nr
- cold_restart
- warm_restart
- initialize_data
- initialize_appl
- start_appl
- stop_appl
- save_config
- enable_unsolicited
- disable_unsolicited
- assign_class
- delay_measure
- record_current_time
- open_file
- close_file
- delete_file
- get_file_info
- authenticate_file
- abort_file
- activate_config
- authenticate_req
- authenticate_err
- response
- unsolicited_response
- authenticate_resp

See `suricata --list-keywords=json | jq .dnp3_func.enum_values`

8.26.2 dnp3_ind

This keyword matches on the DNP3 internal indicator flags in the response application header.

dnp3_ind uses an *unsigned 16-bits integer* with bitmasks.

Syntax

```
dnp3_ind:[*+-=]<flag>{,<flag>...}
```

Where flag is the name of the internal indicator:

- all_stations
- class_1_events
- class_2_events
- class_3_events
- need_time
- local_control
- device_trouble
- device_restart
- no_func_code_support
- object_unknown
- parameter_error
- event_buffer_overflow
- already_executing
- config_corrupt
- reserved_2
- reserved_1

This keyword will match of any of the flags listed are set. To match on multiple flags use prefix +.

Examples

```
dnp3_ind:all_stations;
```

```
dnp3_ind:class_1_events,class_2_events;
```

8.26.3 dnp3_obj

This keyword matches on the DNP3 application data objects.

Syntax

```
dnp3_obj:<group>,<variation>
```

Where <group> and <variation> are integer values between 0 and 255 inclusive.

8.26.4 dnp3_data

This keyword will cause the following content options to match on the re-assembled application buffer. The reassembled application buffer is a DNP3 fragment with CRCs removed (which occur every 16 bytes), and will be the complete fragment, possibly reassembled from multiple DNP3 link layer frames.

Syntax

```
dnp3_data;
```

Example

```
dnp3_data; content:"|c3 06|";
```

8.27 ENIP/CIP Keywords

8.27.1 enip_command

For the ENIP command, we are matching against the command field found in the ENIP encapsulation.

See `suricata --list-keywords=json | jq '."enip_command".enum_values'`

Examples:

```
enip_command:99;
enip_command:list_identity;
```

enip_command uses an *unsigned 16-bits integer*. It can also be specified by text from the enumeration.

8.27.2 cip_service

For the CIP Service, we use a maximum of 3 comma separated values representing the Service, Class and Attribute. These values are described in the CIP specification. CIP Classes are associated with their Service, and CIP Attributes are associated with their Service. If you only need to match up until the Service, then only provide the Service value. If you want to match to the CIP Attribute, then you must provide all 3 values.

Examples:

```
cip_service:75
cip_service:16,246,6
```

(cf. http://read.pudn.com/downloads166/ebook/763211/EIP-CIP-V1-1.0.pdf)

Information on the protocol can be found here: http://literature.rockwellautomation.com/idc/groups/literature/documents/wp/enet-wp001_-en-p.pdf

8.27.3 enip.status

For the ENIP status, we are matching against the status field found in the ENIP encapsulation. It uses a 32-bit unsigned integer as value.

enip.status uses an *unsigned 32-bits integer*. It can also be specified by text from the enumeration.

See `suricata --list-keywords=json | jq '."enip.status".enum_values'`

Examples:

```
enip.status:100;
enip.status:>106;
enip.status:invalid_cmd;
```

8.27.4 enip.protocol_version

Match on the protocol version in different messages. It uses a 16-bit unsigned integer as value.

enip.protocol_version uses an *unsigned 16-bits integer*.

Examples:

```
enip.protocol_version:1;
enip.protocol_version:>1;
```

8.27.5 enip.cip_attribute

Match on the cip attribute in different messages. It uses a 32-bit unsigned integer as value.

This allows to match without needing to match on cip.service.

enip.cip_attribute uses an *unsigned 32-bits integer*.

enip.cip_attribute is also a *multi-integer*.

Examples:

```
enip.cip_attribute:1;
enip.cip_attribute:>1;
```

8.27.6 enip.cip_instance

Match on the cip instance in CIP request path. It uses a 32-bit unsigned integer as value.

enip.cip_instance uses an *unsigned 32-bits integer*.

enip.cip_instance is also a *multi-integer*.

Examples:

```
enip.cip_instance:1;
enip.cip_instance:>1;
```

8.27.7 enip.cip_class

Match on the cip class in CIP request path. It uses a 32-bit unsigned integer as value.

enip.cip_class uses an *unsigned 32-bits integer*.

enip.cip_class is also a *multi-integer*.

This allows to match without needing to match on cip.service.

Examples:

```
enip.cip_class:1;
enip.cip_class:>1;
```

8.27.8 enip.cip_extendedstatus

Match on the cip extended status, if any is present. For multiple service packet, will match on any of the seen statuses. It uses a 16-bit unsigned integer as value.

enip.cip_extendedstatus uses an *unsigned 16-bits integer*.

Examples:

```
enip.cip_extendedstatus:1;
enip.cip_extendedstatus:>1;
```

8.27.9 enip.revision

Match on the revision in identity message. It uses a 16-bit unsigned integer as value.

enip.revision uses an *unsigned 16-bits integer*.

Examples:

```
enip.revision:1;
enip.revision:>1;
```

8.27.10 enip.identity_status

Match on the status in identity message (not in ENIP header). It uses a 16-bit unsigned integer as value.

enip.identity_status uses an *unsigned 16-bits integer*.

Examples:

```
enip.identity_status:1;
enip.identity_status:>1;
```

8.27.11 enip.state

Match on the state in identity message. It uses an 8-bit unsigned integer as value.

enip.state uses an *unsigned 8-bits integer*.

Examples:

```
enip.state:1;
enip.state:>1;
```

8.27.12 enip.serial

Match on the serial in identity message. It uses a 32-bit unsigned integer as value.

enip.serial uses an *unsigned 32-bits integer*.

Examples:

```
enip.serial:1;
enip.serial:>1;
```

8.27.13 enip.product_code

Match on the product code in identity message. It uses a 16-bit unsigned integer as value.

enip.product_code uses an *unsigned 16-bits integer*.

Examples:

```
enip.product_code:1;
enip.product_code:>1;
```

8.27.14 enip.device_type

Match on the device type in identity message. It uses a 16-bit unsigned integer as value.

enip.device_type uses an *unsigned 16-bits integer*.

Examples:

```
enip.device_type:1;
enip.device_type:>1;
```

8.27.15 enip.vendor_id

Match on the vendor id in identity message. It uses a 16-bit unsigned integer as value.

enip.vendor_id uses an *unsigned 16-bits integer*.

Examples:

```
enip.vendor_id:1;
enip.vendor_id:>1;
```

8.27.16 enip.product_name

Match on the product name in identity message.

Examples:

```
enip.product_name; pcre:"/^123[0-9]*/";
enip.product_name; content:"swordfish";
```

`enip.product_name` is a 'sticky buffer' and can be used as `fast_pattern`.

8.27.17 enip.service_name

Match on the service name in list services message.

Examples:

```
enip.service_name; pcre:"/^123[0-9]*/";
enip.service_name; content:"swordfish";
```

`enip.service_name` is a 'sticky buffer' and can be used as `fast_pattern`.

8.27.18 enip.capabilities

Match on the capabilities in list services message. It uses a 16-bit unsigned integer as value.

enip.capabilities uses an *unsigned 16-bits integer*.

Examples:

```
enip.capabilities:1;
enip.capabilities:>1;
```

8.27.19 enip.cip_status

Match on the cip status (one of them in case of multiple service packet). It uses an 8-bit unsigned integer as value.

enip.cip_status uses an *unsigned 8-bits integer*.

Examples:

```
enip.cip_status:1;
enip.cip_status:>1;
```

8.28 FTP/FTP-DATA Keywords

8.28.1 ftpdata_command

Filter ftp-data channel based on command used on the FTP command channel. Currently supported commands are STOR and APPE (put/upload a file), STOU (store a file with a unique name), RETR (get a file), and NLST, LIST and MLSD (list directory contents).

Syntax:

```
ftpdata_command:(stor|appe|stou|retr|nlst|list|mlsd)
```

Signature Example:

alert ftp-data any any -> any any (msg:"FTP store password"; filestore; filename:"password"; ftpdata_command:stor; sid:3; rev:1;)

8.28.2 ftpbounce

Detect FTP bounce attacks.

Syntax:

```
ftpbounce
```

8.28.3 file.name

The `file.name` keyword can be used at the FTP application level.

Signature Example:

alert ftp-data any any -> any any (msg:"FTP file.name usage"; file.name; content:"file.txt"; classtype:bad-unknown; sid:1; rev:1;)

For additional information on the `file.name` keyword, see *File Keywords*.

8.28.4 ftp.command

This keyword matches on the command name from an FTP client request. `ftp.command` is a sticky buffer and can be used as a fast pattern.

Syntax:

```
ftp.command; content: <command>;
```

Signature Example:

alert ftp any any -> any any (ftp.command; content:"PASS"; sid: 1;)

Examples of commands are:

- USER
- PASS
- PORT
- EPRT
- PASV
- RETR

8.28.5 ftp.command_data

This keyword matches on the command data from a FTP client request. `ftp.command_data` is a sticky buffer and can be used as a fast pattern.

Syntax:

```
ftp.command_data; content: <command_data>;
```

Signature Example:

alert ftp any any -> any any (ftp.command_data; content:"anonymous"; sid: 1;)

The `ftp.command_data` matches the data associated with an FTP command. Consider the following FTP command examples:

```
USER anonymous
RETR temp.txt
PORT 192,168,0,13,234,10
```

Example rules for each of the preceding FTP commands and command data.

alert ftp any any -> any any (ftp.command; content: "USER"; ftp.command_data; content:"anonymous"; sid: 1;)

alert ftp any any -> any any (ftp.command_data; content:"anonymous"; sid: 1;)

alert ftp any any -> any any (ftp.command_data; content:"temp.txt"; sid: 2;)

alert ftp any any -> any any (ftp.command_data; content:"192,168,0,13,234,10"; sid: 3;)

8.28.6 ftp.completion_code

This keyword matches on an FTP completion code string. Note that there may be multiple reply strings for an FTP command and hence, multiple completion code values to check. `ftp.completion_code` is a sticky buffer and can be used as a fast pattern. Do not include the response string in the *content* to match upon (see examples).

Syntax:

```
ftp.completion_code; content: <quoted-completion-code>;
```

Signature Example:

alert ftp any any -> any any (ftp.completion_code; content:"226"; sid: 1;)

> **Note**
>
> FTP commands can return multiple reply strings. Specify a single completion code for each `ftp.completion_code` keyword.

This example shows an FTP command (`RETR`) followed by an FTP reply with multiple response strings.

```
RETR temp.txt
150 Opening BINARY mode data connection for temp.txt (1164 bytes).
226 Transfer complete.
```

Because there are multiple completion codes and responses, the rule can match on `ftp.reply` and the `ftp.completion_code`. Suricata cannot guarantee that these come from the *same response*, however.

Signature Examples:

alert ftp any any -> any any (ftp.reply; content:"Opening BINARY mode data connection for temp."; ftp.completion_code; content: "150"; sid: 1;)

alert ftp any any -> any any (ftp.completion_code; content: "226"; sid: 2;)

alert ftp any any -> any any (ftp.reply; content: "Transfer complete."; ftp.completion_code; content: "226"; sid: 3;)

8.28.7 ftp.dynamic_port

This keyword matches on the dynamic port negotiated during an FTP session with the following FTP commands:

- IPv4: `PORT` and `EPRT`
- IPv6: `PASV` and `EPSV`

Syntax:

```
ftp.dynamic_port: <port-spec>;
```

`port-spec` can be one of the following:

- > (greater than)
- < (less than)
- >= (greater than or equal)
- <= (less than or equal)
- `arg1-arg2` (exclusive range)

Signature Example:

alert ftp any any -> any any (ftp.dynamic_port: 59914; sid: 1;)

These rules are will also alert on port `59914`:

alert ftp any any -> any any (ftp.dynamic_port: 59913-59915; sid: 1;)

alert ftp any any -> any any (ftp.dynamic_port: =59914; sid: 1;)

Example rules combining `ftp.dynamic_port` with `ftp.command`

alert ftp any any -> any any (ftp.command; content: "PORT"; ftp.dynamic_port: 59914; sid: 1;)

alert ftp any any -> any any (ftp.command; content: "EPSV"; ftp.dynamic_port: 58612; sid: 1;)

8.28.8 ftp.mode

This keyword matches on whether the FTP session is dynamic or passive. In *active* mode sessions, the server establishes the data channel. In *passive* mode, the client establishes the data channel. Active mode sessions are established in part with the `PORT` (`EPRT` for IPv6) command; passive mode sessions use `PASV` (`EPSV` for IPv6).

Syntax:

```
ftp.mode: active|passive;
```

Signature Example:

alert ftp any any -> any any (ftp.mode: active; sid: 1;)

alert ftp any any -> any any (ftp.mode: passive; sid: 1;)

Example rules combining `ftp.command` with `ftp.mode`

alert ftp any any -> any any (ftp.command; content: "PORT"; ftp.mode: active; sid:1;)

alert ftp any any -> any any (ftp.command; content: "PASV"; ftp.mode: passive; sid:1;)

8.28.9 ftp.reply

This keyword matches on an FTP reply string. Note that there may be multiple reply strings for an FTP command. `ftp.reply` is a sticky buffer and can be used as a fast pattern. Do not include the completion code in the *content* to match upon (see examples).

Syntax:

```
ftp.reply; content: <reply-string>;
```

> **Note**
>
> FTP commands can return multiple reply strings. Specify a single reply for each `ftp.reply` keyword.

This example shows an FTP command (`RETR`) followed by an FTP reply with multiple response strings.

```
RETR temp.txt
150 Opening BINARY mode data connection for temp.txt (1164 bytes).
226 Transfer complete.
```

Signature Example:

alert ftp any any -> any any (ftp.reply; content:"Please specify the password."; sid: 1;)

alert ftp any any -> any any (ftp.reply; content:"Opening BINARY mode data connection for temp."; sid: 1;)

alert ftp any any -> any any (ftp.reply; content:"Transfer complete."; sid: 2;)

8.28.10 ftp.reply_received

This keyword matches on whether an FTP reply string was received. EVE logs with the FTP event_type include a field named `reply_received`. Use this keyword to alert when a reply is (is not) received. `ftp.reply_received` is not a sticky buffer and uses a different syntax to express its value.

> **Note**
>
> Specify the match value without using quotes, e.g., use yes instead of "yes".

Syntax:

```
ftp.reply_received: yes|on|true|1|no|off|false|0;
```

Signature Example:

alert ftp any any -> any any (ftp.reply_received: yes; sid: 1;)

alert ftp any any -> any any (ftp.reply_received: no; sid: 1;)

8.29 Kerberos Keywords

8.29.1 krb5_msg_type

This keyword allows to match the Kerberos messages by its type (integer). It is possible to specify the following values defined in RFC4120:

krb5_msg_type uses *unsigned 32-bit integer*.

See `suricata --list-keywords=json | jq '."krb5_msg_type".enum_values'`

- 10 (AS-REQ)
- 11 (AS-REP)
- 12 (TGS-REQ)
- 13 (TGS-REP)
- 30 (ERROR)

Syntax:

```
krb5_msg_type:(mode) <number or string>
```

Signature examples:

```
alert krb5 any any -> any any (msg:"Kerberos 5 AS-REQ message"; krb5_msg_type:10; sid:3;␣
↪rev:1;)
alert krb5 any any -> any any (msg:"Kerberos 5 AS-REP message"; krb5_msg_type:AS_REP;␣
↪sid:4; rev:1;)
alert krb5 any any -> any any (msg:"Kerberos 5 TGS-REQ message"; krb5_msg_type:12; sid:5;
↪ rev:1;)
alert krb5 any any -> any any (msg:"Kerberos 5 TGS-REP message"; krb5_msg_type:13; sid:6;
↪ rev:1;)
alert krb5 any any -> any any (msg:"Kerberos 5 not ERROR message"; krb5_msg_type:!30;␣
↪sid:7; rev:1;)
```

> **Note**
>
> AP-REQ and AP-REP are not currently supported since those messages are embedded in other application protocols.

8.29.2 krb5_cname

Kerberos client name, provided in the ticket (for AS-REP and TGS-REP messages) or in the error message.

If the client name from the Kerberos message is composed of several parts, the name is compared to each part and the match will succeed if any is identical.

Comparison is case-sensitive.

Syntax:

```
krb5_cname; content:"name";
```

alert krb5 any any -> any any (msg:"Kerberos 5 des server name"; krb5_cname; content:"des"; sid:4; rev:1;)

`krb5_cname` is a 'sticky buffer'.

`krb5_cname` can be used as `fast_pattern`.

`krb5.cname` supports multiple buffer matching, see *Multiple Buffer Matching*.

8.29.3 krb5_sname

Kerberos server name, provided in the ticket (for AS-REP and TGS-REP messages) or in the error message.

If the server name from the Kerberos message is composed of several parts, the name is compared to each part and the match will succeed if any is identical.

Comparison is case-sensitive.

Syntax:

```
krb5_sname; content:"name";
```

alert krb5 any any -> any any (msg:"Kerberos 5 krbtgt server name"; krb5_sname; content:"krbtgt"; sid:5; rev:1;)

`krb5_sname` is a 'sticky buffer'.

`krb5_sname` can be used as `fast_pattern`.

`krb5.sname` supports multiple buffer matching, see *Multiple Buffer Matching*.

8.29.4 krb5_err_code

Kerberos error code (integer). This field is matched in Kerberos error messages only.

For a list of error codes, refer to RFC4120 section 7.5.9.

krb5_err_code uses *unsigned 32-bit integer*.

Syntax:

```
krb5_err_code:<number>
```

alert krb5 any any -> any any (msg:"Kerberos 5 error C_PRINCIPAL_UNKNOWN"; krb5_err_code:6; sid:6; rev:1;)

8.29.5 krb5.weak_encryption (event)

Event raised if the encryption parameters selected by the server are weak or deprecated. For example, using a key size smaller than 128, or using deprecated ciphers like DES.

Syntax:

```
app-layer-event:krb5.weak_encryption
```

alert krb5 any any -> any any (msg:"SURICATA Kerberos 5 weak encryption parameters"; flow:to_client; app-layer-event:krb5.weak_encryption; classtype:protocol-command-decode; sid:2226001; rev:1;)

8.29.6 krb5.malformed_data (event)

Event raised in case of a protocol decoding error.

Syntax:

```
app-layer-event:krb5.malformed_data
```

alert krb5 any any -> any any (msg:"SURICATA Kerberos 5 malformed request data"; flow:to_server; app-layer-event:krb5.malformed_data; classtype:protocol-command-decode; sid:2226000; rev:1;)

8.29.7 krb5.ticket_encryption

Kerberos ticket encryption (enumeration).

For a list of encryption types, refer to RFC3961 section 8.

Syntax:

```
krb5.ticket_encryption: (!)"weak" or (space or comma)-separated list of integer or␣
↪string values for an encryption type
```

alert krb5 any any -> any any (krb5.ticket_encryption: weak; sid:1;)

alert krb5 any any -> any any (krb5.ticket_encryption: 23; sid:2;)

alert krb5 any any -> any any (krb5.ticket_encryption: rc4-hmac,rc4-hmac-exp; sid:3;)

8.30 SMB Keywords

SMB keywords used in both SMB1 and SMB2 protocols.

8.30.1 smb.named_pipe

Match on SMB named pipe in tree connect.

Examples:

```
smb.named_pipe; content:"IPC"; endswith;
smb.named_pipe; content:"strange"; nocase; pcre:"/really$/";
```

`smb.named_pipe` is a 'sticky buffer'.

`smb.named_pipe` can be used as `fast_pattern`.

8.30.2 smb.share

Match on SMB share name in tree connect.

Examples:

```
smb.share; content:"shared"; endswith;
smb.share; content:"strange"; nocase; pcre:"/really$/";
```

`smb.share` is a 'sticky buffer'.

`smb.share` can be used as `fast_pattern`.

8.30.3 smb.ntlmssp_user

Match on SMB ntlmssp user in session setup.

Examples:

```
smb.ntlmssp_user; content:"doe"; endswith;
smb.ntlmssp_user; content:"doe"; nocase; pcre:"/j(ohn|ane).*doe$/";
```

`smb.ntlmssp_user` is a 'sticky buffer'.

`smb.ntlmssp_user` can be used as `fast_pattern`.

8.30.4 smb.ntlmssp_domain

Match on SMB ntlmssp domain in session setup.

Examples:

```
smb.ntlmssp_domain; content:"home"; endswith;
smb.ntlmssp_domain; content:"home"; nocase; pcre:"/home(sweet)*$/";
```

`smb.ntlmssp_domain` is a 'sticky buffer'.

`smb.ntlmssp_domain` can be used as `fast_pattern`.

8.30.5 smb.version

Keyword to match on the SMB version seen in an SMB transaction.

Signature Example:

alert smb $HOME_NET any -> any any (msg:"SMBv1 version rule"; smb.version:1; sid:1;)

alert smb $HOME_NET any -> any any (msg:"SMBv2 version rule"; smb.version:2; sid:2;)

Matching in transition from SMBv1 to SMBv2

In the initial protocol negotiation request, a client supporting SMBv1 and SMBv2 can send an initial SMBv1 request and receive a SMBv2 response from server, indicating that SMBv2 will be used.

This first SMBv2 response made by the server will match as SMBv1, since the entire transaction will be considered a SMBv1 transaction.

Will `smb.version` match SMBv3 traffic?

Yes, it will match SMBv3 messages using *smb.version:2;*, which will match SMBv2 and SMBv3, since they use the same version identifier in the SMB header.

This keyword will use the Protocol ID specified in SMB header to determine the version. Here is a summary of the Protocol ID codes:

- 0xffSMB is SMBv1 header
- 0xfeSMB is SMBv2 normal header (can be sync or async)
- 0xfdSMB is SMBv2 transform header. This is only valid for the SMB 3.x dialect family.

- 0xfcSMB is SMBv2 transform compression header (can be chained or unchained). These ones require the use of the 3.1.1 dialect.

The Protocol ID in the header distinguishes only SMBv1 and SMBv2 since they are completely different protocols with entirely different message formats, types and implementations.

On the other hand, SMBv3 is more like an extension of SMBv2. When using SMBv2 we can select one of the following dialects for the conversation between client and server:

- 2.0.2
- 2.1
- 3.0
- 3.0.2
- 3.1.1

We say we are using SMBv3 when we select a 3.x dialect for the conversation, so you can use SMB 3.0, SMB 3.0.2 or SMB 3.1.1. The higher you choose, the more capabilities you have, but the message syntax and message command number remains the same.

SMB version and dialect are separate components. In the case of SMBv3 for instance, the SMB version will be 2 but the dialect will be 3.x. Dialect specification is not available currently via keyword.

8.30.6 file.name

The `file.name` keyword can be used at the SMB application level.

Signature Example:

alert smb any any -> any any (msg:"SMB file.name usage"; file.name; content:"file.txt"; classtype:bad-unknown; sid:1; rev:1;)

For additional information on the `file.name` keyword, see *File Keywords*.

8.31 SNMP keywords

8.31.1 snmp.version

SNMP protocol version (integer). Expected values are 1, 2 (for version 2c) or 3.

snmp.version uses an, :ref:` unsigned 32-bits integer <rules-integer-keywords>`.

Syntax:

```
snmp.version:[op]<number>
```

The version can be matched exactly, or compared using the _op_ setting:

```
snmp.version:3    # exactly 3
snmp.version:<3   # smaller than 3
snmp.version:>=2  # greater or equal than 2
```

alert snmp any any -> any any (msg:"old SNMP version (<3)"; snmp.version:<3; sid:1; rev:1;)

8.31.2 snmp.community

SNMP community strings are like passwords for SNMP messages in version 1 and 2c. In version 3, the community string is likely to be encrypted. This keyword will not match if the value is not accessible.

The default value for the read-only community string is often "public", and "private" for the read-write community string.

Comparison is case-sensitive.

Syntax:

```
snmp.community; content:"private";
```

alert snmp any any -> any any (msg:"SNMP community private"; snmp.community; content:"private"; sid:2; rev:1;)

`snmp.community` is a 'sticky buffer'.

`snmp.community` can be used as `fast_pattern`.

8.31.3 snmp.usm

SNMP User-based Security Model (USM) is used in version 3. It corresponds to the user name.

Comparison is case-sensitive.

Syntax:

```
snmp.usm; content:"admin";
```

alert snmp any any -> any any (msg:"SNMP usm admin"; snmp.usm; content:"admin"; sid:2; rev:1;)

`snmp.usm` is a 'sticky buffer'.

`snmp.usm` can be used as `fast_pattern`.

8.31.4 snmp.pdu_type

SNMP PDU type (integer).

snmp.pdu_type uses an, :ref:` unsigned 32-bits integer <rules-integer-keywords>`.

You can specify the value as an integer or a string:

- 0: get_request
- 1: get_next_request
- 2: response
- 3: set_request
- 4: trap_v1 (obsolete, was the old Trap-PDU in SNMPv1)
- 5: get_bulk_request
- 6: inform_request
- 7: trap_v2
- 8: report

See `suricata --list-keywords=json | jq '."snmp.pdu_type".enum_values'`

This keyword will not match if the value is not accessible within (for ex, an encrypted SNMP v3 message).

Syntax:

```
snmp.pdu_type:(mode) <number or string>
```

alert snmp any any -> any any (msg:"SNMP response"; snmp.pdu_type:2; sid:3; rev:1;)

8.31.5 snmp.trap_type

SNMP Trap type (integer).

snmp.trap_type uses an, :ref:` unsigned 8-bits integer <rules-integer-keywords>`.

You can specify the value as an integer or a string:

- 0: coldstart
- 1: warmstart
- 2: linkdown
- 3: linkup
- 4: authenticationfailure
- 5: egpneighborloss
- 6: enterprisespecific

See `suricata --list-keywords=json | jq '."snmp.trap_type".enum_values'`

Syntax:

```
snmp.trap_type:(mode) <number or string>
```

Signature example:

```
alert snmp any any -> any 162 (msg:"SNMP trap cold start"; snmp.pdu_type:trap_v1; snmp.
↪trap_type:coldstart; sid:3; rev:1;)
```

8.31.6 snmp.trap_oid

SNMP Trap OID (Object Identifier) is used to uniquely identify the type of trap being sent.

Syntax:

```
snmp.trap_oid; content:"1.3.6.1.4.1.4.1.2.21";
```

alert snmp any any -> any any (msg:"SNMP trap OID example"; snmp.trap_oid; content:"1.3.6.1.4.1.4.1.2.21"; sid:4; rev:1;)

`snmp.trap_oid` is a 'sticky buffer'.

`snmp.trap_oid` can be used as `fast_pattern`.

8.31.7 snmp.trap_address

SNMP Trap Address is the IP address of the agent.

Syntax:

```
snmp.trap_address; content:"192.168.1.2";
```

alert snmp any any -> any any (msg:"SNMP trap address example"; snmp.trap_address; content:"192.168.1.2"; sid:4; rev:1;)

`snmp.trap_address` is a 'sticky buffer'.

`snmp.trap_address` can be used as `fast_pattern`.

8.32 NTP Keywords

8.32.1 ntp.mode

NTP mode. This keyword accepts either an integer or one of the known mode names.

`ntp.mode` uses an *unsigned 8-bit integer*.

See `suricata --list-keywords=json | jq '."ntp.mode".enum_values'`

Syntax:

```
ntp.mode:[op]<number>
ntp.mode:[!]reserved
ntp.mode:[!]symmetric_active
ntp.mode:[!]symmetric_passive
ntp.mode:[!]client
ntp.mode:[!]server
ntp.mode:[!]broadcast
ntp.mode:[!]control
ntp.mode:[!]private
```

Examples:

```
ntp.mode:3
ntp.mode:>3
ntp.mode:client
ntp.mode:!server
```

Signature Example:

alert ntp any any -> any any (msg:"NTP client mode"; ntp.mode:client; sid:1; rev:1;)

8.32.2 ntp.reference_id

`ntp.reference_id` is a sticky buffer exposing the 4-byte NTP reference ID.

Examples:

```
ntp.reference_id; content:"RATE";
ntp.reference_id; content:"|0a 00 00 01|";
```

Signature Example:

alert ntp any any -> any any (msg:"NTP reference ID RATE"; ntp.reference_id; content:"RATE"; sid:4; rev:1;)

8.32.3 ntp.stratum

NTP stratum (integer).

`ntp.stratum` uses an *unsigned 8-bit integer*.

Syntax:

```
ntp.stratum:[op]<number>
```

The stratum can be matched exactly, or compared using the op setting:

```
ntp.stratum:2    # exactly 2
ntp.stratum:<16  # smaller than 16
ntp.stratum:>=1  # greater or equal than 1
```

Signature Example:

alert ntp any any -> any any (msg:"NTP stratum 2"; ntp.stratum:2; sid:2; rev:1;)

8.32.4 ntp.version

NTP protocol version (integer). Expected values are 3 and 4.

`ntp.version` uses an *unsigned 8-bit integer*.

Syntax:

```
ntp.version:[op]<number>
```

The version can be matched exactly, or compared using the op setting:

```
ntp.version:4    # exactly 4
ntp.version:<4   # smaller than 4
ntp.version:>=3  # greater or equal than 3
```

Signature Example:

alert ntp any any -> any any (msg:"NTP version 4"; ntp.version:4; sid:3; rev:1;)

8.33 Base64 keywords

Suricata supports decoding base64 encoded data from buffers and matching on the decoded data.

This is achieved by using two keywords, `base64_decode` and `base64_data`. Both keywords must be used in order to generate an alert.

8.33.1 base64_decode

Decodes base64 data from a buffer and makes it available for the base64_data function.

We recommend using the base64 transform instead -- see *from_base64*.

Syntax:

```
base64_decode:bytes <value>, offset <value>, relative;
```

The `bytes` option specifies how many bytes Suricata should decode and make available for base64_data. This number is limited to 64KiB. The decoding will stop at the end of the buffer.

The `offset` option specifies how many bytes Suricata should skip before decoding. Bytes are skipped relative to the start of the payload buffer if the `relative` is not set.

The `relative` option makes the decoding start relative to the previous content match. Default behavior is to start at the beginning of the buffer. This option makes `offset` skip bytes relative to the previous match.

Note

Regarding `relative` and `base64_decode`:

The content match that you want to decode relative to must be the first match in the stream.

Note

`base64_decode` follows RFC 4648 by default i.e. encounter with any character that is not found in the base64 alphabet leads to rejection of that character and the rest of the string.

See Redmine Bug 5223: https://redmine.openinfosecfoundation.org/issues/5223 and RFC 4648: https://www.rfc-editor.org/rfc/rfc4648#section-3.3

8.33.2 base64_data

base64_data is a `sticky buffer`.

Enables content matching on the data previously decoded by base64_decode.

8.33.3 Example

Here is an example of a rule matching on the base64 encoded string "test" that is found inside the http_uri buffer.

It starts decoding relative to the known string "somestring" with the known offset of 1. This must be the first occurrence of "somestring" in the buffer.

Example:

```
Buffer content:
http_uri = "GET /en/somestring&dGVzdAo=&not_base64"

Rule:
alert http any any -> any any (msg:"Example"; http.uri; content:"somestring"; \
     base64_decode:bytes 8, offset 1, relative; \
     base64_data; content:"test"; sid:10001; rev:1;)

Buffer content:
http_uri = "GET /en/somestring&dGVzdAo=&not_base64"

Rule:
alert http any any -> any any (msg:"Example"; content:"somestring"; http_uri; \
     base64_decode:bytes 8, offset 1, relative; \
     base64_data; content:"test"; sid:10001; rev:1;)
```

> **Note**
>
> `base64_data` cannot be used with `fast_pattern` and will result in a rule load error.

8.34 SIP Keywords

The SIP keywords are implemented as sticky buffers and can be used to match on fields in SIP messages.

As described in RFC3261, common header field names can be represented in a short form. In such cases, the header name is normalized to its regular form to be matched by its corresponding sticky buffer.

Keyword	Direction
sip.method	Request
sip.uri	Request
sip.request_line	Request
sip.stat_code	Response
sip.stat_msg	Response
sip.response_line	Response
sip.protocol	Both
sip.from	Both
sip.to	Both
sip.via	Both
sip.user_agent	Both
sip.content_type	Both
sip.content_length	Both

8.34.1 sip.method

This keyword matches on the method found in a SIP request.

Syntax

```
sip.method; content:<method>;
```

Examples of methods are:

- INVITE
- BYE
- REGISTER
- CANCEL
- ACK
- OPTIONS

Examples

```
sip.method; content:"INVITE";
```

8.34.2 sip.uri

This keyword matches on the uri found in a SIP request.

Syntax

```
sip.uri; content:<uri>;
```

Where <uri> is an uri that follows the SIP URI scheme.

Examples

```
sip.uri; content:"sip:sip.url.org";
```

8.34.3 sip.request_line

This keyword forces the whole SIP request line to be inspected.

Syntax

```
sip.request_line; content:<request_line>;
```

Where <request_line> is a partial or full line.

Examples

```
sip.request_line; content:"REGISTER sip:sip.url.org SIP/2.0"
```

8.34.4 sip.stat_code

This keyword matches on the status code found in a SIP response.

Syntax

```
sip.stat_code; content:<stat_code>
```

Where <status_code> belongs to one of the following groups of codes:

- 1xx - Provisional Responses
- 2xx - Successful Responses
- 3xx - Redirection Responses
- 4xx - Client Failure Responses
- 5xx - Server Failure Responses
- 6xx - Global Failure Responses

Examples

```
sip.stat_code; content:"100";
```

8.34.5 sip.stat_msg

This keyword matches on the status message found in a SIP response.

Syntax

```
sip.stat_msg; content:<stat_msg>
```

Where <stat_msg> is a reason phrase associated to a status code.

Examples

```
sip.stat_msg; content:"Trying";
```

8.34.6 sip.response_line

This keyword forces the whole SIP response line to be inspected.

Syntax

```
sip.response_line; content:<response_line>;
```

Where <response_line> is a partial or full line.

Examples

```
sip.response_line; content:"SIP/2.0 100 OK"
```

8.34.7 sip.protocol

This keyword matches the protocol field from a SIP request or response line.

If the response line is 'SIP/2.0 100 OK', then this buffer will contain 'SIP/2.0'

Syntax

```
sip.protocol; content:<protocol>
```

Where <protocol> is the SIP protocol version.

Example

```
sip.protocol; content:"SIP/2.0"
```

8.34.8 sip.from

This keyword matches on the From field that can be present in SIP headers. It matches both the regular and short forms, though it cannot distinguish between them.

Syntax

```
sip.from; content:<from>
```

Where <from> is the value of the From header.

Example

```
sip.from; content:"user"
```

8.34.9 sip.to

This keyword matches on the To field that can be present in SIP headers. It matches both the regular and short forms, though it cannot distinguish between them.

Syntax

```
sip.to; content:<to>
```

Where <to> is the value of the To header.

Example

```
sip.to; content:"user"
```

8.34.10 sip.via

This keyword matches on the Via field that can be present in SIP headers. It matches both the regular and short forms, though it cannot distinguish between them.

Syntax

```
sip.via; content:<via>
```

Where <via> is the value of the Via header.

Example

```
sip.via; content:"SIP/2.0/UDP"
```

8.34.11 sip.user_agent

This keyword matches on the User-Agent field that can be present in SIP headers.

Syntax

```
sip.user_agent; content:<user_agent>
```

Where <user_agent> is the value of the User-Agent header.

Example

```
sip.user_agent; content:"Asterisk"
```

8.34.12 sip.content_type

This keyword matches on the Content-Type field that can be present in SIP headers. It matches both the regular and short forms, though it cannot distinguish between them.

Syntax

```
sip.content_type; content:<content_type>
```

Where <content_type> is the value of the Content-Type header.

Example

```
sip.content_type; content:"application/sdp"
```

8.34.13 sip.content_length

This keyword matches on the Content-Length field that can be present in SIP headers. It matches both the regular and short forms, though it cannot distinguish between them.

Syntax

```
sip.content_length; content:<content_length>
```

Where <content_length> is the value of the Content-Length header.

Example

```
sip.content_length; content:"200"
```

8.35 SDP Keywords

The SDP keywords are implemented as sticky buffers and can be used to match on fields in SDP messages.

Keyword	Direction
sdp.origin	Both
sdp.session_name	Both
sdp.session_info	Both
sdp.uri	Both
sdp.email	Both
sdp.connection_data	Both
sdp.bandwidth	Both
sdp.time	Both
sdp.repeat_time	Both
sdp.timezone	Both
sdp.encryption_key	Both
sdp.attribute	Both
sdp.media.media	Both
sdp.media.session_info	Both
sdp.media.connection_data	Both
sdp.media.encryption_key	Both

8.35.1 sdp.origin

This keyword matches on the originator found in an SDP request or response.

Syntax

```
sdp.origin; content:<origin>;
```

Where <origin> is an originator that follows the SDP Origin (o=) scheme.

Examples

```
sdp.origin; content:"SIPPS 105015165 105015162 IN IP4 192.168.1.2";
```

8.35.2 sdp.session_name

This keyword matches on the session name found in an SDP request or response.

Syntax

```
sdp.session_name; content:<session_name>;
```

Where <session_name> is a name that follows the SDP Session name (s=) scheme.

Examples

```
sdp.session_name; content:"SIP call";
```

8.35.3 sdp.session_info

This keyword matches on the session information found in an SDP request or response.

Syntax

```
sdp.session_info; content:<session_info>;
```

Where <session_info> is a description that follows the SDP Session information (i=) scheme.

Examples

```
sdp.session_info; content:"Session Description Protocol";
```

8.35.4 sdp.uri

This keyword matches on the URI found in an SDP request or response.

Syntax

```
sdp.uri; content:<uri>;
```

Where <uri> is a URI (u=) that the follows the SDP scheme.

Examples

```
sdp.uri; content:"https://www.sdp.proto"
```

8.35.5 sdp.email

This keyword matches on the email found in an SDP request or response.

Syntax

```
sdp.email; content:<email>
```

Where <email> is an email address (e=) that follows the SDP scheme.

Examples

```
sdp.email; content:"j.doe@example.com (Jane Doe)";
```

8.35.6 sdp.phone_number

This keyword matches on the phone number found in an SDP request or response.

Syntax

```
sdp.phone_number; content:<phone_number>
```

Where <phone_number> is a phone number (p=) that follows the SDP scheme.

Examples

```
sdp.phone_number; content:"+1 617 555-6011 (Jane Doe)";
```

8.35.7 sdp.connection_data

This keyword matches on the connection found in an SDP request or response.

Syntax

```
sdp.connection_data; content:<connection_data>;
```

Where <connection_data> is a connection (c=) that follows the SDP scheme.

Examples

```
sdp.connection_data; content:"IN IP4 192.168.1.2"
```

8.35.8 sdp.bandwidth

This keyword matches on the bandwidths found in an SDP request or response.

Syntax

```
sdp.bandwidth; content:<bandwidth>
```

Where <bandwidth> is a bandwidth (b=) that follows the SDP scheme.

Example

```
sdp.bandwidth; content:"AS:64"
```

8.35.9 sdp.time

This keyword matches on the time found in an SDP request or response.

Syntax

```
sdp.time; content:<time>
```

Where <time> is a time (t=) that follows the SDP scheme.

Example

```
sdp.time; content:"3034423619 3042462419"
```

8.35.10 sdp.repeat_time

This keyword matches on the repeat time found in an SDP request or response.

Syntax

```
sdp.repeat_time; content:<repeat_time>
```

Where <repeat_time> is a repeat time (r=) that follows the SDP scheme.

Example

```
sdp.repeat_time; content:"604800 3600 0 90000"
```

8.35.11 sdp.timezone

This keyword matches on the timezone found in an SDP request or response.

Syntax

```
sdp.timezone; content:<timezone>
```

Where <timezone> is a timezone (z=) that follows the SDP scheme.

Example

```
sdp.timezone; content:"2882844526 -1h 2898848070 0"
```

8.35.12 sdp.encryption_key

This keyword matches on the encryption key found in an SDP request or response.

Syntax

```
sdp.encryption_key; content:<encryption_key>
```

Where <encryption_key> is a key (k=) that follows the SDP scheme.

Example

```
sdp.encryption_key; content:"prompt"
```

8.35.13 sdp.attribute

This keyword matches on the attributes found in an SDP request or response.

Syntax

```
sdp.attribute; content:<attribute>
```

Where <attribute> is an attribute (a=) that follows the SDP scheme.

Example

```
sdp.attribute; content:"sendrecv"
```

8.35.14 sdp.media.media

This keyword matches on the Media subfield of a Media description field found in an SDP request or response.

Syntax

```
sdp.media.media; content:<media>
```

Where <media> is a media (m=) that follows the SDP scheme.

Example

```
sdp.media.media; content:"audio 30000 RTP/AVP 0 8 97 2 3"
```

8.35.15 sdp.media.session_info

This keyword matches on the Session information subfield of a Media description field found in an SDP request or response.

Syntax

```
sdp.media.session_info; content:<session_info>
```

Where <session_info> is a description (i=) that follows the SDP scheme.

Example

```
sdp.media.session_info; content:"Session Description Protocol"
```

8.35.16 sdp.media.connection_data

This keyword matches on the Connection data subfield of a Media description field found in an SDP request or response.

Syntax

```
sdp.media.connection_data; content:<connection_data>
```

Where <connection_data> is a connection (c=) that follows the SDP scheme.

Example

```
sdp.media.connection_data; content:"IN IP4 192.168.1.2"
```

8.35.17 sdp.media.encryption_key

This keyword matches on the Encryption key subfield of a Media description field found in an SDP request or response.

Syntax

```
sdp.media.encryption_key; content:<encryption_key>
```

Where <encryption_key> is a key (k=) that follows the SDP scheme.

Example

```
sdp.media.encryption_key; content:"prompt"
```

8.36 SCTP Keywords

Suricata supports sticky buffers and keywords for matching on SCTP packet headers, chunks, and metadata.

Sticky buffers are expected to be followed by one or more *Payload Keywords*.

8.36.1 sctp.hdr

Sticky buffer to match on the raw SCTP header and all chunks.

Example rule:

alert sctp any any -> any any (msg:"SCTP header match"; sctp.hdr; content:"|01|"; offset:8; depth:1; sid:1; rev:1;)

`sctp.hdr` is a 'sticky buffer'.

`sctp.hdr` can be used as `fast_pattern`.

8.36.2 sctp.chunk_data

Sticky buffer to match on any SCTP DATA chunk user payload.

When a packet contains DATA chunks, the packet payload (`p->payload`) is set to the user data of the first DATA chunk. A bare `content` match (without a sticky buffer) therefore inspects the first DATA chunk's payload. Use `sctp.chunk_data` to inspect all DATA chunks independently.

Example rule:

alert sctp any any -> any any (msg:"SCTP DATA payload match"; sctp.chunk_data; content:"test"; sid:2; rev:1;)

`sctp.chunk_data` is a 'sticky buffer'.

`sctp.chunk_data` can be used as `fast_pattern`.

8.36.3 sctp.vtag

Match on the SCTP verification tag field in the common header.

sctp.vtag uses an *unsigned 32-bit integer*.

Syntax:

```
sctp.vtag:[op]<number>
```

The verification tag can be matched exactly, or compared using the _op_ setting:

```
sctp.vtag:12345         # exactly 12345
sctp.vtag:>0            # greater than 0
sctp.vtag:100-200       # range 100 to 200
```

Example rule:

alert sctp any any -> any any (msg:"SCTP vtag match"; sctp.vtag:0; sid:3; rev:1;)

8.36.4 sctp.chunk_type

Match on the type of any SCTP chunk in the packet.

sctp.chunk_type uses an *unsigned 8-bit integer*.

Syntax:

```
sctp.chunk_type:[!]<value>
sctp.chunk_type:[op]<number>
```

Values can be specified by name or by numeric value. The following named chunk types are supported:

Name	Value
data	0
init	1
init_ack	2
sack	3
heartbeat	4
hb_ack	5
abort	6
shutdown	7
shutdown_ack	8
error	9
cookie_echo	10
cookie_ack	11
ecne	12
cwr	13
shutdown_complete	14
forward_tsn	192

See `suricata --list-keywords=json | jq '."sctp.chunk_type".enum_values'`

Named values are case-insensitive and can be negated with `!`:

```
sctp.chunk_type:init          # INIT chunk
sctp.chunk_type:init_ack      # INIT ACK chunk
sctp.chunk_type:!data         # any chunk that is not DATA
```

Numeric values support comparison operators and ranges:

```
sctp.chunk_type:1             # INIT chunk (type 1)
sctp.chunk_type:0-4           # range 0 to 4
```

Example rules:

alert sctp any any -> any any (msg:"SCTP INIT chunk detected"; sctp.chunk_type:init; sid:4; rev:1;)

alert sctp any any -> any any (msg:"SCTP INIT chunk detected"; sctp.chunk_type:1; sid:5; rev:1;)

8.36.5 sctp.chunk_cnt

Match on the number of SCTP chunks in the packet.

sctp.chunk_cnt uses an *unsigned 16-bit integer*.

Syntax:

```
sctp.chunk_cnt:[op]<number>
```

The chunk count can be matched exactly, or compared using the _op_ setting:

```
sctp.chunk_cnt:1        # exactly 1 chunk
sctp.chunk_cnt:>3       # more than 3 chunks
sctp.chunk_cnt:2-5      # range 2 to 5
```

Example rule:

alert sctp any any -> any any (msg:"SCTP packet with multiple chunks"; sctp.chunk_cnt:>1; sid:5; rev:1;)

8.37 RFB Keywords

The `rfb.name` and `rfb.sectype` keywords can be used for matching on various properties of RFB (Remote Framebuffer, i.e. VNC) handshakes.

8.37.1 rfb.name

Match on the value of the RFB desktop name field.

Examples:

```
rfb.name; content:"Alice's desktop";
rfb.name; pcre:"/.* \(screen [0-9]\)$/";
```

`rfb.name` is a 'sticky buffer'.

`rfb.name` can be used as `fast_pattern`.

8.37.2 rfb.secresult

Match on the value of the RFB security result, e.g. `ok`, `fail`, `toomany` or `unknown`.

rfb.secresult uses an *unsigned 32-bit integer*.

See `suricata --list-keywords=json | jq '."rfb.secresult".enum_values'`

Examples:

```
rfb.secresult: ok;
rfb.secresult: !0;
rfb.secresult: unknown;
```

8.37.3 rfb.sectype

Match on the value of the RFB security type field, e.g. `2` for VNC challenge-response authentication, `0` for no authentication, and `30` for Apple's custom Remote Desktop authentication.

rfb.sectype uses an *unsigned 32-bit integer*.

This keyword takes a numeric argument after a colon and supports additional qualifiers, such as:

- > (greater than)
- < (less than)
- >= (greater than or equal)
- <= (less than or equal)

Examples:

```
rfb.sectype:2;
rfb.sectype:>=3;
```

8.37.4 Additional information

More information on the protocol can be found here: https://tools.ietf.org/html/rfc6143

8.38 MQTT Keywords

Various keywords can be used for matching on fields in fixed and variable headers of MQTT messages as well as payload values.

8.38.1 mqtt.protocol_version

Match on the value of the MQTT protocol version field in the fixed header.

mqtt.protocol_version uses an *unsigned 8-bit integer*.

The format of the keyword:

```
mqtt.protocol_version:<min>-<max>;
mqtt.protocol_version:[<|>]<number>;
mqtt.protocol_version:<value>;
```

Examples:

```
mqtt.protocol_version:5;
```

8.38.2 mqtt.type

Match on the MQTT message type (also: control packet type). Valid values are :

- `CONNECT`
- `CONNACK`
- `PUBLISH`
- `PUBACK`
- `PUBREC`
- `PUBREL`
- `PUBCOMP`
- `SUBSCRIBE`
- `SUBACK`
- `UNSUBSCRIBE`
- `UNSUBACK`
- `PINGREQ`
- `PINGRESP`
- `DISCONNECT`
- `AUTH`
- `UNASSIGNED`

where `UNASSIGNED` refers to message type code 0.

See `suricata --list-keywords=json | jq '."mqtt.type".enum_values'`

mqtt.type uses an *unsigned 8-bits integer*.

mqtt.type is also a *multi-integer*.

Examples:

```
mqtt.type:CONNECT;
mqtt.type:PUBLISH;
mqtt.type:2;
```

8.38.3 mqtt.flags

Match on a combination of MQTT header flags, separated by commas (`,`). Flags may be prefixed by `!` to indicate negation, i.e. a flag prefixed by `!` must *not* be set to match.

mqtt.flags uses an *unsigned 8-bits integer* with bitmasks.

Valid flags are:

- `dup` (duplicate message)
- `retain` (message should be retained on the broker)

Examples:

```
mqtt.flags:dup,!retain;
mqtt.flags:retain;
```

8.38.4 mqtt.qos

Match on the Quality of Service request code in the MQTT fixed header. Valid values are:

- `0` (fire and forget)
- `1` (at least one delivery)
- `2` (exactly one delivery)

Examples:

```
mqtt.qos:0;
mqtt.qos:2;
```

8.38.5 mqtt.reason_code

Match on the numeric value of the reason code that is used in MQTT 5.0 for some message types. Please refer to the specification for the meaning of these values, which are often specific to the message type in question.

mqtt.reason_code uses an *unsigned 8-bits integer*.

mqtt.reason_code is also a *multi-integer*.

Examples:

```
# match on attempts to unsubscribe from a non-subscribed topic
mqtt.type:UNSUBACK; mqtt.reason_code:17;

# match on publications that were accepted but there were no subscribers
mqtt.type:PUBACK; mqtt.reason_code:16;

# match on connection attempts by banned clients
mqtt.CONNACK; mqtt.reason_code:138;

# match on failed connection attempts due to bad credentials
```

(continues on next page)

(continued from previous page)

```
mqtt.CONNACK; mqtt.reason_code:134;

# match on connections terminated by server shutdowns
mqtt.DISCONNECT; mqtt.reason_code:139;
```

This keyword is also available under the alias `mqtt.connack.return_code` for completeness.

8.38.6 mqtt.connack.session_present

Match on the MQTT CONNACK `session_present` flag. Values can be `yes`, `true`, `no` or `false`.

Examples:

```
mqtt.CONNACK; mqtt.connack.session_present:true;
```

8.38.7 mqtt.connect.clientid

Match on the self-assigned client ID in the MQTT CONNECT message.

Examples:

```
mqtt.connect.clientid; pcre:"/^mosq.*/";
mqtt.connect.clientid; content:"myclient";
```

`mqtt.connect.clientid` is a 'sticky buffer' and can be used as `fast_pattern`.

8.38.8 mqtt.connect.flags

Match on a combination of MQTT CONNECT flags, separated by commas (`,`). Flags may be prefixed by `!` to indicate negation, i.e. a flag prefixed by `!` must *not* be set to match.

mqtt.connect.flags uses an *unsigned 8-bits integer* with bitmasks.

Valid flags are:

- `username` (message contains a username)
- `password` (message contains a password)
- `will` (message contains a will definition)
- `will_retain` (will should be retained on broker)
- `clean_session` (start with a clean session)

Examples:

```
mqtt.connect.flags:username,password,!will;
mqtt.connect.flags:username,!password;
mqtt.connect.flags:clean_session;
```

8.38.9 mqtt.connect.password

Match on the password credential in the MQTT CONNECT message.

Examples:

```
mqtt.connect.password; pcre:"/^123[0-9]*/";
mqtt.connect.password; content:"swordfish";
```

`mqtt.connect.password` is a 'sticky buffer' and can be used as `fast_pattern`.

8.38.10 mqtt.connect.protocol_string

Match on the protocol string in the MQTT CONNECT message. In contrast to `mqtt.protocol_version` this is a property that is only really relevant in the initial CONNECT communication and never used again; hence it is organized under `mqtt.connect`.

Examples:

```
mqtt.connect.protocol_string; content:"MQTT";
mqtt.connect.protocol_string; content:"MQIsdp";
```

`mqtt.connect.protocol_string` is a 'sticky buffer' and can be used as `fast_pattern`.

8.38.11 mqtt.connect.username

Match on the username credential in the MQTT CONNECT message.

Examples:

```
mqtt.connect.username; content:"benson";
```

`mqtt.connect.username` is a 'sticky buffer' and can be used as `fast_pattern`.

8.38.12 mqtt.connect.willmessage

Match on the will message in the MQTT CONNECT message, if a will is defined.

Examples:

```
mqtt.connect.willmessage; pcre:"/^fooba[rz]/";
mqtt.connect.willmessage; content:"hunter2";
```

`mqtt.connect.willmessage` is a 'sticky buffer' and can be used as `fast_pattern`.

8.38.13 mqtt.connect.willtopic

Match on the will topic in the MQTT CONNECT message, if a will is defined.

Examples:

```
mqtt.connect.willtopic; pcre:"/^hunter[0-9]/";
```

`mqtt.connect.willtopic` is a 'sticky buffer' and can be used as `fast_pattern`.

8.38.14 mqtt.publish.message

Match on the payload to be published in the MQTT PUBLISH message.

Examples:

```
mqtt.type:PUBLISH; mqtt.publish.message; pcre:"/uid=[0-9]+/";
# match on published JPEG images
mqtt.type:PUBLISH; mqtt.publish.message; content:"|FF D8 FF E0|"; startswith;
```

`mqtt.publish.message` is a 'sticky buffer' and can be used as `fast_pattern`.

8.38.15 mqtt.publish.topic

Match on the topic to be published to in the MQTT PUBLISH message.

Examples:

```
mqtt.publish.topic; content:"mytopic";
```

`mqtt.publish.topic` is a 'sticky buffer' and can be used as `fast_pattern`.

8.38.16 mqtt.subscribe.topic

Match on any of the topics subscribed to in a MQTT SUBSCRIBE message.

Examples:

```
mqtt.subscribe.topic; content:"mytopic";
```

`mqtt.subscribe.topic` is a 'sticky buffer' and can be used as `fast_pattern`.

`mqtt.subscribe.topic` supports multiple buffer matching, see *Multiple Buffer Matching*.

8.38.17 mqtt.unsubscribe.topic

Match on any of the topics unsubscribed from in a MQTT UNSUBSCRIBE message.

Examples:

```
mqtt.unsubscribe.topic; content:"mytopic";
```

`mqtt.unsubscribe.topic` is a 'sticky buffer' and can be used as `fast_pattern`.

`mqtt.unsubscribe.topic` supports multiple buffer matching, see *Multiple Buffer Matching*.

8.38.18 Additional information

More information on the protocol can be found here:

- MQTT 3.1: https://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/mqtt-v3r1.html
- MQTT 3.1.1: https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html
- MQTT 5.0: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html

8.39 IKE Keywords

The keywords

- `ike.init_spi`
- `ike.resp_spi`
- `ike.chosen_sa_attribute`

- ike.exchtype
- ike.vendor
- ike.key_exchange_payload
- ike.key_exchange_payload_length
- ike.nonce_payload
- ike.nonce_payload_length

can be used for matching on various properties of IKE connections.

8.39.1 ike.init_spi, ike.resp_spi

Match on an exact value of the Security Parameter Index (SPI) for the Initiator or Responder.

Examples:

```
ike.init_spi; content:"18fe9b731f9f8034";
ike.resp_spi; content:"a00b8ef0902bb8ec";
```

ike.init_spi and ike.resp_spi are 'sticky buffer'.

ike.init_spi and ike.resp_spi can be used as fast_pattern.

8.39.2 ike.chosen_sa_attribute

Match on an attribute value of the chosen Security Association (SA) by the Responder. Supported for IKEv1 are: alg_enc, alg_hash, alg_auth, alg_dh, alg_prf, sa_group_type, sa_life_type, sa_life_duration, sa_key_length and sa_field_size. IKEv2 supports alg_enc, alg_auth, alg_prf and alg_dh.

If there is more than one chosen SA the event MultipleServerProposal is set. The attributes of the first SA are used for this keyword. You can also use other modes than equality, as in *integer keywords*.

Examples:

```
ike.chosen_sa_attribute:alg_hash=2;
ike.chosen_sa_attribute:sa_key_length=128;
```

8.39.3 ike.exchtype

Match on the value of the Exchange Type.

ike.exchtype uses an *unsigned 8-bit integer*.

This keyword takes a numeric argument after a colon and supports additional qualifiers, such as:

- > (greater than)
- < (less than)
- >= (greater than or equal)
- <= (less than or equal)
- arg1-arg2 (range)

Examples:

```
ike.exchtype:5;
ike.exchtype:>=2;
```

8.39.4 ike.vendor

Match a vendor ID against the list of collected vendor IDs.

Examples:

```
ike.vendor:4a131c81070358455c5728f20e95452f;
```

`ike.vendor` supports multiple buffer matching, see *Multiple Buffer Matching*.

8.39.5 ike.key_exchange_payload

Match against the public key exchange payload (e.g. Diffie-Hellman) of the server or client.

Examples:

```
ike.key_exchange_payload; content:"|6d026d5616c45be05e5b898411e9|"
```

`ike.key_exchange_payload` is a 'sticky buffer'.

`ike.key_exchange_payload` can be used as `fast_pattern`.

8.39.6 ike.key_exchange_payload_length

Match against the length of the public key exchange payload (e.g. Diffie-Hellman) of the server or client.

ike.key_exchange_payload_length uses an *unsigned 32-bit integer*.

This keyword takes a numeric argument after a colon and supports additional qualifiers, such as:

- > (greater than)
- < (less than)
- >= (greater than or equal)
- <= (less than or equal)
- `arg1-arg2` (range)

Examples:

```
ike.key_exchange_payload_length:>132
```

8.39.7 ike.nonce_payload

Match against the nonce of the server or client.

Examples:

```
ike.nonce_payload; content:"|6d026d5616c45be05e5b898411e9|"
```

`ike.nonce_payload` is a 'sticky buffer'.

`ike.nonce_payload` can be used as `fast_pattern`.

8.39.8 ike.nonce_payload_length

Match against the length of the nonce of the server or client.

ike.nonce_payload_length uses an *unsigned 32-bit integer*.

This keyword takes a numeric argument after a colon and supports additional qualifiers, such as:

- > (greater than)
- < (less than)
- >= (greater than or equal)
- <= (less than or equal)
- `arg1-arg2` (range)

Examples:

```
ike.nonce_payload_length:132
ike.nonce_payload_length:>132
```

8.39.9 Additional information

More information on the protocol and the data contained in it can be found here: https://tools.ietf.org/html/rfc2409

8.40 HTTP2 Keywords

HTTP2 frames are grouped into transactions based on the stream identifier it it is not 0. For frames with stream identifier 0, whose effects are global for the connection, a transaction is created for each frame.

8.40.1 Frames

The HTTP2 parser supports the following frames (as defined by Suricata) which are created for each HTTP2 frame (as defined by the HTTP2 RFC) :

- http2.hdr
- http2.data
- http2.pdu

8.40.2 http2.frametype

Match on the frame type present in a transaction.

http2.frametype uses an *unsigned 8-bit integer*.

http2.frametype is also a *multi-integer*.

http2.frametype does not have any corresponding log output.

See `suricata --list-keywords=json | jq '."http2.frametype".enum_values'`

Examples:

```
http2.frametype:GOAWAY;
```

8.40.3 http2.errorcode

Match on the error code in a GOWAY or RST_STREAM frame

http2.errorcode uses an *unsigned 32-bit integer*.

http2.errorcode is also a *multi-integer*.

See `suricata --list-keywords=json | jq '."http2.errorcode".enum_values'`

Examples:

```
http2.errorcode: NO_ERROR;
http2.errorcode: INADEQUATE_SECURITY;
```

8.40.4 http2.priority

Match on the value of the HTTP2 priority field present in a PRIORITY or HEADERS frame.

http2.priority uses an *unsigned 8-bit integer*.

http2.priority is also a *multi-integer*.

This keyword takes a numeric argument after a colon and supports additional qualifiers, such as:

- > (greater than)
- < (less than)
- x-y (range between values x and y)

Examples:

```
http2.priority:2;
http2.priority:>100;
http2.priority:32-64;
```

8.40.5 http2.window

Match on the value of the HTTP2 value field present in a WINDOWUPDATE frame.

http2.window uses an *unsigned 32-bit integer*.

http2.window is also a *multi-integer*.

http2.window does not have any corresponding log output.

This keyword takes a numeric argument after a colon and supports additional qualifiers, such as:

- > (greater than)
- < (less than)
- x-y (range between values x and y)

Examples:

```
http2.window:1;
http2.window:<100000;
```

8.40.6 http2.size_update

Match on the size of the HTTP2 Dynamic Headers Table. More information on the protocol can be found here: https://tools.ietf.org/html/rfc7541#section-6.3

http2.size_update uses an *unsigned 64-bit integer*.

This keyword takes a numeric argument after a colon and supports additional qualifiers, such as:

- > (greater than)
- < (less than)
- x-y (range between values x and y)

Examples:

```
http2.size_update:1234;
http2.size_update:>4096;
```

8.40.7 http2.settings

Match on the name and value of a HTTP2 setting from a SETTINGS frame.

This keyword takes a numeric argument after a colon and supports additional qualifiers, such as:

- > (greater than)
- < (less than)
- `x-y` (range between values x and y)

Examples:

```
http2.settings:SETTINGS_ENABLE_PUSH=0;
http2.settings:SETTINGS_HEADER_TABLE_SIZE>4096;
```

8.40.8 http2.header_name

Match on the name of a HTTP2 header from a HEADER frame (or PUSH_PROMISE or CONTINUATION).

Examples:

```
http2.header_name; content:"agent";
```

`http2.header_name` is a 'sticky buffer'.

`http2.header_name` can be used as `fast_pattern`.

`http2.header_name` supports multiple buffer matching, see *Multiple Buffer Matching*.

8.40.9 Additional information

More information on the protocol can be found here: https://tools.ietf.org/html/rfc7540

8.41 Quic Keywords

Suricata implements initial support for Quic by parsing the Quic version.

Suricata also derives a CYU hash for earlier versions of Quic.

Quic app-layer parsing must be enabled in the Suricata config file (set 'app-layer.protocols.quic.enabled' to 'yes').

8.41.1 quic.cyu.hash

Match on the CYU hash

Examples:

```
alert quic any any -> any any (msg:"QUIC CYU HASH"; \
  quic.cyu.hash; content:"7b3ceb1adc974ad360cfa634e8d0a730"; \
  sid:1;)
```

`quic.cyu.hash` supports multiple buffer matching, see *Multiple Buffer Matching*.

8.41.2 quic.cyu.string

Match on the CYU string

Examples:

```
alert quic any any -> any any (msg:"QUIC CYU STRING"; \
  quic.cyu.string; content:"46,PAD-SNI-VER-CCS-UAID-TCID-PDMD-SMHL-ICSL-NONP-MIDS-SCLS-
↪CSCT-COPT-IRTT-CFCW-SFCW"; \
  sid:2;)
```

`quic.cyu.string` supports multiple buffer matching, see *Multiple Buffer Matching*.

8.41.3 quic.version

Sticky buffer for matching on the Quic header version in long headers.

Examples:

```
alert quic any any -> any any (msg:"QUIC VERSION"; \
  quic.version; content:"Q046"; \
  sid:3;)
```

8.41.4 Additional information

More information on CYU Hash can be found here: https://engineering.salesforce.com/gquic-protocol-analysis-and-fingerprinting-in-zeek-a4178855d75f

More information on the protocol can be found here: https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-17

8.42 NFS Keywords

8.42.1 file.name

The `file.name` keyword can be used at the NFS application level.

Signature Example:

alert nfs any any -> any any (msg:"NFS file.name usage"; file.name; content:"file.txt"; classtype:bad-unknown; sid:1; rev:1;)

For additional information on the `file.name` keyword, see *File Keywords*.

8.42.2 nfs_procedure

This keyword allows to match the nfs procedure by its type (integer).

nfs_procedure uses *unsigned 32-bit integer*.

It is also possible to specify the string values for NFSv3 or NFSv4 procedures. `nfs_procedure: getattr` will match like `nfs_procedure: 1; nfs.version: <4;` or `nfs_procedure: 9; nfs.version: >=4;`

See `suricata --list-keywords=json | jq '."nfs_procedure".enum_values'` to have the list of all the string values for NFS v2, v3 and v4 procedures.

Unlike the other keywords, the usage of range is inclusive.

Syntax:

```
nfs_procedure:(mode) <number or string>
```

8.43 SMTP Keywords

8.43.1 file.name

The `file.name` keyword can be used at the SMTP application level.

Signature Example:

alert smtp any any -> any any (msg:"SMTP file.name usage"; file.name; content:"winmail.dat"; classtype:bad-unknown; sid:1; rev:1;)

For additional information on the `file.name` keyword, see *File Keywords*.

8.43.2 smtp.helo

SMTP helo is the parameter passed to the first HELO command from the client. This keyword matches per transaction, so it can match more than once per flow, even if the helo occured only once at the beginning of the flow.

Syntax:

```
smtp.helo; content:"localhost";
```

alert smtp any any -> any any (msg:"SMTP helo localhost"; smtp.helo; content:"localhost"; sid:2; rev:1;)

`smtp.helo` is a 'sticky buffer'.

`smtp.helo` can be used as `fast_pattern`.

This keyword maps to the eve.json log field `smtp.helo`

8.43.3 smtp.mail_from

SMTP mail from is the parameter passed to the first MAIL FROM command from the client.

Syntax:

```
smtp.mail_from; content:"spam";
```

alert smtp any any -> any any (msg:"SMTP mail from spam"; smtp.mail_from; content:"spam"; sid:2; rev:1;)

`smtp.mail_from` is a 'sticky buffer'.

`smtp.mail_from` can be used as `fast_pattern`.

This keyword maps to the eve.json log field `smtp.mail_from`

8.43.4 smtp.rcpt_to

SMTP rcpt to is the one of the parameters passed to one RCPT TO command from the client.

Syntax:

```
smtp.rcpt_to; content:"sensitive@target";
```

alert smtp any any -> any any (msg:"SMTP rcpt to sensitive"; smtp.rcpt_to; content:"sensitive@target"; sid:2; rev:1;)

`smtp.rcpt_to` is a 'sticky buffer'.

`smtp.rcpt_to` is a 'multi buffer'.

`smtp.rcpt_to` can be used as `fast_pattern`.

This keyword maps to the eve.json log field `smtp.rcpt_to[]`

8.43.5 Frames

The SMTP parser supports the following frames:

- smtp.command_line
- smtp.response_line
- smtp.data
- smtp.stream

smtp.command_line

A single line from the client to the server. Multi-line commands will have a frame per line. Lines part of the SMTP DATA transfer are excluded.

alert smtp any any -> any any (frame:smtp.command_line; content:"MAIL|20|FROM:"; startswith; sid:1;)

smtp.response_line

A single line from the server to the client. Multi-line commands will have a frame per line.

alert smtp any any -> any any (frame:smtp.response_line; content:"354 go ahead"; startswith; sid:1;)

smtp.data

A streaming buffer containing the DATA bytes sent from client to server.

alert smtp any any -> any any (frame:smtp.data; content:"Reply-To:"; startswith; content:"Subject"; distance:0; sid:1;)

smtp.stream

Streaming buffer of the entire TCP data for the SMTP session.

alert smtp any any -> any any (flow:to_client; frame:smtp.stream; content:"250 ok|0d 0a|354 go ahead"; sid:1;)

8.44 WebSocket Keywords

8.44.1 websocket.payload

A sticky buffer on the unmasked payload, limited by suricata.yaml config value `websocket.max-payload-size`.

Examples:

```
websocket.payload; pcre:"/^123[0-9]*/";
websocket.payload content:"swordfish";
```

`websocket.payload` is a 'sticky buffer' and can be used as `fast_pattern`.

8.44.2 websocket.flags

Matches on the websocket flags. It uses a 8-bit unsigned integer as value. Only the four upper bits are used.

The value can also be a list of strings (comma-separated), where each string is the name of a specific bit like *fin* and *comp*, and can be prefixed by *!* for negation.

websocket.flags uses an *unsigned 8-bits integer* with bitmasks.

Examples:

```
websocket.flags:128;
websocket.flags:&0x40=0x40;
websocket.flags:fin,!comp;
```

8.44.3 websocket.mask

Matches on the websocket mask if any. It uses a 32-bit unsigned integer as value (big-endian).

websocket.mask uses an *unsigned 32-bits integer*

Examples:

```
websocket.mask:123456;
websocket.mask:>0;
```

8.44.4 websocket.opcode

Matches on the websocket opcode. It uses a 8-bit unsigned integer as value. Only 16 values are relevant. It can also be specified by text from the enumeration

websocket.opcode uses an *unsigned 8-bits integer*

See `suricata --list-keywords=json | jq '."websocket.opcode".enum_values'`

Examples:

```
websocket.opcode:1;
websocket.opcode:>8;
websocket.opcode:ping;
```

8.45 Generic App Layer Keywords

8.45.1 app-layer-protocol

Match on the detected app-layer protocol.

Syntax:

```
app-layer-protocol:[!]<protocol>[,<qualifier>]...;
app-layer-protocol:[!]<proto1>|<proto2>[|...|<protoN>][,<qualifier>]...;
```

Each `<qualifier>` is either a `<mode>` (at most one, see below) or the `exact` option, in any order.

Examples:

```
app-layer-protocol:ssh;
app-layer-protocol:!tls;
app-layer-protocol:failed;
app-layer-protocol:!http,final;
app-layer-protocol:http,to_server; app-layer-protocol:tls,to_client;
app-layer-protocol:http2,final; app-layer-protocol:http1,original;
app-layer-protocol:unknown;
app-layer-protocol:unknown|tls;
app-layer-protocol:unknown|tls|http;
app-layer-protocol:!tls|http;
app-layer-protocol:tls|http,either;
app-layer-protocol:dns,exact;
app-layer-protocol:tls|dns,either,exact;
```

A special value 'failed' can be used for matching on flows in which protocol detection failed. This can happen if Suricata doesn't know the protocol or when certain 'bail out' conditions happen.

A special value 'unknown' can be used to match on a protocol being not yet known. It can not be negated.

The different modes are * direction : protocol recognized on the direction of the current packet * to_server : protocol recognized in the direction to server * to_client : protocol recognized in the direction to client * either : tries to match protocols found on both directions * final : final protocol chosen by Suricata for parsing * original : original protocol (in case of protocol change)

By default, (if no mode is specified), the mode is `direction`.

Note

when negation is used, like `!http`, it will not match on the "unknown" state in the flow.

Protocol equivalences and the `exact` option

By default a value matches its related protocols as well as itself. For example `http` matches `http1` and `http2`, `dns` also matches `doh2` (DNS over HTTP/2), and `dcerpc` also matches `smb`. This is the long-standing behaviour and keeps existing rules working.

Add the `exact` qualifier to match strictly, with no equivalences: the flow's protocol must equal the configured value exactly. `exact` applies to all values in the list and can be combined with a mode:

```
app-layer-protocol:dns,exact;          # matches dns only, not doh2
app-layer-protocol:tls|dns,either,exact;
```

Because `exact` disables all equivalences, the generic `http` value is not expanded to `http1`/`http2` either. A flow is never classified as the generic `http`, so `app-layer-protocol:http,exact` can never match and is rejected at rule load; use `http1` or `http2` instead.

Here is an example of a rule matching non-http traffic on port 80:

alert tcp any any -> any 80 (msg:"non-HTTP traffic over HTTP standard port"; flow:to_server; app-layer-protocol:!http,final; sid:1;)

Multi-value form

The `app-layer-protocol` keyword also accepts a pipe-separated (`|`) list of protocol values. A rule matches when the flow's resolved application-layer protocol equals **any** value in the list (logical OR).

Syntax:

```
app-layer-protocol:[!]<proto1>|<proto2>[|...|<protoN>](,<mode>);
```

Using `|` for the list keeps the optional trailing `,<mode>` qualifier unambiguous, so the single-value `<protocol>, <mode>` form is unchanged.

Examples:

```
app-layer-protocol:unknown|tls;
app-layer-protocol:unknown|tls|http;
app-layer-protocol:tls|http,either;
app-layer-protocol:!tls|http;
```

The `unknown|<proto>` detection-window idiom

When Suricata has not yet classified a flow's protocol (the "detection window"), the flow's app-layer protocol is `unknown`. Once protocol detection completes, the protocol transitions to its classified value (e.g., `tls`, `http`). Including `unknown` in a multi-value list allows a single rule to cover both the detection window and the confirmed protocol:

```
app-layer-protocol:unknown|tls;
```

This rule matches during the detection window (while the protocol is still `unknown`) **and** after classification (when the protocol is `tls`). If the flow is classified to a protocol not in the list (e.g., `http`), the rule stops matching once the protocol is classified; in firewall mode the flow is then handled by the default policy if no other rule accepts it.

Negated multi-value (NOR semantics)

When the multi-value form is negated with `!`, it implements NOR semantics across the entire list: the rule matches when the resolved application-layer protocol is **known** AND matches **none** of the listed values.

Example:

```
app-layer-protocol:!tls|http;
```

This matches when the flow's protocol is known and is neither `tls` nor `http` (e.g., it matches `dns`, `ssh`, `smtp`, etc.).

> **Note**
>
> Negated multi-value rules do not match during the detection window (when the protocol is still `unknown`). This prevents false positives before protocol classification is complete.

> **Note**
>
> The value `unknown` cannot appear in a negated list. The parser rejects `!unknown` and `!unknown|tls` at rule-load time.

Bail out conditions

Protocol detection gives up in several cases:

- both sides are inspected and no match was found
- side A detection failed, side B has no traffic at all (e.g. FTP data channel)
- side A detection failed, side B has so little data detection is inconclusive

In these last 2 cases the `app-layer-event:applayer_proto_detection_skipped` is set.

8.45.2 app-layer-event

Match on events generated by the App Layer Parsers and the protocol detection engine.

Syntax:

```
app-layer-event:<event name>;
```

Examples:

```
app-layer-event:applayer_mismatch_protocol_both_directions;
app-layer-event:http.gzip_decompression_failed;
```

Protocol Detection

applayer_mismatch_protocol_both_directions

The toserver and toclient directions have different protocols. For example a client talking HTTP to a SSH server.

applayer_wrong_direction_first_data

Some protocol implementations in Suricata have a requirement with regards to the first data direction. The HTTP parser is an example of this.

https://redmine.openinfosecfoundation.org/issues/993

applayer_detect_protocol_only_one_direction

Protocol detection only succeeded in one direction. For FTP and SMTP this is expected.

applayer_proto_detection_skipped

Protocol detection was skipped because of *Bail out conditions*.

8.45.3 app-layer-state

Match on the detected app-layer protocol transaction state. < and > are strict comparisons against the current per-direction transaction progress.

Syntax:

```
app-layer-state:[<>]<state>;
```

Examples:

```
app-layer-state:request_headers;
app-layer-state:>request_body;
```

A comparison that can never hold is rejected at rule load:

- > with a state at or beyond the protocol's completion state;
- < with the protocol's first state (progress 0).

Note

The keyword only resolves on protocols whose parser registers a top-level state table (`get_state_id_by_name`). The `http2` and `doh2` parsers do not, so `app-layer-state` cannot be used on them.

8.46 Generic Decode Layer Keywords

8.46.1 decode-event

Match on events generated by the decode layer. Decode events are generated during the packet decoding phase that indicate structural or invalid values for the Ethernet and layer 2 and layer 3 protocol data.

Syntax:

```
decode-event:<event name>;
```

Examples:

```
decode-event:ipv4.opt_duplicate
decode-event:ethernet.unknown_ethertype
```

Decode Events

ethernet.unknown_ethertype

The ethertype value was not recognized by Suricata. Suricata recognizes the following ethertype values:

```
ETHERNET_TYPE_IP
ETHERNET_TYPE_IPV6
ETHERNET_TYPE_VLAN
ETHERNET_TYPE_8021QINQ
ETHERNET_TYPE_8021AD
ETHERNET_TYPE_8021AH
ETHERNET_TYPE_ARP
ETHERNET_TYPE_MPLS_UNICAST
ETHERNET_TYPE_MPLS_MULTICAST
ETHERNET_TYPE_DCE
ETHERNET_TYPE_ETAG
ETHERNET_TYPE_VNTAG
ETHERNET_TYPE_NSH
ETHERNET_TYPE_PPOE_SESS
ETHERNET_TYPE_PPOE_DISC
```

8.47 Xbits Keyword

Set, unset, toggle and check for bits stored per host, ip_pair or tx.

Syntax:

```
xbits:<set|unset|isset|isnotset|toggle>,<name>,track <ip_src|ip_dst|ip_pair>;
xbits:<set|unset|isset|toggle>,<name>,track <ip_src|ip_dst|ip_pair> \
    [,expire <seconds>];
xbits:<set|unset|isset|toggle>,<name>,track <ip_src|ip_dst|ip_pair> \
    [,expire <seconds>];
xbits:<set|isset>,<name>,track tx;
```

8.47.1 Transaction support (*tx*)

Bits stored per app-layer transaction. What counts as a transaction protocol is implementation dependent.

See *Protocols* for a per protocol explanation.

8.47.2 Notes

- No difference between using `hostbits` and `xbits` with `track ip_<src|dst>`
- If you `set` on a client request and use `track ip_dst`, if you want to match on the server response, you check it (`isset`) with `track ip_src`.
- To not alert, use `noalert;`
- the `toggle` option will flip the value of the xbits.
- See also:
  - https://blog.inliniac.net/2014/12/21/crossing-the-streams-in-suricata/
  - http://www.cipherdyne.org/blog/2013/07/crossing-the-streams-in-ids-signature-languages.html

YAML settings

Bits that are stored per host are stored in the Host table. This means that host table settings affect hostsbits and xbits per host.

Bits that are stored per IP pair are stored in the IPPair table. This means that ippair table settings, especially memcap, affect xbits per ip_pair.

Threading

Due to subtle timing issues between threads the order of sets and checks can be slightly unpredictable.

Unix Socket

Hostbits can be added, removed and listed through the unix socket.

Add:

```
suricatasc -c "add-hostbit <ip> <bit name> <expire in seconds>"
suricatasc -c "add-hostbit 1.2.3.4 blacklist 3600"
```

If a hostbit is added for an existing hostbit, it's expiry timer is updated.

Remove:

```
suricatasc -c "remove-hostbit <ip> <bit name>"
suricatasc -c "remove-hostbit 1.2.3.4 blacklist"
```

List:

```
suricatasc -c "list-hostbit <ip>"
suricatasc -c "list-hostbit 1.2.3.4"
```

This results in:

```
{
    "message":
    {
       "count": 1,
       "hostbits":
            [{
                "expire": 89,
                "name": "blacklist"
            }]
    },
    "return": "OK"
}
```

Examples

Creating a SSH blacklist

Below is an example of rules incoming to a SSH server.

The first 2 rules match on a SSH software version often used in bots. They drop the traffic and create an 'xbit' 'badssh' for the source ip. It expires in an hour:

```
.. container:: example-rule
```

drop ssh any any -> $MYSERVER 22 (msg:"DROP libssh incoming";
flow:to_server,established; ssh.software; content:"libssh"; xbits:set, badssh, track ip_src, expire 3600; sid:4000000005;)

drop ssh any any -> $MYSERVER 22 (msg:"DROP PUTTY incoming";
flow:to_server,established; ssh.software; content:"PUTTY"; xbits:set, badssh, track ip_src, expire 3600; sid:4000000007;)

Then the following rule simply drops any incoming traffic to that server that is on that 'badssh' list:

```
.. container:: example-rule
```

drop ssh any any -> $MYSERVER 22 (msg:"DROP BLACKLISTED";
xbits:isset, badssh, track ip_src; sid:4000000006;)

8.48 Alert Keywords

In addition to the action, alerting behavior can be controlled in the rule body using the `noalert` and `alert` keywords. Additionally, alerting behavior is controlled by *Thresholding Keywords*.

8.48.1 noalert

A rule that specifies `noalert` will not generate an alert when it matches, but rule actions will still be performed.

`noalert` is often used in rules that set a `flowbit` for common patterns.

`noalert` is meant for use with rule actions `alert`, `drop`, `reject` that all explicitly or implicitly include `alert`.

alert http any any -> any any (http.user_agent; content:"Mozilla/5.0"; startswith; endswith; flowbits:set,mozilla-ua; noalert; sid:1;)

This example sets a flowbit "mozilla-ua" on matching, but does not generate an alert due to the presence of `noalert`.

> **Note**
>
> this option is also used as `flowbits:noalert;`, see *Flow Keywords*

8.48.2 alert

A rule that specifies `alert` will generate an alert, even if the rule action doesn't imply alerting.

This keyword can be used to implement an "alert then pass"-logic.

pass http any any -> any any (http.user_agent; content:"Mozilla/5.0"; startswith; endswith; alert; sid:1;)

This example would pass the rest of the HTTP flow with the Mozilla/5.0 user-agent, generating an alert for the "pass" event.

8.49 Thresholding Keywords

Thresholding can be configured per rule and also globally, see *Global-Thresholds*.

Thresholds are tracked in a hash table that is sized according to configuration, see: *Thresholding Settings*.

IMPORTANT NOTES

> **Note**
>
> **For limit, backoff and both**, rule actions `drop` (IPS mode) and `reject` are applied to each packet (not only the one that meets the limit condition). `both` only behaves like that once it has reached its count. `backoff` does not have its rule action applied to the first match, when count > 1.
>
> **For detection_filter, and types both and threshold** neither the rule action nor its alert are applied before (threshold and both) or at (detection_filter) the count limit.
>
> Important Distinction:
>
> - detection_filter: action applied on every match after Cth
> - type threshold: action applied on Cth match, then reset

> **Note**
>
> for any thresholding mechanism, including threshold.config:
>
> Keywords that change state are not subject to threshold limits. They are applied even when the alert is suppressed.
>
> `tag` is skipped when thresholding drops the match completely.
>
> **Post-matching Keywords:**
>
> - flowbits (set / unset)
> - flowint (= / + / -)
> - xbits (set / unset / toggle)

- hostbits (set / unset / toggle)
- bypass
- config
- filestore
- tls_store
- replace
- nfq_set_mark (in NFQ)
- pcre (when it captures into a variable)

Applied at Keyword-evaluating time (even without a match):

- dataset (set / unset)
- lua scripts setting flowvars and flowints

8.49.1 threshold

The `threshold` keyword can be used to control the rule's alert frequency. There are four threshold modes:

1. threshold
2. limit
3. both
4. backoff

Syntax:

```
threshold: type <threshold|limit|both|backoff>, track <by_src|by_dst|by_rule|by_both|by_
↪flow>, count <N>, <seconds <T>|multiplier <M>>
```

Specify `seconds` to control the number of alerts per time period.

type "threshold"

This type sets a minimum threshold for a rule before it generates alerts.

A threshold setting with a `count` value of `C` will generate an alert the `Cth` time the alert matches. If `seconds` is specified, an alert is generated when `count` matches have occurred within `N` seconds.

Syntax:

```
threshold: type threshold, track by_flow, count <C>, seconds <N>;
```

Example:

alert tcp !$HOME_NET any -> $HOME_NET 25 (msg:"ET POLICY Inbound Frequent Emails - Possible Spambot Inbound"; flow:established; content:"mail from|3a|"; nocase; threshold: type threshold, track by_src, count 10, seconds 60; reference:url,doc.emergingthreats.net/2002087; classtype:misc-activity; sid:2002087; rev:10;)

This signature generates an alert if there are 10 or more inbound emails from the same server within one minute.

type "limit"

The `limit` type prevents a flood of alerts by limiting the number of alerts. A limit with a count of `N` won't generate more than `N` alerts.

Limit the number of alerts per time period by specifying `seconds` with `count`.

Syntax:

```
threshold: type limit, track by_dst, count <C>, seconds <N>;
```

Example:

alert http $HOME_NET any -> any any (msg:"ET INFO Internet Explorer 6 in use - Significant Security Risk"; flow:established,to_server; http.user_agent; content:"Mozilla/4.0 (compatible|3b| MSIE 6.0|3b|"; threshold: type limit, track by_src, seconds 180, count 1; classtype:policy-violation; sid:2010706; rev:10; metadata:created_at 2010_07_30, updated_at 2024_03_16;)

In this example, at most 1 alert is generated per host within a period of 3 minutes if "MSIE 6.0" is detected.

type "both"

This type combines `threshold` and `limit` to control when alerts are generated.

Syntax:

```
threshold: type both, track by_flow, count <C>, multiplier <M>;
```

Example:

alert tcp $HOME_NET 5060 -> $EXTERNAL_NET any (msg:"ET VOIP Multiple Unauthorized SIP Responses TCP"; flow:established,from_server; content:"SIP/2.0 401 Unauthorized"; depth:24; threshold: type both, track by_src, count 5, seconds 360; reference:url,doc.emergingthreats.net/2003194; classtype:attempted-dos; sid:2003194; rev:6;)

This rule will generate at most one alert every 6 minutes if there have been 5 or more occurrences of "SIP2.0 401 Unauthorized" responses.

The `type backoff` section describes the `multiplier` keyword.

type "backoff"

This type limits the alert output by using a backoff algorithm between alerts.

> **Note**
>
> `backoff` can only be used with `track by_flow`

Syntax:

```
threshold: type backoff, track by_flow, count <C>, multiplier <M>;
```

- `track`: backoff is only supported for `by_flow`
- `count`: number of alerts before the first match generates an alert.
- `multiplier`: value to multiply `count` with each time the next value is reached

A count of 1 with a multiplier of 10 would generate alerts for matching packets:

```
1, 10, 100, 1000, 10000, 100000, etc.
```

A count of 1 with a multiplier of 2 would generate alerts for matching packets:

```
1, 2, 4, 8, 16, 32, 64, etc.
```

A count of 5 with multiplier 5 would generate alerts for matching packets:

```
5, 25, 125, 625, 3125, 15625, etc
```

In the following example, the `pkt_invalid_ack` would only lead to alerts the 1st, 10th, 100th, etc.

alert tcp any any -> any any (stream-event:pkt_invalid_ack; threshold:type backoff, track by_flow, count 1, multiplier 10; sid:2210045; rev:2;)

track

Option	Tracks By
by_src	source IP
by_dst	destination IP
by_both	pair of src IP and dst IP
by_rule	signature id
by_flow	flow

8.49.2 detection_filter

The `detection_filter` keyword can be used to alert on every match after an initial threshold has been reached. It differs from `threshold` with type `threshold` in that it generates an alert for each rule match after the initial threshold has been reached, where the latter will reset its internal counter and alert each time the threshold has been reached.

Syntax:

```
detection_filter: track <by_src|by_dst|by_rule|by_both|by_flow>, count <N>, seconds <T>[,
↪ unique_on <src_port|dst_port>]
```

`unique_on` (optional) enables distinct counting on a field within the time window:

- `unique_on dst_port`: count distinct destination ports
- `unique_on src_port`: count distinct source ports

> **Note**
>
> `unique_on` requires a transport protocol with ports. Use it only with TCP, UDP or SCTP rules. For non-ported protocols (or `ip` rules), Suricata rejects `unique_on` during rule parsing.

When `unique_on` is specified, alerts start when the number of distinct values exceeds `count` during the `seconds` window, scoped by `track`.

Examples:

alert tcp any any -> $HOME_NET any (msg:"Vertical scan: >=10 distinct dst ports in 60s"; flags:S; ack:0; flow:stateless; detection_filter: track by_dst, count 10, seconds 60, unique_on dst_port; classtype:attempted-recon; sid:100001; rev:1;)

alert tcp any any -> $HOME_NET 22 (msg:"Horizontal scan: >=20 distinct src ports to SSH in 30s"; flow:stateless; detection_filter: track by_dst, count 20, seconds 30, unique_on src_port; classtype:attempted-recon; sid:100002; rev:1;)

Without `unique_on`, the classic behavior applies:

alert http $EXTERNAL_NET any -> $HOME_NET any (msg:"ET WEB_SERVER WebResource.axd access without t (time) parameter - possible ASP padding-oracle exploit"; flow:established,to_server; content:"GET"; http_method; content:"WebResource.axd"; http_uri; nocase; content:!"&t="; http_uri; nocase; content:!"&amp|3b|t="; http_uri; nocase; detection_filter:track by_src,count 15,seconds 2; reference:url,netifera.com/research/; reference:url,www.microsoft.com/technet/security/advisory/2416728.mspx; classtype:web-application-attack; sid:2011807; rev:5;)

This rule will generate alerts after 15 or more matches have occurred within 2 seconds.

8.50 IP Reputation Keyword

IP Reputation can be used in rules through a new rule keyword "iprep".

For more information about IP Reputation see *IP Reputation Config* and *IP Reputation Format*.

8.50.1 iprep

The iprep directive matches on the IP reputation information for a host.

```
iprep:<side to check>,<category>,<operator>,<reputation score>
```

side to check: <any|src|dst|both>

`category`: the category short name

`operator`: <, <=, >, >=, =

`reputation score`: 0-127

alert ip $HOME_NET any -> any any (msg:"IPREP internal host talking to CnC server"; flow:to_server; iprep:dst,CnC,>,30; sid:1; rev:1;)

This rule will alert when a system in `$HOME_NET` acts as a client while communicating with any IP in the CnC category that has a reputation score set to greater than 30.

isset and isnotset

`isset` and `isnotset` can be used to test reputation "membership"

```
iprep:<side to check>,<category>,<isset|issnotset>
```

`side to check`: <any|src|dst|both>

`category`: the category short name

To test whether an IP is part of an iprep set at all, the `isset` can be used. It acts as a `>=,0` statement.

drop ip $HOME_NET any -> any any (iprep:src,known-bad-hosts,isset; sid:1;)

In this example traffic to any IP with a score in `known-bad-hosts` would be blocked.

`isnotset` can be used to test if an IP is not a part of the set.

drop ip $HOME_NET any -> any any (iprep:src,trusted-hosts,isnotset; sid:1;)

In this example traffic for a host w/o a trust score would be blocked.

Compatibility with IP-only

The "iprep" keyword is compatible with "IP-only" rules. This means that a rule like:

alert ip any any -> any any (msg:"IPREP High Value CnC"; iprep:src,CnC,>,100; sid:1; rev:1;)

will only be checked once per flow-direction.

8.51 IP Addresses Match

Matching on IP addresses can be done via the IP tuple parameters or via the iprep keywords (see *IP Reputation Keyword*). Some keywords providing interaction with datasets are also available.

8.51.1 ip.src

The *ip.src* keyword is a sticky buffer to match on source IP address. It matches on the binary representation and is compatible with datasets of types *ip* and *ipv4*.

Example:

```
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"Inbound bad list"; flow:to_server; ip.
↪src; dataset:isset,badips,type ip,load badips.list; sid:1; rev:1;)
```

8.51.2 ip.dst

The *ip.dst* keyword is a sticky buffer to match on destination IP address. It matches on the binary representation and is compatible with the dataset of type *ip* and *ipv4*.

Example:

```
alert tcp $HOME_NET any -> any any (msg:"Outbound bad list"; flow:to_server; ip.dst;␣
↪dataset:isset,badips,type ip,load badips.list; sid:1; rev:1;)
```

8.52 Config Rules

Config rules are rules that when matching, will change the configuration of Suricata for a flow, transaction, packet or other unit.

config dns any any -> any any (dns.query; content:"suricata"; config: logging disable, type tx, scope tx; sid:1;)

This example will detect if a DNS query contains the string *suricata* and if so disable the DNS transaction logging. This means that *eve.json* records, but also Lua output, will not be generated/triggered for this DNS transaction.

config tcp:pre_flow any any <> any 666 (config: tracking disable, type flow, scope packet; sid:1;)

This example skips flow tracking for any packet from or to tcp port 666.

8.52.1 Keyword

The *config* rule keyword provides the setting and the scope of the change.

> **Note**
>
> Currently, the `config` keyword is not supported in Firewall mode, for Threat Detection rules.

Syntax:

```
config:<subsys> <action>, type <type>, scope <scope>;
```

subsys can be set to:

- *logging* setting affects logging.
- *tracking* setting affects tracking.

type can be set to:

- *tx* sub type of the *subsys*. If *subsys* is set to *logging*, setting the *type* to *tx* means transaction logging is affected.
- *flow* sub type of the *subsys*. If *subsys* is set to *flow*, setting the *type* to *flow* means flow tracking is disabled.

scope can be set to:

- *tx* setting affects the matching transaction.
- *packet* setting affects the matching packet.

The *action* in <*subsys*> is currently limited to *disable*.

8.52.2 Action

Config rules can, but don't have to, use the *config* rule action. The *config* rule action won't generate an alert when the rule matches, but the rule actions will still be applied. It is equivalent to *alert ... (noalert; ...)*.

Note

Currently, the `config` action is not supported in Firewall mode, for Threat Detection rules.

8.53 Datasets

Using the `dataset` and `datarep` keyword it is possible to match on large amounts of data against any sticky buffer.

For example, to match against a DNS black list called `dns-bl`:

```
dns.query; dataset:isset,dns-bl;
```

These keywords are aware of transforms. So to look up a DNS query against a MD5 black list:

```
dns.query; to_md5; dataset:isset,dns-bl;
```

8.53.1 Global config (optional)

Datasets can optionally be defined in the main config. Sets can also be declared from the rule syntax.

Example of sets for tracking unique values:

```
datasets:
  ua-seen:
    type: string
    state: ua-seen.lst
  dns-sha256-seen:
    type: sha256
    state: dns-sha256-seen.lst
```

Rules to go with the above:

alert dns any any -> any any (msg:"dns list test"; dns.query; to_sha256; dataset:isset,dns-sha256-seen; sid:123; rev:1;)

alert http any any -> any any (msg: "http user-agent test"; http.user_agent; dataset:set,ua-seen; sid:234; rev:1;)

It is also possible to optionally define global default memcap and hashsize.

Example:

```
datasets:
  defaults:
    memcap: 100mb
    hashsize: 2048
  ua-seen:
    type: string
    load: ua-seen.lst
```

or define memcap and hashsize per dataset.

Example:

```
datasets:
  ua-seen:
    type: string
    load: ua-seen.lst
    memcap: 10mb
    hashsize: 1024
```

Note

The *hashsize* should be close to the amount of entries in the dataset to avoid collisions. If it's set too low, this could result in rather long startup time.

8.53.2 Rule keywords

dataset

Datasets are binary: something is in the set or it's not.

Syntax:

```
dataset:<cmd>,<name>,<options>;

dataset:<set|unset|isset|isnotset>,<name> \
    [, type <string|md5|sha256|ipv4|ip>, save <file name>, load <file name>, state <file␣
↪name>, memcap <size>, hashsize <size>
     , format <csv|json|ndjson>, context_key <output_key>, value_key <json_key>, array_
↪key <json_path>,
     remove_key, match subdomain];
```

type <type>
: the data type: string, md5, sha256, ipv4, ip

load <file name>
: file name for load the data when Suricata starts up

state
sets file name for loading and saving a dataset

save <file name>
advanced option to set the file name for saving the in-memory data when Suricata exits.

memcap <size>
maximum memory limit for the respective dataset

hashsize <size>
allowed size of the hash for the respective dataset

format <type>
the format of the file: csv, json. Defaut to csv. See *dataset with json format* for json and ndjson option

context_key <key>
the key to use for the enrichment of the alert event for json format

value_key <key>
the key to use for the value of the alert for json format

array_key <key>
the key to use for the array of the alert for json format

remove_key
if set, the JSON object pointed by value key will be removed from the alert event

match subdomain
if set to `subdomain`, enables hierarchical domain matching. On lookup, the dataset walks up the domain label hierarchy until a match is found. Only valid with `isset`/`isnotset` commands and `type string`. Best used with the `dotprefix` transform.

Note

'type' is mandatory and needs to be set.

Note

'load' and 'state' or 'save' and 'state' cannot be mixed.

Example rules could look like:

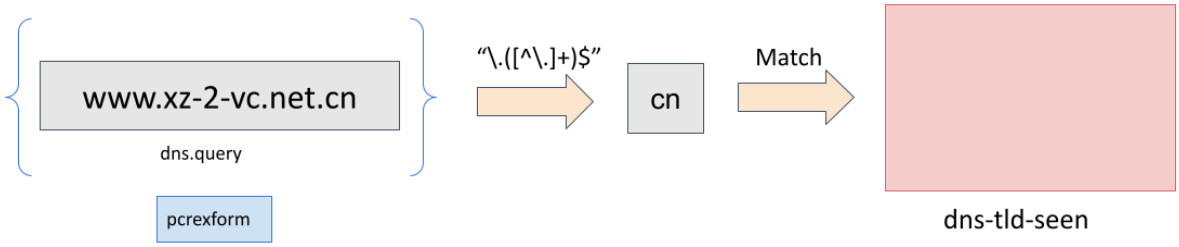
1. Detect unique User-Agents:

alert http any any -> any any (msg:"LOCAL HTTP new UA"; http.user_agent; dataset:set,http-ua-seen, type string, state http-ua-seen.csv; sid:8000001; rev:1;)

2. Detect unique TLDs:

alert dns $HOME_NET any -> any any (msg:"LOCAL DNS unique TLD"; dns.query; pcrexform:"\.([^\.]+)$"; dataset:set,dns-tld-seen, type string, state dns-tld-seen.csv; sid:8000002; rev:1;)

Following image is a pictorial representation of how the `pcrexform` works on domain names to find TLDs in the dataset `dns-tld-seen`:



3. Block domains and all their subdomains using a blocklist dataset:

reject dns any any -> any any (msg:"Blocked domain"; dns.query; dotprefix; dataset:isset,blocked-domains, type string, match subdomain, load blocked-domains.lst; sid:8000003; rev:1;)

The `match subdomain` option walks up the domain hierarchy on each lookup. Combined with `dotprefix`, a query for `mail.evil.com` becomes `.mail.evil.com` and is checked against the dataset as: `.mail.evil.com`, `.evil.com`, `.com`. If `.evil.com` is in the dataset, the rule matches.

The dataset file should contain entries with a leading dot:

```
LmV2aWwuY29tCg==
```

which is the base64 encoding of `.evil.com`.

When using `ndjson` format, use the raw dotted value in the JSON:

```
{"domain": ".evil.com"}
```

Notice how it is not possible to do certain operations alone with datasets (example 2 above), but, it is possible to use a combination of other rule keywords. Keep in mind the cost of additional keywords though e.g. in the second example rule above, negative performance impact can be expected due to `pcrexform`.

datarep

Data Reputation allows matching data against a reputation list.

Syntax:

```
datarep:<name>,<operator>,<value>, \
    [, load <file name>, type <string|md5|sha256|ipv4|ip>, memcap <size>, hashsize <size>
↪];
```

Example rules could look like:

```
alert dns any any -> any any (dns.query; to_md5; datarep:dns_md5, >, 200, load dns_md5.
↪rep, type md5, memcap 100mb, hashsize 2048; sid:1;)
alert dns any any -> any any (dns.query; to_sha256; datarep:dns_sha256, >, 200, load dns_
↪sha256.rep, type sha256; sid:2;)
alert dns any any -> any any (dns.query; datarep:dns_string, >, 200, load dns_string.rep,
↪ type string; sid:3;)
```

In these examples the DNS query string is checked against three different reputation lists. A MD5 list, a SHA256 list, and a raw string (buffer) list. The rules will only match if the data is in the list and the reputation value is higher than 200.

dataset with JSON

Dataset with JSON allows matching data against a set and output data attached to the matching value in the event.

There are two formats supported: `json` and `ndjson`. The difference is that `json` format is a single JSON object, while `ndjson` is handling file with one JSON object per line. The `ndjson` format is useful for large files as the parsing is done line by line.

Syntax:

```
dataset:<cmd>,<name>,<options>;

dataset:<isset|isnotset>,<name> \
    [, type <string|md5|sha256|ipv4|ip>, load <file name>, format <json|ndjson>, memcap
↪<size>, hashsize <size>, context_key <json_key> \
     , value_key <json_key>, array_key <json_path>, match subdomain];
```

Example rules could look like:

```
alert http any any -> any any (msg:"IP match"; ip.dst; dataset:isset,bad_ips, type ip,␣
↪load bad_ips.json, format json, context_key bad_ones, value_key ip; sid:8000001;)
```

In this example, the match will occur if the destination IP is in the set and the alert will have an `alert.content.bad_ones` subobject that will contain the JSON data associated to the value (`bad_ones` coming from `context_key` option).

When format is `json` or `ndjson`, the `value_key` is used to get the value in the line (`ndjson` format) or in the array (`json` format). At least one single element needs to have the `value_key` present in the data file to have a successful load. If `array_key` is present, Suricata will extract the corresponding subobject that has to be a JSON array and search for element to add to the set in this array. This is only valid for `json` format.

If you don't want to have the `value_key` in the alert, you can use the `remove_key` option. This will remove the key from the alert event.

See *Dataset with JSON format* for more information.

8.53.3 Rule Reloads

Sets that are defined in the yaml, or sets that only use *state* or *save*, are considered *dynamic* sets. These are not reloaded during rule reloads.

Sets that are defined in rules using only *load* are considered *static* tests. These are not expected to change during runtime. During rule reloads these are reloaded from disk. This reload is effective when the complete rule reload process is complete.

8.53.4 Unix Socket

dataset-add

Unix Socket command to add data to a set. On success, the addition becomes active instantly.

Syntax:

```
dataset-add <set name> <set type> <data>
```

set name
: Name of an already defined dataset

type
: Data type: string, md5, sha256, ipv4, ip

data
Data to add in serialized form (base64 for string, hex notation for md5/sha256, string representation for ipv4/ip)

Example adding 'google.com' to set 'myset':

```
dataset-add myset string Z29vZ2xlLmNvbQ==
```

dataset-remove

Unix Socket command to remove data from a set. On success, the removal becomes active instantly.

Syntax:

```
dataset-remove <set name> <set type> <data>
```

set name
Name of an already defined dataset

type
Data type: string, md5, sha256, ipv4, ip

data
Data to remove in serialized form (base64 for string, hex notation for md5/sha256, string representation for ipv4/ip)

dataset-clear

Unix Socket command to remove all data from a set. On success, the removal becomes active instantly.

Syntax:

```
dataset-clear <set name> <set type>
```

set name
Name of an already defined dataset

type
Data type: string, md5, sha256, ipv4, ip

dataset-lookup

Unix Socket command to test if data is in a set.

Syntax:

```
dataset-lookup <set name> <set type> <data>
```

set name
Name of an already defined dataset

type
Data type: string, md5, sha256, ipv4, ip

data
Data to test in serialized form (base64 for string, hex notation for md5/sha256, string notation for ipv4/ip)

Example testing if 'google.com' is in the set 'myset':

```
dataset-lookup myset string Z29vZ2xlLmNvbQ==
```

dataset-dump

Unix socket command to trigger a dump of datasets to disk.

Syntax:

```
dataset-dump
```

dataset-add-json

Unix Socket command to add data to a set. On success, the addition becomes active instantly.

Syntax:

```
dataset-add-json <set name> <set type> <data> <json_info>
```

set name
: Name of an already defined dataset

type
: Data type: string, md5, sha256, ipv4, ip

data
: Data to add in serialized form (base64 for string, hex notation for md5/sha256, string representation for ipv4/ip)

Example adding 'google.com' to set 'myset':

```
dataset-add-json myset string Z29vZ2xlLmNvbQ== {"city":"Mountain View"}
```

8.53.5 File formats

Datasets use a simple CSV format where data is per line in the file.

data types

string
: in the file as base64 encoded string

md5
: in the file as hex encoded string

sha256
: in the file as hex encoded string

ipv4
: in the file as string

ip
: in the file as string, it can be IPv6 or IPv4 address (standard notation or IPv4 in IPv6 one)

dataset

Datasets have a simple structure, where there is one piece of data per line in the file.

Syntax:

```
<data>
```

e.g. for ua-seen with type string:

```
TW96aWxsYS80LjAgKGNvbXBhdGlibGU7ICk=
```

which when piped to `base64 -d` reveals its value:

```
Mozilla/4.0 (compatible; )
```

datarep

The datarep format follows the dataset, except that there are 1 more CSV field:

Syntax:

```
<data>,<value>
```

dataset with JSON enrichment

If `format json` is used in the parameters of a dataset keyword, then the loaded file has to contain a valid JSON object.

If `value_key` option is present then the file has to contain a valid JSON object containing an array where the key equal to `value_key` value is present.

For example, if the file `file.json` is like the following example (typical of return of REST API call)

```
{
    "time": "2024-12-21",
    "response": {
        "threats":
            [
                {"host": "toto.com", "origin": "japan"},
                {"host": "grenouille.com", "origin": "french"}
            ]
    }
}
```

then the match to check the list of threats using dataset with JSON can be defined as

```
http.host; dataset:isset,threats,load file.json, context_key threat, value_key host,␣
↪array_key response.threats;
```

If the signature matches, it will result in an alert with the following

```
{
    "alert": {
        "context": {
            "threat": {
                "host": "toto.com",
                "origin": "japan"
            }
        }
    }
}
```

8.53.6 File Locations

Dataset filenames configured in the `suricata.yaml` can exist anywhere on your filesytem.

When a dataset filename is specified in rule, the following *rules* are applied:

- For `load`, the filename is opened relative to the rule file containing the rule. Absolute filenames and parent directory traversals are allowed.
- For `save` and `state` the filename is relative to `$LOCALSTATEDIR/suricata/data`. On many installs this will be `/var/lib/suricata/data`, but run `suricata --build-info` and check the value of `--localstatedir` to verify this location onn your installation.
  - Absolute filenames, or filenames containing parent directory traversal (`..`) are not allowed unless the configuration paramater `datasets.allow-absolute-filenames` is set to `true`.

8.53.7 Security

As datasets potentially allow a rule distributor write access to your system with `save` and `state` dataset rules, the locations allowed are strict by default, however there are two dataset options to tune the security of rules utilizing dataset filenames:

```
datasets:
  rules:
    # Set to true to allow absolute filenames and filenames that use
    # ".." components to reference parent directories in rules that specify
    # their filenames.
    allow-absolute-filenames: false

    # Allow datasets in rules write access for "save" and
    # "state". This is enabled by default, however write access is
    # limited to the data directory.
    allow-write: true
```

By setting `datasets.rules.allow-write` to false, all `save` and `state` rules will fail to load. This option is enabled by default to preserve compatiblity with previous 6.0 Suricata releases, however may change in a future major release.

Pre-Suricata 6.0.13 behavior can be restored by setting `datasets.rules.allow-absolute-filenames` to `true`, however allowing so will allow any rule to overwrite any file on your system that Suricata has write access to.

8.54 Lua Scripting for Detection

There are 2 ways that Lua can be used with detection. These are

- `lua` rule keyword.
- `luaxform` transform.

> **Note**
>
> As of Suricata 8.0, Lua rules are enabled by default and run in a sandboxed environment. See *Lua Sandbox and Available functions*.

8.54.1 Lua Rule Keyword

Syntax:

```
lua:[!]<scriptfilename>;
```

The script filename will be appended to your default rules location.

A Lua rule script has 2 required functions, an `init` function and `match` function, discussed below.

Additionally, the script will run in a limited sandbox by default.

Init function

```
function init (args)
    return {}
end
```

Most Lua rule scripts can simply return an empty table in their init method. To hook into specific protocols states, *Explicit rule hook (states)* may be used. However, some buffers do require explicit initialization:

```
* ja3
* ja3s
* packet
* payload
* stream
```

To request these buffers, use an `init` method like:

```
function init (args)
  return {packet = true}
end
```

Match function

```
local http = require("suricata.http")

function match(args)
    local tx = http.get_tx()
    a = tx:request_line()
    if #a > 0 then
        if a:find("^POST%s+/.*%.php%s+HTTP/1.0$") then
            return 1
        end
    end

    return 0
end
```

The script can return 1 or 0. It should return 1 if the condition(s) it checks for match, 0 if not.

8.54.2 Lua Transform: `luaxform`

More details in *luaxform*.

8.54.3 Lua Sandbox and Available functions

Lua rule scripts are run in a sandbox environment the applies the following restrictions:

- reduced libraries
- only allowed functions available
- instruction count limit
- memory allocation limit

The following table lists the library and functions available:

Package Name	Functions
base	assert, ipairs, next, pairs, print, rawequal, rawlen, select, tonumber, tostring, type, warn, rawget, rawset, error
table	concat, insert, move, pack, remove, sort, unpack
string	byte, char, find, format, gmatch, gsub, len, lower, match, pack, packsize, rep, reverse, sub, unpack, upper
math	abs, acos, asin, atan, atan2, ceil, cos, cosh, deg, exp, floor, fmod, frexp, ldexp, log, log10, max, min, modf, pow, rad, random, randomseed, sin, sinh, sqrt, tan, tanh, tointeger, type, ult
utf8	offset, len, codes, char, codepoint

Of note, the following standard libraries are not available:

- coroutine
- package
- input and output
- operating system facilities
- debug

This behavior can be modified via the `security.lua` section of *Lua*

Note

Suricata 8.0 has moved to Lua 5.4 and now has builtin support for bitwise and utf8 operations.

Script state and budgets

Each rule runs in one Lua state per thread, shared by every packet that thread processes for that rule, unrelated flows included. A value a script stores - a global, a cached table or handle - remains until the per-thread states are rebuilt (a rule reload) or the engine shuts down, and it is the script's own to manage: the engine-owned per-invocation context (flow, transactions, files) is refreshed by the engine on each call and reached only through the lib accessors.

The budgets are per invocation on top of that state. The instruction counter is reset before every script call, so retained state costs no instructions. Retained memory does count against the memory allocation limit: the limit is the setup-time footprint plus `security.lua.max-bytes` (0 disables it), fixed for the state's life and not re-based between calls - so

state a script keeps reduces what later calls may allocate, and a script that caches more than the limit fails its own later invocations with a memory limit error. Keep state small and free what a call no longer needs.

A comprehensive list of existing lua functions - with examples - can be found at *Lua functions* (some of them, however, work only for the lua-output functionality).

8.55 Differences From Snort

This document is intended to highlight the major differences between Suricata and Snort that apply to rules and rule writing.

Where not specified, the statements below apply to Suricata. In general, references to Snort refer to the version 2.9 branch.

8.55.1 Automatic Protocol Detection

- Suricata does automatic protocol detection of the following application layer protocols:
  - dcerpc
  - dnp3
  - dns
  - http
  - imap (detection only by default; no parsing)
  - pop3 (detection only by default; no parsing)
  - ftp
  - modbus (disabled by default; minimalist probe parser; can lead to false positives)
  - smb
  - smb2 (disabled internally inside the engine)
  - smtp
  - ssh
  - tls (SSLv2, SSLv3, TLSv1, TLSv1.1 and TLSv1.2)
- In Suricata, protocol detection is port agnostic (in most cases). In Snort, in order for the `http_inspect` and other preprocessors to be applied to traffic, it has to be over a configured port.
  - Some configurations for app-layer in the Suricata yaml can/do by default specify specific destination ports (e.g. DNS)
  - **You can look on 'any' port without worrying about the performance impact that you would have to be concerned about with Snort.**
- If the traffic is detected as HTTP by Suricata, the `http_*` buffers are populated and can be used, regardless of port(s) specified in the rule.
- You don't have to check for the http protocol (i.e. `alert http ...`) to use the `http_*` buffers although it is recommended.
- If you are trying to detect legitimate (supported) application layer protocol traffic and don't want to look on specific port(s), the rule should be written as `alert <protocol> ...` with `any` in place of the usual protocol port(s). For example, when you want to detect HTTP traffic and don't want to limit detection to a particular port or list of ports, the rules should be written as `alert http ...` with `any` in place of `$HTTP_PORTS`.

- You can also use `app-layer-protocol:<protocol>;` inside the rule instead.

So, instead of this Snort rule:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS ...
```

Do this for Suricata:

```
alert http $HOME_NET -> $EXTERNAL_NET any ...
```

Or:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET any (app-layer-protocol:http; ...
```

8.55.2 `urilen` Keyword

- Ranges given in the `urilen` keyword are inclusive for Snort but not inclusive for Suricata.

  Example: `urilen:2<>10`

  - Snort interprets this as, "the URI length must be **greater than or equal to** 2, and **less than or equal to** 10".
  - Suricata interprets this as "the URI length must be **greater than** 2 and **less than** 10".
  - There is a request to have Suricata behave like Snort in future versions – https://redmine.openinfosecfoundation.org/issues/1416
    - Currently on hold

- By default, with *Suricata*, `urilen` applies to the **normalized** buffer
  - Use `,raw` for raw buffer
  - e.g. `urilen:>20,raw;`
- By default, with *Snort*, `urilen` applies to the **raw** buffer
  - Use `,norm` for normalized buffer
  - e.g. `urilen:>20,norm;`

8.55.3 `http_uri` Buffer

- In Snort, the `http_uri` buffer normalizes '+' characters (0x2B) to spaces (0x20).
  - Suricata can do this as well but you have to explicitly set `query-plusspace-decode: yes` in the `libhtp` section of Suricata's yaml file.
- https://redmine.openinfosecfoundation.org/issues/1035
- https://github.com/inliniac/suricata/pull/620

8.55.4 `http_header` Buffer

- In Snort, the `http_header` buffer includes the CRLF CRLF (0x0D 0x0A 0x0D 0x0A) that separates the end of the last HTTP header from the beginning of the HTTP body. Suricata includes a CRLF after the last header in the `http_header` buffer but not an extra one like Snort does. If you want to match the end of the buffer, use either the `http_raw_header` buffer, a relative `isdataat` (e.g. `isdataat:!1,relative`) or a PCRE (although PCRE will be worse on performance).
- Suricata *will* include CRLF CRLF at the end of the `http_raw_header` buffer like Snort does.

- Snort will include a *leading* CRLF in the `http_header` buffer of *server responses* (but not client requests). Suricata does not have the leading CRLF in the `http_header` buffer of the server response or client request.
- In the `http_header` buffer, Suricata will normalize HTTP header lines such that there is a single space (0x20) after the colon (':') that separates the header name from the header value; this single space replaces zero or more whitespace characters (including tabs) that may be present in the raw HTTP header line immediately after the colon. If the extra whitespace (or lack thereof) is important for matching, use the `http_raw_header` buffer instead of the `http_header` buffer.
- Snort will also normalize superfluous whitespace between the header name and header value like Suricata does but only if there is at least one space character (0x20 only so not 0x90) immediately after the colon. This means that, unlike Suricata, if there is no space (or if there is a tab) immediately after the colon before the header value, the content of the header line will remain unchanged in the `http_header` buffer.
- When there are duplicate HTTP headers (referring to the header name only, not the value), the normalized buffer (`http_header`) will concatenate the values in the order seen (from top to bottom), with a comma and space (", ") between each of them. If this hinders detection, use the `http_raw_header` buffer instead.

  Example request:

```
GET /test.html HTTP/1.1
Content-Length: 44
Accept: */*
Content-Length: 55
```

  The Content-Length header line becomes this in the `http_header` buffer:

```
Content-Length: 44, 55
```

- The HTTP 'Cookie' and 'Set-Cookie' headers are **NOT** included in the `http_header` buffer; instead they are extracted and put into their own buffer – `http_cookie`. See the *http_cookie Buffer* section.
- The HTTP 'Cookie' and 'Set-Cookie' headers **ARE** included in the `http_raw_header` buffer so if you are trying to match on something like particular header ordering involving (or not involving) the HTTP Cookie headers, use the `http_raw_header` buffer.
- If 'enable_cookie' is set for Snort, the HTTP Cookie header names and trailing CRLF (i.e. "Cookie: \r\n" and "Set-Cooke \r\n") are kept in the `http_header` buffer. This is not the case for Suricata which removes the entire "Cookie" or "Set-Cookie" line from the `http_header` buffer.
- Other HTTP headers that have their own buffer (`http_user_agent`, `http_host`) are not removed from the `http_header` buffer like the Cookie headers are.
- When inspecting server responses and `file_data` is used, content matches in `http_*` buffers should come before `file_data` unless you use `pkt_data` to reset the cursor before matching in `http_*` buffers. Snort will not complain if you use `http_*` buffers after `file_data` is set.

8.55.5 `http_cookie` Buffer

- The `http_cookie` buffer will NOT include the header name, colon, or leading whitespace. i.e. it will not include "Cookie: " or "Set-Cookie: ".
- The `http_cookie` buffer does not include a CRLF (0x0D 0x0A) at the end. If you want to match the end of the buffer, use a relative `isdataat` or a PCRE (although PCRE will be worse on performance).
- There is no `http_raw_cookie` buffer in Suricata. Use `http_raw_header` instead.
- You do not have to configure anything special to use the 'http_cookie' buffer in Suricata. This is different from Snort where you have to set `enable_cookie` in the `http_inspect_server` preprocessor config in order to have the `http_cookie` buffer treated separate from the `http_header` buffer.

- If Snort has 'enable_cookie' set and multiple "Cookie" or "Set-Cookie" headers are seen, it will concatenate them together (with no separator between them) in the order seen from top to bottom.
- If a request contains multiple "Cookie" or "Set-Cookie" headers, the values will be concatenated in the Suricata `http_cookie` buffer, in the order seen from top to bottom, with a comma and space (", ") between each of them.

  Example request:

```
GET /test.html HTTP/1.1
Cookie: monster
Accept: */*
Cookie: elmo
```

  Suricata `http_cookie` buffer contents:

```
monster, elmo
```

  Snort `http_cookie` buffer contents:

```
monsterelmo
```

- Corresponding PCRE modifier: `C` (same as Snort)

8.55.6 New HTTP keywords

Suricata supports several HTTP keywords that Snort doesn't have.

Examples are `http_user_agent`, `http_host` and `http_content_type`.

See *HTTP Keywords* for all HTTP keywords.

8.55.7 `byte_extract` Keyword

- Suricata supports `byte_extract` from `http_*` buffers, including `http_header` which does not always work as expected in Snort.
- In Suricata, variables extracted using `byte_extract` must be used in the same buffer, otherwise they will have the value "0" (zero). Snort does allow cross-buffer byte extraction and usage.
- Be sure to always positively and negatively test Suricata rules that use `byte_extract` and `byte_test` to verify that they work as expected.

8.55.8 `byte_jump` Keyword

- Suricata allows a variable name from `byte_extract` or `byte_math` to be specified for the `nbytes` value. The value of `nbytes` must adhere to the same constraints as if it were supplied directly in the rule.

8.55.9 `byte_math` Keyword

- Suricata accepts `dce` as an endian value or as a separate keyword. `endian dce` or `dce` are equivalent.
- Suricata's rule parser rejects rules that repeat keywords in a single rule. E.g., `byte_math: endian big, endian little`.
- Suricata's rule parser accepts `rvalue` values of `0` to the maximum uint32 value. Snort rejects `rvalue` values of `0` and requires values to be between `[1..max-uint32 value]`.
- Suricata will never match if there's a zero divisor. Division by 0 is undefined.

8.55.10 `byte_test` Keyword

- Suricata allows a variable name from `byte_extract` or `byte_math` to be specified for the `nbytes` value. The value of `nbytes` must adhere to the same constraints as though a value was directly supplied by the rule.
- Suricata allows a variable name from `byte_extract` to be specified for the `nbytes` value. The value of `nbytes` must adhere to the same constraints as if it were supplied directly in the rule.

8.55.11 `isdataat` Keyword

- The `rawbytes` keyword is supported in the Suricata syntax but doesn't actually do anything.
- Absolute `isdataat` checks will succeed if the offset used is **less than** the size of the inspection buffer. This is true for Suricata and Snort.
- For *relative* `isdataat` checks, there is a **1 byte difference** in the way Snort and Suricata do the comparisons.
  - Suricata will succeed if the relative offset is **less than or equal to** the size of the inspection buffer. This is different from absolute `isdataat` checks.
  - Snort will succeed if the relative offset is **less than** the size of the inspection buffer, just like absolute `isdataat` checks.
  - Example - to check that there is no data in the inspection buffer after the last content match:
    - Snort: `isdataat:!0,relative;`
    - Suricata: `isdataat:!1,relative;`
- With Snort, the "inspection buffer" used when checking an `isdataat` keyword is generally the packet/segment with some exceptions:
  - With PAF enabled the PDU is examined instead of the packet/segment. When `file_data` or `base64_data` has been set, it is those buffers (unless `rawbytes` is set).
  - With some preprocessors - modbus, gtp, sip, dce2, and dnp3 - the buffer can be particular portions of those protocols (unless `rawbytes` is set).
  - With some preprocessors - rpc_decode, ftp_telnet, smtp, and dnp3 - the buffer can be particular *decoded* portions of those protocols (unless `rawbytes` is set).
- With Suricata, the "inspection buffer" used when checking an absolute `isdataat` keyword is the packet/segment if looking at a packet (e.g. `alert tcp-pkt...`) or the reassembled stream segments.
- In Suricata, a *relative* `isdataat` keyword **will apply to the buffer of the previous content match**. So if the previous content match is a `http_*` buffer, the relative `isdataat` applies to that buffer, starting from the end of the previous content match in that buffer. *Snort does not behave like this!*
- For example, this Suricata rule looks for the string ".exe" at the end of the URI; to do the same thing in the normalized URI buffer in Snort you would have to use a PCRE – `pcre:"/\x2Eexe$/U";`

  alert http $HOME_NET any -> $EXTERNAL_NET any (msg:".EXE File Download Request"; flow:established,to_server; content:"GET"; http_method; content:".exe"; http_uri; isdataat:!1,relative; priority:3; sid:18332111;)
- If you are unclear about behavior in a particular instance, you are encouraged to positively and negatively test your rules that use an `isdataat` keyword.

8.55.12 Relative PCRE

- You can do relative PCRE matches in normalized/special buffers with Suricata. Example:

```
content:".php?sign="; http_uri; pcre:"/^[a-zA-Z0-9]{8}$/UR";
```

- With Snort you can't combine the "relative" PCRE option ('R') with other buffer options like normalized URI ('U') – you get a syntax error.

8.55.13 `tls*` Keywords

In addition to TLS protocol identification, Suricata supports the storing of certificates to disk, verifying the validity dates on certificates, matching against the calculated SHA1 fingerprint of certificates, and matching on certain TLS/SSL certificate fields including the following:

- Negotiated TLS/SSL version.
- Certificate Subject field.
- Certificate Issuer field.
- Certificate SNI Field

For details see *SSL/TLS Keywords*.

8.55.14 `dns_query` Keyword

- Sets the detection pointer to the DNS query.
- Works like `file_data` does ("sticky buffer") but for a DNS request query.
- Use `pkt_data` to reset the detection pointer to the beginning of the packet payload.
- See *DNS Keywords* for details.

8.55.15 IP Reputation and `iprep` Keyword

- Snort has the "reputation" preprocessor that can be used to define whitelist and blacklist files of IPs which are used generate GID 136 alerts as well as block/drop/pass traffic from listed IPs depending on how it is configured.
- Suricata also has the concept of files with IPs in them but provides the ability to assign them:
  - Categories
  - Reputation score
- Suricata rules can leverage these IP lists with the `iprep` keyword that can be configured to match on:
  - Direction
  - Category
  - Value (reputation score)
- *Reputation*
- *IP Reputation Config*
- *IP Reputation Keyword*
- *IP Reputation Format*
- https://blog.inliniac.net/2012/11/21/ip-reputation-in-suricata/

8.55.16 Flowbits

- Suricata fully supports the setting and checking of flowbits (including the same flowbit) on the same packet/stream. Snort does not always allow for this.
- In Suricata, `flowbits:isset` is checked after the fast pattern match but before other `content` matches. In Snort, `flowbits:isset` is checked in the order it appears in the rule, from left to right.
- If there is a chain of flowbits where multiple rules set flowbits and they are dependent on each other, then the order of the rules or the `sid` values can make a difference in the rules being evaluated in the proper order and generating alerts as expected. See bug 1399 - https://redmine.openinfosecfoundation.org/issues/1399.
- *Flow Keywords*

8.55.17 flowbits:noalert;

A common pattern in existing rules is to use `flowbits:noalert;` to make sure a rule doesn't generate an alert if it matches.

Suricata allows using just `noalert;` as well. Both have an identical meaning in Suricata.

8.55.18 Negated Content Match Special Case

- For Snort, a *negated* content match where the starting point for searching is at or beyond the end of the inspection buffer will never return true.
  - For negated matches, you want it to return true if the content is not found.
  - This is believed to be a Snort bug rather than an engine difference but it was reported to Sourcefire and acknowledged many years ago indicating that perhaps it is by design.
  - This is not the case for Suricata which behaves as expected.

    Example HTTP request:

    ```
    POST /test.php HTTP/1.1
    Content-Length: 9

    user=suri
    ```

    This rule snippet will never return true in Snort but will in Suricata:

    ```
    content:!"snort"; offset:10; http_client_body;
    ```

8.55.19 File Extraction

- Suricata has the ability to match on files from FTP, HTTP and SMTP streams and log them to disk.
- Snort has the "file" preprocessor that can do something similar but it is experimental, development of it has been stagnant for years, and it is not something that should be used in a production environment.
- Files can be matched on using a number of keywords including:
  - `filename`
  - `fileext`
  - `filemagic`
  - `filesize`
  - `filemd5`

  - `filesha1`
  - `filesha256`
  - `filesize`
  - See *File Keywords* for a full list.

- The `filestore` keyword tells Suricata to save the file to disk.
- Extracted files are logged to disk with meta data that includes things like timestamp, src/dst IP, protocol, src/dst port, HTTP URI, HTTP Host, HTTP Referer, filename, file magic, md5sum, size, etc.
- There are a number of configuration options and considerations (such as stream reassembly depth and libhtp body-limit) that should be understood if you want fully utilize file extraction in Suricata.
- *File Keywords*
- *File Extraction*
- https://blog.inliniac.net/2011/11/29/file-extraction-in-suricata/
- https://blog.inliniac.net/2014/11/11/smtp-file-extraction-in-suricata/

8.55.20 Lua Scripting

- Suricata has the `lua` keyword which allows for a rule to reference a Lua script that can access the packet, payload, HTTP buffers, etc.
- Provides powerful flexibility and capabilities that Snort does not have.
- More details in: *Lua Scripting for Detection*

8.55.21 Fast Pattern

- Snort's fast pattern matcher is always case insensitive; Suricata's is case sensitive unless 'nocase' is set on the content match used by the fast pattern matcher.
- Snort will truncate fast pattern matches based on the `max-pattern-len` config (default no limit) unless `fast_pattern:only` is used in the rule. Suricata does not do any automatic fast pattern truncation cannot be configured to do so.
- Just like in Snort, in Suricata you can specify a substring of the content string to be use as the fast pattern match. e.g. `fast_pattern:5,20;`
- In Snort, leading NULL bytes (0x00) will be removed from content matches when determining/using the longest content match unless `fast_pattern` is explicitly set. Suricata does not truncate anything, including NULL bytes.
- Snort does not allow for all `http_*` buffers to be used for the fast pattern match (e.g. `http_raw_*`, `http_method`, `http_cookie`, etc.). Suricata lets you use any 'http_*' buffer you want for the fast pattern match, including `http_raw_*' and ``http_cookie` buffers.
- Suricata supports the `fast_pattern:only` syntax but technically it is not really implemented; the `only` is silently ignored when encountered in a rule. It is still recommended that you use `fast_pattern:only` where appropriate in case this gets implemented in the future and/or if the rule will be used by Snort as well.
- With Snort, unless `fast_pattern` is explicitly set, content matches in normalized HTTP Inspect buffers (e.g. http content modifiers such `http_uri`, `http_header`, etc.) take precedence over non-HTTP Inspect content matches, even if they are shorter. Suricata does the same thing and gives a higher 'priority' (precedence) to `http_*` buffers (except for `http_method`, `http_stat_code`, and `http_stat_msg`).
- See *Suricata Fast Pattern Determination Explained* for full details on how Suricata automatically determines which content to use as the fast pattern match.

- When in doubt about what is going to be use as the fast pattern match by Suricata, set `fast_pattern` explicitly in the rule and/or run Suricata with the `--engine-analysis` switch and view the generated file (`rules_fast_pattern.txt`).
- Like Snort, the fast pattern match is checked before `flowbits` in Suricata.
- Using Hyperscan as the MPM matcher (`mpm-algo` setting) for Suricata can greatly improve performance, especially when it comes to fast pattern matching. Hyperscan will also take into account depth and offset when doing fast pattern matching, something the other algorithms and Snort do not do.
- *fast_pattern*

8.55.22 Don't Cross The Streams

Suricata will examine network traffic as individual packets and, in the case of TCP, as part of a (reassembled) stream. However, there are certain rule keywords that only apply to packets only (`dsize`, `flags`, `ttl`) and certain ones that only apply to streams only (`http_*`) and you can't mix packet and stream keywords. Rules that use packet keywords will inspect individual packets only and rules that use stream keywords will inspect streams only. Snort is a little more forgiving when you mix these – for example, in Snort you can use `dsize` (a packet keyword) with `http_*` (stream keywords) and Snort will allow it although, because of `dsize`, it will only apply detection to individual packets (unless PAF is enabled then it will apply it to the PDU).

If `dsize` is in a rule that also looks for a stream-based application layer protocol (e.g. `http`), Suricata will not match on the *first application layer packet* since `dsize` make Suricata evaluate the packet and protocol detection doesn't happen until after the protocol is checked for that packet; *subsequent* packets in that flow should have the application protocol set appropriately and will match rules using `dsize` and a stream-based application layer protocol.

If you need to check sizes on a stream in a rule that uses a stream keyword, or in a rule looking for a stream-based application layer protocol, consider using the `stream_size` keyword and/or `isdataat`.

Suricata also supports these protocol values being used in rules and Snort does not:

- `tcp-pkt` – example:
  - `alert tcp-pkt ...`
  - This tells Suricata to only apply the rule to TCP packets and not the (reassembled) stream.
- `tcp-stream` – example:
  - `alert tcp-stream ...`
  - This tells Suricata to inspect the (reassembled) TCP stream only.

8.55.23 Alerts

- In Snort, the number of alerts generated for a packet/stream can be limited by the `event_queue` configuration.
- Suricata has an internal hard-coded limit of 15 alerts per packet/stream (and this cannot be configured); all rules that match on the traffic being analyzed will fire up to that limit.
- Sometimes Suricata will generate what appears to be two alerts for the same TCP packet. This happens when Suricata evaluates the packet by itself and as part of a (reassembled) stream.

8.55.24 Buffer Reference Chart

Buffer	Snort 2.9.x Support?	Suricata Support?	PCRE flag	Can be used as Fast Pattern?	Suricata Fast Pattern Priority (lower number is higher priority)
content (no modifier)	YES	YES	<none>	YES	3
http_metho	YES	YES	M	Suricata only	3
http_stat_c	YES	YES	S	Suricata only	3
http_stat_n	YES	YES	Y	Suricata only	3
uricontent	YES but deprecated, use http_uri instead	YES but deprecated, use http_uri instead	U	YES	2
http_uri	YES	YES	U	YES	2
http_raw_u	YES	YES	I	Suricata only	2
http_header	YES	YES	H	YES	2
http_raw_h	YES	YES	D	Suricata only	2
http_cookie	YES	YES	C	Suricata only	2
http_raw_c	YES	NO (use http_raw_header instead)	K	NO	n/a
http_host	NO	YES	W	Suricata only	2
http_raw_h	NO	YES	Z	Suricata only	2
http_client_	YES	YES	P	YES	2
http_server	NO	YES	Q	Suricata only	2
http_user_a	NO	YES	V	Suricata only	2
dns_query	NO	YES	n/a*	Suricata only	2
tls_sni	NO	YES	n/a*	Suricata only	2
tls_cert_iss	NO	YES	n/a*	Suricata only	2
tls_cert_sul	NO	YES	n/a*	Suricata only	2
file_data	YES	YES	n/a*	YES	2

* Sticky buffer

8.56 Multiple Buffer Matching

Suricata 7 and newer now supports matching contents in multiple buffers within the same transaction.

For example a single DNS transaction that has two queries in it:

query 1: example.net query 2: something.com

Example rule:

alert dns $HOME_NET any -> $EXTERNAL_NET any (msg:"DNS Multiple Question Example Rule"; dns.query; content:"example"; dns.query; content:".com"; classtype:misc-activity; sid:1; rev:1;)

Within the single DNS query transaction, there are two queries and Suricata will set up two instances of a dns.query buffer.

The first `dns.query` buffer will look for content:"example";

The second `dns.query` buffer will look for content:".com";

The example rule will alert on the example query since all the content matches are satisfied for the rule.

For matching multiple headers in HTTP2 traffic a rule using the new functionality would look like:

alert http2 any any -> any any (msg:"HTTP2 Multiple Header Buffer Example"; flow:established,to_server; http.request_header; content:"method|3a 20|GET"; http.request_header; content:"authority|3a 20|example.com"; classtype:misc-activity; sid:1; rev:1;)

With HTTP2 there are multiple headers seen in the same flow record. We now have a way to write a rule in a more efficient way using the multiple buffer capability.

Note Existing behavior when using sticky buffers still applies:

Example rule:

alert dns $HOME_NET any -> $EXTERNAL_NET any (msg:"DNS Query Sticky Buffer Classic Example Rule"; dns.query; content:"example"; content:".net"; classtype:misc-activity; sid:1; rev:1;)

The above rule will alert on a single dns query containing "example.net" or "example.domain.net" since the rule content matches are within a single `dns.query` buffer and all content match requirements of the rule are met.

Note: This is new behavior. In versions of Suricata prior to version 7 multiple statements of the same sticky buffer did not make a second instance of the buffer. For example:

dns.query; content:"example"; dns.query; content:".com";

would be equivalent to:

dns.query; content:"example"; content:".com";

Using our example from above, the first query is for example.net which matches content:"example"; but does not match content:".com";

The second query is for something.com which would match on the content:".com"; but not the content:"example";

So with the Suricata behavior prior to Suricata 7, the signature would not fire in this case since both content conditions will not be met.

Multiple buffer matching is currently enabled for use with the following keywords:

- `dns.additionals.rrname`
- `dns.answer.name`
- `dns.answers.rrname`
- `dns.authorities.rrname`
- `dns.queries.rrname`
- `dns.query`
- `dns.query.name`
- `email.received`
- `email.url`
- `file.data`
- `file.magic`
- `file.name`
- `ftp.completion_code`
- `ftp.reply`
- `http.request_header`
- `http.response_header`

- http2.header_name
- ike.vendor
- krb5.cname
- krb5.sname
- ldap.request.attribute_type
- ldap.responses.attribute_type
- ldap.responses.dn
- ldap.responses.message
- mdns.additionals.rrname
- mdns.answers.rrname
- mdns.authorities.rrname
- mdns.queries.rrname
- mqtt.subscribe.topic
- mqtt.unsubscribe.topic
- quic.cyu.hash
- quic.cyu.string
- sdp.attribute
- sdp.bandwidth
- sdp.media.connection_data
- sdp.media.encryption_key
- sdp.media.media
- sdp.media.media_info
- sdp.repeat_time
- sdp.time
- sip.content_length
- sip.content_type
- sip.from
- sip.to
- sip.user_agent
- sip.via
- smtp.rcpt_to
- tls.alpn
- tls.certs
- tls.subjectaltname

These can be found by command `suricata --list-keywords=csv | grep "multi buffer" | cut -d';' -f1`

8.57 Tag

The *tag* keyword allows tagging of the current and future packets.

Tagged packets can be logged in *EVE* and conditional PCAP logging.

Tagging is limited to a scope: *host* or *session* (flow). When using *host* a direction can be specified: *src* or *dst*. Tagging will then occur based on the *src* or *dst* IP address of the packet generating the alert.

Tagging is further controlled by count: *packets*, *bytes* or *seconds*. If the count is ommited built-in defaults will be used:

- for *session*: 256 packets
- for *host*: 256 packets for the destination IP of the packet triggering the alert

The *tag* keyword can appear multiple times in a rule.

8.57.1 Syntax

```
tag:<scope>[,<count>, <metric>[,<direction>]];
```

Values for *scope*: *session* and *host* Values for *metric*: *packets*, *bytes*, *seconds* Values for *direction*: *src* and *dst*

> **Note**
>
> "direction" can only be specified if scope is "host" and both "count" and "metric" are also specified.

8.57.2 Examples

Keyword:

```
tag:session;                # tags next 256 packets in the flow
tag:host;                   # tags next 256 packets for the dst ip of the alert
tag:host,100,packets,src;   # tags next 100 packets for src ip of the alert
tag:host,3600,seconds,dst;  # tags packets for dst host for the next hour
```

Full rule examples:

alert dns any any -> any any (dns.query; content:"evil"; tag:host,60,seconds,src; sid:1;)

alert http any any -> any any (http.method; content:"POST"; tag:session; sid:1;)

8.57.3 How to Use Tags

EVE

Tags can be set to generate *EVE tag* records:

```
outputs:
  - eve-log:
      enabled: yes
      filename: eve.json
      types:
        - alert:
            tagged-packets: true
```

The tagged packets will then be logged with *event_type*: *packet*:

```
{
  "timestamp": "2020-06-03T10:29:17.850417+0000",
  "flow_id": 1576832511820424,
  "event_type": "packet",
  "src_ip": "192.168.0.27",
  "src_port": 54634,
  "dest_ip": "192.168.0.103",
  "dest_port": 22,
  "proto": "TCP",
  "pkt_src": "wire/pcap",
  "packet":
↪"CAAn6mWJAPSNvfrHCABFAAAogkVAAIAG9rfAqAAbwKgAZ9VqABZvnJXH5Zf6aFAQEAljEwAAAAAAAAAA",
  "packet_info": {
    "linktype": 1
  }
}
```

EVE: *Eve JSON Output*

Conditional PCAP Logging

Using the conditional PCAP logging option the tag keyword can control which packets are logged by the PCAP logging.

```
outputs:
  - pcap-log:
      enabled: yes
      filename: log.pcap
      limit: 1000mb
      max-files: 2000
      compression: none
      mode: normal
      use-stream-depth: no #If set to "yes" packets seen after reaching stream␣
↪inspection depth are ignored. "no" logs all packets
      honor-pass-rules: no # If set to "yes", flows in which a pass rule matched will␣
↪stop being logged.
      # Use "all" to log all packets or use "alerts" to log only alerted packets and␣
↪flows or "tag"
      # to log only flow tagged via the "tag" keyword
      conditional: tag
```

PCAP Logging: *PCAP log*

8.57.4 Tracking by Host/Flow

When the tags are using the *session* scope, the tag is added to the *Flow* structure. If a packet has no flow, no tagging will happen. No errors/warnings are generated for this.

See *Flow Settings* for managing flow limits and resources.

When tags are using the *host* scope, the tag is stored with a *Host* object in the host table. The Host table size will affect effectiveness of per host tags.

See *Host Settings* for managing host table size.

