Snort 3 Rule Writing Guide

by the Cisco Talos Detection Response Team

Introduction

Snort 3 brings many new features, improvements, and detection capabilities to the Snort engine, as well as updates to the Snort rule language syntax that improve the rule-writing process. This *Snort 3 Rule Writing Guide* elucidates all these new enhancements and contains detailed documentation for all the different rule options available in Snort 3, in a format that is easy to understand and use.

This manual is meant for new and experienced Snort rule-writers alike, and it is intended to *supplement* the documentation provided in the official Snort 3 repository, focusing primarily on the rule-writing process. Each rule option has its own page that describes its functionality, its specific syntax, as well as a few examples to show how the given option might be used in a Snort rule. Additionally, the manual also includes a few overview pages that cover the basic steps needed to help get Snort 3 up and running. We expect to expand on this section in the future.

As Snort 3 continues to evolve, this manual will too. We will provide timely updates to the manual whenever necessary, to keep the greater Snort community abreast of any recent changes.

Getting Started with Snort 3

The section will walk you through the basics of building and running Snort 3, and also help get you started with all things Snort 3. Specifically, this section contains information on building Snort 3, running Snort 3 for the first time, configuring Snort's detection engines, inspecting network traffic with Snort, extending Snort's functionality with "tweaks" and "scripts", and lastly tracing Snort.

While not an exhaustive guide on installing and running Snort 3, this section does provide a foundation that can be built upon to learn and take advantage of the more advanced features that Snort 3 has to offer.

Snort 3 Installation

Required Packages

The very first thing to do is make sure all necessary dependencies are installed. The following is a list of required packages:

- cmake to build from source
- The Snort 3 libdaq for packet IO
- dnet for network utility functions
- flex >= 2.6.0 for JavaScript syntax parsing
- g++ >= 5 or other C++14 compiler
- hwloc for CPU affinity management
- LuaJIT for configuration and scripting
- OpenSSL for SHA and MD5 file signatures, the protected_content rule option, and SSL service detection
- pcap for tcpdump style logging
- pcre for regular expression pattern matching
- pkgconfig to locate build dependencies
- zlib for decompression

Information on where to download each of these *required* packages can be found here: https://github.com/snort3/snort3/blob/master/doc/user/tutorial.txt#L6.

Optional Packages

There are also a few optional packages that can be installed to take advantage of some of Snort's optional features. These include:

- asciidoc to build the HTML manual
- cpputest to run additional unit tests with make check
- dblatex to build the PDF manual included with Snort 3 installs
- flatbuffers for enabling the flatbuffers serialization format
- hyperscan >= 4.4.0 to build the new regex and sd_pattern rule options and hyperscan search engine.
- iconv for converting UTF16-LE filenames to UTF8 (usually included in glibc)
- libunwind to attempt to dump a somewhat readable backtrace when a fatal signal is received
- lzma >= 5.1.2 for decompression of SWF and PDF files

- safec >= 3.5 for runtime bounds checks on certain legacy C-library calls
- source-highlight to generate the dev guide
- w3m from to build the plain text manual
- uuid from uuid-dev package for unique identifiers

Information on where to download each of these *optional* packages can be found here: https://github.com/snort3/snort3/blob/master/doc/user/tutorial.txt#L36.

Installing LibDAQ

We now need to install the Snort 3 LibDAQ, which provides an abstraction layer for communicating with a data source (such as a network interface).

If you have LibDAQ already installed for Snort 2 and want to install a DAQ just for Snort 3, or if you want to install LibDAQ in a custom location, you can change the DAQ install location with the `--prefix` option when configuring: `./configure --prefix=/usr/local/lib/daq_s3`.

To show this in action, first clone the LibDAQ repository from GitHub:

```
$ git clone https://github.com/snort3/libdaq.git
```

Then, run the following commands to generate the configure script, configure with a specified prefix, build, and lastly install:

```
$ cd libdaq
$ ./bootstrap
$ ./configure --prefix=/usr/local/lib/daq_s3
$ make install
```

After installing libdaq, you must then run `ldconfig` to configure your system's dynamic linker run-time bindings. However, if you have installed the DAQ in a nonstandard location, you'll first need to tell your system where to find the new shared libraries. One common solution is to create a file in the `/etc/ld.so.conf.d/` directory that points to where those libraries are located:

```
$ cat /etc/ld.so.conf.d/libdaq3.conf
/usr/local/lib/daq_s3/lib/
```

Once ready you may proceed with the `ldconfig` command to configure the run-time bindings:

```
$ sudo ldconfig
```

Building Snort

After all dependencies have been installed, it is time to build Snort.

To do this, first clone the Snort 3 repository:

```
$ git clone https://github.com/snort3/snort3.git
```

You can choose to install Snort in the system-default directories, or you can specify to install it in some other directory with the `--prefix=<path>` command line argument.

```
$ export my_path=/path/to/snorty
$ mkdir -p $my_path
$ cd snort3
$ ./configure_cmake.sh --prefix=$my_path
```

Additionally, if the LibDAQ has been installed in a non-standard or custom location, then you must include the `--with-daq-libraries` and `--with-daq-includes` arguments and set them accordingly.

```
$ ./configure_cmake.sh --prefix=$my_path \
                       --with-daq-includes=/usr/local/lib/daq_s3/include/ \
                       --with-daq-libraries=/usr/local/lib/daq_s3/lib/
```

There are many more CMake configuration options to choose from (like enabling debug mode, for example), and the full list of options can be seen by running the following command:

```
$ ./configure_cmake.sh --help
```

Once you've configured CMake to your liking and the build files are ready to go, it's time to compile and install Snort. To do this, `cd` to the newly-created `build` directory, and then compile and install:

```
$ cd build
$ make -j $(nproc)
$ make install
```

If all goes well, run `snort -V` at the command line to verify successful installation.

```
$ $my_path/bin/snort -V

   ,,_     -*> Snort++ <*-
  o"  )~   Version 3.1.36.0
   ''''    By Martin Roesch & The Snort Team
           http://snort.org/contact#team
           Copyright (C) 2014-2022 Cisco and/or its affiliates. All rights
reserved.
           Copyright (C) 1998-2013 Sourcefire, Inc., et al.
           Using DAQ version 3.0.6
           Using LuaJIT version 2.1.0-beta3
           Using OpenSSL 1.1.1q  5 Jul 2022
           Using libpcap version 1.10.1 (with TPACKET_V3)
           Using PCRE version 8.45 2021-06-15
           Using ZLIB version 1.2.12
           Using LZMA version 5.2.6
```

Lastly, verify that your installation has the appropriate DAQs available to it:

```
$ $my_path/bin/snort --daq-list
Available DAQ modules:
afpacket(v7): live inline multi unpriv
 Variables:
  buffer_size_mb <arg> - Packet buffer space to allocate in megabytes
  debug - Enable debugging output to stdout
  fanout_type <arg> - Fanout loadbalancing method
  fanout_flag <arg> - Fanout loadbalancing option
  use_tx_ring - Use memory-mapped TX ring
…
```

If, however, you get `No available DAQ modules (try adding directories with --daq-dir).`, then you will need to specify `--daq-dir` as the error points out:

```
$ $my_path/bin/snort --daq-dir /usr/local/lib/daq_s3/lib/daq --daq-list
```

If that's the case, you can create an "alias" for your Snort command so that you don't have to specify `--daq-dir` each time you want to invoke Snort:

```
alias snort='/path/to/bin/snort --daq-dir /usr/local/lib/daq_s3/lib/daq'
```

Using Snort

Snort is an incredibly powerful multipurpose engine. In this section, we'll go over the basics of using Snort on the command line, briefly discuss how to set and tweak one's configuration, and lastly go over how to use Snort to detect and prevent attacks.

Command Line Basics

Running Snort on the command line is easy, but the number of arguments available might be overwhelming at first. So let's start with the basics.

All Snort commands start with `snort`, and running this command by itself will show basic usage instructions:

```
$ snort
usage:
    snort -?: list options
    snort -V: output version
    snort --help: help summary
    snort [-options] -c conf [-T]: validate conf
    snort [-options] -c conf -i iface: process live
    snort [-options] -c conf -r pcap: process readback
```

Fortunately, Snort 3 provides a very robust set of help commands that detail just about every aspect of the engine.

To see the main help "directory", run the following command:

```
$ snort --help

Snort has several options to get more help:

-? list command line options (same as --help)
--help this overview of help
--help-commands [<module prefix>] output matching commands
--help-config [<module prefix>] output matching config options
--help-counts [<module prefix>] output matching peg counts
--help-limits print the int upper bounds denoted by max*
--help-module <module> output description of given module
--help-modules list all available modules with brief help
…
```

As we can see from the output, Snort contains separate help pages for the different parts of the Snort engine, and these subpages can be used to get granular help information about a particular component. Shown below are a few of these help subpages.

Listing all available Snort modules:

```
$ snort --list-modules
```

Getting help on a specific Snort module:

```
$ snort --help-module http_inspect
```

Getting help on a specific rule option module:

```
$ snort --help-module http_uri
```

Listing command line options available:

```
$ snort -?
```

Getting help on the "-A" command line option:

```
$ snort --help-options A
```

Getting help with a specific configuration setting:

```
$ snort --help-config | grep http
```

Outputting help on "rule" options in an AsciiDoc format:

```
$ snort --markup --help-options rule
```

Reading Traffic

Snort is at its best when it has network traffic to inspect, and Snort can perform network inspection in a few different ways. This includes (but is not limited to) reading traffic directly from a packet capture, running passively on a network interface to sniff traffic, and testing Snort's inline injection capabilities locally. Before we can dive into that, we first need to go over how to provide Snort with traffic to inspect.

Specifying LibDAQ directory

First things first, it's important to make sure the Snort 3 install knows where to find the appropriate LibDAQ that we installed earlier. LibDAQ is the "Data Acquisition Library", and at a high-level, it's an abstraction layer used by "modules" to communicate with both hardware and software network data sources. For example, one DAQ module installed by default is the `pcap` module that is built around around the libpcap library to listen on network interfaces or read from `.pcap` files.

If users have Snort both 2 and Snort 3 installed on a single system, then that means they also have two LibDAQ versions installed, one for Snort 2 and another Snort 3. Therefore when using Snort 3 on the command line, users must explicitly set the `--daq-dir` option to tell Snort where to find the appropriate modules.

For example, if the Snort 3 LibDAQ is installed in `/usr/local/lib/daq_s3/`, then users will want to set `--daq-dir` to `/usr/local/lib/daq_s3/lib/daq`:

```
$ snort --daq-dir /usr/local/lib/daq_s3/lib/daq
```

Users can run `snort --daq-list` to see which DAQ modules are available for use.

Reading Packet Captures

The simplest way to see Snort in action is to run it against a packet capture file. Simply pass in a pcap file name to the `-r` option on the command line, and Snort will process it accordingly:

```
$ snort -r get.pcap
```

If successful, Snort will print out basic information about the pcap file that was just read, including details such as the number of packets and the protocols detected.

Users can also run Snort against an entire directory of pcaps with the `--pcap-dir` option. If that directory contains files other pcaps, then the `--pcap-filter` option can be used to tell Snort *which* of those files to process. For instance:

```
$ snort --pcap-dir /path/to/pcap/dir --pcap-filter '*.pcap'
```

Running Snort on Network Interfaces

Snort can also listen on active network interfaces, and specifying it to do so is done with the `-i` option followed by the interface names to run on. The following command, for example, runs Snort on the `eth0` network interface:

```
$ snort -i eth0
```

Modes of operation

With certain DAQ modules, Snort is able to utilize two different modes of operation: **passive** and **inline**. Passive mode gives Snort the ability to observe and detect traffic on a network interface, but it prevents outright *blocking* of traffic. Inline mode on the other hand, *does* give Snort the ability to block traffic if a particular packet warrants such an event.

Snort will infer the particular mode of operation based on the options used at the command line. For example, reading from a pcap file with the `-r` option or listening on an interface with `-i` will cause Snort to run in passive mode by default. If the DAQ supports inline, however, then users can specify the `-Q` flag to run Snort inline.

One DAQ module that supports inline mode is `afpacket`, which is a module that gives Snort access to packets received on Linux network devices.

Using the `afpacket` module inline requires specifying a pair of network interfaces in the `-i` command line option, where each pair is two interface names separated by a colon character.

```
$ snort -Q --daq afpacket -i "eth0:eth1"
```

Configuration

Once we've got Snort set up to process traffic, it's now time to tell Snort *how* to process traffic, and this is done through configuration. Snort configuration handles things like the setting of global variables, the different modules to enable or disable, performance settings, event logging policies, the paths to specific rules files to enable, and much more.

Snort 3 configuration is now all done in Lua, and these configuration options can be supplied to Snort in three different ways: via the command line, with a single Lua configuration file, or with multiple Lua configuration files.

Configuration Files

There are **many** different configuration options that can be tuned in Snort, but luckily open-source Snort 3 comes with a set of standard configuration files that help get Snort users up and running quickly. These default files are located in the `lua/` directory, and the `snort.lua` and `snort_defaults.lua` files present there make up what is considered to be the the base configuration. This default config is an excellent template to build upon, and it can be plugged right into Snort for immediate use.

Module configuration

A big part of one's configuration is the enabling and tuning of Snort "modules", which at a high level control how Snort processes and handles network traffic. Snort contains modules to decipher raw packets, perform traffic normalization, determine whether or not a specific action should be taken against a particular packet, and also control how events should be logged. Snort features eight different types of modules:

- Basic Modules -> handle configuration for basic traffic and rule processing
- Codec Modules -> decode protocols and perform anomaly detection
- Inspector Modules -> analyze and process protocols
- IPS Action Modules -> enable custom actions that can be performed when an event occurs
- IPS Option Modules -> options set in Snort rules to set the detection parameters
- Search Engine -> perform pattern matching against packet data to determine which rules to evaluate
- SO Rule Modules -> perform detection not attainable with the existing IPS options
- Logger Modules -> control the output of events and packet data

A list and brief description of all Snort 3 modules can be seen with the `--help-modules` command:

```
$ snort --help-modules
```

Modules are enabled and configured in a configuration as Lua table literals. For example, the `stream_tcp` inspector module, which handles TCP flow tracking and stream normalization and reassembly, can enabled like so:

```
stream_tcp = { }
```

If a module is initialized as an *empty* table, then that means the module is using its "default" settings. We can view these defaults with the `--help-config` argument:

```
$ snort --help-config stream_tcp
int stream_tcp.flush_factor = 0: flush upon seeing a drop in segment size after
given number of non-decreasing segments { 0:65535 }
int stream_tcp.max_window = 0: maximum allowed TCP window { 0:1073725440 }
int stream_tcp.overlap_limit = 0: maximum number of allowed overlapping segments
per session { 0:max32 }
int stream_tcp.max_pdu = 16384: maximum reassembled PDU size { 1460:32768 }
bool stream_tcp.no_ack = false: received data is implicitly acked immediately
enum stream_tcp.policy = 'bsd': determines operating system characteristics like
reassembly { 'first' | 'last' | 'linux' | 'old_linux' | 'bsd' | 'macos' |
'solaris' | 'irix' | 'hpux11' | 'hpux10' | 'windows' | 'win_2003' | 'vista' |
'proxy' }
bool stream_tcp.reassemble_async = true: queue data for reassembly before
traffic is seen in both directions
int stream_tcp.require_3whs = -1: don't track midstream sessions after given
seconds from start up; -1 tracks all { -1:max31 }
bool stream_tcp.show_rebuilt_packets = false: enable cmg like output of
reassembled packets
int stream_tcp.queue_limit.max_bytes = 4194304: don't queue more than given
bytes per session and direction, 0 = unlimited { 0:max32 }
int stream_tcp.queue_limit.max_segments = 3072: don't queue more than given
segments per session and direction, 0 = unlimited { 0:max32 }
int stream_tcp.small_segments.count = 0: number of consecutive (in the received
order) TCP small segments considered to be excessive (129:12) { 0:2048 }
int stream_tcp.small_segments.maximum_size = 0: minimum bytes for a TCP segment
not to be considered small (129:12) { 0:2048 }
int stream_tcp.session_timeout = 180: session tracking timeout { 1:max31 }
bool stream_tcp.track_only = false: disable reassembly if true
```

This command outputs the different settings for the given module, a description of each setting, and valid values for each one. The `max_pdu` setting, for example, can be set to an integer between 1460 and 32768 (inclusive).

Entries in Lua table literals are effectively just key-value pairs. If we wanted to set `stream_tcp.max_pdu` to its max setting, we would simply add a table entry like so:

```
stream_tcp =
{
    max_pdu = 32768
}
```

Other entries follow the same format and are separated by commas.

The default `snort.lua` configuration file enables and configures many of the core modules relied upon by Snort, and users are encouraged to go through that file and learn about the different ones using the `--help-module` and `--help-config` Snort commands.

Passing configuration files to Snort

Snort doesn't look for a specific configuration file by default, but you can pass one to it very easily with the `-c` argument:

```
$ snort -c $my_path/lua/snort.lua
```

This command simply validates the supplied configuration file, and if everything is in order, the output will include a message that says "Snort successfully validated the configuration".

Tuning Lua Configurations via the Command Line

Sometimes rule writers will want to experiment with a specific configuration to see how it might affect detection. Fortunately, Snort 3 provides the ability to run one-off custom Lua configurations directly from the command line using the `--lua` flag followed by a string enclosed in quotes containing the specific Lua configuration (or configurations) to set.

For example, the following `--lua` command sets the `enable_builtin_rules` boolean from the `ips` module to `true`:

```
$ snort -c $my_path/lua/snort.lua -R local.rules \
        --lua 'ips.enable_builtin_rules = true'
```

Note that the above `--lua` argument uses *dot notation* to *extend* any existing `ips` configuration present in one's Snort configuration file. However, users can also *override* a given module's configuration using *curly braces* instead of using dot notation.

The following argument, for example, overrides any existing `ips` configuration with the one specified:

```
$ snort -c $my_path/lua/snort.lua -R local.rules \
        --lua 'ips = { enable_builtin_rules = true }'
```

This above example sets `ips` back to its default settings, but then also sets `enable_builtin_rules` to `true`.

Snort2Lua

Converting configuration files

For those that are coming from Snort 2 and have a working 2.x configuration file, building Snort 3 also creates a binary named `snort2lua`, which can take one's old Snort 2 configuration and output one that can be plugged into Snort 3.

Running this command is as easy as invoking the binary followed by a `-c` option that points to the Snort 2 configuration file:

```
$ snort2lua -c snort.conf
```

If any errors occur during the conversion, they will be placed in a `snort.rej` file in the current working directory. Once all errors have been taken care of, `snort2lua` will output a `snort.lua` file that can then be passed directly to Snort 3.

Snort Rules

At its core, Snort is an intrusion detection system (IDS) and an intrusion prevention system (IPS), which means that it has the capability to detect intrusions on a network, and also prevent them. A configuration tells Snort how to process network traffic. It is the *rules* that determine whether Snort acts on a particular packet.

Snort rules can be placed directly in one's Lua configuration file(s) via the `ips` module, but for the most part they will live in distinct `.rules` files that get "included". For example, say we had a `malware.rules` file in the same directory as our Lua configuration file. We could "include" that rules file like so:

```
ips = { include = 'malware.rules' }
```

If users want to include multiple `.rules` files, then they can do so like:

```
ips =
{
    rules = [[
        include /path/to/rulesfile1.rules
        include /path/to/rulesfile2.rules
        …
    ]]
}
```

Alternatively, a single rules file or a path to a rules directory can be passed directly to Snort on the command line. This is done either with the `-R` option for a single rules file or the `--rule-path` option to pass in a whole directory of rules files. This is convenient for when you need to verify or troubleshoot a rule or rules against a pcap.

For example, the below command will run all the rules present in `malware.rules` against the traffic in `bad.pcap`:

```
$ snort -c $my_path/lua/snort.lua -R malware.rules -r bad.pcap
```

Generating Alerts

The above command by default will output various statistics about the particular run. These include details about any identified applications, any detection events, types of services detected, and much more. The detection events will show *how many* alerts fired on the provided traffic, but sometimes we want to know more than that.

Snort provides a few different "alert mode" options that can be set on the command line to tweak the way alerts are displayed. These modes include `cmg` which displays alerts alongside a hexdump of the alerting packet(s), as well as a few different `alert_*` modes shown below:

```
$ snort --help-modules | grep alert
alert_csv (logger): output event in csv format
alert_fast (logger): output event with brief text format
alert_full (logger): output event with full packet dump
alert_json (logger): output event in json format
alert_syslog (logger): output event to syslog
alert_talos (logger): output event in Talos alert format
alert_unixsock (logger): output event over unix socket
alerts (basic): configure alerts
```

These modes are set with the `-A` option followed by the desired alert mode, and we can focus solely on the alerts by also including the `-q` (quiet) flag. The `cmg` alert mode, for example, will look something like:

```
$ snort -q -c $my_path/lua/snort.lua -q -r get.pcap -R local.rules -A cmg
10/14-14:59:14.186063 [**] [1:0:0] "GET request" [**] [Classification: Web
Application Attack] [Priority: 1] {TCP} 10.1.2.3:50284 -> 10.9.8.7:80

http_inspect.http_method[3]:
- - - - - - - - - - - -  - - - - - - - - - - - - - -  - - - - - - - - - -
47 45 54                                            GET
- - - - - - - - - - - -  - - - - - - - - - - - - - -  - - - - - - - - - -

http_inspect.http_version[8]:
- - - - - - - - - - - -  - - - - - - - - - - - - - -  - - - - - - - - - -
48 54 54 50 2F 31 2E 31                             HTTP/1.1
- - - - - - - - - - - -  - - - - - - - - - - - - - -  - - - - - - - - - -

http_inspect.http_uri[6]:
- - - - - - - - - - - -  - - - - - - - - - - - - - -  - - - - - - - - - -
2F 68 65 6C 6C 6F                                   /hello
- - - - - - - - - - - -  - - - - - - - - - - - - - -  - - - - - - - - - -

http_inspect.http_header[78]:
- - - - - - - - - - - -  - - - - - - - - - - - - - -  - - - - - - - - - -
48 6F 73 74 3A 20 61 62  63 69 70 2D 68 6F 73 74  Host: ab cip-host
2E 6C 6F 63 61 6C 0D 0A  55 73 65 72 2D 41 67 65  .local.. User-Age
6E 74 3A 20 61 62 63 69  70 0D 0A 41 63 63 65 70  nt: abci p..Accep
74 2D 4C 61 6E 67 75 61  67 65 3A 20 65 6E 2D 75  t-Langua ge: en-u
73 0D 0A 41 63 63 65 70  74 3A 20 2A 2F 2A        s..Accep t: */*
- - - - - - - - - - - -  - - - - - - - - - - - - - -  - - - - - - - - - -
```

The `alert_talos` is another useful mode that displays alerts in a format that is simple and easy-to-understand.

```
$ snort -q -c $my_path/lua/snort.lua -q -r get.pcap -R local.rules -A
alert_talos

##### get.pcap #####
        [1:0:0] GET request (alerts: 1)
#####
```

Lastly, Some of the `alert_*` modes are customizable. `alert_csv` for example allows for customization of the different "fields" that can be outputted. The following example demonstrates a custom CSV alert configuration using the `--lua` command line flag:

```
$ snort -q -c $my_path/lua/snort/lua -r cmd_injection.pcap -R local.rules --lua
'alert_csv = { fields = "action pkt_num gid sid rev msg service src_addr
src_port dst_addr dst_port", separator = "," }'
would_block,5,1,1000000,0,"Command injection
detected",http,10.1.2.3,50284,10.9.8.7,80
would_block,6,1,1000000,0,"Command injection
detected",http,10.1.2.3,50284,10.9.8.7,80
```

Testing Rules Inline

To protect networks, it's also important to make sure that our rules are blocking attacks appropriately, and the `dump` DAQ enables us to do just that.

Specifying the `-Q` option to enable inline mode and then setting the `--daq` to `dump` will "dump" the traffic that would've been passed through, emulating a real inline operation. The resulting traffic will be dumped, by default, to a file named `inline-out.pcap`:

```
$ snort3 -Q --daq dump -q -r get.pcap -R local.rules
```

In the above example, if the `local.rules` file contains a `block` rule that fires on some traffic in the `get.pcap` file, then the resulting `inline-out.pcap` file will contain only the traffic that was not blocked. We can use this functionality to test that our rules are preventing the actual attack packet(s) from getting through.

Converting Snort 2 Rules to Snort 3

Lastly, just like with configuration files, `snort2lua` can also be used to convert old Snort 2 rules to Snort 3 ones. Pass the Snort 2 rules file to the `-c` option and then provide a filename for the new Snort 3 rules file to the `-r` option:

```
$ snort2lua -c in.rules -r out.rules
```

Note that if any errors occur during the conversion, `snort2lua` will output a `snort.rej` file that explains what went wrong:

```
$ snort2lua -c 2.rules -r 3.rules
ERROR: 1 errors occurred while converting
ERROR: see snort.rej for details
```

Snort Wizard and Binder

Snort 3 features two new components to help determine the most likely service of a given flow of traffic and then direct that traffic to the right "service inspector": the **wizard** and **binder**. These next sections go over each of these components in detail, starting with the wizard.

