Snort 3 User Manual

The Snort Team

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
3.1.30.0	2022-05-19 00:39:56 EDT		TST

Contents

1 Overview

Snort 3.0 is an updated version of the Snort Intrusion Prevention System (IPS) which features a new design that provides a superset of Snort 2.X functionality with better throughput, detection, scalability, and usability. Some of the key features of Snort 3.0 are:

- Support multiple packet processing threads
- Use a shared configuration and attribute table
- Autodetect services for portless configuration
- Modular design
- Plugin framework with over 200 plugins
- More scalable memory profile
- LuaJIT configuration, loggers, and rule options
- Hyperscan support
- Rewritten TCP handling
- New rule parser and syntax
- Service rules like alert http
- Rule "sticky" buffers
- Way better SO rules
- New HTTP inspector
- New performance monitor
- New time and space profiling
- New latency monitoring and enforcement

- Inspection Events
- Autogenerate reference documentation

Additional features are on the road map:

- Use a shared network map
- Support hardware offload for fast pattern acceleration
- Provide support for DPDK and ODP
- Support pipelining of packet processing
- Support proxy mode
- Multi-tennant support
- Incremental reload
- New serialization of perf data and events
- Enhanced rule processing
- Windows support
- Anomaly detection
- and more!

The remainder of this section provides a high level survey of the inputs, processing, and outputs available with Snort 3.0.

Snort++ is the project that is creating Snort 3.0. In this manual "Snort" or "Snort 3" refers to the 3.0 version and earlier versions will be referred to as "Snort 2" where the distinction is relevant.

1.1 First Steps

Snort can be configured to perform complex packet processing and deep packet inspection but it is best start simply and work up to more interesting tasks. Snort won't do anything you didn't specifically ask it to do so it is safe to just try things out and see what happens. Let's start by just running Snort with no arguments:

```
$ snort
```

That will output usage information including some basic help commands. You should run all of these commands now to see what is available:

```
$ snort -V
$ snort -?
$ snort --help
```

Note that Snort has extensive command line help available so if anything below isn't clear, there is probably a way to get the exact information you need from the command line.

Now let's examine the packets in a capture file (pcap):

```
$ snort -r a.pcap
```

Snort will decode and count the packets in the file and output some statistics. Note that the output excludes non-zero numbers so it is easy to see what is there.

You may have noticed that there are command line options to limit the number of packets examined or set a filter to select particular packets. Now is a good time to experiment with those options.

If you want to see details on each packet, you can dump the packets to console like this:

```
$ snort -r a.pcap -L dump
```

Add the -d option to see the TCP and UDP payload. Now let's switch to live traffic. Replace eth0 in the below command with an available network interface:

```
$ snort -i eth0 -L dump
```

Unless the interface is taken down, Snort will just keep running, so enter Control-C to terminate or use the -n option to limit the number of packets.

Generally it is better to capture the packets for later analysis like this:

```
$ snort -i eth0 -L pcap -n 10
```

Snort will write 10 packets to log.pcap.# where # is a timestamp value. You can read these back with -r and dump to console or pcap with -L. You get the idea.

Note that you can do similar things with other tools like tcpdump or Wireshark however these commands are very useful when you want to check your Snort setup.

The examples above use the default pcap DAQ. Snort supports non-pcap interfaces as well via the DAQ (data acquisition) library. Other DAQs provide additional functionality such as inline operation and/or higher performance. There are even DAQs that support raw file processing (ie without packets), socket processing, and plain text packets. To load external DAQ libraries and see available DAQs or select a particular DAQ use one of these commands:

```
$ snort --daq-dir <path> --daq-list
$ snort --daq-dir <path> --daq <type>
```

Be sure to put the --daq-dir option ahead of the --daq-list option or the external DAQs won't appear in the list.

To leverage intrusion detection features of Snort you will need to provide some configuration details. The next section breaks down what must be done.

1.2 Configuration

Effective configuration of Snort is done via the environment, command line, a Lua configuration file, and a set of rules.

Note that backwards compatibility with Snort 2 was sacrificed to obtain new and improved functionality. While Snort 3 leverages some of the Snort 2 code base, a lot has changed. The configuration of Snort 3 is done with Lua, so your old conf won't work as is. Rules are still text based but with syntax tweaks, so your 2.X rules must be fixed up. However, snort2lua will help you convert your conf and rules to the new format.

1.2.1 Command Line

A simple command line might look like this:

```
snort -c snort.lua -R cool.rules -r some.pcap -A cmg
```

To understand what that does, you can start by just running snort with no arguments by running snort --help. Help for all configuration and rule options is available via a suitable command line. In this case:

-c snort.lua is the main configuration file. This is a Lua script that is executed when loaded.

-R cool.rules contains some detection rules. You can write your own or obtain them from Talos (native 3.0 rules are not yet available from Talos so you must convert them with snort2lua). You can also put your rules directly in your configuration file.

-r some.pcap tells Snort to read network traffic from the given packet capture file. You could instead use -i eth0 to read from a live interface. There many other options available too depending on the DAQ you use.

-A cmg says to output intrusion events in "cmg" format, which has basic header details followed by the payload in hex and text.

Command line options have precedence over Lua configuration files. This can be used to make a custom run keeping all configuration files unchanged:

```
--daq-batch-size=32
```

will override daq.batch_size value.

Notably, you can add to and/or override anything in your configuration file by using the --lua command line option. For example:

```
--lua 'ips = { enable_builtin_rules = true }'
```

will load the built-in decoder and inspector rules. In this case, ips is overwritten with the config you see above. If you just want to change the config given in your configuration file you would do it like this:

```
--lua 'ips.enable_builtin_rules = true'
```

1.2.2 Configuration File

The configuration file gives you complete control over how Snort processes packets. Start with the default snort.lua included in the distribution because that contains some key ingredients. Note that most of the configurations look like:

```
stream = { }
```

This means enable the stream module using internal defaults. To see what those are, you could run:

```
snort --help-config stream
```

Snort is organized into a collection of builtin and plugin modules. If a module has parameters, it is configured by a Lua table of the same name. For example, we can see what the active module has to offer with this command:

```
$ snort --help-module active

What: configure responses

Type: basic

Configuration:

int active.attempts = 0: number of TCP packets sent per response (with
varying sequence numbers) { 0:20 }

string active.device: use 'ip' for network layer responses or 'eth0' etc
for link layer

string active.dst_mac: use format '01:23:45:67:89:ab'

int active.max_responses = 0: maximum number of responses { 0: }

int active.min_interval = 255: minimum number of seconds between
responses { 1: }
```

This says active is a basic module that has several parameters. For each, you will see:

```
type module.name = default: help { range }
```

For example, the active module has a max_responses parameter that takes non-negative integer values and defaults to zero. We can change that in Lua as follows:

```
active = { max_responses = 1 }
```

or:

```
active = { }
active.max_responses = 1
```

If we also wanted to limit retries to at least 5 seconds, we could do:

```
active = { max_responses = 1, min_interval = 5 }
```

1.2.3 Lua Variables

The following Global Lua Variables are available when Snort is run with a lua config using -c option.

- SNORT_VERSION: points to a string containing snort version and build as follows:

  ```
  SNORT_VERSION = "3.0.2-x"
  ```

- SNORT_MAJOR_VERSION: Snort version's major number.

  ```
  SNORT_MAJOR_VERSION = 3
  ```

- SNORT_MINOR_VERSION: Snort version's minor number.

  ```
  SNORT_MINOR_VERSION = 0
  ```

- SNORT_PATCH_VERSION: Snort version's patch number.

  ```
  SNORT_PATCH_VERSION = 2
  ```

1.2.4 Whitelist

When Snort is run with the --warn-conf-strict option, warnings will be generated for all Lua tables present in the configuration files that do not map to Snort module names. Like with other warnings, these will upgraded to errors when Snort is run in pedantic mode.

To dynamically add exceptions that should bypass this strict validation, two Lua functions are made available to be called during the evaluation of Snort configuration files: snort_whitelist_append() and snort_whitelist_add_prefix(). Each function takes a whitespace-delimited list, the former a list of exact table names and the latter a list of table name prefixes to allow.

Examples: snort_whitelist_append("table1 table2") snort_whitelist_add_prefix("local_ foobar_")

The accumulated contents of the whitelist (both exact and prefix) will be dumped when Snort is run in verbose mode (-v).

1.2.5 Rules

Rules determine what Snort is looking for. They can be put directly in your Lua configuration file with the ips module, on the command line with --lua, or in external files. Generally you will have many rules obtained from various sources such as Talos and loading external files is the way to go so we will summarize that here. Add this to your Lua configuration:

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

to load the external rules file named rules.txt. You can only specify one file this way but rules files can include other rules files with the include statement. In addition you can load rules like:

```
$ sort -c snort.lua -R rules.txt
```

You can use both approaches together.

1.2.6 Includes

Your configuration file may include other files, either directly via Lua or via various parameters. Snort will find relative includes in the following order:

1. If you specify --include-path, this directory will be tried first.
2. Snort will try the directory containing the including file.
3. Snort will try the directory containing the -c configuration file.

Some things to keep in mind:

- If you use the Lua dofile function, then you must specify absolute paths or paths relative to your working directory since Lua will execute the include before Snort sees the file contents.
- For best results, use include in place of dofile. This function is provided to follow Snort's include logic.
- As of now, appid and reputation paths must be absolute or relative to the working directory. These will be updated in a future release.

1.2.7 Converting Your 2.X Configuration

If you have a working 2.X configuration snort2lua makes it easy to get up and running with Snort 3. This tool will convert your configuration and/or rules files automatically. You will want to clean up the results and double check that it is doing exactly what you need.

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

The above command will generate snort.lua based on your 2.X configuration. For more information and options for more sophisticated use cases, see the Snort2Lua section later in the manual.

1.3 Output

Snort can produce quite a lot of data. In the following we will summarize the key aspects of the core output types. Additional data such as from appid is covered later.

1.3.1 Basic Statistics

At shutdown, Snort will output various counts depending on configuration and the traffic processed. Generally, you may see:

- Packet Statistics - this includes data from the DAQ and decoders such as the number of packets received and number of UDP packets.
- Module Statistics - each module tracks activity via a set of peg counts that indicate how many times something was observed or performed. This might include the number of HTTP GET requests processed and the number of TCP reset packets trimmed.
- File Statistics - look here for a breakdown of file type, bytes, signatures.
- Summary Statistics - this includes total runtime for packet processing and the packets per second. Profiling data will appear here as well if configured.

Note that only the non-zero counts are output. Run this to see the available counts:

```
$ snort --help-counts
```

1.3.2 Alerts

If you configured rules, you will need to configure alerts to see the details of detection events. Use the -A option like this:

```
$ snort -c snort.lua -r a.pcap -A cmg
```

There are many types of alert outputs possible. Here is a brief list:

- -A cmg is the same as -A fast -d -e and will show information about the alert along with packet headers and payload.
- -A u2 is the same as -A unified2 and will log events and triggering packets in a binary file that you can feed to other tools for post processing. Note that Snort 3 does not provide the raw packets for alerts on PDUs; you will get the actual buffer that alerted.
- -A csv will output various fields in comma separated value format. This is entirely customizable and very useful for pcap analysis.

To see the available alert types, you can run this command:

```
$ snort --list-plugins | grep logger
```

1.3.3 Files and Paths

Note that output is specific to each packet thread. If you run 4 packet threads with u2 output, you will get 4 different u2 files. The basic structure is:

```
<logdir>/[<run_prefix>][<id#>][<X>]<name>
```

where:

- logdir is set with -l and defaults to ./
- run_prefix is set with --run-prefix else not used
- id# is the packet thread number that writes the file; with one packet thread, id# (zero) is omitted without --id-zero
- X is / if you use --id-subdir, else _ if id# is used
- name is based on module name that writes the file

Additional considerations:

- There is no way to explicitly configure a full path to avoid issues with multiple packet threads.
- All text mode outputs default to stdout

1.3.4 Performance Statistics

Still more data is available beyond the above.

- By configuring the perf_monitor module you can capture a configurable set of peg counts during runtime. This is useful to feed to an external program so you can see what is happening without stopping Snort.
- The profiler module allows you to track time and space used by module and rules. Use this data to tune your system for best performance. The output will show up under Summary Statistics at shutdown.

2 Concepts

This section provides background on essential aspects of Snort's operation.

2.1 Terminology

- **basic module**: a module integrated into Snort that does not come from a plugin.
- **binder**: inspector that maps configuration to traffic
- **builtin rules**: codec and inspector rules for anomalies detected internally.
- **codec**: short for coder / decoder. These plugins are used for basic protocol decoding, anomaly detection, and construction of active responses.
- **data module**: an adjunct configuration plugin for use with certain inspectors.
- **dynamic rules**: plugin rules loaded at runtime. See SO rules.
- **fast pattern**: the content in an IPS rule that must be found by the search engine in order for a rule to be evaluated.
- **fast pattern matcher**: see search engine.
- **hex**: a type of protocol magic that the wizard uses to identify binary protocols.
- **inspector**: plugin that processes packets (similar to the Snort 2 preprocessor)
- **IPS**: intrusion prevention system, like Snort.
- **IPS action**: plugin that allows you to perform custom actions when events are generated. Unlike loggers, these are invoked before thresholding and can be used to control external agents or send active responses.
- **IPS option**: this plugin is the building blocks of IPS rules.
- **logger**: a plugin that performs output of events and packets. Events are thresholded before reaching loggers.
- **module**: the user facing portion of a Snort component. Modules chiefly provide configuration parameters, but may also provide commands, builtin rules, profiling statistics, peg counts, etc. Note that not all modules are plugins and not all plugins have modules.
- **peg count**: the number of times a given event or condition occurs.
- **plugin**: one of several types of software components that can be loaded from a dynamic library when Snort starts up. Some plugins are coupled with the main engine in such a way that they must be built statically, but a newer version can be loaded dynamically.
- **search engine**: a plugin that performs multipattern searching of packets and payload to find rules that should be evaluated. There are currently no specific modules, although there are several search engine plugins. Related configuration is done with the basic detection module. Aka fast pattern matcher.
- **SO rule**: a IPS rule plugin that performs custom detection that can't be done by a text rule. These rules typically do not have associated modules. SO comes from shared object, meaning dynamic library.
- **spell**: a type of protocol magic that the wizard uses to identify ASCII protocols.
- **text rule**: a rule loaded from the configuration that has a header and body. The header specifies action, protocol, source and destination IP addresses and ports, and direction. The body specifies detection and non-detection options.
- **wizard**: inspector that applies protocol magic to determine which inspectors should be bound to traffic absent a port specific binding. See hex and spell.

2.2 Modules

Modules are the building blocks of Snort. They encapsulate the types of data that many components need including parameters, peg counts, profiling, builtin rules, and commands. This allows Snort to handle them generically and consistently. You can learn quite a lot about any given module from the command line. For example, to see what stream_tcp is all about, do this:

```
$ snort --help-module stream_tcp
```

Modules are configured using Lua tables with the same name. So the stream_tcp module is configured with defaults like this:

```
stream_tcp = { }
```

The earlier help output showed that the default session tracking timeout is 30 seconds. To change that to 60 seconds, you can configure it this way:

```
stream_tcp = { session_timeout = 60 }
```

Or this way:

```
stream_tcp = { }
stream_tcp.session_timeout = 60
```

More on parameters is given in the next section.

Other things to note about modules:

- Shutdown output will show the non-zero peg counts for all modules. For example, if stream_tcp did anything, you would see the number of sessions processed among other things.
- Providing the builtin rules allows the documentation to include them automatically and also allows for autogenerating the rules at startup.
- Only a few module provide commands at this point, most notably the snort module.

2.3 Parameters

Parameters are given with this format:

```
type name = default: help { range }
```

The following types are used:

- **addr**: any valid IP4 or IP6 address or CIDR
- **addr_list**: a space separated list of addr values
- **bit_list**: a list of consecutive integer values from 1 to the range maximum
- **bool**: true or false
- **dynamic**: a select type determined by loaded plugins
- **enum**: a string selected from the given range
- **implied**: an IPS rule option that takes no value but means true
- **int**: a whole number in the given range
- **interval**: a set of ints (see below)
- **ip4**: an IP4 address or CIDR

- **mac**: an ethernet address with the form 01:02:03:04:05:06
- **multi**: one or more space separated strings from the given range
- **port**: an int in the range 0:65535 indicating a TCP or UDP port number
- **real**: a real number in the given range
- **select**: a string selected from the given range
- **string**: any string with no more than the given length, if any

The parameter name may be adorned in various ways to indicate additional information about the type and use of the parameter:

- For Lua configuration (not IPS rules), if the name ends with [] it is a list item and can be repeated.
- For IPS rules only, names starting with ~ indicate positional parameters. The names of such parameters do not appear in the rule.
- IPS rules may also have a wild card parameter, which is indicated by a *. Used for unquoted, comma-separated lists such as service and metadata.
- The snort module has command line options starting with a -. The options passed from command line override the options configured via snort module.
- $ denotes variable names.

Some additional details to note:

- Table and variable names are case sensitive; use lower case only.
- String values are case sensitive too; use lower case only.
- Numeric ranges may be of the form low:high where low and high are bounds included in the range. If either is omitted, there is no hard bound. E.g. 0: means any x where x >= 0.
- Strings may have a numeric range indicating a length limit; otherwise there is no hard limit.
- bit_list is typically used to store a set of byte, port, or VLAN ID values.
- interval takes the form [operator]i, j<>k, or j<⇒k where i,j,k are integers and operator is one of =, !, != (same as !), <, ⇐, >, >=. j<>k means j < int < k and j<⇒k means j ⇐ int ⇐ k.
- Ranges may use maxXX like { 1:max32 } since max32 is easier to read than 4294967295. To get the values of maxXX, use snort --help-limits.

Parameter limits:

- max31 = 2147483647
- max32 = 4294967295
- max53 = 9007199254740992
- maxSZ = 9007199254740992

2.4 Plugins

Snort uses a variety of plugins to accomplish much of its processing objectives, including:

- Codec - to decode and encode packets
- Inspector - like Snort 2 preprocessors, for normalization, etc.
- IpsOption - for detection in Snort rules
- IpsAction - for custom actions
- Logger - for handling events
- Mpse - for fast pattern matching
- So - for dynamic rules

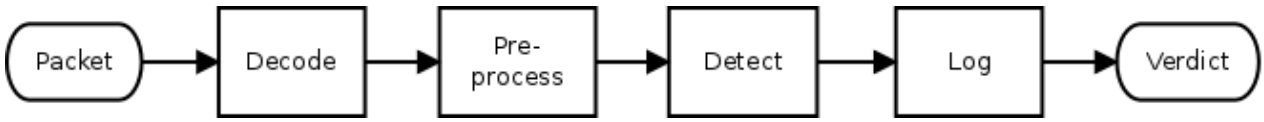
The power of plugins is that they have a very focused purpose and can be created with relative ease. For example, you can extend the rule language by writing your own IpsOption and it will plug in and function just like existing options. The extra directory has examples of each type of plugin.

Most plugins can be built statically or dynamically. By default they are all static. There is no difference in functionality between static or dynamic plugins but the dynamic build generates a slightly lighter weight binary. Either way you can add dynamic plugins with --plugin-path and newer versions will replace older versions, even when built statically.

A single dynamic library may contain more than one plugin. For example, an inspector will typically be packaged together with any associated rule options.

2.5 Operation

Snort is a signature-based IPS, which means that as it receives network packets it reassembles and normalizes the content so that a set of rules can be evaluated to detect the presence of any significant conditions that merit further action. A rough processing flow is as follows:



The steps are:

1. Decode each packet to determine the basic network characteristics such as source and destination addresses and ports. A typical packet might have ethernet containing IP containing TCP containing HTTP (ie eth:ip:tcp:http). The various encapsulating protocols are examined for sanity and anomalies as the packet is decoded. This is essentially a stateless effort.
2. Preprocess each decoded packet using accumulated state to determine the purpose and content of the innermost message. This step may involve reordering and reassembling IP fragments and TCP segments to produce the original application protocol data unit (PDU). Such PDUs are analyzed and normalized as needed to support further processing.
3. Detection is a two step process. For efficiency, most rules contain a specific content pattern that can be searched for such that if no match is found no further processing is necessary. Upon start up, the rules are compiled into pattern groups such that a single, parallel search can be done for all patterns in the group. If any match is found, the full rule is examined according to the specifics of the signature.
4. The logging step is where Snort saves any pertinent information resulting from the earlier steps. More generally, this is where other actions can be taken as well such as blocking the packet.

2.5.1 Snort 2 Processing

The preprocess step in Snort 2 is highly configurable. Arbitrary preprocessors can be loaded dynamically at startup, configured in snort.conf, and then executed at runtime. Basically, the preprocessors are put into a list which is iterated for each packet. Recent versions have tweaked the list handling some, but the same basic architecture has allowed Snort 2 to grow from a sniffer, with no preprocessing, to a full-fledged IPS, with lots of preprocessing.

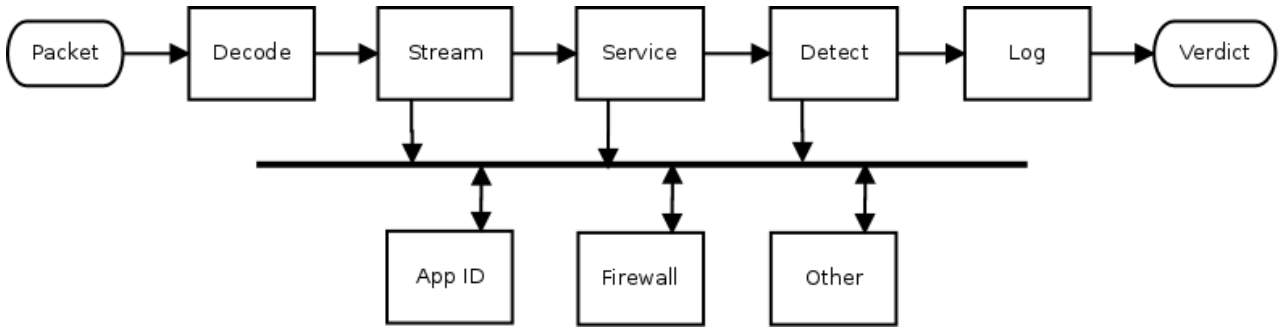
While this "list of plugins" approach has considerable flexibility, it hampers future development when the flow of data from one preprocessor to the next depends on traffic conditions, a common situation with advanced features like application identification. In this case, a preprocessor like HTTP may be extracting and normalizing data that ultimately is not used, or appID may be repeatedly checking for data that is just not available.

Callbacks help break out of the preprocess straitjacket. This is where one preprocessor supplies another with a function to call when certain data is available. Snort has started to take this approach to pass some HTTP and SIP preprocessor data to appID. However, it remains a peripheral feature and still requires the production of data that may not be consumed.

2.5.2 Snort 3 Processing

One of the goals of Snort 3 is to provide a more flexible framework for packet processing by implementing an event-driven approach. Another is to produce data only when needed to minimize expensive normalizations. However, the basic packet processing provides very similar functionality.

The basic processing steps Snort 3 takes are similar to Snort 2 as seen in the following diagram. The preprocess step employs specific inspector types instead of a generalized list, but the basic procedure includes stateless packet decoding, TCP stream reassembly, and service specific analysis in both cases. (Snort 3 provides hooks for arbitrary inspectors, but they are not central to basic flow processing and are not shown.)



However, Snort 3 also provides a more flexible mechanism than callback functions. By using inspection events, it is possible for an inspector to supply data that other inspectors can process. This is known as the observer pattern or publish-subscribe pattern.

Note that the data is not actually published. Instead, access to the data is published, and that means that subscribers can access the raw or normalized version(s) as needed. Normalizations are done only on the first access, and subsequent accesses get the previously normalized data. This results in just in time (JIT) processing.

A basic example of this in action is provided by the extra data_log plugin. It is a passive inspector, ie it does nothing until it receives the data it subscribed for (*other* in the above diagram). By adding the following to your snort.lua configuration, you will get a simple URI logger.

```
data_log = { key = 'http_raw_uri' }
```

Inspection events coupled with pluggable inspectors provide a very flexible framework for implementing new features. And JIT buffer stuffers allow Snort to work smarter, not harder. These capabilities will be leveraged more and more as Snort development continues.

2.6 Rules

Rules tell Snort how to detect interesting conditions, such as an attack, and what to do when the condition is detected. Here is an example rule:

```
alert tcp any any -> 192.168.1.1 80 ( msg:"A ha!"; content:"attack"; sid:1; )
```

The structure is:

```
action proto source dir dest ( body )
```

Where:

action - tells Snort what to do when a rule "fires", ie when the signature matches. In this case Snort will log the event. It can also do thing like block the flow when running inline.

proto - tells Snort what protocol applies. This may be ip, icmp, tcp, udp, http, etc.

source - specifies the sending IP address and port, either of which can be the keyword any, which is a wildcard.

dir - must be either unidirectional as above or bidirectional indicated by <>.

dest - similar to source but indicates the receiving end.

body - detection and other information contained in parenthesis.

There are many rule options available to construct as sophisticated a signature as needed. In this case we are simply looking for the "attack" in any TCP packet. A better rule might look like this:

```
alert http
(
    msg:"Gotcha!";
    flow:established, to_server;
    http_uri:"attack";
    sid:2;
)
```

Note that these examples have a sid option, which indicates the signature ID. In general rules are specified by gid:sid:rev notation, where gid is the generator ID and rev is the revision of the rule. By default, text rules are gid 1 and shared-object (SO) rules are gid 3. The various components within Snort that generate events have 1XX gids, for example the decoder is gid 116. You can list the internal gids and sids with these commands:

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

For details on these and other options, see the reference section.