8.58 VLAN Keywords

8.58.1 vlan.id

Suricata has a `vlan.id` keyword that can be used in signatures to identify and filter network packets based on Virtual Local Area Network IDs. By default, it matches all layers if a packet contains multiple VLAN layers. However, if a specific layer is defined, it will only match that layer.

Syntax:

```
vlan.id: [op]id[,layer];
```

The id can be matched exactly, or compared using the `op` setting:

```
vlan.id:300    # exactly 300
vlan.id:<300,0   # smaller than 300 at layer 0
vlan.id:>=200,1  # greater or equal than 200 at layer 1
```

vlan.id uses *unsigned 16-bit integer*.

The valid range for VLAN id values is `0 - 4095`.

This keyword also supports `all` and `any` as arguments for `layer`. `all` matches only if all VLAN layers match and `any` matches with any layer.

Table 3: **Layer values for vlan.id keyword**

Value	Description
[default]	Match with any layer
0 - 2	Match specific layer
-3 - -1	Match specific layer with back to front indexing
all	Match only if all layers match
any	Match with any layer

This small illustration shows how indexing works for vlan.id:

```
[ethernet]
[vlan 666 (index 0 and -2)]
[vlan 123 (index 1 and -1)]
[ipv4]
[udp]
```

Examples

Example of a signature that would alert if any of the VLAN IDs is equal to 300:

alert ip any any -> any any (msg:"Vlan ID is equal to 300"; vlan.id:300; sid:1;)

Example of a signature that would alert if the VLAN ID at layer 1 is equal to 300:

alert ip any any -> any any (msg:"Vlan ID is equal to 300 at layer 1"; vlan.id:300,1; sid:1;)

Example of a signature that would alert if the VLAN ID at the last layer is equal to 400:

alert ip any any -> any any (msg:"Vlan ID is equal to 400 at the last layer"; vlan.id:400,-1; sid:1;)

Example of a signature that would alert only if all the VLAN IDs are greater than 100:

alert ip any any -> any any (msg:"All Vlan IDs are greater than 100"; vlan.id:>100,all; sid:1;)

It is also possible to use the vlan.id content as a fast_pattern by using the `prefilter` keyword, as shown in the following example.

alert ip any any -> any any (msg:"Vlan ID is equal to 200 at layer 1"; vlan.id:200,1; prefilter; sid:1;)

8.58.2 vlan.layers

Matches based on the number of layers.

Syntax:

```
vlan.layers: [op]number;
```

It can be matched exactly, or compared using the `op` setting:

```
vlan.layers:3    # exactly 3 vlan layers
vlan.layers:<3   # less than 3 vlan layers
vlan.layers:>=2  # more or equal to 2 vlan layers
```

vlan.layers uses *unsigned 8-bit integer*.

The minimum and maximum values that vlan.layers can be are `0` and `3`.

Examples

Example of a signature that would alert if a packet has 0 VLAN layers:

alert ip any any -> any any (msg:"Packet has 0 vlan layers"; vlan.layers:0; sid:1;)

Example of a signature that would alert if a packet has more than 1 VLAN layers:

alert ip any any -> any any (msg:"Packet has more than 1 vlan layer"; vlan.layers:>1; sid:1;)

It is also possible to use the vlan.layers content as a fast_pattern by using the `prefilter` keyword, as shown in the following example.

alert ip any any -> any any (msg:"Packet has 2 vlan layers"; vlan.layers:2; prefilter; sid:1;)

8.59 LDAP Keywords

8.59.1 LDAP Request and Response operations

Table 4: **Operation values for ldap.request.operation and ldap.responses.operation keywords**

Code	Operation
0	bind_request
1	bind_response
2	unbind_request
3	search_request
4	search_result_entry
5	search_result_done
6	modify_request
7	modify_response
8	add_request
9	add_response
10	del_request
11	del_response
12	mod_dn_request
13	mod_dn_response
14	compare_request
15	compare_response
16	abandon_request
19	search_result_reference
23	extended_request
24	extended_response
25	intermediate_response

The keywords ldap.request.operation and ldap.responses.operation accept both the operation code and the operation name as arguments.

8.59.2 ldap.request.operation

Suricata has a `ldap.request.operation` keyword that can be used in signatures to identify and filter network packets based on Lightweight Directory Access Protocol request operations.

Syntax:

```
ldap.request.operation: operation;
```

ldap.request.operation uses *unsigned 8-bit integer*.

See `suricata --list-keywords=json | jq '."ldap.request.operation".enum_values'`

This keyword maps to the EVE field `ldap.request.operation`

Examples

Example of a signatures that would alert if the packet has an LDAP bind request operation:

alert ldap any any -> any any (msg:"Test LDAP bind request"; ldap.request.operation:0; sid:1;)

alert ldap any any -> any any (msg:"Test LDAP bind request"; ldap.request.operation:bind_request; sid:1;)

8.59.3 ldap.bind_request.version

Matches the protocol version in an LDAP BindRequest.

Syntax:

```
ldap.bind_request.version: [op]version;
```

It can be matched exactly, or compared using the `op` setting:

```
ldap.bind_request.version:3    # exactly version 3
ldap.bind_request.version:<3   # less than version 3
ldap.bind_request.version:>=2  # version 2 or later
```

`ldap.bind_request.version` uses *unsigned 8-bit integer*.

This keyword maps to the EVE field `ldap.request.bind_request.version`.

Example

Example of a signature that alerts on an LDAP version 3 BindRequest:

alert ldap any any -> any any (msg:"LDAP version 3 BindRequest"; ldap.bind_request.version:3; sid:1;)

8.59.4 ldap.bind_request.authentication

Matches the authentication choice in an LDAP BindRequest. The accepted values are `simple` and `sasl`. This keyword selects an authentication type; it is not a sticky buffer.

Syntax:

```
ldap.bind_request.authentication:<simple|sasl>;
```

See `suricata --list-keywords=json | jq '."ldap.bind_request.authentication".enum_values'`

Example

Example of a signature that alerts on an LDAP BindRequest with SASL authentication:

alert ldap any any -> any any (msg:"LDAP SASL bind"; ldap.bind_request.authentication:sasl; sid:1;)

8.59.5 ldap.bind_request.sasl.mechanism

Matches the SASL mechanism in an LDAP BindRequest. This is a sticky buffer and can be used as a `fast_pattern`.

Syntax:

```
ldap.bind_request.sasl.mechanism; content:"<mechanism>";
```

This keyword maps to the EVE field `ldap.request.bind_request.sasl.mechanism`.

Example

Example of a signature that alerts on an LDAP BindRequest with SASL mechanism GSS-SPNEGO:

alert ldap any any -> any any (msg:"LDAP SASL mechanism GSS-SPNEGO"; ldap.bind_request.sasl.mechanism; content:"GSS-SPNEGO"; sid:1;)

8.59.6 ldap.bind_request.sasl.credentials

Matches the raw SASL credentials in an LDAP BindRequest. This is a sticky buffer and can be used as a `fast_pattern`. The buffer is unavailable when the optional credentials field is absent.

Syntax:

```
ldap.bind_request.sasl.credentials; content:"<credentials>";
```

This keyword maps to the EVE field `ldap.request.bind_request.sasl.credentials`.

8.59.7 ldap.responses.operation

Suricata has a `ldap.responses.operation` keyword that can be used in signatures to identify and filter network packets based on Lightweight Directory Access Protocol response operations.

Syntax:

```
ldap.responses.operation: operation[,index];
```

ldap.responses.operation uses *unsigned 8-bit integer*.

ldap.responses.operation is also a *multi-integer*.

This keyword maps to the EVE field `ldap.responses[].operation`

An LDAP request operation can receive multiple responses. By default, the ldap.responses.operation keyword matches all indices, but it is possible to specify a particular index for matching and also use flags such as `all` and `any`.

Table 5: **Index values for ldap.responses.operation keyword**

Value	Description
[default]	Match with any index
all	Match only if all indexes match
any	Match with any index
0>=	Match specific index
0<	Match specific index with back to front indexing

Examples

Example of a signatures that would alert if the packet has an LDAP bind response operation:

alert ldap any any -> any any (msg:"Test LDAP bind response"; ldap.responses.operation:1; sid:1;)

alert ldap any any -> any any (msg:"Test LDAP bind response"; ldap.responses.operation:bind_response; sid:1;)

Example of a signature that would alert if the packet has an LDAP search_result_done response operation at index 1:

alert ldap any any -> any any (msg:"Test LDAP search response"; ldap.responses.operation:search_result_done,1; sid:1;)

Example of a signature that would alert if all the responses are of type search_result_entry:

alert ldap any any -> any any (msg:"Test LDAP search response"; ldap.responses.operation:search_result_entry,all; sid:1;)

The keyword ldap.responses.operation supports back to front indexing with negative numbers, this means that -1 will represent the last index, -2 the second to last index, and so on. This is an example of a signature that would alert if a search_result_entry response is found at the last index:

alert ldap any any -> any any (msg:"Test LDAP search response"; ldap.responses.operation:search_result_entry,-1; sid:1;)

8.59.8 ldap.responses.count

Matches based on the number of responses.

Syntax:

```
ldap.responses.count: [op]number;
```

It can be matched exactly, or compared using the op setting:

```
ldap.responses.count:3    # exactly 3 responses
ldap.responses.count:<3   # less than 3 responses
ldap.responses.count:>=2  # more or equal to 2 responses
```

ldap.responses.count uses *unsigned 32-bit integer*.

This keyword maps to the EVE field `len(ldap.responses[])`

Examples

Example of a signature that would alert if a packet has 0 LDAP responses:

alert ldap any any -> any any (msg:"Packet has 0 LDAP responses"; ldap.responses.count:0; sid:1;)

Example of a signature that would alert if a packet has more than 2 LDAP responses:

alert ldap any any -> any any (msg:"Packet has more than 2 LDAP responses"; ldap.responses.count:>2; sid:1;)

8.59.9 ldap.request.dn

Matches on LDAP distinguished names from request operations.

Comparison is case-sensitive.

Syntax:

```
ldap.request.dn; content:"<content to match against>";
```

`ldap.request.dn` is a 'sticky buffer' and can be used as a `fast_pattern`.

This keyword maps to the EVE fields:

- `ldap.request.bind_request.name`
- `ldap.request.add_request.entry`
- `ldap.request.search_request.base_object`
- `ldap.request.modify_request.object`
- `ldap.request.del_request.dn`
- `ldap.request.mod_dn_request.entry`
- `ldap.request.compare_request.entry`

Example

Example of a signature that would alert if a packet has the LDAP distinguished name `uid=jdoe,ou=People,dc=example,dc=com`:

alert ldap any any -> any any (msg:"Test LDAPDN"; ldap.request.dn; content:"uid=jdoe,ou=People,dc=example,dc=com"; sid:1;)

It is possible to use the keyword `ldap.request.operation` in the same rule to specify the operation to match.

Here is an example of a signature that would alert if a packet has an LDAP search request operation and contains the LDAP distinguished name `dc=example,dc=com`.

alert ldap any any -> any any (msg:"Test LDAPDN and operation"; ldap.request.operation:search_request; ldap.request.dn; content:"dc=example,dc=com"; sid:1;)

8.59.10 ldap.responses.dn

Matches on LDAP distinguished names from response operations.

Comparison is case-sensitive.

Syntax:

```
ldap.responses.dn; content:"<content to match against>";
```

`ldap.responses.dn` is a 'sticky buffer' and can be used as a `fast_pattern`.

`ldap.responses.dn` supports multiple buffer matching, see *Multiple Buffer Matching*.

This keyword maps to the EVE fields:

- `ldap.responses[].search_result_entry.base_object`
- `ldap.responses[].bind_response.matched_dn`
- `ldap.responses[].search_result_done.matched_dn`
- `ldap.responses[].modify_response.matched_dn`
- `ldap.responses[].add_response.matched_dn`
- `ldap.responses[].del_response.matched_dn`
- `ldap.responses[].mod_dn_response.matched_dn`
- `ldap.responses[].compare_response.matched_dn`
- `ldap.responses[].extended_response.matched_dn`

Note

If a response within the array does not contain the distinguished name field, this field will be interpreted as an empty buffer.

Example

Example of a signature that would alert if a packet has the LDAP distinguished name `dc=example,dc=com`:

alert ldap any any -> any any (msg:"Test LDAPDN"; ldap.responses.dn; content:"dc=example,dc=com"; sid:1;)

It is possible to use the keyword `ldap.responses.operation` in the same rule to specify the operation to match.

Here is an example of a signature that would alert if a packet has an LDAP search result entry operation at index 1 on the responses array, and contains the LDAP distinguished name `dc=example,dc=com`.

alert ldap any any -> any any (msg:"Test LDAPDN and operation"; ldap.responses.operation:search_result_entry,1; ldap.responses.dn; content:"dc=example,dc=com"; sid:1;)

8.59.11 ldap.responses.result_code

Suricata has a `ldap.responses.result_code` keyword that can be used in signatures to identify and filter network packets based on their LDAP result code.

Syntax:

```
ldap.responses.result_code: code[,index];
```

ldap.responses.result_code uses *unsigned 32-bit integer*.

ldap.responses.result_code is also a *multi-integer*.

See `suricata --list-keywords=json | jq '."ldap.responses.result_code".enum_values'`

This keyword maps to the following eve fields:

- `ldap.responses[].bind_response.result_code`
- `ldap.responses[].search_result_done.result_code`
- `ldap.responses[].modify_response.result_code`
- `ldap.responses[].add_response.result_code`
- `ldap.responses[].del_response.result_code`
- `ldap.responses[].mod_dn_response.result_code`
- `ldap.responses[].compare_response.result_code`
- `ldap.responses[].extended_response.result_code`

Table 6: **Result code values for ldap.responses.result_code**

Code	Name
0	success
1	operations_error
2	protocol_error
3	time_limit_exceeded
4	size_limit_exceeded
5	compare_false
6	compare_true
7	auth_method_not_supported
8	stronger_auth_required
10	referral
11	admin_limit_exceeded
12	unavailable_critical_extension
13	confidentiality_required
14	sasl_bind_in_progress
16	no_such_attribute
17	undefined_attribute_type
18	inappropriate_matching
19	constraint_violation

continues on next page

Table 6 – continued from previous page

Code	Name
20	attribute_or_value_exists
21	invalid_attribute_syntax
32	no_such_object
33	alias_problem
34	invalid_dns_syntax
35	is_leaf
36	alias_dereferencing_problem
48	inappropriate_authentication
49	invalid_credentials
50	insufficient_access_rights
51	busy
52	unavailable
53	unwilling_to_perform
54	loop_detect
60	sort_control_missing
61	offset_range_error
64	naming_violation
65	object_class_violation
66	not_allowed_on_non_leaf
67	not_allowed_on_rdn
68	entry_already_exists
69	object_class_mods_prohibited
70	results_too_large
71	affects_multiple_dsas
76	control_error
80	other
81	server_down
82	local_error
83	encoding_error
84	decoding_error
85	timeout
86	auth_unknown
87	filter_error
88	user_canceled
89	param_error
90	no_memory
91	connect_error
92	not_supported
93	control_not_found
94	no_results_returned
95	more_results_to_return
96	client_loop
97	referral_limit_exceeded
100	invalid_response
101	ambiguous_response
112	tls_not_supported
113	intermediate_response
114	unknown_type
118	canceled
119	no_such_operation

continues on next page

Table 6 – continued from previous page

Code	Name
120	too_late
121	cannot_cancel
122	assertion_failed
123	authorization_denied
4096	e_sync_refresh_required
16654	no_operation

More information about LDAP result code values can be found here: https://ldap.com/ldap-result-code-reference/

An LDAP request operation can receive multiple responses. By default, the ldap.responses.result_code keyword matches with any indices, but it is possible to specify a particular index for matching and also use flags such as `all` and `any`.

Table 7: **Index values for ldap.responses.result_code keyword**

Value	Description
[default]	Match with any index
all	Match only if all indexes match
any	Match with any index
0>=	Match specific index
0<	Match specific index with back to front indexing

Examples

Example of signatures that would alert if the packet has a `success` LDAP result code at any index:

alert ldap any any -> any any (msg:"Test LDAP result code"; ldap.responses.result_code:0; sid:1;)

alert ldap any any -> any any (msg:"Test LDAP result code"; ldap.responses.result_code:success,any; sid:1;)

Example of a signature that would alert if the packet has an `unavailable` LDAP result code at index 1:

alert ldap any any -> any any (msg:"Test LDAP result code at index 1"; ldap.responses.result_code:unavailable,1; sid:1;)

Example of a signature that would alert if all the responses have a `success` LDAP result code:

alert ldap any any -> any any (msg:"Test all LDAP responses have success result code"; ldap.responses.result_code:success,all; sid:1;)

The keyword ldap.responses.result_code supports back to front indexing with negative numbers, this means that -1 will represent the last index, -2 the second to last index, and so on. This is an example of a signature that would alert if a `success` result code is found at the last index:

alert ldap any any -> any any (msg:"Test LDAP success at last index"; ldap.responses.result_code:success,-1; sid:1;)

8.59.12 ldap.responses.message

Matches on LDAP error messages from response operations.

Comparison is case-sensitive.

Syntax:

```
ldap.responses.message; content:"<content to match against>";
```

`ldap.responses.message` is a 'sticky buffer' and can be used as a `fast_pattern`.

`ldap.responses.message` supports multiple buffer matching, see *Multiple Buffer Matching*.

This keyword maps to the EVE fields:

- `ldap.responses[].bind_response.message`
- `ldap.responses[].search_result_done.message`
- `ldap.responses[].modify_response.message`
- `ldap.responses[].add_response.message`
- `ldap.responses[].del_response.message`
- `ldap.responses[].mod_dn_response.message`
- `ldap.responses[].compare_response.message`
- `ldap.responses[].extended_response.message`

Note

If a response within the array does not contain the error message field, this field will be interpreted as an empty buffer.

Example

Example of a signature that would alert if a packet has the LDAP error message `Size limit exceeded`:

alert ldap any any -> any any (msg:"Test LDAP error message"; ldap.responses.message; content:"Size limit exceeded"; sid:1;)

8.59.13 ldap.request.attribute_type

Matches on LDAP attribute type from request operations.

Comparison is case-sensitive.

Syntax:

```
ldap.request.attribute_type; content:"<content to match against>";
```

`ldap.request.attribute_type` is a 'sticky buffer' and can be used as a `fast_pattern`.

`ldap.request.attribute_type` supports multiple buffer matching, see *Multiple Buffer Matching*.

This keyword maps to the EVE fields:

- `ldap.request.search_request.attributes[]`
- `ldap.request.modify_request.changes[].modification.attribute_type`
- `ldap.request.add_request.attributes[].name`
- `ldap.request.compare_request.attribute_value_assertion.description`

Example

Example of a signature that would alert if a packet has the LDAP attribute type `objectClass`:

alert ldap any any -> any any (msg:"Test attribute type"; ldap.request.attribute_type; content:"objectClass"; sid:1;)

It is possible to use the keyword `ldap.request.operation` in the same rule to specify the operation to match.

Here is an example of a signature that would alert if a packet has an LDAP add request operation and contains the LDAP attribute type `objectClass`.

alert ldap any any -> any any (msg:"Test attribute type and operation"; ldap.request.operation:add_request; ldap.request.attribute_type; content:"objectClass"; sid:1;)

8.59.14 ldap.responses.attribute_type

Matches on LDAP attribute type from response operations.

Comparison is case-sensitive.

Syntax:

```
ldap.responses.attribute_type; content:"<content to match against>";
```

`ldap.responses.attribute_type` is a 'sticky buffer' and can be used as a `fast_pattern`.

`ldap.responses.attribute_type` supports multiple buffer matching, see *Multiple Buffer Matching*.

This keyword maps to the EVE field `ldap.responses[].search_result_entry.attributes[].type`

Example

Example of a signature that would alert if a packet has the LDAP attribute type `dc`:

alert ldap any any -> any any (msg:"Test responses attribute type"; ldap.responses.attribute_type; content:"dc"; sid:1;)

It is possible to use the keyword `ldap.responses.operation` in the same rule to specify the operation to match.

Here is an example of a signature that would alert if a packet has an LDAP search result entry operation at index 1 on the responses array, and contains the LDAP attribute type `dc`.

alert ldap any any -> any any (msg:"Test attribute type and operation"; ldap.responses.operation:search_result_entry,1; ldap.responses.attribute_type; content:"dc"; sid:1;)

8.60 PGSQL Keywords

8.60.1 pgsql.query

This keyword is a sticky buffer that allows matching on the contents of PostgreSQL's *query* request messages parsed by the engine. Note that this buffer inspects only the *string* portion of the PostgreSQL message, skipping other fields such as identifier and length, and focusing on the query itself.

Currently, it exposes the contents of the `pgsql.request.simple_query` field from EVE output.

`pgsql.query` can be used as a `fast_pattern` (see *fast_pattern*).

Use `nocase` with this keyword to avoid case sensitivity for the matches.

Examples

alert pgsql any any -> any any (msg:"Simple SELECT rule"; pgsql.query; content:"SELECT *"; sid:1;)

alert pgsql any any -> any any (msg:"Simple delete rule"; pgsql.query; content:"delete"; nocase; sid:2;)

8.61 Email Keywords

8.61.1 email.from

Matches the MIME `From` field of an email.

Comparison is case-sensitive.

Syntax:

```
email.from; content:"<content to match against>";
```

`email.from` is a 'sticky buffer' and can be used as a `fast_pattern`.

This keyword maps to the EVE field `email.from`

Example

Example of a signature that would alert if a packet contains the MIME field `from` with the value `toto <toto@gmail.com>`

alert smtp any any -> any any (msg:"Test mime email from"; email.from; content:"toto <toto@gmail.com>"; sid:1;)

8.61.2 email.subject

Matches the MIME `Subject` field of an email.

Comparison is case-sensitive.

Syntax:

```
email.subject; content:"<content to match against>";
```

`email.subject` is a 'sticky buffer' and can be used as a `fast_pattern`.

This keyword maps to the EVE field `email.subject`

Example

Example of a signature that would alert if a packet contains the MIME field `subject` with the value `This is a test email`

alert smtp any any -> any any (msg:"Test mime email subject"; email.subject; content:"This is a test email"; sid:1;)

8.61.3 email.to

Matches the MIME `To` field of an email.

Comparison is case-sensitive.

Syntax:

```
email.to; content:"<content to match against>";
```

`email.to` is a 'sticky buffer' and can be used as a `fast_pattern`.

This keyword maps to the EVE field `email.to`

Example

Example of a signature that would alert if a packet contains the MIME field `to` with the value `172.16.92.2@linuxbox`

alert smtp any any -> any any (msg:"Test mime email to"; email.to; content:"172.16.92.2@linuxbox"; sid:1;)

8.61.4 email.cc

Matches the MIME `Cc` field of an email.

Comparison is case-sensitive.

Syntax:

```
email.cc; content:"<content to match against>";
```

`email.cc` is a 'sticky buffer' and can be used as a `fast_pattern`.

This keyword maps to the EVE field `email.cc[]`

Example

Example of a signature that would alert if a packet contains the MIME field `cc` with the value `Emily <emily.roberts@example.com>, Ava <ava.johnson@example.com>, Sophia Wilson <sophia.wilson@example.com>`

alert smtp any any -> any any (msg:"Test mime email cc"; email.cc; content:"Emily <emily.roberts@example.com>, Ava <ava.johnson@example.com>, Sophia Wilson <sophia.wilson@example.com>"; sid:1;)

8.61.5 email.date

Matches the MIME `Date` field of an email.

Comparison is case-sensitive.

Syntax:

```
email.date; content:"<content to match against>";
```

`email.date` is a 'sticky buffer' and can be used as a `fast_pattern`.

This keyword maps to the EVE field `email.date`

Example

Example of a signature that would alert if a packet contains the MIME field `date` with the value `Fri, 21 Apr 2023 05:10:36 +0000`

alert smtp any any -> any any (msg:"Test mime email date"; email.date; content:"Fri, 21 Apr 2023 05:10:36 +0000"; sid:1;)

8.61.6 email.message_id

Matches the MIME `Message-Id` field of an email.

Comparison is case-sensitive.

Syntax:

```
email.message_id; content:"<content to match against>";
```

`email.message_id` is a 'sticky buffer' and can be used as a `fast_pattern`.

This keyword maps to the EVE field `email.message_id`

Example

Example of a signature that would alert if a packet contains the MIME field `message id` with the value `<alpine.DEB.2.00.1311261630120.9535@sd-26634.dedibox.fr>`

alert smtp any any -> any any (msg:"Test mime email message id"; email.message_id; content:"<alpine.DEB.2.00.1311261630120.9535@sd-26634.dedibox.fr>"; sid:1;)

8.61.7 email.x_mailer

Matches the MIME `X-Mailer` field of an email.

Comparison is case-sensitive.

Syntax:

```
email.x_mailer; content:"<content to match against>";
```

`email.x_mailer` is a 'sticky buffer' and can be used as a `fast_pattern`.

This keyword maps to the EVE field `email.x_mailer`

Example

Example of a signature that would alert if a packet contains the MIME field `x-mailer` with the value `Microsoft Office Outlook, Build 11.0.5510`

alert smtp any any -> any any (msg:"Test mime email x-mailer"; email.x_mailer; content:"Microsoft Office Outlook, Build 11.0.5510"; sid:1;)

8.61.8 email.url

Matches `URL` extracted of an email.

Comparison is case-sensitive.

Syntax:

```
email.url; content:"<content to match against>";
```

`email.url` is a 'sticky buffer' and can be used as a `fast_pattern`.

`email.url` supports multiple buffer matching, see *Multiple Buffer Matching*.

This keyword maps to the EVE field `email.url[]`

Example

Example of a signature that would alert if an email contains the `url test-site.org/blah/123/`.

alert smtp any any -> any any (msg:"Test mime email url"; email.url; content:"test-site.org/blah/123/"; sid:1;)

8.61.9 email.received

Matches `Received` field of an email.

Comparison is case-sensitive.

Syntax:

```
email.received; content:"<content to match against>";
```

`email.received` is a 'sticky buffer' and can be used as a `fast_pattern`.

`email.received` supports multiple buffer matching, see *Multiple Buffer Matching*.

This keyword maps to the EVE field `email.received[]`

Example

Example of a signature that would alert if a packet contains the MIME field `received` with the value `from [65.201.218.30] (helo=COZOXORY.club)by 173-66-46-112.wash.fios.verizon.net with esmtpa (Exim 4.86)(envelope-from )id 71cF63a9for mirjam@abrakadabra.ch; Mon, 29 Jul 2019 17:01:45 +0000`

alert smtp any any -> any any (msg:"Test mime email received"; email.received; content:"from [65.201.218.30] (helo=COZOXORY.club)by 173-66-46-112.wash.fios.verizon.net with esmtpa (Exim 4.86)(envelope-from)id 71cF63a9for mirjam@abrakadabra.ch; Mon, 29 Jul 2019 17:01:45 +0000"; sid:1;)

8.61.10 email.body_md5

Matches the `md5` hash generated from an email body. This keyword only works if config option `app-layer.protocols.smtp.mime.body-md5` is enabled or auto. It should be used with `requires: keyword email.body_md5;`

Syntax:

```
email.body_md5; content:"<content to match against>";
```

`email.body_md5` is a 'sticky buffer' and can be used as a `fast_pattern`.

This keyword maps to the EVE field `email.body_md5`.

Example

Example of a signature that would alert if the hash `ed00c81b85fa455d60e19f1230977134` is generated from an email body:

alert smtp any any -> any any (msg:"Test mime email body_md5"; requires: keyword email.body_md5; email.body_md5; content:"ed00c81b85fa455d60e19f1230977134"; sid:1;)

8.62 Rule Types and Categorization

Once parsed, Suricata rules are categorized for performance and further processing (as different rule types will be handled by specific engine modules). The signature types are defined in src/detect.h:

Listing 1: src/detect.h

```
enum SignatureType {
    SIG_TYPE_NOT_SET = 0,
```

(continues on next page)

(continued from previous page)

```
    SIG_TYPE_IPONLY,      // rule is handled by IPONLY engine
    SIG_TYPE_LIKE_IPONLY, // rule is handled by pkt engine, has action effect like ip-
↪only
    /** Proto detect only signature.
     *  Inspected once per direction when protocol detection is done. */
    SIG_TYPE_PDONLY, // rule is handled by PDONLY engine
    SIG_TYPE_DEONLY,
    SIG_TYPE_PKT,
    SIG_TYPE_PKT_STREAM,
    SIG_TYPE_STREAM,

    SIG_TYPE_APPLAYER, // app-layer but not tx, e.g. appproto
    SIG_TYPE_APP_TX,   // rule is handled by TX engine

    SIG_TYPE_MAX,
};
```

In more human readable terms:

Table 8: Suricata Rule Types, and their Engine Analysis Term

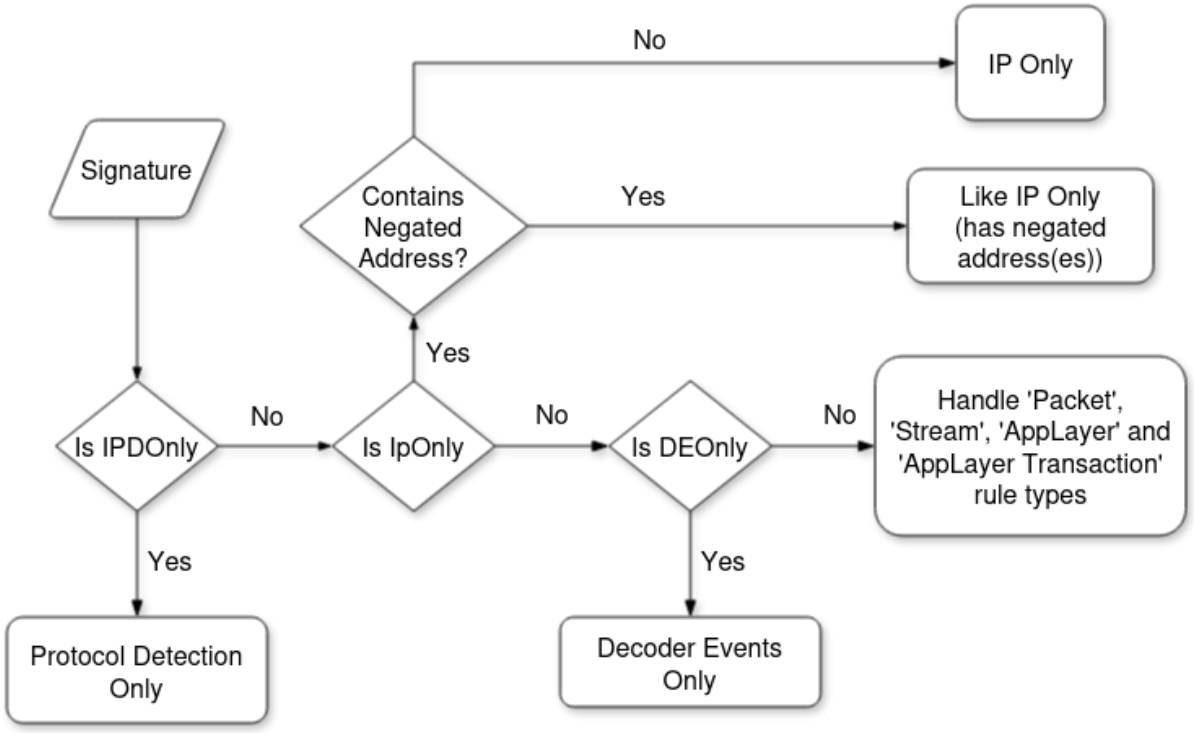
Rule Type	Code Symbol	Engine-Analysis Representation
Decoder Events Only	SIG_TYPE_DEONLY	de_only
Packet	SIG_TYPE_PKT	pkt
IP Only	SIG_TYPE_IPONLY	ip_only
IP Only (contains negated address(es))	SIG_TYPE_LIKE_IPONLY	like_ip_only
Protocol Detection Only	SIG_TYPE_PDONLY	pd_only
Packet-Stream	SIG_TYPE_PKT_STREAM	pkt_stream
Stream	SIG_TYPE_STREAM	stream
Application Layer Protocol	SIG_TYPE_APPLAYER	app_layer
Application Layer Protocol Transactions	SIG_TYPE_APP_TX	app_tx

The rule type will impact:

- To what does the signature action apply, in case of a match (*Action Scope*)
- When is the rule matched against traffic (*Inspection Hook*)
- Against what the rule matches (*Data Exposed*)

This categorization is done taking into consideration the presence or absence of certain rule elements, as well as the type of keywords used. The categorization currently takes place in src/detect-engine-build.c:void SignatureSetType().

The `SignatureSetType()` overall flow is described below:



Flowcharts expanding uncovered functions or portions of the overall algorithm above are shown in the *Detailed Flowcharts* section.

The following table lists all Suricata signature types, and how they impact the aspects aforementioned.

Table 9: Suricata Rule Types

Type	Action Scope	Inspection Hook	Data Exposed	Keyword Examples (non-exhaustive)
Decoder Events Only (de_only	Packet	Per-broken/ invalid packet	Decoding events	decode-event
Packet (pkt)	Packet	Per-packet basis	Packet-level info (e.g.: header info)	tcp-pkt, itype, tcp.hdr, tcp.seq, ttl etc.
IP Only (ip_only	Flow (if existing). Packets (if not part of a flow)	Once per direction	IP addresses on the flow	Source/ Destination field of a rule
IP Only (contains negated address) [2] (like_ip	Flow	All packets	IP addresses on the flow	Source/ Destination field of a rule containing negated address
Protocol Detection Only (pd_only	Flow	Once per direction, when protocol detection is done	Protocol detected for the flow	app-layer-protocol
Packet-Stream (pkt_str	Flow, if stateful [1]	Per stream chunk, if stateful, per-packet if not (stream payload AND packet payload)	The reassembled stream and/or payload data	content with startswith or depth
Stream (stream)	Flow, if stateful [1]	Stream chunks, if stateful, just packets if not	Stream reassembled payload or packet payload data	tcp-stream in protocol field; simple content; byte_extract
Application Layer Protocol (app_lay	Flow	Per-packet basis	'protocol' field in a rule	Protocol field of a rule
Application Layer Protocol Transactions (app_tx)	Flow	Per *transaction* update	Buffer keywords	Application layer protocol-related, e.g. http.host, rfb.secresult, dcerpc.stub_data, frame keywords

Note

Action Scope: *Flow, if stateful*

(1) Apply to the flow. If a segment isn't accepted into a stream for any reason (such as packet anomalies, errors, memcap reached etc), the rule will be applied on a packet level.

Warning

Although both are related to matching on application layer protocols, as the table suggests, since Suricata 7 a Protocol Detection rule (that uses the `app-layer-protocol` keyword) is not internally classified the same as a rule simply matching on the application layer protocol on the `protocol` field.

8.62.1 Signature Properties

The *Action Scope* mentioned above relates to the Signature Properties, as seen in src/detect-engine.c:

Listing 2: src/detect-engine.c

```
const struct SignatureProperties signature_properties[SIG_TYPE_MAX] = {
    /* SIG_TYPE_NOT_SET */      { SIG_PROP_FLOW_ACTION_PACKET, },
    /* SIG_TYPE_IPONLY */       { SIG_PROP_FLOW_ACTION_FLOW, },
    /* SIG_TYPE_LIKE_IPONLY */  { SIG_PROP_FLOW_ACTION_FLOW, },
    /* SIG_TYPE_PDONLY */       { SIG_PROP_FLOW_ACTION_FLOW, },
    /* SIG_TYPE_DEONLY */       { SIG_PROP_FLOW_ACTION_PACKET, },
    /* SIG_TYPE_PKT */          { SIG_PROP_FLOW_ACTION_PACKET, },
    /* SIG_TYPE_PKT_STREAM */   { SIG_PROP_FLOW_ACTION_FLOW_IF_STATEFUL, },
    /* SIG_TYPE_STREAM */       { SIG_PROP_FLOW_ACTION_FLOW_IF_STATEFUL, },
    /* SIG_TYPE_APPLAYER */     { SIG_PROP_FLOW_ACTION_FLOW, },
    /* SIG_TYPE_APP_TX */       { SIG_PROP_FLOW_ACTION_FLOW, },
};
```

Signature: Require Real Packet

Aside from the scope of action of a signature, certain rule conditions will require that it matches against a *real packet* (as opposed to a *pseudo packet*). These rules are flagged with `SIG_MASK_REQUIRE_REAL_PKT` by the engine, and will have `real_pkt` listed as one of the rule's `requirements`. (See `engine-analysis` example output for the *Packet* rule type.)

A *pseudo packet* is an internal resource used by the engine when a flow is over but there is still data to be processed, such as when there is a flow timeout. A fake packet is then injected in the flow to finish up processing before ending it.

Those two types will be more documented soon (tracking #7424).

8.62.2 Signature Types and Variable-like Keywords

Keywords such as flow variables (`flowint`, `flowbits`), `datasets`, and similar ones can alter the rule type, if present in a signature.

That happens because the variable condition can change per packet. Thus, the Signature is categorized as a *packet* rule.

This affects rule types:

- Application Layer (`app_layer`)

- Protocol Detection Only (`pd_only`)
- Decoder Events Only (`de_only`)
- IP Only (`ip_only`) [3]
- Like IP Only (`like_ip_only`) [3]

The rule examples provided further cover some such cases, but the table below lists those keywords with more details:

Table 10: Variable-like Keywords

Keyword	Keyword Option	Rule Type change?
`flow`	`to_server`, `to_client`	no type changes [3]
`flow`	`established`, `not_established`	to *packet*
`flowbits`, `xbits`, `hostbits`	`isset`, `isnotset`	to *packet*
`flowbits`, `xbits`, `hostbits`	`set`, `unset`, `toggle`	no type change
`flowint`	`isset`, `notset`, all operators	to *packet*
`flowint`	defining the variable; unseting;	no type change
`iprep`	`isset`, `notset`, all operators	to *packet*

Note

IP Only and Like IP Only

(3) Unlike the other affected types, signatures that would otherwise be classified as `ip_only` or `like_ip_only` become Packet rules if the `flow` keyword is used, regardless of option.

Note

`dataset`, while may look similar to the keywords above, doesn't pertain to this list as it can only be used with sticky buffer keywords, thus being only available to Application Layer Transaction rules (*app_tx*), which are not affected by this.

Flowbits: `isset`

If a non-stateful rule (e.g. a `pkt` rule) checks if a `flowbit` is set (like in *flowbits:fb6,isset*) and the rule that sets that variable is a stateful one, such as an `app_tx` rule, the engine will set a flag to indicate that that rule is also stateful - without altering its signature type. This flag is currently `SIG_FLAG_INIT_STATE_MATCH` (cf. ticket #7483).

There is a work-in-progress to add information about this to the `engine-analysis` report (ticket #7456).

8.62.3 Signatures per Type

This section offers brief descriptions for each rule type, and illustrates what signatures of each type may look like. It is possible to learn the type of a signature, as well as other important information, by running Suricata in *engine analysis* mode.

For each rule type, there is also a sample of the Engine Analysis report for one or more of rule(s) shown.

Decoder Events Only

Signatures that inspect broken or invalid packets. They expose Suricata decoding events.

For more examples check https://github.com/OISF/suricata/blob/main/rules/decoder-events.rules.

Example

alert pkthdr any any -> any any (msg:"SURICATA IPv6 duplicated Hop-By-Hop Options extension header"; decode-event:ipv6.exthdr_dupl_hh; classtype:protocol-command-decode; sid:1101;)

drop pkthdr any any -> any any (msg:"SURICATA IPv4 invalid option length"; *:example-rule-emphasis: `decode-event:ipv4.opt_invalid_len;* classtype:protocol-command-decode; sid:2200005; rev:2;)

Engine-Analysis Report


```
{
  "raw": "alert pkthdr any any -> any any (msg:\"SURICATA IPv6 duplicated Hop-By-Hop␣
↪Options extension header\"; decode-event:ipv6.exthdr_dupl_hh; classtype:protocol-
↪command-decode; sid:1101;)",
  "id": 1101,
  "gid": 1,
  "rev": 0,
  "msg": "SURICATA IPv6 duplicated Hop-By-Hop Options extension header",
  "app_proto": "unknown",
  "requirements": [
    "engine_event"
  ],
  "type": "de_only",
  "flags": [
    "src_any",
    "dst_any",
    "sp_any",
    "dp_any",
    "toserver",
    "toclient"
  ],
  "pkt_engines": [
    {
      "name": "packet",
      "is_mpm": false
    }
  ],
  "frame_engines": [],
  "lists": {
    "packet": {
      "matches": [
        {
          "name": "decode-event"
        }
      ]
    }
  }
}
```

Packet

Rules that expose/ inspect information on a packet-level (for instance, the header). Certain flow keywords may also turn a rule into a `pkt` rule, if they require per-packet inspection (cf. *Signature Types and Variable-like Keywords*).

Examples

alert tcp-pkt any any -> any any (msg:"tcp-pkt, anchored content"; content:"abc"; startswith; sid:203;)

alert tcp any any -> any any (msg:"ttl"; ttl:123; sid:701;)

alert udp any any -> any any (msg:"UDP with flow direction"; flow:to_server; sid:1001;)

alert tcp any any -> any 443 (flow: to_server; flowbits:set,tls_error; sid:1604; msg:"Allow TLS error handling (outgoing packet) - non-stateful rule";)

alert tcp-pkt any any -> any any (msg:"Flowbit isset"; flowbits:isset,fb6; flowbits:isset,fb7; sid:1919;)

Engine-Analysis Report


```
{
  "raw": "alert tcp-pkt any any -> any any (msg:\"tcp-pkt, anchored content\"; content:\
↪"abc\"; startswith; sid:203;)",
  "id": 203,
  "gid": 1,
  "rev": 0,
  "msg": "tcp-pkt, anchored content",
  "app_proto": "unknown",
  "requirements": [
    "payload",
    "real_pkt"
  ],
  "type": "pkt",
  "flags": [
    "src_any",
    "dst_any",
    "sp_any",
    "dp_any",
    "need_packet",
    "toserver",
    "toclient",
    "prefilter"
  ],
  "pkt_engines": [
    {
      "name": "payload",
      "is_mpm": true
    }
  ],
  "frame_engines": [],
  "lists": {
    "payload": {
      "matches": [
        {
          "name": "content",
```


(continues on next page)

(continued from previous page)


```
          "content": {
            "pattern": "abc",
            "length": 3,
            "nocase": false,
            "negated": false,
            "starts_with": true,
            "ends_with": false,
            "is_mpm": true,
            "no_double_inspect": false,
            "depth": 3,
            "fast_pattern": false,
            "relative_next": false
          }
        }
      ]
    }
  },
  "mpm": {
    "buffer": "payload",
    "pattern": "abc",
    "length": 3,
    "nocase": false,
    "negated": false,
    "starts_with": true,
    "ends_with": false,
    "is_mpm": true,
    "no_double_inspect": false,
    "depth": 3,
    "fast_pattern": false,
    "relative_next": false
  }
}
```


IP Only

The IP ONLY rule type is used when rules match only on source and destination IP addresses, and not on any other flow or content modifier.

Examples

alert tcp-stream any any -> any any (msg:"tcp-stream, no content"; sid:101;)

alert tcp-pkt [192.168.0.0/16,10.0.0.0/8,172.16.0.0/12] any -> any any (msg:"tcp-pkt, no content"; sid:201;)

alert ip any any -> any any (hostbits:set,myflow2; sid:1505;)

alert udp any any -> any any (msg:"UDP with flow direction"; sid:1601;)

Engine-Analysis Report


```
{
  "raw": "alert ip any any -> any any (hostbits:set,myflow2; sid:1505;)",
  "id": 1505,
```


(continues on next page)

(continued from previous page)


```
  "gid": 1,
  "rev": 0,
  "app_proto": "unknown",
  "requirements": [],
  "type": "ip_only",
  "flags": [
    "src_any",
    "dst_any",
    "sp_any",
    "dp_any",
    "toserver",
    "toclient"
  ],
  "pkt_engines": [],
  "frame_engines": [],
  "lists": {
    "postmatch": {
      "matches": [
        {
          "name": "hostbits"
        }
      ]
    }
  }
}
```


IP Only (contains negated address)

A rule that inspects IP only properties, but contains negated IP addresses.

IP Only signatures with negated addresses are *like* IP-only signatures, but currently handled differently due to limitations of the algorithm processing IP Only rules. Impactful differences from a user-perspective are listed on the Signature Types table.

Examples

alert tcp 192.168.0.0/16,10.0.0.0/8,172.16.0.0/12 any -> ![192.168.0.0/16,10.0.0.0/8,172.16.0.0/12] any (msg:"tcp, has negated IP address"; sid:304;)

alert tcp [10.0.0.0/8,!10.10.10.10] any -> [10.0.0.0/8,!10.10.10.10] any (msg:"tcp, has negated IP address"; sid:305;)

Engine-Analysis Report


```
{
  "raw": "alert tcp [10.0.0.0/8,!10.10.10.10] any -> [10.0.0.0/8,!10.10.10.10] any (msg:\
↪"tcp, has negated IP address\"; sid:305;)",
  "id": 305,
  "gid": 1,
  "rev": 0,
  "msg": "tcp, has negated IP address",
  "app_proto": "unknown",
  "requirements": [],
  "type": "like_ip_only",
```


(continues on next page)

(continued from previous page)


```
  "flags": [
    "sp_any",
    "dp_any",
    "toserver",
    "toclient"
  ],
  "pkt_engines": [],
  "frame_engines": [],
  "lists": {}
}
```


Protocol Detection Only

When a signature checks for the application layer protocol but there is no need for a per-packet inspection, protocol detection can be done with the `app-layer-protocol` keyword. Check the keyword documentation full for usage.

See *Protocol Detection Only* for a flowchart representing how the type is defined.

See *Application Layer Protocol* for a packet-based inspection.

> **Warning**
>
> Since Suricata 7, a Protocol Detection rule (that uses the `app-layer-protocol` keyword) is not internally classified the same as a rule simply matching on the application layer protocol on the `protocol` field.

Examples

alert tcp any any -> any any (msg:"tcp, pd negated"; app-layer-protocol:!http; sid:401;)

alert tcp any any -> any any (msg:"tcp, pd positive"; app-layer-protocol:http; sid:402;)

alert tcp any any -> any any (msg:"tcp, pd positive dns"; app-layer-protocol:dns; sid:403;)

alert tcp any any -> any any (msg:"tcp, pd positive, dns, flow:to_server"; app-layer-protocol:dns; flow:to_server; sid:405;)

Engine-Analysis Report


```
{
  "raw": "alert tcp any any -> any any (msg:\"tcp, pd positive dns\"; app-layer-
↪protocol:dns; sid:403;)",
  "id": 403,
  "gid": 1,
  "rev": 0,
  "msg": "tcp, pd positive dns",
  "app_proto": "unknown",
  "requirements": [],
  "type": "pd_only",
  "flags": [
    "src_any",
    "dst_any",
    "sp_any",
    "dp_any",
```


(continues on next page)

(continued from previous page)

```
    "toserver",
    "toclient"
  ],
  "pkt_engines": [
    {
      "name": "packet",
      "is_mpm": false
    }
  ],
  "frame_engines": [],
  "lists": {
    "packet": {
      "matches": [
        {
          "name": "app-layer-protocol"
        }
      ]
    }
  }
}
```

Packet-Stream

A rule is categorized as such when it inspects on traffic in specific portions of the packet payload, using `content` buffer with the `startswith` or `depth` keywords.

Examples

alert tcp any any -> any any (msg:"tcp, anchored content"; content:"abc"; startswith; sid:303;)

alert http any any -> any any (msg:"http, anchored content"; content:"abc"; depth:30; sid:603;)

Engine-Analysis Report

```
{
  "raw": "alert http any any -> any any (msg:\"http, anchored content\"; content:\"abc\";
↪ depth:30; sid:603;)",
  "id": 603,
  "gid": 1,
  "rev": 0,
  "msg": "http, anchored content",
  "app_proto": "http_any",
  "requirements": [
    "payload",
    "flow"
  ],
  "type": "pkt_stream",
  "flags": [
    "src_any",
    "dst_any",
    "sp_any",
    "dp_any",
```

(continues on next page)

(continued from previous page)


```
    "applayer",
    "need_packet",
    "need_stream",
    "toserver",
    "toclient",
    "prefilter"
  ],
  "pkt_engines": [
    {
      "name": "payload",
      "is_mpm": true
    }
  ],
  "frame_engines": [],
  "lists": {
    "payload": {
      "matches": [
        {
          "name": "content",
          "content": {
            "pattern": "abc",
            "length": 3,
            "nocase": false,
            "negated": false,
            "starts_with": false,
            "ends_with": false,
            "is_mpm": true,
            "no_double_inspect": false,
            "depth": 30,
            "fast_pattern": false,
            "relative_next": false
          }
        }
      ]
    }
  },
  "mpm": {
    "buffer": "payload",
    "pattern": "abc",
    "length": 3,
    "nocase": false,
    "negated": false,
    "starts_with": false,
    "ends_with": false,
    "is_mpm": true,
    "no_double_inspect": false,
    "depth": 30,
    "fast_pattern": false,
    "relative_next": false
  }
}
```

Stream

A rule that matches payload traffic without regards to its position, that is, on an unanchored `content` buffer, uses byte extraction or matches on `tcp-stream` is classified a stream rule.

Examples

alert tcp-stream any any -> any any (msg:"tcp-stream, simple content"; content:"abc"; sid:102;)

alert http any any -> any any (msg:"http, simple content"; content:"abc"; sid:602;)

alert tcp any any -> any 443 (flow: to_server; content:"abc"; flowbits:set,tls_error; sid:1605; msg:"Allow TLS error handling (outgoing packet) with simple content - Stream rule";)

alert tcp any any -> any 443 (flow: to_server; content:"abc"; sid:160401; msg:"Allow TLS error handling (outgoing packet) - stream rule";)

alert tcp any any -> any 443 (content:"abc"; sid:160402; msg:"Allow TLS error handling (outgoing packet) - stream rule";)

alert tcp any any -> any any (msg:"byte_extract with dce"; byte_extract:4,0,var,dce; byte_test:4,>,var,4,little; sid:901;)

Engine-Analysis Report


```
{
  "raw": "alert tcp any any -> any any (msg:\"byte_extract with dce\"; byte_extract:4,0,
↪var,dce; byte_test:4,>,var,4,little; sid:901;)",
  "id": 901,
  "gid": 1,
  "rev": 0,
  "msg": "byte_extract with dce",
  "app_proto": "dcerpc",
  "requirements": [
    "payload",
    "flow"
  ],
  "type": "stream",
  "flags": [
    "src_any",
    "dst_any",
    "sp_any",
    "dp_any",
    "applayer",
    "need_stream",
    "toserver",
    "toclient"
  ],
  "pkt_engines": [
    {
      "name": "payload",
      "is_mpm": false
    }
  ],
  "frame_engines": [],
  "lists": {
    "payload": {
```


(continues on next page)

(continued from previous page)

```
      "matches": [
        {
          "name": "byte_extract"
        },
        {
          "name": "byte_test",
          "byte_test": {
            "nbytes": 4,
            "offset": 4,
            "base": "unset",
            "flags": [
              "little_endian"
            ]
          }
        }
      ]
    }
  }
}
```

Application Layer Protocol

For a packet-based inspection of the application layer protocol, a rule should use the protocol field for the matches.

> **Warning**
>
> Since Suricata 7, a simple rule matching traffic on the `protocol` field is not internally classified the same as a rule using the `app-layer-protocol` keyword).

> **Warning**
>
> As per Suricata 7, if `flow:established` or `flow:not_established` is added to a base Application Layer Protocol rule, that signature will become a *Packet* rule.

Examples

alert dns any any -> any any (msg:"app-layer, dns"; sid:404;)

alert http any any -> any any (msg:"http, no content"; sid:601;)

alert tls any any -> any any (msg:"tls, pkt or app-layer?"; flowint:tls_error_int,=,0; sid:613;)

Engine-Analysis Report

```
{
  "raw": "alert dns any any -> any any (msg:\"app-layer, dns\"; sid:404;)",
  "id": 404,
  "gid": 1,
  "rev": 0,
  "msg": "app-layer, dns",
```

(continues on next page)

(continued from previous page)


```
  "app_proto": "dns",
  "requirements": [
    "flow"
  ],
  "type": "app_layer",
  "flags": [
    "src_any",
    "dst_any",
    "sp_any",
    "dp_any",
    "applayer",
    "toserver",
    "toclient"
  ],
  "pkt_engines": [],
  "frame_engines": [],
  "lists": {}
}
```


Application Layer Protocol Transactions

Rules inspecting traffic using keywords related to application layer protocols are classified with this signature type. This also includes *frame* keywords.

Examples

alert tcp any any -> any any (msg:"http, pos event"; app-layer-event:http.file_name_too_long; sid:501;)

alert http any any -> any any (msg:"Test"; flow:established,to_server; http.method; content:"GET"; http.uri; content:".exe"; endswith; http.host; content:!".google.com"; endswith; sid:1102;)

alert udp any any -> any any (msg:"DNS UDP Frame"; flow:to_server; frame:dns.pdu; content:"|01 20 00 01|"; offset:2; content:"suricata"; offset:13; sid:1402; rev:1;)

alert tcp any any -> any any (msg:"byte_extract with dce"; dcerpc.stub_data; content:"abc"; byte_extract:4,0,var,relative; byte_test:4,>,var,4,little; sid:902;)

Engine-Analysis Report


```
{
  "raw": "alert tcp any any -> any any (msg:\"byte_extract with dce\"; dcerpc.stub_data;␣
↪content:\"abc\"; byte_extract:4,0,var,relative; byte_test:4,>,var,4,little; sid:902;)",
  "id": 902,
  "gid": 1,
  "rev": 0,
  "msg": "byte_extract with dce",
  "app_proto": "dcerpc",
  "requirements": [
    "flow"
  ],
  "type": "app_tx",
  "flags": [
    "src_any",
```


(continues on next page)

(continued from previous page)


```
    "dst_any",
    "sp_any",
    "dp_any",
    "applayer",
    "toserver",
    "toclient",
    "prefilter"
  ],
  "pkt_engines": [],
  "frame_engines": [],
  "engines": [
    {
      "name": "dce_stub_data",
      "direction": "toclient",
      "is_mpm": true,
      "app_proto": "dcerpc",
      "progress": 0,
      "matches": [
        {
          "name": "content",
          "content": {
            "pattern": "abc",
            "length": 3,
            "nocase": false,
            "negated": false,
            "starts_with": false,
            "ends_with": false,
            "is_mpm": true,
            "no_double_inspect": false,
            "fast_pattern": false,
            "relative_next": true
          }
        },
        {
          "name": "byte_extract"
        },
        {
          "name": "byte_test",
          "byte_test": {
            "nbytes": 4,
            "offset": 4,
            "base": "unset",
            "flags": [
              "little_endian"
            ]
          }
        }
      ]
    },
    {
      "name": "dce_stub_data",
      "direction": "toserver",
```


(continues on next page)

(continued from previous page)

```
      "is_mpm": true,
      "app_proto": "dcerpc",
      "progress": 0,
      "matches": [
        {
          "name": "content",
          "content": {
            "pattern": "abc",
            "length": 3,
            "nocase": false,
            "negated": false,
            "starts_with": false,
            "ends_with": false,
            "is_mpm": true,
            "no_double_inspect": false,
            "fast_pattern": false,
            "relative_next": true
          }
        },
        {
          "name": "byte_extract"
        },
        {
          "name": "byte_test",
          "byte_test": {
            "nbytes": 4,
            "offset": 4,
            "base": "unset",
            "flags": [
              "little_endian"
            ]
          }
        }
      ]
    },
    {
      "name": "dce_stub_data",
      "direction": "toclient",
      "is_mpm": true,
      "app_proto": "smb",
      "progress": 0,
      "matches": [
        {
          "name": "content",
          "content": {
            "pattern": "abc",
            "length": 3,
            "nocase": false,
            "negated": false,
            "starts_with": false,
            "ends_with": false,
            "is_mpm": true,
```

(continues on next page)

(continued from previous page)


```
            "no_double_inspect": false,
            "fast_pattern": false,
            "relative_next": true
          }
        },
        {
          "name": "byte_extract"
        },
        {
          "name": "byte_test",
          "byte_test": {
            "nbytes": 4,
            "offset": 4,
            "base": "unset",
            "flags": [
              "little_endian"
            ]
          }
        }
      ]
    },
    {
      "name": "dce_stub_data",
      "direction": "toserver",
      "is_mpm": true,
      "app_proto": "smb",
      "progress": 0,
      "matches": [
        {
          "name": "content",
          "content": {
            "pattern": "abc",
            "length": 3,
            "nocase": false,
            "negated": false,
            "starts_with": false,
            "ends_with": false,
            "is_mpm": true,
            "no_double_inspect": false,
            "fast_pattern": false,
            "relative_next": true
          }
        },
        {
          "name": "byte_extract"
        },
        {
          "name": "byte_test",
          "byte_test": {
            "nbytes": 4,
            "offset": 4,
            "base": "unset",
```


(continues on next page)

(continued from previous page)

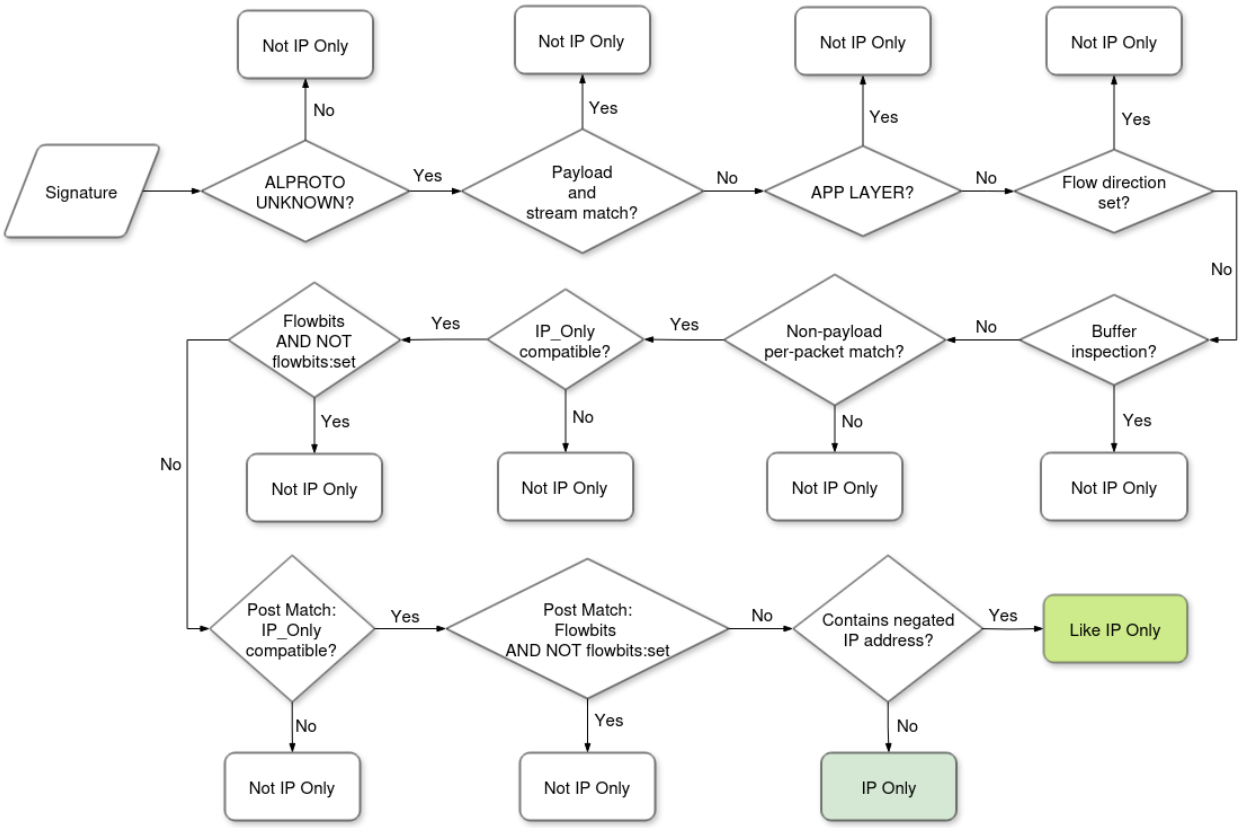

```
            "flags": [
              "little_endian"
            ]
          }
        }
      ]
    }
  ],
  "lists": {},
  "mpm": {
    "buffer": "dce_stub_data",
    "pattern": "abc",
    "length": 3,
    "nocase": false,
    "negated": false,
    "starts_with": false,
    "ends_with": false,
    "is_mpm": true,
    "no_double_inspect": false,
    "fast_pattern": false,
    "relative_next": true
  }
}
```


8.62.4 Detailed Flowcharts

A look into the illustrated overall representation of functions or paths that determine signature types.

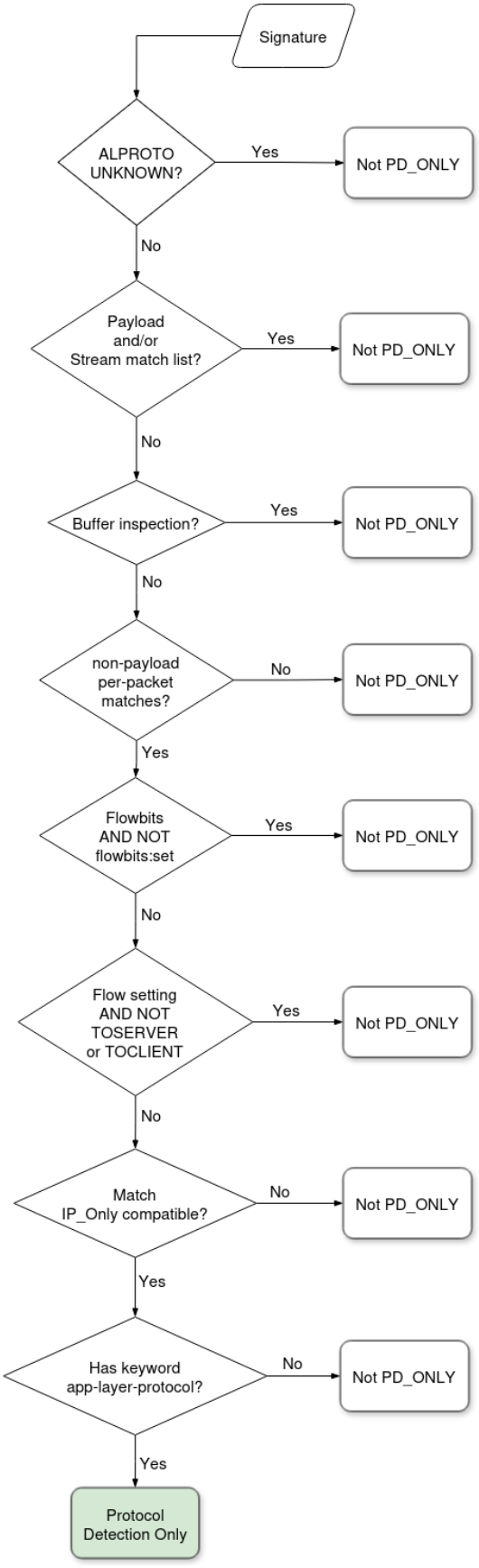
IP Only and IP Only with negated addresses

`ip_only` and `like_ip_only` flows.



Protocol Detection Only

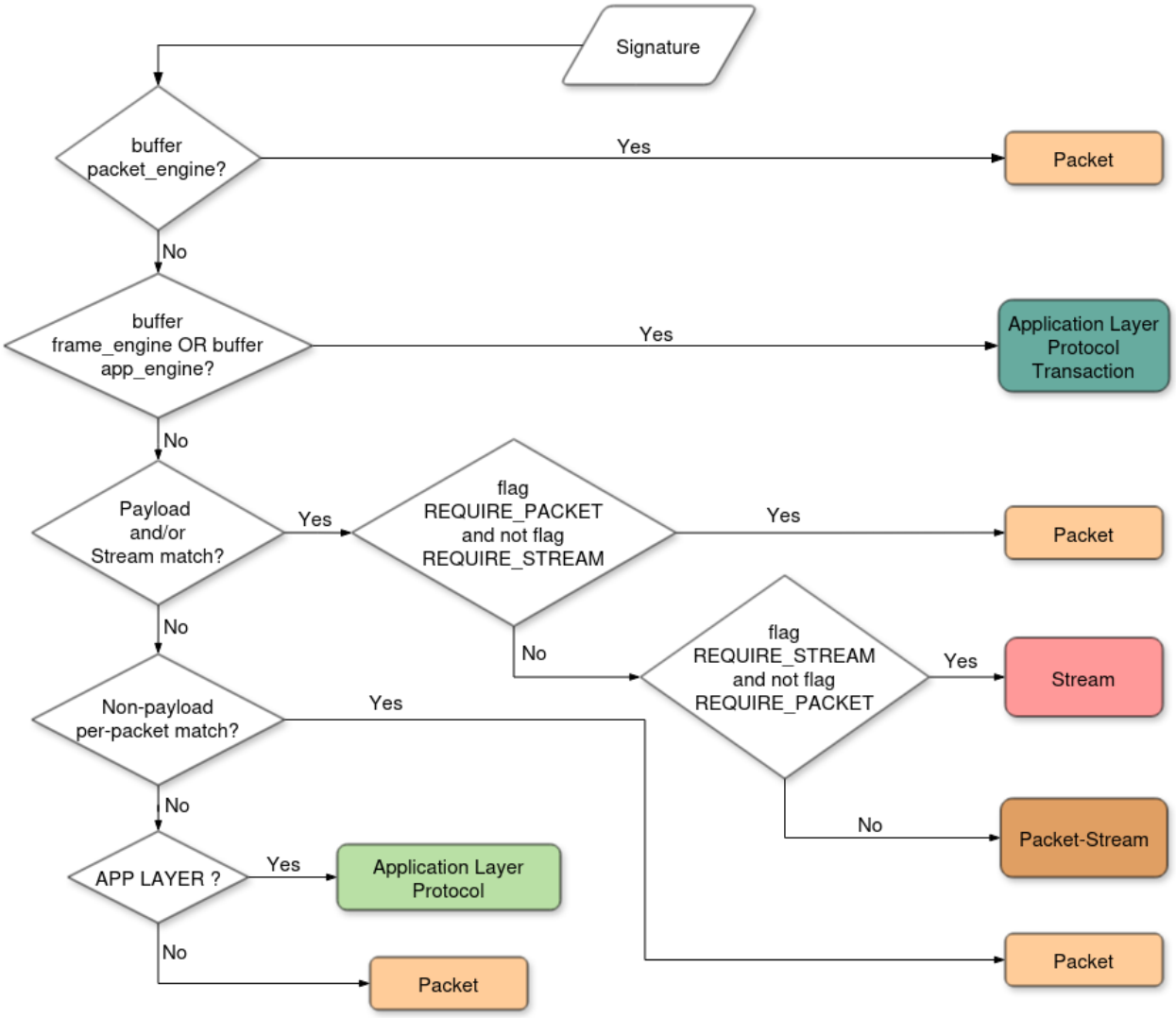
`pd_only` flow.


Signature
ALPROTO UNKNOWN?
Yes
Not PD_ONLY
No
Payload and/or Stream match list?
Yes
Not PD_ONLY
No
Buffer inspection?
Yes
Not PD_ONLY
No
non-payload per-packet matches?
No
Not PD_ONLY
Yes
Flowbits AND NOT flowbits:set
Yes
Not PD_ONLY
No
Flow setting AND NOT TOSERVER or TOCLIENT
Yes
Not PD_ONLY
No
Match IP_Only compatible?
No
Not PD_ONLY
Yes
Has keyword app-layer-protocol?
No
Not PD_ONLY
Yes
Protocol Detection Only

Application Layer Protocol, Transaction, Packet, Stream and Stream-Packet rules

`app_layer`, `app_tx`, `pkt`, `stream` and `stream-pkt` flows.

`REQUIRE_PACKET` and `REQUIRE_STREAM` can be seen as flags `need_packet` and `need_stream` in the `engine-analysis` output.



8.63 Rule Processing

Suricata rules have many elements that influence how they are processed by Suricata and matched against network traffic.

This section explains some key aspects of how Suricata handles rules internally, so it can be easier to understand/predict how different rules may interact in specific scenarios.

This material is intended for: rule writers; developers.

Possible questions one should be better equipped to answer after reading this document:

- What happens if two rules have the same priority value (the keyword)?
- What type of rules will be evaluated first, given a set of rules?

- How does Suricata decide what rule is "more important" when matching traffic?

Important

Rules processing is also heavily affected by rule types, as mentioned in this chapter. You may want to read more on *Rule Types and Categorization*.

Note

Throughout this documentation and for Suricata, the terms "rule" and "signature" are mainly interchangeable, unless context indicates otherwise.

8.63.1 Overview

Rules are provided to Suricata via rules files. Starting from those, the Detection Engine loader will:

1. Load all the signatures;
2. Check for validity (non-existing keywords; duplicated `sid` etc);
3. Report stats on loaded, good and bad signatures;
4. Sort all valid signatures and store them in a list in the Detect Engine Context (`DetectEngineCtx`) structure, taking into consideration several rule aspects according to their order of relevance to the Detection Engine;
5. Attribute internal rule IDs that reflect this *rule prioritization*;
6. During inspection, match rules against the inspected traffic, according to rule and traffic type.

8.63.2 Rule Prioritization

Suricata registers several different ordering functions (with `SCSigRegisterSignatureOrderingFuncs()`), which are then used to compare the rules, sort them, and define their priority. The elements taken into consideration for such are the signature's:

1. *Action*
2. Usage of *flowbits*
3. Usage of *flowint*
4. Usage of flowvar
5. Usage of pktvar
6. Usage of hostbits
7. Usage of ippair
8. *"Priority" keyword*

In this order. Once signatures are ordered, they are attributed a unique internal ID (`Signature::iid`) which symbolizes their priority (the lower the `iid`, the higher the priority). This could mean that a rule with a keyword-defined priority of 1 could have lower priority than another rule that had flowbits set and a rule action with higher priority, for instance.

> **Note**
>
> this list isn't fully comprehensive, in the sense that each item has extra logic for prioritization. For example, considering flowbits, the priority is write (highest) > write + read > read (lowest) > no flowbits.

> **Note**
>
> it is also possible to have a rule priority set implicitly, through the *classtype* keyword. Check *classtype*.

Another important element when considering rule parsing, processing and matching is that the ruleset is optimized into signature group heads based on the signature elements (thus, for instance, a TCP rule and an UDP rule would be loaded into different groups, and their internal ids will not interfere between one another, as they're matched against different traffic). For more on this, see *Detection engine*.

8.63.3 Inspection Process

Once it is time to inspect network traffic against the loaded rules, the Detect Engine will match against - if applicable:

1. IP Only rules;
2. Packet/payload-related rules;
3. Frame keywords;
4. Application layer protocol transaction rules.

During packet inspection, if the signature uses the last two in this list, inspection is left to those steps.

> **Tip**
>
> With the introduction of *Firewall mode*, it is possible to explicitly control to which step of the detection engine flow a rule will be hooked. This is done with *Explicit rule hooks*.

For each rule that is matched, a `PacketAlert` is created. After all matches for a packet have been processed, and the *alert queue limit* is taken into account, the remaining `PacketAlerts` become the alerts in Suricata logs.

8.63.4 Considerations on Inspection Steps

IP Only rules

Without optimization, IP Only signatures would match on every packet on a flow. To improve performance, what Suricata does is to evaluate rules that are `ip-only` only once per flow direction, for the first packet in each direction.

Application layer protocol transactions

Each parser has its own state machine, and uses a per-direction parsing state "progress". Keywords can be registered for each progress value. So, for instance, `http` has a value "request line available" for which there are keywords like *http.uri*, *http.method* etc. registered. While parsing the traffic, if the engine reaches this state, the signatures with those keywords may be already evaluated, even if they have a lower priority than an http body inspecting signature.

Relatedly, a rule with two keywords matching at two different progress stages may be evaluated against two different packets.

8.63.5 Implications

Action precedence and interaction with `ip-only` rules

To illustrate what may be counter-intuitive implications of how inspection steps, action prioritization and rule keywords interact and affect the engine behavior, we will use a real case example. Consider these three rules:

pass tcp 0.0.0.0/0 any <> 0.0.0.0/0 443 (msg:"Allow TCP in port 443"; flow: not_established; sid:1; rev:1;)

pass tcp 0.0.0.0/0 any <> 0.0.0.0/0 80 (msg:"Allow TCP in port 80"; flow: not_established; sid:2; rev:1;)

drop ip 0.0.0.0/0 any -> 0.0.0.0/0 any (msg:"No outbound internet access from host"; sid:3; rev:1;)

The first two are signatures that analyze individual packets and match only if the flow has not been established (`flow:not_established`): the rules grant `PASS` to the matched packet - but not to its flow.

The third signature is considered `ip-only`. This means it will be evaluated for the *first* packet in both directions of a flow, in addition to rules 1 and 2. By extension, **the other packets in the same flow will not be evaluated against this rule.**

With an action order configuration that prioritizes `PASS` over `DROP`, this means that rules 1 and 2 will have a higher internal priority over rule 3, therefore nullifying the `DROP` outcome. The result: a flow for outbound internet traffic from the host, expected to be dropped, wouldn't be.

If the expected behavior with those three signatures was to allow traffic on ports 80 and 443 only, while dropping everything else, the simplest way to achieve this would be to remove the `flow:not_established` portion from rules sid:1 and sid:2. This ensures that the `PASS` action would be applied to the whole flow following the match on the first packet and that all other traffic would be dropped.

Following that, all three rules will be evaluated on the same step, and if a flow isn't flagged with `pass`, it will be dropped with the third rule.

> **Tip**
>
> A more straightforward way to achieve that in Suricata 8 is using the firewall mode. See *Firewall Mode Design*.

Alerts not seen

Another aspect of rule prioritization combined with the alerts queue size is that, in corner case scenarios, if a packet matches against too many rules, signatures with lower priority could be discarded from the `PacketAlert` queue (see the section on *alert queue overflow impact* for more).

The stats counter `detect.alert_queue_overflow` will be higher than zero if an alert was discarded due to `Alert Queue` overflow (cf. *Discarded and Suppressed Alerts Stats*).

CHAPTER

NINE

RULE MANAGEMENT

9.1 Rule Management with Suricata-Update

While it is possible to download and install rules manually, it is recommended to use a management tool for this. `suricata-update` is the official way to update and manage rules for Suricata.

`suricata-update` is bundled with Suricata and is normally installed with it. For instructions on installing manually, see http://suricata-update.readthedocs.io/en/latest/quickstart.html#install-suricata-update

> **Note**
>
> `suricata-update` is bundled with Suricata version 4.1 and later. It can be used with older versions as well. It will have to be installed separately in that case.

To download the Emerging Threats Open ruleset, it is enough to simply run:

```
sudo suricata-update
```

This will download the ruleset into */var/lib/suricata/rules/*

Suricata's configuration will have to be updated to have a rules config like this:

```
default-rule-path: /var/lib/suricata/rules
rule-files:
  - suricata.rules
```

Now (re)start Suricata.

9.1.1 Updating your rules

To update the rules, simply run

```
sudo suricata-update
```

It is recommended to update your rules frequently.

9.1.2 Using other rulesets

Suricata-Update is capable of making other rulesets accessible as well.

To see what is available, fetch the master index from the OISF hosts:

```
sudo suricata-update update-sources
```

Then have a look at what is available:

```
sudo suricata-update list-sources
```

This will give a result similar to

```
Name: oisf/trafficid
  Vendor: OISF
  Summary: Suricata Traffic ID ruleset
  License: MIT
Name: ptresearch/attackdetection
  Vendor: Positive Technologies
  Summary: Positive Technologies Attack Detection Team ruleset
  License: Custom
Name: sslbl/ssl-fp-blacklist
  Vendor: Abuse.ch
  Summary: Abuse.ch SSL Blacklist
  License: Non-Commercial
Name: et/open
  Vendor: Proofpoint
  Summary: Emerging Threats Open Ruleset
  License: MIT
Name: scwx/security
  Vendor: Secureworks
  Summary: Secureworks suricata-security ruleset.
  License: Commercial
  Parameters: secret-code
  Subscription: https://www.secureworks.com/contact/ (Please reference CTU Countermeasures)
Name: scwx/malware
  Vendor: Secureworks
  Summary: Secureworks suricata-malware ruleset.
  License: Commercial
  Parameters: secret-code
  Subscription: https://www.secureworks.com/contact/ (Please reference CTU Countermeasures)
Name: et/pro
  Vendor: Proofpoint
  Summary: Emerging Threats Pro Ruleset
  License: Commercial
  Replaces: et/open
  Parameters: secret-code
  Subscription: https://www.proofpoint.com/us/threat-insight/et-pro-ruleset
```

Each of the rulesets has a name that has a 'vendor' prefix, followed by a set name. For example, OISF's traffic id ruleset is called 'oisf/trafficid'.

To enable 'oisf/trafficid', enter:

```
sudo suricata-update enable-source oisf/trafficid
sudo suricata-update
```

Now restart Suricata again and the rules from the OISF TrafficID ruleset are loaded.

To see which rulesets are currently active, use "list-enabled-sources".

9.1.3 Controlling which rules are used

By default `suricata-update` will merge all rules into a single file "/var/lib/suricata/rules/suricata.rules".

To enable rules that are disabled by default, use */etc/suricata/enable.conf*

```
2019401                   # enable signature with this sid
group:emerging-icmp.rules # enable this rulefile
re:trojan                 # enable all rules with this string
```

Similarly, to disable rules use */etc/suricata/disable.conf*:

```
2019401                   # disable signature with this sid
group:emerging-info.rules # disable this rulefile
re:heartbleed             # disable all rules with this string
```

After updating these files, rerun `suricata-update` again:

```
sudo suricata-update
```

Finally restart Suricata.

9.1.4 Further reading

See https://suricata-update.readthedocs.io/en/latest/

9.2 Adding Your Own Rules

If you would like to create a rule yourself and use it with Suricata, this guide might be helpful.

Start creating a file for your rule. Use one of the following examples in your console/terminal window:

```
sudo nano local.rules
sudo vim local.rules
```

Write your rule, see *Rules Format* and save it.

> **Tip**
>
> To ensure that there are no duplicated SIDs, check rule SID ranges read more about *sid* at *sid (signature ID)*.

Update the Suricata configuration file so your rule is included. Use one of the following examples:

```
sudo nano /etc/suricata/suricata.yaml
sudo vim /etc/suricata/suricata.yaml
```

and make sure your local.rules file is added to the list of rules:

```
default-rule-path: /usr/local/etc/suricata/rules

rule-files:
  - suricata.rules
  - /path/to/local.rules
```

Now, run Suricata and see if your rule is being loaded.

```
suricata -c /etc/suricata/suricata.yaml -i wlan0
```

If the rule failed to load, Suricata will display as much information as it has when it deemed the rule un-loadable. Pay special attention to the details: look for mistakes in special characters, spaces, capital characters, etc.

Next, check if your log-files are enabled in the Suricata configuration file `suricata.yaml`.

If you had to correct your rule and/or modify Suricata's YAML configuration file, you'll have to restart Suricata.

If you see your rule is successfully loaded, you can double check your rule by doing something that should trigger it.

By default, Suricata will log alerts to two places

- `eve.json`
- `fast.log`

These files will be located in the log output directory which is set by one of two methods:

1. Suricata configuration file: see `default-log-dir` for the name of the directory
2. Suricata command line: Using `-l /path/to/log-dir` creates log files in the named directory.

The following example assumes that the log directory is named `/var/log/suricata`

```
tail -f /var/log/suricata/fast.log
```

If you would make a rule like this:

```
alert http any any -> any any (msg:"Do not read gossip during work";
content:"Scarlett"; nocase; classtype:policy-violation; sid:1; rev:1;)
```

Your alert should look like this:

```
09/15/2011-16:50:27.725288  [**] [1:1:1] Do not read gossip during work [**]
[Classification: Potential Corporate Privacy Violation] [Priority: 1] {TCP} 192.168.0.
↪32:55604 -> 68.67.185.210:80
```

9.3 Rule Reloads

Suricata was designed to reload rules while it is actively processing network traffic to minimize service disruption.

Suricata must be administratively directed to reload rules while it is running.

It is also possible to get information about the last reload via dedicated commands. See *Commands in standard running mode* for more information.

9.3.1 Reload Triggers

There are multiple ways to trigger a rule reload. `suricatasc` is a program distributed with Suricata that provides client-side services, including the ability to trigger a Suricata rule reload..

Via process signal

The `USR2` signal will cause Suricata to start a rule reload. The signal can be sent from the command line or from a script/program. Escalation of privileges may be necessary to send the signal.

```
$ kill -USR2 $(pidof suricata)
```

Via the UNIX domain socket

The `suricatasc` program has two commands to initiate a Suricata rule reload.

Blocking reload

This will cause Suricata to reload rules while the caller blocks, or waits.

```
suricatasc -c reload-rules
```

Non-blocking reload

This will cause Suricata to reload rules without the caller blocking or waiting.

```
suricatasc -c ruleset-reload-nonblocking
```

Via Windows service control

Windows does not support the USR2 signal or the UNIX domain socket. Instead, when Suricata is running as a Windows service, a rule reload can be requested by sending the service the user-defined control code `128`:

```
sc.exe control Suricata 128
```

This is a non-blocking reload. `sc.exe` does not report the result of the reload; it is logged by Suricata.

When installed with `--service-install`, only SYSTEM and Administrators may send control codes to the Suricata service. A service installed by older versions of Suricata, or by other tools, may allow any interactive user to send control codes. This can be restricted with:

```
sc.exe sdset Suricata "D:(A;;CCLCSWRPWPDTLOCRRC;;;SY)(A;;CCDCLCSWRPWPDTLOCRSDRCWDWO;;;
↪BA)(A;;CCLCSWLORC;;;IU)(A;;CCLCSWLORC;;;SU)"
```

Note

In PowerShell, `sc` is an alias for `Set-Content`, so `sc.exe` must be used.

9.3.2 Resources Reloaded

There are two types of resources that are reloaded during a rule reload.

- Rule-related configuration:
  - Suricata's configuration file(s): `suricata.yaml` and any specified with the command-line options `--include <config-file.yaml>`. Only rule-related information is reloaded.
    - Rule variables: items in the `vars` section.
    - Rule files from the `rule-files` section (if the `-S` command line option was not used)
  - Ancillary rule-related configuration files: `classification.config`, `reference.config` and `threshold.config`
  - Dataset(s) used by rules.
  - When multi-tenants are configured, rule-related configuration information for each tenant.

9.3.3 When to reload rules

Rule reloads are used in situations when:

- Rules have been changed since the last reload. Vendors often add rules frequently and sometimes update existing rules. Rules should be reloaded according to a security policy that includes Suricata rule and configuration settings.

- Rule variables have been changed. Rule reloads will use rule variables from the Suricata configuration file. When updating these, reload the rules in order for the updated rule variables to take effect.
- Ancillary rule-related configuration files are updated.

9.3.4 Advanced: Rule Reload Steps

When reloading rules, Suricata executes the following steps to ensure a safe and consistent update:

- The main Suricata configuration is reloaded to update rule variables and values, including the rule related files `classification.config`, `reference.config` and `theshold.config`.
- All rule files are reloaded with new rule variables applied.
- A new detection engine is created for the updated rules.
- The previous and newly created detection engines are swapped.
- Ensure all threads are updated.
- Free old detection engine and associated resources.

Suricata will continue to process packets during the update process. Note that additional system memory is used during the reload process as a new detection engine and the reloaded rules are associated with it.

9.4 Rules Profiling

If Suricata is built with the *--enable-profiling-rules* then the ruleset profiling can be activated on demand from the unix socket and dumped from it.

To start profiling

```
suricatasc -c ruleset-profile-start
```

To stop profiling

```
suricatasc -c ruleset-profile-stop
```

To dump profiling

```
suricatasc -c ruleset-profile
```

A typical scenario to get rules performance would be

```
suricatasc -c ruleset-profile-start
sleep 30
suricatasc -c ruleset-profile-stop
suricatasc -c ruleset-profile
```

On busy systems, using the sampling capability to capture performance on a subset of packets can be obtained via the *sample-rate* variable in the *profiling* section in the *suricata.yaml* file.

CHAPTER

TEN

MAKING SENSE OUT OF ALERTS

When an alert happens it's important to figure out what it means. Is it serious? Relevant? A false positive?

To find out more about the rule that fired, it's always a good idea to look at the actual rule.

The first thing to look at in a rule is the description that follows the `msg` keyword. Let's consider an example:

```
msg:"ET SCAN sipscan probe";
```

The "ET" indicates the rule came from the Emerging Threats (Proofpoint) project. "SCAN" indicates the purpose of the rule is to match on some form of scanning. Following that, a more or less detailed description is given.

Most rules contain some pointers to more information in the form of the "reference" keyword.

Consider the following example rule:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS \
  (msg:"ET CURRENT_EVENTS Adobe 0day Shovelware"; \
  flow:established,to_server; content:"GET "; nocase; depth:4; \
  content:!"|0d 0a|Referer\:"; nocase; \
  uricontent:"/ppp/listdir.php?dir="; \
  pcre:"/\/[a-z]{2}\/[a-z]{4}01\/ppp\/listdir\.php\?dir=/U"; \
  classtype:trojan-activity; \
  reference:url,isc.sans.org/diary.html?storyid=7747; \
  reference:url,doc.emergingthreats.net/2010496; \
  reference:url,www.emergingthreats.net/cgi-bin/cvsweb.cgi/sigs/CURRENT_EVENTS/CURRENT_
↪Adobe; \
  sid:2010496; rev:2;)
```

In this rule, the reference keyword indicates 3 urls to visit for more information:

```
isc.sans.org/diary.html?storyid=7747
doc.emergingthreats.net/2010496
www.emergingthreats.net/cgi-bin/cvsweb.cgi/sigs/CURRENT_EVENTS/CURRENT_Adobe
```

Some rules contain a reference like: `"reference:cve,2009-3958;"` should allow you to find info about the specific CVE using your favorite search engine.

It's not always straight forward and sometimes not all of that information is available publicly. Usually asking about it on the signature support channel can be helpful.

In *Rule Management with Suricata-Update* more information on the rule sources and their documentation and support methods can be found.

In many cases, looking at just the alert and the packet that triggered it won't be enough to be conclusive. When using the default Eve settings a lot of metadata will be added to the alert.

For example, if a rule fired that indicates your web application is attacked, looking at the metadata might reveal that the web application replied with `404 not found`. This will usually mean the attack failed but not always.

Not every protocol leads to metadata generation, so when running an IDS engine like Suricata, it's often recommended to combine it with full packet capture. Using tools like Evebox, Sguil or Snorby, the full TCP session or UDP flow can be inspected.

Obviously there is a lot more to Incidence Response, but this should get you started.

CHAPTER

ELEVEN

PERFORMANCE

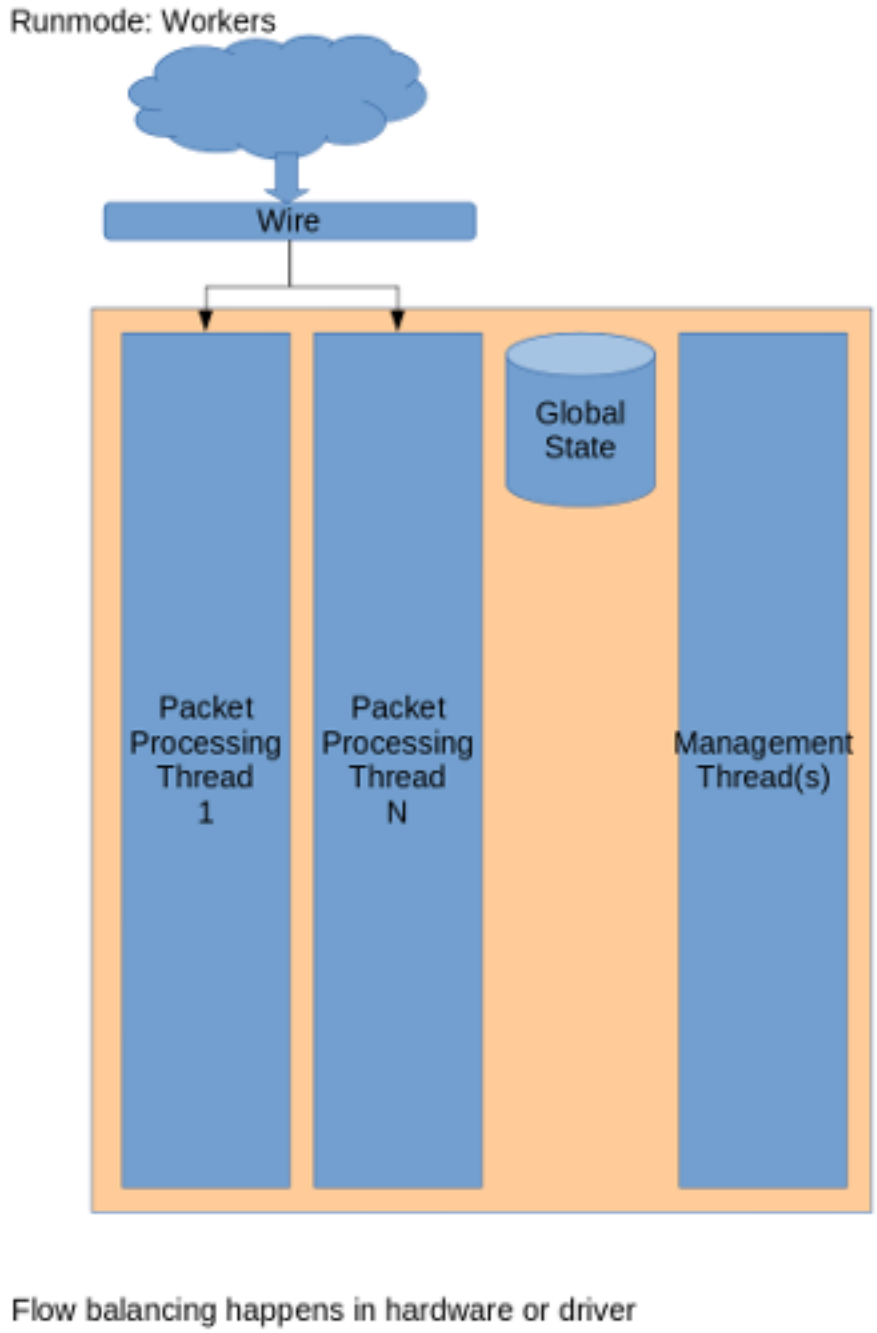
11.1 Runmodes

Suricata consists of several 'building blocks' called threads, thread-modules and queues. A thread is like a process that runs on a computer. Suricata is multi-threaded, so multiple threads are active at once. A thread-module is a part of a functionality. One module is for example for decoding a packet, another is the detect-module and another one the output-module. A packet can be processed by more than one thread. The packet will then be passed on to the next thread through a queue. Packets will be processed by one thread at a time, but there can be multiple packets being processed at a time by the engine (see *Max-pending-packets*). A thread can have one or more thread-modules. If they have more modules, they can only be active one at a time. The way threads, modules and queues are arranged together is called the "Runmode".

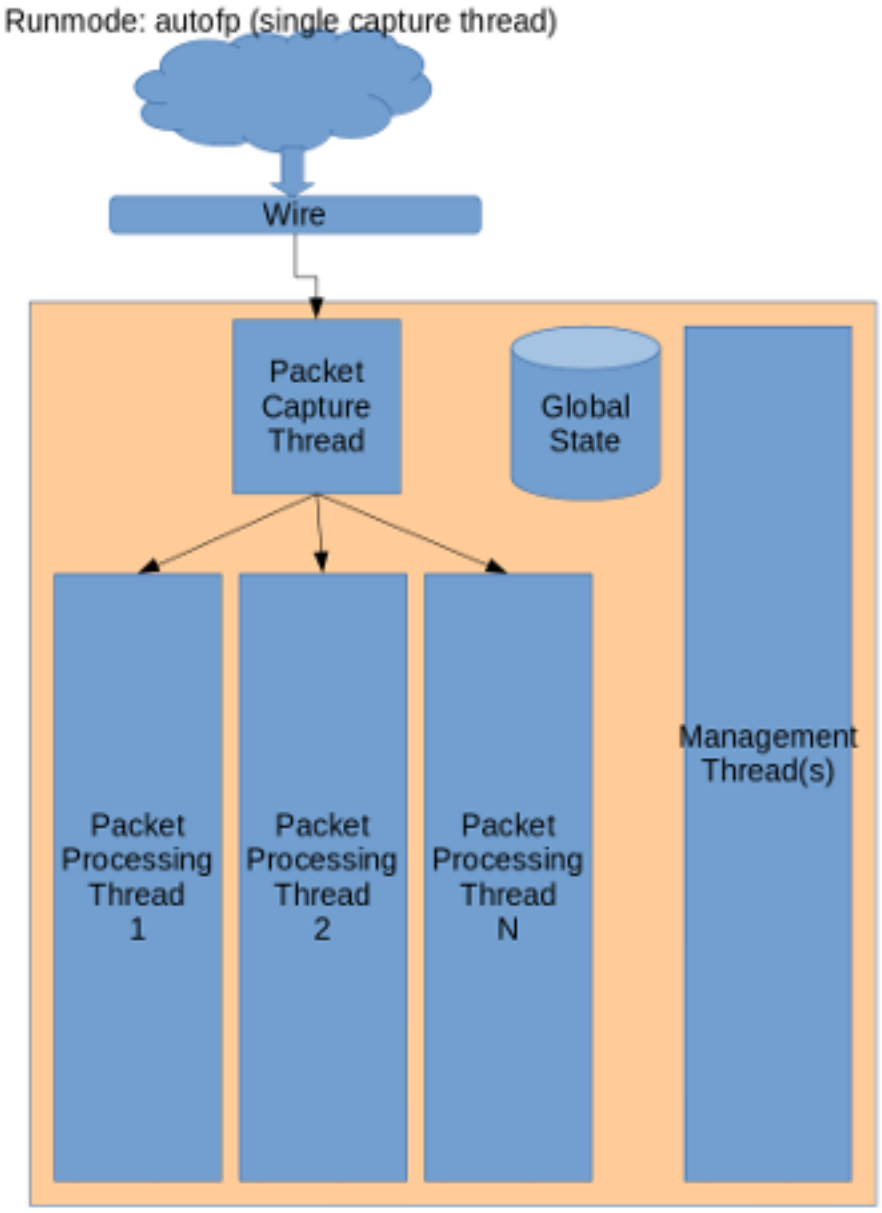
11.1.1 Different runmodes

You can choose a runmode out of several predefined runmodes. The command line option `--list-runmodes` shows all available runmodes. All runmodes have a name: single, workers, autofp.

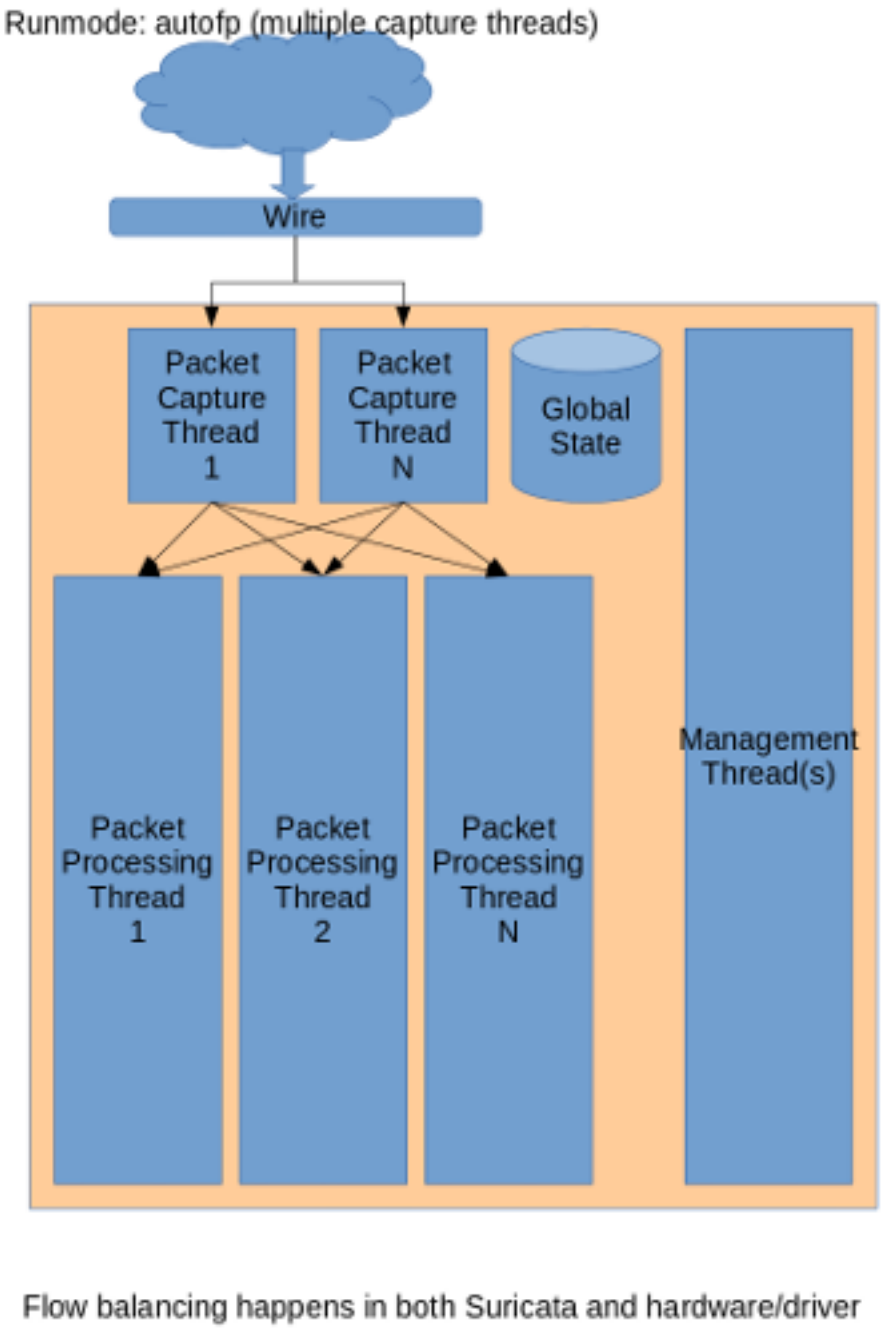
Generally, the `workers` runmode performs the best. In this mode the NIC/driver makes sure packets are properly balanced over Suricata's processing threads. Each packet processing thread then contains the full packet pipeline.



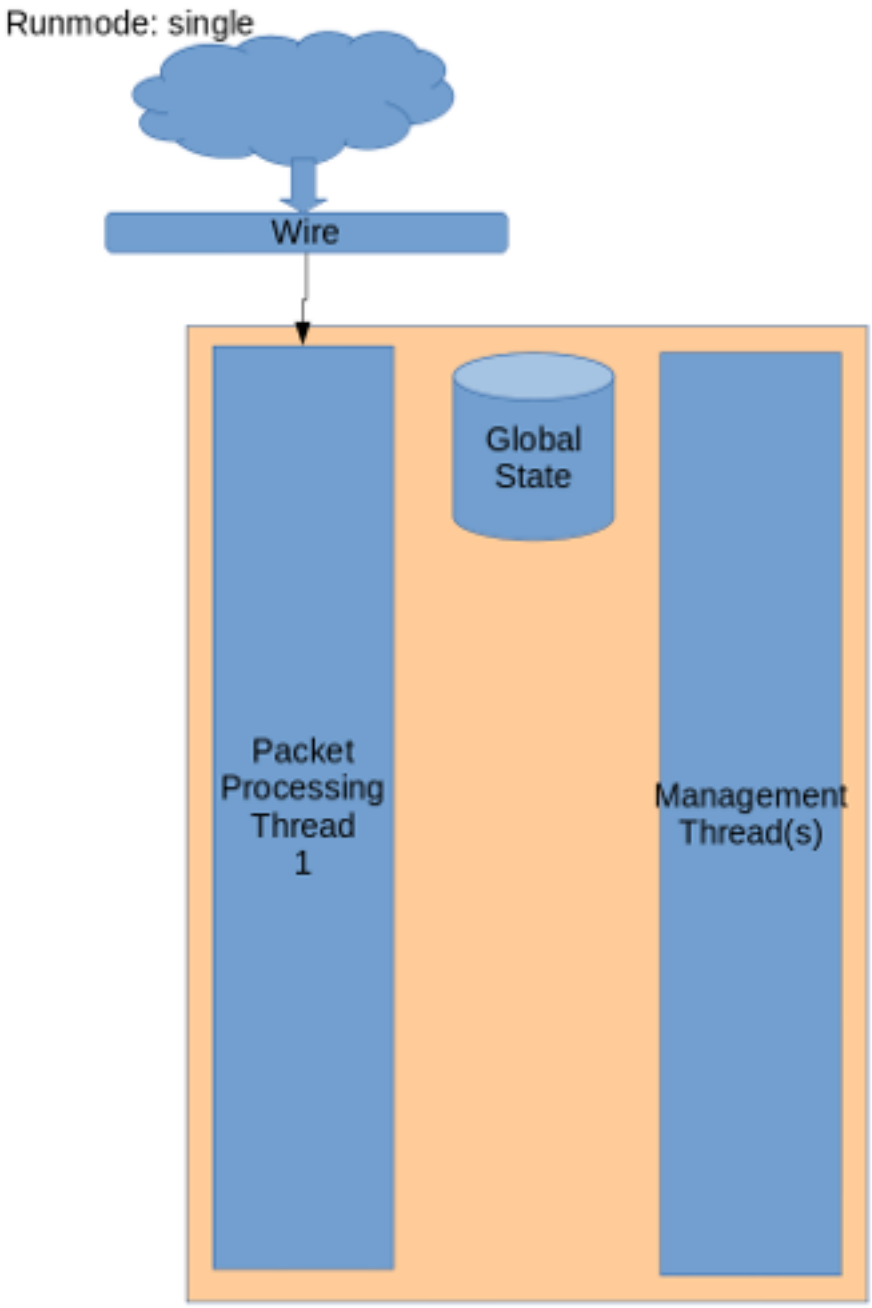
For processing PCAP files, or in case of certain IPS setups (like NFQ), `autofp` is used. Here there are one or more capture threads, that capture the packet and do the packet decoding, after which it is passed on to the `flow worker` threads.



Flow balancing happens inside Suricata



Finally, the `single` runmode is the same as the `workers` mode, however there is only a single packet processing thread. This is mostly useful during development.



For more information about the command line options concerning the runmode, see *Command Line Options*.

11.1.2 Load balancing

Suricata may use different ways to load balance the packets to process between different threads with the configuration option *autofp-scheduler*.

The default value is *hash*, which means the packet is assigned to threads using the 5-7 tuple hash, which is also used anyways to store the flows in memory.

This option can also be set to - *ippair* : packets are assigned to threads using addresses only. - *ftp-hash* : same as *hash* except for flows that may be ftp or ftp-data so that these flows get processed by the same thread. Like so, there is no

concurrency issue in recognizing ftp-data flows due to processing them before the ftp flow got processed. In case of such a flow, a variant of the hash is used.

11.2 Packet Capture

11.2.1 Load balancing

To get the best performance, Suricata will need to run in 'workers' mode. This effectively means that there are multiple threads, each running a full packet pipeline and each receiving packets from the capture method. This means that we rely on the capture method to distribute the packets over the various threads. One critical aspect of this is that Suricata needs to get both sides of a flow in the same thread, in the correct order.

The AF_PACKET and PF_RING capture methods both have options to select the 'cluster-type'. These default to 'cluster_flow' which instructs the capture method to hash by flow (5 tuple). This hash is symmetric. Netmap does not have a cluster_flow mode built-in. It can be added separately by using the "'lb' tool":https://github.com/luigirizzo/netmap/tree/master/apps/lb

On multi-queue NICs, which is almost any modern NIC, RSS settings need to be considered.

11.2.2 RSS

Receive Side Scaling is a technique used by network cards to distribute incoming traffic over various queues on the NIC. This is meant to improve performance but it is important to realize that it was designed for normal traffic, not for the IDS packet capture scenario. RSS using a hash algorithm to distribute the incoming traffic over the various queues. This hash is normally *not* symmetrical. This means that when receiving both sides of a flow, each side may end up in a different queue. Sadly, when deploying Suricata, this is the common scenario when using span ports or taps.

The problem here is that by having both sides of the traffic in different queues, the order of processing of packets becomes unpredictable. Timing differences on the NIC, the driver, the kernel and in Suricata will lead to a high chance of packets coming in at a different order than on the wire. This is specifically about a mismatch between the two traffic directions. For example, Suricata tracks the TCP 3-way handshake. Due to this timing issue, the SYN/ACK may only be received by Suricata long after the client to server side has already started sending data. Suricata would see this traffic as invalid.

None of the supported capture methods like AF_PACKET, PF_RING or NETMAP can fix this problem for us. It would require buffering and packet reordering which is expensive.

To see how many queues are configured:

```
$ ethtool -l ens2f1
Channel parameters for ens2f1:
Pre-set maximums:
RX:             0
TX:             0
Other:          1
Combined:       64
Current hardware settings:
RX:             0
TX:             0
Other:          1
Combined:       8
```

Some NIC's allow you to set it into a symmetric mode. The Intel X(L)710 card can do this in theory, but the drivers aren't capable of enabling this yet (work is underway to try to address this). Another way to address is by setting a special "Random Secret Key" that will make the RSS symmetrical. See http://www.ndsl.kaist.edu/~kyoungsoo/papers/TR-symRSS.pdf (PDF).

In most scenario's however, the optimal solution is to reduce the number of RSS queues to 1:

Example:

```
# Intel X710 with i40e driver:
ethtool -L $DEV combined 1
```

Some drivers do not support setting the number of queues through ethtool. In some cases there is a module load time option. Read the driver docs for the specifics.

11.2.3 Offloading

Network cards, drivers and the kernel itself have various techniques to speed up packet handling. Generally these will all have to be disabled.

LRO/GRO lead to merging various smaller packets into big 'super packets'. These will need to be disabled as they break the dsize keyword as well as TCP state tracking.

Checksum offloading can be left enabled on AF_PACKET and PF_RING, but needs to be disabled on PCAP, NETMAP and others.

11.2.4 Recommendations

Read your drivers documentation! E.g. for i40e the ethtool change of RSS queues may lead to kernel panics if done wrong.

Generic: set RSS queues to 1 or make sure RSS hashing is symmetric. Disable NIC offloading.

AF_PACKET: 1 RSS queue and stay on kernel <=4.2 or make sure you have >=4.4.16, >=4.6.5 or >=4.7. Exception: if RSS is symmetric cluster-type 'cluster_qm' can be used to bind Suricata to the RSS queues. Disable NIC offloading except the rx/tx csum.

PF_RING: 1 RSS queue and use cluster-type 'cluster_flow'. Disable NIC offloading except the rx/tx csum.

NETMAP: 1 RSS queue. There is no flow based load balancing built-in, but the 'lb' tool can be helpful. Another option is to use the 'autofp' runmode. Exception: if RSS is symmetric, load balancing is based on the RSS hash and multiple RSS queues can be used. Disable all NIC offloading.

11.3 Tuning Considerations

Settings to check for optimal performance.

11.3.1 max-pending-packets: <number>

This setting controls the number simultaneous packets that the engine can handle. Setting this higher generally keeps the threads more busy, but setting it too high will lead to degradation.

Suggested setting: 10000 or higher. Max is ~65000. This setting is per thread. The memory is set up at start and the usage is as follows:

```
number_of.threads X max-pending-packets X (default-packet-size + ~750 bytes)
```

11.3.2 mpm-algo: <ac|hs|ac-ks>

Controls the pattern matcher algorithm. AC (`Aho-Corasick`) is the default. On supported platforms, *Hyperscan* is the best option. On commodity hardware if Hyperscan is not available the suggested setting is `mpm-algo: ac-ks` (`Aho-Corasick` Ken Steele variant) as it performs better than `mpm-algo: ac`

11.3.3 detect.profile: <low|medium|high|custom>

The detection engine tries to split out separate signatures into groups so that a packet is only inspected against signatures that can actually match. As in large rule set this would result in way too many groups and memory usage similar groups are merged together. The profile setting controls how aggressive this merging is done. The default setting of `high` usually is good enough.

The "custom" setting allows modification of the group sizes:

```
custom-values:
  toclient-groups: 100
  toserver-groups: 100
```

In general, increasing will improve performance. It will lead to minimal increase in memory usage. The default value for `toclient-groups` and `toserver-groups` with `detect.profile: high` is 75.

11.3.4 detect.sgh-mpm-context: <auto|single|full>

The multi pattern matcher can have it's context per signature group (full) or globally (single). Auto selects between single and full based on the **mpm-algo** selected. ac, ac-ks, hs default to "single". Setting this to "full" with `mpm-algo: ac` or `mpm-algo: ac-ks` offers better performance. Setting this to "full" with `mpm-algo: hs` is not recommended as it leads to much higher startup time. Instead with Hyperscan either `detect.profile: high` or bigger custom group size settings can be used as explained above which offers better performance than `ac` and `ac-ks` even with `detect.sgh-mpm-context: full`.

11.3.5 af-packet

If using `af-packet` (default on Linux) it is recommended that af-packet v3 is used for IDS/NSM deployments. For IPS it is recommended af-packet v2. To make sure af-packet v3 is used it can specifically be enforced it in the `af-packet` config section of suricata.yaml like so:

```
af-packet:
 - interface: eth0
   ....
   ....
   ....
   tpacket-v3: yes
```

11.3.6 ring-size

Ring-size is another `af-packet` variable that can be considered for tuning and performance benefits. It basically means the buffer size for packets per thread. So if the setting is `ring-size: 100000` like below:

```
af-packet:
 - interface: eth0
   threads: 5
   ring-size: 100000
```

it means there will be 100,000 packets allowed in each buffer of the 5 threads. If any of the buffers gets filled (for example packet processing can not keep up) that will result in packet `drop` counters increasing in the stats logs.

The memory used for those is set up and dedicated at start and is calculated as follows:

```
af-packet.threads X af-packet.ring-size X (default-packet-size + ~750 bytes)
```

where `af-packet.threads`, `af-packet.ring-size`, `default-packet-size` are the values set in suricata.yaml. Config values for example for af-packet could be quickly displayed with on the command line as well with `suricata --dump-config |grep af-packet`.

11.3.7 stream.bypass

Another option that can be used to improve performance is `stream.bypass`. In the example below:

```
stream:
 memcap: 64mb
 checksum-validation: yes      # reject wrong csums
 inline: auto                  # auto will use inline mode in IPS mode, yes or no set it␣
↪statically
 bypass: yes
 reassembly:
   memcap: 256mb
   depth: 1mb                  # reassemble 1mb into a stream
   toserver-chunk-size: 2560
   toclient-chunk-size: 2560
   randomize-chunk-size: yes
```

Inspection will be skipped when `stream.reassembly.depth` of 1mb is reached for a particular flow.

11.4 Hyperscan

11.4.1 Introduction

"Hyperscan is a high performance regular expression matching library (...)" (https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-hyperscan.html)

In Suricata it can be used to perform multi pattern matching (mpm) or single pattern matching (spm).

Support for hyperscan in Suricata was initially implemented by Justin Viiret and Jim Xu from Intel via https://github.com/OISF/suricata/pull/1965.

Hyperscan is only for Intel x86 based processor architectures at this time. For ARM processors, vectorscan is a drop in replacement for hyperscan, https://github.com/VectorCamp/vectorscan.

11.4.2 Basic Installation (Package)

Some Linux distributions include hyperscan in their respective package collections.

Fedora 37+/Centos 8+: sudo dnf install hyperscan-devel Ubuntu/Debian: sudo apt-get install libhyperscan-dev

11.4.3 Advanced Installation (Source)

Hyperscan has the following dependencies in order to build from source:

- boost development libraries (minimum boost library version is 1.58)
- cmake
- C++ compiler (e.g. gcc-c++)
- libpcap development libraries
- pcre2 development libraries
- python3

- ragel
- sqlite development libraries

Note: git is an additional dependency if cloning the hyperscan GitHub repository. Otherwise downloading the hyperscan zip from the GitHub repository will work too.

The steps to build and install hyperscan are:

```
git clone https://github.com/intel/hyperscan
cd hyperscan
cmake -DBUILD_STATIC_AND_SHARED=1
cmake --build ./
sudo cmake --install ./
```

Note: Hyperscan can take a long time to build/compile.

Note: It may be necessary to add /usr/local/lib or /usr/local/lib64 to the *ld* search path. Typically this is done by adding a file under /etc/ld.so.conf.d/ with the contents of the directory location of libhs.so.5 (for hyperscan 5.x).

11.4.4 Using Hyperscan

Confirm that the suricata version installed has hyperscan enabled.

```
suricata --build-info | grep Hyperscan
  Hyperscan support:                       yes
```

To use hyperscan support, edit the suricata.yaml. Change the mpm-algo and spm-algo values to 'hs'.

Alternatively, use this command-line option: --set mpm-algo=hs --set spm-algo=hs

Note: The default suricata.yaml configuration settings for mpm-algo and spm-algo are "auto". Suricata will use hyperscan if it is present on the system in case of the "auto" setting.

If the current suricata installation does not have hyperscan support, refer to *Installation*

11.4.5 Hyperscan caching

Upon startup, Hyperscan compiles and optimizes the ruleset into its own internal structure. Suricata optimizes the startup process by saving the Hyperscan internal structures to disk and loading them on the next start. This prevents the recompilation of the ruleset and results in faster initialization. If the ruleset is changed, new necessary cache files are automatically created.

To enable this function, in *suricata.yaml* configure:

```
detect:
  # Cache MPM contexts to the disk to avoid rule compilation at the startup.
  # Cache files are created in the standard library directory.
  sgh-mpm-caching: yes
  sgh-mpm-caching-path: /var/lib/suricata/cache/hs
```

To avoid cache files growing indefinitely, Suricata supports pruning of old cache files. Suricata removes cache files older than the specified age on startup/rule reloads, where age is determined by delta of the file modification time and the current time. Cache files that are actively being used will have their modification time updated when loaded, so they won't be deleted.

In *suricata.yaml* configure:

```
detect:
  sgh-mpm-caching-max-age: 7d
```

The setting accepts a combination of time units (s,m,h,d,w,y), e.g. *1w 3d 12h* for 1 week, 3 days and 12 hours. Setting the value to *0* disables pruning.

Note: You might need to create and adjust permissions to the default caching folder path, especially if you are running Suricata as a non-root user.

Note: If you're running multiple Suricata instances, use separate cache folders for each one to avoid read/write conflicts when they run at the same time.

11.5 High Performance Configuration

11.5.1 NIC

One of the major dependencies for Suricata's performance is the Network Interface Card. There are many vendors and possibilities. Some NICs have and require their own specific instructions and tools of how to set up the NIC. This ensures the greatest benefit when running Suricata. Vendors like Napatech, Netronome, Accolade, Myricom include those tools and documentation as part of their sources.

For Intel, Mellanox and commodity NICs the following suggestions below could be utilized.

It is recommended that the latest available stable NIC drivers are used. In general when changing the NIC settings it is advisable to use the latest `ethtool` version. Some NICs ship with their own `ethtool` that is recommended to be used. Here is an example of how to set up the ethtool if needed:

```
wget https://mirrors.edge.kernel.org/pub/software/network/ethtool/ethtool-5.2.tar.xz
tar -xf ethtool-5.2.tar.xz
cd ethtool-5.2
./configure && make clean && make && make install
/usr/local/sbin/ethtool --version
```

When doing high performance optimisation make sure `irqbalance` is off and not running:

```
service irqbalance stop
```

Depending on the NIC's available queues (for example Intel's x710/i40 has 64 available per port/interface) the worker threads can be set up accordingly. Usually the available queues can be seen by running:

```
/usr/local/sbin/ethtool -l eth1
```

Some NICs - generally lower end 1Gbps - do not support symmetric hashing see *Packet Capture*. On those systems due to considerations for out of order packets the following setup with af-packet is suggested (the example below uses `eth1`):

```
/usr/local/sbin/ethtool -L eth1 combined 1
```

then set up af-packet with number of desired workers threads `threads: auto` (auto by default will use number of CPUs available) and `cluster-type: cluster_flow` (also the default setting)

For higher end systems/NICs a better and more performant solution could be utilizing the NIC itself a bit more. x710/i40 and similar Intel NICs or Mellanox MT27800 Family [ConnectX-5] for example can easily be set up to do a bigger chunk of the work using more RSS queues and symmetric hashing in order to allow for increased performance on the Suricata side by using af-packet with `cluster-type: cluster_qm` mode. In that mode with af-packet all packets

linked by network card to a RSS queue are sent to the same socket. Below is an example of a suggested config set up based on a 16 core one CPU/NUMA node socket system using x710:

```
rmmod i40e && modprobe i40e
ifconfig eth1 down
/usr/local/sbin/ethtool -L eth1 combined 16
/usr/local/sbin/ethtool -K eth1 rxhash on
/usr/local/sbin/ethtool -K eth1 ntuple on
ifconfig eth1 up
/usr/local/sbin/ethtool -X eth1 hkey␣
↪6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A
↪equal 16
/usr/local/sbin/ethtool -A eth1 rx off
/usr/local/sbin/ethtool -C eth1 adaptive-rx off adaptive-tx off rx-usecs 125
/usr/local/sbin/ethtool -G eth1 rx 1024
```

The commands above can be reviewed in detail in the help or manpages of the `ethtool`. In brief the sequence makes sure the NIC is reset, the number of RSS queues is set to 16, load balancing is enabled for the NIC, a low entropy toeplitz key is inserted to allow for symmetric hashing, receive offloading is disabled, the adaptive control is disabled for lowest possible latency and last but not least, the ring rx descriptor size is set to 1024. Make sure the RSS hash function is Toeplitz:

```
/usr/local/sbin/ethtool -X eth1 hfunc toeplitz
```

Let the NIC balance as much as possible:

```
for proto in tcp4 udp4 tcp6 udp6; do
   /usr/local/sbin/ethtool -N eth1 rx-flow-hash $proto sdfn
done
```

In some cases:

```
/usr/local/sbin/ethtool -N eth1 rx-flow-hash $proto sd
```

might be enough or even better depending on the type of traffic. However not all NICs allow it. The `sd` specifies the multi queue hashing algorithm of the NIC (for the particular proto) to use src IP, dst IP only. The `sdfn` allows for the tuple src IP, dst IP, src port, dst port to be used for the hashing algorithm. In the af-packet section of suricata.yaml:

```
af-packet:
 - interface: eth1
   threads: 16
   cluster-id: 99
   cluster-type: cluster_qm
   ...
   ...
```

11.5.2 CPU affinity and NUMA

Intel based systems

If the system has more then one NUMA node there are some more possibilities. In those cases it is generally recommended to use as many worker threads as cpu cores available/possible - from the same NUMA node. The example below uses a 72 core machine and the sniffing NIC that Suricata uses located on NUMA node 1. In such 2 socket configurations it is recommended to have Suricata and the sniffing NIC to be running and residing on the second NUMA

node as by default CPU 0 is widely used by many services in Linux. In a case where this is not possible it is recommended that (via the cpu affinity config section in suricata.yaml and the irq affinity script for the NIC) CPU 0 is never used.

In the case below 36 worker threads are used out of NUMA node 1's CPU, af-packet runmode with `cluster-type: cluster_qm`.

If the CPU's NUMA set up is as follows:

```
lscpu
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              72
On-line CPU(s) list: 0-71
Thread(s) per core:  2
Core(s) per socket:  18
Socket(s):           2
NUMA node(s):        2
Vendor ID:           GenuineIntel
CPU family:          6
Model:               79
Model name:          Intel(R) Xeon(R) CPU E5-2697 v4 @ 2.30GHz
Stepping:            1
CPU MHz:             1199.724
CPU max MHz:         3600.0000
CPU min MHz:         1200.0000
BogoMIPS:            4589.92
Virtualization:      VT-x
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            46080K
NUMA node0 CPU(s):   0-17,36-53
NUMA node1 CPU(s):   18-35,54-71
```

It is recommended that 36 worker threads are used and the NIC set up could be as follows:

```
rmmod i40e && modprobe i40e
ifconfig eth1 down
/usr/local/sbin/ethtool -L eth1 combined 36
/usr/local/sbin/ethtool -K eth1 rxhash on
/usr/local/sbin/ethtool -K eth1 ntuple on
ifconfig eth1 up
./set_irq_affinity local eth1
/usr/local/sbin/ethtool -X eth1 hkey␣
↪6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A
↪equal 36
/usr/local/sbin/ethtool -A eth1 rx off tx off
/usr/local/sbin/ethtool -C eth1 adaptive-rx off adaptive-tx off rx-usecs 125
/usr/local/sbin/ethtool -G eth1 rx 1024
for proto in tcp4 udp4 tcp6 udp6; do
    echo "/usr/local/sbin/ethtool -N eth1 rx-flow-hash $proto sdfn"
    /usr/local/sbin/ethtool -N eth1 rx-flow-hash $proto sdfn
done
```

In the example above the `set_irq_affinity` script is used from the NIC driver's sources. In the cpu affinity section of suricata.yaml config:

```
# Suricata is multi-threaded. Here the threading can be influenced.
threading:
 cpu-affinity:
   management-cpu-set:
     cpu: [ "1-10" ]  # include only these CPUs in affinity settings
   receive-cpu-set:
     cpu: [ "0-10" ]  # include only these CPUs in affinity settings
   worker-cpu-set:
     cpu: [ "18-35", "54-71" ]
     mode: "exclusive"
     prio:
       low: [ 0 ]
       medium: [ "1" ]
       high: [ "18-35","54-71" ]
       default: "high"
```

In the af-packet section of suricata.yaml config :

```
- interface: eth1
  # Number of receive threads. "auto" uses the number of cores
  threads: 18
  cluster-id: 99
  cluster-type: cluster_qm
  defrag: no
  mmap-locked: yes
  tpacket-v3: yes
  ring-size: 100000
  block-size: 1048576
- interface: eth1
  # Number of receive threads. "auto" uses the number of cores
  threads: 18
  cluster-id: 99
  cluster-type: cluster_qm
  defrag: no
  mmap-locked: yes
  tpacket-v3: yes
  ring-size: 100000
  block-size: 1048576
```

That way 36 worker threads can be mapped (18 per each af-packet interface slot) in total per CPUs NUMA 1 range - 18-35,54-71. That part is done via the `worker-cpu-set` affinity settings. `ring-size` and `block-size` in the config section above are decent default values to start with. Those can be better adjusted if needed as explained in *Tuning Considerations*.

AMD based systems

Another example can be using an AMD based system where the architecture and design of the system itself plus the NUMA node's interaction is different as it is based on the HyperTransport (HT) technology. In that case per NUMA thread/lock would not be needed. The example below shows a suggestion for such a configuration utilising af-packet, `cluster-type: cluster_flow`. The Mellanox NIC is located on NUMA 0.

The CPU set up is as follows:

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                128
On-line CPU(s) list:   0-127
Thread(s) per core:    2
Core(s) per socket:    32
Socket(s):             2
NUMA node(s):          8
Vendor ID:             AuthenticAMD
CPU family:            23
Model:                 1
Model name:            AMD EPYC 7601 32-Core Processor
Stepping:              2
CPU MHz:               1200.000
CPU max MHz:           2200.0000
CPU min MHz:           1200.0000
BogoMIPS:              4391.55
Virtualization:        AMD-V
L1d cache:             32K
L1i cache:             64K
L2 cache:              512K
L3 cache:              8192K
NUMA node0 CPU(s):     0-7,64-71
NUMA node1 CPU(s):     8-15,72-79
NUMA node2 CPU(s):     16-23,80-87
NUMA node3 CPU(s):     24-31,88-95
NUMA node4 CPU(s):     32-39,96-103
NUMA node5 CPU(s):     40-47,104-111
NUMA node6 CPU(s):     48-55,112-119
NUMA node7 CPU(s):     56-63,120-127
```

The `ethtool`, `show_irq_affinity.sh` and `set_irq_affinity_cpulist.sh` tools are provided from the official driver sources. Set up the NIC, including offloading and load balancing:

```
ifconfig eno6 down
/opt/mellanox/ethtool/sbin/ethtool -L eno6 combined 15
/opt/mellanox/ethtool/sbin/ethtool -K eno6 rxhash on
/opt/mellanox/ethtool/sbin/ethtool -K eno6 ntuple on
ifconfig eno6 up
/sbin/set_irq_affinity_cpulist.sh 1-7,64-71 eno6
/opt/mellanox/ethtool/sbin/ethtool -X eno6 hfunc toeplitz
/opt/mellanox/ethtool/sbin/ethtool -X eno6 hkey␣
↪6D:5A:6D:5A:6D:5A:
```

```
# Suricata is multi-threaded. Here the threading can be influenced.
threading:
 set-cpu-affinity: yes
 cpu-affinity:
   management-cpu-set:
     cpu: [ "120-127" ]  # include only these cpus in affinity settings
   receive-cpu-set:
     cpu: [ 0 ]  # include only these cpus in affinity settings
   worker-cpu-set:
     cpu: [ "8-55" ]
     mode: "exclusive"
     prio:
       high: [ "8-55" ]
       default: "high"
```

In the af-packet section of suricata.yaml config:

```
- interface: eth1
  # Number of receive threads. "auto" uses the number of cores
  threads: 48 # 48 worker threads on cpus "8-55" above
  cluster-id: 99
  cluster-type: cluster_flow
  defrag: no
  mmap-locked: yes
  tpacket-v3: yes
  ring-size: 100000
  block-size: 1048576
```

In the example above there are 15 RSS queues pinned to cores 1-7,64-71 on NUMA node 0 and 40 worker threads using other CPUs on different NUMA nodes. The reason why CPU 0 is skipped in this set up is as in Linux systems it is very common for CPU 0 to be used by default by many tools/services. The NIC itself in this config is positioned on NUMA 0 so starting with 15 RSS queues on that NUMA node and keeping those off for other tools in the system could offer the best advantage.

Note

Performance and optimization of the whole system can be affected upon regular NIC driver and pkg/kernel upgrades so it should be monitored regularly and tested out in QA/test environments first. As a general suggestion it is always recommended to run the latest stable firmware and drivers as instructed and provided by the particular NIC vendor.

Other considerations

Another advanced option to consider is the `isolcpus` kernel boot parameter is a way of allowing CPU cores to be isolated for use of general system processes. That way ensures total dedication of those CPUs/ranges for the Suricata process only.

`stream.wrong_thread` / `tcp.pkt_on_wrong_thread` are counters available in `stats.log` or `eve.json` as `event_type: stats` that indicate issues with the load balancing. There could be traffic/NICs settings related as well. In very high/heavily increasing counter values it is recommended to experiment with a different load balancing method either via the NIC or for example using XDP/eBPF. There is an issue open https://redmine.openinfosecfoundation.org/issues/2725 that is a placeholder for feedback and findings.

11.6 Statistics

The stats.log produces statistics records on a fixed interval, by default every 8 seconds.

11.6.1 stats.log file

```
-------------------------------------------------------------------
Counter                   | TM Name                   | Value
-------------------------------------------------------------------
flow.memuse               | FlowManagerThread         | 6557568
flow.spare                | FlowManagerThread         | 10002
flow.emerg_mode_entered   | FlowManagerThread         | 0
flow.emerg_mode_over      | FlowManagerThread         | 0
decoder.pkts              | RxPcapem21                | 450001754
decoder.bytes             | RxPcapem21                | 409520714250
decoder.ipv4              | RxPcapem21                | 449584047
decoder.ipv6              | RxPcapem21                | 9212
decoder.ethernet          | RxPcapem21                | 450001754
decoder.raw               | RxPcapem21                | 0
decoder.sll               | RxPcapem21                | 0
decoder.tcp               | RxPcapem21                | 448124337
decoder.udp               | RxPcapem21                | 542040
decoder.sctp              | RxPcapem21                | 0
decoder.icmpv4            | RxPcapem21                | 82292
decoder.icmpv6            | RxPcapem21                | 9164
decoder.ppp               | RxPcapem21                | 0
decoder.pppoe             | RxPcapem21                | 0
decoder.gre               | RxPcapem21                | 0
decoder.vlan              | RxPcapem21                | 0
decoder.avg_pkt_size      | RxPcapem21                | 910
decoder.max_pkt_size      | RxPcapem21                | 1514
defrag.ipv4.fragments     | RxPcapem21                | 4
defrag.ipv4.reassembled   | RxPcapem21                | 1
defrag.ipv4.timeouts      | RxPcapem21                | 0
defrag.ipv6.fragments     | RxPcapem21                | 0
defrag.ipv6.reassembled   | RxPcapem21                | 0
defrag.ipv6.timeouts      | RxPcapem21                | 0
tcp.sessions              | Detect                    | 41184
tcp.ssn_memcap_drop       | Detect                    | 0
tcp.pseudo                | Detect                    | 2087
tcp.invalid_checksum      | Detect                    | 8358
tcp.no_flow               | Detect                    | 0
tcp.reused_ssn            | Detect                    | 11
tcp.memuse                | Detect                    | 36175872
tcp.syn                   | Detect                    | 85902
tcp.synack                | Detect                    | 83385
tcp.rst                   | Detect                    | 84326
tcp.segment_memcap_drop   | Detect                    | 0
tcp.stream_depth_reached  | Detect                    | 109
tcp.reassembly_memuse     | Detect                    | 67755264
tcp.reassembly_gap        | Detect                    | 789
detect.alert              | Detect                    | 14721
```

Detecting packet loss

At shut down, Suricata reports the packet loss statistics it gets from pcap, pfring or afpacket

```
[18088] 30/5/2012 -- 07:39:18 - (RxPcapem21) Packets 451595939, bytes 410869083410
[18088] 30/5/2012 -- 07:39:18 - (RxPcapem21) Pcap Total:451674222 Recv:451596129␣
↪Drop:78093 (0.0%).
```

Usually, this is not the complete story though. These are kernel drop stats, but the NIC may also have dropped packets. Use ethtool to get to those:

```
# ethtool -S em2
NIC statistics:
     rx_packets: 35430208463
     tx_packets: 216072
     rx_bytes: 32454370137414
     tx_bytes: 53624450
     rx_broadcast: 17424355
     tx_broadcast: 133508
     rx_multicast: 5332175
     tx_multicast: 82564
     rx_errors: 47
     tx_errors: 0
     tx_dropped: 0
     multicast: 5332175
     collisions: 0
     rx_length_errors: 0
     rx_over_errors: 0
     rx_crc_errors: 51
     rx_frame_errors: 0
     rx_no_buffer_count: 0
     rx_missed_errors: 0
     tx_aborted_errors: 0
     tx_carrier_errors: 0
     tx_fifo_errors: 0
     tx_heartbeat_errors: 0
     tx_window_errors: 0
     tx_abort_late_coll: 0
     tx_deferred_ok: 0
     tx_single_coll_ok: 0
     tx_multi_coll_ok: 0
     tx_timeout_count: 0
     tx_restart_queue: 0
     rx_long_length_errors: 0
     rx_short_length_errors: 0
     rx_align_errors: 0
     tx_tcp_seg_good: 0
     tx_tcp_seg_failed: 0
     rx_flow_control_xon: 0
     rx_flow_control_xoff: 0
     tx_flow_control_xon: 0
     tx_flow_control_xoff: 0
     rx_long_byte_count: 32454370137414
     rx_csum_offload_good: 35270755306
```

(continues on next page)

(continued from previous page)

```
    rx_csum_offload_errors: 65076
    alloc_rx_buff_failed: 0
    tx_smbus: 0
    rx_smbus: 0
    dropped_smbus: 0
```

11.6.2 Kernel drops

stats.log contains interesting information in the capture.kernel_packets and capture.kernel_drops. The meaning of them is different following the capture mode.

In AF_PACKET mode:

- kernel_packets is the number of packets correctly sent to userspace
- kernel_drops is the number of packets that have been discarded instead of being sent to userspace

In PF_RING mode:

- kernel_packets is the total number of packets seen by pf_ring
- kernel_drops is the number of packets that have been discarded instead of being sent to userspace

In the Suricata stats.log the TCP data gap counter is also an indicator, as it accounts missing data packets in TCP streams:

```
tcp.reassembly_gap        | Detect                    | 789
```

Ideally, this number is 0. Not only pkt loss affects it though, also bad checksums and stream engine running out of memory.

11.6.3 Tools to plot graphs

Some people made nice tools to plot graphs of the statistics file.

- ipython and matplotlib script
- Monitoring with Zabbix or other and Code on GitHub

11.7 Ignoring Traffic

In some cases there are reasons to ignore certain traffic. Certain hosts may be trusted, or perhaps a backup stream should be ignored.

11.7.1 Capture Filters (BPF)

Through BPFs the capture methods pcap, af-packet, netmap and pf_ring can be told what to send to Suricata, and what not. For example a simple filter 'tcp' will only capture tcp packets.

If some hosts and or nets need to be ignored, use something like "not (host IP1 or IP2 or IP3 or net NET/24)".

Example:

```
not host 1.2.3.4
```

Capture filters are specified on the command-line after all other options:

```
suricata -i eth0 -v not host 1.2.3.4
suricata -i eno1 -c suricata.yaml tcp or udp
```

Capture filters can be set per interface in the pcap, af-packet, netmap and pf_ring sections. It can also be put in a file:

```
echo "not host 1.2.3.4" > capture-filter.bpf
suricata -i ens5f0 -F capture-filter.bpf
```

Using a capture filter limits what traffic Suricata processes. So the traffic not seen by Suricata will not be inspected, logged or otherwise recorded.

BPF and IPS

In case of IPS modes using af-packet and netmap, BPFs affect how traffic is forwarded. If a capture NIC does not capture a packet because of a BPF, it will also not be forwarded to the peering NIC.

So in the example of *not host 1.2.3.4*, traffic to and from the IP *1.2.3.4* is effectively dropped.

11.7.2 pass rules

Pass rules are Suricata rules that if matching, pass the packet and in case of TCP the rest of the flow. They look like normal rules, except that instead of *alert* or *drop* they use *pass* as the action.

Example:

```
pass ip 1.2.3.4 any <> any any (msg:"pass all traffic from/to 1.2.3.4"; sid:1;)
```

A big difference with capture filters is that logs such as Eve are still generated for this traffic.

11.7.3 suppress

Suppress rules can be used to make sure no alerts are generated for a host. This is not efficient however, as the suppression is only considered post-matching. In other words, Suricata first inspects a rule, and only then will it consider per-host suppressions.

Example:

```
suppress gen_id 0, sig_id 0, track by_src, ip 1.2.3.4
```

11.7.4 Encrypted Traffic

The TLS and SSH app layer parsers have the ability to stop processing encrypted traffic after the initial handshake. By setting the *app-layer.protocols.tls.encryption-handling* and *app-layer.protocols.quic.encryption-handling* and *app-layer.protocols.ssh.encryption-handling* options to *bypass* Suricata bypasses flows once the handshake is completed and encrypted traffic is detected. The rest of the flow is ignored. The bypass is done in the kernel or in hardware, similar to how flow bypass is done.

11.7.5 Bypassing Traffic

Aside from using the `bypass` keyword in rules, there are three other ways to bypass traffic.

- Within suricata (local bypass). Suricata reads a packet, decodes it, checks it in the flow table. If the corresponding flow is local bypassed then it simply skips all streaming, detection and output and the packet goes directly out in IDS mode and to verdict in IPS mode.

- Within the kernel (capture bypass). When Suricata decides to bypass it calls a function provided by the capture method to declare the bypass in the capture. For NFQ this is a simple mark that will be used by the iptables/nftablesruleset. For AF_PACKET this will be a call to add an element in an eBPF hash table stored in kernel.
- Within the NIC driver. This method relies upon XDP, XDP can process the traffic prior to reaching the kernel.

Additional bypass documentation:

https://suricon.net/wp-content/uploads/2017/12/SuriCon17-Manev_Purzynski.pdf https://www.stamus-networks.com/2016/09/28/suricata-bypass-feature/

11.8 Packet Profiling

In this guide will be explained how to enable packet profiling and use it with the most recent code of Suricata on Ubuntu. It is based on the assumption that you have already installed Suricata once from the GIT repository.

Packet profiling is convenient in case you would like to know how long packets take to be processed. It is a way to figure out why certain packets are being processed quicker than others, and this way a good tool for developing Suricata.

Update Suricata by following the steps from *Installation from GIT*. Start at the end at

```
cd suricata/suricata
git pull
```

And follow the described next steps. To enable packet profiling, make sure you enter the following during the configuring stage:

```
./configure --enable-profiling
```

Find a folder in which you have pcaps. If you do not have pcaps yet, you can get these with Wireshark. See Sniffing Packets with Wireshark.

Go to the directory of your pcaps. For example:

```
cd  ~/Desktop
```

With the ls command you can see the content of the folder. Choose a folder and a pcap file

for example:

```
cd ~/Desktop/2011-05-05
```

Run Suricata with that pcap:

```
suricata -c /etc/suricata/suricata.yaml -r log.pcap.(followed by the number/name of your␣
↪pcap)
```

for example:

```
suricata -c /etc/suricata/suricata.yaml -r log.pcap.1304589204
```

11.9 Rule Profiling

Suricata can generate a rules performance report at the end of each session, if built with the `enable-profiling` option (see configuring *Rule profiling*). With that option, the engine will also generate profiling data for other engine modules, such as packet processing.

Rule profiling can also be enabled by building the engine with `enable-profiling -rules` and using the unix socket to dump the report (see *Rules Profiling*). This will enable profiling of rules' statistics only.

Once the report is generated, it is stored in the default log directory used by Suricata. If not changed, the filename will be `rule_perf.log`.