The Wizard

The wizard uses what are called **hexes**, **spells**, and **curses** to help determine the service present on a given flow. All three look at different parts of the traffic to help identify the service being used. However, the wizard simply helps with service identification; it's the job of the Application Identification inspector to make the definitive determination. Therefore, the wizard's main goal is to quickly identify the *most likely* service so that it can hand it off to the appropriate inspector to do the rest.

Hexes

Hexes are binary-based patterns that are used to detect binary protocols such as `ssl` and `dnp3`. Hexes are configured in Lua as an array of tables, where each table contains keys to define how the hexes should operate. Those keys include the following:

- `service` -> the name of the service that would be assigned
- `proto` -> the protocol to scan (e.g., `tcp`)
- `client_first` -> boolean flag that indicates if the client is the initiator of the data transfer
- `to_server` -> list of text patterns to search in the data sent to the client
- `to_client` -> list of text patterns to search in the data sent to the server

Hex patterns contain a series of hexadecimal bytes enclosed `|` characters. Users can also add an arbitrary number of `?` characters to look for any number of hexadecimal digits. For instance, `|05 ?4|` would match `|05 84|` as well as `|05 34|`.

Here's one hex declaration for DNP3 traffic:

```
{
  service = 'dnp3',
  proto = 'tcp',
  client_first = true,
  to_server = { '|05 64|' },
  to_client = { '|05 64|' }
}
```

This hex and other built-in hexes can be found in the `snort_defaults.lua` configuration file here.

Spells

Hexes are binary-based patterns, whereas *spells* are **text-based**, meaning they are best used with text-dominant protocols such as `http`, `smtp`, `sip`, and so forth. Spells are configured in the same manner that hexes are, and they also have an identical set of keys/options.

The text strings used in spells are case-insensitive and white-space sensitive. Spells can also contain wildcard characters, denoted with `*`, to match any number of bytes until a subsequent text pattern.

For example, an SMTP spell definition might look like this:

```
{
  service = 'smtp',
  proto = 'tcp',
  client_first = true,
  to_server = { 'HELO', 'EHLO' },
  to_client = { '220*SMTP', '220*MAIL' }
}
```

The built-in spells can be found in the `snort_defaults.lua` configuration file here.

Curses

Curses are a little different than hexes and spells in that they are built as C++ state machines, as opposed to being defined in a Snort configuration. Curses exist for services where identification requires more than just a few string or hexadecimal patterns.

Snort currently has algorithms for five services/protocols:

- DCE/RPC over UDP
- DCE/RPC over TCP
- DCE/RPC over SMB
- SSLv2
- S7CommPlus
- MMS

The code for these algorithms can be found in src/service_inspectors/wizard/curses.cc.

Like hexes and spells, curses also get enabled in a Snort configuration. The following line in `snort_defaults.lua` enables the six default curse algorithms present in `curses.cc`:

```
curses = {'dce_udp', 'dce_tcp', 'dce_smb', 'mms', 's7commplus', 'sslv2'}
```

One can view the available curses with the following help command:

```
$ snort --help-config wizard | grep curses
multi wizard.curses: enable service identification based on internal algorithm {
dce_smb | dce_udp | dce_tcp | mms | s7commplus | sslv2 }
```

The Binder

Snort 3's binder is a feature that directs network traffic to a specific service inspector based on some combination of the traffic's detected service, the ports and network ranges involved, and the network protocol being used. Users can create configurations with those options to tell Snort which service inspector to apply to a given network flow.

These configurations are written as an array of Lua tables and placed in one's Snort configuration file(s). These binder entries will typically contain two nested tables, `when` and `use`. The `when` table controls the criteria for invoking the action that's specified in the `use` table.

Here are some examples of a few different binder entries:

```
binder =
{
  -- allow all tcp port 22:
  -- (similar to Snort 2 config ignore_ports)
  { when = { proto = 'tcp', ports = '22' }, use = { action = 'allow' } },

  -- select a config file by vlan
  -- (similar to Snort 2 config binding by vlan)
  { when = { vlans = '1024' }, use = { file = 'vlan.lua' } },

  -- use the DNS service inspector when the server port is TCP port 53
  { when = { proto = 'tcp', ports = '53', role='server' },  use = { type = 'dns'
} },

  -- use a non-default HTTP inspector for port 8080:
  -- (similar to a Snort 2 targeted preprocessor config)
  { when = { nets = '192.168.0.0/16', proto = 'tcp', ports = '8080' },
    use = { name = 'alt_http', type = 'http_inspect' } },

  -- use the default inspectors:
  -- (similar to a Snort 2 default preprocessor config)
  { when = { proto = 'tcp' }, use = { type = 'stream_tcp' } },
  { when = { service = 'http' }, use = { type = 'http_inspect' } },

  -- figure out which inspector to run automatically:
  { use = { type = 'wizard' } }
}
```

One can view all the available binder table parameters with the following Snort command:

```
$ snort --help-module binder
```

Snort Tweaks and Scripts

Tweaks

Snort also provides the ability to add additional tunings to configurations with the `--tweaks` option. This can be used, for example, to employ one of Snort's various policy files that tweak Snort's detection engine to favor either more performance or more security.

Snort 3 comes with four policy tweak files by default: `max_detect`, `security`, `balanced` and `connectivity`. The `max_detect` policy provides the *most* security whereas the `connectivity` policy prioritizes performance and uptime at the expense of security.

These tweaks are more-or-less just configuration extensions; the `snort.lua` and `snort_defaults.lua` files provide a base policy, and then the `tweaks` allow for a particular set of targeted changes.

To use a tweak, simply specify the `--tweaks` option followed by the name of the tweak file to use. For example, to use the `max_detect` policy, one would run Snort like so:

```
$ snort -c $my_path/lua/snort.lua -R local.rules --tweaks max_detect
```

There also exists a `talos` tweaks option that configures Snort to the way Talos analysts will initially test their own rules:

```
$ snort -c $my_path/lua/snort.lua -R local.rules --tweaks talos
```

You can check out each of these tweaks in the `lua/` directory to see what kinds of changes each one makes.

Scripts

Snort 3 is extensible in that it offers the ability for users to create custom LuaJIT scripts to extend its functionality. Scripts are passed to Snort 3 on the command line with the `--script-path <scripts_path>` argument, and they are then called in Snort rules by specifying the script "name" (declared in the .lua file) as a rule option.

One script we commonly work with is `hexdump.lua`, which prints out packet data at the detection cursor's current location. This is incredibly useful when creating rules because it

helps rule writers determine if and what data is present in a specific buffer or at the current cursor location.

For example, if we were to invoke this `hexdump` script right after an `http_uri;` buffer declaration, Snort will print out all the bytes in the `http_uri` buffer:

```
$ ls scripts/
hexdump.lua

$ cat local.rules
alert http ( \
    msg:"GET request"; \
    http_uri; \
    hexdump; \
    classtype:web-application-attack; \
    sid:1000000; \
)

$ snort --talos --script-path scripts/ -q -r get.pcap -R local.rules
[http_uri] (6 bytes)
00000000  2F 68 65 6C 6C 6F                                 /hello
```

Because this particular script is relative to the previous content match, adding a `content:"/h";` match to the rule after `http_uri;` would result in the following change:

```
$ cat local.rules
alert http (
    msg:"GET request";
    http_uri;
    content:"/h";
    hexdump;
    classtype:web-application-attack;
    sid:1000000;
)

$ snort --talos --script-path scripts/ -q -r get.pcap -R local.rules
[http_uri] (4 bytes)
00000000  65 6C 6C 6F                                       ello
```

Snort Trace Modules

Snort 3 also contains new "trace" modules that enable logging Snort's engine output at a very low level to display things such as rule evaluation tracing, buffer dumping, Application ID (wizard) tracing, and much more.

Snort tracing options are configured in Lua, and they can be placed in a tweak file that is included in the Snort configuration being used, or they can be provided directly on the command line.

We can see all the different trace module options and configurations available via the `--help-module trace` command line option:

```
$ snort --help-module trace

trace


Help: configure trace log messages

Type: basic

Usage: global

Configuration:

int trace.modules.all: enable trace for all modules { 0:255 }
int trace.modules.appid.all: enable all trace options { 0:255 }
int trace.modules.dce_smb.all: enable all trace options { 0:255 }
int trace.modules.dce_udp.all: enable all trace options { 0:255 }
int trace.modules.decode.all: enable all trace options { 0:255 }
int trace.modules.detection.all: enable all trace options { 0:255 }
int trace.modules.detection.detect_engine: enable detection engine trace logging
{ 0:255 }
int trace.modules.detection.rule_eval: enable rule evaluation trace logging {
0:255 }
int trace.modules.detection.buffer: enable buffer trace logging { 0:255 }
int trace.modules.detection.rule_vars: enable rule variables trace logging {
0:255 }
…
```

There are many modules to choose from. These include `trace.modules.detection.buffer`, which is useful for packet buffer dumping, and `trace.modules.wizard`, which is used to show application detection information.

Note: To use all available tracing modules, Snort 3 must be configured with the `--`

`enable-debug-msgs` option.

Using a trace option is done by constructing a `trace.modules` Lua table and including in it the Lua table constructor(s) of the module(s) to enable. For example, the Lua table to enable the `trace.modules.wizard`, `trace.modules.detection.buffer`, and `trace.modules.detection.fp_search` modules is done like so:

```
trace.modules = {
    detection = {
        fp_search = 1,
        buffer = 1,
    },
    wizard = {
        all = 1,
    },
}
```

Setting these values to an integer greater than zero will enable them, and conversely, setting them to zero will disable them.

Once the Lua table has been constructed, users will then include the table declaration either in a tweak file or on the command line with the `--lua` option.

Below is a list of common trace options that might be useful when working with Snort rules:

Option	Result	Lua
`fp_search`	Show `fast_pattern` buffer name on entry	`trace.modules = {detection = {fp_search = 1}}`
`buffer`	Print packet buffer	`trace.modules = {detection = {buffer = 1}}`
`rule_vars`	Show rule variables like `byte_extract`	`trace.modules = {detection = {rule_vars = 1}}`
`rule_eval`	Show rule eval tracing	`trace.modules = {detection = {rule_eval = 1}}`
`wizard`	Show application detection information	`trace.modules = {wizard = { all = 1}}`

More Examples

Here are a few examples of specifying tracing on the command line.

This first example incorporates the `wizard` trace module to display the application detection information:

```
$ snort -c $my_path/lua/snort.lua -q -r get.pcap -R local.rules -A alert_talos \
        --lua 'trace.modules = {wizard = {all = 1}}'
P0:wizard:all:1: c2s streaming search found service http

##### get.pcap #####
        [1:0:0] GET request (alerts: 1)
#####
```

This next example uses multiple trace modules, `wizard`, `detection.fp_search`, and `detection.buffer`, to dump application detection information, the packet buffer(s), and rule `fast_pattern` details.

```
$ snort -q -r get.pcap -R local.rules -A alert_talos \
        --lua 'trace.modules = {wizard = {all = 2}, \
        detection = {fp_search = 1, buffer = 1}}'
P0:wizard:all:1: c2s streaming search found service http
P0:detection:fp_search:1: 5 fp http_inspect.key[6]

http_inspect.stream_tcp[6]:
- - - - - - - - - - - -  - - - - - - - - - - - -  - - - - - - - - -
2F 68 65 6C 6C 6F                                /hello
- - - - - - - - - - - -  - - - - - - - - - - - -  - - - - - - - - -
P0:detection:buffer:1: Buffer dump - empty buffer

http_inspect.stream_tcp[6]:
- - - - - - - - - - - -  - - - - - - - - - - - -  - - - - - - - - -
2F 68 65 6C 6C 6F                                /hello
- - - - - - - - - - - -  - - - - - - - - - - - -  - - - - - - - - -
P0:detection:buffer:1: Buffer dump - empty buffer

##### get.pcap #####
        [1:0:0] GET request (alerts: 1)
#####
```

The Basics

Snort Rule Structure

Snort's intrusion detection and prevention system relies on the presence of Snort rules to protect networks, and those rules consist of two main sections:

- The **rule header** defines the action to take upon any matching traffic, as well as the protocols, network addresses, port numbers, and direction of traffic that the rule should apply to.
- The **rule body** section defines the message associated with a given rule, and most importantly the payload and non-payload criteria that need to be met in order for a rule to match. Although rule options are not required, they are essential for making sure a given rule targets the right traffic.

The following is an example of a fully-formed Snort 3 rule with a correct rule header and rule option definitions:

```
alert tcp $EXTERNAL_NET 80 -> $HOME_NET any
(
    msg:"Attack attempt!";
    flow:to_client,established;
    file_data;
    content:"1337 hackz 1337",fast_pattern,nocase;
    service:http;
    sid:1;
)
```

The rule header includes all the text up to the first parenthesis, while the body includes everything between the two parentheses.

The action defined in a given Snort rule's header is not taken unless all of the rule's individual options evaluate to true.

> **Note:** Snort 3 ignores extra whitespace in rules, and so there's no need to escape newlines with backslashes like what was required with Snort 2 rules.

Rule comments

Rule writers can also add comments to their rules to provide additional context or information about a rule or rule option. These comments are added with `#` to start a comment *line* or with `/* … */` to create either inline or multi-line comments.

```
# hash comment here
content:"ABCD";
```

```
/* these can be used to create
   multi-line comments
*/
content:"ABCD"; /* or they can be used like this */
```

Rule Headers

All Snort rules start with a rule header that helps filter the traffic that the rule's body will evaluate.

A *traditional* rule header consists of five main components, and the following example is used to highlight what these five parts are:

```
alert tcp $EXTERNAL_NET 80 -> $HOME_NET any
```

- Rule **actions** tell Snort what to do when a rule "fires":

  ```
  alert tcp $EXTERNAL_NET 80 -> $HOME_NET any
  ```

- The **protocol** tells Snort which protocol applies:

  ```
  alert tcp $EXTERNAL_NET 80 -> $HOME_NET any
  ```

- **IP addresses** tell Snort what networks to evaluate the rule against:

  ```
  alert tcp $EXTERNAL_NET 80 -> $HOME_NET any
  ```

- **Ports** tell Snort which ports to evaluate the rule against:

  ```
  alert tcp $EXTERNAL_NET 80 -> $HOME_NET any
  ```

- The **direction operator** tells Snort which traffic direction to look for:

  ```
  alert tcp $EXTERNAL_NET 80 -> $HOME_NET any
  ```

We will discuss each of these in detail in the next few pages.

Note: Snort 3 also introduces three new rule types that each have their own rule header format, and all three are described in later sections.

Rule Actions

Rule actions tell Snort how to handle matching packets. There are five basic actions:

- `alert` -> generate an alert on the current packet
- `block` -> block the current packet and all the subsequent packets in this flow
- `drop` -> drop the current packet
- `log` -> log the current packet
- `pass` -> mark the current packet as passed

There are also what are known as "active responses" that perform some action in response to the packet being detected:

- `react` -> send response to client and terminate session.
- `reject` -> terminate session with TCP reset or ICMP unreachable
- `rewrite` -> enables overwrite packet contents based on a "replace" option in the rules

The desired action for a given rule is the very first thing declared in a rule.

Examples:

```
alert http (msg:"Generate an alert"; sid:1;)
```

```
drop http (msg:"Drop this packet"; sid:2;)
```

```
block http (msg:"Block this packet and subsequent ones"; sid:3;)
```

Protocols

The protocol field tells Snort what type of protocols a given rule should look at, and the currently supported ones include:

- `ip`
- `icmp`
- `tcp`
- `udp`

A rule can only have one protocol set, and the name of the protocol is placed after the action.

Examples:

```
alert udp $EXTERNAL_NET any -> $HOME_NET 53 (
```

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 80 (
```

```
alert ip any any -> $HOME_NET any (
```

Services in place of protocols

The above four protocols look for specific "Layer 3" (`ip` and `icmp`) and "Layer 4" (`tcp` and `udp`) protocols. However, rule writers also have the option of specifying *application layer* services here—instead of one of the four aforementioned protocols—to tell Snort to only match on traffic of the specified *service*. This means that not only must the networks, ports, and direction of the traffic match what's present in the header, but the specified service *must also match* the service that Snort detects in the traffic.

To utilize this, one must place the *name* of a given service where a protocol would usually go. For example, if we wanted to match only on traffic sent to destination port 443 that Snort detects as SSL/TLS, we would simply specify `ssl` in our rule header like so:

```
alert ssl any any -> any 443
```

It's important to reiterate that the service specified in the header **MUST** match the service detected in the traffic for a rule to be considered a match. This means, for example, that the above rule header *can only match on traffic that Snort has detected as SSL/TLS*.

The names of services that can be used here can be found by looking at the wizard entries in the `snort_defauls.lua` file included in the `lua/` directory, as well as the curse service names present in the `curse_map` in `src/service_inspectors/wizard/curses.cc`.

Examples:

```
# will only run on HTTP traffic sent to destination port 8000
alert http $EXTERNAL_NET any -> $HOME_NET 8000 (
```

```
# will only run on SMTP traffic sent to destination port 5300
alert smtp $EXTERNAL_NET any -> $HOME_NET 5300 (
```

IP Addresses

IP addresses in a rule header tell Snort what source and destination IP addresses a given rule should apply to. A rule will only match if the source and destination IP addresses of a given packet match the IP addresses set in that rule.

They can be declared in one of four ways:

- As a numeric IP address with an optional CIDR block (e.g., `192.168.0.5`, `192.168.1.0/24`)
- As a variable defined in the Snort config that specifies a network address or a set of network addresses (e.g., `$EXTERNAL_NET`, `$HOME_NET`, etc.)
- The keyword `any`, meaning any IP address
- A list of IP addresses, IP address variables, and/or port ranges, enclosed in square brackets and separated by commas (e.g., `[192.168.1.0/24,10.1.1.0/24]`)

Two IP address declarations are made in a single rule header: the source IP addresses declared after the protocol field and the destination IP addresses declared after the direction operator.

> **Note:** IP address declarations can also be negated to tell Snort to match any IP address except for the ones listed. This negation is done with the `!` operator.

Examples:

```
# look for traffic sent from the 192.168.1.0/24 subnet to the
# 192.168.5.0/24 subnet
alert tcp 192.168.1.0/24 any -> 192.168.5.0/24 any (
```

```
# look for traffic sent from addresses included in the
# defined $EXTERNAL_NET variable to addresses included in the defined
# $HOME_NET variable
alert tcp $EXTERNAL_NET any -> $HOME_NET 80 (
```

```
# look for traffic sent from any source network to the IP address, 192.168.1.3
alert tcp any any -> 192.168.1.3 445 (
```

```
alert tcp !192.168.1.0/24 any -> 192.168.1.0/24 23 (
```

```
alert tcp ![192.168.1.0/24,10.1.1.0/24] any -> [192.168.1.0/24,10.1.1.0/24] 80 (
```

Port Numbers

The port numbers in a rule header tell Snort to apply a given rule to traffic sent from or sent to the specified source and destination ports.

Ports are declared in a few different ways:

- As `any` ports (meaning match traffic being sent from or to *any* port)
- As a static port (e.g., `80`, `445`, `21`)
- As a variable defined in the Snort config that specifies a port or set of ports (e.g., `$HTTP_PORTS`)
- As port ranges indicated with the range operator, `:` (e.g., `1:1024`, `500:`)
- A list of static ports, port variables, and/or port ranges, enclosed in square brackets and separated by commas (e.g., `[1:1024,4444,5555,$HTTP_PORTS]`)

A rule header should have two port declarations, one to define the source ports and another to define the destination ports. Source and destination ports are declared after the source and destination IP addresses, respectively.

An important thing to note, however, is that the ports specified in the rule header do not have to match the ports being used in the traffic if a service specified in the service rule option matches the service of the given traffic. For instance, if `service:http` is set in a rule, then Snort will apply that rule to *all* HTTP traffic detected, even if that traffic is being sent to a port that is not included in the rule's destination port list.

If on the other hand you want to look for a particular service **AND** a specific port or multiple ports, then you should specify a service in the "traditional" rule header as mentioned in the protocols page.

> **Note:** Port declarations can also be negated by placing `!` before them.

Examples:

```
# log udp traffic coming from any source port and destination ports
# ranging from 1 to 1024
log udp any any -> 192.168.1.0/24 1:1024 (
```

```
# log tcp traffic from any port going to ports less than or equal to 6000
log tcp any any -> 192.168.1.0/24 :6000 (
```

```
# log tcp traffic from privileged ports less than or equal to 1024 going
# to ports greater than or equal to 500
log tcp any :1024 -> 192.168.1.0/24 500: (
```

Direction Operators

The direction operator of a header indicates the direction of the traffic that the rule should apply to. There are two valid direction operators:

- `->`
- `<>`

The `->` operator is the most common, and it denotes that the IP addresses and port numbers on the left side represent the source and the IP addresses and port numbers on the right side represent the destination.

The `<>` operation is the bidirectional operator, and it tells Snort to consider the two IP address and port pairs as *either* the source or destination.

The direction operator is placed after the first ports declaration in the header.

Examples:

```
alert tcp $EXTERNAL_NET 80 -> $HOME_NET any (
```

```
log tcp !192.168.1.0/24 any <> 192.168.1.0/24 23 (
```

New Rule Types in Snort 3

Snort 3 introduces three new rule types to simplify and enhance rule writing: **service rules**, **file rules**, and **file identification rules**.

Service and file rules allow for the creation of cleaner-looking rules that are service-specific and service-agnostic, respectively, while file identification rules use the new `file_meta` rule option to perform file type identification.

Each of these new rule types are created using a unique rule header, and more info about each can be found in the subsequent chapters.

Service Rules

Service rules are a new rule type in Snort 3 that allows rule writers to match on traffic of a particular service by using a rule header that consists of only an action and the name of an application-layer service. The difference between these headers and the "traditional" headers described here is that *these* ones do not require declarations of network addresses, ports, or a direction operator.

These service rules let rule writers target a particular service regardless of the IP addresses or ports being used in a given network flow. This type of rule is especially useful for services like HTTP where it's not uncommon to see web servers running on TCP ports other than `80`.

For example, the following rule header tells Snort to apply this rule only to traffic that Snort detects as HTTP:

```
alert http (
```

Note however that with these rules, the service specified in the header **MUST** match the service detected in the traffic for a rule to be considered a match. To be more explicit, the above rule header *can only match on traffic that Snort has detected as HTTP*.

The names of services that can be used here can be found by looking at the wizard entries in the `snort_defauls.lua` file included in the `lua/` directory, as well as the curse service names present in the `curse_map` in `src/service_inspectors/wizard/curses.cc`.

Format:

These rules are created with a rule header that includes only an action followed by a service name.

```
action service
```

> **Note**: Service rules do not require a `service` option declaration in the rule.

Example:

The following rule would alert on matching HTTP traffic regardless of ports or IP addresses used in the communication:

```
alert http
(
    msg:"SERVER-WEBAPP This rule only looks at HTTP traffic";
    flow:to_server,established;
    http_uri;
    content:"/admin.php",fast_pattern,nocase;
    content:"cmd=",nocase;
    pcre:"/[?&]cmd=[^&]*?\x3b/i";
    sid:1;
)
```

File Rules

Snort 3's new "file rules" allow rule writers to create rules to match a particular file regardless of the protocol, source IPs, destination IPs, ports, and service.

Snort is able to process files that are sent using any of the following application-layer protocols:

- HTTP
- SMTP
- POP3
- IMAP
- SMB
- FTP

Format:

These rules are created with a rule header containing just an action followed by the keyword `file`:

```
action file
```

When creating file rules, rule writers should make sure to do the following two things:

- Specify the `file_data` buffer for all content matches that should be matched in the file
- Omit any `service` and `flow` rule options from the rule

Example:

To see the advantage of such a rule header, consider the two rules below that look for "secret_encryption_key" in a packet. The first rule looks for the string included in HTTP and IMAP packets sent to a client, while the second rule looks for the string included in SMTP packets sent to some SMTP server.

```
alert tcp $EXTERNAL_NET [80,143] -> $HOME_NET any
(
    msg:"MALWARE-OTHER Win.Ransomware.Agent payload download attempt";
    flow:to_client,established;
    file_data; content:"secret_encryption_key",fast_pattern,nocase;
    service:http, imap;
    classtype:trojan-activity;
    sid:1;
)
alert tcp $EXTERNAL_NET any -> $SMTP_SERVERS 25
(
    msg:"MALWARE-OTHER Win.Ransomware.Agent payload download attempt";
    flow:to_server,established;
    file_data; content:"secret_encryption_key",fast_pattern,nocase;
    service:smtp;
    classtype:trojan-activity;
    sid:2;
)
```

However, this pair of rules can be written as a single `alert file` rule, which will tell Snort to look for "secret_encryption_key" in any file detected on the network, regardless of source, destination, or service.