2.7 Pattern Matching

Snort evaluates rules in a two-step process which includes a fast pattern search and full evaluation of the signature. More details on this process follow.

2.7.1 Rule Groups

When Snort starts or reloads configuration, rules are grouped by protocol, port and service. For example, all TCP rules using the HTTP_PORTS variable will go in one group and all service HTTP rules will go in another group. These rule groups are compiled into multipattern search engines (MPSE) which are designed to search for all patterns with just a single pass through a given packet or buffer. You can select the algorithm to use for fast pattern searches with search_engine.search_method which defaults to *ac_bnfa*, which balances speed and memory. For a faster search at the expense of significantly more memory, use *ac_full*. For best performance and reasonable memory, download the hyperscan source from Intel.

2.7.2 Fast Patterns

Fast patterns are content strings that have the fast_pattern option or which have been selected by Snort automatically to be used as a fast pattern. Snort will by default choose the longest pattern in the rule since that is likely to be most unique. That is not always the case so add fast_pattern to the appropriate content option for best performance. The ideal fast pattern is one which, if found, is very likely to result in a rule match. Fast patterns that match frequently for unrelated traffic will cause Snort to work hard with little to show for it.

Certain contents are not eligible to be used as fast patterns. Specifically, if a content is negated, then if it is also relative to another content, case sensitive, or has non-zero offset or depth, then it is not eligible to be used as a fast pattern.

2.7.3 Rule Evaluation

For each fast pattern match, the corresponding rule(s) are evaluated left-to-right. Rule evaluation requires checking each detection option in a rule and is a fairly costly process which is why fast patterns are so important. Rule evaluation aborts on the first non-matching option.

When rule evaluation takes place, the fast pattern match will automatically be skipped if possible. Note that this differs from Snort 2 which provided the fast_pattern:only option to designate such cases. This is one less thing for the rule writer to worry about.

3 Tutorial

The section will walk you through building and running Snort. It is not exhaustive but, once you master this material, you should be able to figure out more advanced usage.

3.1 Dependencies

Required:

- a compiler that supports the C++14 feature set
- cmake to build from source
- daq from https://github.com/snort3/libdaq for packet IO
- dnet from https://github.com/dugsong/libdnet.git for network utility functions
- hwloc from https://www.open-mpi.org/projects/hwloc/ for CPU affinity management
- LuaJIT from http://luajit.org for configuration and scripting
- OpenSSL from https://www.openssl.org/source/ for SHA and MD5 file signatures, the protected_content rule option, and SSL service detection
- pcap from http://www.tcpdump.org for tcpdump style logging
- pcre from http://www.pcre.org for regular expression pattern matching
- pkgconfig from https://www.freedesktop.org/wiki/Software/pkg-config/ to locate build dependencies
- zlib from http://www.zlib.net for decompression (>= 1.2.8 recommended)

Optional:

- asciidoc from http://www.methods.co.nz/asciidoc/ to build the HTML manual
- cpputest from http://cpputest.github.io to run additional unit tests with make check

- dblatex from http://dblatex.sourceforge.net to build the pdf manual (in addition to asciidoc)
- hyperscan >= 4.4.0 from https://github.com/01org/hyperscan to build new the regex and sd_pattern rule options and hyperscan search engine. Hyperscan is large so it recommended to follow their instructions for building it as a shared library.
- iconv from https://ftp.gnu.org/pub/gnu/libiconv/ for converting UTF16-LE filenames to UTF8 (usually included in glibc)
- libunwind from https://www.nongnu.org/libunwind/ to attempt to dump a somewhat readable backtrace when a fatal signal is received
- lzma >= 5.1.2 from http://tukaani.org/xz/ for decompression of SWF and PDF files
- safec >= 3.5 from https://github.com/rurban/safeclib/ for runtime bounds checks on certain legacy C-library calls
- source-highlight from http://www.gnu.org/software/src-highlite/ to generate the dev guide
- w3m from http://sourceforge.net/projects/w3m/ to build the plain text manual
- uuid from uuid-dev package for unique identifiers

3.2 Building

- Optionally built features are listed in the reference section.
- Create an install path:

```
export my_path=/path/to/snorty
mkdir -p $my_path
```

- If LibDAQ was installed to a custom, non-system path:

```
export PKG_CONFIG_PATH=/libdaq/install/path/lib/pkgconfig:$PKG_CONFIG_PATH
```

- Now do one of the following:

  a. To build with cmake and make, run configure_cmake.sh. It will automatically create and populate a new subdirectory named *build*.

  ```
  ./configure_cmake.sh --prefix=$my_path
  cd build
  make -j
  make install
  ln -s $my_path/conf $my_path/etc
  ```

  b. You can also specify a cmake project generator:

  ```
  ./configure_cmake.sh --generator=Xcode --prefix=$my_path
  ```

  c. Or use ccmake directly to configure and generate from an arbitrary build directory like one of these:

  ```
  ccmake -G Xcode /path/to/Snort++/tree
  open snort.xcodeproj

  ccmake -G "Eclipse CDT4 - Unix Makefiles" /path/to/Snort++/tree
  run eclipse and do File > Import > Existing Eclipse Project
  ```

- To build with g++ on OS X where clang is installed, do this first:

```
export CXX=g++
```

3.3 Running

Examples:

- Get some help:

```
$my_path/bin/snort --help
$my_path/bin/snort --help-module suppress
$my_path/bin/snort --help-config | grep thread
```

- Examine and dump a pcap:

```
$my_path/bin/snort -r <pcap>
$my_path/bin/snort -L dump -d -e -q -r <pcap>
```

- Verify config, with or w/o rules:

```
$my_path/bin/snort -c $my_path/etc/snort/snort.lua
$my_path/bin/snort -c $my_path/etc/snort/snort.lua -R $my_path/etc/snort/sample. ↩
    rules
```

- Run IDS mode. To keep it brief, look at the first n packets in each file:

```
$my_path/bin/snort -c $my_path/etc/snort/snort.lua -R $my_path/etc/snort/sample. ↩
    rules \
     -r <pcap> -A alert_test -n 100000
```

- Let's suppress 1:2123. We could edit the conf or just do this:

```
$my_path/bin/snort -c $my_path/etc/snort/snort.lua -R $my_path/etc/snort/sample. ↩
    rules \
     -r <pcap> -A alert_test -n 100000 --lua "suppress = { { gid = 1, sid = 2123 } ↩
         }"
```

- Go whole hog on a directory with multiple packet threads:

```
$my_path/bin/snort -c $my_path/etc/snort/snort.lua -R $my_path/etc/snort/sample. ↩
    rules \
     --pcap-filter \*.pcap --pcap-dir <dir> -A alert_fast -n 1000 --max-packet- ↩
        threads 8
```

For more examples, see the usage section.

3.4 Tips

One of the goals of Snort 3 is to make it easier to configure your sensor. Here is a summary of tips and tricks you may find useful.

General Use

- Snort tries hard not to error out too quickly. It will report multiple semantic errors.
- Snort always assumes the simplest mode of operation. Eg, you can omit the -T option to validate the conf if you don't provide a packet source.
- Warnings are not emitted unless --warn-* is specified. --warn-all enables all warnings, and --pedantic makes such warnings fatal.

- You can process multiple sources at one time by using the -z or --max-threads option.
- To make it easy to find the important data, zero counts are not output at shutdown.
- Load plugins from the command line with --plugin-path /path/to/install/lib.
- You can process multiple sources at one time by using the -z or --max-threads option.
- Unit tests are configured with --enable-unit-tests. They can then be run with snort --catch-test [tags]|all.
- Benchmark tests are configured with --enable-benchmark-tests. They can then be run with snort --catch-test [tags]|all or built as a separate executable. It is also preferred to configure a non-debug build with optimizations enabled.

Lua Configuration

- Some parameters could be configured via a command line option or snort module. In this case a command line option has the highest precedence, in turn, snort module configuration has precedence over other modules.
- Configure the wizard and default bindings will be created based on configured inspectors. No need to explicitly bind ports in this case.
- You can override or add to your Lua conf with the --lua command line option.
- The Lua conf is a live script that is executed when loaded. You can add functions, grab environment variables, compute values, etc.
- You can also rename symbols that you want to disable. For example, changing normalizer to Xnormalizer (an unknown symbol) will disable the normalizer. This can be easier than commenting in some cases.
- By default, symbols unknown to Snort are silently ignored. You can generate warnings for them with --warn-unknown. To ignore such symbols, export them in the environment variable SNORT_IGNORE.

Writing and Loading Rules

Snort rules allow arbitrary whitespace. Multi-line rules make it easier to structure your rule for clarity. There are multiple ways to add comments to your rules:

- The # character starts a comment to end of line. In addition, all lines between #begin and #end are comments.
- The rem option allows you to write a comment that is conveyed with the rule.
- C style multi-line comments are allowed, which means you can comment out portions of a rule while testing it out by putting the options between /* and */.

There are multiple ways to load rules too:

- Set ips.rules or ips.include.
- include statements can be used in rules files.
- Use -R to load a rules file.
- Use --stdin-rules with command line redirection.
- Use --lua to specify one or more rules as a command line argument.

Ips states are similar to ips rules, except that they are parsed after the rules. That way rules can be overwritten in custom policies.

States without the *enable* option are loaded as stub rules with default gid:0, sid:0. A user should specify *gid*, *sid*, *enable* options to avoid dummy rules.

Output Files

To make it simple to configure outputs when you run with multiple packet threads, output files are not explicitly configured. Instead, you can use the options below to format the paths:

```
<logdir>/[<run_prefix>][<id#>][<X>]<name>
```

- logdir is set with -l and defaults to ./
- run_prefix is set with --run-prefix else not used
- id# is the packet thread number that writes the file; with one packet thread, id# (zero) is omitted without --id-zero
- X is / if you use --id-subdir, else _ if id# is used
- name is based on module name that writes the file
- all text mode outputs default to stdout

3.5 Common Errors

PANIC: unprotected error in call to Lua API (cannot open snort_defaults.lua: No such file or directory)

- export SNORT_LUA_PATH to point to any dofiles

ERROR can't find xyz

- if xyz is the name of a module, make sure you are not assigning a scalar where a table is required (e.g. xyz = 2 should be xyz = { }).

ERROR can't find x.y

- module x does not have a parameter named y. check --help-module x for available parameters.

ERROR invalid x.y = z

- the value z is out of range for x.y. check --help-config x.y for the range allowed.

ERROR: x = { y = z } is in conf but is not being applied

- make sure that x = { } isn't set later because it will override the earlier setting. same for x.y.

FATAL: can't load lua/errors.lua: lua/errors.lua:68: = expected near ';'

- this is a syntax error reported by Lua to Snort on line 68 of errors.lua.

ERROR: rules(2) unknown rule keyword: find.

- this was due to not including the --script-path.

WARNING: unknown symbol x

- if you any variables, you can squelch such warnings by setting them in an environment variable SNORT_IGNORE. to ignore x, y, and z:

```
export SNORT_IGNORE="x y z"
```

3.6 Gotchas

- A nil key in a table will not be caught. Neither will a nil value in a table. Neither of the following will cause errors, nor will they actually set http_inspect.request_depth:

```
http_inspect = { request_depth }
http_inspect = { request_depth = undefined_symbol }
```

- It is not an error to set a value multiple times. The actual value applied may not be the last in the table either. It is best to avoid such cases.

```
http_inspect =
{
    request_depth = 1234,
    request_depth = 4321
}
```

- Snort can't tell you the exact filename or line number of a semantic error but it will tell you the fully qualified name.

3.7 Known Issues

- The dump DAQ will not work with multiple threads unless you use --daq-var output=none. This will be fixed at some point to use the Snort log directory, etc.
- If you build with hyperscan on OS X and see:

```
dyld: Library not loaded: @rpath/libhs.4.0.dylib

when you try to run src/snort, export DYLD_LIBRARY_PATH with the path to
libhs.  You can also do:

install_name_tool -change @rpath/libhs.4.0.dylib \
    /path-to/libhs.4.0.dylib src/snort
```

- Snort built with tcmalloc support (--enable-tcmalloc) on Ubuntu 17.04/18.04 crashes immediately.

```
Workaround:
Uninstall gperftools 2.5 provided by the distribution and install gperftools
2.7 before building Snort.
```

3.7.1 Reload Limitations

The following parameters can't be changed during reload, and require a restart:

- active.attempts
- active.device
- alerts.detection_filter_memcap
- alerts.event_filter_memcap
- alerts.rate_filter_memcap
- attribute_table.max_hosts
- attribute_table.max_services_per_host

- daq.snaplen
- detection.asn1
- file_id.max_files_cached
- process.chroot
- process.daemon
- process.set_gid
- process.set_uid
- snort.--bpf
- snort.-l
- trace.output

In addition, the following scenarios require a restart:

- Enabling file capture for the first time
- Changing file_id.capture_memcap if file capture was previously or currently enabled
- Changing file_id.capture_block_size if file capture was previously or currently enabled
- Adding/removing stream_* inspectors if stream was already configured

In all of these cases reload will fail with the following message: "reload failed - restart required". The original config will remain in use.

4 Usage

For the following examples "$my_path" is assumed to be the path to the Snort install directory. Additionally, it is assumed that "$my_path/bin" is in your PATH.

4.1 Help

Print the help summary:

```
snort --help
```

Get help on a specific module ("stream", for example):

```
snort --help-module stream
```

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

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

Grep for help on threads:

```
snort --help-config | grep thread
```

Output help on "rule" options in AsciiDoc format:

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

Note
Snort stops reading command-line options after the "--help-**" and "--list-**" options, so any other options should be placed before them.

4.2 Sniffing and Logging

Read a pcap:

```
snort -r /path/to/my.pcap
```

Dump the packets to stdout:

```
snort -r /path/to/my.pcap -L dump
```

Dump packets with application data and layer 2 headers

```
snort -r /path/to/my.pcap -L dump -d -e
```

Note
Command line options must be specified separately. "snort -de" won't work. You can still concatenate options and their arguments, however, so "snort -Ldump" will work.

Dump packets from all pcaps in a directory:

```
snort --pcap-dir /path/to/pcap/dir --pcap-filter '*.pcap' -L dump -d -e
```

Log packets to a directory:

```
snort --pcap-dir /path/to/pcap/dir --pcap-filter '*.pcap' -L dump -l /path/to/log/ ↩
   dir
```

4.3 Configuration

Validate a configuration file:

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

Validate a configuration file and a separate rules file:

```
snort -c $my_path/etc/snort/snort.lua -R $my_path/etc/snort/sample.rules
```

Read rules from stdin and validate:

```
snort -c $my_path/etc/snort/snort.lua --stdin-rules < $my_path/etc/snort/sample. ↩
   rules
```

Enable warnings for Lua configurations and make warnings fatal:

```
snort -c $my_path/etc/snort/snort.lua --warn-all --pedantic
```

Tell Snort where to look for additional Lua scripts:

```
snort --script-path /path/to/script/dir
```

4.4 IDS mode

Run Snort in IDS mode, reading packets from a pcap:

```
snort -c $my_path/etc/snort/snort.lua -r /path/to/my.pcap
```

Log any generated alerts to the console using the "-A" option:

```
snort -c $my_path/etc/snort/snort.lua -r /path/to/my.pcap -A alert_full
```

Capture separate stdout, stderr, and stdlog files (out has startup and shutdown output, err has warnings and errors, and log has alerts):

```
snort -c $my_path/etc/snort/snort.lua -r /path/to/my.pcap -A csv \
    1>out 2>err 3>log
```

Add or modify a configuration from the command line using the "--lua" option:

```
snort -c $my_path/etc/snort/snort.lua -r /path/to/my.pcap -A cmg \
    --lua 'ips = { enable_builtin_rules = true }'
```

Note
The "--lua" option can be specified multiple times.

Run Snort in IDS mode on an entire directory of pcaps, processing each input source on a separate thread:

```
snort -c $my_path/etc/snort/snort.lua --pcap-dir /path/to/pcap/dir \
    --pcap-filter '*.pcap' --max-packet-threads 8
```

Run Snort on 2 interfaces, eth0 and eth1:

```
snort -c $my_path/etc/snort/snort.lua -i "eth0 eth1" -z 2 -A cmg
```

Run Snort inline with the afpacket DAQ:

```
snort -c $my_path/etc/snort/snort.lua --daq afpacket -i "eth0:eth1" \
    -A cmg
```

4.5 Plugins

Load external plugins and use the "ex" alert:

```
snort -c $my_path/etc/snort/snort.lua \
    --plugin-path $my_path/lib/snort_extra \
    -A alert_ex -r /path/to/my.pcap
```

Test the LuaJIT rule option *find* loaded from stdin:

```
snort -c $my_path/etc/snort/snort.lua \
    --script-path $my_path/lib/snort_extra \
    --stdin-rules -A cmg -r /path/to/my.pcap << END
alert tcp any any -> any 80 (
    sid:3; msg:"found"; content:"GET";
    find:"pat='HTTP/1%.%d'" ; )
END
```

4.6 Output Files

To make it simple to configure outputs when you run with multiple packet threads, output files are not explicitly configured. Instead, you can use the options below to format the paths:

```
<logdir>/[<run_prefix>][<id#>][<X>]<name>
```

Log to unified in the current directory:

```
snort -c $my_path/etc/snort/snort.lua -r /path/to/my.pcap -A unified2
```

Log to unified in the current directory with a different prefix:

```
snort -c $my_path/etc/snort/snort.lua -r /path/to/my.pcap -A unified2 \
    --run-prefix take2
```

Log to unified in /tmp:

```
snort -c $my_path/etc/snort/snort.lua -r /path/to/my.pcap -A unified2 -l /tmp
```

Run 4 packet threads and log with thread number prefix (0-3):

```
snort -c $my_path/etc/snort/snort.lua --pcap-dir /path/to/pcap/dir \
    --pcap-filter '*.pcap' -z 4 -A unified2
```

Run 4 packet threads and log in thread number subdirs (0-3):

```
snort -c $my_path/etc/snort/snort.lua --pcap-dir /path/to/pcap/dir \
    --pcap-filter '*.pcap' -z 4 -A unified2 --id-subdir
```

Note
subdirectories are created automatically if required. Log filename is based on module name that writes the file. All text mode outputs default to stdout. These options can be combined.

4.7 DAQ Alternatives

Process hext packets from stdin:

```
snort -c $my_path/etc/snort/snort.lua \
    --daq-dir $my_path/lib/snort/daqs --daq hext -i tty << END
$packet 10.1.2.3 48620 -> 10.9.8.7 80
"GET / HTTP/1.1\r\n"
"Host: localhost\r\n"
"\r\n"
END
```

Process raw ethernet from hext file:

```
snort -c $my_path/etc/snort/snort.lua \
    --daq-dir $my_path/lib/snort/daqs --daq hext \
    --daq-var dlt=1 -r <hext-file>
```

Process a directory of plain files (ie non-pcap) with 4 threads with 8K buffers:

```
snort -c $my_path/etc/snort/snort.lua \
    --daq-dir $my_path/lib/snort/daqs --daq file \
    --pcap-dir path/to/files -z 4 -s 8192
```

Bridge two TCP connections on port 8000 and inspect the traffic:

```
snort -c $my_path/etc/snort/snort.lua \
    --daq-dir $my_path/lib/snort/daqs --daq socket
```

4.8 Logger Alternatives

Dump TCP stream payload in hext mode:

```
snort -c $my_path/etc/snort/snort.lua -L hext
```

Output timestamp, pkt_num, proto, pkt_gen, dgm_len, dir, src_ap, dst_ap, rule, action for each alert:

```
snort -c $my_path/etc/snort/snort.lua -A csv
```

Output the old test format alerts:

```
snort -c $my_path/etc/snort/snort.lua \
    --lua "alert_csv = { fields = 'pkt_num gid sid rev', separator = '\t' }"
```

4.9 Shell

You must build with --enable-shell to make the command line shell available.

Enable shell mode:

```
snort --shell <args>
```

You will see the shell mode command prompt, which looks like this:

```
o")~
```

(The prompt can be changed with the SNORT_PROMPT environment variable.)

You can pause immediately after loading the configuration and again before exiting with:

```
snort --shell --pause <args>
```

In that case you must issue the resume() command to continue. Enter quit() to terminate Snort or detach() to exit the shell. You can list the available commands with help().

To enable local telnet access on port 12345:

```
snort --shell -j 12345 <args>
```

The command line interface is still under development. Suggestions are welcome.

4.10 Signals

Note
The following examples assume that Snort is currently running and has a process ID of <pid>.

Modify and Reload Configuration:

```
echo 'suppress = { { gid = 1, sid = 2215 } }' >> $my_path/etc/snort/snort.lua
kill -hup <pid>
```

Dump stats to stdout:

```
kill -usr1 <pid>
```

Shutdown normally:

```
kill -term <pid>
```

Exit without flushing packets:

```
kill -quit <pid>
```

List available signals:

```
snort --help-signals
```

Note
The available signals may vary from platform to platform.

5 Features

This section explains how to use key features of Snort.

5.1 Active Response

Snort can take more active role in securing network by sending active responses to shutdown offending sessions. When active responses is enabled, snort will send TCP RST or ICMP unreachable when dropping a session.

5.1.1 Changes from Snort 2.9

- stream5_global:max_active_responses and min_response_seconds are now active.max_responses and active.min_interval.
- Response actions were removed from IPS rule body to the rule action in the header. This includes react, reject, and rewrite (split out of replace which now just does the detection part). These IPS actions are plugins.
- drop and block are synonymous in Snort 2.9 but in Snort 3.0 drop means don't forward the current packet only whereas block means don't forward this or any following packet on the flow.

5.1.2 Configure Active

Active response is enabled by configuring one of following IPS action plugins:

```
react = { }
reject = { }
```

When these active responses are not configured the default configuration is used.

Active responses will be performed for reject, react or rewrite IPS rule actions, and response packets are encoded based on the triggering packet. TTL will be set to the value captured at session pickup.

Configure the number of attempts to land a TCP RST within the session's current window (so that it is accepted by the receiving TCP). This sequence "strafing" is really only useful in passive mode. In inline mode the reset is put straight into the stream in lieu of the triggering packet so strafing is not necessary.

Each attempt (sent in rapid succession) has a different sequence number. Each active response will actually cause this number of TCP resets to be sent. TCP data is multiplied similarly. At most 1 ICMP unreachable is sent, iff attempts > 0.

Device IP will perform network layer injection. It is probably a better choice to specify an interface and avoid kernel routing tables, etc.

dst_mac will change response destination MAC address, if the device is eth0, eth1, eth2 etc. Otherwise, response destination MAC address is derived from packet.

Example:

```
active =
{
    attempts = 2,
    device = "eth0",
    dst_mac = "00:06:76:DD:5F:E3",
}
```

5.1.3 Reject

IPS action reject perform active response to shutdown hostile network session by injecting TCP resets (TCP connections) or ICMP unreachable packets.

Example:

```
reject = { reset = "both", control = "all" }

local_rules =
[[
reject tcp ( msg:"hostile connection"; flow:established, to_server;
content:"HACK!"; sid:1; )
]]

ips =
{
    rules = local_rules,
}
```

5.1.4 React

IPS action react enables sending an HTML page on a session and then resetting it.

The headers used are:

```
"HTTP/1.1 403 Forbidden\r\n" \
"Connection: close\r\n" \
"Content-Type: text/html; charset=utf-8\r\n" \
"Content-Length: 438\r\n" \
"\r\n"
```

The page to be sent can be read from a file:

```
react = { page = "customized_block_page.html", }
```

or else the default is used:

```
"<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\"\r\n" \
"    \"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\">\r\n" \
"<html xmlns=\"http://www.w3.org/1999/xhtml\" xml:lang=\"en\">\r\n" \
"<head>\r\n" \
"<meta http-equiv=\"Content-Type\" content=\"text/html; charset=UTF-8\" />\r\n" \
"<title>Access Denied</title>\r\n" \
"</head>\r\n" \
"<body>\r\n" \
"<h1>Access Denied</h1>\r\n" \
"<p>You are attempting to access a forbidden site.<br />" \
"Consult your system administrator for details.</p>\r\n" \
"</body>\r\n" \
"</html>\r\n"
```

Note that the file contains the message body only. The headers will be added with an updated value for Content-Length. For HTTP/2 traffic Snort will translate the page to HTTP/2 format.

Limitations for HTTP/2:

- Packet will be injected against the last received stream id.
- Injection triggered while server-to-client flow of traffic is in a middle of a frame is not supported. The traffic will be blocked, but the page will not be injected/displayed.

When using react, payload injector must be configured as well. Also Snort should be in ips mode, so the rule is triggered on the client packet, and not delayed until the server sends ACK. To achieve this use the default normalizer. It will set normalizer.tcp.ips = true. Example:

```
react = { page = "my_block_page.html" }
payload_injector = { }
normalizer = { }

local_rules =
[[
react http ( msg:"Unauthorized Access Prohibited!"; flow:established,
to_server; http_method; content:"GET"; sid:1; )
]]

ips =
{
    rules = local_rules,
}
```

React has debug trace functionality. It can be used to get traces in case injection is not successful. To turn it on:

```
trace =
{
    modules = { react = { all = 1 } }
}
```

5.1.5 Rewrite

IPS action "rewrite" enables overwrite packet contents based on "replace" option in the rules. Note that using "rewrite" action without "replace" option will raise corresponding rule alert, but will not overwrite the packet payload.