A Rules Profile report looks like this:

```
-------------------------------------------------------------------------------
Date: 9/5/2013 -- 14:59:58
-------------------------------------------------------------------------------
 Num      Rule         Gid      Rev      Ticks        %      Checks   Matches  Max Ticks␣
↪  Avg Ticks   Avg Match   Avg No Match
-------- ------------ -------- -------- ------------ ------ -------- -------- -----------
↪ ----------- ----------- --------------
1        2210021      1        3        12037        4.96   1        1        12037     ␣
↪ 12037.00    12037.00    0.00
2        2210054      1        1        107479       44.26  12       0        35805     ␣
↪ 8956.58     0.00        8956.58
3        2210053      1        1        4513         1.86   1        0        4513      ␣
↪ 4513.00     0.00        4513.00
4        2210023      1        1        3077         1.27   1        0        3077      ␣
↪ 3077.00     0.00        3077.00
5        2210008      1        1        3028         1.25   1        0        3028      ␣
↪ 3028.00     0.00        3028.00
6        2210009      1        1        2945         1.21   1        0        2945      ␣
↪ 2945.00     0.00        2945.00
7        2210055      1        1        2945         1.21   1        0        2945      ␣
↪ 2945.00     0.00        2945.00
8        2210007      1        1        2871         1.18   1        0        2871      ␣
↪ 2871.00     0.00        2871.00
9        2210005      1        1        2871         1.18   1        0        2871      ␣
↪ 2871.00     0.00        2871.00
10       2210024      1        1        2846         1.17   1        0        2846      ␣
↪ 2846.00     0.00        2846.00
```

The meaning of the individual fields:

- Ticks -- total ticks spent on this rule, so a sum of all inspections.
- % -- share of this single signature in the total cost of inspection.
- Checks -- number of times a signature was inspected.
- Matches -- number of times it matched. This may not have resulted in an alert due to suppression and thresholding.
- Max ticks -- single most expensive inspection.
- Avg ticks -- per inspection average, so "ticks" / "checks".
- Avg match -- avg ticks spent resulting in match.
- Avg No Match -- avg ticks spent resulting in no match.

The "ticks" are CPU clock ticks: http://en.wikipedia.org/wiki/CPU_time

11.10 Tcmalloc

'tcmalloc' is a library Google created as part of the google-perftools suite for improving memory handling in a threaded program. It's very simple to use and does work fine with Suricata. It leads to minor speed ups and also reduces memory usage quite a bit.

11.10.1 Installation

On Ubuntu, install the libtcmalloc-minimal4 package:

```
apt-get install libtcmalloc-minimal4
```

On Fedora, install the gperftools-libs package:

```
yum install gperftools-libs
```

11.10.2 Usage

Use the tcmalloc by preloading it:

Ubuntu:

```
LD_PRELOAD="/usr/lib/x86_64-linux-gnu/libtcmalloc_minimal.so.4" suricata -c suricata.
↪yaml -i eth0
```

Fedora:

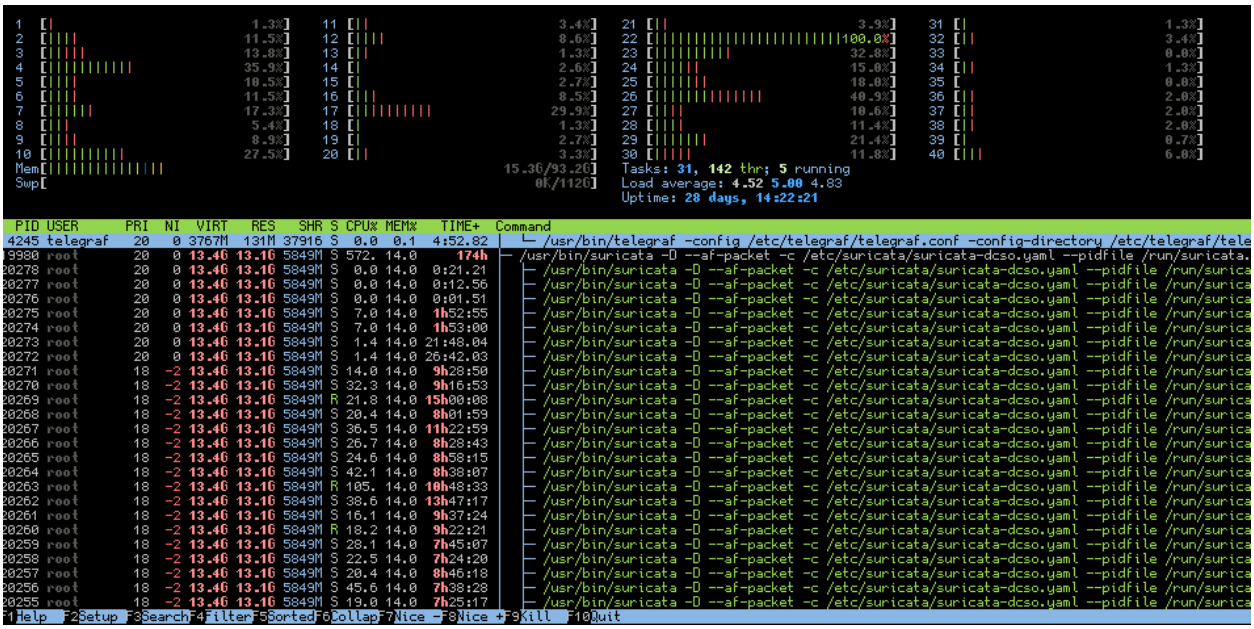
```
LD_PRELOAD="/usr/lib64/libtcmalloc_minimal.so.4" suricata -c suricata.yaml -i eth0
```

11.11 Performance Analysis

There are many potential causes for performance issues. In this section we will guide you through some options. The first part will cover basic steps and introduce some helpful tools. The second part will cover more in-depth explanations and corner cases.

11.11.1 System Load

The first step should be to check the system load. Run a top tool like **htop** to get an overview of the system load and if there is a bottleneck with the traffic distribution. For example if you can see that only a small number of cpu cores hit 100% all the time and others don't, it could be related to a bad traffic distribution or elephant flows like in the screenshot where one core peaks due to one big elephant flow.



If all cores are at peak load the system might be too slow for the traffic load or it might be misconfigured. Also keep an eye on memory usage, if the actual memory usage is too high and the system needs to swap it will result in very poor performance.

The load will give you a first indication where to start with the debugging at specific parts we describe in more detail in the second part.

11.11.2 Logfiles

The next step would be to check all the log files with a focus on **stats.log** and **suricata.log** if any obvious issues are seen. The most obvious indicator is the **capture.kernel_drops** value that ideally would not even show up but should be below 1% of the **capture.kernel_packets** value as high drop rates could lead to a reduced amount of events and alerts.

If **memcap** is seen in the stats the memcap values in the configuration could be increased. This can result to higher memory usage and should be taken into account when the settings are changed.

Don't forget to check any system logs as well, even a **dmesg** run can show potential issues.

11.11.3 Suricata Load

Besides the system load, another indicator for potential performance issues is the load of Suricata itself. A helpful tool for that is **perf** which helps to spot performance issues. Make sure you have it installed and also the debug symbols installed for Suricata or the output won't be very helpful. This output is also helpful when you report performance issues as the Suricata Development team can narrow down possible issues with that.

```
sudo perf top -p $(pidof suricata)
```

If you see specific function calls at the top in red it's a hint that those are the bottlenecks. For example if you see **IPOnlyMatchPacket** it can be either a result of high drop rates or incomplete flows which result in decreased performance. To look into the performance issues on a specific thread you can pass **-t TID** to perf top. In other cases you can see functions that give you a hint that a specific protocol parser is used a lot and can either try to debug a performance bug or try to filter related traffic.

```
Samples: 176K of event 'cycles', 4000 Hz, Event count (approx.): 89969570589 lost: 0/0 drop: 0/0
Overhead  Shared Object                     Symbol
  21.99%  suricata                          [.] IPOnlyMatchPacket
   4.64%  libhs.so.5.1.0                    [.] 0x0000000000739ee1
   2.64%  suricata                          [.] FlowGetFlowFromHash
   2.56%  suricata                          [.] DetectRun.part.16
   1.65%  libpthread-2.28.so                [.] __pthread_mutex_lock
   1.59%  [kernel]                          [k] tpacket_rcv
   1.36%  libhs.so.5.1.0                    [.] 0x0000000000739f32
   1.15%  [kernel]                          [k] i40e_napi_poll
   0.90%  suricata                          [.] AFPReadFromRingV3
   0.86%  libhs.so.5.1.0                    [.] 0x0000000000736ac9
   0.83%  suricata                          [.] FlowManager
   0.82%  libluajit-5.1.so.2.1.0            [.] 0x000000000000bdd2
   0.82%  [kernel]                          [k] memcpy_erms
   0.68%  libhs.so.5.1.0                    [.] 0x0000000000736adc
   0.67%  suricata                          [.] SCHSMatchEvent
   0.65%  [kernel]                          [k] __netif_receive_skb_core
   0.64%  suricata                          [.] DetectPortLookupGroup
   0.61%  suricata                          [.] DetectAddressMatchIPv4
   0.60%  libhs.so.5.1.0                    [.] avx512_hs_scan
   0.58%  libhs.so.5.1.0                    [.] 0x0000000000739ef4
   0.55%  [kernel]                          [k] eth_type_trans
   0.52%  libpthread-2.28.so                [.] __pthread_mutex_unlock_usercnt
   0.51%  suricata                          [.] FlowHandlePacketUpdate
   0.51%  libhs.so.5.1.0                    [.] 0x000000000073a743
   0.51%  [kernel]                          [k] packet_rcv_fanout
   0.50%  suricata                          [.] Prefilter
   0.45%  suricata                          [.] PacketPoolGetPacket
   0.41%  suricata                          [.] OutputLoggerLog
   0.39%  suricata                          [.] FlowWorker
   0.36%  libhs.so.5.1.0                    [.] 0x0000000000736ad5
   0.36%  suricata                          [.] StatsIncr
   0.35%  libhs.so.5.1.0                    [.] 0x000000000073a700
   0.35%  libluajit-5.1.so.2.1.0            [.] 0x000000000001ab7e
   0.35%  libhs.so.5.1.0                    [.] 0x0000000000739f70
   0.34%  suricata                          [.] DecodeIPV4
   0.34%  libhs.so.5.1.0                    [.] 0x0000000000739f23
   0.33%  libhs.so.5.1.0                    [.] 0x0000000000067fec6
   0.33%  libhs.so.5.1.0                    [.] 0x0000000000736aeb
   0.32%  suricata                          [.] hashword
   0.32%  suricata                          [.] TmThreadsSlotVarRun
   0.31%  libhs.so.5.1.0                    [.] 0x0000000000739f14
   0.31%  libtcmalloc_minimal.so.4.5.3      [.] tc_deletearray_aligned_nothrow
   0.31%  [kernel]                          [k] __build_skb
   0.29%  libhs.so.5.1.0                    [.] 0x0000000000739f03
   0.29%  [kernel]                          [k] kmem_cache_free
   0.29%  libhs.so.5.1.0                    [.] 0x0000000000739eed
   0.29%  suricata                          [.] TmqhOutputPacketpool
   0.28%  libc-2.28.so                      [.] 0x000000000015c6ff
   0.28%  suricata                          [.] SCHSSearch
   0.28%  libtcmalloc_minimal.so.4.5.3      [.] tc_malloc
For a higher level overview, try: perf top --sort comm,dso
```

In general try to play around with the different configuration options that Suricata does provide with a focus on the options described in *High Performance Configuration*.

11.11.4 Traffic

In most cases where the hardware is fast enough to handle the traffic but the drop rate is still high it's related to specific traffic issues.

Basics

Some of the basic checks are:

- Check if the traffic is bidirectional, if it's mostly unidirectional you're missing relevant parts of the flow (see **tshark** example at the bottom). Another indicator could be a big discrepancy between SYN and SYN-ACK as well as RST counter in the Suricata stats.

- Check for encapsulated traffic, while GRE, MPLS etc. are supported they could also lead to performance issues. Especially if there are several layers of encapsulation.
- Use tools like **iftop** to spot elephant flows. Flows that have a rate of over 1Gbit/s for a long time can result in one cpu core peak at 100% all the time and increasing the droprate while it might not make sense to dig deep into this traffic.
- Another approach to narrow down issues is the usage of **bpf filter**. For example filter all HTTPS traffic with **not port 443** to exclude traffic that might be problematic or just look into one specific port **port 25** if you expect some issues with a specific protocol. See *Ignoring Traffic* for more details.
- If VLAN is used it might help to disable **vlan.use-for-tracking** in scenarios where only one direction of the flow has the VLAN tag.

Advanced

There are several advanced steps and corner cases when it comes to a deep dive into the traffic.

If VLAN QinQ (IEEE 802.1ad) is used be very cautious if you use **cluster_qm** in combination with Intel drivers and AF_PACKET runmode. While the RFC expects ethertype 0x8100 and 0x88A8 in this case (see https://en.wikipedia.org/wiki/IEEE_802.1ad) most implementations only add 0x8100 on each layer. If the first seen layer has the same VLAN tag but the inner one has different VLAN tags it will still end up in the same queue in **cluster_qm** mode. This was observed with the i40e driver up to 2.8.20 and the firmware version up to 7.00, feel free to report if newer versions have fixed this (see https://suricata.io/support/).

If you want to use **tshark** to get an overview of the traffic direction use this command:

```
sudo tshark -i $INTERFACE -q -z conv,ip -a duration:10
```

The output will show you all flows within 10s and if you see 0 for one direction you have unidirectional traffic, thus you don't see the ACK packets for example. Since Suricata is trying to work on flows this will have a rather big impact on the visibility. Focus on fixing the unidirectional traffic. If it's not possible at all you can enable **async-oneside** in the **stream** configuration setting.

Check for other unusual or complex protocols that aren't supported very well. You can try to filter those to see if it has any impact on the performance. In this example we filter Cisco Fabric Path (ethertype 0x8903) with the bpf filter **not ether proto 0x8903** as it's assumed to be a performance issue (see https://redmine.openinfosecfoundation.org/issues/3637)

Elephant Flows

The so called Elephant Flows or traffic spikes are quite difficult to deal with. In most cases those are big file transfers or backup traffic and it's not feasible to decode the whole traffic. From a network security monitoring perspective it's often enough to log the metadata of that flow and do a packet inspection at the beginning but not the whole flow.

If you can spot specific flows as described above then try to filter those. The easiest solution would be a bpf filter but that would still result in a performance impact. Ideally you can filter such traffic even sooner on driver or NIC level (see eBPF/XDP) or even before it reaches the system where Suricata is running. Some commercial packet broker support such filtering where it's called **Flow Shunting** or **Flow Slicing**.

11.11.5 Rules

The Ruleset plays an important role in the detection but also in the performance capability of Suricata. Thus it's recommended to look into the impact of enabled rules as well.

If you run into performance issues and struggle to narrow it down start with running Suricata without any rules enabled and use the tools again that have been explained at the first part. Keep in mind that even without signatures enabled Suricata still does most of the decoding and traffic analysis, so a fair amount of load should still be seen. If the load is still very high and drops are seen and the hardware should be capable to deal with such traffic loads you should

deep dive if there is any specific traffic issue (see above) or report the performance issue so it can be investigated (see https://suricata.io/join-our-community/).

Suricata also provides several specific traffic related signatures in the rules folder that could be enabled for testing to spot specific traffic issues. Those are found the **rules** and you should start with **decoder-events.rules**, **stream-events.rules** and **app-layer-events.rules**.

It can also be helpful to use *Rule Profiling* and/or *Packet Profiling* to find problematic rules or traffic pattern. This is achieved by compiling Suricata with **--enable-profiling** but keep in mind that this has an impact on performance and should only be used for troubleshooting.

CHAPTER

TWELVE

CONFIGURATION

12.1 Suricata.yaml

Suricata uses the Yaml format for configuration. The Suricata.yaml file included in the source code, is the example configuration of Suricata. This document will explain each option.

At the top of the YAML-file you will find % YAML 1.1. Suricata reads the file and identifies the file as YAML.

12.1.1 Max-pending-packets

With the max-pending-packets setting you can set the number of packets you allow Suricata to process simultaneously. This can range from one packet to tens of thousands/hundreds of thousands of packets. It is a trade of higher performance and the use of more memory (RAM), or lower performance and less use of memory. A high number of packets being processed results in a higher performance and the use of more memory. A low number of packets, results in lower performance and less use of memory. Choosing a low number of packets being processed while having many CPU's/CPU cores, can result in not making use of the whole computer-capacity. (For instance: using one core while having three waiting for processing packets.)

```
max-pending-packets: 1024
```

12.1.2 Runmodes

By default the runmode option is disabled. With the runmodes setting you can set the runmode you would like to use. For all runmodes available, enter **--list-runmodes** in your command line. For more information, see *Runmodes*.

```
runmode: autofp
```

12.1.3 Default-packet-size

For the max-pending-packets option, Suricata has to keep packets in memory. With the default-packet-size option, you can set the size of the packets on your network. It is possible that bigger packets have to be processed sometimes. The engine can still process these bigger packets, but processing it will lower the performance.

```
default-packet-size: 1514
```

12.1.4 User and group

It is possible to set the user and group to run Suricata as:

```
run-as:
  user: suri
  group: suri
```

12.1.5 PID File

This option sets the name of the PID file when Suricata is run in daemon mode. This file records the Suricata process ID.

```
pid-file: /var/run/suricata.pid
```

> **Note**
>
> This configuration file option only sets the PID file when running in daemon mode. To force creation of a PID file when not running in daemon mode, use the `--pidfile` command line option.
>
> Also, if running more than one Suricata process, each process will need to specify a different pid-file location.

12.1.6 Action-order

All signatures have different properties. One of those is the Action property. This one determines what will happen when a signature matches. There are four types of Action. A summary of what will happen when a signature matches and contains one of those Actions:

1) Pass

If a signature matches and contains pass, Suricata stops scanning the packet and skips to the end of all rules (only for the current packet). If the signature matches on a TCP connection, the entire flow will be passed but details of the flow will still be logged.

2) Drop

This only concerns the IPS/inline mode. If the program finds a signature that matches, containing drop, it stops immediately. The packet will not be sent any further. Drawback: The receiver does not receive a message of what is going on, resulting in a time-out (certainly with TCP). Suricata generates an alert for this packet.

3) Reject

This is an active rejection of the packet. Both receiver and sender receive a reject packet. There are two types of reject packets that will be automatically selected. If the offending packet concerns TCP, it will be a Reset-packet. For all other protocols it will be an ICMP-error packet. Suricata also generates an alert. When in Inline/IPS mode, the offending packet will also be dropped like with the 'drop' action.

4) Alert

If a signature matches and contains alert, the packet will be treated like any other non-threatening packet, except for this one an alert will be generated by Suricata. Only the system administrator can notice this alert.

Inline/IPS can block network traffic in two ways. One way is by drop and the other by reject.

Rules will be loaded in the order of which they appear in files. But they will be processed in a different order. Signatures have different priorities. The most important signatures will be scanned first. There is a possibility to change the order of priority. The default order is: pass, drop, reject, alert.

```
action-order:
 - pass
 - drop
 - reject
 - alert
```

This means a pass rule is considered before a drop rule, a drop rule before a reject rule and so on.

12.1.7 Packet alert queue settings

It is possible to configure the size of the alerts queue that is used to append alerts triggered by each packet.

This will influence how many alerts would be perceived to have matched against a given packet. The default value is 15. If an invalid setting or no value is provided, the engine will fall back to the default.

```
#Define maximum number of possible alerts that can be triggered for the same
# packet. Default is 15
packet-alert-max: 15
```

We recommend that you use the default value for this setting unless you are seeing a high number of discarded alerts (`alert_queue_overflow`) - see the *Discarded and Suppressed Alerts Stats* section for more details.

Impact on engine behavior

Internally, the Suricata engine represents each packet with a data structure that has its own alert queue. The max size of the queue is defined by `packet-alert-max`. The same rule can be triggered by the same packet multiple times. As long as there is still space in the alert queue, those are appended.

Rules that have the `noalert` keyword will be checked - in case their signatures have actions that must be applied to the Packet or Flow, then suppressed. They have no effect in the final alert queue.

Rules are queued by priority: higher priority rules may be kept instead of lower priority ones that may have been triggered earlier, if Suricata reaches `packet-alert-max` for a given packet (a.k.a. packet alert queue overflow).

Packet alert queue overflow

Once the alert queue reaches its max size, we are potentially at packet alert queue overflow, so new alerts will only be appended in case their rules have a higher priority id (this is the internal id attributed by the engine, not the signature id).

This may happen in two different situations:

- a higher priority rule is triggered after a lower priority one: the lower priority rule is replaced in the queue;
- a lower priority rule is triggered: the rule is just discarded.

> **Note**
>
> This behavior does not mean that triggered `drop` rules would have their action ignored, in IPS mode.

Discarded and Suppressed Alerts Stats

Both scenarios previously described will be logged as *detect.alert_queue_overflow* in the stats logs (in stats.log and eve-log's stats event).

When `noalert` rules match, they appear in the stats logs as *detect.alerts_suppressed*.

```
Date: 4/6/2022 -- 17:18:08 (uptime: 0d, 00h 00m 00s)
----------------------------------------------------------------------------------
Counter                                       | TM Name                   | Value
----------------------------------------------------------------------------------
detect.alert                                  | Total                     | 3
detect.alert_queue_overflow                   | Total                     | 4
detect.alerts_suppressed                      | Total                     | 1
```

In this example from a stats.log, we read that 8 alerts were generated: 3 were kept in the packet queue while 4 were discarded due to packets having reached max size for the alert queue, and 1 was suppressed due to coming from a `noalert` rule.

12.1.8 Splitting configuration in multiple files

Some users might have a need or a wish to split their suricata.yaml file into separate files, this is available via the 'include' and '!include' keyword. The first example is of taking the contents of the outputs section and storing them in outputs.yaml.

```
# outputs.yaml
- fast
    enabled: yes
    filename: fast.log
    append: yes

...
```

```
# suricata.yaml
...

outputs: !include outputs.yaml

...
```

The second scenario is where multiple sections are migrated to a different YAML file.

```
# host_1.yaml

max-pending-packets: 2048

outputs:
    - fast
        enabled: yes
        filename: fast.log
        append: yes
```

```
# suricata.yaml

include: host_1.yaml

...
```

If the same section, say outputs is later redefined after the include statement it will overwrite the included file. Therefore any include statement at the end of the document will overwrite the already configured sections.

12.1.9 Event output

Default logging directory

In the /var/log/suricata directory, all of Suricata's output (alerts and events) will be stored.

```
default-log-dir: /var/log/suricata
```

This directory can be overridden by entering the -l command line parameter or by changing the directory directly in Yaml. To change it with the -l command line parameter, enter the following:

```
suricata -c suricata.yaml -i eth0 -l /var/log/suricata-logs/
```

Stats

Engine statistics such as packet counters, memory use counters and others can be logged in several ways. A separate text log 'stats.log' and an EVE record type 'stats' are enabled by default.

The stats have a global configuration and a per logger configuration. Here the global config is documented.

```
# global stats configuration
stats:
  enabled: yes
  # The interval field (in seconds) controls at what interval
  # the loggers are invoked.
  interval: 8
  # Add decode events as stats.
  #decoder-events: true
  # Decoder event prefix in stats. Has been 'decoder' before, but that leads
  # to missing events in the eve.stats records. See issue #2225.
  #decoder-events-prefix: "decoder.event"
  # Add stream events as stats.
  #stream-events: false
  # Exception policy stats counters options
  # (Note: if exception policy: ignore, counters are not logged)
  exception-policy:
    #per-app-proto-errors: false  # default: false. True will log errors for
                                  # each app-proto. Warning: VERY verbose
```

Statistics can be *enabled* or disabled here.

Statistics are dumped on an *interval*. Setting this below 3 or 4 seconds is not useful due to how threads are synchronized internally.

The decoder events that the decoding layer generates, can create a counter per event type. This behaviour is enabled by default. The *decoder-events* option can be set to *false* to disable.

In 4.1.x there was a naming clash between the regular decoder counters and the decoder-event counters. This lead to a fair amount of decoder-event counters not being shown in the EVE.stats records. To address this without breaking existing setups, a config option *decoder-events-prefix* was added to change the naming of the decoder-events from decoder.<proto>.<event> to decoder.event.<proto>.<event>. In 5.0 this became the default. See issue 2225.

Similar to the *decoder-events* option, the *stream-events* option controls whether the stream-events are added as counters as well. This is disabled by default.

If any exception policy is enabled, stats counters are logged. To control verbosity for application layer protocol errors, leave *per-app-proto-errors* as false.

Outputs

There are several types of output. The general structure is:

```
outputs:
  - fast:
    enabled: yes
```

(continues on next page)

(continued from previous page)

```
    filename: fast.log
    append: yes/no
    # Compress IPv6 addresses per RFC5952 as they are added to the fast log. The default␣
↪is no.
    ipv6-compress: yes/no
```

Enabling all of the logs, will result in a much lower performance and the use of more disc space, so enable only the outputs you need.

Line based alerts log (fast.log)

This log contains alerts consisting of a single line. Example of the appearance of a single fast.log-file line:

```
10/05/10-10:08:59.667372  [**] [1:2009187:4] ET WEB_CLIENT ACTIVEX iDefense
  COMRaider ActiveX Control Arbitrary File Deletion [**] [Classification: Web
  Application Attack] [Priority: 3] {TCP} xx.xx.232.144:80 -> 192.168.1.4:56068
```

```
-fast:                    #The log-name.
   enabled:yes            #This log is enabled. Set to 'no' to disable.
   filename: fast.log     #The name of the file in the default logging directory.
   append: yes/no         #If this option is set to yes, the last filled fast.log-file␣
↪will not be
                          #overwritten while restarting Suricata.
   # Compress IPv6 addresses per RFC5952 as they are added to the fast log. The default␣
↪is no.
   ipv6-compress: yes/no
```

Eve (Extensible Event Format)

This is an JSON output for alerts and events. It allows for easy integration with 3rd party tools like logstash.

```
outputs:
  # Extensible Event Format (nicknamed EVE) event log in JSON format
  - eve-log:
      enabled: yes
      filetype: regular #regular|syslog|unix_dgram|unix_stream|redis
      filename: eve.json
      # Enable for multi-threaded eve.json output; output files are amended with
      # an identifier, e.g., eve.9.json
      #threaded: false
      # Specify the amount of buffering, in bytes, for
      # this output type. The default value 0 means "no
      # buffering".
      #buffer-size: 0
      #prefix: "@cee: " # prefix to prepend to each log entry
      # the following are valid when type: syslog above
      #identity: "suricata"
      #facility: local5
      #level: Info ## possible levels: Emergency, Alert, Critical,
                   ## Error, Warning, Notice, Info, Debug
      #ethernet: no  # log ethernet header in events when available
      #redis:
```

(continues on next page)

(continued from previous page)

```
      #  server: 127.0.0.1
      #  port: 6379
      #  username: null ## ACL username; if null (default), no username is used
      #  password: null ## AUTH password; if null (default), authentication is disabled
      #  async: true ## if redis replies are read asynchronously
      #  mode: list ## possible values: list|lpush (default), rpush, channel|publish,␣
↪xadd|stream
      #             ## lpush and rpush are using a Redis list. "list" is an alias for␣
↪lpush
      #             ## publish is using a Redis channel. "channel" is an alias for␣
↪publish
      #             ## xadd is using a Redis stream. "stream" is an alias for xadd
      #  key: suricata ## string denoting the key/channel/stream to use (default to␣
↪suricata)
      #  stream-maxlen: 100000        ## Automatically trims the stream length to at most
                                      ## this number of events. Set to 0 to disable␣
↪trimming.
                                      ## Only used when mode is set to xadd/stream.
      #  stream-trim-exact: false     ## Trim exactly to the maximum stream length above.
                                      ## Default: use inexact trimming (inexact by a few
                                      ## tens of items)
                                      ## Only used when mode is set to xadd/stream.
      # Redis pipelining set up. This will enable to only do a query every
      # 'batch-size' events. This should lower the latency induced by network
      # connection at the cost of some memory. There is no flushing implemented
      # so this setting should be reserved to high traffic Suricata deployments.
      #  pipelining:
      #    enabled: yes ## set enable to yes to enable query pipelining
      #    batch-size: 10 ## number of entries to keep in buffer

      # Include top level metadata. Default yes.
      #metadata: no

      # include the name of the input pcap file in pcap file processing mode
      pcap-file: false

      # Compress IPv6 addresses per RFC5952 as they are added to the eve-log. The␣
↪default is no.
      # ipv6-compress: no

      # Community Flow ID
      # Adds a 'community-id' field to EVE records. These are meant to give
      # records a predictable flow ID that can be used to match records to
      # output of other tools such as Zeek (Bro).
      #
      # Takes a 'seed' that needs to be same across sensors and tools
      # to make the id less predictable.

      # enable/disable the community id feature.
      community-id: false
      # Seed value for the ID output. Valid values are 0-65535.
      community-id-seed: 0
```

(continues on next page)

(continued from previous page)

```
    # HTTP X-Forwarded-For support by adding an extra field or overwriting
    # the source or destination IP address (depending on flow direction)
    # with the one reported in the X-Forwarded-For HTTP header. This is
    # helpful when reviewing alerts for traffic that is being reverse
    # or forward proxied.
    xff:
      enabled: no
      # Two operation modes are available: "extra-data" and "overwrite".
      mode: extra-data
      # Two proxy deployments are supported: "reverse" and "forward". In
      # a "reverse" deployment the IP address used is the last one, in a
      # "forward" deployment the first IP address is used.
      deployment: reverse
      # Header name where the actual IP address will be reported. If more
      # than one IP address is present, the last IP address will be the
      # one taken into consideration.
      header: X-Forwarded-For

    types:
      - alert:
          # payload: yes             # enable dumping payload in Base64
          # payload-buffer-size: 4kb # max size of payload buffer to output in eve-log
          # payload-printable: yes   # enable dumping payload in printable (lossy)␣
↪format
          # payload-length: yes      # enable dumping payload length, including the␣
↪gaps
          # packet: yes              # enable dumping of packet (without stream␣
↪segments)
          # metadata: no             # enable inclusion of app layer metadata with␣
↪alert. Default yes
          # If you want metadata, use:
          # metadata:
            # Include the decoded application layer (ie. http, dns)
            #app-layer: true
            # Log the current state of the flow record.
            #flow: true
            #rule:
              # Log the metadata field from the rule in a structured
              # format.
              #metadata: true
              # Log the raw rule text.
              #raw: false
              #reference: false      # include reference information from the rule
          # http-body: yes           # Requires metadata; enable dumping of HTTP body␣
↪in Base64
          # http-body-printable: yes # Requires metadata; enable dumping of HTTP body␣
↪in printable format
          # websocket-payload: yes   # Requires metadata; enable dumping of WebSocket␣
↪Payload in Base64
          # websocket-payload-printable: yes # Requires metadata; enable dumping of␣
↪WebSocket Payload in printable format
```

(continues on next page)

(continued from previous page)

```
            # Enable the logging of tagged packets for rules using the
            # "tag" keyword.
            tagged-packets: yes
            # Enable logging the final action taken on a packet by the engine
            # (e.g: the alert may have action 'allowed' but the verdict be
            # 'drop' due to another alert. That's the engine's verdict)
            # verdict: yes
        # app layer frames
        - frame:
            # disabled by default as this is very verbose.
            enabled: no
            # payload-buffer-size: 4kb # max size of frame payload buffer to output in␣
↪eve-log
        - anomaly:
            # Anomaly log records describe unexpected conditions such
            # as truncated packets, packets with invalid IP/UDP/TCP
            # length values, and other events that render the packet
            # invalid for further processing or describe unexpected
            # behavior on an established stream. Networks which
            # experience high occurrences of anomalies may experience
            # packet processing degradation.
            #
            # Anomalies are reported for the following:
            # 1. Decode: Values and conditions that are detected while
            # decoding individual packets. This includes invalid or
            # unexpected values for low-level protocol lengths as well
            # as stream related events (TCP 3-way handshake issues,
            # unexpected sequence number, etc).
            # 2. Stream: This includes stream related events (TCP
            # 3-way handshake issues, unexpected sequence number,
            # etc).
            # 3. Application layer: These denote application layer
            # specific conditions that are unexpected, invalid or are
            # unexpected given the application monitoring state.
            #
            # By default, anomaly logging is enabled. When anomaly
            # logging is enabled, applayer anomaly reporting is
            # also enabled.
            enabled: yes
            #
            # Choose one or more types of anomaly logging and whether to enable
            # logging of the packet header for packet anomalies.
            types:
              # decode: no
              # stream: no
              # applayer: yes
            #packethdr: no
        - http:
            extended: yes     # enable this for extended logging information
            # custom allows additional HTTP fields to be included in eve-log.
            # the example below adds three additional fields when uncommented
```

(continues on next page)

(continued from previous page)

```
            #custom: [Accept-Encoding, Accept-Language, Authorization]
            # set this value to one and only one from {both, request, response}
            # to dump all HTTP headers for every HTTP request and/or response
            # dump-all-headers: none
        - dns:
            # This configuration uses the new DNS logging format,
            # the old configuration is still available:
            # https://docs.suricata.io/en/latest/output/eve/eve-json-output.html#dns-v1-
↪format

            # As of Suricata 5.0, version 2 of the eve dns output
            # format is the default.
            #version: 2

            # Enable/disable this logger. Default: enabled.
            #enabled: yes

            # Control logging of requests and responses:
            # - requests: enable logging of DNS queries
            # - responses: enable logging of DNS answers
            # By default both requests and responses are logged.
            #requests: no
            #responses: no

            # Format of answer logging:
            # - detailed: array item per answer
            # - grouped: answers aggregated by type
            # Default: all
            #formats: [detailed, grouped]

            # DNS record types to log, based on the query type.
            # Default: all.
            #types: [a, aaaa, cname, mx, ns, ptr, txt]
        - tls:
            extended: yes     # enable this for extended logging information
            # output TLS transaction where the session is resumed using a
            # session id
            #session-resumption: no
            # custom controls which TLS fields that are included in eve-log
            # WARNING: enabling custom disables extended logging.
            #custom: [subject, issuer, session_resumed, serial, fingerprint, sni,␣
↪version, not_before, not_after, certificate, chain, ja3, ja3s, ja4, subjectaltname,␣
↪client, client_certificate, client_chain, client_alpns, server_alpns]
        - files:
            force-magic: no   # force logging magic on all logged files
            # force logging of checksums, available hash functions are md5,
            # sha1 and sha256
            #force-hash: [md5]
        #- drop:
        #    alerts: yes      # log alerts that caused drops
        #    flows: all       # start or all: 'start' logs only a single drop
        #                     # per flow direction. All logs each dropped pkt.
```

(continues on next page)

(continued from previous page)

```
        # Enable logging the final action taken on a packet by the engine
        # (will show more information in case of a drop caused by 'reject')
        # verdict: yes
    - smtp:
        #extended: yes # enable this for extended logging information
        # this includes: bcc, message-id, subject, x_mailer, user-agent
        # custom fields logging from the list:
        #  reply-to, bcc, message-id, subject, x-mailer, user-agent, received,
        #  x-originating-ip, in-reply-to, references, importance, priority,
        #  sensitivity, organization, content-md5, date
        #custom: [received, x-mailer, x-originating-ip, relays, reply-to, bcc]
        # output md5 of fields: body, subject
        # for the body you need to set app-layer.protocols.smtp.mime.body-md5
        # to yes
        #md5: [body, subject]

    #- dnp3
    - websocket
    - ftp
    - ftp-data
    - rdp
    - nfs
    - smb
    - tftp
    - ike
    - dcerpc
    - krb5
    - bittorrent-dht
    - ssh
    - arp:
        enabled: no
    - ntp:
        enabled: no
    - snmp
    - rfb
    - sip
    - quic
    - dhcp:
        enabled: yes
        # When extended mode is on, all DHCP messages are logged
        # with full detail. When extended mode is off (the
        # default), just enough information to map a MAC address
        # to an IP address is logged.
        extended: no
    - mqtt:
        # passwords: yes           # enable output of passwords
        # string-log-limit: 1kb    # limit size of logged strings in bytes.
                                   # Can be specified in kb, mb, gb. Just a number
                                   # is parsed as bytes. Default is 1KB.
                                   # Use a value of 0 to disable limiting.
                                   # Note that the size is also bounded by
                                   # the maximum parsed message size (see
```

(continues on next page)

(continued from previous page)

```
                                    # app-layer configuration)
        - http2
        - pgsql:
            enabled: no
            # passwords: yes           # enable output of passwords. Disabled by default
        - stats:
            totals: yes       # stats for all threads merged together
            threads: no       # per thread stats
            deltas: no        # include delta values
            # Don't log stats counters that are zero. Default: true
            #null-values: false    # False will NOT log stats counters: 0
        # bi-directional flows
        - flow
        # uni-directional flows
        #- netflow

        # Metadata event type. Triggered whenever a pktvar is saved
        # and will include the pktvars, flowvars, flowbits and
        # flowints.
        #- metadata

        # EXPERIMENTAL per packet output giving TCP state tracking details
        # including internal state, flags, etc.
        # This output is experimental, meant for debugging and subject to
        # change in both config and output without any notice.
        #- stream:
        #    all: false                     # log all TCP packets
        #    event-set: false               # log packets that have a decoder/stream␣
↪event
        #    state-update: false            # log packets triggering a TCP state update
        #    spurious-retransmission: false  # log spurious retransmission packets
        #
heartbeat:
  # The output-flush-interval value governs how often Suricata will flush
  # EVE log file output. Specify the value in seconds [1-60] and Suricata will
  # flush all active EVE log files at that interval. A value of 0 means
  # no EVE log output flushes are performed. When the EVE output
  # buffer-size value is non-zero, some EVE output that was written may remain
  # buffered. The output-flush-interval governs how much buffered data exists.
  #
  # The default value is: 0 (no periodic flushing)
  #output-flush-interval: 0
```

For more advanced configuration options, see *Eve JSON Output*.

The format is documented in *Eve JSON Format*.

TLS parameters and certificates logging (tls.log)

Attention

tls-log is deprecated in Suricata 8.0 and will be removed in Suricata 9.0.

The TLS handshake parameters can be logged in a line based log as well. By default, the logfile is *tls.log* in the suricata log directory. See *Custom TLS logging* for details about the configuration and customization of the log format.

Furthermore there is an output module to store TLS certificate files to disk. This is similar to *File-store (File Extraction)*, but for TLS certificates.

Example:

```
# output module to store certificates chain to disk
- tls-store:
    enabled: yes
    #certs-log-dir: certs # directory to store the certificates files
```

Packet log (pcap-log)

With the pcap-log option you can save all packets, that are registered by Suricata, in a log file named _log.pcap_. This way, you can take a look at all packets whenever you want. In the normal mode a pcap file is created in the default-log-dir. It can also be created elsewhere if a absolute path is set in the yaml-file.

The file that is saved in example the `default-log-dir` */var/log/suricata*, can be be opened with every program which supports the pcap file format. This can be Wireshark, TCPdump, Suricata, Snort and many others.

The pcap-log option can be enabled and disabled.

There is a size limit for the pcap-log file that can be set. The default limit is 32 MB. If the log-file reaches this limit, the file will be rotated and a new one will be created. Remember that in the 'normal' mode, the file will be saved in default-log-dir or in the absolute path (if set).

The pcap files can be compressed before being written to disk by setting the compression option to lz4. Note: On Windows, this option increases disk I/O instead of reducing it. When using lz4 compression, you can enable checksums using the lz4-checksum option, and you can set the compression level lz4-level to a value between 0 and 16, where higher levels result in higher compression.

By default all packets are logged except:

- TCP streams beyond stream.reassembly.depth
- encrypted streams after the key exchange
- If a `bpf-filter` is set, packets that don't match the filter will not be logged

It is possible to do conditional pcap logging by using the *conditional* option in the pcap-log section. By default the variable is set to *all* so all packets are logged. If the variable is set to *alerts* then only the flow with alerts will be logged. If the variable is set to *tag* then only packets tagged by signatures using the *tag* keyword will be logged to the pcap file. Please note that if *alerts* or *tag* is used, then in the case of TCP session, Suricata will use available information from the streaming engine to log data that have triggered the alert.

```
- pcap-log:
    enabled:  yes
    filename: log.pcap

    # Limit in MB.
    limit: 32

    mode: normal # "normal" or multi
    conditional: alerts

    # A BPF filter that will be applied to all packets being
    # logged. If set, packets must match this filter otherwise they
```

(continues on next page)

(continued from previous page)

```
    # will not be logged.
    #bpf-filter:
```

In `normal` mode a pcap file "filename" is created in the default-log-dir or as specified by "dir". `normal` mode is generally not as performant as `multi` mode.

In multi mode, multiple pcap files are created (per thread) which performs better than `normal` mode.

In multi mode the filename takes a few special variables:

- %n representing the thread number
- %i representing the thread id
- %t representing the timestamp (secs or secs.usecs based on 'ts-format')

Example: filename: pcap.%n.%t

Note

It is possible to use directories but the directories are not created by Suricata. For example `filename: pcaps/%n/log.%s` will log into the pre-existing `pcaps` directory and per thread sub directories.

Note

that the limit and max-files settings are enforced per thread. So the size limit using 8 threads with 1000mb files and 2000 files is about 16TiB.

Verbose Alerts Log (alert-debug.log)

This is a log type that gives supplementary information about an alert. It is particularly convenient for people who investigate false positives and who write signatures. However, it lowers the performance because of the amount of information it has to store.

```
- alert-debug:                  #The log-name.
    enabled: no                 #This log is not enabled. Set 'yes' to enable.
    filename: alert-debug.log   #The name of the file in the default logging directory.
    append: yes/no              #If this option is set to yes, the last filled fast.log-
↪file will not be
                                # overwritten while restarting Suricata.
    # Compress IPv6 addresses per RFC5952 as they are added to the debug log log. The␣
↪default is no.
    ipv6-compress: yes/no
```

Stats

In stats you can set the options for stats.log. When enabling stats.log you can set the amount of time in seconds after which you want the output-data to be written to the log file.

```
- stats:
     enabled: yes               #By default, the stats-option is enabled
     filename: stats.log        #The log-name. Combined with the default logging␣
↪directory
```

(continues on next page)

(continued from previous page)

```
                                #(default-log-dir) it will result in /var/log/suricata/
↪stats.log.
                                #This directory can be overruled with a absolute path. (A
                                #directory starting with / ).
    append: yes/no              #If this option is set to yes, the last filled fast.log-
↪file will not be
                                #overwritten while restarting Suricata.
```

The interval and several other options depend on the global stats section as described above.

Syslog

> **Attention**
>
> The syslog output is deprecated in Suricata 8.0 and will be removed in Suricata 9.0. Please migrate to the `eve` output which has the ability to send to syslog.

With this option it is possible to send all alert and event output to syslog.

```
- syslog:                       #This is a output-module to direct log-output to several␣
↪directions.
    enabled: no                 #The use of this output-module is not enabled.
    facility: local5            #In this option you can set a syslog facility.
    level: Info                 #In this option you can set the level of output. The␣
↪possible levels are:
                                #Emergency, Alert, Critical, Error, Warning, Notice,␣
↪Info and Debug.
```

File-store (File Extraction)

The *file-store* output enables storing of extracted files to disk and configures where they are stored.

The following shows the configuration options for version 2 of the *file-store* output.

```
- file-store:
    # This configures version 2 of the file-store.
    version: 2

    enabled: no

    # Set the directory for the filestore. If the path is not
    # absolute will be be relative to the default-log-dir.
    #dir: filestore

    # Write out a fileinfo record for each occurrence of a
    # file. Disabled by default as each occurrence is already logged
    # as a fileinfo record to the main eve-log.
    #write-fileinfo: yes

    # Force storing of all files. Default: no.
    #force-filestore: yes
```

(continues on next page)

(continued from previous page)

```
    # Override the global stream-depth for sessions in which we want
    # to perform file extraction. Set to 0 for unlimited; otherwise,
    # must be greater than the global stream-depth value to be used.
    #stream-depth: 0

    # Uncomment the following variable to define how many files can
    # remain open for filestore by Suricata. Default value is 0 which
    # means files get closed after each write
    #max-open-files: 1000

    # Force logging of checksums, available hash functions are md5,
    # sha1 and sha256. Note that SHA256 is automatically forced by
    # the use of this output module as it uses the SHA256 as the
    # file naming scheme.
    #force-hash: [sha1, md5]
```

12.1.10 Detection engine

Inspection configuration

The detection-engine builds internal groups of signatures. Suricata loads signatures, with which the network traffic will be compared. The fact is, that many rules certainly will not be necessary. For instance, if there appears a packet with the UDP-protocol, all signatures for the TCP-protocol won't be needed. For that reason, all signatures will be divided in groups. However, a distribution containing many groups will make use of a lot of memory. Not every type of signature gets its own group. There is a possibility that different signatures with several properties in common, will be placed together in a group. The quantity of groups will determine the balance between memory and performance. A small number of groups will lower the performance yet use little memory. The opposite counts for a higher amount of groups. The engine allows you to manage the balance between memory and performance. To manage this, (by determining the amount of groups) there are several general options: `high` for good performance and more use of memory, `low` for low performance and little use of memory. The option `medium` is the balance between performance and memory usage. This is the default setting. The option `custom-values` is for advanced users. This option has values which can be managed by the user.

```
detect:
  profile: medium
  custom-values:
    toclient-groups: 3
    toserver-groups: 25
  sgh-mpm-context: auto
  inspection-recursion-limit: 3000
  stream-tx-log-limit: 4
  guess-applayer-tx: no
  grouping:
    tcp-priority-ports: 53, 80, 139, 443, 445, 1433, 3306, 3389, 6666, 6667, 8080
    udp-priority-ports: 53, 135, 5060
  flowbits:
    max-per-signature: 8
```

At all of these options, you can add (or change) a value. Most signatures have the adjustment to focus on one direction, meaning focusing exclusively on the server, or exclusively on the client.

If you take a look at example 4, *the Detection-engine grouping tree*, you see it has many branches. At the end of each

branch, there is actually a 'sig group head'. Within that sig group head there is a container which contains a list with signatures that are significant for that specific group/that specific end of the branch. Also within the sig group head the settings for Multi-Pattern-Matcher (MPM) can be found: the MPM-context.

As will be described again in *Pattern matcher settings*, there are several MPM-algorithms of which can be chosen from. Because every sig group head has its own MPM-context, some algorithms use a lot of memory. For that reason there is the option `sgh-mpm-context` to set whether the groups share one MPM-context, or to set that every group has its own MPM-context.

For setting the option sgh-mpm-context, you can choose from auto, full or single. The default setting is 'auto', meaning Suricata selects full or single based on the algorithm you use. 'Full' means that every group has its own MPM-context, and 'single' that all groups share one MPM-context. The algorithm "ac" uses a single MPM-context if the Sgh-MPM-context setting is 'auto'. The rest of the algorithms use full in that case.

The `inspection-recursion-limit` option has to mitigate that possible bugs in Suricata cause big problems. Often Suricata has to deal with complicated issues. It could end up in an 'endless loop' due to a bug, meaning it will repeat its actions over and over again. With the option inspection-recursion-limit you can limit this action.

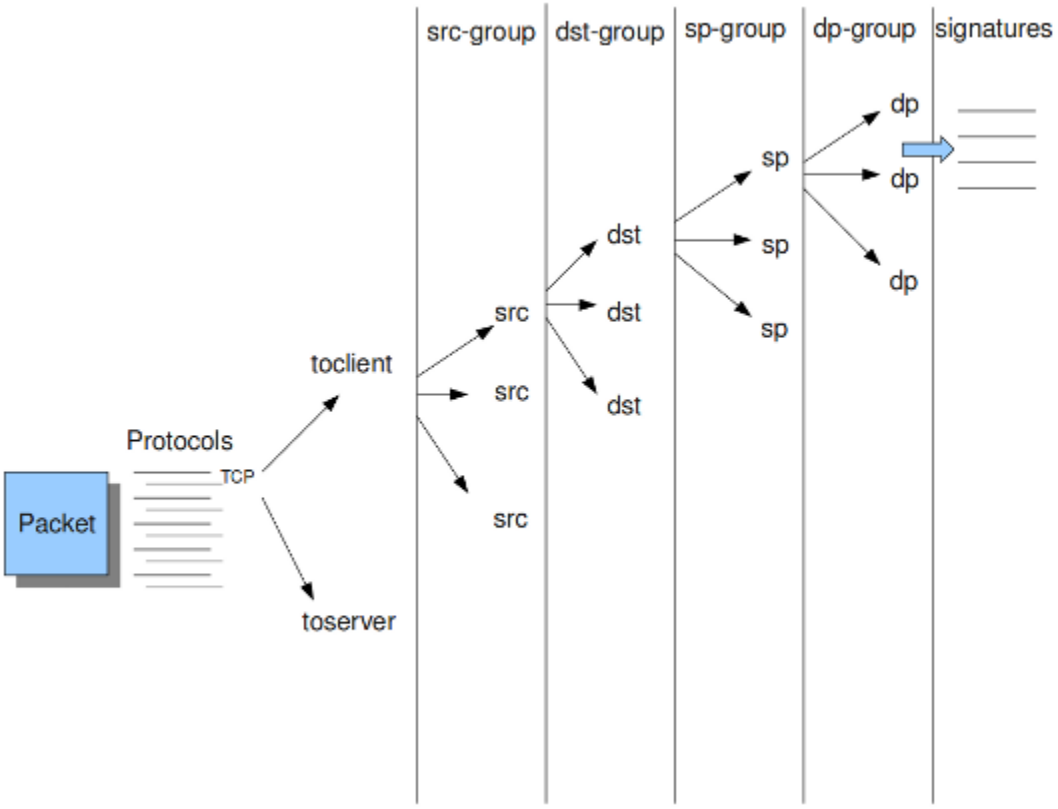
The `stream-tx-log-limit` defines the maximum number of times a transaction will get logged for rules without app-layer keywords. This is meant to avoid logging the same data an arbitrary number of times.

The `guess-applayer-tx` option controls whether the engine will try to guess and tie a transaction to a given alert if the matching signature doesn't have app-layer keywords. If enabled, AND ONLY ONE LIVE TRANSACTION EXISTS, that transaction's data will be added to the alert metadata. Note that this may not be the expected data, from an analyst's perspective.

The `grouping` option allows user to define the most seen ports on their network using `tcp-priority-ports` and `udp-priority-ports` settings to benefit from the internal signature groups created by Suricata. The engine shall then try to club the rules that use the ports defined in groups of their own and put them on top of the list of rules to be matched against traffic on "priority".

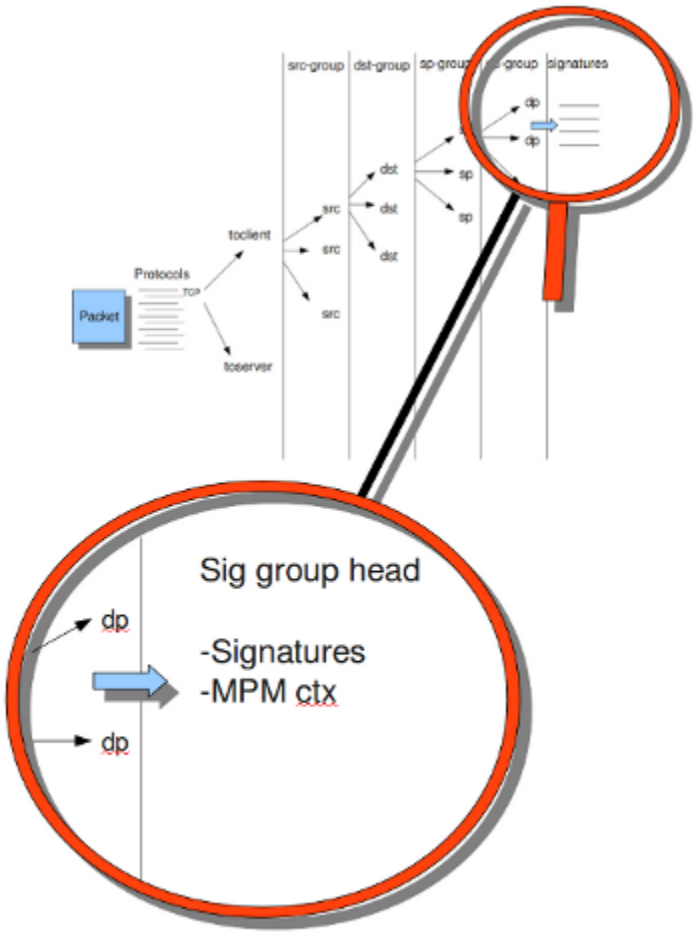
The `flowbits` option carries flowbits detection specific settings. With `max-per-signature` setting, it is possible to define how many times flowbits keyword can be seen in any one signature. This does not include `flowbits:noalert;`. Minimum value allowed is 1 and the default is 8.

Example 4 Detection-engine grouping tree




```
src             Stands for source IP-address.
dst             Stands for destination IP-address.
sp              Stands for source port.
dp              Stands for destination port.
```

Example 5 Detail grouping tree



Prefilter Engines

The concept of prefiltering is that there are far too many rules to inspect individually. The approach prefilter takes is that from each rule one condition is added to prefilter, which is then checked in one step. The most common example is MPM (also known as fast_pattern). This takes a single pattern per rule and adds it to the MPM. Only for those rules that have at least one pattern match in the MPM stage, individual inspection is performed.

Next to MPM, other types of keywords support prefiltering. ICMP itype, icode, icmp_seq and icmp_id for example. TCP window, IP TTL are other examples.

For a full list of keywords that support prefilter, see:

```
suricata --list-keywords=all
```

Suricata can automatically select prefilter options, or it can be set manually.

```
detect:
  prefilter:
    default: mpm
```

By default, only MPM/fast_pattern is used.

The prefilter engines for other non-MPM keywords can then be enabled in specific rules by using the 'prefilter' keyword.

E.g.

alert ip any any -> any any (ttl:123; prefilter; sid:1;)

To let Suricata make these decisions set default to 'auto':

```
detect:
  prefilter:
    default: auto
```

Thresholding Settings

Thresholding uses a central hash table for tracking thresholds of the types: by_src, by_dst, by_both.

```
detect:
  thresholds:
    hash-size: 16384
    memcap: 16mb
```

`detect.thresholds.hash-size` controls the number of hash rows in the hash table. `detect.thresholds.memcap` controls how much memory can be used for the hash table and the data stored in it.

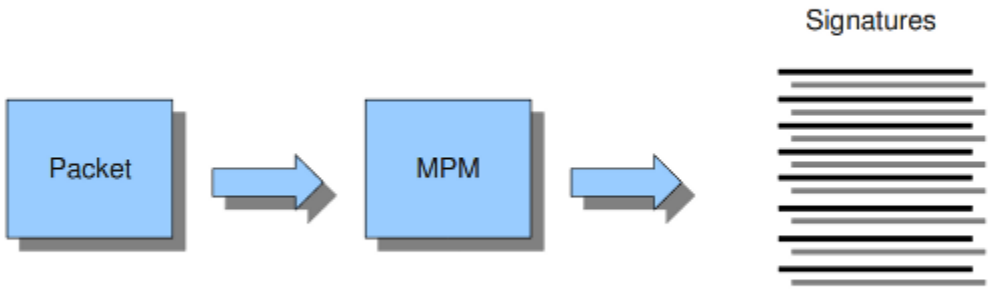
Pattern matcher settings

The multi-pattern-matcher (MPM) is a part of the detection engine within Suricata that searches for multiple patterns at once. Often, signatures have one or more patterns. Of each signature, one pattern is used by the multi-pattern-matcher. That way Suricata can exclude many signatures from being examined, because a signature can only match when all its patterns match.

These are the proceedings:

1) A packet comes in.
2) The packed will be analyzed by the Multi-pattern-matcher in search of patterns that match.
3) All patterns that match, will be further processed by Suricata (signatures).

Example 8 Multi-pattern-matcher



Suricata offers various implementations of different multi-pattern-matcher algorithm's. These can be found below.

To set the multi-pattern-matcher algorithm:

```
mpm-algo: ac
```

After 'mpm-algo', you can enter one of the following algorithms: ac, hs and ac-ks.

On *x86_64* hs (Hyperscan) should be used for best performance.

12.1.11 Threading

Suricata is multi-threaded. Suricata uses multiple CPUs/CPU cores so it can process a lot of network packets simultaneously. (In a single-core engine, the packets will be processed one at a time.)

There are four thread-modules: Packet acquisition, decode and stream application layer, detection, and outputs.

The packet acquisition module reads packets from the network.

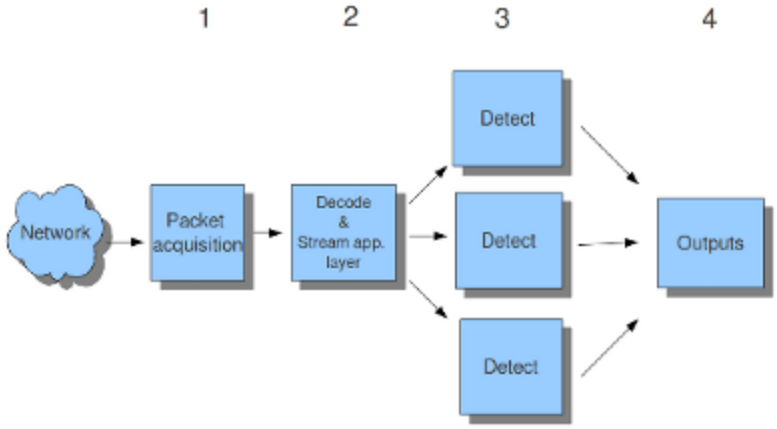
The decode module decodes the packets and the stream application application layer has three tasks:

```
First: it performs stream-tracking, meaning it is making sure all steps will be taken to␣
↪make a correct network-connection.
Second: TCP-network traffic comes in as packets. The Stream-Assembly engine reconstructs␣
↪the original stream.
Finally: the application layer will be inspected. HTTP and DCERPC will be analyzed.
```

The detection threads will compare signatures. There can be several detection threads so they can operate simultaneously.

In Outputs all alerts and events will be processed.

Example 6 Threading




```
Packet acquisition:             Reads packets from the network
Decode:                         Decodes packets.
Stream app. Layer:              Performs stream-tracking and reassembly.
Detect:                         Compares signatures.
Outputs:                        Processes all events and alerts.
```

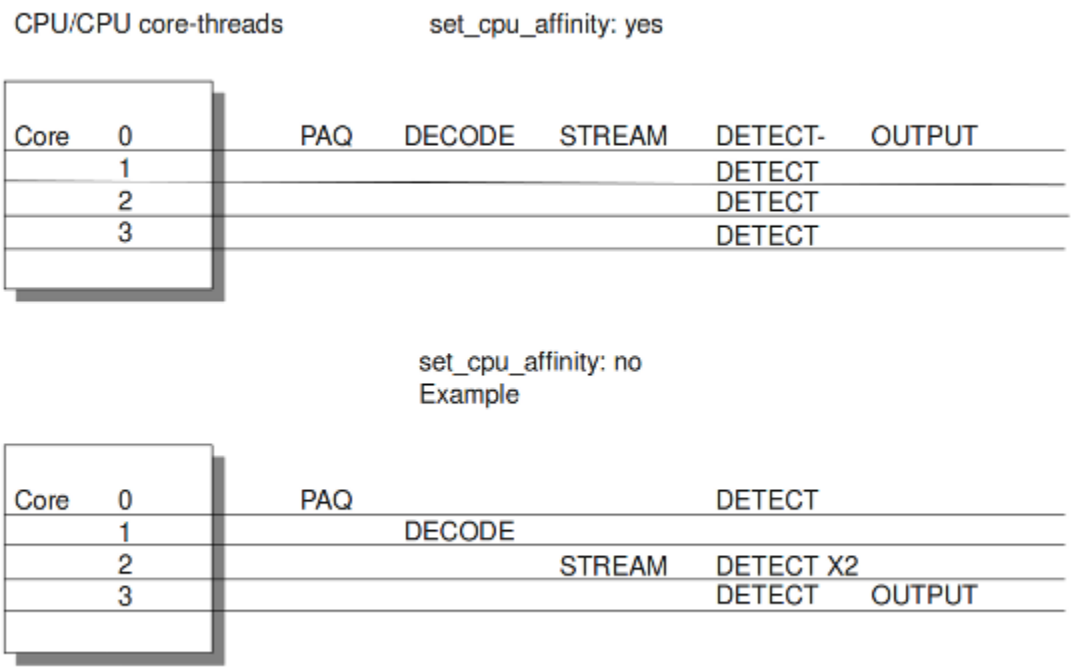
Most computers have multiple CPU's/ CPU cores. By default the operating system determines which core works on which thread. When a core is already occupied, another one will be designated to work on the thread. So, which core works on which thread, can differ from time to time.

There is an option within threading:

```
set-cpu-affinity: no
```

With this option you can cause Suricata setting fixed cores for every thread. In that case 1, 2 and 4 are at core 0 (zero). Each core has its own detect thread. The detect thread running on core 0 has a lower priority than the other threads running on core 0. If these other cores are to occupied, the detect thread on core 0 has not much packets to process. The detect threads running on other cores will process more packets. This is only the case after setting the option to 'yes'.

Example 7 Balancing workload



You can set the detect-thread-ratio:

```
detect-thread-ratio: 1.5
```

The detect thread-ratio will determine the amount of detect threads. By default it will be 1.5 x the amount of CPU's/CPU cores present at your computer. This will result in having more detection threads then CPU's/ CPU cores. Meaning you are oversubscribing the amount of cores. This may be convenient at times when there have to be waited for a detection thread. The remaining detection thread can become active.

You can alter the per-thread stack-size if the default provided by your build system is too small. The default value is provided by your build system; we suggest setting the value to 8MB if the default value is too small.

```
stack-size: 8MB
```

In the option 'cpu affinity' you can set which CPU's/cores work on which thread. In this option there are several sets of threads. The management-, receive-, worker- and verdict-set. These are fixed names and can not be changed. For each set there are several options: cpu, mode, and prio. In the option 'cpu' you can set the numbers of the CPU's/cores which will run the threads from that set. You can set this option to 'all', use a range (0-3) or a comma separated list (0,1). The option 'mode' can be set to 'balanced' or 'exclusive'. When set to 'balanced', the individual threads can be processed by all cores set in the option 'cpu'. If the option 'mode' is set to 'exclusive', there will be fixed cores for each thread. As mentioned before, threads can have different priority's. In the option 'prio' you can set a priority for each thread. This priority can be low, medium, high or you can set the priority to 'default'. If you do not set a priority for a CPU, than the settings in 'default' will count. By default Suricata creates one 'detect' (worker) thread per available CPU/CPU core.

Note

The 'prio' settings could overwrite each other, make sure to not include the same CPU core in different 'prio' settings.

```
threading:
  set-cpu-affinity: yes
  autopin: no
  cpu-affinity:
    management-cpu-set:
      cpu: [ 0 ]  # include only these cpus in affinity settings
    receive-cpu-set:
      cpu: [ 0 ]  # include only these cpus in affinity settings
    worker-cpu-set:
      cpu: [ "all" ]
      mode: "exclusive"
      # Use explicitly 3 threads and don't compute number by using
      # detect-thread-ratio variable:
      # threads: 3
      prio:
        low: [ 0 ]
        medium: [ "1-2" ]
        high: [ 3 ]
        default: "medium"
      interface-specific-cpu-set:
        - interface: "enp4s0f0" # 0000:3b:00.0 # net_bonding0 # ens1f0
          cpu: [ 1,3,5,7,9 ]
          mode: "exclusive"
          prio:
            high: [ "all" ]
            default: "medium"
    verdict-cpu-set:
      cpu: [ 0 ]
      prio:
        default: "high"
```

Relevant cpu-affinity settings for IDS mode

Runmode AutoFp:

```
management-cpu-set - used for management (example - flow.managers, flow.recyclers)
receive-cpu-set - used for receive and decode
worker-cpu-set - used for streamtcp,detect,output(logging),reject
```

Rumode Workers:

```
management-cpu-set - used for management (example - flow.managers, flow.recyclers)
worker-cpu-set - used for receive,streamtcp,decode,detect,output(logging),respond/reject
```

Relevant cpu-affinity settings for IPS mode

Runmode AutoFp:

```
management-cpu-set - used for management (example - flow.managers, flow.recyclers)
receive-cpu-set - used for receive and decode
worker-cpu-set - used for streamtcp,detect,output(logging)
verdict-cpu-set - used for verdict and respond/reject
```

Runmode Workers:

```
management-cpu-set - used for management (example - flow.managers, flow.recyclers)
worker-cpu-set - used for receive,streamtcp,decode,detect,output(logging),respond/reject,
↪ verdict
```

Interface-specific CPU affinity settings

Using the new configuration format introduced in Suricata 8.0 it is possible to set CPU affinity settings per interface. This can be useful when you have multiple interfaces and you want to dedicate specific CPU cores to specific interfaces. This can be useful, for example, when Suricata runs on multiple NUMA nodes and reads from interfaces on each NUMA node.

Interface-specific affinity settings can be configured for the `worker-cpu-set` and the `receive-cpu-set` (only used in autofp mode). This feature is available for capture modes which work with interfaces (af-packet, dpdk, etc.). The value of the interface key can be the kernel interface name (e.g. eth0 for af-packet), the PCI address of the interface (e.g. 0000:3b:00.0 for DPDK capture mode), or the name of the virtual device interface (e.g. net_bonding0 for DPDK capture mode). The interface names needs to be unique and be specified in the capture mode configuration.

The interface-specific settings will override the global settings for the `worker-cpu-set` and `receive-cpu-set`. The CPUs do not need to be contained in the parent node settings. If the interface-specific settings are not defined, the global settings will be used.

```
threading:
  set-cpu-affinity: yes
  cpu-affinity:
    worker-cpu-set:
      interface-specific-cpu-set:
        - interface: "eth0" # 0000:3b:00.0 # net_bonding0
          cpu: [ 1,3,5,7,9 ]
          mode: "exclusive"
          prio:
            high: [ "all" ]
            default: "medium"
```

Automatic NUMA-aware CPU core pinning

When Suricata is running on a system with multiple NUMA nodes, it is possible to automatically use CPUs from the same NUMA node as the network capture interface. CPU cores on the same NUMA node as the network capture interface can have reduced memory access latency and can increase the performance of Suricata. This is enabled by setting the `autopin` option to `yes` in the threading section. This option is available for worker-cpu-set and receive-cpu-set.

```
threading:
  set-cpu-affinity: yes
  autopin: yes
```

(continues on next page)

(continued from previous page)

```
  cpu-affinity:
    worker-cpu-set:
      cpu: [ "all" ]
      mode: "exclusive"
      prio:
        high: [ "all" ]
```

Consider 2 interfaces defined in the capture mode configuration, one on each NUMA node. The `autopin` option is enabled to automatically use CPUs from the same NUMA node as the interface. The worker-cpu-set is set to use all CPUs. When interface on the first NUMA node is used, the worker threads will be pinned to CPUs on the first NUMA node. When interface on the second NUMA node is used, the worker threads will be pinned to CPUs on the second NUMA node. If the number of CPU cores on a given NUMA node is exhausted then the worker threads will be pinned to CPUs on the other NUMA node.

The option `threading.autopin` can be combined with the interface-specific CPU affinity settings. To use the `autopin` option, the system must have the `hwloc` dependency installed and pass `--enable-hwloc` to the configure script.

12.1.12 IP Defrag

Occasionally network packets appear fragmented. On some networks it occurs more often than on others. Fragmented packets exist of many parts. Before Suricata is able to inspect these kind of packets accurately, the packets have to be reconstructed. This will be done by a component of Suricata; the defragment-engine. After a fragmented packet is reconstructed by the defragment-engine, the engine sends on the reassembled packet to rest of Suricata.

At the moment Suricata receives a fragment of a packet, it keeps in memory that other fragments of that packet will appear soon to complete the packet. However, there is a possibility that one of the fragments does not appear. To prevent Suricata for keeping waiting for that packet (thereby using memory) there is a timespan after which Suricata discards the fragments (timeout). This occurs by default after 60 seconds.

In IPS mode, it is possible to tell the engine what to do in case the memcap for the defrag engine is reached: "drop-packet", "pass-packet", or "ignore" (default behavior).

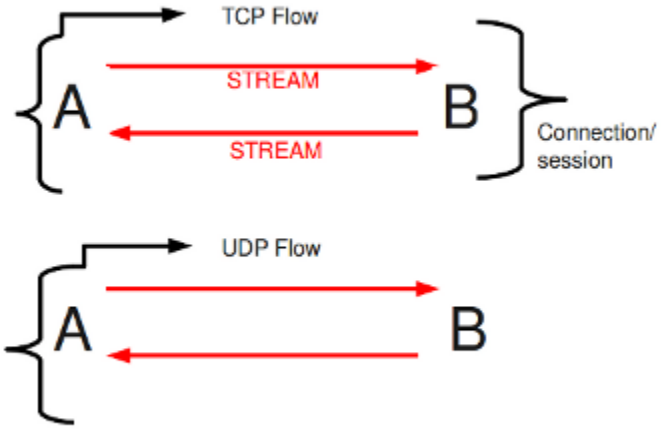
```
defrag:
  memcap: 32mb
  memcap-policy: ignore  # in IPS mode, what to do if memcap is reached
  hash-size: 65536
  trackers: 65535        # number of defragmented flows to follow
  max-frags: 65535       # number of fragments do keep (higher than trackers)
  prealloc: yes
  timeout: 60
```

12.1.13 Flow and Stream handling

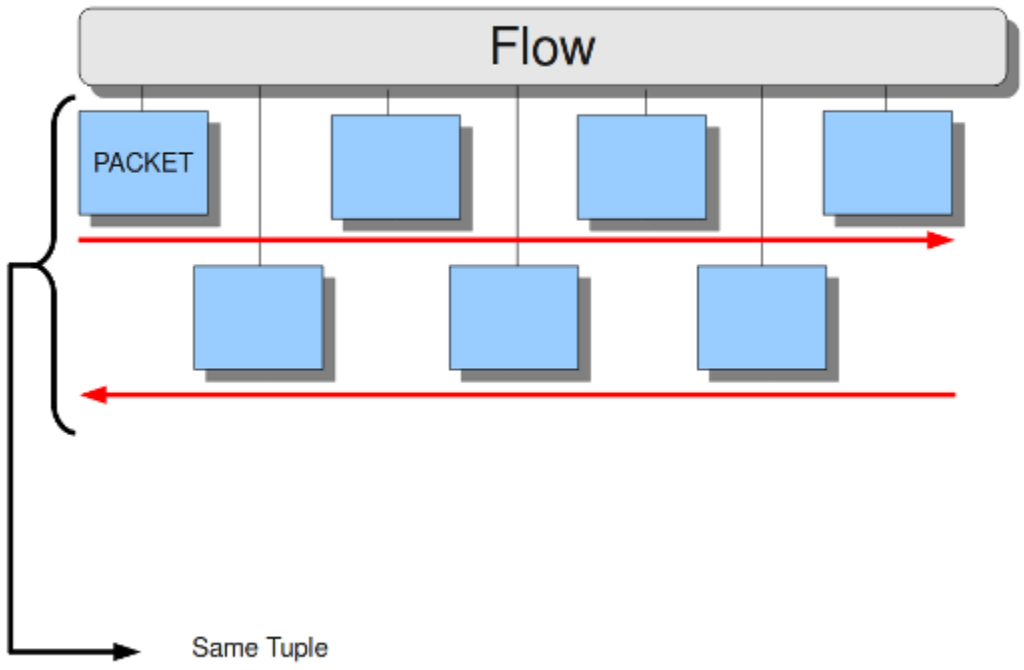
Flow Settings

Within Suricata, Flows are very important. They play a big part in the way Suricata organizes data internally. A flow is a bit similar to a connection, except a flow is more general. All packets having the same Tuple (protocol, source IP, destination IP, source-port, destination-port), belong to the same flow. Packets belonging to a flow are connected to it internally.

Example 9 Flow



Example 10 Tuple



Keeping track of all these flows, uses memory. The more flows, the more memory it will cost.

To keep control over memory usage, there are several options:

The option memcap for setting the maximum amount of bytes the flow-engine will use, hash-size for setting the size of the hash-table and prealloc for the following:

> For packets not yet belonging to a flow, Suricata creates a new flow. This is a relative expensive action. The risk coming with it, is that attackers /hackers can a attack the engine system at this part. When they make sure a computer gets a lot of packets with different tuples, the engine has to make a lot of new flows. This way, an attacker could flood the system. To mitigate the engine from being overloaded, this option instructs Suricata to keep a number of flows ready in memory. This way Suricata is less vulnerable to these kind of attacks.

The flow-engine has a management thread that operates independent from the packet processing. This thread is called the flow-manager. This thread ensures that wherever possible and within the memcap. There will be 10000 flows prepared.

In IPS mode, a memcap-policy exception policy can be set, telling Suricata what to do in case memcap is hit: 'drop-packet', 'pass-packet', 'reject', or 'ignore'.

```
flow:
  memcap: 33554432              #The maximum amount of bytes the flow-engine will make␣
↪use of.
  memcap-policy: bypass         #How to handle the flow if memcap is reached (IPS mode)
  hash-size: 65536              #Flows will be organized in a hash-table. With this␣
↪option you can set the
                                #size of the hash-table.
  prealloc: 10000               #The amount of flows Suricata has to keep ready in␣
↪memory.
  rate-tracking:                #Enable tracking of flows by the following rate␣
↪definition; mark them
                                #as elephant flows if they exceed the defined rate.␣
↪Disabled by default.
    bytes: 1GiB                 #Number of bytes to track
    interval: 10                #Time interval in seconds for which tracking should be␣
↪done
```

At the point the memcap will still be reached, despite prealloc, the flow-engine goes into the emergency-mode. In this mode, the engine will make use of shorter time-outs. It lets flows expire in a more aggressive manner so there will be more space for new Flows.

`emergency-recovery` defines the percentage of flows that the engine needs to prune before clearing the **emergency mode**. The default `emergency-recovery` value is 30. This is the percentage of prealloc'd flows after which the flow -engine will be back to normal (when 30 percent of the 10000 flows are completed).

> If during the **emergency-mode** the aggressive time-outs do not have the desired result, this option is the final resort. It ends some flows even if they have not reached their time-outs yet.

```
emergency-recovery: 30                  #Percentage of 10000 prealloc'd flows.
```

Flow Time-Outs

The amount of time Suricata keeps a flow in memory is determined by the Flow time-out.

There are different states in which a flow can be. Suricata distinguishes three flow-states for TCP and two for UDP. For TCP, these are: New, Established and Closed,for UDP only new and established. For each of these states Suricata can employ different timeouts.

The state new in a TCP-flow, means the period during the three way handshake. The state established is the state when the three way handshake is completed. The state closed in the TCP-flow: there a several ways to end a flow. This is by means of Reset or the Four-way FIN handshake.

New in a UDP-flow: the state in which packets are send from only one direction.

Established in a UDP-flow: packets are send from both directions.

In the example configuration the are settings for each protocol. TCP, UDP, ICMP and default (all other protocols).

```
flow-timeouts:

  default:
    new: 30                     #Time-out in seconds after the last activity in this␣
↪flow in a New state.
    established: 300            #Time-out in seconds after the last activity in this␣
↪flow in a Established
                                #state.
    emergency-new: 10           #Time-out in seconds after the last activity in this␣
↪flow in a New state
                                #during the emergency mode.
    emergency-established: 100  #Time-out in seconds after the last activity in this␣
↪flow in a Established
                                #state in the emergency mode.
  tcp:
    new: 60
    established: 3600
    closed: 120
    emergency-new: 10
    emergency-established: 300
    emergency-closed: 20
  udp:
    new: 30
    established: 300
    emergency-new: 10
    emergency-established: 100
  icmp:
    new: 30
    established: 300
    emergency-new: 10
    emergency-established: 100
```

Stream-engine

The Stream-engine keeps track of the TCP-connections. The engine exists of two parts: The stream tracking- and the reassembly-engine.

The stream-tracking engine monitors the state of a connection. The reassembly-engine reconstructs the flow as it used to be, so it will be recognized by Suricata.

The stream-engine has two memcaps that can be set. One for the stream-tracking-engine and one for the reassembly-engine. For both cases, in IPS mode, an exception policy (memcap-policy) can be set, telling Suricata what to do in case memcap is hit: 'drop-flow', 'drop-packet', 'pass-flow', 'pass-packet', 'bypass', 'reject', or 'ignore'.

The stream-tracking-engine keeps information of the flow in memory. Information about the state, TCP-sequence-numbers and the TCP window. For keeping this information, it can make use of the capacity the memcap allows.

TCP packets have a so-called checksum. This is an internal code which makes it possible to see if a packet has arrived in a good state. The stream-engine will not process packets with a wrong checksum. This option can be set off by entering 'no' instead of 'yes'.

```
stream:
  memcap: 64mb                # Max memory usage (in bytes) for TCP session tracking
  memcap-policy: ignore       # In IPS mode, call memcap policy if memcap is reached
  checksum-validation: yes    # Validate packet checksum, reject packets with invalid␣
↪checksums.
```

To mitigate Suricata from being overloaded by fast session creation, the option prealloc-sessions instructs Suricata to keep a number of sessions ready in memory.

A TCP-session starts with the three-way-handshake. After that, data can be sent and received. A session can last a long time. It can happen that Suricata will be started after a few TCP sessions have already been started. This way, Suricata misses the original setup of those sessions. This setup always includes a lot of information. If you want Suricata to check the stream from that time on, you can do so by setting the option 'midstream' to 'true'. The default setting is 'false'. In IPS mode, it is possible to define a 'midstream-policy', indicating whether Suricata should drop-flow, drop-packet, pass-flow, pass-packet, reject, or bypass a midstream flow. The default is ignore. Normally Suricata is able to see all packets of a connection. Some networks make it more complicated though. Some of the network-traffic follows a different route than the other part, in other words: the traffic goes asynchronous. To make sure Suricata will check the one part it does see, instead of getting confused, the option 'async-oneside' is brought to life. By default the option is set to 'false'.

Suricata inspects content in the normal/IDS mode in chunks. In the inline/IPS mode it does that on the sliding window way (see example ..) In the case Suricata is set in inline mode, it has to inspect packets immediately before sending it to the receiver. This way Suricata is able to drop a packet directly if needed.(see example . . .) It is important for Suricata to note which operating system it is dealing with, because operating systems differ in the way they process anomalies in streams. See *Host-os-policy*.

```
prealloc-sessions: 32768     # 32k sessions prealloc'd
midstream: false             # do not allow midstream session pickups
midstream-policy: drop-flow  # in IPS mode, drop flows that start midstream
async-oneside: false         # do not enable async stream handling
inline: no                   # stream inline mode
drop-invalid: yes            # drop invalid packets
bypass: no
```

The `drop-invalid` option can be set to no to avoid blocking packets that are seen invalid by the streaming engine. This can be useful to cover some weird cases seen in some layer 2 IPS setup.

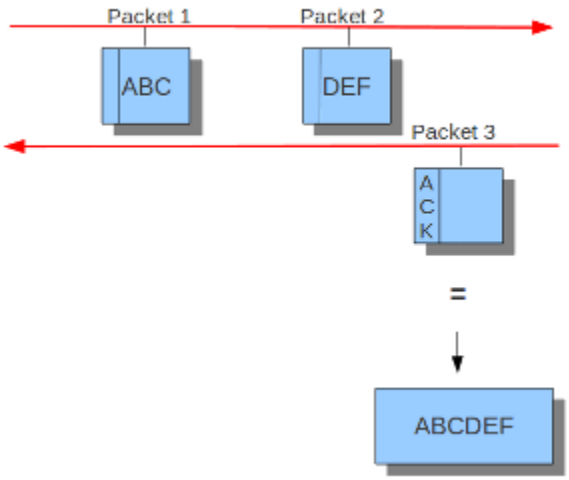
The `bypass` option activates 'bypass' for a flow/session when either side of the session reaches its `depth`.

Warning

`bypass` can lead to missing important traffic. Use with care.

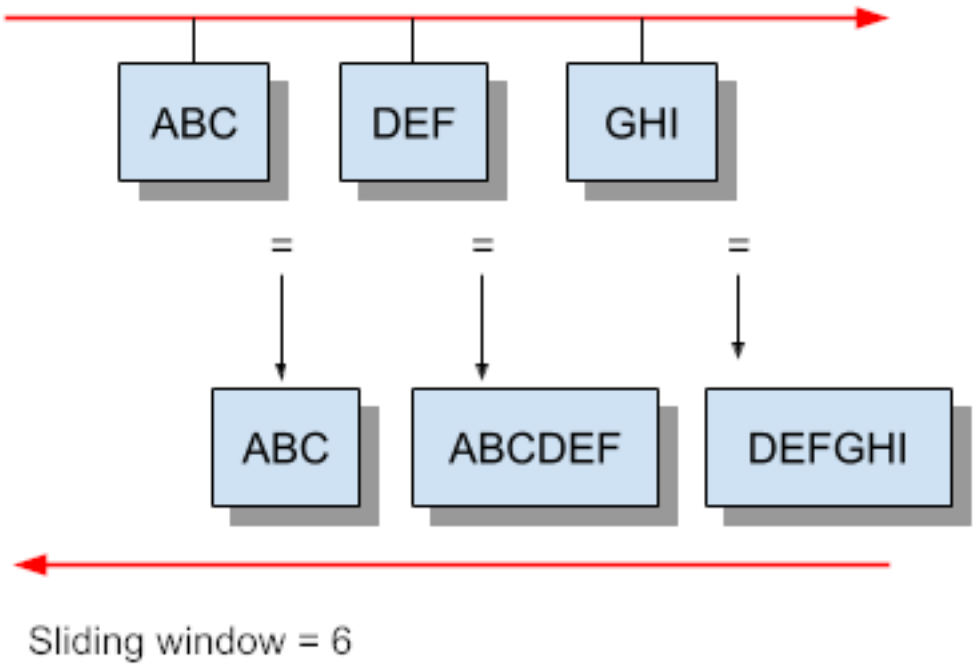
Example 11 Normal/IDS mode

Suricata inspects traffic in chunks.

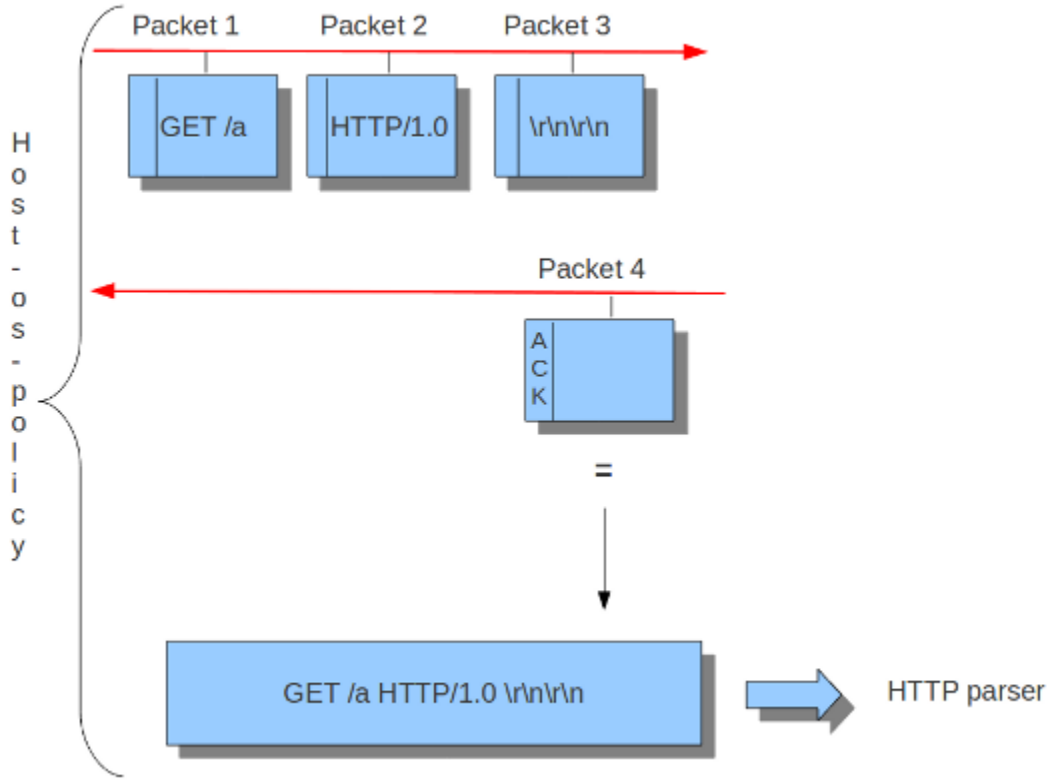


Example 12 Inline/IPS Sliding Window

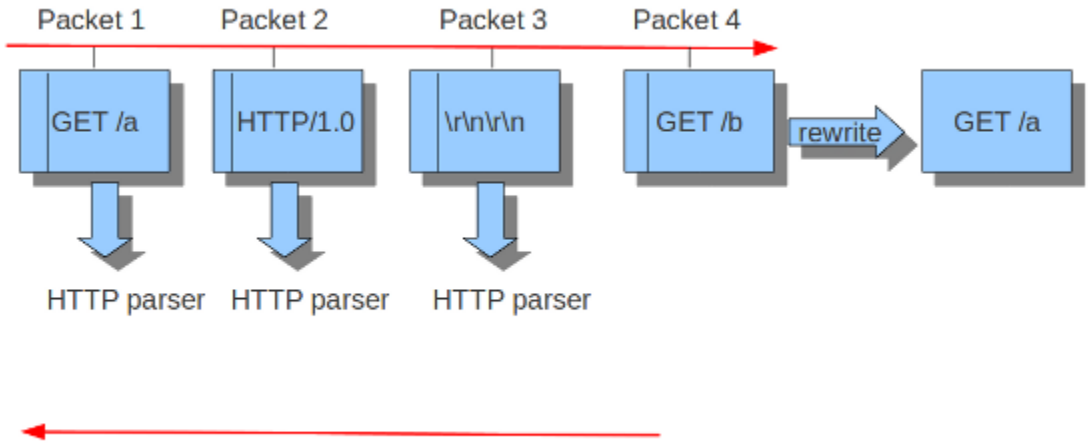
Suricata inspects traffic in a sliding window manner.



Example 13 Normal/IDS (reassembly on ACK'D data)



Example 14 Inline/IPS (reassembly on UNACK'D data)



The reassembly-engine has to keep data segments in memory in order to be able to reconstruct a stream. To avoid resource starvation a memcap is used to limit the memory used. In IPS mode, an exception policy (memcap-policy) can be set, telling Suricata what to do in case memcap is hit: 'drop-flow', 'drop-packet', 'pass-flow', 'pass-packet', 'bypass', 'reject', or 'ignore'.

Reassembling a stream is an expensive operation. With the option depth you can control how far into a stream re-

assembly is done. By default this is 1MB. This setting can be overridden per stream by the protocol parsers that do file extraction.

Inspection of reassembled data is done in chunks. The size of these chunks is set with `toserver-chunk-size` and `toclient-chunk-size`. To avoid making the borders predictable, the sizes can be varied by adding in a random factor.

```
reassembly:
  memcap: 256mb             # Memory reserved for stream data reconstruction (in bytes)
  memcap-policy: ignore     # What to do when memcap for reassembly is hit
  depth: 1mb                # The depth of the reassembling.
  toserver-chunk-size: 2560 # inspect raw stream in chunks of at least this size
  toclient-chunk-size: 2560 # inspect raw stream in chunks of at least
  randomize-chunk-size: yes
  #randomize-chunk-range: 10
```

'Raw' reassembly is done for inspection by simple `content`, `pcre` keywords use and other payload inspection not done on specific protocol buffers like `http_uri`. This type of reassembly can be turned off:

```
reassembly:
  raw: no
```

Incoming segments are stored in a list in the stream. To avoid constant memory allocations a per-thread pool is used.

```
reassembly:
  segment-prealloc: 2048    # pre-alloc 2k segments per thread
```

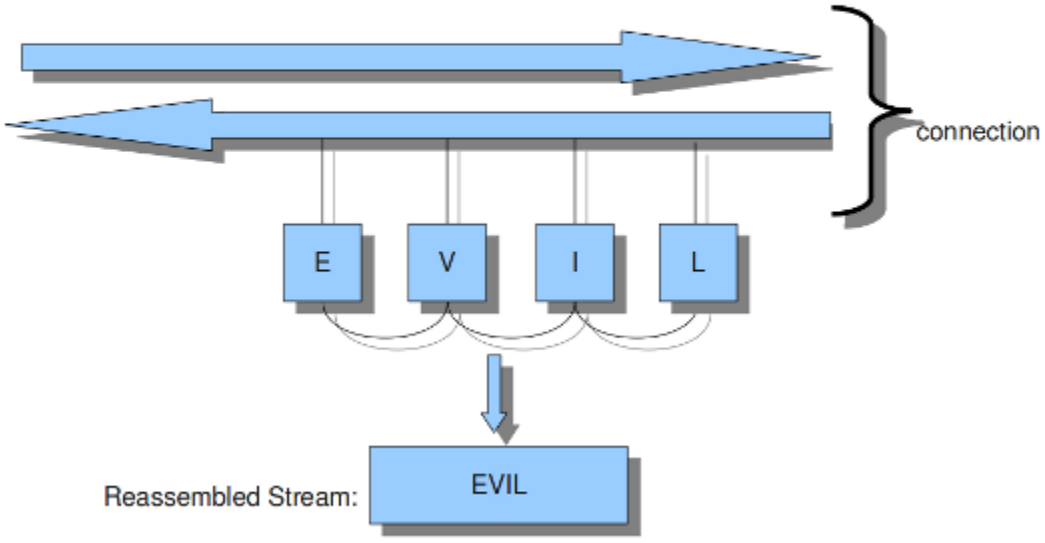
Resending different data on the same sequence number is a way to confuse network inspection.

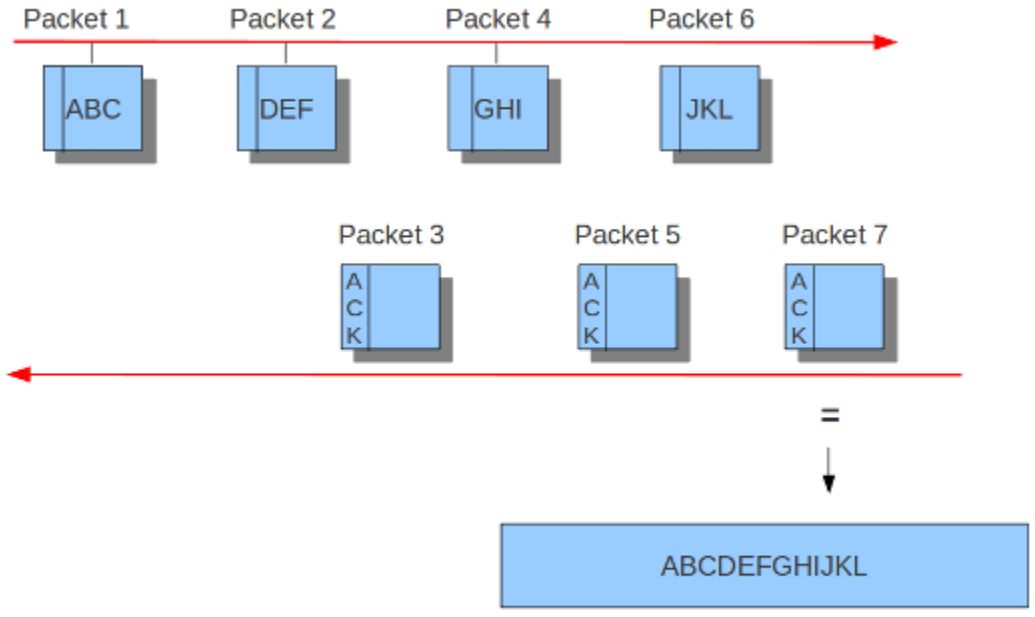
```
reassembly:
  check-overlap-different-data: true
```

Example 15 Stream reassembly

Stream Reassembly

Signature: EVIL





toserver_chunk_size: 10

TCP Urgent Handling

TCP Urgent pointer support is a complicated topic, where it is essentially impossible for a network device to know with certainty what the behavior of the receiving host is.

For this reason, many middleboxes strip the URG flag and reset the urgent pointer (see for example RFC 6093, 3.4).

Several options are provided to control how to deal with the urgent pointer.

```
stream:
  reassembly:
  urgent:
    policy: oob               # drop, inline, oob (1 byte, see RFC 6093, 3.1), gap
    oob-limit-policy: drop
```

***stream.reassembly.urgent.policy*:**

- *drop*: drop URG packets before they affect the stream engine
- *inline*: ignore the urgent pointer and process all data inline
- *oob* (out of band): treat the last byte as out of band
- ***gap*: skip the last byte, but do no adjust sequence offsets, leading to** gaps in the data

If the urgent policy is set to *oob*, there is an additional setting. Since OOB data does advance the TCP sequence number, the stream engine tracks the number of bytes to make sure no GAPs in the non-OOB data are seen by the app-layer parsers and detection engine. This is currently limited to 64k per direction. If the number of OOB bytes exceeds that 64k, an additional policy is triggered: *stream.reassembly.urgent.oob-limit-policy*.

stream.reassembly.urgent.oob-limit-policy: - *drop*: drop URG packets before they affect the stream engine - *inline*: ignore the urgent pointer and process all data inline - *gap*: skip the last byte, but do no adjust sequence offsets, leading to gaps in the data

Observables

Each packet with the URG flag set, will increment the *tcp.urg* counter.

When dropping the URG packets, the packets will have the drop reason *ips.drop_reason.stream_urgent*, which is also a counter in the stats logging.

The stream event *stream-event:reassembly_urgent_oob_limit_reached* allows matching on the packet that reaches the OOB limit. Stream rule *2210066* matches on this.

If *stats.stream-events* are enabled the counter *stream.reassembly_urgent_oob_limit_reached* will be incremented if the OOB limit is reached.

12.1.14 Host Tracking

The Host table is used for tracking per IP address. This is used for tracking per IP thresholding, per IP tagging, storing *iprep* data and storing *hostbit*.

Settings

The configuration allows specifying the following settings: *hash-size*, *prealloc* and *memcap*.

```
host:
  hash-size: 4096
```

(continues on next page)

(continued from previous page)

```
  prealloc: 1000
  memcap: 32mb
```

- *hash-size*: size of the hash table in number of rows
- *prealloc*: number of *Host* objects preallocated for efficiency
- *memcap*: max memory use for hosts, including the hash table size

Hosts are evicted from the hash table by the Flow Manager thread when all data in the host is expired (tag, threshold, etc). Hosts with iprep will not expire.

12.1.15 Application Layer Parsers

The `app-layer` section holds application layer specific configurations.

In IPS mode, a global exception policy accessed via the `error-policy` setting can be defined to indicate what the engine should do in case it encounters an app-layer error. Possible values are "drop-flow", "pass-flow", "bypass", "drop-packet", "pass-packet", "reject" or "ignore" (which maintains the default behavior).

Each supported protocol has a dedicated subsection under `protocols`.

> **Note**
>
> All applayer parsers can be enabled or disabled for specific carrier protocols. Suricata first looks for carrier protocol specific setting and if not found, falls back to the common enabled setting. e.g. if `sip` is being registered, Suricata will first look if `app-layer.protocols.sip.tcp.enabled` and `app-layer.protocols.sip.udp.enabled` are set. If not, then a search would be made for `app-layer.protocols.sip.enabled` and that setting would apply to both SIP/TCP as well as SIP/UDP.

Asn1_max_frames

Asn1 (Abstract Syntax One) is a standard notation to structure and describe data.

Within Asn1-max-frames there are several frames. To protect itself, Suricata will inspect a maximum of 256. You can set this amount differently if wanted.

Application layer protocols such as X.400 electronic mail, X.500 and LDAP directory services, H.323 (VoIP), BACnet and SNMP, use ASN.1 to describe the protocol data units (PDUs) they exchange. It is also extensively used in the Access and Non-Access Strata of UMTS.

Limit for the maximum number of asn1 frames to decode (default 256):

```
asn1-max-frames: 256
```

FTP

The FTP application layer parser is enabled by default and uses dynamic protocol detection.

By default, FTP control channel commands and responses are limited to 4096 bytes, but this value can be changed. When a command request or response exceeds the line length limit, the stored data will be truncated, however the parser will continue to watch for the end of line and acquire the next command. Commands that are truncated will be noted in the *eve* log file with the fields `command_truncated` or `reply_truncated`. Please note that this affects the control messages only, not FTP data (file transfers).

```
ftp:
  enabled: yes
  #memcap: 64mb

  # Maximum line length for control messages before they will be truncated.
  #max-line-length: 4kb

  # Maximum number of preliminary (1xx) responses kept per transaction.
  #max-preliminary-responses: 16
```

`memcap` limits the memory that FTP counts for control-channel transactions, requests and replies, and for FTP-DATA file buffering. It defaults to 64 MiB and is shared by all FTP flows. Once it is reached, new FTP allocations fail: affected flows stop being parsed and file extraction can be truncated. A value of `0` removes the limit.

A transaction keeps at most `max-preliminary-responses` preliminary (`1xx`) responses, 16 by default. Further preliminary responses for the same command are not stored or logged, and the `ftp.too_many_preliminary_responses` event is raised on the transaction. `max-preliminary-responses` must be greater than 0; an invalid value keeps the default.

Configure HTTP (libhtp)

The library Libhtp is being used by Suricata to parse HTTP-sessions.

While processing HTTP-traffic, Suricata has to deal with different kind of servers which each process anomalies in HTTP-traffic differently. The most common web-server is Apache. This is an open source web-server program.

Besides Apache, IIS (Internet Information Services/Server) a web-server program of Microsoft is also well-known.

Like with host-os-policy, it is important for Suricata to know which IP-address/network-address is used by which server. In Libhtp this assigning of web-servers to IP-and network addresses is called personality.

Currently Available Personalities:

- Minimal
- Generic
- IDS (default)
- IIS_4_0
- IIS_5_0
- IIS_5_1
- IIS_6_0
- IIS_7_0
- IIS_7_5
- Apache
- Apache_2_2

You can assign names to each block of settings. Which in this case is -apache and -iis7. Under these names you can set IP-addresses, network-addresses the personality and a set of features.

The version-specific personalities know exactly how web servers behave, and emulate that. The IDS personality would try to implement a best-effort approach that would work reasonably well in the cases where you do not know the specifics.

The default configuration also applies to every IP-address for which no specific setting is available.

HTTP request bodies are often big, so they take a lot of time to process which has a significant impact on the performance. With the option 'request-body-limit' you can set the limit (in bytes) of the client-body that will be inspected. Setting it to 0 will inspect all of the body.

The same goes for HTTP response bodies.

```
libhtp:

  default-config:
    personality: IDS
    request-body-limit: 3072
    response-body-limit: 3072

  server-config:
     - apache:
         address: [192.168.1.0/24, 127.0.0.0/8, "::1"]
         personality: Apache_2_2
         request-body-limit: 0
         response-body-limit: 0

     - iis7:
         address:
           - 192.168.0.0/24
           - 192.168.10.0/24
         personality: IIS_7_0
         request-body-limit: 4096
         response-body-limit: 8192
```

Suricata makes available the whole set of libhtp customisations for its users.

You can now use these parameters in the conf to customise suricata's use of libhtp.

```
# Configures whether backslash characters are treated as path segment
# separators. They are not on Unix systems, but are on Windows systems.
# If this setting is enabled, a path such as "/one\two/three" will be
# converted to "/one/two/three".  Accepted values - yes, no.
#path-convert-backslash-separators: yes

# Configures whether input data will be converted to lowercase.
#path-convert-lowercase: yes

# Configures how the server reacts to encoded NUL bytes.
#path-nul-encoded-terminates: no

# Configures how the server reacts to raw NUL bytes.
#path-nul-raw-terminates: no

# Configures whether consecutive path segment separators will be
# compressed. When enabled, a path such as "/one//two" will be normalized
# to "/one/two". The backslash_separators and decode_separators
# parameters are used before compression takes place. For example, if
# backslash_separators and decode_separators are both enabled, the path
# "/one\\/two\/%5cthree/%2f//four" will be converted to
# "/one/two/three/four".  Accepted values - yes, no.
#path-separators-compress: yes
```

(continues on next page)

(continued from previous page)

```
# Configures whether encoded path segment separators will be decoded.
# Apache does not do this, but IIS does. If enabled, a path such as
# "/one%2ftwo" will be normalized to "/one/two". If the
# backslash_separators option is also enabled, encoded backslash
# characters will be converted too (and subsequently normalized to
# forward slashes).  Accepted values - yes, no.
#path-separators-decode: yes

# Configures whether %u-encoded sequences in path will be decoded. Such
# sequences will be treated as invalid URL encoding if decoding is not
# desireable.  Accepted values - yes, no.
#path-u-encoding-decode: yes

# Configures how server reacts to invalid encoding in path.  Accepted
# values - preserve_percent, remove_percent, decode_invalid, status_400
#path-url-encoding-invalid-handling: preserve_percent

# Controls whether the data should be treated as UTF-8 and converted
# to a single-byte stream using best-fit mapping
#path-utf8-convert-bestfit:yes

# Sets the replacement character that will be used to in the lossy
# best-fit mapping from Unicode characters into single-byte streams.
# The question mark is the default replacement character.
#path-bestfit-replacement-char: ?

# Configures whether plus characters are converted to spaces
# when decoding URL-encoded strings.
#query-plusspace-decode: yes

#   response-body-decompress-layer-limit:
#                           Limit to how many layers of compression will be
#                           decompressed. Defaults to 2.

#   uri-include-all:        Include all parts of the URI. By default the
#                           'scheme', username/password, hostname and port
#                           are excluded.

#   meta-field-limit:       Hard size limit for request and response size
#                           limits.

# inspection limits
    request-body-minimal-inspect-size: 32kb
    request-body-inspect-window: 4kb
    response-body-minimal-inspect-size: 40kb
    response-body-inspect-window: 16kb

# auto will use http-body-inline mode in IPS mode, yes or no set it statically
    http-body-inline: auto

# Decompress SWF files.
```

(continues on next page)

(continued from previous page)

```
# 2 types: 'deflate', 'lzma', 'both' will decompress deflate and lzma
# compress-depth:
# Specifies the maximum amount of data to decompress,
# set 0 for unlimited. The maximum is 4294967282 bytes.
# decompress-depth:
# Specifies the maximum amount of decompressed data to obtain. The
# output buffer grows on demand up to this limit. Set 0 to use the
# default limit of 50 MB. The maximum is 2147483640 bytes.
    swf-decompression:
      enabled: yes
      type: both
      compress-depth: 0
      decompress-depth: 0

# Take a random value for inspection sizes around the specified value.
# This lower the risk of some evasion technics but could lead
# detection change between runs. It is set to 'yes' by default.
#randomize-inspection-sizes: yes
# If randomize-inspection-sizes is active, the value of various
# inspection size will be chosen in the [1 - range%, 1 + range%]
# range
# Default value of randomize-inspection-range is 10.
#randomize-inspection-range: 10

# Can enable LZMA decompression
#lzma-enabled: false
# Memory limit usage for LZMA decompression dictionary
# Data is decompressed until dictionary reaches this size
#lzma-memlimit: 1 Mb
# Maximum decompressed size with a compression ratio
# above 2048 (only reachable by LZMA)
#compression-bomb-limit: 1 Mb
# Maximum time spent decompressing a single transaction in usec
#decompression-time-limit: 100000
# Maximum number of live transactions per flow
#max-tx: 512
# Maximum used number of HTTP1 headers in one request or response
#headers-limit: 1024
```

Other parameters are customizable from Suricata.

```
#   double-decode-path:     Double decode path section of the URI
#   double-decode-query:    Double decode query section of the URI
```

decompression-time-limit

decompression-time-limit was implemented to avoid DOS by resource exhaustion on inputs such as decompression bombs (found by fuzzing). The lower the limit, the better the protection against DOS is, but this may also lead to false positives. In case the time limit is reached, the app-layer event `http.compression_bomb` is set (this event can also set from other conditions). This can happen on slow configurations (hardware, ASAN, etc...)

Configure SMB

The SMB parser will parse version 1, 2 and 3 of the SMB protocol over TCP.

To enable the parser add the following to the `app-layer` section of the YAML.

```
smb:
  enabled: yes
  detection-ports:
    dp: 139, 445
```

The parser uses pattern based protocol detection and will fallback to `probing parsers` if the pattern based detection fails. As usual, the pattern based detection is port independent. The `probing parsers` will only run on the `detection-ports`.

SMB is commonly used to transfer the DCERPC protocol. This traffic is also handled by this parser.

Resource limits

Several options are available for limiting record sizes and data chunk tracking.

```
smb:
  enabled: yes
  max-read-size: 8mb
  max-write-size: 1mb

  max-read-queue-size: 16mb
  max-read-queue-cnt: 16

  max-write-queue-size: 16mb
  max-write-queue-cnt: 16

  dcerpc:
    max-stub-size: 1MiB
```

The *max-read-size* option can be set to control the max size of accepted READ records. Events will be raised if a READ request asks for too much data and/or if READ responses are too big. A value of 0 disables the checks.

The *max-write-size* option can be set to control the max size of accepted WRITE request records. Events will be raised if a WRITE request sends too much data. A value of 0 disables the checks.

Additionally if the *max-read-size* or *max-write-size* values in the "negotiate protocol response" exceeds this limit an event will also be raised.

To control the size of the DCERPC stub data, *dcerpc.max-stub-size* should be used. It is by default set to 1MiB.

For file tracking, extraction and file data inspection the parser queues up out of order data chunks for both READs and WRITEs. To avoid using too much memory the parser allows for limiting both the size in bytes and the number of queued chunks.

```
smb:
  enabled: yes

  max-read-queue-size: 16mb
  max-read-queue-cnt: 16

  max-write-queue-size: 16mb
  max-write-queue-cnt: 16
```

max-read-queue-size controls how many bytes can be used per SMB flow for out of order READs. *max-read-queue-cnt* controls how many READ chunks can be queued per SMB flow. Processing of these chunks will be blocked when any of the limits are exceeded, and an event will be raised.

max-write-queue-size and *max-write-queue-cnt* are as the READ variants, but then for WRITEs.

Cache limits

The SMB parser uses several per flow caches to track data between different records and transactions. These caches have a size ceiling. When the size limit is reached, new additions will automatically evict the oldest entries.

```
smb:
  max-guid-cache-size: 1024
  max-rec-offset-cache-size: 128
  max-tree-cache-size: 512
  max-dcerpc-frag-cache-size: 128
  max-session-cache-size: 512
```

The *max-guid-cache-size* setting controls the size of the hash that maps the GUID to filenames. These are added through CREATE commands and removed by CLOSE commands.

max-rec-offset-cache-size controls the size of the hash that maps the READ offset from READ commands to the READ responses.

The *max-tree-cache-size* option contols the size of the SMB session to SMB tree hash.

max-dcerpc-frag-cache-size controls the size of the hash that tracks partial DCERPC over SMB records. These are buffered in this hash to only parse the DCERPC record when it is fully reassembled.

The *max-session-cache-size* setting controls the size of a generic hash table that maps SMB session to filenames, GUIDs and share names.

Configure NFS

Resource limits

Several options are available for limiting record sizes and queued out of order file data.

```
nfs:
  enabled: yes
  max-read-size: 16mb
  max-write-size: 16mb
  max-metadata-size: 16mb

  max-read-queue-size: 64mb
  max-read-queue-cnt: 64

  max-write-queue-size: 64mb
  max-write-queue-cnt: 64
```

The *max-read-size* option can be set to control the max size of accepted READ responses. Events will be raised if READ responses claim or carry too much data. A value of 0 disables the checks.

The *max-write-size* option can be set to control the max size of accepted WRITE request records. Events will be raised if a WRITE request claims too much data. A value of 0 disables the checks.

The *max-metadata-size* option can be set to control how much of a partial NFSv3 LOOKUP, READDIRPLUS or CREATE reply may be buffered towards the size its RPC record claims, so the file handle to filename mapping is kept.

A value of 0 falls back to the default of 16mb: this limit cannot be disabled. It is separate from *max-read-size*, which only bounds READ data.

For file tracking the parser queues up out of order file data per file transaction. To avoid using too much memory the parser allows for limiting both the size in bytes and the number of queued chunks.

```
nfs:
  enabled: yes

  max-read-queue-size: 64mb
  max-read-queue-cnt: 64

  max-write-queue-size: 64mb
  max-write-queue-cnt: 64
```

max-read-queue-size controls how many bytes can be queued per file for out of order READ data. *max-read-queue-cnt* controls how many chunks can be queued per file. When a limit is exceeded the data is discarded and an event is raised.

max-write-queue-size and *max-write-queue-cnt* are as the READ variants, but then for WRITEs.

Configure DCERPC

DCERPC has one parameter that can be customized. *max-stub-size* is used to control the stub data size of a DCERPC request/response. By default, it is set to 1MiB.

Configure HTTP2

HTTP2 has 2 parameters that can be customized. The point of these 2 parameters is to find a balance between the completeness of analysis and the resource consumption.

http2.max-table-size refers to *SETTINGS_HEADER_TABLE_SIZE* from rfc 7540 section 6.5.2. Its default value is 4096 bytes, but it can be set to any uint32 by a flow.

http2.max-streams refers to *SETTINGS_MAX_CONCURRENT_STREAMS* from rfc 7540 section 6.5.2. Its default value is 256.

SSL/TLS

SSL/TLS parsers track encrypted SSLv2, SSLv3, TLSv1, TLSv1.1 and TLSv1.2 sessions.

Protocol detection is done using patterns and a probing parser running on only TCP/443 by default. The pattern based protocol detection is port independent.

```
tls:
  enabled: yes
  detection-ports:
    dp: 443

  # What to do when the encrypted communications start:
  # - track-only: keep tracking TLS session, check for protocol anomalies,
  #               inspect tls_* keywords. Disables inspection of unmodified
  #               'content' signatures.
  # - bypass:  stop processing this flow as much as possible. No further
  #               TLS parsing and inspection. Offload flow bypass to kernel
  #               or hardware if possible.
  # - full:    keep tracking and inspection as normal. Unmodified content
  #               keyword signatures are inspected as well.
```

(continues on next page)

(continued from previous page)

```
  #
  # For the best performance, select 'bypass'.
  #
  #encryption-handling: track-only
```

Encrypted traffic

There is no decryption of encrypted traffic, so once the handshake is complete continued tracking of the session is of limited use. The `encryption-handling` option in `app-layer.protocols.tls` and `app-layer.protocols.ssh` controls the behavior after the handshake.

If the `encryption-handling` property of the TLS/SSH configuration nodes are set to `track-only` (or are not set), Suricata will continue to track the respective SSL/TLS or SSH session. Inspection will be limited, as raw `content` inspection will still be disabled. There is no point in doing pattern matching on traffic known to be encrypted. Inspection for (encrypted) Heartbleed and other protocol anomalies still happens.

When `encryption-handling` is set to `bypass`, all processing of this session is stopped. No further parsing and inspection happens. This will also lead to the flow being bypassed, either inside Suricata or by the capture method if it supports it and is configured for it.

Finally, if `encryption-handling` is set to `full`, Suricata will process the flow as normal, without inspection limitations or bypass.

The option has replaced the `no-reassemble` option. If `no-reassemble` is present, and `encryption-handling` is not, `false` is interpreted as `encryption-handling: track-only` and `true` is interpreted as `encryption-handling: bypass`.

SSH

Besides `encryption-handling`, ssh parser offers the `hassh` option with 3 values - yes : enables hassh logging - auto : hassh be enabled if rules use hassh keywords - no : disables hassh and will refuse to load rules that use hassh keywords

Modbus

According to MODBUS Messaging on TCP/IP Implementation Guide V1.0b, it is recommended to keep the TCP connection opened with a remote device and not to open and close it for each MODBUS/TCP transaction. In that case, it is important to set the stream-depth of the modbus as unlimited.

```
modbus:
  # Stream reassembly size for modbus, default is 0
  stream-depth: 0
```

MQTT

The maximum size of a MQTT message is 256MB, potentially containing a lot of payload data (such as properties, topics, or published payloads) that would end up parsed and logged. To acknowledge the fact that most MQTT messages, however, will be quite small and to reduce the potential for denial of service issues, it is possible to limit the maximum length of a message that Suricata should parse. Any message larger than the limit will just be logged with reduced metadata, and rules will only be evaluated against a subset of fields. The default is 1 MB.

```
mqtt:
  max-msg-length: 1mb
```

RFB

RFB can have some strings whose maximum length according to the RFC is 4GiB. In order to limit ram consumption and log output, a configuration parameter `max-string-length` is available. This limit will also apply during detection. An event `rfb.too_long_string` will be emitted when a string exceeds the limit. The default is 4 KiB.

```
rfb:
  max-string-length: 4 KiB
```

SMTP

SMTP parsers can extract files from attachments. It is also possible to extract raw conversations as files with the key `raw-extraction`. Note that in this case the whole conversation will be stored as a file, including SMTP headers and body content. The filename will be set to "rawmsg". Usual file-related signatures will match on the raw content of the email. This configuration parameter has a `false` default value. It is incompatible with `decode-mime`. If both are enabled, `raw-extraction` will be automatically disabled.

```
smtp:
  # extract messages in raw format from SMTP
  raw-extraction: true
```

Maximum transactions

SMTP, MQTT, FTP, PGSQL, SMB, DCERPC, HTTP1, ENIP and NFS have each a *max-tx* parameter that can be customized. *max-tx* refers to the maximum number of live transactions for each flow. An app-layer event *protocol.too_many_transactions* is triggered when this value is reached. The point of this parameter is to find a balance between the completeness of analysis and the resource consumption.

When this threshold is reached, a new transaction will not be allocated, and the flow will be put in error state, as it would become too expensive in terms of CPU for Suricata to continue processing it.

For HTTP2, this parameter is named *max-streams* as an HTTP2 stream will get translated into one Suricata transaction. This configuration parameter is used whatever the value of *SETTINGS_MAX_CONCURRENT_STREAMS* negotiated between a client and a server in a specific flow is.

Maximum Responses

LDAP and PGSQL each have a *max-responses* parameter that can be customized. *max-responses* refers to the maximum number of responses a single protocol transaction can hold. An app-layer event *protocol.too_many_responses* is triggered when this value is reached. As with *max-transactions* the point of this setting is to find a balance between visibility and resource consumption. When this cap is reached, new responses to the same transaction will not be pushed nor logged, but the parser will keep inspecting further transactions normally.

> **Note**
>
> For PGSQL, the *too_many_responses* event is issued only once per transaction.

12.1.16 Engine Logging

The engine logging system logs information about the application such as errors and other diagnostic information during startup, runtime and shutdown of the Suricata engine. This does not include Suricata generated alerts and events.

The engine logging system has the following log levels:

- `error`

- warning
- notice
- info
- perf
- config
- debug

Note that debug level logging will only be emitted if Suricata was compiled with the `--enable-debug` configure option.

The first option within the logging configuration is the `default-log-level`. This option determines the severity/importance level of information that will be displayed. Messages of lower levels than the one set here, will not be shown. The default setting is `Notice`. This means that `error`, `warning` and `notice` will be shown and messages for the other levels won't be.

Default Configuration Example

```
# Logging configuration.  This is not about logging IDS alerts/events, but
# output about what Suricata is doing, like startup messages, errors, etc.
logging:
  # The default log level, can be overridden in an output section.
  # Note that debug level logging will only be emitted if Suricata was
  # compiled with the --enable-debug configure option.
  #
  # This value is overridden by the SC_LOG_LEVEL env var.
  default-log-level: notice

  # The default output format.  Optional parameter, should default to
  # something reasonable if not provided.  Can be overridden in an
  # output section.  You can leave this out to get the default.
  #
  # This console log format value can be overridden by the SC_LOG_FORMAT env var.
  #default-log-format: "%D: %S: %M"
  #
  # For the pre-7.0 log format use:
  #default-log-format: "[%i] %t [%S] - (%f:%l) <%d> (%n) -- "

  # A regex to filter output.  Can be overridden in an output section.
  # Defaults to empty (no filter).
  #
  # This value is overridden by the SC_LOG_OP_FILTER env var.
  default-output-filter:

  # Define your logging outputs.  If none are defined, or they are all
  # disabled you will get the default - console output.
  outputs:
  - console:
      enabled: yes
      # type: json
  - file:
      enabled: yes
      level: info
```

(continues on next page)

(continued from previous page)

```
    filename: suricata.log
    # format: "[%i - %m] %z %d: %S: %M"
    # type: json
  - syslog:
    enabled: no
    facility: local5
    format: "[%i] <%d> -- "
    # type: json
```

Default Log Level

Example:

```
logging:
  default-log-level: info
```

This option sets the default log level. The default log level is *notice*. This value will be used in the individual logging configuration (console, file, syslog) if not otherwise set.

Note

The `-v` command line option can be used to quickly increase the log level at runtime. See *the -v command line option*.

The `default-log-level` set in the configuration value can be overridden by the `SC_LOG_LEVEL` environment variable.

Default Log Format

A logging line exists of two parts. First it displays meta information (Log-level, Suricata module), and finally the actual log message. Example:

```
i: suricata: This is Suricata version 7.0.2 RELEASE running in USER mode
```

(Here the part until the second *:* is the meta info, "This is Suricata version 7.0.2 RELEASE running in USER mode" is the actual message.)

It is possible to determine which information will be displayed in this line and (the manner how it will be displayed) in which format it will be displayed. This option is the so called format string:

```
default-log-format: "[%i] %t - (%f:%l) <%d> (%n) -- "
```

The % followed by a character has a special meaning. There are thirteen specified signs:

```
z:      ISO-like formatted timestamp: YYYY-MM-DD HH:MM:SS
t:      Original Suricata log timestamp: DD/MM/YYYY -- HH:MM::SS
p:      Process ID. Suricata's whole processing consists of multiple threads.
i:      Thread ID. ID of individual threads.
m:      Thread module name. (Outputs, Detect etc.)
d:      Log-level of specific log-event. (Error, info, debug etc.)
D:      Compact log format (E for Error, i for info etc.)
S:      Subsystem name.
```

(continues on next page)

(continued from previous page)

```
T:      Thread name.
M:      Log message body.
f:      Name of source code filename where log-event is generated.
l:      Line-number within the source filename, where the log-event is generated.
n:      Function-name in the source code.
```

The last three options, f, l and n, are mainly convenient for developers.

The log-format can be overridden in the command line by the environment variable: `SC_LOG_FORMAT`.

Output Filter

Within logging you can set an output-filter. With this output-filter you can set which part of the event-logs should be displayed. You can supply a regular expression (Regex). A line will be shown if the regex matches.

```
default-output-filter:    # In this option the regular expression can be entered.
```

This value is overridden by the environment var: `SC_LOG_OP_FILTER`.

Logging Outputs

There are different ways of displaying output. The output can appear directly on your screen, it can be placed in a file or via syslog. The last mentioned is an advanced tool for log-management. The tool can be used to direct log-output to different locations (files, other computers etc.)

```
outputs:
  - console:                          # Output to screen (stdout/stderr).
      enabled: yes                    # This option is enabled.
      #level: notice                  # Use a different level than the default.
  - file:                             # Output stored in a file.
      enabled: no                     # This option is not enabled.
      filename: /var/log/suricata.log # Filename and location on disc.
      level: info                     # Use a different level than the default.
  - syslog:                           # Output using syslog.
      enabled: no                     # The use of this program is not enabled.
      facility: local5                # Syslog facility to use.
      format: "[%i] <%d> -- "         # Output format specific to syslog.
      #level: notice                  # Use a different level than the default.
```

12.1.17 Packet Acquisition

Data Plane Development Kit (DPDK)

Data Plane Development Kit is a framework for fast packet processing in data plane applications running on a wide variety of CPU architectures. DPDK's Environment Abstraction Layer (EAL) provides a generic interface to low-level resources. It is a unique way how DPDK libraries access NICs. EAL creates an API for an application to access NIC resources from the userspace level. In DPDK, packets are not retrieved via interrupt handling. Instead, the application polls the NIC for newly received packets.

DPDK allows the user space application to directly access memory where the NIC stores the packets. As a result, neither DPDK nor the application copies the packets for the inspection. The application directly processes packets via passed packet descriptors.

To use DPDK capture module, Suricata must be compiled with DPDK option enabled. Support for DPDK can be enabled in configure step of the build process such as:

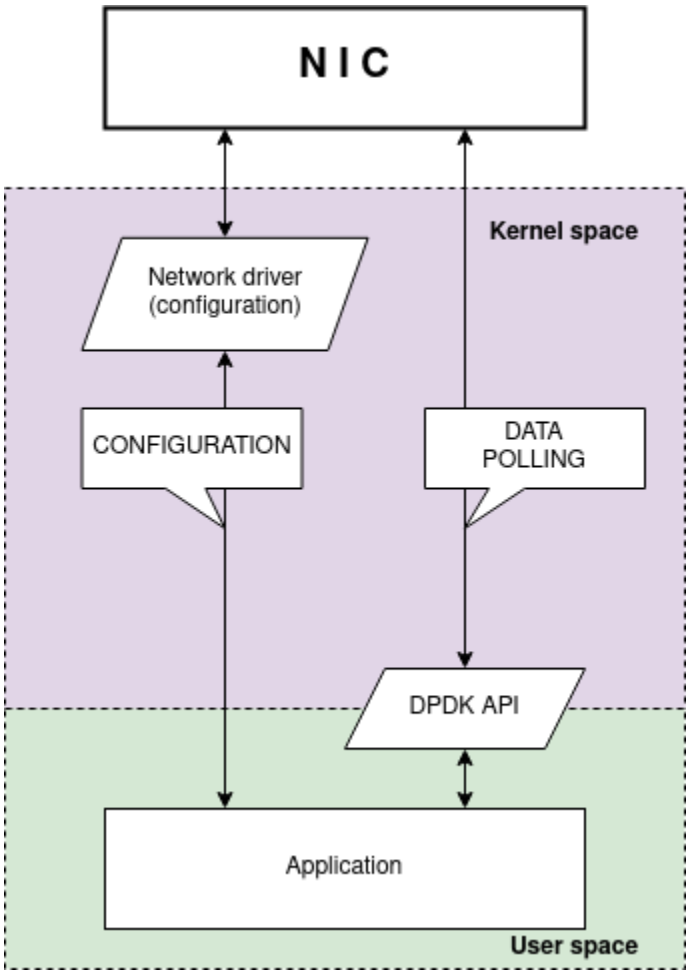


Fig. 1: *High-level overview of DPDK application*

```
./configure --enable-dpdk
```

Suricata makes use of DPDK for packet acquisition in workers runmode. The whole DPDK configuration resides in the `dpdk:` node. This node encapsulates 2 main subnodes, and those are eal-params and interfaces.

```
dpdk:
  eal-params:
    proc-type: primary
    allow: ["0000:3b:00.0", "0000:3b:00.1"]
  interfaces:
    - interface: 0000:3b:00.0
      threads: auto
      promisc: true
      multicast: true
      checksum-checks: true
      checksum-checks-offload: true
      vlan-strip-offload: true
      linkup-timeout: 10
      mtu: 1500
      mempool-size: auto
      mempool-cache-size: auto
      rx-descriptors: auto
      tx-descriptors: auto
      copy-mode: none
      copy-iface: none # or PCIe address of the second interface
```

The DPDK arguments, which are typically provided through the command line, are contained in the node `dpdk.eal-params`. EAL is configured and initialized using these parameters. There are two ways to specify arguments: lengthy and short. Dashes are omitted when describing the arguments. This setup node can be used to set up the memory configuration, accessible NICs, and other EAL-related parameters, among other things. The node `dpdk.eal-params` also supports multiple arguments of the same type. This can be useful for EAL arguments such as `--vdev`, `--allow`, or `--block`. Values for these EAL arguments are specified as a comma-separated list. An example of such usage can be found in the example above where the `allow` argument only makes `0000:3b:00.0` and `0000:3b:00.1` accessible to Suricata. arguments with list node. such as --vdev, --allow, --block eal options. The definition of lcore affinity as an EAL parameter is a standard practice. However, lcore parameters like `-l`, `-c`, and `--lcores` are specified within the *suricata-yaml-threading* section to prevent configuration overlap.

The node `dpdk.interfaces` wraps a list of interface configurations. Items on the list follow the structure that can be found in other capture interfaces. The individual items contain the usual configuration options such as `threads` / `copy-mode` / `checksum-checks` settings. Other capture interfaces, such as AF_PACKET, rely on the user to ensure that NICs are appropriately configured. Configuration through the kernel does not apply to applications running under DPDK. The application is solely responsible for the initialization of the NICs it is using. So, before the start of Suricata, the NICs that Suricata uses, must undergo the process of initialization. As a result, there are extra configuration options (how NICs can be configured) in the items (interfaces) of the `dpdk.interfaces` list. At the start of the configuration process, all NIC offloads are disabled to prevent any packet modification. According to the configuration, checksum validation offload can be enabled to drop invalid packets. Other offloads can not currently be enabled. Additionally, the list items in `dpdk.interfaces` contain DPDK specific settings such as `mempool-size` or `rx-descriptors`. These settings adjust individual parameters of EAL. One of the entries in `dpdk.interfaces` is the `default` interface. When loading interface configuration and some entry is missing, the corresponding value of the `default` interface is used.

The worker threads must be assigned to specific cores. The configuration module `threading` must be used to set thread affinity. Worker threads can be pinned to cores in the array configured in `threading.cpu-affinity["worker-cpu-set"]`. Performance-oriented setups have everything (the NIC, memory, and CPU cores interacting with the NIC) based on one NUMA node. It is therefore required to know the layout of the server architecture to get the best results. The CPU core ids and NUMA locations can be determined for example from the output

of `/proc/cpuinfo` where `physical id` described the NUMA number. The NUMA node to which the NIC is connected to can be determined from the file `/sys/class/net/<KERNEL NAME OF THE NIC>/device/numa_node`.

```
## Check ids and NUMA location of individual CPU cores
cat /proc/cpuinfo | grep 'physical id\|processor'

## Check NUMA node of the NIC
## cat /sys/class/net/<KERNEL NAME OF THE NIC>/device/numa_node e.g.
cat /sys/class/net/eth1/device/numa_node
```

Suricata operates in workers runmode. Packet distribution relies on Receive Side Scaling (RSS), which distributes packets across the NIC queues. Individual Suricata workers then poll packets from the NIC queues. Internally, DPDK runmode uses a symmetric hash (0x6d5a) that redirects bi-flows to specific workers. Each worker operates on 1 RX (and 1 TX) queue. The number of RX queues is always equal to the number of threads/workers. The number of TX queues is the same as the number of RX queues or can be set to 0 if Suricata runs in IDS mode by configuring `tx-descriptors` to 0 or `auto` in the interface configuration node.

Before Suricata can be run, it is required to allocate a sufficient number of hugepages. For efficiency, hugepages are continuous chunks of memory (pages) that are larger (2 MB+) than what is typically used in the operating systems (4 KB). A lower count of pages allows faster lookup of page entries. The hugepages need to be allocated on the NUMA node where the NIC and affiniated CPU cores reside. For example, if the hugepages are allocated only on NUMA node 0 and the NIC is connected to NUMA node 1, then the application will fail to start. As a result, it is advised to identify the NUMA node to which the NIC is attached before allocating hugepages and setting CPU core affinity to that node. In case Suricata deployment uses multiple NICs, hugepages must be allocated on each of the NUMA nodes used by the Suricata deployment.

```
## To check number of allocated hugepages:
sudo dpdk-hugepages.py -s
# alternative (older) way
grep Huge /proc/meminfo

## Allocate 2 GB in hugepages on all available NUMA nodes:
# (number of hugepages depend on the default size of hugepages 2 MB / 1 GB)
sudo dpdk-hugepages.py --setup 2G
# alternative (older) way allocates 1024 2 MB hugepages but only on NUMA 0
echo 1024 | sudo tee \
  /sys/devices/system/node/node0/hugepages/hugepages-2048kB/nr_hugepages
```

DPDK memory pools hold packets received from NICs. These memory pools are allocated in hugepages. Each Suricata worker has independently allocated memory pools per interface. The total size of all mempools of the interface is set with the `mempool-size`. The recommend size of the memory pool can be auto-calculated by setting `mempool-size: auto`. If `mempool-size` is set manually (to e.g. `mempool-size: 65536`), the value is divided by the number of worker cores of the interface (on 4 worker threads, each worker is assigned with a mempool containing 16383 packet objects). Memory (in bytes) for interface's memory pools is calculated as: `mempool-size` * `mtu`. The sum of memory pool requirements divided by the size of one hugepage results in the number of required hugepages. It causes no problem to allocate more memory than required, but it is vital for Suricata to not run out of hugepages.

The mempool cache is local to the individual CPU cores and holds packets that were recently processed. The recommended size of the cache can be auto-calculated by setting `mempool-cache-size: auto`.

To be able to run DPDK on Intel cards, it is required to change the default Intel driver to either `vfio-pci` or `igb_uio` driver. The process is described in DPDK manual page regarding Linux drivers. The Intel NICs have the amount of RX/TX descriptors capped at 4096. This should be possible to change by manually compiling the DPDK while changing the value of respective macros for the desired drivers (e.g. IXGBE_MAX_RING_DESC/I40E_MAX_RING_DESC). DPDK is natively supported by Mellanox and thus their NICs should work "out of the box".

Current DPDK support involves Suricata running on:

- **a physical machine with a physical NICs such as:**
  - mlx5 (ConnectX-4/ConnectX-5/ConnectX-6)
  - ixgbe
  - i40e
  - ice
- **a virtual machine with virtual interfaces such as:**
  - e1000
  - VMXNET3
  - virtio-net

Other NICs using the same driver as mentioned above should work as well. The DPDK capture interface has not been tested neither with the virtual interfaces nor in the virtual environments like VMs, Docker or similar.

The minimal supported DPDK is version 19.11 which should be available in most repositories of major distributions. Alternatively, it is also possible to use `meson` and `ninja` to build and install DPDK from source files. It is required to have correctly configured tool `pkg-config` as it is used to load libraries and CFLAGS during the Suricata configuration and compilation. This can be tested by querying DPDK version as:

```
pkg-config --modversion libdpdk
```

Pf-ring

The Pf_ring is a library that aims to improve packet capture performance over libcap. It performs packet acquisition. There are three options within Pf_ring: interface, cluster-id and cluster-type.

```
pfring:
  interface: eth0    # In this option you can set the network-interface
                     # on which you want the packets of the network to be read.
```

Pf_ring will load balance packets based on flow. All packet acquisition threads that will participate in the load balancing need to have the same cluster-id. It is important to make sure this ID is unique for this cluster of threads, so that no other engine / program is making use of clusters with the same id.

```
cluster-id: 99
```

Pf_ring can load balance traffic using pf_ring-clusters. All traffic for pf_ring can be load balanced according to the configured cluster type value; in a round robin manner or a per flow manner that are part of the same cluster. All traffic for pf_ring will be load balanced across acquisition threads of the same cluster id.

The "inner" flow means that the traffic will be load balanced based on address tuple after the outer vlan has been removed.

Cluster Type	Value
cluster_flow	src ip, src_port, dst ip, dst port, proto, vlan
cluster_inner_flow	src ip, src port, dst ip, dst port, proto, vlan
cluster_inner_flow_2_tuple	src ip, dst ip
cluster_inner_flow_4_tuple	src ip, src port, dst ip, dst port
cluster_inner_flow_5_tuple	src ip, src port, dst ip, dst port, proto
cluster_round_robin	not recommended

The cluster_round_robin manner is a way of distributing packets one at a time to each thread (like distributing playing cards to fellow players). The cluster_flow manner is a way of distributing all packets of the same flow to the same thread. The flows itself will be distributed to the threads in a round-robin manner.

If your deployment has VLANs, the cluster types with "inner" will use the innermost address tuple for distribution.

The default cluster type is `cluster_flow`; the `cluster_round_robin` is not recommended with Suricata.

```
cluster-type: cluster_inner_flow_5_tuple
```

NFQ

Using NFQUEUE in iptables rules, will send packets to Suricata. If the mode is set to 'accept', the packet that has been send to Suricata by a rule using NFQ, will by default not be inspected by the rest of the iptables rules after being processed by Suricata. There are a few more options to NFQ to change this if desired.

If the mode is set to 'repeat', the packets will be marked by Suricata and be re-injected at the first rule of iptables. To mitigate the packet from being going round in circles, the rule using NFQ will be skipped because of the mark.

If the mode is set to 'route', you can make sure the packet will be send to another tool after being processed by Suricata. It is possible to assign this tool at the mandatory option 'route_queue'. Every engine/tool is linked to a queue-number. This number you can add to the NFQ rule and to the route_queue option.

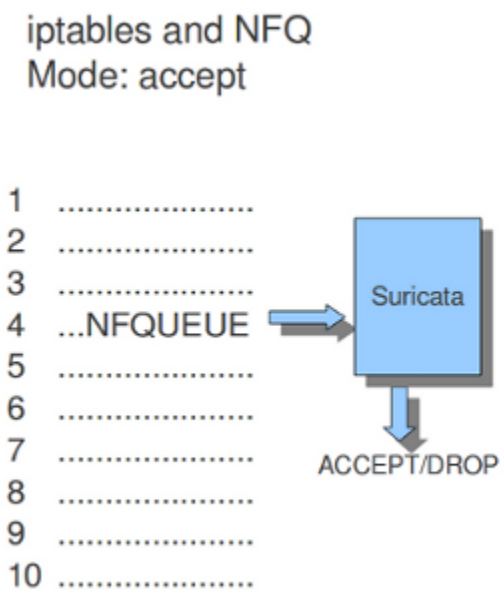
Add the numbers of the options repeat_mark and route_queue to the NFQ-rule:

```
iptables -I FORWARD -m mark ! --mark $MARK/$MASK -j NFQUEUE
```

```
nfq:
   mode: accept                 #By default the packet will be accepted or dropped by␣
↪Suricata
   repeat-mark: 1               #If the mode is set to 'repeat', the packets will be␣
↪marked after being
                                #processed by Suricata.
   repeat-mask: 1
   route-queue: 2               #Here you can assign the queue-number of the tool that␣
↪Suricata has to
                                #send the packets to after processing them.
```

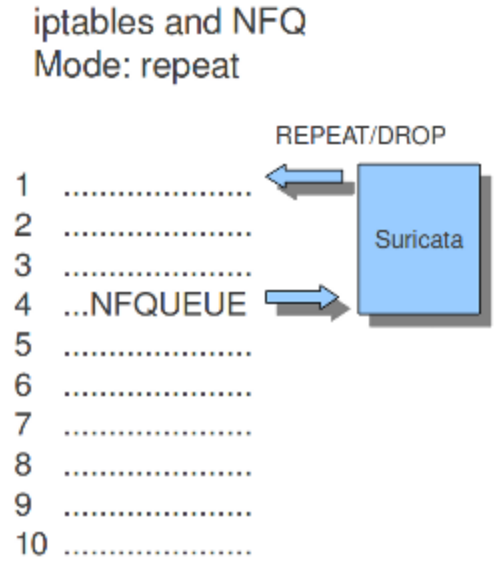
Example 1 NFQ1

mode: accept



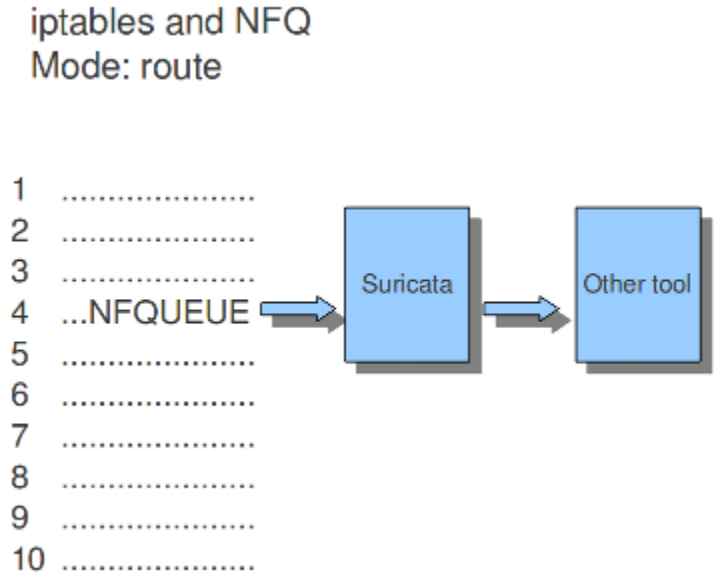
Example 2 NFQ

mode: repeat



Example 3 NFQ

mode: route



Ipfw

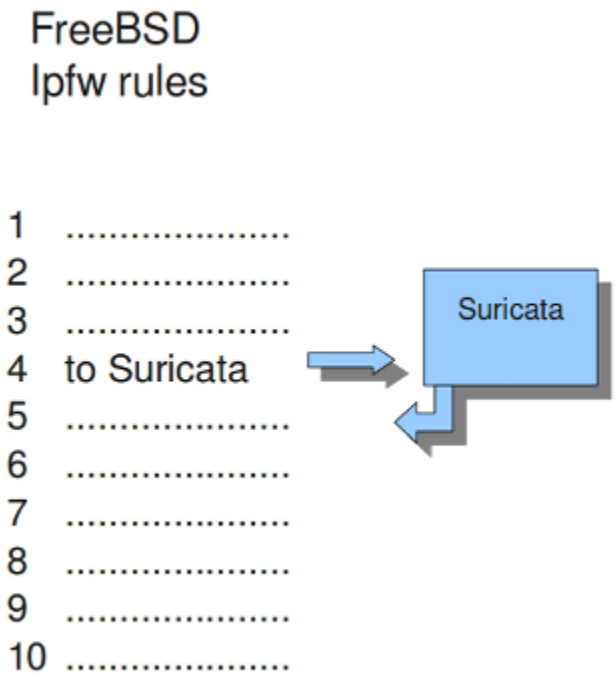
Suricata does not only support Linux, it supports the FreeBSD operating system (this is an open source Unix operating system) and Mac OS X as well. The in-line mode on FreeBSD uses ipfw (IP-firewall).

Certain rules in ipfw send network-traffic to Suricata. Rules have numbers. In this option you can set the rule to which the network-traffic will be placed back. Make sure this rule comes after the one that sends the traffic to Suricata, otherwise it will go around in circles.

The following tells the engine to re-inject packets back into the ipfw firewall at rule number 5500:

```
ipfw:
  ipfw-reinjection-rule-number: 5500
```

Example 16 Ipfw-reinjection.



12.1.18 Rules

Rule Files

Suricata by default is setup for rules to be managed by Suricata-Update with the following rule file configuration:

```
default-rule-path: /var/lib/suricata/rules
rule-files:
  - suricata.rules
```

A default installation of Suricata-Update will write out the rules to /var/lib/suricata/rules/suricata.rules.

You may want to edit this section if you are not using Suricata-Update or want to add rule files that are not managed by Suricata-Update, for example:

```
default-rule-path: /var/lib/suricata/rules
rule-files:
  - suricata.rules
  - /etc/suricata/rules/custom.rules
```

File names can be specific with an absolute path, or just the base name. If just the base name is provided it will be looked for in the `default-rule-path`.

