Snort™ Users Manual
2.6.0

The Snort Project

13th September 2006

Contents

Chapter 1

Snort Overview

This manual is based on *Writing Snort Rules* by Martin Roesch and further work from Chris Green <cmg@snort.org>. It is now maintained by Brian Caswell <bmc@snort.org>. If you have a better way to say something or find that something in the documentation is outdated, drop us a line and we will update it. If you would like to submit patches for this document, you can find the latest version of the documentation in LaTeX format in the Snort CVS repository at `/doc/snort_manual.tex`. Small documentation updates are the easiest way to help out the Snort Project.

1.1 Getting Started

Snort really isn't very hard to use, but there are a lot of command line options to play with, and it's not always obvious which ones go together well. This file aims to make using Snort easier for new users.

Before we proceed, there are a few basic concepts you should understand about Snort. Snort can be configured to run in three modes:

- *Sniffer mode,* which simply reads the packets off of the network and displays them for you in a continuous stream on the console (screen).
- *Packet Logger mode,* which logs the packets to disk.
- *Network Intrusion Detection System (NIDS) mode,* the most complex and configurable configuration, which allows Snort to analyze network traffic for matches against a user-defined rule set and performs several actions based upon what it sees.
- *Inline mode,* which obtains packets from iptables instead of from libpcap and then causes iptables to drop or pass packets based on Snort rules that use inline-specific rule types.

1.2 Sniffer Mode

First, let's start with the basics. If you just want to print out the TCP/IP packet headers to the screen (i.e. sniffer mode), try this:

```
./snort -v
```

This command will run Snort and just show the IP and TCP/UDP/ICMP headers, nothing else. If you want to see the application data in transit, try the following:

```
./snort -vd
```

This instructs Snort to display the packet data as well as the headers. If you want an even more descriptive display, showing the data link layer headers, do this:

```
./snort -vde
```

(As an aside, these switches may be divided up or smashed together in any combination. The last command could also be typed out as:

```
./snort -d -v -e
```

and it would do the same thing.)

1.3 Packet Logger Mode

OK, all of these commands are pretty cool, but if you want to record the packets to the disk, you need to specify a logging directory and Snort will automatically know to go into packet logger mode:

```
./snort -dev -l ./log
```

Of course, this assumes you have a directory named `log` in the current directory. If you don't, Snort will exit with an error message. When Snort runs in this mode, it collects every packet it sees and places it in a directory hierarchy based upon the IP address of one of the hosts in the datagram.

If you just specify a plain -l switch, you may notice that Snort sometimes uses the address of the remote computer as the directory in which it places packets and sometimes it uses the local host address. In order to log relative to the home network, you need to tell Snort which network is the home network:

```
./snort -dev -l ./log -h 192.168.1.0/24
```

This rule tells Snort that you want to print out the data link and TCP/IP headers as well as application data into the directory `./log`, and you want to log the packets relative to the 192.168.1.0 class C network. All incoming packets will be recorded into subdirectories of the log directory, with the directory names being based on the address of the remote (non-192.168.1) host.

Note that if both the source and destination hosts are on the home network, they are logged to a directory with a name based on the higher of the two port numbers or, in the case of a tie, the source address.

If you're on a high speed network or you want to log the packets into a more compact form for later analysis, you should consider logging in binary mode. Binary mode logs the packets in tcpdump format to a single binary file in the logging directory:

```
./snort -l ./log -b
```

Note the command line changes here. We don't need to specify a home network any longer because binary mode logs everything into a single file, which eliminates the need to tell it how to format the output directory structure. Additionally, you don't need to run in verbose mode or specify the -d or -e switches because in binary mode the entire packet is logged, not just sections of it. All you really need to do to place Snort into logger mode is to specify a logging directory at the command line using the -l switch—the -b binary logging switch merely provides a modifier that tells Snort to log the packets in something other than the default output format of plain ASCII text.

Once the packets have been logged to the binary file, you can read the packets back out of the file with any sniffer that supports the tcpdump binary format (such as tcpdump or Ethereal). Snort can also read the packets back by using the -r switch, which puts it into playback mode. Packets from any tcpdump formatted file can be processed through Snort in any of its run modes. For example, if you wanted to run a binary log file through Snort in sniffer mode to dump the packets to the screen, you can try something like this:

```
./snort -dv -r packet.log
```

You can manipulate the data in the file in a number of ways through Snort's packet logging and intrusion detection modes, as well as with the BPF interface that's available from the command line. For example, if you only wanted to see the ICMP packets from the log file, simply specify a BPF filter at the command line and Snort will only see the ICMP packets in the file:

```
./snort -dvr packet.log icmp
```

For more info on how to use the BPF interface, read the Snort and tcpdump man pages.

1.4 Network Intrusion Detection System Mode

To enable Network Intrusion Detection System (NIDS) mode so that you don't record every single packet sent down the wire, try this:

```
./snort -dev -l ./log -h 192.168.1.0/24 -c snort.conf
```

where `snort.conf` is the name of your rules file. This will apply the rules configured in the `snort.conf` file to each packet to decide if an action based upon the rule type in the file should be taken. If you don't specify an output directory for the program, it will default to `/var/log/snort`.

One thing to note about the last command line is that if Snort is going to be used in a long term way as an IDS, the -v switch should be left off the command line for the sake of speed. The screen is a slow place to write data to, and packets can be dropped while writing to the display.

It's also not necessary to record the data link headers for most applications, so you can usually omit the -e switch, too.

```
./snort -d -h 192.168.1.0/24 -l ./log -c snort.conf
```

This will configure Snort to run in its most basic NIDS form, logging packets that trigger rules specified in the `snort.conf` in plain ASCII to disk using a hierarchical directory structure (just like packet logger mode).

1.4.1 NIDS Mode Output Options

There are a number of ways to configure the output of Snort in NIDS mode. The default logging and alerting mechanisms are to log in decoded ASCII format and use full alerts. The full alert mechanism prints out the alert message in addition to the full packet headers. There are several other alert output modes available at the command line, as well as two logging facilities.

Alert modes are somewhat more complex. There are seven alert modes available at the command line: full, fast, socket, syslog, console, cmg, and none. Six of these modes are accessed with the -A command line switch. These options are:

Option	Description
`-A fast`	Fast alert mode. Writes the alert in a simple format with a timestamp, alert message, source and destination IPs/ports.
`-A full`	Full alert mode. This is the default alert mode and will be used automatically if you do not specify a mode.
`-A unsock`	Sends alerts to a UNIX socket that another program can listen on.
`-A none`	Turns off alerting.
`-A console`	Sends "fast-style" alerts to the console (screen).
`-A cmg`	Generates "cmg style" alerts.

Packets can be logged to their default decoded ASCII format or to a binary log file via the -b command line switch. To disable packet logging altogether, use the -N command line switch.

For output modes available through the configuration file, see Section 2.3.

Command line logging options override any output options specified in the configuration file. This allows debugging of configuration issues quickly via the command line.

To send alerts to syslog, use the -s switch. The default facilities for the syslog alerting mechanism are LOG_AUTHPRIV and LOG_ALERT. If you want to configure other facilities for syslog output, use the output plugin directives in the rules files. See Section 2.3.1 for more details on configuring syslog output.

For example, use the following command line to log to default (decoded ASCII) facility and send alerts to syslog:

```
./snort -c snort.conf -l ./log -h 192.168.1.0/24 -s
```

As another example, use the following command line to log to the default facility in /var/log/snort and send alerts to a fast alert file:

```
./snort -c snort.conf -A fast -h 192.168.1.0/24
```

1.4.2 Understanding Standard Alert Output

When Snort generates an alert message, it will usually look like the following:

```
[**] [116:56:1] (snort_decoder): T/TCP Detected [**]
```

The first number is the Generator ID, this tells the user what component of Snort generated this alert. For a list of GIDs, please read etc/generators in the Snort source. In this case, we know that this event came from the "decode" (116) component of Snort.

The second number is the Snort ID (sometimes referred to as Signature ID). For a list of preprocessor SIDs, please see etc/gen-msg.map. Rule-based SIDs are written directly into the rules with the *sid* option. In this case, *56* represents a T/TCP event.

The third number is the revision ID. This number is primarily used when writing signatures, as each rendition of the rule should increment this number with the *rev* option.

1.4.3 High Performance Configuration

If you want Snort to go *fast* (like keep up with a 1000 Mbps connection), you need to use unified logging and a unified log reader such as *barnyard*. This allows Snort to log alerts in a binary form as fast as possible while another program performs the slow actions, such as writing to a database.

If you want a text file that's easily parsable, but still somewhat fast, try using binary logging with the "fast" output mechanism.

This will log packets in tcpdump format and produce minimal alerts. For example:

```
./snort -b -A fast -c snort.conf
```

1.4.4 Changing Alert Order

The default way in which Snort applies its rules to packets may not be appropriate for all installations. The Alert rules are applied first, then the Pass rules, and finally, Log rules are applied. This sequence is somewhat counterintuitive, but it's a more foolproof method than allowing a user to write a hundred alert rules that are then disabled by an errant pass rule. For more information on rule types, see Section 3.2.1.

If you know what you're doing, you can use the -o switch to change the default rule application behavior to apply Pass rules, then Alert rules, then Log rules:

```
./snort -d -h 192.168.1.0/24 -l ./log -c snort.conf -o
```

As of Snort 2.6.0, the command line flags `--alert-before-pass` and `--treat-drop-as-alert` were added to handle changes to rule ordering and fix an issue when pass and drop rules were not always enforced. The `--alert-before-pass` option forces alert rules to take affect in favor of a pass rule. The `--treat-drop-as-alert` causes drop, sdrop, and reject rules and any associated alerts to be logged as alerts, rather then the normal action. This allows use of an inline policy with passive/IDS mode.

Additionally, the `--process-all-events` option causes Snort to process every event associated with a packet, while taking the actions based on the rules ordering. Without this option (default case), only the events for the first action based on rules ordering are processed.

Pass rules are special cases here, in that the event processing is terminated when a pass rule is encountered, regardless of the use of `--process-all-events`.

The additions with Snort 2.6.0 will result in the deprecation of the -o switch in a future release.

1.5 Inline Mode

Snort 2.3.0 RC1 integrated the intrusion prevention system (IPS) capability of `Snort Inline` into the official Snort project. `Snort Inline` obtains packets from iptables instead of libpcap and then uses new rule types to help iptables pass or drop packets based on Snort rules.

In order for `Snort Inline` to work properly, you must download and compile the iptables code to include "make install-devel" (`http://www.iptables.org`). This will install the `libipq` library that allows `Snort Inline` to interface with iptables. Also, you must build and install LibNet, which is available from `http://www.packetfactory.net`.

There are three rule types you can use when running Snort with `Snort Inline`:

- **drop** - The drop rule type will tell iptables to drop the packet and log it via usual Snort means.
- **reject** - The reject rule type will tell iptables to drop the packet, log it via usual Snort means, and send a TCP reset if the protocol is TCP or an icmp port unreachable if the protocol is UDP.

- **sdrop** - The sdrop rule type will tell iptables to drop the packet. Nothing is logged.

You can also replace sections of the packet payload when using `Snort Inline`. See Section 1.5.3 for more information.

When using a `reject` rule, there are two options you can use to send TCP resets:

- You can use a RAW socket (the default behavior for `Snort Inline`), in which case you must have an interface that has an IP address assigned to it. If there is not an interface with an IP address assigned with access to the source of the packet, the packet will be logged and the reset packet will never make it onto the network.
- You can also now perform resets via a physical device when using iptables. We take the indev name from ip_queue and use this as the interface on which to send resets. We no longer need an IP loaded on the bridge, and can remain pretty stealthy as the `config layer2_resets` in snort_inline.conf takes a source MAC address which we substitue for the MAC of the bridge. For example:

  ```
  config layer2resets
  ```

  tells `Snort Inline` to use layer2 resets and uses the MAC address of the bridge as the source MAC in the packet, and:

  ```
  config layer2resets: 00:06:76:DD:5F:E3
  ```

  will tell Snort Inline to use layer2 resets and uses the source MAC of 00:06:76:DD:5F:E3 in the reset packet.

1.5.1 Snort Inline Rule Application Order

The current rule application order is:

```
->activation->dynamic->drop->sdrop->reject->alert->pass->log
```

This will ensure that a drop rule has precedence over an alert or log rule. You can use the -o flag to change the rule application order to:

```
->activation->dynamic->pass->drop->sdrop->reject->alert->log
```

1.5.2 New STREAM4 Options for Use with Snort Inline

When using `Snort Inline`, you can use two additional stream4 options:

- `inline_state` (no arguments)

  This option causes Snort to drop TCP packets that are not associated with an existing TCP session, and is not a valid TCP initiator.

- `midstream_drop_alerts` (no arguments)

  By default, when running in inline mode, Snort will silently drop any packets that were picked up in midstream and would have caused an alert to be generated, if not for the 'flow: established' option. This is to mitigate stick/snot type attacks when the user hasn't enabled inline_state. If you want to see the alerts that are silently dropped, enable this keyword. Note that by enabling this keyword, you have opened yourself up to stick/snot-type attacks.

For more information about Stream4, see Section 2.1.3.

1.5.3 Replacing Packets with Snort Inline

Additionally, Jed Haile's content replace code allows you to modify packets before they leave the network. For example:

```
alert tcp any any <> any 80 (msg: "tcp replace"; content:"GET"; replace:"BET";)
alert udp any any <> any 53 (msg: "udp replace"; \
    content: "yahoo"; replace: "xxxxx";)
```

These rules will comb TCP port 80 traffic looking for GET, and UDP port 53 traffic looking for yahoo. Once they are found, they are replaced with BET and xxxxx, respectively. The only catch is that the replace must be the same length as the content.

1.5.4 Installing Snort Inline

To install Snort inline, use the following command:

```
./configure --enable-inline
make
make install
```

1.5.5 Running Snort Inline

First, you need to ensure that the ip_queue module is loaded. Then, you need to send traffic to Snort Inline using the QUEUE target. For example:

```
iptables -A OUTPUT -p tcp --dport 80 -j QUEUE
```

sends all TCP traffic leaving the firewall going to port 80 to the QUEUE target. This is what sends the packet from kernel space to user space (`Snort Inline`). A quick way to get all outbound traffic going to the QUEUE is to use the rc.firewall script created and maintained by the Honeynet Project (`http://www.honeynet.org/papers/honeynet/tools/`) This script is well-documented and allows you to direct packets to `Snort Inline` by simply changing the QUEUE variable to yes.

Finally, start Snort Inline:

```
snort_inline -QDc ../etc/drop.conf -l /var/log/snort
```

You can use the following command line options:

- `-Q` - Gets packets from iptables.
- `-D` - Runs `Snort Inline` in daemon mode. The process ID is stored at `/var/run/snort_inline.pid`
- `-c` - Reads the following configuration file.
- `-l` - Logs to the following directory.

Ideally, Snort Inline will be run using only its own drop.rules. If you want to use Snort for just alerting, a separate process should be running with its own rule set.

1.5.6 Using the Honeynet Snort Inline Toolkit

The Honeynet Snort Inline Toolkit is a statically compiled `Snort Inline` binary put together by the Honeynet Project for the Linux operating system. It comes with a set of drop.rules, the `Snort Inline` binary, a snort-inline rotation shell script, and a good README. It can be found at:

`http://www.honeynet.org/papers/honeynet/tools/`

1.5.7 Troubleshooting Snort Inline

If you run Snort Inline and see something like this:

```
Initializing Output Plugins!
Reading from iptables
Log directory = /var/log/snort
Initializing Inline mode
InlineInit: : Failed to send netlink message: Connection refused
```

More than likely, the ip_queue module is not loaded or ip_queue support is not compiled into your kernel. Either recompile your kernel to support ip_queue, or load the module.

The ip_queue module is loaded by executing:

```
insmod ip_queue
```

Also, if you want to ensure Snort Inline is getting packets, you can start it in the following manner:

```
snort_inline -Qvc <configuration file>
```

This will display the header of every packet that Snort Inline sees.

1.6 Miscellaneous

1.6.1 Running in Daemon Mode

If you want to run Snort in daemon mode, you can the add -D switch to any combination described in the previous sections. Please notice that if you want to be able to restart Snort by sending a SIGHUP signal to the daemon, you *must* specify the full path to the Snort binary when you start it, for example:

```
/usr/local/bin/snort -d -h 192.168.1.0/24 \
    -l /var/log/snortlogs -c /usr/local/etc/snort.conf -s -D
```

Relative paths are not supported due to security concerns.

Snort PID File

When Snort is run in daemon mode, the daemon creates a PID file in the log directory. In Snort 2.6, the `--pid-path` command line switch causes Snort to write the PID file in the directory specified.

Additionally, the `--create-pidfile` switch can be used to force creation of a PID file even when not running in daemon mode.

The PID file will be locked so that other snort processes cannot start. Use the `--nolock-pidfile` switch to not lock the PID file.

1.6.2 Obfuscating IP Address Printouts

If you need to post packet logs to public mailing lists, you might want to use the -O switch. This switch obfuscates your IP addresses in packet printouts. This is handy if you don't want people on the mailing list to know the IP addresses involved. You can also combine the -O switch with the -h switch to only obfuscate the IP addresses of hosts on the home network. This is useful if you don't care who sees the address of the attacking host. For example, you could use the following command to read the packets from a log file and dump them to the screen, obfuscating only the addresses from the 192.168.1.0/24 class C network:

```
./snort -d -v -r snort.log -O -h 192.168.1.0/24
```

1.6.3 Specifying Multiple-Instance Identifiers

In Snort v2.4, the `-G` command line option was added that specifies an instance identifier for the event logs. This option can be used when running multiple instances of snort, either on different CPUs, or on the same CPU but a different interface. Each Snort instance will use the value specified to generate unique event IDs. Users can specify either a decimal value (`-G 1`) or hex value preceded by 0x (`-G 0x11`). This is also supported via a long option `--logid`.

1.7 More Information

Chapter 2 contains much information about many configuration options available in the configuration file. The Snort manual page and the output of `snort -?` or `snort --help` contain information that can help you get Snort running in several different modes.

In many shells, a backslash (\) is needed to escape the ?, so you may have to type `snort -\?` instead of `snort -?` for a list of Snort command line options.

The Snort web page (`http://www.snort.org`) and the Snort Users mailing list (`http://marc.theaimsgroup.com/?l=snort-users` at `snort-users@lists.sourceforge.net` provide informative announcements as well as a venue for community discussion and support. There's a lot to Snort, so sit back with a beverage of your choosing and read the documentation and mailing list archives.

Chapter 2

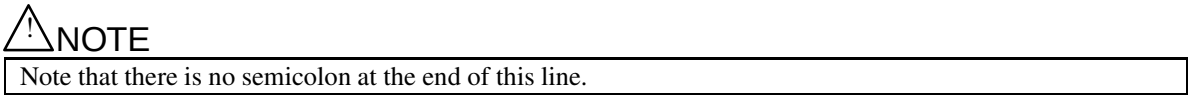
Configuring Snort

2.0.1 Includes

The `include` keyword allows other rules files to be included within the rules file indicated on the Snort command line. It works much like an #include from the C programming language, reading the contents of the named file and adding the contents in the place where the include statement appears in the file.

Format

```
include <include file path/name>
```



Included files will substitute any predefined variable values into their own variable references. See Section 2.0.2 for more information on defining and using variables in Snort rules files.

2.0.2 Variables

Variables may be defined in Snort. These are simple substitution variables set with the `var` keyword as shown in Figure 2.1.

Format

```
var <name> <value>
```

```
var MY_NET [192.168.1.0/24,10.1.1.0/24]
alert tcp any any -> $MY_NET any (flags:S; msg:"SYN packet";)
```

Figure 2.1: Example of Variable Definition and Usage

Rule variable names can be modified in several ways. You can define meta-variables using the $ operator. These can be used with the variable modifier operators `?` and `-`, as described in the following table:

Variable Syntax	Description
var	Defines a meta-variable.
$(var) or $var	Replaces with the contents of variable var.
$(var:-default)	Replaces the contents of the variable var with "default" if var is undefined.
$(var:?message)	Replaces with the contents of variable var or prints out the error message and exits.

See Figure 2.2 for an example of advanced variable usage in action.

```
var MY_NET 192.168.1.0/24
log tcp any any -> $(MY_NET:?MY_NET is undefined!) 23
```

Figure 2.2: Figure Advanced Variable Usage Example

2.0.3 Config

Many configuration and command line options of Snort can be specified in the configuration file.

Format

```
config <directive> [: <value>]
```

Directives

Table 2.1: Config Directives

Command	Example	Description
order	config order: pass alert log activation	Changes the order that rules are evaluated.
alertfile	config alertfile: alerts	Sets the alerts output file.
classification	config classification: misc-activity,Misc activity,3	See Table 3.2 for a list of classifications.
dump_chars_only	config dump_chars_only	Turns on character dumps (snort -C).
dump_payload	config dump_payload	Dumps application layer (snort -d).
decode_data_link	config decode_data_link	Decodes Layer2 headers (snort -e).
bpf_file	config bpf_file: filters.bpf	Specifies BPF filters (snort -F).
daemon	config daemon	Forks as a daemon (snort -D).
interface	config interface: xl0	Sets the network interface (snort -i).
alert_with_interface_name	config alert_with_interface_name	Appends interface name to alert (snort -I).
logdir	config logdir: /var/log/snort	Sets the logdir (snort -l).
umask	config umask: 022	Sets umask when running (snort -m).
pkt_count	config pkt_count: 13	Exits after N packets (snort -n).
nolog	config nolog	Disables logging. Note: Alerts will still occur. (snort -N).
obfuscate	config obfuscate	Obfuscates IP Addresses (snort -O).
no_promisc	config no_promisc	Disables promiscuous mode (snort -p).
quiet	config quiet	Disables banner and status reports (snort -q).
chroot	config chroot: /home/snort	Chroots to specified dir (snort -t).
checksum_mode	config checksum_mode : all	Types of packets to calculate checksums. Values: none, noip, notcp, noicmp, noudp, ip, tcp, udp, icmp or all.

checksum_drop	config checksum_drop : all	Types of packets to drop if invalid checksums. Values: none, noip, notcp, noicmp, noudp, ip, tcp, udp, icmp or all (only applicable in inline mode and for packets checked per checksum_mode config option).
set_gid	config set_gid: 30	Changes GID to specified GID (snort -g).
set_uid	set_uid: snort_user	Sets UID to <id> (snort -u).
utc	config utc	Uses UTC instead of local time for timestamps (snort -U).
verbose	config verbose	Uses verbose logging to STDOUT (snort -v).
dump_payload_verbose	config dump_payload_verbose	Dumps raw packet starting at link layer (snort -X).
show_year	config show_year	Shows year in timestamps (snort -y).
stateful	config stateful	Sets assurance mode for stream4 (est). See the stream4_reassemble configuration in table 2.3.
min_ttl	config min_ttl:30	Sets a Snort-wide minimum ttl to ignore all traffic.
disable_decode_alerts	config disable_decode_alerts	Turns off the alerts generated by the decode phase of Snort.
disable_tcpopt_experimental_alerts	config disable_tcpopt_experimental_alerts	Turns off alerts generated by experimental TCP options.
disable_tcpopt_obsolete_alerts	config disable_tcpopt_obsolete_alerts	Turns off alerts generated by obsolete TCP options.
disable_tcpopt_ttcp_alerts	config disable_tcpopt_ttcp_alerts	Turns off alerts generated by T/TCP options.
disable_ttcp_alerts	config disable_ttcp_alerts	Turns off alerts generated by T/TCP options.
disable_tcpopt_alerts	config disable_tcpopt_alerts	Disables option length validation alerts.
disable_ipopt_alerts	config disable_ipopt_alerts	Disables IP option length validation alerts.
enable_decode_drops	config enable_decode_drops	Enables the dropping of bad packets identified by decoder (only applicable in inline mode).
enable_tcpopt_experimental_drops	config enable_tcpopt_experimental_drops	Enables the dropping of bad packets with experimental TCP option. (only applicable in inline mode).
enable_tcpopt_obsolete_drops	config enable_tcpopt_obsolete_drops	Enables the dropping of bad packets with obsolete TCP option. (only applicable in inline mode).
enable_tcpopt_ttcp_drops	enable_tcpopt_ttcp_drops	Enables the dropping of bad packets with T/TCP option. (only applicable in inline mode).
enable_tcpopt_drops	config enable_tcpopt_drops	Enables the dropping of bad packets with bad/truncated TCP option (only applicable in inline mode).
enable_ipopt_drops	config enable_ipopt_drops	Enables the dropping of bad packets with bad/truncated IP options (only applicable in inline mode).
flowbits_size	config flowbits_size: 128	Specifies the maximum number of flowbit tags that can be used within a rule set.

event_queue	config event_queue: max_queue 512 log 100 order_events priority	Specifies conditions about Snort's event queue. You can use the following options: • max_queue <integer> (max events supported) • log <integer> (number of events to log) • order_events [priority|content_length] (how to order events within the queue) See Section 3.10 for more information and examples.
layer2resets	config layer2resets: 00:06:76:DD:5F:E3	This option is only available when running in inline mode. See Section 1.5.
detection	config detection: search-method ac no_stream_inserts max_queue_events 128	Makes changes to the detection engine. The following options can be used: • search-method<ac|ac-std|acs|ac-banded|ac-sparsebands|mwm|lowmem> – ac Aho-Corasick Full (high memory, best performance) – ac-std Aho-Corasick Standard (moderate memory, high performance) – acs Aho-Corasick Sparse (small memory, moderate performance) – ac-banded Aho-Corasick Banded (small memory, moderate performance) – ac-sparsebands Aho-Corasick Sparse-Banded (small memory, high performance) – lowmem Low Memory Keyword Trie (small memory, low performance) – mwm Wu-Manber (small memory, low performance) • no_stream_inserts • max_queue_events<integer>
asn1	config asn1:256	Specifies the maximum number of nodes to track when doing ASN1 decoding. See Section 2.1.15 for more information and examples.
snaplen	config snaplen: 2048	Set the snaplength of packet, same effect as -P <snaplen> or --snaplen <snaplen> options.

read_bin_file	config read_bin_file: test_alert.pcap	Specifies a pcap file to use (instead of reading from network), same effect as -r <tf> option.
reference	config reference: myref http://myurl.com/?id=	Adds a new reference system to Snort.
ignore_ports	config ignore_ports: udp 1:17 53	Specifies ports to ignore (useful for ignoring noisy NFS traffic). Specify the protocol (TCP, UDP, IP, or ICMP), followed by a list of ports. Port ranges are supported.