For example:

```
local_rules =
[[
rewrite tcp 10.1.1.87 any -> 10.1.1.0/24 80
(
    sid:1000002;
    msg:"test replace rule";
    content:"index.php", nocase;
    replace:"indax.php";
)
]]

ips =
{
    rules = local_rules,
}
```

this rule replaces the first occurrence of "index.php" with "indax.php", and "rewrite" action updates that packet.

Content and replacement are aligned to the right side of the matching content and are limited not by the size of the matching content, but by the boundaries of the packet.

```
Example:

rewrite http any any -> any any
(
    msg:"Small replace";
    content:"content";
    replace:"text";
    sid:1000002;
)

this rule replaces "malicious content" to "malicious context".

Example:

rewrite http any any -> any any
(
    msg:"Big replace";
    content:"content";
    replace:"y favorite page!";
    sid:1000002;
)

this rule replaces "malicious content" to "my favorite page!".
```

Be aware that after the match there should be enough room left for the "replace" content in the matched packet. If there is not enough space for the "replace" content the rule will not match.

"replace" works for raw packets only. So, TCP data must either fit under the "pkt_data" buffer requirements or one should enable detection on TCP payload before reassembly: search_engine.detect_raw_tcp=true. For example:

```
Rule that does not require search_engine.detect_raw_tcp=true:
```

```
rewrite udp any any -> any any
(
    msg:"TEST 1";
    sid:1000002;
    content:"attack";
    replace:"abc123";
)

Rule that does require search_engine.detect_raw_tcp=true:

rewrite http any any -> any any
(
    msg:"TEST 2";
    content:"/content.html";
    replace:"/replace.html";
    sid:1000002;
)
```

5.2 AppId

Network administrators need application awareness in order to fine tune their management of the ever-growing number of applications passing traffic over the network. Application awareness allows an administrator to create rules for applications as needed by the business. The rules can be used to take action based on the application, such as block, allow or alert.

5.2.1 Overview

The AppId inspector provides an application level view when managing networks by providing the following features:

- Network control: The inspector works with Snort rules by providing a set of application identifiers (AppIds) to Snort rule writers.
- Application usage awareness: The inspector outputs statistics to show how many times applications are being used on the network.
- Custom applications: Administrators can create their own application detectors to detect new applications. The detectors are written in Lua and interface with Snort using a well-defined C-Lua API.
- Open Detector Package (ODP): A set of pre-defined application detectors are provided by the Snort team and can be downloaded from snort.org.

5.2.2 Dependency Requirements

For proper functioning of the AppId inspector, at a minimum stream flow tracking must be enabled. In addition, to identify TCP-based or UDP-based applications, the appropriate stream inspector must be enabled, e.g. stream_tcp or stream_udp.

In order to identify HTTP-based applications, the HTTP inspector must be enabled. Otherwise, only non-HTTP applications will be identified.

AppId subscribes to the inspection events published by other inspectors, such as the HTTP and SSL inspectors, to gain access to the data needed. It uses that data to help determine the application ID.

AppId subscribes to the events published by SIP and DCE/RPC inspectors to detect applications on expected flows.

5.2.3 Configuration

The AppId feature can be enabled via configuration. To enable it with the default settings use:

```
appid = { }
```

To use an AppId as a matching parameter in an IPS rule, use the *appids* keyword. For example, to block HTTP traffic that contains a specific header:

```
block tcp any any -> 192.168.0.1 any ( msg:"Block Malicious HTTP header";
  appids:"HTTP"; content:"X-Header: malicious"; sid:18000; )
```

Alternatively, the HTTP application can be specified in place of *tcp* instead of using the *appids* keyword. The AppId inspector will set the service when it is discovered so it can be used in IPS rules like this. Note that this rule also does not specify the IPs or ports which default to *any*.

```
block http ( msg:"Block Malicious HTTP header";
  content:"X-Header: malicious"; sid:18000; )
```

It's possible to specify multiple applications (as many as desired) with the appids keyword. A rule is considered a match if any of the applications on the rule match. Note that this rule does not match specific content which will reduce performance.

```
alert tcp any any -> 192.168.0.1 any ( msg:"Alert ";
  appids:"telnet,ssh,smtp,http";
```

Below is a minimal Snort configuration that is sufficient to block flows based on a specific HTTP header:

```
stream = { }

stream_tcp = { }

binder =
{
    {
        when =
        {
            proto = 'tcp',
            ports = [[ 80 8080 ]],
        },
        use =
        {
            type = 'http_inspect',
        },
    },
}

http_inspect = { }

appid = { }

local_rules =
[[
block http ( msg:"openAppId: test content match for app http";
content:"X-Header: malicious"; sid:18760; rev:4; )
]]
```

```
ips =
{
    rules = local_rules,
}
```

5.2.4 Session Application Identifiers

There are up to four AppIds stored in a session as defined below:

- serviceAppId - An appId associated with server side of a session. Example: http server.
- clientAppId - An appId associated with application on client side of a session. Example: Firefox.
- payloadAppId - For services like http this appId is associated with a webserver host. Example: Facebook.
- miscAppId - For some encapsulated protocols, this is the highest encapsulated application.

For packets originating from the client, a payloadAppid in a session is matched with all AppIds listed on a rule. Thereafter miscAppId, clientAppId and serviceAppId are matched. Since Alert Events contain one AppId, only the first match is reported. If a rule without an appids option matches, then the most specific appId (in order of payload, misc, client, server) is reported.

The same logic is followed for packets originating from the server with one exception. The order of matching is changed to make serviceAppId come before clientAppId.

5.2.5 AppId Usage Statistics

The AppId inspector prints application network usage periodically in the snort log directory in unified2 format. File name, time interval for statistic and file rollover are controlled by appId inspection configuration.

5.2.6 Open Detector Package (ODP) Installation

Application detectors from Snort team will be delivered in a separate package called the Open Detector Package (ODP) that can be downloaded from snort.org. ODP is a package that contains the following artifacts:

- Application detectors in the Lua language.
- appMapping.data file containing application metadata. This file should not be modified. The first column contains application identifier and second column contains application name. Other columns contain internal information.
- Lua library file DetectorCommon.lua.

A user can install the ODP package in any directory and configure this directory via the app_detector_dir option in the appid preprocessor configuration. Installing ODP will not modify any subdirectory named custom, where user-created detectors are located.

When installed, ODP will create following sub-directories:

- odp/lua //Cisco Lua detectors
- odp/libs //Cisco Lua modules

5.2.7 User Created Application Detectors

Users can detect new applications by adding detectors in the Lua language. A document will be posted on the Snort Website with details on API. Users can also copy over Snort team provided detectors and modify them. Users can also use the detector creation tool described in the next section.

Users must organize their Lua detectors and libraries by creating the following directory structure, under the ODP installation directory.

- custom/lua //Lua detectors
- custom/libs //Lua modules

The root path is specified by the "app_detector_dir" parameter of the appid section of snort.conf:

```
appid  =
{
    app_detector_dir = '/usr/local/lib/openappid',
}
```

So the path to the user-created lua files would be /usr/local/lib/openappid/custom/lua/

None of the directories below /usr/local/lib/openappid/ would be added for you.

5.2.8 Application Detector Reload

Both ODP detectors and user created detectors can be reloaded using the command appid.reload_detectors(). Detectors are expected to be updated in the path appid.app_detector_dir before this command is issued. The command takes no parameters.

5.2.9 Application Detector Creation Tool

For rudimentary Lua detectors, there is a tool provided called appid_detector_builder.sh. This is a simple, menu-driven bash script which creates .lua files in your current directory, based on your choices and on patterns you supply.

When you launch the script, it will prompt for the Application Id that you are giving for your detector. This is free-form ASCII with minor restrictions. The Lua detector file will be named based on your Application Id. If the file name already exists you will be prompted to overwrite it.

You will also be prompted for a description of your detector to be placed in the comments of the Lua source code. This is optional.

You will then be asked a series of questions designed to construct Lua code based on the kind of pattern data, protocol, port(s), etc.

When complete, the Protocol menu will be changed to include the option, "Save Detector". Instead of saving the file and exiting the script, you are allowed to give additional criteria for another pattern which may also be incorporated in the detection scheme. Then either pattern, when matched, will be considered a valid detection.

For example, your first choices might create an HTTP detection pattern of "example.com", and the next set of choices would add the HTTP detection pattern of "example.uk.co" (an equally fictional British counterpart). They would then co-exist in the Lua detector, and either would cause a detection with the name you give for your Application Id.

The resulting .lua file will need to be placed in the directory, "custom/lua", described in the previous section of the README above called "User Created Application Detectors"

5.3 Binder

One of the fundamental differences between Snort 2 and Snort 3 concerns configuration related to networks and ports. Here is a brief review of Snort 2 configuration for network and service related components:

- Snort's configuration has a default policy and optional policies selected by VLAN or network (with config binding).
- Each policy contains a user defined set of preprocessor configurations.
- Each preprocessor has a default configuration and some support non-default configurations selected by network.
- Most preprocessors have port configurations.
- The default policy may also contain a list of ports to ignore.

In Snort 3, the above configurations are done in a single module called the binder. Here is an example:

```
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 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' } }
}
```

Bindings are evaluated when a session starts and again if and when service is identified on the session. Essentially, the bindings are a list of when-use rules evaluated from top to bottom. The first matching network and service configurations are applied. binder.when can contain any combination of criteria and binder.use can specify an action, config file, or inspector configuration.

If binder is not explicitly configured (via file *.lua or option --lua), a default binder will be instantiated in which bindings will be created for all service inspectors configured. Some bindings may require a configured wizard to detect the service type.

5.4 Byte rule options

5.4.1 byte_test

This rule option tests a byte field against a specific value (with operator). Capable of testing binary values or converting representative byte strings to their binary equivalent and testing them.

Snort uses the C operators for each of these operators. If the & operator is used, then it would be the same as using

```
if (data & value) { do_something(); }
```

! operator negates the results from the base check. *!<oper>* is considered as

```
!(data <oper> value)
```

Note: The bitmask option applies bitwise AND operator on the bytes converted. The result will be right-shifted by the number of bits equal to the number of trailing zeros in the mask. This applies for the other rule options as well.

Examples

```
alert tcp (byte_test:2, =, 568, 0, bitmask 0x3FF0;)
```

This example extracts 2 bytes at offset 0, performs bitwise and with bitmask 0x3FF0, shifts the result by 4 bits and compares to 568.

```
alert udp (byte_test:4, =, 1234, 0, string, dec;
    msg:"got 1234!";)

alert udp (byte_test:8, =, 0xdeadbeef, 0, string, hex;
    msg:"got DEADBEEF!";)
```

5.4.2 byte_jump

The byte_jump rule option allows rules to be written for length encoded protocols trivially. By having an option that reads the length of a portion of data, then skips that far forward in the packet, rules can be written that skip over specific portions of length-encoded protocols and perform detection in very specific locations.

Examples

```
alert tcp (content:"Begin";
    byte_jump:0, 0, from_end, post_offset -6;
    content:"end..", distance 0, within 5;
    msg:"Content match from end of the payload";)

alert tcp (content:"catalog";
    byte_jump:2, 1, relative, post_offset 2, bitmask 0x03f0;
    byte_test:2, =, 968, 0, relative;
    msg:"Bitmask applied on the 2 bytes extracted for byte_jump";)
```

5.4.3 byte_extract

The byte_extract keyword is another useful option for writing rules against length-encoded protocols. It reads in some number of bytes from the packet payload and saves it to a variable. These variables can be referenced later in the rule, instead of using hard-coded values.

Other options which use byte_extract variables

A byte_extract rule option detects nothing by itself. Its use is in extracting packet data for use in other rule options.

Here is a list of places where byte_extract variables can be used:

- content/uricontent: offset, depth, distance, within
- byte_test: offset, value
- byte_jump: offset, post_offset
- isdataat: offset

Examples

```
alert tcp (byte_extract:1, 0, str_offset;
    byte_extract:1, 1, str_depth;
    content:"bad stuff", offset str_offset, depth str_depth;
    msg:"Bad Stuff detected within field";)

alert tcp (content:"START"; byte_extract:1, 0, myvar, relative;
    byte_jump:1, 3, relative, post_offset myvar;
    content:"END", distance 6, within 3;
    msg: "byte_jump - pass variable to post_offset";)
```

This example uses two variables.

The first variable keeps the offset of a string, read from a byte at offset 0. The second variable keeps the depth of a string, read from a byte at offset 1. These values are used to constrain a pattern match to a smaller area.

```
alert tcp (content:"|04 63 34 35|", offset 4, depth 4;
    byte_extract: 2, 0, var_match, relative, bitmask 0x03ff;
    byte_test: 2, =, var_match, 2, relative;
    msg:"Test value match, after applying bitmask on bytes extracted";)
```

5.4.4 byte_math

Perform a mathematical operation on an extracted value and a specified value or existing variable, and store the outcome in a new resulting variable. These resulting variables can be referenced later in the rule, at the same places as byte_extract variables.

The syntax for this rule option is different. The order of the options is critical for the other rule options and can't be changed. For example, the first option is the number of bytes to extract. Here the name of the option is explicitly written, for example : bytes 2. The order is not important.

Note
Byte_math operations are performed on unsigned 32-bit values. When writing a rule it should be taken into consideration to avoid wrap around.

Examples

```
alert tcp ( byte_math: bytes 2, offset 0, oper *, rvalue 10, result area;
   byte_test:2,>,area,16;)
```

At the zero offset of the payload, extract 2 bytes and apply multiplication operation with value 10. Store result in variable area. The area variable is given as input to byte_test value option.

Let's consider 2 bytes of extracted data is 5. The rvalue is 10. Result variable area is 50 (5 * 10). Area variable can be used in either byte_test offset/value options.

5.4.5 Testing Numerical Values

The rule options byte_test and byte_jump were written to support writing rules for protocols that have length encoded data. RPC was the protocol that spawned the requirement for these two rule options, as RPC uses simple length based encoding for passing data.

In order to understand why byte test and byte jump are useful, let's go through an exploit attempt against the sadmind service.

This is the payload of the exploit:

```
89 09 9c e2 00 00 00 00 00 00 00 02 00 01 87 88  ................
00 00 00 0a 00 00 00 01 00 00 00 01 00 00 00 20  ................
40 28 3a 10 00 00 00 0a 4d 45 54 41 53 50 4c 4f  @(:.....metasplo
49 54 00 00 00 00 00 00 00 00 00 00 00 00 00 00  it..............
00 00 00 00 00 00 00 00 40 28 3a 14 00 07 45 df  ........@(:...e.
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 06 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 04 00 00 00 00 00 00 00 04  ................
7f 00 00 01 00 01 87 88 00 00 00 0a 00 00 00 04  ................
7f 00 00 01 00 01 87 88 00 00 00 0a 00 00 00 11  ................
00 00 00 1e 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 3b 4d 45 54 41 53 50 4c 4f  .......;metasplo
49 54 00 00 00 00 00 00 00 00 00 00 00 00 00 00  it..............
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 06 73 79 73 74 65 6d 00 00  ........system..
00 00 00 15 2e 2e 2f 2e 2e 2f 2e 2e 2f 2e 2e 2f  ....../../../../
2e 2e 2f 62 69 6e 2f 73 68 00 00 00 00 00 04 1e  ../bin/sh.......
```

Let's break this up, describe each of the fields, and figure out how to write a rule to catch this exploit.

There are a few things to note with RPC:

Numbers are written as uint32s, taking four bytes. The number 26 would show up as 0x0000001a.

Strings are written as a uint32 specifying the length of the string, the string, and then null bytes to pad the length of the string to end on a 4-byte boundary. The string *bob* would show up as 0x00000003626f6200.

```
89 09 9c e2     - the request id, a random uint32, unique to each request
00 00 00 00     - rpc type (call = 0, response = 1)
00 00 00 02     - rpc version (2)
00 01 87 88     - rpc program (0x00018788 = 100232 = sadmind)
00 00 00 0a     - rpc program version (0x0000000a = 10)
00 00 00 01     - rpc procedure (0x00000001 = 1)
00 00 00 01     - credential flavor (1 = auth_unix)
00 00 00 20     - length of auth_unix data (0x20 = 32)

## the next 32 bytes are the auth_unix data
40 28 3a 10 - unix timestamp (0x40283a10 = 1076378128 = feb 10 01:55:28 2004 gmt)
00 00 00 0a - length of the client machine name (0x0a = 10)
4d 45 54 41 53 50 4c 4f 49 54 00 00  - metasploit

00 00 00 00 - uid of requesting user (0)
00 00 00 00 - gid of requesting user (0)
00 00 00 00 - extra group ids (0)

00 00 00 00 - verifier flavor (0 = auth_null, aka none)
00 00 00 00 - length of verifier (0, aka none)
```

The rest of the packet is the request that gets passed to procedure 1 of sadmind.

However, we know the vulnerability is that sadmind trusts the uid coming from the client. sadmind runs any request where the client's uid is 0 as root. As such, we have decoded enough of the request to write our rule.

First, we need to make sure that our packet is an RPC call.

```
content:"|00 00 00 00|", offset 4, depth 4;
```

Then, we need to make sure that our packet is a call to sadmind.

```
content:"|00 01 87 88|", offset 12, depth 4;
```

Then, we need to make sure that our packet is a call to the procedure 1, the vulnerable procedure.

```
content:"|00 00 00 01|", offset 20, depth 4;
```

Then, we need to make sure that our packet has auth_unix credentials.

```
content:"|00 00 00 01|", offset 24, depth 4;
```

We don't care about the hostname, but we want to skip over it and check a number value after the hostname. This is where byte_test is useful. Starting at the length of the hostname, the data we have is:

```
00 00 00 0a 4d 45 54 41 53 50 4c 4f 49 54 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00
```

We want to read 4 bytes, turn it into a number, and jump that many bytes forward, making sure to account for the padding that RPC requires on strings. If we do that, we are now at:

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00
```

which happens to be the exact location of the uid, the value we want to check.

In English, we want to read 4 bytes, 36 bytes from the beginning of the packet, and turn those 4 bytes into an integer and jump that many bytes forward, aligning on the 4-byte boundary. To do that in a Snort rule, we use:

```
byte_jump:4,36,align;
```

then we want to look for the uid of 0.

```
content:"|00 00 00 00|", within 4;
```

Now that we have all the detection capabilities for our rule, let's put them all together.

```
content:"|00 00 00 00|", offset 4, depth 4;
content:"|00 01 87 88|", offset 12, depth 4;
content:"|00 00 00 01|", offset 20, depth 4;
content:"|00 00 00 01|", offset 24, depth 4;
byte_jump:4,36,align;
content:"|00 00 00 00|", within 4;
```

The 3rd and fourth string match are right next to each other, so we should combine those patterns. We end up with:

```
content:"|00 00 00 00|", offset 4, depth 4;
content:"|00 01 87 88|", offset 12, depth 4;
content:"|00 00 00 01 00 00 00 01|", offset 20, depth 8;
byte_jump:4,36,align;
content:"|00 00 00 00|", within 4;
```

If the sadmind service was vulnerable to a buffer overflow when reading the client's hostname, instead of reading the length of the hostname and jumping that many bytes forward, we would check the length of the hostname to make sure it is not too large.

To do that, we would read 4 bytes, starting 36 bytes into the packet, turn it into a number, and then make sure it is not too large (let's say bigger than 200 bytes). In Snort, we do:

```
byte_test:4,>,200,36;
```

Our full rule would be:

```
content:"|00 00 00 00|", offset 4, depth 4;
content:"|00 01 87 88|", offset 12, depth 4;
content:"|00 00 00 01 00 00 00 01|", offset 20, depth 8;
byte_test:4,>,200,36;
```

5.5 Consolidated Config

Config dump mode generates a consolidated dump of the config passed to Snort. This output consists of the configured values as well as the module defaults for the values that aren't configured.

In the dump mode Snort validates the config (similar to option -T) and suppresses unrelated messages going to stdout (configuration warnings and errors are still printed to stderr).

The dump mode is activated by the following options: --dump-config-text, --dump-config=all, --dump-config=top. They are described in detail below.

The simple configuration is used in examples. The output contains applied configurations (defaults and configured). To simplify the output we show a brief list of default options.

snort.lua

```
stream =
{
    max_flows = 2
}

stream_tcp =
{
    show_rebuilt_packets = true
}

binder =
{
    { when = { nets = '10.1.2.0/24' }, use = { inspection_policy = 'http.lua' } },
    { when = { nets = '192.168.2.0/24' }, use = { inspection_policy = 'sip.lua' }  ↩
       },
}
```

http.lua

```
wizard =
{
    spells =
    {
        { service = 'http', proto = 'tcp', client_first = true, to_server = { 'GET ↩
           ' }, to_client = { 'HTTP/' } },
    }
}
```

sip.lua

```
wizard =
{
    spells =
    {
        { service = 'sip', to_server = { 'INVITE' } },
    }
}
```

5.5.1 Text Format

The --dump-config-text option verifies the configuration and dumps it to stdout in text format. The output contains a config of the main policy and all other included sub-policies.

Example: snort -c snort.lua --dump-config-text

```
consolidated config for snort.lua
alerts.order="pass reset block drop alert log"
alerts.rate_filter_memcap=1048576
binder[0].when.ips_policy_id=0
binder[0].when.role="any"
binder[0].when.nets="10.1.2.0/24"
binder[0].use.action="inspect"
binder[0].use.inspection_policy="http.lua"
binder[1].when.ips_policy_id=0
binder[1].when.role="any"
binder[1].when.nets="192.168.2.0/24"
binder[1].use.action="inspect"
binder[1].use.inspection_policy="sip.lua"
output.obfuscate=false
output.wide_hex_dump=true
packets.address_space_agnostic=false
packets.limit=0
search_engine.split_any_any=true
search_engine.queue_limit=128
stream.file_cache.idle_timeout=180
stream.file_cache.cap_weight=32
stream.max_flows=2
stream_tcp.small_segments.maximum_size=0
stream_tcp.session_timeout=30
stream_tcp.track_only=false
stream_tcp.show_rebuilt_packets=true
consolidated config for http.lua
wizard.spells[0].proto="tcp"
wizard.spells[0].client_first=true
wizard.spells[0].service="http"
wizard.spells[0].to_client[0].spell="HTTP/"
wizard.spells[0].to_server[0].spell="GET"
consolidated config for sip.lua
wizard.spells[0].proto="tcp"
wizard.spells[0].client_first=true
wizard.spells[0].service="sip"
wizard.spells[0].to_server[0].spell="INVITE"
```

For lists, the index next to the option name designates an element parsing order.

5.5.2 JSON Format

The --dump-config=all command-line option verifies the configuration and dumps it to stdout in JSON format. The output contains a config of the main policy and all other included sub-policies. Snort dumps output in a one-line format.

There is 3rd party tool *jq* for converting to a pretty printed format.

Example: snort -c snort.lua --dump-config=all | jq .


```
[
    {
        "filename": "snort.lua",
        "config": {
        "alerts": {
            "order": "pass reset block drop alert log",
            "rate_filter_memcap": 1048576
        },
        "binder": [
            {
            "when": {
                "ips_policy_id": 0,
                "role": "any",
                "nets": "10.1.2.0/24"
            },
            "use": {
                "action": "inspect",
                "inspection_policy": "http.lua"
            }
            },
            {
            "when": {
                "ips_policy_id": 0,
                "role": "any",
                "nets": "192.168.2.0/24"
            },
            "use": {
                "action": "inspect",
                "inspection_policy": "sip.lua"
            }
            }
        ],
        "output": {
            "obfuscate": false,
            "wide_hex_dump": true
        },
        "packets": {
            "address_space_agnostic": false,
            "limit": 0
        },
        "process": {
            "daemon": false,
            "dirty_pig": false,
            "utc": false
        },
        "search_engine": {
            "split_any_any": true,
            "queue_limit": 128
        },
```

```
        "stream": {
            "file_cache": {
            "idle_timeout": 180,
            "cap_weight": 32
            },
            "max_flows": 2
        },
        "stream_tcp": {
            "small_segments": {
                "maximum_size": 0
            },
            "session_timeout": 30,
            "track_only": false,
            "show_rebuilt_packets": true
        }
        }
    },
    {
        "filename": "http.lua",
        "config": {
        "wizard": {
            "spells": [
            {
                "proto": "tcp",
                "client_first": true,
                "service": "http",
                "to_client": [
                {
                    "spell": "HTTP/"
                }
                ],
                "to_server": [
                {
                    "spell": "GET"
                }
                ]
            }
            ]
        }
        }
    },
    {
        "filename": "sip.lua",
        "config": {
        "wizard": {
            "spells": [
            {
                "proto": "tcp",
                "client_first": true,
                "service": "sip",
                "to_server": [
                {
                    "spell": "INVITE"
                }
                ]
            }
            ]
```

```
        }
        }
    }
]
```

The --dump-config=top command-line option is similar to --dump-config=all, except it produces dump for the main policy only. It verifies the configuration and dumps the main policy configuration to stdout in JSON format.

Example: snort -c snort.lua --dump-config=top | jq .