If a rule file cannot be found, Suricata will log a warning message and continue to load, unless `--init-errors-fatal` has been specified on the command line, in which case Suricata will exit with an error code.

For more information on rule management see *Rule Management*.

Threshold-file

Within this option, you can state the directory in which the threshold-file will be stored. The default directory is: /etc/suricata/threshold.config

Classifications

The Classification-file is a file which makes the purpose of rules clear.

Some rules are just for providing information. Some of them are to warn you for serious risks like when you are being hacked etc.

In this classification-file, there is a part submitted to the rule to make it possible for the system-administrator to distinguish events.

A rule in this file exists of three parts: the short name, a description and the priority of the rule (in which 1 has the highest priority and 4 the lowest).

You can notice these descriptions returning in the rule and events / alerts.

```
Example:

configuration classification: misc-activity,Misc activity,3

Rule:

alert tcp $HOME_NET 21 -> $EXTERNAL_NET any (msg:"ET POLICY FTP Login Successful (non-
↪anonymous)";
flow:from_server,established;flowbits:isset,ET.ftp.user.login; flowbits:isnotset,ftp.
↪user.logged_in;
flowbits:set,ftp.user.logged_in; content:"230 ";pcre:!"/^230(\s+USER)?\s+(anonymous|ftp)/
↪smi";
classtype:misc-activity; reference:urldoc.emergingthreats.net/2003410,;
reference:url,www.emergingthreats.net/cgi-bin/cvsweb.cgi/sigs/POLICY/POLICY_FTP_Login;␣
↪sid:2003410; rev:7;)

Event/Alert:

10/26/10-10:13:42.904785  [**] [1:2003410:7] ET POLICY FTP Login Successful (non-
↪anonymous) [**]
 [Classification: Misc activity[Priority: 3] {TCP} 192.168.0.109:21 -> x.x.x.x:34117
```

You can set the direction of the classification configuration.

```
classification-file: /etc/suricata/classification.config
```

Rule-vars

There are variables which can be used in rules.

Within rules, there is a possibility to set for which IP-address the rule should be checked and for which IP-address it should not.

This way, only relevant rules will be used. To prevent you from having to set this rule by rule, there is an option in which you can set the relevant IP-address for several rules. This option contains the address group vars that will be passed in a rule. So, after HOME_NET you can enter your home IP-address.

```
vars:
  address-groups:
    HOME_NET: "[192.168.0.0/16,10.0.0.0/8,172.16.0.0/12]"        #By using [], it is␣
↪possible to set
                                                                 #complicated variables.
    EXTERNAL_NET: any
    HTTP_SERVERS: "$HOME_NET"                                    #The $-sign tells that␣
↪what follows is
                                                                 #a variable.
    SMTP_SERVERS: "$HOME_NET"
    SQL_SERVERS: "$HOME_NET"
    DNS_SERVERS: "$HOME_NET"
    TELNET_SERVERS: "$HOME_NET"
    AIM_SERVERS: any
```

It is a convention to use upper-case characters.

There are two kinds of variables: Address groups and Port-groups. They both have the same function: change the rule so it will be relevant to your needs.

In a rule there is a part assigned to the address and one to the port. Both have their variable.

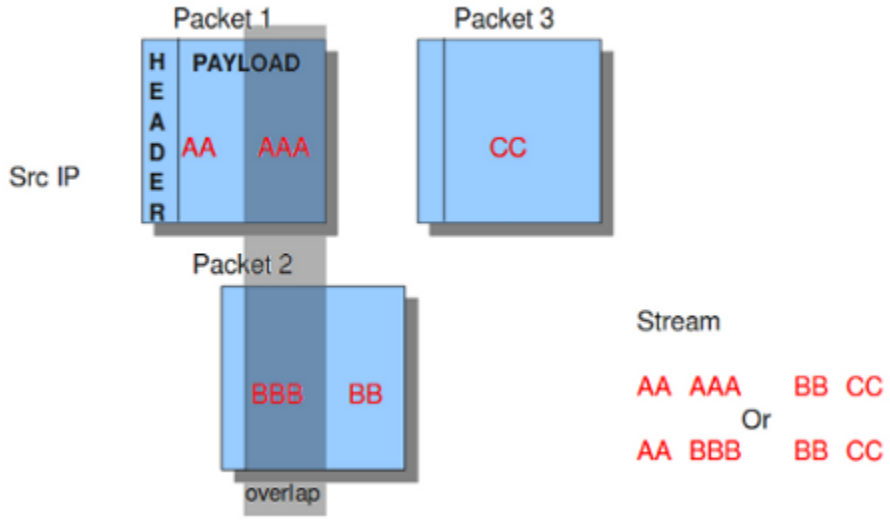
All options have to be set. If it is not necessary to set a specific address, you should enter 'any'.

```
port-groups:
  HTTP_PORTS: "80"
  SHELLCODE_PORTS: "!80"
  ORACLE_PORTS: 1521
  SSH_PORTS: 22
  SIP_PORTS: "[5060, 5061]"
```

Host-os-policy

Operating systems differ in the way they process fragmented packets and streams. Suricata performs differently with anomalies for different operating systems. It is important to set of which operating system your IP-address makes use of, so Suricata knows how to process fragmented packets and streams. For example in stream-reassembly there can be packets with overlapping payloads.

Example 17 Overlapping payloads



In the configuration-file, the operating-systems are listed. You can add your IP-address behind the name of the operating system you make use of.

```
host-os-policy:
  windows: [0.0.0.0/0]
  bsd: []
  bsd-right: []
  old-linux: []
  linux: [10.0.0.0/8, 192.168.1.100, "8762:2352:6241:7245:E000:0000:0000:0000"]
  old-solaris: []
  solaris: ["::1"]
  hpux10: []
  hpux11: []
  irix: []
  macos: []
  vista: []
  windows2k3: []
```

12.1.19 Suricata as a Firewall options

It is possible to run Suricata as a firewall. Please read *Firewall Mode Design* before using this. The existing yaml configuration options are listed below. For more examples on default policy configuration by app-layer hook, check *HTTP example with partially using default policies*.

If the engine is run in firewall mode, dedicated stats counters will be added to the stats logs. To see the stats reported for the firewall mode, refer to *Firewall Mode Stats*.

```
firewall:
  # toggle to enable firewall mode
  #enabled: no
  # Firewall rule file are in their own path and are not managed
  # by Suricata-Update.
  #rule-path: /etc/suricata/firewall/
```

(continues on next page)

(continued from previous page)

```
  # List of files with firewall rules. Order matters, files are loaded
  # in order and rules are applied in that order (per state, see docs)
  #rule-files:
  #  - firewall.rules

  # Default policies
  #
  # Choose a default policy for each firewall hook.
  # It is also possible to specify policies by app-layer protocol.
  # DNS example: Drop and alert on all DNS requests that are not allowed in firewall.
↪rules, accept all responses.
  #
  #policies:
  #  packet-filter: ["drop:packet"]
  #  dns:
  #    request-started: ["accept:hook"]
  #    request-complete: ["drop:flow", "alert"]
  #    response-started: ["accept:tx"]
```

12.1.20 Engine analysis and profiling

Suricata offers several ways of analyzing performance of rules and the engine itself.

Engine-analysis

The option engine-analysis provides information for signature writers about how Suricata organizes signatures internally.

Like mentioned before, signatures have zero or more patterns on which they can match. Only one of these patterns will be used by the multi pattern matcher (MPM). Suricata determines which patterns will be used unless the fast-pattern rule option is used.

The option engine-analysis creates a new log file in the default log dir. In this file all information about signatures and patterns can be found so signature writers are able to see which pattern is used and change it if desired.

To create this log file, you have to run Suricata with ./src/suricata -c suricata.yaml --engine-analysis.

```
engine-analysis:
   rules-fast-pattern: yes
```

Example:

```
[10703] 26/11/2010 -- 11:41:15 - (detect.c:560) <Info> (SigLoadSignatures)
-- Engine-Analysis for fast_pattern printed to file - /var/log/suricata/rules_fast_
↪pattern.txt

alert tcp any any -> any any (content:"Volume Serial Number"; sid:1292;)

== Sid: 1292 ==
Fast pattern matcher: content
Fast pattern set: no
Fast pattern only set: no
Fast pattern chop set: no
```

(continues on next page)

(continued from previous page)

```
Content negated: no
Original content: Volume Serial Number
Final content: Volume Serial Number

---

alert tcp any any -> any any (content:"abc"; content:"defghi"; sid:1;)

== Sid: 1 ==
Fast pattern matcher: content
Fast pattern set: no
Fast pattern only set: no
Fast pattern chop set: no
Content negated: no
Original content: defghi
Final content: defghi

---

alert tcp any any -> any any (content:"abc"; fast_pattern:only; content:"defghi"; sid:1;)

== Sid: 1 ==
Fast pattern matcher: content
Fast pattern set: yes
Fast pattern only set: yes
Fast pattern chop set: no
Content negated: no
Original content: abc
Final content: abc

---

alert tcp any any -> any any (content:"abc"; fast_pattern; content:"defghi"; sid:1;)

== Sid: 1 ==
Fast pattern matcher: content
Fast pattern set: yes
Fast pattern only set: no
Fast pattern chop set: no
Content negated: no
Original content: abc
Final content: abc

---

alert tcp any any -> any any (content:"abc"; fast_pattern:1,2; content:"defghi"; sid:1;)

== Sid: 1 ==
Fast pattern matcher: content
Fast pattern set: yes
Fast pattern only set: no
Fast pattern chop set: yes
```

(continues on next page)

(continued from previous page)

```
Fast pattern offset, length: 1, 2
Content negated: no
Original content: abc
Final content: bc
```

Rule and Packet Profiling settings

Rule profiling is a part of Suricata to determine how expensive rules are. Some rules are very expensive while inspecting traffic. Rule profiling is convenient for people trying to track performance problems and resolving them. Also for people writing signatures.

Compiling Suricata with rule-profiling will have an impact on performance, even if the option is disabled in the configuration file.

To observe the rule-performance, there are several options.

```
profiling:
  rules:
    enabled: yes
```

This engine is not used by default. It can only be used if Suricata is compiled with:

```
-- enable-profiling
```

At the end of each session, Suricata will display the profiling statistics. The list will be displayed sorted.

This order can be changed as pleased. The choice is between ticks, avgticks, checks, maxticks and matches. The setting of your choice will be displayed from high to low.

The amount of time it takes to check the signatures, will be administrated by Suricata. This will be counted in ticks. One tick is one CPU computation. 3 GHz will be 3 billion ticks.

Beside the amount of checks, ticks and matches it will also display the average and the maximum of a rule per session at the end of the line.

The option Limit determines the amount of signatures of which the statistics will be shown, based on the sorting.

```
sort: avgticks
limit: 100
```

Example of how the rule statistics can look like;

```
Rule            Ticks            %     Checks         Matches          Max Tick        ␣
↪Avg
Ticks

7560            107766621        0.02    138            37             105155334       ␣
↪780917.54
11963           1605394413       0.29    2623             1            144418923       ␣
↪612045.14
7040            1431034011       0.26    2500             0            106018209       ␣
↪572413.60
5726            1437574662       0.26    2623             1            115632900       ␣
↪548065.06
7037            1355312799       0.24    2562             0            116048286       ␣
↪529005.78
```

(continues on next page)

(continued from previous page)

```
11964           1276449255      0.23    2623            1               96412347        ␣
↪486637.15
7042            1272562974      0.23    2623            1               96405993        ␣
↪485155.54
5719            1233969192      0.22    2562            0               106439661       ␣
↪481642.93
5720            1204053246      0.21    2562            0               125155431       ␣
↪469966.14
```

Packet Profiling

```
packets:

  # Profiling can be disabled here, but it will still have a
  # performance impact if compiled in.


  enabled: yes                                  #this option is enabled by default
  filename: packet_stats.log                    #name of the file in which packet␣
↪profiling information will be
                                                #stored.
  append: yes                                   #If set to yes, new packet profiling␣
↪information will be added to the
                                                #information that was saved last in the␣
↪file.

  # per packet csv output
  csv:

    # Output can be disabled here, but it will still have a
    # performance impact if compiled in.

    enabled: no                                 #the sending of packet output to a csv-
↪file is by default disabled.
    filename: packet_stats.csv                  #name of the file in which csv packet␣
↪profiling information will be
                                                #stored
```

Packet profiling is enabled by default in suricata.yaml but it will only do its job if you compiled Suricata with --enable profiling.

The filename in which packet profiling information will be stored, is packet-stats.log. Information in this file can be added to the last information that was saved there, or if the append option is set to no, the existing file will be overwritten.

Per packet, you can send the output to a csv-file. This file contains one line for each packet with all profiling information of that packet. This option can be used only if Suricata is build with --enable-profiling and if the packet profiling option is enabled in yaml.

It is best to use runmode 'single' if you would like to profile the speed of the code. When using a single thread, there is no situation in which two threads have to wait for each other. When using two threads, the time threads might have to wait for each other will be taken in account when/during profiling packets. For more information see *Packet Profiling*.

12.1.21 Decoder

Teredo

The Teredo decoder can be disabled. It is enabled by default.

```
decoder:
  # Teredo decoder is known to not be completely accurate
  # it will sometimes detect non-teredo as teredo.
  teredo:
    enabled: true
    # ports to look for Teredo. Max 4 ports. If no ports are given, or
    # the value is set to 'any', Teredo detection runs on _all_ UDP packets.
    ports: $TEREDO_PORTS # syntax: '[3544, 1234]'
```

Using this default configuration, Teredo detection will run on UDP port 1. If the *ports* parameter is missing, or set to *any*, all ports will be inspected for possible presence of Teredo.

VXLAN

The VXLAN decoder can be configured with different reserved bits check modes. It is enabled by default and uses UDP port 4789.

```
decoder:
  # VXLAN decoder is assigned to up to 4 UDP ports. By default only the
  # IANA assigned port 4789 is enabled.
  vxlan:
    enabled: true
    ports: $VXLAN_PORTS # syntax: '[8472, 4789]' or '4789'.
    # Reserved bits check mode. Possible values are:
    #  - strict: check all reserved bits are zero for standard VXLAN (default)
    #  - permissive: do not check any reserved bits (allows VXLAN extensions)
    reserved-bits-check: strict
```

Using this default configuration, VXLAN detection will run on UDP port 4789 with strict reserved bits checking. The `reserved-bits-check` option controls how strictly the decoder validates the VXLAN header:

- `strict`: Validates all reserved bits are zero for standard VXLAN (default). This mode follows RFC 7348 strictly and will reject VXLAN extensions like GBP.
- `permissive`: Does not check any reserved bits. This mode accepts any VXLAN-like traffic regardless of reserved bit values. This is mainly useful when dealing with VXLAN extensions that may use reserved fields.

Recursion Level

Flow matching via recursion level can be disabled. It is enabled by default.

```
decoder:
  # Depending on packet pickup, incoming and outgoing tunnelled packets
  # can be scanned before the kernel has stripped and encapsulated headers,
  # respectively, leading to incoming and outgoing flows not being associated.
  recursion-level:
    use-for-tracking: true
```

Using this default setting, flows will be associated only if the compared packet headers are encapsulated in the same number of headers.

Tunnels

If your packets sources are multiple tunnels encapsulating the traffic, you can configure the `decoder.tunnels` section to assign a tunnel identifier to each of these tunnels. Such tunnel traffic is unidirectional from a sender (like a switch doing traffic mirroring) to the ip address where Suricata is listening.

> **Note**
>
> This setting is thus not meant for firewall nor IPS mode.

These tunnel identifiers are used in flow hashing to be able to distinguish the same-looking flow (same 5-tuple) from different tunnels, meaning it is in fact a different subnetwork (like a VLAN identifier).

Valid tunnel ids range from 1 to 32767.

The supported tunnel types are:

- vxlan
- erspan2

This section is a list of tunnels with the following parameters:

```
- id: 1
  type: erspan2 # or vxlan
  src: 192.168.1.1
  dst: 192.168.1.3 # ip address where suricata is receiving the traffic
  session: 123 # erspan span id or vxlan vni
```

Only IPv4 addresses are supported for now.

These tunnel identifiers can be used for multi-tenancy.

12.1.22 Advanced Options

stacktrace

Display diagnostic stacktraces when a signal unexpectedly terminates Suricata, e.g., such as SIGSEGV or SIGABRT. Requires the `libunwind` library to be available. The default value is to display the diagnostic message if a signal unexpectedly terminates Suricata -- e.g., `SIGABRT` or `SIGSEGV` occurs while Suricata is running.

```
logging:
    # Requires libunwind to be available when Suricata is configured and built.
    # If a signal unexpectedly terminates Suricata, displays a brief diagnostic
    # message with the offending stacktrace if enabled.
    #stacktrace-on-signal: on
```

12.1.23 Configuration hardening

The *security* section of suricata.yaml is meant to provide in-depth security configuration options.

Besides landlock, (see *Using Landlock LSM*), one setting is available. *limit-noproc* is a boolean to prevent process creation by Suricata. If you do not need Suricata to create other processes or threads (you may need it for LUA scripts for instance or plugins), enable this to call *setrlimit* with *RLIMIT_NPROC* argument (see *man setrlimit*). This prevents potential exploits against Suricata to fork a new process, even if it does not prevent the call of *exec*.

Warning! This has no effect on Linux when running as root. If you want a hardened configuration, you probably want to set *run-as* configuration parameter so as to drop root privileges.

Beyond suricata.yaml, other ways to harden Suricata are - compilation : enabling ASLR and other exploit mitigation techniques. - environment : running Suricata on a device that has no direct access to Internet.

Lua

Suricata 8.0 sandboxes Lua rules by default. The restrictions on the sandbox for Lua rules can be modified in the `security.lua` section of the configuration file. This section also applies to Lua transforms. Additionally, Lua rules can be completely disabled in the same way as for as the Suricata 7.0 default:

```
security:
  lua:
    # Allow Lua rules. Enabled by default.
    #allow-rules: true

    # Upper bound of live memory a sandboxed Lua state may grow
    # beyond its setup-time footprint (the loaded libs, the script
    # chunk and the values thread_init() stored). Per-call working
    # memory and memory the script retains between calls (module
    # variables, upvalues, cached handles) count against it; the
    # setup-time footprint and the call's own inputs (the inspected
    # buffer and args) do not.
    #max-bytes: 500000

    # Upper bound of lua instructions by a Lua rule before it will fail
    #max-instructions: 500000

    # Allow dangerous lua operations like external packages and file io
    #allow-restricted-functions: false
```

12.2 Global-Thresholds

Thresholds can be configured in the rules themselves, see *Thresholding Keywords*. They are often set by rule writers based on their intelligence for creating a rule combined with a judgement on how often a rule will alert.

Thresholds are tracked in a hash table that is sized according to configuration, see: *Thresholding Settings*.

12.2.1 Threshold Config

Next to rule thresholding more thresholding can be configured on the sensor using the threshold.config.

threshold/event_filter

Syntax:

```
threshold gen_id <gid>, sig_id <sid>, type <threshold|limit|both>, \
  track <by_src|by_dst|by_rule|by_both|by_flow>, count <N>, seconds <T>
```

rate_filter

Rate filters allow changing of a rule action when a rule matches.

Syntax:

```
rate_filter gen_id <gid>, sig_id <sid>, track <tracker>, \
  count <c>, seconds <s>, new_action <action>, timeout <timeout>
```

Example:

```
rate_filter gen_id 1, sig_id 1000, track by_rule, count 100, seconds 60, \
  new_action alert, timeout 30
```

gen_id

Generator id. Normally 1, but if a rule uses the `gid` keyword to set another value it has to be matched in the `gen_id`.

sig_id

Rule/signature id as set by the rule `sid` keyword.

track

Where to track the rule matches. When using by_src/by_dst the tracking is done per IP-address. The Host table is used for storage. When using by_rule it's done globally for the rule. Option by_both used to track per IP pair of source and destination. Packets going to opposite directions between same addresses tracked as the same pair. The by_flow option tracks the rule matches in the flow.

count

Number of rule hits before the `rate_filter` is activated.

seconds

Time period within which the `count` needs to be reached to activate the `rate_filter`

new_action

New action that is applied to matching traffic when the `rate_filter` is in place.

Values:

```
<alert|drop|pass|reject>
```

Note: 'sdrop' and 'log' are supported by the parser but not implemented otherwise.

timeout

Time in seconds during which the `rate_filter` will remain active.

Example

Let's say we want to limit incoming connections to our SSH server. The rule `888` below simply alerts on SYN packets to the SSH port of our SSH server. If an IP-address triggers this more than 10 or more with a minute, the drop `rate_filter` is set with a timeout of 5 minutes.

Rule:

```
alert tcp any any -> $MY_SSH_SERVER 22 (msg:"Connection to SSH server"; \
  flow:to_server; flags:S,12; sid:888;)
```

Rate filter:

```
rate_filter gen_id 1, sig_id 888, track by_src, count 10, seconds 60, \
  new_action drop, timeout 300
```

suppress

Suppressions can be used to suppress alerts for a rule or a host/network. Actions performed when a rule matches, such as setting a flowbit, are still performed.

Syntax:

```
suppress gen_id <gid>, sig_id <sid>
suppress gen_id <gid>, sig_id <sid>, track <by_src|by_dst|by_either>, ip
↪<ip|subnet|addressvar>
```

Examples:

```
suppress gen_id 1, sig_id 2002087, track by_src, ip 209.132.180.67
```

This will make sure the signature 2002087 will never match for src host 209.132.180.67.

Other possibilities/examples:

```
suppress gen_id 1, sig_id 2003614, track by_src, ip 217.110.97.128/25
suppress gen_id 1, sig_id 2003614, track by_src, ip [192.168.0.0/16,10.0.0.0/8,172.16.0.
↪0/12]
suppress gen_id 1, sig_id 2003614, track by_src, ip $HOME_NET
suppress gen_id 1, sig_id 2003614, track by_either, ip 217.110.97.128/25
```

In the last example above, the `by_either` tracking means that if either the `source ip` or `destination ip` matches `217.110.97.128/25` the rule with sid 2003614 is suppressed.

12.2.2 Global thresholds vs rule thresholds

Note: this section applies to 1.4+ In 1.3 and before mixing rule and global thresholds is not supported.

When a rule has a threshold/detection_filter set a rule can still be affected by the global threshold file.

The rule below will only fire if 10 or more emails are being delivered/sent from a host within 60 seconds.

```
alert tcp any any -> any 25 (msg:"ET POLICY Inbound Frequent Emails - Possible Spambot␣
↪Inbound"; \
    flow:established; content:"mail from|3a|"; nocase;                                ␣
↪       \
    threshold: type threshold, track by_src, count 10, seconds 60;                    ␣
↪       \
    reference:url,doc.emergingthreats.net/2002087; classtype:misc-activity; sid:2002087;
↪ rev:10;)
```

Next, we'll see how global settings affect this rule.

Suppress

Suppressions can be combined with rules with thresholds/detection_filters with no exceptions.

```
suppress gen_id 1, sig_id 2002087, track by_src, ip 209.132.180.67
suppress gen_id 0, sig_id 0, track by_src, ip 209.132.180.67
suppress gen_id 1, sig_id 0, track by_src, ip 209.132.180.67
```

Each of the rules above will make sure 2002087 doesn't alert when the source of the emails is 209.132.180.67. It **will** alert for all other hosts.

```
suppress gen_id 1, sig_id 2002087
```

This suppression will simply convert the rule to "noalert", meaning it will never alert in any case. If the rule sets a flowbit, that will still happen.

Threshold/event_filter

When applied to a specific signature, thresholds and event_filters (threshold from now on) will override the signature setting. This can be useful for when the default in a signature doesn't suit your environment.

```
threshold gen_id 1, sig_id 2002087, type both, track by_src, count 3, seconds 5
threshold gen_id 1, sig_id 2002087, type threshold, track by_src, count 10, seconds 60
threshold gen_id 1, sig_id 2002087, type limit, track by_src, count 1, seconds 15
```

Each of these will replace the threshold setting for 2002087 by the new threshold setting.

Note: overriding all gids or sids (by using gen_id 0 or sig_id 0) is not supported. Bug https://redmine.openinfosecfoundation.org/issues/425.

Rate_filter

see https://redmine.openinfosecfoundation.org/issues/425.

12.3 Exception Policies

Suricata has a set of configuration variables to indicate what should the engine do when certain exception conditions, such as hitting a memcap, are reached.

They are called Exception Policies and are configurable via suricata.yaml. If enabled, the engine will call them when it reaches exception states. Stats for any applied exception policies can be found in counters related to the specific configuration setting (*read more*). Some configuration is available directly via the *stats settings*.

For developers or for researching purposes, there are also simulation options exposed in `qa-simulation` mode and passed via command-line. These exist to force or simulate failures or errors and understand Suricata behavior under such conditions. See *Command-line Options for Simulating Exceptions* for those.

12.3.1 Master Switch

It is possible to set all configuration policies via what we call "master switch". This offers a quick way to define what the engine should do in case of traffic exceptions, while still allowing for the flexibility of indicating a different behavior for specific exception policies your setup/environment may have the need to.

```
# Define a common behavior for all exception policies.
# In IPS mode, the default is drop-flow. For cases when that's not possible, the
# engine will fall to drop-packet. To fallback to old behavior (setting each of
# them individually, or ignoring all), set this to ignore.
# All values available for exception policies can be used, and there is one
# extra option: auto - which means drop-flow or drop-packet (as explained above)
```

(continues on next page)

(continued from previous page)

```
# in IPS mode, and ignore in IDS mode. Exception policy values are: drop-packet,
# drop-flow, reject, bypass, pass-packet, pass-flow, ignore (disable).
exception-policy: auto
```

This value will be overwritten by specific exception policies whose settings are also defined in the yaml file.

Auto

In IPS mode, the default behavior for most of the exception policies is to fail close. This means dropping the flow, or the packet, when the flow action is not supported. The default policy for the midstream exception will be ignore if midstream flows are accepted.

It is possible to disable this default, by setting the exception policies' "master switch" yaml config option to `ignore`.

In IDS mode, setting `auto` mode actually means disabling the `master-switch`, or ignoring the exception policies.

> **Note**
>
> If no exception policy is enabled, Suricata will not log exception policy stats.

12.3.2 Specific settings

Exception policies are implemented for:

Table 1: Exception Policy configuration variables

Config setting	Policy variable	Affects	Expected behavior
stream.memcap	memcap-policy	Flow or packet	If a stream memcap limit is reached, apply the memcap policy to the packet and/or flow.
stream.midstream	midstream-policy	Flow	If a session is picked up midstream, apply the midstream policy to the flow.
stream.reassembly.n	memcap-policy	Flow or packet	If stream reassembly reaches memcap limit, apply memcap policy to the packet and/or flow.
flow.memcap	memcap-policy	Packet	Apply policy when the memcap limit for flows is reached and no flow could be freed up. **Policy can only be applied to the packet.**
defrag.memcap	memcap-policy	Packet	Apply policy when the memcap limit for defrag is reached and no tracker could be picked up. **Policy can only be applied to the packet.**
app-layer	error-policy	Flow or packet	Apply policy if a parser reaches an error state. Policy can be applied to packet and/or flow.

To change any of these, go to the specific section in the suricata.yaml file (for more configuration details, check the *suricata.yaml's* documentation).

The possible values for the exception policies, and the resulting behaviors, are:

- `drop-flow`: disable decoding and parsing for the whole flow (packets, payload, application layer protocol), drop the packet and all future packets in the flow.
- `drop-packet`: drop the packet.
- `reject`: same as `drop-flow`, but reject the current packet as well (see `reject` action in Rule's *Action*).
- `rejectboth`: same as `reject`, except that it sends reject packets to both the source and destination addresses.

- `bypass`: bypass the flow. No further decoding or parsing is done. *Bypass* may be offloaded.
- `pass-flow`: disable payload and packet detection; stream reassembly, app-layer parsing and logging still happen.
- `pass-packet`: disable detection, still does stream updates and app-layer parsing (depending on which policy triggered it).
- `ignore`: do not apply exception policies (default behavior).

The *drop*, *pass* and *reject* are similar to the rule actions described in *rule actions*.

12.3.3 Exception Policies and Midstream Pick-up Sessions

Suricata behavior can be difficult to track in case of midstream session pick-ups. Consider this matrix illustrating the different interactions for midstream pick-ups enabled or not and the various exception policy values:

Table 2: **Exception Policy Behaviors - IDS Mode**

Exception Policy	Midstream pick-up sessions ENABLED (stream.midstream=true)	Midstream pick-up sessions DISABLED (stream.midstream=false)
Ignore	Session and app-layer traffic tracked and parsed, log app-layer traffic, **do** detection.	Session not tracked. No app-layer parsing or logging. No stream reassembly. No detection.
Drop-flow	Not valid.*	Not valid.*
Drop-packet	Not valid.*	Not valid.*
Reject / Reject-both	Not valid.*	Session not tracked, flow REJECTED.
Pass-flow	Session and app-layer traffic tracked and parsed, log app-layer traffic, **no** detection.	Session not tracked. No app-layer parsing or logging. No stream reassembly. No detection.
Pass-packet	Not valid.*	Not valid.*
Bypass	Not valid.*	Session not tracked. No app-layer parsing or logging. No stream reassembly. No detection.
Auto	Midstream policy applied: "ignore". Same behavior.	Midstream policy applied: "ignore". Same behavior.

The main difference between IDS and IPS scenarios is that in IPS mode flows can be allowed or blocked (as in with the PASS and DROP rule actions). Packet actions are not valid, as midstream pick-up is a configuration that affects the whole flow.

Table 3: **Exception Policy Behaviors - IPS Mode**

Exception Policy	Midstream pick-up sessions ENABLED (stream.midstream=true)	Midstream pick-up sessions DISABLED (stream.midstream=false)
Ignore	Session and app-layer traffic tracked and parsed, log app-layer traffic, **do** detection.	Session not tracked. No app-layer parsing or logging. No stream reassembly. No detection.
Drop-flow	Not valid.*	Session not tracked. No app-layer parsing or logging. No stream reassembly. No detection. Flow DROPPED.
Drop-packet	Not valid.*	Not valid.*
Reject / Rejectboth	Not valid.*	Session not tracked, flow DROPPED and REJECTED.
Pass-flow	Track session, parse and log app-layer traffic, **no** detection.	Session not tracked. No app-layer parsing or logging. No stream reassembly. No detection.
Pass-packet	Not valid.*	Not valid.*
Bypass	Not valid.*	Session not tracked. No app-layer parsing or logging. No stream reassembly. No detection. Packets ALLOWED.
Auto	Midstream policy applied: "ignore". Same behavior.	Midstream policy applied: "drop-flow". Same behavior.

Notes:

- Not valid means that Suricata will error out and won't start.
- `REJECT` will make Suricata send a Reset-packet unreach error to the sender of the matching packet.

Exception Policies Interactions

In certain scenarios, one exception policy being applied might trigger another exception. The midstream session policy seems the most prone to be triggered, by its definition.

If a setting can be applied to packets in **IPS mode** with a drop effect, this can result in following packets for the affected flow(s) being considered as a midstream session, thus triggering the related policy.

In particular, these interactions seem probable for the following policies, **if set to drop-packet**:

- Stream Memcap
- Stream Reassembly Memcap
- Flow Memcap
- Defrag Memcap
- Application Layer Protocol Error

12.3.4 Log Output

Flow Event

When an Exception Policy is triggered, this will be indicated in the flow log event for the associated flow, also indicating which target triggered that, and what policy was applied. If no exception policy is triggered, that field won't be present in the logs.

Note that this is true even if the policy is applied only to certain packets from a flow.

In the log sample below, the flow triggered the `midstream policy`, leading to Suricata applying the behavior that had been configured for such scenario: *to pass the flow* (`pass_flow`). It also did trigger the `app_layer_error` exception policy, but that is set up to `ignore`:


```
"flow": {
  "pkts_toserver": 4,
  "pkts_toclient": 5,
  "bytes_toserver": 495,
  "bytes_toclient": 351,
  "start": "2016-07-13T22:42:07.199672+0000",
  "end": "2016-07-13T22:42:07.573174+0000",
  "age": 0,
  "state": "new",
  "reason": "shutdown",
  "alerted": false,
  "action": "pass",
  "exception_policy": [
     {
       "target": "stream_midstream",
       "policy": "pass_flow"
     },
     {
       "target": "app_layer_error",
       "policy": "ignore"
     }
   ]
}
```


Available Stats

There are stats counters for each supported exception policy scenario that will be logged when exception policies are enabled:

Table 4: **Exception Policy Stats Counters**

Setting	Counters
stream.memcap	exception_policy.tcp.ssn_memcap
stream.reassembly.memcap	exception_policy.tcp.reassembly.memcap
stream.midstream	exception_policy.tcp.midstream
defrag.memcap	exception_policy.defrag.memcap
flow.memcap	exception_policy.flow.memcap
app-layer.error	• exception_policy.app_layer.error • app_layer.error.exception_policy

If a given exception policy does not apply for a setting, no related counter is logged.

Stats for application layer errors are available in summarized form or per application layer protocol. As the latter is extremely verbose, by default Suricata logs only the summary. If any further investigation is needed, it is recommended to enable per-app-proto exception policy error counters temporarily (for more, read *stats configuration*).

12.3.5 Command-line Options for Simulating Exceptions

It is also possible to force specific exception scenarios, to check engine behavior under failure or error conditions.

To enable these, you must build Suricata with `qa-simulation` enabled (`--enable-qa-simulation`).

The available command-line options are:

- `simulate-applayer-error-at-offset-ts`: force an applayer error in the to server direction at the given offset.
- `simulate-applayer-error-at-offset-tc`: force an applayer error in the to client direction at the given offset.
- `simulate-packet-loss`: simulate that the packet with the given number (`pcap_cnt`) from the session was lost.
- `simulate-packet-tcp-reassembly-memcap`: simulate that the TCP stream reassembly reached memcap for the specified packet.
- `simulate-packet-tcp-ssn-memcap`: simulate that the TCP session hit the memcap for the specified packet.
- `simulate-packet-flow-memcap`: force the engine to assume that flow memcap is hit at the given packet.
- `simulate-packet-defrag-memcap`: force Suricata to assume memcap is hit when defragmenting specified packet.
- `simulate-alert-queue-realloc-failure`: prevent the engine from dynamically growing the temporary alert queue, during alerts processing.

12.3.6 Glossary

- **decoding**: traffic parsing on the packet level;
- **[app-layer] parsing**: traffic is parsed on the application layer level for events, anomalies and logging;
- **detection**: evaluate traffic against loaded rules to generate alerts and/ or block or allow traffic.

Common abbreviations

- applayer/ app-layer: application layer protocol
- memcap: (maximum) memory capacity available
- defrag: defragmentation

12.4 Snort.conf to Suricata.yaml

This guide is meant for those who are familiar with Snort and the snort.conf configuration format. This guide will provide a 1:1 mapping between Snort and Suricata configuration wherever possible.

12.4.1 Variables

snort.conf

```
ipvar HOME_NET any
ipvar EXTERNAL_NET any
...

portvar HTTP_PORTS [80,81,311,591,593,901,1220,1414,1741,1830,2301,2381,2809,3128,3702,
↪4343,4848,5250,7001,7145,7510,7777,7779,8000,8008,8014,8028,8080,8088,8090,8118,8123,
```

(continues on next page)

(continued from previous page)

```
↪8180,8181,8243,8280,8800,8888,8899,9000,9080,9090,9091,9443,9999,11371,55555]
portvar SHELLCODE_PORTS !80
...
```

suricata.yaml

```
vars:
  address-groups:

    HOME_NET: "[192.168.0.0/16,10.0.0.0/8,172.16.0.0/12]"
    EXTERNAL_NET: "!$HOME_NET"

  port-groups:
    HTTP_PORTS: "80"
    SHELLCODE_PORTS: "!80"
```

Note that Suricata can automatically detect HTTP traffic regardless of the port it uses. So the HTTP_PORTS variable is not nearly as important as it is with Snort, **if** you use a Suricata enabled ruleset.

12.4.2 Decoder alerts

snort.conf

```
# Stop generic decode events:
config disable_decode_alerts

# Stop Alerts on experimental TCP options
config disable_tcpopt_experimental_alerts

# Stop Alerts on obsolete TCP options
config disable_tcpopt_obsolete_alerts

# Stop Alerts on T/TCP alerts
config disable_tcpopt_ttcp_alerts

# Stop Alerts on all other TCPOption type events:
config disable_tcpopt_alerts

# Stop Alerts on invalid ip options
config disable_ipopt_alerts
```

suricata.yaml

Suricata has no specific decoder options. All decoder related alerts are controlled by rules. See #Rules below.

12.4.3 Checksum handling

snort.conf

```
config checksum_mode: all
```

suricata.yaml

Suricata's checksum handling works *on-demand*. The stream engine checks TCP and IP checksum by default:

```
stream:
  checksum-validation: yes      # reject wrong csums
```

Alerting on bad checksums can be done with normal rules. See #Rules, decoder-events.rules specifically.

12.4.4 Various configs

Active response

snort.conf

```
# Configure active response for non inline operation. For more information, see REAMDE.
↪active
# config response: eth0 attempts 2
```

suricata.yaml

Active responses are handled automatically w/o config if rules with the "reject" action are used.

Dropping privileges

snort.conf

```
# Configure specific UID and GID to run snort as after dropping privs. For more␣
↪information see snort -h command line options
#
# config set_gid:
# config set_uid:
```

Suricata

To set the user and group use the --user <username> and --group <groupname> command-line options.

Snaplen

snort.conf

```
# Configure default snaplen. Snort defaults to MTU of in use interface. For more␣
↪information see README
#
# config snaplen:
#
```

Suricata always works at full snap length to provide full traffic visibility.

Bpf

snort.conf

```
# Configure default bpf_file to use for filtering what traffic reaches snort. For more␣
↪information see snort -h command line options (-F)
#
# config bpf_file:
#
```

suricata.yaml

BPF filters can be set per packet acquisition method, with the "bpf-filter: <file>" yaml option and in a file using the -F command line option.

For example:

```
pcap:
  - interface: eth0
    #buffer-size: 16777216
    #bpf-filter: "tcp and port 25"
    #checksum-checks: auto
    #threads: 16
    #promisc: no
    #snaplen: 1518
```

12.4.5 Log directory

snort.conf

```
# Configure default log directory for snort to log to.  For more information see snort -
↪h command line options (-l)
#
# config logdir:
```

suricata.yaml

```
default-log-dir: /var/log/suricata/
```

This value is overridden by the -l command-line option.

12.4.6 Packet acquisition

snort.conf

```
# Configure DAQ related options for inline operation. For more information, see README.
↪daq
#
# config daq: <type>
# config daq_dir: <dir>
# config daq_mode: <mode>
# config daq_var: <var>
#
# <type> ::= pcap | afpacket | dump | nfq | ipq | ipfw
# <mode> ::= read-file | passive | inline
# <var> ::= arbitrary <name>=<value passed to DAQ
# <dir> ::= path as to where to look for DAQ module so's
```

suricata.yaml

Suricata has all packet acquisition support built-in. It's configuration format is very verbose.

```
pcap:
  - interface: eth0
    #buffer-size: 16777216
```

(continues on next page)

(continued from previous page)

```
    #bpf-filter: "tcp and port 25"
    #checksum-checks: auto
    #threads: 16
    #promisc: no
    #snaplen: 1518
pfring:
afpacket:
nfq:
ipfw:
```

Passive vs inline vs reading files is determined by how Suricata is invoked on the command line.

12.4.7 Rules

snort.conf:

In snort.conf a RULE_PATH variable is set, as well as variables for shared object (SO) rules and preprocessor rules.

```
var RULE_PATH ../rules
var SO_RULE_PATH ../so_rules
var PREPROC_RULE_PATH ../preproc_rules

include $RULE_PATH/local.rules
include $RULE_PATH/emerging-activex.rules
...
```

suricata.yaml:

In the suricata.yaml the default rule path is set followed by a list of rule files. Suricata does not have a concept of shared object rules or preprocessor rules. Instead of preprocessor rules, Suricata has several rule files for events set by the decoders, stream engine, http parser etc.

```
default-rule-path: /etc/suricata/rules
rule-files:
 - local.rules
 - emerging-activex.rules
```

The equivalent of preprocessor rules are loaded like normal rule files:

```
rule-files:
 - decoder-events.rules
 - stream-events.rules
 - http-events.rules
 - smtp-events.rules
```

12.5 Multi Tenancy

12.5.1 Introduction

Multi tenancy support allows different tenants to use different rule sets with different rule variables.

Tenants are identified by their *selector*; a *selector* can be a VLAN, interface/device, or from a pcap file ("direct").

12.5.2 YAML

Add a new section in the main ("master") Suricata configuration file -- `suricata.yaml` -- named `multi-detect`.

Settings:

- *enabled*: yes/no -> is multi-tenancy support enabled
- *selector*: direct (for unix socket pcap processing, see below), VLAN, tunnel or device
- *default*: yes/no (default: no) -> inspect traffic not matching any mapping with the rules of the master configuration, see *Traffic not matching any mapping*
- *loaders*: number of *loader* threads, for parallel tenant loading at startup
- *tenants*: list of tenants
- *config-path*: path from where the tenant yamls are loaded
  - id: tenant id (numeric values only)
  - yaml: separate yaml file with the tenant specific settings
- *mappings*:
  - VLAN id or device: The outermost VLAN is used to match.
  - tenant id: tenant to associate with the VLAN id or device

```
multi-detect:
  enabled: yes
  #selector: direct # direct or vlan or tunnel or device
  selector: vlan
  loaders: 3

  tenants:
  - id: 1
    yaml: tenant-1.yaml
  - id: 2
    yaml: tenant-2.yaml
  - id: 3
    yaml: tenant-3.yaml

  mappings:
  - vlan-id: 1000
    tenant-id: 1
  - vlan-id: 2000
    tenant-id: 2
  - vlan-id: 1112
    tenant-id: 3
```

The tenant-1.yaml, tenant-2.yaml, tenant-3.yaml each contain a partial configuration:

```
# Set the default rule path here to search for the files.
# if not set, it will look at the current working dir
default-rule-path: /etc/suricata/rules
rule-files:
  - rules1

# You can specify a threshold config file by setting "threshold-file"
```

(continues on next page)

(continued from previous page)

```
# to the path of the threshold config file:
# threshold-file: /etc/suricata/threshold.config

classification-file: /etc/suricata/classification.config
reference-config-file: /etc/suricata/reference.config

# Holds variables that would be used by the engine.
vars:

  # Holds the address group vars that would be passed in a Signature.
  # These would be retrieved during the Signature address parsing stage.
  address-groups:

    HOME_NET: "[192.168.0.0/16,10.0.0.0/8,172.16.0.0/12]"

    EXTERNAL_NET: "!$HOME_NET"

    ...

  port-groups:

    HTTP_PORTS: "80"

    SHELLCODE_PORTS: "!80"

    ...
```

vlan-id

Assign tenants to VLAN ids. Suricata matches the outermost VLAN id with this value. Multiple VLANs can have the same tenant id. VLAN id values must be between 1 and 4094.

Example of VLAN mapping:

```
mappings:
- vlan-id: 1000
  tenant-id: 1
- vlan-id: 2000
  tenant-id: 2
- vlan-id: 1112
  tenant-id: 3
```

The mappings can also be modified over the unix socket, see below.

Note: can only be used if `vlan.use-for-tracking` is enabled.

device

Assign tenants to devices. A single tenant can be assigned to a device. Multiple devices can have the same tenant id.

Example of device mapping:

```
mappings:
- device: ens5f0
```

(continues on next page)

(continued from previous page)

```
  tenant-id: 1
- device: ens5f1
  tenant-id: 3
```

The mappings are static and cannot be modified over the unix socket.

Note: Not currently supported for IPS.

Note: support depends on a capture method using the 'livedev' API. Currently these are: pcap, AF_PACKET, PF_RING and Netmap.

tunnel

Assign tenants to tunnels. A single tenant can be assigned to a tunnel, see *Tunnels*. Multiple tunnels can have the same tenant id.

Example of tunnel mapping:

```
mappings:
- tunnel-id: 1
  tenant-id: 1
- tunnel-id: 2
  tenant-id: 3
```

Traffic not matching any mapping

Traffic the selector cannot map to a tenant, e.g. untagged traffic, an unmapped VLAN id or device, traffic from a tunnel not defined in `decoder.tunnels` or the outer packets of a defined tunnel, is still decoded, tracked in flows and logged. Only its inspection depends on `default`:

- `default: no` (default): no rule inspects it. The rule files of the master configuration, including the ones passed with `-S`, are not loaded.
- `default: yes`: the rules of the master configuration inspect it. Its alerts have no `tenant_id`.

```
multi-detect:
  enabled: yes
  selector: vlan
  default: yes
```

12.5.3 Per tenant settings

The following settings are per tenant:

- default-rule-path
- rule-files
- classification-file
- reference-config-file
- threshold-file
- address-vars
- port-vars

12.5.4 Unix Socket

Registration

`register-tenant <id> <yaml>`

Examples:

```
register-tenant 1 tenant-1.yaml
register-tenant 2 tenant-2.yaml
register-tenant 3 tenant-3.yaml
register-tenant 5 tenant-5.yaml
register-tenant 7 tenant-7.yaml
```

`unregister-tenant <id>`

```
unregister-tenant 2
unregister-tenant 1
```

Unix socket runmode (pcap processing)

The Unix Socket `pcap-file` command is used to associate the tenant with the pcap:

```
pcap-file traffic1.pcap /logs1/ 1
pcap-file traffic2.pcap /logs2/ 2
pcap-file traffic3.pcap /logs3/ 3
pcap-file traffic4.pcap /logs5/ 5
pcap-file traffic5.pcap /logs7/ 7
```

This runs the traffic1.pcap against tenant 1 and it logs into /logs1/, traffic2.pcap against tenant 2 and logs to /logs2/ and so on.

Live traffic mode

Multi-tenancy supports both VLAN and devices with live traffic.

In the master configuration yaml file, specify `device` or `vlan` for the `selector` setting.

Registration

Tenants can be mapped to vlan ids.

`register-tenant-handler <tenant id> vlan <vlan id>`

```
register-tenant-handler 1 vlan 1000
```

`unregister-tenant-handler <tenant id> vlan <vlan id>`

```
unregister-tenant-handler 4 vlan 1111
unregister-tenant-handler 1 vlan 1000
```

The registration of tenant and tenant handlers can be done on a running engine.

Reloads

Reloading all tenants:

`reload-tenants`

```
reload-tenants
```

Reloading a single tenant:

`reload-tenant <tenant id> [yaml path]`

```
reload-tenant 1 tenant-1.yaml
reload-tenant 5
```

The `[yaml path]` is optional. If it isn't provided, the original path of the tenant will be used during the reload.

12.5.5 Eve JSON output

When multi-tenant support is configured and the detect engine is active then all EVE-types that report based on flows will also report the corresponding `tenant_id` for events matching a tenant configuration.

12.6 Dropping Privileges After Startup

Currently, `libcap-ng` is needed for dropping privileges on Suricata after startup. For libcap, see status of feature request number #276 -- Libcap support for dropping privileges.

Most distributions have `libcap-ng` in their repositories.

To download the current version of libcap-ng from upstream, see also http://people.redhat.com/sgrubb/libcap-ng/ChangeLog

```
wget http://people.redhat.com/sgrubb/libcap-ng/libcap-ng-0.7.8.tar.gz
tar -xzvf libcap-ng-0.7.8.tar.gz
cd libcap-ng-0.7.8
./configure
make
make install
```

Download, configure, compile and install Suricata for your particular setup. See *Installation*. Depending on your environment, you may need to add the --with-libpcap_ng-libraries and --with-libpcap_ng-includes options during the configure step. e.g:

```
./configure --with-libcap_ng-libraries=/usr/local/lib \
  --with-libcap_ng-includes=/usr/local/include
```

Now, when you run Suricata, tell it what user and/or group you want it to run as after startup with the --user and --group options. e.g. (this assumes a 'suri' user and group):

```
suricata -D -i eth0 --user=suri --group=suri
```

You will also want to make sure your user/group permissions are set so Suricata can still write to its log files which are usually located in /var/log/suricata.

```
mkdir -p /var/log/suricata
chown -R root:suri /var/log/suricata
chmod -R 775 /var/log/suricata
```

12.7 Using Landlock LSM

Landlock is a Linux Security Module that has been introduced in Linux 5.13. It allows an application to sandbox itself by selecting access right to directories using a deny by default approach.

Given its nature, Suricata knows where it is going to read files and where it is going to write them. So it is possible to implement an efficient Landlock sandboxing policy.

Landlock is not active by default and needs to be activated in the YAML configuration. Configuration should come with sane default (defined at build time) and the command line options are used to dynamically add some permissions.

Please note that Landlock is in blocking mode by default so careful testing is needed in production.

To enable Landlock, edit the YAML and set `enabled` to `yes`:

```
landlock:
  enabled: yes
  directories:
    write:
      - /var/log/suricata/
      - /var/run/
    read:
      - /usr/
      - /etc/
      - /etc/suricata/
```

Following your running configuration you may have to add some directories. There are two lists you can use, `write` to add directories where write is needed and `read` for directories where read access is needed.

Landlock is not active in some distributions and you may need to activate it at boot by adding `lsm=landock` to the Linux command line. For example, on a Debian distribution with at least a linux 5.13, you can edit `/etc/default/grub` and update the `GRUB_CMDLINE_LINUX_DEFAULT` option:

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet lsm=landlock"
```

Then run `sudo update-grub` and reboot.

You can check at boot if it is running by doing:

```
sudo dmesg | grep landlock || journalctl -kg landlock
```

If you are interested in reading more about Landlock, you can use https://docs.kernel.org/userspace-api/landlock.html as entry point.

12.8 systemd notification

12.8.1 Introduction

Suricata supports systemd notification with the aim of notifying the service manager of successful initialisation. The purpose is to enable the ability to start upon/await successful start-up for services/test frameworks that depend on a fully initialised Suricata .

During the initialisation phase Suricata synchronises the initialisation thread with all active threads to ensure they are in a running state. Once synchronisation has been completed a `READY=1` status notification is sent to the service manager using across the Systemd UNIX socket.

The path of the UNIX socket is taken from the `NOTIFY_SOCKET` env var.

12.8.2 Example

A test framework requires Suricata to be capturing before the tests can be carried out. Writing a `test.service` and ensuring the correct execution order with `After=suricata.service` forces the unit to be started after `suricata.service`. This does not enforce Suricata has fully initialised. By configuring `suricata.service` as `Type=notify` instructs the service manager to wait for the notification before starting `test.service`.

12.8.3 Requirements

This feature is only supported for distributions under the following conditions:

1. Any distribution that runs under **systemd**
2. Unit file configuration: `Type=notify`

For notification to the service manager the unit file must be configured as shown in requirement [2]. Upon all requirements being met the service manager will start and await `READY=1` status from Suricata. Otherwise the service manager will treat the service unit as `Type=simple` and consider it started immediately after the main process `ExecStart=` has been forked.

12.8.4 Additional Information

To confirm the system is running under systemd:

```
ps --no-headers -o comm 1
```

See https://www.freedesktop.org/software/systemd/man/systemd.service.html for help writing systemd unit files.

See https://www.freedesktop.org/software/systemd/man/devel/sd_notify.html#Notes for a discussion of the UNIX socket based notification.

12.9 Includes

A Suricata configuration file (typically `/etc/suricata/suricata.yaml`) may include other files allowing a configuration file to be broken into multiple files. The *special* field name `include` is used to include one or more files.

The contents of the *include* file are inlined at the level of the `include` statement. *Include* fields may also be included at any level within a mapping.

12.9.1 Including a Single File

```
include: filename.yaml
```

12.9.2 Including Multiple Files

```
include:
  - filename1.yaml
  - filename2.yaml
```

12.9.3 Include Inside a Mapping

```
vars:
  address-groups:
    include: address-groups.yaml
```

where `address-groups.yaml` contains:

```
%YAML 1.1
---
HOME_NET: "[192.168.0.0/16,10.0.0.0/8,172.16.0.0/12]"
```

is the equivalent of:

```
vars:
  address-groups:
    HOME_NET: "[192.168.0.0/16,10.0.0.0/8,172.16.0.0/12]"
```

Note

Suricata versions less than 7 required multiple `include` statements to be specified to include more than one file. While Suricata 7.0 still supports this it will issue a deprecation warning. Suricata 8.0 will not allow multiple `include` statements at the same level as this is not allowed by YAML.

CHAPTER

THIRTEEN

REPUTATION

13.1 IP Reputation

13.1.1 IP Reputation Config

IP reputation has a few configuration directives, all disabled by default.

```
# IP Reputation
#reputation-categories-file: /etc/suricata/iprep/categories.txt
#default-reputation-path: /etc/suricata/iprep
#reputation-files:
# - reputation.list
```

reputation-categories-file

The categories file mapping numbered category values to short names.

```
reputation-categories-file: /etc/suricata/iprep/categories.txt
```

default-reputation-path

Path where reputation files from the "reputation-files" directive are loaded from by default.

```
default-reputation-path: /etc/suricata/iprep
```

reputation-files

YAML list of file names to load. In case of a absolute path the file is loaded directly, otherwise the path from "default-reputation-path" is pre-pended to form the final path.

```
reputation-files:
 - badhosts.list
 - knowngood.list
 - sharedhosting.list
```

Hosts

IP reputation information is stored in the host table, so the settings of the host table affect it.

Depending on the number of hosts reputation information is available for, the memcap and hash size may have to be increased.

Reloads

Sending Suricata a USR2 signal will reload the IP reputation data, along with the normal rules reload.

During the reload the host table will be updated to contain the new data. The iprep information is versioned. When the reload is complete, Suricata will automatically clean up the old iprep information.

Only the reputation files will be reloaded, the categories file won't be. If categories change, Suricata should be restarted.

File format

The format of the reputation files is described in the *IP Reputation Format* page.

13.1.2 IP Reputation Format

Description of IP Reputation file formats. For the configuration see *IP Reputation Config* and *IP Reputation Keyword* for the rule format.

Categories file

The categories file provides a mapping between a category number, short name, and long description. It's a simple CSV file:

```
<id>,<short name>,<description>
```

Example:

```
1,BadHosts,Known bad hosts
2,Google,Known google host
```

The maximum value for the category id is hard coded at 60 currently.

Reputation file

The reputation file lists a reputation score for hosts in the categories. It's a simple CSV file:

```
<ip>,<category>,<reputation score>
```

The IP is an IPv4 address in the quad-dotted notation or an IPv6 address. Both IP types support networks in CIDR notation. The category is the number as defined in the categories file. The reputation score is the confidence that this IP is in the specified category, represented by a number between 1 and 127 (0 means no data).

Example:

```
1.2.3.4,1,101
1.1.1.0/24,6,88
```

If an IP address has a score in multiple categories it should be listed in the file multiple times.

Example:

```
1.1.1.1,1,10
1.1.1.1,2,10
```

This lists 1.1.1.1 in categories 1 and 2, each with a score of 10.

The purpose of the IP reputation component is the ranking of IP Addresses within the Suricata Engine. It will collect, store, update and distribute reputation intelligence on IP Addresses. The hub and spoke architecture will allows the

central database (The Hub) to collect, store and compile updated IP reputation details that are then distributed to user-side sensor databases (Spokes) for inclusion in user security systems. The reputation data update frequency and security action taken, is defined in the user security configuration.

The intent of IP Reputation is to allow sharing of intelligence regarding a vast number of IP addresses. This can be positive or negative intelligence classified into a number of categories. The technical implementation requires three major efforts; engine integration, the hub that redistributes reputation, and the communication protocol between hubs and sensors. The hub will have a number of responsibilities. This will be a separate module running on a separate system as any sensor. Most often it would run on a central database that all sensors already have communication with. It will be able to subscribe to one or more external feeds. The local admin should be able to define the feeds to be subscribed to, provide authentication credentials if required, and give a weight to that feed. The weight can be an overall number or a by category weight. This will allow the admin to minimize the influence a feed has on their overall reputation if they distrust a particular category or feed, or trust another implicitly. Feeds can be configured to accept feedback or not and will report so on connect. The admin can override and choose not to give any feedback, but the sensor should report these to the Hub upstream on connect. The hub will take all of these feeds and aggregate them into an average single score for each IP or IP Block, and then redistribute this data to all local sensors as configured. It should receive connections from sensors. The sensor will have to provide authentication and will provide feedback. The hub should redistribute that feedback from sensors to all other sensors as well as up to any feeds that accept feedback. The hub should also have an API to allow outside statistical analysis to be done to the database and fed back into the stream. For instance a local site may choose to change the reputation on all Russian IP blocks, etc.

For more information about IP Reputation see *IP Reputation Config*, *IP Reputation Keyword* and *IP Reputation Format*.

CHAPTER

FOURTEEN

INIT SCRIPTS

For Ubuntu with Upstart, the following can be used in `/etc/init/suricata.conf`:

```
# suricata
description "Intrusion Detection System Daemon"
start on runlevel [2345]
stop on runlevel [!2345]
expect fork
exec suricata -D --pidfile /var/run/suricata.pid -c /etc/suricata/suricata.yaml -i eth1
```

CHAPTER

FIFTEEN

OUTPUT

15.1 EVE

15.1.1 Eve JSON Output

The EVE output facility outputs alerts, anomalies, metadata, file info and protocol specific records through JSON.

The most common way to use this is through 'EVE', which is a firehose approach where all these logs go into a single file.

```
outputs:
  # Extensible Event Format (nicknamed EVE) event log in JSON format
  - eve-log:
      enabled: yes
      filetype: regular #regular|syslog|unix_dgram|unix_stream|redis
      filename: eve.json
      # Enable for multi-threaded eve.json output; output files are amended with
      # an identifier, e.g., eve.9.json
      #threaded: false
      # Specify the amount of buffering, in bytes, for
      # this output type. The default value 0 means "no
      # buffering".
      #buffer-size: 0
      #prefix: "@cee: " # prefix to prepend to each log entry
      # the following are valid when type: syslog above
      #identity: "suricata"
      #facility: local5
      #level: Info ## possible levels: Emergency, Alert, Critical,
                   ## Error, Warning, Notice, Info, Debug
      #ethernet: no  # log ethernet header in events when available
      #redis:
      #  server: 127.0.0.1
      #  port: 6379
      #  username: null ## ACL username; if null (default), no username is used
      #  password: null ## AUTH password; if null (default), authentication is disabled
      #  async: true ## if redis replies are read asynchronously
      #  mode: list ## possible values: list|lpush (default), rpush, channel|publish,␣
↪xadd|stream
      #             ## lpush and rpush are using a Redis list. "list" is an alias for␣
↪lpush
      #             ## publish is using a Redis channel. "channel" is an alias for␣
↪publish
```

(continues on next page)

(continued from previous page)

```
      #            ## xadd is using a Redis stream. "stream" is an alias for xadd
      #  key: suricata ## string denoting the key/channel/stream to use (default to␣
↪suricata)
      #  stream-maxlen: 100000        ## Automatically trims the stream length to at most
                                      ## this number of events. Set to 0 to disable␣
↪trimming.
                                      ## Only used when mode is set to xadd/stream.
      #  stream-trim-exact: false     ## Trim exactly to the maximum stream length above.
                                      ## Default: use inexact trimming (inexact by a few
                                      ## tens of items)
                                      ## Only used when mode is set to xadd/stream.
      # Redis pipelining set up. This will enable to only do a query every
      # 'batch-size' events. This should lower the latency induced by network
      # connection at the cost of some memory. There is no flushing implemented
      # so this setting should be reserved to high traffic Suricata deployments.
      #  pipelining:
      #    enabled: yes ## set enable to yes to enable query pipelining
      #    batch-size: 10 ## number of entries to keep in buffer

      # Include top level metadata. Default yes.
      #metadata: no

      # include the name of the input pcap file in pcap file processing mode
      pcap-file: false

      # Compress IPv6 addresses per RFC5952 as they are added to the eve-log. The␣
↪default is no.
      # ipv6-compress: no

      # Community Flow ID
      # Adds a 'community-id' field to EVE records. These are meant to give
      # records a predictable flow ID that can be used to match records to
      # output of other tools such as Zeek (Bro).
      #
      # Takes a 'seed' that needs to be same across sensors and tools
      # to make the id less predictable.

      # enable/disable the community id feature.
      community-id: false
      # Seed value for the ID output. Valid values are 0-65535.
      community-id-seed: 0

      # HTTP X-Forwarded-For support by adding an extra field or overwriting
      # the source or destination IP address (depending on flow direction)
      # with the one reported in the X-Forwarded-For HTTP header. This is
      # helpful when reviewing alerts for traffic that is being reverse
      # or forward proxied.
      xff:
        enabled: no
        # Two operation modes are available: "extra-data" and "overwrite".
        mode: extra-data
        # Two proxy deployments are supported: "reverse" and "forward". In
```

(continues on next page)

(continued from previous page)

```
        # a "reverse" deployment the IP address used is the last one, in a
        # "forward" deployment the first IP address is used.
        deployment: reverse
        # Header name where the actual IP address will be reported. If more
        # than one IP address is present, the last IP address will be the
        # one taken into consideration.
        header: X-Forwarded-For

      types:
        - alert:
            # payload: yes             # enable dumping payload in Base64
            # payload-buffer-size: 4kb # max size of payload buffer to output in eve-log
            # payload-printable: yes   # enable dumping payload in printable (lossy)␣
↪format
            # payload-length: yes      # enable dumping payload length, including the␣
↪gaps
            # packet: yes              # enable dumping of packet (without stream␣
↪segments)
            # metadata: no             # enable inclusion of app layer metadata with␣
↪alert. Default yes
            # If you want metadata, use:
            # metadata:
              # Include the decoded application layer (ie. http, dns)
              #app-layer: true
              # Log the current state of the flow record.
              #flow: true
              #rule:
                # Log the metadata field from the rule in a structured
                # format.
                #metadata: true
                # Log the raw rule text.
                #raw: false
                #reference: false      # include reference information from the rule
            # http-body: yes           # Requires metadata; enable dumping of HTTP body␣
↪in Base64
            # http-body-printable: yes # Requires metadata; enable dumping of HTTP body␣
↪in printable format
            # websocket-payload: yes   # Requires metadata; enable dumping of WebSocket␣
↪Payload in Base64
            # websocket-payload-printable: yes # Requires metadata; enable dumping of␣
↪WebSocket Payload in printable format

            # Enable the logging of tagged packets for rules using the
            # "tag" keyword.
            tagged-packets: yes
            # Enable logging the final action taken on a packet by the engine
            # (e.g: the alert may have action 'allowed' but the verdict be
            # 'drop' due to another alert. That's the engine's verdict)
            # verdict: yes
        # app layer frames
        - frame:
            # disabled by default as this is very verbose.
```

(continues on next page)

(continued from previous page)

```
            enabled: no
            # payload-buffer-size: 4kb # max size of frame payload buffer to output in␣
↪eve-log
        - anomaly:
            # Anomaly log records describe unexpected conditions such
            # as truncated packets, packets with invalid IP/UDP/TCP
            # length values, and other events that render the packet
            # invalid for further processing or describe unexpected
            # behavior on an established stream. Networks which
            # experience high occurrences of anomalies may experience
            # packet processing degradation.
            #
            # Anomalies are reported for the following:
            # 1. Decode: Values and conditions that are detected while
            # decoding individual packets. This includes invalid or
            # unexpected values for low-level protocol lengths as well
            # as stream related events (TCP 3-way handshake issues,
            # unexpected sequence number, etc).
            # 2. Stream: This includes stream related events (TCP
            # 3-way handshake issues, unexpected sequence number,
            # etc).
            # 3. Application layer: These denote application layer
            # specific conditions that are unexpected, invalid or are
            # unexpected given the application monitoring state.
            #
            # By default, anomaly logging is enabled. When anomaly
            # logging is enabled, applayer anomaly reporting is
            # also enabled.
            enabled: yes
            #
            # Choose one or more types of anomaly logging and whether to enable
            # logging of the packet header for packet anomalies.
            types:
              # decode: no
              # stream: no
              # applayer: yes
            #packethdr: no
        - http:
            extended: yes     # enable this for extended logging information
            # custom allows additional HTTP fields to be included in eve-log.
            # the example below adds three additional fields when uncommented
            #custom: [Accept-Encoding, Accept-Language, Authorization]
            # set this value to one and only one from {both, request, response}
            # to dump all HTTP headers for every HTTP request and/or response
            # dump-all-headers: none
        - dns:
            # This configuration uses the new DNS logging format,
            # the old configuration is still available:
            # https://docs.suricata.io/en/latest/output/eve/eve-json-output.html#dns-v1-
↪format

            # As of Suricata 5.0, version 2 of the eve dns output
```

(continues on next page)

(continued from previous page)

```
            # format is the default.
            #version: 2

            # Enable/disable this logger. Default: enabled.
            #enabled: yes

            # Control logging of requests and responses:
            # - requests: enable logging of DNS queries
            # - responses: enable logging of DNS answers
            # By default both requests and responses are logged.
            #requests: no
            #responses: no

            # Format of answer logging:
            # - detailed: array item per answer
            # - grouped: answers aggregated by type
            # Default: all
            #formats: [detailed, grouped]

            # DNS record types to log, based on the query type.
            # Default: all.
            #types: [a, aaaa, cname, mx, ns, ptr, txt]
        - tls:
            extended: yes     # enable this for extended logging information
            # output TLS transaction where the session is resumed using a
            # session id
            #session-resumption: no
            # custom controls which TLS fields that are included in eve-log
            # WARNING: enabling custom disables extended logging.
            #custom: [subject, issuer, session_resumed, serial, fingerprint, sni,␣
↪version, not_before, not_after, certificate, chain, ja3, ja3s, ja4, subjectaltname,␣
↪client, client_certificate, client_chain, client_alpns, server_alpns]
        - files:
            force-magic: no   # force logging magic on all logged files
            # force logging of checksums, available hash functions are md5,
            # sha1 and sha256
            #force-hash: [md5]
        #- drop:
        #    alerts: yes      # log alerts that caused drops
        #    flows: all       # start or all: 'start' logs only a single drop
        #                     # per flow direction. All logs each dropped pkt.
            # Enable logging the final action taken on a packet by the engine
            # (will show more information in case of a drop caused by 'reject')
            # verdict: yes
        - smtp:
            #extended: yes # enable this for extended logging information
            # this includes: bcc, message-id, subject, x_mailer, user-agent
            # custom fields logging from the list:
            #  reply-to, bcc, message-id, subject, x-mailer, user-agent, received,
            #  x-originating-ip, in-reply-to, references, importance, priority,
            #  sensitivity, organization, content-md5, date
            #custom: [received, x-mailer, x-originating-ip, relays, reply-to, bcc]
```

(continues on next page)

(continued from previous page)

```
            # output md5 of fields: body, subject
            # for the body you need to set app-layer.protocols.smtp.mime.body-md5
            # to yes
            #md5: [body, subject]

        #- dnp3
        - websocket
        - ftp
        - ftp-data
        - rdp
        - nfs
        - smb
        - tftp
        - ike
        - dcerpc
        - krb5
        - bittorrent-dht
        - ssh
        - arp:
            enabled: no
        - ntp:
            enabled: no
        - snmp
        - rfb
        - sip
        - quic
        - dhcp:
            enabled: yes
            # When extended mode is on, all DHCP messages are logged
            # with full detail. When extended mode is off (the
            # default), just enough information to map a MAC address
            # to an IP address is logged.
            extended: no
        - mqtt:
            # passwords: yes           # enable output of passwords
            # string-log-limit: 1kb    # limit size of logged strings in bytes.
                                       # Can be specified in kb, mb, gb. Just a number
                                       # is parsed as bytes. Default is 1KB.
                                       # Use a value of 0 to disable limiting.
                                       # Note that the size is also bounded by
                                       # the maximum parsed message size (see
                                       # app-layer configuration)
        - http2
        - pgsql:
            enabled: no
            # passwords: yes           # enable output of passwords. Disabled by default
        - stats:
            totals: yes       # stats for all threads merged together
            threads: no       # per thread stats
            deltas: no        # include delta values
            # Don't log stats counters that are zero. Default: true
            #null-values: false    # False will NOT log stats counters: 0
```

(continues on next page)

(continued from previous page)

```
        # bi-directional flows
        - flow
        # uni-directional flows
        #- netflow

        # Metadata event type. Triggered whenever a pktvar is saved
        # and will include the pktvars, flowvars, flowbits and
        # flowints.
        #- metadata

        # EXPERIMENTAL per packet output giving TCP state tracking details
        # including internal state, flags, etc.
        # This output is experimental, meant for debugging and subject to
        # change in both config and output without any notice.
        #- stream:
        #    all: false                     # log all TCP packets
        #    event-set: false               # log packets that have a decoder/stream␣
↪event
        #    state-update: false            # log packets triggering a TCP state update
        #    spurious-retransmission: false  # log spurious retransmission packets
        #
heartbeat:
  # The output-flush-interval value governs how often Suricata will flush
  # EVE log file output. Specify the value in seconds [1-60] and Suricata will
  # flush all active EVE log files at that interval. A value of 0 means
  # no EVE log output flushes are performed. When the EVE output
  # buffer-size value is non-zero, some EVE output that was written may remain
  # buffered. The output-flush-interval governs how much buffered data exists.
  #
  # The default value is: 0 (no periodic flushing)
  #output-flush-interval: 0
```

Each alert, http log, etc will go into this one file: 'eve.json'. This file can then be processed by 3rd party tools like Logstash (ELK) or jq.

If `ethernet` is set to yes, then ethernet headers will be added to events if available. If the `pkt_src` value is `stream (flow timeout)`, then the `ethernet` value will be populated with mac addresses from the flow's first packet with ethernet header.

If `suricata-version` is set to yes, then Suricata version, with its git revision if available, will be added to events as `suricata_version`.

If `ipv6-compress` is set to yes, then IPv6 addresses will be displayed in their compressed form according to RFC-5952. The default is `no` (display IPv6 addresses in their expanded form). Here's an example of an IPv6 address displayed with its compress value:

```
``fe80:0000:0000:0000:020c:29ff:faf2:ab42``
``fe80::20c:29ff:faf2:ab42``
```

Output Buffering

Output flushing is controlled by values in the configuration section `heartbeat`. By default, Suricata's output is synchronous with little possibility that written data will not be persisted. However, if `output.buffer-size` has a non-zero value, then some data may be written for the output, but not actually flushed. `buffer-size` bytes may be held in memory and written a short time later opening the possibility -- but limited -- for output data loss.

Hence, a heartbeat mechanism is introduced to limit the amount of time buffered data may exist before being flushed. A heartbeat thread periodically flushes all active EVE log files directly. With default values, there is no change in output buffering and flushing behavior. `output-flush-interval` controls how often Suricata will flush EVE output in a heartbeat fashion. A value of `0` means "never"; non-zero values must be in `[1-60]` seconds.

Flushing should be considered when `outputs.buffer-size` is greater than 0 to limit the amount and age of buffered, but not persisted, output data. Flushing is never needed when `buffer-size` is `0`.

```
heartbeat:
  #output-flush-interval: 0
```

Output types

EVE can output to multiple methods. `regular` is a normal file. Other options are `syslog`, `unix_dgram`, `unix_stream` and `redis`.

Output types:

```
filetype: regular #regular|syslog|unix_dgram|unix_stream|redis
filename: eve.json
# Enable for multi-threaded eve.json output; output files are amended
# with an identifier, e.g., eve.9.json. Default: off
#threaded: off
# Specify the amount of buffering, in bytes, for
# this output type. The default value 0 means "no
# buffering".
#buffer-size: 0
#prefix: "@cee: " # prefix to prepend to each log entry
# the following are valid when type: syslog above
#identity: "suricata"
#facility: local5
#level: Info ## possible levels: Emergency, Alert, Critical,
             ## Error, Warning, Notice, Info, Debug
#ethernet: no # log ethernet header in events when available
#suricata-version: no # include suricata version. Default no.
#redis:
#  server: 127.0.0.1
#  port: 6379
#  username: null ## ACL username; if null (default), no username is used
#  password: null ## AUTH password; if null (default), authentication is disabled
#  async: true ## if redis replies are read asynchronously
#  mode: list ## possible values: list|lpush (default), rpush, channel|publish,␣
↪xadd|stream
#             ## lpush and rpush are using a Redis list. "list" is an alias for lpush
#             ## publish is using a Redis channel. "channel" is an alias for publish
#             ## xadd is using a Redis stream. "stream" is an alias for xadd
#  key: suricata ## string denoting the key/channel/stream to use (default to suricata)
#  stream-maxlen: 100000        ## Automatically trims the stream length to at most
```

(continues on next page)

(continued from previous page)

```
                                ## this number of events. Set to 0 to disable trimming.
                                ## Only used when mode is set to xadd/stream.
#  stream-trim-exact: false     ## Trim exactly to the maximum stream length above.
                                ## Default: use inexact trimming (inexact by a few
                                ## tens of items)
                                ## Only used when mode is set to xadd/stream.
# Redis pipelining set up. This will enable to only do a query every
# 'batch-size' events. This should lower the latency induced by network
# connection at the cost of some memory. There is no flushing implemented
# so this setting as to be reserved to high traffic suricata.
#  pipelining:
#    enabled: yes ## set enable to yes to enable query pipelining
#    batch-size: 10 ## number of entry to keep in buffer
```

Alerts

Alerts are event records for rule matches. They can be amended with metadata, such as the application layer record (HTTP, DNS, etc) an alert was generated for, and elements of the rule.

The alert is amended with application layer metadata for signatures using application layer keywords. It is also the case for protocols over UDP as each single packet is expected to contain a PDU.

For other signatures, the option `guess-applayer-tx` can be used to force the detect engine to tie a transaction to an alert. This transaction is not guaranteed to be the relevant one, depending on your use case and how you define relevant here. **WARNING: If there are multiple live transactions, none will get picked up.** This is to reduce the chances of logging unrelated data, and may lead to alerts being logged without metadata, in some cases. The alert event will have `tx_guessed: true` to recognize such alerts.

Metadata:

```
- alert:
    #payload: yes             # enable dumping payload in Base64
    #payload-buffer-size: 4kb # max size of payload buffer to output in eve-log
    #payload-printable: yes   # enable dumping payload in printable (lossy) format
    #payload-length: yes      # enable dumping payload length, including the gaps
    #packet: yes              # enable dumping of packet (without stream segments)
    #http-body: yes           # Requires metadata; enable dumping of http body in Base64
    #http-body-printable: yes # Requires metadata; enable dumping of http body in␣
↪printable format

    # metadata:

      # Include the decoded application layer (ie. http, dns)
      #app-layer: true

      # Log the current state of the flow record.
      #flow: true

      #rule:
        # Log the metadata field from the rule in a structured
        # format.
        #metadata: true
```

(continues on next page)

(continued from previous page)

```
        # Log the raw rule text.
        #raw: false

        # Include the rule reference information
        #reference: false
```

Anomaly

Anomalies are event records created when packets with unexpected or anomalous values are handled. These events include conditions such as incorrect protocol values, incorrect protocol length values, and other conditions which render the packet suspect. Other conditions may occur during the normal progression of a stream; these are termed `stream` events are include control sequences with incorrect values or that occur out of expected sequence.

Anomalies are reported by and configured by type:

- Decode
- Stream
- Application layer

Metadata:

```
- anomaly:
    # Anomaly log records describe unexpected conditions such as truncated packets,
    # packets with invalid IP/UDP/TCP length values, and other events that render
    # the packet invalid for further processing or describe unexpected behavior on
    # an established stream. Networks which experience high occurrences of
    # anomalies may experience packet processing degradation.
    #
    # Anomalies are reported for the following:
    # 1. Decode: Values and conditions that are detected while decoding individual
    #    packets. This includes invalid or unexpected values for low-level protocol
    #    lengths as well.
    # 2. Stream: This includes stream related events (TCP 3-way handshake issues,
    #    unexpected sequence number, etc).
    # 3. Application layer: These denote application layer specific conditions that
    #    are unexpected, invalid or are unexpected given the application monitoring
    #    state.
    #
    # By default, anomaly logging is disabled. When anomaly logging is enabled,
    # application-layer anomaly reporting is enabled.
    #
    # Choose one or both types of anomaly logging and whether to enable
    # logging of the packet header for packet anomalies.
    types:
      #decode: no
      #stream: no
      #applayer: yes
    #packethdr: no
```

HTTP

HTTP transaction logging.

Config:

```
- http:
    extended: yes     # enable this for extended logging information
    # custom allows additional http fields to be included in eve-log
    # the example below adds three additional fields when uncommented
    #custom: [Accept-Encoding, Accept-Language, Authorization]
    # set this value to one among {both, request, response} to dump all
    # http headers for every http request and/or response
    # dump-all-headers: [both, request, response]
```

List of custom fields:

Yaml Option	HTTP Header
accept	accept
accept_charset	accept-charset
accept_encoding	accept-encoding
accept_language	accept-language
accept_datetime	accept-datetime
authorization	authorization
cache_control	cache-control
cookie	cookie
from	from
max_forwards	max-forwards
origin	origin
pragma	pragma
proxy_authorization	proxy-authorization
range	range
te	te
via	via
x_requested_with	x-requested-with
dnt	dnt
x_forwarded_proto	x-forwarded-proto
x_authenticated_user	x-authenticated-user
x_flash_version	x-flash-version
accept_range	accept-range
age	age
allow	allow
connection	connection
content_encoding	content-encoding
content_language	content-language
content_length	content-length
content_location	content-location
content_md5	content-md5
content_range	content-range
content_type	content-type
date	date
etag	etags
expires	expires

continues on next page

Table 1 – continued from previous page

Yaml Option	HTTP Header
last_modified	last-modified
link	link
location	location
proxy_authenticate	proxy-authenticate
referer	referer
refresh	refresh
retry_after	retry-after
server	server
set_cookie	set-cookie
trailer	trailer
transfer_encoding	transfer-encoding
upgrade	upgrade
vary	vary
warning	warning
www_authenticate	www-authenticate
true_client_ip	true-client-ip
org_src_ip	org-src-ip
x_bluecoat_via	x-bluecoat-via

In the `custom` option values from both columns can be used. The `HTTP Header` column is case insensitive.

DNS

Note

As of Suricata 7.0 the v1 EVE DNS format has been removed.

Version 2 EVE DNS will be removed in Suricata 9.

DNS records are logged as one entry for the request, and one entry for the response.

YAML:

```
- dns:
    #version: 3

    # Enable/disable this logger. Default: enabled.
    #enabled: yes

    # Control logging of requests and responses:
    # - requests: enable logging of DNS queries
    # - responses: enable logging of DNS answers
    # By default both requests and responses are logged.
    #requests: no
    #responses: no

    # Format of answer logging:
    # - detailed: array item per answer
    # - grouped: answers aggregated by type
    # Default: all
```

(continues on next page)

(continued from previous page)

```
    #formats: [detailed, grouped]

    # Types to log, based on the query type.
    # Default: all.
    #types: [a, aaaa, cname, mx, ns, ptr, txt]
```

TLS

TLS records are logged one record per session.

YAML:

```
- tls:
    extended: yes     # enable this for extended logging information
    # custom allows to control which tls fields that are included
    # in eve-log
    #custom: [subject, issuer, serial, fingerprint, sni, version, not_before, not_after,␣
↪certificate, chain, ja3, ja3s, ja4]
```

The default is to log certificate subject and issuer. If `extended` is enabled, then the log gets more verbose.

By using `custom` it is possible to select which TLS fields to log. **Note that this will disable ``extended`` logging.**

ARP

ARP records are logged as one entry for the request, and one entry for the response.

YAML:

```
- arp:
    enabled: no
```

The logger is disabled by default since ARP can generate a large number of events.

Netflow

Netflow records closely relate to flow records except that they are unidirectional while flow records are bidirectional. This means that there will be twice as many netflow records as there are flow records.

Netflow records are disabled by default.

YAML:

```
#- netflow
```

To enable netflow, change this to:

```
- netflow:
    enabled: yes
```

MQTT

EVE-JSON output for MQTT consists of one object per MQTT transaction, with some common and various type-specific fields. Two aspects can be configured:

YAML:

```
- mqtt:
    # passwords: yes           # enable output of passwords
    # string-log-limit: 1kb    # limit size of logged strings in bytes.
                               # Can be specified in kb, mb, gb. Just a number
                               # is parsed as bytes. Default is 1KB.
                               # Use a value of 0 to disable limiting.
                               # Note that the size is also bounded by
                               # the maximum parsed message size (see
                               # app-layer configuration)
```

The default is to output passwords in cleartext and not to limit the size of message payloads. Depending on the kind of context the parser is used in (public output, frequent binary transmissions, ...) this can be configured for regular `mqtt` events.

Drops

Drops are event types logged when the engine drops a packet.

Config:

```
- drop:
    alerts: yes      # log alerts that caused drops
    flows: all       # start or all: 'start' logs only a single drop
                     # per flow direction. All logs each dropped pkt.
    # Enable logging the final action taken on a packet by the engine
    # (will show more information in case of a drop caused by 'reject')
    verdict: yes
```

Stats

Zero-valued Counters

While the human-friendly *stats.log* output will only log out non-zeroed counters, by default EVE Stats logs output all enabled counters, which may lead to fairly verbose logs.

To reduce log file size, one may set *null-values* to false. Do note that this may impact on the visibility of information for which a stats counter as zero is relevant.

Config:

```
- stats:
    # Don't log stats counters that are zero. Default: true
    #null-values: false    # False will NOT log stats counters: 0
```

Date modifiers in filename

It is possible to use date modifiers in the eve-log filename.

```
outputs:
  - eve-log:
      filename: eve-%s.json
```

The example above adds epoch time to the filename. All the date modifiers from the C library should be supported. See the man page for `strftime` for all supported modifiers.

Threaded file output

By default, all output is written to the named filename in the outputs section. The `threaded` option enables each output thread to write to individual files. In this case, the `filename` will include a unique identifier.

With `threaded` enabled, the output will be split among many files -- and the aggregate of each file's contents must be treated together.

```
outputs:
  - eve-log:
      filename: eve.json
      threaded: on
```

This example will cause each Suricata thread to write to its own "eve.json" file. Filenames are constructed by adding a unique identifier to the filename. For example, `eve.7.json`.

Rotate log file

Eve-log can be configured to rotate based on time.

```
outputs:
  - eve-log:
      filename: eve-%Y-%m-%d-%H:%M.json
      rotate-interval: minute
```

The example above creates a new log file each minute, where the filename contains a timestamp. Other supported `rotate-interval` values are `hour` and `day`. These values rotate the log file when the minute, hour or date of the local time changes. For example, `day` rotates at local midnight, also across daylight saving time changes, or at the first local time of the new date where such a change skips midnight. When the clock is set back at the end of daylight saving time, the repeated hour is part of the same hour period, so with `hour` a single file holds both passes of that hour.

In addition to this, it is also possible to specify the `rotate-interval` as a relative value. One example is to rotate the log file each X seconds.

```
outputs:
  - eve-log:
      filename: eve-%Y-%m-%d-%H:%M:%S.json
      rotate-interval: 30s
```

The example above rotates eve-log each 30 seconds. This could be replaced with `30m` to rotate every 30 minutes, `30h` to rotate every 30 hours, `30d` to rotate every 30 days, or `30w` to rotate every 30 weeks. Relative values are counted from startup or the last SIGHUP and are not aligned to the local time, so e.g. `24h` or `1d` do not rotate at local midnight like `day` does.

Multiple Logger Instances

It is possible to have multiple 'EVE' instances, for example the following is valid:

```
outputs:
  - eve-log:
      enabled: yes
      type: file
      filename: eve-ips.json
      types:
        - alert
        - drop
```

(continues on next page)

(continued from previous page)

```
  - eve-log:
      enabled: yes
      type: file
      filename: eve-nsm.json
      types:
        - http
        - dns
        - tls
```

So here the alerts and drops go into 'eve-ips.json', while http, dns and tls go into 'eve-nsm.json'.

With the exception of `drop`, you can specify multiples of the same logger type, however, `drop` can only be used once.

> **Note**
>
> The use of independent json loggers such as alert-json-log, dns-json-log, etc. has been deprecated and will be removed by June 2020. Please use multiple eve-log instances as documented above instead. Please see the deprecation policy for more information.

File permissions

Log file permissions can be set individually for each logger. `filemode` can be used to control the permissions of a log file, e.g.:

```
outputs:
  - eve-log:
      enabled: yes
      filename: eve.json
      filemode: 600
```

The example above sets the file permissions on `eve.json` to 600, which means that it is only readable and writable by the owner of the file.

JSON flags

Several flags can be specified to control the JSON output in EVE:

```
outputs:
  - eve-log:
      json:
        # Sort object keys in the same order as they were inserted
        preserve-order: yes

        # Make the output more compact
        compact: yes

        # Escape all unicode characters outside the ASCII range
        ensure-ascii: yes

        # Escape the '/' characters in string with '\/'
        escape-slash: yes
```

All these flags are enabled by default, and can be modified per EVE instance.

Community Flow ID

Often Suricata is used in combination with other tools like Bro/Zeek. Enabling the community-id option in the eve-log section adds a new `community_id` field to each output.

Example:

```
{
  "timestamp": "2003-12-16T13:21:44.891921+0000",
  "flow_id": 1332028388187153,
  "pcap_cnt": 1,
  "event_type": "alert",
  ...
  "community_id": "1:LQU9qZlK+B5F3KDmev6m5PMibrg=",
  "alert": {
    "action": "allowed",
    "gid": 1,
    "signature_id": 1,
  },
}
{
  "timestamp": "2003-12-16T13:21:45.037333+0000",
  "flow_id": 1332028388187153,
  "event_type": "flow",
  "flow": {
    "pkts_toserver": 5,
    "pkts_toclient": 4,
    "bytes_toserver": 338,
    "bytes_toclient": 272,
    "start": "2003-12-16T13:21:44.891921+0000",
    "end": "2003-12-16T13:21:45.346457+0000",
    "age": 1,
    "state": "closed",
    "reason": "shutdown",
    "alerted": true
  },
  "community_id": "1:LQU9qZlK+B5F3KDmev6m5PMibrg=",
}
```

Options

The output can be enabled per instance of the EVE logger.

The `community-id` option is boolean. If set to `true` it is enabled. The `community-id-seed` option specifies an unsigned 16 bit value that is used to seed the hash that is calculated for the `community-id` output. This must be set to the same value on all tools that output this record.

YAML:

```
- eve-log:
    # Community Flow ID
    # Adds a 'community_id' field to EVE records. These are meant to give
    # a records a predictable flow id that can be used to match records to
```

(continues on next page)

(continued from previous page)

```
    # output of other tools such as Bro.
    #
    # Takes a 'seed' that needs to be same across sensors and tools
    # to make the id less predictable.

    # enable/disable the community id feature.
    community-id: false
    # Seed value for the ID output. Valid values are 0-65535.
    community-id-seed: 0
```

Multi Tenancy

Suricata can be configured to support multiple tenants with different detection engine configurations. When these tenants are configured and the detection engine is running then all EVE logging will also report the `tenant_id` field for traffic matching a specific tenant.

15.1.2 Eve JSON Format

Example:

```
{
  "timestamp": "2017-04-07T22:24:37.251547+0100",
  "flow_id": 586497171462735,
  "pcap_cnt": 53381,
  "event_type": "alert",
  "src_ip": "192.168.2.14",
  "src_port": 50096,
  "dest_ip": "209.53.113.5",
  "dest_port": 80,
  "proto": "TCP",
  "metadata": {
    "flowbits": [
      "http.dottedquadhost"
    ]
  },
  "tx_id": 4,
  "alert": {
    "action": "allowed",
    "gid": 1,
    "signature_id": 2018358,
    "rev": 10,
    "signature": "ET HUNTING GENERIC SUSPICIOUS POST to Dotted Quad with Fake Browser 1",
    "category": "Potentially Bad Traffic",
    "severity": 2
  },
  "app_proto": "http"
}
```

Common Section

All the JSON log types share a common structure:

```
{"timestamp":"2009-11-24T21:27:09.534255","flow_id":ID_NUMBER, "event_type":"TYPE", ...
↪tuple... ,"TYPE":{ ... type specific content ... }}
```

Field: flow_id

Correlates the network protocol, flow logs EVE data and any evidence that Suricata has logged to an `alert` event and that alert's metadata, as well as to `fileinfo`/file transaction and anomaly logs, if available. The same correlation and logs are produced regardless if there is an alert, for any session/flow.

The ability to correlate EVE logs belonging to a specific session/flow was introduced in 2014 (see commit f1185d051c21).

Further below, you can see several examples of events logged by Suricata: an *alert* for an `HTTP` rule, `fileinfo`, *http*, *anomaly*, and *flow* events, all easily correlated using the `flow_id` EVE field:

```
$ jq 'select(.flow_id==1676750115612680)' eve.json
```

Event type: `alert`:

```
{
  "timestamp": "2023-09-18T06:13:41.532140+0000",
  "flow_id": 1676750115612680,
  "pcap_cnt": 130,
  "event_type": "alert",
  "src_ip": "142.11.240.191",
  "src_port": 35361,
  "dest_ip": "192.168.100.237",
  "dest_port": 49175,
  "proto": "TCP",
  "pkt_src": "wire/pcap",
  "ether": {
    "src_mac": "52:54:00:36:3e:ff",
    "dest_mac": "12:a9:86:6c:77:de"
  },
  "tx_id": 1,
  "alert": {
    "action": "allowed",
    "gid": 1,
    "signature_id": 2045001,
    "rev": 1,
    "signature": "ET ATTACK_RESPONSE Win32/LeftHook Stealer Browser Extension Config␣
↪Inbound",
    "category": "A Network Trojan was detected",
    "severity": 1,
    "metadata": {
      "affected_product": [
        "Windows_XP_Vista_7_8_10_Server_32_64_Bit"
      ],
      "attack_target": [
        "Client_Endpoint"
      ],
```

(continues on next page)

(continued from previous page)


```
      "created_at": [
        "2023_04_17"
      ],
      "deployment": [
        "Perimeter"
      ],
      "former_category": [
        "ATTACK_RESPONSE"
      ],
      "signature_severity": [
        "Major"
      ],
      "updated_at": [
        "2023_04_18"
      ]
    }
  },
  "http": {
    "hostname": "142.11.240.191",
    "http_port": 35361,
    "url": "/",
    "http_content_type": "text/xml",
    "http_method": "POST",
    "protocol": "HTTP/1.1",
    "status": 200,
    "length": 5362
  },
  "files": [
    {
      "filename": "/",
      "gaps": false,
      "state": "CLOSED",
      "stored": false,
      "size": 5362,
      "tx_id": 1
    }
  ],
  "app_proto": "http",
  "direction": "to_client",
  "flow": {
    "pkts_toserver": 13,
    "pkts_toclient": 12,
    "bytes_toserver": 1616,
    "bytes_toclient": 8044,
    "start": "2023-09-18T06:13:33.324862+0000",
    "src_ip": "192.168.100.237",
    "dest_ip": "142.11.240.191",
    "src_port": 49175,
    "dest_port": 35361
  }
}
```


Event type: `fileinfo`:

```
{
  "timestamp": "2023-09-18T06:13:33.903924+0000",
  "flow_id": 1676750115612680,
  "pcap_cnt": 70,
  "event_type": "fileinfo",
  "src_ip": "192.168.100.237",
  "src_port": 49175,
  "dest_ip": "142.11.240.191",
  "dest_port": 35361,
  "proto": "TCP",
  "pkt_src": "wire/pcap",
  "ether": {
    "src_mac": "12:a9:86:6c:77:de",
    "dest_mac": "52:54:00:36:3e:ff"
  },
  "http": {
    "hostname": "142.11.240.191",
    "http_port": 35361,
    "url": "/",
    "http_content_type": "text/xml",
    "http_method": "POST",
    "protocol": "HTTP/1.1",
    "status": 200,
    "length": 212
  },
  "app_proto": "http",
  "fileinfo": {
    "filename": "/",
    "gaps": false,
    "state": "CLOSED",
    "stored": false,
    "size": 137,
    "tx_id": 0
  }
}
```


Event type: HTTP:


```
{
  "timestamp": "2023-09-18T06:13:33.903924+0000",
  "flow_id": 1676750115612680,
  "pcap_cnt": 70,
  "event_type": "http",
  "src_ip": "192.168.100.237",
  "src_port": 49175,
  "dest_ip": "142.11.240.191",
  "dest_port": 35361,
  "proto": "TCP",
  "pkt_src": "wire/pcap",
  "ether": {
    "src_mac": "12:a9:86:6c:77:de",
    "dest_mac": "52:54:00:36:3e:ff"
  },
```


(continues on next page)

(continued from previous page)


```
  "tx_id": 0,
  "http": {
    "hostname": "142.11.240.191",
    "http_port": 35361,
    "url": "/",
    "http_content_type": "text/xml",
    "http_method": "POST",
    "protocol": "HTTP/1.1",
    "status": 200,
    "length": 212,
    "request_headers": [
      {
        "name": "Content-Type",
        "value": "text/xml; charset=utf-8"
      },
      {
        "name": "SOAPAction",
        "value": "\"http://tempuri.org/Endpoint/CheckConnect\""
      },
      {
        "name": "Host",
        "value": "142.11.240.191:35361"
      },
      {
        "name": "Content-Length",
        "value": "137"
      },
      {
        "name": "Expect",
        "value": "100-continue"
      },
      {
        "name": "Accept-Encoding",
        "value": "gzip, deflate"
      },
      {
        "name": "Connection",
        "value": "Keep-Alive"
      }
    ],
    "response_headers": [
      {
        "name": "Content-Length",
        "value": "212"
      },
      {
        "name": "Content-Type",
        "value": "text/xml; charset=utf-8"
      },
      {
        "name": "Server",
        "value": "Microsoft-HTTPAPI/2.0"
```


(continues on next page)

(continued from previous page)

```
      },
      {
        "name": "Date",
        "value": "Mon, 18 Sep 2023 06:13:33 GMT"
      }
    ]
  }
}
```

Event type: `anomaly`:

```
{
  "timestamp": "2023-09-18T06:13:58.882971+0000",
  "flow_id": 1676750115612680,
  "pcap_cnt": 2878,
  "event_type": "anomaly",
  "src_ip": "192.168.100.237",
  "src_port": 49175,
  "dest_ip": "142.11.240.191",
  "dest_port": 35361,
  "proto": "TCP",
  "pkt_src": "wire/pcap",
  "ether": {
    "src_mac": "12:a9:86:6c:77:de",
    "dest_mac": "52:54:00:36:3e:ff"
  },
  "tx_id": 3,
  "anomaly": {
    "app_proto": "http",
    "type": "applayer",
    "event": "UNABLE_TO_MATCH_RESPONSE_TO_REQUEST",
    "layer": "proto_parser"
  }
}
```

Event type: `flow`:

```
{
  "timestamp": "2023-09-18T06:13:21.216460+0000",
  "flow_id": 1676750115612680,
  "event_type": "flow",
  "src_ip": "192.168.100.237",
  "src_port": 49175,
  "dest_ip": "142.11.240.191",
  "dest_port": 35361,
  "proto": "TCP",
  "app_proto": "http",
  "flow": {
    "pkts_toserver": 3869,
    "pkts_toclient": 1523,
    "bytes_toserver": 3536402,
    "bytes_toclient": 94102,
```

(continues on next page)

(continued from previous page)


```
    "start": "2023-09-18T06:13:33.324862+0000",
    "end": "2023-09-18T06:14:13.752399+0000",
    "age": 40,
    "state": "closed",
    "reason": "shutdown",
    "alerted": true,
    "exception_policy": [
      {
        "target": "stream_midstream",
        "policy": "ignore"
      }
    ]
  },
  "ether": {
    "dest_macs": [
      "52:54:00:36:3e:ff"
    ],
    "src_macs": [
      "12:a9:86:6c:77:de"
    ]
  },
  "tcp": {
    "tcp_flags": "1e",
    "tcp_flags_ts": "1e",
    "tcp_flags_tc": "1a",
    "syn": true,
    "rst": true,
    "psh": true,
    "ack": true,
    "state": "closed",
    "ts_max_regions": 1,
    "tc_max_regions": 1
  }
}
```


Note

It is possible to have even more detailed alert records, by enabling for instance logging http-body, or alert metadata (*alert output*).

Examples come from pcap found at https://app.any.run/tasks/ce7ca983-9e4b-4251-a7c3-fefa3da02ebe/.

Event types

The common part has a field "event_type" to indicate the log type.


```
"event_type":"TYPE"
```


When an application layer protocol event is detected, the common section will have an `app_proto` field.


```
"app_proto": "http"
```

PCAP fields

If Suricata is processing a pcap file, additional fields are added:

```
"pcap_cnt": 123
```

`pcap_cnt` contains the packet number in the pcap. This can be used to look up a packet in Wireshark for example.

```
"pcap_filename":"/path/to/file.pcap"
```

`pcap_filename` contains the file name and location of the pcap that generated the event.

> **Note**
>
> the pcap fields are only available on "real" packets, and are omitted from internal "pseudo" packets such as flow timeout packets.

Event type: Alert

This field contains data about a signature that matched, such as `signature_id` (`sid` in the rule) and the `signature` (`msg` in the rule).

It can also contain information about Source and Target of the attack in the `alert.source` and `alert.target` field if target keyword is used in the signature.

In firewall mode, the `alert.engine` field identifies which rule engine generated the alert: `fw` for firewall rules and `td` for threat detect rules. This field is omitted outside of firewall mode.

This event will also have the `pcap_cnt` field, when running in pcap mode, to indicate which packet triggered the signature.

```
"alert": {
  "action": "allowed",
  "gid": 1,
  "signature_id": 2024056,
  "rev": 4,
  "signature": "ET MALWARE Win32/CryptFile2 / Revenge Ransomware Checkin M3",
  "category": "Malware Command and Control Activity Detected",
  "severity": 1,
  "metadata": {
    "affected_product": [
      "Windows_XP_Vista_7_8_10_Server_32_64_Bit"
    ],
    "attack_target": [
      "Client_Endpoint"
    ],
    "created_at": [
      "2017_03_15"
    ],
    "deployment": [
      "Perimeter"
    ],
    "former_category": [
      "MALWARE"
    ],
```

(continues on next page)

(continued from previous page)

```
    "malware_family": [
      "CryptFile2"
    ],
    "performance_impact": [
      "Moderate"
    ],
    "signature_severity": [
      "Major"
    ],
    "updated_at": [
      "2020_08_04"
    ]
  }
},
```

Action field

Possible values: "allowed" and "blocked".

Example:

```
"action":"allowed"
```

Action is set to "allowed" unless a rule used the "drop" action and Suricata is in IPS mode, or when the rule used the "reject" action. It is important to note that this does not necessarily indicate the final verdict for a given packet or flow, since one packet may match on several rules.

Verdict

An object containing info on the final action that will be applied to a given packet, based on all the signatures triggered by it and other possible events (e.g., a flow drop). For that reason, it is possible for an alert with an action `allowed` to have a verdict `drop`, in IPS mode, for instance, if that packet was dropped due to a different alert.

- Action: `alert`, `pass`, `drop` (this latter only occurs in IPS mode)
- Reject-target: `to_server`, `to_client`, `both` (only occurs for 'reject' rules)
- Reject: an array of strings with possible reject types: `tcp-reset`, `icmp-prohib` (only occurs for 'reject' rules)

Example:

```
"verdict": {
   "action": "drop",
   "reject_target": "to_client",
   "reject": "[icmp-prohib]"
 }
```

Pcap Field

If pcap log capture is active in *multi* mode, a *capture_file* key will be added to the event with value being the full path of the pcap file where the corresponding packets have been extracted.

Event type: Anomaly

Events with type "anomaly" report unexpected conditions such as truncated packets, packets with invalid values, events that render the packet invalid for further processing or unexpected behaviors.

Networks which experience high occurrences of anomalies may experience packet processing degradation when anomaly logging is enabled.

Fields

- "type": Either "decode", "stream" or "applayer". In rare cases, type will be "unknown". When this occurs, an additional field named "code" will be present. Events with type "applayer" are detected by the application layer parsers.
- "event" The name of the anomalous event. Events of type "decode" are prefixed with "decoder"; events of type "stream" are prefixed with "stream".
- "code" If "type" is "unknown", than "code" contains the unrecognized event code. Otherwise, this field is not present.

The following field is included when "type" has the value "applayer":

- "layer" Indicates the handling layer that detected the event. This will be "proto_parser" (protocol parser), "proto_detect" (protocol detection) or "parser."

When `packethdr` is enabled, the first 32 bytes of the packet are included as a byte64-encoded blob in the main part of record. This applies to events of "type" "packet" or "stream" only.

Examples

```
"anomaly": {
  "type": "decode",
  "event": "decoder.icmpv4.unknown_type"
}

"anomaly": {
  "type": "decode",
  "event": "decoder.udp.pkt_too_small"
}

"anomaly": {
  "type": "decode",
  "event": "decoder.ipv4.wrong_ip_version"
}

"anomaly": {
  "type": "stream",
  "event": "stream.pkt_invalid_timestamp"
}

{
  "timestamp": "1969-12-31T16:04:21.000000-0800",
  "pcap_cnt": 9262,
  "event_type": "anomaly",
  "src_ip": "208.21.2.184",
  "src_port": 0,
```

(continues on next page)

(continued from previous page)


```
  "dest_ip": "10.1.1.99",
  "dest_port": 0,
  "proto": "UDP",
  "packet": "////////AQEBAQEBCABFAAA8xZ5AAP8R1+DQFQK4CgE=",
  "packet_info": {
    "linktype": 1
  },
  "anomaly": {
    "type": "decode",
    "event": "decoder.udp.pkt_too_small"
  }
}

{
  "timestamp": "2016-01-11T05:10:54.612110-0800",
  "flow_id": 412547343494194,
  "pcap_cnt": 1391293,
  "event_type": "anomaly",
  "src_ip": "192.168.122.149",
  "src_port": 49324,
  "dest_ip": "69.195.71.174",
  "dest_port": 443,
  "proto": "TCP",
  "app_proto": "tls",
  "anomaly": {
    "type": "applayer",
    "event": "APPLAYER_DETECT_PROTOCOL_ONLY_ONE_DIRECTION",
    "layer": "proto_detect"
  }
}

{
  "timestamp": "2016-01-11T05:10:52.828802-0800",
  "flow_id": 201217772575257,
  "pcap_cnt": 1391281,
  "event_type": "anomaly",
  "src_ip": "192.168.122.149",
  "src_port": 49323,
  "dest_ip": "69.195.71.174",
  "dest_port": 443,
  "proto": "TCP",
  "tx_id": 0,
  "app_proto": "tls",
  "anomaly": {
    "type": "applayer",
    "event": "INVALID_RECORD_TYPE",
    "layer": "proto_parser"
  }
}
```

Event type: fileinfo

Note that the checksum values for `md5`, `sha1`, and `sha256` are available when

- The command line option `disable-hashing` was not used
- There are no gaps (areas missing)

Fields

- "end: The offset of the last byte captured
- "file_id": Integer value representing the id of a file that has been stored
- "filename": Name of the file as observed in network traffic
- "gaps": Boolean value indicating if there were gaps in the file
- "magic": [optional, requires libmagic] The magic value for the file
- "md5": Iff closed, md5 sum
- "sha1": Iff closed, sha1 sum
- "sha256": The sha256 value for the file, if available
- "sid": One or more signature ids that triggered a *filestore*
- "size": The observed size of the file, in bytes
- "start": The offset of the first byte captured
- "state": The state of the file when the record is written
- "stored": Boolean value indicating whether the file has been stored
- "storing": Boolean value indicating whether the file is in the process of being stored; true when not yet stored
- "tx_id": The transaction id in effect

Offset values

This example shows the offset values from a `fileinfo` event -- note the `http` content range *start* and *end* value are replicated in the `fileinfo` fields:

```
http.content_range.raw: bytes 500-1000/146515
http.content_range.start: 500
http.content_range.end: 1000
http.content_range.size: 146515
fileinfo.start: 500
fileinfo.end: 1000
```

Event type: HTTP

Fields

- "hostname": The hostname this HTTP event is attributed to
- "url": URL at the hostname that was accessed
- "http_user_agent": The user-agent of the software that was used
- "http_content_type": The type of data returned (ex: application/x-gzip)
- "cookie"

In addition to these fields, if the extended logging is enabled in the suricata.yaml file the following fields are (can) also included:

- "length": The content size of the HTTP body
- "status": HTTP status code
- "protocol": Protocol / Version of HTTP (ex: HTTP/1.1)
- "http_method": The HTTP method (ex: GET, POST, HEAD)
- "http_refer": The referer for this action

In addition to the extended logging fields one can also choose to enable/add from more than 50 additional custom logging HTTP fields enabled in the suricata.yaml file. The additional fields can be enabled as following:

```
- eve-log:
    enabled: yes
    type: file #file|syslog|unix_dgram|unix_stream
    filename: eve.json
    # the following are valid when type: syslog above
    #identity: "suricata"
    #facility: local5
    #level: Info ## possible levels: Emergency, Alert, Critical,
                 ## Error, Warning, Notice, Info, Debug
    types:
      - alert
      - http:
          extended: yes     # enable this for extended logging information
          # custom allows additional http fields to be included in eve-log
          # the example below adds three additional fields when uncommented
          #custom: [Accept-Encoding, Accept-Language, Authorization]
          custom: [accept, accept-charset, accept-encoding, accept-language,
          accept-datetime, authorization, cache-control, cookie, from,
          max-forwards, origin, pragma, proxy-authorization, range, te, via,
          x-requested-with, dnt, x-forwarded-proto, accept-range, age,
          allow, connection, content-encoding, content-language,
          content-length, content-location, content-md5, content-range,
          content-type, date, etags, expires, last-modified, link, location,
          proxy-authenticate, referer, refresh, retry-after, server,
          set-cookie, trailer, transfer-encoding, upgrade, vary, warning,
          www-authenticate, x-flash-version, x-authenticated-user]
```

The benefits here of using the extended logging is to see if this action for example was a POST or perhaps if a download of an executable actually returned any bytes.

It is also possible to dump every header for HTTP requests/responses or both via the keyword `dump-all-headers`.

Examples

Event with non-extended logging:

```
"http": {
    "hostname": "www.digip.org",
    "url" :"\/jansson\/releases\/jansson-2.6.tar.gz",
    "http_user_agent": "<User-Agent>",
```

(continues on next page)

(continued from previous page)

```
    "http_content_type": "application\/x-gzip"
}
```

In case the hostname shows a port number, such as in case there is a header "Host: www.test.org:1337":

```
"http": {
    "http_port": 1337,
    "hostname": "www.test.org",
    "url" :"\/this\/is\/test.tar.gz",
    "http_user_agent": "<User-Agent>",
    "http_content_type": "application\/x-gzip"
}
```

Event with extended logging:

```
"http": {
    "hostname": "direkte.vg.no",
    "url":".....",
    "http_user_agent": "<User-Agent>",
    "http_content_type": "application\/json",
    "http_refer": "http:\/\/www.vg.no\/",
    "http_method": "GET",
    "protocol": "HTTP\/1.1",
    "status":"200",
    "length":310
}
```

Event with `dump-all-headers` set to "both":

```
"http": {
    "hostname": "test.co.uk",
    "url":"\/test\/file.json",
    "http_user_agent": "<User-Agent>",
    "http_content_type": "application\/json",
    "http_refer": "http:\/\/www.test.com\/",
    "http_method": "GET",
    "protocol": "HTTP\/1.1",
    "status":"200",
    "length":310,
    "request_headers": [
        {
            "name": "User-Agent",
            "value": "Wget/1.13.4 (linux-gnu)"
        },
        {
            "name": "Accept",
            "value": "*/*"
        },
    ],
    "response_headers": [
        {
            "name": "Date",
            "value": "Wed, 25 Mar 2015 15:40:41 GMT"
```

(continues on next page)

(continued from previous page)

```
        },
    ]
}
```

Event type: DNS

DNS has 2 logging style that can be used together or independently:

- "detailed": "rrname", "rrtype", "rdata" and "ttl" fields are logged for each answer
- "grouped": answers logged are aggregated by their type (A, AAAA, NS, ...)

If no format is chosen, "detailed" will be used by default.

It will be still possible to use the old DNS logging format, you can control it with "version" option in dns configuration section.

Suricata 8.0.0 introduces version 3 of the DNS logging format. This update unifies the DNS logging style used by `dns` events as well as the `dns` object in `alert` records. See DNS Logging Changes for 8.0 for more details on the changes to logging format.

Note

Suricata 7 style DNS logging can be retained by setting the `version` field to 2, however this will be removed in Suricata 9.

Fields

Outline of fields seen in the different kinds of DNS events:

- "type": Indicating DNS message type, can be "request" or "response".
- "id": Identifier field
- "version": Indicating DNS logging version in use
- "flags": Indicating DNS answer flag, in hexadecimal (ex: 8180 , please note 0x is not output)
- "qr": Indicating in case of DNS answer flag, Query/Response flag (ex: true if set)
- "aa": Indicating in case of DNS answer flag, Authoritative Answer flag (ex: true if set)
- "tc": Indicating in case of DNS answer flag, Truncation flag (ex: true if set)
- "rd": Indicating in case of DNS answer flag, Recursion Desired flag (ex: true if set)
- "ra": Indicating in case of DNS answer flag, Recursion Available flag (ex: true if set)
- "z": Indicating in case of DNS answer flag, Reserved bit (ex: true if set)
- "rcode": (ex: NOERROR)
- "ttl": Time-To-Live for this resource record
- "queries": A list of query objects
- "answers": A list of answer objects
- "authorities": A list of authority objects
- "additionals": A list of additional objects

More complex DNS record types may log additional fields for resource data:

- "soa": Section containing fields for the SOA (start of authority) record type
  - "mname": Primary name server for this zone
  - "rname": Authority's mailbox
  - "serial": Serial version number
  - "refresh": Refresh interval (seconds)
  - "retry": Retry interval (seconds)
  - "expire": Upper time limit until zone is no longer authoritative (seconds)
  - "minimum": Minimum ttl for records in this zone (seconds)
- "sshfp": section containing fields for the SSHFP (ssh fingerprint) record type
  - "fingerprint": Hex format of the fingerprint (ex: `12:34:56:78:9a:bc:de:...`)
  - "algo": Algorithm number (ex: 1 for RSA, 2 for DSS)
  - "type": Fingerprint type (ex: 1 for SHA-1)
- "srv": section containing fields for the SRV (location of services) record type
  - "target": Domain name of the target host (ex: `foo.bar.baz`)
  - "priority": Target priority (ex: 20)
  - "weight": Weight for target selection (ex: 1)
  - "port": Port on this target host of this service (ex: 5060)

One can control which RR types are logged by using the "types" field in the suricata.yaml file. If this field is not specified, all RR types are logged. More than 50 values can be specified with this field as shown below:

Configuration:

```
- eve-log:
    enabled: yes
    type: file #file|syslog|unix_dgram|unix_stream
    filename: eve.json
    # the following are valid when type: syslog above
    #identity: "suricata"
    #facility: local5
    #level: Info ## possible levels: Emergency, Alert, Critical,
                 ## Error, Warning, Notice, Info, Debug
    types:
      - alert
      - dns:

        # Logging format. In 8.0 version 3 is the default. Can be
        # set to 2 to keep compatibility with Suricata 7.0.
        # version: 3

        # Control logging of requests and responses:
        # - requests: enable logging of DNS queries
        # - responses: enable logging of DNS answers
        # By default both requests and responses are logged.
        requests: yes
        responses: yes
```

(continues on next page)

(continued from previous page)

```
        # DNS record types to log, based on the query type.
        # Default: all.
        #types: [a, aaaa, cname, mx, ns, ptr, txt]
        types: [a, ns, md, mf, cname, soa, mb, mg, mr, null,
        wks, ptr, hinfo, minfo, mx, txt, rp, afsdb, x25, isdn,
        rt, nsap, nsapptr, sig, key, px, gpos, aaaa, loc, nxt,
        srv, atma, naptr, kx, cert, a6, dname, opt, apl, ds,
        sshfp, ipseckey, rrsig, nsec, dnskey, dhcid, nsec3,
        nsec3param, tlsa, hip, cds, cdnskey, spf, tkey,
        tsig, maila, any, uri]
```

Examples

Example of a DNS query for the IPv4 address of "twitter.com" (resource record type 'A'):

```
"dns": {
    "version": 3,
    "type": "request",
    "id": 16000,
    "queries": [
      {
        "rrname": "twitter.com",
        "rrtype": "A"
      }
    ]
}
```

Example of a DNS answer with "detailed" format:

```
"dns": {
    "version": 3,
    "type": "answer",
    "id": 45444,
    "flags": "8180",
    "qr": true,
    "rd": true,
    "ra": true,
    "rcode": "NOERROR",
    "queries": [
      {
        "rrname": "www.suricata.io",
        "rrtype": "A"
      }
    ],
    "answers": [
      {
        "rrname": "www.suricata.io",
        "rrtype": "CNAME",
        "ttl": 3324,
        "rdata": "suricata.io"
      },
      {
```

(continues on next page)

(continued from previous page)

```
        "rrname": "suricata.io",
        "rrtype": "A",
        "ttl": 10,
        "rdata": "192.0.78.24"
      },
      {
        "rrname": "suricata.io",
        "rrtype": "A",
        "ttl": 10,
        "rdata": "192.0.78.25"
      }
    ]
}
```

Example of a DNS answer with "grouped" format:

```
"dns": {
    "version": 3,
    "type": "answer",
    "id": 18523,
    "flags": "8180",
    "qr": true,
    "rd": true,
    "ra": true,
    "rcode": "NOERROR",
    "grouped": {
      "A": [
        "192.0.78.24",
        "192.0.78.25"
      ],
      "CNAME": [
        "suricata.io"
      ]
    }
}
```

Event type: LLMNR

LLMNR (Link-Local Multicast Name Resolution, RFC 4795) is a protocol that allows hosts on the same local link to perform name resolution. It uses DNS message format, but operates on multicast (224.0.0.252 for IPv4, ff02::1:3 for IPv6) over UDP and TCP port 5355.

Suricata logs both requests and responses as separate events. For UDP, queries are sent to multicast and responses come back as unicast from the responder's IP, resulting in separate flows.

Fields

Outline of fields seen in LLMNR events:

- "type": Indicating LLMNR message type, can be "request" or "response"
- "id": On-wire LLMNR transaction identifier
- "queries": A list of query objects, each containing "rrname" and "rrtype"

- "answers": A list of answer objects, each containing "rrname", "rrtype" with "rdata"
- "authorities": A list of authority objects
- "additionals": A list of additional objects

Examples

Example of an LLMNR request for the A record of "hs2011":

```
"llmnr": {
    "type": "request",
    "tx_id": 0,
    "id": 49985,
    "opcode": 0,
    "queries": [
      {
        "rrname": "hs2011",
        "rrtype": "a"
      }
    ]
}
```

Example of an LLMNR response with an A record answer:

```
"llmnr": {
    "type": "response",
    "tx_id": 0,
    "id": 49985,
    "opcode": 0,
    "queries": [
      {
        "rrname": "hs2011",
        "rrtype": "a"
      }
    ],
    "answers": [
      {
        "rrname": "HS2011",
        "a": "192.168.10.101"
      }
    ]
}
```

Example of an LLMNR NXDOMAIN response:

```
"llmnr": {
    "type": "response",
    "tx_id": 0,
    "id": 12345,
    "opcode": 0,
    "rcode": "NXDOMAIN",
    "queries": [
      {
        "rrname": "unknown-host",
```

(continues on next page)

(continued from previous page)

```
        "rrtype": "a"
      }
    ]
}
```

Example of an LLMNR response with NS authority and additional records:

```
"llmnr": {
    "type": "response",
    "tx_id": 0,
    "id": 54321,
    "opcode": 0,
    "queries": [
      {
        "rrname": "server.local",
        "rrtype": "a"
      }
    ],
    "answers": [
      {
        "rrname": "server.local",
        "a": "192.168.1.100"
      }
    ],
    "authorities": [
      {
        "rrname": "local",
        "ns": "ns.local"
      }
    ],
    "additionals": [
      {
        "rrname": "ns.local",
        "a": "192.168.1.200"
      }
    ]
}
```

Event type: FTP

Fields

- "command": The FTP command.
- "command_data": The data accompanying the command.
- "reply": The command reply, which may contain multiple lines, in array format.
- "completion_code": The 3-digit completion code. The first digit indicates whether the response is good, bad or incomplete. This is also in array format and may contain multiple completion codes matching multiple reply lines.
- "dynamic_port": The dynamic port established for subsequent data transfers, when applicable, with a "PORT" or "EPRT" command.

- "mode": The type of FTP connection. Most connections are "passive" but may be "active".
- "reply_received": Indicates whether a response was matched to the command. In some non-typical cases, a command may lack a response.

Examples

Example of regular FTP logging:

```
"ftp": {
  "command": "RETR",
  "command_data": "100KB.zip",
  "reply": [
    "Opening BINARY mode data connection for 100KB.zip (102400 bytes).",
    "Transfer complete."
  ],
  "completion_code": [
    "150",
    "226"
  ],
```

Example showing all fields:

```
"ftp": {
  "command": "EPRT",
  "command_data": "|2|2a01:e34:ee97:b130:8c3e:45ea:5ac6:e301|41813|",
  "reply": [
    "EPRT command successful. Consider using EPSV."
  ],
  "completion_code": [
    "200"
  ],
  "dynamic_port": 41813,
  "mode": "active",
  "reply_received": "yes"
}
```

Event type: FTP_DATA

Fields

- "command": The FTP command associated with the event.
- "filename": The name of the involved file.

Examples

Example of FTP_DATA logging:

```
"ftp_data": {
  "filename": "temp.txt",
  "command": "RETR"
}
```

Event type: TLS

Fields

- "subject": The subject field from the TLS certificate
- "issuer": The issuer field from the TLS certificate
- "session_resumed": This field has the value of "true" if the TLS session was resumed via a session id. If this field appears, "subject" and "issuer" do not appear, since a TLS certificate is not seen.

If extended logging is enabled the following fields are also included:

- "serial": The serial number of the TLS certificate
- "fingerprint": The (SHA1) fingerprint of the TLS certificate
- "sni": The Server Name Indication (SNI) extension sent by the client
- "version": The SSL/TLS version used
- "notbefore": The NotBefore field from the TLS certificate
- "notafter": The NotAfter field from the TLS certificate
- "ja3": The JA3 fingerprint consisting of both a JA3 hash and a JA3 string
- "ja3s": The JA3S fingerprint consisting of both a JA3 hash and a JA3 string
- "ja4": The JA4 client fingerprint for TLS
- "client_alpns": array of strings with ALPN values
- "server_alpns": array of strings with ALPN values

JA3 and JA4 must be enabled in the Suricata config file (set 'app-layer.protocols.tls.ja3-fingerprints'/'app-layer.protocols.tls.ja4-fingerprints' to 'yes').

In addition to this, custom logging also allows the following fields:

- "certificate": The TLS certificate base64 encoded
- "chain": The entire TLS certificate chain base64 encoded
- "client_handshake": structure containing "version", "ciphers" ([u16]), "exts" ([u16]), "sig_algs" ([u16]), for client hello supported cipher suites, extensions, and signature algorithms, respectively, in the order that they're mentioned (ie. unsorted)
- "server_handshake": structure containing "version", "chosen cipher", "exts" ([u16]), for server hello in the order that they're mentioned (ie. unsorted)

Examples

Example of regular TLS logging:

```
"tls": {
    "subject": "C=US, ST=California, L=Mountain View, O=Google Inc, CN=*.google.com",
    "issuerdn": "C=US, O=Google Inc, CN=Google Internet Authority G2"
}
```

Example of regular TLS logging for resumed sessions:

```
"tls": {
    "session_resumed": true
}
```

Example of extended TLS logging:

```
"tls": {
    "subject": "C=US, ST=California, L=Mountain View, O=Google Inc, CN=*.google.com",
    "issuerdn": "C=US, O=Google Inc, CN=Google Internet Authority G2",
    "serial": "0C:00:99:B7:D7:54:C9:F6:77:26:31:7E:BA:EA:7C:1C",
    "fingerprint": "8f:51:12:06:a0:cc:4e:cd:e8:a3:8b:38:f8:87:59:e5:af:95:ca:cd",
    "sni": "calendar.google.com",
    "version": "TLS 1.2",
    "notbefore": "2017-01-04T10:48:43",
    "notafter": "2017-03-29T10:18:00"
}
```

Example of certificate logging using TLS custom logging (subject, sni, certificate):

```
"tls": {
    "subject": "C=US, ST=California, L=Mountain View, O=Google Inc, CN=*.googleapis.com
    "sni": "www.googleapis.com",
    "certificate": "MIIE3TCCA8WgAwIBAgIIQPsvobRZN0gwDQYJKoZIhvcNAQELBQAwSTELMA [...]"
 }
```

Event type: TFTP

Fields

- "packet": The operation code, can be "read" or "write" or "error"
- "file": The filename transported with the tftp protocol
- "mode": The mode field, can be "octet" or "mail" or "netascii" (or any combination of upper and lower case)

Example of TFTP logging:

```
"tftp": {
    "packet": "write",
    "file": "rfc1350.txt",
    "mode": "octet"
 }
```

Event type: KRB5

KRB5 Fields

- "cname" (string): The client PrincipalName
- "encryption" (string): Encryption used (only in AS-REP and TGS-REP)
- "error_code" (string): Error code, if request has failed
- "failed_request" (string): The request type for which the response had an error_code
- "msg_type" (string): The message type: AS-REQ, AS-REP, etc...
- "realm" (string): The server Realm
- "sname" (string): The server PrincipalName
- "ticket_encryption" (string): Encryption used for ticket
- "ticket_weak_encryption" (boolean): Whether the encryption used for ticket is a weak cipher

- "weak_encryption" (boolean): Whether the encryption used in AS-REP or TGS-REP is a weak cipher

Examples of KRB5 logging:

Pipe open:

```
"krb5": {
  "msg_type": "KRB_TGS_REP",
  "cname": "robin",
  "realm": "CYLERA.LAB",
  "sname": "ldap/dc01",
  "encryption": "aes256-cts-hmac-sha1-96",
  "weak_encryption": false,
  "ticket_encryption": "aes256-cts-hmac-sha1-96",
  "ticket_weak_encryption": false
}
```

Event type: SMB

SMB Fields

- "id" (integer): internal transaction id
- "dialect" (string): the negotiated protocol dialect, or "unknown" if missing
- "command" (string): command name. E.g. SMB2_COMMAND_CREATE or SMB1_COMMAND_WRITE_ANDX
- "status" (string): status string. Can be both NT_STATUS or DOS_ERR and other variants
- "status_code" (string): status code as hex string
- "session_id" (integer): SMB2+ session_id. SMB1 user id.
- "tree_id" (integer): Tree ID
- "filename" (string): filename for CREATE and other commands.
- "disposition" (string): requested disposition. E.g. FILE_OPEN, FILE_CREATE and FILE_OVERWRITE. See https://msdn.microsoft.com/en-us/library/ee442175.aspx#Appendix_A_Target_119
- "access" (string): indication of how the file was opened. "normal" or "delete on close" (field is subject to change)
- "created", "accessed", "modified", "changed" (integer): timestamps in seconds since unix epoch
- "size" (integer): size of the requested file
- "fuid" (string): SMB2+ file GUID. SMB1 FID as hex.
- "share" (string): share name.
- "share_type" (string): FILE, PIPE, PRINT or unknown.
- "client_dialects" (array of strings): list of SMB dialects the client speaks.
- "client_guid" (string): client GUID
- "server_guid" (string): server GUID
- "request.native_os" (string): SMB1 native OS string
- "request.native_lm" (string): SMB1 native Lan Manager string
- "response.native_os" (string): SMB1 native OS string
- "response.native_lm" (string): SMB1 native Lan Manager string

One can restrict which transactions are logged by using the "types" field in the suricata.yaml file. If this field is not specified, all transactions types are logged. 9 values can be specified with this field as shown below:

Configuration:

```
- eve-log:
    enabled: yes
    type: file
    filename: eve.json
    types:
      - smb:
          types: [file, tree_connect, negotiate, dcerpc, create,
            session_setup, ioctl, rename, set_file_path_info, generic]
```

Examples of SMB logging:

Pipe open:

```
"smb": {
  "id": 1,
  "dialect": "unknown",
  "command": "SMB2_COMMAND_CREATE",
  "status": "STATUS_SUCCESS",
  "status_code": "0x0",
  "session_id": 4398046511201,
  "tree_id": 1,
  "filename": "atsvc",
  "disposition": "FILE_OPEN",
  "access": "normal",
  "created": 0,
  "accessed": 0,
  "modified": 0,
  "changed": 0,
  "size": 0,
  "fuid": "0000004d-0000-0000-0005-0000ffffffff"
}
```

File/pipe close:

```
"smb": {
  "id": 15,
  "dialect": "2.10",
  "command": "SMB2_COMMAND_CLOSE",
  "status": "STATUS_SUCCESS",
  "status_code": "0x0",
  "session_id": 4398046511121,
  "tree_id": 1,
}
```

Tree connect (share open):

```
"smb": {
  "id": 3,
  "dialect": "2.10",
  "command": "SMB2_COMMAND_TREE_CONNECT",
```

(continues on next page)

(continued from previous page)


```
  "status": "STATUS_SUCCESS",
  "status_code": "0x0",
  "session_id": 4398046511121,
  "tree_id": 1,
  "share": "\\\\admin-pc\\c$",
  "share_type": "FILE"
}
```


Dialect negotiation from SMB1 to SMB2 dialect 2.10:


```
"smb": {
  "id": 1,
  "dialect": "2.??",
  "command": "SMB1_COMMAND_NEGOTIATE_PROTOCOL",
  "status": "STATUS_SUCCESS",
  "status_code": "0x0",
  "session_id": 0,
  "tree_id": 0,
  "client_dialects": [
    "PC NETWORK PROGRAM 1.0",
    "LANMAN1.0",
    "Windows for Workgroups 3.1a",
    "LM1.2X002",
    "LANMAN2.1",
    "NT LM 0.12",
    "SMB 2.002",
    "SMB 2.???"
  ],
  "server_guid": "aec6e793-2b11-4019-2d95-55453a0ad2f1"
}
"smb": {
  "id": 2,
  "dialect": "2.10",
  "command": "SMB2_COMMAND_NEGOTIATE_PROTOCOL",
  "status": "STATUS_SUCCESS",
  "status_code": "0x0",
  "session_id": 0,
  "tree_id": 0,
  "client_dialects": [
    "2.02",
    "2.10"
  ],
  "client_guid": "601985d2-aad9-11e7-8494-00088bb57f27",
  "server_guid": "aec6e793-2b11-4019-2d95-55453a0ad2f1"
}
```


SMB1 partial SMB1_COMMAND_SESSION_SETUP_ANDX:


```
"request": {
  "native_os": "Unix",
  "native_lm": "Samba 3.9.0-SVN-build-11572"
},
```


(continues on next page)

(continued from previous page)

```
"response": {
  "native_os": "Windows (TM) Code Name \"Longhorn\" Ultimate 5231",
  "native_lm": "Windows (TM) Code Name \"Longhorn\" Ultimate 6.0"
}
```

DCERPC fields

- "request" (string): command. E.g. REQUEST, BIND.
- "response" (string): reply. E.g. RESPONSE, BINDACK or FAULT.
- "opnum" (integer): the opnum
- "call_id" (integer): the call id
- "frag_cnt" (integer): the number of fragments for the stub data
- "stub_data_size": total stub data size
- "interfaces" (array): list of interfaces
- "interfaces.uuid" (string): string representation of the UUID
- "interfaces.version" (string): interface version
- "interfaces.ack_result" (integer): ack result
- "interfaces.ack_reason" (integer): ack reason
- "interfaces.service" (string): service name for the UUID
- "interfaces.procedure" (string): procedure name for the UUID and opnum

DCERPC REQUEST/RESPONSE:

```
"smb": {
  "id": 4,
  "dialect": "unknown",
  "command": "SMB2_COMMAND_IOCTL",
  "status": "STATUS_SUCCESS",
  "status_code": "0x0",
  "session_id": 4398046511201,
  "tree_id": 0,
  "dcerpc": {
    "request": "REQUEST",
    "response": "RESPONSE",
    "opnum": 0,
    "req": {
      "frag_cnt": 1,
      "stub_data_size": 136
    },
    "res": {
      "frag_cnt": 1,
      "stub_data_size": 8
    },
    "call_id": 2
  }
}
```

DCERPC BIND/BINDACK:


```
"smb": {
  "id": 53,
  "dialect": "2.10",
  "command": "SMB2_COMMAND_WRITE",
  "status": "STATUS_SUCCESS",
  "status_code": "0x0",
  "session_id": 35184439197745,
  "tree_id": 1,
  "dcerpc": {
    "request": "BIND",
    "response": "BINDACK",
    "interfaces": [
      {
        "uuid": "12345778-1234-abcd-ef00-0123456789ac",
        "service": "samr",
        "version": "1.0",
        "ack_result": 2,
        "ack_reason": 0
      },
      {
        "uuid": "12345778-1234-abcd-ef00-0123456789ac",
        "service": "samr",
        "version": "1.0",
        "ack_result": 0,
        "ack_reason": 0
      },
      {
        "uuid": "12345778-1234-abcd-ef00-0123456789ac",
        "service": "samr",
        "version": "1.0",
        "ack_result": 3,
        "ack_reason": 0
      }
    ],
    "call_id": 2
  }
```


NTLMSSP fields

- "domain" (string): the Windows domain.
- "user" (string): the user.
- "host" (string): the host.
- "version" (string): the client version.

Example:


```
"ntlmssp": {
  "domain": "VNET3",
  "user": "administrator",
  "host": "BLU",
```


(continues on next page)

(continued from previous page)

```
  "version": "60.230 build 13699 rev 188"
}
```

More complete example:

```
"smb": {
  "id": 3,
  "dialect": "NT LM 0.12",
  "command": "SMB1_COMMAND_SESSION_SETUP_ANDX",
  "status": "STATUS_SUCCESS",
  "status_code": "0x0",
  "session_id": 2048,
  "tree_id": 0,
  "ntlmssp": {
    "domain": "VNET3",
    "user": "administrator",
    "host": "BLU",
    "version": "60.230 build 13699 rev 188"
  },
  "request": {
    "native_os": "Unix",
    "native_lm": "Samba 3.9.0-SVN-build-11572"
  },
  "response": {
    "native_os": "Windows (TM) Code Name \"Longhorn\" Ultimate 5231",
    "native_lm": "Windows (TM) Code Name \"Longhorn\" Ultimate 6.0"
  }
}
```

Kerberos fields

- "kerberos.realm" (string): the Kerberos Realm.
- "kerberos.snames (array of strings): snames.

Example:

```
"smb": {
  "dialect": "2.10",
  "command": "SMB2_COMMAND_SESSION_SETUP",
  "status": "STATUS_SUCCESS",
  "status_code": "0x0",
  "session_id": 35184439197745,
  "tree_id": 0,
  "kerberos": {
    "realm": "CONTOSO.LOCAL",
    "snames": [
      "cifs",
      "DC1.contoso.local"
    ]
  }
}
```

Event type: BITTORRENT-DHT

Common fields:

- "transaction_id" (hex): the unique id of the transaction, generated by node making the request (a.k.a the querying node). Same transaction_id is echoed back by responding nodes.
- "client_version" (hex): identifies the type and version of the bittorrent-dht client. Some implementations may be missing this field.

Extra fields:

Packets should also contain one of either the fields:

error

- **"error": details of an error which occurred while processing the request**
  - "error.num" (num): the error code
  - "error.msg" (string): the error message

request_type and request

- "request_type" (string): the type of the request (a.k.a. the query). Included if this packet was a request
- **"request": a request (a.k.a. a query) sent by the bittorrent-dht client**
  - "request.id" (hex): the node ID of the node which sent the request (20 bytes in network byte order)
  - "request.target" (hex): the target node ID. Used by the find_node request_type
  - "request.info_hash" (hex): info hash of target torrent (20 bytes). Used by the get_peers and announce_peer request_types
  - "request.token" (hex): token key received from previous get_peers request. Used by the announce_peer request type
  - "request.implied_port" (num): 0 or 1, if 1 ignore provided port and use source port of UDP packet. Used by the announce_peer request_type
  - "request.port" (num): port on which peer will download torrent. Used by the announce_peer request_type

response

- **"response": a response to the client's request**
  - "response.id" (hex): the node ID of the node which sent the response (20 bytes in network byte order)
  - "response.nodes" (array): find_node/get_peers - a list of info objects for target node or K(8) closest good nodes in routing table
  - "response.nodes6" (array): find_node/get_peers - a list of info objects for target node or K(8) closest good nodes in routing table (ipv6)
  - "response.values" (array): list of compact peer info strings. Used by the get_peers request_type

- "response.token" (hex): token key required for sender's future announce_peer query

node object

- "id" (hex): node ID
- "ip" (string): IPv4 or IPv6 address of node
- "port" (integer): node port

peer object (values array)

- "ip" (string): IPv6 or IPv6 address of node
- "port" (integer): node port

Examples:

Ping and response:

```
"bittorrent_dht": {
  "transaction_id": "0c17",
  "client_version": "4c540126",
  "request_type": "ping",
  "request": {
    "id": "41aff1580119f074e2f537f231f12adf684f0d1f"
  }
}

"bittorrent_dht": {
  "transaction_id": "0c17",
  "client_version": "5554b50c",
  "response": {
    "id": "42aeb304a0845b3b9ee089327b48967b8e87b2e2"
  }
}
```

Find_node and response:

```
"bittorrent_dht": {
  "transaction_id": "420f0000",
  "client_version": "5554b50c",
  "request_type": "find_node",
  "request": {
    "id": "37579bad1bad166af4329508096fae8c553c6cf4",
    "target": "37579bad1bad166af4329508096fae8c553c6cf4"
  }
}
```

Get_peers and response with values param:

```
"bittorrent_dht": {
  "transaction_id": "05e4",
  "client_version": "4c540126",
  "request_type": "get_peers",
  "request": {
    "id": "41aff1580119f074e2f537f231f12adf684f0d1f",
    "info_hash": "19a6fcfcba6cc2c6d371eb754074d095adb5d291"
  }
}
"bittorrent_dht": {
  "transaction_id": "05e4",
  "client_version": "555462d6",
  "response": {
    "id": "19a6f98be177e32e7b5bd77276d529f03e3ba8a9",
    "values": [
      {
        "ip": "45.238.190.2",
        "port": 6881
      },
      {
        "ip": "185.70.52.245",
        "port": 51215
      },
      {
        "ip": "45.21.238.247",
        "port": 55909
      },
      {
        "ip": "62.28.248.195",
        "port": 6881
      }
    ],
    "token": "c17094641ca8844d711120baecb2b5cf25435614"
  }
}
```


Get_peers and response with nodes param:


```
 "bittorrent_dht": {
  "transaction_id": "44e6",
  "client_version": "4c540126",
  "request_type": "get_peers",
  "request": {
    "id": "41aff1580119f074e2f537f231f12adf684f0d1f",
    "info_hash": "19a6fcfcba6cc2c6d371eb754074d095adb5d291"
  }
}

"bittorrent_dht": {
  "transaction_id": "44e6",
  "response": {
    "id": "19a7c8f4f6d14d9f87a67671720633e551f30cb7",
    "values": [
```


(continues on next page)

(continued from previous page)

```
      {
        "ip": "45.22.252.153",
        "port": 36798
      },
      {
        "ip": "94.41.206.37",
        "port": 30850
      },
      {
        "ip": "84.228.120.50",
        "port": 6881
      },
      {
        "ip": "178.81.206.84",
        "port": 12373
      },
      {
        "ip": "110.188.93.186",
        "port": 22223
      }
    ],
    "token": "c897ee539e02a54595b4d7cfb6319ad48e71b282"
  }
}
```

Announce_peer and response:

```
"bittorrent_dht": {
  "transaction_id": "aa",
  "request_type": "announce_peer",
  "request": {
    "id": "abcdefghij0123456789",
    "info_hash": "mnopqrstuvwxyz123456",
    "token": "aoeusnth",
    "port": 6881
  }
}
"bittorrent_dht": {
  "transaction_id": "aa",
  "response": {
    "id": "mnopqrstuvwxyz123456"
  }
}
```

Announce_peer with implied_port param and response:

```
"bittorrent_dht": {
  "transaction_id": "7fe9",
  "client_version": "4c540126",
  "request_type": "announce_peer",
  "request": {
    "id": "51bc83f53417a62a40e8a48170cad369a13fef3c",
```

(continues on next page)

(continued from previous page)

```
    "info_hash": "19a6fcfcba6cc2c6d371eb754074d095adb5d291",
    "token": "cacbef35",
    "implied_port": 1,
    "port": 54892
  }
}

"bittorrent_dht": {
  "transaction_id": "7fe9",
  "client_version": "4c54012f",
  "response": {
    "id": "19a66dece45e0288ab75d141e0255738a1ce8508"
  }
}
```

Sample error responses:

```
"bittorrent_dht": {
  "transaction_id": "aa",
  "error": {
    "num": 201,
    "msg": "A Generic Error Ocurred"
  }
}
"bittorrent_dht": {
  "transaction_id": "aa",
  "error": {
    "num": 203,
    "msg": "Malformed Packet"
  }
}
```

Event type: SSH

Fields

- "proto_version": The protocol version transported with the ssh protocol (1.x, 2.x)
- "software_version": The software version used by end user
- "hassh.hash": MD5 of hassh algorithms of client or server
- "hassh.string": hassh algorithms of client or server
- "error": (per direction) unrecoverable parse failure: "invalid_banner" or "invalid_record". Present only when the direction failed; a failed direction is logged even when it parsed no banner, so the error field may be the only content of the direction object.
- "state": (per direction) the state the failure occurred in (banner, kex or session). Present only alongside the error field.

Hassh must be enabled in the Suricata config file (set 'app-layer.protocols.ssh.hassh' to 'yes').

Example of SSH logging:

```
"ssh": {
  "client": {
      "proto_version": "2.0",
      "software_version": "OpenSSH_6.7",
      "hassh": {
          "hash": "ec7378c1a92f5a8dde7e8b7a1ddf33d1",
          "string": "curve25519-sha256,diffie-hellman-group14-sha256,diffie-hellman-
↪group14-sha1,ext-info-c",
      }
   },
  "server": {
      "proto_version": "2.0",
      "software_version": "OpenSSH_6.7",
      "hassh": {
          "hash": "ec7378c1a92f5a8dde7e8b7a1ddf33d1",
          "string": "curve25519-sha256,curve25519-sha256@libssh.org,ecdh-sha2-nistp256",
      }
   }
}
```

Event type: Flow

Fields

- "pkts_toserver": total number of packets to server, include bypassed packets
- "pkts_toclient": total number of packets to client
- "bytes_toserver": total bytes count to server
- "bytes_toclient": total bytes count to client
- "bypassed.pkts_toserver": number of bypassed packets to server
- "bypassed.pkts_toclient": number of bypassed packets to client
- "bypassed.bytes_toserver": bypassed bytes count to server
- "bypassed.bytes_toclient": bypassed bytes count to client
- "start": date of start of the flow
- "end": date of end of flow (last seen packet)
- "age": duration of the flow
- "bypass": if the flow has been bypassed, it is set to "local" (internal bypass) or "capture"
- "state": display state of the flow (include "new", "established", "closed", "bypassed")
- "reason": mechanism that did trigger the end of the flow (include "timeout", "forced" and "shutdown")
- "alerted": "true" or "false" depending if an alert has been seen on flow
- "action": "pass" or "drop" depending if flow was PASS'ed or DROP'ed (no present if none)
- "tx_cnt": number of transactions seen in the flow (only present if flow has an application layer)
- "exception_policy": array consisting of exception policies that have been triggered by the flow:
  - "target": if an exception policy was triggered, what setting exceptions led to this (cf. *Exception Policy - Specific Settings*).

- "policy": if an exception policy was triggered, what policy was applied (to the flow or to any packet(s) from it).

Example


```
"flow": {
  "pkts_toserver": 23,
  "pkts_toclient": 21,
  "bytes_toserver": 4884,
  "bytes_toclient": 7392,
  "bypassed": {
    "pkts_toserver": 10,
    "pkts_toclient": 8,
    "bytes_toserver": 1305,
    "bytes_toclient": 984
  },
  "start": "2019-05-28T23:32:29.025256+0200",
  "end": "2019-05-28T23:35:28.071281+0200",
  "age": 179,
  "bypass": "capture",
  "state": "bypassed",
  "reason": "timeout",
  "alerted": false,
  "action": "pass",
  "exception_policy": [
    {
      "target": "stream_midstream",
      "policy": "pass_flow"
    }
  ]
}
```


Event type: RDP

Initial negotiations between RDP client and server are stored as transactions and logged.

Each RDP record contains a per-flow incrementing "tx_id" field.

The "event_type" field indicates an RDP event subtype. Possible values:

- "initial_request"
- "initial_response"
- "connect_request"
- "connect_response"
- "tls_handshake"

RDP type: Initial Request

The optional "cookie" field is a string identifier the RDP client has chosen to provide.

The optional "flags" field is a list of client directives. Possible values:

- "restricted_admin_mode_required"
- "redirected_authentication_mode_required"

- "correlation_info_present"

RDP type: Initial Response

In the event of a standard initial response:

The "protocol" field is the selected protocol. Possible values:

- "rdp"
- "ssl"
- "hybrid"
- "rds_tls"
- "hybrid_ex"

The optional "flags" field is a list of support server modes. Possible values:

- "extended_client_data"
- "dynvc_gfx"
- "restricted_admin"
- "redirected_authentication"

Alternatively, in the event of an error-indicating initial response:

There will be no "protocol" or "flags" fields.

The "error_code" field will contain the numeric code provided by the RDP server.

The "reason" field will contain a text summary of this code. Possible values:

- "ssl required by server" (error code 0x1)
- "ssl not allowed by server" (error code 0x2)
- "ssl cert not on server" (error code 0x3)
- "inconsistent flags" (error code 0x4)
- "hybrid required by server" (error code 0x5)
- "ssl with user auth required by server" (error code 0x6)

RDP type: Connect Request

The optional "channel" field is a list of requested data channel names.

Common channels:

- "rdpdr" (device redirection)
- "cliprdr" (shared clipboard)
- "rdpsnd" (sound)

The optional "client" field is a sub-object that may contain the following:

- "version": RDP protocol version. Possible values are "v4", "v5", "v10.0", "v10.1", "v10.2", "v10.3", "v10.4", "v10.5", "v10.6", "v10.7", "unknown".
- "desktop_width": Numeric desktop width value.
- "desktop_height": Numeric desktop height value.

- "color_depth": Numeric color depth. Possible values are 4, 8, 15, 16, 24.
- "keyboard_layout": Locale identifier name, e.g., "en-US".
- "build": OS and SP level, e.g., "Windows XP", "Windows 7 SP1".
- "client_name": Client computer name.
- "keyboard_type": Possible values are "xt", "ico", "at", "enhanced", "1050", "9140", "jp".
- "keyboard_subtype": Numeric code for keyboard.
- "function_keys": Number of function keys on client keyboard.
- "ime": Input method editor (IME) file name.
- "product_id": Product id string.
- "serial_number": Numeric value.
- "capabilities": List of any of the following: "support_errinfo_pdf", "want_32bpp_session", "support_statusinfo_pdu", "strong_asymmetric_keys", "valid_connection_type", "support_monitor_layout_pdu", "support_netchar_autodetect", "support_dynvc_gfx_protocol", "support_dynamic_time_zone", "support_heartbeat_pdu".
- "id": Client product id string.
- "connection_hint": Possible values are "modem", "low_broadband", "satellite", "high_broadband", "wan", "lan", "autodetect".
- "physical_width": Numeric physical width of display.
- "physical_height": Numeric physical height of display.
- "desktop_orientation": Numeric angle of orientation.
- "scale_factor": Numeric scale factor of desktop.
- "device_scale_factor": Numeric scale factor of display.

RDP type: Connect Response

With this event, the initial RDP negotiation is complete in terms of tracking and logging.

RDP type: TLS Handshake

With this event, the initial RDP negotiation is complete in terms of tracking and logging.

The session will use TLS encryption.

The "x509_serials" field is a list of observed certificate serial numbers, e.g., "16ed2aa0495f259d4f5d99edada570d1".

Examples

RDP logging:

```
"rdp": {
  "tx_id": 0,
  "event_type": "initial_request",
  "cookie": "A70067"
}

"rdp": {
```

(continues on next page)

(continued from previous page)


```
  "tx_id": 1,
  "event_type": "initial_response"
}

"rdp": {
  "tx_id": 2,
  "event_type": "connect_request",
  "client": {
    "version": "v5",
    "desktop_width": 1152,
    "desktop_height": 864,
    "color_depth": 15,
    "keyboard_layout": "en-US",
    "build": "Windows XP",
    "client_name": "ISD2-KM84178",
    "keyboard_type": "enhanced",
    "function_keys": 12,
    "product_id": 1,
    "capabilities": [
      "support_errinfo_pdf"
    ],
    "id": "55274-OEM-0011903-00107"
  },
  "channels": [
    "rdpdr",
    "cliprdr",
    "rdpsnd"
  ]
}

"rdp": {
  "tx_id": 3,
  "event_type": "connect_response"
}
```


RDP logging, with transition to TLS:


```
"rdp": {
  "tx_id": 0,
  "event_type": "initial_request",
  "cookie": "AWAKECODI"
}

"rdp": {
  "tx_id": 1,
  "event_type": "initial_response",
  "server_supports": [
    "extended_client_data"
  ],
  "protocol": "hybrid"
}
```


(continues on next page)

(continued from previous page)

```
"rdp": {
  "tx_id": 2,
  "event_type": "tls_handshake",
  "x509_serials": [
    "16ed2aa0495f259d4f5d99edada570d1"
  ]
}
```

Event type: RFB

Fields

- "server_protocol_version.major", "server_protocol_version.minor": The RFB protocol version offered by the server.
- "client_protocol_version.major", "client_protocol_version.minor": The RFB protocol version agreed by the client.
- "authentication.security_type": Security type agreed upon in the logged transaction, e.g. 2 is VNC auth.
- "authentication.vnc.challenge", "authentication.vnc.response": Only available when security type 2 is used. Contains the challenge and response byte buffers exchanged by the server and client as hex strings.
- "authentication.security_result": Result of the authentication process (`OK`, `FAIL` or `TOOMANY`).
- "screen_shared": Boolean value describing whether the client requested screen sharing.
- "framebuffer": Contains metadata about the initial screen setup process. Only available when the handshake completed this far.
- "framebuffer.width", "framebuffer.height": Screen size as offered by the server.
- "framebuffer.name": Desktop name as advertised by the server.
- "framebuffer.pixel_format": Pixel representation information, such as color depth. See RFC6143 (https://tools.ietf.org/html/rfc6143) for details.

Examples

Example of RFB logging, with full VNC style authentication parameters:

```
"rfb": {
  "server_protocol_version": {
    "major": "003",
    "minor": "007"
  },
  "client_protocol_version": {
    "major": "003",
    "minor": "007"
  },
  "authentication": {
    "security_type": 2,
    "vnc": {
      "challenge": "0805b790b58e967f2b350a0c99de3881",
      "response": "aecb26faeaaa62179636a5934bac1078"
    },
    "security_result": "OK"
```

(continues on next page)

(continued from previous page)


```
  },
  "screen_shared": false,
  "framebuffer": {
    "width": 1280,
    "height": 800,
    "name": "foobar@localhost.localdomain",
    "pixel_format": {
      "bits_per_pixel": 32,
      "depth": 24,
      "big_endian": false,
      "true_color": true,
      "red_max": 255,
      "green_max": 255,
      "blue_max": 255,
      "red_shift": 16,
      "green_shift": 8,
      "blue_shift": 0
    }
  }
```


Event type: MQTT

EVE-JSON output for MQTT consists of one object per MQTT transaction, with some common and various type-specific fields.

Transactions

A single MQTT communication can consist of multiple messages that need to be exchanged between broker and client. For example, some actions at higher QoS levels (> 0) usually involve a combination of requests and acknowledgement messages that are linked by a common identifier:

- `CONNECT` followed by `CONNACK`
- `PUBLISH` followed by `PUBACK` (QoS 1) or `PUBREC/PUBREL/PUBCOMP` (QoS 2)
- `SUBSCRIBE` followed by `SUBACK`
- `UNSUBSCRIBE` followed by `UNSUBACK`

The MQTT parser merges individual messages into one EVE output item if they belong to one transaction. In such cases, the source and destination information (IP/port) reflect the direction of the initial request, but contain messages from both sides.

Example for a PUBLISH at QoS 2:


```
{
  "timestamp": "2020-05-19T18:00:39.016985+0200",
  "flow_id": 1454127794305760,
  "pcap_cnt": 65,
  "event_type": "mqtt",
  "src_ip": "0000:0000:0000:0000:0000:0000:0000:0001",
  "src_port": 60105,
  "dest_ip": "0000:0000:0000:0000:0000:0000:0000:0001",
  "dest_port": 1883,
  "proto": "TCP",
```


(continues on next page)

(continued from previous page)


```
  "mqtt": {
    "publish": {
      "qos": 2,
      "retain": false,
      "dup": false,
      "topic": "house/bulbs/bulb1",
      "message_id": 3,
      "message": "OFF"
    },
    "pubrec": {
      "qos": 0,
      "retain": false,
      "dup": false,
      "message_id": 3
    },
    "pubrel": {
      "qos": 1,
      "retain": false,
      "dup": false,
      "message_id": 3
    },
    "pubcomp": {
      "qos": 0,
      "retain": false,
      "dup": false,
      "message_id": 3
    }
  }
}
```


Note that some message types (aka control packet types), such as `PINGREQ` and `PINGRESP`, have no type-specific data, nor do they have information that facilitate grouping into transactions. These will be logged as single items and only contain the common fields listed below.

Common fields

Common fields from the MQTT fixed header:

- "*.qos": Quality of service level for the message, integer between 0 and 2.
- "*.retain": Boolean value of the MQTT 'retain' flag.
- "*.dup": Boolean value of the MQTT 'dup' (duplicate) flag.

MQTT CONNECT fields

- "connect.protocol_string": Protocol string as defined in the spec, e.g. `MQTT` (MQTT 3.1.1 and later) or `MQIsdp` (MQTT 3.1).
- "connect.protocol_version": Protocol version as defined in the specification:
  - protocol version 3: MQTT 3.1
  - protocol version 4: MQTT 3.1.1
  - protocol version 5: MQTT 5.0

- "connect.flags.username", "connect.flags.password": Set to *true* if credentials are submitted with the connect request.
- "connect.flags.will": Set to *true* if a will is set.
- "connect.flags.will_retain": Set to *true* if the will is to be retained on the broker.
- "connect.will.clean_session": Set to *true* if the connection is to made with a clean session.
- "connect.client_id": Client ID string submitted my the connecting client.
- "connect.username", "connect.password": User/password authentication credentials submitted with the connect request. Passwords are only logged when the corresponding configuration setting is enabled (`mqtt.passwords: yes`).
- "connect.will.topic": Topic to publish the will message to.
- "connect.will.message": Message to be published on connection loss.
- "connect.will.properties": (Optional, MQTT 5.0) Will properties set on this request. See 3.1.3.2 in the spec for more information on will properties.
- "connect.properties": (Optional, MQTT 5.0) CONNECT properties set on this request. See 3.1.2.11 in the spec for more information on CONNECT properties.

Example of MQTT CONNECT logging:

```
"connect": {
  "qos": 0,
  "retain": false,
  "dup": false,
  "protocol_string": "MQTT",
  "protocol_version": 5,
  "flags": {
    "username": true,
    "password": true,
    "will_retain": false,
    "will": true,
    "clean_session": true
  },
  "client_id": "client",
  "username": "user",
  "password": "pass",
  "will": {
    "topic": "willtopic",
    "message": "willmessage",
    "properties": {
      "content_type": "mywilltype",
      "correlation_data": "3c32aa4313b3e",
      "message_expiry_interval": 133,
      "payload_format_indicator": 144,
      "response_topic": "response_topic1",
      "user_properties": [{"key": "userprop", "value": "uservalue"}],
      "will_delay_interval": 200
    }
  },
  "properties": {
    "maximum_packet_size": 11111,
```

(continues on next page)

(continued from previous page)

```
    "receive_maximum": 222,
    "session_expiry_interval": 555,
    "topic_alias_maximum": 666,
    "user_properties": [{"key": "userprop1", "value": "userval1"}, {"key": "userprop2",
↪"value": "userval2"}]
  }
}
```

MQTT CONNACK fields

- "connack.session_present": Set to *true* if a session is continued on connection.
- "connack.return_code": Return code/reason code for this reply. See 3.2.2.2 in the spec for more information on these codes.
- "connect.properties": (Optional, MQTT 5.0) CONNACK properties set on this request. See 3.2.2.3 in the spec for more information on CONNACK properties.

Example of MQTT CONNACK logging:

```
"connack": {
  "qos": 0,
  "retain": false,
  "dup": false,
  "session_present": false,
  "return_code": 0,
  "properties": {
    "topic_alias_maximum": 10
  }
}
```

MQTT PUBLISH fields

- "publish.topic": Topic this message is published to.
- "publish.message_id": (Only present if QOS level > 0) Message ID for this publication.
- "publish.message": Message to be published.
- "publish.properties": (Optional, MQTT 5.0) PUBLISH properties set on this request. See 3.3.2.3 in the spec for more information on PUBLISH properties.

Example of MQTT PUBLISH logging:

```
"publish": {
  "qos": 1,
  "retain": false,
  "dup": false,
  "topic": "topic",
  "message_id": 1,
  "message": "baa baa sheep",
  "properties": {
    "content_type": "mytype",
    "correlation_data": "3c32aa4313b3e",
    "message_expiry_interval": 77,
```

(continues on next page)

(continued from previous page)

```
    "payload_format_indicator": 88,
    "response_topic": "response_topic1",
    "topic_alias": 5,
    "user_properties": [{"key": "userprop", "value":"userval"}]
  }
}
```

MQTT PUBACK/PUBREL/PUBREC/PUBCOMP fields

- "[puback|pubrel|pubrec|pubcomp].message_id": Original message ID this message refers to.
- "[puback|pubrel|pubrec|pubcomp].reason_code": Return code/reason code for this reply. See the spec for more information on these codes.
- "[puback|pubrel|pubrec|pubcomp].properties": (Optional, MQTT 5.0) Properties set on this request. See the spec for more information on these properties.

Example of MQTT PUBACK/PUBREL/PUBREC/PUBCOMP logging:

```
"puback": {
  "qos": 0,
  "retain": false,
  "dup": false,
  "message_id": 1,
  "reason_code": 16
}
```

MQTT SUBSCRIBE fields

- "subscribe.message_id": (Only present if QOS level > 0) Message ID for this subscription.
- "subscribe.topics": Array of pairs describing the subscribed topics:
  - "subscribe.topics[].topic": Topic to subscribe to.
  - "subscribe.topics[].qos": QOS level to apply for when subscribing.
- "subscribe.properties": (Optional, MQTT 5.0) SUBSCRIBE properties set on this request. See 3.8.2.1 in the spec for more information on SUBSCRIBE properties.

Example of MQTT SUBSCRIBE logging:

```
"subscribe": {
  "qos": 1,
  "retain": false,
  "dup": false,
  "message_id": 1,
  "topics": [
    {
      "topic": "topicX",
      "qos": 0
    },
    {
      "topic": "topicY",
      "qos": 0
```

(continues on next page)

(continued from previous page)

```
    }
  ]
}
```

MQTT SUBACK fields

- "suback.message_id": Original message ID this message refers to.
- "suback.qos_granted": Array of QOS levels granted for the subscribed topics, in the order of the original request.
- "suback.properties": (Optional, MQTT 5.0) SUBACK properties set on this request. See 3.9.2.1 in the spec for more information on SUBACK properties.

Example of MQTT SUBACK logging:

```
"suback": {
  "qos": 0,
  "retain": false,
  "dup": false,
  "message_id": 1,
  "qos_granted": [
    0,
    0
  ]
}
```

MQTT UNSUBSCRIBE fields

- "unsubscribe.message_id": (Only present if QOS level > 0) Message ID for this unsubscribe action.
- "unsubscribe.topics": Array of topics to be unsubscribed from.
- "unsubscribe.properties": (Optional, MQTT 5.0) UNSUBSCRIBE properties set on this request. See 3.10.2.1 in the spec for more information on UNSUBSCRIBE properties.

Example of MQTT UNSUBSCRIBE logging:

```
"unsubscribe": {
  "qos": 1,
  "retain": false,
  "dup": false,
  "message_id": 1,
  "topics": [
    "topicX",
    "topicY"
  ]
}
```

MQTT UNSUBACK fields

- "unsuback.message_id": Original message ID this message refers to.

Example of MQTT UNSUBACK logging:

```
"unsuback": {
  "qos": 0,
  "retain": false,
  "dup": false,
  "message_id": 1
}
```

MQTT AUTH fields (MQTT 5.0)

- "auth.reason_code": Return code/reason code for this message. See 3.15.2.1 in the spec for more information on these codes.
- "auth.properties": (Optional, MQTT 5.0) Properties set on this request. See 3.15.2.2 in the spec for more information on these properties.

Example of MQTT AUTH logging:

```
"auth": {
  "qos": 0,
  "retain": false,
  "dup": false,
  "reason_code": 16
}
```

MQTT DISCONNECT fields

- "auth.reason_code": (Optional) Return code/reason code for this message. See 3.14.2.1 in the spec for more information on these codes.
- "auth.properties": (Optional, MQTT 5.0) Properties set on this request. See 3.14.2.2 in the spec for more information on DISCONNECT properties.

Example of MQTT DISCONNECT logging:

```
"disconnect": {
  "qos": 0,
  "retain": false,
  "dup": false,
  "reason_code": 4,
  "properties": {
    "session_expiry_interval": 122,
  }
}
```

Truncated MQTT data

Messages exceeding the maximum message length limit (config setting `app-layer.protocols.mqtt.max-msg-length`) will not be parsed entirely to reduce the danger of denial of service issues. In such cases, only reduced metadata will be included in the EVE-JSON output. Furthermore, since no message ID is parsed, such messages can not be placed into transactions, hence, they will always appear as a single transaction.

These truncated events will -- besides basic communication metadata -- only contain the following fields:

- "truncated": Set to *true* if the entry is truncated.
- "skipped_length": Size of the original message.

Example of a truncated MQTT PUBLISH message (with 10000 being the maximum length):


```
{
  "timestamp": "2020-06-23T16:25:48.729785+0200",
  "flow_id": 1872904524326406,
  "pcap_cnt": 107,
  "event_type": "mqtt",
  "src_ip": "0000:0000:0000:0000:0000:0000:0000:0001",
  "src_port": 53335,
  "dest_ip": "0000:0000:0000:0000:0000:0000:0000:0001",
  "dest_port": 1883,
  "proto": "TCP",
  "mqtt": {
    "publish": {
      "qos": 0,
      "retain": false,
      "dup": false,
      "truncated": true,
      "skipped_length": 100011
    }
```


Event type: HTTP2

Fields

There are the two fields "request" and "response" which can each contain the same set of fields : * "settings": a list of settings with "name" and "value" * "headers": a list of headers with either "name" and "value", or "table_size_update", or "error" if any * "error_code": the error code from GOAWAY or RST_STREAM, which can be "NO_ERROR" * "priority": the stream priority.

Examples

Example of HTTP2 logging, of a settings frame:


```
"http2": {
  "request": {
    "settings": [
      {
        "settings_id": "SETTINGSMAXCONCURRENTSTREAMS",
        "settings_value": 100
      },
      {
        "settings_id": "SETTINGSINITIALWINDOWSIZE",
        "settings_value": 65535
      }
    ]
  },
  "response": {}
}
```


Example of HTTP2 logging, of a request and response:


```
"http2": {
  "request": {
```


(continues on next page)

(continued from previous page)


```
    "headers": [
      {
        "name": ":authority",
        "value": "localhost:3000"
      },
      {
        "name": ":method",
        "value": "GET"
      },
      {
        "name": ":path",
        "value": "/doc/manual/html/index.html"
      },
      {
        "name": ":scheme",
        "value": "http"
      },
      {
        "name": "accept",
        "value": "*/*"
      },
      {
        "name": "accept-encoding",
        "value": "gzip, deflate"
      },
      {
        "name": "user-agent",
        "value": "nghttp2/0.5.2-DEV"
      }
    ]
  },
  "response": {
    "headers": [
      {
        "name": ":status",
        "value": "200"
      },
      {
        "name": "server",
        "value": "nghttpd nghttp2/0.5.2-DEV"
      },
      {
        "name": "content-length",
        "value": "22617"
      },
      {
        "name": "cache-control",
        "value": "max-age=3600"
      },
      {
        "name": "date",
        "value": "Sat, 02 Aug 2014 10:50:25 GMT"
```


(continues on next page)

(continued from previous page)

```
      },
      {
        "name": "last-modified",
        "value": "Sat, 02 Aug 2014 07:58:59 GMT"
      }
    ]
  }
}
```

Event type: PGSQL

PGSQL eve-logs reflect the bidirectional nature of the protocol transactions. Each PGSQL event lists at most one "Request" message field and one or more "Response" messages.

The PGSQL parser merges individual messages into one EVE output item if they belong to the same transaction. In such cases, the source and destination information (IP/port) reflect the direction of the initial request, but contain messages from both sides.

Example of `pgsql` event for a SimpleQuery transaction complete with request with a `SELECT` statement and its response:

```
{
  "timestamp": "2021-11-24T16:56:24.403417+0000",
  "flow_id": 1960113262002448,
  "pcap_cnt": 780,
  "event_type": "pgsql",
  "src_ip": "172.18.0.1",
  "src_port": 54408,
  "dest_ip": "172.18.0.2",
  "dest_port": 5432,
  "proto": "TCP",
  "pgsql": {
    "tx_id": 4,
    "request": {
      "simple_query": "select * from rule limit 5000;"
    },
    "response": {
      "field_count": 7,
      "data_rows": 5000,
      "data_size": 3035751,
      "command_completed": "SELECT 5000"
    }
  }
}
```

While on the wire PGSQL messages follow basically two types (startup messages and regular messages), those may have different subfields and/or meanings, based on the message type. Messages are logged based on their type and relevant fields.

We list a few possible message types and what they mean in Suricata. For more details on message types and formats as well as what each message and field mean for PGSQL, check PostgreSQL's official documentation.

Fields

- "tx_id": internal transaction id.
- "request": each PGSQL transaction may have up to one request message. The possible messages will be described in another section.
- "response": even when there are several "Response" messages, there is one `response` field that summarizes all responses for that transaction. The possible messages will be described in another section.

Request Messages

Requests are sent by the frontend (client), which would be the source of a pgsql flow. Some of the possible request messages are:

- "startup_message": message sent to start a new PostgreSQL connection
- "password": if password output for PGSQL is enabled in suricata.yaml, carries the password sent during Authentication phase
- "password_redacted": set to true in case there is a password message, but its logging is disabled
- "simple_query": issued SQL command during simple query subprotocol. PostgreSQL identifies specific sets of commands that change the set of expected messages to be exchanged as subprotocols.
- `"message": "cancel_request"`: sent after a query, when the frontend attempts to cancel said query. This message is sent over a different port, thus bring shown as a different flow. It has no direct answer from the backend, but if successful will lead to an `ErrorResponse` in the transaction where the query was sent.
- "message": requests which do not have meaningful payloads are logged like this, where the field value is the message type
- "copy_data_in": object. Part of the CopyIn subprotocol, consolidated data resulting from a `Copy From Stdin` query
- "copy_done": string. Similar to `command_completed` but sent after the frontend finishes sending a batch of `CopyData` messages

There are several different authentication messages possible, based on selected authentication method. (e.g. the SASL authentication will have a set of authentication messages different from when `md5` authentication is chosen).

Response Messages

Responses are sent by the backend (server), which would be the destination of a pgsql flow. Some of the possible request messages are:

- "authentication_sasl_final": final SCRAM `server-final-message`, as explained at https://www.postgresql.org/docs/14/sasl-authentication.html#SASL-SCRAM-SHA-256
- "message": Backend responses which do not have meaningful payloads are logged like this, where the field value is the message type
- "error_response"
- "notice_response"
- "notification_response"
- "authentication_md5_password": a string with the `md5` salt value
- "parameter_status": logged as an array
- "backend_key_data"

- "data_rows": integer. When one or many `DataRow` messages are parsed, the total returned rows
- "data_size": in bytes. When one or many `DataRow` messages are parsed, the total size in bytes of the data returned
- "command_completed": string. Informs the command just completed by the backend
- "copy_in_response": object. Indicates the beginning of a CopyIn mode, shows how many columns will be copied from STDIN (`columns` field)
- "copy_out_response": object. Indicates the beginning of a CopyTo mode, shows how many columns will be copied to STDOUT (`columns` field)
- "copy_data_out": object. Consolidated data on the CopyData sent by the backend in a CopyOut transaction
- "copy_done": string. Similar to `command_completed` but sent after the backend finishes sending a batch of `CopyData` messages
- "field_count": integer. The number of fields declared by a `RowDescription` message
- "row_description_malformed": bool. True when the `field_count` declared by a `RowDescription` message disagreed with the message body delimited by its length field. The declared `field_count` is still logged, as seen on the wire
- "ssl_accepted": bool. With this event, the initial PGSQL SSL Handshake negotiation is complete in terms of tracking and logging. The session will be upgraded to use TLS encryption

Examples

The two `pgsql` events in this example represent a rejected `SSL handshake` and a following connection request where the authentication method indicated by the backend was `md5`:


```
{
  "timestamp": "2021-11-24T16:56:19.435242+0000",
  "flow_id": 1960113262002448,
  "pcap_cnt": 21,
  "event_type": "pgsql",
  "src_ip": "172.18.0.1",
  "src_port": 54408,
  "dest_ip": "172.18.0.2",
  "dest_port": 5432,
  "proto": "TCP",
  "pgsql": {
    "tx_id": 1,
    "request": {
      "message": "SSL Request"
    },
    "response": {
      "accepted": false
    }
  }
}
{
  "timestamp": "2021-11-24T16:56:19.436228+0000",
  "flow_id": 1960113262002448,
  "pcap_cnt": 25,
  "event_type": "pgsql",
  "src_ip": "172.18.0.1",
  "src_port": 54408,
```


(continues on next page)

(continued from previous page)


```
  "dest_ip": "172.18.0.2",
  "dest_port": 5432,
  "proto": "TCP",
  "pgsql": {
    "tx_id": 2,
    "request": {
      "protocol_version": "3.0",
      "startup_parameters": {
        "user": "rules",
        "database": "rules",
        "optional_parameters": [
          {
            "application_name": "psql"
          },
          {
            "client_encoding": "UTF8"
          }
        ]
      }
    },
    "response": {
      "authentication_md5_password": "Z\\xdc\\xfdf"
    }
  }
}
```


`AuthenticationOk`: a response indicating that the connection was successfully established.:


```
{
  "pgsql": {
    "tx_id": 3,
    "response": {
      "message": "authentication_ok",
      "parameter_status": [
        {
          "application_name": "psql"
        },
        {
          "client_encoding": "UTF8"
        },
        {
          "date_style": "ISO, MDY"
        },
        {
          "integer_datetimes": "on"
        },
        {
          "interval_style": "postgres"
        },
        {
          "is_superuser": "on"
        },
```


(continues on next page)

(continued from previous page)


```
        {
          "server_encoding": "UTF8"
        },
        {
          "server_version": "13.6 (Debian 13.6-1.pgdg110+1)"
        },
        {
          "session_authorization": "rules"
        },
        {
          "standard_conforming_strings": "on"
        },
        {
          "time_zone": "Etc/UTC"
        }
      ],
      "process_id": 28954,
      "secret_key": 889887985
    }
  }
}
```


Note

In Suricata, the `AuthenticationOk` message is also where the backend's `process_id` and `secret_key` are logged. These must be sent by the frontend when it issues a `CancelRequest` message (seen below).

A `CancelRequest` message:


```
{
   "timestamp": "2023-12-07T15:46:56.971150+0000",
   "flow_id": 775771889500133,
   "event_type": "pgsql",
   "src_ip": "100.88.2.140",
   "src_port": 39706,
   "dest_ip": "100.96.199.113",
   "dest_port": 5432,
   "proto": "TCP",
   "pkt_src": "stream (flow timeout)",
   "pgsql": {
     "tx_id": 1,
     "request": {
       "message": "cancel_request",
       "process_id": 28954,
       "secret_key": 889887985
     }
   }
}
```

> **Note**
>
> As the `CancelRequest` message is sent over a new connection, the way to correlate it with the proper frontend/flow from which it originates is by querying on `process_id` and `secret_key` seen in the `AuthenticationOk` event.

References:

- PostgreSQL protocol - Canceling Requests in Progress
- PostgreSQL message format - BackendKeyData

Field Reference

Top Level (object)

Name	Type	Description
request	object	
response	object	
tx_id	integer	

response (object)

Name	Type	Description
authentication_md5_password	string	
authentication_sasl_final	string	
code	string	
command_completed	string	
copy_data_out	object	CopyData message from CopyOut mode
copy_in_response	object	Backend/server response accepting CopyIn mode
copy_out_response	object	Backend/server response accepting CopyOut mode
data_rows	integer	
data_size	integer	
field_count	integer	
file	string	
line	string	
message	string	The PostgreSQL message type seen on the wire. Or the primary human-readable error/notice PostgreSQL message present in an ErrorResponse message
parameter_status	array of objects	
position	string	String, representing a decimal ASCII integer, related to an error cursor position as an index to original PostgreSQL query string, as seen in an ErrorResponse message
process_id	integer	The process id of the backend process, as seen in BackendKeyData messages
routine	string	
row_description_malformed	boolean	True when the field count declared in a RowDescription message disagreed with the message body delimited by its length field
secret_key	integer	
severity_localizable	string	
severity_non_localizable	string	
ssl_accepted	boolean	
where	string	'where' field in an ErrorResponse PostgreSQL message, indicating the context in which the error occurred

response.parameter_status (array of objects)

Name	Type	Description
application_name	string	
client_encoding	string	
date_style	string	
integer_datetimes	string	
interval_style	string	
is_superuser	string	
server_encoding	string	
server_version	string	
session_authorization	string	
standard_conforming_strings	string	
time_zone	string	

response.copy_out_response (object)

Name	Type	Description
columns	integer	Number of columns that will be copied in the CopyData message
malformed	boolean	True in case the message body format on the wire disagreed with specifications

response.copy_in_response (object)

Name	Type	Description
columns	integer	Number of columns that will be copied in the CopyData message
malformed	boolean	True in case the message body format on the wire disagreed with specifications

response.copy_data_out (object)

Name	Type	Description
data_size	integer	Accumulated data size of all CopyData messages sent
row_count	integer	Number of rows sent in CopyData messages

request (object)

Name	Type	Description
copy_data_in	object	CopyData message from CopyIn mode
message	string	
password	string	
password_redacted	boolean	Indicates if a password message was received but not logged due to Suricata settings
process_id	integer	The process id of the target backend, as seen in CancelRequests messages
protocol_version	string	
sasl_authentication_mechanism	string	
sasl_param	string	
sasl_response	string	
secret_key	integer	
simple_query	string	
startup_parameters	object	

request.startup_parameters (object)

Name	Type	Description
optional_parameters	array of objects	
user	string	

request.startup_parameters.optional_parameters (array of objects)

Name	Type	Description
application_name	string	
client_encoding	string	
database	string	
datestyle	string	
extra_float_digits	string	
options	string	
replication	string	

request.copy_data_in (object)

Name	Type	Description
data_size	integer	Accumulated data size of all CopyData messages sent
msg_count	integer	How many CopyData messages were sent (does not necessarily match number of rows from the query)

Event type: IKE

The parser implementations for IKEv1 and IKEv2 have a slightly different feature set. They can be distinguished using the "version_major" field (which equals either 1 or 2). The unique properties are contained within a separate "ikev1" and "ikev2" sub-object.

Fields

- "init_spi", "resp_spi": The Security Parameter Index (SPI) of the initiator and responder.
- "version_major": Major version of the ISAKMP header.
- "version_minor": Minor version of the ISAKMP header.
- "payload": List of payload types in the current packet.
- "exchange_type": Type of the exchange, as numeric values.
- "exchange_type_verbose": Type of the exchange, in human-readable form. Needs `extended: yes` set in the `ike` EVE output option.
- "alg_enc", "alg_hash", "alg_auth", "alg_dh", "alg_esn": Properties of the chosen security association by the server.
- "ikev1.encrypted_payloads": Set to `true` if the payloads in the packet are encrypted.
- "ikev1.doi": Value of the domain of interpretation (DOI).
- "ikev1.server.key_exchange_payload", "ikev1.client.key_exchange_payload": Public key exchange payloads of the server and client.
- "ikev1.server.key_exchange_payload_length", "ikev1.client.key_exchange_payload_length": Length of the public key exchange payload.
- "ikev1.server.nonce_payload", "ikev1.client.nonce_payload": Nonce payload of the server and client.
- "ikev1.server.nonce_payload_length", "ikev1.client.nonce_payload_length": Length of the nonce payload.
- "ikev1.client.client_proposals": List of the security associations proposed to the server.
- "ikev1.vendor_ids": List of the vendor IDs observed in the communication.
- "server_proposals": List of server proposals with parameters, if there are more than one. This is a non-standard case; this field is only present if such a situation was observed in the inspected traffic.

Examples

Example of IKE logging:


```
"ike": {
  "version_major": 1,
  "version_minor": 0,
  "init_spi": "8511617bfea2f172",
  "resp_spi": "c0fc6bae013de0f5",
  "message_id": 0,
  "exchange_type": 2,
  "exchange_type_verbose": "Identity Protection",
  "sa_life_type": "LifeTypeSeconds",
  "sa_life_type_raw": 1,
  "sa_life_duration": "Unknown",
  "sa_life_duration_raw": 900,
```


(continues on next page)

(continued from previous page)


```
  "alg_enc": "EncAesCbc",
  "alg_enc_raw": 7,
  "alg_hash": "HashSha2_256",
  "alg_hash_raw": 4,
  "alg_auth": "AuthPreSharedKey",
  "alg_auth_raw": 1,
  "alg_dh": "GroupModp2048Bit",
  "alg_dh_raw": 14,
  "sa_key_length": "Unknown",
  "sa_key_length_raw": 256,
  "alg_esn": "NoESN",
  "payload": [
    "VendorID",
    "Transform",
    "Proposal",
    "SecurityAssociation"
  ],
  "ikev1": {
    "doi": 1,
    "encrypted_payloads": false,
    "client": {
      "key_exchange_payload": "0bf7907681a656aabed38fb1ba8918b10d707a8e635a...",
      "key_exchange_payload_length": 256,
      "nonce_payload": "1427d158fc1ed6bbbc1bd81e6b74960809c87d18af5f0abef14d5274ac232904
↪",
      "nonce_payload_length": 32,
      "proposals": [
        {
          "sa_life_type": "LifeTypeSeconds",
          "sa_life_type_raw": 1,
          "sa_life_duration": "Unknown",
          "sa_life_duration_raw": 900,
          "alg_enc": "EncAesCbc",
          "alg_enc_raw": 7,
          "alg_hash": "HashSha2_256",
          "alg_hash_raw": 4,
          "alg_auth": "AuthPreSharedKey",
          "alg_auth_raw": 1,
          "alg_dh": "GroupModp2048Bit",
          "alg_dh_raw": 14,
          "sa_key_length": "Unknown",
          "sa_key_length_raw": 256
        }
      ]
    },
    "server": {
      "key_exchange_payload": "1e43be52b088ec840ff81865074b6d459b5ca7813b46...",
      "key_exchange_payload_length": 256,
      "nonce_payload": "04d78293ead007bc1a0f0c6c821a3515286a935af12ca50e08905b15d6c8fcd4
↪",
      "nonce_payload_length": 32
    },
```


(continues on next page)

(continued from previous page)

```
    "vendor_ids": [
      "4048b7d56ebce88525e7de7f00d6c2d3",
      "4a131c81070358455c5728f20e95452f",
      "afcad71368a1f1c96b8696fc77570100",
      "7d9419a65310ca6f2c179d9215529d56",
      "cd60464335df21f87cfdb2fc68b6a448",
      "90cb80913ebb696e086381b5ec427b1f"
    ]
  },
}
```

Event type: Modbus

Common fields

- "id": The unique transaction number given by Suricata

Request/Response fields

- "transaction_id": The transaction id found in the packet
- "protocol_id": The modbus version
- "unit_id": ID of the remote server to interact with
- "function_raw": Raw value of the function code byte
- "function_code": Associated name of the raw function value
- "access_type": Type of access requested by the function
- "category": The function code's category
- "error_flags": Errors found in the data while parsing

Exception fields

- "raw": Raw value of the exception code byte
- "code": Associated name of the raw exception value

Diagnostic fields

- "raw": Raw value of the subfunction code bytes
- "code": Associated name of the raw subfunction value
- "data": Bytes following the subfunction code

MEI fields

- "raw": Raw value of the mei function code bytes
- "code": Associated name of the raw mei function value
- "data": Bytes following the mei function code

Read Request fields

- "address": Starting address to read from
- "quantity": Amount to read

Read Response fields

- "data": Data that was read

Multiple Write Request fields

- "address": Starting address to write to
- "quantity": Amount to write
- "data": Data to write

Mask Write fields

- "address": Starting address of content modification
- "and_mask": And mask to modify content with
- "or_mask": Or mask to modify content with

Other Write fields

- "address": Starting address to write to
- "data": Data to write

Generic Data fields

- "data": Data following the function code

Example

Example of Modbus logging of a request and response:

```
"modbus": {
  "id": 1,
  "request": {
    "transaction_id": 0,
    "protocol_id": 0,
    "unit_id": 0,
    "function_raw": 1,
    "function_code": "RdCoils",
    "access_type": "READ | COILS",
    "category": "PUBLIC_ASSIGNED",
    "error_flags": "NONE",
  },
  "response": {
    "transaction_id": 0,
    "protocol_id": 0,
    "unit_id": 0,
    "function_raw": 1,
```

(continues on next page)

(continued from previous page)

```
    "function_code": "RdCoils",
    "access_type": "READ | COILS",
    "category": "PUBLIC_ASSIGNED",
    "error_flags": "DATA_VALUE",
  },
}
```

Event type: QUIC

Fields

- "version": Version of the QUIC packet if contained in the packet, 0 if not
- "cyu": List of found CYUs in the packet
- "cyu[].hash": CYU hash
- "cyu[].string": CYU string
- "ja3": The JA3 fingerprint consisting of both a JA3 hash and a JA3 string
- "ja3s": The JA3S fingerprint consisting of both a JA3 hash and a JA3 string
- "ja4": The JA4 client fingerprint for QUIC

Examples

Example of QUIC logging with CYU, JA3 and JA4 hashes (note that the JA4 hash is only an example to illustrate the format and does not correlate with the others):

```
"quic": {
  "version": 1362113590,
  "cyu": [
      {
          "hash": "7b3ceb1adc974ad360cfa634e8d0a730",
          "string": "46,PAD-SNI-STK-SNO-VER-CCS-NONC-AEAD-UAID-SCID-TCID-PDMD-SMHL-ICSL-
↪NONP-PUBS-MIDS-SCLS-KEXS-XLCT-CSCT-COPT-CCRT-IRTT-CFCW-SFCW"
      }
  ],
  "ja3": {
      "hash": "324f8c50e267adba4b5dd06c964faf67",
      "string": "771,4865-4866-4867,51-43-13-27-17513-16-45-0-10-57,29-23-24,"
  },
  "ja4": "q13d0310h3_55b375c5d22e_cd85d2d88918"
}
```

Output Reference

Top Level (object)

Name	Type	Description
cyu	array of objects	JA3-like fingerprint for versions of QUIC before standardization
extensions	array of objects	list of extensions in hello
ja3	object	JA3 from client, as in TLS
ja3s	object	JA3 from server, as in TLS
ja4	string	
sni	string	Server Name Indication
ua	string	User Agent for versions of QUIC before standardization
version	string	Quic protocol version

ja3s (object)

Name	Type	Description
hash	string	JA3s hex representation
string	string	JA3s string representation

ja3 (object)

Name	Type	Description
hash	string	JA3 hex representation
string	string	JA3 string representation

extensions (array of objects)

Name	Type	Description
name	string	Human-friendly name of the extension
type	integer	Integer identifier of the extension
values	array of strings	Extension values

cyu (array of objects)

Name	Type	Description
hash	string	CYU hash hex representation
string	string	CYU hash string representation

Event type: DHCP

The default DHCP logging level only logs enough information to map a MAC address to an IP address. Enable extended mode to log all DHCP message types in full detail.

When a DHCP message carries the Option Overload entry (option 52, RFC 2132), the BOOTP `sname` and `file`

header fields are used as extra option storage. Suricata parses any options found in those continuation areas alongside the standard options block, so values carried in either area show up in the same EVE fields below.

Fields

- "type": message type (e.g. request, reply)
- "id": DHCP transaction id
- "client_mac": client MAC address
- "assigned_ip": IP address given by DHCP server
- "client_ip": client IP address
- "dhcp_type": DHCP message type
- "client_id": DHCP client identifier
- "hostname": DHCP client host name
- "params": DHCP parameter request list
- "requested_ip": DHCP client requesting specific IP address
- "relay_ip": BOOTP relay agent IP address
- "next_server_ip": BOOTP next IP address to use for booting process
- "subnet_mask": subnet mask to use with client IP address
- "routers": IP address(es) to be used as default gateways on DHCP client
- "lease_time": Duration of IP address assignment to client
- "renewal_time": Time in seconds since client began IP address request or renewal process
- "rebinding_time": Time in seconds before the client begins to renew its IP address lease
- "dns_servers": IP address(es) of servers the client will use for DNS queries

Examples

Example of DHCP log entry (default logging level):

```
"dhcp": {
  "type":"reply",
  "id":755466399,
  "client_mac":"54:ee:75:51:e0:66",
  "assigned_ip":"100.78.202.125",
  "dhcp_type":"ack",
  "renewal_time":21600,
  "client_id":"54:ee:75:51:e0:66"
}
```

Example of DHCP log entry (extended logging enabled):

```
"dhcp": {
  "type":"reply",
  "id":2787908432,
  "client_mac":"54:ee:75:51:e0:66",
  "assigned_ip":"192.168.1.120",
```

(continues on next page)

(continued from previous page)


```
  "client_ip":"0.0.0.0",
  "relay_ip":"192.168.1.1",
  "next_server_ip":"0.0.0.0",
  "dhcp_type":"offer",
  "subnet_mask":"255.255.255.0",
  "routers":["192.168.1.100"],
  "hostname":"test",
  "lease_time":86400,
  "renewal_time":21600,
  "rebinding_time":43200,
  "client_id":"54:ee:75:51:e0:66",
  "dns_servers":["192.168.1.50","192.168.1.49"]
}
```


Event type: ARP

Fields

- "hw_type": network link protocol type
- "proto_type": internetwork protocol for which the request is intended
- "opcode": operation that the sender is performing (e.g. request, response)
- "src_mac": source MAC address
- "src_ip": source IP address
- "dest_mac": destination MAC address
- "dest_ip": destination IP address

Examples

Example of ARP logging: request and response


```
"arp": {
  "hw_type": "ethernet",
  "proto_type": "ipv4",
  "opcode": "request",
  "src_mac": "00:1a:6b:6c:0c:cc",
  "src_ip": "10.10.10.2",
  "dest_mac": "00:00:00:00:00:00",
  "dest_ip": "10.10.10.1"
}
```

```
"arp": {
  "hw_type": "ethernet",
  "proto_type": "ipv4",
  "opcode": "reply",
  "src_mac": "00:1a:6b:6c:0c:cc",
  "src_ip": "10.10.10.2",
  "dest_mac": "00:1d:09:f0:92:ab",
  "dest_ip": "10.10.10.1"
}
```

Event type: POP3

Fields

- **"request" (optional): a request sent by the pop3 client**
  - "request.command" (string): a pop3 command, for example "USER" or "STAT", if unknown but valid *UnknownCommand* event will be set
  - "request.args" (array of strings): pop3 command arguments, if incorrect number for command *IncorrectArgumentCount* event will be set
- **"response" (optional): a response sent by the pop3 server**
  - "response.success" (boolean): whether the response is successful, ie. +OK
  - "response.status" (string): the response status, one of "OK" or "ERR"
  - "response.header" (string): the content of the first line of the response
  - "response.data" (array of strings): the response data, which may contain multiple lines

Example of POP3 logging:

```
"pop3": {
    "request": {
        "command": "USER",
        "args": ["user@example.com"],
    },
    "response": {
        "success": true,
        "status": "OK",
        "header": "+OK password required for \"user@example.com\"",
        "data": []
    }
 }
```

Event type: Netflow

Fields

- "age": duration of the flow (measured from timestamp of last packet and first packet)
- "bytes": total number of bytes to client
- "end": date of the end of the flow
- "max_ttl": maximum observed Time-To-Live (TTL) value
- "min_ttl": minimum observed TTL value
- "pkts": total number of packets to client
- "start": date of start of the flow
- "tx_cnt": number of transactions seen in the flow (only present if flow has an application layer)

Example

```
"netflow": {
   "pkts": 1,
   "bytes": 160,
```

(continues on next page)

(continued from previous page)

```
  "start": "2013-02-26T17:02:42.907340-0500",
  "end": "2013-02-26T17:02:42.907340-0500",
  "age": 0,
  "min_ttl": 1,
  "max_ttl": 1
}
```

15.1.3 Eve JSON 'jq' Examples

The jq tool is very useful for quickly parsing and filtering JSON files. This page is contains various examples of how it can be used with Suricata's Eve.json.

The basics are discussed here:

- https://www.stamus-networks.com/2015/05/18/looking-at-suricata-json-events-on-command-line/

Colorize output

```
tail -f eve.json | jq -c '.'
```

DNS NXDOMAIN

```
tail -f eve.json|jq -c 'select(.dns.rcode=="NXDOMAIN")'
```

Unique HTTP User Agents

```
cat eve.json | jq -s '[.[]|.http.http_user_agent]|group_by(.)|map({key:.[0],value:(.
↪|length)})|from_entries'
```

Source: https://twitter.com/mattarnao/status/601807374647750657

Data use for a host

```
tail -n500000 eve.json | jq -s 'map(select(.event_type=="netflow" and .dest_ip=="192.168.
↪1.3").netflow.bytes)|add'|numfmt --to=iec
1.3G
```

Note: can use a lot of memory. Source: https://twitter.com/pkt_inspector/status/605524218722148352

Monitor part of the stats

```
$ tail -f eve.json | jq -c 'select(.event_type=="stats")|.stats.decoder'
```

Inspect Alert Data

```
cat eve.json | jq -r -c 'select(.event_type=="alert")|.payload'|base64 --decode
```

Top 10 Destination Ports

```
cat eve.json | jq -c 'select(.event_type=="flow")|[.proto, .dest_port]'|sort |uniq -
↪c|sort -nr|head -n10
```

15.2 Lua Output

Suricata offers the possibility to get more detailed output on specific kinds of network traffic via pluggable lua scripts. You can write these scripts yourself and only need to define four hook functions.

For lua output scripts suricata offers a wide range of lua functions. They all return information on specific engine internals and aspects of the network traffic. They are described in the following sections, grouped by the event/traffic type. But let's start with an example explaining the four hook functions, and how to make suricata load a lua output script.

15.2.1 Script structure

A lua output script needs to define 4 hook functions: init(), setup(), log(), deinit()

- init() -- registers where the script hooks into the output engine
- setup() -- does per output thread setup
- log() -- logging function
- deinit() -- clean up function

Example:

```
local config = require("suricata.config")
local logger = require("suricata.log")
local http = require("suricata.http")
local packet = require("suricata.packet")
local flow = require("suricata.flow")

function init (args)
    local needs = {}
    needs["protocol"] = "http"
    return needs
end

function setup (args)
    filename = config.log_path() .. "/" .. name
    file = assert(io.open(filename, "a"))
    logger.info("HTTP Log Filename " .. filename)
    http = 0
end

function log(args)
    local tx = http.get_tx()

    http_uri = tx:request_uri_raw()
    if http_uri == nil then
        http_uri = "<unknown>"
    end
    http_uri = string.gsub(http_uri, "%c", ".")

    http_host = tx:request_host()
    if http_host == nil then
        http_host = "<hostname unknown>"
    end
```

(continues on next page)

(continued from previous page)

```
    http_host = string.gsub(http_host, "%c", ".")

    http_ua = tx:request_header("User-Agent")
    if http_ua == nil then
        http_ua = "<useragent unknown>"
    end
    http_ua = string.gsub(http_ua, "%g", ".")

    local p = packet.get()
    timestring = p:timestring_legacy()
    ip_version, src_ip, dst_ip, protocol, src_port, dst_port = p:tuple()

    file:write (timestring .. " " .. http_host .. " [**] " .. http_uri .. " [**] " ..
           http_ua .. " [**] " .. src_ip .. ":" .. src_port .. " -> " ..
           dst_ip .. ":" .. dst_port .. "\n")
    file:flush()

    http = http + 1
end

function deinit (args)
    logger.info ("HTTP transactions logged: " .. http);
    file:close(file)
end
```

15.2.2 YAML

To enable the lua output, add the 'lua' output and add one or more scripts like so:

```
outputs:
  - lua:
      enabled: yes
      scripts-dir: /etc/suricata/lua-output/

      # By default the Lua module search paths are empty. If you plan
      # to use external modules these paths will need to be set. The
      # examples below are likely suitable for finding modules
      # installed with a package manager on a 64 bit Linux system, but
      # may need tweaking.
      #path: "/usr/share/lua/5.4/?.lua;/usr/share/lua/5.4/?/init.lua;/usr/lib64/lua/5.4/?
↪.lua;/usr/lib64/lua/5.4/?/init.lua;./?.lua;./?/init.lua"
      #cpath: "/usr/lib64/lua/5.4/?.so;/usr/lib64/lua/5.4/loadall.so;./?.so"

      scripts:
        - tcp-data.lua
        - flow.lua
```

The scripts-dir option is optional. It makes Suricata load the scripts from this directory. Otherwise scripts will be loaded from the current workdir.

15.2.3 Developing lua output script

You can use functions described in *Lua Functions*

15.3 Syslog Alerting Compatibility

Suricata can alert via syslog which is a very handy feature for central log collection, compliance, and reporting to a SIEM. Instructions on setting this up can be found in the .yaml file in the section where you can configure what type of alert (and other) logging you would like.

However, there are different syslog daemons and there can be parsing issues with the syslog format a SIEM expects and what syslog format Suricata sends. The syslog format from Suricata is dependent on the syslog daemon running on the Suricata sensor but often the format it sends is not the format the SIEM expects and cannot parse it properly.

Attention

The syslog output is deprecated in Suricata 8.0 and will be removed in Suricata 9.0. Please migrate to the `eve` output which has the ability to send to syslog.

15.3.1 Popular syslog daemons

- **syslogd** - logs system messages
- **syslog-ng** - logs system messages but also supports TCP, TLS, and other enhanced enterprise features
- **rsyslogd** - logs system messages but also support TCP, TLS, multi-threading, and other enhanced features
- **klogd** - logs kernel messages
- **sysklogd** - basically a bundle of syslogd and klogd

If the syslog format the Suricata sensor is sending is not compatible with what your SIEM or syslog collector expects, you will need to fix this. You can do this on your SIEM if it is capable of being able to be configured to interpret the message, or by configuring the syslog daemon on the Suricata sensor itself to send in a format you SIEM can parse. The latter can be done by applying a template to your syslog config file.

15.3.2 Finding what syslog daemon you are using

There are many ways to find out what syslog daemon you are using but here is one way:

```
cd /etc/init.d
ls | grep syslog
```

You should see a file with the word syslog in it, e.g. "syslog", "rsyslogd", etc. Obviously if the name is "rsyslogd" you can be fairly confident you are running rsyslogd. If unsure or the filename is just "syslog", take a look at that file. For example, if it was "rsyslogd", run:

```
less rsyslogd
```

At the top you should see a comment line that looks something like this:

```
# rsyslog        Starts rsyslogd/rklogd.
```

Locate those files and look at them to give you clues as to what syslog daemon you are running. Also look in the *start()* section of the file you ran "less" on and see what binaries get started because that can give you clues as well.

15.3.3 Example

Here is an example where the Suricata sensor is sending syslog messages in rsyslogd format but the SIEM is expecting and parsing them in a sysklogd format. In the syslog configuration file (usually in /etc with a filename like rsyslog.conf or syslog.conf), first add the template:

```
$template sysklogd, "<%PRI%>%syslogtag:1:32%%msg:::sp-if-no-1st-sp%%msg%"
```

Then send it to the syslog server with the template applied:

```
user.alert @10.8.75.24:514;sysklogd
```

Of course this is just one example and it will probably be different in your environment depending on what syslog daemons and SIEM you use but hopefully this will point you in the right direction.

15.4 Custom tls logging

Attention

tls-log is deprecated in Suricata 8.0 and will be removed in Suricata 9.0.

In your Suricata.yaml, find the tls-log section and edit as follows:

```
- tls-log:
    enabled: yes       # Log TLS connections.
    filename: tls.log # File to store TLS logs.
    append: yes
    custom: yes        # enabled the custom logging format (defined by customformat)
    customformat: "%{%D-%H:%M:%S}t.%z %a:%p -> %A:%P %v %n %d %D"
```

And in your tls.log file you would get the following, for example:

```
12/03/16-19:20:14.85859 10.10.10.4:58274 -> 192.0.78.24:443 VERSION='TLS 1.2' suricata.
↪io NOTBEFORE='2016-10-27T20:36:00' NOTAFTER='2017-01-25T20:36:00'
```

The list of supported format strings is the following:

- %n - client SNI
- %v - TLS/SSL version
- %d - certificate date not before
- %D - certificate date not after
- %f - certificate fingerprint SHA1
- %s - certificate subject
- %i - certificate issuer dn
- %E - extended format
- %{strftime_format}t - timestamp of the TLS transaction in the selected strftime format. ie: 08/28/12-22:14:30
- %z - precision time in useconds. ie: 693856
- %a - client IP address

- %p - client port number
- %A - server IP address
- %P - server port number

Any non printable character will be represented by its byte value in hexadecimal format (|XX|, where XX is the hex code)

15.5 Log Rotation

All outputs in the *outputs* section of the configuration file can be subject to log rotation.

For most outputs an external tool like *logrotate* is required to rotate the log files in combination with sending a SIGHUP to Suricata to notify it that the log files have been rotated.

On receipt of a SIGHUP, Suricata simply closes all open log files and then re-opens them in append mode. If the external tool has renamed any of the log files, new files will be created, otherwise the files will be re-opened and new data will be appended to them with no noticeable affect.

The following is an example *logrotate* configuration file that will rotate Suricata log files then send Suricata a SIGHUP triggering Suricata to open new files:

```
/var/log/suricata/*.log /var/log/suricata/*.json
{
    rotate 3
    missingok
    nocompress
    create
    sharedscripts
    postrotate
            /bin/kill -HUP `cat /var/run/suricata.pid 2>/dev/null` 2>/dev/null || true
    endscript
}
```

Note

The above *logrotate* configuration file depends on the existence of a Suricata PID file. If running in daemon mode a PID file will be created by default, otherwise the `--pidfile` option should be used to create a PID file.

In addition to the SIGHUP style rotation discussed above, some outputs support their own time and date based rotation, however removal of old log files is still the responsibility of external tools. These outputs include:

- *Eve*
- *PCAP log*

15.5.1 Windows

Windows does not support SIGHUP. Instead, when Suricata is running as a Windows service, it can be told to re-open its log files by sending the service the user-defined control code `129`:

```
sc.exe control Suricata 129
```

As with SIGHUP, an external tool is responsible for renaming the log files before sending the control code. Suricata opens log files on Windows with shared delete access so they can be renamed while open. See *Via Windows service control* for who may send control codes.

Note

In PowerShell, `sc` is an alias for `Set-Content`, so `sc.exe` must be used.

CHAPTER

SIXTEEN

LUA SUPPORT

16.1 Lua usage in Suricata

Lua scripting can be used in two components of Suricata:

- Output
- Detection: `lua` keyword and `luaxform` transform

Both features are using a list of functions to access the data extracted by Suricata. You can get the list of functions in the *Lua functions* page.

Note

Currently, there is a difference in the `needs` key in the `init` function, depending on what is the usage: `output` or `detection`. The list of available functions may also differ. The `luaxform` doesn't use the `needs` key.

16.1.1 Lua output

Lua scripts can be used to write arbitrary output. See *Lua Output* for more information.

16.1.2 Lua detection

Lua scripts can be used as a filter condition in signatures. See *Lua Scripting for Detection* for more information.

16.1.3 Lua transform

The `luaxform` transform can be used in signatures. See *luaxform* for more information.

16.2 Lua functions

16.2.1 Differences between *output* and *detect*:

Currently the `table` returned from the `init` method varies, depending on whether it is in an output script or a detection script.

Lua scripts for `luaxform` do not require an `init` method.

If the script is for detection, the `init` method should return a table, for example, if a packet is required:

```
function init (args)
  local needs = {}
  needs["packet"] = true
  return needs
end
```

See *Lua Scripting for Detection* for more detection script examples.

For output scripts, follow the pattern below. (The complete script structure can be seen at *Lua Output*:)

```
function init (args)
    local needs = {}
    needs["protocol"] = "tls"
    return needs
end
```

Do notice that the functions and protocols available for `log` and `match` may also vary. DNP3, for instance, is not available for logging.

Note

By convention, many scripts use a variable name of `needs` for this table, however this is not a hard requirement.

16.2.2 packet

Initialize with:

```
function init (args)
    local needs = {}
    needs["type"] = "packet"
    return needs
end
```

16.2.3 flow

```
function init (args)
    local needs = {}
    needs["type"] = "flow"
    return needs
end
```

16.2.4 http

For output, init with:

```
function init (args)
    local needs = {}
    needs["protocol"] = "http"
    return needs
end
```

For detection, rule hooks are used to execute the Lua script at specific protocol states, for example:

```
alert http1:request_line any any -> any any (
    msg: "Test HTTP Lua request.line";
    lua: test-request-line.lua; sid:1;)
```

where `test-request-line.lua` might look like:

```
local http = require("suricata.http")

function init (args)
  return {}
end

function match(args)
  local tx, err = http.get_tx()
  http_request_line, err = tx:request_line()

  if #http_request_line > 0 then
      --GET /base64-hello-world.txt HTTP/1.1
      if http_request_line:find("^GET") then
          return 1
      end
  end

  return 0
end
```

For more information on rule hooks, see *Explicit rule hook (states)*.

16.2.5 Streaming Data

Streaming data can currently log out reassembled TCP data and normalized HTTP data. The script will be invoked for each consecutive data chunk.

In case of TCP reassembled data, all possible overlaps are removed according to the host OS settings.

```
function init (args)
    return {streaming = "tcp"}
end
```

In case of HTTP body data, the bodies are unzipped and dechunked if applicable.

```
function init (args)
    return {streaming = "http"}
end
```

The streaming data will be provided in the `args` to the log function within a `stream` subtable:

```
function log(args)
  -- The data (buffer)
  local data = args["stream"]["data"]

  -- Buffer open?
  local open = args["stream"]["open"]
```

(continues on next page)

(continued from previous page)

```
  -- Buffer closed?
  local close = args["stream"]["close"]

  -- To server?
  local ts = args["stream"]["to_server"]

  -- To client?
  local tc = args["stream"]["to_client"]
end
```

16.3 Lua Libraries

Suricata provides Lua extensions, or libraries to Lua scripts with the `require` keyword. These extensions are particularly important in Lua rules as Lua rules are executed in a restricted sandbox environment without access to additional modules.

16.3.1 Base64

Base64 functions are exposed to Lua scripts with the `suricata.base64` library. For example:

```
local base64 = require("suricata.base64")
```

Functions

encode(string)

Encode a buffer with standard base64 encoding. This standard encoding includes padding.

decode(string)

Decode a base64 string that contains padding.

encode_nopad(string)

Encode a buffer with standard base64 encoding but don't include any padding.

decode_nopad(string)

Decode a base64 string that contains no padding.

decode_padopt(string)

Decode a base64 string that may or may not contain trailing padding.

decode_rfc2045(string)

Decode an RFC 2045 formatted base64 string.

decode_rfc4648(string)

Decode an RFC 4648 formatted base64 string.

Implementation Details

The base64 functions provided come from the Rust base64 library documented at https://docs.rs/base64 and correspond to the `STANDARD` and `STANDARD_NO_PAD` base64 engines provided in that library.

16.3.2 Bytevar

The `suricata.bytevar` module provides access to variables defined by `byte_extract` and `byte_math` keywords in Suricata rules.

It is only available in Suricata Lua rules, not output scripts.

Setup

```
local bytevars = require("suricata.bytevar")
```

Module Functions

`bytevars.map`(*sig*, *varname*)

Ensures that the `bytevar` exists and sets it up for further use in the script by mapping it into the Lua context. Must be called during `init()`.

Parameters

- **sig** -- The signature object passed to `init()`
- **varname** (*string*) -- Name of the variable as defined in the rule

Raises

- **error** -- If the variable name is unknown
- **error** -- If too many byte variables are mapped

Example:

```
function init(sig)
    bytevars.map(sig, "var1")
    bytevars.map(sig, "var2")
    return {}
end
```

`bytevars.get`(*name*)

Returns a byte variable object for the given name. May be called during `thread_init()` to save a handle to the bytevar.

Parameters

name (*number*) -- Name of the variable previously setup with `map()`.

Raises

error -- If variable name is not mapped with `map()`.

Returns

A byte variable object

Example:

```
function thread_init()
    bv_var1 = bytevars.get("var1")
    bv_var2 = bytevars.get("var2")
end
```

Byte Variable Object Methods

bytevar:value()

Returns the current value of the byte variable.

Returns
The value of the byte variable.

Example:

```
function match(args)
    local var1 = bv_var1:value()
    if var1 then
        -- Use the value
    end
end
```

16.3.3 Config Library

The config library provides access to Suricata configuration settings.

To use this library, you must require it:

```
local config = require("suricata.config")
```

Functions

log_path()

Returns the configured log directory path.

Example:

```
local config = require("suricata.config")

local log_path, err = config.log_path()
if log_path == nil then
   print("failed to get log path " .. err)
end
```

16.3.4 DNP3

The `suricata.dnp3` module provides access to DNP3 (Distributed Network Protocol 3) transaction data in Suricata Lua rules.

It is only available in Suricata Lua rules, not output scripts.

Setup

```
local dnp3 = require("suricata.dnp3")
```

Module Functions

dnp3.**get_tx**()

Returns the current DNP3 transaction object containing request or response data.

Returns
A table containing the DNP3 transaction data, or nil on error

Raises

- **error** -- If the protocol is not DNP3
- **error** -- If no transaction is available

Example:

```
function match(args)
    local tx = dnp3.get_tx()
    if tx and tx.is_request then
        -- Process DNP3 request
    end
end
```

Transaction Object Structure

The transaction object returned by `get_tx()` contains the following fields:

tx_num

Transaction number (integer)

is_request

Boolean indicating if this is a request (true) or response (false)

request

Table containing request data (only present when `is_request` is true)

response

Table containing response data (only present when `is_request` is false)

Request/Response Structure

Both request and response tables contain:

done

Boolean indicating if the transaction is complete

complete

Boolean indicating if all data has been received

link_header

Table containing DNP3 link layer header fields:

- `len`: Frame length
- `control`: Control byte

- `dst`: Destination address
- `src`: Source address
- `crc`: CRC value

`transport_header`
Transport layer header byte (integer)

`application_header`
Table containing DNP3 application layer header fields:

- `control`: Application control byte
- `function_code`: DNP3 function code

`objects`
Array of DNP3 objects in the message

Additionally, response tables contain:

`indicators`
Internal Indication (IIN) field as a 16-bit integer combining IIN1 and IIN2

Objects Structure

Each object in the `objects` array contains:

`group`
DNP3 object group number (integer)

`variation`
DNP3 object variation number (integer)

`points`
Data points of this object, as an array-like value. `#points` gives the point count, `points[i]` (1-based) returns the point as a table, and reads outside `1..#points` return nil; the value supports `pairs()` and numeric loops. Objects of 32 points or fewer carry a copy that stays readable; objects with more points are backed by the transaction itself, so they are valid while that transaction is open - once it is freed (complete, inspected and logged, or at flow teardown) `points[i]` returns nil.

Points Structure

Each point (table) of the `points` value contains:

`index`
Point index (integer)

Additional point fields depend on the object group and variation. Common fields include:

- `state`: Binary state value
- `online`: Online status flag
- `restart`: Restart flag
- `comm_lost`: Communication lost flag
- `remote_forced`: Remote forced flag
- `local_forced`: Local forced flag
- `chatter_filter`: Chatter filter flag

- `reserved`: Reserved bits
- `value`: Analog value (for analog objects)
- `timestamp`: Timestamp value (for time-tagged objects)

For all available fields, see `app-layer-dnp3-objects.h` in the Suricata source code.

Example Usage

Complete example checking for specific DNP3 function codes:

```
local dnp3 = require("suricata.dnp3")

function init(args)
    return {}
end

function match(args)
    local tx = dnp3.get_tx()

    if not tx then
        return 0
    end

    -- Check for write function code in request
    if tx.is_request and tx.request then
        local func_code = tx.request.application_header.function_code
        if func_code == 2 then  -- WRITE function
            return 1
        end
    end

    -- Check for specific object types
    if tx.request and tx.request.objects then
        for _, obj in ipairs(tx.request.objects) do
            if obj.group == 12 and obj.variation == 1 then
                -- Control Relay Output Block
                return 1
            end
        end
    end

    return 0
end
```

16.3.5 DNS

DNS transaction details are exposed to Lua scripts with the `suricata.dns` library, for example:

```
local dns = require("suricata.dns")
```

Setup

If your purpose is to create a logging script, initialize the buffer as:

```
function init (args)
   local needs = {}
   needs["protocol"] = "dns"
   return needs
end
```

If you are going to use the script for rule matching, choose one of the available DNS buffers listed in *Lua Scripting for Detection* and follow the pattern:

For use in rule matching, the rule must **hook** into a DNS transaction state. Available states are `request_complete` and `response_complete`. For example:

alert dns:request_complete any any -> any any (sid: 1;)

Then to initialize the script:

```
function init (args)
   return {}
end
```

Transaction

DNS is transaction based, and the current transaction must be obtained before use:

```
local tx, err = dns.get_tx()
if tx == err then
    print(err)
end
```

All other functions are methods on the transaction table.

Transaction Methods

answers()

Get the `answers` response section as a table of tables.

Example:

```
local tx = dns.get_tx()
local answers = tx:answers()
if answers ~= nil then
    for n, t in pairs(answers) do
        rrname = t["rrname"]
        rrtype = t["type"]
        ttl = t["ttl"]

        print ("ANSWER: " .. ts .. " " .. rrname .. " [**] " .. rrtype .. " [**] " ..
               ttl .. " [**] " .. srcip .. ":" .. sp .. " -> " ..
               dstip .. ":" .. dp)
    end
end
```

authorities()

Get the authorities response section as a table of tables.

Example:

```
local tx = dns.get_tx()
local authorities = tx:authorities();
if authorities ~= nil then
    for n, t in pairs(authorities) do
        rrname = t["rrname"]
        rrtype = t["type"]
        ttl = t["ttl"]
         print ("AUTHORITY: " .. ts .. " " .. rrname .. " [**] " .. rrtype .. " [**] " ..
               ttl .. " [**] " .. srcip .. ":" .. sp .. " -> " ..
               dstip .. ":" .. dp)
    end
end
```

queries()

Get the queries request or response section as a table of tables.

Example:

```
local tx = dns.get_tx()
local queries = tx:queries();
if queries ~= nil then
    for n, t in pairs(queries) do
        rrname = t["rrname"]
        rrtype = t["type"]

        print ("QUERY: " .. ts .. " " .. rrname .. " [**] " .. rrtype .. " [**] " ..
               "TODO" .. " [**] " .. srcip .. ":" .. sp .. " -> " ..
               dstip .. ":" .. dp)
    end
end
```

rcode()

Get the rcode value as an integer.

Example:

```
local tx = dns.get_tx()
local rcode = tx:rcode()
print (rcode)
```

rcode_string()

Get the rcode value as a string.

Example:

```
local tx = dns.get_tx()
local rcode_string = tx:rcode_string();
print (rcode_string)
```

recursion_desired()

Return the value of the recursion desired (RD) flag as a boolean.

Example:

```
local tx = dns.get_tx()
if tx:recursion_desired() == true then
    print ("RECURSION DESIRED")
end
```

rrname()

Return the resource name from the first query object.

Example:

```
local tx = dns.get_tx()
local rrname = tx:rrname()
print(rrname)
```

txid()

Return the DNS transaction ID found in the DNS message.

Example:

```
local tx = dns.get_tx()
local txid = tx:txid()
print(txid)
```

16.3.6 File

File information is exposed to Lua scripts with the `suricata.file` library, for example:

```
local filelib = require("suricata.file")
```

Setup

If your purpose is to create a logging script, initialize the script as:

```
function init (args)
   local needs = {}
   needs["type"] = "file"
   return needs
end
```

Currently the Lua file library is not implemented for rules.

API

File Object

File data is accessed through the file object, which must be obtained before use:

```
local file, err = filelib.get_file()
if file == nil then
    print(err)
end
```

File Methods

file_id()

Returns the ID number of the file.

Example:

```
local file = filelib.get_file()
local id = file:file_id()
print("File ID: " .. id)
```

tx_id()

Returns the transaction ID associated with the file.

Example:

```
local file = filelib.get_file()
local tx_id = file:tx_id()
print("Transaction ID: " .. tx_id)
```

name()

Returns the file name.

Example:

```
local file = filelib.get_file()
local name = file:name()
if name ~= nil then
    print("Filename: " .. name)
end
```

size()

Returns the file size.

Example:

```
local file = filelib.get_file()
local size = file:size()
print("File size: " .. size .. " bytes")
```

magic()

Returns the file type based on libmagic (if available). Will return nil if magic is not available.

Example:

```
local file = filelib.get_file()
local magic = file:magic()
if magic ~= nil then
    print("File type: " .. magic)
end
```

md5()

Returns the MD5 hash of the file (if calculated). Will return nil if the MD5 hash was not calculated.

Example:

```
local file = filelib.get_file()
local md5 = file:md5()
if md5 ~= nil then
    print("MD5: " .. md5)
end
```

sha1()

Returns the SHA1 hash of the file (if calculated). Will return nil if the SHA1 hash was not calculated.

Example:

```
local file = filelib.get_file()
local sha1 = file:sha1()
if sha1 ~= nil then
    print("SHA1: " .. sha1)
end
```

sha256()

Returns the SHA256 hash of the file (if calculated). Will return nil if the SHA256 hash was not calculated.

Example:

```
local file = filelib.get_file()
local sha256 = file:sha256()
if sha256 ~= nil then
    print("SHA256: " .. sha256)
end
```

get_state()

Returns the current state of the file.

Returns:

- **State: "CLOSED", "TRUNCATED", "ERROR", "OPENED", "NONE", or** "UNKNOWN"

Example:

```
local file = filelib.get_file()
local state = file:get_state()
if state ~= nil then
    print("File state: " .. state)
end
```

is_stored()

Returns true if the file has been stored to disk, false otherwise.

Example:

```
local file = filelib.get_file()
local stored = file:is_stored()
print("File stored: " .. tostring(stored))
```

16.3.7 Flow

Flows are exposed to Lua scripts with the `suricata.flow` library. To use it, the script must require it. For example:

```
local flow = require("suricata.flow")
```

Following are the functions currently available for acessing Flow details.

Initialization

get

Init the flow for use in the script. The flow is the current one the engine is processing. `flow.get()` returns the current flow in packet, buffer and app-layer rule contexts; in transform contexts, and for packets without a flow, it returns `nil` and an error message. In output scripts the tx, file, alert, packet, flow and streaming loggers set the flow; the stats logger does not.

```
f = flow.get()
```

Time

timestamps

Get timestamps of the first and the last packets from the flow, as seconds and microseconds since *1970-01-01 00:00:00* UTC, returning 4 numbers:

```
f = flow.get()
local start_sec, last_sec, start_usec, last_usec = f:timestamps()
```

timestring_legacy

Get the timestamp of the first packet from the flow, as a string in the format: *11/24/2009-18:57:25.179869*. This is the format used by *fast.log*.

```
f = flow.get()
print f:timestring_legacy()
```

timestring_iso8601

Get the timestamp of the first packet from the flow, as a string in the format: *2015-10-06T15:16:43.136733+0000*. This is the format used by EVE outputs.

```
f = flow.get()
print f:timestring_iso8601()
```

Ports and Addresses

tuple

Using the *tuple* method, the IP version (4 or 6), src IP and dest IP (as string), IP protocol (int), and ports (ints) are retrieved.

For a flow with an IP header this returns 6 values. For a packet without an IP header it returns 1 value only: the IP version, 0 (the packet is neither IPv4 nor IPv6, so there is no IP version).

The protocol value comes from the IP header. See further https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml .

```
f = flow.get()
ipver, srcip, dstip, proto, sp, dp = f:tuple()
```

App Layer Protocols

app_layer_proto

Get *alproto* from the flow as a string. If an *alproto* is not (yet) known, it returns "unknown".

Returns 5 values: <alproto>, <alproto_ts>, <alproto_tc>, <alproto_orig>, <alproto_expect>.

Example:

```
f = flow.get()
alproto, alproto_ts, alproto_tc, alproto_orig, alproto_expect = f:app_layer_proto()
```

orig and *expect* are used when changing and upgrading protocols. In an SMTP STARTTLS case, *orig* would normally be set to "smtp" and *expect* to "tls".

Misc

has_alerts

Returns *true* if the flow has alerts.

```
f = flow.get()
alerted = f:has_alerts()
```

id

Get the flow id. Note that simply printing the `id` will likely result in printing a scientific notation. To avoid that, simply do:

```
f = flow.get()
id = f:id()
```

(continues on next page)

(continued from previous page)

```
id_str = string.format("%.0f", id)
print ("Flow ID: " .. id_str .."\n")
```

stats

Get the packet and byte counts (for both directions), as 4 numbers, per flow.

```
f = flow.get()
tscnt, tsbytes, tccnt, tcbytes = f:stats()
```

Example

A simple `log` function for a script to output Flow details if the flow triggered an alert:

```
function log(args)
    local f = flow.get()
    ts = f:timestring_iso8601()
    has_alerts = f:has_alerts()
    ipver, srcip, dstip, proto, sp, dp = f:tuple()
    alproto, alproto_ts, alproto_tc, alproto_orig, alproto_expect = f:app_layer_proto()
    start_sec, start_usec, last_sec, last_usec = f:timestamps()
    id = f:id()

    if has_alerts then
        file:write ("[**] Start time " .. ts .. " [**] -> alproto " .. alproto .. " [**]
↪" .. proto .. " [**] alerted: true\n[**] First packet: " .. start_sec .." [**] Last␣
↪packet: " .. last_sec .. " [**] Flow id: " .. id .. "\n")
        file:flush()
    end
end
```

For complete scripts using these and other lua functions, the Suricata-verify can be a good resource: https://github.com/OISF/suricata-verify/tree/master/tests .

16.3.8 Flowint Library

The `suricata.flowint` library exposes `flowint` variables to Lua scripts.

Initialization

First, the `flowint` module must be loaded:

```
local flowintlib = require("suricata.flowint")
```

Then in the `init` method, any flow integers used in the script should be registered. This is optional and could be skipped if you know for sure the flow integers will be registered by some other means.

Example:

```
local flowintlib = require("suricata.flowint")

function init ()
    flowintlib.register("count")
```

(continues on next page)

(continued from previous page)

```
    return {}
end
```

Finally, in the `thread_init` function a handle is acquired for the flow integers and stored as a global:

```
function thread_init ()
    count_flow_int = flowintlib.get("count")
end
```

Flow Integer Methods

decr()

Decrement the value of the `flowint` by 1. The new value is returned. If the value is 0, it will remain 0.

incr()

Increment the value of the `flowint` by 1. The new value is returned.

value()

Get the current value of the flow integer. Note that `nil` may be returned if the flow integer does not have a value.

set(value)

Set the value of the flowint to the value provided.

16.3.9 Flowvar

The `suricata.flowvar` library exposes flow variables to Lua scripts.

Initialization

First, the `flowvar` lib module must be loaded:

```
local flowvarlib = require("suricata.flowvar")
```

Then in the `init` method, any flow variables used in the script should be registered. This is optional and could be skipped if you know for sure the flow variable will be registered by some other means.

Example:

```
local flowvarlib = require("suricata.flowvar")

function init ()
    flowvarlib.register("count")
    return {}
end
```

Finally, in the `thread_init` function a handle is acquired for the flow variables and stored as a global:

```
function thread_init ()
    count_flow_var = flowvarlib.get("count")
end
```

Flow Variable Methods

value()

Get the current value of the flow variable as a string. Note that `nil` may be returned if the flow variable does not have a value.

set(value, len)

Set the value of the flow variable to the value provided. The length of the value must also be provided.

Example

```
local flowvarlib = require("suricata.flowvar")

function init ()
    flowvarlib.register("count")
    return {}
end

function thread_init ()
    count_var = flowvarlib.get("count")
end

function match ()
    local value = count_var:value()
    if value == nil then
        -- Initialize value to 1.
        value = tostring(1)
        count_var:set(value, #value)
    else
        value = tostring(tonumber(value) + 1)
        count_var:set(value, #value)
    fi

    -- Return 1 or 0 based on your own logic.
    return 1
end
```

16.3.10 Hashing

Hashing functions are exposed to Lua scripts with `suricata.hashlib` library. For example:

```
local hashing = require("suricata.hashlib")
```

If `--disable-hashing` is set, every hashlib call raises a Lua error instead of computing a digest, consistent with the hashing transforms and JA3/JA4. Inputs of 4 GiB (2 to the 32nd bytes) or more are rejected, matching the 32-bit length the FFI takes.

SHA-256

sha256_digest(string)

SHA-256 hash the provided string returning the digest as bytes.

sha256_hexdigest(string)

SHA-256 hash the provided string returning the digest as a hex string.

sha256()

Returns a SHA-256 hasher that can be updated multiple times, for example:

```
local hashing = require("suricata.hashlib")
hasher = hashing.sha256()
hasher:update("www.suricata")
hasher:update(".io")
hash = hasher:finalize_to_hex()
```

The methods on the hasher object include:

- `update(string)`: Add more data to the hasher
- `finalize()`: Finalize the hash returning the hash as a byte string
- `finalize_to_hex()`: Finalize the hash returning the has as a hex string

SHA-1

sha1_digest(string)

SHA-1 hash the provided string returning the digest as bytes.

sha1_hexdigest(string)

SHA-1 hash the provided string returning the digest as a hex string.

sha1()

Returns a SHA-1 hasher that can be updated multiple times, for example:

```
local hashing = require("suricata.hashlib")
hasher = hashing.sha1()
hasher.update("www.suricata")
hasher.update(".io")
hash = hasher.finalize_to_hex()
```

The methods on the hasher object include:

- `update(string)`: Add more data to the hasher
- `finalize()`: Finalize the hash returning the hash as a byte string
- `finalize_to_hex()`: Finalize the hash returning the has as a hex string

MD5

md5_digest(string)

MD5 hash the provided string returning the digest as bytes.

md5_hexdigest(string)

MD5 hash the provided string returning the digest as a hex string.

md5()

Returns a MD5 hasher that can be updated multiple times, for example:

```
local hashing = require("suricata.hashlib")
hasher = hashing.md5()
hasher:update("www.suricata")
hasher:update(".io")
hash = hasher:finalize_to_hex()
```

The methods on the hasher object include:

- `update(string)`: Add more data to the hasher
- `finalize()`: Finalize the hash returning the hash as a byte string
- `finalize_to_hex()`: Finalize the hash returning the hash as a hex string

16.3.11 HTTP

HTTP transaction details are exposes to Lua scripts with the `suricata.http` library, For example:

```
local http = require("suricata.http")
```

Setup

If your purpose is to create a logging script, initialize the buffer as:

```
function init (args)
   local needs = {}
   needs["protocol"] = "http"
   return needs
end
```

If you are going to use the script for rule matching, choose one of the available HTTP buffers listed in *Lua Scripting for Detection* and follow the pattern:

```
function init (args)
   local needs = {}
   needs["http.request_line"] = tostring(true)
   return needs
end
```

Transaction

HTTP is transaction based, and the current transaction must be obtained before use:

```
local tx, err = http.get_tx()
if tx == err then
    print(err)
end
```

All other functions are methods on the transaction table.

Transaction Methods

request_header()

Get the HTTP request header value by key.

Example:

```
local tx = http.get_tx()
local ua = tx:request_header("User-Agent")
if ua ~= nil then
      print(ua)
end
```

response_header()

Get the HTTP response header value by key.

Example:

```
local tx = http.get_tx()
local content_type = tx:response_header("Content-Type")
if content_type ~= nil then
      print(content_type)
end
```

request_line

Get the HTTP request line as a string.

Example:

```
local tx = http.get_tx()
local http_request_line = tx:request_line();
if #http_request_line > 0 then
    if http_request_line:find("^GET") then
        print(http_request_line)
    end
end
```

response_line

Get the HTTP response line as a string.

Example:

```
local tx = http.get_tx()
local http_response_line = tx:response_line();
if #http_response_line > 0 then
      print(http_response_line)
end
```

request_headers_raw()

Get the raw HTTP request headers.

Example:

```
http_request_headers_raw = tx:request_headers_raw()

if #http_request_headers_raw > 0 then
    if http_request_headers_raw:find("User%-Agent: curl") then
        print(http_request_headers_raw)
    end
end
```

response_headers_raw()

Get the raw HTTP response headers.

Example:

```
http_response_headers_raw = tx:response_headers_raw()

if #http_response_headers_raw > 0 then
      print(http_response_headers_raw)
end
```

request_uri_raw()

Get the raw HTTP request URI.

Example:

```
local tx = http.get_tx()
http_request_uri_raw = tx:request_uri_raw()
print(http_request_uri_raw)
```

request_uri_normalized()

Get the normalized HTTP request URI.

Example:

```
local tx = http.get_tx()
http_request_uri_normalized = tx:request_uri_normalized()
print(http_request_uri_normalized)
```

request_headers()

Get the HTTP request headers.

Example:

```
local tx = http.get_tx()
http_request_headers = tx:request_headers()
print(http_request_headers)
```

response_headers()

Get the HTTP response headers.

Example:

```
local tx = http.get_tx()
http_response_headers = tx:response_headers()
print(http_response_headers)
```

request_body()

Get the HTTP request body.

Example:

```
local tx = http.get_tx()
http_request_body = tx:request_body()
print(http_request_body)
```

response_body()

Get the HTTP response body.

Example:

```
local tx = http.get_tx()
http_response_body = tx:response_body()
print(http_response_body)
```

request_host()

Get the HTTP request host.

Example:

```
local tx = http.get_tx()
http_host = tx:request_host()
print(http_host)
```

16.3.12 Log

The `suricata.log` Lua library exposes the Suricata application logging functions to Lua scripts. These are equivalant to `SCLogNotice`, `SCLogError`, etc, in the Suricata source.

In Suricata, the logging priority order is:

- Error
- Warning
- Notice
- Info
- Perf
- Config
- Debug

> **Note**
>
> Debug logging will only work if Suricata was compiled with `--enable-debug`.

Setup

To use the logging functions, first require the module:

```
local logger = require("suricata.log")
```

Functions

info

Log an informational message:

```
logger.info("Processing HTTP request")
```

This is equivalent to `SCLogInfo`.

notice

Log a notice message:

```
logger.notice("Unusual pattern detected")
```

This is equivalent to `SCLogNotice`.

warning

Log a warning message:

```
logger.warning("Connection limit approaching")
```

This is equivalent to `SCLogWarning`.

error

Log an error message:

```
logger.error("Failed to parse data")
```

This is equivalent to `SCLogError`.

debug

Log a debug message (only visible when debug logger.ing is enabled):

```
logger.debug("Variable value: " .. tostring(value))
```

This is equivalent to `SCLogDebug`.

config

Log a configuration-related message:

```
logger.config("Loading configuration from " .. filename)
```

This is equivalent to `SCLogConfig`.

perf

Log a performance-related message:

```
logger.perf("Processing took " .. elapsed .. " seconds")
```

This is equivalent to `SCLogPerf`.

16.3.13 NTP

NTP transaction details are exposed to Lua scripts with the `suricata.ntp` library, for example:

```
local ntp = require("suricata.ntp")
```

Setup

If your purpose is to create a logging script, initialize the buffer as:

```
function init (args)
   local needs = {}
   needs["protocol"] = "ntp"
   return needs
end
```

Transaction

NTP is transaction based, and the current transaction must be obtained before use:

```
local tx, err = ntp.get_tx()
if tx == nil then
    print(err)
end
```

All other functions are methods on the transaction table.

Transaction Methods

version()

Get the NTP version as an integer.

mode()

Get the NTP mode as an integer.

stratum()

Get the NTP stratum as an integer.

reference_id()

Get the NTP reference ID as a raw 4-byte binary string.

Example:

```
local tx, err = ntp.get_tx()
local ref_id = tx:reference_id()
if ref_id == "\x4c\x4f\x43\x4c" then
  -- ref_id matches "LOCL"
end

-- If looking for a specific printable string, this is also valid
if ref_id == "LOCL" then
  ...
end
```

16.3.14 Packet

Packets are exposed to Lua scripts with `suricata.packet` library. For example:

```
local packet = require("suricata.packet")
```

Initialization

get

Init the packet for use in the script. The packet is the current packet the engine is processing. `packet.get()` is only available in packet context; in buffer, app-layer and transform contexts it returns `nil` and an error message. In output scripts the tx, file, alert and packet loggers set the packet; the flow, streaming and stats loggers do not.

```
p = packet.get()
```

Time

timestamp

Get packet timestamp as 2 numbers: seconds & microseconds elapsed since 1970-01-01 00:00:00 UTC.

```
p = packet.get()
local sec, usec = p:timestamp()
```

timestring_legacy

Get packet timestamp as a string in the format: *11/24/2009-18:57:25.179869*. This is the format used by *fast.log*.

```
p = packet.get()
print p:timestring_legacy()
```

timestring_iso8601

Get packet timestamp as a string in the format: *2015-10-06T15:16:43.137833+0000*. This is the format used by *eve*.

```
p = packet.get()
print p:timestring_iso8601()
```

Ports and Addresses

tuple

Using the *tuple* method the IP version (4 or 6), src IP and dest IP (as string), IP protocol (int) and ports (ints) are retrieved.

For a packet with an IP header this returns 6 values. For non-IP packets it returns 1 value only: the IP version, 0.

The protocol value comes from the IP header, see further https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml

```
p = packet.get()
ipver, srcip, dstip, proto, sp, dp = p:tuple()
```

If the protocol is ICMPv4 or ICMPv6, so when *proto == 1* or *proto == 58*, then the final two results are *icmp type* and *icmp code*.

```
p = packet.get()
ipver, srcip, dstip, proto, itype, icode = p:tuple()
if ipver == 6 and proto == 1 then
    -- weird, ICMPv4 on IPv6
    return 1
end
```

sp

Get the packets TCP, UDP or SCTP source port as an int. Returns *nil* for other protocols.

```
p = packet.get()
source_port = p:sp()
if source_port == 31337 then
    return 1
end
```

dp

Get the packets TCP, UDP or SCTP destination port as an int. Returns *nil* for other protocols.

```
p = packet.get()
dest_port = p:dp()
-- not port 443
if dest_port ~= 443 then
    return 1
end
```

Data

payload

Packet payload.

```
payload = p:payload()
```

packet

Entire packet, including headers for protocols like TCP, Ethernet, VLAN, etc.

```
raw_packet = p:packet()
```

Misc

pcap_cnt

The packet number when reading from a pcap file.

```
p = packet.get()
print p:pcap_cnt()
```

Example

Example *match* function that takes a packet, inspect the payload line by line and checks if it finds the HTTP request line. If it is found, issue a notice log with packet details.

```
local logger = require("suricata.log")

function match (args)
    p = packet.get()
    payload = p:payload()
    ts = p:timestring()

    for line in payload:gmatch("([^\r\n]*)[\r\n]+") do
        if line == "GET /index.html HTTP/1.0" then
            ipver, srcip, dstip, proto, sp, dp = p:tuple()
            logger.notice(string.format("%s %s->%s %d->%d (pcap_cnt:%d) match! %s", ts,␣
↪srcip, dstip, sp, dp, p:pcap_cnt(), line));
            return 1
        end
    end

    return 0
end
```

16.3.15 Rule

Rule details for an alert are exposed to Lua scripts with the `suricata.rule` library, for example:

```
local rule = require("suricata.rule")
```

Rule Setup

For use in Suricata Lua rules, no additional setup is required.

Output Setup

For use in Suricata Lua output scripts, some additional setup is required:

```
function init(args)
    return {
        type = "packet",
        filter = "alerts",
    }
end
```

Getting a Rule Instance

To obtain a rule object, use the `get_rule()` function on the `rule` library:

```
local sig = rule.get_rule()
```

Rule Methods

`action()`

Returns the action of the rule, for example: *alert*, *pass*.

`class_description()`

Returns the classification description.

`gid()`

Returns the generator ID of the rule.

`rev()`

Returns the revision of the rule.

`msg()`

Returns the rule message (`msg`).

`priority`

Returns the priority of the rule as a number.

`sid()`

Returns the signature ID of the rule.

16.3.16 SMTP

SMTP transaction details are exposed to Lua scripts with the `suricata.smtp` library, for example:

```
local smtp = require("suricata.smtp")
```

Setup

If your purpose is to create a logging script, initialize the buffer as:

```
function init (args)
   local needs = {}
   needs["protocol"] = "smtp"
   return needs
end
```

Otherwise if a detection script:

```
function init (args)
  return {}
end
```

API

Transaction

SMTP is transaction based, and the current transaction must be obtained before use:

```
local tx, err = smtp.get_tx()
if tx == nil then
    print(err)
end
```

All other functions are methods on the transaction table.

How long a transaction handle is valid

A handle is valid while its transaction remains open: it is resolved by identity within the flow of the current invocation, so a method call after the transaction is gone (freed once it is complete, inspected and logged, or at flow teardown) aborts the script with a Lua error - counted in `detect.lua.errors` for detection scripts - and never returns data from another transaction or flow. The sandbox blocks `pcall`/`xpcall`, so a detection script cannot catch the error (output scripts are unsandboxed). Keep handles short-lived: do not cache them across packets.

`smtp.get_tx()` with no transaction available returns the usual `nil` plus `error: no SMTP transaction found` - the invocation continues and nothing is logged or counted - which is what keeps it distinguishable from a stale handle.

`get_mail_from()` on a transaction whose field has not been seen yet returns a single `nil` with no error, so an empty `mail_from` is distinguishable from a stale handle. `get_mime_field()` and `get_mime_list()` return `nil` plus an error before DATA or for a field not seen.

Transaction Methods

`get_mime_field(name)`

Get a specific MIME header field by name from the SMTP transaction.

Example:

```
local tx = smtp.get_tx()
local encoding = tx:get_mime_field("Content-Transfer-Encoding")
if encoding ~= nil then
    print("Encoding: " .. subject)
end
```

get_mime_list()

Get all the MIME header field names from the SMTP transaction as a table, starting with the first header.

> **Note**
>
> The list is indexed from the first header. Earlier releases (up to 8.0.x as of this change) started at the header after the first one, so scripts written against those shift by one.

Example:

```
local tx = smtp.get_tx()
local mime_fields = tx:get_mime_list()
if mime_fields ~= nil then
    for i, name in pairs(mime_fields) do
        local value = tx:get_mime_field(name)
        print(name .. ": " .. value)
    end
end
```

get_mail_from()

Get the sender email address from the MAIL FROM command.

Example:

```
local tx = smtp.get_tx()
local mail_from = tx:get_mail_from()
if mail_from ~= nil then
    print("Sender: " .. mail_from)
end
```

get_rcpt_list()

Get all recipient email addresses from RCPT TO commands as a table.

Example:

```
local tx = smtp.get_tx()
local recipients = tx:get_rcpt_list()
if recipients ~= nil then
    for i, recipient in pairs(recipients) do
        print("Recipient " .. i .. ": " .. recipient)
    end
end
```

16.3.17 SSH

SSH transaction details are exposes to Lua scripts with the `suricata.ssh` library, For example:

```
local ssh = require("suricata.ssh")
```

If you want to use hassh, you can either set suricata.yaml option `app-layer.protocols.ssh.hassh` to true, or specify it in the `init` function of your lua script by calling `ssh.enable_hassh()`:

```
function init (args)
  ssh.enable_hassh()
  return {}
end
```

`ssh.enable_hassh()` is a process-global, parse-time switch that only affects flows parsed after the call, and returns no value.

For use in rule matching, the rule must **hook** into a SSH transaction state. Available states are listed in *Hooks*. For example:

alert ssh:response_kex any any -> any any (sid: 1;)

Setup

If your purpose is to create a logging script, initialize the buffer as:

```
function init (args)
   local needs = {}
   return needs
end
```

If you are going to use the script for rule matching, choose one of the available SSH buffers listed in *Lua Scripting for Detection* and follow the pattern:

```
function init (args)
   local needs = {}
   return needs
end
```

Transaction

SSH is transaction based, and the current transaction must be obtained before use:

```
local tx, err = ssh.get_tx()
if tx == err then
    print(err)
end
```

All other functions are methods on the transaction table.

Transaction Methods

`server_proto()`

Get the `server_proto` value as a string.

Example:

```
local tx = ssh.get_tx()
local proto = tx:server_proto();
print (proto)
```

client_proto()

Get the client_proto value as a string.

Example:

```
local tx = ssh.get_tx()
local proto = tx:client_proto();
print (proto)
```

server_software()

Get the server_software value as a string.

Example:

```
local tx = ssh.get_tx()
local software = tx:server_software();
print (software)
```

client_software()

Get the client_software value as a string.

Example:

```
local tx = ssh.get_tx()
local software = tx:client_software();
print (software)
```

client_hassh()

Should be used with ssh.enable_hassh().

Get MD5 of hassh algorithms used by the client through client_hassh.

Example:

```
local tx = ssh.get_tx()
local h = tx:client_hassh();
print (h)
```

client_hassh_string()

Should be used with ssh.enable_hassh().

Get hassh algorithms used by the client through client_hassh_string.

Example:

```
local tx = ssh.get_tx()
local h = tx:client_hassh_string();
print (h)
```

server_hassh()

Should be used with `ssh.enable_hassh()`.

Get MD5 of hassh algorithms used by the server through server_hassh.

Example:

```
local tx = ssh.get_tx()
local h = tx:server_hassh();
print (h)
```

server_hassh_string()

Should be used with `ssh.enable_hassh()`.

Get hassh algorithms used by the server through server_hassh_string.

Example:

```
local tx = ssh.get_tx()
local h = tx:server_hassh_string();
print (h)
```

16.3.18 TLS

TLS details are exposed to Lua scripts with the `suricata.tls` library, for example:

```
local tls = require("suricata.tls")
```

Setup

If your purpose is to create a logging script, initialize the buffer as:

```
function init (args)
   local needs = {}
   needs["protocol"] = "tls"
   return needs
end
```

Otherwise if a detection script:

```
function init (args)
  return {}
end
```

API

Transaction

TLS is transaction based, and the current transaction must be obtained before use:

```
local tx, err = tls.get_tx()
if tx == nil then
    print(err)
end
```

All other functions are methods on the transaction table.

Client Methods

get_client_version

Get the negotiated version in a TLS session as a string through `get_client_version`.

Example:

```
function log (args)
    t, err = tls.get_tx()
    version = t:get_client_version()
    if version ~= nil then
        -- do something
    end
end
```

get_client_cert_chain

Make certificate chain available to the script through `get_client_cert_chain`

The output is an array of certificate with each certificate being an hash with *data* and *length* keys.

Example:

```
-- Use debian lua-luaossl coming from https://github.com/wahern/luaossl
local x509 = require"openssl.x509"

   chain = t:get_client_cert_chain()
   for k, v in pairs(chain) do
      -- v.length is length of data
      -- v.data is raw binary data of certificate
      print("data length is" .. v["length"] .. "\n")
      cert = x509.new(v["data"], "DER")
      print(cert:text() .. "\n")
   end
```

get_client_cert_info

Make certificate information available to the script through `get_client_cert_info`

Example:

```
function log (args)
    version, subject, issuer, fingerprint = t:get_client_cert_info()
    if version ~= nil then
        -- do something
    end
end
```

get_client_cert_not_after

Get the Unix timestamp of end of validity of certificate.

Example:

```
function log (args)
    notafter = t:get_client_cert_not_after()
    if notafter < os.time() then
        -- expired certificate
    end
end
```

get_client_cert_not_before

Get the Unix timestamp of beginning of validity of certificate.

Example:

```
function log (args)
    notbefore = t:get_client_cert_not_before()
    if notbefore > os.time() then
        -- not yet valid certificate
    end
end
```

get_client_serial

Get TLS certificate serial number through `get_client_serial`.

Example:

```
function log (args)
    serial = t:get_client_serial()
    if serial ~= nil then
        -- do something
    end
end
```

get_client_sni

Get the Server name Indication from a TLS connection.

Example:

```
function log (args)
    asked_domain = t:get_client_sni()
    if string.find(asked_domain, "badguys") then
        -- ok connection to bad guys let's do something
    end
end
```

Server Methods

get_server_cert_info

Make certificate information available to the script through `get_server_cert_info`

Example:

```
function log (args)
    version, subject, issuer, fingerprint = t:get_server_cert_info()
    if version ~= nil then
        -- do something
    end
end
```

get_server_cert_chain

Make certificate chain available to the script through `get_server_cert_chain`

The output is an array of certificate with each certificate being an hash with *data* and *length* keys.

Example:

```
-- Use debian lua-luaossl coming from https://github.com/wahern/luaossl
local x509 = require"openssl.x509"

   chain = t:get_server_cert_chain()
   for k, v in pairs(chain) do
      -- v.length is length of data
      -- v.data is raw binary data of certificate
      print("data length is" .. v["length"] .. "\n")
      cert = x509.new(v["data"], "DER")
      print(cert:text() .. "\n")
   end
```

get_server_cert_not_after

Get the Unix timestamp of end of validity of certificate.

Example:

```
function log (args)
    notafter = t:get_server_cert_not_after()
    if notafter < os.time() then
        -- expired certificate
    end
end
```

get_server_cert_not_before

Get the Unix timestamp of beginning of validity of certificate.

Example:

```
function log (args)
    notbefore = t:get_server_cert_not_before()
    if notbefore > os.time() then
        -- not yet valid certificate
    end
end
```

get_server_serial

Get TLS certificate serial number through `get_server_serial`.

Example:

```
function log (args)
    serial = t:get_server_serial()
    if serial ~= nil then
        -- do something
    end
end
```

16.3.19 JA3

JA3 details are exposed to Lua scripts with the `suricata.ja3` library. For example:

```
local ja3 = require("suricata.ja3")
```

If you want to use ja3, you can either set suricata.yaml option `app-layer.protocols.tls.ja3-fingerprints` to true, or specify it in the `init` function of your lua script by calling `ja3.enable_ja3()`:

```
function init (args)
  ja3.enable_ja3()
  return {}
end
```

`ja3.enable_ja3()` will not enable ja3 if they are explicitly disabled, so you should add `requires: feature ja3;` (see *requires*) to your rule. It is a process-global, parse-time switch that only affects flows parsed after the call, and returns no value.

For use in rule matching, the rule should use need `ja3` or `ja3s` in your init script:

```
function init (args)
  ja3.enable_ja3()
  local needs = {}
  needs["ja3s"] = true
  return needs
end
```

Transaction

JA3 is transaction based, and the current transaction must be obtained before use:

```
local tx, err = ja3.get_tx()
if tx == err then
```

(continues on next page)

(continued from previous page)

```
    print(err)
end
```

All other functions are methods on the transaction (either a QUIC or a TLS one).

Transaction Methods

ja3_get_hash()

Get the ja3 value as a hash.

Example:

```
local tx = ja3.get_tx()
local h = tx:ja3_get_hash();
print (h)
```

ja3_get_string()

Get the ja3 value as a string.

Example:

```
local tx = ja3.get_tx()
local s = tx:ja3_get_string();
print (s)
```

ja3s_get_hash()

Get the ja3s value as a hash.

Example:

```
local tx = ja3.get_tx()
local h = tx:ja3s_get_hash();
print (h)
```

ja3s_get_string()

Get the ja3s value as a string.

Example:

```
local tx = ja3.get_tx()
local s = tx:ja3s_get_string();
print (s)
```

16.3.20 Util

The `suricata.util` library provides utility functions for Lua scripts.

Setup

The library must be loaded prior to use:

```
local util = require("suricata.util")
```

Functions

thread_info()

Get information about the current thread.

Returns

Table containing thread information with the following fields:

- `id` (number): Thread ID
- `name` (string): Thread name

Example:

```
local util = require("suricata.util")

local info = util.thread_info()
print("Thread ID: " .. info.id)
print("Thread Name: " .. info.name)
```

CHAPTER

SEVENTEEN

FILE EXTRACTION

17.1 Architecture

The file extraction code works on top of selected protocol parsers (see supported protocols below). The application layer parsers run on top of the stream reassembly engine and the UDP flow tracking.

In case of HTTP, the parser takes care of dechunking and unzipping the request and/or response data if necessary.

This means that settings in the stream engine, reassembly engine and the application layer parsers all affect the workings of the file extraction.

The rule language controls which files are extracted and stored on disk.

Supported protocols are:

- HTTP
- SMTP
- FTP
- NFS
- SMB
- HTTP2

17.2 Settings

stream.checksum_validation controls whether or not the stream engine rejects packets with invalid checksums. This is a good idea normally, but if the network interface performs checksum offloading, a lot of packets may seem to be broken. This setting is enabled by default, and can be disabled by setting to "no". Note that the checksum handling can be controlled per interface; see "checksum_checks" in example configuration.

file-store.stream-depth controls how far into a stream reassembly is done. Beyond this value no reassembly will be done. This means that after this value the HTTP session will no longer be tracked. By default a setting of 1 Megabyte is used. 0 sets it to unlimited. If set to "no", it is disabled and *stream.reassembly.depth* is considered. Non-zero values must be greater than `stream.stream-depth` to be used.

libhtp.default-config.request-body-limit / *libhtp.server-config.<config>.request-body-limit* controls how much of the HTTP request body is tracked for inspection by the *http_client_body* keyword, but also used to limit file inspection. A value of 0 means unlimited.

libhtp.default-config.response-body-limit / *libhtp.server-config.<config>.response-body-limit* is like the request body limit, only it applies to the HTTP response body.

17.3 Output

17.3.1 File-Store and Eve Fileinfo

There are two output modules for logging information about extracted files. The first is `eve.files` which is an `eve` sub-logger that logs `fileinfo` records. These `fileinfo` records provide metadata about the file, but not the actual file contents.

This must be enabled in the `eve` output:

```
- outputs:
    - eve-log:
      types:
          - files:
              force-magic: no
              force-hash: [md5,sha256]
```

See *Eve (Extensible Event Format)* for more details on working with the *eve* output.

The other output module, `file-store` stores the actual files to disk.

The `file-store` module uses its own log directory (default: *filestore* in the default logging directory) and logs files using the SHA256 of the contents as the filename. Each file is then placed in a directory named *00* to *ff* where the directory shares the first 2 characters of the filename. For example, if the SHA256 hex string of an extracted file starts with "f9bc6d..." the file we be placed in the directory *filestore/f9*.

The size of a file that can be stored depends on `file-store.stream-depth`, if this value is reached a file can be truncated and might not be stored completely. If not enabled, `stream.reassembly.depth` will be considered.

Setting `file-store.stream-depth` to 0 permits store of the entire file; here, 0 means "unlimited."

`file-store.stream-depth` will always override `stream.reassembly.depth` when filestore keyword is used. However, it is not possible to set `file-store.stream-depth` to a value less than `stream.reassembly.depth`. Values less than this amount are ignored and a warning message will be displayed.

A protocol parser, like modbus, could permit to set a different store-depth value and use it rather than `file-store.stream-depth`.

Using the SHA256 for file names allows for automatic de-duplication of extracted files. However, the timestamp of a preexisting file will be updated if the same files is extracted again, similar to the *touch* command.

Optionally a `fileinfo` record can be written to its own file sharing the same SHA256 as the file it references. To handle recording the metadata of each occurrence of an extracted file, these filenames include some extra fields to ensure uniqueness. Currently the format is:

```
<SHA256>.<SECONDS>.<ID>.json
```

where <SECONDS> is the seconds from the packet that triggered the stored file to be closed and <ID> is a unique ID for the runtime of the Suricata instance. These values should not be depended on, and are simply used to ensure uniqueness.

These `fileinfo` records are identical to the `fileinfo` records logged to the `eve` output.

See *File-store (File Extraction)* for more information on configuring the file-store output.

> **Note**
>
> This section documents version 2 of the `file-store`. Version 1 of the file-store has been removed as of Suricata version 6.

17.4 Rules

Without rules in place no extraction will happen. The simplest rule would be:

```
alert http any any -> any any (msg:"FILE store all"; filestore; sid:1; rev:1;)
```

This will simply store all files to disk.

Want to store all files with a pdf extension?

```
alert http any any -> any any (msg:"FILE PDF file claimed"; fileext:"pdf"; filestore;␣
↪sid:2; rev:1;)
```

Or rather all actual pdf files?

```
alert http any any -> any any (msg:"FILE pdf detected"; filemagic:"PDF document";␣
↪filestore; sid:3; rev:1;)
```

Or rather only store files from black list checksum md5 ?

```
alert http any any -> any any (msg:"Black list checksum match and extract MD5";␣
↪filemd5:fileextraction-chksum.list; filestore; sid:4; rev:1;)
```

Or only store files from black list checksum sha1 ?

```
alert http any any -> any any (msg:"Black list checksum match and extract SHA1";␣
↪filesha1:fileextraction-chksum.list; filestore; sid:5; rev:1;)
```

Or finally store files from black list checksum sha256 ?

```
alert http any any -> any any (msg:"Black list checksum match and extract SHA256";␣
↪filesha256:fileextraction-chksum.list; filestore; sid:6; rev:1;)
```

Bundled with the Suricata download, is a file with more example rules. In the archive, go to the *rules* directory and check the `files.rules` file.

17.5 MD5

Suricata can calculate MD5 checksums of files on the fly and log them. See *Storing MD5s checksums* for an explanation on how to enable this.

17.5.1 Storing MD5s checksums

Configuration

In the Suricata config file:

```
- file-store:
    enabled: yes       # set to yes to enable
    dir: filestore     # directory to store the files
    force-hash: [md5]  # force logging of md5 checksums
```

For JSON output:

```
outputs:
  - eve-log:
    enabled: yes
    filetype: regular #regular|syslog|unix_dgram|unix_stream|redis
    filename: eve.json
    types:
      - files:
        force-magic: no   # force logging magic on all logged files
        # force logging of checksums, available hash functions are md5,
        # sha1 and sha256
        #force-hash: [md5]
```

Other settings affecting *File Extraction*

```
stream:
  memcap: 64mb
  checksum-validation: yes      # reject wrong csums
  inline: no                    # no inline mode
  reassembly:
    memcap: 32mb
    depth: 0                     # reassemble all of a stream
    toserver-chunk-size: 2560
    toclient-chunk-size: 2560
```

Make sure we have *depth: 0* so all files can be tracked fully.

```
libhtp:
  default-config:
    personality: IDS
    # Can be specified in kb, mb, gb.  Just a number indicates
    # it's in bytes.
    request-body-limit: 0
    response-body-limit: 0
```

Make sure we have *request-body-limit: 0* and *response-body-limit: 0*

Testing

For the purpose of testing we use this rule only in a file.rules (a test/example file):

```
alert http any any -> any any (msg:"FILE store all"; filestore; sid:1; rev:1;)
```

This rule above will save all the file data for files that are opened/downloaded through HTTP

Start Suricata (`-S` option *ONLY loads* the specified rule file and disregards any other rules that are enabled in suricata.yaml):

```
suricata -c /etc/suricata/suricata.yaml -S file.rules -i eth0
```

Meta data:

```
TIME:              05/01/2012-11:09:52.425751
SRC IP:            2.23.144.170
DST IP:            192.168.1.91
```

(continues on next page)

(continued from previous page)

```
PROTO:              6
SRC PORT:           80
DST PORT:           51598
HTTP URI:           /en/US/prod/collateral/routers/ps5855/prod_brochure0900aecd8019dc1f.
↪pdf
HTTP HOST:          www.cisco.com
HTTP REFERER:       http://www.cisco.com/c/en/us/products/routers/3800-series-integrated-
↪services-routers-isr/index.html
FILENAME:           /en/US/prod/collateral/routers/ps5855/prod_brochure0900aecd8019dc1f.
↪pdf
MAGIC:              PDF document, version 1.6
STATE:              CLOSED
MD5:                59eba188e52467adc11bf2442ee5bf57
SIZE:               9485123
```

and in files-json.log (or eve.json) :

```
{ "id": 1, "timestamp": "05\/01\/2012-11:10:27.693583", "ipver": 4, "srcip": "2.23.144.
↪170", "dstip": "192.168.1.91", "protocol": 6, "sp": 80, "dp": 51598, "http_uri": "\/en\
↪/US\/prod\/collateral\/routers\/ps5855\/prod_brochure0900aecd8019dc1f.pdf", "http_host
↪": "www.cisco.com", "http_referer": "http:\/\/www.google.com\/url?sa=t&rct=j&q=&esrc=s&
↪source=web&cd=1&ved=0CDAQFjAA&url=http%3A%2F%2Fwww.cisco.com%2Fen%2FUS%2Fprod
↪%2Fcollateral%2Frouters%2Fps5855%2Fprod_brochure0900aecd8019dc1f.pdf&
↪ei=OqyfT9eoJubi4QTyiamhAw&usg=AFQjCNGdjDBpBDfQv2r3VogSH41V6T5x9Q", "filename": "\/en\/
↪US\/prod\/collateral\/routers\/ps5855\/prod_brochure0900aecd8019dc1f.pdf", "magic":
↪"PDF document, version 1.6", "state": "CLOSED", "md5":
↪"59eba188e52467adc11bf2442ee5bf57", "stored": true, "size": 9485123 }
{ "id": 12, "timestamp": "05\/01\/2012-11:12:57.421420", "ipver": 4, "srcip": "2.23.144.
↪170", "dstip": "192.168.1.91", "protocol": 6, "sp": 80, "dp": 51598, "http_uri": "\/en\
↪/US\/prod\/collateral\/routers\/ps5855\/prod_brochure0900aecd8019dc1f.pdf", "http_host
↪": "www.cisco.com", "http_referer": "http:\/\/www.google.com\/url?sa=t&rct=j&q=&esrc=s&
↪source=web&cd=1&ved=0CDAQFjAA&url=http%3A%2F%2Fwww.cisco.com%2Fen%2FUS%2Fprod
↪%2Fcollateral%2Frouters%2Fps5855%2Fprod_brochure0900aecd8019dc1f.pdf&
↪ei=OqyfT9eoJubi4QTyiamhAw&usg=AFQjCNGdjDBpBDfQv2r3VogSH41V6T5x9Q", "filename": "\/en\/
↪US\/prod\/collateral\/routers\/ps5855\/prod_brochure0900aecd8019dc1f.pdf", "magic":
↪"PDF document, version 1.6", "state": "CLOSED", "md5":
↪"59eba188e52467adc11bf2442ee5bf57", "stored": true, "size": 9485123 }
```

Log all MD5s without any rules

If you would like to log MD5s for everything and anything that passes through the traffic that you are inspecting with Suricata, but not log the files themselves, all you have to do is disable file-store and enable only the JSON output with forced MD5s - in suricata.yaml like so:

```
- file-store:
    version: 2
    enabled: no       # set to yes to enable
    log-dir: files    # directory to store the files
    force-filestore: no
    force-hash: [md5]  # force logging of md5 checksums
```

17.5.2 Public SHA1 MD5 data sets

National Software Reference Library - http://www.nsrl.nist.gov/Downloads.html

17.6 Updating Filestore Configuration

17.6.1 Update File-store v1 Configuration to V2

Given a file-store configuration like:

```
- file-store:
    enabled: yes        # set to yes to enable
    log-dir: files      # directory to store the files
    force-magic: no     # force logging magic on all stored files
    force-hash: [md5]   # force logging of md5 checksums
    force-filestore: no # force storing of all files
    stream-depth: 1mb   # reassemble 1mb into a stream, set to no to disable
    waldo: file.waldo   # waldo file to store the file_id across runs
    max-open-files: 0   # how many files to keep open (0 means none)
    write-meta: yes     # write a .meta file if set to yes
    include-pid: yes    # include the pid in filenames if set to yes.
```

The following changes will need to be made to convert to a v2 style configuration:

- The `version` field must be set to 2.
- The `log-dir` field should be renamed to `dir`. It is recommended to use a new directory instead of an existing v1 directory.
- Remove the `waldo` option. It is no longer used.
- Remove the `write-meta` option.
- Optionally set `write-fileinfo` to enable writing of a metadata file along side the extracted file. Not that this option is disabled by default as a `fileinfo` event can be written to the Eve log file.
- Remove the `include-pid` option. There is no equivalent to this option in file-store v2.

Example converted configuration:

```
- file-store:
    version: 2
    enabled: yes
    dir: filestore
    force-hash: [md5]
    file-filestore: no
    stream-depth: 1mb
    max-open-files: 0
    write-fileinfo: yes
```

Refer to the *File Extraction* section of the manual for information about the format of the file-store directory for file-store v2.

CHAPTER

EIGHTEEN

PROTOCOLS

18.1 App-Layer

18.1.1 HTTP

The HTTP protocol parser handles HTTP 0.9, 1.0 and 1.1 support.

Rule Keywords

HTTP rule keywords are documented in the rule guide *HTTP Keywords*.

In addition to these specific keywords, file transactions can be inspected with the *File Keywords*.

Transactions

Transactions in the HTTP implementation are *bidirectional*. A request and its response together form the transaction.

18.1.2 HTTP/2

HTTP/2 is generally encrypted on the wire, although it *can* be unencrypted. But it's most likely this traffic will only be seen after some form of TLS decryption.

Rule Keywords

HTTP rule keywords apply to HTTP/2 as well and are documented in the rule guide *HTTP Keywords*. HTTP/2 specific rule keywords are documented in the rule guide *HTTP2 Keywords*.

In addition to these specific keywords, file transactions can be inspected with the *File Keywords*.

18.1.3 TLS

TLS support includes SSLv2 and SSLv3.

Rule Keywords

TLS rule keywords are documented in the rule guide *SSL/TLS Keywords*.

In addition to these specific keywords, the traffic can be inspected with the *JA3/JA4 Keywords*.

Transactions

The TLS implementation uses a single *bidirectional* transaction for the entire TLS flow. It includes the TLS handshake and the handling of the encrypted portion the traffic.

18.1.4 DNS

Rule Keywords

DNS rule keywords are documented in the rule guide *DNS Keywords*.

Transactions

Transactions in the DNS implementation are *unidirectional*. A DNS request will form a transaction, and a response will form its own transaction.

18.1.5 SMB

SMB is a complex protocol with many dialects and capabilities. The parser supports SMBv1, SMBv2 and SMBv3.

Rule Keywords

SMB rule keywords are documented in the rule guide *SMB Keywords*.

In addition to these specific keywords, file transactions can be inspected with the *File Keywords*.

DCERPC over SMB traffic can be inspected using *DCERPC Keywords*.

Transactions

Transactions in the SMB implementation are *bidirectional*. There are different types:

- generic request/response pairs
- file transfer, this may include many write/read commands and their responses, including close commands
- session setup, including several related commands and their responses
- DCERPC over SMB, this may include several read/write commands to create a DCERPC transaction that has a single DCEPRC request and its matching response

18.2 Further Reading

Description of transactional rules *Transactional rules*.

More implementation details can be found in the *App-Layer* developer guide section.

CHAPTER

NINETEEN

PUBLIC DATASETS (PCAPS)

Collections of public datasets and packet captures for testing and profiling. Not to be confused with Suricata `datasets`.

MIT Lincoln Laboratory
: Cyber-focused datasets from MIT, including DARPA datasets.

MAWI Samplepoint-F
: WIDE transit-link traces (packet headers only; no payloads).

MACCDC
: Mid-Atlantic Collegiate Cyber Defense Competition traffic captures.

Netresec
: Public pcap files for network forensics and malware analysis.

Wireshark Sample Captures
: Curated sample captures useful for protocol testing.

Security Onion PCAP collection
: Collection of links and downloads for lab datasets.

Stratosphere IPS Malware Capture Facility
: Malware-oriented captures with related dataset material.

UNSW-NB15
: UNSW Cyber Range dataset with IXIA-generated benign/attack traffic, raw pcap captures, and labeled flow-feature CSV files.

CIC-IDS2017
: Intrusion Detection Evaluation Dataset

CSE-CIC-IDS2018
: CSE-CIC-IDS2018 on AWS: A collaborative project between the Communications Security Establishment (CSE) & the Canadian Institute for Cybersecurity (CIC).

CIC-DDoS2019
: Dataset containing DDoS attacks resembling true real world data.

CHAPTER

TWENTY

USING CAPTURE HARDWARE

20.1 AF_PACKET

20.1.1 Introduction

AF_PACKET is a capture interface to the Linux Kernel.

20.1.2 Config Options

enable-hwtimestamp

Boolean option to enable hardware timestamping on an interface.

By default the hardware timestamping support is disabled.

Hardware timestamping can lead to issue of the NIC and kernel getting out of sync. See ticket 7585.

```
af-packet:
  - interface: eth0
    cluster-id: 99
    enable-hwtimestamp: true
    cluster-type: cluster_flow
```

20.2 Endace DAG

Suricata comes with native Endace DAG card support. This means Suricata can use the *libdag* interface directly, instead of a libpcap wrapper (which should also work).

Steps:

Configure with DAG support:

```
./configure --enable-dag --prefix=/usr --sysconfdir=/etc --localstatedir=/var
make
sudo make install
```

Results in:

```
Suricata Configuration:
  AF_PACKET support:                       no
  PF_RING support:                         no
  NFQueue support:                         no
  IPFW support:                            no
```

(continues on next page)

(continued from previous page)

```
  DAG enabled:                              yes
  Napatech enabled:                         no
```

Start with:

```
suricata -c suricata.yaml --dag 0:0
```

Started up!

```
[5570] 10/7/2012 -- 13:52:30 - (source-erf-dag.c:262) <Info> (ReceiveErfDagThreadInit) --
↪ Attached and started stream: 0 on DAG: /dev/dag0
[5570] 10/7/2012 -- 13:52:30 - (source-erf-dag.c:288) <Info> (ReceiveErfDagThreadInit) --
↪ Starting processing packets from stream: 0 on DAG: /dev/dag0
```

20.3 Napatech

20.3.1 Contents

- Introduction
- Package Installation
- Basic Configuration
- Advanced Multithreaded Configuration

20.3.2 Introduction

Napatech packet capture accelerator cards can greatly improve the performance of your Suricata deployment using these hardware based features:

- On board burst buffering (up to 12GB)
- Zero-copy kernel bypass DMA
- Non-blocking PCIe performance
- Port merging
- Load distribution to up 128 host buffers
- Precise timestamping
- Accurate time synchronization

The package uses a proprietary shell script to handle the installation process. In either case, gcc, make and the kernel header files are required to compile the kernel module and install the software.

20.3.3 Package Installation

Note that make, gcc, and the kernel headers are required for installation

Root privileges are also required

The latest driver and tools installation package can be downloaded from: https://www.napatech.com/downloads.

Note that you will be prompted to install the Napatech libpcap library. Answer "yes" if you would like to use the Napatech card to capture packets in Wireshark, tcpdump, or another pcap based application. Libpcap is not needed for Suricata as native Napatech API support is included

Red Hat Based Distros:

```
$ yum install kernel-devel-$(uname -r) gcc make
      $ ./package_install_3gd.sh
```

Debian Based Distros:

```
$ apt-get install linux-headers-$(uname .r) gcc make
      $ ./package_install_3gd.sh
```

To complete installation for all distros `ntservice`:

```
$ /opt/napatech3/bin/ntstart.sh -m
```

20.3.4 Suricata Installation

After downloading and extracting the Suricata tarball, you need to run configure to enable Napatech support and prepare for compilation:

```
$ ./configure --enable-napatech --with-napatech-includes=/opt/napatech3/include --with-
↪napatech-libraries=/opt/napatech3/lib
$ make
$ make install-full
```

20.3.5 Suricata configuration

Now edit the suricata.yaml file to configure the system. There are three ways the system can be configured:

1. Auto-config without cpu-affinity: In this mode you specify the stream configuration in suricata.yaml file and allow the threads to roam freely. This is good for single processor systems where NUMA node configuration is not a performance concern.

2. Auto-config with cpu-affinity: In this mode you use the cpu-affinity of the worker threads to control the creation and configuration of streams. One stream and one worker thread will be created for each cpu identified in suricata.yaml. This is best in systems with multiple NUMA nodes (i.e. multi-processor systems) as the NUMA node of the host buffers is matched to the core on which the thread is running.

3. Manual-config (legacy): In this mode the underlying Napatech streams are configured by issuing NTPL commands prior to running Suricata. Suricata then connects to the existing streams on startup.

20.3.6 Example Configuration - Auto-config without cpu-affinity:

If cpu-affinity is not used it is necessary to explicitly define the streams in the Suricata configuration file. To use this option the following options should be set in the Suricata configuration file:

1. Turn off cpu-affinity
2. Enable the Napatech "auto-config" option
3. Specify the streams that should be created on startup
4. Specify the ports that will provide traffic to Suricata
5. Specify the hashmode used to distribute traffic to the streams

Below are the options to set:

```
threading:
  set-cpu-affinity: no
    .
    .
    .
napatech:
    auto-config: yes
    streams: ["0-3"]
    ports: [all]
    hashmode: hash5tuplesorted
```

Now modify `ntservice.ini`. You also need make sure that you have allocated enough host buffers in `ntservice.ini` for the streams. It's a good idea to also set the `TimeSyncReferencePriority`. To do this make the following changes to ntservice.ini:

> HostBuffersRx = [4,16,-1] # [number of host buffers, Size(MB), NUMA node] TimeSyncReferencePriority = OSTime # Timestamp clock synchronized to the OS

Stop and restart `ntservice` after making changes to ntservice:

```
$ /opt/napatech3/bin/ntstop.sh
$ /opt/napatech3/bin/ntstart.sh
```

Now you are ready to start Suricata:

```
$ suricata -c /usr/local/etc/suricata/suricata.yaml --napatech --runmode workers
```

20.3.7 Example Configuration - Auto-config with cpu-affinity:

This option will create a single worker-thread and stream for each CPU defined in the `worker-cpu-set`. To use this option make the following changes to suricata.yaml:

1. Turn on cpu-affinity
2. Specify the worker-cpu-set
3. Enable the Napatech "auto-config" option
4. Specify the ports that will provide traffic to Suricata
5. Specify the hashmode that will be used to control the distribution of traffic to the different streams/cpus.

When you are done it should look similar to this:

```
threading:
  set-cpu-affinity: yes
  cpu-affinity:
    management-cpu-set:
      cpu: [ 0 ]
    receive-cpu-set:
      cpu: [ 0 ]
    worker-cpu-set:
      cpu: [ all ]
      .
      .
      .
napatech:
```

(continues on next page)

(continued from previous page)

```
  auto-config: yes
  ports: [all]
  hashmode: hash5tuplesorted
```

Prior to running Suricata in this mode you also need to configure a sufficient number of host buffers on each NUMA node. So, for example, if you have a two processor server with 32 total cores and you plan to use all of the cores you will need to allocate 16 host buffers on each NUMA node. It is also desirable to set the Napatech cards time source to the OS.

To do this make the following changes to ntservice.ini:

```
TimeSyncReferencePriority = OSTime  # Timestamp clock synchronized to the OS
HostBuffersRx = [16,16,0],[16,16,1] # [number of host buffers, Size(MB), NUMA node]
```

Stop and restart `ntservice` after making changes to ntservice:

```
$ /opt/napatech3/bin/ntstop.sh -m
$ /opt/napatech3/bin/ntstart.sh -m
```

Now you are ready to start Suricata:

```
$ suricata -c /usr/local/etc/suricata/suricata.yaml --napatech --runmode workers
```

20.3.8 Example Configuration - Manual Configuration

For Manual Configuration the Napatech streams are created by running NTPL commands prior to running Suricata.

Note that this option is provided primarily for legacy configurations as previously this was the only way to configure Napatech products. Newer capabilities such as flow-awareness and inline processing cannot be configured manually.

In this example we will setup the Napatech capture accelerator to merge all physical ports, and then distribute the merged traffic to four streams that Suricata will ingest.

The steps for this configuration are:

1. Disable the Napatech auto-config option in suricata.yaml
2. Specify the streams that Suricata is to use in suricata.yaml
3. Create a file with NTPL commands to create the underlying Napatech streams.

First suricata.yaml should be configured similar to the following:

```
napatech:
  auto-config: no
  streams: ["0-3"]
```

Next you need to make sure you have enough host buffers defined in ntservice.ini. As it's also a good idea to set up the TimeSync. Here are the lines to change:

```
TimeSyncReferencePriority = OSTime      # Timestamp clock synchronized to the OS
HostBuffersRx = [4,16,-1]               # [number of host buffers, Size(MB), NUMA node]
```

Stop and restart ntservice after making changes to ntservice:

```
$ /opt/napatech3/bin/ntstop.sh
$ /opt/napatech3/bin/ntstart.sh
```

Now that ntservice is running we need to execute a few NTPL (Napatech Programming Language) commands to complete the setup. Create a file will the following commands:

```
Delete=All                                  # Delete any existing filters
Assign[streamid=(0..3)]= all    # Assign all physical ports to stream ID 0
```

Next execute those command using the `ntpl` tool:

```
$ /opt/napatech3/bin/ntpl -f <my_ntpl_file>
```

Now you are ready to start Suricata:

```
$ suricata -c /usr/local/etc/suricata/suricata.yaml --napatech --runmode workers
```

It is possible to specify much more elaborate configurations using this option. Simply by creating the appropriate NTPL file and attaching Suricata to the streams.

20.3.9 Bypassing Flows

On flow-aware Napatech products, traffic from individual flows can be automatically dropped or, in the case of inline configurations, forwarded by the hardware after an inspection of the initial packet(s) of the flow by Suricata. This will save CPU cycles since Suricata does not process packets for a flow that has already been adjudicated. This is enabled via the hardware-bypass option in the Napatech section of the configuration file.

When hardware bypass is used it is important that the ports accepting upstream and downstream traffic from the network are configured with information on which port the two sides of the connection will arrive. This is needed for the hardware to properly process traffic in both directions. This is indicated in the "ports" section as a hyphen separated list of port-pairs that will be receiving upstream and downstream traffic E.g.:

```
napatech:
  hardware-bypass: true
  ports[0-1,2-3]
```

Note that these "port-pairings" are also required for IDS configurations as the hardware needs to know on which port(s) two sides of the connection will arrive.

For configurations relying on optical taps the two sides of the pairing will typically be different ports. For SPAN port configurations where both upstream and downstream traffic are delivered to a single port both sides of the "port-pair" will reference the same port.

For example tap configurations have a form similar to this:

```
ports[0-1,2-3]
```

Whereas SPAN port configurations it would look similar to this:

```
ports[0-0,1-1,2-2,3-3]
```

Note that SPAN and tap configurations may be combined on the same adapter.

There are multiple ways that Suricata can be configured to bypass traffic. One way is to enable stream.bypass in the configuration file. E.g.:

```
stream:
  bypass: true
```

When enabled once Suricata has evaluated the first chunk of the stream (the size of which is also configurable) it will indicate that the rest of the packets in the flow can be bypassed. In IDS mode this means that the subsequent packets of

the flow will be dropped and not delivered to Suricata. In inline operation the packets will be transmitted on the output port but not delivered to Suricata.

Another way is by specifying the "bypass" keyword in a rule. When a rule is triggered with this keyword then the "pass" or "drop" action will be applied to subsequent packets of the flow automatically without further analysis by Suricata. For example given the rule:

```
drop tcp any 443 <> any any (msg: "SURICATA Test rule"; bypass; sid:1000001; rev:2;)
```

Once Suricata initially evaluates the fist packet(s) and identifies the flow, all subsequent packets from the flow will be dropped by the hardware; thus saving CPU cycles for more important tasks.

The timeout value for how long to wait before evicting stale flows from the hardware flow table can be specified via the FlowTimeout attribute in ntservice.ini.

20.3.10 Inline Operation

Napatech flow-aware products can be configured for inline operation. This is specified in the configuration file. When enabled, ports are specified as port-pairs. With traffic received from one port it is transmitted out the the peer port after inspection by Suricata. E.g. the configuration:

```
napatech:
 inline: enabled
 ports[0-1, 2-3]
```

Will pair ports 0 and 1; and 2 and 3 as peers. Rules can be defined to pass traffic matching a given signature. For example, given the rule:

```
pass tcp any 443 <> any any (msg: "SURICATA Test rule";  bypass; sid:1000001; rev:2;)
```

Suricata will evaluate the initial packet(s) of the flow and program the flow into the hardware. Subsequent packets from the flow will be automatically be shunted from one port to it's peer.

20.3.11 Counters

The following counters are available:

- napa_total.pkts - The total of packets received by the card.
- napa_total.byte - The total count of bytes received by the card.
- napa_total.overflow_drop_pkts - The number of packets that were dropped because the host buffers were full. (I.e. the application is not able to process packets quickly enough.)
- napa_total.overflow_drop_byte - The number of bytes that were dropped because the host buffers were full. (I.e. the application is not able to process packets quickly enough.)

On flow-aware products the following counters are also available:

- napa_dispatch_host.pkts, napa_dispatch_host.byte:

  The total number of packets/bytes that were dispatched to a host buffer for processing by Suricata. (Note: this count includes packets that may be subsequently dropped if there is no room in the host buffer.)

- napa_dispatch_drop.pkts, napa_dispatch_drop.byte:

  The total number of packets/bytes that were dropped at the hardware as a result of a Suricata "drop" bypass rule or other adjudication by Suricata that the flow packets should be dropped. These packets are not delivered to the application.

- napa_dispatch_fwd.pkts, napa_dispatch_fwd.byte:

  When inline operation is configured this is the total number of packets/bytes that were forwarded as result of a Suricata "pass" bypass rule or as a result of stream or encryption bypass being enabled in the configuration file. These packets were not delivered to the application.

- napa_bypass.active_flows:

  The number of flows actively programmed on the hardware to be forwarded or dropped.

- napa_bypass.total_flows:

  The total count of flows programmed since the application started.

If enable-stream-stats is enabled in the configuration file then, for each stream that is being processed, the following counters will be output in stats.log:

- napa<streamid>.pkts: The number of packets received by the stream.
- napa<streamid>.bytes: The total bytes received by the stream.
- napa<streamid>.drop_pkts: The number of packets dropped from this stream due to buffer overflow conditions.
- napa<streamid>.drop_byte: The number of bytes dropped from this stream due to buffer overflow conditions.

This is useful for fine-grain debugging to determine if a specific CPU core or thread is falling behind resulting in dropped packets.

Debugging:

For debugging configurations it is useful to see what traffic is flowing as well as what streams are created and receiving traffic. There are two tools in /opt/napatech3/bin that are useful for this:

- monitoring: this tool will, among other things, show what traffic is arriving at the port interfaces.
- profiling: this will show host-buffers, streams and traffic flow to the streams.

If Suricata terminates abnormally stream definitions, which are normally removed at shutdown, may remain in effect. If this happens they can be cleared by issuing the "delete=all" NTPL command as follows:

```
# /opt/napatech3/bin/ntpl -e "delete=all"
```

20.3.12 Napatech configuration options:

These are the Napatech options available in the Suricata configuration file:

```
napatech:
  # When use_all_streams is set to "yes" the initialization code will query
  # the Napatech service for all configured streams and listen on all of them.
  # When set to "no" the streams config array will be used.
  #
  # This option necessitates running the appropriate NTPL commands to create
  # the desired streams prior to running Suricata.
  #use-all-streams: no

  # The streams to listen on when auto-config is disabled or when threading
  # cpu-affinity is disabled. This can be either:
  #   an individual stream (e.g. streams: [0])
  # or
  #   a range of streams (e.g. streams: ["0-3"])
  #
```

(continues on next page)

(continued from previous page)

```
streams: ["0-3"]

# Stream stats can be enabled to provide fine grain packet and byte counters
# for each thread/stream that is configured.
#
enable-stream-stats: no

# When auto-config is enabled the streams will be created and assigned
# automatically to the NUMA node where the thread resides. If cpu-affinity
# is enabled in the threading section, then the streams will be created
# according to the number of worker threads specified in the worker cpu set.
# Otherwise, the streams array is used to define the streams.
#
# This option cannot be used simultaneous with "use-all-streams".
#
auto-config: yes

# Enable hardware level flow bypass.
#
hardware-bypass: yes

# Enable inline operation. When enabled traffic arriving on a given port is
# automatically forwarded out it's peer port after analysis by Suricata.
# hardware-bypass must be enabled when this is enabled.
#
inline: no

# Ports indicates which napatech ports are to be used in auto-config mode.
# these are the port ID's of the ports that will be merged prior to the
# traffic being distributed to the streams.
#
# When hardware-bypass is enabled the ports must be configured as a segment
# specify the port(s) on which upstream and downstream traffic will arrive.
# This information is necessary for the hardware to properly process flows.
#
# When using a tap configuration one of the ports will receive inbound traffic
# for the network and the other will receive outbound traffic. The two ports on a
# given segment must reside on the same network adapter.
#
# When using a SPAN-port configuration the upstream and downstream traffic
# arrives on a single port. This is configured by setting the two sides of the
# segment to reference the same port.  (e.g. 0-0 to configure a SPAN port on
# port 0).
#
# port segments are specified in the form:
#    ports: [0-1,2-3,4-5,6-6,7-7]
#
# For legacy systems when hardware-bypass is disabled this can be specified in any
# of the following ways:
#
#   a list of individual ports (e.g. ports: [0,1,2,3])
#
```

(continues on next page)

(continued from previous page)

```
#   a range of ports (e.g. ports: [0-3])
#
#   "all" to indicate that all ports are to be merged together
#   (e.g. ports: [all])
#
# This parameter has no effect if auto-config is disabled.
#
ports: [0-1,2-3]

# When auto-config is enabled the hashmode specifies the algorithm for
# determining to which stream a given packet is to be delivered.
# This can be any valid Napatech NTPL hashmode command.
#
# The most common hashmode commands are: hash2tuple, hash2tuplesorted,
# hash5tuple, hash5tuplesorted and roundrobin.
#
# See Napatech NTPL documentation other hashmodes and details on their use.
#
# This parameter has no effect if auto-config is disabled.
#
hashmode: hash5tuplesorted
```

Make sure that there are enough host-buffers declared in `ntservice.ini` to accommodate the number of cores/streams being used.

20.3.13 Support

Contact a support engineer at: ntsupport@napatech.com

Napatech Documentation can be found at: https://docs.napatech.com (Click the search icon, with no search text, to see all documents in the portal.)

20.4 Myricom

From: https://blog.inliniac.net/2012/07/10/suricata-on-myricom-capture-cards/

In this guide I'll describe using the Myricom libpcap support. I'm going to assume you installed the card properly, installed the Sniffer driver and made sure that all works. Make sure `dmesg` shows that the card is in sniffer mode:

```
[ 2102.860241] myri_snf INFO: eth4: Link0 is UP
[ 2101.341965] myri_snf INFO: eth5: Link0 is UP
```

I have installed the Myricom runtime and libraries in `/opt/snf`

Compile Suricata against Myricom's libpcap:

```
./configure --with-libpcap-includes=/opt/snf/include/ --with-libpcap-libraries=/opt/snf/
↪lib/ --prefix=/usr --sysconfdir=/etc --localstatedir=/var
make
sudo make install
```

Next, configure the amount of ringbuffers. I'm going to work with 8 here, as my quad core + hyper threading has 8 logical CPUs. *See below* for additional information about the buffer-size parameter.