The Wu-Manber pattern matching engine (search-method mwm) will be deprecated in a future Snort release in favor of pattern matching algorithms with better performance and smaller memory consumption.

2.1 Preprocessors

Preprocessors were introduced in version 1.5 of Snort. They allow the functionality of Snort to be extended by allowing users and programmers to drop modular plugins into Snort fairly easily. Preprocessor code is run before the detection engine is called, but after the packet has been decoded. The packet can be modified or analyzed in an out-of-band manner using this mechanism.

Preprocessors are loaded and configured using the `preprocessor` keyword. The format of the preprocessor directive in the Snort rules file is:

```
preprocessor <name>: <options>

preprocessor minfrag: 128
```

Figure 2.3: Preprocessor Directive Format Example

2.1.1 Frag2

Frag2 is deprecated in Snort 2.4.0 and later in favor of frag3. See Section 2.1.2 for more information about frag3.

2.1.2 Frag3

The frag3 preprocessor is a target-based IP defragmentation module for Snort. Frag3 is intended as a replacement for the frag2 defragmentation module and was designed with the following goals:

1. Faster execution than frag2 with less complex data management.
2. Target-based host modeling anti-evasion techniques.

The frag2 preprocessor used splay trees extensively for managing the data structures associated with defragmenting packets. Splay trees are excellent data structures to use when you have some assurance of locality of reference for the data that you are handling but in high speed, heavily fragmented environments the nature of the splay trees worked against the system and actually hindered performance. Frag3 uses the sfxhash data structure and linked lists for data handling internally which allows it to have much more predictable and deterministic performance in any environment which should aid us in managing heavily fragmented environments.

Target-based analysis is a relatively new concept in network-based intrusion detection. The idea of a target-based system is to model the actual targets on the network instead of merely modeling the protocols and looking for attacks within them. When IP stacks are written for different operating systems, they are usually implemented by people who read the RFCs and then write their interpretation of what the RFC outlines into code. Unfortunately, there are ambiguities in the way that the RFCs define some of the edge conditions that may occurr and when this happens different people implement certain aspects of their IP stacks differently. For an IDS this is a big problem.

In an environment where the attacker can determine what style of IP defragmentation is being used on a particular target, the attacker can try to fragment packets such that the target will put them back together in a specific manner while any passive systems trying to model the host traffic have to guess which way the target OS is going to handle the overlaps and retransmits. As I like to say, if the attacker has more information about the targets on a network than the IDS does, it is possible to evade the IDS. This is where the idea for "target-based IDS" came from. For more detail on this issue and how it affects IDS, check out the famous Ptacek & Newsham paper at `http://www.snort.org/docs/idspaper/`.

The basic idea behind target-based IDS is that we tell the IDS information about hosts on the network so that it can avoid Ptacek & Newsham style evasion attacks based on information about how an individual target IP stack operates. Vern Paxson and Umesh Shankar did a great paper on this very topic in 2003 that detailed mapping the hosts on a network and determining how their various IP stack implementations handled the types of problems seen in IP defragmentation and TCP stream reassembly. Check it out at `http://www.icir.org/vern/papers/activemap-oak03.pdf`.

We can also present the IDS with topology information to avoid TTL-based evasions and a variety of other issues, but that's a topic for another day. Once we have this information we can start to really change the game for these complex modeling problems.

Frag3 was implemented to showcase and prototype a target-based module within Snort to test this idea.

Frag 3 Configuration

Frag3 configuration is somewhat more complex than frag2. There are at least two preprocessor directives required to activate frag3, a global configuration directive and an engine instantiation. There can be an arbitrary number of engines defined at startup with their own configuration, but only one global configuration.

Global Configuration

- Preprocessor name: `frag3_global`
- Available options:
  - `max_frags` <number> - Maximum simultaneous fragments to track. Default is 8192.
  - `memcap` <bytes> - Memory cap for self preservation. Default is 4MB.
  - `prealloc_frags` <number> - Alternate memory management mode. Use preallocated fragment nodes (faster in some situations).

Engine Configuration

- Preprocessor name: `frag3_engine`
- Available options:
  - `timeout` <seconds> - Timeout for fragments. Fragments in the engine for longer than this period will be automatically dropped. Default is 60 seconds.
  - `ttl_limit` <hops> - Max TTL delta acceptable for packets based on the first packet in the fragment. Default is 5.
  - `min_ttl` <value> - Minimum acceptable TTL value for a fragment packet. Default is 1.
  - `detect_anomalies` - Detect fragment anomalies.
  - `bind_to` <ip_list> - IP List to bind this engine to. This engine will only run for packets with destination addresses contained within the IP List. Default value is `all`.
  - `policy` <type> - Select a target-based defragmentation mode. Available types are first, last, bsd, bsd-right, linux. Default type is bsd.

    The Paxson Active Mapping paper introduced the terminology frag3 is using to describe policy types. The known mappings are as follows. Anyone who develops more mappings and would like to add to this list please feel free to send us an email!

Platform	Type
AIX 2	BSD
AIX 4.3 8.9.3	BSD
Cisco IOS	Last
FreeBSD	BSD
HP JetDirect (printer)	BSD-right
HP-UX B.10.20	BSD
HP-UX 11.00	First
IRIX 4.0.5F	BSD
IRIX 6.2	BSD
IRIX 6.3	BSD
IRIX64 6.4	BSD
Linux 2.2.10	linux
Linux 2.2.14-5.0	linux
Linux 2.2.16-3	linux
Linux 2.2.19-6.2.10smp	linux
Linux 2.4.7-10	linux
Linux 2.4.9-31SGI 1.0.2smp	linux
Linux 2.4 (RedHat 7.1-7.3)	linux
MacOS (version unknown)	First
NCD Thin Clients	BSD
OpenBSD (version unknown)	linux
OpenBSD (version unknown)	linux
OpenVMS 7.1	BSD
OS/2 (version unknown)	BSD
OSF1 V3.0	BSD
OSF1 V3.2	BSD
OSF1 V4.0,5.0,5.1	BSD
SunOS 4.1.4	BSD
SunOS 5.5.1,5.6,5.7,5.8	First
Tru64 Unix V5.0A,V5.1	BSD
Vax/VMS	BSD
Windows (95/98/NT4/W2K/XP)	First

format

```
preprocessor frag3_global
preprocessor frag3_engine
```

Figure 2.4: Example configuration (Basic)

```
preprocessor frag3_global: prealloc_nodes 8192
preprocessor frag3_engine: policy linux, bind_to 192.168.1.0/24
preprocessor frag3_engine: policy first, bind_to [10.1.47.0/24,172.16.8.0/24]
preprocessor frag3_engine: policy last, detect_anomalies
```

Figure 2.5: Example configuration (Advanced)

Note in the advanced example (Figure 2.5), there are three engines specified running with *Linux*, `first` and `last` policies assigned. The first two engines are bound to specific IP address ranges and the last one applies to all other traffic. Packets that don't fall within the address requirements of the first two engines automatically fall through to the third one.

Frag 3 Alert Output

Frag3 is capable of detecting eight different types of anomalies. Its event output is packet-based so it will work with all output modes of Snort. Read the documentation in the `doc/signatures` directory with filenames that begin with "123-" for information on the different event types.

2.1.3 Stream4

The Stream4 module provides TCP stream reassembly and stateful analysis capabilities to Snort. Robust stream reassembly capabilities allow Snort to ignore "stateless" attacks (which include the types of attacks that Stick and Snot produce). Stream4 also gives large scale users the ability to track many simultaneous TCP streams. Stream4 is set to handle 8192 simultaneous TCP connections in its default configuration; however, it scales to handle over 100,000 simultaneous connections.

Stream4 contains two configurable modules: the Stream4 preprocessor and the associated Stream4 reassemble plugin. The `stream4_reassemble` options are listed below.

Additional options can be used if Snort is running in inline mode. See Section 1.5.2 for more information.

Stream4 Format

```
preprocessor stream4: [noinspect], [asynchronous_link], [keepstats [<machine|binary>]], \
                      [detect_scans], [log_flushed_streams], [detect_state_problems], \
                      [disable_evasion_alerts], [timeout <seconds>], [memcap <bytes>],  \
                      [max_sessions <num sessions>], [cache_clean_percent <% of sessions>], \
                      [cache_clean_sessions <num of sessions>], [ttl_limit <count>], \
                      [self_preservation_threshold], [self_preservation_period], \
                      [suspend_threshold], [suspend_period], [enforce_state], \
                      [state_protection], [server_inspect_limit <bytes>]
```

Option	Description
asynchronous_link	Uses state transitions based only on one-sided conversation (no tracking of acknowledge/sequence numbers).
cache_clean_percent	Purges this percent of least-recently used sessions from the session cache (overrides cache_clean_sessions).
cache_clean_sessions	Purges this number of least-recently used sessions from the session cache.
detect_scans	Turns on alerts for portscan events.
detect_state_problems	Turns on alerts for stream events of note, such as evasive RST packets, data on the SYN packet, and out of window sequence numbers.
enforce_state	Enforces statefulness so that sessions aren't picked up mid-stream.
keepstats	Records session summary information in <logdir>/session.log. If no options are specified, output is human readable.
log_flushed_streams	Log the packets that are part of reassembled stream.
disable_evasion_alerts	Turns off alerts for events such as TCP overlap.
timeout <seconds>	Amount of time to keep an inactive stream in the state table; sessions that are flushed will automatically be picked up again if more activity is seen. The default value is 30 seconds.
memcap <bytes>	Sets the number of bytes used to store packets for reassembly.
max_sessions	Sets the maximum number of simultaneous sessions.
noinspect	Disables stateful inspection.
ttl_limit	Sets the delta value that will set off an evasion alert.
self_preservation_threshold	Set limit on number of sessions before entering self-preservation mode (only reassemble data on the default ports).
self_preservation_period	Set length of time (seconds) to remain in self-preservation mode.
suspend_threshold	Sets limit on number of sessions before entering suspend mode (no reassembly).
suspend_period	Sets length of time (seconds) to remain in suspend mode.
server_inspect_limit	Restricts inspection of server traffic to this many bytes until another client request is seen (ie: client packet with data).
state_protection	Protects self against DoS attacks.

stream4_reassemble Format

```
preprocessor stream4_reassemble: [clientonly], [serveronly], [both], [noalerts], \
                                 [favor_old], [favor_new], [flush_on_alert], \
                                 [flush_behavior <number>], [flush_base <number>], \

                                 [flush_range <number>], [flush_seed <number>], \
                                 [overlap_limit], [ports <portlist>], \
                                 [zero_flushed_packets], [flush_data_diff_size <number>]
```

Option	Description
clientonly	Provides reassembly for the client side of a connection only.
serveronly	Provides reassembly for the server side of a connection only.
both	Reassemble for client and server sides of connection.
noalerts	Won't alert on events that may be insertion or evasion attacks.
favor_old	Favor old segments based on sequence number over a new segments.
favor_new	Favor new segments based on sequence number over a old segments.
flush_on_alert	Flush a stream when an individual packet causes an alert.
flush_behavior <number>	Use specified flush behavior. Number greater than 0 means use old static flush points. Number equal to 0 means use new larger flush points. Number less than 0 means use random flush points defined by flush_base, flush_seed, and flush_range.
flush_base <number>	Lowest allowed random flushpoint. The default value is 512 bytes. Only used if flush_behavior is less than 0.
flush_range <number>	Space within random flushpoints are generated. The default value is 1213. Only used if flush_behavior is less than 0.
flush_seed <number>	Random seed for flushpoints. The default value is computed from Snort PID + time. Only used if flush_behavior is less than 0.
overlap_limit	Alert when the number of overlapping data bytes reaches a threshold.
ports <portlist>	Provides reassembly for a whitespace-separated list of ports. By default, reassembly is performed for ports 21, 23, 25, 42, 53, 80, 110, 111, 135, 136, 137, 139, 143, 445, 513, 1443, 1521, and 3306. To perform reassembly for all ports, use all as the port list.
flush_data_diff_size <number>	minumum size of a packet to zero out the empty space in a rebuilt packet.
zero_flushed_packets	Zero out any space that is not filled in when flushing a rebuilt packet.

Notes

Just setting the stream4 and stream4_reassemble directives without arguments in the snort.conf file will set them up in their default configurations shown in Table 2.2 and Table 2.3.

Table 2.2: Stream4 Defaults

Option	Default
session timeout (timeout)	30 seconds
session memory cap (memcap)	8388608 bytes
stateful inspection (noinspect)	active (noinspect disabled)
stream stats (keepstats)	inactive
state problem alerts (detect_state_problems)	inactive (detect_state_problems disabled)
evasion alerts (disable_evasion_alerts)	inactive (disable_evasion_alerts enabled)
asynchronous link (asynchronous_link)	inactive
log flushed streams (log_flushed_streams)	inactive
max sessions (max_sessions)	8192
session cache purge (cache_clean_sessions)	5
session cache purge percent (cache_clean_percent)	inactive
self preservation threshold (self_preservation_threshold)	50 sessions/sec
self preservation period (self_preservation_period)	90 seconds
suspend threshold (suspend_threshold)	200 sessions/sec
suspend period (suspend_period)	30 seconds
state protection (state_protection)	inactive
server inspect limit (server_inspect_limit)	-1 (inactive)

Table 2.3: stream4_reassemble Defaults

Option	**Default**
reassemble client (`clientonly`)	active
reassemble server (`serveronly`)	inactive
reassemble both (`both`)	inactive
reassemble ports (`ports`)	21 23 25 42 53 80 110 111 135 136 137 139 143 445 513 1433 1521 3306
reassembly alerts (`noalerts`)	active (`noalerts` disabled)
favor old packet (`favor_old`)	active
favor new packet (`favor_new`)	inactive
flush on alert (`flush_on_alert`)	inactive
overlap limit (`overlap_limit`)	-1 (inactive)

2.1.4 Flow

The Flow tracking module is meant to start unifying the state keeping mechanisms of Snort into a single place. As of Snort 2.1.0, only a portscan detector is implemented, but in the long term, many of the stateful subsystems of Snort will be migrated over to becoming flow plugins. With the introduction of flow, this effectively makes the conversation preprocessor obsolete.

An IPv4 flow is unique when the IP protocol (`ip_proto`), source IP (`sip`), source port (`sport`), destination IP (`dip`), and destination port (`dport`) are the same. The `dport` and `sport` are 0 unless the protocol is TCP or UDP.

Format

```
preprocessor flow: [memcap <bytes>], [rows <count>], \
                   [stats_interval <seconds>], [hash <1|2>]
```

Table 2.4: Flow Options

Option	Description
`memcap`	Number of bytes to allocate.
`rows`	Number of rows for the flow hash table. [a]
`stats_interval`	Interval (in seconds) to dump statistics to STDOUT. Set this to 0 to disable.
`hash`	Hashing method to use.[b]

[a] This number can be increased, at the cost of using more memory, to enhance performance. Increasing rows provides a larger hash table.

[b] 1 - hash by byte, 2 - hash by integer (faster, not as much of a chance to become diverse). The hash table has a pseudo-random salt picked to make algorithmic complexity attacks much more difficult.

Example Configuration

```
preprocessor flow: stats_interval 0 hash 2
```

2.1.5 Portscan

The ”Portscan” preprocessor was deprecated in Snort 2.2, in favor of Flow Portscan, which was deprecated in Snort 2.3, in favor of sfPortscan.

2.1.6 Flow-Portscan

The Flow-Portscan preprocessor was deprecated in Snort 2.3, in favor of sfPortscan.

2.1.7 sfPortscan

The sfPortscan module, developed by Sourcefire, 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 service ports 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. sfPortscan was designed to be able to detect the different types of scans Nmap can produce.

sfPortscan will currently alert for the following types of Nmap scans:

- TCP Portscan
- UDP Portscan
- IP Portscan

These alerts are for one→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.

sfPortscan also alerts for the following types of decoy portscans:

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

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

sfPortscan alerts for the following types of distributed portscans:

- TCP Distributed Portscan
- UDP Distributed Portscan
- IP Distributed Portscan

These are many→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.

Negative queries will be distributed among scanning hosts, so we track this type of scan through the scanned host.

sfPortscan alerts for the following types of portsweeps:

- TCP Portsweep
- UDP Portsweep
- IP Portsweep
- ICMP Portsweep

These alerts are for one→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.

sfPortscan alerts on the following filtered portscans and portsweeps:

- 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 of 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.

sfPortscan only generates one alert for each host pair in question during the time window (more on windows below). On TCP scan alerts, sfPortscan will also display any open ports that were scanned. On TCP sweep alerts however, sfPortscan will only track open ports after the alert has been triggered. Open port events are not individual alerts, but tags based on the orginal scan alert.

sfPortscan Configuration

You may want to use the following line in your `snort.conf` to disable evasion alerts within stream4 because some scan packets can cause these alerts to be generated:

```
preprocessor stream4: disable_evasion_alerts
```

Use of the Flow preprocessor is required for sfPortscan. Flow gives portscan direction in the case of connectionless protocols like ICMP and UDP. You should enable the Flow preprocessor in your `snort.conf` by using the following:

```
preprocessor flow: stats_interval 0 hash 2
```

The parameters you can use to configure the portscan module are:

1. **proto** <**protocol**>
   Available options:

- TCP
- UDP
- IGMP
- ip_proto
- all

2. **scan_type** <**scan_type**>

   Available options:

   - portscan
   - portsweep
   - decoy_portscan
   - distributed_portscan
   - all

3. **sense_level** <**level**>

   Available options:

   - low - "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 postives. 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, afterwhich this window is reset.
   - medium - "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 - "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 sfPortscan.

4. **watch_ip** <**ip1**|**ip2/cidr[:[port**|**port2-port3]]**>

   Defines which IPs, networks, and specific ports on those hosts to watch. The list is a comma seperated list of IP addresses, IP address using CIDR notation. Optionally, ports are specified after the IP address/CIDR using a colon and can be either a single port or a range denoted by a dash. IPs or networks not falling into this range are ignored if this option is used.

5. **ignore_scanners** <**ip_list**>

   Ignores the source of scan alerts. ip_list can be a comma seperated list of IP addresses or IP addresses using CIDR notation.

6. **ignore_scanned** <**ip_list**>

   Ignores the destination of scan alerts. ip_list can be a comma seperated list of IP addresses or IP addresses using CIDR notation.

7. **logfile** <**file**>

   This option will output portscan events to the file specified. If file does not contain a leading slash, this file will be placed in the Snort config dir.

Format

```
preprocessor sfportscan: proto <protocols> \
scan_type <portscan|portsweep|decoy_portscan|distributed_portscan|all>\
sense_level <low|medium|high> watch_ip <IP or IP/CIDR> ignore_scanners <IP list>\
ignore_scanned <IP list> logfile <path and filename>
```

```
preprocessor flow: stats_interval 0 hash 2
preprocessor sfportscan: proto { all } \
    scan_type { all } \
    sense_level { low }
```

Figure 2.6: sfPortscan Preprocessor Configuration

sfPortscan Alert Output

Unified Output In order to get all the portscan information logged with the alert, snort generates a pseudo-packet and uses the payload portion to store the additional portscan information of priority count, connection count, IP count, port count, IP range, and port range. The characteristics of the packet are:

```
Src/Dst MAC Addr == MACDAD
IP Protocol == 255
IP TTL == 0
```

Other than that, the packet looks like the IP portion of the packet that caused the portscan alert to be generated. This includes any IP options, etc. The payload and payload size of the packet are equal to the length of the additional portscan information that is logged. The size tends to be around 100 - 200 bytes.

Open port alerts differ from the other portscan alerts, because open port alerts utilize the tagged packet output system. This means that if an output system that doesn't print tagged packets is used, then the user won't see open port alerts. The open port information is stored in the IP payload and contains the port that is open.

The sfPortscan alert output was designed to work with unified packet logging, so it is possible to extend favorite Snort GUIs to display portscan alerts and the additional information in the IP payload using the above packet characteristics.

Log File Output Log file output is displayed in the following format, and explained further below:

```
Time: 09/08-15:07:31.603880
event_id: 2
192.168.169.3 -> 192.168.169.5 (portscan) TCP Filtered Portscan
Priority Count: 0
Connection Count: 200
IP Count: 2
Scanner IP Range: 192.168.169.3:192.168.169.4
Port/Proto Count: 200
Port/Proto Range: 20:47557
```

If there are open ports on the target, one or more additional tagged packet(s) will be appended:

```
Time: 09/08-15:07:31.603881
event_ref: 2
192.168.169.3 -> 192.168.169.5 (portscan) Open Port
Open Port: 38458
```

1. **Event_id/Event_ref**

   These fields are used to link an alert with the corresponding `Open Port` tagged packet

2. **Priority Count**

   `Priority Count` keeps track of bad responses (resets, unreachables). The higher the priority count, the more bad responses have been received.

3. **Connection Count**

   `Connection Count` lists how many connections are active on the hosts (src or dst). This is accurate for connection-based protocols, and is more of an estimate for others. Whether or not a portscan was filtered is determined here. High connection count and low priority count would indicate filtered (no response received from target).

4. **IP Count**

   IP Count keeps track of the last IP to contact a host, and increments the count if the next IP is different. For one-to-one scans, this is a low number. For active hosts this number will be high regardless, and one-to-one scans may appear as a distributed scan.

5. **Scanned/Scanner IP Range**

   This field changes depending on the type of alert. Portsweep (one-to-many) scans display the scanned IP range; Portscans (one-to-one) display the scanner IP.

6. **Port Count**

   Port Count keeps track of the last port contacted and increments this number when that changes. We use this count (along with IP Count) to determine the difference between one-to-one portscans and one-to-one decoys.

Tuning sfPortscan

The most important aspect in detecting portscans is tuning the detection engine for your network(s). Here are some tuning tips:

1. **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, sfPortscan 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. sfPortscan may not generate false positives for these types of hosts, but be aware when first tuning sfPortscan 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.

2. **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 sfPortscan 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.

3. **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 legimite 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).

4. **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 generate 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.

2.1.8 Telnet Decode

The `telnet_decode` preprocessor allows Snort to normalize Telnet control protocol characters from the session data. In Snort 1.9.0 and above, it accepts a list of ports to run on as arguments. Also in 1.9.0, it normalizes into a separate data buffer from the packet itself so that the raw data may be logged or examined with the `rawbytes` content modifier. See section 3.5.3.

By default, `telnet_decode` runs against traffic on ports 21, 23, 25, and 119.

Format

```
preprocessor telnet_decode: <ports>
```

⚠ NOTE

The `telnet_decode` preprocessor is being deprecated in the next release in favor of the FTP/Telnet preprocessor. See section 2.1.13.

2.1.9 RPC Decode

The rpc_decode preprocessor normalizes RPC multiple fragmented records into a single un-fragmented record. It does this by normalizing the packet into the packet buffer. If stream4 is enabled, it will only process client-side traffic. By default, it runs against traffic on ports 111 and 32771.

Table 2.5: RPC Decoder Options

Option	Description
`alert_fragments`	Alert on any fragmented RPC record.
`no_alert_multiple_requests`	Don't alert when there are multiple records in one packet.
`no_alert_large_fragments`	Don't alert when the sum of fragmented records exceeds one packet.
`no_alert_incomplete`	Don't alert when a single fragment record exceeds the size of one packet.