```
alert file
(
    msg:"MALWARE-OTHER Win.Ransomware.Agent payload download attempt";
    file_data;
    content:"secret_encryption_key",fast_pattern,nocase;
    classtype:trojan-activity;
    sid:3;
)
```

File Identification Rules

File identification rules take advantage of Snort's detection engine to enable file type identification. These rules are basic Snort 3 rules, but instead of alerting on and/or blocking traffic, they *identify* files based on the contents of that file and then define a file *type* that can be used in subsequent rules with file_type options.

File identification rules have two key components:

- a rule header consisting of only `file_id`, which tells Snort that the rule that follows is a file type definition
- a `file_meta` rule option that set the file metadata for a given file identification rule

Because these rules are used to identify a particular file, rule-writers should look for any and all payload options in the `file_data` buffer.

More info on using the `file_meta` option is available on the file_meta manual page.

Examples:

```
file_id (
    msg:"Windows/DOS executable file";
    file_meta:type MSEXE, id 21, category "Executables,Dynamic Analysis
Capable,Local Malware Analysis Capable";
    file_data;
    content:"| 4D 5A |", depth 2, offset 0;
    gid:4;
    sid:16;
    rev:1;
)
```

A `file_id` entry can also define a specific file type *version*, which is set via the `version` argument.

```
file_id (
    msg:"PDF file";
    file_meta:type PDF, id 282, category "PDF files,Dynamic Analysis
Capable,Local Malware Analysis Capable", version "1.0";
    file_data;
    content:"| 25 50 44 46 2D 31 2E 30 |", depth 8, offset 0;
    gid:4;
    sid:158;
    rev:1;
)
```

Enabling file identification

Use of file identification rules requires that the `file_id` and `file_policy` builtins are enabled in one's Snort 3 config. Fortunately, both are enabled by default in the standard `snort.lua` file:

```
file_id = { rules_file = 'file_magic.rules' }
file_policy = { }
```

Rule Options

Rule options are the heart and soul of a Snort rule, as they determine if a given packet should be passed along to its destination, or if it should instead be stopped in its tracks.

Each rule option has its own set of option-specific critera, but they all follow the same general structure. First, all rule options are enclosed in parentheses after the rule header. Then, each rule option is declared with its name followed optionally by a `:` character and any option-specific criteria. Lastly, each rule option is terminated with a `;` character.

It's important to note that not all options have arguments, and some options have multiple arguments that are separated by commas.

There are four major categories of rule options:

- general options provide additional context for a given rule
- payload options set payload-specific criteria
- non-payload options set non-payload specific criteria
- post-detection options set actions to take on a given packet after the rule has "fired"

All of these are discussed in great detail later in the manual, but here are a few example options and their specific structure:

```
# this is an example of a buffer modifier, "http_uri"
# it tells Snort to look for subsequent content matches only in that buffer
http_uri;

# content match specific criteria, as well as others, require double quotes
content:"/web_form.php";
```

```
# not all options have criteria that require quotes
service:http;
```

```
# some options require specifying a "sub-option"
reference:url,www.example.com;
```

Rule Option Syntax Key

Each rule option page features a "Format" section that describes how the specific rule option can be formatted. Some rule options are simple and specified with just the option name, while others are more complicated and have a mix of required and optional "arguments". This page serves as a key to help understand the syntax that you will find in those sections.

Italicized strings

Italicized strings serve as placeholders for arbitrary values. When one sees this option they must replace it with an appropriate value.

It's important to note italicized strings can be placeholders for different *types* of data, such as strings or integers, and so one must make sure take that into account when specifying a value or values.

In the following example, the user must write `food` followed a food value.

```
food food
```

For example, `food pizza`, `food cookies`, and `food cheese` would all be valid entries.

Square brackets

Square brackets (`[]`) indicate that the enclosed item or items are *optional*. If the items in the square brackets are separated between pipe characters (`|`), then the rule writer can choose one of the items or none of them.

In the following example, the rule writer can choose `pizza` or `cookies`, or none of them.

```
[pizza|cookies]
```

And in the following example, the rule writer can optionally add the `, nocase` string.

```
[, nocase]
```

Curly braces

Curly braces (`{}`) indicate that the rule writer **must** select one—but *only one*—of the items separated by pipe characters.

In the following example, the rule writer must choose either `pizza` or `cookies`, but not both.

```
{pizza|cookies}
```

Ellipses

Three dots (`…`) indicates that a rule option, a rule option argument, or some other modifier can be repeated any number of times. An ellipsis applies to the entire group that it's in, whether that's a curly-brace group or a square-bracket group.

In the following example, the rule writer must specify the string `food` plus any number of *`food`* items separated by commas.

```
food food[, food]…
```

For example, `food pizza, cookies, bacon` would be a valid entry.

General Rule Options

General rule options provide information about a rule, but they do not at all change what a given rule looks for in a packet. General options are not required for a rule, but it is strongly recommended that they are used to provide additional context for a rule should that rule ever generate an event.

Each general option is described in subsequent sections, but the following table lists each one for quick reference.

keyword	description
msg	`msg` sets the message to be printed out when a rule matches
reference	`reference` is used to provide additional context to rules in the form of links to relevant attack identification systems
gid	`gid` identifies the specific Snort component that generates a given event
sid	`sid` identifies the unique signature number assigned to a given Snort rule
rev	`rev` identifies the particular revision number of a given Snort rule
classtype	`classtype` assigns a classification to the rule to indicate the type of attack associated with an event
priority	`priority` sets a severity level for appropriate event prioritizing
metadata	`metadata` adds additional and arbitrary information to a rule in the form of name-value pairs
service	`service` sets the list of services to be associated with a given rule
rem	`rem` is used to convey an arbitrary comment in the rule body
file_meta	`file_meta` is used to set the file metadata for a given file identification rule

msg

The `msg` rule option is used to add a message describing the rule. The message should summarize the rule's purpose, and it will be outputted along with events generated by the rule.

This option takes just a single argument: a text string enclosed in double quotes that explains what kind of traffic the rule will match.

`msg` is typically the first one present in a Snort rule.

> **Note**: Snort rules have a few reserved characters (e.g., `"`, `;`), and rule-writers must escape them with `\` to use them in the rule's `msg` option.

Format:

```
msg:"message";
```

Examples:

```
msg:"SERVER-WEBAPP /etc/inetd.conf file access attempt";
```

```
msg:"Malicious file download attempt";
```

reference

The `reference` rule option provides additional context to rules in the form of links to relevant attack identification systems.

This option takes in two arguments separated by commas. The first argument is the *`scheme`*, which is the attack identification system being referenced, and the second argument is the `id`, which is the specific identifier within that system.

There are a few *`scheme`* types known to Snort by default, but the two most common ones used are `cve` and `url`.

For example, consider the CVE identification system, which identifies a software vulnerability via a CVE record that is formatted like "CVE-XXXX-YYYY", where "XXXX" is the year the vulnerability was identified and "YYYY" is a unique numeric identifier. Snort rule writers can put references to CVE records in rules with a `reference` option that has `scheme` set to `cve` and the `id` set to the "XXXX-YYYY" portion of the record. For example, `reference:cve,2020-1234` puts in the rule a reference to `CVE-2020-1234`.

Format:

```
reference:scheme,id;
```

Examples:

```
reference:url,www.example.com;
```

```
reference:cve,2020-1234;
```

gid

The `gid` keyword stands for "generator id" and it identifies the specific part of Snort that generated a given event. The different parts include, but are not limited to, standard text rules, shared object rules, and builtin rules, and each have part their own generator ID. Standard text rules, for example, are identified with as GID 1, shared object rules are identified with GID 3, and builtin rules are identified with gids over 100.

The GIDs included within Snort can be listed with the following command:

```
$ snort --list-gids
```

It's important to note that the `gid` keyword is **optional**, and if it is not specified in a rule, then it will default to 1 and the rule will be part of the standard rule subsystem.

Format:

```
gid:generator_id;
```

Examples:

```
# sets the rule's generator ID to 1 to tell Snort it's a standard text rule
gid:1;
```

```
# sets the rule's generator ID to 3 to tell Snort it's a shared object rule
gid:3;
```

sid

The `sid` keyword uniquely identifies a given Snort rule. This rule option takes in a single argument that is a numeric value that must be unique to the rule.

While not technically required, all Snort rules should have a `sid` option to be able to quickly identify a rule should it ever generate an alert.

Snort "reserves" `sid` values 0-999999 because those are used in rules included with the Snort distribution. Therefore users should use for local rules `sid` values that start at 1000000, incrementing the `sid` values by one for each additional local rule.

Format:

```
sid:signature_id;
```

Example:

```
sid:44763;
```

```
sid:1000001;
```

rev

The `rev` keyword uniquely identifies the revision number of a given Snort rule. This option should be used along with the `sid` keyword and should be incremented by one each time a change is made to a rule.

This option takes in a single argument, a numeric value that identifies the rule's current revision number. Revision values start at `1`, and rules will default to this value if the option is omitted from them.

Format:

```
rev:revision;
```

Examples:

```
sid:1000001; rev:1;
```

```
sid:1000001; rev:2;
```

classtype

The `classtype` assigns a classification to the rule to indicate the type of attack associated with an event. Snort provides a list of default classifications that rule-writers can use to better organize rule event data.

Note that a rule should only have one `classtype` declaration.

Attack classifications provided by Snort reside in the `snort_defaults.lua` configuration file, and they use a table syntax like so with three entries:

```
{ name = 'attempted-user', priority = 1,
      text = 'Attempted User Privilege Gain' }
```

Snort's current default classifications use priority values 1-4, with 1 being the most severe and 4 being the least severe. The following is a table of all default classifications provided by Snort:

Classtype	Description	Priority
`not-suspicious`	Not Suspicious Traffic	3
`unknown`	Unknown Traffic	3
`bad-unknown`	Potentially Bad Traffic	2
`attempted-recon`	Attempted Information Leak	2
`successful-recon-limited`	Information Leak	2
`successful-recon-largescale`	Large Scale Information Leak	2
`attempted-dos`	Attempted Denial of Service	2
`successful-dos`	Denial of Service	2
`attempted-user`	Attempted User Privilege Gain	1
`unsuccessful-user`	Unsuccessful User Privilege Gain	1
`succesful-user`	Successful User Privilege Gain	1
`attempted-admin`	Attempted Administrator Privilege Gain	1
`successful-admin`	Successful Administrator Privilege Gain	1
`rpc-portmap-decode`	Decode of an RPC Query	2
`shellcode-detect`	Executable code was detected	1
`string-detect`	A suspicious string was detected	3

Classtype	Description	Priority
`suspicious-filename-detect`	A suspicious filename was detected	2
`suspicious-login`	An attempted login using a suspicious username was detected	2
`system-call-detect`	A system call was detected	2
`tcp-connection`	A TCP connection was detected	4
`trojan-activity`	A Network Trojan was detected	1
`unusual-client-port-connection`	A client was using an unusual port	2
`network-scan`	Detection of a Network Scan	3
`denial-of-service`	Detection of a Denial of Service Attack	2
`non-standard-protocol`	Detection of a non-standard protocol or event	2
`protocol-command-decode`	Generic Protocol Command Decode	3
`web-application-activity`	Access to a potentially vulnerable web application	2
`web-application-attack`	Web Application Attack	1
`misc-activity`	Misc Activity	3
`misc-attack`	Misc Attack	2
`icmp-event`	Generic ICMP event	3
`inappropriate-content`	Inappropriate Content was Detected	1
`policy-violation`	Potential Corporate Privacy Violation	1
`default-login-attempt`	Attempt to login by a default username and password	2
`sdf`	Sensitive Data	2
`file-format`	Known malicious file or file based exploit	1
`malware-cnc`	Known malware command and control traffic	1
`client-side-exploit`	Known client side exploit attempt	1

Format

```
classtype:classification;
```

Examples:

```
classtype:web-application-attack;
```

```
classtype:attempted-user;
```

priority

The `priority` option assigns a severity level to a given rule to enable appropriate event prioritizing. Although the builtin classifications set with `classtype` come with their own priority levels, rule writers can override those by using the `priority` option.

Valid priority levels are 1-2147483647, with 1 being the most severe and 2147483647 being the least severe.

A given rule should only have one `priority` declaration.

Format:

```
priority:severity_level;
```

Examples:

```
priority:1;
```

```
priority:10;
```

metadata

The `metadata` option adds additional and arbitrary information to a rule in the form of key-value pairs. There are a few keys that have special meanings to Snort and Snort products (such as `policy`), but generally speaking this option is free-form and can contain arbitrary keys and values.

Key-value pairs set in this option are separated by spaces, and rule writers can also include multiple key-value pairs in this option by separating them with commas. It's important to note that a key's value *can* have spaces in it, but it's that *first* space that separates the key from the value.

> **Note**: Service declarations were made in the `metadata` option in Snort 2, but Snort 3 has moved these declarations to an entirely new keyword, `service`.

Format:

```
metadata:key value[, key value]…;
```

Examples:

```
metadata:policy max-detect-ips drop;
```

```
metadata:policy max-detect-ips drop, policy security-ips drop;
```

service

The `service` rule option is used to tell Snort what application-layer service or services you want your rule to apply to. When Snort receives traffic, one of the things it will do first is determine the service of the traffic so that it can process and parse it correctly, and rule writers can target those services using this `service` rule option.

This rule option takes as arguments one or more comma-separated service *names*. The names of services that can be used here can be found by looking at the wizard entries in the `snort_defauls.lua` file included in the `lua/` directory, as well as the curse service names present in the `curse_map` in `src/service_inspectors/wizard/curses.cc`.

Note that Snort handles services specified in this rule option differently than how it handles a service specified in the rule *header* (as seen here and here). More specifically, with *this* rule option, the services specified **do not** have to match the actual service of the traffic **as long as the ports match**. Additionally, the converse of that statement is true as well; the *ports* do not have to match as long as the services match. Another way to look at the `service` option is like an "OR ports" statement.

For instance, if the source or destination ports present in a rule's header match those used in the traffic, but the service specified in the rule does not match the service present in the traffic, the rule can still match as long as the rest of the rule evaluates to true. The following rule, for example, will apply **either** to traffic Snort detects as HTTP or traffic that is destined for TCP port `8000`:

```
alert tcp any any -> any any 8000 (
    msg:"HTTP traffic or dst port 8000 please";
    service:http;
    sid:1000000;
)
```

Note that if you are already specifying a service in the rule header or creating a file rule, then you should not include this `service` rule option in your rule.

> **Note:** This rule option should be used instead of placing service declarations in the `metadata` option like what was done in Snort 2.

Format:

```
service:service[,service]…;
```

Examples:

```
service:http;
```

```
service:http,imap,pop3;
```

rem

The `rem` rule option is used to convey an arbitrary comment in the rule body. This option has no effect on detection and should be used simply to provide meta information about the rule its in.

This option takes in a single argument, a string enclosed in double quotes.

Format:

```
rem:"comment_string";
```

Examples:

```
rem:"check for a malicious URI string";
http_uri;
content:"/php_backdoor.php";
```

file_meta

The `file_meta` rule option is used to set the file metadata for a given file identification rule. This option sets the type name, numerical id value, category, groups, and version for the file that will be matched.

This rule option has two required arguments:

1. `type *type*`: string to give the file identifier a name
2. `id *type_id*`: integer value that should be unique to this particular file identifier rule

There are also three additional optional arguments that rule-writers can use to add greater specificity to the `file_id` rule:

1. `category *type_category*`: string that sets the category of the file being identified
2. `group *type_groups*`: string that contains a list of groups—separated by commas—associated with that particular file type
3. `version *type_version*`: string that sets the version of the particular file that's being identified

Each of these arguments is separated by commas.

Note: This rule option should only be used in `file_id` rules.

Format:

```
file_meta:type type, id type_id[, category "type_category"] \
          [, group "type_groups"][, version "type_version"];
```

Examples:

```
# defines the parameters for Windows/DOS executable files
file_meta:type MSEXE, id 21, category "Executables,Dynamic Analysis Capable";
```

```
file_meta:type PDF, id 282, category "PDF files", version "1.0";
```

```
file_meta:type MOV, id 4, category "Multimedia", group "video";
```

Payload Detection Rule Options

Snort rules are best at evaluating a network packet's "payload" (e.g., the TCP or UDP data fields), and this chapter covers what are referred to as "payload detection" options. These options tell Snort *what* kind of packet data to look for, *where* to look for that data, and lastly *how* to look for said data.

A single Snort rule can contain multiple options, and those options are evaluated against the packet data in the order they are placed in the rule (except for certain `fast_pattern` matches).

Inspection Buffers

When Snort receives network traffic and begins processing, it places the packet data into various "buffers" that rule writers can evaluate payload options against. Snort provides buffers for the raw packet data, normalized packet data, "file" data, individual HTTP elements, like `http_header` and `http_uri`, and more. Not all buffers will be available for a given packet, and so rule writers should use table at the end of this page to make sure they are using the appropriate one(s).

As Snort evaluates payload options against a given buffer, it keeps track of its current location there with a detection-offset-end (DOE) pointer (also sometimes referred to as a cursor). By default, this pointer points to the start of the current buffer, but some rule options will "move" this pointer forward and backwards, which allow for the use of relative payload options.

Sticky buffers

By default, rule options are evaulated against data present in the `pkt_data` buffer. Looking for data in one of the other buffers is done by using what are called "sticky buffers", which are rule options that, when set, move the DOE pointer to the start of that particular buffer. Then, all subsequent payload options will be looked for in that buffer unless some other sticky buffer is specified.

One of those sticky buffers, for example, is `http_uri`, which contains the URI portion of an HTTP request. Setting this buffer and looking for data there might be done like so:

```
http_uri;
content:"/index.php";
```

> **Note**: Depending on one's Snort configuration, Snort will place certain payload data—such as HTTP-specific elements—**only** in their respective buffers. This means, for example, that trying to look for HTTP request elements in the default, normalized packet data buffer will result in a detection failure.

This section breaks down each of those rule payload options to explain how they are used and how they work, starting with the `content` option.

Quick Reference

keyword	description
content	`content` is used to perform basic string and/or hexadecimal pattern matching
fast_pattern	`fast_pattern` is a `content` modifier that tells Snort to use that particular match to determine if further rule processing should continue against the traffic
nocase	`nocase` is a `content` modifier that tells Snort to ignore case when looking for a specified pattern
offset	`offset` is a `content` modifier that specifies where to start searching for a pattern relative to the beginning of the packet or buffer
depth	`depth` is a `content` modifier that specifies how far into a Snort packet or buffer to look for the specified pattern relative to the beginning of the packet or buffer
distance	`distance` is a `content` modifier that specifies where to start searching for a pattern relative to the previous content match
within	`within` is a `content` modifier that specifies how far into a Snort packet or buffer to look for the specified pattern relative to the previous content match
HTTP buffers	`http_*` options are sticky buffer declarations that set the detection cursor to the beginning of the various HTTP parts
bufferlen	`bufferlen` checks the length of a given buffer
isdataat	`isdataat` verifies the payload data exists at a specified location
dsize	`dsize` tests packet payload size
pcre	`pcre` is used to create perl compatible regular expressions
regex	`regex` is used to create perl compatible regular expressions that are checked against payload data with the hyperscan engine

keyword	description
pkt_data	`pkt_data` is a sticky buffer declaration that sets the detection cursor to the beginning of the normalized packet data
raw_data	`raw_data` is a sticky buffer declaration that sets the detection cursor to the beginning of the raw packet data
file_data	`file_data` is a sticky buffer declaration that sets the detection cursor to either the HTTP response body for HTTP traffic or file data sent via other application protocols that has been processed and captured by Snort's "file API"
js_data	`js_data` is a sticky buffer declaration that sets the detection cursor to the normalized JavaScript data buffer
vba_data	`vba_data` is a sticky buffer declaration that sets the detection cursor to the buffer containing VBA macro code
base64_decode	`base64_decode` is used to decode base64-encoded data in a packet
base64_data	`base64_data` is a sticky buffer declaration that sets the detection cursor to the beginning of the base64 decoded buffer
byte_extract	`byte_extract` reads some number of bytes from packet data and stores the extracted byte or bytes into a named variable
byte_test	`byte_test` tests a byte or multiple bytes from the packet against a specific value with a specified operator
byte_math	`byte_math` extracts bytes from the packet and performs a mathematical operation on the extracted value, storing the result in a new variable
byte_jump	`byte_jump` reads some number of bytes from the packet, converts them from their numeric representation if necessary, and moves that many bytes forward
ber_data and ber_skip	`ber_*` rule options evaluate and work with BER-encoded data
ssl_state and ssl_version	`ssl_*` rule options evaluate and work with SSL/TLS sessions
DCE Specific Options	`dce_*` rule options evaluate and work with DCERPC traffic
SIP Specific Options	`sip_*` rule options evaluate and work with SIP traffic
sd_pattern	`sd_pattern` detects sensitive data, such as credit card and social security numbers
cvs	`cvs` looks for a specific attack types
md5, sha256, and sha512	`md5`, `sha256`, and `sha512` check payload data against a specified hash value

keyword	description
GTP Specific Options	`gtp_*` rule options evaluate specific elements of GTP traffic
DNP3 Specific Options	`dnp3_*` rule options evaluate specific elements of DNP3 traffic
CIP Specific Options	`cip_*` rule options evaluate specific elements of CIP traffic
IEC 104 Specific Options	`iec104_*` rule options evaluate specific elements of IEC 104 traffic
MMS Specific Options	`mms_*` rule options evaluate specific elements of MMS traffic
Modbus Specific Options	`modbus_*` rule options evaluate specific elements of Modbus traffic
S7CommPlus Specific Options	`s7commplus_*` rule options evaluate specific elements of S7CommPlus traffic

content

The first option we will discuss is `content`, which is used to perform basic pattern matching against packet data. This option is declared with the `content` keyword, followed by a `:` character, and lastly followed the content string enclosed in double quotes. Matches can also be "negated" with a `!` character immediately after the colon, telling Snort only to handle packets that *do not* contain some string or hex sequence.

Content matches can contain ASCII strings, hex bytes, or a mix of both. Hex bytes must be enclosed in `|` characters.

A rule can contain multiple content matches, and each match is evaluated in the order they are declared in the rule (except `fast_pattern` matches, which is discussed in the next chapter). This means of course that Snort will continue checking subsequent matches against packet data as long as the content checks continue to pass. As a result, it's often times beneficial to place the most unique sequence of mathces towards the top of the rule to allow Snort the opportunity to exit processing early.

Format:

```
content:[!]"content_string";
```

Examples:

```
# Simple ascii string match
content:"USER root";
```

```
# Combining of ascii characters and hex bytes
content:"PK|03 04|";
```

> **Note:** Certain characters must be either escaped (with '\' characters) or encoded in hex. These are: ';', '\', and '"'.

Content match modifiers

Snort content matches can be written with option modifiers to set additional evaluation requirements for a given content match, offering users greater specificity when defining rule parameters.

These modifiers include `fast_pattern`, `nocase`, `within`, `distance`, `offset`, and `depth`, and they are written alongside the content string, separated by commas. Certain modifiers

also require "arguments" that define the parameters to go along with them.

Format:

```
, content_modifer [content_modifer_argument]
```

Examples:

```
content:"pizza", nocase;
```

```
content:"cheese";
content:" pizza", within 6;
```

We will talk about each one content match modifier in depth in the ensuing sections, starting with the `fast_pattern` option.

fast_pattern

Snort's fast pattern matcher is crucial for performance, as it helps determine which packets qualify for the additional processing that comes with rule option evaluation. At a high-level, the fast pattern engine uses a single content match from a rule and evaluates it against the packet to determine if further rule processing should continue against the traffic. The ideal fast pattern is one which, if found, is very likely to result in a rule match. Fast patterns that match frequently against unrelated traffic will cause Snort to work hard with little to show for it.

Fast pattern matches are either explicitly set with the `fast_pattern` option or set automatically to the longest content match if the option is not specified. However, it's important to keep in mind that the longest pattern is sometimes not the most unique, and so one can add the `fast_pattern` modifier to a content option to maximize performance.

During rule evaluation, the content string selected as the `fast_pattern` match will *automatically be skipped if possible*. This is a change from Snort 2. Previously, users would have to specify `fast_pattern:only` to evaluate a fast_pattern match only once; Snort 3 now intelligently evaluates the `fast_pattern` match only once if it is able.

> **Note:** Certain buffers are not eligible to contain `fast_pattern` content matches, and those include the following: `http_raw_cookie`, `http_param`, `http_raw_body`, `http_version`, `http_raw_request`, `http_raw_status`, `http_raw_trailer`, and `http_true_ip`.

Format:

```
fast_pattern
```

Example:

```
content:"super_secret_encryption_key",fast_pattern;
```

fast_pattern_offset, fast_pattern_length

Users can also specify that only a portion of a content match be used as as `fast_pattern`. This is specified with two modifiers, `fast_pattern_offset` and `fast_pattern_length`. The former sets the number of leading characters of this content the fast pattern should

exclude, while the latter sets the number of characters from this content to include in the fast pattern matcher. Valid values are 0:65535 and 1:65535 for offset and length, respectively.

Format:

```
fast_pattern_offset offset, fast_pattern_length length
```

Example:

```
# Only the "/not_a_cnc_endpoint.php" portion of the match is used as the fast
pattern
content:"/index/not_a_cnc_endpoint.php",fast_pattern_offset
6,fast_pattern_length 23;
```

This above option will, however, still evaluate the full content match normally as long as the fast pattern check is successful.

nocase

Content matches are case-sensitive by default, but the `nocase` content modifier tells Snort to ignore case and look for the specified string match case-insensitively.

Format:

```
nocase
```

Examples:

```
content:"/index/vulnerable_endpoint.php",nocase;
```

```
# It is common to see nocase used with `fast_pattern`
content:"super_secret_encryption_key",fast_pattern,nocase;
```

> **Note:** The `nocase` option also applies to hex bytes; specifying `content:"|41|",nocase;` will look for either 'A' *or* 'a' in the packet.

---

offset, depth, distance, and within

These four content modifiers, `depth`, `offset`, `distance`, and `within`, let rule writers specify where to look for a given pattern relative to either the start of a packet or a previous content match. These four options, however, let users write nuanced rules to look for matches at specific locations. More specifically, `depth` and `offset` are used to look for a match relative to the start of a packet or buffer, whereas `distance` and `within` are instead relative to a previous content match.

offset

The `offset` modifier allows the rule writer to specify where to start searching for a pattern relative to the beginning of the packet or buffer. So, an offset of 5 would tell Snort to look for the specified pattern after the first 5 bytes of the payload.

This keyword allows values from -65535 to 65535, and it can also be set to a string value referencing a variable extracted by the `byte_extract` keyword in the same rule.

Format:

```
offset {offset|variable_name}
```

Example:

```
content:"|FE|SMB", offset 4;
```

depth

The `depth` modifier allows the rule writer the ability to specify how far into a Snort packet or buffer to look for the specified pattern. For example, setting `depth` to 5 would tell Snort to only look for the pattern within the first 5 bytes of the payload.

Specifying `depth` without offset will implicitly look at the start of the payload (offset 0), so there's no need to specify an additional `offset 0` option in those cases.

The value chosen must be greater than or equal to the length of the content string, and the max value is 65535. It can also be set to a string value referencing a variable extracted by the `byte_extract` keyword in the same rule.

Format:

```
depth {depth|variable_name}
```

Examples:

```
# Can combine depth and offset for a single content match
content:"|FE|SMB", depth 4, offset 4;
```

```
content:"PK|03 04|", depth 4;
```

distance

The `distance` keyword is similar to `offset` but is relative to a preceding content match instead of the start of the payload/buffer. It tells Snort to look skip *X* number of bytes after the last content match before looking for this one.

This keyword allows values from -65535 to 65535, and it can also be set to a string value referencing a variable extracted by the `byte_extract` keyword in the same rule.

Format:

```
distance {distance|variable_name}
```

Example:

```
content:"ABC";
content:"EFG", distance 1;
```

> **Note:** To put this example in the context of cursors, the "ABC" match moves the detection cursor to point to the byte immediately after the 'C' character. And then the `distance 1` modifier in the very next match tells Snort not to reset the cursor back to the beginning of the buffer, but to instead increase the cursor by 1 so that it points to second byte in the buffer after the 'C' character.

---

within

The `within` keyword is similar to `depth` but is relative to a preceding content match instead of the start of the payload. It tells Snort to look this content match within *X* number of bytes of the last one.

Specifying `within` without distance will implicitly look immediately after the last content match, so there's no need to specify an additional `distance 0` option in those cases.

The value chosen must be greater than or equal to the length of the content string, and the max value is 65535. It can also be set to a string value referencing a variable extracted by the `byte_extract` keyword in the same rule.

Format:

```
within {within|variable_name}
```

Examples:

```
content:"ABC";
content:"EFG", within 10;
```

```
# Distance and within can be used together
content:"ABC";
content:"EFG", distance 1, within 3;
```

```
content:"DEF";
content:"GHI", within 3;
```

> **Note:** Content matches specified without any of these four modifiers will always be looked for starting from the beginning of (1) the default buffer or (2) an explicitly-set sticky buffer.

> **Note:** Rule writers are free to use all four options in a single rule, but only `distance` and `within` and `offset` and `depth` can appear together attached a single content match. For example, the second content option in `content:"DEF"; content:"GHI", offset 6, within 3;` is not valid due to the combination of `offset` and `within`.

HTTP Specific Options

Snort operates with a bevy of "service inspectors" that can identify specific TCP/UDP applications and divide the application data into distinct buffers. One of those service inspectors that does exactly this is the "HTTP inspector".

Whenever HTTP traffic is detected in a packet, the HTTP service inspector scans the payload data to parse the different HTTP elements (e.g., URIs, headers, methods, etc.) and populates individual buffers with those different pieces. This powerful inspector allows rule writers to then develop rules with content matches targeting only specific parts of an HTTP packet.

Most HTTP options in Snort 3 rules are "sticky buffers", as opposed to content-modifiers like they were in Snort 2, meaning they should be placed *before* a content match option to set the desired buffer (e.g., `http_uri; content:"/pizza.php";`). In addition to these sticky buffers, there are also a few non-sticky-buffer HTTP rule options that are used to run checks on specific parts of an HTTP message.

This section goes over each of these rule options in great detail, describing how to use each one, what HTTP data is included in each sticky buffer, and how that data gets formatted in the different buffers.

Quick Reference

keyword	description
http_uri	Normalized HTTP URI
http_raw_uri	Unnormalized HTTP URI
http_header	Normalized HTTP headers
http_raw_header	Unnormalized HTTP headers
http_cookie	Normalized HTTP cookies
http_raw_cookie	Unnormalized HTTP cookies
http_client_body	Normalized HTTP request body
http_raw_body	Unnormalized HTTP request body and response data
http_param	Specific HTTP parameter values
http_method	HTTP request methods
http_version	HTTP request and response versions
http_stat_code	HTTP response status codes
http_stat_msg	HTTP response status messages

keyword	description
http_raw_request	Unnormalized HTTP start lines
http_raw_status	Unnormalized HTTP status lines
http_trailer	Normalized HTTP trailers
http_raw_trailer	Unnormalized HTTP trailers
http_true_ip	Original client IP address as stored in various request proxy headers
http_version_match	Non-sticky buffer option used to test an HTTP message's version against a list of versions
http_num_headers	Non-sticky buffer option used to test the number of HTTP headers against a specific value or a range of values
http_num_trailers	Non-sticky buffer option used to test the number of HTTP trailers against a specific value or a range of values
http_num_cookies	Non-sticky buffer option used to test the number of HTTP cookies against a specific value or a range of values
Combining Request and Response Detection	Explains how to create rules that examine an HTTP response and the HTTP request associated with that response

http_uri and http_raw_uri

These two sticky buffers, `http_uri` and `http_raw_uri`, look for data in HTTP request URIs. The `http_uri` buffer contains the full *normalized* URI whereas the `http_raw_uri` contains the *unnormalized* URI.

Snort 3 also parses HTTP URIs into six individual components and makes them available as optional selectors to these two buffers. Those six components are `path`, `query`, `fragment`, `host`, `port`, and `scheme`.

The following HTTP request line contains an absolute URI with all six components, and they are broken down below it:

```
GET https://www.samplehost.com:287/basic/example/of/path?with-query=value#and-fragment HTTP/1.1\r\n
```

1. path -> `/basic/example/of/path`
2. query -> `with-query=value`
3. fragment -> `#and-fragment`
4. host -> `www.samplehost.com`
5. port -> `287`
6. scheme -> `https`

The contents of the general URI buffer and each individual component depend on request URI type. There are four main URI types that Snort can parse. These include *asterisk* URIs, which contain just a '*' character, *absolute URI* URIs, which contain all six components (if they're all present), *absolute path* URIs, which contain the path, query, and fragment (which is not often sent over the network), and lastly *authority* URIs, which contain the host and port.

Specifying an optional URI selector is done with a colon after `http_uri` or `http_raw_uri` followed by the selector name.

http_uri

As mentioned above, `http_uri` sticky buffer searches proceeding payload options in the normalized URI. Snort parses an HTTP request, normalizes anything in the URI that needs normalization, and then places the end-result in the `http_uri` buffer. This normalization does things like decode percent-encoded values (e.g., "%41%41" -> "AA"), replace backslashes with forward slashes, replace plus characters with spaces, remove path

directory traversals, simplify paths (e.g., "/nothing/../to/./././see//////detour/to/nowhere/../.././../example" -> "/very/easy/example"), and more.

There is not an exhaustive list, and some are disabled by default, but the full list can be found in the http_inspect documentation.

> **Note:** Any data in URIs that are not "unnormalized" will of course be left intact and included in the `http_uri` buffer as-is.

Format:

```
http_uri[:{scheme|host|port|path|query|fragment}];
```

Examples:

```
http_uri;
content:"/basic/example/of/path?query=value",fast_pattern,nocase;
```

```
http_uri;
content:"/basic/example/of/path",fast_pattern,nocase;
http_uri:query;
content:"query=value",nocase;
```

http_raw_uri

The `http_raw_uri` sticky buffer searches proceeding content matches in the *unnormalized* URI. Testing against this buffer is often useful when one wants to explicitly look for things like percent-encoded data and directory traversal attempts.

Consider the following raw absolute path URI:

```
/%63%68%6F%63%6F%6C%61%74%65/%63%61%6B%65
```

Snort will normalize this data and put the percent-decoded values in the `http_uri` buffer. However, the `http_raw_uri` buffer will still contain the URI in its raw form, allowing us to check for this percent-encoded string.

Format:

```
http_raw_uri[:{scheme|host|port|path|query|fragment}];
```

Examples:

```
http_raw_uri;
content:"/%63%68%6F%63%6F%6C%61%74%65/%63%61%6B%65";
```

```
http_raw_uri;
content:"../";
```

http_header and http_raw_header

Snort makes HTTP request and response headers available in two sticky buffers, `http_header` and `http_raw_header`. The `http_header` buffer contains the *normalized* request/response headers, whereas the `http_raw_header` buffer contains *unnormalized* ones.

The header normalization that occurs is similar to the URI normalization and includes things like percent-decoding and path-simplification.

Snort 3 also allows users the ability to look for content matches in specific HTTP header fields with the optional `field header_name` argument. This option is specified with a colon character after `http_header`, followed by the word "field", and lastly followed by the specific header field name (which is case-insensitive). For example:

```
http_header:field user-agent;
```

Specifying individual headers like this creates a more efficient and accurate rule.

`http_header` and `http_raw_header` also allow for an optional `request` argument, which is useful if writing detection that looks at both an HTTP client request and the HTTP server response to that request. More specifically, this `request` argument is used to signify that the `http_header` or `http_raw_header` match or matches should apply to the headers from the *request* if other parts of the rule are examining the *response*. More information on this topic can be found on this page.

http_header

Format:

```
http_header[:field header_name][,request];
```

Examples:

```
http_header;
content:"User-Agent: abcip",fast_pattern,nocase;
content:"Accept-Language: en-us",nocase,distance 0;
```

```
# http_header field name arguments are case-insensitive
http_header:field user-agent;
content:"abcip";
```

http_raw_header

Format:

```
http_raw_header[:field header_name][,request];
```

Examples:

```
http_raw_header;
content:"Accept-Language: en-us",fast_pattern,nocase;
```

```
http_raw_header;
content:"Accept-Language:",fast_pattern,nocase;
content:"%60whoami",within 30;
```

http_cookie and http_raw_cookie

HTTP request and response Cookie values are placed into two sticky buffers, `http_cookie` and `http_raw_cookie`. The `http_cookie` buffer contains the normalized Cookie header values, whereas the `http_raw_cookie` buffer contains unnormalized ones.

The cookie normalization that occurs is also similar to the URI normalization and includes things like percent-decoding and path-simplification.

Snort 3 has also made `http_cookie` matches eligible for fast patterns.

If an HTTP request contains multiple Cookie headers, then each Cookie header value is extracted and placed into the two `*_cookie` buffers, with each full header value separated by commas.

For example, consider the following request with two Cookie headers:

```
Cookie: name=value; name2=value2; name3=value3
Cookie: name4=value4; name5=value5; name6=value6
```

Snort 3 will combine the two Cookie values and place them in the two buffers like so:

```
[http_cookie]
00000000  6E 61 6D 65 3D 76 61 6C 75 65 3B 20 6E 61 6D 65  name=value; name
00000010  32 3D 76 61 6C 75 65 32 3B 20 6E 61 6D 65 33 3D  2=value2; name3=
00000020  76 61 6C 75 65 33 2C 6E 61 6D 65 34 3D 76 61 6C  value3,name4=val
00000030  75 65 34 3B 20 6E 61 6D 65 35 3D 76 61 6C 75 65  ue4; name5=value
00000040  35 3B 20 6E 61 6D 65 36 3D 76 61 6C 75 65 36     5; name6=value6
```

and

```
[http_raw_cookie]
00000000  6E 61 6D 65 3D 76 61 6C 75 65 3B 20 6E 61 6D 65  name=value; name
00000010  32 3D 76 61 6C 75 65 32 3B 20 6E 61 6D 65 33 3D  2=value2; name3=
00000020  76 61 6C 75 65 33 2C 6E 61 6D 65 34 3D 76 61 6C  value3,name4=val
00000030  75 65 34 3B 20 6E 61 6D 65 35 3D 76 61 6C 75 65  ue4; name5=value
00000040  35 3B 20 6E 61 6D 65 36 3D 76 61 6C 75 65 36     5; name6=value6
```

The same is also true for Set-Cookie headers.

`http_cookie` and `http_raw_cookie` also allow for an optional `request` argument, which is useful if writing detection that looks at both an HTTP client request and the HTTP server response to that request. More specifically, this `request` argument is used to signify that

the `http_cookie` or `http_raw_cookie` match or matches should apply to the cookies from the *request* if other parts of the rule are examining the *response*. More information on this topic can be found on this page.

> **Note:** `http_cookie` matches *are* eligible for fast patterns, which is a change new to Snort 3.

> **Note:** The "Cookie:" and "Set-Cookie:" portions of these headers are not included in either of the two `*_cookie` buffers.

---

http_cookie

Format:

```
http_cookie[:request];
```

Examples:

```
http_cookie;
content:"name=value",depth 10;
```

```
http_cookie;
content:"name=value",fast_pattern;
content:"name6=value6",distance 0;
```

http_raw_cookie

Format:

```
http_raw_cookie[:request];
```

Examples:

```
http_raw_cookie;
content:"name=value";
```

```
http_raw_cookie;
content:"name=";
content:"%60whoami",nocase,within 25;
```

http_client_body and http_raw_body

Snort places HTTP message data into two sticky buffers, `http_client_body` and `http_raw_body`. The former contains normalized message *request* data, while the latter contains unnormalized request *and response* message data.

The request data normalization that occurs is also similar to the URI normalization and includes things like percent-decoding and path-simplification.

Snort is also able to decompress request and response data (e.g., gzip-compression), and so it will, depending on one's configuration, place the decompressed data in both the `http_client_body` and `http_raw_body` buffers. Furthermore, the `http_raw_body` will contain de-chunked and decompressed data if applicable, but it will not be modified/normalized in any other way.

http_client_body

Format:

```
http_client_body;
```

Examples:

```
http_client_body;
content:"user=root",fast_pattern,nocase;
```

```
http_client_body;
content:"pizza_type=",fast_pattern,nocase;
content:"../",within 20;
```

http_raw_body

Format:

```
http_raw_body;
```

Examples:

```
http_raw_body;
content:"user=root",nocase;
```

```
http_raw_body;
content:"pizza_type=",nocase;
content:"%2e%2e",nocase,distance 0;
```

http_param

Rule writers can access the value of a specific HTTP parameter with the `http_param` sticky buffer. This buffer will contain only the value of the specified parameter. This option is perfect for when rule-writers want to match a particular parameter's value but aren't sure if that parameter is sent via the URI or the client body.

The `http_param` buffer will be populated with the specified parameter's value whenever that parameter appears in either a query string sent via the URL or in a urlencoded form (key-value pair) sent in the client body. `http_param` currently **does not** support multipart/form-data requests, and so a separate rule that does not use `http_param` would be needed in order to detect a parameter sent using a multipart request.

The parameter value *will* be URL decoded but not path normalized. The parameter name argument is case-sensitive by default, but Snort can be instructed to ignore case by adding `,nocase` after the param name.

For example, given a request like `/food.php?favoriteFood=pizza`, users can set the `http_param` argument to `favoriteFood` to look only at that param's value.

It is also recommended that rule writers make the first rule option used after `http_param` "relative" by adding either `distance 0` to a content match or adding the `R` flag to a pcre. This is because `http_param` by default will look at only the first instance of the specified parameter in the request, even if that parameter "key" appears multiple times. Making the first option relative instructs Snort to look at *all* instances of that parameter.

> **Note:** `http_param` matches *are not* eligible to be used as fast patterns.

Format:

```
http_param:"param_name"[,nocase];
```

Examples:

```
http_param:"favoriteFood",nocase;
# note the "distance 0" relative modifier used to make the option relative
content:"pizza",nocase,distance 0;
```

```
http_uri;
content:"/food.php",fast_pattern,nocase;
http_param:"favoriteFood",nocase;
# note the 'R' flag used to make the option relative
pcre:"/([\x27\x22\x3b\x23\x28]|\x2f\x2a|\x2d\x2d)/R";
```

http_method

The HTTP request method is accessible to rule writers via the `http_method` sticky buffer. Common values are `GET`, `POST`, `OPTIONS`, `HEAD`, `DELETE`, `PUT`, `TRACE`, and `CONNECT`.

> **Note:** `http_method` matches *are* eligible for fast patterns, which is a change new to Snort 3.

Format:

```
http_method;
```

Examples:

```
http_method;
content:"POST";
```

```
http_method;
content:"GET",fast_pattern;
```

http_version

Snort parses the HTTP version from request and response start/status lines and makes it accessible to rule-writers via the `http_version` sticky buffer. This is usually `HTTP/1.0` or `HTTP/1.1`.

`http_version` also allows for an optional `request` argument, which is useful if writing detection that looks at both an HTTP client request and the HTTP server response to that request. More specifically, this `request` argument is used to signify that the `http_version` match or matches should apply to the HTTP version from the *request* if other parts of the rule are examining the *response*. More information on this topic can be found on this page.

Format:

```
http_version[:request];
```

Examples:

```
http_version;
content:"HTTP/1.1";
```

```
http_version;
content:"HTTP/1.0";
```

http_stat_code

The `http_stat_code` sticky buffer contains the status code field of an HTTP response status line. This includes values such as `200`, `403`, and `404`.

Format:

```
http_stat_code;
```

Examples:

```
http_stat_code;
content:"200";
```

```
http_stat_code;
content:"403";
```

http_stat_msg

The `http_stat_msg` sticky buffer contains the status text field of an HTTP response status line. This includes values such as `OK`, `Forbidden`, and `Not Found`.

Format:

```
http_stat_msg;
```

Examples:

```
http_stat_msg;
content:"OK";
```

```
http_stat_msg;
content:"Forbidden";
```

http_raw_request and http_raw_status

The sticky buffers `http_raw_request` and `http_raw_status` contain the unmodified first line of HTTP request and HTTP response messages, respectively. These rule options are a safety valve in case one needs to do something that can't otherwise be done with the specific start and status-line buffers.

http_raw_request

Format:

```
http_raw_request;
```

Examples:

```
http_raw_request; content:"GET /robots.txt HTTP/1.1";
```

```
http_raw_request; content:"POST /totally_not_vulnerable.php HTTP/1.1";
```

http_raw_status

Format:

```
http_raw_status;
```

Examples:

```
http_raw_status; content:"HTTP/1.1 200 OK";
```

```
http_raw_status; content:"HTTP/1.1 404 Not Found";
```

http_trailer and http_raw_trailer

HTTP allows header lines to appear after a chunked body ends, and those are referred to as "trailers". Snort makes these trailers available via the `http_trailer` and `http_raw_trailer` sticky buffers. These are identical to their `*_header` counterparts but apply to end headers instead.

Take the following chunked response, for example:

```
HTTP/1.1 200 OK
Host:abcip-host.local
Content-Type: text/plain
Transfer-Encoding: chunked
Trailer: Expires

7
Mozilla
9
Developer
7
Network
0
Expires: Wed, 21 Oct 2015 07:28:00 GMT
```

The `http_trailer` and `http_raw_trailer` buffers will set to the first Trailer header, which in this case is `Expires`:

```
[http_trailer]
00000000  45 78 70 69 72 65 73 3A 20 57 65 64 2C 20 32 31  Expires: Wed, 21
00000010  20 4F 63 74 20 32 30 31 35 20 30 37 3A 32 38 3A   Oct 2015 07:28:
00000020  30 30 20 47 4D 54                                00 GMT
```

Like with `http_header`, users can tell Snort to look at only a particular trailer field, with the optional `field` argument.

`http_trailer` and `http_raw_trailer` also allow for an optional `request` argument, which is useful if writing detection that looks at both an HTTP client request and the HTTP server response to that request. More specifically, this `request` argument is used to signify that the `http_trailer` or `http_raw_trailer` match or matches should apply to the HTTP trailers from the *request* if other parts of the rule are examining the *response*. More information on this topic can be found on this page.

http_trailer

Format:

```
http_trailer[:field field_name][,request];
```

Examples:

```
http_trailer;
content:"Expires:";
```

```
http_trailer;
content:"Expires:";
content:"2015", within 30;
```

http_raw_trailer

Format:

```
http_raw_trailer[:field field_name][,request];
```

Examples:

```
http_raw_trailer;
content:"Expires:";
```

```
http_raw_trailer;
content:"Expires:";
content:"2015", within 30;
```

http_true_ip

The `http_true_ip` sticky buffer provides the original IP address of the client sending the request as it's stored by a proxy in the request message headers. Specifically, it is the last IP address listed in the "X-Forwarded-For" and "True-Client-IP" headers, or any other custom "X-Forwarded-For-type" header. If a request contains multiple headers of this type, then the preference defined in the `xff_headers` configuration variable is taken into account when determining which header value to include in this buffer.

Format:

```
http_true_ip;
```

Examples:

```
http_true_ip;
content:"192.168.1.2";
```

```
http_true_ip;
content:"150.172.238.178";
```

http_version_match

The `http_version_match` rule option is used to match the HTTP version of an HTTP message against one from a *list* of versions. This option will check the version present in an HTTP request or status line and use that for comparision.

Valid values to include in the version list include `1.0`, `1.1`, `2.0`, `3.0`, `0.9`, `malformed`, and `other`. The list of versions should be wrapped in double quotes, and specifying multiple versions is done by separating each one with a space character.

The first five values mentioned above are used to match specific version numbers, while a `malformed` version is any version present in the request or status line that is not formatted like `[0-9].[0-9]` (i.e., single digit followed by a dot, followed by another single digit). An example of a malformed HTTP version would be `1.a`.

A version value that is `other` is a value that is formatted correctly but is not one of the five specific version values listed above. For example, an HTTP message that specifies HTTP version `8.4` would be considered `other`.

Additionally, HTTP messages that *falsely claim* to be one version via their request or status lines are also considered `other`. For example, an HTTP request that follows the `1.1` format but has `0.9` in its request line would be `other`.

Note that you can also optionally add `,request` to the rule option to match only against the version found in the *request* message, even when examining the response.

Format:

```
http_version_match:"version[ version]…"[,request];
```

Examples:

```
http_version_match:"0.9 1.0 1.1";
```

```
http_version_match:"2.0 3.0";
```

```
http_version_match:"other";
```

```
http_version_match:"malformed";
```

http_num_headers

The `http_num_headers` rule option is used to compare the number of HTTP headers present in an HTTP packet against a specific value.

Users can check whether the total number of headers present is less than, greater than, equal to, not equal to, less than or equal to, or greater than or equal to a specified integer value.

Additionally, users can also use `http_num_headers` to look for a count value that is between two numbers. This is done by setting the sign to `<>` or `<=>` and putting the minimum count to the left of the sign and the maximum count to the right of it. The `<>` case is for an exclusive min-max check, while the `<=>` is for an inclusive min-max check.

Valid `http_num_headers` number values are 0 through 65535 (inclusive).

Note that you can also optionally add `,request` to the rule option to only count the number of *request* headers, even when examining the response.

Format:

Single value comparison:

```
http_num_headers:[<|>|=|!|<=|>=]count[,request];
```

Range comparison:

```
http_num_headers:min_count{<>|<=>}max_count[,request];
```

Examples:

```
# Look for an HTTP packet containing more than 100 headers
http_num_headers:>100;
```

```
# Look for an HTTP packet containing exactly 100 headers
http_num_headers:100;
```

