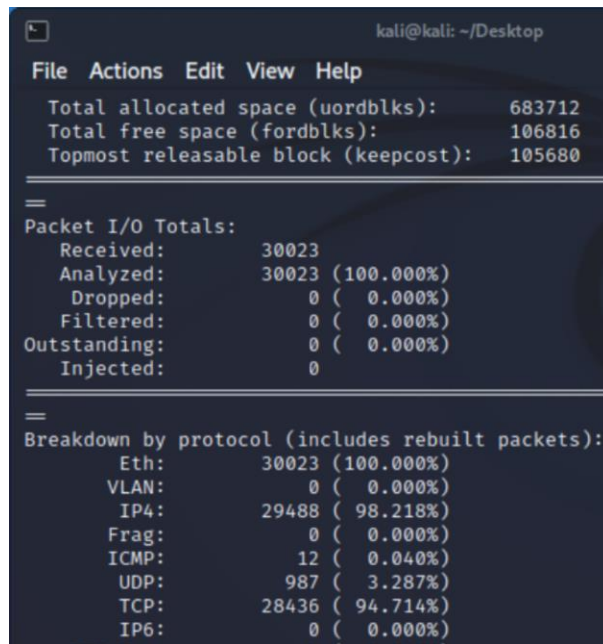


Snort is a powerful open-source intrusion detection system (IDS) and intrusion prevention system (IPS) that provides real-time network traffic analysis and data packet logging. Snort uses a rule-based language that combines anomaly, protocol, and signature inspection methods to detect potentially malicious activity.

Using Snort, network admins can spot denial-of-service (DoS) attacks and distributed DoS (DDoS) attacks, Common Gateway Interface (CGI) attacks, buffer overflows, and stealth port scans. Snort creates a series of rules that define malicious network activity, identify malicious packets, and send alerts to users.

Snort can also examine packets in a capture file (PCAP). Snort will decode and count the packets in the file and output statistics such as number of packets analyzed and their breakdown by protocol.

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



```
kali@kali: ~/Desktop
File Actions Edit View Help
Total allocated space (uordblks): 683712
Total free space (fordblks): 106816
Topmost releasable block (keepcost): 105680
=====
Packet I/O Totals:
Received: 30023
Analyzed: 30023 (100.000%)
Dropped: 0 ( 0.000%)
Filtered: 0 ( 0.000%)
Outstanding: 0 ( 0.000%)
Injected: 0
=====
Breakdown by protocol (includes rebuilt packets):
Eth: 30023 (100.000%)
VLAN: 0 ( 0.000%)
IP4: 29488 ( 98.218%)
Frag: 0 ( 0.000%)
ICMP: 12 ( 0.040%)
UDP: 987 ( 3.287%)
TCP: 28436 ( 94.714%)
IP6: 0 ( 0.000%)
```

To leverage the intrusion detection features of Snort you will need to provide some configuration details. Effective configuration of Snort is done via the environment, command line, a Lua configuration file, and a set of rules. A simple command line might look like this:

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

- **-c snort.lua** is the main configuration file. This is a Lua script that is executed when loaded.
- **-R cool.rules** contains some detection rules. You can write your own or obtain them from Talos. You can also put your rules directly in your configuration file.
- **-r some.pcap** tells Snort to read network traffic from the given packet capture file. You could instead use **-i eth0** to read from a live interface.
- **-A cmg** says to output intrusion events in "cmg" format, which has basic header details followed by the payload in hex and text.

Rules determine what Snort is looking for. They can be put directly in your Lua configuration file with the `ips` module, on the command line with `--lua`, or in external files. Add this to your Lua configuration ( `/etc/snort/snort.lua` ) to load the external rules file named `rules.txt`:

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

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

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

Rule structure is as follows: *action protocol source direction destination ( body )*

- **action** - tells Snort what to do when a rule matches. In this case Snort will alert (log) the event. When run inline, Snort can be configured to *drop*, and not just *alert*, to matches.
- **protocol** - tells Snort what protocol applies. This may be ip, icmp, tcp, udp, http, etc.
- **source** - specifies the sending IP address and port. CIDR blocks **[10.0.0.0/8]**, Comma-Separated Lists **[80,443]** **[1.1.1.1,2.2.2.2]**, Negation **![22]**, Wildcards **any**, Variables such as **\$HTTP\_PORTS**, can all be used
- **direction** - must be either unidirectional -> or bidirectional indicated by <>.
- **destination** - specifies the receiving IP address and port. Uses same arguments as source
- **body** - detection and other information contained in parenthesis.