Format

```
preprocessor rpc_decode: <ports> [ alert_fragments ] \
   [no_alert_multiple_requests] [no_alert_large_fragments] \
   [no_alert_incomplete]
```

2.1.10 Performance Monitor

This preprocessor measures Snort's real-time and theoretical maximum performance. Whenever this preprocessor is turned on, it should have an output mode enabled, either "console" which prints statistics to the console window or "file" with a file name, where statistics get printed to the specified file name. By default, Snort's real-time statistics are processed. This includes:

- Time Stamp
- Drop Rate
- Mbits/Sec (wire) [duplicated below for easy comparison with other rates]
- Alerts/Sec
- K-Pkts/Sec (wire) [duplicated below for easy comparison with other rates]
- Avg Bytes/Pkt (wire) [duplicated below for easy comparison with other rates]
- Pat-Matched [percent of data received that Snort processes in pattern matching]
- Syns/Sec
- SynAcks/Sec
- New Sessions Cached/Sec
- Sessions Del fr Cache/Sec
- Current Cached Sessions
- Max Cached Sessions
- Stream Flushes/Sec
- Stream Session Cache Faults
- Stream Session Cache Timeouts
- New Frag Trackers/Sec
- Frag-Completes/Sec
- Frag-Inserts/Sec
- Frag-Deletes/Sec
- Frag-Auto Deletes/Sec [memory DoS protection]
- Frag-Flushes/Sec
- Frag-Current [number of current Frag Trackers]
- Frag-Max [max number of Frag Trackers at any time]
- Frag-Timeouts
- Frag-Faults

- Number of CPUs [*** Only if compiled with LINUX_SMP ***, the next three appear for each CPU]
- CPU usage (user)
- CPU usage (sys)
- CPU usage (Idle)
- Mbits/Sec (wire) [average mbits of total traffic]
- Mbits/Sec (ipfrag) [average mbits of IP fragmented traffic]
- Mbits/Sec (ipreass) [average mbits Snort injects after IP reassembly]
- Mbits/Sec (tcprebuilt) [average mbits Snort injects after stream4 reassembly]
- Mbits/Sec (applayer) [average mbits seen by rules and protocol decoders]
- Avg Bytes/Pkt (wire)
- Avg Bytes/Pkt (ipfrag)
- Avg Bytes/Pkt (ipreass)
- Avg Bytes/Pkt (tcprebuilt)
- Avg Bytes/Pkt (applayer)
- K-Pkts/Sec (wire)
- K-Pkts/Sec (ipfrag)
- K-Pkts/Sec (ipreass)
- K-Pkts/Sec (tcprebuilt)
- K-Pkts/Sec (applayer)
- Total Packets Received
- Total Packets Dropped (not processed)
- Total Packets Blocked (inline)

The following options can be used with the performance monitor:

- `flow` - Prints out statistics about the type of traffic and protocol distributions that Snort is seeing. This option can produce large amounts of output.
- `events` - Turns on event reporting. This prints out statistics as to the number of signatures that were matched by the setwise pattern matcher (*non-qualified events*) and the number of those matches that were verified with the signature flags (*qualified events*). This shows the user if there is a problem with the rule set that they are running.
- `max` - Turns on the theoretical maximum performance that Snort calculates given the processor speed and current performance. This is only valid for uniprocessor machines, since many operating systems don't keep accurate kernel statistics for multiple CPUs.
- `console` - Prints statistics at the console. This is enabled by default.
- `file` - Prints statistics in a comma-delimited format to the file that is specified. Not all statistics are output to this file. You may also use `snortfile` which will output into your defined Snort log directory. Both of these directives can be overridden on the command line with the `-Z` or `--perfmon-file` options.
- `pktcnt` - Adjusts the number of packets to process before checking for the time sample. This boosts performance, since checking the time sample reduces Snort's performance. By default, this is 10000.

- time - Represents the number of seconds between intervals.
- accumulate or reset - Defines which type of drop statistics are kept by the operating system. By default, accumulate is used.
- atexitonly - Dump stats for entire life of Snort.

Examples

```
preprocessor perfmonitor: time 30 events flow file stats.profile max \
    console pktcnt 10000
preprocessor perfmonitor: time 300 file /var/tmp/snortstat pktcnt 10000
```

2.1.11 HTTP Inspect

HTTP Inspect is a generic HTTP decoder for user applications. Given a data buffer, HTTP Inspect will decode the buffer, find HTTP fields, and normalize the fields. HTTP Inspect works on both client requests and server responses.

The current version of HTTP Inspect only handles stateless processing. This means that HTTP Inspect looks for HTTP fields on a packet-by-packet basis, and will be fooled if packets are not reassembled. This works fine when there is another module handling the reassembly, but there are limitations in analyzing the protocol. Future versions will have a stateful processing mode which will hook into various reassembly modules.

HTTP Inspect has a very "rich" user configuration. Users can configure individual HTTP servers with a variety of options, which should allow the user to emulate any type of web server. Within HTTP Inspect, there are two areas of configuration: global and server.

Global Configuration

The global configuration deals with configuration options that determine the global functioning of HTTP Inspect. The following example gives the generic global configuration format:

Format

```
preprocessor http_inspect: global \
                           iis_unicode_map <map_filename> \
                           codemap <integer> \
                           [detect_anomalous_servers] \
                           [proxy_alert]
```

You can only have a single global configuration, you'll get an error if you try otherwise.

Configuration

1. iis_unicode_map <map_filename> [codemap <integer>]

   This is the global iis_unicode_map file. The iis_unicode_map is a required configuration parameter. The map file can reside in the same directory as snort.conf or be specified via a fully-qualified path to the map file.

   The iis_unicode_map file is a Unicode codepoint map which tells HTTP Inspect which codepage to use when decoding Unicode characters. For US servers, the codemap is usually 1252.

   A Microsoft US Unicode codepoint map is provided in the Snort source etc directory by default. It is called unicode.map and should be used if no other codepoint map is available. A tool is supplied with Snort to generate custom Unicode maps--ms_unicode_generator.c, which is available at http://www.snort.org/dl/contrib/.

⚠NOTE

Remember that this configuration is for the global IIS Unicode map, individual servers can reference their own IIS Unicode map.

2. `detect_anomalous_servers`

   This global configuration option enables generic HTTP server traffic inspection on non-HTTP configured ports, and alerts if HTTP traffic is seen. Don't turn this on if you don't have a default server configuration that encompasses all of the HTTP server ports that your users might access. In the future, we want to limit this to specific networks so it's more useful, but for right now, this inspects all network traffic.

3. `proxy_alert`

   This enables global alerting on HTTP server proxy usage. By configuring HTTP Inspect servers and enabling `allow_proxy_use`, you will only receive proxy use alerts for web users that aren't using the configured proxies or are using a rogue proxy server.

   Please note that if users aren't required to configure web proxy use, then you may get a lot of proxy alerts. So, please only use this feature with traditional proxy environments. Blind firewall proxies don't count.

Example Global Configuration

```
preprocessor http_inspect: global iis_unicode_map unicode.map 1252
```

Server Configuration

There are two types of server configurations: default and by IP address.

Default This configuration supplies the default server configuration for any server that is not individually configured. Most of your web servers will most likely end up using the default configuration.

Example Default Configuration

```
preprocessor http_inspect_server: server default profile all ports { 80 }
```

Configuration by IP Address This format is very similar to "default", the only difference being that specific IPs can be configured.

Example IP Configuration

```
preprocessor http_inspect_server: server 10.1.1.1 profile all ports { 80 }
```

Server Configuration Options

Important: Some configuration options have an argument of 'yes' or 'no'. This argument specifies whether the user wants the configuration option to generate an HTTP Inspect alert or not. The 'yes/no' argument does not specify whether the configuration option itself is on or off, only the alerting functionality. In other words, whether set to 'yes' or 'no', HTTP normalization will still occur, and rules based on HTTP traffic will still trigger.

1. `profile <all|apache|iis>`

   Users can configure HTTP Inspect by using pre-defined HTTP server profiles. Profiles allow the user to easily configure the preprocessor for a certain type of server, but are not required for proper operation.

   There are three profiles available: all, apache, and iis.

1-A. `all`

The `all` profile is meant to normalize the URI using most of the common tricks available. We alert on the more serious forms of evasions. This is a great profile for detecting all types of attacks, regardless of the HTTP server. `profile all` sets the configuration options described in Table 2.6.

Table 2.6: Options for the "all" Profile

Option	Setting
flow_depth	300
chunk encoding	alert on chunks larger than 500000 bytes
iis_unicode_map	codepoint map in the global configuration
ascii decoding	on, alert off
multiple slash	on, alert off
directory normalization	on, alert off
apache whitespace	on, alert off
double decoding	on, alert on
%u decoding	on, alert on
bare byte decoding	on, alert on
iis unicode codepoints	on, alert on
iis backslash	on, alert off
iis delimiter	on, alert off
webroot	on, alert on
non_strict URL parsing	on
tab_uri_delimiter	is set

1-B. `apache`

The `apache` profile is used for Apache web servers. This differs from the `iis` profile by only accepting UTF-8 standard Unicode encoding and not accepting backslashes as legitimate slashes, like IIS does. Apache also accepts tabs as whitespace. `profile apache` sets the configuration options described in Table 2.7.

Table 2.7: Options for the `apache` Profile

Option	Setting
flow_depth	300
chunk encoding	alert on chunks larger than 500000 bytes
ascii decoding	on, alert off
multiple slash	on, alert off
directory normalization	on, alert off
webroot	on, alert on
apache whitespace	on, alert on
utf_8 encoding	on, alert off
non_strict url parsing	on
tab_uri_delimiter	is set

1-C. `iis`

The `iis` profile mimics IIS servers. So that means we use IIS Unicode codemaps for each server, %u encoding, bare-byte encoding, double decoding, backslashes, etc. `profile iis` sets the configuration options described in Table 2.8.

The default options used by HTTP Inspect do not use a profile and are described in Table 2.9.

Table 2.8: Options for the `iis` Profile

Option	Setting
flow_depth	300
chunk encoding	alert on chunks larger than 500000 bytes
iis_unicode_map	codepoint map in the global configuration
ascii decoding	on, alert off
multiple slash	on, alert off
directory normalization	on, alert off
webroot	on, alert on
double decoding	on, alert on
%u decoding	on, alert on
bare byte decoding	on, alert on
iis unicode codepoints	on, alert on
iis backslash	on, alert off
iis delimiter	on, alert on
apache whitespace	on, alert on
non_strict URL parsing	on

Table 2.9: Default HTTP Inspect Options

Option	Setting
port	80
flow_depth	300
chunk encoding	alert on chunks larger than 500000 bytes
ascii decoding	on, alert off
utf_8 encoding	on, alert off
multiple slash	on, alert off
directory normalization	on, alert off
webroot	on, alert on
iis backslash	on, alert off
apache whitespace	on, alert off
iis delimiter	on, alert off
non_strict URL parsing	on

Profiles must be specified as the first server option and cannot be combined with any other options except:

- ports
- iis_unicode_map
- allow_proxy_use
- flow_depth
- no_alerts
- inspect_uri_only
- oversize_dir_length

These options must be specified after the profile option.

Example

```
preprocessor http_inspect_server: server 1.1.1.1 profile all ports { 80 3128 }
```

2. ports {<port> [<port>< ... >]}

   This is how the user configures which ports to decode on the HTTP server. Encrypted traffic (SSL) cannot be decoded, so adding port 443 will only yield encoding false positives.

3. iis_unicode_map <map_filename> codemap <integer>

   The IIS Unicode map is generated by the program ms_unicode_generator.c. This program is located on the Snort.org web site at http://www.snort.org/dl/contrib/ directory. Executing this program generates a Unicode map for the system that it was run on. So, to get the specific Unicode mappings for an IIS web server, you run this program on that server and use that Unicode map in this configuration.

   When using this option, the user needs to specify the file that contains the IIS Unicode map and also specify the Unicode map to use. For US servers, this is usually 1252. But the ms_unicode_generator program tells you which codemap to use for you server; it's the ANSI code page. You can select the correct code page by looking at the available code pages that the ms_unicode_generator outputs.

4. flow_depth <integer>

   This specifies the amount of server response payload to inspect. This option significantly increases IDS performance because we are ignoring a large part of the network traffic (HTTP server response payloads). A small percentage of Snort rules are targeted at this traffic and a small flow_depth value may cause false negatives in some of these rules. Most of these rules target either the HTTP header, or the content that is likely to be in the first hundred or so bytes of non-header data. Headers are usually under 300 bytes long, but your mileage may vary.

   This value can be set from -1 to 1460. A value of -1 causes Snort to ignore all server side traffic for ports defined in ports. Inversely, a value of 0 causes Snort to inspect all HTTP server payloads defined in ports (note that this will likely slow down IDS performance). Values above 0 tell Snort the number of bytes to inspect in the first packet of the server response.

5. ascii <yes|no>

   The ascii decode option tells us whether to decode encoded ASCII chars, a.k.a %2f = /, %2e = ., etc. It is normal to see ASCII encoding usage in URLs, so it is recommended that you disable HTTP Inspect alerting for this option.

6. utf_8 <yes|no>

   The utf-8 decode option tells HTTP Inspect to decode standard UTF-8 Unicode sequences that are in the URI. This abides by the Unicode standard and only uses % encoding. Apache uses this standard, so for any Apache servers, make sure you have this option turned on. As for alerting, you may be interested in knowing when you have a UTF-8 encoded URI, but this will be prone to false positives as legitimate web clients use this type of encoding. When utf_8 is enabled, ASCII decoding is also enabled to enforce correct functioning.

7. `u_encode <yes|no>`

   This option emulates the IIS %u encoding scheme. How the %u encoding scheme works is as follows: the encoding scheme is started by a %u followed by 4 characters, like %uxxxx. The xxxx is a hex-encoded value that correlates to an IIS Unicode codepoint. This value can most definitely be ASCII. An ASCII character is encoded like %u002f = /, %u002e = ., etc. If no iis_unicode_map is specified before or after this option, the default codemap is used.

   You should alert on %u encodings, because we are not aware of any legitimate clients that use this encoding. So it is most likely someone trying to be covert.

8. `bare_byte <yes|no>`

   Bare byte encoding is an IIS trick that uses non-ASCII characters as valid values when decoding UTF-8 values. This is not in the HTTP standard, as all non-ASCII values have to be encoded with a %. Bare byte encoding allows the user to emulate an IIS server and interpret non-standard encodings correctly.

   The alert on this decoding should be enabled, because there are no legitimate clients that encode UTF-8 this way since it is non-standard.

9. `base36 <yes|no>`

   This is an option to decode base36 encoded chars. This option is based on of info from `http://www.yk.rim.or.jp/~shikap/patch/spp\_http\_decode.patch`.

   If %u encoding is enabled, this option will not work. You have to use the `base36` option with the `utf_8` option. Don't use the %u option, because base36 won't work. When `base36` is enabled, ASCII encoding is also enabled to enforce correct behavior.

10. `iis_unicode <yes|no>`

    The `iis_unicode` option turns on the Unicode codepoint mapping. If there is no iis_unicode_map option specified with the server config, `iis_unicode` uses the default codemap. The `iis_unicode` option handles the mapping of non-ASCII codepoints that the IIS server accepts and decodes normal UTF-8 requests.

    You should alert on the `iis_unicode option`, because it is seen mainly in attacks and evasion attempts. When `iis_unicode` is enabled, ASCII and UTF-8 decoding are also enabled to enforce correct decoding. To alert on UTF-8 decoding, you must enable also enable `utf_8 yes`.

11. `double_decode <yes|no>` The `double_decode` option is once again IIS-specific and emulates IIS functionality. How this works is that IIS does two passes through the request URI, doing decodes in each one. In the first pass, it seems that all types of iis encoding is done: utf-8 unicode, ascii, bare byte, and %u. In the second pass, the following encodings are done: ascii, bare byte, and %u. We leave out utf-8 because I think how this works is that the % encoded utf-8 is decoded to the Unicode byte in the first pass, and then UTF-8 is decoded in the second stage. Anyway, this is really complex and adds tons of different encodings for one character. When `double_decode` is enabled, so ASCII is also enabled to enforce correct decoding.

12. `non_rfc_char {<byte> [<byte ...>]}`

    This option lets users receive an alert if certain non-RFC chars are used in a request URI. For instance, a user may not want to see null bytes in the request URI and we can alert on that. Please use this option with care, because you could configure it to say, alert on all '/' or something like that. It's flexible, so be careful.

13. `multi_slash <yes|no>`

    This option normalizes multiple slashes in a row, so something like: "foo////////bar" get normalized to "foo/bar."

    If you want an alert when multiple slashes are seen, then configure with a `yes`; otherwise, use `no`.

14. `iis_backslash <yes|no>`

    Normalizes backslashes to slashes. This is again an IIS emulation. So a request URI of "/foo\bar" gets normalized to "/foo/bar."

15. `directory <yes|no>`

    This option normalizes directory traversals and self-referential directories.

    The directory:

```
/foo/fake\_dir/../bar
```

gets normalized to:

```
/foo/bar
```

The directory:

```
/foo/./bar
```

gets normalized to:

```
/foo/bar
```

If you want to configure an alert, specify yes, otherwise, specify no. This alert may give false positives, since some web sites refer to files using directory traversals.

16. apache_whitespace <yes|no>

This option deals with the non-RFC standard of using tab for a space delimiter. Apache uses this, so if the emulated web server is Apache, enable this option. Alerts on this option may be interesting, but may also be false positive prone.

17. iis_delimiter <yes|no>

This started out being IIS-specific, but Apache takes this non-standard delimiter was well. Since this is common, we always take this as standard since the most popular web servers accept it. But you can still get an alert on this option.

18. chunk_length <non-zero positive integer>

This option is an anomaly detector for abnormally large chunk sizes. This picks up the Apache chunk encoding exploits, and may also alert on HTTP tunneling that uses chunk encoding.

19. no_pipeline_req

This option turns HTTP pipeline decoding off, and is a performance enhancement if needed. By default, pipeline requests are inspected for attacks, but when this option is enabled, pipeline requests are not decoded and analyzed per HTTP protocol field. It is only inspected with the generic pattern matching.

20. non_strict

This option turns on non-strict URI parsing for the broken way in which Apache servers will decode a URI. Only use this option on servers that will accept URIs like this: ”get /index.html alsjdfk alsj lj aj la jsj s\n”. The non_strict option assumes the URI is between the first and second space even if there is no valid HTTP identifier after the second space.

21. allow_proxy_use

By specifying this keyword, the user is allowing proxy use on this server. This means that no alert will be generated if the proxy_alert global keyword has been used. If the proxy_alert keyword is not enabled, then this option does nothing. The allow_proxy_use keyword is just a way to suppress unauthorized proxy use for an authorized server.

22. no_alerts

This option turns off all alerts that are generated by the HTTP Inspect preprocessor module. This has no effect on HTTP rules in the rule set. No argument is specified.

23. oversize_dir_length <non-zero positive integer>

This option takes a non-zero positive integer as an argument. The argument specifies the max char directory length for URL directory. If a url directory is larger than this argument size, an alert is generated. A good argument value is 300 characters. This should limit the alerts to IDS evasion type attacks, like whisker -i 4.

24. inspect_uri_only

This is a performance optimization. When enabled, only the URI portion of HTTP requests will be inspected for attacks. As this field usually contains 90-95% of the web attacks, you'll catch most of the attacks. So if you need extra performance, enable this optimization. It's important to note that if this option is used without any uricontent rules, then no inspection will take place. This is obvious since the URI is only inspected with uricontent rules, and if there are none available, then there is nothing to inspect.

For example, if we have the following rule set:

```
alert tcp any any -> any 80 ( msg:"content"; content: "foo"; )
```

and the we inspect the following URI:

```
get /foo.htm http/1.0\r\n\r\n
```

No alert will be generated when inspect_uri_only is enabled. The inspect_uri_only configuration turns off all forms of detection except uricontent inspection.

25. webroot <yes|no>

This option generates an alert when a directory traversal traverses past the web server root directory. This generates much fewer false positives than the directory option, because it doesn't alert on directory traversals that stay within the web server directory structure. It only alerts when the directory traversals go past the web server root directory, which is associated with certain web attacks.

26. tab_uri_delimiter

This option turns on the use of the tab character (0x09) as a delimiter for a URI. Apache accepts tab as a delimiter; IIS does not. For IIS, a tab in the URI should be treated as any other character. Whether this option is on or not, a tab is treated as whitespace if a space character (0x20) precedes it. No argument is specified.

Examples

```
preprocessor http_inspect_server: server 10.1.1.1 \
                        ports { 80 3128 8080 } \
                        flow_depth 0 \
                        ascii no \
                        double_decode yes \
                        non_rfc_char { 0x00 } \
                        chunk_length 500000 \
                        non_strict \
                        no_alerts

preprocessor http_inspect_server: server default \
                        ports  { 80 3128 }  \
                        non_strict \
                        non_rfc_char  { 0x00 }  \
                        flow_depth 300  \
                        apache_whitespace yes \
                        directory no \
                        iis_backslash no \
                        u_encode yes \
                        ascii no \
                        chunk_length 500000 \
                        bare_byte yes \
                        double_decode yes \
                        iis_unicode yes \
                        iis_delimiter yes \
                        multi_slash no
```

```
preprocessor http_inspect_server: server default \
                        profile all \
                        ports { 80 8080 }
```

2.1.12 SMTP Preprocessor

The SMTP preprocessor is an SMTP decoder for user applications. Given a data buffer, SMTP will decode the buffer and find SMTP commands and responses. It will also mark the command, data header data body sections, and TLS data.

SMTP handles stateless and stateful processing. It saves state between individual packets. However maintaining correct state is dependent on the reassembly of the client side of the stream (ie, a loss of coherent stream data results in a loss of state).

Configuration

SMTP has the usual configuration items, such as `port` and `inspection_type`. Also, SMTP command lines can be normalized to remove extraneous spaces. TLS-encrypted traffic can be ignored, which improves performance. In addition, regular mail data can be ignored for an additional performance boost. Since so few (none in the current snort rule set) exploits are against mail data, this is relatively safe to do and can improve the performance of data inspection.

The configuration options are described below:

1. `ports { <port> [<port>] ... }`

   This specifies on what ports to check for SMTP data. Typically, this will include 25 and possibly 465, for encrypted SMTP.

2. `inspection_type <stateful | stateless>`

   Indicate whether to operate in stateful or stateless mode.

3. `normalize <all | none | cmds>`

   This turns on normalization. Normalization checks for more than one space character after a command. Space characters are defined as space (ASCII 0x20) or tab (ASCII 0x09).

   `all` checks all commands

   `none` turns off normalization for all commands.

   `cmds` just checks commands listed with the `normalize_cmds` parameter.

4. `ignore_data`

   Ignore data section of mail (except for mail headers) when processing rules.

5. `ignore_tls_data`

   Ignore TLS-encrypted data when processing rules.

6. `max_command_line_len <int>`

   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.

7. `max_header_line_len <int>`

   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.

8. `max_response_line_len <int>`

   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.

9. `alt_max_command_line_len <int> { <cmd> [<cmd>] }`

   Overrides `max_command_line_len` for specific commands.

10. `no_alerts`

    Turn off all alerts for this preprocessor.

11. `invalid_cmds { <Space-delimited list of commands> }`

    Alert if this command is sent from client side. Default is an empty list.

12. `valid_cmds { <Space-delimited list of commands> }`

    List of valid commands. We do not alert on commands in this list. Default is an empty list, but preprocessor has this list hard-coded: { ATRN AUTH BDAT DATA DEBUG EHLO EMAL ESAM ESND ESOM ETRN EVFY EXPN } { HELO HELP IDENT MAIL NOOP QUIT RCPT RSET SAML SOML SEND ONEX QUEU } { STARTTLS TICK TIME TURN TURNME VERB VRFY X-EXPS X-LINK2STATE } { XADR XAUTH XCIR XEXCH50 XGEN XLICENSE XQUE XSTA XTRN XUSR }

13. `alert_unknown_cmds`

    Alert if we don't recognize command. Default is off.

14. `normalize_cmds { <Space-delimited list of commands> }`

    Normalize this list of commands Default is { RCPT VRFY EXPN }.

15. `xlink2state { enable | disable [drop] }`

    Enable/disable xlink2state alert. Drop if alerted. Default is `enable`.

16. `print_cmds`

    List all commands understood by the preprocessor. This not normally printed out with the configuration because it can print so much data.

Example

```
preprocessor SMTP: \
  ports { 25 } \
  inspection_type stateful \
  normalize cmds \
  normalize_cmds { EXPN VRFY RCPT } \
  ignore_data \
  ignore_tls_data \
  max_command_line_len  512 \
  max_header_line_len   1024 \
  max_response_line_len 512 \
  no_alerts \
  alt_max_command_line_len 300 { RCPT } \
  invalid_cmds { } \
  valid_cmds { } \
  xlink2state { disable } \
  print_cmds
```

Default

```
preprocessor SMTP: \
  ports { 25 } \
  inspection_type stateful \
  normalize cmds \
  normalize_cmds { EXPN VRFY RCPT } \
```

```
alt_max_command_line_len 260 { MAIL } \
alt_max_command_line_len 300 { RCPT } \
alt_max_command_line_len 500 { HELP HELO ETRN } \
alt_max_command_line_len 255 { EXPN VRFY }
```

Note

`RCPT TO:` and `MAIL FROM:` are SMTP commands. For the preprocessor configuration, they are referred to as RCPT and MAIL, respectively. Within the code, the preprocessor actually maps RCPT and MAIL to the correct command name.