```
{
    "alerts": {
        "order": "pass reset block drop alert log",
        "rate_filter_memcap": 1048576,
    },
    "binder": [
        {
            "when": {
                "ips_policy_id": 0,
                "role": "any",
                "nets": "10.1.2.0/24"
            },
            "use": {
                "action": "inspect",
                "inspection_policy": "http.lua"
            }
        },
        {
            "when": {
                "ips_policy_id": 0,
                "role": "any",
                "nets": "192.168.2.0/24"
            },
            "use": {
                "action": "inspect",
                "inspection_policy": "sip.lua"
            }
        }
    ],
    "output": {
        "obfuscate": false,
        "wide_hex_dump": true
    },
    "packets": {
        "address_space_agnostic": false,
        "limit": 0,
    },
    "process": {
        "daemon": false,
        "dirty_pig": false,
        "utc": false
    },
    "search_engine": {
        "split_any_any": true,
        "queue_limit": 128
    },
    "stream": {
        "file_cache": {
```

```
            "idle_timeout": 180,
            "cap_weight": 32
        }
        "max_flows": 2
    },
    "stream_tcp": {
        "small_segments": {
            "count": 0,
            "maximum_size": 0
        },
        "session_timeout": 30,
        "track_only": false,
        "show_rebuilt_packets": true
    },
}
```

5.6 DCE Inspectors

The main purpose of these inspector are to perform SMB desegmentation and DCE/RPC defragmentation to avoid rule evasion using these techniques.

5.6.1 Overview

The following transports are supported for DCE/RPC: SMB, TCP, and UDP. New rule options have been implemented to improve performance, reduce false positives and reduce the count and complexity of DCE/RPC based rules.

Different from Snort 2, the DCE-RPC preprocessor is split into three inspectors - one for each transport: dce_smb, dce_tcp, dce_udp. This includes the configuration as well as the inspector modules. The Snort 2 server configuration is now split between the inspectors. Options that are meaningful to all inspectors, such as policy and defragmentation, are copied into each inspector configuration. The address/port mapping is handled by the binder. Autodetect functionality is replaced by wizard curses.

5.6.2 Quick Guide

A typical dcerpce configuration looks like this:

```
binder =
{
   {
       when =
       {
           proto = 'tcp',
           ports = '139 445 1025',
        },
       use =
       {
           type = 'dce_smb',
       },
    },
    {
       when =
       {
           proto = 'tcp',
           ports = '135 2103',
       },
       use =
```

```
        {
            type = 'dce_tcp',
        },
    },
    {
        when =
        {
            proto = 'udp',
            ports = '1030',
        },
        use =
        {
            type = 'dce_udp',
        },
    }
}

dce_smb = { }

dce_tcp = { }

dce_udp = { }
```

In this example, it defines smb, tcp and udp inspectors based on port. All the configurations are default.

5.6.3 Target Based

There are enough important differences between Windows and Samba versions that a target based approach has been implemented. Some important differences:

- Named pipe instance tracking
- Accepted SMB commands
- AndX command chaining
- Transaction tracking
- Multiple Bind requests
- DCE/RPC Fragmented requests - Context ID
- DCE/RPC Fragmented requests - Operation number
- DCE/RPC Stub data byte order

Because of those differences, each inspector can be configured to different policy. Here are the list of policies supported:

- WinXP (default)
- Win2000
- WinVista
- Win2003
- Win2008
- Win7

- Samba
- Samba-3.0.37
- Samba-3.0.22
- Samba-3.0.20

5.6.4 Reassembling

Both SMB inspector and TCP inspector support reassemble. Reassemble threshold specifies a minimum number of bytes in the DCE/RPC desegmentation and defragmentation buffers before creating a reassembly packet to send to the detection engine. This option is useful in inline mode so as to potentially catch an exploit early before full defragmentation is done. A value of 0 s supplied as an argument to this option will, in effect, disable this option. Default is disabled.

5.6.5 SMB

SMB inspector is one of the most complex inspectors. In addition to supporting rule options and lots of inspector rule events, it also supports file processing for both SMB version 1, 2, and 3.

Finger Print Policy

In the initial phase of an SMB session, the client needs to authenticate with a SessionSetupAndX. Both the request and response to this command contain OS and version information that can allow the inspector to dynamically set the policy for a session which allows for better protection against Windows and Samba specific evasions.

File Inspection

SMB inspector supports file inspection. A typical configuration looks like this:

```
binder =
{
   {
       when =
       {
           proto = 'tcp',
           ports = '139 445',
       },
       use =
       {
           type = 'dce_smb',
       },
   },
}

dce_smb =
{
    smb_file_inspection = 'on',
    smb_file_depth = 0,
 }

file_id =
{
    enable_type = true,
    enable_signature = true,
```

```
    enable_capture = true,
    file_rules = magics,
}
```

First, define a binder to map tcp port 139 and 445 to smb. Then, enable file inspection in smb inspection and set the file depth as unlimited. Lastly, enable file inspector to inspect file type, calculate file signature, and capture file. The details of file inspector are explained in file processing section.

SMB inspector does inspection of normal SMB file transfers. This includes doing file type and signature through the file processing as well as setting a pointer for the "file_data" rule option. Note that the "file_depth" option only applies to the maximum amount of file data for which it will set the pointer for the "file_data" rule option. For file type and signature it will use the value configured for the file API. If "only" is specified, the inspector will only do SMB file inspection, i.e. it will not do any DCE/RPC tracking or inspection. If "on" is specified with no arguments, the default file depth is 16384 bytes. An argument of -1 to "file-depth" disables setting the pointer for "file_data", effectively disabling SMB file inspection in rules. An argument of 0 to "file_depth" means unlimited. Default is "off", i.e. no SMB file inspection is done in the inspector.

5.6.6 TCP

dce_tcp inspector supports defragmentation, reassembling, and policy that is similar to SMB.

5.6.7 UDP

dce_udp is a very simple inspector that only supports defragmentation

5.6.8 Rule Options

New rule options are supported by enabling the dcerpc2 inspectors:

- dce_iface
- dce_opnum
- dce_stub_data

New modifiers to existing byte_test and byte_jump rule options:

- byte_test: dce
- byte_jump: dce

dce_iface

For DCE/RPC based rules it has been necessary to set flow-bits based on a client bind to a service to avoid false positives. It is necessary for a client to bind to a service before being able to make a call to it. When a client sends a bind request to the server, it can, however, specify one or more service interfaces to bind to. Each interface is represented by a UUID. Each interface UUID is paired with a unique index (or context id) that future requests can use to reference the service that the client is making a call to. The server will respond with the interface UUIDs it accepts as valid and will allow the client to make requests to those services. When a client makes a request, it will specify the context id so the server knows what service the client is making a request to. Instead of using flow-bits, a rule can simply ask the inspector, using this rule option, whether or not the client has bound to a specific interface UUID and whether or not this client request is making a request to it. This can eliminate false positives where more than one service is bound to successfully since the inspector can correlate the bind UUID to the context id used in the request. A DCE/RPC request can specify whether numbers are represented as big endian or little endian. The representation of the interface UUID is different depending on the endianness specified in the DCE/RPC previously requiring two rules - one for big endian and one for little endian. The inspector eliminates the need for two rules by normalizing the UUID. An interface contains a version. Some versions of an interface may not be vulnerable to a certain exploit. Also, a DCE/RPC request can be

broken up into 1 or more fragments. Flags (and a field in the connectionless header) are set in the DCE/RPC header to indicate whether the fragment is the first, a middle or the last fragment. Many checks for data in the DCE/RPC request are only relevant if the DCE/RPC request is a first fragment (or full request), since subsequent fragments will contain data deeper into the DCE/RPC request. A rule which is looking for data, say 5 bytes into the request (maybe it's a length field), will be looking at the wrong data on a fragment other than the first, since the beginning of subsequent fragments are already offset some length from the beginning of the request. This can be a source of false positives in fragmented DCE/RPC traffic. By default it is reasonable to only evaluate if the request is a first fragment (or full request). However, if the "any_frag" option is used to specify evaluating on all fragments.

Examples:

```
dce_iface: 4b324fc8-1670-01d3-1278-5a47bf6ee188;
dce_iface: 4b324fc8-1670-01d3-1278-5a47bf6ee188,<2;
dce_iface: 4b324fc8-1670-01d3-1278-5a47bf6ee188,any_frag;
dce_iface: 4b324fc8-1670-01d3-1278-5a47bf6ee188,=1,any_frag;
```

This option is used to specify an interface UUID. Optional arguments are an interface version and operator to specify that the version be less than (<), greater than (>), equal to (=) or not equal to (!) the version specified. Also, by default the rule will only be evaluated for a first fragment (or full request, i.e. not a fragment) since most rules are written to start at the beginning of a request. The "any_frag" argument says to evaluate for middle and last fragments as well. This option requires tracking client Bind and Alter Context requests as well as server Bind Ack and Alter Context responses for connection-oriented DCE/RPC in the inspector. For each Bind and Alter Context request, the client specifies a list of interface UUIDs along with a handle (or context id) for each interface UUID that will be used during the DCE/RPC session to reference the interface. The server response indicates which interfaces it will allow the client to make requests to - it either accepts or rejects the client's wish to bind to a certain interface. This tracking is required so that when a request is processed, the context id used in the request can be correlated with the interface UUID it is a handle for.

hexlong and hexshort will be specified and interpreted to be in big endian order (this is usually the default way an interface UUID will be seen and represented). As an example, the following Messenger interface UUID as taken off the wire from a little endian Bind request:

```
|f8 91 7b 5a 00 ff d0 11 a9 b2 00 c0 4f b6 e6 fc|
```

must be written as:

```
5a7b91f8-ff00-11d0-a9b2-00c04fb6e6fc
```

The same UUID taken off the wire from a big endian Bind request:

```
|5a 7b 91 f8 ff 00 11 d0 a9 b2 00 c0 4f b6 e6 fc|
```

must be written the same way:

```
5a7b91f8-ff00-11d0-a9b2-00c04fb6e6fc
```

This option matches if the specified interface UUID matches the interface UUID (as referred to by the context id) of the DCE/RPC request and if supplied, the version operation is true. This option will not match if the fragment is not a first fragment (or full request) unless the "any_frag" option is supplied in which case only the interface UUID and version need match. Note that a defragmented DCE/RPC request will be considered a full request.

Using this rule option will automatically insert fast pattern contents into the fast pattern matcher. For UDP rules, the interface UUID, in both big and little endian format will be inserted into the fast pattern matcher. For TCP rules, (1) if the rule option "flow:to_server|from_client" is used, |05 00 00| will be inserted into the fast pattern matcher, (2) if the rule option "flow:from_server|to_client" is used, |05 00 02| will be inserted into the fast pattern matcher and (3) if the flow isn't known, |05 00| will be inserted into the fast pattern matcher. Note that if the rule already has content rule options in it, the best (meaning longest) pattern will be used. If a content in the rule uses the fast_pattern rule option, it will unequivocally be used over the above mentioned patterns.

dce_opnum

The opnum represents a specific function call to an interface. After is has been determined that a client has bound to a specific interface and is making a request to it (see above - dce_iface) usually we want to know what function call it is making to that service. It is likely that an exploit lies in the particular DCE/RPC function call.

Examples:

```
dce_opnum: 15;
dce_opnum: 15-18;
dce_opnum: 15,18-20;
dce_opnum: 15,17,20-22;
```

This option is used to specify an opnum (or operation number), opnum range or list containing either or both opnum and/or opnum-range. The opnum of a DCE/RPC request will be matched against the opnums specified with this option. This option matches if any one of the opnums specified match the opnum of the DCE/RPC request.

dce_stub_data

Since most DCE/RPC based rules had to do protocol decoding only to get to the DCE/RPC stub data, i.e. the remote procedure call or function call data, this option will alleviate this need and place the cursor at the beginning of the DCE/RPC stub data. This reduces the number of rule option checks and the complexity of the rule.

This option takes no arguments.

Example:

```
dce_stub_data;
```

This option is used to place the cursor (used to walk the packet payload in rules processing) at the beginning of the DCE/RPC stub data, regardless of preceding rule options. There are no arguments to this option. This option matches if there is DCE/RPC stub data.

The cursor is moved to the beginning of the stub data. All ensuing rule options will be considered "sticky" to this buffer. The first rule option following dce_stub_data should use absolute location modifiers if it is position-dependent. Subsequent rule options should use a relative modifier if they are meant to be relative to a previous rule option match in the stub data buffer. Any rule option that does not specify a relative modifier will be evaluated from the start of the stub data buffer. To leave the stub data buffer and return to the main payload buffer, use the "pkt_data" rule option.

byte_test and byte_jump

A DCE/RPC request can specify whether numbers are represented in big or little endian. These rule options will take as a new argument "dce" and will work basically the same as the normal byte_test/byte_jump, but since the DCE/RPC inspector will know the endianness of the request, it will be able to do the correct conversion.

Examples:

```
byte_test: 4,>,35000,0,relative,dce;
byte_test: 2,!=,2280,-10,relative,dce;
```

When using the "dce" argument to a byte_test, the following normal byte_test arguments will not be allowed: "big", "little", "string", "hex", "dec" and "oct".

Examples:

```
byte_jump:4,-4,relative,align,multiplier 2,post_offset -4,dce;
```

When using the dce argument to a byte_jump, the following normal byte_jump arguments will not be allowed: "big", "little", "string", "hex", "dec", "oct" and "from_beginning"

5.7 File Processing

With the volume of malware transferred through network increasing, network file inspection becomes more and more important. This feature will provide file type identification, file signature creation, and file capture capabilities to help users deal with those challenges.

5.7.1 Overview

There are two parts of file services: file APIs and file policy. File APIs provides all the file inspection functionalities, such as file type identification, file signature calculation, and file capture. File policy provides users ability to control file services, such as enable/disable/configure file type identification, file signature, or file capture.

In addition to all capabilities from Snort 2, we support customized file policy along with file event log.

- Supported protocols: HTTP, SMTP, IMAP, POP3, FTP, and SMB.
- Supported file signature calculation: SHA256

5.7.2 Quick Guide

A very simple configuration has been included in lua/snort.lua file. A typical file configuration looks like this:

```
dofile('magic.lua')

my_file_policy =
{
    {  when = { file_type_id = 0 }, use = { verdict = 'log', enable_file_signature ↩
        = true, enable_file_capture = true } }
    {  when = { file_type_id = 22 }, use = { verdict = 'log',  ↩
       enable_file_signature = true } },
    {  when = { sha256 = " ↩
       F74DC976BC8387E7D4FC0716A069017A0C7ED13F309A523CC41A8739CCB7D4B6" }, use =  ↩
       { verdict = 'block'} },
}

file_id =
{
    enable_type = true,
    enable_signature = true,
    enable_capture = true,
    file_rules = magics,
    trace_type = true,
    trace_signature = true,
    trace_stream = true,
    file_policy = my_file_policy,
 }

file_log =
{
    log_pkt_time = true,
    log_sys_time = false,
}
```

There are 3 steps to enable file processing:

- First, you need to include the file magic rules.
- Then, define the file policy and configure the inspector
- At last, enable file_log to get detailed information about file event

5.7.3 Pre-packaged File Magic Rules

A set of file magic rules is packaged with Snort. They can be located at "lua/file_magic.lua". To use this feature, it is recommended that these pre-packaged rules are used; doing so requires that you include the file in your Snort configuration as such (already in snort.lua):

```
dofile('magic.lua')
```

Example:

```
{ type = "GIF", id = 62, category = "Graphics", rev = 1,
  magic = { { content = "| 47 49 46 38 37 61 |",offset = 0 } } },

{ type = "GIF", id = 63, category = "Graphics", rev = 1,
  magic = { { content = "| 47 49 46 38 39 61 |",offset = 0 } } },
```

The previous two rules define GIF format, because two file magics are different. File magics are specified by content and offset, which look at content at particular file offset to identify the file type. In this case, two magics look at the beginning of the file. You can use character if it is printable or hex value in between "|".

5.7.4 File Policy

You can enabled file type, file signature, or file capture by configuring file_id. In addition, you can enable trace to see file stream data, file type, and file signature information.

Most importantly, you can configure a file policy that can block/alert some file type or an individual file based on SHA. This allows you build a file blacklist or whitelist.

Example:

```
file_policy =
{
    {  when = { file_type_id = 22 }, use = { verdict = 'log',  ↩
       enable_file_signature = true } },
    {  when = { sha256 = " ↩
       F74DC976BC8387E7D4FC0716A069017A0C7ED13F309A523CC41A8739CCB7D4B6" }, use =  ↩
       { verdict = 'block'} },
    {  when = { file_type_id = 0 }, use = { verdict = 'log', enable_file_signature ↩
        = true, enable_file_capture = true } }
}
```

In this example, it enables this policy:

- For PDF files, they will be logged with signatures.
- For the file matching this SHA, it will be blocked
- For all file types identified, they will be logged with signature, and also captured onto log folder.

5.7.5 File Capture

File can be captured and stored to log folder. We use SHA as file name instead of actual file name to avoid conflicts. You can capture either all files, some file type, or a particular file based on SHA.

You can enable file capture through this config:

```
enable_capture = true,
```

or enable it for some file or file type in your file policy:

```
{  when = { file_type_id = 22 }, use = { verdict = 'log', enable_file_capture =  ↩
   true } },
```

The above rule will enable PDF file capture.

5.7.6 File Events

File inspect preprocessor also works as a dynamic output plugin for file events. It logs basic information about file. The log file is in the same folder as other log files with name starting with "file.log".

Example:

```
file_log = { log_pkt_time = true, log_sys_time = false }
```

All file events will be logged in packet time, system time is not logged.

File event example:

```
08/14-19:14:19.100891  10.22.75.72:33734 -> 10.22.75.36:80,
[Name: "malware.exe"] [Verdict: Block] [Type: MSEXE]
[SHA: 6F26E721FDB1AAFD29B41BCF90196DEE3A5412550615A856DAE8E3634BCE9F7A]
[Size: 1039328]
```

5.8 High Availability

High Availability includes the HA flow synchronization and the SideChannel messaging subsystems.

5.8.1 HA

HighAvailability (or HA) is a Snort module that provides state coherency between two partner snort instances. It uses SideChannel for messaging.

There can be multiple types of HA within Snort and Snort plugins. HA implements an extensible architecture to enable plugins to subscribe to the base flow HA messaging. These plugins can then include their own messages along with the flow cache HA messages.

HA produces and consumes two type of messages:

- Update - Update flow status. Plugins may add their own data to the messages
- Delete - A flow has been removed from the cache

The HA module is configured with these items:

```
high_availability =
{
    ports = "1",
    enable = true,
    min_age = 0,
    min_sync = 0
}
```

The *ports* item maps to the SideChannel port to use for the HA messaging.

The *enabled* item controls the overall HA operation.

The items min_age and min_sync are used in the stream HA logic. min_age is the number of milliseconds that a flow must exist in the flow cache before sending HA messages to the partner. min_sync is the minimum time between HA status updates. HA messages for a particular flow will not be sent faster than min_sync. Both are expressed as a number of milliseconds.

HA messages are composed of the base *stream* information plus any content from additional modules. Modules subscribe HA in order to add message content. The *stream* HA content is always present in the messages while the ancillary module content is only present when requested via a status change request.

5.8.2 Connector

Connectors are a set of modules that are used to exchange message-oriented data among Snort threads and the external world. A typical use-case is HA (High Availability) message exchange. Connectors serve to decouple the message transport from the message creation/consumption. Connectors expose a common API for several forms of message transport.

Connectors are a Snort plugin type.

Connector (parent plugin class)

Connectors may either be a simplex channel and perform unidirectional communications. Or may be duplex and perform bidirectional communications. The TcpConnector is duplex while the FileConnector is simplex.

All subtypes of Connector have a *direction* configuration element and a *connector* element. The *connector* string is the key used to identify the element for sidechannel configuration. The *direction* element may have a default value, for instance TcpConnector's are *duplex*.

There are currently two implementations of Connectors:

- TcpConnector - Exchange messages over a tcp channel.
- FileConnector - Write messages to files and read messages from files.

TcpConnector

TcpConnector is a subclass of Connector and implements a DUPLEX type Connector, able to send and receive messages over a tcp session.

TcpConnector adds a few session setup configuration elements:

- setup = *call* or *answer* - *call* is used to have TcpConnector initiate the connection. *answer* is used to have TcpConnector accept incoming connections.
- address = <*addr*> - used for *call* setup to specify the partner
- base_port = port - used to construct the actual port number for *call* and *answer* modes. Actual port used is (base_port + instance_id).

An example segment of TcpConnector configuration:

```
tcp_connector =
{
    {
        connector = 'tcp_1',
        address = '127.0.0.1',
        setup = 'call',
        base_port = 11000
    },
}
```

FileConnector

FileConnector implements a Connector that can either read from files or write to files. FileConnector's are simplex and must be configured to be CONN_TRANSMIT or CONN_RECEIVE.

FileConnector configuration adds two additional element:

- name = string - used as part of the message file name
- format = *text* or *binary* - FileConnector supports two file types

The configured *name* string is used to construct the actual names as in:

- file_connector_NAME_transmit and file_connector_NAME_receive

All messages for one Snort invocation are read and written to one file.

In the case of a receive FileConnector, all messages are read from the file prior to the start of packet processing. This allows the messages to establish state information for all processed packets.

Connectors are used solely by SideChannel

An example segment of FileConnector configuration:

```
file_connector =
{
    {
        connector = 'file_tx_1',
        direction = 'transmit',
        format = 'text',
        name = 'HA'
    },
    {
        connector = 'file_rx_1',
        direction = 'receive',
        format = 'text',
        name = 'HA'
    },
}
```

5.8.3 Side Channel

SideChannel is a Snort module that uses Connectors to implement a messaging infrastructure that is used to communicate between Snort threads and the outside world.

SideChannel adds functionality onto the Connector as:

- message multiplexing/demultiplexing - An additional protocol layer is added to the messages. This port number is used to direct message to/from various SideClass instancs.
- application receive processing - handler for received messages on a specific port.

SideChannel's are always implement a duplex (bidirectional) messaging model and can map to separate transmit and receive Connectors.

The message handling model leverages the underlying Connector handling. So please refer to the Connector documentation.

SideChannel's are instantiated by various applications. The SideChannel port numbers are the configuration element used to map SideChannel's to applications.

The SideChannel configuration mostly serves to map a port number to a Connector or set of connectors. Each port mapping can have at most one transmit plus one receive connector or one duplex connector. Multiple SideChannel's may be configured and instantiated to support multiple applications.

An example SideChannel configuration along with the corresponding Connector configuration:

```
side_channel =
{
    {
        ports = '1',
        connectors =
        {
            {
                connector = 'file_rx_1',
            },
            {
                connector = 'file_tx_1',
            }
        },
    },
}

file_connector =
{
    {
        connector = 'file_tx_1',
        direction = 'transmit',
        format = 'text',
        name = 'HA'
    },
    {
        connector = 'file_rx_1',
        direction = 'receive',
        format = 'text',
        name = 'HA'
    },
}
```

5.9 FTP

Given an FTP command channel buffer, FTP will interpret the data, identifying FTP commands and parameters, as well as FTP response codes and messages. It will enforce correctness of the parameters, determine when an FTP command connection is encrypted, and determine when an FTP data channel is opened.

5.9.1 Configuring the inspector to block exploits and attacks

ftp_server configuration

- ftp_cmds

This specifies additional FTP commands outside of those checked by default within the inspector. The inspector may be configured to generate an alert when it sees a command it does not recognize.

Aside from the default commands recognized, it may be necessary to allow the use of the "X" commands, specified in RFC 775. To do so, use the following ftp_cmds option. Since these are rarely used by FTP client implementations, they are not included in the defaults.

```
ftp_cmds = [[ XPWD XCWD XCUP XMKD XRMD ]]
```

- def_max_param_len

This specifies the default maximum parameter length for all commands in bytes. If the parameter for an FTP command exceeds that length, and the inspector is configured to do so, an alert will be generated. This is used to check for buffer overflow exploits within FTP servers.

- cmd_validity

This specifies the valid format and length for parameters of a given command.

- cmd_validity[].len

This specifies the maximum parameter length for the specified command in bytes, overriding the default. If the parameter for that FTP command exceeds that length, and the inspector is configured to do so, an alert will be generated. It can be used to restrict specific commands to small parameter values. For example the USER command — usernames may be no longer than 16 bytes, so the appropriate configuration would be:

```
cmd_validity =
{
    {
        command = 'USER',
        length = 16,
    }
}
```

- cmd_validity[].format

format is as follows:

```
int                 Param must be an integer
number              Param must be an integer between 1 and 255
char <chars>        Param must be a single char, and one of <chars>
date <datefmt>      Param follows format specified where
                    # = Number, C=Char, []=optional, |=OR, {}=choice,
                    anything else=literal (i.e., .+- )
string              Param is string (effectively unrestricted)
host_port           Param must a host port specifier, per RFC 959.
long_host_port      Parameter must be a long host port specified, per RFC 1639
extended_host_port  Parameter must be an extended host port specified, per RFC  ↩
    2428
```

Examples of the cmd_validity option are shown below. These examples are the default checks (per RFC 959 and others) performed by the inspector.