Body structure is as follows: *Rule message; Content matches; Modifiers; Alternate data buffers; Byte operations; Regular expressions; Signature identification*

**Rule Message:** Category, Malware Family, Attempted Action.

msg: "MALWARE-CNC Win.Trojan.Zeus variant outbound connection"

**Content Matches.** Keywords as well as hex values can be specified:

content: "TALOS";

content: "|54 14 4C 4F 53| Intelligence";

**Modifiers.** Specify packet offset, depth, distance, proximity; rule performance.

content: "MALWARE" offset:10; depth:10;

content: "TROJAN"; content: "ZEUS"; distance:12; within:3;

content: "%pdf"; fast\_pattern;

**Alternate Data Buffers:**

content: "/documents/1.doc"; http\_uri;

content: "User-Agent| MalwareBot"; http\_header; fast\_pattern:only;

**Byte Operations:**

byte\_jump:4,0,relative,big; #Extract 4 byte offset to IFD

byte\_extract:1,23,name\_length,relative; #Extract length of file name

**PCRE (Regular Expressions):**

content: "?message="; pcre:"/(info|app)\x2ephp";

**Signature Identification (SID):** Uniquely identifies Snort rules. They must be included.

( msg:"Warning!"; content:"attack"; sid:1; )

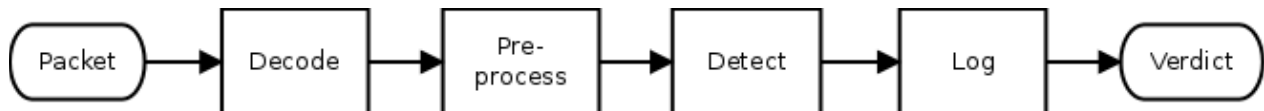
If you configured rules, you will need to configure alerts to see the details of detection events. Use the `-A` option like this:

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

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

- **-A cmg** will show information about the alert along with packet headers and payload.
- **-A u2** will log events and triggering packets in a binary file that you can feed to other tools for post processing.
- **-A csv** will output various fields in comma separated value format. This is entirely customizable and very useful for pcap analysis.

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



The steps are:

**Decode** each packet to determine the basic network characteristics such as source and destination addresses and ports. The various encapsulating protocols are examined for sanity and anomalies as the packet is decoded.

**Pre-process** each decoded packet using accumulated state to determine the purpose and content of the innermost message. This step may involve reordering and reassembling IP fragments and TCP segments to produce the original application protocol data unit (PDU). Such PDUs are analyzed and normalized as needed to support further processing.

**Detect.** Upon start up, the rules are compiled into pattern groups such that a single, parallel search can be done for all patterns in the group. If any match is found, the full rule is examined according to the specifics of the signature.

**Log.** The logging step is where Snort saves any pertinent information resulting from the earlier steps.

Snort evaluates rules in a two-step process which includes a fast pattern search and full evaluation of the signature. Fast patterns are content strings that have the *fast\_pattern* option or which have been selected by Snort automatically to be used as a fast pattern. Snort will by default choose the longest pattern in the rule since that is likely to be most unique. That is not always the case so add *fast\_pattern* to the appropriate content option for best performance.

## Example Snort Commands:

Log packets to a directory:

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

Validate a configuration file and a separate rules file:

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

Tell Snort where to look for additional Lua scripts:

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

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

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

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

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

Run Snort in IDS mode on an entire directory of pcaps on separate threads:

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

Run Snort inline with the afpacket DAQ:

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

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

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

```
react = { } ... reject = { } ... rewrite = { }
```

IPS action *reject* shutdowns hostile network sessions by injecting TCP resets (TCP connections) or ICMP unreachable packets. Example:

```
reject = { reset = "both", control = "all" } local_rules = [[ reject tcp ( msg:"hostile connection"; flow:established, to_server; content:"HACK!"; sid:1; ) ]] ips = { rules = local_rules, }
```

One of the major undertakings for Snort 3 is developing a completely new HTTP inspector. You can configure it by adding: *http\_inspect = { }* to your snort.lua configuration file.