2.1.13 FTP/Telnet Preprocessor

FTP/Telnet is an improvement to the Telnet decoder and provides stateful inspection capability for both FTP and Telnet data streams. FTP/Telnet will decode the stream, identifying FTP commands and responses and Telnet escape sequences and normalize the fields. FTP/Telnet works on both client requests and server responses.

FTP/Telnet has the capability to handle stateless processing, meaning it only looks for information on a packet-by-packet basis.

The default is to run FTP/Telent in stateful inspection mode, meaning it looks for information and handles reassembled data correctly.

FTP/Telnet has a very "rich" user configuration, similar to that of HTTP Inspect (See 2.1.11). Users can configure individual FTP servers and clients with a variety of options, which should allow the user to emulate any type of FTP server or FTP Client. Within FTP/Telnet, there are four areas of configuration: Global, Telnet, FTP Client, and FTP Server.

Some configuration options have an argument of `yes` or `no`. This argument specifies whether the user wants the configuration option to generate a ftptelnet alert or not. The presence of the option indicates the option itself is on, while the `yes/no` argument applies to the alerting functionality associated with that option.

Global Configuration

The global configuration deals with configuration options that determine the global functioning of FTP/Telnet. The following example gives the generic global configuration format:

Format

```
preprocessor ftptelnet: global \
                        inspection_type stateful \
                        encrypted_traffic yes \
                        check_encrypted
```

You can only have a single global configuration, you'll get an error if you try otherwise. The FTP/Telnet global configuration must appear before the other three areas of configuration.

Configuration

1. `inspection_type`

   This indicates whether to operate in stateful or stateless mode.

2. `encrypted_traffic <yes|no>`

   This option enables detection and alerting on encrypted Telnet and FTP command channels.

   ⚠NOTE

   When `inspection_type` is in stateless mode, checks for encrypted traffic will occur on every packet, whereas in stateful mode, a particular session will be noted as encrypted and not inspected any further.

3. `check_encrypted`

   Instructs the the preprocessor to continue to check an encrypted session for a subsequent command to cease encryption.

Example Global Configuration

```
preprocessor ftptelnet: global inspection_type stateful encrypted_traffic no
```

Telnet Configuration

The telnet configuration deals with configuration options that determine the functioning of the Telnet portion of the preprocessor. The following example gives the generic telnet configuration format:

Format

```
preprocessor ftptelnet: telnet \
                        ports { 23 } \
                        normalize \
                        ayt_attack_thresh 6
```

There should only be a single telnet configuration, and subsequent instances will override previously set values.

Configuration

1. `ports {<port> [<port>< ... >]}`

   This is how the user configures which ports to decode as telnet traffic. SSH tunnels cannot be decoded, so adding port 22 will only yield false positives. Typically port 23 will be included.

2. `normalize`

   This option tells the preprocessor to normalize the telnet traffic by eliminating the telnet escape sequences. It functions similarly to its predecessor, the telnet_decode preprocessor. Rules written with 'raw' content options will ignore the normailzed buffer that is created when this option is in use.

3. `ayt_attach_thresh < number >`

   This option causes the preprocessor to alert when the number of consecutive telnet Are You There (AYT) commands reaches the number specified. It is only applicable when the mode is stateful.

Example Telnet Configuration

```
preprocessor ftptelnet: telnet ports { 23 } normalize ayt_attack_thresh 6
```

FTP Server Configuration

There are two types of FTP server configurations: default and by IP address.

Default This configuration supplies the default server configuration for any FTP server that is not individually configured. Most of your FTP servers will most likely end up using the default configuration.

Example Default FTP Server Configuration

```
preprocessor ftp_inspect_server: ftp server default ports { 21 }
```

Configuration by IP Address This format is very similar to "default", the only difference being that specific IPs can be configured.

Example IP specific FTP Server Configuration

```
preprocessor ftptelnet: ftp server 10.1.1.1 ports { 21 } ftp_cmds { XPWD XCWD }
```

FTP Server Configuration Options

1. `ports` {<port> [<port>< ... >]}

   This is how the user configures which ports to decode as FTP command channel traffic. Typically port 21 will be included.

2. `print_cmds`

   During initialization, this option causes the preprocessor to print the configuration for each of the FTP commands for this server.

3. `ftp_cmds` {*cmd*[*cmd*]}

   The preprocessor is configured to alert when it sees an FTP command that is not allowed by the server.

   This option specifies a list of additional commands allowed by this server, outside of the default FTP command set as specified in RFC 959. This may be used to allow the use of the 'X' commands identified in RFC 775, as well as any additional commands as needed.

   For example:

   ```
   ftp_cmds { XPWD XCWD XCUP XMKD XRMD }
   ```

4. `def_max_param_len` <number>

   This specifies the default maximum allowed parameter length for an FTP command. It can be used as a basic buffer overflow detection.

5. `alt_max_param_len` <number> {*cmd*[*cmd*]}

   This specifies the maximum allowed parameter length for the specified FTP command(s). It can be used as a more specific buffer overflow detection. For example the USER command – usernames may be no longer than 16 bytes, so the appropriate configuration would be:

   ```
   alt_max_param_len 16 { USER }
   ```

6. `chk_str_fmt` {*cmd*[*cmd*]}

   This option causes a check for string format attacks in the specified commands.

7. `cmd_validity cmd < fmt >`

   This option specifies the valid format for parameters of a givven command.

   fmt must be enclosed in <>'s and may contain the following:

Value		Description
int		Parameter must be an integer
number		Parameter must be an integer between 1 and 255
char _chars		Parameter must be a single character, one of _chars
date _datefmt		Parameter follows format specified, where:
#	Number	
C	Character	
[]	optional format enclosed	
|	OR	
{}	choice of options	
other	literal (ie, . + -)	
string		Parameter is a string (effectively unrestricted)
host_port		Parameter must be a host/port specified, per RFC 959
, |		One of choices enclosed within, separated by |
[]		Optional value enclosed within

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

```
cmd_validity MODE <char SBC>
cmd_validity STRU <char FRP>
cmd_validity ALLO < int [ char R int ] >
cmd_validity TYPE < { char AE [ char NTC ] | char I | char L [ number ] } >
cmd_validity PORT < host_port >
```

   A cmd_validity line can be used to override these defaults and/or add a check for other commands.

```
# This allows additional modes, including mode Z which allows for
# zip-style compression.
cmd_validity MODE < char ASBCZ >

# Allow for a date in the MDTM command.
cmd_validity MDTM < [ date nnnnnnnnnnnnnn[.n[n[n]]] ] string >
```

   MDTM is an off 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 (time format as specified in http://www.ietf.org/internet-drafts/draft-ietf-ftpext-mlst-16.txt)

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

```
cmd_validity MDTM < [ date nnnnnnnnnnnnnn[{+|-}n[n]] ] string >
```

8. `telnet_cmds <yes|no>`

   This option turns on detection and alerting when telnet escape sequences are seen on the FTP command channel. Injection of telnet escape sequences could be used as an evasion attempt on an FTP command channel.

9. `data_chan`

   This option causes the rest of snort (rules, other preprocessors) to ignore FTP data channel connections. Using this option means that **NO INSPECTION other than TCP state will be performed on FTP data transfers. It can be used to improve performance, especially with large file transfers from a trusted source. If your rule set includes virus-type rules, it is recommended that this option not be used.**

FTP Client Configuration

Similar to the FTP Server configuration, the FTP client configurations has two types: default, and by IP address.

Default This configuration supplies the default client configuration for any FTP client that is not individually configured. Most of your FTP clients will most likely end up using the default configuration.

Example Default FTP Client Configuration

```
preprocessor ftptelnet: ftp client default bounce no max_resp_len 200
```

Configuration by IP Address This format is very similar to "default", the only difference being that specific IPs can be configured.

Example IP specific FTP Client Configuration

```
preprocessor ftptelnet: ftp client 10.1.1.1 bounce yes max_resp_len 500
```

FTP Client Configuration Options

1. `max_resp_len <number>`

   This specifies the maximum allowed response length to an FTP command accepted by the client. It can be used as a basic buffer overflow detection.

2. `bounce <yes|no>`

   This option turns on detection and alerting of FTP bounce attacks. An FTP bounce attack occurs when the FTP PORT command is issued and the specified host does not match the host of the client.

3. `bounce_to < CIDR,[port|portlow,porthi] >`

   When the bounce option is turned on, this allows the PORT command to use the IP address (in CIDR format) and port (or inclusive port range) without generating an alert. It can be used to deal with proxied FTP connections where the FTP data channel is different from the client.

   A few examples:

   - Allow bounces to 192.162.1.1 port 20020 – ie, the use of `PORT 192,168,1,1,78,52`.

     ```
     bounce_to { 192.168.1.1,20020 }
     ```

   - Allow bounces to 192.162.1.1 ports 20020 through 20040 – ie, the use of `PORT 192,168,1,1,78,xx`, where xx is 52 through 72 inclusive.

     ```
     bounce_to { 192.168.1.1,20020,20040 }
     ```

   - Allow bounces to 192.162.1.1 port 20020 and 192.168.1.2 port 20030.

     ```
     bounce_to { 192.168.1.1,20020 192.168.1.2,20030}
     ```

4. `telnet_cmds <yes|no>`

   This option turns on detection and alerting when telnet escape sequences are seen on the FTP command channel. Injection of telnet escape sequences could be used as an evasion attempt on an FTP command channel.

Examples/Default Configuration from snort.conf

```
preprocessor ftp_telnet: global \
            encrypted_traffic yes \
            inspection_type stateful

preprocessor ftp_telnet_protocol: telnet \
            normalize \
            ayt_attack_thresh 200

# This is consistent with the FTP rules as of 18 Sept 2004.
# Set CWD to allow parameter length of 200
# MODE has an additional mode of Z (compressed)
# Check for string formats in USER & PASS commands
# Check MDTM commands that set modification time on the file.
preprocessor ftp_telnet_protocol: ftp server default \
            def_max_param_len 100 \
            alt_max_param_len 200 { CWD } \
            cmd_validity MODE < char ASBCZ > \
            cmd_validity MDTM < [ date nnnnnnnnnnnnnn[.n[n[n]]] ] string > \
            chk_str_fmt { USER PASS RNFR RNTO SITE MKD } \
            telnet_cmds yes \
            data_chan

preprocessor ftp_telnet_protocol: ftp client default \
            max_resp_len 256 \
            bounce yes \
            telnet_cmds yes
```

2.1.14 DNS

The DNS preprocessor decodes DNS Responses and can detect the following exploits: DNS Client RData Overflow, Obsolete Record Types, and Experimental Record Types.

DNS looks are DNS Response traffic over UDP and TCP and it requires Stream preprocessor to be enabled for TCP decoding.

Configuration

By default, all alerts are disabled and the preprocessor checks traffic on port 53.

The available configuration options are described below.

1. `ports` {<port> [<port>< ... >]}

   This option specifies the source ports that the DNS preprocessor should inspect traffic.

2. `enable_obsolete_types`

   Alert on Obsolete (per RFC 1035) Record Types

3. `enable_experimental_types`

   Alert on Experimental (per RFC 1035) Record Types

4. `enable_rdata_overflow`

   Check for DNS Client RData TXT Overflow

The DNS preprocessor does nothing if none of the 3 vulnerabilities it checks for are enabled. It will not operate on TCP sessions picked up midstream, and it will cease operation on a session if it loses state because of missing data (dropped packets).

Examples/Default Configuration from snort.conf

Looks for traffic on DNS server port 53. Check for the DNS Client RData overflow vulnerability. Do not alert on obsolete or experimental RData record types.

```
preprocessor dns: server_ports { 53 } \
                  enable_rdata_overflow
```

2.1.15 ASN.1 Detection

The asn.1 detection plugin decodes a packet or a portion of a packet, and looks for various malicious encodings.

The general configuration of the asn.1 rule option is as follows:

```
asn1: [keyword [argument]], . . .
```

Multiple keywords can be used in an 'asn1' option and the implied logic is boolean OR. So if any of the arguments evaluate as true, the whole option evaluates as true.

ASN.1 Keywords

The ASN.1 keywords provide programmatic detection capabilities as well as some more dynamic type detection. Most of the keywords don't have arguments as the detection is looking for non-configurable information. If a keyword does have an argument, the keyword is followed by a comma and the argument is the next word. If a keyword has multiple arguments, then a comman separates each.

1. `bitstring_overflow`

   The `bitstring_overflow` option detects invalid bitstring encodings that are known to be remotely exploitable.

2. `double_overflow`

   The double_overflow detects a double ASCII encoding that is larger than a standard buffer. This is known to be an exploitable function in Microsoft, but it is unknown at this time which services may be exploitable.

3. `oversize_length`

   This detection keyword compares ASN.1 type lengths with the supplied argument. The syntax looks like, "oversize_length 500". This means that if an ASN.1 type is greater than 500, then this keyword is evaluated as true. This keyword must have one argument which specifies the length to compare against.

4. `absolute_offset`

   This is the absolute offset from the beginning of the packet. For example, if you wanted to decode snmp packets, you would say "absolute_offset, 0". `absolute_offset` has one argument—the offset value. Offset may be positive or negative.

5. `relative_offset`

   This is the relative offset from the last content match or byte_test/jump. `relative_offset` has one argument—the offset number. So if you wanted to start decoding and ASN.1 sequence right after the content "foo", you would specify `'content:"foo"; asn1: bitstring_overflow, relative_offset, 0'`. Offset values may be positive or negative.

ASN.1 Examples

The following rules use ASN.1 decoding options:

```
alert udp any any -> any 161 (msg:"Oversize SNMP Length"; \
    asn1: oversize_length, 10000, absolute_offset, 0;)

alert tcp any any -> any 80 (msg:"ASN1 Relative Foo"; content:"foo"; \
    asn1: bitstring_overflow, print, relative_offset, 0;)
```

2.2 Event Thresholding

You can use event thresholding to reduce the number of logged alerts for noisy rules. This can be tuned to significantly reduce false alarms, and it can also be used to write a newer breed of rules. Thresholding commands limit the number of times a particular event is logged during a specified time interval. See Section 3.8 for more information.

2.3 Output Modules

Output modules are new as of version 1.6. They allow Snort to be much more flexible in the formatting and presentation of output to its users. The output modules are run when the alert or logging subsystems of Snort are called, after the preprocessors and detection engine. The format of the directives in the rules file is very similar to that of the preprocessors.

Multiple output plugins may be specified in the Snort configuration file. When multiple plugins of the same type (log, alert) are specified, they are stacked and called in sequence when an event occurs. As with the standard logging and alerting systems, output plugins send their data to /var/log/snort by default or to a user directed directory (using the -l command line switch).

Output modules are loaded at runtime by specifying the output keyword in the rules file:

```
output <name>: <options>

output alert_syslog: log_auth log_alert
```

Figure 2.7: Output Module Configuration Example

2.3.1 alert_syslog

This module sends alerts to the syslog facility (much like the -s command line switch). This module also allows the user to specify the logging facility and priority within the Snort rules file, giving users greater flexibility in logging alerts.

Available Keywords

Facilities

- log_auth
- log_authpriv
- log_daemon

- log_local0
- log_local1
- log_local2
- log_local3
- log_local4
- log_local5
- log_local6
- log_local7
- log_user

Priorities

- log_emerg
- log_alert
- log_crit
- log_err
- log_warning
- log_notice
- log_info
- log_debug

Options

- log_cons
- log_ndelay
- log_perror
- log_pid

Format

```
alert_syslog: <facility> <priority> <options>
```

As WIN32 does not run syslog servers locally by default, a hostname and port can be passed as options. The default host is 127.0.0.1. The default port is 514.

```
output alert_syslog: [host=<hostname[:<port>],] <facility> <priority> <options>
```

```
output alert_syslog: 10.1.1.1:514, <facility> <priority> <options>
```

Figure 2.8: Syslog Configuration Example

2.3.2 alert_fast

This will print Snort alerts in a quick one-line format to a specified output file. It is a faster alerting method than full alerts because it doesn't need to print all of the packet headers to the output file

Format

```
alert_fast: <output filename>
```

```
output alert_fast: alert.fast
```

Figure 2.9: Fast Alert Configuration

2.3.3 alert_full

This will print Snort alert messages with full packet headers. The alerts will be written in the default logging directory (/var/log/snort) or in the logging directory specified at the command line.

Inside the logging directory, a directory will be created per IP. These files will be decoded packet dumps of the packets that triggered the alerts. The creation of these files slows Snort down considerably. This output method is discouraged for all but the lightest traffic situations.

Format

```
alert_full: <output filename>
```

```
output alert_full: alert.full
```

Figure 2.10: Full Alert Configuration

2.3.4 alert_unixsock

Sets up a UNIX domain socket and sends alert reports to it. External programs/processes can listen in on this socket and receive Snort alert and packet data in real time. This is currently an experimental interface.

Format

```
alert_unixsock
```

```
output alert_unixsock
```

Figure 2.11: UNIXSock Alert Configuration

2.3.5 log_tcpdump

The log_tcpdump module logs packets to a tcpdump-formatted file. This is useful for performing post-process analysis on collected traffic with the vast number of tools that are available for examining tcpdump-formatted files. This module only takes a single argument: the name of the output file. Note that the file name will have the UNIX timestamp in seconds appended the file name. This is so that data from separate Snort runs can be kept distinct.

Format

```
log_tcpdump: <output filename>
```

```
output log_tcpdump: snort.log
```

Figure 2.12: Tcpdump Output Module Configuration Example

2.3.6 database

This module from Jed Pickel sends Snort data to a variety of SQL databases. More information on installing and configuring this module can be found on the [91]incident.org web page. The arguments to this plugin are the name of the database to be logged to and a parameter list. Parameters are specified with the format parameter = argument. see Figure 2.13 for example usage.

Format

```
database: <log | alert>, <database type>, <parameter list>
```

The following parameters are available:

host - Host to connect to. If a non-zero-length string is specified, TCP/IP communication is used. Without a host name, it will connect using a local UNIX domain socket.

port - Port number to connect to at the server host, or socket filename extension for UNIX-domain connections.

dbname - Database name

user - Database username for authentication

password - Password used if the database demands password authentication

sensor_name - Specify your own name for this Snort sensor. If you do not specify a name, one will be generated automatically

encoding - Because the packet payload and option data is binary, there is no one simple and portable way to store it in a database. Blobs are not used because they are not portable across databases. So i leave the encoding option to you. You can choose from the following options. Each has its own advantages and disadvantages:

hex (default) - Represent binary data as a hex string.

Storage requirements - 2x the size of the binary
Searchability - very good
Human readability - not readable unless you are a true geek, requires post processing

base64 - Represent binary data as a base64 string.

Storage requirements - ∼1.3x the size of the binary
Searchability - impossible without post processing

Human readability - not readable requires post processing

ascii - Represent binary data as an ASCII string. This is the only option where you will actually lose data. Non-ASCII Data is represented as a '.'. If you choose this option, then data for IP and TCP options will still be represented as hex because it does not make any sense for that data to be ASCII.

Storage requirements - slightly larger than the binary because some characters are escaped (&,<,>)

Searchability - very good for searching for a text string impossible if you want to search for binary

human readability - very good

detail - How much detailed data do you want to store? The options are:

full (default) - Log all details of a packet that caused an alert (including IP/TCP options and the payload)

fast - Log only a minimum amount of data. You severely limit the potential of some analysis applications if you choose this option, but this is still the best choice for some applications. The following fields are logged: `timestamp`, `signature`, `source ip`, `destination ip`, `source port`, `destination port`, `tcp flags`, and `protocol`)

Furthermore, there is a logging method and database type that must be defined. There are two logging types available, `log` and `alert`. Setting the type to log attaches the database logging functionality to the log facility within the program. If you set the type to log, the plugin will be called on the log output chain. Setting the type to alert attaches the plugin to the alert output chain within the program.

There are five database types available in the current version of the plugin. These are `mssql`, `mysql`, `postgresql`, `oracle`, and `odbc`. Set the type to match the database you are using.

The database output plugin does not have the ability to handle alerts that are generated by using the `tag` keyword. See section 3.7.5 for more details.

```
output database: log, mysql, dbname=snort user=snort host=localhost password=xyz
```

Figure 2.13: Database Output Plugin Configuration

2.3.7 csv

The csv output plugin allows alert data to be written in a format easily importable to a database. The plugin requires 2 arguments: a full pathname to a file and the output formatting option.

The list of formatting options is below. If the formatting option is default, the output is in the order the formatting option is listed.

- `timestamp`
- `sig_generator`
- `sig_id`
- `sig_rev`
- `msg`
- `proto`
- `src`
- `srcport`

- dst
- dstport
- ethsrc
- ethdst
- ethlen
- tcpflags
- tcpseq
- tcpack
- tcplen
- tcpwindow
- ttl
- tos
- id
- dgmlen
- iplen
- icmptype
- icmpcode
- icmpid
- icmpseq

Format

```
output alert_csv: <filename> <format>
```

```
output alert_csv: /var/log/alert.csv default

output alert_csv: /var/log/alert.csv timestamp, msg
```

Figure 2.14: CSV Output Configuration

2.3.8 unified

The unified output plugin is designed to be the fastest possible method of logging Snort events. The unified output plugin logs events in binary format, allowing another programs to handle complex logging mechanisms that would otherwise diminish the performance of Snort.

The name *unified* is a misnomer, as the unified output plugin creates two different files, an *alert* file, and a *log* file. The alert file contains the high-level details of an event (eg: IPs, protocol, port, message id). The log file contains the detailed packet information (a packet dump with the associated event ID). Both file types are written in a bimary format described in *spo_unified.h*.

⚠NOTE

Files have the file creation time (in Unix Epoch format) appended to each file when it is created.

Format

```
output alert_unified: <base file name> [, <limit <file size limit in MB>]
output log_unified: <base file name> [, <limit <file size limit in MB>]


output alert_unified: snort.alert, limit 128
output log_unified: snort.log, limit 128
```

Figure 2.15: Unified Configuration Example

2.3.9 alert_prelude

⚠NOTE

support to use alert_prelude is not built in by default. To use alert_prelude, snort must be built with the –enable-prelude arguement passed to ./configure.

The alert_prelude output plugin is used to log to a Prelude database. For more information on Prelude, see `http://www.prelude-ids.org/`.

format

```
output alert_prelude: profile <name of prelude profile>
    [, info <priority number for info priority alerts>]
    [, low <priority number for low priority alerts>]
    [, medium <priority number for medium priority alerts>]
    [, high <priority number for high priority alerts>]


output alert_prelude: profile snort, info 4, low 3, medium 2, high  1
```

Figure 2.16: alert_prelude configuration example

2.3.10 log null

Sometimes it is useful to be able to create rules that will alert to certain types of traffic but will not cause packet log entries. In Snort 1.8.2, the log_null plugin was introduced. This is equivalent to using the -n command line option but it is able to work within a ruletype.

Format

```
output log_null
```

```
output log_null  # like using snort -n

ruletype info {
    type alert
    output alert_fast: info.alert
    output log_null
}
```

Figure 2.17: Log Null Usage Example

2.4 Dynamic Modules

Dynamically loadable modules were introduced with Snort 2.6. They can be loaded via directives in `snort.conf` or via command-line options.

To use dynamic modules, Snort must be configured with the –enable-dynamicplugin flag.

2.4.1 Format

```
<directive> <parameters>
```

2.4.2 Directives

Table 2.10: Dynamic Directives

Directive	Syntax	Description
dynamicpreprocessor	dynamicpreprocessor [file <shared library path> | directory <directory of shared libraries>]	Tells snort to load the dynamic preprocessor shared library (if file is used) or all dynamic preprocessor shared libraries (if directory is used). Specify 'file', followed by the full or relative path to the shared library. Or, specify 'directory, followed by the full or relative path to a directory of preprocessor shared libraries. (Same effect as --dynamic-preprocessor-lib or --dynamic-preprocessor-lib-dir options). See chapter 5 for more information on dynamic preprocessor libraries.
dynamicengine	dynamicengine <shared library path>	Tells snort to load the dynamic engine shared library. Specify the full or relative path to the shared library. (Same effect as --dynamic-engine-lib option). See chapter 5 for more information on dynamic engine libraries.
dynamicdetection	dynamicdetection [file <shared library path> | directory <directory of shared libraries>]	Tells snort to load the dynamic detection rules shared library (if file is used) or all dynamic detection rules shared libraries (if directory is used). Specify 'file', followed by the full or relative path to the shared library. Or, specify 'directory, followed by the full or relative path to a directory of detection rules shared libraries. (Same effect as --dynamic-detection-lib or --dynamic-detection-lib-dir options). See chapter 5 for more information on dynamic detection rules libraries.

Chapter 3

Writing Snort Rules: How to Write Snort Rules and Keep Your Sanity

3.1 The Basics

Snort uses a simple, lightweight rules description language that is flexible and quite powerful. There are a number of simple guidelines to remember when developing Snort rules.

Most Snort rules are written in a single line. This was required in versions prior to 1.8. In current versions of Snort, rules may span multiple lines by adding a backslash \ to the end of the line.