```
# Look for an HTTP packet containing somewhere between
# 50 and 100 headers (exclusive)
http_num_headers:50<>100;
```

http_num_trailers

The `http_num_trailers` rule option is used to compare the number of HTTP trailers present in an HTTP packet against a specific value.

Users can check whether the total number of trailers present is less than, greater than, equal to, not equal to, less than or equal to, or greater than or equal to a specified integer value.

Additionally, users can also use `http_num_trailers` to look for a count value that is between two numbers. This is done by setting the sign to `<>` or `<=>` and putting the minimum count to the left of the sign and the maximum count to the right of it. The `<>` case is for an exclusive min-max check, while the `<=>` is for an inclusive min-max check.

Valid `http_num_trailers` number values are 0 through 65535 (inclusive).

Format:

Single value comparison:

```
http_num_trailers:[<|>|=|!|<=|>=]count;
```

Range comparison:

```
http_num_trailers:min_count{<>|<=>}max_count;
```

Examples:

```
# Look for an HTTP packet containing more than 100 trailers
http_num_trailers:>100;
```

```
# Look for an HTTP packet containing exactly 100 trailers
http_num_trailers:100;
```

```
# Look for an HTTP packet containing somewhere between
# 50 and 100 trailers (exclusive)
http_num_trailers:50<>100;
```

http_num_cookies

The `http_num_cookies` rule option is used to compare the number of HTTP cookies present in an HTTP packet against a specific value.

This rule option works against HTTP requests and HTTP responses, and users can check for the number of cookies present in a `Cookie:` header or in a `Set-Cookie:` header.

For example, the following `Cookie:` header has two cookies:

```
GET /cookies HTTP/1.1
Host: www.cookie-store.com
Cookie: SID=31d4d96e407aad42; lang=en-US
```

Users can check whether the total number of cookies present is less than, greater than, equal to, not equal to, less than or equal to, or greater than or equal to a specified integer value.

Additionally, users can also use `http_num_cookies` to look for a count value that is between two numbers. This is done by setting the sign to `<>` or `<=>` and putting the minimum count to the left of the sign and the maximum count to the right of it. The `<>` case is for an exclusive min-max check, while the `<=>` is for an inclusive min-max check.

Valid `http_num_cookies` number values are 0 through 65535 (inclusive).

Note that you can also optionally add `,request` to the rule option to only count the number of *request* cookies, even when examining the response.

Format:

Single value comparison:

```
http_num_cookies:[<|>|=|!|<=|>=]count[,request];
```

Range comparison:

```
http_num_cookies:min_count{<>|<=>}max_count[,request];
```

Examples:

```
# Look for an HTTP packet containing more than 100 cookies
http_num_cookies:>100;
```

```
# Look for an HTTP packet containing exactly 100 cookies
http_num_cookies:100;
```

```
# Look for an HTTP packet containing somewhere between
# 50 and 100 cookies (exclusive)
http_num_cookies:50<>100;
```

http_header_test

The `http_header_test` rule option is used to perform various tests on a specific HTTP header field. The tests that can be performed in this option include a test to see whether the header field's value is numeric or not, a check to see whether a header field's value is within a given range, and a check to see whether a given field is absent.

This rule option takes a few different arguments, one of which is the name of the header field to run the tests against. Specifying this field is done with the `field` argument followed by the name of the field we want to test. The subsequent arguments are the specific tests to run against that particular field, and rule writers can specify multiple tests at once by separating them with commas.

As mentioned above, there are three tests that can be performed on a given header. First is the `numeric` test to check if a given field is numeric or not. This argument requires specifying either `true` or `false` after it to look for numeric headers and non-numeric headers, respectively.

Next is the `check` argument, which is used to check a numeric header against a given number range. This range is specified after `check`, and the format for specifying a range is described below.

Last is the `absent` argument, which simply checks if a given field is absent.

Note that you can also optionally add `,request` to the rule option to only perform tests against the *request* headers, even when examining the response.

Format:

```
http_header_test:field header_name[,numeric {true|false}][,check range][,absent]
[,request];
```

A range can either be a single value comparison:

```
[<|>|=|!|<=|>=]number
```

Or it can be a range between two integer values:

```
min_number{<>|<=>}max_number
```

Examples:

```
# check that the Content-Length header value is numeric
# and that its value is >40000000
http_header_test:field content-length,numeric true,check >40000000;
```

```
# check that the Content-Length header value is not numeric
http_header_test:field content-length,numeric false;
```

```
# check that the User-Agent field is absent
http_header_test:field user-agent,absent;
```

http_trailer_test

The `http_trailer_test` rule option is used to perform various tests on a specific HTTP trailer field. The tests that can be performed in this option include a test to see whether the trailer field's value is numeric or not, a check to see whether a trailer field's value is within a given range, and a check to see whether a given field is absent.

This rule option takes a few different arguments, one of which is the name of the trailer field to run the tests against. Specifying this field is done with the `field` argument followed by the name of the field we want to test. The subsequent arguments are the specific tests to run against that particular field, and rule writers can specify multiple tests at once by separating them with commas.

As mentioned above, there are three tests that can be performed on a given trailer. First is the `numeric` test to check if a given field is numeric or not. This argument requires specifying either `true` or `false` after it to look for numeric trailers and non-numeric trailers, respectively.

Next is the `check` argument, which is used to check a numeric trailer against a given number range. This range is specified after `check`, and the format for specifying a range is described below.

Last is the `absent` argument, which simply checks if a given field is absent.

Format:

```
http_trailer_test:field trailer_name[,numeric {true|false}][,check range]
[,absent];
```

A range can either be a single value comparison:

```
[<|>|=|!|<=|>=]number
```

Or it can be a range between two integer values:

```
min_number{<>|<=>}max_number
```

Examples:

```
# check that Expires trailer value is numeric
http_trailer_test:field expires,numeric true;
```

Rules Examining HTTP Request and Response Messages

The stateful nature of the Snort 3 HTTP inspector provides the ability for rule writers to create detection that targets both an HTTP client request and the HTTP server response to that request. Creating detection to do this simply requires including options that target both data from an HTTP response as well as data from an associated HTTP request.

The following Snort rule is a simple example showcasing what this might look like:

```
alert http (
    msg:"Rule examining a response and the request associated with that
response";
    flow:to_client,established;
    file_data;
    content:"pizza",fast_pattern;
    http_uri;
    content:"/index.php";
    classtype:misc-activity;
)
```

Here the `to_client` and `file_data` declarations tell us that the rule is targeting an HTTP response message, and the `http_uri` declaration tells Snort to get the request message associated with that response and see if the URI contains that content string.

Note that these kinds of rules must have the `fast_pattern` set on a response-specific rule option since Snort will always check the response before examining the associated request.

The following is a list of request-specific `http_*` buffers that can be used in this manner:

- `http_uri`
- `http_raw_uri`
- `http_cookie`
- `http_raw_cookie`
- `http_header`
- `http_raw_header`
- `http_trailer`
- `http_raw_trailer`
- `http_version`
- `http_true_ip`

The optional "request" argument

Certain HTTP rule options can apply to either an HTTP request or an HTTP response. For example, the `http_header` rule option can be used to match HTTP request headers *or* HTTP response headers. As a result, these rule options also allow for an optional `request` argument that signifies that the given rule option should apply to the HTTP *request* if the rule contains other options that examine the HTTP response.

For example, the following rule looks for the string "pizza" in an HTTP response message:

```
alert http (
    msg:"Rule examining just a response message";
    flow:to_client,established;
    file_data;
    content:"pizza",fast_pattern;
    classtype:misc-activity;
)
```

If we were to add an `http_header` match, then Snort would see that we are examining the HTTP response message and thus apply `http_header` to the response headers:

```
alert http (
    msg:"Rule examining just a response message";
    flow:to_client,established;
    file_data;
    content:"pizza",fast_pattern;
    http_header;
    content:"User-Agent: bad";
    classtype:misc-activity;
)
```

The above rule would look for a User-Agent string in the response headers, which likely would not match since "User-Agent" is a request header. However, by adding the `request` argument to `http_header`, we tell Snort to instead look at the *request headers* in the request associated with that particular response:

```
alert http (
    msg:"Rule examining a response and the request associated with that
response";
    flow:to_client,established;
    file_data;
    content:"pizza",fast_pattern;
    http_header:request;
    content:"User-Agent: bad";
    classtype:misc-activity;
)
```

> Note that the `request` argument is not needed for the `http_uri` rule shown above because a URI will only apply to request messages.

---

bufferlen

The `bufferlen` option enables rule-writers to check the length of a given buffer. Users can check that the length of a buffer equals an exact size, or they can use a mathematical equality/inequality sign to compare a buffer's length to a given size or sizes.

Declaring a `bufferlen` option is done with the `bufferlen` keyword, followed by a colon character, optionally followed by an equality/inequality sign, and lastly followed by the number to compare against. A `bufferlen` check can also be made relative to a previous cursor move by adding `,relative` after the number.

Users can also use `bufferlen` to look for a length value that is between two numbers. This is done by setting the sign to `<>` or `<=>` and putting the minimum number the left of the sign and the maximum number to the right of it. The `<>` case is for an *exclusive* min-max check, while the `<=>` is for an *inclusive* min-max check.

Valid `bufferlen` number values are 0 through 65535 (inclusive).

`bufferlen` will be tested against the `pkt_data` buffer unless some other sticky buffer is specified before it.

> **Note:** Snort 2's `urilen` option has been removed, and Snort 3 rule-writers should use the `http_uri` sticky buffer + `bufferlen` to check URI lengths.

Format:

Single value comparison:

```
bufferlen:[<|>|=|!|<=|>=]length[,relative];
```

Range comparison:

```
bufferlen:min_length{<>|<=>}max_length[,relative];
```

Examples:

```
# check that the packet payload contains more than 100 bytes
bufferlen:>100;
```

```
http_uri;
content:"/pizza.php?";
# check that the http_uri buffer contains exactly 10 bytes of data after the
content match
bufferlen:10,relative;
```

```
http_client_body;
# check that the client body contains between 2 and 10 bytes (inclusive)
bufferlen:2<=>10;
```

```
http_client_body;
# check that the client body contains between 2 and 10 bytes (exclusive)
bufferlen:2<>10;
```

isdataat

The `isdataat` rule option verifies the payload data exists at a specified location.

`isdataat` is specified with the `isdataat` keyword, followed by a colon character, followed by a number that signals where to look for packet data, and lastly followed optionally by `,relative` to tell Snort to look for data starting from a previous cursor move.

Users can also specify a negated `isdataat` check with `!` placed before the number to check that certain amount of data is not present at a specified location in the payload.

Valid `isdataat` numbers are 0 through 65535 (inclusive). This means that `isdataat:0` checks that there is at least one byte present after the current cursor location.

Format:

```
isdataat:[!]location[,relative];
```

Examples:

```
isdataat:100;
```

```
content:"USER";
# checks for at least 30 bytes after "USER" since valid isdataat numbers start
at 0
isdataat:29,relative;
content:!"|0a|", within 30;
```

dsize

The `dsize` rule option is used to test a packet's payload size. This option can be specified to look for a packet size that is less than, greater than, equal to, not equal to, less than or equal to, or greater than or equal to a specified integer value. This rule option can also be used to check that a payload size is between a range of numbers, using the `<>` range operator for an exclusive range check or the `<=>` for an inclusive one.

The valid `dsize` number range is 0-65535.

Format:

Single value comparison:

```
dsize:[<|>|=|!|<=|>=]size;
```

Range comparison:

```
dsize:min_size{<>|<=>}max_size;
```

Examples:

```
dsize:300<>400;
```

```
dsize:>10000;
```

```
dsize:<10;
```

pcre

The `pcre` rule option matches regular expression strings against packet data.

Regular expressions written for these two options use perl-compatible regular expression (PCRE) syntax, which can be read about here.

The regular expression written is enclosed in double quotes and must start and end with forward slashes. Users can specify optional "flags" after the ending forward slash to denote pcre modifiers. A table of these flags/modifiers can be found below in the two "Format" sections.

> **Note:** Snort 3 no longer contains HTTP-specific pcre flags since HTTP buffers are now sticky. Simply specify the `http_*` buffer before declaring `pcre` to evaluate the regular expression there.

A `pcre` rule option can be negated to tell Snort to alert only if that regular expression is not matched.

Format:

```
pcre:[!]"/pcre_string/[flag…]";
```

Flag	Explanation
`i`	case insensitive
`s`	include newlines in the dot metacharacter
`m`	By default, a pcre string is treated as one big line of characters, and '^' and '$' match at the beginning and ending of the string. When `m` is set, '^' and '$' match immediately following or immediately before any newline in the buffer, as well as the very start and very end of the buffer.
`x`	specifies that whitespace data characters in the pattern are ignored except when escaped or inside a character class
`A`	specifies the pattern must match only at the start of the buffer (same as specifying the '^' character)
`E`	sets '$' to match only at the end of the subject string
`G`	inverts the "greediness" of the quantifiers so that they are not greedy by default, but become greedy if followed by '?'

Flag	Explanation
`O`	overrides the configured pcre match limit and pcre match limit recursion for this expression
`R`	start the regex search from the end of the last match instead of start of buffer

Examples:

```
pcre:"/^file\x3a\x2f\x2f[^\n]{400}/mi";
```

```
http_uri;
content:"/vulnerable_endpoint.php",fast_pattern,nocase;
# pcre gets evaluated against data in the specified sticky buffer
pcre:"/[?&]interface=[\x60\x3b]/i";
```

> **Note:** Because regular expressions are relatively costly from a performance standpoint, rules that use a `pcre` should also have at least one `content` match to take advantage of Snort's fast-pattern engine.

---

regex

The `regex` rule option matches regular expressions against payload data via the hyperscan search engine.

One of the main advantages to using `regex` options over `pcre` options is the ability to use `regex` regular expressions as `fast_pattern` matches. Doing so, however, requires that the hyperscan libraries are installed and hyperscan is enabled in the Snort 3 Lua configuration like so:

```
search_engine = { search_method = "hyperscan" }
```

Like the `pcre` option, these regular expressions follow the perl-compatible regular expression (PCRE) syntax, are enclosed in double quotes, and must start and end with forward slashes.

Similar to `pcre`, `regex` options are evaluated against any sticky buffer that precedes it.

Regular expressions written for `regex` options have access to only a limited set of flags/modifiers, and those compatible flags are shown below.

Format:

```
regex:"/regex_string/[flag…]"[,fast_pattern][,nocase];
```

Flag	Explanation
`i`	case insensitive
`s`	include newlines in the dot metacharacter
`m`	By default, a pcre string is treated as one big line of characters, and '^' and '$' match at the beginning and ending of the string. When `m` is set, '^' and '$' match immediately following or immediately before any newline in the buffer, as well as the very start and very end of the buffer.
`R`	start the regex search from the end of the last match instead of start of buffer

Examples:

```
regex:"/^file\x3a\x2f\x2f[^\n]{400}/mi",fast_pattern;
```

```
http_uri;
# regex gets evaluated against data in the specified sticky buffer
regex:"/\x2fvulnerable_endpoint\x2ephp?interface=[\x60\x3b]/i",fast_pattern;
```

pkt_data

The `pkt_data` rule option sets the detection cursor to the start of normalized packet data.

Rules that do not include any buffer specifiers will check payload options against the `pkt_data` buffer by default. However, one might want to use the `pkt_data` buffer explicitly either for clarification purposes, or to return the cursor to normalized packet data after using some other sticky buffer.

It's important to note that unless `search_engine.detect_raw_tcp` is set to `true` in one's Snort configuration, not everything from a packet's payload gets placed in the `pkt_data` buffer. For example, if Snort detects traffic as HTTP, then elements like the URI and headers would not placed in a `pkt_data` buffer. Therefore, users will want to use this buffer to detect normalized packet payload bytes that are not available for detection in other buffers (such as `http_*` buffers).

This `search_engine.detect_raw_tcp` configuration option is set to `false` by default for performance reasons.

Format:

```
pkt_data;
```

Examples:

```
pkt_data;
content:"pizza", depth 5;
```

```
pkt_data;
content:"AAAAAA";
bufferlen:>1000;
```

raw_data

The `raw_data` rule option in Snort 3 replaces the old `rawbytes` keyword from Snort 2, and it sets the cursor to raw packet data. It is different from `pkt_data` in that it will ignore certain preprocessing and normalization done by Snort.

> **Note:** This option will likely not be used often as it was introduced in Snort 2 to remediate Telnet-related issues back in the day.

Format:

```
raw_data;
```

Examples:

```
# telnet NOP
raw_data;
content:"|FF F1|";
```

file_data

The `file_data` option sets the detection cursor to either the HTTP response body for HTTP traffic or file data sent via other application protocols that has been processed and captured by Snort's "file API". Data in this buffer can contain normalized and decoded data depending on the service used to send the file data, as well as the specific configurations enabled for the different service inspectors included in Snort.

Using this option to detect file data is as simple as specifying `file_data;` before any and and all payload options one wants to match there.

This rule option can be used several times in a rule if desired.

Services that support the `file_data` buffer include:

- `http`
- `pop3`
- `imap`
- `smtp`
- `ftp-data`
- `netbios-ssn`

The following sub-sections explain what the `file_data` buffer *could* contain for each of the above services. One should refer to each inspector's default configurations, and adjust them accordingly in the Snort Lua configuration.

HTTP

For HTTP traffic, the `file_data` buffer points to the normalized HTTP response body, and the specific normalizations that occur depend on one's Snort configuration. This includes things like the decompression of ZIP, SWF, and PDF files, the decoding of UTF-* encodings, JavaScript normalization, and deflate and gzip decompression. Additionally, Snort can also de-chunk chunked messages and place the de-chunked message body in the `file_data` buffer.

The default configurations can be seen with the following Snort command:

```
$ snort --help-module http_inspect
```

POP3/IMAP/SMTP

The `file_data` buffer for mail traffic also depends on one's specific configuration. By default, Snort will place email headers, decoded MIME attachments, and non-decoded MIME attachments in the `file_data` buffer. The decoders enabled by default in Snort include base64, quoted-printable, MIME, and Unix-to-Unix.

Note that if MIME decoding is disabled, then Snort will place the unencoded MIME data in the `file_data` buffer.

The default configurations can be seen with the following commands:

```
$ snort --help-module pop
$ snort --help-module imap
$ snort --help-module smtp
```

FTP data

For FTP traffic, the `file_data` buffer will contain any raw files sent over an FTP-data session.

SMB

Snort 3's DCE/RPC service inspector is aware of SMB request and response command codes and will process files seen in `SMB2 READ` responses and `SMB2 WRITE` requests. The `file_data` buffer is then set to the raw files processed in both of these two SMB message types.

The default configurations can be seen with the following commands:

```
$ snort --help-module dce_smb
```

Format:

```
file_data;
```

Examples:

```
alert http (
  …
  flow:to_client,established;
  file_data;
  content:"<script>var aaaaaaa";
  …
)
```

```
alert file (
  …
  flow:to_client,established;
  file_data;
  content:"MZ",depth 2;
  …
)
```

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 25 (
  …
  file_data;
  content:"decoded SMTP file here"
  …
)
```

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 445 (
  …
  content:"|FE|SMB";
  content:"|08|", distance 8, within 1;
  file_data;
  content:"MALWARE";
  …
)
```

js_data

The `js_data` option sets the detection cursor to the normalized JavaScript data buffer, which contains data normalized by the new "Enhanced Normalizer". Snort can detect whether JavaScript is present in the message body and then perform normalization on it if so.

Snort's "Enhanced Normalizer" is able to perform the following normalizations:

- Normalize whitespace
- Concatenate string literals
- Unify identifier and property names
- Expand escaped text

To see the default normalizations enabled, run the following command and look at the `http_inspect.js_*` configurations:

```
$ snort --help-module http_inspect
```

For example, consider the following JavaScript code:

```
<script>var o = {};

o.__defineGetter__('vuln');</script>
```

The Enhanced Normalizer will normalize the whitespace as well as unify the variable name to something predictable. If we look at the `js_data` buffer, we can see exactly that:

```
[js_data]
00000000  76 61 72 20 76 61 72 5F 30 30 30 30 3D 7B 7D 3B  var var_0000={};
00000010  76 61 72 5F 30 30 30 30 2E 5F 5F 64 65 66 69 6E  var_0000.__defin
00000020  65 47 65 74 74 65 72 5F 5F 28 27 76 75 6C 6E 27  eGetter__('vuln'
00000030  29 3B                                            );
```

Note that to use this buffer, the `http_inspect.js_normalization_depth` field must be set in one's Snort configuration:

```
http_inspect = { js_normalization_depth = depth }
```

> **Note:** The `js_data` option is still under active development, so rule-writers should use caution when using it in their rules.

Format:

```
js_data;
```

Examples:

```
js_data;
content:"=new Uint32Array(-1)|3B|";
```

```
js_data;
content:"var var_0000={}|3B|o.__defineGetter__(";
```

```
js_data;
content:"0xFFFFFFFF";
content:"-1";
bufferlen:<200;
```

vba_data

The `vba_data` rule option is used to set the detection cursor to the Microsoft Office Visual Basic for Applications (VBA) macros buffer.

VBA macros can be included in documents and spreadsheets to automate common Office tasks and operations, but they unfortunately can also be used by malicious actors to execute arbitrary code on an unsuspecting victim's machine. To be able to protect against malicious macros, Snort provides the `vba_data` sticky buffer to look at VBA macros present in Office documents that are sent over the wire.

Note that because VBA macros and Office documents are usually compressed, this option requires that the `decompress_zip` and `decompress_vba` options are enabled in one's Snort configuration. For example to enable it for the HTTP inspector, you would add the following lines to your configuration:

```
http_inspect.decompress_zip = true
http_inspect.decompress_vba = true
```

Format:

```
vba_data;
```

Examples:

```
vba_data;
content:"URLDownloadToFileA",nocase;
```

base64_decode and base64_data

Snort can decode base64-encoded data present in a packet's payload via the `base64_decode` option. If base64-encoded data is found, it gets decoded and the base64-decoded data is then placed in the `base64_data` sticky buffer.

base64_decode

The `base64_decode` option tells Snort exactly which bytes to decode as base64. It can be declared by itself with just `base64_decode;`, or it can take any combination of three optional arguments after `base64_decode:`. Those optional arguments are:

Argument	Description
`bytes` *`bytes`*	How many bytes to decode
`offset` *`offset`*	Where in the payload to look for the base64 data to decode
`relative`	Applies the offset relative to cursor instead of start of buffer

As noted above, all three arguments are optional. Omitting the `bytes` argument tells Snort to decode any base64-encoded data present until either the end of the buffer or the end of a present base64-encoded string. If the `offset` argument is omitted, Snort will look for base64 data either at the start of the buffer or the current cursor position (i.e., it implicitly sets `offset` to 0). And lastly, omitting `relative` tells Snort to look for the bytes relative to beginning of the currently-set buffer rather than the current cursor position.

Format:

```
base64_decode[:[bytes bytes][, offset offset][, relative]];
```

Examples:

```
base64_decode;
```

```
base64_decode:relative;
```

```
http_header;
content:"Authorization:",nocase;
base64_decode:bytes 12, offset 6, relative;
```

base64_data

The `base64_data` options sets the detection cursor to the beginning of the base64 decoded —provided `base64_decode` was in fact able to decode data.

This option does not take any arguments but requires that `base64_decode` be specified before it.

> **Note:** If matching content at the beginning of a base64 data buffer, users can use either `depth`/`offset` or `distance`/`within`.

Format:

```
base64_data;
```

Examples:

```
base64_decode:relative;
base64_data;
content:"AAAA";
```

```
http_header;
content:"Authorization:",nocase;
base64_decode:bytes 12, offset 6, relative;
base64_data;
content:"NTLMSSP", within 8;
```

byte_extract

The `byte_extract` keyword is used to read a some number of bytes from packet data and store the extracted byte or bytes into a named variable. This option does nothing by itself, and the extracted value should be used with other options later in the rule. The named variable can be used as arguments to any of the following options:

- `distance`, `within`, `offset`, or `depth` modifiers
- `byte_test`
- `byte_jump`
- `isdataat`

`byte_extract` is declared with the keyword, followed by a colon character, followed by three required arguments separated by commas: (1) number of bytes to extract, (2) the offset of the bytes to extract, and (3) the name of variable that will receive the extracted value. These three arguments MUST be specified in this exact order.

There are also a few additional optional arguments that can be added after the three required arguments, which are also separated by commas, and they are listed and described below in the formatting section.

> **Note:** The `byte_extract` option moves the detection cursor forward the number of bytes extracted.

Format:

```
byte_extract:count, offset, name[, relative][, multiplier multiplier] \
             [, endian][, string[, {dec|hex|oct}]][, align align][, dce] \
             [, bitmask bitmask];
```

Argument	Description
count	Number of bytes to pick up from the buffer (valid values include `1:10` if `string` argument is used and `1:4` if `string` argument is not used)
`offset`	Number of bytes into the buffer to start processing (valid values include `-65535:65535`)
`name`	Name of the variable to be used in other rule options
`relative`	Offset from cursor instead of start of buffer

Argument	Description
`multiplier` *`multiplier`*	Multiply the extracted value by the specified amount (valid values include `1:65535`)
`align` *`align`*	Round the number of converted bytes up to the next 2- or 4-byte boundary (valid values may be `2` or `4`)
`endian`	Set to either `big` or `little` to specify whether to process the data as little-endian or big-endian (extracted data is processed as big-endian by default)
`dce`	Use the DCE/RPC 2 inspector engine to determine the byte endianness
`string`	Extract bytes from packet that are stored in string format
`hex`	Convert the string bytes in the packet from a hexadecimal string (must be accompanied by `string`)
`oct`	Convert the string bytes in the packet from an octal string (must be accompanied by `string`)
`dec`	Convert the string bytes in the packet from a decimal string (the default option when `string` is set)
`bitmask` *`bitmask`*	Perform an AND bitwise operation with the specified bitmask on the extracted value before storing it in `name` (valid values are `1:`*`count`*)

Examples:

```
byte_extract:1, 0, str_offset;
byte_extract:1, 1, str_depth;
content:"bad stuff", offset str_offset, depth str_depth;
```

```
# multiplies the extracted byte by 8 and stores the result in "multiplier_ex1"
byte_extract:1, 0, multiplier_ex1, multiplier 8;
content:"AAAAA", within multiplier_ex1;
```

```
content:"MAGIC";
# extracts 4 bytes after "MAGIC", processes those bytes as little-endian,
# and stores the value in "field_sz"
byte_extract:4, 0, field_sz, relative, little;
content:"next field", distance field_sz;
```

```
http_header;
content:"Content-Length: ";
# extracts 4 bytes represented as a decimal string
# from the packet immediately after "Content-Length: "
byte_extract:4, 0, content_len, relative, string;
isdataat:!content_len;
```

```
# extracts 4 bytes represented as a hexadecimal string
# from the beginning of the packet
byte_extract:4, 0, hex_string_var, string, hex;
content:"BBBBB", distance hex_string_var;
```

byte_test

The `byte_test` rule option tests a byte field against a specific value with a specified operator. This option is able to test binary values right from the packet, and it can also convert string-representations of numbers (e.g., decimal, hexadecimal, and octal-representations) for testing purposes as well.

`byte_test` is declared with the keyword, followed by a colon character, followed by four required arguments separated by commas: (1) number of bytes to grab from the packet, (2) the operator to test against the bytes in the packet, (3) the value to test the bytes in the packet against, and (4) the offset of the bytes to grab. These four arguments MUST be specified in this exact order.

There are also a few optional arguments that can be declared after the four required arguments, which are also separated by commas. They are listed and described below.

Snort uses C operators for testing, and valid ones include `<`, `>`, `<=`, `>=`, `=`, `&`, and `^`.

The `&` operator does an AND bitwise test:

```
if (packet_data_bytes & value)
```

Conversely, the `^` does an XOR bitwise test:

```
if (packet_data_bytes ^ value)
```

A `byte_test` operator can also look for the negation of the test result by adding `!` before the selected operator.

> **Note:** A `byte_test` option *does not* move the detection cursor.

Format:

```
byte_test:count, [!]operator, compare, offset[, relative][, endian] \
          [, string[, {dec|hex|oct}]][, dce][, bitmask bitmask];
```

Argument	Description
`count`	Number of bytes to pick up from the buffer (valid values include `1:10`)
`operator`	Operation to test against the bytes in the packet (valid

Argument	Description
	operators include `<`, `>`, `<=`, `>=`, `=`, `&`, and `^`)
`compare`	Variable name or value to test the converted result against (valid values include `0:4294967295` and can be represented as decimal or hexadecimal)
`offset`	Variable name or number of bytes into the payload to start processing (valid values include `-65535:65535`)
`relative`	Offset from cursor instead of start of buffer
`endian`	Set to either `big` or `little` to specify whether to process the data as little-endian or big-endian (packet bytes are processed as big-endian by default)
`dce`	Use the DCE/RPC 2 inspector engine to determine the byte endianness
`string`	Pick up bytes from packet that are stored in string format
`hex`	Grab the string bytes in the packet from a hexadecimal string (must be accompanied by `string`)
`oct`	Grab the string bytes in the packet from an octal string (must be accompanied by `string`)
`dec`	Grab the string bytes in the packet from a decimal string (the default option when `string` is set)
`bitmask` *`bitmask`*	Perform an AND bitwise operation with the specified bitmask on the picked-up bytes before testing it against *`value`* (valid values are `1:`*`count`*)

Note: The `bitmask` argument result will be right-shifted by the number of bits equal to the number of trailing zeros in the mask.

Examples:

```
content:"ABCD", depth 4;
# grabs the 2 bytes immediately after "ABCD"
# that are represented as little-endian
# and checks that those bytes are greater than 0x7fff
byte_test:2, >, 0x7fff, 0, relative, little;
```

```
# grabs 2 bytes at offset 0, performs a
# bitwise AND with 0x3FF0, right-shifts the result by 4 bits,
# and compares the final result to 568 to see if they are equal
byte_test:2, =, 568, 0, bitmask 0x3FF0;
```

```
# grabs 4 bytes af offset 0, converts those bytes
# as a decimal string, and tests that the converted number
# is greater than 1234
byte_test:4, >, 1234, 0, string, dec;
```

```
content:"AAAA";
byte_extract:4, 0, a_sz, relative;
content:"BBBB";
# grabs 4 bytes right after "BBBB"
# and tests if those bytes are greater than
# the extracted bytes from earlier
byte_test:4, >, a_sz, 0, relative;
```

byte_math

The `byte_math` operation extracts bytes from the packet, performs a mathematical operation on the extracted value with specified value or existing variable, and stores the outcome in a new variable. These variables can be referenced later in the rule in the same places that `byte_extract` variables can be used.

`byte_math` is declared with the keyword, followed by a colon character, followed by five required arguments separated by commas: (1) "bytes" followed by the number of bytes to extract from the packet, (2) "offset" followed by the offset of the bytes to extract, (3) "oper" followed by the mathematical operation to perform on the extracted value, (4) "rvalue" followed by the value to use with the mathematical operation on the extracted value, and (5) "result" followed by the name of variable that will receive the final result.

There are also a few additional optional arguments that are listed and described below in the formatting section.

Valid math operations that can be used in this option include `+`, `-`, `*`, `/`, `<<`, and `>>`.

Note: The `byte_math` option *does not* move the detection cursor forward.

Note: `byte_math` operations are performed on unsigned 32-bit values, and this should be taken into consideration when writing rules to avoid integer wrap arounds.

Format:

```
byte_math:bytes count, offset offset, oper operator, rvalue rvalue, result
result \
          [, relative][, endian endian][, string[, {dec|hex|oct}]][, dce] \
          [, bitmask bitmask];
```

Argument	Description
count	Number of bytes to pick up from the buffer (valid values include `1:10` if `string` argument is used and `1:4` if `string` argument is not used)
offset	Variable name or number of bytes into the buffer to start processing (valid values include (valid values include `-65535:65535`)

Argument	Description
operator	Mathematical operation to perform on the extracted value (valid operations include +, -, *, /, <<, and >>)
rvalue	Value to use the mathematical operation with the extracted bytes (valid values include 1:4294967295 or an extracted byte_extract variable)
result	Name of the variable to store the result in
relative	Offset relative from cursor instead of start of buffer
endian *endian*	Set to either big or little to specify whether to process the data as little-endian or big-endian (packet bytes are processed as big-endian by default)
dce	Use the DCE/RPC 2 inspector engine to determine the byte endianness
string	Pick up bytes from packet that are stored in string format (must be followed by , hex, , oct, or , dec)
hex	Extract the string bytes in the packet from a hexadecimal string (must be accompanied by string)
oct	Extract the string bytes in the packet from an octal string (must be accompanied by string)
dec	Extract the string bytes in the packet from a decimal string (must be accompanied by string)
bitmask *bitmask*	Perform an AND bitwise operation with the specified bitmask on the extracted bytes *before* executing the math operation (valid values are 1:*count*)

Note: If using either the << or >> operation, then (1) the *count* value must be between 1 and 4 bytes, and (2) the *rvalue* must be less than 32.

Note: The bitmask argument result will be right-shifted by the number of bits equal to the number of trailing zeros in the mask.

Examples:

```
# extracts 2 bytes at offset 0, multiplies the extracted number by 10,
# and stores the result in the variable "area", and then uses
# the resulting variable in a `byte_test` option
byte_math:bytes 2, offset 0, oper *, rvalue 10, result area;
byte_test:2, >, area, 16;
```

```
http_header;
content:"Content-Length: ",nocase;
# extracts 8 decimal string bytes immediately after "Content-Length: " (if they
exist), multiplies
# the value by 10, and stores the result in "content_length"
byte_math:bytes 8, offset 0, oper *, rvalue 10, result content_length, relative,
string dec;
```

```
content:"ABCD";
# this is a valid byte_math option despite the odd ordering of the arguments
# extracts a single byte 10 bytes from the previous match, adds 10 to it,
# performs on it a bitwise AND with 0x12, and stores the result in "var"
byte_math:oper +, rvalue 10, offset 10, result var, bytes 1, bitmask 0x12,
relative;
```

byte_jump

The `byte_jump` rule option reads some number of bytes from the packet, converts them from their numeric representation if necessary, and moves that many bytes forward. By having an option that reads the length of a portion of data, rules can be written that skip over specific portions of length-encoded protocols and perform detection in very specific locations.

`byte_jump` is declared with the keyword, followed by a colon character, followed by just two required arguments separated by commas: (1) the number of bytes to grab from the packet and (2) the offset of the bytes to grab. These two arguments must be specified in this order, and there are also a few optional arguments that can be declared afterwards, which are also separated by commas. They are listed and described below.

Format:

```
byte_jump:count, offset[, relative][, multiplier multiplier][,endian] \
          [, string[, {dec|hex|oct}]][, align][, from_beginning][, from_end] \
          [, post_offset adjustment_value][, dce][, bitmask bitmask];
```

Argument	Description
count	Number of bytes to pick up from the buffer (valid values include `0:10` if `string` argument is used and 0:4 if `string` argument is not used)
offset	Variable name or number of bytes into the buffer to start processing (valid values include `-65535:65535`)
`relative`	Offset from cursor instead of start of buffer
`multiplier` *multiplier*	Multiply the grabbed value by the given amount (valid values include `1:65535`)
endian	Set to either `big` or `little` to specify whether to process the data as little-endian or big-endian (extracted data is processed as big-endian by default)
`align`	Round the number of converted bytes up to the next 32-bit boundary
`dce`	Use the DCE/RPC 2 inspector engine to determine the byte endianness
`string`	Pick up bytes from the packet that are stored in string format

Argument	Description
`hex`	Grab the string bytes in the packet from a hexadecimal string (must be accompanied by `string`)
`oct`	Grab the string bytes in the packet from an octal string (must be accompanied by `string`)
`dec`	Grab the string bytes in the packet from a decimal string (the default option when `string` is set)
`from_beginning`	Jump from the beginning of the packet payload instead of the current cursor location
`from_end`	Jump from the end of the packet payload instead of the current cursor location
`post_offset` *`adjustment_value`*	Number of bytes to skip forward or backward after the jump (valid values include a variable name or an integer in the following range: `-65535:65535`)
`bitmask` *`bitmask`*	Perform an AND bitwise operation with the specified bitmask on the grabbed value before jumping (valid values are `1:`*`count`*)

Note: The `bitmask` argument result will be right-shifted by the number of bits equal to the number of trailing zeros in the mask.

Examples:

```
# grab the 2 bytes at offset 0 and
# jump that many bytes forward
byte_jump:2,0;
```

```
content:"START";
byte_extract:1, 0, myvar, relative;
# grab a single byte 3 bytes after the previous
# byte_extract location, jump forward that number
# of bytes, and then adjust forward "myvar"
# number of bytes after the jump
byte_jump:1,3,relative,post_offset myvar;
content:"END", distance 6, within 3;
```

```
# grab 2 bytes at offset 1 from the current cursor location,
# bitmask AND the grabbed bytes by 0x03f0, jump the
# resulting number of bytes, and then adjust forward
# 2 number of bytes after the jump
byte_jump:2,1,relative,post_offset 2,bitmask 0x03f0;
byte_test:2,=,968,0,relative;
```

```
# this grabs 0 bytes so that it can
# jump backwards 6 bytes from the end of the
# current payload
byte_jump:0,0,from_end,post_offset -6;
content:"end..", distance 0, within 5;
```

ber_data and ber_skip

Snort 3 introduces two new rule options that are useful when writing detection for ASN.1 Basic Encoding Rules (BER), `ber_data` and `ber_skip`. Both options move the detection cursor with respect to BER data, but the difference lies in *where* the cursor is moved.

BER is a type-length-value (TLV) encoding format that encodes data elements in a stream containing (in this order):

- **Type**: typically a single byte that identifies the kind of data being represented
- **Length**: one or more bytes identifying the size of the value field
- **Value**: *length*-number of bytes of the specified type's data

For example, consider the following BER-encoded element (assume the below bytes are hex):

```
02 03 01 00 01
```

The *type* is `0x02`, which corresponds an `INTEGER` type. The *length* is `0x03`, which tells us that the *value* is 3-bytes long. Then we just parse those three bytes after the *length* field to get the integer *value*:

```
01 00 01
```

Encoded length fields

As mentioned above, the *length* field can either be a single byte, or it can span *multiple* bytes. More specifically, the *length* field will span multiple bytes if the *value* field contains more than 128 bytes. If that's the case, then bit 7 of the *length* field will be set, and bits 0-6 will indicate *how many* length bytes follow. For example, consider the start of this element:

```
30 82 04 9A 41 41 41 41 …
```

The first byte tells us the *type*, which in this case is `0x30`. However, notice that our *length* field is `> 0x7f`. This means that bit 7 of the length field is set to 1, which means that the *length* field requires more than one byte. So, we can look at the binary of `0x82` to figure out what bits 0-6 give us:

```
10000010
```

As we can see, we get *2* from reading bytes 0-6, and so we know that **two** length bytes follow. We then grab those two bytes – `0x049A` – to get the *length* of the subsequent *value*. Then we can just parse *length*-number of bytes as we normally would to get the *value*.

Now let's look at how Snort 3's `ber_data` and `ber_skip` rule options are used to work with this kind of data.

ber_data

The `ber_data` option moves the detection cursor to the *value* portion of a specified BER element.

Using this option is done with `ber_data:` followed by the element's *type* code. The code can be written in decimal (e.g., 1, 2, 3, 4) or in a hexadecimal (e.g., 0x1, 0x2, 0x3, 0x4), and the valid range of codes is `0:255`.

For example, let's assume we are looking at a packet starting with `02 03 01 00 01`. We can move to the *value* portion of the element with `ber_data:0x02;`.

- Detection cursor before the `ber_data:0x02;`:

  ```
  00000000  02 03 01 00 01
  ```

- Detection cursor after the `ber_data:0x02;`:

  ```
  00000000 01 00 01
  ```

Note that `ber_data` works relative to the cursor's current location in a given buffer. This means that if you don't move the cursor (for example with a `content` match) prior to using this option, then `ber_data` will try to pick up bytes at the start of the buffer.

Format:

```
ber_data:type;
```

Examples:

```
ber_data:0x02;
content:"|01 00 01|", within 3;
```

BER elements are commonly nested such that one element's *value* is the start of another element. Multiple `ber_data` options can be used to verify that a specific element is present in another:

```
ber_data:0x30;
# checks that 0x02 is the first element present in 0x30's value
ber_data:0x02;
```

ber_skip

The `ber_skip` option skips an entire BER element. It reads in the *length* value of the specified *type*, and jumps that many bytes, moving the cursor to the data immediately following the specified element.

Using this option is done with `ber_skip:` followed by the *type* code of the element to skip. The code can be written in decimal (e.g., 1, 2, 3, 4) or in hexadecimal (e.g., 0x1, 0x2, 0x3, 0x4), and the valid range of elements is `0:255`.

For example, let's assume we are looking an `INTEGER` element (identified by 0x02) followed by a `BOOLEAN` element (identified by 0x01): `02 01 FF 01 01 05`. We can skip the `INTEGER` element with `ber_skip:0x02;` to move the cursor to the start of the very next element, the `BOOLEAN`.

- Detection cursor before the `ber_skip:0x02;`:

  ```
  00000000  02 01 FF 01 01 05
  ```

- Detection cursor after the `ber_skip:0x02;`:

  ```
  00000000  01 01 05
  ```

By default, this rule option will return false if the specified BER element is not found. However, you can specify `,optional` after the element number to tell Snort that the BER element you want to skip is *optional*.

Note that `ber_skip` works relative to the cursor's current location in a given buffer. This means that if you don't move the cursor (for example with a `content` match) prior to using this option, then `ber_skip` will try to pick up bytes at the start of the buffer.

Format:

```
ber_skip:type[,optional];
```

Examples:

```
ber_skip:0x02;
content:"|01 01 05|", within 3;
```

```
# will still continue evaluating even if the 0x02 is not present
ber_skip:0x02,optional;
content:"|01 01 05|", within 3;
```

Using ber_data and ber_skip together

`ber_data` and `ber_skip` will often be used together to iterate through BER-encoded structures. LDAP traffic is one common use case. For the below example, consider the following LDAP packet:

```
00000000  30 33 02 01 02 63 2E 04 00 0A 01 02 0A 01 00 02  03...c..........
00000010  01 00 02 01 00 01 01 00 A0 19 A4 0C 04 02 63 6E  ..............cn
00000020  30 06 81 04 66 72 65 64 A3 09 04 02 64 6E 04 03  0...fred....dn..
00000030  6A 6F 65 30 00                                   joe0.
```

The below example will iterate through the various elements to get to the data of the first `0x0A` tag:

```
# LDAP is a common example
ber_data:0x30;
ber_skip:0x02,optional;
ber_data:0x63;
ber_skip:0x04;
ber_data:0x0a;
content:"|02|", within 1;
```

ssl_state and ssl_version

Snort features an SSL/TLS service inspector that inspects stream reassembled SSL and TLS traffic and keeps track of the records sent throughout a given session. It provides two options to rule writers, `ssl_state` and `ssl_version`, which enable checking for a specific SSL/TLS state and a specific SSL/TLS version, respectively.

These options are declared with the keyword, followed by a ':' character, and then lastly followed by one or more identifiers that are the states or versions to match. The valid identifiers are listed below in their respective sections.

Both options can also be "negated" by placing `!` after the colon to check that a given SSL/TLS packet does *not* match a version or state.

ssl_state

The `ssl_state` rule option tracks the **state** of the SSL/TLS session. The list of states that can be matched are `client_hello`, `server_hello`, `client_keyx`, `server_keyx`, and `unknown`. Multiple states can be specified in a single option, via a comma separated list, and are `OR`-ed together, meaning that if any of them match, the rule option evaluates to true.

Format:

```
ssl_state:[!]{client_hello|server_hello|client_keyx|server_keyx|unknown}
          [,{client_hello|server_hello|client_keyx|server_keyx|unknown}]…;
```

Examples

```
ssl_state:client_hello;
```

```
# client_keyx OR server_keyx
ssl_state:client_keyx,server_keyx;
```

```
# NOT server_hello
ssl_state:!server_hello;
```

ssl_version

The `ssl_version` rule option tracks the specific SSL/TLS version agreed upon by the two parties. The list of versions that can be matched are `sslv2`, `sslv3`, `tls1.0`, `tls1.1`, and `tls1.2`. More than one identifier can be specified, via a comma separated list, and are `OR`-ed together, meaning that if any of them match, the rule option matches.

Format:

```
ssl_version:[!]{sslv2|sslv3|tls1.0|tls1.1|tls1.2}
            [,{sslv2|sslv3|tls1.0|tls1.1|tls1.2}]…;
```

Examples

```
ssl_version:sslv3;
```

```
# TLS 1.0, TLS 1.1, OR TLS 1.2
ssl_version:tls1.0,tls1.1,tls1.2;
```

```
# NOT SSLv2
ssl_version:!sslv2;
```

DCE Specific Options

Snort features a DCE-RPC service inspector that keeps track of the DCE-RPC sessions, staying aware of (1) the DCE-RPC interfaces bound to, (2) the specific operation numbers (opnums) invoked, and (3) the stub data associated with DCE-RPC requests and responses. This service inspector also provides three rule options for each of these components, `dce_iface`, `dce_opnum`, and `dce_stub_data`, that are used to check packets against specific DCE-RPC requests or responses.

dce_iface

The `dce_iface` option is used to specify an interface UUID that a client has bound to. It is declared with `dce_iface:`, followed by `uuid`, and lastly followed by the actual UUID to match. Users can also optionally specify an interface version and operator to check that the DCE-RPC version is less than (`<`), greater than (`>`), equal to (`=`) or not equal to (`!`) the version specified. Valid version numbers include `0:65535`.

Also, by default the rule will only be evaluated against a first fragment (or the full request if not fragmented) since most rules are written to start at the beginning of a request. However, rule-writers can specify the optional `any_frag` argument to evaluate against middle and last fragments as well.

Format:

DCE-RPC interface versions are not required, but there are two ways to check for a specific version or versions. Both formats are below.

> Note: UUIDs are formatted like: *4hexbytes-2hexbytes-2hexbytes-2hexbytes-6hexbytes*

Single value version comparison:

```
dce_iface:uuid uuid [, version [<|>|=|!|<=|>=]version][, any_frag];
```

Range of versions comparison:

```
dce_iface:uuid uuid [, version min_version{<>|<=>}max_version][, any_frag];
```

Examples:

```
dce_iface:uuid 4b324fc8-1670-01d3-1278-5a47bf6ee188;
```

```
dce_iface:uuid 4b324fc8-1670-01d3-1278-5a47bf6ee188, version <2;
```

```
dce_iface:uuid 4b324fc8-1670-01d3-1278-5a47bf6ee188, any_frag;
```

```
dce_iface:uuid 4b324fc8-1670-01d3-1278-5a47bf6ee188, version =1, any_frag;
```

dce_opnum

The `dce_opnum` option enables users to check that a packet belongs to a specific DCE-RPC operation invocation. It is declared with `dce_opnum:` followed by either a single opnum number, an opnum number range, or an opnum list containing a combination of opnums and/or opnum ranges, and the entire argument is enclosed in double quotes. The rule option option matches if *any one* of the opnums specified match the opnum associated with a DCE/RPC request or response.

An opnum range is declared with a hyphen between the two numbers of the range (e.g., `1-10`), and multiple opnums or opnum ranges are separated by spaces.

> **Note:** Multiple opnums and/or opnum ranges were previously separated by commas. Separating them by spaces is a change new to Snort 3.

Format:

```
dce_opnum:"{opnum|min_opnum-max_opnum}[ {opnum|min_opnum-max_opnum}]…";
```

> **Note**: Valid opnum numbers are `0-65535`.

Examples:

```
dce_opnum:"15";
dce_opnum:"15-18";
dce_opnum:"15 18-20";
dce_opnum:"15 17 20-22";
```

dce_stub_data

The `dce_stub_data` option is a sticky buffer that is used to set the detection cursor to the beginning of the DCE/RPC stub data, regardless of preceding rule options. All ensuing rule options are checked for in this stub data buffer, and the first rule option following `dce_stub_data` should use absolute location modifiers if it is position-dependent.

This option takes no arguments and is set just with the keyword itself.

> **Note:** To leave the stub data buffer and return to the main payload buffer, use the "pkt_data" rule option after done inspecting `dce_stub_data` data.

Format:

```
dce_stub_data;
```

Examples:

```
dce_stub_data;
byte_test:4,>,128,8,dce;
```

```
dce_stub_data;
pcre:"/^(\x00\x00\x00\x00|.{12})/s";
```

SIP Specific Options

Snort 3's Session Initiation Protocol (SIP) inspector keeps track of SIP request and response messages, and it provides four rule options that let rule-writers look for specific SIP components: `sip_method`, `sip_header`, `sip_body`, and `sip_stat_code`.

sip_method

The `sip_method` rule option enables rule writers to check packets against a specific SIP method or multiple SIP methods. The list of methods that can be matched include `invite`, `cancel`, `ack`, `bye`, `register`, `options`, `refer`, `subscribe`, `update`, `join`, `info`, `message`, `notify`, and `prack`.

It's declared with `sip_method:` followed by one or more methods to look for. Multiple methods are specified via a comma separated list, and evaluation checks for any of the specified methods against any SIP methods extracted from a given packet.

This option can also be "negated" by placing `!` after the colon to check that a given SIP method does not match a particular method.

Format:

```
sip_method:[!]method[,method]…;
```

Examples:

```
sip_method:invite, cancel;
```

```
sip_method:!invite;
```

```
# check that a SIP message is not an INVITE AND also not a BYE
sip_method:!invite;
sip_method:!bye;
```

> **Note:** While SIP methods are case-sensitive, the arguments for this option are case-insensitive.

---

sip_header

The `sip_header` rule option is a sticky buffer that sets the detection cursor to the buffers containing extracted SIP headers from a SIP message request or response. This option takes no arguments and is declared before all other payload options that one wants to match against the SIP header portion of a message.

Format:

```
sip_header;
```

Examples:

```
sip_header;
content:"CSeq";
```

sip_body

The `sip_body` rule option is a sticky buffer that sets the detection cursor to a SIP message body. This option takes no arguments and is declared before all other payload options that one wants to match against the SIP body portion of a message.

Format:

```
sip_body;
```

Examples:

```
sip_body;
content:"v=0|0D 0A|", within 5;
```

sip_stat_code

The `sip_stat_code` option is used to check the status code of a SIP response packet.

This option is declared with `sip_stat_code:` followed by a status code or status codes to match. Multiple status codes are specified via a comma separated list, and evaluation checks for any of the specified codes are present in a given SIP response packet.

Valid stat codes are `1-9` and `100-999`.

Format:

```
sip_stat_code:stat_code[,stat_code]…;
```

Note: `1-9` codes mean to check for `1xx`, `2xx`, `3xx`, `4xx`, etc. responses.

Examples:

```
sip_stat_code:200;
```

```
# match any 2xx SIP status codes
sip_stat_code:2;
```

```
# match a SIP status code of 200 or 180
sip_stat_code:200, 180;
```

```
# match any 2xx SIP status codes or any 4xx SIP status codes
sip_stat_code:200, 180;
```

sd_pattern

The `sd_pattern` rule option detects and filters Personally Identifiable Information (PII) and other sensitive information, such as credit card numbers, U.S. Social Security numbers, and email addresses.

This rule option has just one required argument, which is the specific pattern to detect. Snort has three built-in patterns:

- `"credit_card"`,
- `"us_social"`
- `"us_social_nodashes"`,

If used in the rule option, Snort will replace those strings with the actual patterns themselves.

Snort users can also define their own patterns by including a PCRE-compatible regular expression as the argument instead. The pattern argument will be enclosed in double quotes, regardless if it's a built-in pattern or not. If a built-in one is used, however, Snort will replace it with the appropriate pattern and then validate that data in the packet matches it.

There exists also one optional argument that can be added after the pattern string: `threshold`. By default, `sd_pattern` looks for just one instance of the pattern before firing, but users can specify the `threshold` argument to require that there are multiple hits on that pattern in a single packet before firing. The format of this option can be seen in the format section below.

> **Note**: The `sd_pattern` rule option uses the hyperscan engine to perform pattern matching, meaning Snort must be built with the hyperscan libraries to use it.

Obfuscating PII

By default, Snort will not obfuscate credit card and social security numbers when outputting packet data to logs. However, users can enable obfuscation with the `ips.obfuscate_pii` configuration, which will mask all but the last four characaters of credit card and social security numbers. Enabling this is as easy as setting this configuration option to `true`, either in a Snort config or on the command line.

Here's an example showcasing how this works. Consider a credit card number that is "5555555555554444". Looking for `sd_pattern:"credit_card";` in a rule and outputting the alerts as `cmg` will produce the following output:

```
$ snort -q -r cc.pcap -R local.rules -A cmg
10/19-10:29:55.494550 [**] [1:1:0] "credit card found" [**] [Priority: 0] {TCP}
10.1.2.3:50284 -> 10.9.8.7:1234
02:01:02:03:04:05 -> 02:09:08:07:06:05 type:0x800 len:0x46
10.1.2.3:50284 -> 10.9.8.7:1234 TCP TTL:64 TOS:0x0 ID:3 IpLen:20 DgmLen:56
***A**** Seq: 0x2  Ack: 0x2  Win: 0x2000  TcpLen: 20

snort.raw[16]:
- - - - - - - - - - - -  - - - - - - - - - - - -  - - - - - - - -
35 35 35 35 35 35 35 35  35 35 35 35 34 34 34 34  55555555 55554444
- - - - - - - - - - - -  - - - - - - - - - - - -  - - - - - - - -
```

However, we can enable obfuscation and conceal the credit card number with the above configuration setting:

```
$ snort -q -r cc.pcap -R local.rules -A cmg --lua 'ips.obfuscate_pii = true'
10/19-10:29:55.494550 [**] [1:1:0] "credit card found" [**] [Priority: 0] {TCP}
10.1.2.3:50284 -> 10.9.8.7:1234
02:01:02:03:04:05 -> 02:09:08:07:06:05 type:0x800 len:0x46
10.1.2.3:50284 -> 10.9.8.7:1234 TCP TTL:64 TOS:0x0 ID:3 IpLen:20 DgmLen:56
***A**** Seq: 0x2  Ack: 0x2  Win: 0x2000  TcpLen: 20

snort.raw[16]:
- - - - - - - - - - - -  - - - - - - - - - - - -  - - - - - - - -
58 58 58 58 58 58 58 58  58 58 58 58 34 34 34 34  XXXXXXXX XXXX4444
- - - - - - - - - - - -  - - - - - - - - - - - -  - - - - - - - -
```

We should note, however, that log obfuscation is only applicable to CMG and Unified2 logging formats and that it doesn't support user defined PII patterns.

Format:

```
sd_pattern:"pattern"[, threshold count];
```

Examples

```
# matches all credit card patterns
sd_pattern:"credit_card";
```

```
# matches likely email addresses for the site "ourdomain.com"
sd_pattern:"\b\w+@ourdomain\.com\b";
```

```
# look for that string literal 300 times before firing
sd_pattern:"This is a string literal", threshold 300;
```

CVS

The `cvs` rule option is used to detect CVS vulnerabilities, and currently only one vulnerability is able to be detected: "Malformed Entry Modified and Unchanged flag insertion". This optional specifically looks for attempts to exploit a heap buffer overflow in CVS via malformed "Entry lines".

Each CVS vulnerability corresponds to an option name in Snort, and that option name is what's passed to the `cvs` option as an argument. But because there's currently only one CVS vulnerability that Snort can detect, there's only one option available: `invalid-entry`.

Format:

```
cvs:option;
```

Examples:

```
cvs:invalid-entry;
```

md5, sha256, and sha512

There exists three "hash" options that are each used to check some payload data against a hash value: `md5`, `sha256`, and `sha512`.

Each option requires two arguments: the actual hash enclosed in double quotes and the number of bytes from the payload to get the hash of. Additionally, users can also specify an `offset` *`offset`* argument to check for bytes at a specific location and/or the `relative` argument to hash the bytes relative to the current cursor location.

The offset value can be either an integer from `-65535:65535`, or it can be the name of a variable extracted with `byte_extract`.

Lastly, each hash option can be negated by placing a `!` before the hash value.

All three options are structured the same with the only difference being the option name.

> **Note**: Because hash values are typically represented in hex format, the hash values should also be enclosed in vertical bars (`|`) inside the quotes.

---

md5

Format:

```
md5:[!]"|hash|", length length[, offset offset][, relative];
```

Examples:

```
md5:"|7cf2db5ec261a0fa27a502d3196a6f60|", length 100, offset 0, relative;
```

sha256

Format:

```
sha256:[!]"|hash|", length length[, offset offset][, relative];
```

Examples:

```
sha256:"|9ed1515819dec61fd361d5fdabb57f41ecce1a5fe1fe263b98c0d6943b9b232e|", \
       length 100, offset 0, relative;
```

sha512

Format:

```
sha512:[!]"|hash|", length length[, offset offset][, relative];
```

Examples:

```
sha512:"|d8fefb4255686e6bf365b0f4763fad983f624beb7cbbb59b617c745c346b8db51a870fe
0a89cfba036cfbf2d011686b881acd8ab3278b318a304227ac2a99072|", \
       length 100, offset 0, relative;
```

GTP Specific Options

There exists a general-packet-radio-service tunneling protocol (GTP) service inspector that enables GTP control channel inspection in Snort. There are three GTP-specific rule options: `gtp_info`, `gtp_type`, and `gtp_version`.

gtp_info

The `gtp_info` rule option is used to check the "Information Element" field of GTP packets. This option takes in a single argument that can be either an integer or a string that maps to one of the info elements defined in the snort_defaults.lua file. Please note that the information element values do vary depending on the GTP version.

> **Note**: Both the integer and string info values should *not* be enclosed in double quotes when supplied to this option.

Format:

```
gtp_info:info_element;
```

Examples:

```
gtp_info:16;
```

```
gtp_info:packet_flow_id;
```

gtp_type

The `gtp_type` rule option is used to check for specific GTP Message Type values. This rule option takes in a "type list" that is one or more type values separated by spaces. The type values can either be an integer or a string that corresponds to one of the table entries in the snort_defaults.lua file. Please note that the type values do vary depending on the GTP version.

Additionally, the entire type list should also be enclosed in double quotes.

If more than one type is provided, Snort will trigger an event if *any* one of them is seen on the wire.

Format:

```
gtp_type:"type[ type]…";
```

Examples:

```
gtp_type:"255 16";
```

```
gtp_type:"255 create_pdp_context_request";
```

gtp_version

The `gtp_version` option is used to check GTP version numbers. This option takes just a single option—an integer that is the version to look for. Valid version number arguments include `0`, `1`, and `2`.

Format:

```
gtp_version:version_num;
```

Examples:

```
gtp_version:1;
```

```
gtp_version:2;
```

DNP3 Specific Options

The DNP3 service inspector in Snort 3 provides anomaly detection and IPS rule options for matching on data, functions, indications and objects in DNP3 messages. Configuration of the ports is handled by the binder. This service inspector offers four IPS rule options: `dnp3_func`, `dnp3_ind`, `dnp3_obj`, and `dnp3_data`.

dnp3_func

The `dnp3_func` rule option is used to check for DNP3 function codes. This rule option takes in either a function code number or a function code string. The supported function codes and their string and number mappings are defined in Snort's dnp3_map.cc file.

Format:

```
dnp3_func:function_code;
```

Examples:

```
dnp3_func:0;
```

```
dnp3_func:confirm;
```

dnp3_ind

The `dnp3_ind` rule option is used to check DNP3 indicator flags. This rule option takes in a "indicator flag list" that is one or more indicator flag values separated by spaces. The indicator flag values are string values corresponding to the table defined in Snort's dnp3_map.cc file.

Additionally, the entire flag list should also be enclosed in double quotes.

If more than one flag is provided, Snort will trigger an event if any one of them is seen on the wire.

Format:

```
dnp3_ind:"indicator_flag[ indicator_flag ]…";
```

Examples:

```
dnp3_ind:"config_corrupt event_buffer_overflow need_time";
```

```
dnp3_ind:"need_time";
```

dnp3_obj

The `dnp3_obj` rule option is used to check DNP3 object headers. This rule option enables users the ability to check for specific DNP3 data object types by checking for a specific **object group** and a specific **object group variation**. Thus, this option takes as arguments two things: a group number and a variation number.

Format:

```
dnp3_obj:group group_number, var varation_number;
```

Examples:

```
dnp3_obj:group 80, var 1;
```

```
dnp3_obj:group 60, var 2;
```

dnp3_data

The `dnp3_data` rule option sets the cursor to the beginning of DNP3 Application Layer data. This is a sticky buffer, so once set, all subsequent rule options will be evaluated against `dnp3_data` unless the cursor is reset or moved to another buffer.

Format:

```
dnp3_data;
```

Examples:

```
dnp3_data;
content:"badstuff";
```

CIP Specific Options

Snort 3 features a Common Industrial Protocol (CIP) service inspector that does CIP decoding and enables detection of various CIP fields. This service inspector is also able to decode Ethernet/IP (ENIP) protocols as well.

The CIP inspector provides 11 different rule options: `cip_attribute`, `cip_class`, `cip_conn_path_class`, `enip_command`, `enip_req`, `enip_rsp`, `cip_instance`, `cip_req`, `cip_rsp`, `cip_service`, and `cip_status`. Each of these options is broken down below.

This inspector is disabled by default, and so you will need to enable the `cip` service inspector and add entries to the binder in your Snort config to tell Snort which traffic to run the `cip` service inspector on. Below is a snippet of a Snort config showing these two things:

```
-- enable the CIP inspector
cip = { }
-- add CIP binder entries
binder =
{
    { when = { proto = 'tcp', ports = '44818', role = 'server' }, use = { type =
'cip' } },
    { when = { proto = 'udp', ports = '22222', role = 'server' }, use = { type =
'cip' } },
```

cip_attribute

The `cip_attribute` rule option is used to look for a single CIP attribute or multiple CIP attributes.

This rule option takes in one argument, which is either a single attribute value used with an optional comparison operator sign, or a range of attributes using one of two range operator signs. The range comparison is done by including either the `<>` or `<=>` sign and putting the minimum number the left of the sign and the maximum number to the right of it. The `<>` case is for an *exclusive* min-max check, while the `<=>` is for an *inclusive* min-max check.

Format:

Single value comparison:

```
cip_attribute:[<|>|=|!|<=|>=]attribute;
```

Range comparison:

```
cip_attribute:min_attribute{<>|<=>}max_attribute;
```

Examples:

```
cip_attribute:5;
```

```
cip_attribute:>5;
```

cip_class

The `cip_class` rule option is used to look for a single CIP class or multiple CIP classes.

This rule option takes in one argument, which is either a single class value used with an optional comparison operator sign, or a range of classes using one of two range operator signs. The range comparison is done by including either the `<>` or `<=>` sign and putting the minimum number the left of the sign and the maximum number to the right of it. The `<>` case is for an *exclusive* min-max check, while the `<=>` is for an *inclusive* min-max check.

Format:

Single value comparison:

```
cip_class:[<|>|=|!|<=|>=]class;
```

Range comparison:

```
cip_class:min_class{<>|<=>}max_class;
```

Examples:

```
cip_class:2;
```

```
cip_class:2<>5;
```

cip_conn_path_class

The `cip_conn_path_class` rule option is used to match a single CIP connection path class or multiple CIP connection path classes.

This rule option takes in one argument, which is either a single connection path class value used with an optional comparison operator sign, or a range of connection path class using one of two range operator signs. The range comparison is done by including either the `<>` or `<=>` sign and putting the minimum number the left of the sign and the maximum number to the right of it. The `<>` case is for an *exclusive* min-max check, while the `<=>` is for an *inclusive* min-max check.

Format:

Single value comparison:

```
cip_conn_path_class:[<|>|=|!|<=|>=]conn_path_class;
```

Range comparison:

```
cip_conn_path_class:min_conn_path_class{<>|<=>}max_conn_path_class;
```

The chosen *`conn_path_class`* values must be in the following range: `0:65535`.

Examples:

```
cip_conn_path_class:10;
```

```
cip_conn_path_class:!0;
```

cip_instance

The `cip_instance` rule option is used to match a single CIP instance value or multiple CIP instance values.

This rule option takes in one argument, which is either a single instance value used with an optional comparison operator sign, or a range of instance values using one of two range operator signs. The range comparison is done by including either the `<>` or `<=>` sign and putting the minimum number the left of the sign and the maximum number to the right of it. The `<>` case is for an *exclusive* min-max check, while the `<=>` is for an *inclusive* min-max check.

Format:

Single value comparison:

```
cip_instance:[<|>|=|!|<=|>=]cip_instance;
```

Range comparison:

```
cip_instance:min_cip_instance{<>|<=>}max_cip_instance;
```

The chosen `cip_instance` values must be in the following range: `0:4294967295`.

Examples:

```
cip_instance:33;
```

```
cip_instance:!33;
```

cip_req

The `cip_req` rule option is used to match CIP request packets.

This rule option takes no arguments.

Format:

```
cip_req;
```

Examples:

```
cip_req;
```

cip_rsp

The `cip_rsp` rule option is used to match CIP response packets.

This rule option takes no arguments.

Format:

```
cip_rsp;
```

Examples:

```
cip_rsp;
```

cip_service

The `cip_service` rule option is used to match a single CIP service value or multiple CIP service values.

This rule option takes in one argument, which is either a single service value used with an optional comparison operator sign, or a range of service values using one of two range operator signs. The range comparison is done by including either the `<>` or `<=>` sign and putting the minimum number the left of the sign and the maximum number to the right of it. The `<>` case is for an *exclusive* min-max check, while the `<=>` is for an *inclusive* min-max check.

Format:

Single value comparison:

```
cip_service:[<|>|=|!|<=|>=]service;
```

Range comparison:

```
cip_service:min_service{<>|<=>}max_service;
```

Note: The chosen `service` values must be in the following range: `0:127`.

Examples:

```
cip_service:10;
```

```
cip_service:127;
```

cip_status

The `cip_status` rule option is used to match a single CIP status value or multiple CIP status values.

This rule option takes in one argument, which is either a single status value used with an optional comparison operator sign, or a range of status values using one of two range operator signs. The range comparison is done by including either the `<>` or `<=>` sign and putting the minimum number the left of the sign and the maximum number to the right of it. The `<>` case is for an *exclusive* min-max check, while the `<=>` is for an *inclusive* min-max check.

Format:

Single value comparison:

```
cip_status:[<|>|=|!|<=|>=]status;
```

Range comparison:

```
cip_status:min_status{<>|<=>}max_status;
```

Note: The chosen `status` values must be in the following range: `0:255`.

Examples:

```
cip_status:0;
```

```
cip_status:!0;
```

enip_command

The `enip_command` rule option is used to match a single ENIP command value or multiple ENIP command values.

This rule option takes in one argument, which is either a single ENIP command value used with an optional comparison operator sign, or a range of ENIP command values using one of two range operator signs. The range comparison is done by including either the `<>` or `<=>` sign and putting the minimum number the left of the sign and the maximum number to the

right of it. The `<>` case is for an *exclusive* min-max check, while the `<=>` is for an *inclusive* min-max check.

Format:

Single value comparison:

```
enip_command:[<|>|=|!|<=|>=]enip_command;
```

Range comparison:

```
enip_command:min_enip_command{<>|<=>}max_enip_command;
```

Note: The chosen *command* values must be in the following range: `0:65535`.

Examples:

```
enip_command:5<>100;
```

```
enip_command:<7;
```

enip_req

The `enip_req` rule option is used to match ENIP response packets.

This rule option takes no arguments.

Format:

```
enip_req;
```

Examples:

```
enip_req;
```

enip_rsp

The `enip_rsp` rule option is used to match ENIP response packets.

This rule option takes no arguments.

Format:

```
enip_rsp;
```

Examples:

```
enip_rsp;
```

IEC 104 Specific Options

Snort 3 contains a service inspector for the IEC 60870-5-104 (IEC 104) protocol, a protocol that is distributed by the International Electrotechnical Commission (IEC) and provides a standardized method of sending telecontrol messages between central stations and outstations. Traffic using this protocol typically runs on TCP port 2404.

This inspector decodes IEC104 traffic and provides two rule options to let rule writers evaluate the (1) IEC104 APCI type and (2) IEC104 function code.

To be able to use the various `iec104` rule options, one must make sure to enable the inspector and add an approriate Binder entry in the Snort 3 config. For example:

```
-- enable the IEC104 inspector
iec104 = { }
-- add the IEC104 binder entry
binder =
{
    { when = { proto = 'tcp', ports = '2404' }, use = { type = 'iec104' }, },
}
```

iec104_apci_type

The `iec104_apci_type` rule option is used to verify that the message being processed is of the specified type. The argument passed to this rule option can be specified in one of three ways: the full type name, the lowercase type abbreviation, or the uppercase type abbreviation.

Format:

```
iec104_apci_type:
{u|U|unnumbered_control_function|s|S|numbered_supervisory_function \
    |i|I|information_transfer_format};
```

Examples:

```
iec104_apci_type:unnumbered_control_function;
```

```
iec104_apci_type:S;
```

```
iec104_apci_type:i;
```

iec104_asdu_func

The `iec104_asdu_func` rule option is used to verify that the message being processed is using the specified ASDU function. The argument passed to this rule option can be specified in one of two ways: the uppercase function name, or the lowercase function name.

A list of accepted function cames can be found in the Snort 3 code here.

Format:

```
iec104_asdu_func:function_name;
```

Examples:

```
iec104_asdu_func:M_SP_NA_1;
```

```
iec104_asdu_func:m_ps_na_1;
```

MMS Specific Options

IEC 61850 is a family of protocols, including the Manufacturing Message Specification (MMS), distributed by the International Electrotechnical Commission (IEC). It provides a standardized method of sending service messages between various manufacturing and process control devices, typically running on TCP port 102.

This inspector decodes MMS traffic and provides two rule options: `mms_func` and `mms_data`.

To be able to use the various `mms` rule options, one must make sure to enable the inspector and add an approriate Binder entry in the Snort 3 config. For example:

```
-- enable the mms wizard support
wizard = { curses = {'mms'}, }
-- enable the mms service inspector
mms = { }
-- add the mms binder entry
binder =
{
    { when = { service = 'mms' },              use = { type = 'mms' } },
    { use = { type = 'wizard' } }
}
```

mms_func

The `mms_func` rule option takes a single argument, a service name or a service number, and compares it with the `Confirmed Service` field in the MMS request or response packet that's being analyzed. The argument passed to this rule option can be specified in one of two ways: either as the full service name, or as a number that corresponds to a particular service. The full list of service names and their associated numbers can be found in the Snort3 source code here.

Format:

```
mms_func:{full_service_name|service_number};
```

Examples:

In this example the rule is using the `mms_func` rule option with a string argument containing the `Confirmed Service Request` service name on which to alert:

```
mms_func:get_name_list;
```

The following example also uses the `mms_func` rule option to alert on a `GetNameList` message, but this time an integer argument containing the function number is used:

```
mms_func:1;
```

mms_data

The `mms_data` rule option is used to set the detection cursor to the start of MMS PDU, bypassing all of the OSI encapsulation layers and allowing subsequent rule options to start processing from the MMS PDU field.

Format:

```
mms_data;
```

Examples:

In this example, the rule is using the `mms_data` rule option to set the cursor position to the beginning of the MMS PDU, and then checking the byte at that position for the value indicative of an `Initiate-Request` message:

```
mms_data;
content:"|A8|", depth 1;
```

Modbus Specific Options

Modbus is a protocol used in SCADA networks, and its traffic is typically seen on TCP port 502 (aka Modbus TCP). The Modbus service inspector decodes the Modbus protocol and provides three rule options that rule writers can use to evaluate Modbus traffic.

Those three options are `modbus_data`, `modbus_func`, and `modbus_unit`.

To be able to use the various `modbus` rule options, one must make sure to enable the inspector and add an approriate Binder entry in the Snort 3 config. For example:

```
-- enable the Modbus service inspector
modbus = {}
-- add the Modbus binder entry
binder =
{
    { when = { proto = 'tcp', ports = '502' }, use = { type = 'modbus' }, },
}
```

modbus_data

The `modbus_data` rule option is used to set the detection cursor to the start of Modbus data.

Format:

```
modbus_data;
```

Examples:

```
modbus_data;
content:"modbus stuff";
```

modbus_func

The `modbus_func` rule option is used to check for a particular Modbus function code or function name.

Rule writers can provide either the function code—an integer value—or the function's string name. The list of valid function strings can be found in the Snort 3 source code here. If providing the function name as the argument, one *should not* enclose the string in double quotes.

Format:

```
modbus_func:{function_name|function_code};
```

Examples:

```
modbus_func:5;
```

```
modbus_func:write_single_coil;
```

modbus_unit

The `modbus_unit` rule option is used to check for a particular Modbus unit identifier.

This rule option takes in a single argument, an integer in the following range: `0-255`.

Format:

```
modbus_unit:modbus_unit_id;
```

Examples:

```
modbus_unit:0;
```

```
modbus_unit:73;
```

S7CommPlus Specific Options

S7 Communication (S7Comm) and S7CommPlus are Siemens protocols that run between programmable logic controllers (PLCs) of the Siemens S7-300/400 and S7-1500/1200(v4.0) families. Traffic of this service typically runs on TCP port 102.

S7Comm and S7CommPlus data is sent in the payload of Connection Oriented Transport Protocol (COTP) data packets. Snort 3 features a S7CommPlus service inspector that provides three rule options: `s7commplus_content`, `s7commplus_func`, and `s7commplus_opcode`.

To be able to use the various `s7commplus` rule options, one must make sure to enable the inspector and add an approriate Binder entry in the Snort 3 config. For example:

```
-- enable the s7commplus wizard support
wizard = { curses = {'s7commplus'}, }
-- enable the s7commplus service inspector
s7commplus = { }
-- add the s7commplus binder entry
binder =
{
    { when = { service = 's7commplus' },       use = { type = 's7commplus' } },
    { use = { type = 'wizard' } }
}
```

s7commplus_content

The `s7commplus_content` rule option is used to set the detection cursor to the start of S7CommPlus content.

Format:

```
s7commplus_content;
```

Examples:

```
s7commplus_content;
content:"|01 02 03 04|";
```

s7commplus_func

The `s7commplus_func` rule option is used to check for a particular S7CommPlus function code.

This option takes in a single argument, either the name of a function code or the integer value of a function code. The currently supported function names and codes can be found in the Snort 3 source code here.

Format:

```
s7commplus_func:{function_code_name|function_code_number};
```

Examples:

```
s7commplus_func:explore;
```

```
s7commplus_func:0x586;
```

s7commplus_opcode

The `s7commplus_opcode` rule option is used to check for a particular S7CommPlus opcode code.

This option takes in a single argument, either the name of the opcode code or an integer value of the opcode code. The currently supported opcode names and codes can be found in the Snort 3 source code here.

Format:

```
s7commplus_opcode:{opcode_code_name|opcode_code_number};
```

Examples:

```
s7commplus_opcode:request;
```

```
s7commplus_opcode:0x31;
```

Non-Payload Detection Rule Options

The non-payload rule options look for non-payload-related data. All of these options are described in detail in subsequent sections, but essentially, these options enable users to evaluate parts of a packet other than the TCP and UDP data sections, as well as keep track of packet states for future evaluation.

Quick Reference

keyword	description
fragoffset	`fragoffset` looks for specific IP header fragment offset values
ttl	`ttl` looks for specific IP header TTL values
tos	`tos` looks for specific IP header ToS values
id	`id` looks for specific IP header ID values
ipopts	`ipopts` looks for the prescence of specific IP options
fragbits	`fragbits` checks the IP header for fragmentation or reserved bits
ip_proto	`ip_proto` looks for specific IP header protocol fields
flags	`flags` checks the TCP header for specific TCP flag bits
flow	`flow` checks the session properties associated with given packet
flowbits	`flowbits` is used to set and test arbitrary boolean flags to track states during a transport protocol session
file_type	`file_type` is used to create rules that are constrained to a specific file type, a specific version of a file type
seq	`seq` looks for specific TCP header sequence numbers
ack	`ack` looks for specific TCP header acknowledgment numbers
window	`window` looks for specific TCP header window sizes
itype	`itype` looks for specific ICMP type values
icode	`icode` looks for specific ICMP code values
icmp_id	`icmp_id` looks for specific ICMP ID values
icmp_seq	`icmp_seq` looks for specific ICMP sequence values

keyword	description
rpc	`rpc` looks for specific SUNRPC CALL request parameters
stream_reassemble	`stream_reassemble` is used to enable or disable TCP stream reassembly on matching traffic
stream_size	`stream_size` is used to perform stream size checking

fragoffset

The `fragoffset` rule option is used to check that the IP fragment offset value in the IP header is less than, greater than, equal to, not equal to, less than or equal to, or greater than or equal to a specified integer value. This rule option can also check that the header's fragment's offset value is between a range of numbers, using the `<>` range operator for an exclusive range check or the `<=>` for an inclusive one.

Format:

Single value comparison:

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

Range comparison:

```
fragoffset:fragoffset_min{<>|<=>}fragoffset_max;
```

Examples:

```
# Check that the fragoffset equals 0
fragoffset:0;
```

```
# Check that the fragoffset does not equal 0
fragoffset:!0;
```

ttl

The `ttl` rule option is used to check that the IP time-to-live (TTL) value in the IP header is less than, greater than, equal to, not equal to, less than or equal to, or greater than or equal to a specified integer value. This rule option can also check that the header's TTL value is between a range of numbers, using the `<>` range operator for an exclusive range check or the `<=>` for an inclusive one.

Format:

Single value comparision:

```
ttl:[<|>|=|!|<=|>=]ttl;
```

Range comparison:

```
ttl:ttl_min{<>|<=>}ttl_max;
```

Examples:

```
# Check that the TTL equals 64
ttl:64;
```

```
# Check that the TTL does not equal 64
ttl:!64;
```

```
# Check that the TTL is less than 3
ttl:<3;
```

```
# Check that the TTL is between 3 and 5 (inclusive)
ttl:3<=>5;
```

```
# Check that the TTL is equal to 5
ttl:=5;
```

tos

The `tos` rule option is used to check an IP header's type of service (ToS) value is less than, greater than, equal to, not equal to, less than or equal to, or greater than or equal to a specified integer value. This rule option can also check that the header's ToS value is between a range of numbers, using the `<>` range operator for an exclusive range check or the `<=>` for an inclusive one.

This option is useful to detect things like the "bubonic" DoS tool.

Format:

Single value comparison:

```
tos:[<|>|=|!|<=|>=]tos;
```

Range comparison:

```
tos:tos_min{<>|<=>}tos_max;
```

Examples:

```
# Check that the ToS value does not equal 4
tos:!4;
```

```
# Check that the ToS value equals 4
tos:4;
```

id

The `id` rule option is used to check an IP header's identification (ID) field value is less than, greater than, equal to, not equal to, less than or equal to, or greater than or equal to a specified integer value. This rule option can also check that the header's ID value is between a range of numbers, using the `<>` range operator for an exclusive range check or the `<=>` range operator for an inclusive one.

Format:

Single value comparison

```
id:[<|>|=|!|<=|>=]id;
```

Range comparison

```
id:id_min{<>|<=>}id_max;
```

Examples:

```
# Check for an ID value that equals 31337
id:31337;
```

```
# Check for an ID value that is greater than 31337
id:>31337;
```

ipopts

The `ipopts` rule option is used to check if a specified IP option is present in an IP header.

There are 11 possible `ipopts` arguments to choose from, and an `ipopts` option can only have one argument. These options include the following:

- `rr`: Record Route
- `eol`: End of Options List
- `nop`: No Operation
- `ts`: Time Stamp
- `sec`: Security
- `esec`: Extended Security
- `lsrr`: Loose Source Routing
- `lsrre`: Loose Source Routing (For MS99-038 and CVE-1999-0909)
- `ssrr`: Strict Source Route
- `satid`: Stream ID
- `any`: Any IP options are set

Format:

```
ipopts:{rr|eol|nop|ts|sec|esec|lsrr|lsrre|ssrr|satid|any};
```

Examples:

```
# Match packets with IP headers containing the
# Record Route option
ipopts:rr;
```

```
# Match packets with IP headers containing any option
ipopts:any;
```

fragbits

The `fragbits` option checks the IP header to see if specific fragmentation and reserved bits are set or not.

Rule writers can check for the following bits:

- `M` -> More Fragments
- `D` -> Don't Fragment
- `R` -> Reserved Bit

Additionally, rule options can include one of the following optional modifiers to change how the criteria is evaluated:

- `+` -> Match on the specified bits, plus any others
- `*` -> Match if any of the specified bits are set
- `!` -> Match if the specified bits are not set

Format:

```
fragbits:[modifier]fragbit…;
```

Examples:

```
## Checks if only the More Fragments bit is set
fragbits:M;
```

```
# Checks if the More Fragments bit and the
# Do not Fragment bit are set, plus any others
fragbits:+MD;
```

ip_proto

The `ip_proto` rule option is used to check the IP header protocol field against an IP protocol number or name. Valid protocol numbers and names can be found on the IANA's "Protocol Numbers" page here: https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml.

Rule writers can also use the `!`, `<`, or `>` operators to check for IP header protocol numbers that are not equal to, less than, or greater than the specified protocol number or protocol name's corresponding number.

Format:

```
ip_proto:[!|>|<]{proto_number|proto_name};
```

Examples:

```
# Check that the IP protocol field equals IGMP
ip_proto:igmp;
```

```
# Check that the IP protocol field does not equal TCP
ip_proto:!tcp;
```

flags

The `flags` rule option checks to see if the specified flag bits are set in the TCP header.

The following flag bits may be checked:

- `F` -> FIN (Finish)
- `S` -> SYN (Synchronize sequence numbers)
- `R` -> RST (Reset the connection)
- `P` -> PSH (Push buffered data)
- `A` -> ACK (Acknowledgement)
- `U` -> URG (Urgent pointer)
- `C` -> CWR (Congestion window reduced)
- `E` -> ECE (ECN-Echo)
- `0` -> No TCP flags set

One can look for multiple flags at once by specifying more than one flag character. Doing this tells Snort to look for *all* of the flags specified in the option.

Additionally, rule options can also include one of the following optional modifiers to change how the criteria is evaluated:

- `+` -> match any of the specified bits, plus any others
- `*` -> match if any of the specified bits are set
- `!` -> match if the specified bits are not set

Rule writers can also specify flags to *ignore* by placing a comma after the initial set of flags followed by a flag character or flag characters to ignore.

Format:

```
flags:[modifier]test_flag…[,mask_flag…];
```

Examples:

```
# Check for TCP packets where only the SYN flag is set
flags:S;
```

```
# Check for TCP packets where only the SYN and ACK flags is set
flags:SA;
```

```
# Check for TCP packets where the SYN and/or ACK flags are set
flags:*SA;
```

```
# Check if the SYN and FIN bits are set, ignoring the CWR and ECN bits
flags:SF,CE;
```

flow

The `flow` option is used to check session properties of a given packet. There are four main property categories that one can check with this option:

- The direction of the packet, specifically whether it's from a client to a server or from a server to a client
- Whether the packet is part of an established TCP connection or not
- Whether the packet is a reassembled packet or not
- Whether the packet is a rebuilt frag packet or not

Each of these property categories has a few different arguments that can be specified in a `flow` option, but only one property from each "category" can be included in a single option. All possible arguments are listed and described in the below table, and specifying multiple properties is done by adding commas in between them. The order is not important.

Argument	Description
`to_client`	Match on server responses
`to_server`	Match on client requests
`from_client`	Match on client requests
`from_server`	Match on server responses
`established`	Match only on established TCP connections
`not_established`	Match only when no TCP connection is established
`stateless`	Match regardless of stream state
`no_stream`	Match only on non-reassembled packets
`only_stream`	Match only on reassembled packets
`no_frag`	Match only on fragmented packets
`only_frag`	Match only on de-fragmented packets

Format:

```
flow:[{established|not_established|stateless}] \
     [,{to_client|to_server|from_client|from_server}] \
     [,{no_stream|only_stream}] \
     [,{no_frag|only_frag}];
```

Examples:

```
flow:to_server,established;
```

```
flow:to_client,established;
```

```
flow:stateless;
```

flowbits

The `flowbits` rule option is used to set and test arbitrary boolean flags to track states throughout the entirety of a transport protocol session (UDP or TCP).

There are five `flowbit` operations, all of which are listed below, that rule writers can use to track states. These are described in the following table:

Argument	Description
`set`	Sets the specified states for the current flow
`unset`	Unsets the specified states for the current flow
`isset`	Checks if the specified states are set
`isnotset`	Checks if the specified states are not set
`noalert`	Cause the rule to not generate an alert, regardless of the rest of the detection options

Setting and checking flowbits

The first four operations, `set`, `unset`, `isset`, and `isnotset`, are used to track states throughout a transport protocol session. These four operations require an additional argument, the flowbit flag *name*, which is the name of the flag to be associated with that particular state.

Tracking states is done properly by creating at least two rules: (1) a "flowbit setter" rule that tells Snort to set a flag if the other conditions in it are met and (2) a "flowbit checker" rule to check whether that particular flag has been set or not set previously in the current transport protocol session, using that as one of its conditions. Rule writers can also "unset" a flag if there's something in a particular packet to warrant such a thing.

Lastly, rule writers can also set and evaluate *multiple* bits at once using the `&` and `|` operators. However, if setting or unsetting multiple flowbit flags with one `flowbit` option, one must use `&`.

Format:

Setting or unsetting bits

```
flowbits:{set|unset},bit[&bit]…;
```

Checking if any bit is set

```
flowbits:{isset|isnotset},bit[|bit]…;
```

Checking if all bits are set

```
flowbits:{isset|isnotset},bit[&bit]…;
```

> **Note**: The names of the flowbit names should be limited to alphanumeric strings and can include periods, dashes, or underscores.

Examples:

```
# this example sets a "logged_in" flag that is used to denote
# that an IMAP login has occurred
alert tcp any 143 -> any any (
    msg:"IMAP login";
    content:"OK LOGIN";
    flowbits:set,logged_in;
    flowbits:noalert;
)

# this rule then will only "alert" if "LIST" is found in a packet AND
# the "logged_in" flag has been set previously during the
# current transport protocol session
alert tcp any any -> any 143 (
    msg:"IMAP LIST";
    content:"LIST";
    flowbits:isset,logged_in;
)
```

```
# check that flag1 AND flag2 have been set previously in the
# current transport protocol session
flowbits:isset,flag1&flag2;
```

```
# check that flag1 OR flag2 have been set previously in the
# current transport protocol session
flowbits:isset,flag1|flag2;
```

```
# set the flowbits, flag1 AND flag2, for the current transport
# protocol session
flowbits:set,flag1&flag2;
```

```
# unset the flowbits, flag1 AND flag2, for the current transport
# protocol session
flowbits:unset,flag1&flag2;
```

The noalert flowbit

The last flowbit operation is `noalert`. Invoking this operation simply tells Snort not to generate an alert for that particular rule. There is not bit name required for this one.

This operation is most commonly used in flowbit setter rules since those are usually just a precursor to what one actually wants to detect.

Format:

```
flowbits:noalert;
```

Examples:

```
flowbits:noalert;
```

file_type

The `file_type` rule option is used write rules that are constrained to a given file type, a specific version of a file type, several different file types, or several file types of varying versions.

Rule writers can use this option by specifying either a single file type name, a file type name and a specific version, or multiple file type names with optional version numbers. File type version numbers are specified with a comma followed by the specific version number to look for, and multiple type names are then separated by a single space character.

The entire `file_type` argument should be wrapped in double quotes if specifying a version as well.

Use of file identification rules

It's important to note that successful use of `file_type` requires the prescence of "file identification rules" that leverage the Snort rule engine to define the matches that indicate a particular file type is present in the traffic currently being inspected. Open source Snort 3 includes definitions for the most common file types, such as EXE, PDF, and Office files, and those are located in file_magic.rules.

These identification rules are created as `file_id` rules, and more info about them and their syntax can be found in the file_id manual page.

Format:

```
file_type:"type_name[,type_version]…[ type_name[,type_version]…]…";
```

> **Note**: This is one of the few rule options where whitespace *does* matter.

Examples:

```
# look for PDF files
file_type:"PDF";
```

```
# look for version 1.6 PDF files
file_type:"PDF,1.6";
```

```
# look for version 1.6 or version 1.7 PDF files
file_type:"PDF,1.6,1.7";
```

```
# look for MSEXE, MSCAB, or MSOLE files
file_type:"MSEXE MSCAB MSOLE2";
```

seq

The `seq` rule option is used to check that the TCP header sequence number is less than, greater than, equal to, not equal to, less than or equal to, or greater than or equal to a specified integer value. This rule option can also check that the sequence number is between a range of numbers, using the `<>` range operator for an exclusive range check or the `<=>` for an inclusive one.

Format:

Single value comparison:

```
seq:[<|>|=|!|<=|>=]seq;
```

Range comparison:

```
seq:seq_min{<>|<=>}seq_max;
```

Examples:

```
# Look for a sequence number of 0
seq:0;
```

ack

The `ack` rule option is used to check that the TCP header acknowledgment number is less than, greater than, equal to, not equal to, less than or equal to, or greater than or equal to a specified integer value. This rule option can also check that the acknowlegment number is between a range of numbers, using the `<>` range operator for an exclusive range check or the `<=>` for an inclusive one.

Format:

Single value comparison:

```
ack:[<|>|=|!|<=|>=]ack;
```

Range comparison:

```
ack:ack_min{<>|<=>}ack_max;
```

Examples:

```
# Look for an acknowledgment number of 0
ack:0;
```

```
# Look for an acknowledgment number between
# 0 and 1000 (inclusive)
ack:0<=>1000;
```

window

The `window` rule option is used to check that the TCP header window number is less than, greater than, equal to, not equal to, less than or equal to, or greater than or equal to a specified integer value. This rule option can also check that the window number is between a range of numbers, using the `<>` range operator for an exclusive range check or the `<=>` for an inclusive one.

Format:

Single value comparison:

```
window:[<|>|=|!|<=|>=]window;
```

Range comparison:

```
window:window_min{<>|<=>}window_max;
```

Examples:

```
# Check for a Window size that equals 55808
window:55808;
```

```
# Check for a Window size that is between 0 and 55808 (exclusive)
window:0<>55808;
```

itype

The `itype` rule option is used to check that a packet's ICMP type is less than, greater than, equal to, not equal to, less than or equal to, or greater than or equal to a specified integer value. This rule option can also check that an ICMP type is between a range of numbers, using the `<>` range operator for an exclusive range check or the `<=>` for an inclusive one.

Format:

Single value comparison:

```
itype:[<|>|=|!|<=|>=]itype;
```

Range comparison:

```
itype:itype_min{<>|<=>}itype_max;
```

Examples:

```
# Check for an ICMP type greater than 30
itype:>30;
```

icode

The `icode` rule option is used to check that an ICMP code value is less than, greater than, equal to, not equal to, less than or equal to, or greater than or equal to a specified integer value. This rule option can also check that ICMP code value is between a range of numbers, using the `<>` range operator for an exclusive range check or the `<=>` for an inclusive one.

Format:

Single value comparison:

```
icode:[<|>|=|!|<=|>=]icode;
```

Range comparison:

```
icode:icode_min{<>|<=>}icode_max;
```

Examples:

```
# Check for an ICMP code value greater than 30
icode:>30;
```

```
# Check for an ICMP code greater than or equal to zero and less than 30
icode:0<=>30;
```

icmp_id

The `icmp_id` rule option is used to check that an ICMP ID value is less than, greater than, equal to, not equal to, less than or equal to, or greater than or equal to a specified integer value. This rule option can also check that ICMP ID value is between a range of numbers, using the `<>` range operator for an exclusive range check or the `<=>` for an inclusive one.

Format:

Single value comparison:

```
icmp_id:[<|>|=|!|<=|>=]icmp_id;
```

Range comparison:

```
icmp_id:icmp_id_min{<>|<=>}icmp_id_max;
```

Examples:

```
# Check for an ICMP ID value of 0
icmp_id:0;
```

icmp_seq

The `icmp_seq` rule option is used to check that an ICMP sequence value is less than, greater than, equal to, not equal to, less than or equal to, or greater than or equal to a specified integer value. This rule option can also check that ICMP Sequence value is between a range of numbers, using the `<>` range operator for an exclusive range check or the `<=>` for an inclusive one.

Format:

Single value comparison:

```
icmp_seq:[<|>|=|!|<=|>=]icmp_seq;
```

Range comparison:

```
icmp_seq:icmp_seq_min{<>|<=>}icmp_seq_max;
```

Examples:

```
# Check for an ICMP sequence value of 0
icmp_seq:0;
```

rpc

The `rpc` rule option is used to look for specific RPC application/program numbers, version numbers, and procedure numbers in SUNRPC CALL requests. The RPC application number is a required argument, but the version and procedure numbers are optional. Rule writers can either specify specific version and procedure numbers, or they can use the `*` character to look for any version number or any procedure number.

Format:

```
rpc:application_number[,{version_number|*}][,{procedure_number|*}];
```

Examples:

```
# Look for the 100000 application number, any version number,
# and procedure number 3
rpc:100000, *, 3;
```

stream_reassemble

The `stream_reassemble` rule option is used to enable or disable TCP stream reassembly on matching traffic.

This rule option takes two required arguments: (1) whether to enable or disable stream reassembly and (2) whether the action applies to client traffic, server traffic, or both client and server traffic. Rule writers can have two optional arguments to choose from: (1) `noalert` to prevent alerting on matching traffic and (2) `fastpath` to ingore the rest of the session.

Format:

```
stream_reassemble:action {enable|disable}, direction {server|client|both} \
                  [, noalert][, fastpath];
```

Example:

```
flow:to_client,established;
content:"ABCDEF";
# this will disable stream reassembly on client traffic when
# "ABCDEF" is seen in server response traffic and will also
# prevent the rule from generating an alert
stream_reassemble:action disable, direction client, noalert;
```

stream_size

The `stream_size` rule option is used to check the stream size of a given TCP session.

Rule writers can check whether the `stream_size` is less than, greater than, equal to, not equal to, less than or equal to, or greater than or equal to a specified integer value, or they can check that the window number is between a range of numbers, using the `<>` range operator for an exclusive range check or the `<=>` for an inclusive one.

By default, the specified value gets checked against both the client and server's TCP sequence numbers, marking it as a "match" if either check passes. However, rule writers can also specify that the `stream_size` apply only to TCP sequence numbers from the server, client, or both server and client. This is done by placing a comma at the end of the argument followed by one of four possible options: `either`, `to_server`, `to_client`, and `both`.

Format:

Single value comparison:

```
stream_size:[<|>|=|!|<=|>=]bytes[,{either|to_server|to_client|both}];
```

Range comparison:

```
stream_size:min_bytes{<>|<=>}max_bytes[,{either|to_server|to_client|both}];
```

Examples:

```
# Look for sessions containg traffic to the server where the
# stream size is equal to 125 bytes
stream_size:=125,to_server;
```

```
# Look for sessions where the stream size values from both
# the client and server are between 0 and 100 bytes (exclusive)
stream_size:0<>100,both;
```

Post-Detection Rule Options

Post-detection rule options are specific triggers that happen after a rule has "fired". All post-detection options are discussed in the next few sections, but a quick reference of them all can be found below.

Quick Reference

keyword	description
detection_filter	`detection_filter` sets the rate in which the rule must hit before an event gets generated
replace	`replace` is used to match and then overwrite payload data
tag	`tag` is used to log additional packets after a rule event

detection_filter

The `detection_filter` option is used to require multiple rule hits before generating an "event". Rule writers use this option to define a rate (count per seconds) that must be exceeded by a source or destination host before a rule can generate an event.

This option is used by declaring three things: (1) whether to track from a source or destination host, (2) the maximum number of rule matches in `s` seconds allowed before the detection filter limit is exceeded, and (3) the period over which the count is accrued.

Snort evaluates a `detection_filter` option last, after evaluating all other rule options (regardless of the position of the filter within the rule source). Only one `detection_filter` option is permitted per rule.

Format:

```
detection_filter:track {by_src|by_dst}, count c, seconds s;
```

Examples:

```
# this rule looks for 30 SSH login attempts occurring
# in 60 seconds from a single source IP
flow:established,to_server;
content:"SSH",nocase,offset 0,depth 4;
detection_filter:track by_src, count 30, seconds 60;
```

replace

The `replace` rule option is used to overwrite prior matching content with the string provided to the option. This option should be used with the `rewrite` rule action, and it works for raw packets only.

Format:

```
replace:"string";
```

Examples:

```
content:"ABCD";
replace:"EFGH";
```

tag

The `tag` rule option is used to tell Snort to continue to log additional packets or bytes following a rule "event". Rule writers can specify whether to tag packets or bytes seen from both the source host and destination host, just the source host, or just the destination host. Furthermore, they can also control whether to log a certain number of packets or bytes, or to log all packets occurring over a specified number of seconds.

This rule option requires two arguments: (1) the tag type that defines from whom to log the packets and (2) the number of packets or bytes to log or the number of seconds to log for.

Format:

```
tag:{session|host_src|host_dst}, {packets packets|seconds seconds|bytes bytes};
```

Examples:

```
# tag the next 10 packets from the entire session
tag:session, packets 10;
```

```
# tag the next 4000 bytes from the source IP address of the
# packet that generated the initial event
tag:host_src, bytes 4000;
```

Shared Object Rules

Shared object rules are written in C++ and loaded by Snort at runtime.

The following is a simple Snort3 SO rule example:

```
#include "main/snort_types.h"
#include "framework/so_rule.h"

using namespace snort;

static const char* rule_1337 = R"[Snort_SO_Rule](
alert tcp any any -> any any (
    msg:"SERVER-OTHER SO Example";
    soid:1337;
    flow:to_server,established;
    content:"foo",nocase;
    so:eval;
    gid:3;
    sid:1337;
)
)[Snort_SO_Rule]";

static const unsigned rule_1337_len = 0;

static IpsOption::EvalStatus rule_1337_eval(void*, Cursor&, Packet*)
{
    return IpsOption::MATCH;
}

static SoEvalFunc rule_1337_ctor(const char* /*so*/, void** pv)
{
    *pv = nullptr;
    return rule_1337_eval;
}

static const SoApi so_1337 =
{
    {
        PT_SO_RULE,
        sizeof(SoApi),
        SOAPI_VERSION,
        0, // version
        API_RESERVED,
        API_OPTIONS,
        "1337",
        "SERVER-OTHER SO Example",
        nullptr,
        nullptr
    },
    (uint8_t*)rule_1337,
    rule_1337_len,
    nullptr,        // pinit
    nullptr,        // pterm
    nullptr,        // tinit
    nullptr,        // tterm
    rule_1337_ctor, // ctor
    nullptr         // dtor
};

const BaseApi* pso_1337 = &so_1337.base;

SO_PUBLIC const BaseApi* snort_plugins[] =
```

```
{
    pso_1337,
    nullptr
};
```