```
cmd_validity =
{
    {
        command = 'CWD',
        length = 200,
    },
    {
        command = 'MODE',
```

```
        format = '< char SBC >',
    },
    {
        command = 'STRU',
        format = '< char FRP >',
    },
    {
        command = 'ALLO',
        format = '< int [ char R int ] >',
    },
    {
        command = 'TYPE',
        format = [[ < { char AE [ char NTC ] | char I | char L [ number ]
            } > ]],
    },
    {
        command = 'PORT',
        format = '< host_port >',
    },
}
```

A cmd_validity entry in the configuration can be used to override these defaults and/or add a check for other commands. A few examples follow.

This allows additional modes, including mode Z which allows for zip-style compression:

```
cmd_validity =
{
    {
        command = 'MODE',
        format = '< char ASBCZ >',
    },
}
```

Allow for a date in the MDTM command:

```
cmd_validity =
{
    {
        command = 'MDTM',
        format = '< [ date nnnnnnnnnnnnnn[.n[n[n]]] ] string >',
    },
}
```

MDTM is an odd case that is worth discussing...

While not part of an established standard, certain FTP servers accept MDTM commands that set the modification time on a file. The most common among servers that do, accept a format using YYYYMMDDHHmmss[.uuu]. Some others accept a format using YYYYMMDDHHmmss[+|-]TZ format. The example above is for the first case.

To check validity for a server that uses the TZ format, use the following:

```
cmd_validity =
{
    {
        command = 'MDTM',
        format = '< [ date nnnnnnnnnnnnnn[{+|-}n[n]] ] string >',
    },
}
```

- chk_str_fmt

This causes the inspector to check for string format attacks on the specified commands.

- telnet_cmds

Detect and alert when telnet cmds are seen on the FTP command channel.

- ignore_telnet_erase_cmds

This option allows Snort to ignore telnet escape sequences for erase character (TNC EAC) and erase line (TNC EAL) when normalizing FTP command channel. Some FTP servers do not process those telnet escape sequences.

- ignore_data_chan

When set to true, causes the FTP inspector to force the rest of snort to ignore the FTP data channel connections. NO INSPECTION other than state (inspector AND rules) will be performed on that data channel. It can be turned on to improve performance — especially with respect to large file transfers from a trusted source — by ignoring traffic. If your rule set includes virus-type rules, it is recommended that this option not be used.

ftp_client configuration

- max_resp_len

This specifies the maximum length for all response messages in bytes. If the message for an FTP response (everything after the 3 digit code) exceeds that length, and the inspector is configured to do so, an alert will be generated. This is used to check for buffer overflow exploits within FTP clients.

- telnet_cmds

Detect and alert when telnet cmds are seen on the FTP command channel.

- ignore_telnet_erase_cmds

This option allows Snort to ignore telnet escape sequences for erase character (TNC EAC) and erase line (TNC EAL) when normalizing FTP command channel. Some FTP clients do not process those telnet escape sequences.

ftp_data

In order to enable file inspection for ftp, the following should be added to the configuration:

```
ftp_data = {}
```

5.10 HTTP Inspector

One of the major undertakings for Snort 3 is developing a completely new HTTP inspector.

5.10.1 Overview

You can configure it by adding:

```
http_inspect = {}
```

to your snort.lua configuration file. Or you can read about it in the source code under src/service_inspectors/http_inspect.

So why a new HTTP inspector?

For starters it is object-oriented. That's good for us because we maintain this software. But it should also be really nice for open-source developers. You can make meaningful changes and additions to HTTP processing without having to understand the whole thing. In fact much of the new HTTP inspector's knowledge of HTTP is centralized in a series of tables where it can be easily reviewed and modified. Many significant changes can be made just by updating these tables.

http_inspect is the first inspector written specifically for the new Snort 3 architecture. This provides access to one of the very best features of Snort 3: purely PDU-based inspection. The classic preprocessor processes HTTP messages, but even while doing so it is constantly aware of IP packets and how they divide up the TCP data stream. The same HTTP message might be processed differently depending on how the sender (bad guy) divided it up into IP packets.

http_inspect is free of this burden and can focus exclusively on HTTP. This makes it much simpler, easier to test, and less prone to false positives. It also greatly reduces the opportunity for adversaries to probe the inspector for weak spots by adjusting packet boundaries to disguise bad behavior.

Dealing solely with HTTP messages also opens the door for developing major new features. The http_inspect design supports true stateful processing. Want to ask questions that involve both the client request and the server response? Or different requests in the same session? These things are possible.

http_inspect is taking a very different approach to HTTP header fields. The classic preprocessor divides all the HTTP headers following the start line into cookies and everything else. It normalizes the two pieces using a generic process and puts them in buffers that one can write rules against. There is some limited support for examining individual headers within the inspector but it is very specific.

The new concept is that every header should be normalized in an appropriate and specific way and individually made available for the user to write rules against it. If for example a header is supposed to be a date then normalization means put that date in a standard format.

5.10.2 Legacy and Enhanced Normalizers

Currently, there are Legacy and Enhanced Normalizers for JavaScript normalization. Both normalizers are independent and can be configured separately. The Legacy normalizer should be considered deprecated. The Enhanced Normalizer is encouraged to use for JavaScript normalization in the first place as we continue improving functionality and quality.

Legacy Normalizer

The Legacy Normalizer can normalize obfuscated data within the JavaScript functions such as unescape, String.fromCharCode, decodeURI, and decodeURIComponent. It also replaces consecutive whitespaces with a single space and normalizes the plus by concatenating the strings. For more information on how to enable Legacy Normalizer, check the http_inspect.normalize_javascript option. Legacy Normalizer is deprecated preferably to use Enhanced Normalizer. After supporting backward compatibility in the Enhanced Normalizer, Legacy Normalizer will be removed.

Enhanced Normalizer

Having ips option *js_data* in the rules automatically enables Enhanced Normalizer. The Enhanced Normalizer can normalize inline/external scripts. It supports scripts over multiple PDUs. It is a stateful JavaScript whitespace and identifiers normalizer. Normalizer concatenates string literals whenever it's possible to do. This also works with any other normalizations that result in string literals. All JavaScript identifier names, except those from the ignore lists, will be substituted with unified names in the following format: var_0000 $\rightarrow$ var_ffff. But the unescape-like function names will be removed from the normalized data. The Normalizer tries to expand an escaped text, so it will appear in a usual form in the output. Moreover, Normalizer validates

the syntax concerning ECMA-262 Standard, including scope tracking and restrictions for script elements. For more information on how additionally configure Enhanced Normalizer check with the following configuration options: js_norm_bytes_depth, js_norm_identifier_depth, js_norm_max_tmpl_nest, js_norm_max_bracket_depth, js_norm_max_scope_depth, js_norm_ident_ignore, js_norm_prop_ignore. Eventually Enhanced Normalizer will completely replace Legacy Normalizer.

5.10.3 Configuration

Configuration can be as simple as adding:

```
http_inspect = {}
```

to your snort.lua file. The default configuration provides a thorough inspection and may be all that you need. But there are some options that provide extra features, tweak how things are done, or conserve resources by doing less.

request_depth and response_depth

These replace the flow depth parameters used by the old HTTP inspector but they work differently.

The default is to inspect the entire HTTP message body. That's a very sound approach but if your HTTP traffic includes many very large files such as videos the load on Snort can become burdensome. Setting the request_depth and response_depth parameters will limit the amount of body data that is sent to the rule engine. For example:

```
request_depth = 10000,
response_depth = 80000,
```

would examine only the first 10000 bytes of POST, PUT, and other message bodies sent by the client. Responses from the server would be limited to 80000 bytes.

These limits apply only to the message bodies. HTTP headers are always completely inspected.

If you want to only inspect headers and no body, set the depth to 0. If you want to inspect the entire body set the depth to -1 or simply omit the depth parameter entirely because that is the default.

These limits have no effect on how much data is forwarded to file processing.

script_detection

Script detection is a feature that enables Snort to more quickly detect and block response messages containing malicious JavaScript. When http_inspect detects the end of a script it immediately forwards the available part of the message body for early detection. This enables malicious Javascripts to be detected more quickly but consumes somewhat more of the sensor's resources.

This feature is off by default. script_detection = true will activate it.

gzip

http_inspect by default decompresses deflate and gzip message bodies before inspecting them. This feature can be turned off by unzip = false. Turning off decompression provides a substantial performance improvement but at a very high price. It is unlikely that any meaningful inspection of message bodies will be possible. Effectively HTTP processing would be limited to the headers.

normalize_utf

http_inspect will decode utf-8, utf-7, utf-16le, utf-16be, utf-32le, and utf-32be in response message bodies based on the Content-Type header. This feature is on by default: normalize_utf = false will deactivate it.

decompress_pdf

decompress_pdf = true will enable decompression of compressed portions of PDF files encountered in a message body. http_inspect will examine the message body for PDF files that are then parsed to locate PDF streams with a single /FlateDecode filter. The compressed content is decompressed and made available through the file data rule option.

decompress_swf

decompress_swf = true will enable decompression of compressed SWF (Adobe Flash content) files encountered in a message body. The available decompression modes are 'deflate' and 'lzma'. http_inspect will search for the file signatures CWS for Deflate/ZLIB and ZWS for LZMA. The compressed content is decompressed and made available through the file data rule option. The compressed SWF file signature is converted to FWS to indicate an uncompressed file.

decompress_zip

decompress_zip = true will enable decompression of compressed zip archives encountered in a message body. The compressed content is decompressed and made available through the file_data rule option.

decompress_vba

decompress_vba = true will enable decompression of RLE (Run Length Encoding) compressed vba (Visual Basic for Applications) macro data of MS Office files encountered in a message body. The MS office files are PKZIP compressed which are parsed to locate the OLE (Object Linking and Embedding) file embedded with the files containing RLE compressed vba macro data. The decompressed vba macro data is then made available through the vba_data ips rule option.

normalize_javascript

normalize_javascript = true will enable legacy normalizer of JavaScript within the HTTP response body. http_inspect looks for JavaScript by searching for the <script> tag without a type. Obfuscated data within the JavaScript functions such as unescape, String.fromCharCode, decodeURI, and decodeURIComponent are normalized. The different encodings handled within the unescape, decodeURI, or decodeURIComponent are %XX, %uXXXX, XX and uXXXXi. http_inspect also replaces consecutive whitespaces with a single space and normalizes the plus by concatenating the strings. Such normalizations refer to basic JavaScript normalization.

js_norm_bytes_depth

js_norm_bytes_depth = N {-1 : max53} will set a number of input JavaScript bytes to normalize. When the depth is reached, normalization will be stopped. It's implemented per-script. By default js_norm_bytes_depth = -1, will set unlimited depth. The enhanced normalizer provides more precise whitespace normalization of JavaScript, that removes all redundant whitespaces and line terminators from the JavaScript syntax point of view (between identifier and punctuator, between identifier and operator, etc.) according to ECMAScript 5.1 standard. Additionally, it performs normalization of JavaScript identifiers making a substitution of unique names with unified names representation: var_0000:var_ffff. The identifiers are variables and function names. The normalized data is available through the js_data rule option.

js_norm_identifier_depth

js_norm_identifier_depth = N {0 : 65536} will set a number of unique JavaScript identifiers to normalize. When the depth is reached, a built-in alert is generated. Every HTTP response has its own identifier substitution context, which means that identifier will retain same normal form in multiple scripts, if they are a part of the same HTTP response, and that this limit is set for a single HTTP response and not a single script. By default, the value is set to 65536, which is the max allowed number of unique identifiers. The generated names are in the range from var_0000 to var_ffff.

js_norm_max_tmpl_nest

js_norm_max_tmpl_nest = N {0 : 255} (default 32) is an option of the enhanced JavaScript normalizer that determines the deepest level of nested template literals to be processed. Introduced in ES6, template literals provide syntax to define a literal multiline string, which can have arbitrary JavaScript substitutions, that will be evaluated and inserted into the string. Such substitutions can be nested, and require keeping track of every layer for proper normalization. This option is present to limit the amount of memory dedicated to template nesting tracking.

js_norm_max_bracket_depth

js_norm_max_bracket_depth = N {1 : 65535} (default 256) is an option of the enhanced JavaScript normalizer that determines the maximum depth of nesting brackets, i.e. parentheses, braces and square brackets, nested within a matching pair, in any combination. This option is present to limit the amount of memory dedicated to bracket tracking.

js_norm_max_scope_depth

js_norm_max_scope_depth = N {1 : 65535} (default 256) is an option of the enhanced JavaScript normalizer that determines the deepest level of nested variable scope, i.e. functions, code blocks, etc. including the global scope. This option is present to limit the amount of memory dedicated to scope tracking.

js_norm_ident_ignore

js_norm_ident_ignore = {<list of ignored identifiers>} is an option of the enhanced JavaScript normalizer that defines a list of identifiers to keep intact.

Identifiers in this list will not be put into normal form (var_0000). Subsequent accessors, after dot, in square brackets or after function call, will not be normalized as well.

For example:

```
console.log("bar")
document.getElementById("id").text
eval("script")
console["log"]
```

Every entry has to be a simple identifier, i.e. not include dots, brackets, etc. For example:

```
http_inspect.js_norm_ident_ignore = { 'console', 'document', 'eval', 'foo' }
```

When a variable assignment that *aliases* an identifier from the list is found, the assignment will be tracked, and subsequent occurrences of the variable will be replaced with the stored value. This substitution will follow JavaScript variable scope limits.

For example:

```
var a = console.log
a("hello") // will be substituted to 'console.log("hello")'
```

The default list of ignore-identifiers is present in "snort_defaults.lua".

Unescape function names should remain intact in the output. They ought to be included in the ignore list. If for some reason the user wants to disable unescape related features, then removing function's name from the ignore list does the trick.

js_norm_prop_ignore

js_norm_prop_ignore = {<list of ignored properties>} is an option of the enhanced JavaScript normalizer that defines a list of object properties and methods that will be kept intact during the identifiers normalization. This list should include methods and properties of objects that will not be tracked by assignment substitution functionality, for example, those that can be created implicitly.

Subsequent accessors, after dot, in square brackets or after function call, will not be normalized as well.

For example:

```
http_inspect.js_norm_prop_ignore = { 'split' }

in: "string".toUpperCase().split("").reverse().join("");
out: "string".var_0000().split("").reverse().join("");
```

The default list of ignored properties is present in "snort_defaults.lua".

xff_headers

This configuration supports defining custom x-forwarded-for type headers. In a multi-vendor world, it is quite possible that the header name carrying the original client IP could be vendor-specific. This is due to the absence of standardization which would otherwise standardize the header name. In such a scenario, this configuration provides a way with which such headers can be introduced to HI. The default value of this configuration is "x-forwarded-for true-client-ip". The default definition introduces the two commonly known headers and is preferred in the same order by the inspector as they are defined, e.g "x-forwarded-for" will be preferred than "true-client-ip" if both headers are present in the stream. The header names should be delimited by a space.

maximum_host_length

Setting maximum_host_length causes http_inspect to generate 119:25 if the Host header value including optional white space exceeds the specified length. In the abnormal case of multiple Host headers, the total length of the combined values is used. The default value is -1, meaning do not perform this check.

maximum_chunk_length

http_inspect strictly limits individual chunks within a chunked message body to be less than four gigabytes.

A lower limit may be configured by setting maximum_chunk_length. Any chunk longer than maximum chunk length will generate a 119:16 alert.

URI processing

Normalization and inspection of the URI in the HTTP request message is a key aspect of what http_inspect does. The best way to normalize a URI is very dependent on the idiosyncrasies of the HTTP server being accessed. The goal is to interpret the URI the same way as the server will so that nothing the server will see can be hidden from the rule engine.

The default URI inspection parameters are oriented toward following the HTTP RFCs—reading the URI the way the standards say it should be read. Most servers deviate from this ideal in various ways that can be exploited by an attacker. The options provide tools for the user to cope with that.

```
utf8 = true
plus_to_space = true
percent_u = false
utf8_bare_byte = false
iis_unicode = false
iis_double_decode = true
```

The HTTP inspector normalizes percent encodings found in URIs. For instance it will convert "%48%69%64%64%65%6e" to "Hidden". All the options listed above control how this is done. The options listed as true are fairly standard features that are decoded by default. You don't need to list them in snort.lua unless you want to turn them off by setting them to false. But that is not recommended unless you know what you are doing and have a definite reason.

The other options are primarily for the protection of servers that support irregular forms of decoding. These features are off by default but you can activate them if you need to by setting them to true in snort.lua.

```
bad_characters = "0x25 0x7e 0x6b 0x80 0x81 0x82 0x83 0x84"
```

That's a list of 8-bit Ascii characters that you don't want present in any normalized URI after the percent decoding is done. For example 0x25 is a hexadecimal number (37 in decimal) which stands for the % character. The % character is legitimately used for encoding special characters in a URI. But if there is still a percent after normalization one might conclude that something is wrong. If you choose to configure 0x25 as a bad character there will be an alert whenever this happens.

Another example is 0x00 which signifies the null character zero. Null characters in a URI are generally wrong and very suspicious.

The default is not to alert on any of the 256 8-bit Ascii characters. Add this option to your configuration if you want to define some bad characters.

```
ignore_unreserved = "abc123"
```

Percent encoding common characters such as letters and numbers that have no special meaning in HTTP is suspicious. It's legal but why would you do it unless you have something to hide? http_inspect will alert whenever an upper-case or lower-case letter, a digit, period, underscore, tilde, or minus is percent-encoded. But if a legitimate application in your environment encodes some of these characters for some reason this allows you to create exemptions for those characters.

In the example, the lower-case letters a, b, and c and the digits 1, 2, and 3 are exempted. These may be percent-encoded without generating an alert.

```
simplify_path = true
backslash_to_slash = true
```

HTTP inspector simplifies directory paths in URIs by eliminating extra traversals using ., .., and /.

For example I can take a simple URI such as

```
/very/easy/example
```

and complicate it like this:

```
/very/../very/./././/easy//////detour/to/nowhere/../.././../example
```

which may be very difficult to match with a detection rule. simplify_path is on by default and you should not turn it off unless you have no interest in URI paths.

backslash_to_slash is a tweak to path simplification for servers that allow directories to be separated by backslashes:

```
/this/is/the/normal/way/to/write/a/path

\this\is\the\other\way\to\write\a\path
```

backslash_to_slash is turned on by default. It replaces all the backslashes with slashes during normalization.

5.10.4 CONNECT processing

The HTTP CONNECT method is used by a client to establish a tunnel to a destination via an HTTP proxy server. If the connection is successful the server will send a 2XX success response to the client, then proceed to blindly forward traffic between the client and destination. That traffic belongs to a new session between the client and destination and may be of any protocol, so clearly the HTTP inspector will be unable to continue processing traffic following the CONNECT message as if it were just a continuation of the original HTTP/1.1 session.

Therefore upon receiving a success response to a CONNECT request, the HTTP inspector will stop inspecting the session. The next packet will return to the wizard, which will determine the appropriate inspector to continue processing the flow. If the tunneled protocol happens to be HTTP/1.1, the HTTP inspector will again start inspecting the flow, but as an entirely new session.

There is one scenario where the cutover to the wizard will not occur despite a 2XX success response to a CONNECT request. HTTP allows for pipelining, or sending multiple requests without waiting for a response. If the HTTP inspector sees any further traffic from the client after a CONNECT request before it has seen the CONNECT response, it is unclear whether this traffic should be interpreted as a pipelined HTTP request or tunnel traffic sent in anticipation of a success response from the server. Due to this potential evasion tactic, the HTTP inspector will not cut over to the wizard if it sees any early client-to-server traffic, but will continue normal HTTP processing of the flow regardless of the eventual server response.

5.10.5 Trace messages

When a user needs help to sort out things going on inside HTTP inspector, Trace module becomes handy.

```
$ snort --help-module trace | grep http_inspect
```

Messages for the enhanced JavaScript Normalizer follow (more verbosity available in debug build):

trace.module.http_inspect.js_proc

Messages from script processing flow and their verbosity levels:

1. Script opening tag location.
2. Attributes of the detected script.
3. Return codes from Normalizer.

trace.module.http_inspect.js_dump

JavaScript data dump and verbosity levels:

1. js_data buffer as it is passed to detection.
2. (no messages available currently)
3. Current script as it is passed to Normalizer.

5.10.6 Detection rules

http_inspect parses HTTP messages into their components and makes them available to the detection engine through rule options. Let's start with an example:

```
alert tcp any any -> any any ( msg:"URI example"; flow:established,
to_server; http_uri; content:"chocolate"; sid:1; rev:1; )
```

This rule looks for chocolate in the URI portion of the request message. Specifically, the http_uri rule option is the normalized URI with all the percent encodings removed. It will find chocolate in both:

```
GET /chocolate/cake HTTP/1.1
```

and

```
GET /%63%68$6F%63%6F%6C%61%74%65/%63%61%6B%65 HTTP/1.1
```

It is also possible to search the unnormalized URI

```
alert tcp any any -> any any ( msg:"Raw URI example"; flow:established,
to_server; http_raw_uri; content:"chocolate"; sid:2; rev:1; )
```

will match the first message but not the second. If you want to detect someone who is trying to hide his request for chocolate then

```
alert tcp any any -> any any ( msg:"Raw URI example"; flow:established,
to_server; http_raw_uri; content:"%63%68$6F%63%6F%6C%61%74%65";
sid:3; rev:1; )
```

will do the trick.

Let's look at possible ways of writing a rule to match HTTP response messages with the Content-Language header set to "da" (Danish). You could write:

```
alert tcp any any -> any any ( msg:"whole header search";
flow:established, to_client; http_header; content:
"Content-Language: da", nocase; sid:4; rev:1; )
```

This rule leaves much to be desired. Modern headers are often thousands of bytes and seem to get longer every year. Searching all of the headers consumes a lot of resources. Furthermore this rule is easily evaded:

```
HTTP/1.1 ... Content-Language:  da ...
```

the extra space before the "da" throws the rule off. Or how about:

```
HTTP/1.1 ... Content-Language: xx,da ...
```

By adding a made up second language the attacker has once again thwarted the match.

A better way to write this rule is:

```
alert tcp any any -> any any ( msg:"individual header search";
flow:established, to_client; http_header: field content-language;
content:"da", nocase; sid:4; rev:2; )
```

The field option improves performance by narrowing the search to the Content-Language field of the header. Because it uses the header parsing abilities of http_inspect to find the field of interest it will not be thrown off by extra spaces or other languages in the list.

In addition to the headers there are rule options for virtually every part of the HTTP message.

http_uri and http_raw_uri

These provide the URI of the request message. The raw form is exactly as it appeared in the message and the normalized form is determined by the URI normalization options you selected. In addition to searching the entire URI there are six components that can be searched individually:

```
alert tcp any any -> any any ( msg:"URI path"; flow:established,
to_server; http_uri: path; content:"chocolate"; sid:1; rev:2; )
```

By specifying "path" the search is limited to the path portion of the URI. Informally this is the part consisting of the directory path and file name. Thus it will match:

```
GET /chocolate/cake HTTP/1.1
```

but not:

```
GET /book/recipes?chocolate+cake HTTP/1.1
```

The question mark ends the path and begins the query portion of the URI. Informally the query is where parameter values are set and often contains a search to be performed.

The six components are:

1. path: directory and file
2. query: user parameters
3. fragment: part of the file requested, normally found only inside a browser and not transmitted over the network

4. host: domain name of the server being addressed
5. port: TCP port number being addressed
6. scheme: normally "http" or "https" but others are possible such as "ftp"

Here is an example with all six:

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

The URI is everything between the first space and the last space. "https" is the scheme, "www.samplehost.com" is the host, "287" is the port, "/basic/example/of/path" is the path, "with-query" is the query, and "and-fragment" is the fragment.

http_uri represents the normalized uri, normalization of components depends on uri type. If the uri is of type absolute (contains all six components) or absolute path (contains path, query and fragment) then the path and query components are normalized. In these cases, http_uri represents the normalized path, query, and fragment (/path?query#fragment). If the uri is of type authority (host and port), the host is normalized and http_uri represents the normalized host with the port number. In all other cases http_uri is the same as http_raw_uri.

Note: this section uses informal language to explain some things. Nothing here is intended to conflict with the technical language of the HTTP RFCs and the implementation follows the RFCs.

http_header and http_raw_header

These cover all the header lines except the first one. You may specify an individual header by name using the field option as shown in this earlier example:

```
alert tcp any any -> any any ( msg:"individual header search";
flow:established, to_client; http_header: field content-language;
content:"da", nocase; sid:4; rev:2; )
```

This rule searches the value of the Content-Language header. Header names are not case sensitive and may be written in the rule in any mixture of upper and lower case.

With http_header the individual header value is normalized in a way that is appropriate for that header.

If you don't specify a header you get all of the headers. http_raw_header includes the unmodified header names and values as they appeared in the original message. http_header is the same except percent encodings and cookies are removed and paths are simplified exactly as if the headers were a URI.

In most cases specifying individual headers creates a more efficient and accurate rule. It is recommended that new rules be written using individual headers whenever possible.

http_trailer and http_raw_trailer

HTTP permits header lines to appear after a chunked body ends. Typically they contain information about the message content that was not available when the headers were created. For convenience we call them trailers.

http_trailer and http_raw_trailer are identical to their header counterparts except they apply to these end headers. If you want a rule to inspect both kinds of headers you need to write two rules, one using header and one using trailer.

http_cookie and http_raw_cookie

These provide the value of the Cookie header for a request message and the Set-Cookie for a response message. If multiple cookies are present they will be concatenated into a comma-separated list.

Normalization for http_cookie is the same URI-style normalization applied to http_header when no specific header is specified.

http_true_ip

This provides the original IP address of the client sending the request as it was stored by a proxy in the request message headers. Specifically it is the last IP address listed in the X-Forwarded-For, True-Client-IP or any other custom x-forwarded-for type header. If multiple headers are present the preference defined in xff_headers configuration is considered.

http_client_body

This is the body of a request message such as POST or PUT. Normalization for http_client_body is the same URI-like normalization applied to http_header when no specific header is specified.

http_raw_body

This is the body of a request or response message. It will be dechunked and unzipped if applicable but will not be normalized in any other way.

http_method

The method field of a request message. Common values are "GET", "POST", "OPTIONS", "HEAD", "DELETE", "PUT", "TRACE", and "CONNECT".

http_stat_code

The status code field of a response message. This is normally a 3-digit number between 100 and 599. In this example it is 200.