Snort rules are divided into two logical sections, the rule header and the rule options. The rule header contains the rule's action, protocol, source and destination IP addresses and netmasks, and the source and destination ports information. The rule option section contains alert messages and information on which parts of the packet should be inspected to determine if the rule action should be taken.

Figure 3.1 illustrates a sample Snort rule.

```
alert tcp any any -> 192.168.1.0/24 111 (content:"|00 01 86 a5|"; msg:"mountd access";)
```

Figure 3.1: Sample Snort Rule

The text up to the first parenthesis is the rule header and the section enclosed in parenthesis contains the rule options. The words before the colons in the rule options section are called option *keywords*.

Note that the rule options section is not specifically required by any rule, they are just used for the sake of making tighter definitions of packets to collect or alert on (or drop, for that matter).

All of the elements in that make up a rule must be true for the indicated rule action to be taken. When taken together, the elements can be considered to form a logical AND statement. At the same time, the various rules in a Snort rules library file can be considered to form a large logical OR statement.

3.2 Rules Headers

3.2.1 Rule Actions

The rule header contains the information that defines the who, where, and what of a packet, as well as what to do in the event that a packet with all the attributes indicated in the rule should show up. The first item in a rule is the rule action. The rule action tells Snort what to do when it finds a packet that matches the rule criteria. There are 5 available default actions in Snort, alert, log, pass, activate, and dynamic. In addition, if you are running Snort in inline mode, you have additional options which include drop, reject, and sdrop.

1. alert - generate an alert using the selected alert method, and then log the packet
2. log - log the packet
3. pass - ignore the packet
4. activate - alert and then turn on another dynamic rule
5. dynamic - remain idle until activated by an activate rule , then act as a log rule
6. drop - make iptables drop the packet and log the packet
7. reject - make iptables drop the packet, log it, and then send a TCP reset if the protocol is TCP or an ICMP port unreachable message if the protocol is UDP.
8. sdrop - make iptables drop the packet but does not log it.

You can also define your own rule types and associate one or more output plugins with them. You can then use the rule types as actions in Snort rules.

This example will create a type that will log to just tcpdump:

```
ruletype suspicious
{
    type log
    output log_tcpdump: suspicious.log
}
```

This example will create a rule type that will log to syslog and a MySQL database:

```
ruletype redalert
{
      type alert
      output alert_syslog: LOG_AUTH LOG_ALERT
      output database: log, mysql, user=snort dbname=snort host=localhost
}
```

3.2.2 Protocols

The next field in a rule is the protocol. There are four protocols that Snort currently analyzes for suspicious behavior – TCP, UDP, ICMP, and IP. In the future there may be more, such as ARP, IGRP, GRE, OSPF, RIP, IPX, etc.

3.2.3 IP Addresses

The next portion of the rule header deals with the IP address and port information for a given rule. The keyword any may be used to define any address. Snort does not have a mechanism to provide host name lookup for the IP address fields in the rules file. The addresses are formed by a straight numeric IP address and a CIDR[3] block. The CIDR block indicates the netmask that should be applied to the rule's address and any incoming packets that are tested against the rule. A CIDR block mask of /24 indicates a Class C network, /16 a Class B network, and /32 indicates a specific machine address. For example, the address/CIDR combination 192.168.1.0/24 would signify the block of addresses from 192.168.1.1 to 192.168.1.255. Any rule that used this designation for, say, the destination address would match on any address in that range. The CIDR designations give us a nice short-hand way to designate large address spaces with just a few characters.

In Figure 3.1, the source IP address was set to match for any computer talking, and the destination address was set to match on the 192.168.1.0 Class C network.

There is an operator that can be applied to IP addresses, the negation operator. This operator tells Snort to match any IP address except the one indicated by the listed IP address. The negation operator is indicated with a !. For example, an easy modification to the initial example is to make it alert on any traffic that originates outside of the local net with the negation operator as shown in Figure 3.2.

```
alert tcp !192.168.1.0/24 any -> 192.168.1.0/24 111 \
     (content: "|00 01 86 a5|"; msg: "external mountd access";)
```

Figure 3.2: Example IP Address Negation Rule

This rule's IP addresses indicate any tcp packet with a source IP address not originating from the internal network and a destination address on the internal network.

You may also specify lists of IP addresses. An IP list is specified by enclosing a comma separated list of IP addresses and CIDR blocks within square brackets. For the time being, the IP list may not include spaces between the addresses. See Figure 3.3 for an example of an IP list in action.

```
alert tcp ![192.168.1.0/24,10.1.1.0/24] any -> \
     [192.168.1.0/24,10.1.1.0/24] 111 (content: "|00 01 86 a5|"; \
     msg: "external mountd access";)
```

Figure 3.3: IP Address Lists

3.2.4 Port Numbers

Port numbers may be specified in a number of ways, including any ports, static port definitions, ranges, and by negation. Any ports are a wildcard value, meaning literally any port. Static ports are indicated by a single port number, such as 111 for portmapper, 23 for telnet, or 80 for http, etc. Port ranges are indicated with the range operator :. The range operator may be applied in a number of ways to take on different meanings, such as in Figure 3.4.

Port negation is indicated by using the negation operator !. The negation operator may be applied against any of the other rule types (except any, which would translate to none, how Zen...). For example, if for some twisted reason you wanted to log everything except the X Windows ports, you could do something like the rule in Figure 3.5.

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

```
log tcp any any -> 192.168.1.0/24 :6000
```

log tcp traffic from any port going to ports less than or equal to 6000

```
log tcp any :1024 -> 192.168.1.0/24 500:
```

log tcp traffic from privileged ports less than or equal to 1024 going to ports greater than or equal to 500

Figure 3.4: Port Range Examples

```
log tcp any any -> 192.168.1.0/24 !6000:6010
```

Figure 3.5: Example of Port Negation

3.2.5 The Direction Operator

The direction operator -> indicates the orientation, or direction, of the traffic that the rule applies to. The IP address and port numbers on the left side of the direction operator is considered to be the traffic coming from the source host, and the address and port information on the right side of the operator is the destination host. There is also a bidirectional operator, which is indicated with a <> symbol. This tells Snort to consider the address/port pairs in either the source or destination orientation. This is handy for recording/analyzing both sides of a conversation, such as telnet or POP3 sessions. An example of the bidirectional operator being used to record both sides of a telnet session is shown in Figure 3.6.

Also, note that there is no <- operator. In Snort versions before 1.8.7, the direction operator did not have proper error checking and many people used an invalid token. The reason the <- does not exist is so that rules always read consistently.

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

Figure 3.6: Snort rules using the Bidirectional Operator

3.2.6 Activate/Dynamic Rules

Activate and Dynamic rules are being phased out in favor of a combination of tagging (3.7.5) and flowbits (3.6.10).

Activate/dynamic rule pairs give Snort a powerful capability. You can now have one rule activate another when it's action is performed for a set number of packets. This is very useful if you want to set Snort up to perform follow on recording when a specific rule goes off. Activate rules act just like alert rules, except they have a *required* option field: activates. Dynamic rules act just like log rules, but they have a different option field: activated_by. Dynamic rules have a second required field as well, count.

Activate rules are just like alerts but also tell Snort to add a rule when a specific network event occurs. Dynamic rules are just like log rules except are dynamically enabled when the activate rule id goes off.

Put 'em together and they look like Figure 3.7.

```
activate tcp !$HOME_NET any -> $HOME_NET 143 (flags: PA; \
     content: "|E8C0FFFFFF|/bin"; activates: 1;  \
     msg: "IMAP buffer overflow!";)
dynamic tcp !$HOME_NET any -> $HOME_NET 143 (activated_by: 1; count: 50;)
```

Figure 3.7: Activate/Dynamic Rule Example

These rules tell Snort to alert when it detects an IMAP buffer overflow and collect the next 50 packets headed for port 143 coming from outside $HOME_NET headed to $HOME_NET. If the buffer overflow happened and was successful, there's a very good possibility that useful data will be contained within the next 50 (or whatever) packets going to that same service port on the network, so there's value in collecting those packets for later analysis.

3.3 Rule Options

Rule options form the heart of Snort's intrusion detection engine, combining ease of use with power and flexibility. All Snort rule options are separated from each other using the semicolon (;) character. Rule option keywords are separated from their arguments with a colon (:) character.

There are four major categories of rule options.

meta-data These options provide information about the rule but do not have any affect during detection

payload These options all look for data inside the packet payload and can be inter-related

non-payload These options look for non-payload data

post-detection These options are rule specific triggers that happen after a rule has "fired."

3.4 Meta-Data Rule Options

3.4.1 msg

The msg rule option tells the logging and alerting engine the message to print along with a packet dump or to an alert. It is a simple text string that utilizes the \ as an escape character to indicate a discrete character that might otherwise confuse Snort's rules parser (such as the semi-colon ; character).

Format

```
msg: "<message text>";
```

3.4.2 reference

The reference keyword allows rules to include references to external attack identification systems. The plugin currently supports several specific systems as well as unique URLs. This plugin is to be used by output plugins to provide a link to additional information about the alert produced.

Make sure to also take a look at `http://www.snort.org/pub-bin/sigs-search.cgi/` for a system that is indexing descriptions of alerts based on of the sid (See Section 3.4.3).

Table 3.1: Supported Systems

System	URL Prefix
bugtraq	http://www.securityfocus.com/bid/
cve	http://cve.mitre.org/cgi-bin/cvename.cgi?name=
nessus	http://cgi.nessus.org/plugins/dump.php3?id=
arachnids	(currently down) http://www.whitehats.com/info/IDS
mcafee	http://vil.nai.com/vil/dispVirus.asp?virus_k=
url	http://

Format

```
reference: <id system>,<id>; [reference: <id system>,<id>;]
```

```
alert tcp any any -> any 7070 (msg:"IDS411/dos-realaudio"; \
   flags:AP; content:"|fff4 fffd 06|"; reference:arachnids,IDS411;)

alert tcp any any -> any 21 (msg:"IDS287/ftp-wuftp260-venglin-linux"; \
   flags:AP; content:"|31c031db 31c9b046 cd80 31c031db|"; \
   reference:arachnids,IDS287; reference:bugtraq,1387; \
   reference:cve,CAN-2000-1574;)
```

Figure 3.8: Reference Usage Examples

3.4.3 sid

The sid keyword is used to uniquely identify Snort rules. This information allows output plugins to identify rules easily. This option should be used with the rev keyword. (See section 3.4.4)

- <100 Reserved for future use
- 100-1,000,000 Rules included with the Snort distribution
- >1,000,000 Used for local rules

The file sid-msg.map contains a mapping of alert messages to Snort rule IDs. This information is useful when post-processing alert to map an ID to an alert message.

Format

```
sid: <snort rules id>;
```

Example

This example is a rule with the Snort Rule ID of 1000983.

```
alert tcp any any -> any 80 (content:"BOB"; sid:1000983; rev:1;)
```

3.4.4 rev

The sid keyword is used to uniquely identify revisions of Snort rules. Revisions, along with Snort rule id's, allow signatures and descriptions to be refined and replaced with updated information. This option should be used with the sid keyword. (See section 3.4.3)

Format

```
rev: <revision integer>
```

Example

This example is a rule with the Snort Rule Revision of 1.

```
alert tcp any any -> any 80 (content:"BOB"; sid:1000983; rev:1;)
```

3.4.5 classtype

The classtype keyword categorizes alerts to be attack classes. By using the and prioritized. The user can specify what priority each type of rule classification has. Rules that have a classification will have a default priority set.

Format

```
classtype: <class name>;
```

Rule classifications are defined in the `classification.config` file. The config file uses the following syntax:

```
config classification:  <class name>,<class description>,<default priority>
```

The standard classifications included with Snort are listed in Table 3.2. The standard classifications are ordered with 3 default priorities currently. A priority 1 is the most severe priority level of the default rule set and 4 is the least severe.

Table 3.2: Snort Default Classifications

Classtype	Description	Priority
attempted-admin	Attempted Administrator Privilege Gain	high
attempted-user	Attempted User Privilege Gain	high
shellcode-detect	Executable code was detected	high
successful-admin	Successful Administrator Privilege Gain	high
successful-user	Successful User Privilege Gain	high
trojan-activity	A Network Trojan was detected	high
unsuccessful-user	Unsuccessful User Privilege Gain	high
web-application-attack	Web Application Attack	high
attempted-dos	Attempted Denial of Service	medium
attempted-recon	Attempted Information Leak	medium
bad-unknown	Potentially Bad Traffic	medium
denial-of-service	Detection of a Denial of Service Attack	medium
misc-attack	Misc Attack	medium
non-standard-protocol	Detection of a non-standard protocol or event	medium
rpc-portmap-decode	Decode of an RPC Query	medium
successful-dos	Denial of Service	medium
successful-recon-largescale	Large Scale Information Leak	medium
successful-recon-limited	Information Leak	medium
suspicious-filename-detect	A suspicious filename was detected	medium
suspicious-login	An attempted login using a suspicious username was detected	medium
system-call-detect	A system call was detected	medium
unusual-client-port-connection	A client was using an unusual port	medium
web-application-activity	access to a potentially vulnerable web application	medium
icmp-event	Generic ICMP event	low
misc-activity	Misc activity	low
network-scan	Detection of a Network Scan	low
not-suspicious	Not Suspicious Traffic	low
protocol-command-decode	Generic Protocol Command Decode	low
string-detect	A suspicious string was detected	low
unknown	Unknown Traffic	low

```
alert tcp any any -> any 80 (msg:"EXPLOIT ntpdx overflow"; \
    dsize: >128; classtype:attempted-admin; priority:10 );

alert tcp any any -> any 25 (msg:"SMTP expn root"; flags:A+; \
   content:"expn root"; nocase; classtype:attempted-recon;)
```

Figure 3.9: Example Classtype Rules

Warnings

classtype uses classifications defined by the classification config option. The classifications used by the rules provided with Snort are defined in etc/classification.config

3.4.6 Priority

The priority tag assigns a severity level to rules. A classtype rule assigns a default priority that may be overridden with a priority rule. For an example in conjunction with a classification rule refer to Figure 3.9. For use by itself, see Figure 3.10

Format

```
priority: <priority integer>;
```

```
alert TCP any any -> any 80 (msg: "WEB-MISC phf attempt"; flags:A+; \
      content: "/cgi-bin/phf"; priority:10;)
```

Figure 3.10: Example Priority Rule

3.5 Payload Detection Rule Options

3.5.1 content

The content keyword is one of the more important features of Snort. It allows the user to set rules that search for specific content in the packet payload and trigger response based on that data. Whenever a content option pattern match is performed, the Boyer-Moore pattern match function is called and the (rather computationally expensive) test is performed against the packet contents. If data exactly matching the argument data string is contained anywhere within the packet's payload, the test is successful and the remainder of the rule option tests are performed. Be aware that this test is case sensitive.

The option data for the content keyword is somewhat complex; it can contain mixed text and binary data. The binary data is generally enclosed within the pipe (|) character and represented as bytecode. Bytecode represents binary data as hexadecimal numbers and is a good shorthand method for describing complex binary data. Figure 3.11 contains an example of mixed text and binary data in a Snort rule.

Note that multiple content rules can be specified in one rule. This allows rules to be tailored for less false positives.

If the rule is preceded by a `!`, the alert will be triggered on packets that do not contain this content. This is useful when writing rules that want to alert on packets that do not match a certain pattern

Also note that the following characters must be escaped inside a content rule:
`: ; \ "`

Format

```
content: [!] "<content string>";
```

Example

```
alert tcp any any -> any 139 (content:"|5c 00|P|00|I|00|P|00|E|00 5c|";)
```

Figure 3.11: Mixed Binary Bytecode and Text in a 'content' keyword

```
alert tcp any any -> any 80 (content:!"GET";)
```

Figure 3.12: Negation Example

Changing content behavior

The `content` keyword has a number of modifier keywords. The modifier keywords change how the previously specified content works. These modifier keywords are:

1. depth
2. offset
3. distance
4. within
5. nocase
6. rawbytes

3.5.2 nocase

The nocase keyword allows the rule writer to specify that the Snort should look for the specific pattern, ignoring case. nocase modifies the previous 'content' keyword in the rule.

Format

```
nocase;
```

Example

```
alert tcp any any -> any 21 (msg:"FTP ROOT"; content:"USER root"; nocase;)
```

Figure 3.13: Content rule with nocase modifier

3.5.3 rawbytes

The rawbytes keyword allows rules to look at the raw packet data, ignoring any decoding that was done by preprocessors. This acts as a modifier to the previous content 3.5.1 option.

format

```
rawbytes;
```

Example

This example tells the content pattern matcher to look at the raw traffic, instead of the decoded traffic provided by the Telnet decoder.

```
alert tcp any any -> any 21 (msg: "Telnet NOP"; content: "|FF F1|"; rawbytes;)
```

3.5.4 depth

The depth keyword allows the rule writer to specify how far into a packet Snort should search for the specified pattern. depth modifies the previous ‘content’ keyword in the rule.

A depth of 5 would tell Snort to only look look for the specified pattern within the first 5 bytes of the payload.

As the depth keyword is a modifier to the previous ‘content’ keyword, there must be a content in the rule before ‘depth’ is specified.

See Figure 3.14 for an example of a combined content, offset, and depth search rule.

Format

```
depth: <number>;
```

3.5.5 offset

The offset keyword allows the rule writer to specify where to start searching for a pattern within a packet. offset modifies the previous ’content’ keyword in the rule.

An offset of 5 would tell Snort to start looking for the specified pattern after the first 5 bytes of the payload.

As this keyword is a modifier to the previous ’content’ keyword, there must be a content in the rule before ’offset’ is specified.

See Figure 3.14 for an example of a combined content, offset, and depth search rule.

Format

```
offset: <number>;
```

```
alert tcp any any -> any 80 (content: "cgi-bin/phf"; offset:4; depth:20;)
```

Figure 3.14: Combined Content, Offset and Depth Rule. Skip the first 4 bytes, and look for cgi-bin/phf in the next 20 bytes

3.5.6 distance

The distance keyword allows the rule writer to specify how far into a packet Snort should ignore before starting to search for the specified pattern relative to the end of the previous pattern match.

This can be thought of as exactly the same thing as depth (See Section 3.5.5), except it is relative to the end of the last pattern match instead of the beginning of the packet.

Format

```
distance: <byte count>;
```

Example

The rule listed in Figure 3.15 maps to a regular expression of /ABCDE.{1}EFGH/.

```
alert tcp any any -> any any (content:"ABC"; content: "DEF"; distance:1;)
```

Figure 3.15: distance usage example

3.5.7 within

The within keyword is a content modifier that makes sure that at most N bytes are between pattern matches using the Content (See Section 3.5.1). It's designed to be used in conjunction with the distance (Section 3.5.6) rule option.

The rule listed in Figure 3.16 constrains the search to not go past 10 bytes past the ABCDE match.

Format

```
within: <byte count>;
```

Examples

```
alert tcp any any -> any any (content:"ABC"; content: "EFG"; within:10;)
```

Figure 3.16: within usage example

3.5.8 uricontent

The `uricontent` parameter in the Snort rule language searches the NORMALIZED request URI field. This means that if you are writing rules that include things that are normalized, such as %2f or directory traversals, these rules will not alert. The reason is that the things you are looking for are normalized out of the URI buffer.

For example, the URI:

```
/scripts/..%c0%af../winnt/system32/cmd.exe?/c+ver
```

will get normalized into:

```
/winnt/system32/cmd.exe?/c+ver
```

```
Another example, the URI:
\begin{verbatim} /cgi-bin/aaaaaaaaaaaaaaaaaaaaaaaaa/..%252fp%68f?
```

will get normalized into:

```
/cgi-bin/phf?
```

When writing a `uricontent` rule, write the content that you want to find in the context that the URI will be normalized. For example, if Snort normalizes directory traversals, do not include directory traversals.

You can write rules that look for the non-normalized content by using the content option. (See Section 3.5.1)

For a description of the parameters to this function, see the content rule options in Section 3.5.1.

This option works in conjunction with the HTTP Inspect preprocessor specified in Section 2.1.11.

Format

```
uricontent:[!]<content string>;
```

3.5.9 isdataat

Verify that the payload has data at a specified location, optionally looking for data relative to the end of the previous content match.

Format

```
isdataat:<int>[,relative];
```

Example

```
alert tcp any any -> any 111 (content:"PASS"; isdataat:50,relative; \
   content:!"|0a|"; distance:0;)
```

This rule looks for the string PASS exists in the packet, then verifies there is at least 50 bytes after the end of the string PASS, then verifies that there is not a newline character within 50 bytes of the end of the PASS string.

3.5.10 pcre

The pcre keyword allows rules to be written using perl compatible regular expressions. For more detail on what can be done via a pcre regular expression, check out the PCRE web site `http://www.pcre.org`

Format

```
pcre:[!]"(/<regex>/|m<delim><regex><delim>)[ismxAEGRUB]";
```

The post-re modifiers set compile time flags for the regular expression.

Table 3.3: Perl compatible modifiers

i	case insensitive
s	include newlines in the dot metacharacter
m	By default, the string is treated as one big line of characters. ^ and $ match at the beginning and ending of the string. When m is set, ^ and $ match immediately following or immediately before any newline in the buffer, as well as the very start and very end of the buffer.
x	whitespace data characters in the pattern are ignored except when escaped or inside a character class

Table 3.4: PCRE compatible modifiers

A	the pattern must match only at the start of the buffer (same as ^)
E	Set $ to match only at the end of the subject string. Without E, $ also matches immediately before the final character if it is a newline (but not before any other newlines).
G	Inverts the ”greediness” of the quantifiers so that they are not greedy by default, but become greedy if followed by ”?”.

Table 3.5: Snort specific modifiers

R	Match relative to the end of the last pattern match. (Similar to distance:0;)
U	Match the decoded URI buffers (Similar to `uricontent`)
B	Do not use the decoded buffers (Similar to rawbytes)

The modifiers R and B should not be used together.

Example

This example performs a case-insensitive search for the string BLAH in the payload.

```
alert ip any any -> any any (pcre:"/BLAH/i";)
```

Snort’s handling of multiple URIs with PCRE does not work as expected. PCRE when used without a `uricontent` only evaluates the first URI. In order to use pcre to inspect all URIs, you must use either a content or a uricontent.

3.5.11 byte_test

Test 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.

For a more detailed explanation, please read Section 3.11.5.

Format

```
byte_test: <bytes to convert>, [!]<operator>, <value>, <offset>  \
    [,relative] [,<endian>] [,<number type>, string];
```

Option	Description
bytes_to_convert	Number of bytes to pick up from the packet
operator	Operation to perform to test the value: • < - less than • > - greater than • = - equal • ! - not • & - bitwise AND • ^ bitwise OR
value	Value to test the converted value against
offset	Number of bytes into the payload to start processing
relative	Use an offset relative to last pattern match
endian	Endian type of the number being read: • big - Process data as big endian (default) • little - Process data as little endian
string	Data is stored in string format in packet
number type	Type of number being read: • hex - Converted string data is represented in hexadecimal • dec - Converted string data is represented in decimal • oct - Converted string data is represented in octal

Any of the operators can also include *!* to check if the operator is not true. If *!* is specified without an operator, then the operator is set to =.

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();}*

3.5.12 byte_jump

The byte_jump 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.

The byte_jump option does this by reading some number of bytes, convert them to their numeric representation, move that many bytes forward and set a pointer for later detection. This pointer is known as the detect offset end pointer, or doe_ptr.

For a more detailed explanation, please read Section 3.11.5.

Format

```
byte_jump: <bytes_to_convert>, <offset> \
        [,relative] [,multiplier <multiplier value>] [,big] [,little][,string]\
```

```
alert udp $EXTERNAL_NET any -> $HOME_NET any \
       (msg:"AMD procedure 7 plog overflow "; \
       content: "|00 04 93 F3|"; \
       content: "|00 00 00 07|"; distance: 4; within: 4; \
       byte_test: 4,>, 1000, 20, relative;)

alert tcp $EXTERNAL_NET any -> $HOME_NET any \
     (msg:"AMD procedure 7 plog overflow "; \
     content: "|00 04 93 F3|"; \
     content: "|00 00 00 07|"; distance: 4; within: 4; \
     byte_test: 4, >,1000, 20, relative;)

alert udp any any -> any 1234 \
     (byte_test: 4, =, 1234, 0, string, dec; \
     msg: "got 1234!";)

alert udp any any -> any 1235 \
     (byte_test: 3, =, 123, 0, string, dec; \
     msg: "got 123!";)

alert udp any any -> any 1236 \
     (byte_test: 2, =, 12, 0, string, dec; \
     msg: "got 12!";)

alert udp any any -> any 1237 \
     (byte_test: 10, =, 1234567890, 0, string, dec; \
     msg: "got 1234567890!";)

alert udp any any -> any 1238 \
     (byte_test: 8, =, 0xdeadbeef, 0, string, hex; \
     msg: "got DEADBEEF!";)
```

Figure 3.17: Byte Test Usage Example

```
[,hex] [,dec] [,oct] [,align] [,from_beginning];
```

Option	Description
bytes_to_convert	Number of bytes to pick up from the packet
offset	Number of bytes into the payload to start processing
relative	Use an offset relative to last pattern match
multiplier <value>	Multiply the number of calculated bytes by <value> and skip forward that number of bytes.
big	Process data as big endian (default)
little	Process data as little endian
string	Data is stored in string format in packet
hex	Converted string data is represented in hexadecimal
dec	Converted string data is represented in decimal
oct	Converted string data is represented in octal
align	Round the number of converted bytes up to the next 32-bit boundary
from_beginning	Skip forward from the beginning of the packet payload instead of from the current position in the packet.

```
alert udp any any -> any 32770:34000 (content: "|00 01 86 B8|"; \
              content: "|00 00 00 01|"; distance: 4; within: 4; \
              byte_jump: 4, 12, relative, align; \
              byte_test: 4, >, 900, 20, relative; \
              msg: "statd format string buffer overflow";)
```

Figure 3.18: byte jump Usage Example

3.5.13 ftpbounce

The ftpbounce keyword detects FTP bounce attacks.