The *http\_inspect* design supports true stateful processing. It allows analysts to examine both the client request and server response, or different requests in the same session. The new concept is that every header should be normalized in an appropriate and specific way and individually made available for the user to write rules against it.

Script detection is a feature that enables Snort to more quickly detect and block response messages containing malicious JavaScript. When *http\_inspect* detects the end of a script it immediately forwards the available part of the message body for early detection. This enables malicious Javascripts to be detected more quickly but consumes somewhat more of the sensor's resources. This feature is off by default. *script\_detection = true* will activate it.

Portscan is a module to detect the first phase in a network attack: reconnaissance (i.e. port scanning). The following are a list of the types of Nmap scans Portscan can alert for. TCP Portscan; UDP Portscan; IP Portscan.

Portscan only generates one alert for each host pair in question during the time window. On TCP scan alerts, Portscan will also display any open ports that were scanned.

There are 3 default scan levels that can be set.

- 1) default\_hi\_port\_scan
- 2) default\_med\_port\_scan
- 3) default\_low\_port\_scan

The default scan level can be set as follows:

```
port_scan = default_low_port_scan
```

**Low** alerts are only generated on error packets sent from the target host, and because of the nature of error responses, this setting should see very few false positives. However, this setting will never trigger a Filtered Scan alert because of a lack of error responses. This setting is based on a static time window of 60 seconds, after which this window is reset.

**Medium** alerts track Connection Counts, and so will generate Filtered Scan alerts. This setting may false positive on active hosts (NATs, proxies, DNS caches, etc), so the user may need to deploy the use of Ignore directives to properly tune this directive.

**High** alerts continuously track hosts on a network using a time window to evaluate portscan statistics for that host. A "High" setting will catch some slow scans because of the continuous monitoring, but is very sensitive to active hosts. This most definitely will require the user to tune Portscan.

Each of these default levels have separate options that can be edited to alter the scan sensitivity levels (scans, rejects, nets or ports)

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

Use the *watch\_ip* option. The analyst should set this option to the list of CIDR blocks and IPs that they want to watch. If no *watch\_ip* is defined, Portscan will watch all network traffic.

The *ignore\_scanners* and *ignore\_scanned* options come into play in weeding out legitimate hosts that are very active on your network. Some of the most common examples are NAT IPs, DNS cache servers, syslog servers, and nfs servers.

Most of the false positives that Portscan may generate are of the filtered scan alert type. So be much more suspicious of filtered portscans. Many times, this just indicates that a host was very active during the time period in question. If the host continually generates these types of alerts, add it to the *ignore\_scanners* list or use a lower sensitivity level.

Make use of the Priority Count, Connection Count, IP Count, Port Count, IP range, and Port range to determine false positives. The portscan alert details are vital in determining the scope of a portscan and also the confidence of the portscan.

The easiest way to determine false positives is through simple ratio estimations. The following is a list of ratios to estimate and the associated values that indicate a legitimate scan and not a false positive.

- Connection Count / IP Count: This ratio indicates an estimated average of connections per IP. For portscans, this ratio should be high, the higher the better. For portsweeps, this ratio should be low.
- Port Count / IP Count: This ratio indicates an estimated average of ports connected to per IP. For portscans, this ratio should be high and indicates that the scanned host's ports were connected to by fewer IPs. For portsweeps, this ratio should be low, indicating that the scanning host connected to few ports but on many hosts.
- Connection Count / Port Count: This ratio indicates an estimated average of connections per port. For portscans, this ratio should be low. This indicates that each connection was to a different port. For portsweeps, this ratio should be high. This indicates that there were many connections to the same port.

If none of these other tuning techniques work or the analyst doesn't have the time for tuning, lower the sensitivity level. You get the best protection the higher the sensitivity level, but it's also important that the portscan detection engine generates alerts that the analyst will find informative.