```
HTTP/1.1 200 OK
```

http_stat_msg

The reason phrase field of a response message. This is the human-readable text following the status code. "OK" in the previous example.

http_version

The protocol version information that appears on the first line of an HTTP message. This is usually "HTTP/1.0" or "HTTP/1.1".

http_raw_request and http_raw_status

These are the unmodified first header line of the HTTP request and response messages respectively. These rule options are a safety valve in case you need to do something you cannot otherwise do. In most cases it is better to use a rule option for a specific part of the first header line. For a request message those are http_method, http_raw_uri, and http_version. For a response message those are http_version, http_stat_code, and http_stat_msg.

file_data

The file_data contains the normalized message body. This is the normalization described above under gzip, normalize_utf, decompress_pdf, decompress_swf, and normalize_javascript.

js_data

The js_data contains normalized JavaScript text collected from the whole PDU (inline or external scripts). It requires the Enhanced Normalizer enabled: http_inspect = { js_norm_bytes_depth = N }, js_norm_bytes_depth option is described above. Despite what js_data has, file_data still contains the whole HTTP body with an original JavaScript in it.

vba_data

The vba_data will contain the decompressed Visual Basic for Applications (vba) macro data embedded in MS office files. It requires decompress_zip and decompress_vba options enabled.

http_num_headers and http_num_trailers

These rule options are used to check the number of headers and trailers, respectively. Checks available: equal to "=" or just value, not "!" or "!=", less than "<", greater than ">", less or equal to "⇐", less or greater than ">=", in range "<>", in range or equal to "<⇒".

http_version_match

Rule option that matches HTTP version to one of the listed version values. Possible match values: 1.0, 1.1, 2.0, 0.9, other, and malformed. When receiving a request line or status line, if the version is present it will be used for comparison. If the version doesn't have a format of [0-9].[0-9] it is considered malformed. A [0-9].[0-9] that is not 1.0 or 1.1 is considered other. 0.9 refers to the original HTTP protocol version that uses simple GET requests without headers and includes no version number. 2.0 refers to the actual HTTP/2 protocol with framed data. Messages that follow the general HTTP/1 format but contain version fields falsely claiming to be HTTP/2.0 or HTTP/0.9 will match "other" as described above. The http_version rule option is available to examine the actual bytes in the version field.

http_header_test and http_trailer_test

Rule options that perform various tests against a specific header and trailer field, respectively. It can perform a range test, check whether the value is numeric or whether it is absent. Negative values are considered non-numeric. Values with more than 18 digits are considered non-numeric.

5.10.7 Timing issues and combining rule options

HTTP inspector is stateful. That means it is aware of a bigger picture than the packet in front of it. It knows what all the pieces of a message are, the dividing lines between one message and the next, which request message triggered which response message, pipelines, and how many messages have been sent over the current connection.

Some rules use a single rule option:

```
alert tcp any any -> any any ( msg:"URI example"; flow:established,
to_server; http_uri; content:"chocolate"; sid:1; rev:1; )
```

Whenever a new URI is available this rule will be evaluated. Nothing complicated about that, but suppose we use more than one rule option:

```
alert tcp any any -> any any ( msg:"combined example"; flow:established,
to_server; http_uri: with_body; content:"chocolate"; file_data;
content:"sinister POST data"; sid:5; rev:1; )
```

The with_body option to http_uri causes the URI to be made available with the message body. Use with_body for header-related rule options in rules that also examine the message body.

The with_trailer option is analogous and causes an earlier message element to be made available at the end of the message when the trailers following a chunked body arrive.

```
alert tcp any any -> any any ( msg:"double content-language";
flow:established, to_client; http_header: with_trailer, field
content-language; content:"da", nocase; http_trailer: field
content-language; content:"en", nocase; sid:6; rev:1; )
```

This rule will alert if the Content-Language changes from Danish in the headers to English in the trailers. The with_trailer option is essential to make this rule work.

It is also possible to write rules that examine both the client request and the server response to it.

```
alert tcp any any -> any any ( msg:"request and response example";
flow:established, to_client; http_uri: with_body; content:"chocolate";
file_data; content:"white chocolate"; sid:7; rev:1; )
```

This rule looks for white chocolate in a response message body where the URI of the request contained chocolate. Note that this is a "to_client" rule that will alert on and potentially block a server response containing white chocolate, but only if the client URI requested chocolate. If the rule were rewritten "to_server" it would be nonsense and not work. Snort cannot block a client request based on what the server response will be because that has not happened yet.

Another point is "with_body" for http_uri. This ensures the rule works on the entire response body. If we were looking for white chocolate in the response headers this would not be necessary.

Response messages do not have a URI so there was only one thing http_uri could have meant in the previous rule. It had to be referring to the request message. Sometimes that is not so clear.

```
alert tcp any any -> any any ( msg:"header ambiguity example 1";
flow:established, to_client; http_header: with_body; content:
"chocolate"; file_data; content:"white chocolate"; sid:8; rev:1; )


alert tcp any any -> any any ( msg:"header ambiguity example 2";
flow:established, to_client; http_header: with_body, request; content:
"chocolate"; file_data; content:"white chocolate"; sid:8; rev:2; )
```

Our search for chocolate has moved from the URI to the message headers. Both the request and response messages have headers—which one are we asking about? Ambiguity is always resolved in favor of looking in the current message which is the response. The first rule is looking for a server response containing chocolate in the headers and white chocolate in the body.

The second rule uses the "request" option to explicitly say that the http_header to be searched is the request header.

Let's put all of this together. There are six opportunities to do detection:

1. When the the request headers arrive. The request line and all of the headers go through detection at the same time.
2. When sections of the request message body arrive. If you want to combine this with something from the request line or headers you must use the with_body option.
3. When the request trailers arrive. If you want to combine this with something from the request line or headers you must use the with_trailer option.
4. When the response headers arrive. The status line and all of the headers go through detection at the same time. These may be combined with elements from the request line, request headers, or request trailers. Where ambiguity arises use the request option.
5. When sections of the response message body arrive. These may be combined with the status line, response headers, request line, request headers, or request trailers as described above.
6. When the response trailers arrive. Again these may be combined as described above.

Message body sections can only go through detection at the time they are received. Headers may be combined with later items but the body cannot.

5.11 HTTP/2 Inspector

New in Snort 3, the HTTP/2 inspector enables Snort to process HTTP/2 traffic.

5.11.1 Overview

Despite the name, it is better to think of HTTP/2 not as a newer version of HTTP/1.1, but rather a separate protocol layer that runs under HTTP/1.1 and on top of TLS or TCP. It supports several new features with the goal of improving the performance of HTTP requests, notably the ability to multiplex many requests over a single TCP connection, HTTP header compression, and server push.

HTTP/2 is a perfect fit for the new Snort 3 PDU-based inspection architecture. The HTTP/2 inspector parses and strips the HTTP/2 protocol framing and outputs HTTP/1.1 messages, exactly what http_inspect wants to input. The HTTP/2 traffic then undergoes the same processing as regular HTTP/1.1 traffic discussed above. So if you haven't already, take a look at the HTTP Inspector section; those features also apply to HTTP/2 traffic.

5.11.2 Configuration

You can configure the HTTP/2 inspector with the default configuration by adding:

```
http2_inspect = {}
```

to your snort.lua configuration file. Since processing HTTP/2 traffic relies on the HTTP inspector, http_inspect must also be configured. Keep in mind that the http_inspect configuration will also impact HTTP/2 traffic.

concurrent_streams_limit

This limits the maximum number of HTTP/2 streams Snort will process concurrently in a single HTTP/2 flow. The default and minimum configurable value is 100. It can be configured up to a maximum of 1000.

5.11.3 Detection rules

Since HTTP/2 traffic is processed through the HTTP inspector, all of the rule options discussed above are also available for HTTP/2 traffic. To smooth the transition to inspecting HTTP/2, rules that specify service:http will be treated as if they also specify service:http2. Thus:

```
alert tcp any any -> any any (flow:established, to_server;
http_uri; content:"/foo";
service: http; sid:10; rev:1;)
```

is understood to mean:

```
alert tcp any any -> any any (flow:established, to_server;
http_uri; content:"/foo";
service: http,http2; sid:10; rev:1;)
```

Thus it will alert on "/foo" in the URI for both HTTP/1 and HTTP/2 traffic.

The reverse is not true. "service: http2" without http will match on HTTP/2 flows but not HTTP/1 flows.

This feature makes it easy to add HTTP/2 inspection without modifying large numbers of existing rules. New rules should explicitly specify "service http,http2;" if that is the desired behavior. Eventually support for http implies http2 may be deprecated and removed.

5.12 IEC104 Inspector

iec104 inspector is a service inspector for the IEC 60870-5-104 protocol.

5.12.1 Overview

IEC 60870-5-104 (iec104) is a protocol distributed by the International Electrotechnical Commission (IEC) that provides a standardized method of sending telecontrol messages between central stations and outstations, typically running on TCP port 2404.

It is used in combination with the companion specifications in the IEC 60870-5 family, most notably IEC 60870-5-101, to provide reliable transport via TCP/IP.

An iec104 Application Protocol Data Unit (APDU) consists of one of three Application Protocol Control Information (APCI) structures, each beginning with the start byte 0x68. In the case of an Information Transfer APCI, an Application Service Data Unit (ASDU) follows the APCI.

The iec104 inspector decodes the iec104 protocol and provides rule options to access certain protocol fields and data content. This allows the user to write rules for iec104 packets without decoding the protocol.

5.12.2 Configuration

iec104 messages can be normalized to either combine a message spread across multiple frames, or to split apart multiple messages within one frame. No manual configuration is necessary to leverage this functionality.

5.12.3 Quick Guide

A typical iec104 configuration looks like this:

```
binder =
{
    {
        when =
        {
            proto = 'tcp',
            ports = '2404'
        },
        use =
        {
            type = 'iec104'
        },
    },
}

iec104 = { }
```

In this example, the tcp inspector is defined based on port. All configurations are default.

Debug logging can be enabled with the following additional configuration:

```
trace =
{
    modules =
    {
        iec104 =
        {
            all = 1
        }
    }
}
```

5.12.4 Rule Options

New rule options are supported by enabling the iec104 inspector:

- iec104_apci_type
- iec104_asdu_func

iec104_apci_type

Determining the APCI type of an iec104 message involves checking the state of one to two bits in the message's first control field octet. This can be completed with a `byte_test` in a plaintext rule, however it adds unnecessary complexity to the rule. Since most rules inspecting iec104 traffic will target APCI Type I messages, this option was created to alleviate the need to manually check the type and subsequently reduce the complexity of the rule.

This option takes one argument with three acceptable configurations.

Examples:

```
iec104_apci_type:unnumbered_control_function;
iec104_apci_type:S;
iec104_apci_type:i;
```

This 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.

iec104_asdu_func

Determining the ASDU function of an iec104 message can be completed with a plaintext rule that checks a single byte in the message, however it also requires verifying that the message's APCI is of Type I. Since a rule writer may not necessarily know that this additional check must be made, this option was created to simplify the process of verifying the function type and subsequently reduce the complexity of the rule.

This option takes one argument with two acceptable configurations.

Examples:

```
iec104_asdu_func:M_SP_NA_1;
iec104_asdu_func:m_ps_na_1;
```

This 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.

5.13 MMS Inspector

MMS inspector is a service inspector for the MMS protocol within the IEC 61850 specification.

5.13.1 Overview

IEC 61850 is a family of protocols, including MMS, distributed by the International Electrotechnical Commission (IEC) that provide a standardized method of sending service messages between various manufacturing and process control devices, typically running on TCP port 102.

It is used in combination with various parts of the OSI model, most notably the TPKT, COTP, Session, Presentation, and ACSE layers, to provide reliable transport via TCP/IP.

The MMS inspector decodes the OSI layers encapsulating the MMS protocol and provides rule writers access to certain protocol fields and data content through rule options. This allows the user to write rules for MMS messages without decoding the protocol.

5.13.2 Configuration

MMS messages can be sent in a variety of ways including multiple PDUs within one TCP packet, one PDU split across multiple TCP packets, or a combination of the two. It is the aim of the MMS service inspector to normalize the traffic such that only complete MMS messages are presented to the user. No manual configuration other than enabling the MMS service inspector is necessary to leverage this functionality.

5.13.3 Quick Guide

A typical MMS configuration looks like this:

```
wizard = { curses = {'mms'}, }
mms = { }

binder =
{
    { when = { service = 'mms' }, use = { type = 'mms' } },
    { use = { type = 'wizard' } }
}
```

In this example, the mms inspector is defined based on patterns known to be consistent with MMS messages.

5.13.4 Rule Options

New rule options are supported by enabling the MMS inspector:

- mms_data
- mms_func

mms_data

mms_data moves the cursor to the start of the MMS message, bypassing all of the OSI encapsulation layers and allowing subsequent rule options to start processing from the MMS PDU field.

This option takes no arguments.

In the following 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.

```
alert tcp ( \
  msg: "PROTOCOL-SCADA MMS Initiate-Request"; \
  flow: to_server, established; \
  mms_data; \
  content:"|A8|", depth 1; \
  sid:1000000; \
)
```

mms_func

mms_func takes the supplied function name or number and compares it with the Confirmed Service Request/Response in the message being analyzed.

This option takes one argument.

In the following 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. This is combined with a content match for a `Confirmed Service Request` message (0xA0) to allow for use of the fast pattern matcher.

```
alert tcp ( \
  msg: "PROTOCOL-SCADA MMS svc get_name_list"; \
  flow: to_server, established; \
  content:"|A0|"; \
  mms_func: get_name_list; \
  sid:1000000; \
)
```

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.

```
alert tcp ( \
  msg: "PROTOCOL-SCADA MMS svc get_name_list"; \
  flow: to_server, established; \
  content:"|A0|"; \
  mms_func:1; \
  sid:1000001; \
)
```

5.14 Performance Monitor

The new and improved performance monitor! Is your sensor being bogged down by too many flows? perf_monitor! Why are certain TCP segments being dropped without hitting a rule? perf_monitor! Why is a sensor leaking water? Not perf_monitor, check with stream...

5.14.1 Overview

The Snort performance monitor is the built-in utility for monitoring system and traffic statistics. All statistics are separated by processing thread. perf_monitor supports several trackers for monitoring such data:

5.14.2 Base Tracker

The base tracker is used to gather running statistics about Snort and its running modules. All Snort modules gather, at the very least, counters for the number of packets reaching it. Most supplement these counts with those for domain specific functions, such as http_inspect's number of GET requests seen.

Statistics are gathered live and can be reported at regular intervals. The stats reported correspond only to the interval in question and are reset at the beginning of each interval.

These are the same counts displayed when Snort shuts down, only sorted amongst the discrete intervals in which they occurred.

Base differs from prior implementations in Snort in that all stats gathered are only raw counts, allowing the data to be evaluated as needed. Additionally, base is entirely pluggable. Data from new Snort plugins can be added to the existing stats either automatically or, if specified, by name and function.

All plugins and counters can be enabled or disabled individually, allowing for only the data that is actually desired instead of overly verbose performance logs.

To enable everything:

```
perf_monitor = { modules = {} }
```

To enable everything within a module:

```
perf_monitor =
{
    modules =
    {
        {
            name = 'stream_tcp',
            pegs = [[ ]]
        },
    }
}
```

To enable specific counts within modules:

```
perf_monitor =
{
    modules =
    {
        {
            name = 'stream_tcp',
            pegs = [[ overlaps gaps ]]
        },
    }
```

Note: Event stats from prior Snorts are now located within base statistics.

5.14.3 Flow Tracker

Flow tracks statistics regarding traffic and L3/L4 protocol distributions. This data can be used to build a profile of traffic for inspector tuning and for identifying where Snort may be stressed.

To enable:

```
perf_monitor = { flow = true }
```

5.14.4 FlowIP Tracker

FlowIP provides statistics for individual hosts within a network. This data can be used for identifying communication habits, such as generating large or small amounts of data, opening a small or large number of sessions, and tendency to send smaller or larger IP packets.

To enable:

```
perf_monitor = { flow_ip = true }
```

5.14.5 CPU Tracker

This tracker monitors the CPU and wall time spent by a given processing thread.

To enable:

```
perf_monitor = { cpu = true }
```

5.14.6 Formatters

Performance monitor allows statistics to be output in a few formats. Along with human readable text (as seen at shutdown) and csv formats, a JSON format format is also available.

5.15 POP and IMAP

POP inspector is a service inspector for POP3 protocol and IMAP inspector is for IMAP4 protocol.

5.15.1 Overview

POP and IMAP inspectors examine data traffic and find POP and IMAP commands and responses. The inspectors also identify the command, header, body sections and extract the MIME attachments and decode it appropriately. The pop and imap also identify and whitelist the pop and imap traffic.

5.15.2 Configuration

POP inspector and IMAP inspector offer same set of configuration options for MIME decoding depth. These depths range from 0 to 65535 bytes. Setting the value to 0 ("do none") turns the feature off. Alternatively the value -1 means an unlimited amount of data should be decoded. If you do not specify the default value is -1 (unlimited).

The depth limits apply per attachment. They are:

b64_decode_depth

Set the base64 decoding depth used to decode the base64-encoded MIME attachments.

qp_decode_depth

Set the Quoted-Printable (QP) decoding depth used to decode QP-encoded MIME attachments.

bitenc_decode_depth

Set the non-encoded MIME extraction depth used for non-encoded MIME attachments.

uu_decode_depth

Set the Unix-to-Unix (UU) decoding depth used to decode UU-encoded attachments.

Examples

```
stream = { }

stream_tcp = { }

stream_ip = { }

binder =
{
    {
        {
            when = { proto = 'tcp', ports = '110', },
            use = { type = 'pop', },
        },
        {
            when = { proto = 'tcp', ports = '143', },
            use =  { type = 'imap', },
        },
    },
}
```

```
imap =
{
    qp_decode_depth = 500,
}

pop =
{
    qp_decode_depth = -1,
    b64_decode_depth = 3000,
}
```

5.16 Port Scan

A module to detect port scanning

5.16.1 Overview

This module is designed to detect the first phase in a network attack: Reconnaissance. In the Reconnaissance phase, an attacker determines what types of network protocols or services a host supports. This is the traditional place where a portscan takes place. This phase assumes the attacking host has no prior knowledge of what protocols or services are supported by the target, otherwise this phase would not be necessary.

As the attacker has no beforehand knowledge of its intended target, most queries sent by the attacker will be negative (meaning that the services are closed). In the nature of legitimate network communications, negative responses from hosts are rare, and rarer still are multiple negative responses within a given amount of time. Our primary objective in detecting portscans is to detect and track these negative responses.

One of the most common portscanning tools in use today is Nmap. Nmap encompasses many, if not all, of the current portscanning techniques. Portscan was designed to be able to detect the different types of scans Nmap can produce.

The following are a list of the types of Nmap scans Portscan will currently alert for.

- TCP Portscan
- UDP Portscan
- IP Portscan

These alerts are for one to one portscans, which are the traditional types of scans; one host scans multiple ports on another host. Most of the port queries will be negative, since most hosts have relatively few services available.

- TCP Decoy Portscan
- UDP Decoy Portscan
- IP Decoy Portscan

Decoy portscans are much like regular, only the attacker has spoofed source address inter-mixed with the real scanning address. This tactic helps hide the true identity of the attacker.

- TCP Distributed Portscan
- UDP Distributed Portscan
- IP Distributed Portscan

These are many to one portscans. Distributed portscans occur when multiple hosts query one host for open services. This is used to evade an IDS and obfuscate command and control hosts.

Note
Negative queries will be distributed among scanning hosts, so we track this type of scan through the scanned host.

- TCP Portsweep
- UDP Portsweep
- IP Portsweep
- ICMP Portsweep

These alerts are for one to many portsweeps. One host scans a single port on multiple hosts. This usually occurs when a new exploit comes out and the attacker is looking for a specific service.

Note
The characteristics of a portsweep scan may not result in many negative responses. For example, if an attacker portsweeps a web farm for port 80, we will most likely not see many negative responses.

- TCP Filtered Portscan
- UDP Filtered Portscan
- IP Filtered Portscan
- TCP Filtered Decoy Portscan
- UDP Filtered Decoy Portscan
- IP Filtered Decoy Portscan
- TCP Filtered Portsweep
- UDP Filtered Portsweep
- IP Filtered Portsweep
- ICMP Filtered Portsweep
- TCP Filtered Distributed Portscan
- UDP Filtered Distributed Portscan
- IP Filtered Distributed Portscan

"Filtered" alerts indicate that there were no network errors (ICMP unreachables or TCP RSTs) or responses on closed ports have been suppressed. It's also a good indicator on whether the alert is just a very active legitimate host. Active hosts, such as NATs, can trigger these alerts because they can send out many connection attempts within a very small amount of time. A filtered alert may go off before responses from the remote hosts are received.

Portscan only generates one alert for each host pair in question during the time window. On TCP scan alerts, Portscan will also display any open ports that were scanned. On TCP sweep alerts however, Portscan will only track open ports after the alert has been triggered. Open port events are not individual alerts, but tags based off the original scan alert.

5.16.2 Scan levels

There are 3 default scan levels that can be set.

```
1) default_hi_port_scan
2) default_med_port_scan
3) default_low_port_scan
```

Each of these default levels have separate options that can be edited to alter the scan sensitivity levels (scans, rejects, nets or ports)

Example:

```
port_scan = default_low_port_scan

port_scan.tcp_decoy.ports = 1
port_scan.tcp_decoy.scans = 1
port_scan.tcp_decoy.rejects = 1
port_scan.tcp_ports.nets = 1
```

The example above would change each of the individual settings to 1.

NOTE:The default levels for scans, rejects, nets and ports can be seen in the snort_defaults.lua file.

The counts can be seen in the alert outputs (-Acmg shown below):

```
50 72 69 6F 72 69 74 79  20 43 6F 75 6E 74 3A 20  Priority  Count:
30 0A 43 6F 6E 6E 65 63  74 69 6F 6E 20 43 6F 75  0.Connec tion Cou
6E 74 3A 20 34 35 0A 49  50 20 43 6F 75 6E 74 3A  nt: 45.I P Count:
20 31 0A 53 63 61 6E 6E  65 72 20 49 50 20 52 61  1.Scann er IP Ra
6E 67 65 3A 20 31 2E 32  2E 33 2E 34 3A 31 2E 32  nge: 1.2 .3.4:1.2
2E 33 2E 34 0A 50 6F 72  74 2F 50 72 6F 74 6F 20  .3.4.Por t/Proto
43 6F 75 6E 74 3A 20 33  37 0A 50 6F 72 74 2F 50  Count: 3 7.Port/P
72 6F 74 6F 20 52 61 6E  67 65 3A 20 31 3A 39 0A  roto Ran ge: 1:9.
```

"Low" alerts are only generated on error packets sent from the target host, and because of the nature of error responses, this setting should see very few false positives. However, this setting will never trigger a Filtered Scan alert because of a lack of error responses. This setting is based on a static time window of 60 seconds, after which this window is reset.

"Medium" alerts track Connection Counts, and so will generate Filtered Scan alerts. This setting may false positive on active hosts (NATs, proxies, DNS caches, etc), so the user may need to deploy the use of Ignore directives to properly tune this directive.

"High" alerts continuously track hosts on a network using a time window to evaluate portscan statistics for that host. A "High" setting will catch some slow scans because of the continuous monitoring, but is very sensitive to active hosts. This most definitely will require the user to tune Portscan.

5.16.3 Tuning Portscan

The most important aspect in detecting portscans is tuning the detection engine for your network(s). Here are some tuning tips:

Use the watch_ip, ignore_scanners, and ignore_scanned options. It's important to correctly set these options. The watch_ip option is easy to understand. The analyst should set this option to the list of CIDR blocks and IPs that they want to watch. If no watch_ip is defined, Portscan will watch all network traffic. The ignore_scanners and ignore_scanned options come into play in weeding out legitimate hosts that are very active on your network. Some of the most common examples are NAT IPs, DNS cache servers, syslog servers, and nfs servers. Portscan may not generate false positives for these types of hosts, but be aware when first tuning Portscan for these IPs. Depending on the type of alert that the host generates, the analyst will know which to ignore it as. If the host is generating portsweep events, then add it to the ignore_scanners option. If the host is generating portscan alerts (and is the host that is being scanned), add it to the ignore_scanned option.

Filtered scan alerts are much more prone to false positives. When determining false positives, the alert type is very important. Most of the false positives that Portscan may generate are of the filtered scan alert type. So be much more suspicious of filtered

portscans. Many times this just indicates that a host was very active during the time period in question. If the host continually generates these types of alerts, add it to the ignore_scanners list or use a lower sensitivity level.

Make use of the Priority Count, Connection Count, IP Count, Port Count, IP range, and Port range to determine false positives. The portscan alert details are vital in determining the scope of a portscan and also the confidence of the portscan. In the future, we hope to automate much of this analysis in assigning a scope level and confidence level, but for now the user must manually do this. The easiest way to determine false positives is through simple ratio estimations. The following is a list of ratios to estimate and the associated values that indicate a legitimate scan and not a false positive.

Connection Count / IP Count: This ratio indicates an estimated average of connections per IP. For portscans, this ratio should be high, the higher the better. For portsweeps, this ratio should be low.

Port Count / IP Count: This ratio indicates an estimated average of ports connected to per IP. For portscans, this ratio should be high and indicates that the scanned host's ports were connected to by fewer IPs. For portsweeps, this ratio should be low, indicating that the scanning host connected to few ports but on many hosts.

Connection Count / Port Count: This ratio indicates an estimated average of connections per port. For portscans, this ratio should be low. This indicates that each connection was to a different port. For portsweeps, this ratio should be high. This indicates that there were many connections to the same port.

The reason that Priority Count is not included, is because the priority count is included in the connection count and the above comparisons take that into consideration. The Priority Count play an important role in tuning because the higher the priority count the more likely it is a real portscan or portsweep (unless the host is firewalled).

If all else fails, lower the sensitivity level. If none of these other tuning techniques work or the analyst doesn't have the time for tuning, lower the sensitivity level. You get the best protection the higher the sensitivity level, but it's also important that the portscan detection engine generates alerts that the analyst will find informative. The low sensitivity level only generates alerts based on error responses. These responses indicate a portscan and the alerts generated by the low sensitivity level are highly accurate and require the least tuning. The low sensitivity level does not catch filtered scans, since these are more prone to false positives.

5.17 Sensitive Data Filtering

The `sd_pattern` IPS option provides detection and filtering of Personally Identifiable Information (PII). This information includes credit card numbers, U.S. Social Security numbers, and email addresses. A rich regular expression syntax is available for defining your own PII.

5.17.1 Hyperscan

The `sd_pattern` rule option is powered by the open source Hyperscan library from Intel. It provides a regex grammar which is mostly PCRE compatible. To learn more about Hyperscan see https://intel.github.io/hyperscan/dev-reference/

5.17.2 Syntax

Snort provides `sd_pattern` as IPS rule option with no additional inspector overhead. The Rule option takes the following syntax.