Format

```
ftpbounce;
```

Example

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"FTP PORT bounce attempt"; \
flow:to_server,established; content:"PORT"; nocase; ftpbounce; pcre:"/^PORT/smi";\
classtype:misc-attack; sid:3441; rev:1;)
```

3.5.14 regex

The regex keyword has been superceded by PCRE. See Section 3.5.10.

3.5.15 content-list

The content-list keyword is broken and should not be used.

3.6 Non-Payload Detection Rule Options

3.6.1 fragoffset

The fragoffset keyword allows one to compare the IP fragment offset field against a decimal value. To catch all the first fragments of an IP session, you could use the fragbits keyword and look for the More fragments option in conjunction with a fragoffset of 0.

Format

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

```
alert ip any any -> any any \
      (msg: "First Fragment"; fragbits: M; fragoffset: 0;)
```

Figure 3.19: Fragoffset Usage Example

3.6.2 ttl

The ttl keyword is used to check the IP time-to-live value. This option keyword was intended for use in the detection of traceroute attempts.

Format

```
ttl:[[<number>-]><=]<number>;
```

Example

This example checks for a time-to-live value that is less than 3.

```
ttl:<3;
```

This example checks for a time-to-live value that between 3 and 5.

```
ttl:3-5;
```

3.6.3 tos

The tos keyword is used to check the IP TOS field for a specific value.

Format

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

Example

This example looks for a tos value that is not 4

```
tos:!4;
```

3.6.4 id

The id keyword is used to check the IP ID field for a specific value. Some tools (exploits, scanners and other odd programs) set this field specifically for various purposes, for example, the value 31337 is very popular with some hackers.

Format

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

Example

This example looks for the IP ID of 31337.

```
id:31337;
```

3.6.5 ipopts

The ipopts keyword is used to check if a specific IP option is present.

The following options may be checked:

rr - Record route

eol - End of list

nop - No op

ts - Time Stamp

sec - IP security option

lsrr - Loose source routing

ssrr - Strict source routing

satid - Stream identifier

any - any IP options are set

The most frequently watched for IP options are strict and loose source routing which aren't used in any widespread internet applications.

Format

```
ipopts:<rr|eol|nop|ts|sec|lsrr|ssrr|satid|any>;
```

Example

This example looks for the IP Option of Loose Source Routing.

```
ipopts:lsrr;
```

Warning

Only a single ipopts keyword may be specified per rule.

3.6.6 fragbits

The `fragbits` keyword is used to check if fragmentation and reserved bits are set in the IP header.

The following bits may be checked:

M - More Fragments

D - Don't Fragment

R - Reserved Bit

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

+ match on the specified bits, plus any others

***** match if any of the specified bits are set

! match if the specified bits are not set

Format

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

Example

This example checks if the More Fragments bit and the Do not Fragment bit are set.

```
fragbits:MD+;
```

3.6.7 dsize

The dsize keyword is used to test the packet payload size. This may be used to check for abnormally sized packets. In many cases, it is useful for detecting buffer overflows.

Format

```
dsize: [<>]<number>[<><number>];
```

Example

This example looks for a dsize that is between 300 and 400 bytes.

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

Warning

dsize will fail on stream rebuilt packets, regardless of the size of the payload.

3.6.8 flags

The flags keyword is used to check if specific TCP flag bits are present.

The following bits may be checked:

F - FIN (LSB in TCP Flags byte)

S - SYN

R - RST

P - PSH

A - ACK

U - URG

1 - Reserved bit 1 (MSB in TCP Flags byte)

2 - Reserved bit 2

0 - No TCP Flags Set

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

+ - match on the specified bits, plus any others

***** - match if any of the specified bits are set

! - match if the specified bits are not set

To handle writing rules for session initiation packets such as ECN where a SYN packet is sent with the previously reserved bits 1 and 2 set, an option mask may be specified. A rule could check for a flags value of S,12 if one wishes to find packets with just the syn bit, regardless of the values of the reserved bits.

Format

```
flags:[!|*|+]<FSRPAU120>[,<FSRPAU120>];
```

Example

This example checks if just the SYN and the FIN bits are set, ignoring reserved bit 1 and reserved bit 2.

```
alert tcp any any -> any any (flags:SF,12;)
```

3.6.9 flow

The flow rule option is used in conjunction with TCP stream reassembly (see Section 2.1.3). It allows rules to only apply to certain directions of the traffic flow.

This allows rules to only apply to clients or servers. This allows packets related to $HOME_NET clients viewing web pages to be distinguished from servers running the $HOME_NET.

The established keyword will replace the `flags: A+` used in many places to show established TCP connections.

Options

Option	Description
to_client	Trigger on server responses from A to B
to_server	Trigger on client requests from A to B
from_client	Trigger on client requests from A to B
from_server	Trigger on server responses from A to B
established	Trigger only on established TCP connections
stateless	Trigger regardless of the state of the stream processor (useful for packets that are designed to cause machines to crash)
no_stream	Do not trigger on rebuilt stream packets (useful for dsize and stream4)
only_stream	Only trigger on rebuilt stream packets

Format

```
flow: [(established|stateless)]
      [,(to_client|to_server|from_client|from_server)]
      [,(no_stream|only_stream)]
```

```
alert tcp !$HOME_NET any -> $HOME_NET 21 (msg:"cd incoming detected"; \
    flow:from_client; content:"CWD incoming"; nocase;)

alert tcp !$HOME_NET 0 -> $HOME_NET 0 (msg: "Port 0 TCP traffic"; \
    flow:stateless;)
```

Figure 3.20: Flow usage examples

3.6.10 flowbits

The flowbits rule option is used in conjunction with conversation tracking from the Flow preprocessor (see Section2.1.4). It allows rules to track states across transport protocol sessions. The flowbits option is most useful for TCP sessions, as it allows rules to generically track the state of an application protocol.

There are seven keywords associated with flowbits. Most of the options need a user-defined name for the specific state that is being checked. This string should be limited to any alphanumeric string including periods, dashes, and underscores.

Option	Description
set	Sets the specified state for the current flow.
unset	Unsets the specified state for the current flow.
toggle	Sets the specified state if the state is unset, otherwise unsets the state if the state is set.
isset	Checks if the specified state is set.
isnotset	Checks if the specified state is not set.
noalert	Cause the rule to not generate an alert, regardless of the rest of the detection options.

Format

```
flowbits: [set|unset|toggle|isset,reset,noalert][,<STATE_NAME>];
```

3.6.11 seq

The seq keyword is used to check for a specific TCP sequence number.

```
alert tcp any 143 -> any any (msg:"IMAP login";
  content:"OK LOGIN"; flowbits:set,logged_in;
  flowbits:noalert;)

alert tcp any any -> any 143 (msg:"IMAP LIST"; content:"LIST";
  flowbits:isset,logged_in;)
```

Figure 3.21: Flowbits Usage Examples

Format

```
seq:<number>;
```

Example

This example looks for a TCP sequence number of 0.

```
seq:0;
```

3.6.12 ack

The ack keyword is used to check for a specific TCP acknowledge number.

Format

```
ack: <number>;
```

Example

This example looks for a TCP acknowledge number of 0.

```
ack:0;
```

3.6.13 window

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

Format

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

Example

This example looks for a TCP window size of 55808.

```
window:55808;
```

3.6.14 itype

The itype keyword is used to check for a specific ICMP type value.

Format

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

Example

This example looks for an ICMP type greater than 30.

```
itype:>30;
```

3.6.15 icode

The itype keyword is used to check for a specific ICMP code value.

Format

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

Example

This example looks for an ICMP code greater than 30.

```
code:>30;
```

3.6.16 icmp_id

The itype keyword is used to check for a specific ICMP ID value.

This is useful because some covert channel programs use static ICMP fields when they communicate. This particular plugin was developed to detect the stacheldraht DDoS agent.

Format

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

Example

This example looks for an ICMP ID of 0.

```
icmp_id:0;
```

3.6.17 icmp_seq

The itype keyword is used to check for a specific ICMP sequence value.

This is useful because some covert channel programs use static ICMP fields when they communicate. This particular plugin was developed to detect the stacheldraht DDoS agent.

Format

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

Example

This example looks for an ICMP Sequence of 0.

```
icmp_seq:0;
```

3.6.18 rpc

The rpc keyword is used to check for a RPC application, version, and procedure numbers in SUNRPC CALL requests. Wildcards are valid for both version and procedure numbers by using '*';

Format

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

Example

The following example looks for an RPC portmap GETPORT request.

```
alert tcp any any -> any 111 (rpc: 100000,*,3;);
```

Warning

Because of the fast pattern matching engine, the RPC keyword is slower than looking for the RPC values by using normal content matching.

3.6.19 ip_proto

The ip_proto keyword allows checks against the IP protocol header. For a list of protocols that may be specified by name, see /etc/protocols.

Format

```
ip_proto:[!><] <name or number>;
```

Example

This example looks for IGMP traffic.

```
alert ip any any -> any any (ip_proto:igmp;)
```

3.6.20 sameip

The sameip keyword allows rules to check if the source ip is the same as the destination IP.

Format

```
sameip;
```

Example

This example looks for any traffic where the Source IP and the Destination IP is the same.

```
alert ip any any -> any any (sampeip;)
```

3.7 Post-Detection Rule Options

3.7.1 logto

The logto option tells Snort to log all packets that trigger this rule to a special output log file. This is especially handy for combining data from things like NMAP activity, HTTP CGI scans, etc. It should be noted that this option does not work when Snort is in binary logging mode.

Format

```
logto:"filename";
```

3.7.2 session

The session keyword is built to extract user data from TCP Sessions. There are many cases where seeing what users are typing in telnet, rlogin, ftp, or even web sessions is very useful.

There are two available argument keywords for the session rule option, printable or all. The printable keyword only prints out data that the user would normally see or be able to type.

The all keyword substitutes non-printable characters with their hexadecimal equivalents.

Format

```
session: [printable|all];
```

Example

The following example logs all printable strings in a telnet packet.

```
log tcp any any <> any 23 (session:printable;)
```

Warnings

Using the session keyword can slow Snort down considerably, so it should not be used in heavy load situations. The session keyword is best suited for post-processing binary (pcap) log files.

3.7.3 resp

The resp keyword is used attempt to close sessions when an alert is triggered. In Snort, this is called flexible response.

Flexible Response supports the following mechanisms for attempting to close sessions:

Option	Description
rst_snd	Send TCP-RST packets to the sending socket
rst_rcv	Send TCP-RST packets to the receiving socket
rst_all	Send TCP_RST packets in both directions
icmp_net	Send a ICMP_NET_UNREACH to the sender
icmp_host	Send a ICMP_HOST_UNREACH to the sender
icmp_port	Send a ICMP_PORT_UNREACH to the sender
icmp_all	Send all above ICMP packets to the sender

These options can be combined to send multiple responses to the target host.

Format

```
resp: <resp_mechanism>[,<resp_mechanism>[,<resp_mechanism>]];
```

Warnings

This functionality is not built in by default. Use the – –enable-flexresp flag to configure when building Snort to enable this functionality.

Be very careful when using Flexible Response. It is quite easy to get Snort into an infinite loop by defining a rule such as:

```
alert tcp any any -> any any (resp:rst_all;)
```

It is easy to be fooled into interfering with normal network traffic as well.

Example

The following example attempts to reset any TCP connection to port 1524.

```
alert tcp any any -> any 1524 (flags:S; resp:rst_all;)
```

3.7.4 react

This keyword implements an ability for users to react to traffic that matches a Snort rule. The basic reaction is blocking interesting sites users want to access: New York Times, slashdot, or something really important - napster and porn sites. The React code allows Snort to actively close offending connections and/or send a visible notice to the browser. The notice may include your own comment. The following arguments (basic modifiers) are valid for this option:

- block - close connection and send the visible notice
- warn - send the visible, warning notice (will be available soon)

The basic argument may be combined with the following arguments (additional modifiers):

- msg - include the msg option text into the blocking visible notice
- proxy: <port_nr> - use the proxy port to send the visible notice (will be available soon)

Multiple additional arguments are separated by a comma. The react keyword should be placed as the last one in the option list.

```
alert tcp any any <> 192.168.1.0/24 80 (content: "bad.htm"; \
    msg: "Not for children!"; react: block, msg;)
```

Figure 3.22: React Usage Example

Format

```
react: <react_basic_modifier[, react_additional_modifier]>;
```

Warnings

React functionality is not built in by default. This code is currently bundled under Flexible Response, so enabling Flexible Response (–enable-flexresp) will also enable React.

Be very careful when using react. Causing a network traffic generation loop is very easy to do with this functionality.

3.7.5 tag

The tag keyword allow rules to log more than just the single packet that triggered the rule. Once a rule is triggered, additional traffic involving the source and/or destination host is *tagged*. Tagged traffic is logged to allow analysis of response codes and post-attack traffic. *tagged* alerts will be sent to the same output plugins as the original alert, but it is the responsibility of the output plugin to properly handle these special alerts. Currently, the database output plugin, described in Section 2.3.6, does not properly handle *tagged* alerts.

Format

```
tag: <type>, <count>, <metric>, [direction]
```

`type`

- `session` - Log packets in the session that set off the rule
- `host` - Log packets from the host that caused the tag to activate (uses [direction] modifier)

`count` - Count is specified as a number of units. Units are specified in the <metric> field.

`metric`

- `packets` - Tag the host/session for <count> packets
- `seconds` - Tag the host/session for <count> seconds

Note, any packets that generate an alert will not be tagged. For example, it may seem that the following rule will tag the first 600 seconds of any packet involving 10.1.1.1.

```
alert tcp any any <> any 10.1.1.1 (tag:host,600,seconds,src;)
```

However, since the rule will fire on every packet involving 10.1.1.1, no packets will get tagged. The *flowbits* option would be useful here.

```
alert tcp any any <> any 10.1.1.1 (flowbits:isnotset,tagged;
    flowbits:set,tagged; tag:host,600,seconds,src;)
```

Example

This example logs the first 10 seconds of any telnet session.

```
alert tcp any any -> any 23 (flags:s,12; tag:session,10,seconds;)
```

3.8 Event Thresholding

Event thresholding can be used to reduce the number of logged alerts for noisy rules. This can be tuned to significantly reduce false alarms, and it can also be used to write a newer breed of rules. Thresholding commands limit the number of times a particular event is logged during a specified time interval.

There are 3 types of thresholding:

- `limit`

  Alerts on the 1st *m* events during the time interval, then ignores events for the rest of the time interval.

- `threshold`

  Alerts every *m* times we see this event during the time interval.

- `both`

  Alerts once per time interval after seeing *m* occurrences of the event, then ignores any additional events during the time interval.

Thresholding commands can be included as part of a rule, or you can use standalone threshold commands that reference the generator and SID they are applied to. There is no functional difference between adding a threshold to a rule, or using a separate threshold command applied to the same rule. There is a logical difference. Some rules may only make sense with a threshold. These should incorporate the threshold command into the rule. For instance, a rule for detecting a too many login password attempts may require more than 5 attempts. This can be done using the 'limit' type of threshold command. It makes sense that the threshold feature is an integral part of this rule.

In order for rule thresholds to apply properly, these rules must contain a SID.

Only one threshold may be applied to any given generator and SID pair. If more than one threshold is applied to a generator and SID pair, Snort will terminate with an error while reading the configuration information.

3.8.1 Standalone Options

This format supports 6 threshold options as described in Table 3.6—all are required.

Table 3.6: Standalone Options

Option	Arguments
`gen_id`	<generator ID>
`sig_id`	<Snort signature ID>
`type`	`limit`, `threshold`, or `both`
`track`	`by_src` or `by_dst`
`count`	<number of events>
`seconds`	<time period over which count is accrued>

3.8.2 Standalone Format

```
threshold gen_id <gen-id>, sig_id <sig-id>, \
    type <limit|threshold|both>, \
    track <by_src|by_dst>, count <s>, seconds <m>
```

3.8.3 Rule Keyword Format

This format supports 4 threshold options as described in Table 3.7—all are required.

Table 3.7: Rule Keyword Options

Option	Arguments
type	limit, threshold, or both
track	by_src or by_dst
count	<number of events>
seconds	<time period over which count is accrued>

3.8.4 Rule Keyword Format

```
threshold: type <limit|threshold|both>, track <by_src|by_dst>, \
     count <n>, seconds <m>;
```

For either standalone or rule format, all tracking is by src or by dst ip, ports or anything else are not tracked.

Thresholding can also be used globally, this allows you to specify a threshold for every rule. Standard thresholding tests are applied first to an event, if they do not block a rule from being logged, and then the global thresholding test is applied—thresholds in a rule will override a global threshold. Global thresholds do not override what's in a signature or a more specific stand-alone threshold.

The global threshold options are the same as the standard threshold options with the exception of the 'sig_id' field. The sig_id field must be set to 0 to indicate that this threshold command applies to all sig_id values with the specified gen_id. To apply the same threshold to all gen_id's at the same time, and with just one command specify a value of gen_id=0.

The format for global threshold commands is as such:

```
threshold gen_id <gen-id>, sig_id 0, \
    type <limit|threshold|both>, \
    track <by_src|by_dst>, \
    count <n>, \
    seconds <m>
```

This applies a threshold to every event from <gen-id>.

or

```
threshold gen_id 0 , sig_id 0, \
    type <limit|threshold|both>, \
    track <by_src|by_dst>, \
    count <n>, \
    seconds <m>
```

This applies a threshold to every event from every gen-id.

3.8.5 Examples

Standalone Thresholds

Limit logging to 1 event per 60 seconds:

```
threshold gen_id 1, sig_id 1851, \
    type limit, track by_src,  \
    count 1, seconds 60
```

Limit logging to every 3rd event:

```
threshold gen_id 1, sig_id 1852,  \
    type threshold, track by_src, \
    count 3, seconds 60
```

Limit logging to just 1 event per 60 seconds, but only if we exceed 30 events in 60 seconds:

```
threshold gen_id 1, sig_id 1853, \
    type both, track by_src, \
    count 30, seconds 60
```

Rule Thresholds

This rule logs the first event of this SID every 60 seconds.

```
alert tcp $external_net any -> $http_servers $http_ports \
    (msg:"web-misc robots.txt access"; flow:to_server, established; \
    uricontent:"/robots.txt"; nocase; reference:nessus,10302; \
    classtype:web-application-activity; threshold: type limit, track \
    by_src, count 1 , seconds 60 ; sid:1000852; rev:1;)
```

This rule logs every 10th event on this SID during a 60 second interval. So if less than 10 events occur in 60 seconds, nothing gets logged. Once an event is logged, a new time period starts for type=threshold.

```
alert tcp $external_net any -> $http_servers $http_ports \
    (msg:"web-misc robots.txt access"; flow:to_server, established; \
    uricontent:"/robots.txt"; nocase; reference:nessus,10302; \
    classtype:web-application-activity; threshold: type threshold, \
    track by_dst, count 10 , seconds 60 ; sid:1000852; rev:1;)
```

This rule logs at most one event every 60 seconds if at least 10 events on this SID are fired.

```
alert tcp $external_net any -> $http_servers $http_ports \
    (msg:"web-misc robots.txt access"; flow:to_server, established; \
    uricontent:"/robots.txt"; nocase; reference:nessus,10302; \
    classtype:web-application-activity; threshold: type both , track \
    by_dst, count 10 , seconds 60 ; sid:1000852; rev:1;)
```

Global Thresholds

Limit to logging 1 event per 60 seconds per IP triggering each rule (rule gen_id is 1):

```
threshold gen_id 1, sig_id 0, type limit, track by_src, count 1, seconds 60
```

Limit to logging 1 event per 60 seconds per IP, triggering each rule for each event generator:

```
threshold gen_id 0, sig_id 0, type limit, track by_src, count 1, seconds 60
```

Events in Snort are generated in the usual way, thresholding is handled as part of the output system. Read gen-msg.map for details on gen ids.

Users can also configure a memcap for threshold with a “config:” option:

```
config threshold: memcap <bytes>
```

3.9 Event Suppression

Event suppression stops specified events from firing without removing the rule from the rule base. Suppression uses a CIDR block notation to select specific networks and users for suppression. Suppression tests are performed prior to either standard or global thresholding tests.

Suppression commands are standalone commands that reference generators, SIDs, and IP addresses via a CIDR block. This allows a rule to be completely suppressed, or suppressed when the causative traffic is going to or coming from a specific IP or group of IP addresses.

You may apply multiple suppression commands to a SID. You may also combine one threshold command and several suppression commands to the same SID.

3.9.1 Format

The suppress command supports either 2 or 4 options, as described in Table 3.8.

Table 3.8: Suppression Options

Option	Argument	Required?
gen_id	<generator id>	required
sig_id	<Snort signature id>	required
track	by_src or by_dst	optional, requires ip
ip	ip[/mask]	optional, requires track

```
suppress gen_id <gen-id>, sig_id <sig-id>, \
    track <by_src|by_dst>, ip <ip|mask-bits>
```

3.9.2 Examples

Suppress this event completely:

```
suppress gen_id 1, sig_id 1852:
```

Suppress this event from this IP:

```
suppress gen_id 1, sig_id 1852, track by_src, ip 10.1.1.54
```

Suppress this event to this CIDR block:

```
suppress gen_id 1, sig_id 1852, track by_dst, ip 10.1.1.0/24
```

3.10 Snort Multi-Event Logging (Event Queue)

Snort supports logging multiple events per packet/stream that are prioritized with different insertion methods, such as max content length or event ordering using the event queue.

The general configuration of the event queue is as follows:

```
config event_queue: [max_events [size]] [log [size]] [order_events [TYPE]]
```

3.10.1 Event Queue Configuration Options

There are three configuration options to the configuration parameter 'event_queue'.

1. `max_queue`

   This determines the maximum size of the event queue. For example, if the event queue has a max size of 8, only 8 events will be stored for a single packet or stream.

   The default value is 8.

2. `log`

   This determines the number of events to log for a given packet or stream. You can't log more than the max_event number that was specified.

   The default value is 3.

3. `order_events`

   This argument determines the way that the incoming events are ordered. We currently have two different methods:

   - `priority` - The highest priority (1 being the highest) events are ordered first.
   - `content_length` - Rules are ordered before decode or preprocessor alerts, and rules that have a longer content are ordered before rules with shorter contents.

   The method in which events are ordered does not affect rule types such as pass, alert, log, etc.

   The default value is content_length.

3.10.2 Event Queue Configuration Examples

The default configuration:

```
config event_queue: max_queue 8 log 3 order_events content_length
```

Example of a reconfigured event queue:

```
config event_queue: max_queue 10 log 3 order_events content_length
```

Use the default event queue values, but change event order:

```
config event_queue: order_events priority
```

Use the default event queue values but change the number of logged events:

```
config event_queue: log 2
```

3.11 Writing Good Rules

There are some general concepts to keep in mind when developing Snort rules to maximize efficiency and speed.

3.11.1 Content Matching

The 2.0 detection engine changes the way Snort works slightly by having the first phase be a setwise pattern match. The longer a content option is, the more *exact* the match. Rules without *content* (or *uricontent*) slow the entire system down.

While some detection options, such as *pcre* and *byte_test*, perform detection in the payload section of the packet, they do not use the setwise pattern matching engine. If at all possible, try and have at least one *content* option if at all possible.

3.11.2 Catch the Vulnerability, Not the Exploit

Try to write rules that target the vulnerability, instead of a specific exploit.

For example, look for a the vulnerable command with an argument that is too large, instead of shellcode that binds a shell.

By writing rules for the vulnerability, the rule is less vulnerable to evasion when an attacker changes the exploit slightly.

3.11.3 Catch the Oddities of the Protocol in the Rule

Many services typically send the commands in upper case letters. FTP is a good example. In FTP, to send the username, the client sends:

```
user username_here
```

A simple rule to look for FTP root login attempts could be:

```
alert tcp any any -> any any 21 (content:"user root";)
```

While it may *seem* trivial to write a rule that looks for the username root, a good rule will handle all of the odd things that the protocol might handle when accepting the user command.

For example, each of the following are accepted by most FTP servers:

```
user root
user root
user root
user     root
user<tab>root
```

To handle all of the cases that the FTP server might handle, the rule needs more smarts than a simple string match.

A good rule that looks for root login on ftp would be:

```
alert tcp any any -> any 21 (flow:to_server,established; content:"root";
   pcre:"/user\s+root/i";)
```

There are a few important things to note in this rule:

- The rule has a *flow* option, verifying this is traffic going to the server on an enstablished session.
- The rule has a *content* option, looking for *root*, which is the longest, most unique string in the attack. This option is added to allow Snort's setwise pattern match detection engine to give Snort a boost in speed.
- The rule has a *pcre* option, looking for user, followed at least one space character (which includes tab), followed by root, ignoring case.

3.11.4 Optimizing Rules

The content matching portion of the detection engine has recursion to handle a few evasion cases. Rules that are not properly written can cause Snort to waste time duplicating checks.

The way the recursion works now is if a pattern matches, and if any of the detection options after that pattern fail, then look for the pattern again after where it was found the previous time. Repeat until the pattern is not found again or the opt functions all succeed.

On first read, that may not sound like a smart idea, but it is needed. For example, take the following rule:

```
alert ip any any -> any any (content:"a"; content:"b"; within:1;)
```

This rule would look for "a", immediately followed by "b". Without recursion, the payload "aab" would fail, even though it is obvious that the payload "aab" has "a" immediately followed by "b", because the first "a" is not immediately followed by "b".

While recursion is important for detection, the recursion implementation is not very smart.

For example, the following rule options are not optimized:

```
content:"|13|"; dsize:1;
```

By looking at this rule snippit, it is obvious the rule looks for a packet with a single byte of 0x13. However, because of recursion, a packet with 1024 bytes of 0x13 could cause 1023 too many pattern match attempts and 1023 too many dsize checks. Why? The content 0x13 would be found in the first byte, then the dsize option would fail, and because of recursion, the content 0x13 would be found again starting after where the previous 0x13 was found, once it is found, then check the dsize again, repeating until 0x13 is not found in the payload again.

Reordering the rule options so that discrete checks (such as dsize) are moved to the begining of the rule speed up Snort.

The optimized rule snipping would be:

```
dsize:1; content:"|13|";
```

A packet of 1024 bytes of 0x13 would fail immediately, as the dsize check is the first option checked and dsize is a discrete check without recursion.

The following rule options are discrete and should generally be placed at the begining of any rule:

- `dsize`
- `flags`
- `flow`
- `fragbits`
- `icmp_id`
- `icmp_seq`
- `icode`

- id
- ipopts
- ip_proto
- itype
- seq
- session
- tos
- ttl
- ack
- window
- resp
- sameip

3.11.5 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.......
<snip>
```

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:16; depth:4;
```

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

```
content:"|00 00 00 01|"; offset:20; 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:"g00 01 87 88|"; offset:12; depth:4;
content:"|00 00 00 01|"; offset:16; depth:4;
content:"|00 00 00 01|"; offset:20; 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:16; 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:16; depth:8;
byte_test:4,>,200,36;
```

Chapter 4

Making Snort Faster

4.1 MMAPed pcap

On Linux, a modified version of libpcap is available that implements a shared memory ring buffer. Phil Woods (cpw@lanl.gov) is the current maintainer of the libpcap implementation of the shared memory ring buffer. The shared memory ring buffer libpcap can be downloaded from his website at `http://public.lanl.gov/cpw/`.

Instead of the normal mechanism of copying the packets from kernel memory into userland memory, by using a shared memory ring buffer, libpcap is able to queue packets into a shared buffer that Snort is able to read directly. This change speeds up Snort by limiting the number of times the packet is copied before Snort gets to perform its detection upon it.

Once Snort linked against the shared memory libpcap, enabling the ring buffer is done via setting the enviornment variable *PCAP_FRAMES*. *PCAP_FRAMES* is the size of the ring buffer. According to Phil, the maximum size is 32768, as this appears to be the maximum number of iovecs the kernel can handle. By using *PCAP_FRAMES=max*, libpcap will automatically use the most frames possible. On Ethernet, this ends up being 1530 bytes per frame, for a total of around 52 Mbytes of memory for the ring buffer alone.

Chapter 5

Dynamic Modules

Preprocessors, detection capabilities, and rules can now be developed as dynamically loadable module to snort. When enabled via the *–enable-dynamicplugin* configure option, the dynamic API presents a means for loading dynamic libraries and allowing the module to utilize certain functions within the main snort code.

The remainder of this chapter will highlight the data structures and API functions used in developing preprocessors, detection engines, and rules as a dynamic plugin to snort.

5.1 Data Structures

A number of data structures are central to the API. The definition of each is defined in the following sections.

5.1.1 DynamicPluginMeta

The *DynamicPluginMeta* structure defines the type of dynamic module (preprocessor, rules, or detection engine), the version information, and path to the shared library. A shared library can implement all three types, but typically is limited to a single functionality such as a preprocessor. It is defined in `sf_dynamic_meta.h` as:

```
#define TYPE_ENGINE 0x01
#define TYPE_DETECTION 0x02
#define TYPE_PREPROCESSOR 0x04

typedef struct _DynamicPluginMeta
{
    int type;
    int major;
    int minor;
    int build;
    char uniqueName[MAX_NAME_LEN];
    char *libraryPath;
} DynamicPluginMeta;
```

5.1.2 DynamicPreprocessorData

The *DynamicPreprocessorData* structure defines the interface the preprocessor uses to interact with snort itself. This inclues functions to register the preprocessor's configuration parsing, restart, exit, and processing functions. It includes function to log messages, errors, fatal errors, and debugging info. It also includes information for setting alerts, handling Inline drops, access to the StreamAPI, and it provides access to the normalized http and alternate data

buffers. This data structure should be initialized when the preprocessor shared library is loaded. It is defined in sf_dynamic_preprocessor.h as:

```
typedef struct _DynamicPreprocessorData
{
    int version;
    char *altBuffer;
    unsigned int altBufferLen;
    UriInfo *uriBuffers[MAX_URIINFOS];
    LogMsgFunc logMsg;
    LogMsgFunc errMsg;
    LogMsgFunc fatalMsg;
    DebugMsgFunc debugMsg;

    PreprocRegisterFunc registerPreproc;
    AddPreprocFunc addPreproc;
    AddPreprocRestart addPreprocRestart;
    AddPreprocExit addPreprocExit;
    AddPreprocConfCheck addPreprocConfCheck;
    RegisterPreprocRuleOpt preprocOptRegister;
    AddPreprocProfileFunc addPreprocProfileFunc;
    ProfilingFunc profilingPreprocsFunc;
    void *totalPerfStats;

    AlertQueueAdd alertAdd;
    ThresholdCheckFunc thresholdCheck;

    InlineFunc  inlineMode;
    InlineDropFunc  inlineDrop;

    DetectFunc detect;
    DisableDetectFunc disableDetect;
    DisableDetectFunc disableAllDetect;

    SetPreprocBitFunc setPreprocBit;

    StreamAPI *streamAPI;
    SearchAPI *searchAPI;

    char **config_file;
    int *config_line;
    printfappendfunc printfappend;
    TokenSplitFunc tokenSplit;
    TokenFreeFunc tokenFree;

    GetRuleInfoByNameFunc getRuleInfoByName;
    GetRuleInfoByIdFunc getRuleInfoById;
} DynamicPreprocessorData;
```

5.1.3 DynamicEngineData

The *DynamicEngineData* structure defines the interface a detection engine uses to interact with snort itself. This includes functions for logging messages, errors, fatal errors, and debugging info as well as a means to register and check flowbits. It also includes a location to store rule-stubs for dynamic rules that are loaded, and it provides access to the normalized http and alternate data buffers. It is defined in sf_dynamic_engine.h as:

```
typedef struct _DynamicEngineData
{
    int version;
    char *altBuffer;
    UriInfo *uriBuffers[MAX_URIINFOS];
    RegisterRule ruleRegister;
    RegisterBit flowbitRegister;
    CheckFlowbit flowbitCheck;
    DetectAsn1 asn1Detect;
    LogMsg logMsg;
    LogMsg errMsg;
    LogMsg fatalMsg;
    char *dataDumpDirectory;

    GetPreprocRuleOptFuncs getPreprocOptFuncs;
} DynamicEngineData;
```

5.1.4 SFSnortPacket

The *SFSnortPacket* structure mirrors the snort Packet structure and provides access to all of the data contained in a given packet.

It and the data structures it incorporates are defined in `sf_snort_packet.h` as follows. Additional data structures may be defined to reference other protocol fields.

```
#define IP_RESBIT       0x8000
#define IP_DONTFRAG     0x4000
#define IP_MOREFRAGS    0x2000

typedef struct _IPV4Header
{
    u_int8_t version_headerlength;
    u_int8_t type_service;
    u_int16_t data_length;
    u_int16_t identifier;
    u_int16_t offset;
    u_int8_t time_to_live;
    u_int8_t proto;
    u_int16_t checksum;
    struct in_addr source;
    struct in_addr destination;
} IPV4Header;

#define MAX_IP_OPTIONS 40
/* ip option codes */
#define IPOPTION_EOL            0x00
#define IPOPTION_NOP            0x01
#define IPOPTION_RR             0x07
#define IPOPTION_RTRALT         0x14
#define IPOPTION_TS             0x44
#define IPOPTION_SECURITY       0x82
#define IPOPTION_LSRR           0x83
#define IPOPTION_LSRR_E         0x84
#define IPOPTION_SATID          0x88
#define IPOPTION_SSRR           0x89

typedef struct _IPOptions
```

```
{
    u_int8_t option_code;
    u_int8_t length;
    u_int8_t *option_data;
} IPOptions;

typedef struct _TCPHeader
{
    u_int16_t source_port;
    u_int16_t destination_port;
    u_int32_t sequence;
    u_int32_t acknowledgement;
    u_int8_t offset_reserved;
    u_int8_t flags;
    u_int16_t window;
    u_int16_t checksum;
    u_int16_t urgent_pointer;
} TCPHeader;

#define TCPHEADER_FIN  0x01
#define TCPHEADER_SYN  0x02
#define TCPHEADER_RST  0x04
#define TCPHEADER_PUSH 0x08
#define TCPHEADER_ACK  0x10
#define TCPHEADER_URG  0x20
#define TCPHEADER_RES2 0x40
#define TCPHEADER_RES1 0x80
#define TCPHEADER_NORESERVED (TCPHEADER_FIN|TCPHEADER_SYN|TCPHEADER_RST \
                              |TCPHEADER_PUSH|TCPHEADER_ACK|TCPHEADER_URG)
#define MAX_TCP_OPTIONS 40
/* tcp option codes */
#define TCPOPT_EOL              0x00
#define TCPOPT_NOP              0x01
#define TCPOPT_MSS              0x02
#define TCPOPT_WSCALE           0x03    /* window scale factor (rfc1072) */
#define TCPOPT_SACKOK           0x04    /* selective ack ok (rfc1072) */
#define TCPOPT_SACK             0x05    /* selective ack (rfc1072) */
#define TCPOPT_ECHO             0x06    /* echo (rfc1072) */
#define TCPOPT_ECHOREPLY        0x07    /* echo (rfc1072) */
#define TCPOPT_TIMESTAMP        0x08    /* timestamps (rfc1323) */
#define TCPOPT_CC               0x11    /* T/TCP CC options (rfc1644) */
#define TCPOPT_CCNEW            0x12    /* T/TCP CC options (rfc1644) */
#define TCPOPT_CCECHO           0x13    /* T/TCP CC options (rfc1644) */

typedef IPOptions TCPOptions;

typedef struct _UDPHeader
{
    u_int16_t source_port;
    u_int16_t destination_port;
    u_int16_t data_length;
    u_int16_t checksum;
} UDPHeader;

typedef struct _ICMPSequenceID
{
    u_int16_t id;
```

```
    u_int16_t seq;
} ICMPSequenceID;

typedef struct _ICMPHeader
{
    u_int8_t type;
    u_int8_t code;
    u_int16_t checksum;

    union
    {
        /* type 12 */
        u_int8_t parameter_problem_ptr;

        /* type 5 */
        struct in_addr gateway_addr;

        /* type 8, 0 */
        ICMPSequenceID echo;

        /* type 13, 14 */
        ICMPSequenceID timestamp;

        /* type 15, 16 */
        ICMPSequenceID info;

        int voidInfo;

        /* type 3/code=4 (Path MTU, RFC 1191) */
        struct path_mtu
        {
            u_int16_t voidInfo;
            u_int16_t next_mtu;
        } path_mtu;

        /* type 9 */
        struct router_advertisement
        {
            u_int8_t number_addrs;
            u_int8_t entry_size;
            u_int16_t lifetime;
        } router_advertisement;
    } icmp_header_union;
#define icmp_parameter_ptr  icmp_header_union.parameter_problem_ptr
#define icmp_gateway_addr   icmp_header_union.gateway_waddr
#define icmp_echo_id        icmp_header_union.echo.id
#define icmp_echo_seq       icmp_header_union.echo.seq
#define icmp_timestamp_id   icmp_header_union.timestamp.id
#define icmp_timestamp_seq  icmp_header_union.timestamp.seq
#define icmp_info_id        icmp_header_union.info.id
#define icmp_info_seq       icmp_header_union.info.seq
#define icmp_void           icmp_header_union.void
#define icmp_nextmtu        icmp_header_union.path_mtu.nextmtu
#define icmp_ra_num_addrs   icmp_header_union.router_advertisement.number_addrs
#define icmp_ra_entry_size  icmp_header_union.router_advertisement.entry_size
#define icmp_ra_lifetime    icmp_header_union.router_advertisement.lifetime
```

```
    union
    {
        /* timestamp */
        struct timestamp
        {
            u_int32_t orig;
            u_int32_t receive;
            u_int32_t transmit;
        } timestamp;

        /* IP header for unreach */
        struct ipv4_header
        {
            IPV4Header *ip;
            /* options and then 64 bits of data */
        } ipv4_header;

        /* Router Advertisement */
        struct router_address
        {
            u_int32_t addr;
            u_int32_t preference;
        } router_address;

        /* type 17, 18 */
        u_int32_t mask;

        char    data[1];
    } icmp_data_union;
#define icmp_orig_timestamp     icmp_data_union.timestamp.orig
#define icmp_recv_timestamp     icmp_data_union.timestamp.receive
#define icmp_xmit_timestamp     icmp_data_union.timestamp.transmit
#define icmp_ipheader           icmp_data_union.ip_header
#define icmp_ra_addr0           icmp_data_union.router_address
#define icmp_mask               icmp_data_union.mask
#define icmp_data               icmp_data_union.data
} ICMPHeader;

#define ICMP_ECHO_REPLY             0    /* Echo Reply                  */
#define ICMP_DEST_UNREACHABLE       3    /* Destination Unreachable     */
#define ICMP_SOURCE_QUENCH          4    /* Source Quench               */
#define ICMP_REDIRECT               5    /* Redirect (change route)     */
#define ICMP_ECHO_REQUEST           8    /* Echo Request                */
#define ICMP_ROUTER_ADVERTISEMENT   9    /* Router Advertisement        */
#define ICMP_ROUTER_SOLICITATION    10    /* Router Solicitation         */
#define ICMP_TIME_EXCEEDED          11    /* Time Exceeded               */
#define ICMP_PARAMETER_PROBLEM      12    /* Parameter Problem           */
#define ICMP_TIMESTAMP_REQUEST      13    /* Timestamp Request           */
#define ICMP_TIMESTAMP_REPLY        14    /* Timestamp Reply             */
#define ICMP_INFO_REQUEST           15    /* Information Request         */
#define ICMP_INFO_REPLY             16    /* Information Reply           */
#define ICMP_ADDRESS_REQUEST        17    /* Address Mask Request        */
#define ICMP_ADDRESS_REPLY          18    /* Address Mask Reply          */

#define CHECKSUM_INVALID_IP 0x01
#define CHECKSUM_INVALID_TCP 0x02
#define CHECKSUM_INVALID_UDP 0x04
```

```
#define CHECKSUM_INVALID_ICMP 0x08
#define CHECKSUM_INVALID_IGMP 0x10

typedef struct _SFSnortPacket
{
    struct pcap_pkthdr *pcap_header;
    u_int8_t *pkt_data;

    void *fddi_header;
    void *fddi_saps;
    void *fddi_sna;
    void *fddi_iparp;
    void *fddi_other;

    void *tokenring_header;
    void *tokenring_header_llc;
    void *tokenring_header_mr;

    void *sll_header;

    void *pflog_header;
    void *old_pflog_header;

    void *ether_header;
    void *vlan_tag_header;

    void *ether_header_llc;
    void *ether_header_other;

    void *wifi_header;

    void *ether_arp_header;

    void *ether_eapol_header; /* 802.1x */
    void *eapol_headear;
    u_int8_t *eapol_type;
    void *eapol_key;

    void *ppp_over_ether_header;

    IPV4Header *ip4_header, *orig_ip4_header;
    u_int32_t ip4_options_length;
    void *ip4_options_data;

    TCPHeader *tcp_header, *orig_tcp_header;
    u_int32_t tcp_options_length;
    void *tcp_options_data;

    UDPHeader *udp_header, *orig_udp_header;
    ICMPHeader *icmp_header, *orig_icmp_header;

    u_int8_t *payload;
    u_int16_t payload_size;
    u_int16_t normalized_payload_size;

    u_int16_t actual_ip_length;
```

```
    u_int8_t ip_fragmented;
    u_int16_t ip_fragment_offset;
    u_int8_t ip_more_fragments;
    u_int8_t ip_dont_fragment;
    u_int8_t ip_reserved;

    u_int16_t src_port;
    u_int16_t dst_port;
    u_int16_t orig_src_port;
    u_int16_t orig_dst_port;
    u_int32_t pcap_cap_len;

    u_int8_t num_uris;

    void *stream_session_ptr;
    void *fragmentation_tracking_ptr;
    void *flow_ptr;
    void *stream_ptr;

    IPOptions ip_options[MAX_IP_OPTIONS];
    u_int32_t num_ip_options;
    u_int8_t ip_last_option_invalid_flag;

    TCPOptions tcp_options[MAX_TCP_OPTIONS];
    u_int32_t num_tcp_options;
    u_int8_t tcp_last_option_invalid_flag;

    u_int8_t checksums_invalid;
    u_int32_t flags;
#define FLAG_REBUILT_FRAG     0x00000001
#define FLAG_REBUILT_STREAM   0x00000002
#define FLAG_STREAM_UNEST_UNI 0x00000004
#define FLAG_STREAM_UNEST_BI  0x00000008
#define FLAG_STREAM_EST       0x00000010
#define FLAG_FROM_SERVER      0x00000040
#define FLAG_FROM_CLIENT      0x00000080
#define FLAG_HTTP_DECODE      0x00000100
#define FLAG_STREAM_INSERT    0x00000400
#define FLAG_ALT_DECODE       0x00000800
    u_int32_t number_bytes_to_check;

    void *preprocessor_bit_mask;
} SFSnortPacket;
```

5.1.5 Dynamic Rules

A dynamic rule should use any of the following data structures. The following structures are defined in `sf_snort_plugin_api.h`.

Rule

The *Rule* structure defines the basic outline of a rule and contains the same set of information that is seen in a text rule. That includes protocol, address and port information and rule information (classification, generator and signature IDs, revision, priority, classification, and a list of references). It also includes a list of rule options and an optional evaluation function.

```
#define RULE_MATCH 1
#define RULE_NOMATCH 0

typedef struct _Rule
{
    IPInfo ip;
    RuleInformation info;

    RuleOption **options; /* NULL terminated array of RuleOption union */

    ruleEvalFunc evalFunc;

    char initialized;     /* Rule Initialized, used internally */
    u_int32_t numOptions; /* Rule option count, used internally */
    char noAlert;         /* Flag with no alert, used internally */
    void *ruleData;    /* Hash table for dynamic data pointers */
} Rule;
```

The rule evaluation function is defined as

```
int (*ruleEvalFunc)(void *);
```

where the parameter is a pointer to the SFSnortPacket structure.

RuleInformation

The *RuleInformation* structure defines the meta data for a rule and includes generator ID, signature ID, revision, classification, priority, message text, and a list of references.

```
int (*ruleEvalFunc)(void *);
truct _RuleInformation
{
    u_int32_t genID;
    u_int32_t sigID;
    u_int32_t revision;
    char     *classification; /* String format of classification name */
    u_int32_t priority;
    char     *message;
    RuleReference **references; /* NULL terminated array of references */
} RuleInformation;
```

RuleReference

The *RuleReference* structure defines a single rule reference, including the system name and rereference identifier.

```
typedef struct _RuleReference
{
    char *systemName;
    char *refIdentifier;
} RuleReference;
```

IPInfo

The *IPInfo* structure defines the initial matching criteria for a rule and includes the protocol, src address and port, destination address and port, and direction. Some of the standard strings and variables are predefined - any, HOME_NET, HTTP_SERVERS, HTTP_PORTS, etc.

```
typedef struct _IPInfo
{
    u_int8_t protocol;
    char *   src_addr;
    char *   src_port; /* 0 for non TCP/UDP */
    char     direction;     /* non-zero is bi-directional */
    char *   dst_addr;
    char *   dst_port; /* 0 for non TCP/UDP */
} IPInfo;

#define ANY_NET         "any"
#define HOME_NET        "$HOME_NET"
#define EXTERNAL_NET    "$EXTERNAL_NET"
#define ANY_PORT        "any"
#define HTTP_SERVERS    "$HTTP_SERVERS"
#define HTTP_PORTS      "$HTTP_PORTS"
#define SMTP_SERVERS    "$SMTP_SERVERS"
```

RuleOption

The *RuleOption* structure defines a single rule option as an option type and a reference to the data specific to that option. Each option has a flags field that contains specific flags for that option as well as a ”Not” flag. The ”Not” flag is used to negate the results of evaluating that option.

```
#define OPTION_TYPE_CONTENT      0x01
#define OPTION_TYPE_PCRE         0x02
#define OPTION_TYPE_FLOWBIT      0x03
#define OPTION_TYPE_FLOWFLAGS    0x04
#define OPTION_TYPE_ASN1         0x05
#define OPTION_TYPE_CURSOR       0x06
#define OPTION_TYPE_HDR_CHECK    0x07
#define OPTION_TYPE_BYTE_TEST    0x08
#define OPTION_TYPE_BYTE_JUMP    0x09
#define OPTION_TYPE_BYTE_EXTRACT 0x10
#define OPTION_TYPE_SET_CURSOR   0x11
#define OPTION_TYPE_LOOP         0x12

typedef struct _RuleOption
{
    int optionType;
    union
    {
        void *ptr;
        ContentInfo *content;
        CursorInfo *cursor;
        PCREInfo *pcre;
        FlowBitsInfo *flowBit;
        ByteData *byte;
        ByteExtract *byteExtract;
        FlowFlags *flowFlags;
```

```
        Asn1Context *asn1;
        HdrOptCheck *hdrData;
        LoopInfo    *loop;
    } option_u;
} RuleOption;

#define NOT_FLAG                0x10000000
```

Some options also contain information that is initialized at run time, such as the compiled PCRE information, Boyer-Moore content information, the integer ID for a flowbit, etc.

The option types and related structures are listed below.

- OptionType: Content & Structure: *ContentInfo*

  The *ContentInfo* structure defines an option for a content search. It includes the pattern, depth and offset, and flags (one of which must specify the buffer – raw, URI or normalized – to search). Additional flags include nocase, relative, unicode, and a designation that this content is to be used for snorts fast pattern evaluation. The most unique content, that which distinguishes this rule as a possible match to a packet, should be marked for fast pattern evaluation. In the dynamic detection engine provided with Snort, if no *ContentInfo* structure in a given rules uses that flag, the one with the longest content length will be used.

  ```
  typedef struct _ContentInfo
  {
      u_int8_t *pattern;
      u_int32_t depth;
      int32_t   offset;
      u_int32_t flags;        /* must include a CONTENT_BUF_X */
      void     *boyer_ptr;
      u_int8_t *patternByteForm;
      u_int32_t patternByteFormLength;
      u_int32_t incrementLength;
  } ContentInfo;

  #define CONTENT_NOCASE          0x01
  #define CONTENT_RELATIVE        0x02
  #define CONTENT_UNICODE2BYTE    0x04
  #define CONTENT_UNICODE4BYTE    0x08
  #define CONTENT_FAST_PATTERN    0x10
  #define CONTENT_END_BUFFER      0x20

  #define CONTENT_BUF_NORMALIZED  0x100
  #define CONTENT_BUF_RAW         0x200
  #define CONTENT_BUF_URI         0x400
  ```

- OptionType: PCRE & Structure: *PCREInfo*

  The *PCREInfo* structure defines an option for a PCRE search. It includes the PCRE expression, pcre_flags such as caseless, as defined in PCRE.h, and flags to specify the buffer.