```
pcap:
  - interface: eth5
    threads: 8
    buffer-size: 512kb
    checksum-checks: no
```

The 8 threads setting causes Suricata to create 8 reader threads for eth5. The Myricom driver makes sure each of those is attached to its own ringbuffer.

Then start Suricata as follows:

```
SNF_NUM_RINGS=8 SNF_FLAGS=0x1 suricata -c suricata.yaml -i eth5 --runmode=workers
```

If you want 16 ringbuffers, update the "threads" variable in the Suricata configuration file to *16* and start Suricata:

```
SNF_NUM_RINGS=16 SNF_FLAGS=0x1 suricata -c suricata.yaml -i eth5 --runmode=workers
```

Note that the `pcap.buffer-size` configuration setting shown above is currently ignored when using Myricom cards. The value is passed through to the `pcap_set_buffer_size` libpcap API within the Suricata source code. From Myricom support:

```
"The libpcap interface to Sniffer10G ignores the pcap_set_buffer_size() value.  The call␣
↪to snf_open() uses zero as the dataring_size which informs the Sniffer library to use␣
↪a default value or the value from the SNF_DATARING_SIZE environment variable."
```

The following pull request opened by Myricom in the libpcap project indicates that a future SNF software release could provide support for setting the SNF_DATARING_SIZE via the pcap.buffer-size yaml setting:

- https://github.com/the-tcpdump-group/libpcap/pull/435

Until then, the data ring and descriptor ring values can be explicitly set using the SNF_DATARING_SIZE and SNF_DESCRING_SIZE environment variables, respectively.

The SNF_DATARING_SIZE is the total amount of memory to be used for storing incoming packet data. This size is shared across all rings. The SNF_DESCRING_SIZE is the total amount of memory to be used for storing meta information about the packets (packet lengths, offsets, timestamps). This size is also shared across all rings.

Myricom recommends that the descriptor ring be 1/4 the size of the data ring, but the ratio can be modified based on your traffic profile. If not set explicitly, Myricom uses the following default values: SNF_DATARING_SIZE = 256MB, and SNF_DESCRING_SIZE = 64MB

Expanding on the 16 thread example above, you can start Suricata with a 16GB Data Ring and a 4GB Descriptor Ring using the following command:

```
SNF_NUM_RINGS=16 SNF_DATARING_SIZE=17179869184 SNF_DESCRING_SIZE=4294967296 SNF_
↪FLAGS=0x1 suricata -c suricata.yaml -i eth5 --runmode=workers
```

20.4.1 Debug Info

Myricom also provides a means for obtaining debug information. This can be useful for verifying your configuration and gathering additional information. Setting SNF_DEBUG_MASK=3 enables debug information, and optionally setting the SNF_DEBUG_FILENAME allows you to specify the location of the output file.

Following through with the example:

```
SNF_NUM_RINGS=16 SNF_DATARING_SIZE=17179869184 SNF_DESCRING_SIZE=4294967296 SNF_
↪FLAGS=0x1 SNF_DEBUG_MASK=3 SNF_DEBUG_FILENAME="/tmp/snf.out" suricata -c suricata.yaml␣
↪-i eth5 --runmode=workers
```

20.4.2 Additional Info

- http://www.40gbe.net/index_files/be59da7f2ab5bf0a299ab99ef441bb2e-28.html
- https://www.broadcom.com/support/knowledgebase/1211161394432/how-to-use-emulex-oneconnect-oce12000-d-adapters-with

20.5 eBPF and XDP

20.5.1 Introduction

eBPF stands for extended BPF. This is an extended version of Berkeley Packet Filter available in recent Linux kernel versions (at least kernel version 4.13 or greater).

It provides more advanced features with eBPF programs developed in C and capability to use structured data shared between kernel and userspace.

eBPF is used for three things in Suricata:

- eBPF filter: any BPF like filter can be developed. An example of filter accepting only packet for some VLANs is provided. A bypass implementation is also provided.
- eBPF load balancing: provide programmable load balancing. Simple ippair load balancing is provided.
- XDP programs: Suricata can load XDP programs. A bypass program is provided.

Bypass can be implemented in eBPF and XDP. The advantage of XDP is that the packets are dropped at the earliest stage possible. So performance is better. But bypassed packets don't reach the network so you can't use this on regular traffic but only on duplicated/sniffed traffic.

The bypass implementation relies on one of the most powerful concept of eBPF: maps. A map is a data structure shared between user space and kernel space/hardware. It allows user space and kernel space to interact, pass information. Maps are often implemented as arrays or hash tables that can contain arbitrary key, value pairs.

XDP

XDP provides another Linux native way of optimising Suricata's performance on sniffing high speed networks:

> XDP or eXpress Data Path provides a high performance, programmable network data path in the Linux kernel as part of the IO Visor Project. XDP provides bare metal packet processing at the lowest point in the software stack which makes it ideal for speed without compromising programmability. Furthermore, new functions can be implemented dynamically with the integrated fast path without kernel modification.

More info about XDP:

- IOVisor's XDP page
- Cilium's BPF and XDP reference guide

20.5.2 Requirements

You will need a kernel that supports XDP and, for the most performance improvement, a network card that support XDP in the driver.

Suricata XDP code has been tested with 4.13.10 but 4.15 or later is necessary to use all features like the CPU redirect map.

If you are using an Intel network card, you will need to stay with in tree kernel NIC drivers. The out of tree drivers do not contain the XDP support.

Having a network card with support for RSS symmetric hashing is a good point or you will have to use the XDP CPU redirect map feature.

20.5.3 Prerequisites

This guide has been confirmed on Debian/Ubuntu "LTS" Linux.

Disable irqbalance

`irqbalance` may cause issues in most setups described here, so it is recommended to deactivate it

```
systemctl stop irqbalance
systemctl disable irqbalance
```

Kernel

You need to run a kernel 4.13 or newer.

Clang and dependencies

Make sure you have `clang` (>=3.9) installed on the system

```
On Ubuntu/Debian:
sudo apt install clang

On Fedora/RHEL/CentOS:
sudo dnf install clang
```

libbpf

Suricata uses libbpf to interact with eBPF and XDP

```
On Ubuntu/Debian:
sudo apt install libbpf-dev

On Fedora/RHEL/CentOS:
sudo dnf install libbpf-devel
```

If the libbpf package is unavailable, it can be cloned from the repository

```
git clone https://github.com/libbpf/libbpf.git
```

Now, you can build and install the library

```
cd libbpf/src/
make && sudo make install

sudo make install_headers
sudo ldconfig
```

In some cases your system will not find the libbpf library that is installed under `/usr/lib64` so you may need to modify your ldconfig configuration.

20.5.4 Compile and install Suricata

To get Suricata source, you can use the usual

```
git clone https://github.com/OISF/suricata.git
cd suricata && ./scripts/bundle.sh

./autogen.sh
```

Then you need to add the eBPF flags to configure and specify the Clang compiler for building all C sources, including the eBPF programs

```
CC=clang ./configure --prefix=/usr/ --sysconfdir=/etc/ --localstatedir=/var/ \
--enable-ebpf --enable-ebpf-build

make clean && make
sudo make install-full
sudo ldconfig
sudo mkdir /usr/libexec/suricata/ebpf/
```

The `clang` compiler is needed if you want to build eBPF files as the build is done via a specific eBPF backend available only in llvm/clang suite. If you don't want to use Clang for building Suricata itself, you can still specify it separately, using the `--with-clang` parameter

```
./configure --prefix=/usr/ --sysconfdir=/etc/ --localstatedir=/var/ \
--enable-ebpf --enable-ebpf-build --with-clang=/usr/bin/clang
```

20.5.5 Setup bypass

If you plan to use eBPF or XDP for a kernel/hardware level bypass, you need to enable some of the following features:

First, enable *bypass* in the *stream* section in `suricata.yaml`

```
stream:
  bypass: true
```

This will bypass flows as soon as the stream depth will be reached.

If you want, you can also bypass encrypted flows by setting *encryption-handling* to *bypass* in the app-layer tls section

```
app-layer:
  protocols:
    tls:
      enabled: yes
      detection-ports:
        dp: 443

      encryption-handling: bypass
```

Another solution is to use a set of signatures using the `bypass` keyword to obtain a selective bypass. Suricata traffic ID defines flowbits that can be used in other signatures. For instance one could use

```
.. container:: example-rule
```

alert ip any any -> any any (msg:"bypass video"; flowbits:isset,traffic/label/video; noalert; bypass; sid:1000000; rev:1;)

alert ip any any -> any any (msg:"bypass Skype"; flowbits:isset,traffic/id/skype; noalert; bypass; sid:1000001; rev:1;)

20.5.6 Setup eBPF filter

The file *ebpf/vlan_filter.c* contains a list of VLAN id in a switch that you need to edit to get something adapted to your network. Another filter dropping packets from or to a set of IPv4 address is also available in *ebpf/filter.c*. See *Pinned maps usage* for more information.

Suricata can load as eBPF filter any eBPF code exposing a `filter` section.

Once modifications and build via `make` are complete, you can copy the resulting eBPF filter as needed

```
cp ebpf/vlan_filter.bpf /usr/libexec/suricata/ebpf/
```

Then setup the *ebpf-filter-file* variable in af-packet section in `suricata.yaml`

```
- interface: eth3
  threads: 16
  cluster-id: 97
  cluster-type: cluster_flow # choose any type suitable
  defrag: yes
  # eBPF file containing a 'filter' function that will be inserted into the
  # kernel and used as load balancing function
  ebpf-filter-file:  /usr/libexec/suricata/ebpf/vlan_filter.bpf
  ring-size: 200000
```

You can then run Suricata normally

```
/usr/bin/suricata --pidfile /var/run/suricata.pid  --af-packet=eth3 -vvv
```

20.5.7 Setup eBPF bypass

You can also use eBPF bypass. To do that load the *bypass_filter.bpf* file and update af-packet configuration in `suricata.yaml` to set bypass to *yes*

```
- interface: eth3
  threads: 16
  cluster-id: 97
  cluster-type: cluster_qm # symmetric RSS hashing is mandatory to use this mode
  # eBPF file containing a 'filter' function that will be inserted into the
  # kernel and used as packet filter function
  ebpf-filter-file:  /usr/libexec/suricata/ebpf/bypass_filter.bpf
  bypass: yes
  ring-size: 200000
```

Constraints on eBPF code to have a bypass compliant code are stronger than for regular filters. The filter must expose *flow_table_v4* and *flow_table_v6* per CPU array maps with similar definitions as the one available in *bypass_filter.c*. These two maps will be accessed and maintained by Suricata to handle the lists of flows to bypass.

If you are not using VLAN tracking (`vlan.use-for-tracking` set to *false* in suricata.yaml) then you also have to set the `VLAN_TRACKING` define to *0* in `bypass_filter.c`.

20.5.8 Setup eBPF load balancing

eBPF load balancing allows to load balance the traffic on the listening sockets With any logic implemented in the eBPF filter. The value returned by the function tagged with the `loadbalancer` section is used with a modulo on the CPU count to know in which socket the packet has to be send.

An implementation of a simple symmetric IP pair hashing function is provided in the `lb.bpf` file.

Copy the resulting eBPF filter as needed

```
cp ebpf/lb.bpf /usr/libexec/suricata/ebpf/
```

Then use `cluster_ebpf` as load balancing method in the interface section of af-packet and point the `ebpf-lb-file` variable to the `lb.bpf` file

```
- interface: eth3
  threads: 16
  cluster-id: 97
  cluster-type: cluster_ebpf
  defrag: yes
  # eBPF file containing a 'loadbalancer' function that will be inserted into the
  # kernel and used as load balancing function
  ebpf-lb-file:  /usr/libexec/suricata/ebpf/lb.bpf
  ring-size: 200000
```

20.5.9 Setup XDP bypass

XDP bypass allows Suricata to tell the kernel that packets for some flows have to be dropped via the XDP mechanism. This is an early drop that occurs before the datagram reaches the Linux kernel network stack.

Linux 4.15 or newer are recommended to use that feature. You can use it on older kernel if you set `BUILD_CPUMAP` to *0* in `ebpf/xdp_filter.c`.

Copy the resulting XDP filter as needed:

```
cp ebpf/xdp_filter.bpf /usr/libexec/suricata/ebpf/
```

Setup af-packet section/interface in `suricata.yaml`.

We will use `cluster_qm` as we have symmetric hashing on the NIC, `xdp-mode: driver` and we will also use the `/usr/libexec/suricata/ebpf/xdp_filter.bpf` (in our example TCP offloading/bypass)

```
- interface: eth3
  threads: 16
  cluster-id: 97
  cluster-type: cluster_qm # symmetric hashing is a must!
  defrag: yes
  # Xdp mode, "soft" for skb based version, "driver" for network card based
  # and "hw" for card supporting eBPF.
  xdp-mode: driver
  xdp-filter-file:  /usr/libexec/suricata/ebpf/xdp_filter.bpf
  # if the ebpf filter implements a bypass function, you can set 'bypass' to
  # yes and benefit from these feature
  bypass: yes
  ring-size: 200000
  # Uncomment the following if you are using hardware XDP with
```

(continues on next page)

(continued from previous page)

```
  # a card like Netronome (default value is yes)
  # use-percpu-hash: no
```

XDP bypass is compatible with AF_PACKET IPS mode. Packets from bypassed flows will be send directly from one card to the second card without going by the kernel network stack.

If you are using hardware XDP offload you may have to set `use-percpu-hash` to false and build and install the XDP filter file after setting `USE_PERCPU_HASH` to 0.

In the XDP filter file, you can set `ENCRYPTED_TLS_BYPASS` to 1 if you want to bypass the encrypted TLS 1.2 packets in the eBPF code. Be aware that this will mean that Suricata will be blind on packets on port 443 with the correct pattern.

If you are not using VLAN tracking (`vlan.use-for-tracking` set to false in suricata.yaml) then you also have to set the VLAN_TRACKING define to 0 in `xdp_filter.c`.

Intel NIC setup

Intel network card don't support symmetric hashing but it is possible to emulate it by using a specific hashing function.

Follow these instructions closely for desired result:

```
ifconfig eth3 down
```

Use in tree kernel drivers: XDP support is not available in Intel drivers available on Intel website.

Enable symmetric hashing

```
ifconfig eth3 down
ethtool -L eth3 combined 16 # if you have at least 16 cores
ethtool -K eth3 rxhash on
ethtool -K eth3 ntuple on
ifconfig eth3 up
./set_irq_affinity 0-15 eth3
ethtool -X eth3 hkey␣
↪6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A
↪equal 16
ethtool -x eth3
ethtool -n eth3
```

In the above setup you are free to use any recent `set_irq_affinity` script. It is available in any Intel x520/710 NIC sources driver download.

NOTE: We use a special low entropy key for the symmetric hashing. More info about the research for symmetric hashing set up

Disable any NIC offloading

Run the following command to disable offloading

```
for i in rx tx tso ufo gso gro lro tx nocache copy sg txvlan rxvlan; do
       /sbin/ethtool -K eth3 $i off 2>&1 > /dev/null;
done
```

Balance as much as you can

Try to use the network card's flow balancing as much as possible

```
for proto in tcp4 udp4 ah4 esp4 sctp4 tcp6 udp6 ah6 esp6 sctp6; do
   /sbin/ethtool -N eth3 rx-flow-hash $proto sd
done
```

This command triggers load balancing using only source and destination IPs. This may be not optimal in term of load balancing fairness but this ensures all packets of a flow will reach the same thread even in the case of IP fragmentation (where source and destination port will not be available for some fragmented packets).

The XDP CPU redirect case

If ever your hardware is not able to do a symmetric load balancing but support XDP in driver mode, you can then use the CPU redirect map support available in the *xdp_filter.bpf* and *xdp_lb.bpf* file. In this mode, the load balancing will be done by the XDP filter and each CPU will handle the whole packet treatment including the creation of the skb structure in kernel.

You will need Linux 4.15 or newer to use that feature.

To do so set the *xdp-cpu-redirect* variable in af-packet interface configuration to a set of CPUs. Then use the *cluster_cpu* as load balancing function. You will also need to set the affinity to be certain that CPU cores that have the skb assigned are used by Suricata.

Also to avoid out of order packets, you need to set the RSS queue number to 1. So if our interface is *eth3*

```
/sbin/ethtool -L eth3 combined 1
```

In case your system has more then 64 core, you need to set *CPUMAP_MAX_CPUS* to a value greater than this number in *xdp_lb.c* and *xdp_filter.c*.

A sample configuration for pure XDP load balancing could look like

```
- interface: eth3
  threads: 16
  cluster-id: 97
  cluster-type: cluster_cpu
  xdp-mode: driver
  xdp-filter-file:  /usr/libexec/suricata/ebpf/xdp_lb.bpf
  xdp-cpu-redirect: ["1-17"] # or ["all"] to load balance on all CPUs
  ring-size: 200000
```

It is possible to use *xdp_monitor* to have information about the behavior of CPU redirect. This program is available in Linux tree under the *samples/bpf* directory and will be build by the make command. Sample output is the following

```
sudo ./xdp_monitor --stats
XDP-event       CPU:to  pps          drop-pps     extra-info
XDP_REDIRECT    11      2,880,212    0            Success
XDP_REDIRECT    total   2,880,212    0            Success
XDP_REDIRECT    total   0            0            Error
cpumap-enqueue   11:0   575,954      0            5.27       bulk-average
cpumap-enqueue  sum:0   575,954      0            5.27       bulk-average
cpumap-kthread  0       575,990      0            56,409     sched
cpumap-kthread  1       576,090      0            54,897     sched
```

Start Suricata with XDP

You can now start Suricata with XDP bypass activated

```
/usr/bin/suricata -c /etc/suricata/xdp-suricata.yaml --pidfile /var/run/suricata.pid  --
↪af-packet=eth3 -vvv
```

Confirm you have the XDP filter engaged in the output (example):

```
...
...
(runmode-af-packet.c:220) <Config> (ParseAFPConfig) -- Enabling locked memory for mmap␣
↪on iface eth3
(runmode-af-packet.c:231) <Config> (ParseAFPConfig) -- Enabling tpacket v3 capture on␣
↪iface eth3
(runmode-af-packet.c:326) <Config> (ParseAFPConfig) -- Using queue based cluster mode␣
↪for AF_PACKET (iface eth3)
(runmode-af-packet.c:424) <Info> (ParseAFPConfig) -- af-packet will use '/usr/libexec/
↪suricata/ebpf/xdp_filter.bpf' as XDP filter file
(runmode-af-packet.c:429) <Config> (ParseAFPConfig) -- Using bypass kernel functionality␣
↪for AF_PACKET (iface eth3)
(runmode-af-packet.c:609) <Config> (ParseAFPConfig) -- eth3: enabling zero copy mode by␣
↪using data release call
(util-runmodes.c:296) <Info> (RunModeSetLiveCaptureWorkersForDevice) -- Going to use 8␣
↪thread(s)
...
...
```

20.5.10 Pinned maps usage

Pinned maps stay attached to the system if the creating process disappears and they can also be accessed by external tools. In Suricata bypass case, this can be used to keep bypassed flow tables active, so Suricata is not hit by previously bypassed flows when restarting. In the socket filter case, this can be used to maintain a map from tools outside of Suricata.

To use pinned maps, you first have to mount the *bpf* pseudo filesystem

```
sudo mount -t bpf none /sys/fs/bpf
```

You can also add to your */etc/fstab*

```
bpffs                     /sys/fs/bpf            bpf     defaults 0 0
```

and run *sudo mount -a*.

Pinned maps will be accessible as file from the */sys/fs/bpf* directory. Suricata will pin them under the name *suricata-$IFACE_NAME-$MAP_NAME*.

To activate pinned maps for a interface, set *pinned-maps* to *true* in the *af-packet* configuration of this interface

```
- interface: eth3
  pinned-maps: true
```

20.5.11 XDP and pinned-maps

This option can be used to expose the maps of a socket filter to other processes. This allows for example, the external handling of a accept list or block list of IP addresses. See bpfctrl for an example of external list handling.

In the case of XDP, the eBPF filter is attached to the interface so if you activate *pinned-maps* the eBPF will remain attached to the interface and the maps will remain accessible upon Suricata start. If XDP bypass is activated, Suricata will try at start to open the pinned maps *flow_v4_table* and *flow_v6_table*. If they are present, this means the XDP filter is still there and Suricata will just use them instead of attaching the XDP file to the interface.

So if you want to reload the XDP filter, you need to remove the files from */sys/fs/bpf/* before starting Suricata.

In case, you are not using bypass, this means that the used maps are managed from outside Suricata. As their names are not known by Suricata, you need to specify a name of a map to look for, that will be used to check for the presence of the XDP filter

```
- interface: eth3
  pinned-maps: true
  pinned-maps-name: ipv4_drop
  xdp-filter-file: /usr/libexec/suricata/ebpf/xdp_filter.bpf
```

If XDP bypass is used in IPS mode stopping Suricata will trigger an interruption in the traffic. To fix that, the provided XDP filter *xdp_filter.bpf* is containing a map that will trigger a global bypass if set to 1. You need to use *pinned-maps* to benefit from this feature.

To use it you need to set *#define USE_GLOBAL_BYPASS 1* (instead of 0) in the *xdp_filter.c* file and rebuild the eBPF code and install the eBPF file in the correct place. If you write *1* as key *0* then the XDP filter will switch to global bypass mode. Set key *0* to value *0* to send traffic to Suricata.

The switch must be activated on all sniffing interfaces. For an interface named *eth0* the global switch map will be */sys/fs/bpf/suricata-eth0-global_bypass*.

Pinned maps and eBPF filter

Pinned maps can also be used with regular eBPF filters. The main difference is that the map will not persist after Suricata is stopped because it is attached to a socket and not an interface which is persistent.

The eBPF filter *filter.bpf* uses a *ipv4_drop* map that contains the set of IPv4 addresses to drop. If *pinned-maps* is set to *true* in the interface configuration then the map will be pinned under */sys/fs/bpf/suricata-eth3-ipv4_drop*.

You can then use a tool like *bpfctrl* to manage the IPv4 addresses in the map.

20.5.12 Hardware bypass with Netronome

Netronome cards support hardware bypass. In this case the eBPF code is running in the card itself. This introduces some architectural differences compared to driver mode and the configuration and eBPF filter need to be updated.

On eBPF side, as of Linux 4.19 CPU maps and interfaces redirect are not supported and these features need to be disabled. By architecture, per CPU hash should not be used and has to be disabled. To achieve this, edit the beginning of *ebpf/xdp_filter.c* and do

```
#define BUILD_CPUMAP        0
/* Increase CPUMAP_MAX_CPUS if ever you have more than 64 CPUs */
#define CPUMAP_MAX_CPUS     64

#define USE_PERCPU_HASH    0
#define GOT_TX_PEER    0
```

Then build the bpf file with *make* and install it in the expected place.

The Suricata configuration is rather simple as you need to activate hardware mode and the *use-percpu-hash* option in the *af-packet* configuration of the interface

```
xdp-mode: hw
use-percpu-hash: no
```

The load balancing will be done on IP pairs inside the eBPF code, so using *cluster_qm* as cluster type is a good idea

```
cluster-type: cluster_qm
```

As of Linux 4.19, the number of threads must be a power of 2. So set *threads* variable of the *af-packet* interface to a power of 2 and in the eBPF filter set the following variable accordingly

```
#define RSS_QUEUE_NUMBERS   32
```

20.5.13 Getting live info about bypass

You can get information about bypass via the stats event and through the unix socket. `iface-stat` will return the number of bypassed packets (adding packets for a flow when it timeout)

```
suricatasc -c "iface-stat enp94s0np0" | jq
{
  "message": {
    "pkts": 56529854964,
    "drop": 932328611,
    "bypassed": 1569467248,
    "invalid-checksums": 0
  },
  "return": "OK"
}
```

`iface-bypassed-stats` command will return the number of elements in IPv4 and IPv6 flow tables for each interface

```
# suricatasc
>>> iface-bypassed-stats
Success:
{
    "enp94s0np0": {
       "ipv4_fail": 0,
       "ipv4_maps_count": 2303,
       "ipv4_success": 4232,
       "ipv6_fail": 0,
       "ipv6_maps_count": 13131,
       "ipv6_success": 13500

    }
}
```

The stats entry also contains a *stats.flow_bypassed* object that has local and capture bytes and packets counters as well as a bypassed and closed flow counter

```
{
  "local_pkts": 0,
  "local_bytes": 0,
```

(continues on next page)

(continued from previous page)

```
  "local_capture_pkts": 20,
  "local_capture_bytes": 25000,
  "closed": 84,
  "pkts": 4799,
  "bytes": 2975133
}
```

local_pkts and *local_bytes* are for Suricata bypassed flows. This can be because local bypass is used or because the capture method can not bypass more flows. *pkts* and *bytes* are counters coming from the capture method. They can take some time to appear due to the accounting at timeout. *local_capture_pkts* and *local_capture_bytes* are counters for packets that are seen by Suricata before the capture method efficiently bypass the traffic. There is almost always some for each flow because of the buffer in front of Suricata reading threads.

20.6 Netmap

Netmap is a high speed capture framework for Linux and FreeBSD. In Linux it is available as an external module, while in FreeBSD 11+ it is available by default.

20.6.1 Compiling Suricata

FreeBSD

On FreeBSD 11 and up, NETMAP is included and enabled by default in the kernel.

To build Suricata with NETMAP, add `--enable-netmap` to the configure line. The location of the NETMAP includes (/usr/src/sys/net/) does not have to be specified.

Linux

On Linux, NETMAP is not included by default. It can be pulled from github. Follow the instructions on installation included in the NETMAP repository.

When NETMAP is installed, add `--enable-netmap` to the configure line. If the includes are not added to a standard location, the location can be specified when configuring Suricata.

Example:

```
./configure --enable-netmap --with-netmap-includes=/usr/local/include/netmap/
```

20.6.2 Starting Suricata

When opening an interface, netmap can take various special characters as options in the interface string.

> **Warning**
>
> the interface that netmap reads from will become unavailable for normal network operations. You can lock yourself out of your system.

IDS

Suricata can be started in 2 ways to use netmap:

```
suricata --netmap=<interface>
suricata --netmap=igb0
```

In the above example Suricata will start reading from the *igb0* network interface. The number of threads created depends on the number of RSS queues available on the NIC.

```
suricata --netmap
```

In the above example Suricata will take the `netmap` block from the Suricata configuration and open each of the interfaces listed.

```
netmap:
  - interface: igb0
    threads: 2
  - interface: igb1
    threads: 4
```

For the above configuration, both `igb0` and `igb1` would be opened. With 2 threads for `igb0` and 4 capture threads for `igb1`.

Warning

This multi threaded setup only works correctly if the NIC has symmetric RSS hashing. If this is not the case, consider using the 'lb' method below.

IPS

Suricata's Netmap based IPS mode is based on the concept of creating a layer 2 software bridge between 2 interfaces. Suricata reads packets on one interface and transmits them on another.

Packets that are blocked by the IPS policy, are simply not transmitted.

```
netmap:
  - interface: igb0
    copy-mode: ips
    copy-iface: igb1
  - interface: igb1
    copy-mode: ips
    copy-iface: igb0
```

20.6.3 Advanced setups

20.6.4 lb (load balance)

"lb" is a tool written by Seth Hall to allow for load balancing for single or multiple tools. One common use case is being able to run Suricata and Zeek together on the same traffic.

starting lb:

```
lb -i eth0 -p suricata:6 -p zeek:6
```

> **Note**
>
> On FreeBSD 11, the named prefix doesn't work.

yaml:

```
netmap:
  - interface: netmap:suricata
    threads: 6
```

startup:

```
suricata --netmap=netmap:suricata
```

The interface name as passed to Suricata includes a 'netmap:' prefix. This tells Suricata that it's going to read from netmap pipes instead of a real interface.

Then Zeek (formerly Bro) can be configured to load 6 instances. Both will get a copy of the same traffic. The number of netmap pipes does not have to be equal for both tools.

FreeBSD 11

On FreeBSD 11 the named pipe is not available.

starting lb:

```
lb -i eth0 -p 6
```

yaml:

```
netmap:
  - interface: netmap:eth0
    threads: 6
```

startup:

```
suricata --netmap
```

> **Note**
>
> "lb" is bundled with netmap.

Single NIC

When an interface enters NETMAP mode, it is no longer available to the OS for other operations. This can be undesirable in certain cases, but there is a workaround.

By running Suricata in a special inline mode, the interface will show it's traffic to the OS.

```
netmap:
  - interface: igb0
    copy-mode: tap
    copy-iface: igb0^
  - interface: igb0^
```

(continues on next page)

(continued from previous page)

```
    copy-mode: tap
    copy-iface: igb0
```

The copy-mode can be both 'tap' and 'ips', where the former never drops packets based on the policies in use, and the latter may drop packets.

Warning

Misconfiguration can lead to connectivity loss. Use with care.

Note

This set up can also be used to mix NETMAP with firewall setups like pf or ipfw.

VALE switches

VALE is a virtual switch that can be used to create an all virtual network or a mix of virtual and real nics.

A simple all virtual setup:

```
vale-ctl -n vi0
vale-ctl -a vale0:vi0
vale-ctl -n vi1
vale-ctl -a vale0:vi1
```

We now have a virtual switch "vale0" with 2 ports "vi0" and "vi1".

We can start Suricata to listen on one of the ports:

```
suricata --netmap=vale0:vi1
```

Then we can

20.6.5 Inline IDS

The inline IDS is almost the same as the IPS setup above, but it will not enforce `drop` policies.

```
netmap:
  - interface: igb0
    copy-mode: tap
    copy-iface: igb1
  - interface: igb1
    copy-mode: tap
    copy-iface: igb0
```

The only difference with the IPS mode is that the `copy-mode` setting is set to `tap`.

20.7 AF_XDP

AF_XDP (eXpress Data Path) is a high speed capture framework for Linux that was introduced in Linux v4.18. AF_XDP aims at improving capture performance by redirecting ingress frames to user-space memory rings, thus bypassing the network stack.

Note that during `af_xdp` operation the selected interface cannot be used for regular network usage.

Further reading:

- https://www.kernel.org/doc/html/latest/networking/af_xdp.html

20.7.1 Compiling Suricata

Linux

libxdp and libpbf are required for this feature. When building from source the development files will also be required.

Example:

```
dnf -y install libxdp-devel libbpf-devel
```

This feature is enabled provided the libraries above are installed, the user does not need to add any additional command line options.

The command line option `--disable-af-xdp` can be used to disable this feature.

Example:

```
./configure --disable-af-xdp
```

20.7.2 Starting Suricata

IDS

Suricata can be started as follows to use af-xdp:

```
af-xdp:
  suricata --af-xdp=<interface>
  suricata --af-xdp=igb0
```

In the above example Suricata will start reading from the *igb0* network interface.

20.7.3 AF_XDP Configuration

Each of these settings can be configured under `af-xdp` within the "Configure common capture settings" section of suricata.yaml configuration file.

The number of threads created can be configured in the suricata.yaml configuration file. It is recommended to use threads equal to NIC queues/CPU cores.

Another option is to select `auto` which will allow Suricata to configure the number of threads based on the number of RSS queues available on the NIC.

With `auto` selected, Suricata spawns receive threads equal to the number of configured RSS queues on the interface.

```
af-xdp:
  threads: <number>
```

(continues on next page)

(continued from previous page)

```
  threads: auto
  threads: 8
```

20.7.4 Advanced setup

af-xdp capture source will operate using the default configuration settings. However, these settings are available in the suricata.yaml configuration file.

Available configuration options are:

force-xdp-mode

There are two operating modes employed when loading the XDP program, these are:

- XDP_DRV: Mode chosen when the driver supports AF_XDP
- XDP_SKB: Mode chosen when no AF_XDP support is unavailable

XDP_DRV mode is the preferred mode, used to ensure best performance.

```
af-xdp:
  force-xdp-mode: <value> where: value = <skb|drv|none>
  force-xdp-mode: drv
```

force-bind-mode

During binding the kernel will first attempt to use zero-copy (preferred). If zero-copy support is unavailable it will fallback to copy mode, copying all packets out to user space.

```
af-xdp:
  force-bind-mode: <value> where: value = <copy|zero|none>
  force-bind-mode: zero
```

For both options, the kernel will attempt the 'preferred' option first and fallback upon failure. Therefore the default (none) means the kernel has control of which option to apply. By configuring these options the user is forcing said option. Note that if enabled, the bind will only attempt this option, upon failure the bind will fail i.e. no fallback.

mem-unaligned

AF_XDP can operate in two memory alignment modes, these are:

- Aligned chunk mode
- Unaligned chunk mode

Aligned chunk mode is the default option which ensures alignment of the data within the UMEM.

Unaligned chunk mode uses hugepages for the UMEM. Hugepages start at the size of 2MB but they can be as large as 1GB. Lower count of pages (memory chunks) allows faster lookup of page entries. The hugepages need to be allocated on the NUMA node where the NIC and CPU resides. Otherwise, if the hugepages are allocated only on NUMA node 0 and the NIC is connected to NUMA node 1, then the application will fail to start. Therefore, it is recommended to first find out to which NUMA node the NIC is connected to and only then allocate hugepages and set CPU cores affinity to the given NUMA node.

Memory assigned per socket/thread is 16MB, so each worker thread requires at least 16MB of free space. As stated above hugepages can be of various sizes, consult the OS to confirm with `cat /proc/meminfo`.

Example

```
8 worker threads * 16Mb = 128Mb
hugepages = 2048 kB
so: pages required = 62.5 (63) pages
```

See https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt for detailed description.

To enable unaligned chunk mode:

```
af-xdp:
  mem-unaligned: <yes/no>
  mem-unaligned: yes
```

Introduced from Linux v5.11 a SO_PREFER_BUSY_POLL option has been added to AF_XDP that allows a true polling of the socket queues. This feature has been introduced to reduce context switching and improve CPU reaction time during traffic reception.

Enabled by default, this feature will apply the following options, unless disabled (see below). The following options are used to configure this feature.

enable-busy-poll

Enables or disables busy polling.

```
af-xdp:
  enable-busy-poll: <yes/no>
  enable-busy-poll: yes
```

busy-poll-time

Sets the approximate time in microseconds to busy poll on a `blocking receive` when there is no data.

```
af-xdp:
  busy-poll-time: <time>
  busy-poll-time: 20
```

busy-poll-budget

Budget allowed for batching of ingress frames. Larger values means more frames can be stored/read. It is recommended to test this for performance.

```
af-xdp:
  busy-poll-budget: <budget>
  busy-poll-budget: 64
```

Linux tunables

The SO_PREFER_BUSY_POLL option works in concert with the following two Linux knobs to ensure best capture performance. These are not socket options:

- gro-flush-timeout
- napi-defer-hard-irq

The purpose of these two knobs is to defer interrupts and to allow the NAPI context to be scheduled from a watchdog timer instead.

The `gro-flush-timeout` indicates the timeout period for the watchdog timer. When no traffic is received for `gro-flush-timeout` the timer will exit and softirq handling will resume.

The `napi-defer-hard-irq` indicates the number of queue scan attempts before exiting to interrupt context. When enabled, the softirq NAPI context will exit early, allowing busy polling.

```
af-xdp:
  gro-flush-timeout: 2000000
  napi-defer-hard-irq: 2
```

20.7.5 Hardware setup

Intel NIC setup

Intel network cards don't support symmetric hashing but it is possible to emulate it by using a specific hashing function.

Follow these instructions closely for desired result:

```
ifconfig eth3 down
```

Enable symmetric hashing

```
ifconfig eth3 down
ethtool -L eth3 combined 16 # if you have at least 16 cores
ethtool -K eth3 rxhash on
ethtool -K eth3 ntuple on
ifconfig eth3 up
./set_irq_affinity 0-15 eth3
ethtool -X eth3 hkey␣
↪6D:5A:6D:5A:6D:5A:
```

```
for proto in tcp4 udp4 ah4 esp4 sctp4 tcp6 udp6 ah6 esp6 sctp6; do
   /sbin/ethtool -N eth3 rx-flow-hash $proto sd
done
```

This command triggers load balancing using only source and destination IPs. This may be not optimal in terms of load balancing fairness but this ensures all packets of a flow will reach the same thread even in the case of IP fragmentation (where source and destination port will not be available for some fragmented packets).

20.8 DPDK

20.8.1 Introduction

The Data Plane Development Kit (DPDK) is a set of libraries and drivers that enhance and speed up packet processing in the data plane. Its primary use is to provide faster packet processing by bypassing the kernel network stack, which can provide significant performance improvements. For detailed instructions on how to setup DPDK, please refer to *Suricata.yaml* to learn more about the basic setup for DPDK. The following sections contain examples of how to set up DPDK and Suricata for more obscure use-cases.

20.8.2 Hugepage analysis

Suricata can analyse utilized hugepages on the system. This can be particularly beneficial when there's a potential overallocation of hugepages. The hugepage analysis is designed to examine the hugepages in use and provide recommendations on an adequate number of hugepages. This then ensures Suricata operates optimally while leaving sufficient memory for other applications on the system. The analysis works by comparing snapshots of the hugepages before and after Suricata is initialized. After the initialization, no more hugepages are allocated by Suricata. The hugepage analysis can be seen in the Perf log level and is printed out during the Suricata start. It is only printed when Suricata detects some disrepancies in the system related to hugepage allocation.

It's recommended to perform this analysis from a "clean" state - that is a state when all your hugepages are free. It is especially recommended when no other hugepage-dependent applications are running on your system. This can be checked in one of two ways:

```
# global check
cat /proc/meminfo

HugePages_Total:    1024
HugePages_Free:     1024

# per-numa check depends on NUMA node ID, hugepage size,
# and nr_hugepages/free_hugepages - e.g.:
cat /sys/devices/system/node/node0/hugepages/hugepages-2048kB/free_hugepages
```

After the termination of Suricata and other hugepage-related applications, if the count of free hugepages is not equal with the total number of hugepages, it indicates some hugepages were not freed completely. This can be fixed by removing DPDK-related files from the hugepage-mounted directory (filesystem). It's important to exercise caution while removing hugepages, especially when other hugepage-dependent applications are in operation, as this action will disrupt their memory functionality. Removing the DPDK files from the hugepage directory can often be done as:

```
sudo rm -rf /dev/hugepages/rtemap_*

# To check where hugepages are mounted:
dpdk-hugepages.py -s
# or
mount | grep huge
```

20.8.3 Bond interface

Link Bonding Poll Mode Driver (Bond PMD), is a software mechanism provided by the Data Plane Development Kit (DPDK) for aggregating multiple physical network interfaces into a single logical interface. Bonding can be e.g. used to:

- deliver bidirectional flows of tapped interfaces to the same worker,
- establish redundancy by monitoring multiple links,
- improve network performance by load-balancing traffic across multiple links.

Bond PMD is essentially a virtual driver that manipulates with multiple physical network interfaces. It can operate in multiple modes as described in the DPDK docs The individual bonding modes can accustom user needs. DPDK Bond PMD has a requirement that the aggregated interfaces must be the same device types - e.g. both physical ports run on mlx5 PMD. Bond PMD supports multiple queues and therefore can work in workers runmode. It should have no effect on traffic distribution of the individual ports and flows should be distributed by physical ports according to the RSS configuration the same way as if they would be configured independently.

As an example of Bond PMD, we can setup Suricata to monitor 2 interfaces that receive TAP traffic from optical interfaces. This means that Suricata receive one direction of the communication on one interface and the other direction is received on the other interface.

```
...
dpdk:
  eal-params:
    proc-type: primary
    vdev: 'net_bonding0,mode=0,slave=0000:04:00.0,slave=0000:04:00.1'

  # DPDK capture support
  # RX queues (and TX queues in IPS mode) are assigned to cores in 1:1 ratio
  interfaces:
    - interface: net_bonding0 # PCIe address of the NIC port
      # Threading: possible values are either "auto" or number of threads
      # - auto takes all cores
      # in IPS mode it is required to specify the number of cores and the
      # numbers on both interfaces must match
      threads: 4
...
```

In the DPDK part of suricata.yaml we have added a new parameter to the eal-params section for virtual devices - *vdev*. DPDK Environment Abstraction Layer (EAL) can initialize some virtual devices during the initialization of EAL. In this case, EAL creates a new device of type *net_bonding*. Suffix of *net_bonding* signifies the name of the interface (in this case the zero). Extra arguments are passed after the device name, such as the bonding mode (*mode=0*). This is the round-robin mode as is described in the DPDK documentation of Bond PMD. Members (slaves) of the *net_bonding0* interface are appended after the bonding mode parameter.

When the device is specified within EAL parameters, it can be used within Suricata *interfaces* list. Note that the list doesn't contain PCIe addresses of the physical ports but instead the *net_bonding0* interface. Threading section is also adjusted according to the items in the interfaces list by enablign set-cpu-affinity and listing CPUs that should be used in management and worker CPU set.

```
...
threading:
  set-cpu-affinity: yes
  cpu-affinity:
    management-cpu-set:
```

(continues on next page)

(continued from previous page)

```
      cpu: [ 0 ]  # include only these CPUs in affinity settings
    receive-cpu-set:
      cpu: [ 0 ]  # include only these CPUs in affinity settings
    worker-cpu-set:
      cpu: [ 2,4,6,8 ]
...
```

20.8.4 Interrupt (power-saving) mode

The DPDK is traditionally recognized for its polling mode operation. In this mode, CPU cores are continuously querying for packets from the Network Interface Card (NIC). While this approach offers benefits like reduced latency and improved performance, it might not be the most efficient in scenarios with sporadic or low traffic. The constant polling can lead to unnecessary CPU consumption. To address this, DPDK offers an *interrupt* mode.

The obvious advantage that interrupt mode brings is power efficiency. So far in our tests, we haven't observed a decrease in performance. Suricata's performance has actually seen a slight improvement. The (IPS runmode) users should be aware that interrupts can introduce non-deterministic latency. However, the latency should never be higher than in other (e.g. AF_PACKET/AF_XDP/...) capture methods.

Interrupt mode in DPDK can be configured on a per-interface basis. This allows for a hybrid setup where some workers operate in polling mode, while others utilize the interrupt mode. The configuration for the interrupt mode can be found and modified in the DPDK section of the suricata.yaml file.

Below is a sample configuration that demonstrates how to enable the interrupt mode for a specific interface:

```
...
dpdk:
    eal-params:
      proc-type: primary

    interfaces:
      - interface: 0000:3b:00.0
        interrupt-mode: true
        threads: 4
```

20.8.5 Automatic interface configuration

A number of interface properties can be manually configured. However, Suricata can automatically configure the interface properties based on the NIC capabilities. This can be done by setting `auto` to `mempool-size`, `mempool-cache-size`, `rx-descriptors`, and `tx-descriptors` interface node properties. This will allow Suricata to automatically set the sizes of individual properties according to the best-effort calculation based on the NIC capabilities. For example, receive (RX) descriptors are calculated based on the maximal "power of 2" that is lower or equal to the number of descriptors supported by the NIC. Number of TX descriptors depends on the configured `copy-mode`. IDS (none) mode uses no TX descriptors and does not create any TX queues by default. IPS and TAP mode uses the same number of TX descriptors as RX descriptors. The number of mempool and its cache is then derived from the count of descriptors.

Rx (and Tx) descriptors are set to the highest possible value to allow more buffer room when traffic spikes occur. However, it requires more memory. Individual properties can still be set manually if needed.

Note

Mellanox ConnectX-4 NICs may not support auto-configuration of `RX /TX descriptors`. Instead it can be set to a fixed value (e.g. 16384).

20.8.6 Link State Change timeout

The *linkup-timeout* YAML configuration option allows the user to set a timeout period to wait until the interface's link is detected. This ensures that Suricata does not start processing packets until the link is up. This option is particularly useful for Intel E810 (Ice) NICs, which begin receiving packets only after a few seconds have passed since the interface started. In such cases, if this check is disabled, Suricata reports as started but only begins processing packets after a few seconds. This issue has not been observed with other cards.

Setting the value to 0 causes Suricata to skip the link check. If the interface's link remains down after the timeout period, Suricata warns the user but continues with the engine initialization.

20.8.7 Encapsulation stripping

Suricata supports stripping the hardware-offloaded encapsulation stripping on the supported NICs. Currently, VLAN encapsulation stripping is supported. VLAN encapsulation stripping can be enabled with *vlan-strip-offload*.

20.9 PCAP File Reading

Suricata offers a `pcap-file` capture method to process PCAP files and directories of PCAP files in an offline or live-feed manner.

20.9.1 Configuration

```
pcap-file:
  checksum-checks: auto
  # buffer-size: 128 KiB
  # tenant-id: none
  # Applies to file and directory. Options: false (no deletion), true (always delete),
  #   "non-alerts" (delete only files with no alerts)
  # delete-when-done: false
  # recursive: false
  # continuous: false
  # delay: 30
  # poll-interval: 5
```

20.9.2 Buffer Size

This option specifies the size of the read buffer for the PCAP file. The larger the buffer, the more data Suricata can read at once. This can improve performance, especially for large files. The size can be specified through the command line option, see *--pcap-file-buffer-size*

Setting `buffer-size` to `0` disables `setvbuf` buffering. This is the explicit opt-out for non-seekable sources such as `/dev/stdin` or named pipes, where buffering the underlying file descriptor is not supported.

20.9.3 Directory-related options

The **recursive** option enables Suricata to traverse subdirectories within the specified directory, up to a maximum depth of 255. This allows for processing of PCAP files located in nested folders. Note that the recursive option cannot be used together with the `continuous` option. The command-line option is *--pcap-file-recursive*.

The **continuous** option allows Suricata to monitor the specified directory for new files, processing them as they appear. This is useful for live environments where new PCAP files are continuously added. The continuous option cannot be combined with the `recursive` option. The command-line option is *--pcap-file-continuous*..

The **delay** option specifies the amount of time, in seconds, that Suricata waits before processing newly detected files. This helps prevent the processing of incomplete files that are still being written. The delay option is applicable with the `continuous` mode.

The **poll-interval** option determines how frequently, in seconds, Suricata checks the directory for new files. Adjusting this interval can help balance responsiveness and resource usage.

Note

`continuous` and `recursive` cannot be enabled simultaneously.

Note

Symlinks are ignored during recursive traversal.

20.9.4 Other options

checksum-checks

- **auto** (default): Suricata detects checksum offloading statistically.
- **yes**: Forces checksum validation.
- **no**: Disables checksum validation.
- The command-line option is *-k*

tenant-id

- Specifies the tenant for multi-tenant setups with direct select.
- The PCAP is processed by the detection engine assigned to the specified tenant.

delete-when-done

Controls when PCAP files are deleted after processing. Three values are supported:

- `false` (default): Files are never deleted
- `true`: Files are always deleted after processing
- `"non-alerts"`: Files are deleted only if they didn't generate any alerts

Note

The command-line option *--pcap-file-delete* overrides this configuration and forces "always delete" mode (`true`).

> **Warning**
>
> When using "`non-alerts`" mode, file deletion is deferred until thread cleanup to ensure alert counts are finalized. This may delay deletion compared to other modes.

BPF filter

- Suricata supports BPF filters for packet capture that is also applicable to the `pcap-file` capture method.
- The BPF filter is specified in the file with the *-F* command-line option.

CHAPTER

TWENTYONE

INTERACTING VIA UNIX SOCKET

21.1 Introduction

Suricata can listen to a unix socket and accept commands from the user. The exchange protocol is JSON-based and the format of the message is generic.

An application called `suricatasc` is provided and installed automatically when installing/updating Suricata.

The unix socket is always enabled by default.

The creation of the socket is managed by setting enabled to 'yes' or 'auto' under unix-command in Suricata YAML configuration file:

```
unix-command:
  enabled: yes
  #filename: custom.socket # use this to specify an alternate file
```

The `filename` variable can be used to set an alternate socket filename. The filename is always relative to the local state base directory.

Clients are implemented for some programming languages and can be used as code example to write custom scripts:

- Rust: https://github.com/OISF/suricata/blob/main/rust/suricatasc (version provided in Suricata 8+)
- Python: https://github.com/OISF/suricata/blob/main-7.0.x/python/suricata/sc/suricatasc.py (Python version from older versions of Suricata)
- Perl: https://github.com/aflab/suricatac (a simple Perl client with interactive mode)
- C: https://github.com/regit/SuricataC (a Unix socket mode client in C without interactive mode)

21.2 Commands in standard running mode

The `suricatasc` command should automatically be installed in the same directory as the main `suricata` program.

The set of existing commands is the following:

- command-list: list available commands
- shutdown: shutdown Suricata
- iface-list: list interfaces where Suricata is sniffing packets
- iface-stat: list statistics for an interface
- help: alias of command-list
- version: display Suricata's version

- uptime: display Suricata's uptime
- running-mode: display running mode (workers, autofp, simple)
- capture-mode: display capture system used
- conf-get: get configuration item (see example below)
- dump-counters: dump Suricata's performance counters
- reopen-log-files: reopen log files (to be run after external log rotation)
- ruleset-reload-rules: reload ruleset and wait for completion
- ruleset-reload-nonblocking: reload ruleset and proceed without waiting
- ruleset-reload-time: return time of last reload
- ruleset-stats: display the number of rules loaded and failed
- ruleset-failed-rules: display the list of failed rules
- memcap-set: update memcap value of the specified item
- memcap-show: show memcap value of the specified item
- memcap-list: list all memcap values available
- reload-rules: alias of ruleset-reload-rules
- register-tenant-handler: register a tenant handler with the specified mapping
- unregister-tenant-handler: unregister a tenant handler with the specified mapping
- register-tenant: register tenant with a particular ID and filename
- unregister-tenant: unregister tenant with a particular ID
- reload-tenant: reload a tenant with specified ID and filename
- add-hostbit: add hostbit on a host IP with a particular bit name and time of expiry
- remove-hostbit: remove hostbit on a host IP with specified bit name
- list-hostbit: list hostbit for a particular host IP
- get-flow-stats-by-id: list information for a specific `flow_id`

A typical session with `suricatasc` looks like:

```
# suricatasc
Command list: shutdown, command-list, help, version, uptime, running-mode, capture-mode,␣
↪conf-get, dump-counters, iface-stat, iface-list, quit
>>> iface-list
Success: {'count': 2, 'ifaces': ['eth0', 'eth1']}
>>> iface-stat eth0
Success: {'pkts': 378, 'drop': 0, 'invalid-checksums': 0}
>>> conf-get unix-command.enabled
Success:
"yes"
```

21.3 Commands on the cmd prompt

You can use `suricatasc` directly on the command prompt:

```
root@debian64:~# suricatasc -c version
{'message': '5.0.3 RELEASE', 'return': 'OK'}
root@debian64:~#
root@debian64:~# suricatasc -c uptime
{'message': 35264, 'return': 'OK'}
root@debian64:~#
```

NOTE: You need to quote commands with more than one argument:

```
root@debian64:~# suricatasc -c "iface-stat eth0"
{'message': {'pkts': 5110429, 'drop': 0, 'invalid-checksums': 0}, 'return': 'OK'}
root@debian64:~#
```

21.4 PCAP processing mode

This mode is one of main motivations behind this code. The idea is to be able to provide different pcap files to Suricata without having to restart Suricata for each file. This saves time since you don't need to wait for the signature engine to initialize.

To use this mode, start Suricata with your preferred configuration YAML file and provide the option `--unix-socket` as argument:

```
suricata -c /etc/suricata-full-sigs.yaml --unix-socket
```

It is also possible to specify the socket filename as an argument:

```
suricata --unix-socket=custom.socket
```

In this last case, you will need to provide the complete path to the socket to `suricatasc`. To do so, you need to pass the filename as first argument of `suricatasc`:

```
suricatasc custom.socket
```

Once Suricata is started, you can use `suricatasc` to connect to the command socket and provide different pcap files:

```
root@tiger:~# suricatasc
>>> pcap-file /home/benches/file1.pcap /tmp/file1
Success: Successfully added file to list
>>> pcap-file /home/benches/file2.pcap /tmp/file2
Success: Successfully added file to list
>>> pcap-file-continuous /home/pcaps /tmp/dirout
Success: Successfully added file to list
```

You can add multiple files without waiting for each to be processed; they will be sequentially processed and the generated log/alert files will be put into the directory specified as second argument of the pcap-file command. You need to provide an absolute path to the files and directory as Suricata doesn't know from where the script has been run. If you pass a directory instead of a file, all files in the directory will be processed. If using `pcap-file-continuous` and passing in a directory, the directory will be monitored for new files being added until you use `pcap-interrupt` or delete/move the directory.

To display how many files are waiting to get processed, you can do:

```
>>> pcap-file-number
Success: 3
```

To display the list of queued files, do:

```
>>> pcap-file-list
Success: {'count': 2, 'files': ['/home/benches/file1.pcap', '/home/benches/file2.pcap']}
```

To display current processed file:

```
>>> pcap-current
Success:
"/tmp/test.pcap"
```

When passing in a directory, you can see last processed time (modified time of last file) in milliseconds since epoch:

```
>>> pcap-last-processed
Success:
1509138964000
```

To interrupt directory processing which terminates the current state:

```
>>> pcap-interrupt
Success:
"Interrupted"
```

21.5 Build your own client

The protocol is documented in the following page https://redmine.openinfosecfoundation.org/projects/suricata/wiki/Unix_Socket#Protocol

The following session show what is sent (SND) and received (RCV) by the server. Initial negotiation is the following:

```
# suricatasc
SND: {"version": "0.1"}
RCV: {"return": "OK"}
```

Once this is done, commands can be issued:

```
>>> iface-list
SND: {"command": "iface-list"}
RCV: {"message": {"count": 1, "ifaces": ["wlan0"]}, "return": "OK"}
Success: {'count': 1, 'ifaces': ['wlan0']}
>>> iface-stat wlan0
SND: {"command": "iface-stat", "arguments": {"iface": "wlan0"}}
RCV: {"message": {"pkts": 41508, "drop": 0, "invalid-checksums": 0}, "return": "OK"}
Success: {'pkts': 41508, 'drop': 0, 'invalid-checksums': 0}
```

In pcap-file mode, this gives:

```
>>> pcap-file /home/eric/git/oisf/benches/sandnet.pcap /tmp/bench
SND: {"command": "pcap-file", "arguments": {"output-dir": "/tmp/bench", "filename": "/
↪home/eric/git/oisf/benches/sandnet.pcap"}}
```

(continues on next page)

(continued from previous page)

```
RCV: {"message": "Successfully added file to list", "return": "OK"}
Success: Successfully added file to list
>>> pcap-file-number
SND: {"command": "pcap-file-number"}
RCV: {"message": 1, "return": "OK"}
>>> pcap-file-list
SND: {"command": "pcap-file-list"}
RCV: {"message": {"count": 1, "files": ["/home/eric/git/oisf/benches/sandnet.pcap"]},
↪"return": "OK"}
Success: {'count': 1, 'files': ['/home/eric/git/oisf/benches/sandnet.pcap']}
>>> pcap-file-continuous /home/eric/git/oisf/benches /tmp/bench 0 true
SND: {"command": "pcap-file", "arguments": {"output-dir": "/tmp/bench", "filename": "/
↪home/eric/git/oisf/benches/sandnet.pcap", "tenant": 0, "delete-when-done": true}}
RCV: {"message": "Successfully added file to list", "return": "OK"}
Success: Successfully added file to list
```

There is one thing to be careful about: a Suricata message is sent in multiple send operations. This result in possible incomplete read on client side. The worse workaround is to sleep a bit before trying a recv call. An other solution is to use non blocking socket and retry a recv if the previous one has failed.

Pcap-file json format is:

```
{
  "command": "pcap-file",
  "arguments": {
    "output-dir": "path to output dir",
    "filename": "path to file or directory to run",
    "tenant": 0,
    "continuous": false,
    "delete-when-done": false
  }
}
```

output-dir and *filename* are required. *tenant* is optional and should be a number, indicating which tenant the file or directory should run under. *continuous* is optional and should be true/false, indicating that file or directory should be run until *pcap-interrupt* is sent or ctrl-c is invoked. *delete-when-done* is optional and should be true/false, indicating that the file or files under the directory specified by *filename* should be deleted when processing is complete. *delete-when-done* defaults to false, indicating files will be kept after processing.

CHAPTER

TWENTYTWO

PLUGINS

Suricata bundles a few plugins that can't be built-in by default.

22.1 nDPI

22.1.1 Installation

Before using nDPI, Suricata must be built with nDPI support, for example:

```
./configure --enable-ndpi --with-ndpi=/home/user/src/nDPI
```

Then make sure the plugin is loaded in your `suricata.yaml`:

```
plugins:
  - /usr/lib/suricata/ndpi.so
```

Which should also be present in the default configuration file after building Suricata with nDPI support.

For more information on nDPI, see https://www.ntop.org/products/deep-packet-inspection/ndpi/.

22.1.2 Keywords

Once the nDPI plugin is loaded, the following keyword are available:

- `ndpi-protocol`
- `ndpi-risk`

`ndpi-protocol`

Match on the Layer-7 protocol detected by nDPI.

Note that rules using the `ndpi-protocol` should check if the `ndpi-protocol` keyword exists with `requires`, for example:

```
requires: keyword ndpi-protocol
```

Syntax:

```
ndpi-protocol:[!]<protocol>;
```

Where *<protocol>* is one of the application protocols detected by nDPI. Plase check *ndpiReader -H* for the full list. It is possible to specify the transport protocol, the application protocol, or both (dot-separated).

Examples:

```
ndpi-protocol:HTTP;
ndpi-protocol:!TLS;
ndpi-protocol:TLS.YouTube;
```

Here is an example of a rule matching TLS traffic on port 53:

alert tcp any any -> any 53 (msg:"TLS traffic over DNS standard port"; requires:keyword ndpi-protocol; ndpi-protocol:TLS; sid:1;)

ndpi-risk

Match on the flow risks detected by nDPI. Risks are potential issues detected by nDPI during the packet dissection and include:

- Known protocol on non-standard port
- Binary application transfer
- Self-signed certificate
- Suspected DGA Domain name
- Malware host contacted
- and many others...

Additionally, rules using the `ndpi-risk` keyword should check if the keyword exists using the `requires` keyword, for example:

```
requires: keyword ndpi-risk
```

Syntax:

```
ndpi-risk:[!]<risk>;
```

Where risk is one (or multiple comma-separated) of the risk codes supported by nDPI (e.g. NDPI_BINARY_APPLICATION_TRANSFER). Please check `ndpiReader -H` for the full list.

Examples:

```
ndpi-risk:NDPI_BINARY_APPLICATION_TRANSFER;
ndpi-risk:NDPI_TLS_OBSOLETE_VERSION,NDPI_TLS_WEAK_CIPHER;
```

Here is an example of a rule matching HTTP traffic transferring a binary application:

alert tcp any any -> any any (msg:"Binary application transfer over HTTP"; requires:keyword ndpi-protocol, keyword ndpi-risk; ndpi-protocol:HTTP; ndpi-risk:NDPI_BINARY_APPLICATION_TRANSFER; sid:1;)

CHAPTER

TWENTYTHREE

IPS MODE

23.1 IPS Concept

Intrusion Prevention System mode, or IPS mode, is the Suricata mode that makes it act as a traffic filter.

By default it will allow all traffic, and will use `drop` or `reject` rules to block unwanted traffic.

It is generally used `inline`, where threat detection rules are used to drop known bad traffic.

The `inline` operations are either on layer 2 (bridge, for example using AF_PACKET or DPDK) or on layer 3 (routing, for example in NFQueue or IPFW).

23.1.1 Differences from the passive IDS mode

TCP stream engine

Where in IDS mode TCP traffic is only inspected after the acknowledgement (ACK) for it has been received, in IPS mode the default behavior is different: new data is inspected immediately, together with previous data where possible. The inspection happens in a sliding window. This behavior is controlled by the `stream.inline` setting.

In case of overlapping data, the first data Suricata receives is accepted. Follow-up data that overlaps with this is then checked against the first data. If it is different, the traffic on the wire is rewritten to match the first data.

The sliding window inspection can be visualized as such:

```
Packet 1: [hdr][segment data 1     ]
Segments: [segment data 1     ]
Window:   [ inspection window ]

Packet 2: [hdr][segment data 2]
Segments: [segment data 1     ][segment data 2]
Window:   [ inspection window                 ]

Packet 3: [hdr][segment data 3]
Segments: [segment data 1     ][segment data 2][segment data 3]
Window:            [ inspection window                        ]

Packet 4: [hdr][segment data 4]
Segments: [segment data 2][segment data 3][segment data 4]
Window:            [ inspection window                   ]
```

Each segment's data is inspected together with the other available data. One consequence of this is that there can be significant rescanning of data, which has a performance impact.

http body inspection

Similar to the sliding window approach in the TCP stream engine, the HTTP body inspection will happen in a sliding manner by default. In IDS mode body data is buffered to the configured settings before inspection.

```
app-layer:
  protocols:
    http:
      libhtp:
         default-config:
           # auto will use http-body-inline mode in IPS mode, yes or no set it statically
           http-body-inline: auto
```

file.data

For HTTP, the `file.data` logic is the same as the body inspection above.

23.1.2 Exception Policies

By default, when IPS mode is enabled, the exception policies are set to block (`drop`). This is to make sure rules cannot be bypassed due to Suricata reaching an error state in parsing, reaching internal resource limits and other exception conditions.

See *Exception Policies documentation*.

23.1.3 Differences from Firewall Mode

The main difference is that unlike IPS mode, Firewall Mode has a default `drop` policy. This means that a ruleset must be created to allow traffic to be accepted, instead of accepting traffic by default and filtering out unwanted traffic.

See *Firewall Mode Design*.

23.2 Setting up IPS/inline for Linux

23.2.1 Setting up IPS with Netfilter

In this guide, we'll discuss how to work with Suricata in layer3 *inline mode* using `iptables`.

First, start by compiling Suricata with NFQ support. For instructions see Ubuntu Installation. For more information about NFQ and `iptables`, see *NFQ*.

To check if you have NFQ enabled in your Suricata build, enter the following command:

```
suricata --build-info
```

and make sure that NFQ is listed in the output.

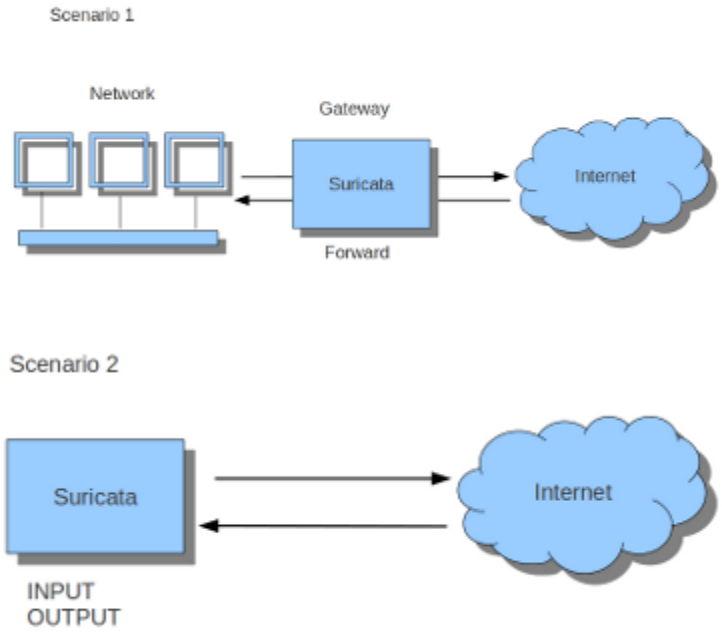
To run Suricata with the NFQ mode, you have to make use of the `-q` option. This option tells Suricata which queue numbers it should use.

```
sudo suricata -c /etc/suricata/suricata.yaml -q 0
```

Iptables configuration

First of all, it is important to know which traffic you would like to send to Suricata. There are two choices:

1. Traffic that passes your computer
2. Traffic that is generated by your computer.



If Suricata is running on a gateway and is meant to protect the computers behind that gateway you are dealing with the first scenario: *forward_ing* .

If Suricata has to protect the computer it is running on, you are dealing with the second scenario: *host* (see drawing 2).

These two ways of using Suricata can also be combined.

The easiest rule in case of the gateway-scenario to send traffic to Suricata is:

```
sudo iptables -I FORWARD -j NFQUEUE
```

In this case, all forwarded traffic goes to Suricata.

In case of the host situation, these are the two most simple `iptables` rules;

```
sudo iptables -I INPUT -j NFQUEUE
sudo iptables -I OUTPUT -j NFQUEUE
```

It is possible to set a queue number. If you do not, the queue number will be 0 by default.

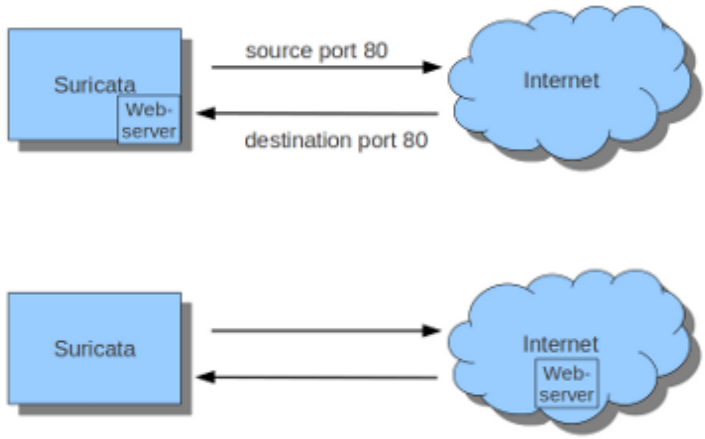
Imagine you want Suricata to check for example just TCP traffic, or all incoming traffic on port 80, or all traffic on destination-port 80, you can do so like this:

```
sudo iptables -I INPUT -p tcp  -j NFQUEUE
sudo iptables -I OUTPUT -p tcp -j NFQUEUE
```

In this case, Suricata checks just TCP traffic.

```
sudo iptables -I INPUT -p tcp --sport 80  -j NFQUEUE
sudo iptables -I OUTPUT -p tcp --dport 80 -j NFQUEUE
```

In this example, Suricata checks all packets for outgoing connections to port 80.



To see if you have set your `iptables` rules correct make sure Suricata is running and enter:

```
sudo iptables -vnL
```

In the example you can see if packets are being logged.

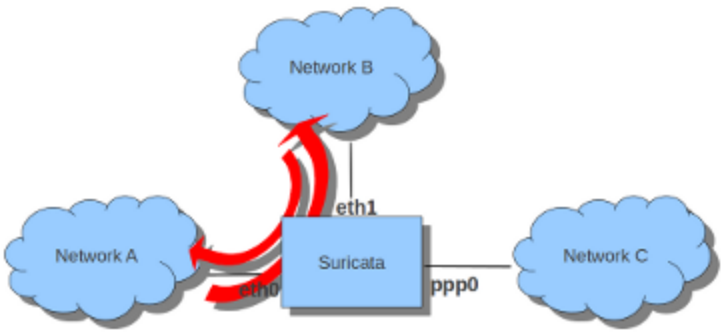
```
anne-fleur@t60:~$ sudo iptables -vnL
Chain INPUT (policy ACCEPT 258 packets, 43900 bytes)
 pkts bytes target     prot opt in     out     source               destination
 4979 5846K NFQUEUE    tcp  --  *      *       0.0.0.0/0            0.0.0.0/0           tcp spt:80 NFQUEUE num 0

Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain OUTPUT (policy ACCEPT 278 packets, 43459 bytes)
 pkts bytes target     prot opt in     out     source               destination
 5286  388K NFQUEUE    tcp  --  *      *       0.0.0.0/0            0.0.0.0/0           tcp dpt:80 NFQUEUE num 0
anne-fleur@t60:~$
```

This description of the use of `iptables` is the way to use it with IPv4. To use it with IPv6 all previous mentioned commands have to start with `ip6tables`. It is also possible to let Suricata check both kinds of traffic.

There is also a way to use `iptables` with multiple networks (and interface cards). Example:




```
sudo iptables -I FORWARD -i eth0 -o eth1 -j NFQUEUE
sudo iptables -I FORWARD -i eth1 -o eth0 -j NFQUEUE
```

The options `-i` (input) `-o` (output) can be combined with all previous mentioned options.

If you would stop Suricata and use internet, the traffic will not come through. To make internet work correctly, first delete all `iptables` rules.

To erase all `iptables` rules, enter:

```
sudo iptables -F
```

NFtables configuration

The NFtables configuration is straight forward and allows mixing firewall rules with IPS. The concept is to create a dedicated chain for the IPS that will be evaluated after the firewalling rule. If your main table is named *filter* it can be created like so:

```
nft> add chain filter IPS { type filter hook forward priority 10;}
```

To send all forwarded packets to Suricata one can use

```
nft> add rule filter IPS queue
```

To only do it for packets exchanged between eth0 and eth1

```
nft> add rule filter IPS iif eth0 oif eth1 queue
nft> add rule filter IPS iif eth1 oif eth0 queue
```

NFQUEUE advanced options

The NFQUEUE mechanism supports some interesting options. The `nftables` configuration will be shown there but the features are also available in `iptables`.

The full syntax of the queuing mechanism is as follows:

```
nft add rule filter IPS queue num 3-5 options fanout,bypass
```

This rule sends matching packets to 3 load-balanced queues starting at 3 and ending at 5. To get the packets in Suricata with this setup, you need to specify multiple queues on command line:

```
suricata -q 3 -q 4 -q 5
```

fanout and *bypass* are the two available options:

- *fanout*: When used together with load balancing, this will use the CPU ID instead of connection hash as an index to map packets to the queues. The idea is that you can improve performance if there's one queue per CPU. This requires total with a number of queues superior to 1 to be specified.
- *bypass*: By default, if no userspace program is listening on an Netfilter queue, then all packets that are to be queued are dropped. When this option is used, the queue rule behaves like ACCEPT if there is no program listening, and the packet will move on to the next table.

The *bypass* option can be used to avoid downtime of link when Suricata is not running but this also means that the blocking feature will not be present.

23.2.2 Setting up IPS at Layer 2

AF_PACKET IPS mode

AF_PACKET capture method is supporting a IPS/Tap mode. In this mode, you just need the interfaces to be up. Suricata will take care of copying the packets from one interface to the other. No `iptables` or `nftables` configuration is necessary.

You need to dedicate two network interfaces for this mode. The configuration is made via configuration variable available in the description of an AF_PACKET interface.

For example, the following configuration will create a Suricata acting as IPS between interface `eth0` and `eth1`:

```
af-packet:
  - interface: eth0
    threads: 1
    defrag: no
    cluster-type: cluster_flow
    cluster-id: 98
    copy-mode: ips
    copy-iface: eth1
    buffer-size: 64535
  - interface: eth1
    threads: 1
    cluster-id: 97
    defrag: no
    cluster-type: cluster_flow
    copy-mode: ips
    copy-iface: eth0
    buffer-size: 64535
```

This is a basic af-packet configuration using two interfaces. Interface `eth0` will copy all received packets to `eth1` because of the *copy-** configuration variable

```
copy-mode: ips
copy-iface: eth1
```

The configuration on `eth1` is symmetric

```
copy-mode: ips
copy-iface: eth0
```

There are some important points to consider when setting up this mode:

- MTU on both interfaces have to be equal: the copy from one interface to the other is direct and packets bigger then the MTU will be dropped by kernel.
- Set different values of *cluster-id* on both interfaces to avoid conflict.
- Any network card offloading creating bigger then physical layer datagram (like GRO, LRO, TSO) will result in dropped packets as the transmit path can not handle them.
- Set *stream.inline* to *auto* or *yes* so Suricata switches to blocking mode.

The *copy-mode* variable can take the following values:

- *ips*: the drop keyword is honored and matching packets are dropped.
- *tap*: no drop occurs, Suricata acts as a bridge

Some specific care must be taken to scale the capture method on multiple threads. As we can't use defrag that will generate too big frames, the in kernel load balancing will not be correct: the IP-only fragment will not reach the same thread as the full featured packet of the same flow because the port information will not be present.

A solution is to use eBPF load balancing to get an IP pair load balancing without fragmentation. The AF_PACKET IPS Configuration using multiple threads and eBPF load balancing looks like the following:

```
af-packet:
  - interface: eth0
    threads: 16
    defrag: no
    cluster-type: cluster_ebpf
    ebpf-lb-file: /usr/libexec/suricata/ebpf/lb.bpf
    cluster-id: 98
    copy-mode: ips
    copy-iface: eth1
    buffer-size: 64535
  - interface: eth1
    threads: 16
    cluster-id: 97
    defrag: no
    cluster-type: cluster_ebpf
    ebpf-lb-file: /usr/libexec/suricata/ebpf/lb.bpf
    copy-mode: ips
    copy-iface: eth0
    buffer-size: 64535
```

The eBPF file `/usr/libexec/suricata/ebpf/lb.bpf` may not be present on disk. See *eBPF and XDP* for more information.

DPDK IPS mode

In the same way as you would configure AF_PACKET IPS mode, you can configure the DPDK capture module. Prior to starting with IPS (inline) setup, it is recommended to go over *Data Plane Development Kit (DPDK)* manual page to understand the setup essentials.

DPDK IPS mode, similarly to AF-Packet, uses two interfaces. Packets received on the first network interface (`0000:3b:00.1`) are transmitted by the second network interface (`0000:3b:00.0`) and similarly, packets received on the second interface (`0000:3b:00.0`) are transmitted by the first interface (`0000:3b:00.1`). Packets are not altered in any way in this mode.

The following configuration snippet configures Suricata DPDK IPS mode between two NICs:

```
dpdk:
  eal-params:
    proc-type: primary

  interfaces:
  - interface: 0000:3b:00.1
    threads: 4
    promisc: true
    multicast: true
    checksum-checks: true
    checksum-checks-offload: true
    mempool-size: 262143
    mempool-cache-size: 511
    rx-descriptors: 4096
    tx-descriptors: 4096
    copy-mode: ips
    copy-iface: 0000:3b:00.0
    mtu: 3000

  - interface: 0000:3b:00.0
    threads: 4
    promisc: true
    multicast: true
    checksum-checks: true
    checksum-checks-offload: true
    mempool-size: 262143
    mempool-cache-size: 511
    rx-descriptors: 4096
    tx-descriptors: 4096
    copy-mode: ips
    copy-iface: 0000:3b:00.1
    mtu: 3000
```

The previous DPDK configuration snippet outlines several things to consider:

- `copy-mode` - see Section *AF_PACKET IPS mode* for more details.
- `copy-iface` - see Section *AF_PACKET IPS mode* for more details.
- `threads` - all interface entries must have their thread count configured and paired/connected interfaces must be configured with the same amount of threads.
- `mtu` - MTU must be the same on both paired interfaces.

DPDK capture module also requires having CPU affinity set in the configuration file. For the best performance, every Suricata worker should be pinned to a separate CPU core that is not shared with any other Suricata thread (e.g. management threads). The following snippet shows a possible *Threading* configuration set-up for DPDK IPS mode.

```
threading:
  set-cpu-affinity: yes
  cpu-affinity:
    management-cpu-set:
      cpu: [ 0 ]
    worker-cpu-set:
      cpu: [ 2,4,6,8,10,12,14,16 ]
```

Netmap IPS mode

Using Netmap to support IPS requires setting up pairs of interfaces; packets are received on one interface within the pair, inspected by Suricata, and transmitted on the other paired interface. You can use native or host stack mode; host stack mode is used when the interface name contains the ^ character, e.g, `enp6s0f0^`. host stack mode does not require multiple physical network interfaces.

Netmap Host Stack Mode

Netmap's host stack mode allows packets that flow through Suricata to be used with other host OS applications, e.g., a firewall or similar. Additionally, host stack mode allows traffic to be received and transmitted on one network interface card.

With host stack mode, Netmap establishes a pair of host stack mode rings (one each for RX and TX). Packets pass through the host operating system network protocol stack. Ingress network packets flow from the network interface card to the network protocol stack and then into the host stack mode rings. Outbound packets flow from the host stack mode rings to the network protocol stack and finally, to the network interface card. Suricata receives packets from the host stack mode rings and, in IPS mode, places packets to be transmitted into the host stack mode rings. Packets transmitted by Suricata into the host stack mode rings are available for other host OS applications.

Paired network interfaces are specified in the `netmap` configuration section. For example, the following configuration will create a Suricata acting as IPS between interface `enp6s0f0` and `enp6s0f1`

```
netmap:
  - interface: enp6s0f0
    threads: auto
    copy-mode: ips
    copy-iface: enp6s0f1

  - interface: enp6s0f1
    threads: auto
    copy-mode: ips
    copy-iface: enp6s0f0
```

You can specify the `threads` value; the default value of `auto` will create a thread for each queue supported by the NIC; restrict the thread count by specifying a value, e.g., `threads: 1`

This is a basic netmap configuration using two interfaces. Suricata will copy packets between interfaces `enp6s0f0` and `en60sf1` because of the *copy-** configuration variable in interface's `enp6s0f0` configuration

```
copy-mode: ips
copy-iface: enp6s0f1
```

The configuration on `enp6s0f1` is symmetric

```
copy-mode: ips
copy-iface: enp6s0f0
```

The host stack mode feature of Netmap can be used. host stack mode doesn't require a second network interface.

This example demonstrates host stack mode with a single physical network interface `enp6s0f01`

```
- interface: enp60s0f0
  copy-mode: ips
  copy-iface: enp6s0f0^
```

The configuration on `enp6s0f0^` is symmetric

```
- interface: enp60s0f0^
  copy-mode: ips
  copy-iface: enp6s0f0
```

Suricata will use zero-copy mode when the runmode is `workers`.

There are some important points to consider when setting up this mode:

- Any network card offloading creating bigger then physical layer datagram (like GRO, LRO, TSO) will result in dropped packets as the transmit path can not handle them.
- Set *stream.inline* to *auto* or *yes* so Suricata switches to blocking mode. The default value is *auto*.

The *copy-mode* variable can take the following values:

- *ips*: the drop keyword is honored and matching packets are dropped.
- *tap*: no drop occurs, Suricata acts as a bridge

23.3 Setting up IPS/inline for Windows

This guide explains how to work with Suricata in layer 4 inline mode using WinDivert on Windows.

First start by compiling Suricata with WinDivert support. For building instructions, see ../install/windows.

`WinDivert.dll` and `WinDivert.sys` must be in the same directory as the Suricata executable. WinDivert automatically installs the driver when it is run. For more information about WinDivert, see https://www.reqrypt.org/windivert-doc.html.

To check if you have WinDivert enabled in your Suricata, enter the following command in an elevated command prompt or terminal:

```
suricata -c suricata.yaml --windivert [filter string]
```

For information on the WinDivert filter language, see https://www.reqrypt.org/windivert-doc.html#filter_language

If Suricata is running on a gateway and is meant to protect the network behind that gateway, you need to run WinDivert at the *NETWORK_FORWARD* layer. This can be achieved using the following command:

```
suricata -c suricata.yaml --windivert-forward [filter string]
```

The filter is automatically stopped and normal traffic resumes when Suricata is stopped.

A quick start is to examine all traffic, in which case you can use the following command:

```
suricata -c suricata.yaml --windivert[-forward] true
```

A few additional examples:

Only TCP traffic:

```
suricata -c suricata.yaml --windivert tcp
```

Only TCP traffic on port 80:

```
suricata -c suricata.yaml --windivert "tcp.DstPort == 80"
```

TCP and ICMP traffic:

```
suricata -c suricata.yaml --windivert "tcp or icmp"
```

CHAPTER

TWENTYFOUR

FIREWALL MODE

24.1 Firewall Mode Design

> **Note**
>
> Suricata firewall mode features and behavior are still subject to change.

The firewall mode in Suricata allows the use of a ruleset that has different properties than the default "threat detection" rulesets:

1. default policy is `drop`, meaning a firewall ruleset needs to specify what is allowed
2. firewall rules are loaded from separate files
3. firewall rules use a new action `accept`
4. firewall rules are required to use explicit action scopes and rule hooks (see below)
5. evaluation order is as rules are in the file(s), per protocol state

24.1.1 Concepts

Firewall vs Threat Detection (TD)

The interaction between firewall and TD is concepualized as if they are 2 seperate instances, where the firewall instance runs first, and it passes along to the TD instance what is accepted by the firewall.

This is reflected in the stats, where a packet accepted by the firewall is counted as `firewall.accepted`. If it was also allowed by TD, it will additionally be counted as `ips.accepted`. If it was dropped by firewall, only `firewall.blocked` will be incremented. No `ips.*` counter will be updated as conceptually the TD instance won't have seen the packet.

Tables

A `table` is a collection of rules with different properties. These tables are built-in. No custom tables can be created. Tables are available within the scope of packet layer and application layer (if available). Each rule can define its own *action scope*.

Packet layer tables

Rules categorized in the following tables apply to all packets.

Table	Description	Default Policy	Rule Order
packet:pre_flow	Firewall rules to be evaluated before flow is created/updated	accept:hook	As appears in the rule file
packet:pre_stre	Firewall rules to be evaluated before stream is updated	accept:hook	As appears in the rule file
packet:filter	Firewall rules to be evaluated against every packet after decoding	drop:packe	As appears in the rule file
packet:td	Generic IDS/IPS threat detection rules	accept:hook	Internal IDS/IPS rule ordering

Application layer tables

If applayer is available, rules from the following tables apply. The tables for the application layer are per app layer protocol and per protocol state. e.g. `http1:request_line`.

Table	Description	Default Policy	Rule Order
app:filter	Firewall rules to be evaluated per applayer protocol and state	drop:flow	As appears in the rule file
app:td	App-layer IDS/IPS threat detection rules	accept:hook	Internal IDS/IPS rule ordering

Actions and Action Scopes

Firewall rules require action scopes to be explicitly specified.

accept

`accept` is used to issue an accept verdict to the packet, flow or hook.

- `packet` accept this packet
- `flow` accept the rest of the packets in this flow
- `hook` accept rules for the current hook/state, evaluate the next tables
- `tx` accept rules for the current transaction, evaluate the next tables

The `accept` action is only available in firewall rules.

> **Note**
>
> some protocol implementations like `dns` use a transaction per direction. For those `accept:tx` will only accept packets that are part of that direction.

drop

`drop` is used to drop either the packet or the flow

- `packet` drop this packet directly, don't eval any further rules
- `flow` drop this packet as with `packet` and drop all future packets in this flow

> **Note**
>
> unlike in threat detection mode rules, a `drop` in a firewall rule does not imply alert

pass

`pass` is not available as a primary firewall action, but can be used as a secondary action in firewall rules. The effect of the action will only apply to threat detection rules.

alert

`alert` is not available as a primary firewall action, but can be used as a secondary action in firewall rules. The effect will be the creation of an alert event when the firewall rule matches.

For application layer transactions the alert event carries a `firewall` object with the resolved `policy` and, when the protocol registers a state name callback, the `hook` the rule was registered at. Protocols without such a callback (`quic`, `modbus`, `rdp`) omit the `hook` key; the policy is still reported.

config

`config` is a primary firewall action used to apply the setting of the `config` rule keyword when the rule matches, see *Config Rules*. The `config` action does not issue a verdict for the packet or the flow, so the other tables are still evaluated. It is not available as a secondary action.

Multi action rules

In firewall rules, multiple actions can be specified: a primary firewall action, followed by one or more secondary actions.

Example:

```
accept:flow,pass:flow,alert tls:client_hello ... tls.sni; ...
```

In this example the first action `accept:flow` is the primary firewall action. When the rule matches, the flow will be accepted. The secondary actions `pass:flow` and `alert` are evaluated in the context of the threat detection engine.

> **Note**
>
> the secondary actions are only evaluated if the primary firewall action is accepted. This is different from the behavior of the `pass` action in threat detection mode.

Explicit rule hook (states)

In the regular IDS/IPS rules the engine infers from the rule's matching logic where the rule should be "hooked" into the engine. While this works well for these types of rules, it does lead to many edge cases that are not acceptable in a firewall ruleset. For this reason in the firewall rules the hook needs to be explicitly set.

There are two types of hooks available based on the layer.

Packet layer hooks

- `flow_start`: evaluate the rule only on the first packet in both the directions
- `pre_flow`: evaluate the rule before the flow is created/updated
- `pre_stream`: evaluate the rule before the stream is updated

- `all`: evaluate the rule on every packet

Application layer hooks

The application layer states / hooks are defined per protocol. Each of the hooks has its own default-`drop` policy, so a ruleset needs an `accept` rule for each of the states to allow the traffic to flow through.

This is done in the protocol field of the rule. Where in threat detection a rule might look like:

```
alert http ... http.uri; ...
```

In the firewall case it will be:

```
accept:hook http1:request_line ... http.uri; ...
```

All available applayer hooks are available via commandline option `--list-app-layer-hooks`.

general

Each protocol has at least the default states.

Request (`to_server`) side:

- `request_started`
- `request_complete`

Response (`to_client`) side:

- `response_started`
- `response_complete`

http

For the HTTP protocol there are a number of states to hook into. These apply to HTTP 0.9, 1.0 and 1.1. HTTP/2 uses its own state machine.

Available states:

Request (`to_server`) side:

- `request_started`
- `request_line`
- `request_headers`
- `request_body`
- `request_trailer`
- `request_complete`

Response (`to_client`) side:

- `response_started`
- `response_line`
- `response_headers`
- `response_body`
- `response_trailer`

- response_complete

tls

Available states:

Request (to_server) side:

- client_started
- client_hello
- client_cert
- client_data
- client_finished

The client_hello state names the ClientHello message being parsed: it is entered when the message starts, so the fragments of a hello split over several TLS records are all decided at client_hello. The hello data (SNI, version, ...) is available once the final fragment completes the message, which moves the state to client_cert; rules evaluated on the earlier fragments see the buffers still empty, the same way a split HTTP request line is in request_line before the line is complete.

Response (to_client) side:

- server_started
- server_hello
- server_cert
- server_data
- server_finished

ssh

Available states are listed in *Hooks*.

Auto-accept prior states

To avoid creating lots of boilerplate accept rules there is a special notation to have a rule accept not just the hook it matches in, but also the hooks before it.

Example:

```
accept:flow tls:<client_hello ... tls.sni; content:"suricata.io"; ...
```

The main matching logic here is in the tls:client_hello hook. The state before it, tls:client_started, is also accepted, as if the ruleset was actually:

```
accept:hook tls:client_started ...
accept:flow tls:client_hello ... tls.sni; content:"suricata.io"; ...
```

This logic only applies to the app:filter table.

Firewall pipeline

The firewall pipeline works in the detection engine, and is invoked after packet decoding, flow update, stream tracking and reassembly and app-layer parsing are all done in the context of a single packet.

For each packet rules in the first firewall hook `packet:filter` are then evaluated. Assuming the verdict of this hook is `accept:hook`, the next hook is evaluated: `packet:td` (packet threat detection). In this hook the IDS/IPS rules are evaluated. Rule actions here are not immediate, as they can still be modified by alert postprocessing like rate_filter, thresholding, etc.

The default `drop` for the `packet:filter` table is `drop:packet`. Thus the `drop` is only applied to the current packet.

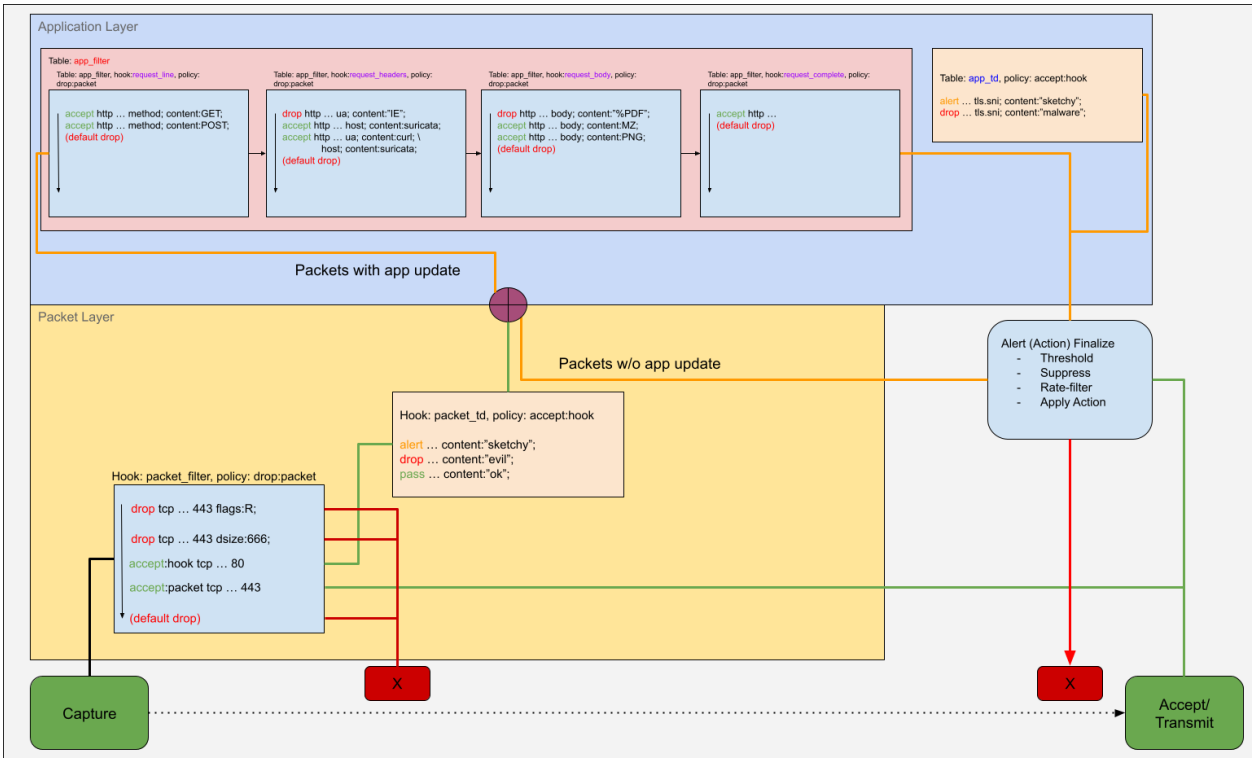
If the packet has been marked internally as a packet with an application layer update, then the next table is `app:*:*`.

In `app:*:*` the per application layer states are all evaluated at least once. At each of these states an `accept:hook` is required to progress to the next state. When all available states have been accepted, the pipeline moves to the final table `app:td` (application layer threat detection). A `drop` in the `app:filter` table is immediate, however and `accept` is conditional on the verdict of the `app:td` table.

The default `drop` in one of the `app:*:*` tables is a `drop:flow`. This means that the current packet as well as all future packets from that flow are dropped.

In `app:td` the IDS/IPS rules for the application layer are evaluated. `drop` actions in this table are queued in the alert queue.

When all tables have been evaluated, the alert finalize process orders threat detection alerts by `action-order` logic. It can then apply a `drop` or default to `accept`-ing.



Pass rules with Firewall mode

In IDS/IPS mode, a `pass` rule with app-layer matches will bypass the detection engine for the rest of the flow. In firewall mode, this bypass no longer happens in the same way, as `pass` rules do not affect firewall rules. So the detection engine is still invoked on packets of such a flow, but the `packet:td` and `app:td` tables are skipped.

24.1.2 Firewall rules

Firewall rules are loaded first and separately from the following section of `suricata.yaml`:

```
firewall-rule-path: /etc/suricata/firewall/
firewall-rule-files:
  - fw.rules
```

One can optionally, also load firewall rules exclusively from commandline using the `--firewall-rules-exclusive` option. Note that this option blocks hot rule reloads, just like the `-S` option in thread detection rules.

Firewall rules are available in the file `firewall.json` as a part of the output of *engine analysis*.

24.1.3 Bridge vs router

The firewall mode can be used with capture methods in bridge and router mode. When using the bridge mode, the default drop policy will also apply to non-IP protocols, like ARP.

For ARP to work, a rule to accept it is required:

```
accept:packet arp:all any any -> any any (sid:200;)
```

Other ethernet types can be accepted by using generic ethernet rules, with the `ether.hdr` keyword.

The example below accepts ARP again, using this mechanism.

```
accept:packet ether:all any any -> any any (ether.hdr; content:"|08 06|"; offset:12;␣
↪depth:2; sid:1;)
```

24.1.4 Default policies

Each hook has a default policy applied to traffic that no firewall rule handled. By default `packet.filter` enforces `drop:packet`, `packet.pre-flow` and `packet.pre-stream` enforce `accept:hook`, and every `app` hook enforces `drop:flow`.

Defaults are configured in the `firewall.policies` block. A `default-policy` for any hook may be given at several levels and the most specific present setting wins:

```
firewall:
  policies:
    default-policy: ["accept:hook"]     # global fallback (all hooks)
    packet:
      default-policy: ["drop:packet"]   # fallback for packet hooks
      filter:     ["reject:packet"]
      pre-flow:   ["accept:hook"]
      pre-stream: ["accept:hook"]
    app:
      default-policy: ["drop:flow"]     # fallback for all app hooks
      dns:
        default-policy: ["drop:flow"]   # fallback for dns hooks
        request-started: ["accept:hook"]
        # Drop and alert on all DNS requests that are not allowed in
        # firewall.rules.
        request-complete: ["drop:flow", "alert"]
        # Accept all responses.
        response-started: ["accept:tx"]
```

(continues on next page)

(continued from previous page)

```
    # Define default policies for protocols with sub states
    http2:
      default-policy: ["drop:flow"]   # fallback for all http2 hooks
      stream:
        default-policy: ["drop:flow"] # fallback for http2 stream hooks
        request-started: ["accept:hook"]
      global:
        request-started: ["accept:hook"]
```

Precedence:

- packet hook: `packet.<hook>` > `packet.default-policy` > `policies.default-policy` > built-in (`drop:packet` or `accept:hook`)
- app hook: `app.<proto>.<hook>` > `app.<proto>.default-policy` > `app.default-policy` > `policies.default-policy` > built-in (`drop:flow`)
- app hook in a sub state: `app.<proto>.<sub state>.<hook>` > `app.<proto>.<sub state>.default-policy` > `app.<proto>.default-policy` > `app.default-policy` > `policies.default-policy` > built-in (`drop:flow`)

An action scope must be valid for the hook it is applied to. For example, defining `accept:tx` as a global default policy will fail to start Suricata, because `packet` policies do not accept `tx`. Similarly, `packet.pre-flow` only accepts the `packet` and `hook` scopes, so a `drop:flow` set in `policies.default-policy` or `packet.default-policy` will fail to start Suricata. Cover such hooks with a more specific setting so the incompatible default never reaches them:

```
firewall:
  policies:
    default-policy: ["drop:flow"]
    packet:
      pre-flow: ["accept:hook"]
```

A < hook rule at the protocol's first state (progress 0) is accepted and is equivalent to the plain hook form, as there are no prior states to auto-accept.

24.2 Firewall Ruleset Examples

24.2.1 HTTP

In this example a simple HTTP ruleset will be shown. It will allow HTTP to flow as long as:

- method is GET or POST
- User-Agent is "curl"
- Status code is 200.

It starts by allowing the TCP port 80 traffic.

```
accept:hook tcp:all any any <> any 80 (sid:10;)
```

The stream tracking combined with the default exception policy handling will enforce a proper TCP handshake, etc.

The HTTP rules need to `accept` each state:

```
# allow traffic before the request line is complete
accept:hook http1:request_started any any -> any any (sid:100;)
# allow GET
accept:hook http1:request_line any any -> any any ( \
        http.method; content:"GET"; sid:101;)
# or allow POST
accept:hook http1:request_line any any -> any any ( \
        http.method; content:"POST"; sid:102;)
# allow User-Agent curl
accept:hook http1:request_headers any any -> any any ( \
        http.user_agent; content:"curl"; sid:103;)
# allow the body, if any
accept:hook http1:request_body any any -> any any (sid:104;)
# allow trailers, if any
accept:hook http1:request_trailer any any -> any any (sid:105;)
# allow completion
accept:hook http1:request_complete any any -> any any (sid:106;)

# allow traffic before the response line is complete
accept:hook http1:response_started any any -> any any (sid:200;)
# allow the 200 ok stat code.
accept:hook http1:response_line any any -> any any ( \
        http.stat_code; content:"200"; sid:201;)
# allow all other states
accept:hook http1:response_headers any any -> any any (sid:202;)
accept:hook http1:response_body any any -> any any (sid:203;)
accept:hook http1:response_trailer any any -> any any (sid:204;)
accept:hook http1:response_complete any any -> any any (sid:205;)
```

Each state needs an `accept` rule. Each state is evaluated at least once.

HTTP example with partially using default policies

In the example below: the config auto accepts various hooks, leaving just `http1:request_line`, `http1:request_headers` and `http1:response_line` for the ruleset to accept.

```
firewall:
  policies:
    app:
      http1:
        request-started:
          - "accept:hook"
        request-line:
          - "drop:flow"
          - "alert"
        request-headers:
          - "drop:flow"
          - "alert"
        request-body:
          - "accept:hook"
        request-trailer:
          - "accept:hook"
        request-complete:
```

(continues on next page)

(continued from previous page)

```
          - "accept:hook"

        response-started:
          - "accept:hook"
        response-line:
          - "drop:flow"
          - "alert"
        response-headers:
          - "accept:hook"
        response-body:
          - "accept:hook"
        response-trailer:
          - "accept:hook"
        response-complete:
          - "accept:hook"
```

```
# allow GET
accept:hook http1:request_line any any -> any any ( \
        http.method; content:"GET"; sid:101;)
# or allow POST
accept:hook http1:request_line any any -> any any ( \
        http.method; content:"POST"; sid:102;)
# allow User-Agent curl
accept:hook http1:request_headers any any -> any any ( \
        http.user_agent; content:"curl"; sid:103;)

# allow the 200 ok stat code.
accept:hook http1:response_line any any -> any any ( \
        http.stat_code; content:"200"; sid:201;)
```

24.2.2 TLS SNI with complex TCP rules

In this example the `packet:filter` rules will be more opinionated about the traffic:

```
# allow 3-way handshake
accept:hook tcp:all $HOME_NET any -> $EXTERNAL_NET 443 (flags:S; \
        flow:not_established; flowbits:set,syn; sid:1;)
accept:hook tcp:all $EXTERNAL_NET 443 -> $HOME_NET any (flags:SA; \
        flow:not_established; flowbits:isset,syn; flowbits:set,synack; sid:2;)
accept:hook tcp:all $HOME_NET any -> $EXTERNAL_NET 443 (flags:A; \
        flow:not_established; flowbits:isset,synack;             \
        flowbits:unset,syn; flowbits:unset,synack; sid:3;)
# allow established
accept:hook tcp:all $HOME_NET any <> $EXTERNAL_NET 443 (flow:established; sid:4;)
```

Then on the TLS level this will be a TLS SNI firewall. *Migrating from the old TLS state names* covers migrating rules and config keys from the pre-phase state names.

Again all the states need to be accepted. Only in the `client_hello` state will there be additional constraints:

```
accept:hook tls:client_started $HOME_NET any -> $EXTERNAL_NET any (sid:100;)
# allow the good sites
```

(continues on next page)

(continued from previous page)

```
accept:flow tls:client_hello $HOME_NET any -> $EXTERNAL_NET any (tls.sni; \
        pcre:"/^(suricata.io|oisf.net)$/R"; sid:101;)
# drop every ClientHello that carried an SNI the rule above does
# not allow; the SNI buffer is required, so a fragment still in
# flight (no SNI buffer yet) cannot match
drop:flow tls:client_hello $HOME_NET any -> $EXTERNAL_NET any (alert; tls.sni; \
        pcre:!"/^(suricata.io|oisf.net)$/R"; sid:102;)
# hello packets that carry no verdict of their own: the in-flight
# fragments
accept:hook tls:client_hello $HOME_NET any -> $EXTERNAL_NET any (sid:103;)
# no accept at the certificate state: a hello without an SNI matches
# no SNI rule and is decided by the default policy of the states it
# walked over (the implicit drop:flow) - the same outcome the base
# release gave it at the hello-completion milestone. If you want to
# accept SNI-less hellos, add an explicit accept here and audit the
# relaxation.
accept:hook tls:client_data $HOME_NET any -> $EXTERNAL_NET any (sid:105;)
accept:hook tls:client_finished $HOME_NET any -> $EXTERNAL_NET any (sid:106;)

accept:hook tls:server_started $EXTERNAL_NET any -> $HOME_NET any (sid:200;)
accept:hook tls:server_hello $EXTERNAL_NET any -> $HOME_NET any (sid:201;)
accept:hook tls:server_cert $EXTERNAL_NET any -> $HOME_NET any (sid:202;)
accept:hook tls:server_data $EXTERNAL_NET any -> $HOME_NET any (sid:204;)
accept:hook tls:server_finished $EXTERNAL_NET any -> $HOME_NET any (sid:205;)
```

The `client_hello` state names the ClientHello being parsed: it is entered at the first byte of the message, and its completion moves the state to `client_cert`. The SNI buffer is owned by the hello state, so `tls.sni` rules must live in `client_hello` even though the record that completes the hello is decided at `client_cert`: a flow that matches the SNI accept is carried over by the flow-scoped accept (101) and a flow whose SNI is not allowed is dropped by 102 on that same record - the drop is rule-scoped and alerts, where the implicit default policy would drop silently. The plain `accept:hook` at 103 carries the fragments in flight (the SNI buffer is not available before the message completes, so it also cannot gate them). A ClientHello that carries no SNI at all matches no SNI rule either - and with no accept at the certificate state it is decided by the default policy of the states it walked over (the implicit drop:flow), the same outcome the base release gave it at the hello-completion milestone. A flow whose hello is accepted by 101 is carried by the flow-scoped accept: the flow action is checked before any further firewall rule of a packet is evaluated, so the later states of that flow never reach the implicit default. `client_started` covers the packets before the hello.

TLS SNI with auto-accept logic

Rule that has the same effect as the 11 TLS rules above:

```
accept:flow tls:<client_hello $HOME_NET any -> $EXTERNAL_NET any (tls.sni; \
        pcre:"/^(suricata.io|oisf.net)$/R"; sid:101;)
```

Explanation: `accept:flow` accepts all of the TLS flow from the moment the rule has matched. The `tls:client_started` hook is auto-accepted by the use of the < modifier in the hook `tls:<client_hello`. The < modifier is only valid for firewall rules: a threat detection rule using it fails to load.

TLS SNI with auto-accept logic, plus disabling TD matching

To allow-list a connection to a specific SNI, w/o threat detection rules matching on this flow either, the example above can be extended by adding `pass:flow` as a secondary action:

```
accept:flow,pass:flow tls:<client_hello $HOME_NET any -> $EXTERNAL_NET any \
    (tls.sni; pcre:"/^(suricata.io|oisf.net)$/; sid:101;)
```

Explanation: as soon as this rule fully matches at the `tls:client_hello` hook, a `pass` is applied to the flow effectively bypassing the threat detection engine.

24.3 Firewall Mode Stats

Statistics counters for the firewall mode cover:

- drop reasons: `stats.firewall.drop_reason`
- discarded alerts: `stats.firewall.discarded_alerts`
- blocked packets: `stats.firewall.blocked`
- accepted packets: `stats.firewall.accepted`
- rejected packets: `stats.firewall.rejected`

These will be present in the stats logs if the engine is run in firewall mode, only.

24.3.1 Drop reasons

If a drop was caused by the firewall, the corresponding counter will be incremented. The existing ones are:

- `rules`: a firewall rule triggered the drop
- `default_packet_policy`: drop caused by the default fail closed firewall behavior, on the packet hook level
- `default_app_policy`: drop caused by the default fail close firewall behavior, on the app-layer hook level
- `pre_flow_hook`: drop caused by the pre-flow hook
- `pre_stream_hook`: drop caused by the pre-stream hook
- `flow_drop`: the whole flow was dropped after a firewall action.

24.3.2 Discarded alerts

When the alert queue is full for a firewall rule match, the alert is discarded. These are reported with the counter `stats.firewall.discarded_alerts`. Note that the drop may be caused by non-firewall rules.

The rule's primary action as well as other secondary actions will be applied to the traffic as normal.

24.4 Migrating from the old TLS state names

In this release the TLS state names changed from completion milestones to active phases, following the HTTP1 unit states: a state names the handshake message (or data exchange) the state progression is working through - it is entered when that unit starts being parsed and left when the unit completes. The phase data (SNI, certificates) is filled only once the message has been fully parsed; rules evaluated while it is still arriving (a fragmented message) see no data at all, so their content keywords cannot match. The state names used in rules (`accept:hook tls:<state>`, `alert tls:<state>`) and in the `firewall.policies.app.tls` config keys were renamed accordingly, and the flight-completion state was removed.

24.4.1 Old and new names

old state (mile-stone)	new state (phase)	note
client_in	client_	covers the packets before the hello
client_he	client_	the phase is entered at the start of the hello message
client_ce	client_	entered when the client hello completes, before any certificate byte: the records up to the completing certificate message carry no certificate data - add a plain `accept:hook tls:client_cert` (or a drop-based cert check plus that accept) for them
client_ha	client_	
client_fi	client_	unchanged
server_in	server_	
server_he	server_	unchanged
server_ce	server_	entered when the server hello completes, before any certificate byte: the records up to the completing certificate message carry no certificate data - add a plain `accept:hook tls:server_cert` (or a drop-based cert check plus that accept) for them
server_he	server_	the flight-completion milestone no longer exists
server_ha	server_	both map to the data phase
server_fi	server_	unchanged

A plain accept at `client_cert` or `server_cert` - including the pairing of a drop-based cert check with that accept - also decides the record that completes the hello, and with it a ClientHello that carries no SNI at all: the SNI buffer is only filled once the message is fully parsed, so no `tls.sni` rule (including a negated one) can match it. The base release dropped that hello at the `client_hello_done` milestone through the implicit default policy; a certificate-state accept accepts it. Keep the certificate state free of an accept to preserve the base outcome - the implicit default policy decides it, identically to the milestone - or add an explicit rule for the SNI-less case if it must be accepted. The SNI example in the firewall user guide and the upgrade chapter carry the same warning.

24.4.2 Semantics that changed with the rename

- `client_hello` names the ClientHello being parsed: it is entered at the first byte of the message, so every fragment of a hello split over several TLS records is decided at `client_hello` (with the SNI and the other hello buffers filled only once the final fragment completes the message). Completion moves the state to `client_cert` before a single certificate byte arrives, and the record that completes the hello is decided there; a Certificate message also names the `client_cert` phase at its own start, which keeps a state progression that begins mid-connection or after a failed hello coherent. Rules on hello data (`tls.sni`) still live in `client_hello` - that state owns the buffer - but their accept must carry the flow: the verdict of the completing record is rendered at `client_cert`, so an `accept:hook` at `client_hello` alone does not let the flow through. A hello that fails to parse keeps the state progression in `client_hello`: the hand-over to the certificate phase is suppressed, but the fields the parser already filled (version, session id, an SNI extension parsed before the failure) stay in the connp and remain matchable - the phase data may be incomplete rather than absent, as in the previous model. `server_hello` behaves identically for the ServerHello.
- `server_data` is entered on the first server (to_client) app-data record, or as soon as the server certificate is parsed (the subject/issuer shortcut that establishes the data phase mid-handshake) - not when the server flight ends. A session that completes the handshake but sends no server app data and no parseable certificate never reaches `server_data`; app data no longer re-enters a completion state. The shortcut is per direction: a direction's own certificate data decides that direction only. In particular, in mutual TLS the server certificate (which arrives before the CertificateRequest) does not push the client-side progression through `client_cert` - the client certificate message is still evaluated at `client_cert`.
- The `server_hello_done` milestone is gone. The server record progression between hello and data is ServerHello, the certificate messages, then the ChangeCipherSpec/Finished flight; the flight carries no inspectable data,

which is why no state exists for it. A rule that wanted to act at "flight complete, before data" now belongs to the tail of `server_cert` or the head of `server_data` depending on intent - the rename hint (`server_hello_done -> server_data`) is approximate in the same way; at the old milestone the state progression was usually still in `server_cert`.

- The firewall applies one decision per state: after an `accept:hook` rule matches at a state, further rules at the same state are not evaluated. Remapping two old rules onto one new state therefore leaves the second one dead - keep at most one rule per state.
- The TLS event carrying the `ja3`/`ja3s` fingerprint fields is a session level event: it flushes as soon as everything it logs is final, that is once a certificate of the state progression has decoded (or the progression advanced beyond the certificate phase, showing it carries none) and, when the server requested a client certificate, the same for the client certificate. While the server flight has not yet revealed whether a client certificate is required - the CertificateRequest normally rides after the server Certificate - the event waits for that to be decided (a message at or past the CertificateRequest position, a change cipher spec, the first application data record, or the close of the handshake; a TLS 1.3 flight is encrypted past the ServerHello, so application data is its mid-flow signal), so an mTLS client certificate and its decode failures are still included, as they were when the old logger flushed at server flight end. A session that never reaches that point publishes the event at connection close. Its packet attribution matches the old model's, because the old logger gated on the same certificate material.
- TLS handshake records often share a TCP segment (e.g. Certificate, CertificateRequest and ServerHelloDone in one flight). Every state walked over by such a packet is evaluated in order, but the packet decision renders at the state the progress reaches: rules of the earlier states in the segment can only accept weakly (mid-walk).
- The state names are also exposed by the `app-layer-state` keyword and in the `ts_progress`/`tc_progress` output fields. The name table changed with this rename: rules using `app-layer-state` against the old TLS state names no longer match, and consumers filtering on the progress fields see the new names.
- SSLv2: a v2 client hello moves the client-side progression to `client_hello`, a v2 server hello moves the server-side progression to `server_hello`, and a v2 client certificate message moves the client-side progression to `client_cert` (the v2 certificate payload itself is not extracted, so cert keywords stay empty there). The old `server_cert_done` and `server_hello_done` milestones are gone. TLS records embedded in a v2 message are parsed by the regular v3 state machine. A rule keyed on the post-hello v2 flight belongs to the state the progression is in when it arrives.
- The per-direction state progressions are monotonic: once a progression reaches `*_data` or `*_finished`, a renegotiated handshake does not re-enter the hello/cert phases on that progression; a second ClientHello does not re-evaluate the `client_hello` rules or re-expose the SNI.
- A ruleset that accepted the pre-hello state and then matched the SNI on the completing record now drops the first fragment of every fragmented hello: the fragment is decided at `client_hello` (not at the pre-hello state as before), it carries no data to match, and an unmatched packet falls on the implicit default policy (drop in firewall mode). If you accept fragmented handshake messages today, add a plain `accept:hook tls:client_hello` / `tls:server_hello` rule for the fragments; only the SNI/other-data rules keep their content match.

24.4.3 Migration steps

1. **Rules.** Search the ruleset for the old names (`client_in_progress`, `client_hello_done`, `client_cert_done`, `client_handshake_done`, `server_in_progress`, `server_hello_done`, `server_cert_done`, `server_handshake_done`) and update them to the new names. A rule with an unknown state fails to load with a `does not support hook` error and is **not enforced**, so check the load output after updating.
2. **Firewall config keys.** Config keys use the **dash** form (`firewall.policies.app.tls.client-hello-done`); rules use the underscore form (`tls:client_hello_done`). A key that no longer matches a state is reported at load: a key known to be a renamed state is an error naming the replacement (fatal when `init-failure-fatal` is set), other unknown keys are a warning and are ignored. Update the keys, or the state falls back to the implicit default policy (drop in firewall mode).

3. **Monitoring.** The `ts_progress` field in alert events now carries the new state names; update any correlation on the old values.

24.5 Migrating from the old SSH state names

In this release the SSH parser was rebuilt on per-direction active phases: a state names the part of the handshake the state progression of one direction is working through - it is entered when that unit starts being parsed and left when the unit completes. The two directions progress independently. The state names used in rules (`accept:hook ssh:<state>`, `alert ssh:<state>`) and in the `firewall.policies.app.ssh` config keys were renamed accordingly, and the removed `banner_wait_eol` state has no replacement: a long banner line now keeps its direction in the banner state. Rules using an old state name fail to load.

24.5.1 Old and new names

old state (milestone)	new state (phase)	note
`request_in_pr`	`request_b`	the banner (version) phase: the first packets of the direction
`request_banne`	`request_b`	the state is gone; a banner line that does not end within 256 bytes keeps the direction in `request_banner` until the end-of-line arrives (the `ssh.long_banner` event)
`request_banne`	`request_k`	entered when the banner line completes; `ssh.proto`, `ssh.software` and the hassh buffers are registered here
`request_finis`	`request_s`	a `new keys` record was seen in this direction; the old top state required both directions to reach it
`response_in_p`	`response_`	the server-side banner phase
`response_bann`	`response_`	as `request_banner_wait_eol` above, in the other direction
`response_bann`	`response_`	as `request_banner_done` above, in the other direction
`response_fini`	`response_`	as `request_finished` above, in the other direction

`request_done` / `response_done` is the registered completion state. No direction ever reports it: a successful direction tops out at `session`, and a failed direction is frozen in the state the failure occurred in. The completion state is neither hookable nor a firewall policy key, so `ssh:request_complete` (and the `request-complete` policy key) matches nothing for a well-formed flow - a session-admit rule like that no longer fires. Only a disrupted flow (depth truncation or async reassembly) reports the completion state, because the engine then returns it as the transaction progress instead of querying the parser. Failures are reported through the `ssh.invalid_banner` / `ssh.invalid_record` events instead.

24.5.2 Semantics that changed with the rename

- The scale is per direction: each direction reports its own raw phase, where the old progress accessor clamped both directions to one shared value. A rule hooking the old top state therefore meant "the flow completed", not "this direction completed the handshake". `session` is now set on the packet's own direction only, when that direction sees a `new keys` record; the two directions are independent.
- A parse failure freezes the failing direction in the state it occurred in: the direction neither advances nor regresses, and the frozen state is the last progress it reports. Nothing after the failing packet is parsed or delivered, mid-flow or at the flow-end flush. The failure reason is available through the `ssh.invalid_banner` / `ssh.invalid_record` app-layer events and the per-direction `error` / `state` fields of the eve `ssh` object (`invalid_banner` / `invalid_record` and the state the failure occurred in; a failed direction is logged even when it parsed no banner). A keyword registered at the failed state gets no eof run, so its flow-end evaluation stays non-terminal.
- A banner line of 256 bytes or more without an end-of-line keeps the direction in the banner state. The line is consumed greedily until the end-of-line arrives, and the direction then advances to `kex`. The condition is

observable through the `ssh.long_banner` event (the banner data is published when the long line first parses). If the line does not parse as a banner when it completes, the direction fails and is frozen in `banner`.

- The firewall applies one decision per state, as for TLS: after an `accept:hook` rule matches at a state, further rules at the same state are not evaluated, and the verdict applies while the transaction stays at the hooked state. Remapping two old rules onto one new state therefore leaves the second one dead - keep at most one rule per state. The old milestones collapsed both directions onto one progress value; a rule now hooks one direction's phase.
- The state names are also exposed by the `app-layer-state` keyword and the `ts_progress` / `tc_progress` output fields. The name table changed with this rename: rules using `app-layer-state` against the old SSH state names no longer match. `done` is out of both name tables, so `app-layer-state:>request_session` matches only a direction disrupted past `session`; a successful direction stops at `request_session` and a failed one is frozen where it failed.
- The generic hook lists for the completion state run only at a disrupted flow end, so Lua hooks on SSH completion no longer fire for well-formed flows; before this change the accessor reported the terminal state once both directions had sent NewKeys.

24.5.3 Migration steps

1. **Rules.** Search the ruleset for the old names (`request_in_progress`, `request_banner_wait_eol`, `request_banner_done`, `request_finished`, and the `response_` equivalents) and update them to the new names. A rule with an unknown state fails to load with a `does not support hook` error and is **not enforced**, so check the load output after updating. A rule that hooked `request_banner_wait_eol` maps to `request_banner`; there is no separate long-line state.
2. **Firewall config keys.** Config keys use the **dash** form (`firewall.policies.app.ssh.request-banner-done`); rules use the underscore form (`ssh:request_banner_done`). Update the keys, or the state falls back to the implicit default policy (drop in firewall mode). Review `request-complete` policies: the completion state is never reported on a clean flow.
3. **``app-layer-state`` and hook users.** Rules and Lua hooks keyed on the old state names no longer match, and a hook on the completion state fires only for a disrupted flow. Use the `ssh.invalid_*` events to observe parse failures.
4. **Monitoring.** The eve `ssh` object now carries the per-direction `error` / `state` fields for a parse failure; a failed direction is logged even when it parsed no banner. The `ts_progress` / `tc_progress` alert fields carry the new names.

24.6 Firewall Mode Unsupported Keywords

Certain rule keywords are not supported in firewall mode, either completely or based on rulesets.

A keyword may be unsupported in firewall rules, in threat detection rules, or in firewall mode altogether.

This is done when there is a chance that the keyword would lead to the firewall verdict being skipped, contradicted, or when there could happen inconsistent states from interactions between firewall and threat detection rules matching.

> **Note**
>
> Future releases may add support for such keywords where it makes sense.

24.6.1 Unsupported Keywords

Firewall Mode

- Bypass
- Replace

Threat Detection Rules

- Config (both the `config` keyword and the `config` action)

CHAPTER

TWENTYFIVE

3RD PARTY INTEGRATION

25.1 Symantec SSL Visibility (BlueCoat)

As Suricata itself cannot decrypt SSL/TLS traffic, some organizations use a decryption product to handle this. This document will offer some advice on using Suricata with the Symantec SSL Visibility appliance (formerly known as BlueCoat).

25.1.1 Appliance Software Version

The appliance comes with two major software version options. The 3.x and 4.x series. Suricata works best with the 4.x series.

TLS1.3 is only properly supported in the 4.x version of the appliance software.

25.1.2 Magic Markers

The appliance has an indicator that data is decrypted. This is done using a special magic source MAC address, or using a special VLAN header. Since Suricata can use VLANs as part of flow tracking, it is recommended to use the source MAC method.

In the 3.x version of the software these markers are always there, the config just allows setting which type will be used. In the 4.x software the markers are optional.

25.1.3 TCP handling

In the 3.x software, a bit of care is required in TCP stream reassembly handling in Suricata. The decrypted traffic is presented to the IDS as TCP data packets, that are not ack'd as regularly as would be expected in a regular TCP session. A large TCP window is used to not violate the TCP specs. Since in IDS mode Suricata waits for ACKs for much of its processing, this can lead to delays in detection and logging, as well as increased resource usage due to increased data buffering.

To avoid this, enable the 'stream.inline' mode, which processed data segments as they come in without waiting for the ACKs.

The 4.x software sends more regular ACKs and does not need any special handling on the Suricata side.

25.1.4 TLS matching in Suricata

The appliance takes care of the TLS handling and decryption, presenting only the decrypted data to Suricata. This means that Suricata will not see the TLS handshake. As a consequence of this, Suricata cannot inspect the TLS handshake or otherwise process it. This means that for decrypted TLS sessions, Suricata will not do any TLS keyword inspection (such as fingerprint matching and ja3), TLS logging or TLS certificate extraction.

If it is important to match on and/or log such information as well, the appliance facilities for matching and logging themselves will have to be used.

For TLS traffic where the appliance security policy does not lead to decryption of the traffic, the TLS handshake is presented to Suricata for analysis and logging.

25.1.5 IPS

When using Suricata in IPS mode with the appliance, some things will have to be considered:

- if Suricata DROPs a packet in the decrypted traffic, this will be seen by the appliance after which it will trigger a RST session teardown.
- if a packet takes more than one second to process, it will automatically be considered a DROP by the appliance. This should not happen in normal traffic, but with very inefficient Lua scripts this could perhaps happen. The appliance can also be configured to wait for 5 seconds.
- When using the Suricata 'replace' keyword to modify data, be aware that the 3.x appliance software will not pass the modification on to the destination so this will not have any effect. The 4.x appliance software does support passing on modifications that were made to the unencrypted text, by default this feature is disabled but you can enable it if you want modifications to be passed on to the destination in the re-encrypted stream. Due to how Suricata works, the size of the payloads cannot be changed.

CHAPTER

TWENTYSIX

MAN PAGES

26.1 Suricata

26.1.1 SYNOPSIS

suricata [OPTIONS] [BPF FILTER]

26.1.2 DESCRIPTION

suricata is a high performance Network IDS, IPS and Network Security Monitoring engine. Open Source and owned by a community run non-profit foundation, the Open Information Security Foundation (OISF).

suricata can be used to analyze live traffic and pcap files. It can generate alerts based on rules. **suricata** will generate traffic logs.

When used with live traffic **suricata** can be passive or active. Active modes are: inline in a L2 bridge setup, inline with L3 integration with host firewall (NFQ, IPFW, WinDivert), or out of band using active responses.

26.1.3 OPTIONS

-h

Display a brief usage overview.

-V

Displays the version of Suricata.

-c <path>

Path to configuration file.

--include <path>

Additional configuration files to include. Multiple additional configuration files can be provided and will be included in the order specified on the command line. These additional configuration files are loaded as if they existed at the end of the main configuration file.

Example including one additional file:

```
--include /etc/suricata/other.yaml
```

Example including more than one additional file:

```
--include /etc/suricata/other.yaml --include /etc/suricata/extra.yaml
```

-T

Test configuration.

`-v`

Increase the verbosity of the Suricata application logging by increasing the log level from the default. This option can be passed multiple times to further increase the verbosity.

- -v: INFO
- -vv: PERF
- -vvv: CONFIG
- -vvvv: DEBUG

This option will not decrease the log level set in the configuration file if it is already more verbose than the level requested with this option.

`-r` <path>

Run in pcap offline mode (replay mode) reading files from pcap file. If <path> specifies a directory, all files in that directory will be processed in order of modified time maintaining flow state between files.

`--pcap-file-continuous`

Used with the -r option to indicate that the mode should stay alive until interrupted. This is useful with directories to add new files and not reset flow state between files.

`--pcap-file-recursive`

Used with the -r option when the path provided is a directory. This option enables recursive traversal into sub-directories to a maximum depth of 255. This option cannot be combined with --pcap-file-continuous. Symlinks are ignored.

`--pcap-file-delete`

Used with the -r option to force deletion of pcap files after they have been processed. This is useful with pcap-file-continuous to continuously feed files to a directory and have them cleaned up when done.

command-line vs Configuration: This command-line option overrides the `pcap-file.delete-when-done` configuration option in `suricata.yaml` and forces "always delete" mode (equivalent to `delete-when-done: true`).

For more control, use the `pcap-file.delete-when-done` configuration option instead, which supports three values:

- `false` (default): No files are deleted
- `true`: All files are deleted after processing
- `"non-alerts"`: Only files that generated no alerts are deleted

If neither `--pcap-file-delete` nor `delete-when-done` is configured, pcap files will not be deleted after processing.

`--pcap-file-buffer-size` <value>

Set read buffer size using `setvbuf` to speed up pcap reading. Valid values are 0, which disables `setvbuf` buffering, or 4 KiB to 64 MiB. Default value is 128 KiB. Supported on Linux only.

`-i` <interface>

After the -i option you can enter the interface card you would like to use to sniff packets from. This option will try to use the best capture method available. Can be used several times to sniff packets from several interfaces.

`--pcap`[=<device>]

Run in PCAP mode. If no device is provided the interfaces provided in the *pcap* section of the configuration file will be used.

--af-packet[=<device>]

Enable capture of packet using AF_PACKET on Linux. If no device is supplied, the list of devices from the af-packet section in the yaml is used.

--af-xdp[=<device>]

Enable capture of packet using AF_XDP on Linux. If no device is supplied, the list of devices from the af-xdp section in the yaml is used.

-q <queue id>

Run inline of the NFQUEUE queue ID provided. May be provided multiple times.

-s <filename.rules>

With the -s option you can set a file with signatures, which will be loaded together with the rules set in the yaml.

It is possible to use globbing when specifying rules files. For example, `-s '/path/to/rules/*.rules'`

-S <filename.rules>

With the -S option you can set a file with signatures, which will be loaded exclusively, regardless of the rules set in the yaml.

It is possible to use globbing when specifying rules files. For example, `-S '/path/to/rules/*.rules'`

-l <directory>

With the -l option you can set the default log directory. If you already have the default-log-dir set in yaml, it will not be used by Suricata if you use the -l option. It will use the log dir that is set with the -l option. If you do not set a directory with the -l option, Suricata will use the directory that is set in yaml.

-D

Normally if you run Suricata on your console, it keeps your console occupied. You can not use it for other purposes, and when you close the window, Suricata stops running. If you run Suricata as daemon (using the -D option), it runs at the background and you will be able to use the console for other tasks without disturbing the engine running.

--runmode <runmode>

With the *--runmode* option you can set the runmode that you would like to use. This command line option can override the yaml runmode option.

Runmodes are: *workers*, *autofp* and *single*.

For more information about runmodes see *Runmodes* in the user guide.

-F <bpf filter file>

Use BPF filter from file.

-k [all|none]

Force (all) the checksum check or disable (none) all checksum checks.

--user=<user>

Set the process user after initialization. Overrides the user provided in the *run-as* section of the configuration file.

--group=<group>

Set the process group to group after initialization. Overrides the group provided in the *run-as* section of the configuration file.

--pidfile <file>

Write the process ID to file. Overrides the *pid-file* option in the configuration file and forces the file to be written when not running as a daemon.

--init-errors-fatal

Exit with a failure when errors are encountered loading signatures.

--strict-rule-keywords[=all|<keyword>|<keywords(csv)]

Applies to: classtype, reference and app-layer-event.

By default missing reference or classtype values are warnings and not errors. Additionally, loading outdated app-layer-event events are also not treated as errors, but as warnings instead.

If this option is enabled these warnings are considered errors.

If no value, or the value 'all', is specified, the option applies to all of the keywords above. Alternatively, a comma separated list can be supplied with the keyword names it should apply to.

--disable-detection

Disable the detection engine.

--disable-hashing

Disable support for hash algorithms such as md5, sha1 and sha256.

By default hashing is enabled. Disabling hashing will also disable some Suricata features such as the filestore, ja3, and rule keywords that use hash algorithms.

--dump-config

Dump the configuration loaded from the configuration file to the terminal and exit.

--dump-features

Dump the features provided by Suricata modules and exit. Features list (a subset of) the configuration values and are intended to assist with comparing provided features with those required by one or more rules.

--build-info

Display the build information the Suricata was built with.

--list-app-layer-protos

List all supported application layer protocols.

--list-keywords=[all|json|csv|<kword>]

List all supported rule keywords.

--list-runmodes

List all supported run modes.

--plugin <path>

Load a plugin from *path* in addition to the plugins listed in the configuration file. This option can be specified multiple times.

If *path* is a directory, Suricata will attempt to load each `.so` file in that directory.

--set <key>=<value>

Set a configuration value. Useful for overriding basic configuration parameters. For example, to change the default log directory:

```
--set default-log-dir=/var/tmp
```

This option cannot be used to add new entries to a list in the configuration file, such as a new output. It can only be used to modify a value in a list that already exists.

For example, to disable the `eve-log` in the default configuration file:

```
--set outputs.1.eve-log.enabled=no
```

Also note that the index values may change as the `suricata.yaml` is updated.

See the output of `--dump-config` for existing values that could be modified with their index.

--engine-analysis

Print reports on analysis of different sections in the engine and exit. Please have a look at the conf parameter engine-analysis on what reports can be printed

--unix-socket=<file>

Use file as the Suricata unix control socket. Overrides the *filename* provided in the *unix-command* section of the configuration file.

--reject-dev=<device>

Use *device* to send out RST / ICMP error packets with the *reject* keyword.

--pcap-buffer-size=<size>

Set the size of the PCAP buffer (0 - 2147483647).

--netmap[=<device>]

Enable capture of packet using NETMAP on FreeBSD or Linux. If no device is supplied, the list of devices from the netmap section in the yaml is used.

--pfring[=<device>]

Enable PF_RING packet capture. If no device provided, the devices in the Suricata configuration will be used.

--pfring-cluster-id <id>

Set the PF_RING cluster ID.

--pfring-cluster-type <type>

Set the PF_RING cluster type (cluster_round_robin, cluster_flow).

-d <divert-port>

Run inline using IPFW divert mode.

--dag <device>

Enable packet capture off a DAG card. If capturing off a specific stream the stream can be select using a device name like "dag0:4". This option may be provided multiple times read off multiple devices and/or streams.

--napatech

Enable packet capture using the Napatech Streams API.

--erf-in=<file>

Run in offline mode reading the specific ERF file (Endace extensible record format).

--simulate-ips

Simulate IPS mode when running in a non-IPS mode.

26.1.4 OPTIONS FOR DEVELOPERS

-u

Run the unit tests and exit. Requires that Suricata be configured with *--enable-unittests*.

-U, --unittest-filter=REGEX

With the -U option you can select which of the unit tests you want to run. This option uses REGEX. Example of use: suricata -u -U http

--list-unittests
Lists available unit tests.

--fatal-unittests
Enables fatal failure on a unit test error. Suricata will exit instead of continuing more tests.

--unittests-coverage
Display unit test coverage report.

26.1.5 SIGNALS

Suricata will respond to the following signals:

SIGUSR2

Causes Suricata to perform a live rule reload.

SIGHUP

Causes Suricata to close and re-open all log files. This can be used to re-open log files after they may have been moved away by log rotation utilities.

On Windows, when running as a service, the service control codes 128 and 129 are the equivalent of SIGUSR2 and SIGHUP respectively.

26.1.6 FILES AND DIRECTORIES

/etc/suricata/suricata.yaml
Default location of the Suricata configuration file.

/var/log/suricata
Default Suricata log directory.

26.1.7 EXAMPLES

To capture live traffic from interface *eno1*:

```
suricata -i eno1
```

To analyze a pcap file and output logs to the CWD:

```
suricata -r /path/to/capture.pcap
```

To capture using *AF_PACKET* and override the flow memcap setting from the *suricata.yaml*:

```
suricata --af-packet --set flow.memcap=1gb
```

To analyze a pcap file with a custom rule file:

```
suricata -r /pcap/to/capture.pcap -S /path/to/custom.rules
```

26.1.8 BUGS

Please visit Suricata's support page for information about submitting bugs or feature requests.

26.1.9 NOTES

- Suricata Home Page

  https://suricata.io/

- Suricata Support Page

  https://suricata.io/support/

26.2 Suricata Socket Control

26.2.1 SYNOPSIS

suricatasc [OPTIONS] [SOCKET]

26.2.2 DESCRIPTION

Suricata socket control tool

26.2.3 SOCKET

Optional path to Suricata unix socket

26.2.4 OPTIONS

-v, --verbose
: Enable verbose output

-c, --command <COMMAND>
: Execute command and return JSON

-h, --help
: Print help

26.2.5 COMMANDS

shutdown
: Shut Suricata instance down.

command-list
: List available commands.

help
: Get help about the available commands.

version
: Print the version of Suricata instance.

uptime
: Display the uptime of Suricata.

running-mode
: Display running mode. This can either be *workers*, *autofp* or *single*.

capture-mode

Display the capture mode. This can be either of *PCAP_DEV*, *PCAP_FILE*, *PFRING(DISABLED)*, *NFQ*, *NFLOG*, *IPFW*, *ERF_FILE*, *ERF_DAG*, *AF_PACKET_DEV*, *NETMAP(DISABLED)*, *UNIX_SOCKET* or *WINDIVERT(DISABLED)*.

conf-get <variable>

Get configuration value for a given variable. Variable to be provided can be either of the configuration parameters that are written in suricata.yaml.

dump-counters

Dump Suricata's performance counters.

ruleset-reload-rules

Reload the ruleset and wait for completion.

reload-rules

Alias .. describe *ruleset-reload-rules*.

ruleset-reload-nonblocking

Reload ruleset and proceed without waiting.

ruleset-reload-time

Return time of last reload.

ruleset-stats

Display the number of rules loaded and failed.

ruleset-failed-rules

Display the list of failed rules.

register-tenant-handler <id> <htype> [hargs]

Register a tenant handler with the specified mapping.

unregister-tenant-handler <id> <htype> [hargs]

Unregister a tenant handler with the specified mapping.

register-tenant <id> <filename>

Register tenant with a particular ID and filename.

reload-tenant <id> [filename]

Reload a tenant with specified ID. A filename to a tenant yaml can be specified. If it is omitted, the original yaml that was used to load / last reload the tenant is used.

reload-tenants

Reload all registered tenants by reloading their yaml.

unregister-tenant <id>

Unregister tenant with a particular ID.

add-hostbit <ipaddress> <hostbit> <expire>

Add hostbit on a host IP with a particular bit name and time of expiry.

remove-hostbit <ipaddress> <hostbit>

Remove hostbit on a host IP with specified IP address and bit name.

list-hostbit <ipaddress>

List hostbit for a particular host IP.

reopen-log-files

Reopen log files to be run after external log rotation.

memcap-set <config> <memcap>

Update memcap value of a specified item.

memcap-show <config>

Show memcap value of a specified item.

memcap-list

List all memcap values available.

get-flow-stats-by-id <flow_id>

Display information for a specific flow using `flow_id` values.

26.2.6 PCAP MODE COMMANDS

pcap-file <file> <dir> [tenant] [continuous] [delete-when-done]

Add pcap files to Suricata for sequential processing. The generated log/alert files will be put into the directory specified as second argument. Make sure to provide absolute path to the files and directory. It is acceptable to add multiple files without waiting the result.

pcap-file-continuous <file> <dir> [tenant] [delete-when-done]

Add pcap files to Suricata for sequential processing. Directory will be monitored for new files being added until there is a use of **pcap-interrupt** or directory is moved or deleted.

pcap-file-number

Number of pcap files waiting to get processed.

pcap-file-list

List of queued pcap files.

pcap-last-processed

Processed time of last file in milliseconds since epoch.

pcap-interrupt

Terminate the current state by interrupting directory processing.

pcap-current

Currently processed file.

26.2.7 BUGS

Please visit Suricata's support page for information about submitting bugs or feature requests.

26.2.8 NOTES

- Suricata Home Page

  https://suricata.io/

- Suricata Support Page

  https://suricata.io/support/

26.3 Suricata Control

26.3.1 SYNOPSIS

suricatactl [-h] <command> [<args>]

26.3.2 DESCRIPTION

This tool helps control Suricata's features.

26.3.3 OPTIONS

-h

Get help about the available commands.

26.3.4 COMMANDS

suricatactl-filestore(1)

26.3.5 BUGS

Please visit Suricata's Community page for information about submitting bugs or feature requests.

26.3.6 NOTES

- Suricata Home Page

  https://suricata.io/

- Suricata Community Page

  https://suricata.io/community/

26.4 Suricata Control Filestore

26.4.1 SYNOPSIS

suricatactl filestore [-h] <command> [<args>]

26.4.2 DESCRIPTION

This command lets you perform certain operations on Suricata filestore.

26.4.3 OPTIONS

-h

Get help about the available commands.

26.4.4 COMMANDS

prune [-h|--help] [-n|--dry-run] [-v|verbose] [-q|--quiet] -d <DIRECTORY> --age <AGE>

Prune files older than a given age.

-d <DIRECTORY> | --directory <DIRECTORY> is a required argument which tells that user must provide the suricata filestore directory on which all the specified operations are to be performed.

--age <AGE> is a required argument asking the age of the files. Files older than the age mentioned with this option shall be pruned.

-h | --help is an optional argument with which you can ask for help about the command usage.

-n | --dry-run is an optional argument which makes the utility print only what would happen

-v | --verbose is an optional argument to increase the verbosity of command.

-q | --quiet is an optional argument that helps log errors and warnings only and keep silent about everything else.

26.4.5 BUGS

Please visit Suricata's support page for information about submitting bugs or feature requests.

26.4.6 NOTES

- Suricata Home Page

  https://suricata.io/

- Suricata Support Page

  https://suricata.io/support/

CHAPTER

TWENTYSEVEN

ACKNOWLEDGEMENTS

Thank you to the following for their Wiki and documentation contributions that have made this user guide possible:

- Aaron Bungay
- Abbed
- Alex Savage
- Alexandre Iooss
- Alfredo Cardigliano
- Alice Akaki
- AlirezaPourchali
- Andi
- Andreas Dolp
- Andreas Herz
- Andreas Moe
- Anne-Fleur Koolstra
- Bazzan Don
- Benjamin Wilkins
- Bill Meeks
- Brandon Sterne
- brandonlattin
- Bryant Smith
- Carl Smith
- Chris Speidel
- Christophe Vandeplas
- Cole Dishington
- Comfort Amaechi
- Daisu
- Dana Helwig
- Daniel Olatunji

- Danny Browning
- Darren Spruell
- David Cannings
- David Diallo
- David Wharton
- Dean Balandin
- Eloy Pérez González
- Eloïse Brocas
- EmilienCourt
- Emmanuel Thompson
- Eric Leblond
- fooinha
- Frank Honza
- Gaurav Singh
- Giuseppe Longo
- god lol
- Hadiqa Alamdar Bukhari
- Haleema Khan
- Haris Haq
- Hilko Bengen
- Ignacio Sanchez
- Jacob Masen-Smith
- James
- James Dutrisac
- Jason Ish
- Jason Taylor
- Jason Williams
- Jeff Lucovsky
- Jeremy MountainJohnson
- Jo Johnson
- John Dewey
- Jon Zeolla
- Josh Smith
- Josh Stroschein
- Joshua Lumb
- Joyce Yu

- Julian
- Juliana Fajardini
- Justin Ossevoort
- Ken Steele
- Kirby Kuehl
- Kirjan Kohuladas
- Konstantin Klinger
- kwong
- Lancer Cheng
- Les Syv
- liaozhiyuan
- Lukas Sismis
- Mark Solaris
- Martin Holste
- Martin Natano
- Mats Klepsland
- Matt Jonkman
- Maurizio Abba
- Max Fillinger
- Michael Bentley
- Michael Hrishenko
- Morris Chan
- myr463
- Nathan Jimerson
- Nathan Scrivens
- Nick Price
- Nicolas Merle
- Nicolas Thill
- Odin Jenseg
- Pascal Delalande
- Peter Manev
- Peter Sanders
- Phil Young
- Philipp Buehler
- Philippe Antoine
- Pierre Chifflier

- Rafael Girão
- Ralph Broenink
- Ralph Eastwood
- Ray Ruvinskiy
- Richard McConnell
- Richard Sailer
- Rob MacGregor
- Ruslan Usmanov
- Russel Fulton
- Sascha Steinbiss
- Sebastian Garcia
- Selivanov Pavel
- Shivani Bhardwaj
- showipintbri
- Simon Dugas
- Steven Hostetler
- Todd Mortimer
- Travis Green
- Tristan Fletcher
- Vadym Malakhatko
- Victor Julien
- Vincent Fang
- Vinjar Hillestad
- Wes Hurd
- William Harding
- Zach Kelly
- Zach Rasmor

CHAPTER

TWENTYEIGHT

LICENSES

28.1 GNU General Public License

Version 2, June 1991 Copyright © 1989, 1991 Free Software Foundation, Inc. 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

28.1.1 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Lesser General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: **(1)** copyright the software, and **(2)** offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

28.1.2 TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

- **a)** You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
- **b)** You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
- **c)** If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

- **a)** Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

- **b)** Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- **c)** Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical

distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

28.1.3 NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

28.1.4 How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> <one line to give the program's name and a brief idea of what it does.> Copyright (C) <year> <name of author>
>
> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

> You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

> Gnomovision version 69, Copyright (C) year name of author Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'. This is free software, and you are welcome to redistribute it under certain conditions; type `show c' for details.

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c`; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the program `Gnomovision' (which makes passes at compilers) written by James Hacker.
>
> <signature of Ty Coon>, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License.

28.2 Creative Commons Attribution-NonCommercial 4.0 International Public License

By exercising the Licensed Rights (defined below), You accept and agree to be bound by the terms and conditions of this Creative Commons Attribution-NonCommercial 4.0 International Public License ("Public License"). To the extent this Public License may be interpreted as a contract, You are granted the Licensed Rights in consideration of Your acceptance of these terms and conditions, and the Licensor grants You such rights in consideration of benefits the Licensor receives from making the Licensed Material available under these terms and conditions.

28.2.1 Section 1 – Definitions.

a. **Adapted Material** means material subject to Copyright and Similar Rights that is derived from or based upon the Licensed Material and in which the Licensed Material is translated, altered, arranged, transformed, or otherwise modified in a manner requiring permission under the Copyright and Similar Rights held by the Licensor. For purposes of this Public License, where the Licensed Material is a musical work, performance, or sound recording, Adapted Material is always produced where the Licensed Material is synched in timed relation with a moving image.

b. **Adapter's License** means the license You apply to Your Copyright and Similar Rights in Your contributions to Adapted Material in accordance with the terms and conditions of this Public License.

c. **Copyright and Similar Rights** means copyright and/or similar rights closely related to copyright including, without limitation, performance, broadcast, sound recording, and Sui Generis Database Rights, without regard to how the rights are labeled or categorized. For purposes of this Public License, the rights specified in Section 2(b)(1)-(2) are not Copyright and Similar Rights.

d. **Effective Technological Measures** means those measures that, in the absence of proper authority, may not be circumvented under laws fulfilling obligations under Article 11 of the WIPO Copyright Treaty adopted on December 20, 1996, and/or similar international agreements.

e. **Exceptions and Limitations** means fair use, fair dealing, and/or any other exception or limitation to Copyright and Similar Rights that applies to Your use of the Licensed Material.

f. **Licensed Material** means the artistic or literary work, database, or other material to which the Licensor applied this Public License.

g. **Licensed Rights** means the rights granted to You subject to the terms and conditions of this Public License, which are limited to all Copyright and Similar Rights that apply to Your use of the Licensed Material and that the Licensor has authority to license.

h. **Licensor** means the individual(s) or entity(ies) granting rights under this Public License.

i. **NonCommercial** means not primarily intended for or directed towards commercial advantage or monetary compensation. For purposes of this Public License, the exchange of the Licensed Material for other material subject to Copyright and Similar Rights by digital file-sharing or similar means is NonCommercial provided there is no payment of monetary compensation in connection with the exchange.

j. **Share** means to provide material to the public by any means or process that requires permission under the Licensed Rights, such as reproduction, public display, public performance, distribution, dissemination, communication, or importation, and to make material available to the public including in ways that members of the public may access the material from a place and at a time individually chosen by them.

k. **Sui Generis Database Rights** means rights other than copyright resulting from Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, as amended and/or succeeded, as well as other essentially equivalent rights anywhere in the world.

l. **You** means the individual or entity exercising the Licensed Rights under this Public License. Your has a corresponding meaning.

28.2.2 Section 2 – Scope.

a. **License grant.**

1. Subject to the terms and conditions of this Public License, the Licensor hereby grants You a worldwide, royalty-free, non-sublicensable, non-exclusive, irrevocable license to exercise the Licensed Rights in the Licensed Material to:

A. reproduce and Share the Licensed Material, in whole or in part, for NonCommercial purposes only; and

B. produce, reproduce, and Share Adapted Material for NonCommercial purposes only.

2. **Exceptions and Limitations.** For the avoidance of doubt, where Exceptions and Limitations apply to Your use, this Public License does not apply, and You do not need to comply with its terms and conditions.

3. **Term.** The term of this Public License is specified in Section 6(a).

4. **Media and formats; technical modifications allowed.** The Licensor authorizes You to exercise the Licensed Rights in all media and formats whether now known or hereafter created, and to make technical modifications necessary to do so. The Licensor waives and/or agrees not to assert any right or authority to forbid You from making technical modifications necessary to exercise the Licensed Rights, including technical modifications necessary to circumvent Effective Technological Measures. For purposes of this Public License, simply making modifications authorized by this Section 2(a)(4) never produces Adapted Material.

5. **Downstream recipients.**

A. **Offer from the Licensor – Licensed Material.** Every recipient of the Licensed Material automatically receives an offer from the Licensor to exercise the Licensed Rights under the terms and conditions of this Public License.

B. **No downstream restrictions.** You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, the Licensed Material if doing so restricts exercise of the Licensed Rights by any recipient of the Licensed Material.

6. **No endorsement.** Nothing in this Public License constitutes or may be construed as permission to assert or imply that You are, or that Your use of the Licensed Material is, connected with, or sponsored, endorsed, or granted official status by, the Licensor or others designated to receive attribution as provided in Section 3(a)(1)(A)(i).

b. **Other rights.**

1. Moral rights, such as the right of integrity, are not licensed under this Public License, nor are publicity, privacy, and/or other similar personality rights; however, to the extent possible, the Licensor waives and/or agrees not to assert any such rights held by the Licensor to the limited extent necessary to allow You to exercise the Licensed Rights, but not otherwise.
2. Patent and trademark rights are not licensed under this Public License.
3. To the extent possible, the Licensor waives any right to collect royalties from You for the exercise of the Licensed Rights, whether directly or through a collecting society under any voluntary or waivable statutory or compulsory licensing scheme. In all other cases the Licensor expressly reserves any right to collect such royalties, including when the Licensed Material is used other than for NonCommercial purposes.

28.2.3 Section 3 – License Conditions.

Your exercise of the Licensed Rights is expressly made subject to the following conditions.

a. **Attribution.**

1. If You Share the Licensed Material (including in modified form), You must:

A. retain the following if it is supplied by the Licensor with the Licensed Material:

i. identification of the creator(s) of the Licensed Material and any others designated to receive attribution, in any reasonable manner requested by the Licensor (including by pseudonym if designated);

ii. a copyright notice;

iii. a notice that refers to this Public License;

iv. a notice that refers to the disclaimer of warranties;

v. a URI or hyperlink to the Licensed Material to the extent reasonably practicable;

B. indicate if You modified the Licensed Material and retain an indication of any previous modifications; and

C. indicate the Licensed Material is licensed under this Public License, and include the text of, or the URI or hyperlink to, this Public License.

2. You may satisfy the conditions in Section 3(a)(1) in any reasonable manner based on the medium, means, and context in which You Share the Licensed Material. For example, it may be reasonable to satisfy the conditions by providing a URI or hyperlink to a resource that includes the required information.
3. If requested by the Licensor, You must remove any of the information required by Section 3(a)(1)(A) to the extent reasonably practicable.
4. If You Share Adapted Material You produce, the Adapter's License You apply must not prevent recipients of the Adapted Material from complying with this Public License.

28.2.4 Section 4 – Sui Generis Database Rights.

Where the Licensed Rights include Sui Generis Database Rights that apply to Your use of the Licensed Material:

a. for the avoidance of doubt, Section 2(a)(1) grants You the right to extract, reuse, reproduce, and Share all or a substantial portion of the contents of the database for NonCommercial purposes only;

b. if You include all or a substantial portion of the database contents in a database in which You have Sui Generis Database Rights, then the database in which You have Sui Generis Database Rights (but not its individual contents) is Adapted Material; and

c. You must comply with the conditions in Section 3(a) if You Share all or a substantial portion of the contents of the database.

For the avoidance of doubt, this Section 4 supplements and does not replace Your obligations under this Public License where the Licensed Rights include other Copyright and Similar Rights.

28.2.5 Section 5 – Disclaimer of Warranties and Limitation of Liability.

a. **Unless otherwise separately undertaken by the Licensor, to the extent possible, the Licensor offers the Licensed Material as-is and as-available, and makes no representations or warranties of any kind concerning the Licensed Material, whether express, implied, statutory, or other. This includes, without limitation, warranties of title, merchantability, fitness for a particular purpose, non-infringement, absence of latent or other defects, accuracy, or the presence or absence of errors, whether or not known or discoverable. Where disclaimers of warranties are not allowed in full or in part, this disclaimer may not apply to You.**

b. **To the extent possible, in no event will the Licensor be liable to You on any legal theory (including, without limitation, negligence) or otherwise for any direct, special, indirect, incidental, consequential, punitive, exemplary, or other losses, costs, expenses, or damages arising out of this Public License or use of the Licensed Material, even if the Licensor has been advised of the possibility of such losses, costs, expenses, or damages. Where a limitation of liability is not allowed in full or in part, this limitation may not apply to You.**

c. The disclaimer of warranties and limitation of liability provided above shall be interpreted in a manner that, to the extent possible, most closely approximates an absolute disclaimer and waiver of all liability.

28.2.6 Section 6 – Term and Termination.

a. This Public License applies for the term of the Copyright and Similar Rights licensed here. However, if You fail to comply with this Public License, then Your rights under this Public License terminate automatically.

b. Where Your right to use the Licensed Material has terminated under Section 6(a), it reinstates:

   1. automatically as of the date the violation is cured, provided it is cured within 30 days of Your discovery of the violation; or

   2. upon express reinstatement by the Licensor.

   For the avoidance of doubt, this Section 6(b) does not affect any right the Licensor may have to seek remedies for Your violations of this Public License.

c. For the avoidance of doubt, the Licensor may also offer the Licensed Material under separate terms or conditions or stop distributing the Licensed Material at any time; however, doing so will not terminate this Public License.

d. Sections 1, 5, 6, 7, and 8 survive termination of this Public License.

28.2.7 Section 7 – Other Terms and Conditions.

a. The Licensor shall not be bound by any additional or different terms or conditions communicated by You unless expressly agreed.

b. Any arrangements, understandings, or agreements regarding the Licensed Material not stated herein are separate from and independent of the terms and conditions of this Public License.

28.2.8 Section 8 – Interpretation.

a. For the avoidance of doubt, this Public License does not, and shall not be interpreted to, reduce, limit, restrict, or impose conditions on any use of the Licensed Material that could lawfully be made without permission under this Public License.

b. To the extent possible, if any provision of this Public License is deemed unenforceable, it shall be automatically reformed to the minimum extent necessary to make it enforceable. If the provision cannot be reformed, it shall be severed from this Public License without affecting the enforceability of the remaining terms and conditions.

c. No term or condition of this Public License will be waived and no failure to comply consented to unless expressly agreed to by the Licensor.

d. Nothing in this Public License constitutes or may be interpreted as a limitation upon, or waiver of, any privileges and immunities that apply to the Licensor or You, including from the legal processes of any jurisdiction or authority.

Creative Commons is not a party to its public licenses. Notwithstanding, Creative Commons may elect to apply one of its public licenses to material it publishes and in those instances will be considered the "Licensor." Except for the limited purpose of indicating that material is shared under a Creative Commons public license or as otherwise permitted by the Creative Commons policies published at creativecommons.org/policies, Creative Commons does not authorize the use of the trademark "Creative Commons" or any other trademark or logo of Creative Commons without its prior written consent including, without limitation, in connection with any unauthorized modifications to any of its public licenses or any other arrangements, understandings, or agreements concerning use of licensed material. For the avoidance of doubt, this paragraph does not form part of the public licenses.

Creative Commons may be contacted at creativecommons.org.

28.3 Suricata Source Code

The Suricata source code is licensed under version 2 of the *GNU General Public License*.

28.4 Suricata Documentation

The Suricata documentation (this documentation) is licensed under the *Creative Commons Attribution-NonCommercial 4.0 International Public License*.

CHAPTER

TWENTYNINE

SURICATA DEVELOPER GUIDE

29.1 Working with the Codebase

29.1.1 Installation from GIT

Ubuntu Installation from GIT

This document will explain how to install and use the most recent code of Suricata on Ubuntu. Installing from GIT on other operating systems is basically the same, except that some commands are Ubuntu-specific (like sudo and apt-get). In case you are using another operating system, you should replace those commands with your OS-specific commands.

> **Note**
>
> These instructions were tested on Ubuntu 22.04.

Pre-installation requirements

Before you can build Suricata for your system, run the following command to ensure that you have everything you need for the installation.