```
sd_pattern: "<pattern>"[, threshold <count>];
```

Pattern

Pattern is the most important and is the only required parameter to `sd_pattern`. It supports 3 built in patterns which are configured by name: "credit_card", "us_social" and "us_social_nodashes", as well as user defined regular expressions of the Hyperscan dialect (see https://intel.github.io/hyperscan/dev-reference/compilation.html#pattern-support).

```
sd_pattern:"credit_card";
```

When configured, Snort will replace the pattern *credit_card* with the built in pattern. In addition to pattern matching, Snort will validate that the matched digits will pass the Luhn-check algorithm. Currently the only pattern that performs extra verification.

```
sd_pattern:"us_social";
sd_pattern:"us_social_nodashes";
```

These special patterns will also be replaced with a built in pattern. Naturally, "us_social" is a pattern of 9 digits separated by −'s in the canonical form.

```
sd_pattern:"\b\w+@ourdomain\.com\b"
```

This is a user defined pattern which matches what is most likely email addresses for the site "ourdomain.com". The pattern is a PCRE compatible regex, \b matches a word boundary (whitespace, end of line, non-word characters) and \w+ matches one or more word characters. \. matches a literal ..

The above pattern would match "a@ourdomain.com", "aa@ourdomain.com" but would not match `1@ourdomain.com` `ab12@ourdo` or `@ourdomain.com`.

Note: This is just an example, this pattern is not suitable to detect many correctly formatted emails.

Threshold

Threshold is an optional parameter allowing you to change built in default value (default value is *1*). The following two instances are identical. The first will assume the default value of *1* the second declaration explicitly sets the threshold to *1*.

```
sd_pattern:"This rule requires 1 match";
sd_pattern:"This rule requires 1 match", threshold 1;
```

That's pretty easy, but here is one more example anyway.

```
sd_pattern:"This is a string literal", threshold 300;
```

This example requires 300 matches of the pattern "This is a string literal" to qualify as a positive match. That is, if the string only occurred 299 times in a packet, you will not see an event.

Obfuscating Credit Cards and Social Security Numbers

Snort provides discreet logging for the built in patterns "credit_card", "us_social" and "us_social_nodashes". Enabling `ips.obfuscat` makes Snort obfuscate the suspect packet payload which was matched by the patterns. This configuration is disabled by default.

```
ips =
{
    obfuscate_pii = true
}
```

5.17.3 Example

A complete Snort IPS rule

```
alert tcp ( sid:1; msg:"Credit Card"; sd_pattern:"credit_card"; )
```

Logged output when running Snort in "cmg" alert format.

```
02/25-21:19:05.125553 [**] [1:1:0] "Credit Card" [**] [Priority: 0] {TCP}  ↩
   10.1.2.3:48620 -> 10.9.8.7:8
02:01:02:03:04:05 -> 02:09:08:07:06:05 type:0x800 len:0x46
10.1.2.3:48620 -> 10.9.8.7:8 TCP TTL:64 TOS:0x0 ID:14 IpLen:20 DgmLen:56
***A**** Seq: 0xB2  Ack: 0x2  Win: 0x2000  TcpLen: 20
```

```
- - - raw[16] - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
58 58 58 58 58 58 58 58 58 58 58 39 32 39 34              XXXXXXXXXXXX9294
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

5.17.4 Caveats

1. Snort currently requires setting the fast pattern engine to use "hyperscan" in order for `sd_pattern` ips option to function correctly.

   ```
   search_engine = { search_method = 'hyperscan' }
   ```

2. Log obfuscation is only applicable to CMG and Unified2 logging formats.
3. Log obfuscation doesn't support user defined PII patterns. It is currently only supported for the built in patterns for Credit Cards and US Social Security numbers.
4. Log obfuscation doesn't work with stream rebuilt packet payloads. (This is a known bug).

5.18 SMTP

SMTP inspector is a service inspector for SMTP protocol.

5.18.1 Overview

The SMTP inspector examines SMTP connections looking for commands and responses. It also identifies the command, header and body sections, TLS data and extracts the MIME attachments. This inspector also identifies and whitelists the SMTP traffic.

SMTP inspector logs the filename, email addresses, attachment names when configured.

5.18.2 Configuration

SMTP command lines can be normalized to remove extraneous spaces. TLS-encrypted traffic can be ignored, which improves performance. In addition, plain-text mail data can be ignored for an additional performance boost.

The configuration options are described below:

normalize and normalize_cmds

Normalization checks for more than one space character after a command. Space characters are defined as space (ASCII 0x20) or tab (ASCII 0x09). "normalize" provides options all|none|cmds, *all* checks all commands, *none* turns off normalization for all commands. *cmds* just checks commands listed with the "normalize_cmds" parameter. For example:

```
smtp = { normalize = 'cmds', normalize_cmds = 'RCPT VRFY EXPN' }
```

ignore_data

Set it to true to ignore data section of mail (except for mail headers) when processing rules.

ignore_tls_data

Set it to true to ignore TLS-encrypted data when processing rules.

max_command_line_len

Alert if an SMTP command line is longer than this value. Absence of this option or a "0" means never alert on command line length. RFC 2821 recommends 512 as a maximum command line length.

max_header_line_len

Alert if an SMTP DATA header line is longer than this value. Absence of this option or a "0" means never alert on data header line length. RFC 2821 recommends 1024 as a maximum data header line length.

max_response_line_len

Alert if an SMTP response line is longer than this value. Absence of this option or a "0" means never alert on response line length. RFC 2821 recommends 512 as a maximum response line length.

alt_max_command_line_len

Overrides max_command_line_len for specific commands For example:

```
alt_max_command_line_len =
{
    {
        command = 'MAIL',
        length = 260,
    },
    {
        command = 'RCPT',
        length = 300,
    },
}
```

invalid_cmds

Alert if this command is sent from client side.

valid_cmds

List of valid commands. We do not alert on commands in this list.

DEFAULT empty list, but SMTP inspector has this list hard-coded: [[ATRN AUTH BDAT DATA DEBUG EHLO EMAL ESAM ESND ESOM ETRN EVFY EXPN HELO HELP IDENT MAIL NOOP ONEX QUEU QUIT RCPT RSET SAML SEND SIZE STARTTLS SOML TICK TIME TURN TURNME VERB VRFY X-EXPS X-LINK2STATE XADR XAUTH XCIR XEXCH50 XGEN XLICENSE XQUE XSTA XTRN XUSR]]

data_cmds

List of commands that initiate sending of data with an end of data delimiter the same as that of the DATA command per RFC 5321 - "<CRLF>.<CRLF>".

binary_data_cmds

List of commands that initiate sending of data and use a length value after the command to indicate the amount of data to be sent, similar to that of the BDAT command per RFC 3030.

auth_cmds

List of commands that initiate an authentication exchange between client and server.

xlink2state

Enable/disable xlink2state alert, options are {disable | alert | drop}. See CVE-2005-0560 for a description of the vulnerability.

MIME processing depth parameters

These four MIME processing depth parameters are identical to their POP and IMAP counterparts. See that section for further details.

b64_decode_depth qp_decode_depth bitenc_decode_depth uu_decode_depth

Log Options

Following log options allow SMTP inspector to log email addresses and filenames. Please note, this is logged only with the unified2 output and is not logged with the console output (-A cmg). u2spewfoo can be used to read this data from the unified2.

log_mailfrom

This option enables SMTP inspector to parse and log the sender's email address extracted from the "MAIL FROM" command along with all the generated events for that session. The maximum number of bytes logged for this option is 1024.

log_rcptto

This option enables SMTP inspector to parse and log the recipient email addresses extracted from the "RCPT TO" command along with all the generated events for that session. Multiple recipients are appended with commas. The maximum number of bytes logged for this option is 1024.

log_filename

This option enables SMTP inspector to parse and log the MIME attachment filenames extracted from the Content-Disposition header within the MIME body along with all the generated events for that session. Multiple filenames are appended with commas. The maximum number of bytes logged for this option is 1024.

log_email_hdrs

This option enables SMTP inspector to parse and log the SMTP email headers extracted from SMTP data along with all generated events for that session. The number of bytes extracted and logged depends upon the email_hdrs_log_depth.

email_hdrs_log_depth

This option specifies the depth for logging email headers. The allowed range for this option is 0 - 20480. A value of 0 will disable email headers logging. The default value for this option is 1464.

5.18.3 Example

```
smtp =
{
    normalize = 'cmds',
    normalize_cmds = 'EXPN VRFY RCPT',
    b64_decode_depth = 0,
    qp_decode_depth = 0,
    bitenc_decode_depth = 0,
    uu_decode_depth = 0,
    log_mailfrom = true,
    log_rcptto = true,
    log_filename = true,
    log_email_hdrs = true,
```

```
    max_command_line_len = 512,
    max_header_line_len = 1000,
    max_response_line_len = 512,
    max_auth_command_line_len = 50,
    xlink2state = 'alert',
    alt_max_command_line_len =
    {
        {
            command = 'MAIL',
            length = 260,
        },
        {
            command = 'RCPT',
            length = 300,
        },
        {
            command = 'HELP',
            length = 500,
        },
        {
            command = 'HELO',
            length = 500,
        },
        {
            command = 'ETRN',
            length = 500,
        },
        {
            command = 'EXPN',
            length = 255,
        },
        {
            command = 'VRFY',
            length = 255,
        },
    },
}
```

5.19 Telnet

Given a telnet data buffer, Telnet will normalize the buffer with respect to telnet commands and option negotiation, eliminating telnet command sequences per RFC 854. It will also determine when a telnet connection is encrypted, per the use of the telnet encryption option per RFC 2946.

5.19.1 Configuring the inspector to block exploits and attacks

ayt_attack_thresh number

Detect and alert on consecutive are you there [AYT] commands beyond the threshold number specified. This addresses a few specific vulnerabilities relating to bsd-based implementations of telnet.

5.20 Trace

Snort 3 retired the different flavors of debug macros that used to be set through the SNORT_DEBUG environment variable. It was replaced by per-module trace functionality. Trace is turned on by setting the specific trace module configuration in snort.lua. As

before, to enable debug tracing, Snort must be configured at build time with --enable-debug-msgs. However, a growing number of modules (such as wizard and snort.inspector_manager) are providing non-debug trace messages in normal production builds.

5.20.1 Trace module

The trace module is responsible for configuring traces and supports the following parameters:

```
output - configure the output method for trace messages
modules - trace configuration for specific modules
constraints - filter traces by the packet constraints
ntuple - on/off packet n-tuple info logging
timestamp - on/off message timestamps logging
```

The following lines, added in snort.lua, will enable trace messages for detection and codec modules. The messages will be printed to syslog if the packet filtering constraints match. Messages will be in extended format, including timestamp and n-tuple packet info at the beginning of each trace message.

```
trace =
{
    output = "syslog",
    modules =
    {
        detection = { detect_engine = 1 },
        decode = { all = 1 }
    },
    constraints =
    {
        ip_proto = 17,
        dst_ip = "10.1.1.2",
        src_port = 100,
        dst_port = 200
    },
    ntuple = true,
    timestamp = true
}
```

The trace module supports config reloading. Also, it's possible to set or clear modules traces and packet filter constraints via the control channel command.

5.20.2 Trace module - configuring traces

The trace module has the **modules** option - a table with trace configuration for specific modules. The following lines placed in snort.lua will enable trace messages for detection, codec and wizard modules:

```
trace =
{
    modules =
    {
        detection = { all = 1 },
        decode = { all = 1 },
        wizard = { all = 1 }
    }
}
```

The detection and snort modules are currently the only modules to support multiple trace options. Others have only the default **all** option, which will enable or disable all traces in a given module. It's available for multi-option modules also and works as a global switcher:

```
trace =
{
    modules =
    {
        detection = { all = 1 }  -- set each detection option to level 1
    }
}

trace =
{
    modules =
    {
        detection = { all = 1, tag = 2 }  -- set each detection option to level 1  ↩
           but the 'tag' to level 2
    }
}
```

Also, it's possible to enable or disable traces for all modules with a top-level **all** option.

The following configuration states that:

- all traces are enabled with verbosity level 5
- traces for the decode module are enabled with level 3
- rule_eval traces for the detection module are enabled with level 1

```
trace =
{
    modules =
    {
        all = 5,
        decode = { all = 3 },
        detection = { rule_eval = 1 }
    }
}
```

The full list of available trace parameters is placed into the "Basic Modules.trace" chapter.

Each option must be assigned an integer value between 0 and 255 to specify a level of verbosity for that option:

```
0 - turn off trace messages printing for the option
1 - print most significant trace messages for the option
255 - print all available trace messages for the option
```

Tracing is disabled by default (verbosity level equals 0). The verbosity level is treated as a threshold, so specifying a higher value will result in all messages with a lower level being printed as well. For example:

```
trace =
{
    modules =
    {
        decode = { all = 3 }  -- messages with levels 1, 2, and 3 will be printed
    }
}
```

5.20.3 Trace module - configuring packet filter constraints for packet related trace messages

There is a capability to filter traces by the packet constraints. The trace module has the **constraints** option - a table with filtering configuration that will be applied to all trace messages that include a packet. Filtering is done on a flow that packet is related. By default filtering is disabled.

Available constraints options:

```
ip_proto - numerical IP protocol ID
src_ip - match all packets with a flow that has this client IP address (passed as ↩
    a string)
src_port - match all packets with a flow that has this source port
dst_ip - match all packets with a flow that has this server IP address (passed as ↩
    a string)
dst_port - match all packets with a flow that has this destination port
match - boolean flag to enable/disable whether constraints will ever match ( ↩
    enabled by default)
```

The following lines placed in snort.lua will enable all trace messages for detection filtered by ip_proto, dst_ip, src_port and dst_port:

```
trace =
{
    modules =
    {
        detection = { all = 1 }
    },
    constraints =
    {
        ip_proto = 6, -- tcp
        dst_ip = "10.1.1.10",
        src_port = 150,
        dst_port = 250
    }
}
```

To create constraints that will never successfully match, set the **match** parameter to *false*. This is useful for situations where one is relying on external packet filtering from the DAQ module, or for preventing all trace messages in the context of a packet. The following is an example of such configuration:

```
trace =
{
    modules =
    {
        snort = { all = 1 }
    },
    constraints =
    {
        match = false
    }
}
```

5.20.4 Trace module - configuring trace output method

There is a capability to configure the output method for trace messages. The trace module has the **output** option with two acceptable values:

```
"stdout" - printing to stdout
"syslog" - printing to syslog
```

By default, the output method will be set based on the Snort run mode. Normally it will use stdout, but if -D (daemon mode) and/or -M (alert-syslog mode) are set, it will instead use syslog.

Example - set output method as syslog:

In snort.lua, the following lines were added:

```
trace =
{
    output = "syslog",
    modules =
    {
        detection = { all = 1 }
    }
}
```

As a result, each trace message will be printed into syslog (the Snort run-mode will be ignored).

5.20.5 Configuring traces via control channel command

There is a capability to configure module trace options and packet constraints via the control channel command by using a Snort shell. In order to enable shell, Snort has to be configured and built with --enable-shell.

The trace control channel command is a way how to configure module trace options and/or packet filter constraints directly during Snort run and without reloading the entire config.

Control channel also allow adjusting trace output format by setting ntuple and timestamp switchers.

After entering the Snort shell, there are two commands available for the trace module:

```
trace.set({ modules = {...}, constraints = {...} }) - set modules traces and  ↩
    constraints (should pass a valid Lua-entry)

trace.set({ modules = { all = N } }) - enable traces for all modules with  ↩
    verbosity level N

trace.set({ ntuple = true/false }) - on/off packet n-tuple info logging

trace.set({ timestamp = true/false }) - on/off timestamp logging

trace.clear() - clear modules traces and constraints
```

Also, it's possible to omit tables in the trace.set() command:

```
trace.set({constraints = {...}}) - set only filtering configuration keeping old  ↩
    modules traces

trace.set({modules = {...}}) - set only module trace options keeping old filtering ↩
     constraints

trace.set({}) - disable traces and constraints (set to empty)
```

5.20.6 Trace messages format

Each tracing message has a standard format:

```
<module_name>:<option_name>:<message_log_level>: <particular_message>
```

The stdout logger also prints thread type and thread instance ID at the beginning of each trace message in a colon-separated manner.

The capital letter at the beginning of the trace message indicates the thread type.

Possible thread types: C – main (control) thread P – packet thread O – other thread

Setting the option - **ntuple** allows you to change the trace message format, expanding it with information about the processed packet.

It will be added at the beginning, right after the thread type and instance ID, in the following format:

```
src_ip src_port -> dst_ip dst_port ip_proto AS=address_space
```

Where:

```
src_ip - source IP address
src_port - source port
dst_ip - destination IP address
dst_port - destination port
ip_proto - IP protocol ID
address_space - unique ID of the address space
```

Those info can be displayed only for IP packets. Port defaults to zero if a packet doesn't have it.

The **timestamp** option extends output format by logging the message time in the next format:

```
MM/DD-hh:mm:ss.SSSSSS
```

Where:

```
M - month
D - day
h - hours
m - minutes
s - seconds
S - milliseconds
```

5.20.7 Example - Debugging rules using detection trace

The detection engine is responsible for rule evaluation. Turning on the trace for it can help with debugging new rules.

The relevant options for detection are as follow:

```
rule_eval - follow rule evaluation
buffer - print evaluated buffer if it changed (level 1) or at every step (level 5)
rule_vars - print value of ips rule options vars
fp_search - print information on fast pattern search
```

Buffer print is useful, but in case the buffer is very big can be too verbose. Choose between verbosity levels 1, 5, or no buffer trace accordingly.

rule_vars is useful when the rule is using ips rule options vars.

In snort.lua, the following lines were added:

```
trace =
{
    modules =
    {
        detection =
        {
            rule_eval = 1,
            buffer = 1,
            rule_vars = 1,
            fp_search = 1
        }
    }
}
```

The pcap has a single packet with payload:

```
10.AAAAAAAfoobar
```

Evaluated on rules:

```
# byte_math + oper with byte extract and content
# VAL = 1, byte_math = 0 + 10
alert tcp ( byte_extract: 1, 0, VAL, string, dec;
byte_math:bytes 1,offset VAL,oper +, rvalue 10, result var1, string dec;
content:"foo", offset var1; sid:3)


#This rule should not trigger
alert tcp (content:"AAAAA"; byte_jump:2,0,relative;
content:"foo", within 3; sid:2)
```

The output:

```
detection:rule_eval:1: packet 1 C2S 127.0.0.1:1234 127.0.0.1:5678 (fast-patterns)
detection:rule_eval:1: Fast pattern search
detection:fp_search:1: 1 fp packet[16]

snort.raw[16]:
- - - - - - - - - - - - - - - -  - - - - - - - - - - - - - - - -  - - - - -  - - - - -
31 30 00 41 41 41 41 41 41 41  66 6F 6F 62 61 72              10.AAAAAAAfoobar
- - - - - - - - - - - - - - - -  - - - - - - - - - - - - - - - -  - - - - -  - - - - -
detection:rule_eval:1: Processing pattern match #1
detection:rule_eval:1: Fast pattern packet[5] = 'AAAAA' |41 41 41 41 41 | ( )
detection:rule_eval:1: Starting tree eval
detection:rule_eval:1: Evaluating option content, cursor name pkt_data, cursor ↩
   position 0

snort.raw[16]:
- - - - - - - - - - - - - - - -  - - - - - - - - - - - - - - - -  - - - - -  - - - - -
31 30 00 41 41 41 41 41 41 41  66 6F 6F 62 61 72              10.AAAAAAAfoobar
- - - - - - - - - - - - - - - -  - - - - - - - - - - - - - - - -  - - - - -  - - - - -
detection:rule_vars:1: Rule options variables: var[0]=0 var[1]=0 var[2]=0
detection:rule_eval:1: Evaluating option byte_jump, cursor name pkt_data, cursor ↩
   position 8
```

```
snort.raw[8]:
- - - - - - - - - - - - - - -  - - - - - - - - - - - - - - - -  - - - - -  - - - - -
41 41 66 6F 6F 62 61 72                                        AAfoobar
- - - - - - - - - - - - - - -  - - - - - - - - - - - - - - - -  - - - - -  - - - - -
detection:rule_eval:1: no match
detection:rule_vars:1: Rule options variables: var[0]=0 var[1]=0 var[2]=0
detection:rule_eval:1: Evaluating option byte_jump, cursor name pkt_data, cursor ↩
   position 9

snort.raw[7]:
- - - - - - - - - - - - - - -  - - - - - - - - - - - - - - - -  - - - - -  - - - - -
41 66 6F 6F 62 61 72                                           Afoobar
- - - - - - - - - - - - - - -  - - - - - - - - - - - - - - - -  - - - - -  - - - - -
detection:rule_eval:1: no match
detection:rule_vars:1: Rule options variables: var[0]=0 var[1]=0 var[2]=0
detection:rule_eval:1: Evaluating option byte_jump, cursor name pkt_data, cursor ↩
   position 10

snort.raw[6]:
- - - - - - - - - - - - - - -  - - - - - - - - - - - - - - - -  - - - - -  - - - - -
66 6F 6F 62 61 72                                              foobar
- - - - - - - - - - - - - - -  - - - - - - - - - - - - - - - -  - - - - -  - - - - -
detection:rule_eval:1: no match
detection:rule_eval:1: no match
detection:rule_eval:1: Processing pattern match #2
detection:rule_eval:1: Fast pattern packet[3] = 'foo' |66 6F 6F | ( )
detection:rule_eval:1: Starting tree eval
detection:rule_eval:1: Evaluating option byte_extract, cursor name pkt_data, ↩
   cursor position 0

snort.raw[16]:
- - - - - - - - - - - - - - -  - - - - - - - - - - - - - - - -  - - - - -  - - - - -
31 30 00 41 41 41 41 41 41 41  66 6F 6F 62 61 72               10.AAAAAAAfoobar
- - - - - - - - - - - - - - -  - - - - - - - - - - - - - - - -  - - - - -  - - - - -
detection:rule_vars:1: Rule options variables: var[0]=1 var[1]=0 var[2]=0
detection:rule_eval:1: Evaluating option byte_math, cursor name pkt_data, cursor ↩
   position 1

snort.raw[15]:
- - - - - - - - - - - - - - -  - - - - - - - - - - - - - - - -  - - - - -  - - - - -
30 00 41 41 41 41 41 41 41 66  6F 6F 62 61 72                  0.AAAAAAAfoobar
- - - - - - - - - - - - - - -  - - - - - - - - - - - - - - - -  - - - - -  - - - - -
detection:rule_vars:1: Rule options variables: var[0]=1 var[1]=10 var[2]=0
detection:rule_eval:1: Evaluating option content, cursor name pkt_data, cursor ↩
   position 2

snort.raw[14]:
- - - - - - - - - - - - - - -  - - - - - - - - - - - - - - - -  - - - - -  - - - - -
00 41 41 41 41 41 41 41 66 6F  6F 62 61 72                     .AAAAAAAfoobar
- - - - - - - - - - - - - - -  - - - - - - - - - - - - - - - -  - - - - -  - - - - -
detection:rule_vars:1: Rule options variables: var[0]=1 var[1]=10 var[2]=0
detection:rule_eval:1: Reached leaf, cursor name pkt_data, cursor position 13

snort.raw[3]:
```

```
- - - - - - - - - - - - - - -  - - - - - - - - - - - - - - - - -  - - - - - -  - - - - - -
62 61 72                                                        bar
- - - - - - - - - - - - - - -  - - - - - - - - - - - - - - - - -  - - - - - -  - - - - - -
detection:rule_eval:1: Matched rule gid:sid:rev 1:3:0
detection:rule_vars:1: Rule options variables: var[0]=1 var[1]=10 var[2]=0
04/22-20:21:40.905630, 1, TCP, raw, 56, C2S, 127.0.0.1:1234, 127.0.0.1:5678,  ↩
    1:3:0, allow
```

5.20.8 Example - Protocols decoding trace

Turning on decode trace will print out information about the packets decoded protocols. Can be useful in case of tunneling.