  ```
  /*
  pcre.h provides flags:

  PCRE_CASELESS
  PCRE_MULTILINE
  PCRE_DOTALL
  PCRE_EXTENDED
  PCRE_ANCHORED
  ```

```
PCRE_DOLLAR_ENDONLY
PCRE_UNGREEDY
*/

typedef struct _PCREInfo
{
    char     *expr;
    void     *compiled_expr;
    void     *compiled_extra;
    u_int32_t compile_flags;
    u_int32_t flags; /* must include a CONTENT_BUF_X */
} PCREInfo;
```

- OptionType: Flowbit & Structure: *FlowBitsInfo*

  The *FlowBitsInfo* structure defines a flowbits option. It includes the name of the flowbit and the operation (set, unset, toggle, isset, isnotset).

```
#define FLOWBIT_SET       0x01
#define FLOWBIT_UNSET     0x02
#define FLOWBIT_TOGGLE    0x04
#define FLOWBIT_ISSET     0x08
#define FLOWBIT_ISNOTSET  0x10
#define FLOWBIT_RESET     0x20
#define FLOWBIT_NOALERT   0x40

typedef struct _FlowBitsInfo
{
    char      *flowBitsName;
    u_int8_t   operation;
    u_int32_t  id;
    u_int32_t  flags;
} FlowBitsInfo;
```

- OptionType: Flow Flags & Structure: *FlowFlags*

  The *FlowFlags* structure defines a flow option. It includes the flags, which specify the direction (from_server, to_server), established session, etc.

```
#define FLOW_ESTABLISHED 0x10
#define FLOW_IGNORE_REASSEMBLED 0x1000
#define FLOW_ONLY_REASSMBLED    0x2000
#define FLOW_FR_SERVER   0x40
#define FLOW_TO_CLIENT   0x40 /* Just for redundancy */
#define FLOW_TO_SERVER   0x80
#define FLOW_FR_CLIENT   0x80 /* Just for redundancy */

typedef struct _FlowFlags
{
    u_int32_t   flags;
} FlowFlags;
```

- OptionType: ASN.1 & Structure: *Asn1Context*

  The *Asn1Context* structure defines the information for an ASN1 option. It mirrors the ASN1 rule option and also includes a flags field.

```
#define ASN1_ABS_OFFSET 1
```

```
#define ASN1_REL_OFFSET 2

typedef struct _Asn1Context
{
    int bs_overflow;
    int double_overflow;
    int print;
    int length;
    unsigned int max_length;
    int offset;
    int offset_type;
    u_int32_t  flags;
} Asn1Context;
```

- OptionType: Cursor Check & Structure: *CursorInfo*

  The *CursorInfo* structure defines an option for a cursor evaluation. The cursor is the current position within the evaluation buffer, as related to content and PCRE searches, as well as byte tests and byte jumps. It includes an offset and flags that specify the buffer. This can be used to verify there is sufficient data to continue evaluation, similar to the isdataat rule option.

```
typedef struct _CursorInfo
{
    int32_t   offset;
    u_int32_t flags;        /* specify one of CONTENT_BUF_X */
} CursorInfo;
```

- OptionType: Protocol Header & Structure: *HdrOptCheck*

  The *HdrOptCheck* structure defines an option to check a protocol header for a specific value. It incldues the header field, the operation (¡,¿,=,etc), a value, a mask to ignore that part of the header field, and flags.

```
#define IP_HDR_ID           0x0001  /* IP Header ID */
#define IP_HDR_PROTO        0x0002  /* IP Protocol */
#define IP_HDR_FRAGBITS     0x0003  /* Frag Flags set in IP Header */
#define IP_HDR_FRAGOFFSET   0x0004  /* Frag Offset set in IP Header */
#define IP_HDR_OPTIONS      0x0005  /* IP Options -- is option xx included */
#define IP_HDR_TTL          0x0006  /* IP Time to live */
#define IP_HDR_TOS          0x0007  /* IP Type of Service */
#define IP_HDR_OPTCHECK_MASK 0x000f

#define TCP_HDR_ACK         0x0010  /* TCP Ack Value */
#define TCP_HDR_SEQ         0x0020  /* TCP Seq Value */
#define TCP_HDR_FLAGS       0x0030  /* Flags set in TCP Header */
#define TCP_HDR_OPTIONS     0x0040  /* TCP Options -- is option xx included */
#define TCP_HDR_WIN         0x0050  /* TCP Window */
#define TCP_HDR_OPTCHECK_MASK 0x00f0

#define ICMP_HDR_CODE       0x1000  /* ICMP Header Code */
#define ICMP_HDR_TYPE       0x2000  /* ICMP Header Type */
#define ICMP_HDR_ID         0x3000  /* ICMP ID for ICMP_ECHO/ICMP_ECHO_REPLY */
#define ICMP_HDR_SEQ        0x4000  /* ICMP ID for ICMP_ECHO/ICMP_ECHO_REPLY */
#define ICMP_HDR_OPTCHECK_MASK 0xf000

typedef struct _HdrOptCheck
{
    u_int16_t hdrField;   /* Field to check */
    u_int32_t op;         /* Type of comparison */
```

```
    u_int32_t value;      /* Value to compare value against */
    u_int32_t mask_value; /* bits of value to ignore */
    u_int32_t flags;
} HdrOptCheck;
```

- OptionType: Byte Test & Structure: *ByteData*

  The *ByteData* structure defines the information for both ByteTest and ByteJump operations. It includes the number of bytes, an operation (for ByteTest, ¡,¿,=,etc), a value, an offset, multiplier, and flags. The flags must specify the buffer.

```
#define CHECK_EQ            0
#define CHECK_NEQ           1
#define CHECK_LT            2
#define CHECK_GT            3
#define CHECK_LTE           4
#define CHECK_GTE           5
#define CHECK_AND           6
#define CHECK_XOR           7
#define CHECK_ALL           8
#define CHECK_ATLEASTONE    9
#define CHECK_NONE          10

typedef struct _ByteData
{
    u_int32_t bytes;      /* Number of bytes to extract */
    u_int32_t op;         /* Type of byte comparison, for checkValue */
    u_int32_t value;      /* Value to compare value against, for checkValue, or extracted value */
    int32_t   offset;     /* Offset from cursor */
    u_int32_t multiplier; /* Used for byte jump -- 32bits is MORE than enough */
    u_int32_t flags;      /* must include a CONTENT_BUF_X */
} ByteData;
```

- OptionType: Byte Jump & Structure: *ByteData*

  See *Byte Test* above.

- OptionType: Set Cursor & Structure: *CursorInfo*

  See *Cursor Check* above.

- OptionType: Loop & Structures: *LoopInfo,ByteExtract,DynamicElement*

  The *LoopInfo* structure defines the information for a set of options that are to be evaluated repeatedly. The loop option acts like a FOR loop and includes start, end, and increment values as well as the comparison operation for termination. It includes a cursor adjust that happens through each iteration of the loop, a reference to a RuleInfo structure that defines the RuleOptions are to be evaluated through each iteration. One of those options may be a ByteExtract.

```
typedef struct _LoopInfo
{
    DynamicElement *start;      /* Starting value of FOR loop (i=start) */
    DynamicElement *end;        /* Ending value of FOR loop (i OP end) */
    DynamicElement *increment;  /* Increment value of FOR loop (i+= increment) */
    u_int32_t op;               /* Type of comparison for loop termination */
    CursorInfo *cursorAdjust;   /* How to move cursor each iteration of loop */
    struct _Rule *subRule;      /* Pointer to SubRule & options to evaluate within
                                 * the loop */
    u_int8_t initialized;       /* Loop initialized properly (safeguard) */
    u_int32_t flags;            /* can be used to negate loop results, specifies
} LoopInfo;
```

The *ByteExtract* structure defines the information to use when extracting bytes for a DynamicElement used a in Loop evaltion. It includes the number of bytes, an offset, multiplier, flags specifying the buffer, and a reference to the DynamicElement.

```
typedef struct _ByteExtract
{
    u_int32_t bytes;      /* Number of bytes to extract */
    int32_t   offset;     /* Offset from cursor */
    u_int32_t multiplier; /* Multiply value by this (similar to byte jump) */
    u_int32_t flags;      /* must include a CONTENT_BUF_X */
    char *refId;          /* To match up with a DynamicElement refId */
    void *memoryLocation; /* Location to store the data extracted */
} ByteExtract;
```

The *DynamicElement* structure is used to define the values for a looping evaluation. It includes whether the element is static (an integer) or dynamic (extracted from a buffer in the packet) and the value. For a dynamic element, the value is filled by a related ByteExtract option that is part of the loop.

```
#define DYNAMIC_TYPE_INT_STATIC 1
#define DYNAMIC_TYPE_INT_REF    2

typedef struct _DynamicElement
{
    char dynamicType;           /* type of this field - static or reference */
    char *refId;                /* reference ID (NULL if static) */
    union
    {
        void *voidPtr;          /* Holder */
        int32_t staticInt;        /* Value of static */
        int32_t *dynamicInt;  /* Pointer to value of dynamic */
    } data;
} DynamicElement;
```

5.2 Required Functions

Each dynamic module must define a set of functions and data objects to work within this framework.

5.2.1 Preprocessors

Each dynamic preprocessor library must define the following functions. These are defined in the file `sf_dynamic_preproc_lib.c`. The metadata and setup function for the preprocessor should be defined `sf_preproc_info.h`.

- *int LibVersion(DynamicPluginMeta *)*

  This function returns the metadata for the shared library.

- *int InitializePreprocessor(DynamicPreprocessorData *)*

  This function initializes the data structure for use by the preprocessor into a library global variable, `_dpd` and invokes the setup function.

5.2.2 Detection Engine

Each dynamic detection engine library must define the following functions.

- *int LibVersion(DynamicPluginMeta *)*

  This function returns the metadata for the shared library.

- *int InitializeEngineLib(DynamicEngineData *)*

  This function initializes the data structure for use by the engine.

The sample code provided with Snort predefines those functions and defines the following APIs to be used by a dynamic rules library.

- *int RegisterRules(Rule **)*

  This is the function to iterate through each rule in the list, initialize it to setup content searches, PCRE evalution data, and register flowbits.

- *int DumpRules(char *,Rule **)*

  This is the function to iterate through each rule in the list and write a rule-stop to be used by snort to control the action of the rule (alert, log, drop, etc).

- *int ruleMatch(void *p, Rule *rule)*

  This is the function to evaluate a rule if the rule does not have its own Rule Evaluation Function. This uses the individual functions outlined below for each of the rule options and handles repetitive content issues.

  Each of the functions below returns RULE_MATCH if the option matches based on the current criteria (cursor position, etc).

  - *int contentMatch(void *p, ContentInfo* content, u_int8_t **cursor)*

    This function evaluates a single content for a given packet, checking for the existence of that content as delimited by ContentInfo and cursor. Cursor position is updated and returned in *cursor.

    With a text rule, the with option corresponds to depth, and the distance option corresponds to offset.

  - *int checkFlow(void *p, FlowFlags *flowflags)*

    This function evaluates the flow for a given packet.

  - *int extractValue(void *p, ByteExtract *byteExtract, u_int8_t *cursor)*

    This function extracts the bytes from a given packet, as specified by ByteExtract and delimited by cursor. Value extracted is stored in ByteExtract memoryLocation paraneter.

  - *int processFlowbits(void *p, FlowBitsInfo *flowbits)*

    This function evaluates the flowbits for a given packet, as specified by FlowBitsInfo. It will interact with flowbits used by text-based rules.

  - *int setCursor(void *p, CursorInfo *cursorInfo, u_int8_t **cursor)*

    This function adjusts the cursor as delimited by CursorInfo. New cursor position is returned in *cursor. It handles bounds checking for the specified buffer and returns RULE_NOMATCH if the cursor is moved out of bounds.

    It is also used by contentMatch, byteJump, and pcreMatch to adjust the cursor position after a successful match.

  - *int checkCursor(void *p, CursorInfo *cursorInfo, u_int8_t *cursor)*

    This function validates that the cursor is within bounds of the specified buffer.

  - *int checkValue(void *p, ByteData *byteData, u_int32_t value, u_int8_t *cursor)*

    This function compares the *value* to the value stored in ByteData.

  - *int byteTest(void *p, ByteData *byteData, u_int8_t *cursor)*

    This is a wrapper for extractValue() followed by checkValue().

  - *int byteJump(void *p, ByteData *byteData, u_int8_t **cursor)*

    This is a wrapper for extractValue() followed by setCursor().

  - *int pcreMatch(void *p, PCREInfo *pcre, u_int8_t **cursor)*

    This function evaluates a single pcre for a given packet, checking for the existence of the expression as delimited by PCREInfo and cursor. Cursor position is updated and returned in *cursor.

- *int detectAsn1(void *p, Asn1Context *asn1, u_int8_t *cursor)*
  This function evaluates an ASN.1 check for a given packet, as delimited by Asn1Context and cursor.
- *int checkHdrOpt(void *p, HdrOptCheck *optData)*
  This function evaluates the given packet's protocol headers, as specified by HdrOptCheck.
- *int loopEval(void *p, LoopInfo *loop, u_int8_t **cursor)*
  This function iterates through the SubRule of LoopInfo, as delimited by LoopInfo and cursor. Cursor position is updated and returned in *cursor.
- *int preprocOptionEval(void *p, PreprocessorOption *preprocOpt, u_int8_t **cursor)*
  This function evaluates the preprocessor defined option, as spepcifed by PreprocessorOption. Cursor position is updated and returned in *cursor.
- *void setTempCursor(u_int8_t **temp_cursor, u_int8_t **cursor)*
  This function is used to handled repetitive contents to save off a cursor position temporarily to be reset at later point.
- *void revertTempCursor(u_int8_t **temp_cursor, u_int8_t **cursor)*
  This function is used to revert to a previously saved temporary cursor position.

If you decide to write you own rule evaluation function, patterns that occur more than once may result in false negatives. Take extra care to handle this situation and search for the matched pattern again if subsequent rule options fail to match. This should be done for both content and PCRE options.

5.2.3 Rules

Each dynamic rules library must define the following functions. Examples are defined in the file `sfnort_dynamic_detection_lib.c`. The metadata and setup function for the preprocessor should be defined `sfsnort_dynamic_detection_lib.h`.

- *int LibVersion(DynamicPluginMeta *)*
  This function returns the metadata for the shared library.
- *int EngineVersion(DynamicPluginMeta *)*
  This function defines the version requirements for the corresponding detection engine library.
- *int DumpSkeletonRules()*
  This functions writes out the rule-stubs for rules that are loaded.
- *int InitializeDetection()*
  This function registers each rule in the rules library. It should set up fast pattern-matcher content, register flowbits, etc.

The sample code provided with Snort predefines those functions and uses the following data within the dynamic rules library.

- *Rule *rules[]*
  A NULL terminated list of Rule structures that this library defines.

5.3 Examples

This section provides a simple example of a dynamic preprocessor and a dynamic rule.

5.3.1 Preprocessor Example

The following is an example of a simple preprocessor. This preprocessor always alerts on a Packet if the TCP port matches the one configured.

This assumes the the files *sf_dynamic_preproc_lib.c* and *sf_dynamic_preproc_lib.h* are used.

This is the metadata for this preprocessor, defined in *sf_preproc_info.h*.

```
#define MAJOR_VERSION 1
#define MINOR_VERSION 0
#define BUILD_VERSION 0
#define PREPROC_NAME    "SF_Dynamic_Example_Preprocessor"

#define DYNAMIC_PREPROC_SETUP   ExampleSetup
extern void ExampleSetup();
```

The remainder of the code is defined in *spp_example.c* and is compiled together with *sf_dynamic_preproc_lib.c* into lib_sfdynamic_preprocessor_example.so.

Define the Setup function to register the initialization function.

```
#define GENERATOR_EXAMPLE 256
extern DynamicPreprocessorData _dpd;

void ExampleInit(unsigned char *);
void ExampleProcess(void *, void *);

void ExampleSetup()
{
    _dpd.registerPreproc("dynamic_example", ExampleInit);

    DEBUG_WRAP(_dpd.debugMsg(DEBUG_PLUGIN, "Preprocessor: Example is setup\n"););
}
```

The initialization function to parse the keywords from `snort.conf`.

```
u_int16_t portToCheck;

void ExampleInit(unsigned char *args)
{
    char *arg;
    char *argEnd;
    unsigned long port;

    _dpd.logMsg("Example dynamic preprocessor configuration\n");

    arg = strtok(args, " \t\n\r");

    if(!strcasecmp("port", arg))
    {
        arg = strtok(NULL, "\t\n\r");
        if (!arg)
        {
            _dpd.fatalMsg("ExamplePreproc: Missing port\n");
        }
```

```
        port = strtoul(arg, &argEnd, 10);
        if (port < 0 || port > 65535)
        {
            _dpd.fatalMsg("ExamplePreproc: Invalid port %d\n", port);
        }
        portToCheck = port;

        _dpd.logMsg("    Port: %d\n", portToCheck);
    }
    else
    {
        _dpd.fatalMsg("ExamplePreproc: Invalid option %s\n", arg);
    }

    /* Register the preprocessor function, Transport layer, ID 10000 */
    _dpd.addPreproc(ExampleProcess, PRIORITY_TRANSPORT, 10000);

    DEBUG_WRAP(_dpd.debugMsg(DEBUG_PLUGIN, "Preprocessor: Example is initialized\n"););
}
```

The function to process the packet and log an alert if the either port matches.

```
#define SRC_PORT_MATCH  1
#define SRC_PORT_MATCH_STR "example_preprocessor: src port match"
#define DST_PORT_MATCH  2
#define DST_PORT_MATCH_STR "example_preprocessor: dest port match"
void ExampleProcess(void *pkt, void *context)
{
    SFSnortPacket *p = (SFSnortPacket *)pkt;
    if (!p->ip4_header || p->ip4_header->proto != IPPROTO_TCP || !p->tcp_header)
    {
        /* Not for me, return */
        return;
    }

    if (p->src_port == portToCheck)
    {
        /* Source port matched, log alert */
        _dpd.alertAdd(GENERATOR_EXAMPLE, SRC_PORT_MATCH,
                      1, 0, 3, SRC_PORT_MATCH_STR, 0);
        return;
    }

    if (p->dst_port == portToCheck)
    {
        /* Destination port matched, log alert */
        _dpd.alertAdd(GENERATOR_EXAMPLE, DST_PORT_MATCH,
                      1, 0, 3, DST_PORT_MATCH_STR, 0);
        return;
    }
}
```

5.3.2 Rules

The following is an example of a simple rule, take from the current rule set, SID 109. It is implemented to work with the detection engine provided with snort.

The snort rule in normal format:

```
alert tcp $HOME_NET 12345:12346 -> $EXTERNAL_NET any \
(msg:"BACKDOOR netbus active"; flow:from_server,established; \
content:"NetBus"; reference:arachnids,401; classtype:misc-activity; \
sid:109; rev:5;)
```

This is the metadata for this rule library, defined in *detection_lib_meta.h*.

```
/* Version for this rule library */
#define DETECTION_LIB_MAJOR_VERSION 1
#define DETECTION_LIB_MINOR_VERSION 0
#define DETECTION_LIB_BUILD_VERSION 1
#define DETECTION_LIB_NAME "Snort_Dynamic_Rule_Example"

/* Required version and name of the engine */
#define REQ_ENGINE_LIB_MAJOR_VERSION 1
#define REQ_ENGINE_LIB_MINOR_VERSION 0
#define REQ_ENGINE_LIB_NAME "SF_SNORT_DETECTION_ENGINE"
```

The definition of each data structure for this rule is in *sid109.c*.

Declaration of the data structures.

- Flow option

  Define the *FlowFlags* structure and its corresponding *RuleOption*. Per the text version, flow is from_server,established.

  ```
  static FlowFlags sid109flow =
  {
      FLOW_ESTABLISHED|FLOW_TO_CLIENT
  };

  static RuleOption sid109option1 =
  {
      OPTION_TYPE_FLOWFLAGS,
      {
          &sid109flow
      }
  };
  ```

- Content Option

  Define the *ContentInfo* structure and its corresponding *RuleOption*. Per the text version, content is ”NetBus”, no depth or offset, case sensitive, and non-relative. Search on the normalized buffer by default. **NOTE**: This content will be used for the fast pattern matcher since it is the longest content option for this rule and no contents have a flag of *CONTENT_FAST_PATTERN*.

  ```
  static ContentInfo sid109content =
  {
      "NetBus",               /* pattern to search for */
      0,                      /* depth */
      0,                      /* offset */
      CONTENT_BUF_NORMALIZED, /* flags */
      NULL,                   /* holder for boyer/moore info */
      NULL,                   /* holder for byte representation of "NetBus" */
  ```

```
    0,                      /* holder for length of byte representation */
    0                       /* holder for increment length */
};

static RuleOption sid109option2 =
{
    OPTION_TYPE_CONTENT,
    {
        &sid109content
    }
};
```

- Rule and Meta Data

  Define the references.

```
static RuleReference sid109ref_arachnids =
{
    "arachnids",    /* Type */
    "401"           /* value */
};

static RuleReference *sid109refs[] =
{
    &sid109ref_arachnids,
    NULL
};
```

The list of rule options. Rule options are evaluated in the order specified.

```
RuleOption *sid109options[] =
{
    &sid109option1,
    &sid109option2,
    NULL
};
```

The rule itself, with the protocl header, meta data (sid, classification, message, etc).

```
Rule sid109 =
{
    /* protocol header, akin to => tcp any any -> any any               */
    {
        IPPROTO_TCP,        /* proto */
        HOME_NET,           /* source IP     */
        "12345:12346",      /* source port(s) */
        0,                  /* Direction */
        EXTERNAL_NET,       /* destination IP     */
        ANY_PORT,           /* destination port   */
    },
    /* metadata */
    {
        3,                  /* genid -- use 3 to distinguish a C rule */
        109,                /* sigid */
        5,                  /* revision */
        "misc-activity",    /* classification */
        0,                  /* priority */
```

```
        "BACKDOOR netbus active",     /* message */
        sid109refs          /* ptr to references */
    },
    sid109options,          /* ptr to rule options */
    NULL,                   /* Use internal eval func */
    0,                      /* Holder, not yet initialized, used internally */
    0,                      /* Holder, option count, used internally */
    0,                      /* Holder, no alert, used internally for flowbits */
    NULL                    /* Holder, rule data, used internally */
```

- The List of rules defined by this rules library

  The NULL terminated list of rules. The InitializeDetection iterates through each Rule in the list and initializes the content, flowbits, pcre, etc.

```
extern Rule sid109;
extern Rule sid637;

Rule *rules[] =
{
    &sid109,
    &sid637,
    NULL
};
```

Chapter 6

Snort Development

Currently, this chapter is here as a place holder. It will someday contain references on how to create new detection plugins and preprocessors. End users don't really need to be reading this section. This is intended to help developers get a basic understanding of whats going on quickly.

If you are going to be helping out with Snort development, please use the HEAD branch of cvs. We've had problems in the past of people submitting patches only to the stable branch (since they are likely writing this stuff for their own IDS purposes). Bugfixes are what goes into STABLE. Features go into HEAD.

6.1 Submitting Patches

Patches to Snort should be sent to the `snort-devel@lists.sourceforge.net` mailing list. Patches should done with the command `diff -nu snort-orig snort-new`.

6.2 Snort Data Flow

First, traffic is acquired from the network link via libpcap. Packets are passed through a series of decoder routines that first fill out the packet structure for link level protocols then are further decoded for things like TCP and UDP ports.

Packets are then sent through the registered set of preprocessors. Each preprocessor checks to see if this packet is something it should look at.

Packets are then sent through the detection engine. The detection engine checks each packet against the various options listed in the Snort rules files. Each of the keyword options is a plugin. This allows this to be easily extensible.

6.2.1 Preprocessors

For example, a TCP analysis preprocessor could simply return if the packet does not have a TCP header. It can do this by checking:

```
if (p->tcph==null)
   return;
```

Similarly, there are a lot of packet_flags available that can be used to mark a packet as "reassembled" or logged. Check out src/decode.h for the list of pkt_* constants.

6.2.2 Detection Plugins

Basically, look at an existing output plugin and copy it to a new item and change a few things. Later, we'll document what these few things are.

6.2.3 Output Plugins

Generally, new output plugins should go into the barnyard project rather than the Snort project. We are currently cleaning house on the available output options.

6.3 The Snort Team

Creator and Lead Architect	Marty Roesch
Lead Snort Developers	Marc Norton Andrew Mullican Steve Sturges
Snort Rules Maintainer	Brian Caswell
Snort Rules Team	Nigel Houghton Alex Kirk Judy Novak Matt Watchinski
Win32 Maintainer	Chris Reid
RPM Maintainers	JP Vossen Daniel Wittenberg
Inline Developers	Victor Julien Rob McMillen William Metcalf
Major Contributors	Daniel Roelker Andrew Baker Erek Adams Scott Campbell Roman D. Michael Davis Chris Green Jed Haile Jeremy Hewlett Glenn Mansfield Keeni Chad Kreimendahl Jeff Nathan Andreas Ostling Dragos Ruiu Fyodor Yarochkin Phil Wood

Bibliography

[1] http://packetstorm.securify.com/mag/phrack/phrack49/p49-06

[2] http://www.nmap.org

[3] http://public.pacbell.net/dedicated/cidr.html

[4] http://www.whitehats.com

[5] http://www.incident.org/snortdb

[6] http://www.pcre.org