```
sudo apt-get -y install libpcre2-dev build-essential autoconf \
automake libtool libpcap-dev libnet1-dev libyaml-0-2 libyaml-dev \
pkg-config zlib1g zlib1g-dev libcap-ng-dev libcap-ng0 make \
libmagic-dev libjansson-dev rustc cargo jq git-core
```

Add `${HOME}/.cargo/bin` to your path:

```
export PATH=$PATH:${HOME}/.cargo/bin
cargo install --force cbindgen
```

Depending on the current status of your system, it may take a while to complete this process.

IPS

By default, Suricata works as an IDS. If you want to use it as an IDS and IPS program, enter:

```
sudo apt-get -y install libnetfilter-queue-dev libnetfilter-queue1 \
libnfnetlink-dev libnfnetlink0
```

Suricata

First, it is convenient to create a directory for Suricata. Name it 'suricata' or 'oisf', for example. Open the terminal and enter:

```
mkdir suricata  # mkdir oisf
```

Followed by:

```
cd suricata  # cd oisf
```

Next, enter the following line in the terminal:

```
git clone https://github.com/OISF/suricata.git
cd suricata
```

Suricata-update is not bundled. Get it by doing:

```
./scripts/bundle.sh
```

Followed by:

```
./autogen.sh
```

To configure, please enter:

```
./configure
```

To compile, please enter:

```
make
```

To install Suricata, enter:

```
sudo make install
sudo ldconfig
```

Auto-setup

You can also use the available auto-setup features of Suricata. Ex:

```
./configure && make && sudo make install-conf
```

make install-conf would do the regular "make install" and then it would automatically create/setup all the necessary directories and `suricata.yaml` for you.

```
./configure && make && make install-rules
```

make install-rules would do the regular "make install" and then it would automatically download and set-up the latest ruleset from Emerging Threats available for Suricata.

```
./configure && make && make install-full
```

make install-full would combine everything mentioned above (install-conf and install-rules) - and will present you with a ready to run (configured and set-up) Suricata.

Post installation

Please continue with *Basic setup*.

In case you have already created your Suricata directory and cloned the repository in it, if you want to update your local repository with the most recent code, please run:

```
cd suricata/suricata
```

next, enter:

```
git pull
```

After that, you should run *./autogen.sh* again.

29.1.2 Coding Style

Suricata uses a fairly strict coding style. This document describes it.

Formatting

clang-format

`clang-format` is configured to help you with formatting C code.

> **Note**
>
> The `.clang-format` script requires clang 9 or newer. At this time `clang-format-14` is used to validate formatting in CI.

Format your Changes

Before opening a pull request, please also try to ensure it is formatted properly. We use `clang-format` for this, which has git integration through the `git-clang-format` script to only format your changes.

On some systems, it may already be installed (or be installable via your package manager). If so, you can simply run it.

It is recommended to format each commit as you go. However, you can always reformat your whole branch after the fact.

> **Note**
>
> Depending on your installation, you might have to use the version-specific `git clang-format` in the commands below, e.g. `git clang-format-14`, and possibly even provide the `clang-format` binary with `--binary clang-format-14`.
>
> As an alternative, you can use the provided `scripts/clang-format.sh` that isolates you from the different versions.

Formatting the most recent commit only

The following command will format only the code changed in the most recent commit:

```
$ git clang-format HEAD^
# Or with script:
$ scripts/clang-format.sh commit
```

Note that this modifies the files, but doesn't commit them. If the changes are trivial, you'll likely want to run

```
$ git commit --amend -a
```

in order to update the last commit with all pending changes.

For bigger formatting changes, we do ask you to add them to separate, dedicated commits.

Formatting code in staging

The following command will format the changes in staging, i.e. files you `git add`-ed:

```
$ git clang-format
# Or with script:
$ scripts/clang-format.sh cached
```

If you also want to change the unstaged changes, do:

```
$ git clang-format --force
# Or with script:
$ scripts/clang-format.sh cached --force
```

Formatting your branch's commits

In case you have multiple commits on your branch already and forgot to format them you can fix that up as well.

The following command will format every commit in your branch off main and rewrite history using the existing commit metadata.

Tip: Create a new version of your branch first and run this off the new version.

```
# In a new version of your pull request:
$ scripts/clang-format.sh rewrite-branch
```

Note that the above should only be used for rather minimal formatting changes. As mentioned, we prefer that you add such changes to a dedicated commit for formatting changes:

```
# Format all changes by commits on your branch:
$ git clang-format first_commit_on_your_branch^
# Or with script:
$ scripts/clang-format.sh branch
```

Note the usage of `first_commit_on_your_branch^`, not `main`, to avoid picking up new commits on `main` in case you've updated main since you've branched.

Check formatting

Check if your branch changes' formatting is correct with:

```
$ scripts/clang-format.sh check-branch
```

Add the `--diffstat` parameter if you want to see the files needing formatting. Add the `--diff` parameter if you want to see the actual diff of the formatting change.

Formatting a whole file

> **Note**
> Do not reformat whole files by default, i.e. do not use `clang-format` proper in general.

If you were ever to do so, formatting changes of existing code with clang-format shall be a different commit and must not be mixed with actual code changes.

```
$ clang-format -i {file}
```

Disabling clang-format

There might be times, where the clang-format's formatting might not please. This might mostly happen with macros, arrays (single or multi-dimensional ones), struct initialization, or where one manually formatted code.

You can always disable clang-format.

```
/* clang-format off */
#define APP_LAYER_INCOMPLETE(c, n) (AppLayerResult){1, (c), (n)}
/* clang-format on */
```

Installing clang-format and git-clang-format

clang-format 9 or newer is required.

On Ubuntu 24.04:

- It is sufficient to only install clang-format, e.g.

  ```
  $ sudo apt-get install clang-format-14
  ```

- See http://apt.llvm.org for other releases in case the clang-format version is not found in the default repos.

On Fedora:

- Install the `clang` and `git-clang-format` packages with

  ```
  $ sudo dnf install clang git-clang-format
  ```

Line length

Limit line lengths to 100 characters.

When wrapping lines that are too long, they should be indented at least 8 spaces from previous line. You should attempt to wrap the minimal portion of the line to meet the 100 character limit.

clang-format:

- ColumnLimit: 100
- ContinuationIndentWidth: 8
- ReflowComments: true

Indent

We use 4 space indentation.

```
int DecodeEthernet(ThreadVars *tv, DecodeThreadVars *dtv, Packet *p,
        uint8_t *pkt, uint16_t len, PacketQueue *pq)
{
    SCPerfCounterIncr(dtv->counter_eth, tv->sc_perf_pca);

    if (unlikely(len < ETHERNET_HEADER_LEN)) {
        ENGINE_SET_INVALID_EVENT(p, ETHERNET_PKT_TOO_SMALL);
        return TM_ECODE_FAILED;
    }

    ...

    DecodeNetworkLayer(tv, dtv, SCNtohs(p->ethh->eth_type), p,
            pkt + ETHERNET_HEADER_LEN, len - ETHERNET_HEADER_LEN);

    return TM_ECODE_OK;
}
```

Use 8 space indentation when wrapping function parameters, loops and if statements.

Use 4 space indentation when wrapping variable definitions.

```
const SCPlugin PluginSpec = {
    .name = OUTPUT_NAME,
    .author = "Some Developer",
    .license = "GPLv2",
    .Init = TemplateInit,
};
```

clang-format:

- AlignAfterOpenBracket: DontAlign
- Cpp11BracedListStyle: false
- IndentWidth: 4
- TabWidth: 8 [llvm]
- UseTab: Never [llvm]

Braces

Functions should have the opening brace on a newline:

```
int SomeFunction(void)
{
    DoSomething();
}
```

Note: you may encounter non-compliant code.

Control and loop statements should have the opening brace on the same line:

```
if (unlikely(len < ETHERNET_HEADER_LEN)) {
    ENGINE_SET_INVALID_EVENT(p, ETHERNET_PKT_TOO_SMALL);
    return TM_ECODE_FAILED;
}

for (ascii_code = 0; ascii_code < 256; ascii_code++) {
    ctx->goto_table[ctx->state_count][ascii_code] = SC_AC_FAIL;
}

while (funcs != NULL) {
    temp = funcs;
    funcs = funcs->next;
    SCFree(temp);
}
```

Opening and closing braces go on the same line as as the _else_ (also known as a "cuddled else").

```
if (this) {
    DoThis();
} else {
    DoThat();
}
```

Structs, unions and enums should have the opening brace on the same line:

```
union {
    TCPVars tcpvars;
    ICMPV4Vars icmpv4vars;
    ICMPV6Vars icmpv6vars;
} l4vars;

struct {
    uint8_t type;
    uint8_t code;
} icmp_s;

enum {
    DETECT_TAG_TYPE_SESSION,
    DETECT_TAG_TYPE_HOST
};
```

clang-format:

- BreakBeforeBraces: Custom [breakbeforebraces]
- BraceWrapping:
  - AfterClass: true
  - AfterControlStatement: false
  - AfterEnum: false
  - AfterFunction: true
  - AfterStruct: false
  - AfterUnion: false

- AfterExternBlock: true
- BeforeElse: false
- IndentBraces: false

Flow

Don't use conditions and statements on the same line. E.g.

```
if (a) b = a; // <- wrong

if (a)
    b = a; // <- right

for (int i = 0; i < 32; ++i) f(i); // <- wrong

for (int i = 0; i < 32; ++i)
    f(i); // <- right
```

Don't put short or empty functions and structs on one line.

```
void empty_function(void)
{
}

int short_function(void)
{
    return 1;
}
```

Don't use unnecessary branching. E.g.:

```
if (error) {
    goto error;
} else {
    a = b;
}
```

Can be written as:

```
if (error) {
    goto error;
}
a = b;
```

clang-format:

- AllowShortBlocksOnASingleLine: false [llvm]
- AllowShortBlocksOnASingleLine: Never [llvm] (breaking change in clang 10!) [clang10]
- AllowShortEnumsOnASingleLine: false [clang11]
- AllowShortFunctionsOnASingleLine: None
- AllowShortIfStatementsOnASingleLine: Never [llvm]
- AllowShortLoopsOnASingleLine: false [llvm]

- BreakBeforeBraces: Custom [breakbeforebraces]
- BraceWrapping:
  - SplitEmptyFunction: true
  - SplitEmptyRecord: true

Alignment

Pointers

Pointers shall be right aligned.

```
void *ptr;
void f(int *a, const char *b);
void (*foo)(int *);
```

clang-format:

- PointerAlignment: Right
- DerivePointerAlignment: false

Declarations and Comments

Trailing comments should be aligned for consecutive lines.

```
struct bla {
    int a;       /* comment */
    unsigned bb; /* comment */
    int *ccc;    /* comment */
};

void alignment()
{
    // multiple consecutive vars
    int a = 13;           /* comment */
    int32_t abc = 1312;   /* comment */
    int abcdefghikl = 13; /* comment */
}
```

clang-format:

- AlignConsecutiveAssignments: false
- AlignConsecutiveDeclarations: false
- AlignTrailingComments: true

Functions

parameter names

TODO

Function names

Function names are SCNamedLikeThis(). All non-static functions should be prefixed with *SC*.

```
static SCConfNode *SCConfGetNodeOrCreate(char *name, int final)
```

static vs non-static

Functions should be declared static whenever possible.

inline

The inlining of functions should be used only in critical paths.

Variables

Names

A variable is `named_like_this` in all lowercase.

```
SCConfNode *parent_node = root;
```

Generally, use descriptive variable names.

In loop vars, make sure `i` is a signed int type.

Scope

TODO

Macros

Macro names are ALL_CAPS_WITH_UNDERSCORES. Enclose parameters in parens on each usage inside the macro.

Align macro values on consecutive lines.

```
#define ACTION_ALERT       0x01
#define ACTION_DROP        0x02
#define ACTION_REJECT      0x04
#define ACTION_REJECT_DST  0x08
#define ACTION_REJECT_BOTH 0x10
#define ACTION_PASS        0x20
```

Align escape for multi-line macros right-most at ColumnLimit.

```
#define MULTILINE_DEF(a, b)                                    \
    if ((a) > 2) {                                             \
        auto temp = (b) / 2;                                   \
        (b) += 10;                                             \
        someFunctionCall((a), (b));                            \
    }
```

clang-format:

- AlignConsecutiveMacros: true [clang9]
- AlignEscapedNewlines: Right

Comments

Function comments

We use Doxygen, functions are documented using Doxygen notation:

```
/**
 * \brief Helper function to get a node, creating it if it does not
 * exist.
 *
 * This function exits on memory failure as creating configuration
 * nodes is usually part of application initialization.
 *
 * \param name The name of the configuration node to get.
 * \param final Flag to set created nodes as final or not.
 *
 * \retval The existing configuration node if it exists, or a newly
 * created node for the provided name. On error, NULL will be returned.
 */
static SCConfNode *SCConfGetNodeOrCreate(char *name, int final)
```

General comments

We use `/* foobar */` style and try to avoid `//` style.

File names

File names are all lowercase and have a .c .h or .rs (Rust) extension.

Most files have a _subsystem_ prefix, e.g. `detect-dsize.c`, `util-ip.c`

Some cases have a multi-layer prefix, e.g. `util-mpm-ac.c`

Enums

Use a common prefix for all enum values. Value names are ALL_CAPS_WITH_UNDERSCORES.

Put each enum values on a separate line. Tip: Add a trailing comma to the last element to force "one-value-per-line" formatting in clang-format.

Enums exposed in a header file should be prefixed with `SC_`.

```
enum { VALUE_ONE, VALUE_TWO };  // <- wrong

// right
enum {
    VALUE_ONE,
    VALUE_TWO, // <- force one-value-per-line
};
```

clang-format:

- AllowShortEnumsOnASingleLine: false [clang11]

Structures and typedefs

Structures and typedefs use `TitleCase` naming. When exposed in a header file they must be prefixed with `SC`.

For example:

```
typedef struct SCPlugin_ {
} SCPlugin;
```

switch statements

Switch statements are indented like in the following example, so the 'case' is indented from the switch:

```
switch (ntohs(p->ethh->eth_type)) {
    case ETHERNET_TYPE_IP:
        DecodeIPV4(tv, dtv, p, pkt + ETHERNET_HEADER_LEN,
                   len - ETHERNET_HEADER_LEN, pq);
        break;
```

Fall through cases will be commented with `/* fall through */`. E.g.:

```
switch (suri->run_mode) {
    case RUNMODE_PCAP_DEV:
    case RUNMODE_AFP_DEV:
    case RUNMODE_PFRING:
        /* find payload for interface and use it */
        default_packet_size = GetIfaceMaxPacketSize(suri->pcap_dev);
        if (default_packet_size)
            break;
        /* fall through */
    default:
        default_packet_size = DEFAULT_PACKET_SIZE;
```

Do not put short case labels on one line. Put opening brace on same line as case statement.

```
switch (a) {
    case 13: {
        int a = bla();
        break;
    }
    case 15:
        blu();
        break;
    default:
        gugus();
}
```

clang-format:

- IndentCaseLabels: true
- IndentCaseBlocks: false [clang11]
- AllowShortCaseLabelsOnASingleLine: false [llvm]
- BreakBeforeBraces: Custom [breakbeforebraces]
- BraceWrapping:

- AfterCaseLabel: false (default)

const

TODO

goto

Goto statements should be used with care. Generally, we use it primarily for error handling. E.g.:

```
static DetectFileextData *DetectFileextParse (char *str)
{
    DetectFileextData *fileext = NULL;

    fileext = SCMalloc(sizeof(DetectFileextData));
    if (unlikely(fileext == NULL))
        goto error;

    memset(fileext, 0x00, sizeof(DetectFileextData));

    if (DetectContentDataParse("fileext", str, &fileext->ext, &fileext->len, &fileext->
↪flags) == -1) {
        goto error;
    }

    return fileext;

error:
    if (fileext != NULL)
        DetectFileextFree(fileext);
    return NULL;
}
```

Put goto labels at brace level.

```
int goto_style_nested()
{
    if (foo()) {
    label1:
        bar();
    }

label2:
    return 1;
}
```

clang-format:

- IndentGotoLabels: true (default) [clang10]

Includes

A .c file shall include it's own header first, or immediately after `suricata-common.h`.

clang-format:

- SortIncludes: false

Unittests

When writing unittests that use a data array containing a protocol message, please put an explanatory comment that contain the readable content of the message

So instead of:

```
int SMTPProcessDataChunkTest02(void)
{
    char mimemsg[] = {0x4D, 0x49, 0x4D, 0x45, 0x2D, 0x56, 0x65, 0x72,
```

you should have something like:

```
int SMTPParserTest14(void)
{
    /* 220 mx.google.com ESMTP d15sm986283wfl.6<CR><LF> */
    static uint8_t welcome_reply[] = { 0x32, 0x32, 0x30, 0x20,
```

Banned functions

function	replacement	reason
strtok	strtok_r	
sprintf	snprintf	unsafe
strcat	strlcat	unsafe
strcpy	strlcpy	unsafe
strncpy	strlcat	
strncat	strlcpy	
strndup		OS specific
strchrnul		
rand		
rand_r		
index		
rindex		
bzero	memset	

Also, check the existing code. If yours is wildly different, it's wrong. Example: https://github.com/oisf/suricata/blob/main/src/decode-ethernet.c

Rust

Pure Rust Code

Rust functions should follow normal Rust style where appropriate, for example:

```
pub fn try_new_array() -> Result<()> {
    Ok(())
}
```

New Rust code should be formatted with `rustfmt` or `cargo fmt`. If reformatting an existing file, format and commit before making any changes. Such reformatting may be rejected in a PR based on a variety of factors.

FFI

Rust code that is exposed to C should follow our C code style with respect to naming. This applies to all functions marked as `#[no_mangle]`. For example:

```
#[no_mangle]
pub extern "C" SCJbNewArray() -> *mut JsonBuilder {
}
```

29.1.3 Fuzz Testing

To enable fuzz targets compilation, add *--enable-fuzztargets* to configure.

> **Note**
>
> This changes various parts of Suricata making the *suricata* binary unsafe for production use.

The targets can be used with libFuzzer, AFL and other fuzz platforms.

Running the Fuzzers

TODO. For now see src/tests/fuzz/README

Reproducing issues

Extending Coverage

Adding Fuzz Targets

Oss-Fuzz

Suricata is continuously fuzz tested in Oss-Fuzz. See https://github.com/google/oss-fuzz/tree/master/projects/suricata

29.1.4 Testing Suricata

Table of Contents

- *Testing Suricata*
  - *General Concepts*
  - *Unit tests*
    - *Code Examples*
  - *Suricata-Verify*
  - *Generating Input*
    - *Using real traffic*
    - *Crafting input samples with Scapy*
    - *Other examples from our Suricata-Verify tests:*
    - *Finding Capture Samples*

General Concepts

There are a few ways of testing Suricata:

- **Unit tests**: for independently checking specific functions or portions of code. This guide has specific sections to further explain those, for C and Rust;
- Suricata-Verify: those are used to check more complex behavior, like the log output or the alert counts for a given input, where that input is usually comprised of several packets;
- **Static and dynamic analysis tools**: to help in finding bugs, memory leaks and other issues (like scan-build, from *clang*, which is also used for our C formatting checks; or ASAN, which checks for memory issues);
- **Fuzz testing**: especially good for uncovering existing, often non-trivial bugs. For more on how to fuzz test Suricata, check *Fuzz Testing*;
- **CI checks**: each PR submitted to the project's public repositories will be run against a suit of Continuous Integration workflows, as part of our QA process. Those cover: formatting and commit checks; fuzz tests (CI Fuzz), and several builds. See our github workflows for details and those in action at https://github.com/OISF/suricata/actions.

> **Note**
>
> If you can run unit tests or other checks and report failures in our issue tracker, that is rather useful and appreciated!

The focus of this document are Unit tests and Suricata-Verify tests, especially on offering some guidance regarding when to use each type of test, and how to prepare input for them.

Unit tests

Use these to check that specific functions behave as expected, in success and in failure scenarios. Specially useful during development, for nom parsers in the Rust codebase, for instance, or for checking that messages or message parts of a protocol/stream are processed as they should.

To execute all unit tests (both from C and Rust code) from the Suricata main directory, run:

```
make check
```

Check the Suricata Devguide on *Unit Tests - C* or *Unit tests - Rust* for more on how to write and run unit tests, given that the way to do so differs, depending on the language.

Code Examples

An example from the DNS parser. This checks that the given raw input (note the comments indicating what it means), once processed by `dns_parse_name` yields the expected result, including the unparsed portion.

```
/// Parse a simple name with no pointers.
#[test]
fn test_dns_parse_name() {
    let buf: &[u8] = &[
                                          0x09, 0x63, /* .......c */
        0x6c, 0x69, 0x65, 0x6e, 0x74, 0x2d, 0x63, 0x66, /* lient-cf */
        0x07, 0x64, 0x72, 0x6f, 0x70, 0x62, 0x6f, 0x78, /* .dropbox */
        0x03, 0x63, 0x6f, 0x6d, 0x00, 0x00, 0x01, 0x00, /* .com.... */
    ];
```

(continues on next page)

(continued from previous page)

```
    let expected_remainder: &[u8] = &[0x00, 0x01, 0x00];
    let (remainder,name) = dns_parse_name(buf, buf).unwrap();
    assert_eq!("client-cf.dropbox.com".as_bytes(), &name[..]);
    assert_eq!(remainder, expected_remainder);
}
```

From the C side, `decode-ethernet.c` offers an good example:

```
/**
 * Test a DCE ethernet frame that is too small.
 */
static int DecodeEthernetTestDceTooSmall(void)
{
    uint8_t raw_eth[] = {
        0x00, 0x10, 0x94, 0x55, 0x00, 0x01, 0x00, 0x10,
        0x94, 0x56, 0x00, 0x01, 0x89, 0x03,
    };

    Packet *p = PacketGetFromAlloc();
    FAIL_IF_NULL(p);
    ThreadVars tv;
    DecodeThreadVars dtv;

    memset(&dtv, 0, sizeof(DecodeThreadVars));
    memset(&tv,  0, sizeof(ThreadVars));

    DecodeEthernet(&tv, &dtv, p, raw_eth, sizeof(raw_eth));

    FAIL_IF_NOT(ENGINE_ISSET_EVENT(p, DCE_PKT_TOO_SMALL));

    PacketFree(p);
    PASS;
}
```

Suricata-Verify

As mentioned above, these tests are used to check more complex behavior that involve a complete flow, with exchange of requests and responses. This can be done in an easier and more straightforward way, since one doesn't have to simulate the network traffic and Suricata engine mechanics - one simply runs it, with the desired input packet capture, configuration and checks.

A Suricata-verify test can help to ensure that code refactoring doesn't affect protocol logs, or signature detection, for instance, as this could have a major impact to Suricata users and integrators.

For simpler tests, providing the pcap input is enough. But it is also possible to provide Suricata rules to be inspected, and have Suricata Verify match for alerts and specific events.

Refer to the Suricata Verify readme for details on how to create this type of test. It suffices to have a packet capture representative of the behavior one wants to test, and then follow the steps described there.

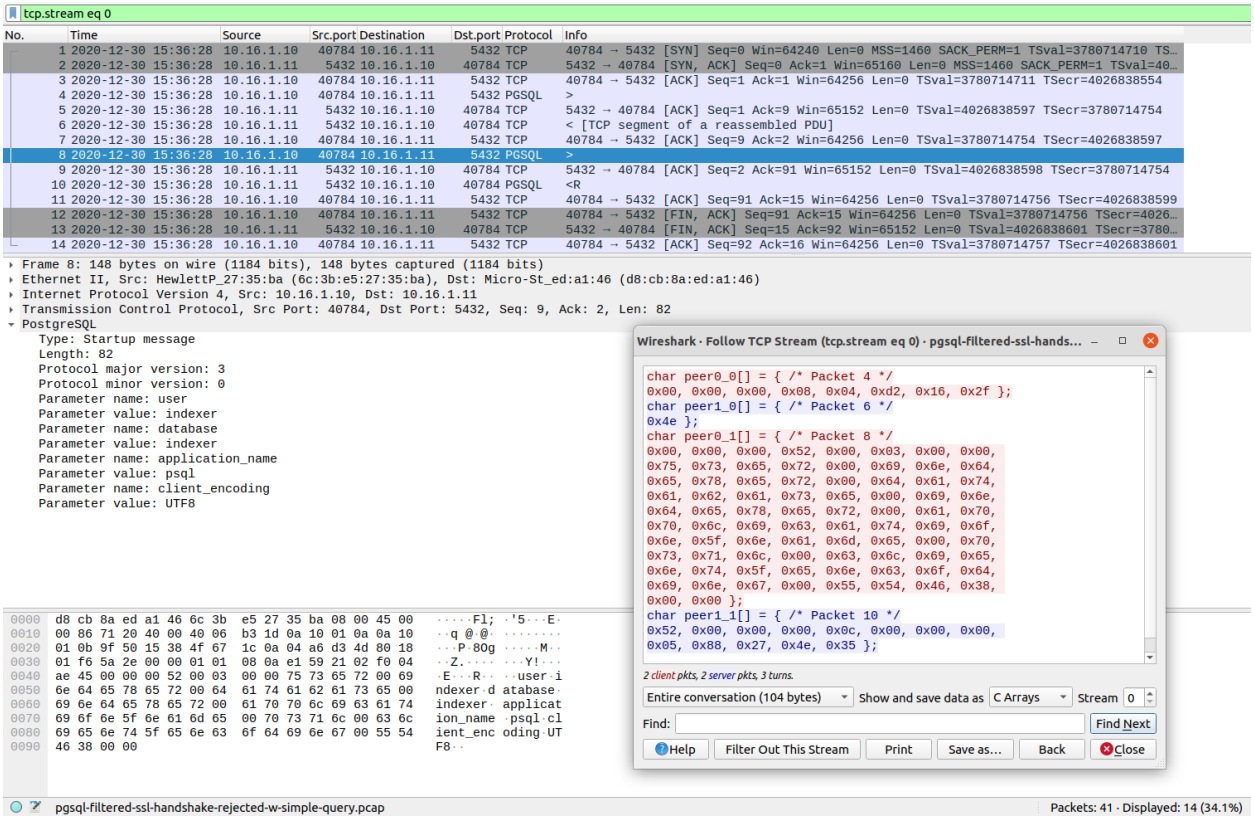
The Git repository for the Suricata Verify tests is a great source for examples, like the app-layer-template one.

Generating Input

Using real traffic

Having a packet capture for the desired protocol you want to test, open it in Wireshark, and select the specific packet chosen for the test input, then use the Wireshark option `Follow [TCP/UDP/HTTP/HTTP2/QUIC] Stream`. This allows for inspecting the whole network traffic stream in a different window. There, it's possible to choose to `Show and save data as C Arrays`, as well as to select if one wants to see the whole conversation or just **client** or **server** packets. It is also possible to reach the same effect by accessing the **Analyze->Follow->TCP Stream** top menu in Wireshark. (There are other stream options, the available one will depend on the type of network traffic captured).

This option will show the packet data as hexadecimal compatible with C-array style, and easily adapted for Rust, as well. As shown in the image:



Wireshark can be also used to capture sample network traffic and generate pcap files.

Crafting input samples with Scapy

It is also possible to use Scapy to create specific traffic: Scapy usage

Suricata-verify tests have several examples of pcaps generated in such a way. Look for Python scripts like the one used for the dce-udp-scapy.

Other examples from our Suricata-Verify tests:

Going through Suricata-Verify tests *readme* files it is also possible to find an assorted collection of pcap generation possibilities, some with explanation on the how-tos. To list a few:

- http2-range
- http-range

- smb2-delete
- smtp-rset
- http-auth-unrecognized

Finding Capture Samples

If you can't capture traffic for the desired protocol from live traffic, or craft something up, you can try finding the type of traffic you are interested in in public data sets. There's a thread for Sharing good sources of sample captures in our forum.

29.1.5 Unit Tests - C

Unit tests are a great way to create tests that can check the internal state of parsers, structures and other objects.

Tests should:

- use `FAIL`/`PASS` macros
- be deterministic
- not leak memory on `PASS`
- not use conditions

Unit tests are used by developers of Suricata and advanced users who would like to contribute by debugging and testing the engine. Unit tests are small pieces (units) of code which check certain code functionalities in Suricata. If Suricata's code is modified, developers can run unit tests to see if there are any unforeseen effects on other parts of the engine's code. Unit tests will not be compiled with Suricata by default. If you would like to compile Suricata with unit tests, enter the following during the configure-stage:

```
./configure --enable-unittests
```

The unit tests specific command line options can be found at Command Line Options.

Example: You can run tests specifically on flowbits. This is how you should do that:

```
suricata -u -U flowbit
```

It is highly appreciated if you would run unit tests and report failing tests in our issue tracker.

If you want more info about the unittests, regular debug mode can help. This is enabled by adding the configure option:

```
--enable-debug
```

Then, set the debug level from the command-line:

```
SC_LOG_LEVEL=Debug suricata -u
```

This will be very verbose. You can also add the `SC_LOG_OP_FILTER` to limit the output, it is grep-like:

```
SC_LOG_LEVEL=Debug SC_LOG_OP_FILTER="(something|somethingelse)" suricata -u
```

This example will show all lines (debug, info, and all other levels) that contain either something or something else. Keep in mind the log level precedence: if you choose *Info* level, for instance, Suricata won't show messages from the other levels.

Writing Unit Tests - C codebase

Suricata unit tests are somewhat different in C and in Rust. In C, they are comprised of a function with no arguments and returning 0 for failure or 1 for success. Instead of explicitly returning a value, FAIL_* and PASS macros should be used. For example:

```
void MyUnitTest(void)
{
    int n = 1;
    void *p = NULL;

    FAIL_IF(n != 1);
    FAIL_IF_NOT(n == 1);
    FAIL_IF_NOT_NULL(p);
    FAIL_IF_NULL(p);

    PASS;
}
```

Each unit test needs to be registered with `UtRegisterTest()`. Example:

```
UtRegisterTest("MyUnitTest", MyUnitTest);
```

where the first argument is the name of the test, and the second argument is the function. Existing modules should already have a function that registers its unit tests. Otherwise the unit tests will need to be registered. Look for a module similar to your new module to see how best to register the unit tests or ask the development team for help.

Examples

From `conf-yaml-loader.c`:

```
/**
 * Test that a configuration section is overridden but subsequent
 * occurrences.
 */
static int
ConfYamlOverrideTest(void)
{
    char config[] =
        "%YAML 1.1\n"
        "---\n"
        "some-log-dir: /var/log\n"
        "some-log-dir: /tmp\n"
        "\n"
        "parent:\n"
        "  child0:\n"
        "    key: value\n"
        "parent:\n"
        "  child1:\n"
        "    key: value\n"
        ;
    const char *value;

    SCConfCreateContextBackup();
```

(continues on next page)

(continued from previous page)

```
    SCConfInit();

    FAIL_IF(SCConfYamlLoadString(config, strlen(config)) != 0);
    FAIL_IF_NOT(SCConfGet("some-log-dir", &value));
    FAIL_IF(strcmp(value, "/tmp") != 0);

    /* Test that parent.child0 does not exist, but child1 does. */
    FAIL_IF_NOT_NULL(SCConfGetNode("parent.child0"));
    FAIL_IF_NOT(SCConfGet("parent.child1.key", &value));
    FAIL_IF(strcmp(value, "value") != 0);

    SCConfDeInit();
    SCConfRestoreContextBackup();

    PASS;
}
```

In `detect-ike-chosen-sa.c`, it is possible to see the freeing of resources (`DetectIkeChosenSaFree`) and the function that should group all the `UtRegisterTest` calls:

```
#ifdef UNITTESTS
.
.
.
static int IKEChosenSaParserTest(void)
{
    DetectIkeChosenSaData *de = NULL;
    de = DetectIkeChosenSaParse("alg_hash=2");

    FAIL_IF_NULL(de);
    FAIL_IF(de->sa_value != 2);
    FAIL_IF(strcmp(de->sa_type, "alg_hash") != 0);

    DetectIkeChosenSaFree(NULL, de);
    PASS;
}

#endif /* UNITTESTS */

void IKEChosenSaRegisterTests(void)
{
#ifdef UNITTESTS
    UtRegisterTest("IKEChosenSaParserTest", IKEChosenSaParserTest);
#endif /* UNITTESTS */
```

29.1.6 Unit tests - Rust

Rust tests with Cargo check

Rust offers a built-in tool for running unit and integration tests. To do so, one makes usage of:

```
cargo test [options][testname][-- test-options]
```

The Cargo Book explains all options in more detail.

For testing a specific Rust module from Suricata, it suffices to go to the `rust` directory and run the above command, specifying the desired module (like `http2`).

```
cargo test http2
```

The line above will make *rustc* compile the Rust side of Suricata and run unit tests in the `http2` rust module.

For running all Suricata unit tests from our Rust codebase, just run `cargo test`.

Adding unit tests

> **Note**
>
> If you want to understand *when* to use a unit test, please read the devguide section on *Testing Suricata*.

In general, it is preferable to have the unit tests in the same file that they test. At the end of the file, after all other functions. Add a `tests` module, if there isn't one yet, and add the `#[test]` attribute before the unit test function. It is also necessary to import (`use`) the module to test, as well as any other modules used. As seen in the example below:

Example

From `nfs > rpc_records.rs`:

```
mod tests {
    use crate::nfs::rpc_records::*;
    use nom::Err::Incomplete;
    use nom::Needed::Size;

    #[test]
    fn test_partial_input_ok() {
        let buf: &[u8] = &[
            0x80, 0x00, 0x00, 0x9c, // flags
            0x8e, 0x28, 0x02, 0x7e, // xid
            0x00, 0x00, 0x00, 0x01, // msgtype
            0x00, 0x00, 0x00, 0x02, // rpcver
            0x00, 0x00, 0x00, 0x03, // program
            0x00, 0x00, 0x00, 0x04, // progver
            0x00, 0x00, 0x00, 0x05, // procedure
        ];
        let expected = RpcRequestPacketPartial {
            hdr: RpcPacketHeader {
                    frag_is_last: true,
                    frag_len: 156,
                    xid: 2384986750,
                    msgtype: 1
                },
            rpcver: 2,
            program: 3,
            progver: 4,
            procedure: 5
        };
```

(continues on next page)

(continued from previous page)

```
        let r = parse_rpc_request_partial(buf);
        match r {
            Ok((rem, hdr)) => {
                assert_eq!(rem.len(), 0);
                assert_eq!(hdr, expected);
            },
            _ => { panic!("failed {:?}",r); }
        }
    }
}
```

Once that is done, Rust should recognize the new test. If you want to check a single test, run:

```
cargo test module::file_name::tests::test_name
```

Where `tests` refers to `mod tests`. If you know the test name is unique, you can even run:

```
cargo test test_name
```

Following the same idea, it is also possible to test specific modules or submodules. For instance:

```
cargo test nfs::rpc_records
```

29.2 Contributing

29.2.1 Contributing to Suricata

This guide describes what steps to take if you want to contribute a patch or patchset to Suricata.

Essentially, these are:

1. Agree to and sign our *Contribution Agreement*
2. Communicate early, and use the *preferred channels*
3. *Claim (or open) a ticket*
4. *Fork from main*
5. Follow our *Coding Style*
6. Use our *Documentation Style*
7. Stick to our *commit guidelines*
8. Add version numbers to your *Pull Requests*
9. Incorporate *Feedback* into new PRs
10. [Work merged] *Wrap up!*

The rest of this document will cover those in detail.

> **Note**
>
> Important!
>
> Before contributing, please review and sign our Contribution Agreement.

Communication is Key!

To clarify questions, discuss or suggest new features, talk about bugs and optimizations, and/or ask for help, it is important to communicate.

These are our main channels:

- Suricata's issue tracker
- Suricata's forum
- Suricata's Discord server

Claim (or open) a ticket

For features and bugs we need tickets. Tickets help us keep track of the work done, indicate when changes need backports etc.

They are also important if you would like to see your new feature officially added to our tool: the ticket documents your ideas so we can analyze how do they fit in our plans for Suricata, and, if the feature is accepted, we can properly track progress etc.

The ticket should clearly reflect the intention as per the tracker. For example, if the ticket is a "Bug", the title should only say what the bug is.

Good ticket title examples

1. **Ticket:** [Bug #00000] stream: segfault in case of increasing gaps

Why is it good? It shows subsystem affected and exactly what the bug is.

2. **Ticket:** [Bug #19999] dcerpc: memleak in case of invalid data

Why is it good? It talks about the bug itself as the Tracker indicates.

3. **Ticket:** [Bug #44444] stream: excess memuse in *TcpTracking*

Why is it good? Title is to the point and conveys what the issue is.

> **Note**
>
> The ticket titles are used to auto generate ChangeLog with each release. If the ticket titles are unclear, the ChangeLog does not properly convey what issues were fixed with a release.

> **Note**
>
> If you want to add new functionalities (e.g. a new application layer protocol), please ask us first whether we see that being merged into Suricata or not. This helps both sides understand how the new feature will fit in our roadmap, and prevents wasting time and motivation with contributions that we may not accept. Therefore, *before* starting any code related to a new feature, do request comments from the team about it.

For really trivial fixes or cleanups we won't need that.

Once work on the issue has been agreed upon:

Assign the ticket to yourself. For this, you will need to have the "developer" role. You can ask for that directly on the ticket you want to claim or mention that you are interested in working on *ticket number* on our Developer's channel on Discord.

> **Attention**
>
> If you submit work for a ticket without it being assigned to you, **we may close your PR and request you to do so**. Please, communicate with us when before working on an issue. Comment on the ticket, assign it to yourself.

If a ticket is already assigned to someone: please reach out on the ticket or ask the person first.

You can reach out to other community members via Suricata's Discord server.

Expectations

If you submit a new feature that is not part of Suricata's core functionalities, it will have the *community supported* status. This means we would expect some commitment from you, or the organization who is sponsoring your work, before we could approve the new feature, as the Suricata development team is pretty lean (and many times overworked).

This means we expect that:

- the new contribution comes with a set of Suricata-verify tests (and possibly unit tests, where those apply), before we can approve it;
- proof of compatibility with existing keywords/features is provided, when the contribution is for replacing an existing feature;
- you would maintain the feature once it is approved - or some other community member would do that, in case you cannot.

> **Note**
>
> Regardless of contribution size or complexity, we expect that you respect our guidelines and processes. We appreciate community contributors: Suricata wouldn't be what it is without them; and the value of our tool and community also comes from how seriously we take all this, so we ask that our contributors do the same!

What does "community supported" and "supporting a feature" mean?

If a feature is *community supported*, the Suricata team will try to spend minimal time on it - to be able to focus on the core functionalities. If for any reason you're not willing or able to commit to supporting a feature, please indicate this.

The team and/or community members can then consider offering help. It is best to indicate this prior to doing the actual work, because we will reject features if no one steps up.

It is also important to note that *community supported* features will be disabled by default, and if it brings in new dependencies (libraries or Rust crates) those will also be optional and disabled by default.

Supporting a feature means to actually *maintain* it:

- fixing bugs
- writing documentation
- keeping it up to date
- offering end-user support via forum and/or Discord chat

Stale tickets policy

We understand that people's availability and interested to volunteer their time to our project may change. Therefore, to prevent tickets going stale (not worked on), and issues going unsolved for a long time, we have a policy to unclaim tickets if there are no contribution updates within 6 months.

If you claim a ticket and later on find out that you won't be able to work on it, it is also appreciated if you inform that to us in the ticket and unclaim it, so everyone knows that work is still open and waiting to be done.

What branch to work on

There are usually 2 or 3 active branches:

- main-x.x.x (e.g. main-7.0.x)
- main-x.x.x (e.g. main-8.0.x)
- main

The ones with version numbers are stable branches. **main** is the development branch.

The stable branch should only be worked on for important bug fixes or other needed *backports*. Those are mainly expected from more experienced contributors.

Development of new features or large scale redesign is done in the development branch. New development and new contributors should work with *main* except in very special cases - which should and would be discussed with us first.

If in doubt, please reach out to us via *Redmine, Discord or forum*.

Create your own branch

It's useful to create descriptive branch names. You're working on ticket 123 to improve GeoIP? Name your branch "geoip-feature-123-v1". The "-v1" addition is for feedback. When incorporating feedback you will have to create a new branch for each pull request. So, when you address the first feedback, you will work in "geoip-feature-123-v2" and so on.

For more details check: Creating a branch to do your changes

Coding Style

We have a *Coding Style* that must be followed.

Documentation Style

For documenting *code*, please follow Rust documentation and/or Doxygen guidelines, according to what your contribution is using (Rust or C). The rest of this section refers to the user and developer documentation.

The user and developer guide documentation (what you are reading now) is written in *reStructuredText* and rendered with Sphinx. For a primer *reStucturedText* please see the reStrucutredText Primer.

When writing or updating *documentation pages*, please:

- wrap up lines at 79 (80 at most) characters;
- when adding diagrams or images, we prefer alternatives that can be generated automatically, if possible;
- bear in mind that our documentation is published on Read the Docs and can also be built to pdf, so it is important that it looks good in such formats.

Headings

reStructuredText allows for flexible header order, for consistency please use the following order:

- #: for h1
- *: for h2
- =: for h3
- -: for h4
- ~: for h5
- ^: for h6

For example, in a new documentation page:

```
Page Title
##########

Section
*******

Sub-Section
===========
```

Rule examples

For rule documentation, we have a special container:

```
example-rule
```

This will present the rule in a box with an easier to read font size, and also allows highlighting specific elements in the signature, as the names indicate - action, header, options, or emphasize custom portions:

- example-rule-action
- example-rule-header
- example-rule-options
- example-rule-emphasis

When using these, indicate the portion to be highlighted by surrounding it with ` . Before using them, one has to invoke the specific role, like so:

```
.. role:: example-rule-role
```

It is only necessary to invoke the role once per document. One can see these being invoked in our introduction to the rule language (see Rules intro).

A rule example like:

```
.. container:: example-rule

:example-rule-action:`alert` :example-rule-header:`http $HOME_NET any ->
$EXTERNAL_NET any` :example-rule-options:`(msg:"HTTP GET Request Containing
Rule in URI"; flow:established,to_server; http.method; content:"GET"; http.uri;
content:"rule"; fast_pattern; classtype:bad-unknown; sid:123; rev:1;)`
```

Results in:

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"HTTP GET Request Containing Rule in URI"; flow:established,to_server; http.method; content:"GET"; http.uri; content:"rule"; fast_pattern; classtype:bad-unknown; sid:123; rev:1;)

Example - emphasis:

```
.. container:: example-rule

alert ssh any any -> any any (msg:"match SSH protocol version";
:example-rule-emphasis:`ssh.proto;` content:"2.0"; sid:1000010;)
```

Renders as:

alert ssh any any -> any any (msg:"match SSH protocol version"; ssh.proto; content:"2.0"; sid:1000010;)

Commit History matters

Please consider our *Commit guidelines* before submitting your PR.

Send a Pull Request

The pull request is used to request inclusion of your patches into the main repository. Before it is merged, it will be reviewed and pushed through a QA process.

Please consider our *Pull Requests Criteria* when submitting.

We have 'GitHub-CI' integration enabled. This means some automated build check, suricata-verity and unit tests are performed on the pull request. Generally, this is ready after a few minutes. If the test fails, the pull request won't be considered. So please, when you submit something, keep an eye on the checks, and address any failures - if you do not understand what they are, it is fine to ask about them on the failing PR itself.

Before merge, we also perform other integration tests in our private QA-lab. If those fail, we may request further changes, even if the GitHub-CI has passed.

Feedback

You'll likely get some feedback. Even our most experienced devs do, so don't feel bad about it.

After discussing what needs to be changed (usually on the PR itself), it's time to go back to "*Create your own branch*" and do it all again. This process can iterate quite a few times, as the contribution is refined.

Wrapping up

Merged! Cleanup

Congrats! Your change has been merged into the main repository. Many thanks!

We strongly suggest cleaning up: delete your related branches, both locally and on GitHub - this helps you in keeping things organized when you want to make new contributions.

Update ticket

You can now put the URL of the *merged* pull request in the Redmine ticket. Next, mark the ticket as "Closed" or "Resolved".

Well done! You are all set now.

29.2.2 Code Submission Process

Commits

1. Commits need to be logically separated. Don't fix unrelated things in one commit.
2. Don't add unnecessary commits, if commit 2 fixes commit 1 merge them together (squash)
3. Commits need to have proper messages, explaining anything that is non-trivial
4. Commits should not, at the same time, change, rename and/or move code. Use separate commits for each of this, e.g, a commit to rename files, then a commit to change the code.
5. If your code changes or adds new behavior, add the related documentation updates in their own commit, but make sure to add the same ticket number to both commit messages.
6. **Commit messages need to be properly formatted (check the example further below in this section).**
    - Meaningful and short (50 chars max) subject line followed by an empty line
    - Naming convention: prefix message with sub-system (**"rule parsing: fixing foobar"**). If you're not sure what to use, look at past commits to the file(s) in your PR.
    - Description, wrapped at ~72 characters
7. Commits should be individually compilable, starting with the oldest commit. Make sure that each commit can be built if it and the preceding commits in the PR are used.
8. Commits should be authored with the format: "FirstName LastName <name@example.com>"

We recommend that you use git commit message template with the following command: `git config commit.template /path/to/suricata/git-template/commit-template.txt` The template lists items that help describe the context and include requisite information in the commit message. We reserve the right to strictly enforce the template in the future:

Information that needs to be part of a commit (if applicable):

1. Ticket it fixes. E.g. "Fixes Bug #123."
2. Compiler warnings addressed.
3. Coverity Scan issues addressed.
4. Static analyzer error it fixes (cppcheck/scan-build/etc)

Note

When in doubt, check our git history for other messages or changes done to the same module your're working on. This is a good example of a commit message:

```
pcap/file: normalize file timestamps

Normalize the timestamps that are too far in the past to epoch.

Bug: #6240.
```

Pull Requests

A github pull request is actually just a pointer to a branch in your tree. GitHub provides a review interface that we use.

1. A branch can only be used in for an individual PR.
2. A branch should not be updated after the pull request

3. A pull request always needs a good description (link to issue tracker if related to a ticket).
4. Incremental pull requests need to link to the prior iteration
5. Incremental pull requests need to describe changes since the last PR
6. Link to the ticket(s) that are addressed to it.
7. When fixing an issue, update the issue status to `In Review` after submitting the PR.
8. Pull requests are automatically tested using github actions (https://github.com/OISF/suricata/blob/main/.github/workflows/builds.yml). Failing builds won't be considered and should be closed immediately.
9. Pull requests that change, or add a feature should include a documentation update commit

Tests and QA

As much as possible, new functionality should be easy to QA.

1. Add `suricata-verify` tests for verification. See https://github.com/OISF/suricata-verify
2. Add unittests if a `suricata-verify` test isn't possible.
3. Provide pcaps that reproduce the problem. Try to trim as much as possible to the pcap includes the minimal set of packets that demonstrate the problem.
4. Provide example rules if the code added new keywords or new options to existing keywords

AI Tools

Suricata is maintained by people. If you use AI tools to contribute, we still want to be talking to you.

1. You are the author. Whatever tools you use, you are responsible for your contribution. You should understand the changes well enough to explain and defend them in review. If you cannot, say so clearly in the pull request.
2. Write pull request descriptions and respond to reviewers based on your own understanding. Reviewers are asking you questions, and they expect answers you can explain and stand behind.
3. If you include AI-generated text in a pull request description, comment or reply, say so and clearly mark it. Translation, spelling corrections and minor wording edits do not require disclosure.
4. Verify claims, references and test results before you share them. Only report tests as passed if they were actually run and you checked the results.

The Suricata team uses AI tools for code review. Any AI-generated review comments will be clearly marked as such. If you disagree with an AI review comment, please say so. The team makes the final call, not AI.

29.2.3 GitHub Pull Request Workflow

Draft Pull Requests

A Pull Request (PR) should be marked as *draft* if it is not intended to be merged as is, but is waiting for some sort of feedback. The author of the PR should be explicit with what kind of feedback is expected (CI/QA run, discussion on the code, etc...)

The GitHub filter is `is:pr is:open draft:true sort:updated-asc`.

A draft may be closed if it has not been updated in two months.

Mergeable Pull Requests

When a Pull Request is intended to be merged as is, the workflow is the following:

1. get reviewed, and either request changes or get approved
2. if approved, get staged in a next branch (with other PRs), wait for CI validation (and eventually request changes if CI finds anything)
3. get merged and closed

Once submitted, we aim at providing a first PR review within two weeks and a month.

If either code, documentation wording or commit messages need re-work, the reviewer will set the PR state to *changes requested*.

> **Note**
>
> It is expected that the author will create a new PR with a new version of the patch as described in *Pull Requests Criteria*. A PR may be closed as stale if it has not been updated in two months after changes were requested.

A PR may be labeled *decision-required* if the reviewer thinks the team needs more time to analyze the best approach to a proposed solution or discussion raised by the PR.

Once in approved state, the PRs are in the responsibility of the maintainer, along with the next branches/PRs.

Reviewers and Maintainers

A newly created PR should match the filter:

```
is:pr is:open draft:false review:none sort:updated-asc no:assignee
```

The whole team is responsible to assign a PR to someone precise within 2 weeks.

When someone gets assigned a PR, it should get a review status within 2 weeks: either changes requested, approved, or assigned to someone else if more expertise is needed.

The GitHub filter for changes-requested PRs is:

```
is:pr is:open draft:false sort: updated-asc review:changes-requested
```

The command to get approved PRs is:

```
gh pr list --json number,reviewDecision --search "state:open type:pr -review:none" | jq
↪'.[] | select(.reviewDecision=="")'
```

An approved PR should match the filter: `is:open is:pr review:approved`.

29.2.4 Suricata Backports Guide

This document describes the processes used to backport content to current stable Suricata releases. Most often, this means security and/or bug fixes; however, in some cases, features may be backported to previous Suricata releases.

There are multiple versions of Suricata at any given time:

- Main
- Major stable release
- Old stable release

For example, at the moment, there are 3 releases based on these Suricata branches:

- main: 9.0.0-dev, current development branch
- main-8.0.x: major stable release
- main-7.0.x: old stable release

For Suricata's release cadence and *end of life* policies, please check https://suricata.io/our-story/eol-policy/.

The next sections discuss when and what to backport, and some guidelines when doing so.

What should be backported?

Usually, when the team creates a ticket, we'll add the *Needs backport* related labels, so necessary backporting tickets will be automatically created. If you are working on a ticket that doesn't have such labels, nor backporting tasks associated, it probably doesn't need backporting. If you understand that the issue should be backported, please let us know in the ticket or related PR. But sometimes we'll miss those.

The general principle used to determine what will be backported is:

- security fixes (please see our Security Policy)
- bug fixes
- in some cases, new features are backported if there are sufficient reasons to backport a new feature.

Note

7.0.x backports

With the release of Suricata 8.0, Suricata 7 has reached a mature stage of its release cycle. Thus, as a rule, only critical issues (bugs and security), should be considered for backporting.

Note

Exceptions

There can be cases where backports may be "missed" -- some issues may not be labeled as needing backports and some PRs may be merged without an issue.

This guide may be insufficient for some situations. When in doubt, please reach out to the team on the backport ticket or PR.

Selection overview

All items considered for backports should be reviewed with the following:

- risk estimate: will the change introduce new bugs? Consider the scope and items affected by the change.
- behavioral change: how much will the behavior of the system be changed by the backport. For example, a small change to decode additional encapsulation protocols may result in more traffic being presented to Suricata.
- default settings: if the issue alters behavior, can it be made optional, and at what cost?

Creating backport tickets -- new issues

Redmine: for security and bug fixes, when creating a new Redmine issue, label the Redmine issue with "Needs backport to x.0", where x.0 is a supported Suricata release, e.g, 8.0.x.

Creating backport tickets -- existing issues/PRs

We want to minimize the occurrence of "missed backports" -- that is, work that should be backported but wasn't. Sometimes this happens when there is no Redmine issue, or the Redmine issue wasn't labeled as needing a backport.

Therefore, we will be periodically reviewing:

- Redmine issues without backport labels, including recently closed issues, to see which require backport labels.
- PRs without associated Redmine issues. Those requiring backports should be labeled with *needs backport*.

Then, also periodically, we will create backport issues from those items identified in the previous steps. When doing so, we will evaluate what are the relevant target backport releases. Some issues reported against main or the current Suricata release may not apply to older releases.

Git Backport Workflow

If you are working on a task that needs to be backported, only start the backporting process once the PR for main has been merged. Then:

- *Identify the commit(s) needed* for the backport. Start with the PR that merged the commits into main and select only the commits from the issue being backported.
- *Bring each commit into the new branch,* one at a time -- starting with the oldest commit. Use `git cherry-pick -x commit-hash`, where `commit-hash` is the hash to the commit already in main or main-8.0x that is being backported, as it maintains the linkage with said cherry-picked commit.
- *Resolve conflicts:* Some of the cherry-picked commits may contain merge conflicts. If the conflicts are small, include the corrections in the cherry-picked commit.
- *Add additional commits*, if any are needed (e.g., to adjust cherry-picked code to old behavior).

> **Note**
>
> Commit hashes
>
> We have a CI check that ensures the validity of the cherry-pick line.

> **Note**
>
> Exceptions
>
> Sometimes, the fix for main will not work for the stable or old releases. In such cases, the backporting process won't be through cherry-picking, but through actually implementing a fix for the specific version.

Create a PR:

Please indicate in the title that this is a backport PR, with something like *(8.0.x-backport)*, and add the related milestone label.

In the PR description, indicate the backport ticket.

QA

Add suricata-verify PRs when needed. Some existing suricata-verify tests may require version specification changes.

29.3 Suricata Internals

29.3.1 Packet Pipeline

29.3.2 Threading

29.3.3 Important Data Structures

Introduction

This section explains the most important Suricata Data structures.

For a complete overview, see the doxygen: https://doxygen.openinfosecfoundation.org

29.3.4 Engines

Flow

Stream

Inspection of raw stream data

Stream Engine

Suricata's Stream Engine tracks and processes all the TCP stream data. Its responsibilities include

- TCP segment reassembly
- TCP data normalization
- gap management and handling
- maintaining internal caches
- handling of special cases like TCP URG ptr
- applying user-defined constraints like stream depth etc

for IDS as well as inline mode.

Internal storage of stream data

For a stream with small gaps, a Red Black Tree is used to store the streaming buffer blocks.

For a stream with large gaps (>=262144 bytes), regions (list of blocks of data) are used.

For a stream without gaps, one continuous streaming buffer is used (i.e. just one region).

These different data structures are used in a quest to make efficient use of memory in exceptional and regular conditions.

Role of stream reassembly

TCP stream data can arrive in any manner. For example, 100 bytes of data can arrive as 100 bytes at once or 1 byte at a time in 100 segments. The possibilities are insanely high! So, if it's 100 bytes of data, there are 2^{99} ways this data can be received in a world where this data arrives in order!

Hence, it is important for the engine to reassemble the TCP stream data to avoid unnecessary inspection on incomplete data and to avoid leaving room for evasion techniques based on small segments. Stream reassembly makes sure that the data to be matched upon is reliable.

Role of Detection Engine and Applayer Parser

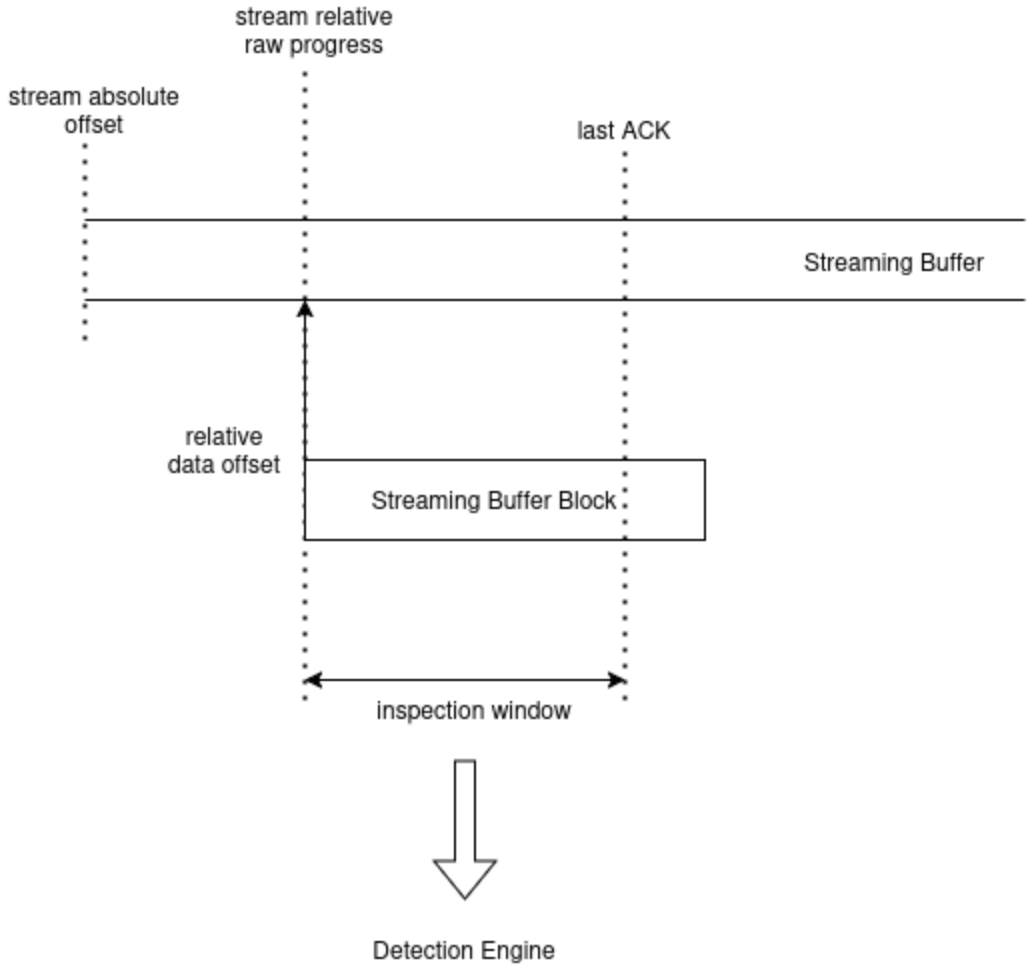
In order to conduct inspection on certain stream data, the Detection Engine has to request the Stream Engine for data. Doing this for every parseable data can be expensive and unreliable, so, the engine requests data in chunks. The size of these chunks can be defined in *suricata.yaml*. It is recommended to randomize the chunk size to avoid possible evasions on predictable boundaries. By default, the chunk size is randomized by Suricata.

Note that in some cases these chunk sizes may be too far into the future resulting in delayed inspection of data. This could lead to several issues like the one listed in Bug 7004. To deal with this, most applayer parsers request inspection of data as soon as they have fully and reliably parsed a certain entity like a request or a response in a respective direction.

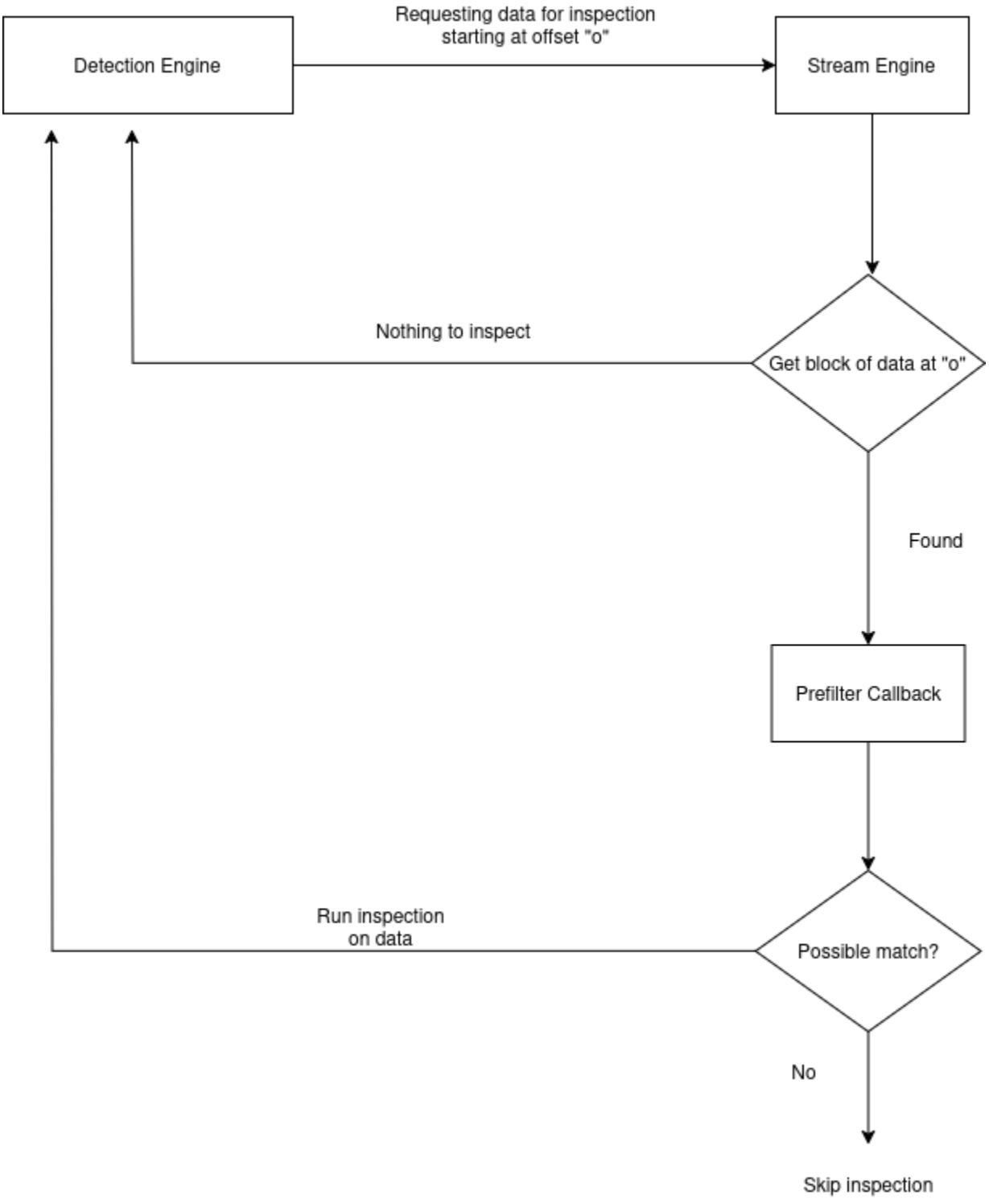
It is important to note that the inspection window can be limited by certain special conditions like stream depth being reached or end of stream being reached, etc.

Tracking of inspection

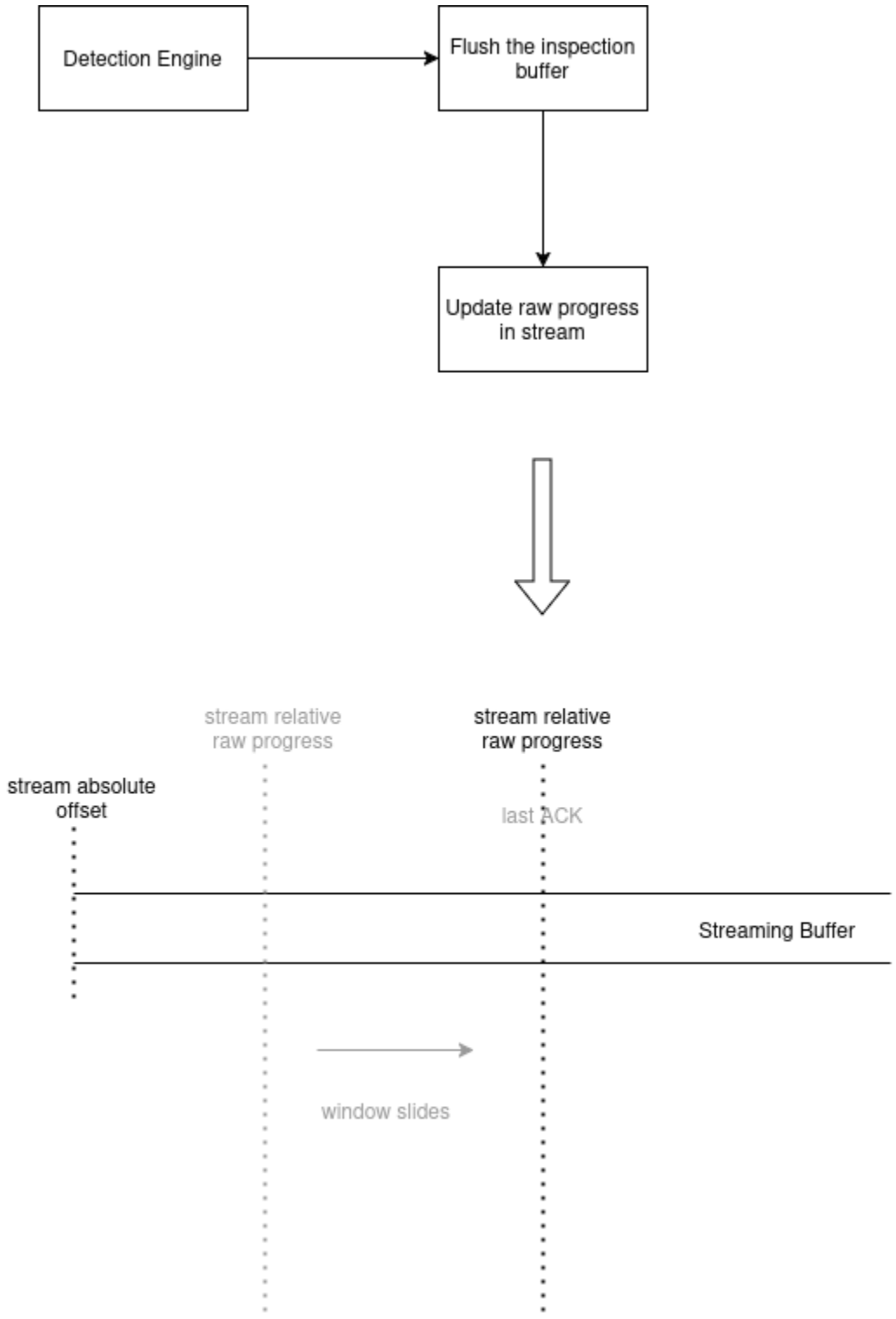
The Stream Engine must keep track of the point until which the inspection is already done. This helps the engine know what data has been consumed and can be slid out of the window. For a given stream without gaps, in IDS mode, from a very high level, assuming there are no overlaps in the tracking/data, no special conditions at play, the oversimplified tracking would look like the following.



On a very high level, the communication that takes place between the Detection Engine and the Stream Engine about data inspection is as follows.



Of course this means that the Detection Engine also maintains a copy of the raw progress of the data it has consumed so far. After the inspection is completed, the streaming buffer window slides if the data was consumed successfully. Additionally, the relative raw tracker is updated.



Relevant configuration

The following *suricata.yaml* settings can impact the internal inspection of data.

Stream Engine related settings:

```
stream:
  memcap: 64 MiB
  #memcap-policy: ignore
  checksum-validation: yes      # reject incorrect csums
  #midstream: false
  #midstream-policy: ignore
  inline: auto                  # auto will use inline mode in IPS mode, yes#
  reassembly:
    urgent:
      policy: oob              # drop, inline, oob (1 byte, see RFC 6093, 3.#
      oob-limit-policy: drop
    memcap: 256 MiB
    #memcap-policy: ignore
    depth: 1 MiB                # reassemble 1 MiB into a stream
    toserver-chunk-size: 2560
    toclient-chunk-size: 2560
    randomize-chunk-size: yes
    #randomize-chunk-range: 10
    #raw: yes
    #segment-prealloc: 2048
    #check-overlap-different-data: true
    #max-regions: 8
```

Prefilter/MPM related settings:

```
mpm-algo: hs
```

Defrag

Protocol detection

For each flow, Suricata will try to recognize the application layer protocol.

Protocol detection is run for TCP and UDP flows. Protocol detection is run (generally) independently for both directions of the flow. A flow can change its app-layer protocol during its lifetime (TLS upgrade for example). Protocol detection can, in the midstream case, reverse a flow direction. (If the first packet we see is a DNS over UDP response for example.)

Decision process

For each flow+direction, Suricata tries the following:

1. Multi pattern matching (port-independent)

Each app-layer protocol may register a set of patterns for each direction. (for example `HTTP/1.` for HTTP1 responses.)

As this is done by multi-pattern matching, this method scales, meaning that its CPU time cost is O(1) relative to the number of protocols and patterns. This is why it is the first method being run.

Debug validation ensures that the same pattern is not registered for multiple protocols (as may have happened with SIP and HTTP1).

An app-layer may also register a pattern with a probing parser, meaning that it will only recognise the protocol if: first the pattern is found, and then the probing parser also matches.

2. Probing parser

Each app-layer protocol may register arbitrary code to recognize a protocol. This code will only be run for some configured ports.

The probing function returns one of the 3 values - ALPROTO_FAILED : this is definitely not the protocol - ALPROTO_UNKNOWN : needs more data to take a decision - ALPROTO_XYZ : if it is indeed protocol xyz

An application-layer protocol can have both a set of patterns registered, and a probing parser.

3. Expectations

This is used now only for FTP-DATA. A flow can set an expected flow between a source IP and a server IP+port.

Output

For each flow event, we have different fields that represent the application layer protocol:

- "app_proto": the final app-layer protocol detected and parsed by Suricata
- "app_proto_tc": the app-layer protocol detected by Suricata in the direction to client, only logged if different than the app_proto
- "app_proto_ts": the app-layer protocol detected by Suricata in the direction to server, only logged if different than the app_proto
- "app_proto_orig": the original app-layer protocol detected by Suricata if the flow changed its protocol
- "app_proto_expected": the expected app-layer protocol if the flow changed its protocol to an unexpected protocol

> **Note**
>
> For detection the keyword *app-layer-protocol* may be used for these different fields.

Suricata also emits anomalies about protocol detection (for which you can use rules with `app-layer-event` keyword):

- APPLAYER_DETECT_PROTOCOL_ONLY_ONE_DIRECTION : only one side was recognised, the other is unknown
- APPLAYER_MISMATCH_PROTOCOL_BOTH_DIRECTIONS : the two sides were recognised but are different
- APPLAYER_PROTO_DETECTION_SKIPPED : no side was recognised
- APPLAYER_UNEXPECTED_PROTOCOL : a protocol change was requested to a specific one, but this specific protocol was not recognised
- APPLAYER_NO_TLS_AFTER_STARTTLS : same as above, but specialized for TLS
- APPLAYER_WRONG_DIRECTION_FIRST_DATA : the protocol recognised received the first data in the unexpected side (like HTTP1 flow beginning by a response)

Suricata stats events also count the number of flows per app-layer protocol : `.stats.app_layer.flow.xyz` for xyz protocol. For the app-layer protocols that can be recognised above both TCP and UDP, these counters are split in 2 fields like `nfs_tcp` and `nfs_udp`. These statistics are known to be not entirely consistent with the number of flows for a certain app-layer protocol (because of protocol change for a known edge case).

29.4 Extending Suricata

29.4.1 Packet Capture

29.4.2 Packet Decoder

29.4.3 App-Layer

Application Layer Overview

Table of Contents

- *Application Layer Overview*
  - *Parser*
    - *Transactions*
    - *Gap support*
  - *Logger*
  - *Detection engine*

This section aims to give an overview of what is needed to add an application-layer protocol to Suricata.

After a generic first step of collecting data about this application-layer protocol, especially pcaps for testing, we can dive into the Suricata specifics.

An application-layer protocol has three logic components in Suricata:

- parser
- logger
- detecting keywords

Both detection engine and logger will depend on the processing done by the parser.

For security reasons, we now develop application-layer protocol code only in Rust and not in C.

The script `scripts/setup-app-layer.py` may help you get started for adding a new app-layer protocol.

Parser

The parser is described by an instance of the structure `RustParser`.

A parser has:

- a name (where it is better to avoid dashes)
- an ipproto (if an app-layer is both over UDP and TCP, it needs to be registered with 2 RustParser)
- flags: only one flag `APP_LAYER_PARSER_OPT_ACCEPT_GAPS`
- some app-layer detection logic see *Protocol detection*.
- some logic around (one) State and (one) Transaction structures
- some stringer functions (frames, events)

So each app-layer protocol needs to define two structures: one State and one Transaction. A State will live throughout the flow (or until there is a protocol change in the flow). As such, it is useful to retain data that needs such a scope (for example HTTP2 dynamic headers table). And it is also useful if the parsing uses a state-machine logic, for example for file streaming. A State will own a list of *Transactions*.

Transactions

Transactions are the basic logical unit used by Suricata for an application-layer protocol.

The big decision is how to assemble PDUs into a transaction. Simplest design is to have one transaction per PDU (like DNS). But it may add value for example to combine request and response into a single transaction (like HTTP).

The `RustParser` structure contains callbacks to parse network traffic in both directions. These callbacks will create the transactions.

For protocols over TCP, this callback has to loop as one callback may run for a network traffic containing multiple PDUs, and thus resulting in the creation of multiple transactions.

> **Note**
>
> If a protocol may have multiple long-lived transactions, it is good to enforce limits on the number of live transactions, and bound any other data owned by the State.

In case of parsing anomalies, a transaction can set anomaly events, which are specific to the application-layer protocol. There is currently no good standardized way to have this kind of event outside transactions.

Gap support

It is good to develop an app-layer support first without gap support, then improve it by adding gap support.

Pcaps for testing can be created by removing some packets in previous testing pcaps.

After adding the flag `APP_LAYER_PARSER_OPT_ACCEPT_GAPS`, a generic way to handle this is:

- add two booleans to the State, like request_gap and response_gap
- have the parsing functions set these booleans `if stream_slice.is_gap()`
- have the parsing functions test these booleans, and try to resync with a beginning of a PDU

> **Note**
>
> This generic best-effort approach is vulnerable to request/response smuggling.

Another less generic approach is to handle gaps only in the case the gap happens in the middle of a known-length PDU (like HTTP1 content-length).

Logger

Besides the logging function, the logger also has a direction which may be either `LOG_DIR_PACKET` or `LOG_DIR_FLOW`.

UDP unidirectional transactions will be better interpreted using `LOG_DIR_PACKET` while TCP transactions are usually better interpreted using `LOG_DIR_FLOW`.

The logging function returns a boolean which must be false if there is nothing to log, for example if the resulting json object is empty.

Support for application-layer specific logging options is not yet standardized, especially for alerts.

Detection engine

A simple callback should register the keywords matching the log output fields.

Application Layer Frame Support

Table of Contents

- *Application Layer Frame Support*
  - *Baseline*
  - *General Concepts*
  - *Adding Frame Support to a Parser*
    - *Basic steps*
    - *Implementation Examples & API Callbacks*
      - *Rust*
      - *C code*
  - *Visual context*

Baseline

- Suricata rules format

General Concepts

Frame support was introduced with Suricata 7.0. Up until 6.0.x, Suricata's architecture and state of parsers meant that the network traffic available to the detection engine was just a stream of data, without detail about higher level parsers.

Note

For Suricata, *Frame* is a generic term that can represent any unit of network data we are interested in, which could be comprised of one or several records of other, lower level protocol(s). Frames work as "stream annotations", allowing Suricata to tell the detection engine what type of record exists at a specific offset in the stream.

The normal pipeline of detection in Suricata implied that:

- Certain rules could be quite costly performance-wise. This happened because the same stream could be inspected several times for different rules, since for certain signatures the detection is done when Suricata is still inspecting a lower level stream, not the application layer protocol (e.g., *TCP* traffic, in place of *SMB* one);
- Rules could be difficult and tedious to write (and read), requiring that writers went in byte-detail to express matching on specific payload patterns.

What the Frame support offers is the ability to "point" to a specific portion of stream and identify what type of traffic Suricata is looking at. Then, as the engine reassembles the stream, one can have "read access" to that portion of the stream, aggregating concepts like what type of application layer protocol that is, and differentiating between `header`, `data` or even protocol versions (*SMB1*, *SMB2*...).

The goal of the stream *Frame* is to expose application layer protocol PDUs and other such arbitrary elements to the detection engine directly, instead of relying on Transactions. The main purpose is to bring *TCP data* processing times

down by specialising/ filtering down traffic detection.

Adding Frame Support to a Parser

The application layer parser exposes frames it supports to the detect engine, by tagging them as they're parsed. The rest works automatically.

In order to allow the engine to identify frames for records of a given application layer parser, thought must be given as to which frames make sense for the specific protocol you are handling. Some parsers may have clear `header` and `data` fields that form its *protocol data unit* (pdu). For others, the distinction might be between `request` and `response`, only. Whereas for others it may make sense to have specific types of data. This is better understood by seeing the different types of frame keywords, which vary on a per-protocol basis.

It is also important to keep follow naming conventions when defining Frame Types. While a protocol may have strong naming standards for certain structures, do compare those with what Suricata already has registered:

- `hdr`: used for the record header portion
- `data`: is used for the record data portion
- `pdu`: unless documented otherwise, means the whole record, comprising `hdr` and `data`
- `request`: a message from a client to a server
- `response`: a message from a server to a client

Basic steps

Once the frame types that make sense for a given protocol are defined, the basic steps for adding them are:

- create an enum with the frame types;
- identify the parsing function(s) where application layer records are parsed;
- identify the correct moment to register the frames;
- use the Frame API calls directly or build upon them and use your functions to register the frames;
- register the relevant frame callbacks when registering the parser.

Once these are done, you can enable frame eve-output to confirm that your frames are being properly registered. It is important to notice that some hard coded limits could influence what you see on the logs (max size of log output; type of logging for the payload, cf. https://redmine.openinfosecfoundation.org/issues/4988).

If all the steps are successfully followed, you should be able to write a rule using the *frame* keyword and the frame types you have registered with the application layer parser.

Using the *SMB* parser as example, before frame support, a rule would look like:

```
alert tcp ... flow:to_server; content:"|ff|SMB"; content:"some smb 1 issue";
```

With frame support, one is able to do:

```
alert smb ... flow:to_server; frame:smb1.data; content:"some smb 1 issue";
```

Implementation Examples & API Callbacks

Though the steps are the same, there are a few differences when implementing frame support in Rust or in C. The following sections elaborate on that, as well as on the process itself. (Note that the code snippets have omitted portions of code that weren't so relevant to this document).

Rust

This section shows how Frame support is added in Rust, using examples from the SIP parser, and the telnet parser.

Define the frame types. The frame types are defined as an enum. In Rust, make sure to derive from the `AppLayerFrameType`:

Listing 1: rust/src/sip/sip.rs

```
#[derive(AppLayerFrameType)]
pub enum SIPFrameType {
    Pdu,
    RequestLine,
    ResponseLine,
    RequestHeaders,
    ResponseHeaders,
    RequestBody,
    ResponseBody,
}
```

Frame registering. Some understanding of the parser will be needed in order to find where the frames should be registered. It makes sense that it will happen when the input stream is being parsed into records. See when some pdu and request frames are created for SIP:

Listing 2: rust/src/sip/sip.rs

```
fn parse_request(&mut self, flow: *mut Flow, stream_slice: StreamSlice) ->␣
↪AppLayerResult {
    let input = stream_slice.as_slice();
    let _pdu = Frame::new(
        flow,
        &stream_slice,
        input,
        input.len() as i64,
        SIPFrameType::Pdu as u8,
        None,
    );
    SCLogDebug!("ts: pdu {:?}", _pdu);

    match parse_request(input) {
        Ok((_, request)) => {
            let mut tx = self.new_tx(Direction::ToServer);
            sip_frames_ts(flow, &stream_slice, &request, tx.id);
            tx.request = Some(request);
            if let Ok((_, req_line)) = sip_take_line(input) {
                tx.request_line = req_line;
            }
            self.transactions.push_back(tx);
            return AppLayerResult::ok();
        }
```

Note

when to create PDU frames

The standard approach we follow for frame registration is that a frame pdu will always be created, regardless of parser status (in practice, before the parser is called). The other frames are then created when and if only the parser succeeds.

Use the Frame API or build upon them as needed. These are the frame registration functions highlighted above:

Listing 3: rust/src/sip/sip.rs

```
fn sip_frames_ts(flow: *mut Flow, stream_slice: &StreamSlice, r: &Request, tx_id: u64) {
    let oi = stream_slice.as_slice();
    let _f = Frame::new(
        flow,
        stream_slice,
        oi,
        r.request_line_len as i64,
        SIPFrameType::RequestLine as u8,
        Some(tx_id),
    );
    SCLogDebug!("ts: request_line {:?}", _f);
    let hi = &oi[r.request_line_len as usize..];
    let _f = Frame::new(
        flow,
        stream_slice,
        hi,
        r.headers_len as i64,
        SIPFrameType::RequestHeaders as u8,
        Some(tx_id),
    );
    SCLogDebug!("ts: request_headers {:?}", _f);
    if r.body_len > 0 {
        let bi = &oi[r.body_offset as usize..];
        let _f = Frame::new(
            flow,
            stream_slice,
            bi,
            r.body_len as i64,
            SIPFrameType::RequestBody as u8,
            Some(tx_id),
        );
        SCLogDebug!("ts: request_body {:?}", _f);
    }
}
```

Register relevant frame callbacks. As these are inferred from the `#[derive(AppLayerFrameType)]` statement, all that is needed is:

Listing 4: rust/src/sip/sip.rs

```
get_frame_id_by_name: Some(SIPFrameType::ffi_id_from_name),
get_frame_name_by_id: Some(SIPFrameType::ffi_name_from_id),
```

> **Note**
>
> on frame_len
>
> For protocols which search for an end of frame char, like telnet, indicate unknown length by passing `-1`. Once the length is known, it must be updated. For those where length is a field in the record (e.g. *SIP*), the frame is set to match said length, even if that is bigger than the current input

The telnet parser has examples of using the Frame API directly for registering telnet frames, and also illustrates how that is done when length is not yet known:

Listing 5: rust/src/telnet/telnet.rs

```
fn parse_request(
    &mut self, flow: *mut Flow, stream_slice: &StreamSlice, input: &[u8],
) -> AppLayerResult {
    let mut start = input;
    while !start.is_empty() {
        if self.request_frame.is_none() {
            self.request_frame = Frame::new(
                flow,
                stream_slice,
                start,
                -1_i64,
                TelnetFrameType::Pdu as u8,
                None,
            );
        }
        if self.request_specific_frame.is_none() {
            if let Ok((_, is_ctl)) = parser::peek_message_is_ctl(start) {
                let f = if is_ctl {
                    Frame::new(
                        flow,
                        stream_slice,
                        start,
                        -1_i64,
                        TelnetFrameType::Ctl as u8,
                        None,
                    )
                } else {
                    Frame::new(
                        flow,
                        stream_slice,
                        start,
```

We then update length later on (note especially lines 3 and 10):

Listing 6: rust/src/telnet/telnet.rs

```
1 match parser::parse_message(start) {
2     Ok((rem, request)) => {
3         let consumed = start.len() - rem.len();
4         if rem.len() == start.len() {
```

(continues on next page)

(continued from previous page)

```
            panic!("lockup");
        }
        start = rem;

        if let Some(frame) = &self.request_frame {
            frame.set_len(flow, consumed as i64);
```

The Frame API calls parameters represent:

- `flow`: dedicated data type, carries specific flow-related data
- `stream_slice`: dedicated data type, carries stream data, shown further bellow
- `frame_start`: a pointer to the start of the frame buffer in the stream (`cur_i` in the SMB code snippet)
- `frame_len`: what we expect the frame length to be (the engine may need to wait until it has enough data. See what is done in the telnet snippet request frames registering)
- `frame_type`: type of frame it's being registering (defined in an enum, as shown further above)
- `tx_id`: an optional transaction id, if the frame belongs to a transaction. May be set later like *frame_len*

`StreamSlice` contains the input data to the parser, alongside other Stream-related data important in parsing context. Definition is found in *applayer.rs*:

Listing 7: src/app-layer-parser.h

```
typedef struct StreamSlice {
    const uint8_t *input;
    uint32_t input_len;
    /// STREAM_* flags
    uint8_t flags;
    uint64_t offset;
} StreamSlice;
```

C code

Implementing Frame support in C involves a bit more manual work, as one cannot make use of the Rust derives. Code snippets from the *HTTP* parser:

Defining the frame types with the enum means:

Listing 8: src/app-layer-htp.c

```
enum HttpFrameTypes {
    HTTP_FRAME_REQUEST,
    HTTP_FRAME_RESPONSE,
};

SCEnumCharMap http_frame_table[] = {
    {
            "request",
            HTTP_FRAME_REQUEST,
    },
    {
            "response",
```

(continues on next page)

(continued from previous page)

```
            HTTP_FRAME_RESPONSE,
    },
    { NULL, -1 },
};
```

The HTTP parser uses the Frame registration functions from the C API (`app-layer-frames.c`) directly for registering request Frames. Here we also don't know the length yet. The `0` indicates flow direction: `toserver`, and `1` would be used for `toclient`:

Listing 9: src/app-layer-htp.c

```
Frame *frame = AppLayerFrameNewByAbsoluteOffset(
        hstate->f, hstate->slice, consumed, -1, 0, HTTP_FRAME_REQUEST);
if (frame) {
    SCLogDebug("frame %p/%" PRIi64, frame, frame->id);
    hstate->request_frame_id = frame->id;
    AppLayerFrameSetTxId(frame, HtpGetActiveRequestTxID(hstate));
}
```

Updating `frame->len` later:

Listing 10: src/app-layer-htp.c

```
if (hstate->request_frame_id > 0) {
    Frame *frame = AppLayerFrameGetById(hstate->f, 0, hstate->request_frame_id);
    if (frame) {
        const uint64_t request_size = abs_right_edge - hstate->last_request_data_stamp;

        SCLogDebug("HTTP request complete: data offset %" PRIu64 ", request_size %"␣
↪PRIu64,
                hstate->last_request_data_stamp, request_size);
        SCLogDebug("frame %p/%" PRIi64 " setting len to  %" PRIu64, frame, frame->id,
                request_size);
        frame->len = (int64_t)request_size;
```

Register relevant callbacks (note that the actual functions will also have to be written, for C):

Listing 11: src/app-layer-htp.c

```
AppLayerParserRegisterGetFrameFuncs(
        IPPROTO_TCP, ALPROTO_HTTP1, HTTPGetFrameIdByName, HTTPGetFrameNameById);
```

Note

The `GetFrameIdByName` functions can be "probed", so they should not generate any output or that could be misleading (for instance, Suricata generating a log message stating that a valid frame type is unknown).

Visual context

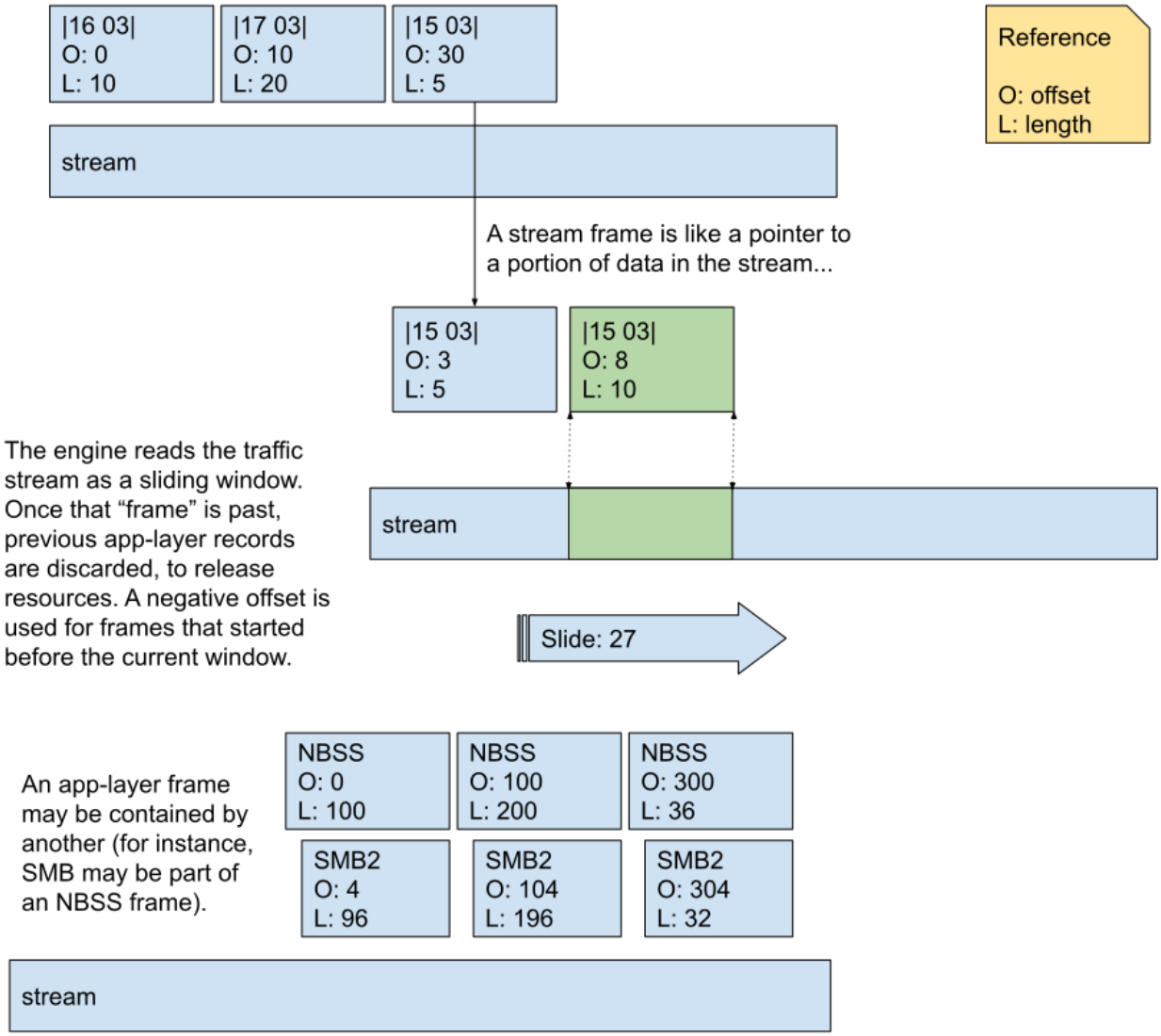
`input` and `input_len` are used to calculate the proper offset, for storing the frame. The stream buffer slides forward, so frame offsets/frames have to be updated. The *relative offset* (`rel_offset`) reflects that:

```
Start:
[ stream ]
  [ frame   ..........]
   rel_offset: 2
   len: 19

Slide:
     [ stream ]
[ frame ....          .]
 rel_offset: -10
 len: 19

Slide:
            [ stream ]
[ frame ...........    ]
 rel_offset: -16
 len: 19
```

The way the engine handles stream frames can be illustrated as follows:



Parsers

Callbacks

The API calls callbacks that are registered at the start of the program.

The function prototype is:

```
typedef AppLayerResult (*AppLayerParserFPtr)(Flow *f, void *protocol_state,
        AppLayerParserState *pstate,
        const uint8_t *buf, uint32_t buf_len,
        void *local_storage, const uint8_t flags);
```

Examples

A C example:

```
static AppLayerResult HTPHandleRequestData(Flow *f, void *htp_state,
        AppLayerParserState *pstate,
        const uint8_t *input, uint32_t input_len,
        void *local_data, const uint8_t flags);
```

In Rust, the callbacks are similar.

```
#[no_mangle]
pub extern "C" fn rs_dns_parse_response_tcp(_flow: *const core::Flow,
        state: *mut std::os::raw::c_void,
        _pstate: *mut AppLayerParserState,
        input: *const u8,
        input_len: u32,
        _data: *mut std::os::raw::c_void,
        _flags: u8)
-> AppLayerResult
```

Return Types

Parsers return the type *AppLayerResult*.

There are 3 possible results:

- *APP_LAYER_OK* - parser consumed the data successfully
- *APP_LAYER_ERROR* - parser encountered a unrecoverable error
- *APP_LAYER_INCOMPLETE(c,n)* - parser consumed *c* bytes, and needs *n* more before being called again

Rust parsers follow the same logic, but can return

- *AppLayerResult::ok()*
- *AppLayerResult::err()*
- *AppLayerResult::incomplete(c,n)*

For *i32* and *bool*, Rust parsers can also use *.into()*.

APP_LAYER_OK / AppLayerResult::ok()

When a parser returns "OK", it signals to the API that all data has been consumed. The parser will be called again when more data is available.

APP_LAYER_ERROR / AppLayerResult::err()

Returning "ERROR" from the parser indicates to the API that the parser encountered an unrecoverable error and the processing of the protocol should stop for the rest of this flow.

> **Note**
>
> This should not be used for recoverable errors. For those events should be set.

APP_LAYER_INCOMPLETE / AppLayerResult::incomplete()

Using "INCOMPLETE" a parser can indicate how much more data is needed. Many protocols use records that have the size as one of the first parameters. When the parser receives a partial record, it can read this value and then tell the API to only call the parser again when enough data is available.

consumed is used how much of the current data has been processed *needed* is the number of bytes that the parser needs on top of what was consumed.

Example:

```
[ 32 record 1 ][ 32 record 2 ][ 32 r.. ]
 0          31  32         63  64    72
                            ^   ^
consumed: 64 ---------------/   |
needed:   32 --------------------/
```

Note

"INCOMPLETE" is only supported for TCP

The parser will be called again when the *needed* data is available OR when the stream ends. In the latter case the data will be incomplete. It's up to the parser to decide what to do with it in this case.

Supporting incomplete data

In some cases it may be preferable to actually support dealing with incomplete records. For example protocols like SMB and NFS can use very large records during file transfers. Completely queuing these before processing could be a waste of resources. In such cases the "INCOMPLETE" logic could be used for just the record header, while the record data is streamed into the parser.

Transactions

Table of Contents

- *Transactions*
  - *General Concepts*
  - *How the engine uses transactions*
    - *Logging*
    - *Rule Matching*
  - *Progress Tracking*
    - *In Summary - Transactions and State*
  - *Examples*
    - *Enums*
    - *API Callbacks*
    - *Sequence Diagrams*

    * *Template Protocol*
  - *Work In Progress changes*
  - *Common words and abbreviations*

General Concepts

For Suricata, transactions are an abstraction that help with detecting and logging. An example of a complete transaction is a pair of messages in the form of a request (from client to server) and a response (from server to client) in HTTP.

In order to know when to log an event for a given protocol, the engine tracks the progress of each transaction - that is, when is it complete, or when it reaches a key intermediate state. They aid during the detection phase, when dealing with protocols that can have large PDUs (protocol data units), like TCP, in controlling state for partial rule matching -- in case of rules that mention more than one field.

Transactions are implemented and stored in the per-flow state. The engine interacts with them using a set of callbacks the parser registers.

How the engine uses transactions

All transactions share a common structure `AppLayerTxData` and a unique increasing id.

Logging

Suricata controls when logging should happen based on transaction completeness. For simpler protocols, such as `dns` or `ntp`, that will most likely happen once per transaction, by the time of its completion. In other cases, like with HTTP, this may happen at intermediary states.

In `OutputTxLog`, the engine will compare current state with the value defined for the logging to happen, per flow direction (`logger->tc_log_progress`, `logger->ts_log_progress`). If state is less than that value, the engine skips to the next logger. Code snippet from: suricata/src/output-tx.c:

```
static TmEcode OutputTxLog(ThreadVars *tv, Packet *p, void *thread_data)
{
    .
    .
    .
        if ((ts_eof && tc_eof) || last_pseudo) {
            SCLogDebug("EOF, so log now");
        } else {
            if (logger->LogCondition) {
                int r = logger->LogCondition(tv, p, alstate, tx, tx_id);
                if (r == FALSE) {
                    SCLogDebug("conditions not met, not logging");
                    goto next_logger;
                }
            } else {
                if (tx_progress_tc < logger->tc_log_progress) {
                    SCLogDebug("progress not far enough, not logging");
                    goto next_logger;
                }

                if (tx_progress_ts < logger->ts_log_progress) {
```

(continues on next page)

(continued from previous page)

```
                    SCLogDebug("progress not far enough, not logging");
                    goto next_logger;
                }
            }
        }
    .
    .
    .
}
```

Rule Matching

Transaction progress is also used for certain keywords to know what is the minimum state before we can expect a match: until that, Suricata won't even try to look for the patterns.

As seen in `DetectAppLayerMpmRegister` that has `int progress` as parameter, and `DetectAppLayerInspectEngineRegister`, which expects `uint8_t tx_min_progress`, for instance. In the code snippet, `HTTP2ProgData` and `0` are the values passed to the functions - in the last example, for `FTPDATA`, the existence of a transaction implies that a file is being transferred. Hence the `0` value.

```
void DetectFiledataRegister(void)
{
    .
    .
    DetectAppLayerMpmRegister("file_data", SIG_FLAG_TOSERVER, 2,
            PrefilterMpmFiledataRegister, NULL,
            ALPROTO_HTTP2, HTTP2ProgData);
    DetectAppLayerMpmRegister("file_data", SIG_FLAG_TOCLIENT, 2,
            PrefilterMpmFiledataRegister, NULL,
            ALPROTO_HTTP2, HTTP2ProgData);
    .
    .
    DetectAppLayerInspectEngineRegister("file_data",
        ALPROTO_HTTP2, SIG_FLAG_TOCLIENT, HTTP2ProgData,
        DetectEngineInspectFiledata, NULL);
    DetectAppLayerInspectEngineRegister(
            "file_data", ALPROTO_FTPDATA, SIG_FLAG_TOSERVER, 0,␣
↪DetectEngineInspectFiledata, NULL);
    .
    .
}
```

Progress Tracking

As a rule of thumb, transactions will follow a request-response model: if a transaction has had a request and a response, it is complete.

But if a protocol has situations where a request or response won't expect or generate a message from its counterpart, it is also possible to have uni-directional transactions. In such cases, transaction is set to complete at the moment of creation.

For example, DNS responses may be considered as completed transactions, because they also contain the request data, so all information needed for logging and detection can be found in the response.

In addition, for file transfer protocols, or similar ones where there may be several messages before the file exchange is completed (NFS, SMB), it is possible to create a level of abstraction to handle such complexity. This could be achieved by adding phases to the model implemented by the protocol (e.g., protocol negotiation phase (SMB), request parsed (HTTP), and so on).

This is controlled by implementing progress states. In Suricata, those will be enums that are incremented as the parsing progresses. A state will start at 0. The higher its value, the closer the transaction would be to completion. Due to how the engine tracks detection across states, there is an upper limit of 48 to the state progress (it must be < 48).

Entering a state does not imply that the state's data is present: a state is entered when the unit it names starts being parsed, and the data (buffers, extracted fields) exists only once the unit has been fully parsed. Keywords reading that data must tolerate the unit still arriving - buffer keywords bail out when the buffer's GetData callback reports the buffer unavailable, and keywords reading extracted fields must gate on the field being present rather than assume the state implies the data.

The engine interacts with transactions' state using a set of callbacks the parser registers. State is defined per flow direction (`STREAM_TOSERVER` / `STREAM_TOCLIENT`).

In Summary - Transactions and State

- Initial State value: `0`.
- Simpler scenarios: State is simply a bool. `1` represents transaction completion, per direction.
- Complex Transaction State in Suricata: `enum` (Rust: `i32`). Completion is indicated by the highest enum value (some examples are: SSH, HTTP, HTTP2, DNS, SMB).

Examples

This section shares some examples from Suricata codebase, to help visualize how Transactions work and are handled by the engine.

Enums

Code snippet from: rust/src/ssh/ssh.rs:

```
pub enum SSHConnectionState {
    SshStateBanner = 0,
    SshStateKex = 1,
    SshStateSession = 2,
    SshStateDone = 3,
}
```

From src/app-layer-ftp.h:

```
enum {
    FTP_STATE_IN_PROGRESS,
    FTP_STATE_PORT_DONE,
    FTP_STATE_FINISHED,
};
```

From src/app-layer-ssl.h:

```
enum {
    TLS_STATE_IN_PROGRESS = 0,
    TLS_STATE_CERT_READY = 1,
```

(continues on next page)

(continued from previous page)

```
    TLS_HANDSHAKE_DONE = 2,
    TLS_STATE_FINISHED = 3
};
```

API Callbacks

In Rust, this is done via the RustParser struct. As seen in rust/src/applayer.rs:

```
/// Rust parser declaration
pub struct RustParser {
        .
        .
        .
    /// Progress values at which the tx is considered complete in a direction
    pub tx_comp_st_ts:      c_int,
    pub tx_comp_st_tc:      c_int,
    .
    .
    .
}
```

In C, the callback API is:

```
void AppLayerParserRegisterStateProgressCompletionStatus(
    AppProto alproto, const int ts, const int tc)
```

Simple scenario described, in Rust:

rust/src/dhcp/dhcp.rs:

```
tx_comp_st_ts: 1,
tx_comp_st_tc: 1,
```

For SSH, this looks like this:

rust/src/ssh/ssh.rs:

```
tx_comp_st_ts: SSHConnectionState::SshStateDone as i32,
tx_comp_st_tc: SSHConnectionState::SshStateDone as i32,
```

In C, callback usage would be as follows:

src/app-layer-dcerpc.c:

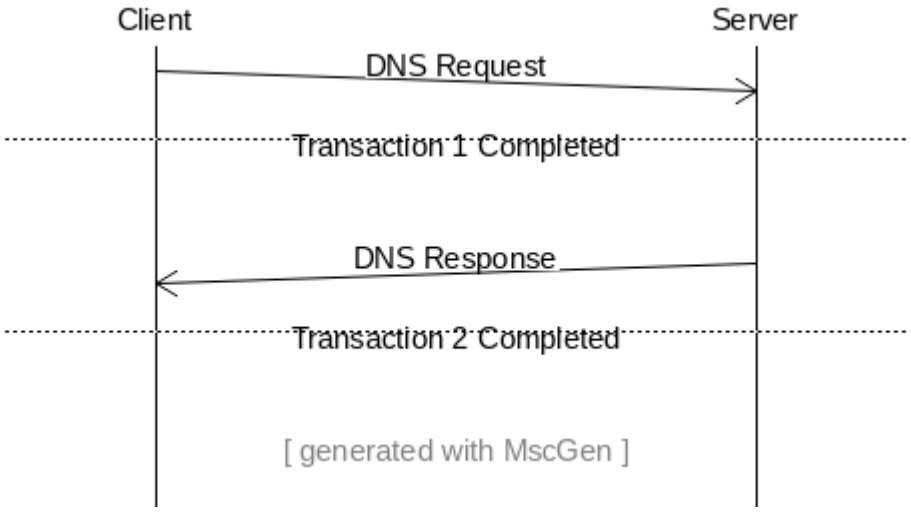
```
AppLayerParserRegisterStateProgressCompletionStatus(ALPROTO_DCERPC, 1, 1);
```

src/app-layer-ftp.c:

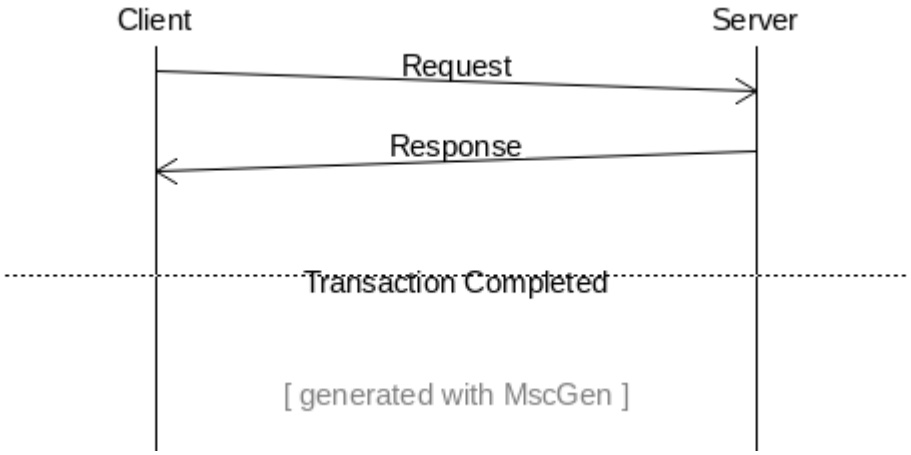
```
AppLayerParserRegisterStateProgressCompletionStatus(
    ALPROTO_FTP, FTP_STATE_FINISHED, FTP_STATE_FINISHED);
```

Sequence Diagrams

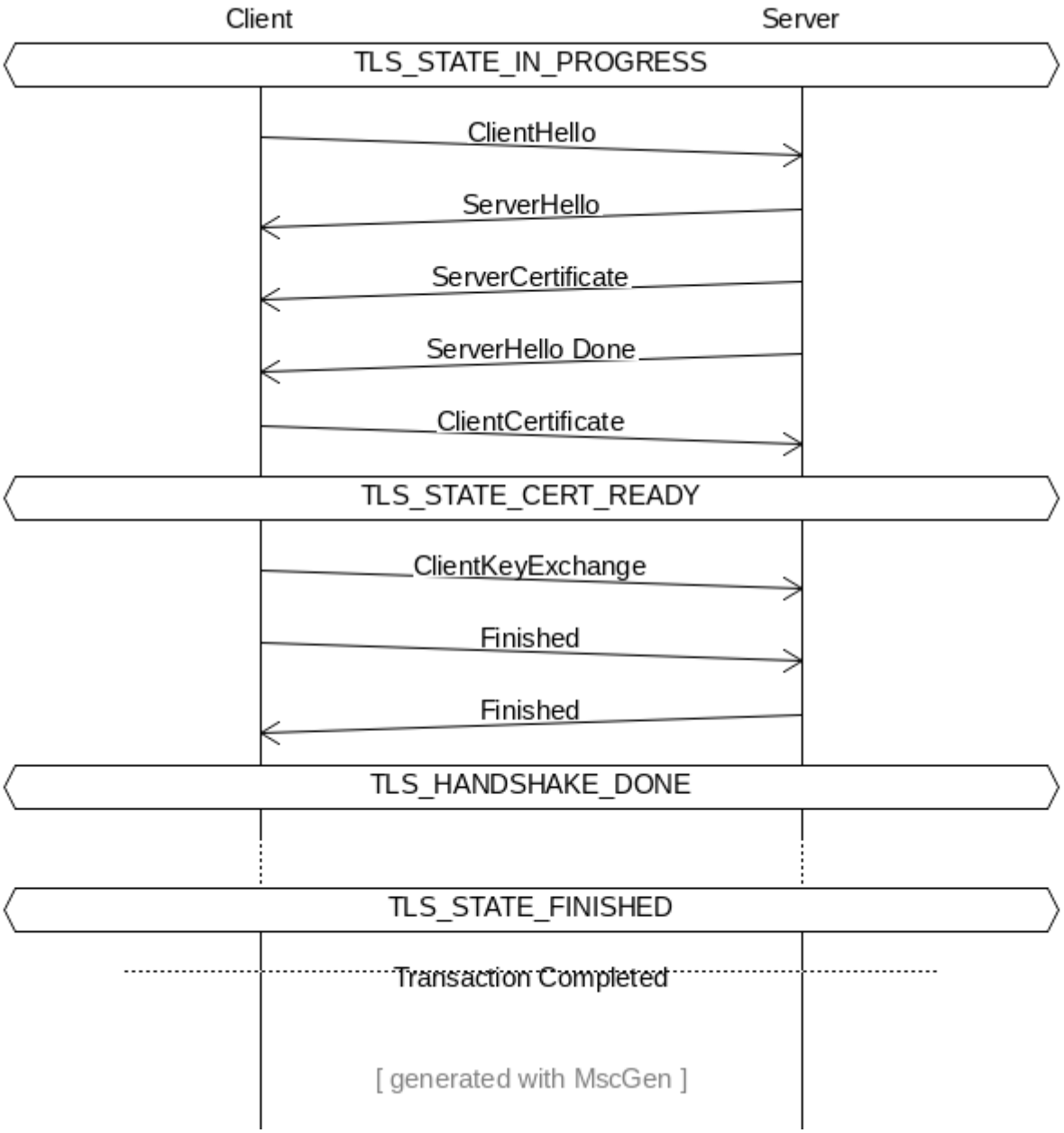
A DNS transaction in Suricata can be considered unidirectional:



An HTTP2 transaction is an example of a bidirectional transaction, in Suricata (note that, while HTTP2 may have multiple streams, those are mapped to transactions in Suricata. They run in parallel, scenario not shown in this Sequence Diagram - which shows one transaction, only):



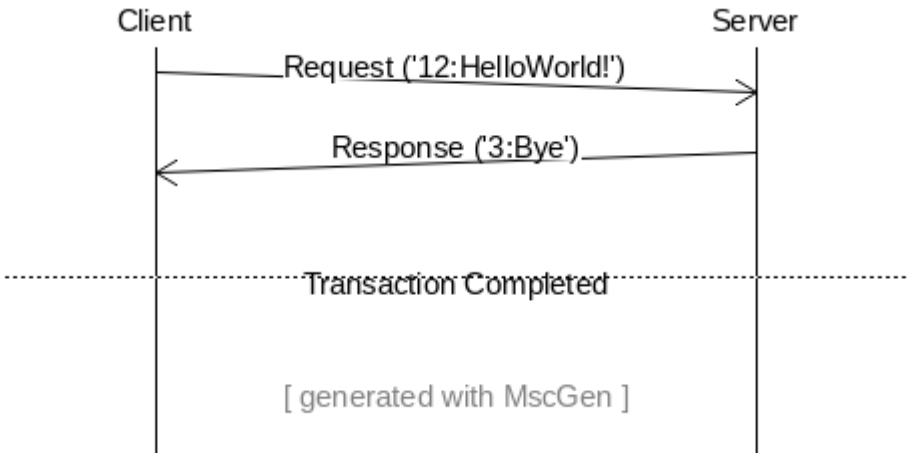
A TLS Handshake is a more complex example, where several messages are exchanged before the transaction is considered completed:



Template Protocol

Suricata has a template protocol for educational purposes, which has simple bidirectional transactions.

A completed transaction for the template looks like this:



Following are the functions that check whether a transaction is considered completed, for the Template Protocol. Those are called by the Suricata API. Similar functions exist for each protocol, and may present implementation differences, based on what is considered a transaction for that given protocol.

In C:

```
static int TemplateGetStateProgress(void *txv, uint8_t direction)
{
    TemplateTransaction *tx = txv;

    SCLogNotice("Transaction progress requested for tx ID %"PRIu64
        ", direction=0x%02x", tx->tx_id, direction);

    if (direction & STREAM_TOCLIENT && tx->response_done) {
        return 1;
    }
    else if (direction & STREAM_TOSERVER) {
        /* For the template, just the existence of the transaction means the
         * request is done. */
        return 1;
    }

    return 0;
}
```

And in Rust:

```
pub extern "C" fn rs_template_tx_get_alstate_progress(
    tx: *mut std::os::raw::c_void,
    _direction: u8,
) -> std::os::raw::c_int {
    let tx = cast_pointer!(tx, TemplateTransaction);

    // Transaction is done if we have a response.
    if tx.response.is_some() {
        return 1;
    }
    return 0;
}
```

Work In Progress changes

An ongoing effort is to limit the number of transactions per flow, to prevent Denial of Service (DoS) by quadratic complexity - a type of attack that may happen to protocols which can have multiple transactions at the same time - such as HTTP2 so-called streams (see https://redmine.openinfosecfoundation.org/issues/4530).

Common words and abbreviations

- al, applayer: application layer
- alproto: application layer protocol
- alstate: application layer state
- engine: refers to Suricata core detection logic
- flow: a bidirectional flow of packets with the same 5-tuple elements (protocol, source ip, destination ip, source port, destination port. Vlans can be added as well)
- PDU: Protocol Data Unit
- rs: rust
- tc: to client
- ts: to server
- tx: transaction

29.4.4 Detection

Transforms

Overview

Transforms modify the contents of an inspection buffer before content keywords inspect it. They are applied in the order they appear in the rule, with each transform's output becoming the next transform's input.

Execution Order

Transforms run at two points in the detection pipeline:

1. **Prefilter (MPM):** when the buffer is set up for the multi-pattern matcher. The MPM searches the *transformed* buffer for fast-pattern content.
2. **Full inspection:** when the rule's keywords are evaluated against the (transformed) buffer.

In both cases, transforms execute before detection-time keywords such as `byte_extract` and `byte_math`. The transformed buffer is what all subsequent keywords — including `content` — operate on.

Transform Identity

The engine deduplicates inspection buffers: rules that use the same buffer keyword with the same transform configuration share one pre-computed buffer. Equivalence is determined at rule load time by comparing an identity value that each transform instance produces from its configuration. Instances with identical identity share a buffer; instances with different identity (or where a transform produces no identity) get independent buffers.

For `xor` with a static key the key bytes are the identity. For a variable key the identity is the key's offset and byte count — so two rules reading their key from the same buffer location share a buffer, while two rules with different key locations each get their own correctly transformed buffer.

Rate Filter Callback

A callback can be registered for any signature hit whose action has been modified by the rate filter. This allows for the user to modify the action, if needed using their own custom logic.

For an example, see `examples/lib/custom/main.c` in the Suricata source code.

The Callback

The callback function will be called with the packet, signature details (sid, gid, rev), original action, the new action, and a user provided argument. It will only be called if the Suricata rate filter modified the action:

```
/**
 * \brief Function type for rate filter callback.
 *
 * This function should return the new action to be applied. If no change to the
 * action is to be made, the callback should return the current action provided
 * in the new_action parameter.
 */
typedef uint8_t (*SCDetectRateFilterFunc)(const Packet *p, uint32_t sid, uint32_t gid,␣
↪uint32_t rev,
       uint8_t original_action, uint8_t new_action, void *arg);
```

Callback Registration

To register the rate filter callback, use the `SCDetectEngineRegisterRateFilterCallback` function with your callback and a user provided argument which will be provided to the callback.

```
/**
 * \brief Register a callback when a rate_filter has been applied to
 *     an alert.
 *
 * This callback is added to the current detection engine and will be
 * copied to all future detection engines over rule reloads.
 */
bool SCDetectEngineRegisterRateFilterCallback(SCDetectRateFilterFunc cb, void *arg);
```

29.4.5 Exception Policies

This section offers an overview on the Suricata Exception Policies from a development perspective, for those interested in extending them. For more on usage, configuration etc, see the chapter Exception Policies.

Exception Policies allow control on how the engine will behave when it reaches exception scenarios where it could lose visibility into traffic being processed, for instance.

Briefly, they give the ability to fail close or open, for a set of known conditions, currently related to (This may be expanded by future work):

- hitting memory capacity limits;
- encountering application layer protocol errors;
- picking a TCP session midstream.

When failing close or open, the chosen policies may affect:

- individual packets, or
- the whole flow.

In the Suricata documentation, it is common to refer to anything in this scope as Exception Policies, but more strictly, what the engine proactively does when it reaches any of those conditions is an Exception Policy.

Extending

The policy scope should be decided based on what is the setting about, as well as on its possible impacts. Allowing for a midstream pickup policy to affect individual packets of a flow, for instance, would only lead to the policy being effectively applied to every single packet, and thus this isn't allowed.

Besides those aspects, there are differences in impacts when the engine is running in *IDS* or *IPS* mode.

There are two main ways to extend Suricata adding new Exception Policies:

- add a new *policy*, that is, a new possible behavior;
- add a new *exception policy*, that is, covering another exception scenario.

The next sections will cover what are the steps and files to take into account when extending Suricata Exception Policies.

Adding a New Policy

This section indicates the main steps and entry points for changes when adding a new policy.

Currently Suricata supports the following exception policies:

Listing 12: src/util-exception-policy-types.h

```
enum ExceptionPolicy {
    EXCEPTION_POLICY_NOT_SET = 0,
    EXCEPTION_POLICY_AUTO,
    EXCEPTION_POLICY_PASS_PACKET,
    EXCEPTION_POLICY_PASS_FLOW,
    EXCEPTION_POLICY_BYPASS_FLOW,
    EXCEPTION_POLICY_DROP_PACKET,
    EXCEPTION_POLICY_DROP_FLOW,
    EXCEPTION_POLICY_REJECT,     /**< reject src */
    EXCEPTION_POLICY_REJECT_BOTH /**< reject both src and dest */
};
```

If a new policy is needed, once it is clear what the new policy type and nomenclature, will be, the following give an overview of where to go to add the patch.

For the behavior type and logic:

- src/util-exception-policy-types.h
- src/util-exception-policy.c

To indicate whether the new policy is valid for each scenario

- src/app-layer.c
- src/decode.c
- src/stream-tcp.c

Don't forget to document:

- suricata.yaml.in
- doc/userguide/configuration/suricata-yaml.rst
- doc/userguide/configuration/exception-policies.rst

Ensure the new policy is valid in the JSON logs, by adding it to:

- etc/schema.json

For an example PR on adding a new policy, see: https://github.com/OISF/suricata/pull/14225/files

Adding a New Exception Policy

When the goal is to expand the exception scenarios that the exception policies cover, the files to modify will depend on what the exception is.

Consider:

- What is the exception;
- When to get the Exception Policy setting;
- At what moment should the policy be applied.

Once you know how it should work, the related function calls are:

```
- ExceptionPolicyParse
- ExceptionPolicyApply
```

It will also be necessary to update:

- suricata.yaml.in - to add the new setting options;
- src/decode.h - `PacketDropReason` to add the corresponding drop reason.

Documentation updates:

- Exception Policy section: add new scenario to tables, add explanations;
- This DevGuide section, in case new files or exception conditions should be listed;
- Corresponding suricata-yaml section.

For an example PR on adding a new exception policy, see: https://github.com/OISF/suricata/pull/7791, especially commit aa5bb2c329aff5.

Testing

For any significant fixes or new features, tests are required. For Exception Policies, please add Suricata-Verify tests. To learn more about testing Suricata refer to:

- *Suricata Testing: Suricata Verify*
- Suricata-verify: https://github.com/OISF/suricata-verify/blob/master/README.md
- Suricata Testing 101 - Building Robust Security Tools (SuriCon 2024 talk by Haleema Khan and Modupe Falodun): https://www.youtube.com/watch?v=9gOGkrSIcXQ

29.4.6 Output

Low Level Logging

Suricata's alert, protocol, and other types of output are built up from a set of low level loggers. These loggers include:

- Packet logging (alerts)
- Flow logging
- Transaction logging (application layer)
- File information logging

- File data logging (file extraction)
- Statistics

These low level logging facilities are used to build up Suricata's logging include EVE, but they can also be hooked into by plugins or applications using Suricata as a library.

> **Note**
>
> At this time only a C API exists to hook into the low level logging functions.

The Suricata source code contains an example plugin demonstrating how to hook into some of these APIs. See https://github.com/OISF/suricata/blob/main/examples/plugins/c-custom-loggers/custom-logger.c.

Packet Logging

Packet loggers can be registered with the `SCOutputRegisterPacketLogger` function:

```
/** \brief Register a packet logger.
 *
 * \param logger_id An ID used to distinguish this logger from others
 *     while profiling.
 * \param name An informational name for this logger. Used only for
 *     debugging.
 * \param LogFunc A function that will be called to log each packet
 *     that passes the condition test.
 * \param ConditionFunc A function to test if the packet should be passed to
 *     the logging function.
 * \param initdata Initialization data that will pass to the
 *     ThreadInitFunc.
 * \param ThreadInitFunc Thread initialization function.
 * \param ThreadDeinitFunc Thread de-initialization function.
 *
 * \retval 0 on success, -1 on failure.
 */
int SCOutputRegisterPacketLogger(LoggerId logger_id, const char *name, PacketLogger␣
↪LogFunc,
       PacketLogCondition ConditionFunc, void *initdata, ThreadInitFunc,␣
↪ThreadDeinitFunc);
```

Flow Logging

Flow loggers can be registered with the `SCOutputRegisterFlowLogger` function:

```
/** \brief Register a flow logger.
 *
 * \param name An informational name for this logger. Used only for
 *     debugging.
 * \param LogFunc A function that will be called to log each flow.
 * \param initdata A pointer to initialization data that will be
 *     passed the ThreadInit.
 * \param ThreadInit Thread initialization callback.
 * \param ThreadDeinit Thread de-initialization callback.
```

(continues on next page)

(continued from previous page)

```
 *
 * \retval 0 on success, -1 on failure.
 */
int SCOutputRegisterFlowLogger(const char *name, FlowLogger LogFunc, void *initdata,
        ThreadInitFunc ThreadInit, ThreadDeinitFunc ThreadDeinit);
```

Transaction Logging

Transaction logger can be registered with the `SCOutputRegisterTxLogger` function:

Attention

Transaction loggers cannot be registered from a plugin at this time, see https://redmine.openinfosecfoundation.org/issues/7236 for more information.

```
/** \brief Register a transaction logger.
 *
 * \param logger_id An ID used to distinguish this logger from others
 *     while profiling. For transaction logging this is only used for
 *     some internal state tracking.
 *
 * \param name An informational name for this logger. Used for
 *     debugging.
 *
 * \param alproto The application layer protocol this logger is for,
 *     for example ALPROTO_DNS.
 *
 * \param LogFunc A pointer to the logging function.
 *
 * \param initdata Initialization data that will be provided to the
 *     ThreadInit callback.
 *
 * \param tc_log_progress The to_client progress state required for
 *     the log function to be called.
 *
 * \param ts_log_progress The to_server progress state required for
 *     the log function to be called.
 *
 * \param LogCondition A pointer to a function that will be called
 *     before the log function to test if the log function should be
 *     called.
 *
 * \param ThreadInitFunc Callback a thread initialization function,
 *     initdata will be provided.
 *
 * \param ThreadDeinitFunc Callback to a thread de-initialization
 *     function for cleanup.
 */
int SCOutputRegisterTxLogger(LoggerId id, const char *name, AppProto alproto, TxLogger␣
↪LogFunc,
```

(continues on next page)

(continued from previous page)

```
        void *, int tc_log_progress, int ts_log_progress, TxLoggerCondition LogCondition,
        ThreadInitFunc, ThreadDeinitFunc);
```

Stream Logging

Stream logging allows for the logging of streaming data such as TCP reassembled data and HTTP body data. The provided log function will be called each time a new chunk of data is available.

Stream loggers can be registered with the `SCOutputRegisterStreamingLogger` function:

```
/** \brief Register a streaming logger.
 *
 * \param logger_id An ID to uniquely identify this logger.
 *
 * \param name An informational name for this logger.
 *
 * \param LogFunc Pointer to logging function.
 *
 * \param initdata Initialization data that will be passed the
 *     ThreadInit.
 *
 * \param stream_type Type of stream to log, see
 *     SCOutputStreamingType.
 *
 * \param ThreadInit Pointer to thread initialization function.
 *
 * \param ThreadDeinit Pointer to thread de-initialization function.
 */
int SCOutputRegisterStreamingLogger(LoggerId logger_id, const char *name,␣
↪SCStreamingLogger LogFunc,
        void *initdata, enum SCOutputStreamingType stream_type, ThreadInitFunc␣
↪ThreadInit,
        ThreadDeinitFunc ThreadDeinit);
```

File Logging

File loggers can be registered with the `SCOutputRegisterFileLogger` function:

```
/** \brief Register a file logger.
 *
 * \param logger_id An ID used to distinguish this logger from others
 *     while profiling.
 *
 * \param name An informational name for this logger. Used only for
 *     debugging.
 *
 * \param LogFunc A function that will be called to log each file to be logged.
 *
 * \param initdata Initialization data that will pass to the
 *     ThreadInitFunc.
 *
 * \param ThreadInitFunc Thread initialization function.
```

(continues on next page)

(continued from previous page)

```
 *
 * \param ThreadDeinitFunc Thread de-initialization function.
 *
 * \retval 0 on success, -1 on failure.
 */
int SCOutputRegisterFileLogger(LoggerId id, const char *name, SCFileLogger LogFunc, void␣
↪*initdata,
        ThreadInitFunc ThreadInit, ThreadDeinitFunc ThreadDeinit);
```

File-data Logging

File-data loggers can be registered with the `SCOutputRegisterFileDataLogger` function:

```
/** \brief Register a file-data logger.
 *
 * \param logger_id An ID used to distinguish this logger from others
 *     while profiling.
 *
 * \param name An informational name for this logger. Used only for
 *     debugging.
 *
 * \param LogFunc A function that will be called to log each file-data.
 *
 * \param initdata Initialization data that will pass to the
 *     ThreadInitFunc.
 *
 * \param ThreadInitFunc Thread initialization function.
 *
 * \param ThreadDeinitFunc Thread de-initialization function.
 *
 * \retval 0 on success, -1 on failure.
 */
int SCOutputRegisterFiledataLogger(LoggerId id, const char *name, SCFiledataLogger␣
↪LogFunc,
        void *initdata, ThreadInitFunc ThreadInit, ThreadDeinitFunc ThreadDeinit);
```

29.4.7 EVE Filetypes

Introduction

The Suricata EVE/JSON output supports filetypes to extend how EVE records are processed and delivered. Custom filetypes provide alternatives to standard file output by implementing a file-like interface that Suricata can write to. These filetypes can send events to databases, sockets, or other destinations, and can even perform custom processing on the output before storing it.

EVE Filetype Life Cycle

The life-cycle of an EVE filetype along with the callbacks are discussed in `output-eve.h`:

```
/** \brief Structure used to define an EVE output file type.
 *
 * EVE filetypes implement an object with a file-like interface and
```

(continues on next page)

(continued from previous page)

```
 * are used to output EVE log records to files, syslog, or
 * database. They can be built-in such as the syslog (see
 * SyslogInitialize()) and nullsink (see NullLogInitialize()) outputs,
 * registered by a library user or dynamically loaded as a plugin.
 *
 * The life cycle of an EVE filetype is:
 *   - Init: called once for each EVE instance using this filetype
 *   - ThreadInit: called once for each output thread
 *   - Write: called for each log record
 *   - ThreadDeinit: called once for each output thread on exit
 *   - Deinit: called once for each EVE instance using this filetype on exit
 *
 * Examples:
 * - built-in syslog: \ref src/output-eve-syslog.c
 * - built-in nullsink: \ref src/output-eve-null.c
 * - example plugin: \ref examples/plugins/c-json-filetype/filetype.c
 *
 * ### Multi-Threaded Note:
 *
 * The EVE logging system can be configured by the Suricata user to
 * run in threaded or non-threaded modes. In the default non-threaded
 * mode, ThreadInit will only be called once and the filetype does not
 * need to be concerned with threads.
 *
 * However, in **threaded** mode, ThreadInit will be called multiple
 * times and the filetype needs to be thread aware and thread-safe. If
 * utilizing a unique resource such as a file for each thread then you
 * may be naturally thread safe. However, if sharing a single file
 * handle across all threads then your filetype will have to take care
 * of locking, etc.
 */
typedef struct SCEveFileType_ {
    /**
     * \brief The name of the output, used in the configuration.
     *
     * This name is used by the configuration file to specify the EVE
     * filetype used.
     *
     * For example:
     *
     * \code{.yaml}
     * outputs:
     *   - eve-log:
     *       filetype: my-output-name
     * \endcode
     */
    const char *name;

    /**
     * \brief Function to initialize this filetype.
     *
     * \param conf The ConfNode of the `eve-log` configuration
```

(continues on next page)

(continued from previous page)

```
     *     section this filetype is being initialized for
     *
     * \param threaded Flag to specify if the EVE sub-systems is in
     *     threaded mode or not
     *
     * \param init_data An output pointer for filetype specific data
     *
     * \retval 0 on success, -1 on failure
     */
    SCEveFileTypeInitFunc Init;

    /**
     * \brief Initialize thread specific data.
     *
     * Initialize any thread specific data. For example, if
     * implementing a file output you might open the files here, so
     * you have one output file per thread.
     *
     * \param init_data Data setup during Init
     *
     * \param thread_id A unique ID to differentiate this thread from
     *     others. If EVE is not in threaded mode this will be called
     *     once with a ThreadId of 0. In threaded mode the ThreadId of
     *     0 correlates to the main Suricata thread.
     *
     * \param thread_data Output pointer for any data required by this
     *     thread.
     *
     * \retval 0 on success, -1 on failure
     */
    SCEveFileTypeThreadInitFunc ThreadInit;

    /**
     * \brief Called for each EVE log record.
     *
     * The Write function is called for each log EVE log record. The
     * provided buffer contains a fully formatted EVE record in JSON
     * format.
     *
     * \param buffer The fully formatted JSON EVE log record
     *
     * \param buffer_len The length of the buffer
     *
     * \param init_data The data setup in the call to Init
     *
     * \param thread_data The data setup in the call to ThreadInit
     *
     * \retval 0 on success, -1 on failure
     */
    SCEveFileTypeWriteFunc Write;

    /**
```

(continues on next page)

(continued from previous page)

```
     * \brief Called to deinitialize each thread.
     *
     * This function will be called for each thread. It is where any
     * resources allocated in ThreadInit should be released.
     *
     * \param init_data The data setup in Init
     *
     * \param thread_data The data setup in ThreadInit
     */
    SCEveFileTypeThreadDeinitFunc ThreadDeinit;

    /**
     * \brief Final call to deinitialize this filetype.
     *
     * Called, usually on exit to deinitialize and free any resources
     * allocated during Init.
     *
     * \param init_data Data setup in the call to Init.
     */
    SCEveFileTypeDeinitFunc Deinit;

    /* Internal list management. */
    TAILQ_ENTRY(SCEveFileType_) entries;
} SCEveFileType;
```

Threading Considerations

It is the user's Suricata EVE output configuration that enables multi-threaded logging, not the filetype. So all filetypes should be designed to be thread safe.

If your filetype can absolutely not be made thread safe, it would be best to error out on initialization. This can be done during the filetype initialization:

```
static int MyFiletypeInit(const SCConfNode *node, const bool threaded, void **data)
{
    if (threaded) {
        FatalError("EVE filetype does not support threaded logging.");
    }

    /* Continue with initialization. */
}
```

Write Considerations

The `Write` callback is called in a packet processing thread so any blocking (other than writing to a file) should be avoided. If writing to a blocking resource it is recommended to copy the buffer into another thread for further processing to avoid packet loss.

Registration

Registering an EVE filetype requires registering the filetype early in the Suricata start-up or lifecycle, or if a plugin, in the plugin initialization function.