Example for a icmpv4-in-ipv6 packet:

In snort.lua, the following line was added:

```
trace =
{
    modules =
    {
        decode = { all = 1 }
    }
}
```

The output:

```
decode:all:1: Codec eth (protocol_id: 34525) ip header starts at: 0x7f70800110f0,  ↩
    length is 14
decode:all:1: Codec ipv6 (protocol_id: 1) ip header starts at: 0x7f70800110f0,  ↩
    length is 40
decode:all:1: Codec icmp4 (protocol_id: 256) ip header starts at: 0x7f70800110f0,  ↩
    length is 8
decode:all:1: Codec unknown (protocol_id: 256) ip header starts at: 0x7f70800110f0 ↩
    , length is 0
```

5.20.9 Example - Track the time packet spends in each inspector

There is a capability to track which inspectors evaluate a packet, and how much time the inspector consumes doing so. These trace messages could be enabled by the Snort module trace options:

```
main - command execution traces (main trace logging)
inspector_manager - inspectors execution and time tracking traces
```

Example for a single packet with payload:

```
10.AAAAAAAfoobar
```

In snort.lua, the following lines were added:

```
trace =
{
    modules =
    {
        snort =
        {
            -- could be replaced by 'all = 1'
            main = 1,
```

```
            inspector_manager = 1
        }
    }
}
```

The output:

```
snort:main:1: [0] Queuing command START for execution (refcount 1)
snort:main:1: [0] Queuing command RUN for execution (refcount 1)
snort:main:1: [0] Destroying completed command START
snort:inspector_manager:1: start inspection, raw, packet 1, context 1
snort:inspector_manager:1: enter stream
snort:inspector_manager:1: exit stream, elapsed time: 2 usec
snort:inspector_manager:1: stop inspection, raw, packet 1, context 1, total time:  ↩
   14 usec
snort:inspector_manager:1: post detection inspection, raw, packet 1, context 1
snort:inspector_manager:1: end inspection, raw, packet 1, context 1, total time: 0 ↩
    usec
snort:main:1: [0] Destroying completed command RUN
```

5.20.10 Example - trace filtering by packet constraints:

In snort.lua, the following lines were added:

```
ips =
{
    rules =
    [[
        alert tcp any any -> any any ( msg: "ALERT_TCP"; gid: 1001; sid: 1001 )
        alert udp any any -> any any ( msg: "ALERT_UDP"; gid: 1002; sid: 1002 )
    ]]
}

trace =
{
    modules =
    {
        detection = { rule_eval = 1 }
    },
    constraints =
    {
        ip_proto = 17, -- udp
        dst_ip = "10.1.1.2",
        src_port = 100,
        dst_port = 200
    }
}
```

The processed traffic was next:

```
d ( stack="eth:ip4:udp" )

c ( ip4:a="10.1.1.1", ip4:b="10.1.1.2", udp:a=100, udp:b=200 )
a ( pay="pass" )
b ( pay="pass" )
```

```
c ( ip4:a="10.2.1.1" )
a ( pay="pass" )
b ( pay="pass" )

c ( udp:a=101 )
a ( pay="block" )
b ( pay="block" )
```

The output:

```
detection:rule_eval:1: packet 1 UNK 10.1.1.1:100 10.1.1.2:200 (fast-patterns)
detection:rule_eval:1: Fast pattern processing - no matches found
detection:rule_eval:1: packet 1 UNK 10.1.1.1:100 10.1.1.2:200 (non-fast-patterns)
detection:rule_eval:1: packet 2 UNK 10.1.1.2:200 10.1.1.1:100 (fast-patterns)
detection:rule_eval:1: Fast pattern processing - no matches found
detection:rule_eval:1: packet 2 UNK 10.1.1.2:200 10.1.1.1:100 (non-fast-patterns)
detection:rule_eval:1: packet 3 UNK 10.2.1.1:100 10.1.1.2:200 (fast-patterns)
detection:rule_eval:1: Fast pattern processing - no matches found
detection:rule_eval:1: packet 3 UNK 10.2.1.1:100 10.1.1.2:200 (non-fast-patterns)
detection:rule_eval:1: packet 4 UNK 10.1.1.2:200 10.2.1.1:100 (fast-patterns)
detection:rule_eval:1: Fast pattern processing - no matches found
detection:rule_eval:1: packet 4 UNK 10.1.1.2:200 10.2.1.1:100 (non-fast-patterns)
```

The trace messages for two last packets (numbers 5 and 6) weren't printed.

5.20.11 Example - configuring traces via trace.set() command

In snort.lua, the following lines were added:

```
ips =
{
    rules =
    [[
        alert tcp any any -> any any ( msg: "ALERT_TCP"; gid: 1001; sid: 1001 )
        alert udp any any -> any any ( msg: "ALERT_UDP"; gid: 1002; sid: 1002 )
    ]]
}

trace =
{
    constraints =
    {
        ip_proto = 17, -- udp
        dst_ip = "10.1.1.2",
        src_port = 100,
        dst_port = 200
    },
    modules =
    {
        detection = { rule_eval = 1 }
    }
}
```

The processed traffic was next:

```
# Flow 1
d ( stack="eth:ip4:udp" )
c ( ip4:a="10.1.1.1", ip4:b="10.1.1.2", udp:a=100, udp:b=200 )
a ( data="udp packet 1" )
a ( data="udp packet 2" )

# Flow 2
d ( stack="eth:ip4:tcp" )
c ( ip4:a="10.1.1.3", ip4:b="10.1.1.4", tcp:a=5000, tcp:b=6000 )
a ( syn )
b ( syn, ack )
a ( ack )
a ( ack, data="tcp packet 1" )
a ( ack, data="tcp packet 2" )
a ( fin, ack )
b ( fin, ack )
```

After 1 packet, entering shell and pass the trace.set() command as follows:

```
trace.set({ constraints = { ip_proto = 6, dst_ip = "10.1.1.4", src_port = 5000,  ↩
    dst_port = 6000 }, modules = { decode = { all = 1 }, detection = { rule_eval =  ↩
    1 } } })
```

The output (not full, only descriptive lines):

```
detection:rule_eval:1: packet 1 UNK 10.1.1.1:100 10.1.1.2:200 (fast-patterns)
detection:rule_eval:1: packet 1 UNK 10.1.1.1:100 10.1.1.2:200 (non-fast-patterns)
decode:all:1: Codec udp (protocol_id: 256) ip header starts length is 8
decode:all:1: Codec tcp (protocol_id: 256) ip header starts length is 20
detection:rule_eval:1: packet 3 UNK 10.1.1.3:5000 10.1.1.4:6000 (fast-patterns)
detection:rule_eval:1: packet 3 UNK 10.1.1.3:5000 10.1.1.4:6000 (non-fast-patterns ↩
    )
decode:all:1: Codec tcp (protocol_id: 256) ip header starts length is 20
detection:rule_eval:1: packet 4 UNK 10.1.1.4:6000 10.1.1.3:5000 (fast-patterns)
detection:rule_eval:1: packet 4 UNK 10.1.1.4:6000 10.1.1.3:5000 (non-fast-patterns ↩
    )
decode:all:1: Codec tcp (protocol_id: 256) ip header starts length is 20
detection:rule_eval:1: packet 5 UNK 10.1.1.3:5000 10.1.1.4:6000 (fast-patterns)
detection:rule_eval:1: packet 5 UNK 10.1.1.3:5000 10.1.1.4:6000 (non-fast-patterns ↩
    )
decode:all:1: Codec tcp (protocol_id: 256) ip header starts length is 20
detection:rule_eval:1: packet 6 UNK 10.1.1.3:5000 10.1.1.4:6000 (fast-patterns)
detection:rule_eval:1: packet 6 UNK 10.1.1.3:5000 10.1.1.4:6000 (non-fast-patterns ↩
    )
decode:all:1: Codec tcp (protocol_id: 256) ip header starts length is 20
detection:rule_eval:1: packet 7 UNK 10.1.1.3:5000 10.1.1.4:6000 (fast-patterns)
detection:rule_eval:1: packet 7 UNK 10.1.1.3:5000 10.1.1.4:6000 (non-fast-patterns ↩
    )
decode:all:1: Codec tcp (protocol_id: 256) ip header starts length is 20
detection:rule_eval:1: packet 8 UNK 10.1.1.3:5000 10.1.1.4:6000 (fast-patterns)
detection:rule_eval:1: packet 8 UNK 10.1.1.3:5000 10.1.1.4:6000 (non-fast-patterns ↩
    )
decode:all:1: Codec tcp (protocol_id: 256) ip header starts length is 20
detection:rule_eval:1: packet 9 UNK 10.1.1.4:6000 10.1.1.3:5000 (fast-patterns)
detection:rule_eval:1: packet 9 UNK 10.1.1.4:6000 10.1.1.3:5000 (non-fast-patterns ↩
    )
```

The new configuration was applied. **decode:all:1** messages aren't filtered because they don't include a packet (a packet isn't well-formed at the point when the message is printing).

5.20.12 Other available traces

There are more trace options supported by detection:

```
detect_engine - prints statistics about the engine
pkt_detect - prints a message when disabling content detect for packet
opt_tree - prints option tree data structure
tag - prints a message when a new tag is added
```

The rest support only 1 option, and can be turned on by adding all = 1 to their table in trace lua config.

- stream module trace:

When turned on prints a message in case inspection is stopped on a flow. Example for output:

```
stream:all:1: stop inspection on flow, dir BOTH
```

- stream_ip, stream_user: trace will output general processing messages

Other modules that support trace have messages as seemed fit to the developer. Some are for corner cases, others for complex data structures.

5.21 Wizard

Using the wizard enables port-independent configuration and the detection of malware command and control channels. If the wizard is bound to a session, it peeks at the initial payload to determine the service. For example, *GET* would indicate HTTP and *HELO* would indicate SMTP. Upon finding a match, the service bindings are reevaluated so the session can be handed off to the appropriate inspector. The wizard is still under development; if you find you need to tweak the defaults please let us know.

5.21.1 Wizard patterns

Wizard supports 3 kinds of patterns:

1. Hexes
2. Spells
3. Curses

Each kind of pattern has its own purpose and features. It should be noted that the types of patterns are evaluated exactly in the order in which they are described above. Thus, if some data matches a hex, it will not be processed by spells and curses.

The depth of search for a pattern in the data can be configured using the `max_search_depth` option

TCP packets form a flow, so wizard checks all data in the flow for a match. If no pattern matches and *max_search_depth* is reached, the flow is abandoned by wizard.

UDP packets form a "meta-flow" based on the addresses and ports of the packets. However, unlike TCP processing, for UDP wizard only looks at the first arriving packet from the meta-flow. If no pattern matches that packet or wizard's *max_search_depth* is reached, the meta-flow is abandoned by wizard.

5.21.2 Wizard patterns - Spells

Spell is a text based pattern. The best area of usage - text protocols: http, smtp, sip, etc. Spells are:

- Case insensitive
- Whitespace sensitive
- Able to match by a wildcard symbol

In order to match any sequence of characters in pattern, you should use "*" (glob) symbol in pattern.

```
Example:
  Pattern: '220-*FTP'
  Traffic that would match: '220- Hello world! It's a new FTP server'
```

To escape "*" symbol, put "**" in the pattern.

Spells are configured as a Lua array, each element of which can contain following options:

- *service* - name of the service that would be assigned
- *proto* - protocol to scan
- *client_first* - indicator of which end initiates 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

```
  Example of a spell definition in Lua:
    {
      service = 'smtp',
      proto = 'tcp',
      client_first = true,
      to_server = { 'HELO', 'EHLO' },
      to_client = { '220*SMTP', '220*MAIL' }
    }
```

5.21.3 Wizard patterns - Hexes

Hexes can be used to match binary protocols: dnp3, http2, ssl, etc. Hexes use hexadecimal representation of the data for pattern matching.

Wildcard in hex pattern is a placeholder for exactly one occurrence of any hexadecimal digit and denoted by the symbol "?".

```
Example:
  Pattern: '|05 ?4|'
  Traffic that would match: '|05 84|'
```

Hexes are configured in the same way as spells and have an identical set of options.

```
Example of a hex definition in Lua:
  {
    service = 'dnp3',
    proto = 'tcp',
    client_first = true,
    to_server = { '|05 64|' },
    to_client = { '|05 64|' }
  }
```

5.21.4 Wizard patterns - Curses

Curses are internal algorithms of service identification. They are implemented as state machines in C++ code and can have their own unique state information stored on the flow.

A list of available services can be obtained using `snort --help-config wizard | grep curses`.

```
A configuration which enables some curses:
  curses = {'dce_udp', 'dce_tcp', 'dce_smb', 'sslv2'}
```

5.21.5 Additional Details:

- Note that usually more specific patterns have higher precedence.

```
For example:
  The following spells against 'foobar' payload. The 3rd spell matches.
    { service = 'first',  to_server = { 'foo' } },
    { service = 'second',  to_server = { 'bar' } }
    { service = 'third',  to_server = { 'foobar' } }
```

- If the wizard and one or more service inspectors are configured w/o explicitly configuring the binder, default bindings will be generated which should work for most common cases.
- Also note that while Snort 2 bindings can only be configured in the default policy, each Snort 3 policy can contain a binder leading to an arbitrary hierarchy.
- The entire configuration can be reloaded and hot-swapped during run-time via signal or command in both Snort 2 and Snort 3. Ultimately, Snort 3 will support commands to update the binder on the fly, thus enabling incremental reloads of individual inspectors.
- Both Snort 2 and Snort 3 support server specific configurations via a hosts table (XML in Snort 2 and Lua in Snort 3). The table allows you to map network, protocol, and port to a service and policy. This table can be reloaded and hot-swapped separately from the config file.
- You can find the specifics on the binder, wizard, and hosts tables in the manual or command line like this: snort --help-module binder, etc.

6 DAQ Configuration and Modules

The Data AcQuisition library (DAQ), provides pluggable packet I/O. LibDAQ replaces direct calls to libraries like libpcap with an abstraction layer that facilitates operation on a variety of hardware and software interfaces without requiring changes to Snort. It is possible to select the DAQ module and mode when invoking Snort to perform pcap readback or inline operation, etc. The DAQ library may be useful for other packet processing applications and the modular nature allows you to build new modules for other platforms.

The DAQ library exists as a separate repository on the official Snort 3 GitHub project (https://github.com/snort3/libdaq) and contains a number of bundled DAQ modules including AFPacket, Divert, NFQ, PCAP, and Netmap implementations. Snort 3 itself contains a few new DAQ modules mostly used for testing as described below. Additionally, DAQ modules developed by third parties to facilitate the usage of their own hardware and software platforms exist.

6.1 Building the DAQ Library and Its Bundled DAQ Modules

Refer to the READMEs in the LibDAQ source tarball for instructions on how to build the library and modules as well as details on configuring and using the bundled DAQ modules.

6.2 Configuration

As with a number of features in Snort 3, the LibDAQ and DAQ module configuration may be controlled using either the command line options or by configuring the *daq* Snort module in the Lua configuration (command line option has higher precedence).

DAQ modules may be statically built into Snort, but the more common case is to use DAQ modules that have been built as dynamically loadable objects. Because of this, the first thing to take care of is informing Snort of any locations it should search for dynamic DAQ modules. From the command line, this can be done with one or more invocations of the --daq-dir option, which takes a colon-separated set of paths to search as its argument. All arguments will be collected into a list of locations to be searched. In the Lua configuration, the *daq.module_dirs[]* property is a list of paths for the same purpose.

Next, one must select which DAQ modules they wish to use by name. At least one base module and zero or more wrapper modules may be selected. This is done using the --daq options from the command line or the *daq.modules[]* list-type property. To get a list of the available modules, run Snort with the --daq-list option making sure to specify any DAQ module search directories beforehand. If no DAQ module is specified, Snort will default to attempting to find and use a DAQ module named *pcap*.

Some DAQ modules can be further directly configured using DAQ module variables. All DAQ module variables come in the form of either just a key or a key and a value separated by an equals sign. For example, *debug* or *fanout_type=hash*. The command line option for specifying these is --daq-var and the configuration file equivalent is the *daq.modules[].variables[]* property. The available variables for each module will be shown when listing the available DAQ modules with --daq-list.

The LibDAQ concept of operational mode (passive, inline, or file readback) is automatically configured based on inferring the mode from other Snort configuration. The presence of -r or --pcap-* options implies *read-file*, -i without -Q implies *passive*, and -i with -Q implies *inline*. The mode can be overridden on a per-DAQ module basis with the --daq-mode option on the command line or the *daq.modules[].mode* property.

The DAQ module receive timeout is always configured to 1 second. The packet capture length (snaplen) defaults to 1518 bytes and can be overridden by the -s command line option or *daq.snaplen* property.

Finally, and most importantly, is the input specification for the DAQ module. In readback mode, this is simply the file to be read back and analyzed. For live traffic processing, this is the name of the interface or other necessary input specification as required by the DAQ module to understand what to operate upon. From the command line, the -r option is used to specify a file to be read back and the -i option is used to indicate a live interface input specification. Both are covered by the *daq.inputs[]* property.

For advanced use cases, one additional LibDAQ configuration exists: the number of DAQ messages to request per receive call. In Snort, this is referred to as the DAQ "batch size" and defaults to 64. The default can be overridden with the --daq-batch-size command line option or *daq.batch_size* property. The message pool size requested from the DAQ module will be four times this batch size.

6.2.1 Command Line Example

```
    snort --daq-dir /usr/local/lib/daq --daq-dir /opt/lib/daq --daq afpacket
--daq-var debug --daq-var fanout_type=hash -i eth1:eth2 -Q
```

6.2.2 Configuration File Example

The following is the equivalent of the above command line DAQ configuration in Lua form:

```
daq =
{
    module_dirs =
    {
        '/usr/local/lib/daq',
        '/opt/lib/daq'
    },
    modules =
    {
        {
```

```
            name = 'afpacket',
            mode = 'inline',
            variables =
            {
                'debug',
                'fanout_type=hash'
            }
        }
    },
    inputs =
    {
        'eth1:eth2',
    },
    snaplen = 1518
}
```

The *daq.snaplen* property was included for completeness and may be omitted if the default value is acceptable.

6.2.3 DAQ Module Configuration Stacks

Like briefly mentioned above, a DAQ configuration consists of a base DAQ module and zero or more wrapper DAQ modules. DAQ wrapper modules provide additional functionality layered on top of the base module in a decorator pattern. For example, the Dump DAQ module will capture all passed or injected packets and save them to a PCAP savefile. This can be layered on top of something like the PCAP DAQ module to assess which packets are making it through Snort without being dropped and what actions Snort has taken that involved sending new or modified packets out onto the network (e.g., TCP reset packets and TCP normalizations).

To configure a DAQ module stack from the command line, the --daq option must be given multiple times with the base module specified first followed by the wrapper modules in the desired order (building up the stack). Each --daq option changes which module is being configured by subsequent --daq-var and --daq mode options.

When configuring the same sort of stack in Lua, everything lives in the *daq.modules[]* property. *daq.modules[]* is an array of module configurations pushed onto the stack from top to bottom. Each module configuration **must** contain the name of the DAQ module. Additionally, it may contain an array of variables (*daq.modules[].variables[]*) and/or an operational mode (*daq.modules[].mode*).

If only wrapper modules were specified, Snort will default to implicitly configuring a base module with the name *pcap* in *read-file* mode. This is a convenience to mimic the previous behavior when selecting something like the old Dump DAQ module that may be removed in the future.

For any particularly complicated setup, it is recommended that one configure via a Lua configuration file rather than using the command line options.

6.3 Interaction With Multiple Packet Threads

All packet threads will receive the same DAQ instance configuration with the potential exception of the input specification.

If Snort is in file readback mode, a full set of files will be constructed from the -r/--pcap-file/--pcap-list/--pcap-dir/--pcap-filter options. A number of packet threads will be started up to the configured maximum (-z) to process these files one at a time. As a packet thread completes processing of a file, it will be stopped and then started again with a different file input to process. If the number of packet threads configured exceeds the number of files to process, or as the number of remaining input files dwindles below that number, Snort will stop spawning new packet threads when it runs out of unhandled input files.

When Snort is operating on live interfaces (-i), all packet threads up to the configured maximum will always be started. By default, if only one input specification is given, all packet threads will receive the same input in their configuration. If multiple inputs are given, each thread will be given the matching input (ordinally), falling back to the first if the number of packet threads exceeds the number of inputs.

6.4 DAQ Modules Included With Snort 3

6.4.1 Socket Module

The socket module provides provides a stream socket server that will accept up to 2 simultaneous connections and bridge them together while also passing data to Snort for inspection. The first connection accepted is considered the client and the second connection accepted is considered the server. If there is only one connection, stream data can't be forwarded but it is still inspected.

Each read from a socket of up to snaplen bytes is passed as a packet to Snort along with the ability to retrieve a DAQ_UsrHdr_t structure via ioctl. DAQ_UsrHdr_t conveys IP4 address, ports, protocol, and direction. Socket packets can be configured to be TCP or UDP. The socket DAQ can be operated in inline mode and is able to block packets.

Packets from the socket DAQ module are handled by Snort's stream_user module, which must be configured in the Snort configuration.

To use the socket DAQ, start Snort like this:

```
./snort --daq-dir /path/to/lib/snort_extra/daq \
    --daq socket [--daq-var port=<port>] [--daq-var proto=<proto>] [-Q]

<port> ::= 1..65535; default is 8000
<proto> ::= tcp | udp
```

- This module only supports ip4 traffic.
- This module is only supported by Snort 3. It is not compatible with Snort 2.
- This module is primarily for development and test.

6.4.2 File Module

The file module provides the ability to process files directly without having to extract them from pcaps. Use the file module with Snort's stream_file to get file type identification and signature services. The usual IPS detection and logging, etc. is also available.

You can process all the files in a directory recursively using 8 threads with these Snort options:

```
--pcap-dir path -z 8
```

- This module is only supported by Snort 3. It is not compatible with Snort 2.
- This module is primarily for development and test.

6.4.3 Hext Module

The hext module generates packets suitable for processing by Snort from hex/plain text. Raw packets include full headers and are processed normally. Otherwise the packets contain only payload and are accompanied with flow information (4-tuple) suitable for processing by stream_user.

The first character of the line determines it's purpose:

```
'$' command
'#' comment
'"' quoted string packet data
'x' hex packet data
' ' empty line separates packets
```

The available commands are:

```
$client <ip4> <port>
$server <ip4> <port>

$packet -> client
$packet -> server

$packet <addr> <port> -> <addr> <port>

$sof <i32:ingressZone> <i32:egressZone> <i32:ingressIntf> <i32:egressIntf> <s: ↩
    srcIp> <i16:srcPort> <s:destIp> <i16:dstPort> <u32:opaque> <u64:initiatorPkts>  ↩
    <u64:responderPkts> <u64:initiatorPktsDropped> <u64:responderPktsDropped> <u64: ↩
    initiatorBytesDropped> <u64:responderBytesDropped> <u8:isQosAppliedOnSrcIntf> < ↩
    timeval:sof_timestamp> <timeval:eof_timestamp> <u32:address_space_id> <u32: ↩
    tenant_id> <u16:vlan> <u8:protocol> <u8:flags>
$eof <i32:ingressZone> <i32:egressZone> <i32:ingressIntf> <i32:egressIntf> <s: ↩
    srcIp> <i16:srcPort> <s:destIp> <i16:dstPort> <u32:opaque> <u64:initiatorPkts>  ↩
    <u64:responderPkts> <u64:initiatorPktsDropped> <u64:responderPktsDropped> <u64: ↩
    initiatorBytesDropped> <u64:responderBytesDropped> <u8:isQosAppliedOnSrcIntf> < ↩
    timeval:sof_timestamp> <timeval:eof_timestamp> <u32:address_space_id> <u32: ↩
    tenant_id> <u16:vlan> <u8:protocol> <u8:flags>
```

Client and server are determined as follows. $packet → client indicates to the client (from server) and $packet → server indicates a packet to the server (from client). $packet followed by a 4-tuple uses the heuristic that the client is the side with the greater port number.

The default client and server are 192.168.1.1 12345 and 10.1.2.3 80 respectively. $packet commands with a 4-tuple do not change client and server set with the other $packet commands.

$packet commands should be followed by packet data, which may contain any combination of hex and strings. Data for a packet ends with the next command or a blank line. Data after a blank line will start another packet with the same tuple as the prior one.

$sof and $eof commands generate Start of Flow and End of Flow metapackets respectively. They are followed by a definition of a DAQ_FlowStats_t data structure which will be fed into Snort via the metadata callback.

Strings may contain the following escape sequences:

```
\r = 0x0D = carriage return
\n = 0x0A = new line
\t = 0x09 = tab
\\ = 0x5C = \
```

Format your input carefully; there is minimal error checking and little tolerance for arbitrary whitespace. You can use Snort's -L hext option to generate hext input from a pcap.

- This module only supports ip4 traffic.
- This module is only supported by Snort 3. It is not compatible with Snort 2.
- This module is primarily for development and test.

The hext DAQ also supports a raw mode which is activated by setting the data link type. For example, you can input full ethernet packets with --daq-var dlt=1 (Data link types are defined in the DAQ include sfbpf_dlt.h.) Combine that with the hext logger in raw mode for a quick (and dirty) way to edit pcaps. With --lua "log_hext = { raw = true }", the hext logger will dump the full packet in a way that can be read by the hext DAQ in raw mode. Here is an example:

```
# 3 [96]
```

```
x02 09 08 07 06 05 02 01 02 03 04 05 08 00 45 00 00 52 00 03  # ..............E..R ↩
    ..
x00 00 40 06 5C 90 0A 01 02 03 0A 09 08 07 BD EC 00 50 00 00  # ..@.\............P ↩
    ..
x00 02 00 00 00 02 50 10 20 00 8A E1 00 00 47 45 54 20 2F 74  # ......P.  .....GET ↩
     /t
x72 69 67 67 65 72 2F 31 20 48 54 54 50 2F 31 2E 31 0D 0A 48  # rigger/1 HTTP ↩
    /1.1..H
x6F 73 74 3A 20 6C 6F 63 61 6C 68 6F 73 74 0D 0A              # ost: localhost..
```

A comment indicating packet number and size precedes each packet dump. Note that the commands are not applicable in raw mode and have no effect.