```
SCEveFileType *filetype = SCCalloc(1, sizeof(SCEveFileType));

filetype->name = "my-custom-filetype";
filetype->Init = FiletypeInit;
filetype->Deinit = FiletypeDeinit;
filetype->ThreadInit = FiletypeThreadInit;
filetype->ThreadDeinit = FiletypeThreadDeinit;
filetype->Write = FiletypeWrite;

if (!SCRegisterEveFileType(filetype)) {
    FatalError("Failed to register EVE filetype");
}
```

Then to use this filetype, set the `filetype` in your `suricata.yaml` `eve-log` configuration to the name of the filetype:

```
outputs:
  - eve-log:
      enabled: true
      filetype: my-custom-filetype
```

Examples

Suricata built-ins:

- `null`: see `output-eve-null.c` in the Suricata source code
- `syslog`: see `output-eve-syslog.c` in the Suricata source code

Plugin:

- The Suricata source code contains an example as a plugin, see: `examples/plugins/c-json-filetype`.

29.4.8 EVE Hooks

The EVE output provides a callback for additional data to be added to an EVE record before it is written.

It is important to note that it does not allow for modification of the EVE record due to the append only nature of Suricata's EVE output.

C API

Registration

In C, registering the callback is done with `SCEveRegisterCallback`.

```
/** \brief Register a callback for adding extra information to EVE logs.
 *
 * Allow users to register a callback for each EVE log. The callback
 * is called just before the root object on the SCJsonBuilder is to be
 * closed.
 *
 * New objects and fields can be appended, but existing entries cannot be modified.
```

(continues on next page)

(continued from previous page)

```
 *
 * Packet and Flow will be provided if available, but will otherwise be
 * NULL.
 *
 * Limitations: At this time the callbacks will only be called for EVE
 * loggers that use SCJsonBuilder, notably this means it won't be called
 * for stats records at this time.
 *
 * \returns true if callback is registered, false is not due to memory
 *     allocation error.
 */
bool SCEveRegisterCallback(SCEveUserCallbackFn fn, void *user);
```

Callback

The callback function is provided with an open `SCJsonBuilder` instance just before being closed out with a final `}`. Additional fields can be added with the `SCJsonBuilder` API.

```
/** \brief Function type for EVE callbacks.
 *
 * The function type for callbacks registered with
 * SCEveRegisterCallback. This function will be called with the
 * SCJsonBuilder just prior to the top-level object being closed. New
 * fields may be added, however, there is no way to alter existing
 * objects already added to the SCJsonBuilder.
 *
 * \param tv The ThreadVars for the thread performing the logging.
 * \param p Packet if available.
 * \param f Flow if available.
 * \param user User data provided during callback registration.
 */
typedef void (*SCEveUserCallbackFn)(
        ThreadVars *tv, const Packet *p, Flow *f, SCJsonBuilder *jb, void *user);
```

Example

For a real-life C example, see the `ndpi` plugin included in the Suricata source.

That example demonstrates:

- Registering an EVE callback during plugin initialization
- Using thread-local storage to maintain state
- Adding protocol-specific information to EVE records
- Properly checking for NULL pointers before accessing data

Rust API

In Rust, use `suricata_ffi::eve::register_callback`. This wraps the C API and lets the callback be registered as a Rust closure instead of a C function pointer plus `user` pointer.

The closure receives:

- `tv`: the `ThreadVars` for the thread performing the logging

- `p`: the `Packet`, if available
- `f`: the `Flow`, if available
- `jb`: a Rust `JsonBuilder` wrapper for the current EVE record

Unlike the C API, the Rust callback returns `Result<(), Error>`. If it returns `Err`, any JSON emitted by that callback is discarded.

```
use suricata_ffi::eve;

eve::register_callback(|_tv, _p, _f, jb| {
    jb.open_object("my_plugin")?;
    jb.set_string("key", "value")?;
    jb.close()?;
    Ok(())
}).expect("failed to register EVE callback");
```

The Rust callback is invoked at the same point, but it receives a `JsonBuilder` wrapper instead of a raw `SCJsonBuilder` pointer.

The raw pointers passed into the callback are only valid for the duration of the callback and must not be stored. The callback must also not panic.

This API is safe for library and plugins.

29.4.9 Threads

Rust API

The `suricata_ffi::thread` module provides Rust wrappers for thread lifecycle callbacks.

Thread Init Callback

Register a callback with `thread::register_init_callback` to run code for each Suricata thread as it is initialized. The callback receives a `ThreadVars` wrapper for the thread that has just been initialized.

The current Rust thread lifecycle API exposes an init callback only; there is no Rust thread deinit callback.

```
use suricata_ffi::thread::{self, ThreadVars};
use suricata_ffi::SCLogNotice;

fn on_thread_init(tv: &mut ThreadVars) {
    SCLogNotice!("thread initialized: {:p}", tv.as_ptr());
}

fn register_thread_callbacks() -> Result<(), &'static str> {
    thread::register_init_callback(on_thread_init)
}
```

The wrapper accepts function items or closures that implement `Fn(&mut ThreadVars) + Send + Sync + 'static` and returns `Result<(), &'static str>`. An error means the callback could not be registered. Registered callbacks are kept for the Suricata process lifetime.

`ThreadVars` carries a lifetime tied to the callback invocation, so the borrow checker prevents it from being stored beyond the call. Rust callbacks must not panic, as they are invoked across an FFI boundary.

Thread Storage

`thread::ThreadStorage<T>` provides typed, per-thread storage backed by Suricata's thread storage API. Each registered slot holds an independent value of type `T` for every thread.

Register a slot once during initialization with `ThreadStorage::<T>::register`. Registration must happen before Suricata finalizes its storage registration, which is the case during plugin initialization.

```
use suricata_ffi::thread::{self, ThreadStorage, ThreadVars};

#[derive(Default)]
struct ThreadState {
    flows: u64,
}

fn register(storage: ThreadStorage<ThreadState>) -> Result<(), &'static str> {
    thread::register_init_callback(move |tv| on_thread_init(storage, tv))
}
```

Values are owned by Suricata's thread storage and are dropped automatically when the thread's storage is freed.

Access the value for a thread through the `ThreadVars` wrapper. `get` takes `&ThreadVars` and returns `Option<&T>`. `get_mut` takes `&mut ThreadVars` and returns `Option<&mut T>`. `get_or_insert_with` also takes `&mut ThreadVars` and returns `Result<&mut T, _>`, inserting a value produced by the closure if the slot is empty:

```
fn on_thread_init(storage: ThreadStorage<ThreadState>, tv: &mut ThreadVars) {
    let _ = storage.get_or_insert_with(tv, ThreadState::default);
}

fn on_flow_init(storage: ThreadStorage<ThreadState>, tv: &mut ThreadVars) {
    if let Some(state) = storage.get_mut(tv) {
        state.flows += 1;
    }
}
```

29.4.10 Flow Life Cycle Callbacks

Flow lifecycle callbacks let plugins and library users observe when Suricata initializes a flow, updates a flow with a packet, and finishes with a flow.

These callbacks are useful for maintaining plugin state that follows the lifetime of a Suricata flow. For example, a plugin can allocate per-flow state from the init callback, update it as packets are seen, and perform final accounting from the finish callback.

C API

Flow Init Callback

The init callback is called when Suricata initializes a flow.

```
/** \brief Function type for flow initialization callbacks.
 *
 * Once registered with SCFlowRegisterInitCallback, this function will
 * be called every time a flow is initialized, or in other words,
 * every time Suricata picks up a flow.
 *
```

(continues on next page)

(continued from previous page)

```
 * \param tv The ThreadVars data structure for the thread creating the
 *     flow.
 * \param f The newly initialized flow.
 * \param p The packet related to creating the new flow.
 * \param user The user data provided during callback registration.
 */
typedef void (*SCFlowInitCallbackFn)(ThreadVars *tv, Flow *f, const Packet *p, void␣
↪*user);
```

Register an init callback with `SCFlowRegisterInitCallback`.

```
/** \brief Register a flow init callback.
 *
 * Register a user provided function to be called every time a flow is
 * initialized for use.
 *
 * \param fn Pointer to function to be called
 * \param user Additional user data to be passed to callback
 *
 * \returns true if callback was registered, otherwise false if the
 *     callback could not be registered due to memory allocation error.
 */
bool SCFlowRegisterInitCallback(SCFlowInitCallbackFn fn, void *user);
```

Flow Update Callback

The update callback is called when Suricata updates a flow with a packet.

```
/** \brief Function type for flow update callbacks.
 *
 * Once registered with SCFlowRegisterUpdateCallback, this function
 * will be called every time a flow is updated by a packet (basically
 * everytime a packet is seen on a flow).
 *
 * \param tv The ThreadVars data structure for the thread updating the
 *     flow.
 * \param f The flow being updated.
 * \param p The packet responsible for the flow update.
 * \param user The user data provided during callback registration.
 */
typedef void (*SCFlowUpdateCallbackFn)(ThreadVars *tv, Flow *f, Packet *p, void *user);
```

Register an update callback with `SCFlowRegisterUpdateCallback`.

```
/** \brief Register a flow update callback.
 *
 * Register a user provided function to be called everytime a flow is
 * updated.
 *
 * \param fn Pointer to function to be called
 * \param user Additional user data to be passed to callback
 *
```

(continues on next page)

(continued from previous page)

```
 * \returns true if callback was registered, otherwise false if the
 *     callback could not be registered due to memory allocation error.
 */
bool SCFlowRegisterUpdateCallback(SCFlowUpdateCallbackFn fn, void *user);
```

Flow Finish Callback

The finish callback is called when Suricata is done with a flow.

```
/** \brief Function type for flow finish callbacks.
 *
 * Once registered with SCFlowRegisterFinshCallback, this function
 * will be called when Suricata is done with a flow.
 *
 * \param tv The ThreadVars data structure for the thread finishing
 *     the flow.
 * \param f The flow being finshed.
 * \param user The user data provided during callback registration.
 */
typedef void (*SCFlowFinishCallbackFn)(ThreadVars *tv, Flow *f, void *user);
```

Register a finish callback with `SCFlowRegisterFinishCallback`.

```
bool SCFlowRegisterFinishCallback(SCFlowFinishCallbackFn fn, void *user);
```

Example

```
static void ExampleFlowInit(ThreadVars *tv, Flow *f, const Packet *p, void *user)
{
    SCLogNotice("flow initialized: %p", f);
}

static void ExampleFlowUpdate(ThreadVars *tv, Flow *f, Packet *p, void *user)
{
    SCLogNotice("flow updated: %p packet: %p", f, p);
}

static void ExampleFlowFinish(ThreadVars *tv, Flow *f, void *user)
{
    SCLogNotice("flow finished: %p", f);
}

static void ExampleInit(void)
{
    SCFlowRegisterInitCallback(ExampleFlowInit, NULL);
    SCFlowRegisterUpdateCallback(ExampleFlowUpdate, NULL);
    SCFlowRegisterFinishCallback(ExampleFlowFinish, NULL);
}
```

Rust API

In Rust, use the `suricata_ffi::flow` module:

- `flow::register_init_callback`
- `flow::register_update_callback`
- `flow::register_finish_callback`

The Rust wrappers register closures or function items and return `Result<(), &'static str>`.

```
use suricata_ffi::flow::{self, Flow};
use suricata_ffi::thread::ThreadVars;
use suricata_sys::sys::Packet;
use suricata_ffi::SCLogNotice;

fn flow_init(_tv: &mut ThreadVars, f: &mut Flow, _p: *const Packet) {
    SCLogNotice!("flow initialized: {:p}", f.as_ptr());
}

fn flow_update(_tv: &mut ThreadVars, f: &mut Flow, p: *mut Packet) {
    SCLogNotice!("flow updated: {:p} packet: {:p}", f.as_ptr(), p);
}

fn flow_finish(_tv: &mut ThreadVars, f: &mut Flow) {
    SCLogNotice!("flow finished: {:p}", f.as_ptr());
}

fn register_flow_callbacks() -> Result<(), &'static str> {
    flow::register_init_callback(flow_init)?;
    flow::register_update_callback(flow_update)?;
    flow::register_finish_callback(flow_finish)?;
    Ok(())
}
```

The raw pointers passed into callbacks are only valid for the duration of the callback invocation and must not be stored. Rust callbacks must not panic.

Flow Storage

`flow::FlowStorage<T>` provides typed, per-flow storage backed by Suricata's flow storage API. Each registered slot holds an independent value of type `T` for every flow.

Register a slot once during initialization with `FlowStorage::<T>::register`. Registration must happen before Suricata finalizes its storage registration, which is the case during plugin initialization.

```
use suricata_ffi::flow::{self, Flow, FlowStorage};
use suricata_ffi::thread::ThreadVars;
use suricata_sys::sys::Packet;

#[derive(Default)]
struct FlowState {
    packets: u64,
}

fn register(storage: FlowStorage<FlowState>) -> Result<(), &'static str> {
```

(continues on next page)

(continued from previous page)

```
    flow::register_update_callback(move |tv, f, p| on_flow_update(storage, tv, f, p))
}
```

Values are owned by Suricata's flow storage and are dropped automatically when the flow's storage is freed.

```
fn on_flow_init(storage: FlowStorage<FlowState>, _tv: &mut ThreadVars, f: &mut Flow, _p:␣
↪*const Packet) {
    let _ = storage.get_or_insert_with(f, FlowState::default);
}

fn on_flow_update(storage: FlowStorage<FlowState>, _tv: &mut ThreadVars, f: &mut Flow, _
↪p: *mut Packet) {
    if let Some(state) = storage.get_mut(f) {
        state.packets += 1;
    }
}
```

29.5 LibSuricata and Plugins

29.5.1 Using Suricata as a Library

The ability to turn Suricata into a library that can be utilized in other tools is currently a work in progress, tracked by Redmine Ticket #2693: https://redmine.openinfosecfoundation.org/issues/2693.

29.5.2 Plugins

A related work are Suricata plugins, also in progress and tracked by Redmine Ticket #4101: https://redmine.openinfosecfoundation.org/issues/4101.

Plugins can be used by modifying the suricata.yaml `plugins` section to include the path of the dynamic library to load.

Plugins should export a `SCPluginRegister` function that will be the entry point used by Suricata.

Application-layer plugins

Application layer plugins can be added as demonstrated by example https://github.com/OISF/suricata/blob/main/examples/plugins/altemplate/

The plugin code contains the same files as an application layer in the source tree:

- alname.rs : entry point of protocol with its registration
- detect.rs : signature keywords
- lib.rs : list the files in the rust module
- log.rs : logging to eve.json
- parser.rs : parsing functions

These files will have different `use` statements, targeting the suricata crate.

> **Attention**
>
> A plugin should not use rust structures from suricata crate if they are not repr(C), especially JsonBuilder.

This is because the rust compiler does not guarantee the structure layout unless you specify this representation. Thus, the plugin may expect the `JsonBuilder` fields at different offsets than they are supplied by Suricata at runtime. The solution is to go through the `JsonBuilder` C API which uses an opaque pointer.

And the plugin contains also additional files:

- plugin.rs : defines the entry point of the plugin -- `SCPluginRegister`

`SCPluginRegister` should register a callback that should then call `SCPluginRegisterAppLayer` passing a `SCAppLayerPlugin` structure to Suricata. It should also call `suricata::plugin::init();` to ensure the plugin has initialized its value of the Suricata Context. This is a structure needed by rust, to call some C functions, that cannot be found at compile time because of circular dependencies, and are therefore resolved at runtime.

The `SCPlugin` begins by a version number `SC_API_VERSION` for runtime compatibility between Suricata and the plugin.

Known limitations are:

- Plugins can only use simple logging as defined by `EveJsonSimpleTxLogFunc` without suricata.yaml configuration, see https://github.com/OISF/suricata/pull/11160
- Keywords cannot use validate callbacks, see https://redmine.openinfosecfoundation.org/issues/5634

> **Attention**
>
> A pure rust plugin needs to be compiled with `RUSTFLAGS=-Clink-args=-Wl,-undefined,dynamic_lookup`

This is because the plugin will link dynamically at runtime the functions defined in Suricata runtime. You can define this rust flag in a `.cargo/config.toml` file.

29.6 Upgrading

29.6.1 Upgrading 8.0 to 9.0

Alert Logging

Alert logging is done by iterating the *PacketAlert* entries in *Packet::alerts*. In 9.0 it is important to check the *PacketAlert::action* field for the *ACTION_ALERT* flag. If this flag is not set, no alert should be generated. This is to support the *pass*-rule usecase better.

29.6.2 Upgrading 7.0 to 8.0

EVE File Types

- The `ThreadInit` function will now be called when in *threaded* and *non-threaded* modes. This simplifies the initialization for EVE filetypes as they can use the same flow of execution for both modes. To upgrade, either remove the call to `ThreadInit` from `Init`, or move per-thread setup code from `Init` to `ThreadInit`.
- Many of the function arguments to the callbacks have been made `const` where it made sense.

Please see the latest example EVE filetype plugin for an up to date example.

CHAPTER

THIRTY

VERIFYING SURICATA SOURCE DISTRIBUTION FILES

Once the Suricata release distribution file has been downloaded, the PGP signature should be verified. This can be done using the GPG application and is usually available on Linux/BSD systems without having to manually install any additional packages. For Mac or Windows systems installation packages can be found at https://gnupg.org/.

30.1 Verification Steps

These verification steps are for general guidance, the exact process and commands may vary between operating systems.

30.1.1 Downloading the Signature File

The signature file needs to be downloaded as well as the distribution file. Both files can be found at https://suricata.io/download/.

30.1.2 Importing the OISF Signing Key

Once both the signature file and Suricata distribution files are obtained, the OISF signing key should be imported to the local gpg keyring. This can be done by running the following command:

```
$ gpg --receive-keys 2BA9C98CCDF1E93A
```

The above command should produce output similar to the following:

```
gpg: key 2BA9C98CCDF1E93A: public key "Open Information Security Foundation
(OISF) <releases@openinfosecfoundation.org>" imported
gpg: Total number processed: 1
gpg:               imported: 1
```

30.1.3 Verifying the Suricata Distribution File

To verify the contents of the Suricata distribution file the following command could be ran on the Suricata 8.0.0 distribution file. (Keep in mind that this should always be updated to the actual version and file names you are verifying.):

```
$ gpg --verify suricata-8.0.0.tar.gz.sig suricata-8.0.0.tar.gz
```

Depending on the trust level assigned to the OISF signing keys, something similar to the following output should be seen:

```
$ gpg --verify suricata-8.0.0.tar.gz.sig suricata-8.0.0.tar.gz
gpg: Signature made Tue 08 Jul 2025 11:12:04 AM UTC
gpg:                using RSA key B36FDAF2607E10E8FFA89E5E2BA9C98CCDF1E93A
```

(continues on next page)

(continued from previous page)

```
gpg: checking the trustdb
gpg: marginals needed: 3  completes needed: 1  trust model: pgp
gpg: depth: 0  valid:   1  signed:   0  trust: 0-, 0q, 0n, 0m, 0f, 1u
gpg: next trustdb check due at 2025-08-06
gpg: Good signature from "Open Information Security Foundation (OISF)
<releases@openinfosecfoundation.org>" [ultimate]
```

This indicates a valid signature and that the signing key is trusted.

Note

If output from the *--verify* command is similar to the following:

```
gpg: Signature made Tue 08 Jul 2025 11:12:04 AM UTC
gpg:                using RSA key B36FDAF2607E10E8FFA89E5E2BA9C98CCDF1E93A
gpg: Can't check signature: No public key
```

This indicates that the OISF signing key was not imported to the local GPG keyring.

Note

If output from the *--verify* command is similar to the following:

```
gpg: Signature made Tue 08 Jul 2025 11:12:04 AM UTC
gpg:                using RSA key B36FDAF2607E10E8FFA89E5E2BA9C98CCDF1E93A
gpg: Good signature from "Open Information Security Foundation (OISF)
<releases@openinfosecfoundation.org>" [unknown]
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
Primary key fingerprint: B36F DAF2 607E 10E8 FFA8  9E5E 2BA9 C98C CDF1 E93A
```

This indicates that the OISF signing key was imported and the signatures are valid, but either the keys have not been marked as trusted OR the keys are possibly a forgery.

If there are questions regarding the validity of the downloaded file, the OISF team can be reached at *security @ oisf.net* (remove the spaces between the @ before sending).

CHAPTER

THIRTYONE

APPENDIX

31.1 EVE JSON Schema

The Suricata source distribution contains a JSON schema for the EVE log files. This schema follows the JSON Schema specification and can be found in `etc/schema.json`. If your distribution does not contain this file, it can be viewed online at https://github.com/OISF/suricata/blob/main/etc/schema.json, but note that it is version-specific and may change between major versions of Suricata.

This schema attempts to log all possible fields that may be seen in Suricata's **EVE** output, including their datatype. It also includes extensions to help map log fields to related detection keywords.

31.1.1 Suricata Schema Extensions

We have extended JSON schema with a `suricata` object to add extra Suricata context such as detection keywords related to a log field, for example:

```
"rrname": {
    "type": "string",
    "suricata": {
        "keywords": [
            "dns.answers.rrname",
            "dns.response.rrname"
        ]
    }
}
```

The above shows that a field named `rrname` has 2 keywords that are related. Please refer to the keyword documentation to see precisely how they are used and related to the field being logged.

Extension Reference

The `suricata` extension object is valid on objects inside the `properties` object. The `suricata` object may accept the following fields:

keywords

Type: `array` or `boolean`

- **When an array:** Contains keyword names that are related to this JSON property. Each keyword in the array represents a detection rule keyword that can be used to match against the corresponding field value.
- **When ``false``:** Indicates that this JSON property has no applicable keyword. This is used for metadata fields that don't correspond to actual network data. For example, the `version` field inside a DNS object denotes the version of the log format and is unrelated to any aspect of a DNS message, therefore no keyword is applicable.

> **Note**
>
> As of Suricata 8.0, mapping log fields to detection keywords is a work in progress. Any field that does not have a `suricata.keywords` value still needs to be evaluated.

31.1.2 Schema Tooling

- Suricata-Verify: Our own tool for verifying every Suricata pull request, validates all EVE logs generated against the schema.
- `./scripts/eve-parity.py`: Found inside the Suricata source code when checked out with `git`, is a tool to provide information on how log fields map to keywords, or how keywords map to log entries.
- `./scripts/evedoc.py`: Generate documentation from the schema, such as the *EVE Index* included in this documentation.

31.2 EVE Index

31.2.1 Top Level (object)

Name	Type	Description
alert	object	
anomaly	object	
app_proto	string	Application layer protocol of the flow
app_proto_expected	string	In case of a protocol change to a specific protocol, and this specific protocol was not recognised, this field will have the value of the expected protocol
app_proto_orig	string	Original application layer protocol of the flow after a protocol change
app_proto_tc	string	Application layer protocol detected to client in case of mismatch
app_proto_ts	string	Application layer protocol detected to server in case of mismatch
arp	object	
bittorrent_dht	object	
capture_file	string	
community_id	string	
dcerpc	object	
dest_ip	string	
dest_port	integer	
dhcp	object	
direction	string	
dnp3	object	
dns	object	
drop	object	
email	object	
engine	object	
enip	object	
ether	object	
event_type	string	
fileinfo	object	

continues on next page

Table 1 – continued from previous page

Name	Type	Description
files	array of objects	
firewall	object	
flow	object	
flow_id	integer	
frame	object	
ftp	object	
ftp_data	object	
host	string	the sensor-name, if configured
http	object	
icmp_code	integer	
icmp_type	integer	
igmp	object	
ike	object	
in_iface	string	
ip_v	integer	IP version of the packet or flow
krb5	object	
ldap	object	
llmnr	object	LLMNR requests and responses
log_level	string	
mdns	object	mDNS requests and responses
metadata	object	
modbus	object	
mqtt	object	
ndpi	object	nDPI plugin, contents provided by 3rd party library
netflow	object	
nfs	object	
ntp	object	
packet	string	
packet_info	object	
parent_id	integer	
payload	string	
payload_length	integer	
payload_printable	string	
pcap_cnt	integer	
pcap_filename	string	
pgsql	object	
pkt_src	string	
pop3	object	
proto	string	
quic	object	
rdp	object	
response_icmp_code	integer	
response_icmp_type	integer	
rfb	object	
rgmp	object	
rpc	object	
sctp	object	SCTP protocol information
sip	object	
smb	object	

continues on next page

Table 1 – continued from previous page

Name	Type	Description
smtp	object	
snmp	object	
spi	integer	
src_ip	string	
src_port	integer	
ssh	object	
stats	object	
stream	integer	
stream_tcp	object	
sub_state	string	Transaction type or sub state.
suricata_version	string	
tc_progress	string	
tcp	object	
template	object	
tenant_id	integer	
tftp	object	
timestamp	string	
tls	object	
traffic	object	
ts_progress	string	
tunnel	object	
tunnel_id	integer	if any, the tunnel identifier defined in suricata.yaml decoder.tunnels section
tx_guessed	boolean	The signature that triggered this alert didn't tie to a transaction, so the transaction (and metadata) logged is a forced estimation and may not be the one you expect
tx_id	integer	
verdict	object	
vlan	array of numbers	
websocket	object	

31.2.2 websocket (object)

Name	Type	Description
fin	boolean	
mask	integer	
opcode	string	
payload_base64	string	
payload_printable	string	

31.2.3 verdict (object)

Name	Type	Description
action	string	
reject	array of strings	
reject_target	string	

31.2.4 tunnel (object)

Name	Type	Description
depth	integer	
dest_ip	string	
dest_port	integer	
pcap_cnt	integer	
pkt_src	string	
proto	string	
src_ip	string	
src_port	integer	
tunnel_id	integer	If any, the tunnel identifier defined in suricata.yaml decoder.tunnels section

31.2.5 traffic (object)

Name	Type	Description
id	array of strings	
label	array of strings	

31.2.6 tls (object)

Name	Type	Description
certificate	string	
chain	array of strings	
client	object	
client_alpns	array of strings	TLS client ALPN field(s)
client_handshake	object	
fingerprint	string	
from_proto	string	
issuerdn	string	
ja3	object	
ja3s	object	
ja4	string	
notafter	string	
notbefore	string	
serial	string	
server_alpns	array of strings	TLS server ALPN field(s)
server_handshake	object	
session_resumed	boolean	
sni	string	
subject	string	
subjectaltname	array of strings	TLS Subject Alternative Name field
version	string	

31.2.7 tls.server_handshake (object)

Name	Type	Description
cipher	integer	TLS server's chosen cipher
exts	array of integers	TLS server extension(s)
version	string	TLS version in server hello

31.2.8 tls.ja3s (object)

Name	Type	Description
hash	string	
string	string	

31.2.9 tls.ja3 (object)

Name	Type	Description
hash	string	
string	string	

31.2.10 tls.client_handshake (object)

Name	Type	Description
ciphers	array of integers	TLS client cipher(s)
exts	array of integers	TLS client extension(s)
sig_algs	array of integers	TLS client signature algorithm(s)
version	string	TLS version in client hello

31.2.11 tls.client (object)

Name	Type	Description
certificate	string	
chain	array of strings	
fingerprint	string	
issuerdn	string	
notafter	string	
notbefore	string	
serial	string	
subject	string	
subjectaltname	array of strings	TLS Subject Alternative Name field

31.2.12 tftp (object)

Name	Type	Description
file	string	
mode	string	
packet	string	

31.2.13 template (object)

Name	Type	Description
request	string	
response	string	

31.2.14 tcp (object)

Name	Type	Description
ack	boolean	
cwr	boolean	
ecn	boolean	
fin	boolean	
psh	boolean	
rst	boolean	
state	string	
syn	boolean	
tc_gap	boolean	
tc_max_regions	integer	
tc_urgent_oob_data	integer	Number of Out-of-Band bytes sent by server using TCP urgent packets
tcp_flags	string	
tcp_flags_tc	string	
tcp_flags_ts	string	
ts_gap	boolean	
ts_max_regions	integer	
ts_urgent_oob_data	integer	Number of Out-of-Band bytes sent by client using TCP urgent packets
urg	boolean	

31.2.15 stats (object)

Name	Type	Description
app_layer	object	Module with observational and performance-related statistics from application layer protocol parsers and flows
capture	object	Observational statistics for packet capture module
decoder	object	Statistics for packet decoding engine
defrag	object	Statistics on IP (de)fragmentation
detect	object	Statistics related to the detection engines
exception_policy	object	Statistics on exception policies hit and applied
file_store	object	Performance-related statistics for the file storing module
firewall	object	
flow	object	Stats on flow-related diagnostics
flow_bypassed	object	Observational statistics on flow bypassing
ftp	object	Performance statistics for global memory use and memory capacity for FTP app-layer parser
host	object	Performance statistics for global memory use and memory capacity for Host table
http	object	Performance statistics for global memory use and memory capacity for HTTP app-layer parser
ippair	object	Performance statistics for global memory use and memory capacity for IP Pair table
ips	object	Statistics for IPS mode
memcap	object	Performance statistics on global memory capacity / usage. Calculated for flow, stream, stream-reassembly, app-layer http, defrag, ippair and host
pcap_log	object	Statistics for pcap logging
sctp	object	Statistics on SCTP chunk types
stream	object	Observational statistics on TCP stream events
tcp	object	Statistics on TCP stream tracking and reassembly
transform	object	Statistics related to the transform functions
uptime	integer	Suricata engine's uptime

31.2.16 stats.transform (object)

Name	Type	Description
lua	object	

31.2.17 stats.transform.lua (object)

Name	Type	Description
blocked_function_errors	integer	Counter for Lua transform scripts failing due to blocked functions being called
errors	integer	Errors encountered while running Lua transform functions
instruction_limit_errors	integer	Count of Lua transform functions exceeding the instruction limit
memory_limit_errors	integer	Count of Lua transform functions exceeding the memory limit

31.2.18 stats.tcp (object)

Name	Type	Description
ack_unseen_data	integer	
active_sessions	integer	
insert_data_normal_fail	integer	
insert_data_overlap_fail	integer	
invalid_checksum	integer	
memuse	integer	
midstream_pickups	integer	
no_flow	integer	
overlap	integer	
overlap_diff_data	integer	
pkt_on_wrong_thread	integer	
pseudo	integer	
reassembly_gap	integer	
reassembly_memuse	integer	
rst	integer	
segment_from_cache	integer	
segment_from_pool	integer	
segment_memcap_drop	integer	
sessions	integer	
ssn_from_cache	integer	
ssn_from_pool	integer	
ssn_memcap_drop	integer	
stream_depth_reached	integer	
syn	integer	
synack	integer	
urg	integer	Number of TCP packets with the urgent flag set
urgent_oob_data	integer	Number of OOB bytes tracked in TCP urgent handling

31.2.19 stats.stream (object)

Name	Type	Description
3whs_ack_data_inject	integer	

continues on next page

Table 2 – continued from previous page

Name	Type	Description
3whs_ack_in_wrong_dir	integer	
3whs_async_wrong_seq	integer	
3whs_right_seq_wrong_ack_evas	integer	
3whs_syn_flood	integer	
3whs_syn_resend_diff_seq_on_sy	integer	
3whs_syn_toclient_on_syn_recv	integer	
3whs_synack_flood	integer	
3whs_synack_in_wrong_direction	integer	
3whs_synack_resend_with_diff_a	integer	
3whs_synack_resend_with_diff_s	integer	
3whs_synack_tfo_data_ignored	integer	
3whs_synack_toserver_on_syn_re	integer	
3whs_synack_with_wrong_ack	integer	
3whs_wrong_seq_wrong_ack	integer	
4whs_invalid_ack	integer	
4whs_synack_with_wrong_ack	integer	
4whs_synack_with_wrong_syn	integer	
4whs_wrong_seq	integer	
closewait_ack_out_of_window	integer	
closewait_fin_out_of_window	integer	
closewait_invalid_ack	integer	
closewait_pkt_before_last_ack	integer	
closing_ack_wrong_seq	integer	
closing_invalid_ack	integer	
est_ack_zwp_data	integer	
est_invalid_ack	integer	
est_packet_out_of_window	integer	
est_pkt_before_last_ack	integer	
est_syn_resend	integer	
est_syn_resend_diff_seq	integer	
est_syn_toclient	integer	
est_synack_resend	integer	
est_synack_resend_with_diff_ack	integer	
est_synack_resend_with_diff_seq	integer	
est_synack_toserver	integer	
fin1_ack_wrong_seq	integer	
fin1_fin_wrong_seq	integer	
fin1_invalid_ack	integer	
fin2_ack_wrong_seq	integer	
fin2_fin_wrong_seq	integer	
fin2_invalid_ack	integer	
fin_but_no_session	integer	
fin_invalid_ack	integer	
fin_out_of_window	integer	
fin_syn	integer	
lastack_ack_wrong_seq	integer	
lastack_invalid_ack	integer	
pkt_bad_window_update	integer	
pkt_broken_ack	integer	
pkt_invalid_ack	integer	

continues on next page

Table 2 – continued from previous page

Name	Type	Description
pkt_invalid_timestamp	integer	
pkt_retransmission	integer	
pkt_spurious_retransmission	integer	
reassembly_depth_reached	integer	
reassembly_insert_invalid	integer	
reassembly_insert_limit	integer	
reassembly_insert_memcap	integer	
reassembly_no_segment	integer	
reassembly_overlap_different_data	integer	
reassembly_segment_before_base_seq	integer	
reassembly_seq_gap	integer	
reassembly_urgent_oob_limit_reached	integer	
rst_but_no_session	integer	
rst_invalid_ack	integer	
rst_with_data	integer	
shutdown_syn_resend	integer	
suspected_rst_inject	integer	
timewait_ack_wrong_seq	integer	
timewait_invalid_ack	integer	
wrong_thread	integer	

31.2.20 stats.sctp (object)

Name	Type	Description
abort	integer	Number of SCTP packets with ABORT chunk
data	integer	Number of SCTP packets with DATA chunk
init	integer	Number of SCTP packets with INIT chunk
init_ack	integer	Number of SCTP packets with INIT_ACK chunk
shutdown	integer	Number of SCTP packets with SHUTDOWN chunk

31.2.21 stats.pcap_log (object)

Name	Type	Description
filtered_bpf	integer	Number of packets filtered out by bpf (not written)
written	integer	Number of packets written

31.2.22 stats.memcap (object)

Name	Type	Description
pressure	integer	Percentage of memcaps used by flow, stream, stream-reassembly and app-layer-http
pressure_max	integer	Maximum pressure seen by the engine

31.2.23 stats.ips (object)

Name	Type	Description
accepted	integer	Number of accepted packets
blocked	integer	Number of blocked packets
drop_reason	object	Number of dropped packets, grouped by drop reason
rejected	integer	Number of rejected packets
replaced	integer	Number of packets replaced by the stream engine or based on a match of the 'replaced' keyword.

31.2.24 stats.ips.drop_reason (object)

Name	Type	Description
applayer_error	integer	Number of packets dropped due to app-layer error exception policy
applayer_memcap	integer	Number of packets dropped due to applayer memcap
decode_error	integer	Number of packets dropped due to decoding errors
defrag_error	integer	Number of packets dropped due to defragmentation errors
defrag_memcap	integer	Number of packets dropped due to defrag memcap exception policy
exception_policy_flow_drop	integer	Number of packets dropped due to an exception policy flow dropping
flow_drop	integer	Number of packets dropped due to dropped flows
flow_memcap	integer	Number of packets dropped due to flow memcap exception policy
nfq_error	integer	Number of packets dropped due to no NFQ verdict
rules	integer	Number of packets dropped due to rule actions
stream_error	integer	Number of packets dropped due to invalid TCP stream
stream_memcap	integer	Number of packets dropped due to stream memcap exception policy
stream_midstream	integer	Number of packets dropped due to stream midstream exception policy
stream_reassembly	integer	Number of packets dropped due to stream reassembly exception policy
stream_urgent	integer	Number of packets dropped due to TCP urgent flag
threshold_detection_filter	integer	Number of packets dropped due to threshold detection filter
tunnel_packet_drop	integer	Number of packets dropped due to inner tunnel packet being dropped

31.2.25 stats.ippair (object)

Name	Type	Description
memcap	integer	Global memory capacity reached for IP Pair table
memuse	integer	Global memory usage for IP Pair table

31.2.26 stats.http (object)

Name	Type	Description
byterange	object	
memcap	integer	Global memory capacity reached for HTTP parser
memuse	integer	Global memory usage for HTTP parser

31.2.27 stats.http.byterange (object)

Name	Type	Description
memcap	integer	Global memory capacity reached for Byte Range containers
memuse	integer	Global memory usage for Byte Range containers

31.2.28 stats.host (object)

Name	Type	Description
memcap	integer	Global memory capacity reached for Host table
memuse	integer	Global memory usage for Host table

31.2.29 stats.ftp (object)

Name	Type	Description
memcap	integer	Global memory capacity reached for FTP parser
memuse	integer	Global memory usage for FTP parser

31.2.30 stats.flow_bypassed (object)

Name	Type	Description
bytes	integer	
closed	integer	
local_bytes	integer	
local_capture_bytes	integer	
local_capture_pkts	integer	
local_pkts	integer	
pkts	integer	

31.2.31 stats.flow (object)

Name	Type	Description
active	integer	Number of currently active flows
elephant	integer	Total number of elephant flows
elephant_toclient	integer	Total number of elephant flows in toclient direction
elephant_toserver	integer	Total number of elephant flows in toserver direction
emerg_mode_entered	integer	Number of times emergency mode was entered
emerg_mode_over	integer	Number of times recovery was made from emergency mode
end	object	
get_used	integer	Number of reused flows from the hash table in case memcap was reached and spare pool was empty
get_used_eval	integer	Number of attempts at getting a flow directly from the hash
get_used_eval_busy	integer	Number of times a flow was found in the hash but the lock for hash bucket could not be obtained
get_used_eval_reject	integer	Number of flows that were evaluated but rejected from reuse as they were still alive/active
get_used_failed	integer	Number of times retrieval of flow from hash was attempted but was unsuccessful
icmpv4	integer	Number of ICMPv4 flows
icmpv6	integer	Number of ICMPv6 flows
memcap	integer	Number of times memcap was reached for flows
memuse	integer	Memory currently in use by the flows
mgr	object	Flow manager stats counters
recycler	object	
spare	integer	Number of flows in the spare pool
tcp	integer	Number of TCP flows
tcp_reuse	integer	Number of TCP flows that were reused as they seemed to share the same flow tuple
total	integer	Total number of flows
udp	integer	Number of UDP flows
wrk	object	Flow worker threads stats

31.2.32 stats.flow.wrk (object)

Name	Type	Description
flows_evicted	integer	Number of flows that were evicted
flows_evicted_needs_work	integer	Number of TCP flows that were returned to the workers in case reassembly, detection, logging still needs work
flows_evicted_pkt_inject	integer	Number of pseudo packets injected into worker threads to complete flows' processing. For any flow this can be between 0-2, this is the total for all flows.
flows_injected	integer	Number of flows injected into the worker thread from another thread
flows_injected_max	integer	Maximum number of flows injected into the worker thread from another thread
spare_sync	integer	Number of times the engine attempted to fetch flows from the master flow pool/spare queue
spare_sync_avg	integer	Average number of flows a thread could fetch from the master flow pool/spare queue
spare_sync_empty	integer	Number of times the master spare pool was empty when requesting flows from it
spare_sync_incomplete	integer	Number of times spare flow syncs were incomplete (fetched with less than 100 flows in sync)

31.2.33 stats.flow.recycler (object)

Name	Type	Description
queue_avg	integer	Average number of recycled flows per queue
queue_max	integer	Maximum number of recycled flows per queue
recycled	integer	Number of recycled flows

31.2.34 stats.flow.mgr (object)

Name	Type	Description
flows_checked	integer	Number of flows checked for timeout in the last pass
flows_evicted	integer	Number of flows that were evicted
flows_evicted_needs_work	integer	Number of TCP flows that were returned to the workers in case reassembly, detection, logging still needs work
flows_notimeout	integer	Number of flows that did not time out
flows_timeout	integer	Number of flows that reached the time out
full_hash_pass	integer	Number of times a full pass of the hash table was done
rows_maxlen	integer	Size of the biggest row in the hash table
rows_per_sec	integer	Number of rows to be scanned every second by a worker

31.2.35 stats.flow.end (object)

Name	Type	Description
state	object	
tcp_liberal	integer	Number of TCP flows ended that had liberal state
tcp_state	object	

31.2.36 stats.flow.end.tcp_state (object)

Name	Type	Description
close_wait	integer	Number of TCP sessions in CLOSE_WAIT state
closed	integer	Number of TCP sessions in CLOSED state
closing	integer	Number of TCP sessions in CLOSING state
established	integer	Number of TCP sessions in ESTABLISHED state
fin_wait1	integer	Number of TCP sessions in FIN_WAIT_1 state
fin_wait2	integer	Number of TCP sessions in FIN_WAIT_2 state
last_ack	integer	Number of TCP sessions in LAST_ACK state
none	integer	Number of TCP sessions newly created
syn_recv	integer	Number of TCP sessions in SYN_RECV state
syn_sent	integer	Number of TCP sessions in SYN_SENT state
time_wait	integer	Number of TCP sessions in TIME_WAIT state

31.2.37 stats.flow.end.state (object)

Name	Type	Description
capture_bypassed	integer	Number of flows bypassed at the capture level -- counted at the time of flow end
closed	integer	Number of flows in 'closed' state at the time of flow end
established	integer	Number of flows in 'established' state at the time of flow end
local_bypassed	integer	Number of flows bypassed internally -- counted at the time of flow end
new	integer	Number of flows in 'new' state at the time of flow end

31.2.38 stats.firewall (object)

Name	Type	Description
accepted	integer	Count of accepted packets due to firewall policies
blocked	integer	Count of blocked packets due to firewall policies
discarded_alerts	integer	Count of alerts discarded due max alerts reached
drop_reason	object	
rejected	integer	Count of packets rejected due to firewall policies

31.2.39 stats.firewall.drop_reason (object)

Name	Type	Description
default_app_policy	integer	Count of packets dropped due to firewall's mode default app policy
default_packet_policy	integer	Count of packets dropped due to firewall's mode default packet policy
flow_drop	integer	Count of packets dropped due to a firewall policy that led to flow drop
pre_flow_hook	integer	Count of packets dropped due to pre-flow hook
pre_stream_hook	integer	Count of packets dropped due to pre-stream hook
rules	integer	Count of packets dropped due to firewall rules

31.2.40 stats.file_store (object)

Name	Type	Description
fs_errors	integer	
open_files	integer	
open_files_max_hit	integer	

31.2.41 stats.exception_policy (object)

Name	Type	Description
app_layer	object	
defrag	object	
flow	object	
tcp	object	

31.2.42 stats.exception_policy.tcp (object)

Name	Type	Description
midstream	object	
reassembly	object	
ssn_memcap	object	

31.2.43 stats.exception_policy.tcp.ssn_memcap (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.44 stats.exception_policy.tcp.reassembly (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.45 stats.exception_policy.tcp.midstream (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.46 stats.exception_policy.flow (object)

Name	Type	Description
memcap	object	

31.2.47 stats.exception_policy.flow.memcap (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.48 stats.exception_policy.defrag (object)

Name	Type	Description
memcap	object	

31.2.49 stats.exception_policy.defrag.memcap (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.50 stats.exception_policy.app_layer (object)

Name	Type	Description
error	object	

31.2.51 stats.exception_policy.app_layer.error (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.52 stats.detect (object)

Name	Type	Description
alert	integer	Count of alerts triggered
alert_queue_overflow	integer	Count of alerts discarded due to alert queue overflow
alerts_suppressed	integer	Count of alerts not logged due to noalert keyword usage or thresholding
engines	array of objects	
lua	object	
match_list	integer	If profiling is enabled, average count of signature matched against a packet
mpm_list	integer	If profiling is enabled, average count of signatures in the mpm prefilter list
thresholds	object	

31.2.53 stats.detect.thresholds (object)

Name	Type	Description
bitmap_alloc_fail	integer	Count of bitmap allocation failures
bitmap_memuse	integer	Memory usage by detection_filter bitmaps
cache	object	
memcap	integer	Memory cap for threshold hash table
memuse	integer	Memory usage by threshold hash table

31.2.54 stats.detect.thresholds.cache (object)

Name	Type	Description
memuse	integer	Memory usage by the per-thread threshold decision cache

31.2.55 stats.detect.lua (object)

Name	Type	Description
blocked_function_errors	integer	Counter for Lua scripts failing due to blocked functions being called
errors	integer	Errors encountered while running Lua scripts
instruction_limit_errors	integer	Count of Lua rules exceeding the instruction limit
memory_limit_errors	integer	Count of Lua rules exceeding the memory limit

31.2.56 stats.detect.engines (array of objects)

Name	Type	Description
id	integer	If multi-tenancy is enabled, the tenant id
last_reload	string	Last time the rules were reloaded, in TimeString format
rules_failed	integer	Count of rules that failed to load
rules_loaded	integer	Count of rules successfully loaded
rules_skipped	integer	Count of rules that were skipped due to missing requirements

31.2.57 stats.defrag (object)

Name	Type	Description
ipv4	object	
ipv6	object	
max_frags_reached	integer	How many times a fragment wasn't stored due to max-frags limit being reached
max_trackers_reached	integer	How many times a packet wasn't reassembled due to max-trackers limit being reached
memuse	integer	Current memory use.
mgr	object	
tracker_hard_reuse	integer	Active tracker force closed before completion and reused for new tracker
tracker_soft_reuse	integer	Finished tracker re-used from hash table before being moved to spare pool
wrk	object	

31.2.58 stats.defrag.wrk (object)

Name	Type	Description
tracker_timeout	integer	

31.2.59 stats.defrag.mgr (object)

Name	Type	Description
tracker_timeout	integer	

31.2.60 stats.defrag.ipv6 (object)

Name	Type	Description
fragments	integer	
reassembled	integer	
timeouts	integer	

31.2.61 stats.defrag.ipv4 (object)

Name	Type	Description
fragments	integer	
reassembled	integer	
timeouts	integer	

31.2.62 stats.decoder (object)

Name	Type	Description
arp	integer	Number of ARP packets decoded
avg_pkt_size	integer	Average packet size decoded
bytes	integer	Number of bytes decoded by the engine
chdlc	integer	Number of Cisco HDLC packets decoded
erspan	integer	Number of ERSPAN packets decoded
esp	integer	Number of ESP packets decoded
etag	integer	Number of ETAG packets decoded
ethernet	integer	Number of Ethernet packets decoded
event	object	Statistics on events raised during packet decoding
geneve	integer	Number of GENEVE packets decoded
gre	integer	Number of GRE packets decoded
icmpv4	integer	Number of ICMPv4 packets decoded
icmpv6	integer	Number of ICMPv6 packets decoded
ieee8021ah	integer	Number of IEEE802.1ah packets decoded
igmp	integer	Number of IGMP packets decoded
invalid	integer	Number of invalid packets decoded
ipv4	integer	Number of IPv4 packets decoded
ipv4_in_ipv4	integer	Number of IPv4 in IPv4 packets decoded
ipv4_in_ipv6	integer	Number of IPv4 in IPv6 packets decoded
ipv6	integer	Number of IPv6 packets decoded
ipv6_in_ipv4	integer	Number of IPv6 in IPv4 packets decoded
ipv6_in_ipv6	integer	Number of IPv6 in IPv6 packets decoded
max_mac_addrs_dst	integer	Maximum amount of destination MAC addresses seen per flow (only if ethernet header logging enabled)
max_mac_addrs_src	integer	Maximum amount of source MAC addresses seen per flow (only if ethernet header logging enabled)
max_pkt_size	integer	Maximum packet size decoded by the engine
mpls	integer	Number of MPLS packets decoded
nsh	integer	Number of NSH packets decoded
null	integer	Number of LINKTYPE_NULL packets decoded
pkts	integer	Number of packets decoded
ppp	integer	Number of PPP packets decoded
pppoe	integer	Number of PPPOE packets decoded
raw	integer	Number of RAW packets decoded
sctp	integer	Number of STCP packets decoded
sll	integer	Number of SLL packets decoded
sll2	integer	The number of SLL2 frames encountered
tcp	integer	Number of TCP packets decoded
teredo	integer	Number of Teredo packets decoded
too_many_layers	integer	Number of decoded packets that reach maximum layers for the engine
udp	integer	Number of UDP packets decoded
unknown_ethertype	integer	Number of decoded packets with unknown ethertype
vlan	integer	Number of VLAN layer 2 packets decoded
vlan_qinq	integer	Number of VLAN layer 2 (Q-in-Q) packets decoded

continues on next page

Table 3 – continued from previous page

Name	Type	Description
vlan_qinqinq	integer	Number of VLAN layer 3 (Q-in-Q-in-Q) packets decoded
vntag	integer	Number of VNTAG packets decoded
vxlan	integer	Number of VXLAN packets decoded

31.2.63 stats.decoder.event (object)

Name	Type	Description
afpacket	object	
arp	object	
chdlc	object	
dce	object	
erspan	object	
esp	object	
etag	object	
ethernet	object	
geneve	object	
gre	object	
icmpv4	object	
icmpv6	object	
ieee8021ah	object	
igmp	object	
ipraw	object	
ipv4	object	
ipv6	object	
ltnull	object	
mpls	object	
nsh	object	
ppp	object	
pppoe	object	
sctp	object	
sll	object	
sll2	object	
tcp	object	
udp	object	
vlan	object	
vntag	object	
vxlan	object	

31.2.64 stats.decoder.event.vxlan (object)

Name	Type	Description
unknown_payload_type	integer	

31.2.65 stats.decoder.event.vntag (object)

Name	Type	Description
header_too_small	integer	
unknown_type	integer	

31.2.66 stats.decoder.event.vlan (object)

Name	Type	Description
header_too_small	integer	
too_many_layers	integer	
unknown_type	integer	

31.2.67 stats.decoder.event.udp (object)

Name	Type	Description
hlen_invalid	integer	
hlen_too_small	integer	
len_invalid	integer	
pkt_too_small	integer	

31.2.68 stats.decoder.event.tcp (object)

Name	Type	Description
hlen_too_small	integer	
invalid_optlen	integer	
opt_duplicate	integer	
opt_invalid_len	integer	
opt_wscale_over_max	integer	TCP option window scale had a value exceeding its maximum (14) and was clamped to it
pkt_too_small	integer	

31.2.69 stats.decoder.event.sll2 (object)

Name	Type	Description
pkt_too_small	integer	The number of times the SLL2 header was too small to be valid

31.2.70 stats.decoder.event.sll (object)

Name	Type	Description
pkt_too_small	integer	Number of SLL decoded packets that were too small

31.2.71 stats.decoder.event.sctp (object)

Name	Type	Description
chunk_len_invalid	integer	SCTP chunk length < 4 or exceeds remaining packet
chunk_too_small	integer	Remaining data too small for SCTP chunk header
data_with_zero_vtag	integer	SCTP DATA chunk with verification tag == 0
init_chunk_bundled	integer	RFC 4960 sec 6.10 violation: INIT/INIT_ACK bundled with other chunks
init_with_non_zero_vtag	integer	SCTP INIT with verification tag != 0
pkt_too_small	integer	SCTP packet smaller than minimum size
too_many_chunks	integer	More chunks than SCTP_MAX_TRACKED_CHUNKS
too_many_data_chunks	integer	More DATA chunks than SCTP_MAX_DATA_CHUNKS

31.2.72 stats.decoder.event.pppoe (object)

Name	Type	Description
malformed_tags	integer	
pkt_too_small	integer	
wrong_code	integer	

31.2.73 stats.decoder.event.ppp (object)

Name	Type	Description
ip4_pkt_too_small	integer	
ip6_pkt_too_small	integer	
pkt_too_small	integer	
unsup_proto	integer	
vju_pkt_too_small	integer	
wrong_type	integer	

31.2.74 stats.decoder.event.nsh (object)

Name	Type	Description
bad_header_length	integer	
header_too_small	integer	
reserved_type	integer	
unknown_payload	integer	
unsupported_type	integer	
unsupported_version	integer	

31.2.75 stats.decoder.event.mpls (object)

Name	Type	Description
bad_label_implicit_null	integer	
bad_label_reserved	integer	
bad_label_router_alert	integer	
header_too_small	integer	
pkt_too_small	integer	
unknown_payload_type	integer	

31.2.76 stats.decoder.event.ltnull (object)

Name	Type	Description
pkt_too_small	integer	
unsupported_type	integer	

31.2.77 stats.decoder.event.ipv6 (object)

Name	Type	Description
data_after_none_header	integer	Number of IPv6 packets with data after the 'none' header
dstopts_only_padding	integer	Number of IPv6 packets with all DST options as only padding
dstopts_unknown_opt	integer	Number of IPv6 packets with unknown DST option
exthdr_ah_res_not_null	integer	Number of IPv6 packets with AH header reserved fields not null
exthdr_dupl_ah	integer	Number of IPv6 packets with duplicated 'authentication' header in IPv6 extension headers
exthdr_dupl_dh	integer	Number of IPv6 packets with duplicated 'destination' header in IPv6 extension headers
exthdr_dupl_eh	integer	Number of IPv6 packets with duplicated 'ESP' header in IPv6 extension headers
exthdr_dupl_fh	integer	Number of IPv6 packets with duplicated 'fragment' header in IPv6 extension headers
exthdr_dupl_hh	integer	Number of IPv6 packets with duplicated 'hop-by-hop' header in IPv6 extension headers
exthdr_dupl_rh	integer	Number of IPv6 packets with duplicated'routing' header in IPv6 extension headers
exthdr_invalid_optlen	integer	Number of IPv6 packets flagged invalid due to invalid option length in a hop or dst extended header
exthdr_useless_fh	integer	Number of IPv6 packets with useless 'fragment header' in the extended headers
fh_non_zero_reserved_field	integer	Number of IPv6 packets with 'fragment header' with non-zero reserved field
frag_ignored	integer	Number of IPv6 fragments ignored due to resource allocation errors
frag_invalid_length	integer	Number of IPv6 fragments with invalid length

continues on next page

Table 5 – continued from previous page

Name	Type	Description
frag_overlap	integer	Number of IPv6 fragments with overlapping data
frag_pkt_too_large	integer	Number of IPv6 fragments ignored due to being too large
hopopts_only_padding	integer	Number of IPv6 packets with all HOP options as only padding
hopopts_unknown_opt	integer	Number of IPv6 packets with unknown HOP option
icmpv4	integer	Number of IPv6 packets with ICMPv4 header
ipv4_in_ipv6_too_small	integer	Number of IPv4-in-IPv6 packets flagged invalid due to being too small
ipv4_in_ipv6_wrong_version	integer	Number of IPv4-in-IPv6 packets with wrong IP version
ipv6_in_ipv6_too_small	integer	Number of IPv6-in-IPv6 packets flagged invalid due to being too small
ipv6_in_ipv6_wrong_version	integer	Number of IPv6-in-IPv6 packets with wrong IP version
pkt_too_small	integer	Number of IPv6 packets flagged invalid due to size smaller than minimum header size
rh_type_0	integer	Number of IPv6 packets with extended header 'routing' with type 0
trunc_exthdr	integer	Number of IPv6 packets flagged invalid due to truncated extension header
trunc_pkt	integer	Number of IPv6 packets flagged invalid due to truncated packet
unknown_next_header	integer	Number of IPv6 packets with unknown next header
wrong_ip_version	integer	Number of IPv6 packets flagged invalid due to wrong IP version
zero_len_padn	integer	Number of IPv6 packets with PadN option without data (length zero)

31.2.78 stats.decoder.event.ipv4 (object)

Name	Type	Description
frag_ignored	integer	Number of IPv6 fragments ignored due to resource allocation errors
frag_overlap	integer	Number of IPv4 fragments with overlapping data
frag_pkt_too_large	integer	Number of IPv4 fragments ignored due to being too large
hlen_too_small	integer	Number of IPv4 packets flagged invalid due to header smaller than minimum size
icmpv6	integer	Number of IPv4 packets flagged invalid due to having an ICMPV6 header
iplen_smaller_than_hlen	integer	Number of IPv4 packets flagged invalid due to length being smaller than IP header size
opt_duplicate	integer	Number of IPv4 packets with duplicated IP options
opt_eol_required	integer	Number of IPv4 packets with 'end of list' option not present, but required, in IP options
opt_invalid	integer	Number of IPv4 packets with invalid IP options
opt_invalid_len	integer	Number of IPv4 packets flagged invalid due to IP options with invalid length
opt_malformed	integer	Number of IPv4 packets flagged invalid due to malformed IP options
opt_pad_required	integer	Number of IPv4 packets with padding bytes required in IP options
opt_unknown	integer	Number of IPv4 packets flagged invalid due to unknown IP option
pkt_too_small	integer	Number of IPv4 packets flagged invalid due to size smaller than minimum header size
trunc_pkt	integer	Number of IPv4 packets flagged invalid due to truncated packet
unknown_protocol	integer	Number of IPv4 packets with unknown protocol
wrong_ip_version	integer	Number of IPv4 packets flagged invalid due to having wrong IP version in IP options

31.2.79 stats.decoder.event.ipraw (object)

Name	Type	Description
invalid_ip_version	integer	Number of RAW packets with invalid IP version

31.2.80 stats.decoder.event.igmp (object)

Name	Type	Description
malformed	integer	IGMP with malformed data
pkt_too_small	integer	IGMP packets too small to fit a IGMP header
v3_pkt_too_small	integer	IGMPv3 packets too small to fit a IGMP header

31.2.81 stats.decoder.event.ieee8021ah (object)

Name	Type	Description
header_too_small	integer	Number of IEEE802.1ah packets with header too small

31.2.82 stats.decoder.event.icmpv6 (object)

Name	Type	Description
experimentation_type	integer	Number of ICMPv6 packets with private experimentation type
ipv6_trunc_pkt	integer	Number of truncated ICMPv6 packets
ipv6_unknown_version	integer	Number of ICMPv6 packets with unknown version
mld_message_with_invalid_hl	integer	Number of ICMPv6 packets with MLD messages and invalid HL (not 1)
pkt_too_small	integer	Number of packets too small for ICMPv6
unassigned_type	integer	Number of ICMPv6 packets with unassigned type
unknown_code	integer	Number of ICMPv6 packets with unknown code
unknown_type	integer	Number of ICMPv6 packets with unknown type

31.2.83 stats.decoder.event.icmpv4 (object)

Name	Type	Description
ipv4_trunc_pkt	integer	Number of truncated packets for ICMPv4
ipv4_unknown_ver	integer	Number of ICMPv4 packets with unknown version
pkt_too_small	integer	Number of packets too small for ICMPv4
unknown_code	integer	Number of ICMPv4 packets with unknown code
unknown_type	integer	Number of ICMPv4 packets with unknown type

31.2.84 stats.decoder.event.gre (object)

Name	Type	Description
pkt_too_small	integer	Number of packets too small for GRE
version0_flags	integer	Number of packets with version 0 flags set for GRE
version0_hdr_too_big	integer	Number of packets with version 0 and header too big for GRE
version0_malformed_sre_hdr	integer	Number of packets of with version 0 and malformed SRE header for GRE
version0_recur	integer	Number of packets with version 0 and flag recursion control set for GRE
version1_chksum	integer	Number of packets with version 1 and checksum flag set for GRE
version1_flags	integer	Number of packets with version 1 flags set for GRE
version1_hdr_too_big	integer	Number of packets with version 1 and header too big for GRE
version1_malformed_sre_hdr	integer	Number of packets with version 1 and malformed SRE header for GRE
version1_no_key	integer	Number of packets with version 1 and no key flag set for GRE
version1_recur	integer	Number of packets with version 1 and flag recursion control set for GRE
version1_route	integer	Number of packets with version 1 and flag route set for GRE
version1_ssr	integer	Number of packets with version 1 and flag SSR set for GRE
version1_wrong_protocol	integer	Number of packets with version 1 and wrong protocol set for GRE
wrong_version	integer	Number of packets with wrong version set for GRE

31.2.85 stats.decoder.event.geneve (object)

Name	Type	Description
unknown_payload_type	integer	Number of packets with unknown payload type for Geneve

31.2.86 stats.decoder.event.ethernet (object)

Name	Type	Description
pkt_too_small	integer	Number of packets too small for Ethernet
unknown_ethertype	integer	Number of packets with Unkonwn Ethertype for Ethernet

31.2.87 stats.decoder.event.etag (object)

Name	Type	Description
header_too_small	integer	Number of packets with header too small for ETAG
unknown_type	integer	Number of ETAG packets with unknown type

31.2.88 stats.decoder.event.esp (object)

Name	Type	Description
pkt_too_small	integer	Number of packets too small for ESP

31.2.89 stats.decoder.event.erspan (object)

Name	Type	Description
header_too_small	integer	Number of packets with header too small for ERSPAN
too_many_vlan_layers	integer	Number of packets with too many VLAN layers for ERSPAN
unsupported_version	integer	Number of packets with unsupported version for ERSPAN

31.2.90 stats.decoder.event.dce (object)

Name	Type	Description
pkt_too_small	integer	Number of packets too small for DCE

31.2.91 stats.decoder.event.chdlc (object)

Name	Type	Description
pkt_too_small	integer	Number of packets too small for CHDLC

31.2.92 stats.decoder.event.arp (object)

Name	Type	Description
invalid_hardware_size	integer	Number of ARP packets with invalid hardware size (valid size is 6)
invalid_pkt	integer	Number of invalid decoded ARP packets
invalid_protocol_size	integer	Number of ARP packets with invalid protocol size (valid size is 4)
pkt_too_small	integer	Number of ARP packets with header length too small
unsupported_hardware	integer	Number of ARP packets with unsupported hardware
unsupported_opcode	integer	Number of ARP packets with unsupported Operation Codes
unsupported_protocol	integer	Number of ARP packets with unsupported protocol

31.2.93 stats.decoder.event.afpacket (object)

Name	Type	Description
trunc_pkt	integer	Number of packets truncated by AF_PACKET

31.2.94 stats.capture (object)

Name	Type	Description
afpacket	object	Statistics for AF_PACKET capture module
errors	integer	Number of Suricata errors reported while reading capture module
kernel_drops	integer	Number of packets dropped by the kernel
kernel_ifdrops	integer	Number of packets dropped by the interface
kernel_packets	integer	Number of packets received from the kernel

31.2.95 stats.capture.afpacket (object)

Name	Type	Description
busy_loop_avg	integer	
poll_data	integer	
poll_errors	integer	
poll_signal	integer	
poll_timeout	integer	
polls	integer	
send_errors	integer	

31.2.96 stats.app_layer (object)

Name	Type	Description
error	object	
expectations	integer	Expectation (dynamic parallel flow) counter
flow	object	
tx	object	

31.2.97 stats.app_layer.tx (object)

Name	Type	Description
bittorrent_dht	integer	Number of transactions for BitTorrent DHT protocol
dcerpc_tcp	integer	Number of transactions for DCERPC/TCP protocol
dcerpc_udp	integer	Number of transactions for DCERPC/UDP protocol
dhcp	integer	Number of transactions for DHCP
dnp3	integer	Number of transactions for DNP3
dns_tcp	integer	Number of transactions for DNS/TCP protocol
dns_udp	integer	Number of transactions for DNS/UDP protocol
doh2	integer	
enip_tcp	integer	Number of transactions for ENIP/TCP
enip_udp	integer	Number of transactions for ENIP/UDP
ftp	integer	Number of transactions for FTP
ftp_data	integer	Number of transactions for FTP data protocol
http	integer	Number of transactions for HTTP
http2	integer	Number of transactions for HTTP/2
ike	integer	Number of transactions for IKE protocol
ikev2	integer	Number of transactions for IKE v2 protocol
imap	integer	Number of transactions for IMAP
krb5_tcp	integer	Number of transactions for Kerberos v5/TCP protocol
krb5_udp	integer	Number of transactions for Kerberos v5/UDP protocol
ldap_tcp	integer	Number of transactions for LDAP/TCP protocol
ldap_udp	integer	Number of transactions for LDAP/UDP protocol
llmnr_tcp	integer	Number of transactions for LLMNR/TCP protocol
llmnr_udp	integer	Number of transactions for LLMNR/UDP protocol
mdns	integer	Number of transactions for mDNS
modbus	integer	Number of transactions for Modbus protocol
mqtt	integer	Number of transactions for MQTT protocol
nfs_tcp	integer	Number of transactions for NFS/TCP protocol
nfs_udp	integer	Number of transactions for NFS/UDP protocol
ntp	integer	Number of transactions for NTP
pgsql	integer	Number of transactions for PostgreSQL protocol
pop3	integer	
quic	integer	Number of transactions for QUIC protocol
rdp	integer	Number of transactions for RDP
rfb	integer	Number of transactions for RFB protocol

continues on next page

Table 6 – continued from previous page

Name	Type	Description
sip_tcp	integer	Number of transactions for SIP/TCP protocol
sip_udp	integer	Number of transactions for SIP/UDP protocol
smb	integer	Number of transactions for SMB protocol
smtp	integer	Number of transactions for SMTP
snmp	integer	Number of transactions for SNMP
ssh	integer	Number of transactions for SSH protocol
telnet	integer	Number of transactions for Telnet protocol
tftp	integer	Number of transactions for TFTP
tls	integer	Number of transactions for TLS protocol
websocket	integer	

31.2.98 stats.app_layer.flow (object)

Name	Type	Description
bittorrent_dht	integer	Number of flows for BitTorrent DHT protocol
dcerpc_tcp	integer	Number of flows for DCERPC/TCP protocol
dcerpc_udp	integer	Number of flows for DCERPC/UDP protocol
dhcp	integer	Number of flows for DHCP
dnp3	integer	Number of flows for DNP3
dns_tcp	integer	Number of flows for DNS/TCP protocol
dns_udp	integer	Number of flows for DNS/UDP protocol
doh2	integer	
enip_tcp	integer	Number of flows for ENIP/TCP
enip_udp	integer	Number of flows for ENIP/UDP
failed_tcp	integer	Number of failed flows for TCP
failed_udp	integer	Number of failed flows for UDP
ftp	integer	Number of flows for FTP
ftp_data	integer	Number of flows for FTP data protocol
http	integer	Number of flows for HTTP
http2	integer	Number of flows for HTTP/2
ike	integer	Number of flows for IKE protocol
ikev2	integer	Number of flows for IKE v2 protocol
imap	integer	Number of flows for IMAP
krb5_tcp	integer	Number of flows for Kerberos v5/TCP protocol
krb5_udp	integer	Number of flows for Kerberos v5/UDP protocol
ldap_tcp	integer	Number of flows for LDAP/TCP protocol
ldap_udp	integer	Number of flows LDAP/UDP protocol
llmnr_tcp	integer	Number of flows for LLMNR/TCP protocol
llmnr_udp	integer	Number of flows for LLMNR/UDP protocol
mdns	integer	Number of flows for mDNS
modbus	integer	Number of flows for Modbus protocol
mqtt	integer	Number of flows for MQTT protocol
nfs_tcp	integer	Number of flows for NFS/TCP protocol
nfs_udp	integer	Number of flows for NFS/UDP protocol
ntp	integer	Number of flows for NTP
pgsql	integer	Number of flows for PostgreSQL protocol
pop3	integer	
quic	integer	Number of flows for QUIC protocol
rdp	integer	Number of flows for RDP

continues on next page

Table 7 – continued from previous page

Name	Type	Description
rfb	integer	Number of flows for RFB protocol
sip_tcp	integer	Number of flows for SIP/TCP protocol
sip_udp	integer	Number of flows for SIP/UDP protocol
smb	integer	Number of flows for SMB protocol
smtp	integer	Number of flows for SMTP
snmp	integer	Number of flows for SNMP
ssh	integer	Number of flows for SSH protocol
telnet	integer	Number of flows for Telnet protocol
tftp	integer	Number of flows for TFTP
tls	integer	Number of flows for TLS protocol
websocket	integer	

31.2.99 stats.app_layer.error (object)

Name	Type	Description
bittorrent_dht	object	
dcerpc_tcp	object	
dcerpc_udp	object	
dhcp	object	
dnp3	object	
dns_tcp	object	
dns_udp	object	
doh2	object	
enip_tcp	object	
enip_udp	object	
failed_tcp	object	
ftp	object	
ftp_data	object	
http	object	
http2	object	
ike	object	
imap	object	
krb5_tcp	object	
krb5_udp	object	
ldap_tcp	object	
ldap_udp	object	
llmnr_tcp	object	
llmnr_udp	object	
mdns	object	
modbus	object	
mqtt	object	
nfs_tcp	object	
nfs_udp	object	
ntp	object	
pgsql	object	
pop3	object	
quic	object	
rdp	object	
rfb	object	

continues on next page

Table 8 – continued from previous page

Name	Type	Description
sip_tcp	object	
sip_udp	object	
smb	object	
smtp	object	
snmp	object	
ssh	object	
telnet	object	
tftp	object	
tls	object	
websocket	object	

31.2.100 stats.app_layer.error.websocket (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.101 stats.app_layer.error.websocket.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.102 stats.app_layer.error.tls (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.103 stats.app_layer.error.tls.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.104 stats.app_layer.error.tftp (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.105 stats.app_layer.error.tftp.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.106 stats.app_layer.error.telnet (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.107 stats.app_layer.error.telnet.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.108 stats.app_layer.error.ssh (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.109 stats.app_layer.error.ssh.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.110 stats.app_layer.error.snmp (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.111 stats.app_layer.error.snmp.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.112 stats.app_layer.error.smtp (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.113 stats.app_layer.error.smtp.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.114 stats.app_layer.error.smb (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.115 stats.app_layer.error.smb.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.116 stats.app_layer.error.sip_udp (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.117 stats.app_layer.error.sip_udp.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.118 stats.app_layer.error.sip_tcp (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.119 stats.app_layer.error.sip_tcp.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.120 stats.app_layer.error.rfb (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.121 stats.app_layer.error.rfb.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.122 stats.app_layer.error.rdp (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.123 stats.app_layer.error.rdp.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.124 stats.app_layer.error.quic (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.125 stats.app_layer.error.quic.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.126 stats.app_layer.error.pop3 (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.127 stats.app_layer.error.pop3.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.128 stats.app_layer.error.pgsql (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.129 stats.app_layer.error.pgsql.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.130 stats.app_layer.error.ntp (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.131 stats.app_layer.error.ntp.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.132 stats.app_layer.error.nfs_udp (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.133 stats.app_layer.error.nfs_udp.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.134 stats.app_layer.error.nfs_tcp (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.135 stats.app_layer.error.nfs_tcp.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.136 stats.app_layer.error.mqtt (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.137 stats.app_layer.error.mqtt.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.138 stats.app_layer.error.modbus (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.139 stats.app_layer.error.modbus.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.140 stats.app_layer.error.mdns (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.141 stats.app_layer.error.mdns.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.142 stats.app_layer.error.llmnr_udp (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.143 stats.app_layer.error.llmnr_udp.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.144 stats.app_layer.error.llmnr_tcp (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.145 stats.app_layer.error.llmnr_tcp.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.146 stats.app_layer.error.ldap_udp (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.147 stats.app_layer.error.ldap_udp.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.148 stats.app_layer.error.ldap_tcp (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.149 stats.app_layer.error.ldap_tcp.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.150 stats.app_layer.error.krb5_udp (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.151 stats.app_layer.error.krb5_udp.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.152 stats.app_layer.error.krb5_tcp (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.153 stats.app_layer.error.krb5_tcp.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.154 stats.app_layer.error.imap (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.155 stats.app_layer.error.imap.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.156 stats.app_layer.error.ike (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.157 stats.app_layer.error.ike.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.158 stats.app_layer.error.http2 (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.159 stats.app_layer.error.http2.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.160 stats.app_layer.error.http (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.161 stats.app_layer.error.http.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.162 stats.app_layer.error.ftp_data (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.163 stats.app_layer.error.ftp_data.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.164 stats.app_layer.error.ftp (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.165 stats.app_layer.error.ftp.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.166 stats.app_layer.error.failed_tcp (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.167 stats.app_layer.error.failed_tcp.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.168 stats.app_layer.error.enip_udp (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.169 stats.app_layer.error.enip_udp.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.170 stats.app_layer.error.enip_tcp (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.171 stats.app_layer.error.enip_tcp.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.172 stats.app_layer.error.doh2 (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.173 stats.app_layer.error.doh2.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.174 stats.app_layer.error.dns_udp (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.175 stats.app_layer.error.dns_udp.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.176 stats.app_layer.error.dns_tcp (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.177 stats.app_layer.error.dns_tcp.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.178 stats.app_layer.error.dnp3 (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.179 stats.app_layer.error.dnp3.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.180 stats.app_layer.error.dhcp (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.181 stats.app_layer.error.dhcp.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.182 stats.app_layer.error.dcerpc_udp (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.183 stats.app_layer.error.dcerpc_udp.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.184 stats.app_layer.error.dcerpc_tcp (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.185 stats.app_layer.error.dcerpc_tcp.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.186 stats.app_layer.error.bittorrent_dht (object)

Name	Type	Description
alloc	integer	Number of errors allocating memory
exception_policy	object	
gap	integer	Number of errors processing gaps
internal	integer	Number of internal parser errors
parser	integer	Number of errors reported by parser

31.2.187 stats.app_layer.error.bittorrent_dht.exception_policy (object)

Name	Type	Description
bypass	integer	
drop_flow	integer	
drop_packet	integer	
pass_flow	integer	
pass_packet	integer	
reject	integer	
reject_both	integer	

31.2.188 ssh (object)

Name	Type	Description
client	object	
server	object	

31.2.189 ssh.server (object)

Name	Type	Description
error	string	Unrecoverable parse failure in this direction (invalid_banner or invalid_record)
hassh	object	
proto_version	string	
software_version	string	
state	string	State the parse failure occurred in (banner, kex or session); present only alongside the error field

31.2.190 ssh.server.hassh (object)

Name	Type	Description
hash	string	
string	string	

31.2.191 ssh.client (object)

Name	Type	Description
error	string	Unrecoverable parse failure in this direction (invalid_banner or invalid_record)
hassh	object	
proto_version	string	
software_version	string	
state	string	State the parse failure occurred in (banner, kex or session); present only alongside the error field

31.2.192 ssh.client.hassh (object)

Name	Type	Description
hash	string	
string	string	

31.2.193 snmp (object)

Name	Type	Description
community	string	
pdu_type	string	
trap_address	string	
trap_oid	string	
trap_type	string	
usm	string	
vars	array of strings	
version	integer	

31.2.194 smtp (object)

Name	Type	Description
helo	string	
mail_from	string	
rcpt_to	array of strings	

31.2.195 smb (object)

Name	Type	Description
access	string	
accessed	integer	
changed	integer	
client_dialects	array of strings	
client_guid	string	
command	string	
created	integer	
dcerpc	object	
dialect	string	
directory	string	
disposition	string	
filename	string	
fuid	string	
function	string	
id	integer	
kerberos	object	
level_of_interest	string	
max_read_size	integer	
max_write_size	integer	

continues on next page

Table 9 – continued from previous page

Name	Type	Description
modified	integer	
named_pipe	string	
ntlmssp	object	
rename	object	
request	object	
request_done	boolean	
response	object	
response_done	boolean	
server_guid	string	
service	object	
session_id	integer	
set_info	object	
share	string	
share_type	string	
size	integer	
status	string	
status_code	string	
subcmd	string	
tree_id	integer	

31.2.196 smb.set_info (object)

Name	Type	Description
class	string	
info_level	string	

31.2.197 smb.service (object)

Name	Type	Description
request	string	
response	string	

31.2.198 smb.response (object)

Name	Type	Description
native_lm	string	
native_os	string	

31.2.199 smb.request (object)

Name	Type	Description
native_lm	string	
native_os	string	

31.2.200 smb.rename (object)

Name	Type	Description
from	string	
to	string	

31.2.201 smb.ntlmssp (object)

Name	Type	Description
domain	string	
host	string	
user	string	
version	string	
warning	boolean	

31.2.202 smb.kerberos (object)

Name	Type	Description
realm	string	
snames	array of strings	

31.2.203 smb.dcerpc (object)

Name	Type	Description
call_id	integer	
interfaces	array of objects	
opnum	integer	
req	object	
request	string	
res	object	
response	string	

31.2.204 smb.dcerpc.res (object)

Name	Type	Description
frag_cnt	integer	
stub_data_size	integer	

31.2.205 smb.dcerpc.req (object)

Name	Type	Description
frag_cnt	integer	
stub_data_size	integer	

31.2.206 smb.dcerpc.interfaces (array of objects)

Name	Type	Description
ack_reason	integer	
ack_result	integer	
procedure	string	DCERPC Procedure name corresponding to the interface UUID and Opnum
service	string	DCERPC Service name corresponding to the interface UUID
uuid	string	
version	string	

31.2.207 sip (object)

Name	Type	Description
code	string	
method	string	
reason	string	
request_line	string	
response_line	string	
sdp	object	SDP message body
uri	string	
version	string	

31.2.208 sip.sdp (object)

Name	Type	Description
attributes	array of strings	A list of attributes to extend SDP
bandwidths	array of strings	Proposed bandwidths to be used by the session or media
connection_data	string	Connection data
email	string	Email address for the person responsible for the conference
encryption_key	string	Field used to convey encryption keys if SDP is used over a secure channel
media_descriptions	array of objects	A list of media descriptions for a session
origin	string	Owner of the session
phone_number	string	Phone number for the person responsible for the conference
session_info	string	Textual information about the session
session_name	string	Session name
time_descriptions	array of objects	A list of time descriptions for a session
timezone	string	Timezone to specify adjustments for times and offsets from the base time
uri	string	A pointer to additional information about the session
version	integer	SDP protocol version

31.2.209 sip.sdp.time_descriptions (array of objects)

Name	Type	Description
repeat_time	string	Specify repeat times for a session
time	string	Start and stop times for a session

31.2.210 sip.sdp.media_descriptions (array of objects)

Name	Type	Description
attributes	array of strings	A list of attributes specified for a media description
bandwidths	array of strings	A list of bandwidth proposed for a media
connection_data	string	Connection data per media description
encryption_key	string	Field used to convey encryption keys if SDP is used over a secure channel
media	string	Media description
media_info	string	Media information primarily intended for labelling media streams

31.2.211 sctp (object)

Name	Type	Description
chunk_cnt	integer	Number of SCTP chunks in the packet
chunk_types	array of strings	Array of SCTP chunk type names present in the packet
has_abort	boolean	Whether the packet contains an ABORT chunk
has_data	boolean	Whether the packet contains a DATA chunk
has_init	boolean	Whether the packet contains an INIT chunk
has_init_ack	boolean	Whether the packet contains an INIT_ACK chunk
vtag	integer	SCTP verification tag

31.2.212 rpc (object)

Name	Type	Description
auth_type	string	
creds	object	
status	string	
xid	integer	

31.2.213 rpc.creds (object)

Name	Type	Description
gid	integer	
machine_name	string	
uid	integer	

31.2.214 rgmp (object)

Name	Type	Description
type	integer	

31.2.215 rfb (object)

Name	Type	Description
authentication	object	
client_protocol_version	object	
framebuffer	object	
screen_shared	boolean	
server_protocol_version	object	
server_security_failure_reason	string	The string describing the reason why the server failed the security Handshake, see rfc 6143 section 7.1

31.2.216 rfb.server_protocol_version (object)

Name	Type	Description
major	string	
minor	string	

31.2.217 rfb.framebuffer (object)

Name	Type	Description
height	integer	
name	string	
pixel_format	object	
width	integer	

31.2.218 rfb.framebuffer.pixel_format (object)

Name	Type	Description
big_endian	boolean	
bits_per_pixel	integer	
blue_max	integer	
blue_shift	integer	
depth	integer	
green_max	integer	
green_shift	integer	
red_max	integer	
red_shift	integer	
true_color	boolean	

31.2.219 rfb.client_protocol_version (object)

Name	Type	Description
major	string	
minor	string	

31.2.220 rfb.authentication (object)

Name	Type	Description
security_result	string	
security_type	integer	
vnc	object	

31.2.221 rfb.authentication.vnc (object)

Name	Type	Description
challenge	string	
response	string	

31.2.222 rdp (object)

Name	Type	Description
channels	array of strings	
client	object	
cookie	string	
event_type	string	
tx_id	integer	

31.2.223 rdp.client (object)

Name	Type	Description
build	string	
capabilities	array of strings	
client_name	string	
color_depth	integer	
desktop_height	integer	
desktop_width	integer	
function_keys	integer	
id	string	
keyboard_layout	string	
keyboard_type	string	
product_id	integer	
version	string	

31.2.224 quic (object)

Name	Type	Description
cyu	array of objects	JA3-like fingerprint for versions of QUIC before standardization
extensions	array of objects	list of extensions in hello
ja3	object	JA3 from client, as in TLS
ja3s	object	JA3 from server, as in TLS
ja4	string	
sni	string	Server Name Indication
ua	string	User Agent for versions of QUIC before standardization
version	string	Quic protocol version

31.2.225 quic.ja3s (object)

Name	Type	Description
hash	string	JA3s hex representation
string	string	JA3s string representation

31.2.226 quic.ja3 (object)

Name	Type	Description
hash	string	JA3 hex representation
string	string	JA3 string representation

31.2.227 quic.extensions (array of objects)

Name	Type	Description
name	string	Human-friendly name of the extension
type	integer	Integer identifier of the extension
values	array of strings	Extension values

31.2.228 quic.cyu (array of objects)

Name	Type	Description
hash	string	CYU hash hex representation
string	string	CYU hash string representation

31.2.229 pop3 (object)

Name	Type	Description
request	object	
response	object	

31.2.230 pop3.response (object)

Name	Type	Description
data	array of strings	
header	string	First line of response
status	string	
success	boolean	Response indicated positive status ie +OK

31.2.231 pop3.request (object)

Name	Type	Description
args	array of strings	Pop3 request arguments
command	string	A pop3 command, for example *USER* or *STAT*

31.2.232 pgsql (object)

Name	Type	Description
request	object	
response	object	
tx_id	integer	

31.2.233 pgsql.response (object)

Name	Type	Description
authentication_md5_password	string	
authentication_sasl_final	string	
code	string	
command_completed	string	
copy_data_out	object	CopyData message from CopyOut mode
copy_in_response	object	Backend/server response accepting CopyIn mode
copy_out_response	object	Backend/server response accepting CopyOut mode
data_rows	integer	
data_size	integer	
field_count	integer	
file	string	
line	string	
message	string	The PostgreSQL message type seen on the wire. Or the primary human-readable error/notice PostgreSQL message present in an ErrorResponse message
parameter_status	array of objects	
position	string	String, representing a decimal ASCII integer, related to an error cursor position as an index to original PostgreSQL query string, as seen in an ErrorResponse message
process_id	integer	The process id of the backend process, as seen in BackendKeyData messages
routine	string	
row_description_malformed	boolean	True when the field count declared in a RowDescription message disagreed with the message body delimited by its length field
secret_key	integer	
severity_localizable	string	
severity_non_localizable	string	
ssl_accepted	boolean	
where	string	'where' field in an ErrorResponse PostgreSQL message, indicating the context in which the error occurred

31.2.234 pgsql.response.parameter_status (array of objects)

Name	Type	Description
application_name	string	
client_encoding	string	
date_style	string	
integer_datetimes	string	
interval_style	string	
is_superuser	string	
server_encoding	string	
server_version	string	
session_authorization	string	
standard_conforming_strings	string	
time_zone	string	

31.2.235 pgsql.response.copy_out_response (object)

Name	Type	Description
columns	integer	Number of columns that will be copied in the CopyData message
malformed	boolean	True in case the message body format on the wire disagreed with specifications

31.2.236 pgsql.response.copy_in_response (object)

Name	Type	Description
columns	integer	Number of columns that will be copied in the CopyData message
malformed	boolean	True in case the message body format on the wire disagreed with specifications

31.2.237 pgsql.response.copy_data_out (object)

Name	Type	Description
data_size	integer	Accumulated data size of all CopyData messages sent
row_count	integer	Number of rows sent in CopyData messages

31.2.238 pgsql.request (object)

Name	Type	Description
copy_data_in	object	CopyData message from CopyIn mode
message	string	
password	string	
password_redacted	boolean	Indicates if a password message was received but not logged due to Suricata settings
process_id	integer	The process id of the target backend, as seen in CancelRequests messages
protocol_version	string	
sasl_authentication_mechanism	string	
sasl_param	string	
sasl_response	string	
secret_key	integer	
simple_query	string	
startup_parameters	object	

31.2.239 pgsql.request.startup_parameters (object)

Name	Type	Description
optional_parameters	array of objects	
user	string	

31.2.240 pgsql.request.startup_parameters.optional_parameters (array of objects)

Name	Type	Description
application_name	string	
client_encoding	string	
database	string	
datestyle	string	
extra_float_digits	string	
options	string	
replication	string	

31.2.241 pgsql.request.copy_data_in (object)

Name	Type	Description
data_size	integer	Accumulated data size of all CopyData messages sent
msg_count	integer	How many CopyData messages were sent (does not necessarily match number of rows from the query)

31.2.242 packet_info (object)

Name	Type	Description
linktype	integer	
linktype_name	string	The descriptive name of the linktype

31.2.243 ntp (object)

Name	Type	Description
mode	integer	The mode of the NTP message
reference_id	string	Identifies specific server or reference clock as a colon-separated 4-byte hex string
stratum	integer	Indicates distance from the reference clock
version	integer	The NTP version number, typically 3 or 4

31.2.244 nfs (object)

Name	Type	Description
file_tx	boolean	
filename	string	
hhash	string	
id	integer	
procedure	string	
read	object	
rename	object	
status	string	
type	string	
version	integer	
write	object	

31.2.245 nfs.write (object)

Name	Type	Description
chunks	integer	
first	boolean	
last	boolean	
last_xid	integer	

31.2.246 nfs.rename (object)

Name	Type	Description
from	string	
to	string	

31.2.247 nfs.read (object)

Name	Type	Description
chunks	integer	
first	boolean	
last	boolean	
last_xid	integer	

31.2.248 netflow (object)

Name	Type	Description
age	integer	Duration of the flow (measured from timestamp of last packet and first packet)
bytes	integer	Total number of bytes transferred to server/client
end	string	Date of the end of the flow
max_ttl	integer	Maximum observed Time-To-Live (TTL) value
min_ttl	integer	Minimum observed TTL value
pkts	integer	Total number of packets transferred to server,client
start	string	Date of start of the flow
tx_cnt	integer	Number of transactions seen in the flow (only present if flow has an application layer)

31.2.249 mqtt (object)

Name	Type	Description
connack	object	
connect	object	
disconnect	object	
pingreq	object	
pingresp	object	
puback	object	
pubcomp	object	
publish	object	
pubrec	object	
pubrel	object	
suback	object	
subscribe	object	
unsuback	object	
unsubscribe	object	

31.2.250 mqtt.unsubscribe (object)

Name	Type	Description
dup	boolean	
message_id	integer	
qos	integer	
retain	boolean	
topics	array of strings	

31.2.251 mqtt.unsuback (object)

Name	Type	Description
dup	boolean	
message_id	integer	
qos	integer	
reason_codes	array of integers	
retain	boolean	

31.2.252 mqtt.subscribe (object)

Name	Type	Description
dup	boolean	
message_id	integer	
properties	object	
qos	integer	
retain	boolean	
topics	array of objects	

31.2.253 mqtt.subscribe.topics (array of objects)

Name	Type	Description
qos	integer	
topic	string	

31.2.254 mqtt.subscribe.properties (object)

Name	Type	Description
assigned_client_identifier	string	
authentication_data	string	
authentication_method	string	
content_type	string	
correlation_data	string	
maximum_packet_size	integer	
maximum_qos	integer	
message_expiry_interval	integer	
payload_format_indicator	integer	
reason_string	string	
receive_maximum	integer	
request_problem_information	integer	
request_response_information	integer	
response_information	string	
response_topic	string	
retain_available	integer	
server_keep_alive	integer	
server_reference	string	
session_expiry_interval	integer	
shared_subscription_available	integer	
subscription_identifier	integer	
subscrip- tion_identifier_available	integer	
topic_alias	integer	
topic_alias_maximum	integer	
user_properties	array of objects	
wild- card_subscription_available	integer	
will_delay_interval	integer	

31.2.255 mqtt.subscribe.properties.user_properties (array of objects)

Name	Type	Description
key	string	
value	string	

31.2.256 mqtt.suback (object)

Name	Type	Description
dup	boolean	
message_id	integer	
qos	integer	
qos_granted	array of integers	
retain	boolean	

31.2.257 mqtt.pubrel (object)

Name	Type	Description
dup	boolean	
message_id	integer	
properties	object	
qos	integer	
reason_code	integer	
retain	boolean	

31.2.258 mqtt.pubrel.properties (object)

Name	Type	Description
assigned_client_identifier	string	
authentication_data	string	
authentication_method	string	
content_type	string	
correlation_data	string	
maximum_packet_size	integer	
maximum_qos	integer	
message_expiry_interval	integer	
payload_format_indicator	integer	
reason_string	string	
receive_maximum	integer	
request_problem_information	integer	
request_response_information	integer	
response_information	string	
response_topic	string	
retain_available	integer	
server_keep_alive	integer	
server_reference	string	
session_expiry_interval	integer	
shared_subscription_available	integer	
subscription_identifier	integer	
subscrip- tion_identifier_available	integer	
topic_alias	integer	
topic_alias_maximum	integer	
user_properties	array of objects	
wild- card_subscription_available	integer	
will_delay_interval	integer	

31.2.259 mqtt.pubrel.properties.user_properties (array of objects)

Name	Type	Description
key	string	
value	string	

31.2.260 mqtt.pubrec (object)

Name	Type	Description
dup	boolean	
message_id	integer	
properties	object	
qos	integer	
reason_code	integer	
retain	boolean	

31.2.261 mqtt.pubrec.properties (object)

Name	Type	Description
assigned_client_identifier	string	
authentication_data	string	
authentication_method	string	
content_type	string	
correlation_data	string	
maximum_packet_size	integer	
maximum_qos	integer	
message_expiry_interval	integer	
payload_format_indicator	integer	
reason_string	string	
receive_maximum	integer	
request_problem_information	integer	
request_response_information	integer	
response_information	string	
response_topic	string	
retain_available	integer	
server_keep_alive	integer	
server_reference	string	
session_expiry_interval	integer	
shared_subscription_available	integer	
subscription_identifier	integer	
subscrip- tion_identifier_available	integer	
topic_alias	integer	
topic_alias_maximum	integer	
user_properties	array of objects	
wild- card_subscription_available	integer	
will_delay_interval	integer	

31.2.262 mqtt.pubrec.properties.user_properties (array of objects)

Name	Type	Description
key	string	
value	string	

31.2.263 mqtt.publish (object)

Name	Type	Description
dup	boolean	
message	string	
message_id	integer	
properties	object	
qos	integer	
retain	boolean	
skipped_length	integer	
topic	string	
truncated	boolean	

31.2.264 mqtt.publish.properties (object)

Name	Type	Description
assigned_client_identifier	string	
authentication_data	string	
authentication_method	string	
content_type	string	
correlation_data	string	
maximum_packet_size	integer	
maximum_qos	integer	
message_expiry_interval	integer	
payload_format_indicator	integer	
reason_string	string	
receive_maximum	integer	
request_problem_information	integer	
request_response_information	integer	
response_information	string	
response_topic	string	
retain_available	integer	
server_keep_alive	integer	
server_reference	string	
session_expiry_interval	integer	
shared_subscription_available	integer	
subscription_identifier	integer	
subscrip- tion_identifier_available	integer	
topic_alias	integer	
topic_alias_maximum	integer	
user_properties	array of objects	
wild- card_subscription_available	integer	
will_delay_interval	integer	

31.2.265 mqtt.publish.properties.user_properties (array of objects)

Name	Type	Description
key	string	
value	string	

31.2.266 mqtt.pubcomp (object)

Name	Type	Description
dup	boolean	
message_id	integer	
properties	object	
qos	integer	
reason_code	integer	
retain	boolean	

31.2.267 mqtt.pubcomp.properties (object)

Name	Type	Description
assigned_client_identifier	string	
authentication_data	string	
authentication_method	string	
content_type	string	
correlation_data	string	
maximum_packet_size	integer	
maximum_qos	integer	
message_expiry_interval	integer	
payload_format_indicator	integer	
reason_string	string	
receive_maximum	integer	
request_problem_information	integer	
request_response_information	integer	
response_information	string	
response_topic	string	
retain_available	integer	
server_keep_alive	integer	
server_reference	string	
session_expiry_interval	integer	
shared_subscription_available	integer	
subscription_identifier	integer	
subscrip- tion_identifier_available	integer	
topic_alias	integer	
topic_alias_maximum	integer	
user_properties	array of objects	
wild- card_subscription_available	integer	
will_delay_interval	integer	

31.2.268 mqtt.pubcomp.properties.user_properties (array of objects)

Name	Type	Description
key	string	
value	string	

31.2.269 mqtt.puback (object)

Name	Type	Description
dup	boolean	
message_id	integer	
properties	object	
qos	integer	
reason_code	integer	
retain	boolean	

31.2.270 mqtt.puback.properties (object)

Name	Type	Description
assigned_client_identifier	string	
authentication_data	string	
authentication_method	string	
content_type	string	
correlation_data	string	
maximum_packet_size	integer	
maximum_qos	integer	
message_expiry_interval	integer	
payload_format_indicator	integer	
reason_string	string	
receive_maximum	integer	
request_problem_information	integer	
request_response_information	integer	
response_information	string	
response_topic	string	
retain_available	integer	
server_keep_alive	integer	
server_reference	string	
session_expiry_interval	integer	
shared_subscription_available	integer	
subscription_identifier	integer	
subscrip- tion_identifier_available	integer	
topic_alias	integer	
topic_alias_maximum	integer	
user_properties	array of objects	
wild- card_subscription_available	integer	
will_delay_interval	integer	

31.2.271 mqtt.puback.properties.user_properties (array of objects)

Name	Type	Description
key	string	
value	string	

31.2.272 mqtt.pingresp (object)

Name	Type	Description
dup	boolean	
qos	integer	
retain	boolean	

31.2.273 mqtt.pingreq (object)

Name	Type	Description
dup	boolean	
qos	integer	
retain	boolean	

31.2.274 mqtt.disconnect (object)

Name	Type	Description
dup	boolean	
properties	object	
qos	integer	
reason_code	integer	
retain	boolean	

31.2.275 mqtt.disconnect.properties (object)

Name	Type	Description
assigned_client_identifier	string	
authentication_data	string	
authentication_method	string	
content_type	string	
correlation_data	string	
maximum_packet_size	integer	
maximum_qos	integer	
message_expiry_interval	integer	
payload_format_indicator	integer	
reason_string	string	
receive_maximum	integer	
request_problem_information	integer	
request_response_information	integer	
response_information	string	
response_topic	string	
retain_available	integer	
server_keep_alive	integer	
server_reference	string	
session_expiry_interval	integer	
shared_subscription_available	integer	
subscription_identifier	integer	
subscrip- tion_identifier_available	integer	
topic_alias	integer	
topic_alias_maximum	integer	
user_properties	array of objects	
wild- card_subscription_available	integer	
will_delay_interval	integer	

31.2.276 mqtt.disconnect.properties.user_properties (array of objects)

Name	Type	Description
key	string	
value	string	

31.2.277 mqtt.connect (object)

Name	Type	Description
client_id	string	
dup	boolean	
flags	object	
password	string	
properties	object	
protocol_string	string	
protocol_version	integer	
qos	integer	
retain	boolean	
username	string	
will	object	

31.2.278 mqtt.connect.will (object)

Name	Type	Description
message	string	
properties	object	
topic	string	

31.2.279 mqtt.connect.will.properties (object)

Name	Type	Description
assigned_client_identifier	string	
authentication_data	string	
authentication_method	string	
content_type	string	
correlation_data	string	
maximum_packet_size	integer	
maximum_qos	integer	
message_expiry_interval	integer	
payload_format_indicator	integer	
reason_string	string	
receive_maximum	integer	
request_problem_information	integer	
request_response_information	integer	
response_information	string	
response_topic	string	
retain_available	integer	
server_keep_alive	integer	
server_reference	string	
session_expiry_interval	integer	
shared_subscription_available	integer	
subscription_identifier	integer	
subscrip- tion_identifier_available	integer	
topic_alias	integer	
topic_alias_maximum	integer	
user_properties	array of objects	
wild- card_subscription_available	integer	
will_delay_interval	integer	

31.2.280 mqtt.connect.will.properties.user_properties (array of objects)

Name	Type	Description
key	string	
value	string	

31.2.281 mqtt.connect.properties (object)

Name	Type	Description
assigned_client_identifier	string	
authentication_data	string	
authentication_method	string	
content_type	string	
correlation_data	string	
maximum_packet_size	integer	
maximum_qos	integer	
message_expiry_interval	integer	
payload_format_indicator	integer	
reason_string	string	
receive_maximum	integer	
request_problem_information	integer	
request_response_information	integer	
response_information	string	
response_topic	string	
retain_available	integer	
server_keep_alive	integer	
server_reference	string	
session_expiry_interval	integer	
shared_subscription_available	integer	
subscription_identifier	integer	
subscrip- tion_identifier_available	integer	
topic_alias	integer	
topic_alias_maximum	integer	
user_properties	array of objects	
wild- card_subscription_available	integer	
will_delay_interval	integer	

31.2.282 mqtt.connect.properties.user_properties (array of objects)

Name	Type	Description
key	string	
value	string	

31.2.283 mqtt.connect.flags (object)

Name	Type	Description
clean_session	boolean	
password	boolean	
username	boolean	
will	boolean	
will_retain	boolean	

31.2.284 mqtt.connack (object)

Name	Type	Description
dup	boolean	
properties	object	
qos	integer	
retain	boolean	
return_code	integer	
session_present	boolean	

31.2.285 mqtt.connack.properties (object)

Name	Type	Description
assigned_client_identifier	string	
authentication_data	string	
authentication_method	string	
content_type	string	
correlation_data	string	
maximum_packet_size	integer	
maximum_qos	integer	
message_expiry_interval	integer	
payload_format_indicator	integer	
reason_string	string	
receive_maximum	integer	
request_problem_information	integer	
request_response_information	integer	
response_information	string	
response_topic	string	
retain_available	integer	
server_keep_alive	integer	
server_reference	string	
session_expiry_interval	integer	
shared_subscription_available	integer	
subscription_identifier	integer	
subscrip- tion_identifier_available	integer	
topic_alias	integer	
topic_alias_maximum	integer	
user_properties	array of objects	
wild- card_subscription_available	integer	
will_delay_interval	integer	

31.2.286 mqtt.connack.properties.user_properties (array of objects)

Name	Type	Description
key	string	
value	string	

31.2.287 modbus (object)

Name	Type	Description
id	integer	
request	object	
response	object	

31.2.288 modbus.response (object)

Name	Type	Description
access_type	string	
category	string	
data	string	
diagnostic	object	
error_flags	string	
exception	object	
function_code	string	
function_raw	integer	
protocol_id	integer	
read	object	
transaction_id	integer	
unit_id	integer	
write	object	

31.2.289 modbus.response.write (object)

Name	Type	Description
address	integer	
data	integer	

31.2.290 modbus.response.read (object)

Name	Type	Description
data	string	

31.2.291 modbus.response.exception (object)

Name	Type	Description
code	string	
raw	integer	

31.2.292 modbus.response.diagnostic (object)

Name	Type	Description
code	string	
data	string	
raw	integer	

31.2.293 modbus.request (object)

Name	Type	Description
access_type	string	
category	string	
data	string	
diagnostic	object	
error_flags	string	
function_code	string	
function_raw	integer	
mei	object	
protocol_id	integer	
read	object	
transaction_id	integer	
unit_id	integer	
write	object	

31.2.294 modbus.request.write (object)

Name	Type	Description
address	integer	
data	integer	

31.2.295 modbus.request.read (object)

Name	Type	Description
address	integer	
quantity	integer	

31.2.296 modbus.request.mei (object)

Name	Type	Description
code	string	
data	string	
raw	integer	

31.2.297 modbus.request.diagnostic (object)

Name	Type	Description
code	string	
data	string	
raw	integer	

31.2.298 metadata (object)

Name	Type	Description
entropy	object	
flowbits	array of strings	
flowints	object	
flowvars	array of objects	
pktvars	array of objects	

31.2.299 metadata.pktvars (array of objects)

Name	Type	Description
uid	string	
username	string	

31.2.300 metadata.flowvars (array of objects)

Name	Type	Description
gid	string	
key	string	
value	string	

31.2.301 mdns (object)

Name	Type	Description
additionals	array of objects	mDNS additional records
answers	array of objects	mDNS answer records
authorities	array of objects	mDNS authority records
flags	array of unknowns	mDNS message flags
id	integer	mDNS transaction ID
opcode	integer	mDNS opcode value
queries	array of objects	mDNS query records
rcode	integer	mDNS reply (error) code
type	string	Type of message, either a request or response

31.2.302 mdns.queries (array of objects)

Name	Type	Description
rrname	string	Resource name being requested
rrname_truncated	boolean	Name was truncated by Suricata due to length
rrtype	string	Type of resource being requested

31.2.303 mdns.authorities (array of objects)

Name	Type	Description
rrname	string	Resource name of the record being returned
rrname_truncated	boolean	Name was truncated by Suricata due to length

31.2.304 mdns.answers (array of objects)

Name	Type	Description
ptr	string	Value of the requested PTR record
rrname	string	Resource name of the record being returned
rrname_truncated	boolean	Name was truncated by Suricata due to length
txt	array of strings	Value of the requested TXT record

31.2.305 mdns.additionals (array of objects)

Name	Type	Description
ptr	string	Value of the requested PTR record
rrname	string	Resource name of the record being returned
rrname_truncated	boolean	Name was truncated by Suricata due to length
txt	array of strings	Value of the requested TXT record

31.2.306 llmnr (object)

Name	Type	Description
additionals	array of objects	LLMNR additional records
answers	array of objects	LLMNR answer records
authorities	array of objects	LLMNR authority records
flags	array of strings	LLMNR message flags
grouped	object	Grouped answer records by type
id	integer	LLMNR transaction ID
opcode	integer	LLMNR opcode value
queries	array of objects	LLMNR query records
tx_id	integer	Internal transaction ID
type	string	Type of message, either a request or response

31.2.307 llmnr.queries (array of objects)

Name	Type	Description
rrname	string	Resource name being requested
rrname_truncated	boolean	Name was truncated by Suricata due to length
rrtype	string	Type of resource being requested

31.2.308 llmnr.grouped (object)

Name	Type	Description
A	array of strings	IPv4 address records
AAAA	array of strings	IPv6 address records
CNAME	array of strings	Canonical name records
MX	array of strings	Mail exchange records
NS	array of strings	Name server records
NULL	array of strings	NULL record data
PTR	array of strings	Pointer records
SOA	array of unknowns	Start of authority records
TXT	array of strings	Text records

31.2.309 llmnr.authorities (array of objects)

Name	Type	Description
a	string	IPv4 address
aaaa	string	IPv6 address
cname	string	Canonical name
mx	string	Mail exchange
ns	string	Name server
null	string	NULL record data
ptr	string	Pointer record
rrname	string	Resource name of the record
rrname_truncated	boolean	Name was truncated by Suricata due to length
rrtype	string	Resource record type
soa	object	
ttl	integer	Time to live in seconds
txt	array of strings	TXT record values
unknown	string	Unknown record type data

31.2.310 llmnr.authorities.soa (object)

Name	Type	Description
expire	integer	
minimum	integer	
mname	string	
mname_truncated	boolean	Set to true if the mname was too long and truncated by Suricata
refresh	integer	
retry	integer	
rname	string	
serial	integer	

31.2.311 llmnr.answers (array of objects)

Name	Type	Description
a	string	IPv4 address
aaaa	string	IPv6 address
cname	string	Canonical name
mx	string	Mail exchange
ns	string	Name server
null	string	NULL record data
ptr	string	Pointer record
rrname	string	Resource name of the record
rrname_truncated	boolean	Name was truncated by Suricata due to length
rrtype	string	Resource record type
soa	object	
srv	object	SRV record data
sshfp	object	SSH fingerprint record
ttl	integer	Time to live in seconds
txt	array of strings	TXT record values
unknown	string	Unknown record type data

31.2.312 llmnr.answers.sshfp (object)

Name	Type	Description
algo	integer	Algorithm number
fingerprint	string	Fingerprint value
type	integer	Fingerprint type

31.2.313 llmnr.answers.srv (object)

Name	Type	Description
name	string	Target hostname
port	integer	Target port
priority	integer	Priority value
weight	integer	Weight value

31.2.314 llmnr.answers.soa (object)

Name	Type	Description
expire	integer	
minimum	integer	
mname	string	
mname_truncated	boolean	Set to true if the mname was too long and truncated by Suricata
refresh	integer	
retry	integer	
rname	string	
serial	integer	

31.2.315 llmnr.additionals (array of objects)

Name	Type	Description
a	string	IPv4 address
aaaa	string	IPv6 address
cname	string	Canonical name
mx	string	Mail exchange
ns	string	Name server
null	string	NULL record data
opt	array of objects	EDNS OPT records
ptr	string	Pointer record
rrname	string	Resource name of the record
rrname_truncated	boolean	Name was truncated by Suricata due to length
rrtype	string	Resource record type
ttl	integer	Time to live in seconds
txt	array of strings	TXT record values
unknown	string	Unknown record type data

31.2.316 llmnr.additionals.opt (array of objects)

Name	Type	Description
code	integer	EDNS option code
data	string	EDNS option data

31.2.317 ldap (object)

Name	Type	Description
request	object	
responses	array of objects	

31.2.318 ldap.responses (array of objects)

Name	Type	Description
add_response	object	
bind_response	object	
compare_response	object	
del_response	object	
extended_response	object	
intermediate_response	object	
message_id	integer	
mod_dn_response	object	
modify_response	object	
operation	string	
search_result_done	object	
search_result_entry	object	

31.2.319 ldap.responses.search_result_entry (object)

Name	Type	Description
attributes	array of objects	
base_object	string	

31.2.320 ldap.responses.search_result_entry.attributes (array of objects)

Name	Type	Description
type	string	
values	array of strings	

31.2.321 ldap.responses.search_result_done (object)

Name	Type	Description
matched_dn	string	
message	string	
result_code	string	

31.2.322 ldap.responses.modify_response (object)

Name	Type	Description
matched_dn	string	
message	string	
result_code	string	

31.2.323 ldap.responses.mod_dn_response (object)

Name	Type	Description
matched_dn	string	
message	string	
result_code	string	

31.2.324 ldap.responses.intermediate_response (object)

Name	Type	Description
name	string	
value	string	

31.2.325 ldap.responses.extended_response (object)

Name	Type	Description
matched_dn	string	
message	string	
name	string	
result_code	string	
value	string	

31.2.326 ldap.responses.del_response (object)

Name	Type	Description
matched_dn	string	
message	string	
result_code	string	

31.2.327 ldap.responses.compare_response (object)

Name	Type	Description
matched_dn	string	
message	string	
result_code	string	

31.2.328 ldap.responses.bind_response (object)

Name	Type	Description
matched_dn	string	
message	string	
result_code	string	
server_sasl_creds	string	

31.2.329 ldap.responses.add_response (object)

Name	Type	Description
matched_dn	string	
message	string	
result_code	string	

31.2.330 ldap.request (object)

Name	Type	Description
abandon_request	object	
add_request	object	
bind_request	object	
compare_request	object	
del_request	object	
extended_request	object	
message_id	integer	
mod_dn_request	object	
modify_request	object	
operation	string	
search_request	object	

31.2.331 ldap.request.search_request (object)

Name	Type	Description
attributes	array of strings	
base_object	string	
deref_alias	integer	
scope	integer	
size_limit	integer	
time_limit	integer	
types_only	boolean	

31.2.332 ldap.request.modify_request (object)

Name	Type	Description
changes	array of objects	
object	string	

31.2.333 ldap.request.modify_request.changes (array of objects)

Name	Type	Description
modification	object	
operation	string	

31.2.334 ldap.request.modify_request.changes.modification (object)

Name	Type	Description
attribute_type	string	
attribute_values	array of strings	

31.2.335 ldap.request.mod_dn_request (object)

Name	Type	Description
delete_old_rdn	boolean	
entry	string	
new_rdn	string	
new_superior	string	

31.2.336 ldap.request.extended_request (object)

Name	Type	Description
name	string	
value	string	

31.2.337 ldap.request.del_request (object)

Name	Type	Description
dn	string	

31.2.338 ldap.request.compare_request (object)

Name	Type	Description
attribute_value_assertion	object	
entry	string	

31.2.339 ldap.request.compare_request.attribute_value_assertion (object)

Name	Type	Description
description	string	
value	string	

31.2.340 ldap.request.bind_request (object)

Name	Type	Description
name	string	
sasl	object	
version	integer	

31.2.341 ldap.request.bind_request.sasl (object)

Name	Type	Description
credentials	string	
mechanism	string	

31.2.342 ldap.request.add_request (object)

Name	Type	Description
attributes	array of objects	
entry	string	

31.2.343 ldap.request.add_request.attributes (array of objects)

Name	Type	Description
name	string	
values	array of strings	

31.2.344 ldap.request.abandon_request (object)

Name	Type	Description
message_id	integer	

31.2.345 krb5 (object)

Name	Type	Description
cname	string	The client PrincipalName
encryption	string	Encryption used (only in AS-REP and TGS-REP)
error_code	string	Error code, if request has failed
failed_request	string	The request type for which the response had an error_code
msg_type	string	The message type: AS-REQ, AS-REP, etc...
realm	string	The server Realm
sname	string	The server PrincipalName
ticket_encryption	string	Encryption used for ticket
ticket_weak_encryption	boolean	Whether the encryption used for ticket is a weak cipher
weak_encryption	boolean	Whether the encryption used in AS-REP or TGS-REP is a weak cipher

31.2.346 ike (object)

Name	Type	Description
_v	integer	
attributes	array of objects	
exchange_type	integer	
exchange_type_verbose	string	
ikev1	object	
ikev2	object	
init_spi	string	
message_id	integer	
payload	array of strings	
resp_spi	string	
role	string	
version_major	integer	
version_minor	integer	

31.2.347 ike.ikev2 (object)

Name	Type	Description
errors	integer	
notify	array of unknowns	

31.2.348 ike.ikev1 (object)

Name	Type	Description
client	object	
doi	integer	
encrypted_payloads	boolean	
server	object	
vendor_ids	array of strings	

31.2.349 ike.ikev1.server (object)

Name	Type	Description
key_exchange_payload	string	
key_exchange_payload_length	integer	
nonce_payload	string	
nonce_payload_length	integer	

31.2.350 ike.ikev1.client (object)

Name	Type	Description
key_exchange_payload	string	
key_exchange_payload_length	integer	
nonce_payload	string	
nonce_payload_length	integer	
proposals	array of objects	

31.2.351 ike.ikev1.client.proposals (array of objects)

Name	Type	Description
key	string	
raw	string or number	
value	string	

31.2.352 ike.attributes (array of objects)

Name	Type	Description
key	string	
raw	string or number	
value	string	

31.2.353 igmp (object)

Name	Type	Description
type	integer	
version	integer	

31.2.354 http (object)

Name	Type	Description
content_range	object	
hostname	string	The domain name of the server, the Host header
http2	object	
http_content_type	string	The media type of the resource or data, the Content-Type header
http_method	string	The HTTP request method
http_port	integer	The port in the Host header if any
http_refer	string	An absolute or partial address of the web page that makes the request, the Referer header
http_request_body	string	Base64 of the request body
http_request_body_printable	string	The ascii-printable characters of the request body
http_response_body	string	Base64 of the response body
http_response_body_printable	string	The ascii-printable characters of the response body
http_user_agent	string	A characteristic string that lets servers and network peers identify the application, operating system, vendor, and/or version of the requesting user agent, the User-Agent header
length	integer	The response message length
protocol	string	The HTTP protocol with its version
redirect	string	The URL to redirect a page to, the Location header
request_headers	array of objects	
response_headers	array of objects	
status	integer	Status as integer value
status_string	string	Status string when it is not a valid integer (like 2XX)
url	string	The HTTP request URI
version	string	HTTP major version : 2 when HTTP/2 is used
xff	string	A de-facto standard header for identifying the originating IP address of a client connecting to a web server through a proxy server, the X-Forwarded-For header request header

31.2.355 http.response_headers (array of objects)

Name	Type	Description
name	string	
table_size_update	integer	
value	string	

31.2.356 http.request_headers (array of objects)

Name	Type	Description
name	string	
table_size_update	integer	
value	string	

31.2.357 http.http2 (object)

Name	Type	Description
request	object	
response	object	
stream_id	integer	

31.2.358 http.http2.response (object)

Name	Type	Description
error_code	string	
has_multiple	string	
settings	array of objects	

31.2.359 http.http2.response.settings (array of objects)

Name	Type	Description
settings_id	string	
settings_value	integer	

31.2.360 http.http2.request (object)

Name	Type	Description
error_code	string	
has_multiple	string	
priority	integer	
settings	array of objects	

31.2.361 http.http2.request.settings (array of objects)

Name	Type	Description
settings_id	string	
settings_value	integer	

31.2.362 http.content_range (object)

Name	Type	Description
end	integer	Range end in Content-Range header
raw	string	Raw Content-Range header
size	integer	Total length of document in Content-Range header
start	integer	Range start in Content-Range header

31.2.363 ftp_data (object)

Name	Type	Description
command	string	
filename	string	

31.2.364 ftp (object)

Name	Type	Description
command	string	
command_data	string	
command_truncated	boolean	
completion_code	array of strings	
dynamic_port	integer	
mode	string	
reply	array of strings	
reply_received	string	
reply_truncated	boolean	

31.2.365 frame (object)

Name	Type	Description
complete	boolean	
direction	string	
id	integer	
length	integer	
payload	string	
payload_printable	string	
stream_offset	integer	
tx_id	integer	
type	string	

31.2.366 flow (object)

Name	Type	Description
action	string	
age	integer	
alerted	boolean	
bypass	string	
bypassed	object	
bytes_toclient	integer	
bytes_toserver	integer	
dest_ip	string	
dest_port	integer	
elephant	boolean	
elephant_direction	array of unknowns	Direction(s) in which flow was found to be elephant
emergency	boolean	
end	string	
exception_policy	array of unknowns	The exception policy(ies) triggered by the flow. Not logged if none was triggered
pkts_toclient	integer	
pkts_toserver	integer	
reason	string	
src_ip	string	
src_port	integer	
start	string	
state	string	
tx_cnt	integer	
wrong_thread	boolean	

31.2.367 flow.bypassed (object)

Name	Type	Description
bytes_toclient	integer	
bytes_toserver	integer	
pkts_toclient	integer	
pkts_toserver	integer	

31.2.368 firewall (object)

Name	Type	Description
hook	string	Firewall hook for the match
policy	string	Firewall actions for the match (from rule or default policy)

31.2.369 files (array of objects)

Name	Type	Description
end	integer	
file_id	integer	
filename	string	
gaps	boolean	
magic	string	
md5	string	
sha1	string	
sha256	string	
sid	array of integers	
size	integer	
start	integer	
state	string	
stored	boolean	
storing	boolean	The file is set to be stored when completed
tx_id	integer	

31.2.370 fileinfo (object)

Name	Type	Description
end	integer	The offset of the last byte captured
file_id	integer	Represents the id of a file that has been stored
filename	string	Name of the file as observed in network traffic
gaps	boolean	Indicates if there were gaps in the file
magic	string	[optional, requires libmagic] The magic value for the file
md5	string	[optional, if state is `CLOSED`] When closed, md5 sum
sha1	string	[optional, if state is `CLOSED`] When closed, sha1 sum
sha256	string	The sha256 value for the file, if available
sid	array of integers	
size	integer	The observed size fo the file, in bytes
start	integer	The offset of the first byte captured
state	string	The state of the file when the record is written
stored	boolean	Indicates whether the file has been stored
storing	boolean	Indicates whether the file is in the process of being stored; true when not yet stored
tx_id	integer	The transaction id in effect

31.2.371 ether (object)

Name	Type	Description
dest_mac	string	
dest_macs	array of strings	
ether_type	integer	Ethernet type value
src_mac	string	
src_macs	array of strings	

31.2.372 enip (object)

Name	Type	Description
request	object	
response	object	

31.2.373 enip.response (object)

Name	Type	Description
cip	object	
command	string	
identity	object	
list_services	object	
register_session	object	
status	string	

31.2.374 enip.response.register_session (object)

Name	Type	Description
options	integer	
protocol_version	integer	

31.2.375 enip.response.list_services (object)

Name	Type	Description
capabilities	integer	
protocol_version	integer	
service_name	string	

31.2.376 enip.response.identity (object)

Name	Type	Description
device_type	string	
product_code	integer	
product_name	string	
protocol_version	integer	
revision	string	
serial	integer	
state	integer	
status	integer	
vendor_id	string	

31.2.377 enip.response.cip (object)

Name	Type	Description
multiple	array of objects	
service	string	
status	string	
status_extended	string	
status_extended_meaning	string	

31.2.378 enip.response.cip.multiple (array of objects)

Name	Type	Description
service	string	
status	string	
status_extended	string	
status_extended_meaning	string	

31.2.379 enip.request (object)

Name	Type	Description
cip	object	
command	string	
register_session	object	
status	string	

31.2.380 enip.request.register_session (object)

Name	Type	Description
options	integer	
protocol_version	integer	

31.2.381 enip.request.cip (object)

Name	Type	Description
class_name	string	
multiple	array of objects	
path	array of objects	
service	string	

31.2.382 enip.request.cip.path (array of objects)

Name	Type	Description
segment_type	string	
value	integer	

31.2.383 enip.request.cip.multiple (array of objects)

Name	Type	Description
class_name	string	
path	array of objects	
service	string	

31.2.384 enip.request.cip.multiple.path (array of objects)

Name	Type	Description
segment_type	string	
value	integer	

31.2.385 engine (object)

Name	Type	Description
error	string	
error_code	integer	
message	string	
module	string	
thread_name	string	

31.2.386 email (object)

Name	Type	Description
attachment	array of strings	
body_md5	string	
cc	array of strings	
date	string	
from	string	
message_id	string	
received	array of strings	
status	string	
subject	string	
subject_md5	string	
to	array of strings	
url	array of strings	
x_mailer	string	

31.2.387 drop (object)

Name	Type	Description
ack	boolean	
fin	boolean	
flowlbl	integer	
hoplimit	integer	
icmp_id	integer	
icmp_seq	integer	
ipid	integer	
len	integer	
psh	boolean	
reason	string	
rst	boolean	
syn	boolean	
tc	integer	
tcpack	integer	
tcpres	integer	
tcpseq	integer	
tcpurgp	integer	
tcpwin	integer	
tos	integer	
ttl	integer	
udplen	integer	
urg	boolean	
verdict	object	

31.2.388 drop.verdict (object)

Name	Type	Description
action	string	
reject	array of strings	
reject_target	string	

31.2.389 dns (object)

Name	Type	Description
aa	boolean	
additionals	array of objects	
answer	object	
answers	array of objects	
authorities	array of objects	
flags	string	
grouped	object	
id	integer	
opcode	integer	DNS opcode as an integer
qr	boolean	
queries	array of objects	
query	array of objects	
ra	boolean	
rcode	string	
rd	boolean	
rrname	string	
rrtype	string	
tc	boolean	DNS truncation flag
tx_id	integer	
type	string	
version	integer	The version of this EVE DNS event
z	boolean	

31.2.390 dns.query (array of objects)

Name	Type	Description
id	integer	
opcode	integer	DNS opcode as an integer
rrname	string	
rrtype	string	
tx_id	integer	
type	string	
z	boolean	

31.2.391 dns.queries (array of objects)

Name	Type	Description
id	integer	
opcode	integer	DNS opcode as an integer
rrname	string	
rrname_truncated	boolean	Set to true if the rrname was too long and truncated by Suricata
rrtype	string	
tx_id	integer	
type	string	
z	boolean	

31.2.392 dns.grouped (object)

Name	Type	Description
A	array of strings	
AAAA	array of strings	
CNAME	array of strings	
MX	array of strings	
NS	array of strings	
NULL	array of strings	
PTR	array of strings	
SOA	array of unknowns	
SRV	array of objects	
SSHFP	array of objects	A Secure Shell fingerprint is used to verify the system's authenticity
TXT	array of strings	

31.2.393 dns.grouped.SSHFP (array of objects)

Name	Type	Description
algo	integer	
fingerprint	string	
type	integer	

31.2.394 dns.grouped.SRV (array of objects)

Name	Type	Description
name	string	
port	integer	
priority	integer	
weight	integer	

31.2.395 dns.authorities (array of objects)

Name	Type	Description
rdata	string	
rdata_truncated	boolean	Set to true if the rdata was too long and truncated by Suricata
rrname	string	
rrname_truncated	boolean	Set to true if the rrname was too long and truncated by Suricata
rrtype	string	
soa	object	
ttl	integer	

31.2.396 dns.authorities.soa (object)

Name	Type	Description
expire	integer	
minimum	integer	
mname	string	
mname_truncated	boolean	Set to true if the mname was too long and truncated by Suricata
refresh	integer	
retry	integer	
rname	string	
serial	integer	

31.2.397 dns.answers (array of objects)

Name	Type	Description
rdata	string	
rrname	string	
rrtype	string	
soa	object	
srv	object	
sshfp	object	A Secure Shell fingerprint, used to verify the system's authenticity
ttl	integer	

31.2.398 dns.answers.sshfp (object)

Name	Type	Description
algo	integer	
fingerprint	string	
type	integer	

31.2.399 dns.answers.srv (object)

Name	Type	Description
name	string	
port	integer	
priority	integer	
weight	integer	

31.2.400 dns.answers.soa (object)

Name	Type	Description
expire	integer	
minimum	integer	
mname	string	
mname_truncated	boolean	Set to true if the mname was too long and truncated by Suricata
refresh	integer	
retry	integer	
rname	string	
serial	integer	

31.2.401 dns.answer (object)

Name	Type	Description
additionals	array of objects	
authorities	array of objects	
flags	string	
id	integer	
opcode	integer	DNS opcode as an integer
qr	boolean	
ra	boolean	
rcode	string	
rd	boolean	
rrname	string	
rrtype	string	
type	string	
version	integer	

31.2.402 dns.answer.authorities (array of objects)

Name	Type	Description
rdata	string	
rdata_truncated	boolean	Set to true if the rdata was too long and truncated by Suricata
rrname	string	
rrname_truncated	boolean	Set to true if the rrname was too long and truncated by Suricata
rrtype	string	
soa	object	
ttl	integer	

31.2.403 dns.answer.authorities.soa (object)

Name	Type	Description
expire	integer	
minimum	integer	
mname	string	
mname_truncated	boolean	Set to true if the mname was too long and truncated by Suricata
refresh	integer	
retry	integer	
rname	string	
serial	integer	

31.2.404 dns.answer.additionals (array of objects)

Name	Type	Description
opt	array of objects	
rdata	string	
rrname	string	
rrtype	string	
ttl	integer	

31.2.405 dns.answer.additionals.opt (array of objects)

Name	Type	Description
code	integer	
data	string	

31.2.406 dns.additionals (array of objects)

Name	Type	Description
opt	array of objects	
rdata	string	
rrname	string	
rrtype	string	
ttl	integer	

31.2.407 dns.additionals.opt (array of objects)

Name	Type	Description
code	integer	
data	string	

31.2.408 dnp3 (object)

Name	Type	Description
application	object	
control	object	
dst	integer	
iin	object	
request	object	
response	object	
src	integer	
type	string	

31.2.409 dnp3.response (object)

Name	Type	Description
application	object	
control	object	
dst	integer	
iin	object	
src	integer	
type	string	

31.2.410 dnp3.response.iin (object)

Name	Type	Description
indicators	array of strings	

31.2.411 dnp3.response.control (object)

Name	Type	Description
dir	boolean	
fcb	boolean	
fcv	boolean	
function_code	integer	
pri	boolean	

31.2.412 dnp3.response.application (object)

Name	Type	Description
complete	boolean	
control	object	
function_code	integer	
objects	array of objects	

31.2.413 dnp3.response.application.objects (array of objects)

Name	Type	Description
count	integer	
group	integer	
points	array of objects	
prefix_code	integer	
qualifier	integer	
range_code	integer	
start	integer	
stop	integer	
variation	integer	

31.2.414 dnp3.response.application.control (object)

Name	Type	Description
con	boolean	
fin	boolean	
fir	boolean	
sequence	integer	
uns	boolean	

31.2.415 dnp3.request (object)

Name	Type	Description
application	object	
control	object	
dst	integer	
src	integer	
type	string	

31.2.416 dnp3.request.control (object)

Name	Type	Description
dir	boolean	
fcb	boolean	
fcv	boolean	
function_code	integer	
pri	boolean	

31.2.417 dnp3.request.application (object)

Name	Type	Description
complete	boolean	
control	object	
function_code	integer	
objects	array of objects	

31.2.418 dnp3.request.application.objects (array of objects)

Name	Type	Description
count	integer	
group	integer	
points	array of objects	
prefix_code	integer	
qualifier	integer	
range_code	integer	
start	integer	
stop	integer	
variation	integer	

31.2.419 dnp3.request.application.control (object)

Name	Type	Description
con	boolean	
fin	boolean	
fir	boolean	
sequence	integer	
uns	boolean	

31.2.420 dnp3.iin (object)

Name	Type	Description
indicators	array of strings	

31.2.421 dnp3.control (object)

Name	Type	Description
dir	boolean	
fcb	boolean	
fcv	boolean	
function_code	integer	
pri	boolean	

31.2.422 dnp3.application (object)

Name	Type	Description
complete	boolean	
control	object	
function_code	integer	
objects	array of objects	

31.2.423 dnp3.application.objects (array of objects)

Name	Type	Description
count	integer	
group	integer	
points	array of objects	
prefix_code	integer	
qualifier	integer	
range_code	integer	
start	integer	
stop	integer	
variation	integer	

31.2.424 dnp3.application.control (object)

Name	Type	Description
con	boolean	
fin	boolean	
fir	boolean	
sequence	integer	
uns	boolean	

31.2.425 dhcp (object)

Name	Type	Description
assigned_ip	string	
client_id	string	
client_ip	string	
client_mac	string	
dhcp_type	string	
dns_servers	array of strings	
hostname	string	
id	integer	
lease_time	integer	
next_server_ip	string	
params	array of strings	
rebinding_time	integer	
relay_ip	string	
renewal_time	integer	
requested_ip	string	
routers	array of strings	
subnet_mask	string	
type	string	
vendor_class_identifier	string	

31.2.426 dcerpc (object)

Name	Type	Description
activityuuid	string	
call_id	integer	
interfaces	array of objects	
req	object	
request	string	
res	object	
response	string	
rpc_version	string	
seqnum	integer	

31.2.427 dcerpc.res (object)

Name	Type	Description
frag_cnt	integer	
stub_data_size	integer	

31.2.428 dcerpc.req (object)

Name	Type	Description
frag_cnt	integer	
opnum	integer	
stub_data_size	integer	

31.2.429 dcerpc.interfaces (array of objects)

Name	Type	Description
ack_result	integer	
procedure	string	DCERPC Procedure name corresponding to the interface UUID and Opnum
service	string	DCERPC Service name corresponding to the interface UUID
uuid	string	
version	string	

31.2.430 bittorrent_dht (object)

Name	Type	Description
client_version	string	
error	object	
request	object	
request_type	string	
response	object	
transaction_id	string	

31.2.431 bittorrent_dht.response (object)

Name	Type	Description
id	string	
nodes	array of objects	
nodes6	array of objects	
token	string	
values	array of objects	

31.2.432 bittorrent_dht.response.values (array of objects)

Name	Type	Description
ip	string	
port	number	

31.2.433 bittorrent_dht.response.nodes6 (array of objects)

Name	Type	Description
id	string	
ip	string	
port	number	

31.2.434 bittorrent_dht.response.nodes (array of objects)

Name	Type	Description
id	string	
ip	string	
port	number	

31.2.435 bittorrent_dht.request (object)

Name	Type	Description
id	string	
implied_port	integer	
info_hash	string	
port	integer	
target	string	
token	string	

31.2.436 bittorrent_dht.error (object)

Name	Type	Description
msg	string	
num	integer	

31.2.437 arp (object)

Name	Type	Description
dest_ip	string	Logical address of the intended receiver
dest_mac	string	Physical address of the intended receiver
hw_type	string	Network link protocol type
opcode	string	Specifies the operation that the sender is performing
proto_type	string	Internetwork protocol for which the ARP request is intended
src_ip	string	Logical address of the sender
src_mac	string	Physical address of the sender

31.2.438 anomaly (object)

Name	Type	Description
app_proto	string	
code	integer	
event	string	
layer	string	
type	string	

31.2.439 alert (object)

Name	Type	Description
action	string	
category	string	
context	object	Extra context data created by keywords such as dataset with JSON
engine	string	Engine that generated the alert in firewall mode: fw for firewall rules, td for threat detect rules.
gid	integer	
metadata	object	
references	array of strings	
rev	integer	
rule	string	
severity	integer	
signature	string	
signature_id	integer	
source	object	
target	object	
tenant_id	integer	
xff	string	

31.2.440 alert.target (object)

Name	Type	Description
ip	string	
port	integer	

31.2.441 alert.source (object)

Name	Type	Description
ip	string	
port	integer	

31.2.442 alert.metadata (object)

Name	Type	Description
affected_product	array of strings	
attack_target	array of strings	
created_at	array of strings	
deployment	array of strings	
former_category	array of strings	
malware_family	array of strings	
policy	array of strings	
signature_severity	array of strings	
tag	array of strings	
updated_at	array of strings	

BIBLIOGRAPHY

[llvm] Default LLVM clang-format Style

[clang9] Requires clang 9

[clang10] Requires clang 10

[clang11] Requires clang 11

[breakbeforebraces] BreakBeforeBraces: Mozilla is closest, but does not split empty functions/structs

INDEX

Symbols

-D
    command line option, 37, 635
-F
    command line option, 37, 635
-S
    command line option, 36, 635
-T
    command line option, 35, 633
-U
    command line option, 39, 637
-V
    command line option, 35, 633
--af-packet
    command line option, 36, 634
--af-xdp
    command line option, 36, 635
--build-info
    command line option, 38, 636
--command
    command line option, 639
--dag
    command line option, 39, 637
--disable-detection
    command line option, 37, 636
--disable-gccmarch-native
    command line option, 9
--disable-hashing
    command line option, 37, 636
--dump-config
    command line option, 38, 636
--dump-features
    command line option, 38, 636
--enable-dpdk
    command line option, 9
--enable-geoip
    command line option, 9
--engine-analysis
    command line option, 38, 637
--erf-in
    command line option, 39, 637
--fatal-unittests
    command line option, 39, 638
--group
    command line option, 37, 635
--help
    command line option, 639
--include
    command line option, 35, 633
--init-errors-fatal
    command line option, 37, 635
--list-app-layer-protos
    command line option, 38, 636
--list-keywords
    command line option, 38, 636
--list-runmodes
    command line option, 38, 636
--list-unittests
    command line option, 39, 637
--localstatedir
    command line option, 9
--napatech
    command line option, 39, 637
--netmap
    command line option, 39, 637
--pcap
    command line option, 36, 634
--pcap-buffer-size
    command line option, 38, 637
--pcap-file-buffer-size
    command line option, 36, 634
--pcap-file-continuous
    command line option, 35, 634
--pcap-file-delete
    command line option, 36, 634
--pcap-file-recursive
    command line option, 36, 634
--pfring
    command line option, 39, 637
--pfring-cluster-id
    command line option, 39, 637
--pfring-cluster-type
    command line option, 39, 637
--pidfile

    command line option, 37, 635
--plugin
    command line option, 38, 636
--prefix
    command line option, 9
--reject-dev
    command line option, 38, 637
--runmode
    command line option, 37, 635
--set
    command line option, 38, 636
--simulate-ips
    command line option, 39, 637
--strict-rule-keywords
    command line option, 37, 636
--sysconfdir
    command line option, 9
--unittest-filter
    command line option, 39, 637
--unittests-coverage
    command line option, 39, 638
--unix-socket
    command line option, 38, 637
--user
    command line option, 37, 635
--verbose
    command line option, 639
-c
    command line option, 35, 633, 639
-d
    command line option, 39, 637
-h
    command line option, 35, 633, 639, 642
-i
    command line option, 36, 634
-k
    command line option, 37, 635
-l
    command line option, 37, 635
-q
    command line option, 36, 635
-r
    command line option, 35, 634
-s
    command line option, 36, 635
-u
    command line option, 39, 637
-v
    command line option, 35, 633, 639

A

application_header, 512

B

built-in function
    bytevars.get(), 509
    bytevars.map(), 509
    dnp3.get_tx(), 511
    thread_info(), 545
bytevars.get()
    built-in function, 509
bytevars.map()
    built-in function, 509

C

command line option
    -D, 37, 635
    -F, 37, 635
    -S, 36, 635
    -T, 35, 633
    -U, 39, 637
    -V, 35, 633
    --af-packet, 36, 634
    --af-xdp, 36, 635
    --build-info, 38, 636
    --command, 639
    --dag, 39, 637
    --disable-detection, 37, 636
    --disable-gccmarch-native, 9
    --disable-hashing, 37, 636
    --dump-config, 38, 636
    --dump-features, 38, 636
    --enable-dpdk, 9
    --enable-geoip, 9
    --engine-analysis, 38, 637
    --erf-in, 39, 637
    --fatal-unittests, 39, 638
    --group, 37, 635
    --help, 639
    --include, 35, 633
    --init-errors-fatal, 37, 635
    --list-app-layer-protos, 38, 636
    --list-keywords, 38, 636
    --list-runmodes, 38, 636
    --list-unittests, 39, 637
    --localstatedir, 9
    --napatech, 39, 637
    --netmap, 39, 637
    --pcap, 36, 634
    --pcap-buffer-size, 38, 637
    --pcap-file-buffer-size, 36, 634
    --pcap-file-continuous, 35, 634
    --pcap-file-delete, 36, 634
    --pcap-file-recursive, 36, 634
    --pfring, 39, 637
    --pfring-cluster-id, 39, 637
    --pfring-cluster-type, 39, 637

--pidfile, 37, 635
--plugin, 38, 636
--prefix, 9
--reject-dev, 38, 637
--runmode, 37, 635
--set, 38, 636
--simulate-ips, 39, 637
--strict-rule-keywords, 37, 636
--sysconfdir, 9
--unittest-filter, 39, 637
--unittests-coverage, 39, 638
--unix-socket, 38, 637
--user, 37, 635
--verbose, 639
-c, 35, 633, 639
-d, 39, 637
-h, 35, 633, 639, 642
-i, 36, 634
-k, 37, 635
-l, 37, 635
-q, 36, 635
-r, 35, 634
-s, 36, 635
-u, 39, 637
-v, 35, 633, 639
complete, 511

D

dnp3.get_tx()
built-in function, 511
done, 511

G

group, 512

I

index, 512
indicators, 512
is_request, 511

L

link_header, 511

O

objects, 512

P

points, 512

R

request, 511
response, 511

T

thread_info()
built-in function, 545
transport_header, 512
tx_num, 511

V

variation, 512