

3

Dissecting Snort

THE CLICHÉ YOUR GRAMMAR SCHOOL TEACHERS told you, “It’s what’s on the inside that counts!” still applies in the real world. Snort contains many configurable internal components that can vastly influence false positives and negatives as well as general packet logging performance. Knowledge of Snort’s internals is required to make Snort run and monitor for intrusions effectively. Snort is a powerful application, but it takes a little more in-depth research on your part than other, less potent, IDSs. Understanding the function of these internal components will help you customize Snort to your network and help you avoid some of the common Snort pitfalls.

Snort can be divided into five major components that are each critical to intrusion detection. The first is the packet capturing mechanism. Snort relies on an external packet capturing library (libpcap) to sniff packets. After packets have been captured in a raw form, they are passed into the packet decoder. The decoder is the first step into Snort’s own architecture. The packet decoder translates specific protocol elements into an internal data structure. After the initial preparatory packet capture and decode is completed, traffic is handled by the preprocessors. Any number of pluggable preprocessors either examine or manipulate packets before handing them to the next component: the detection engine. The detection engine performs simple tests on a single aspect of each packet to detect intrusions. The last component is the output plugins, which generate alerts to present suspicious activity to you. A simplified graphical representation of the dataflow is shown in Figure 3.1.

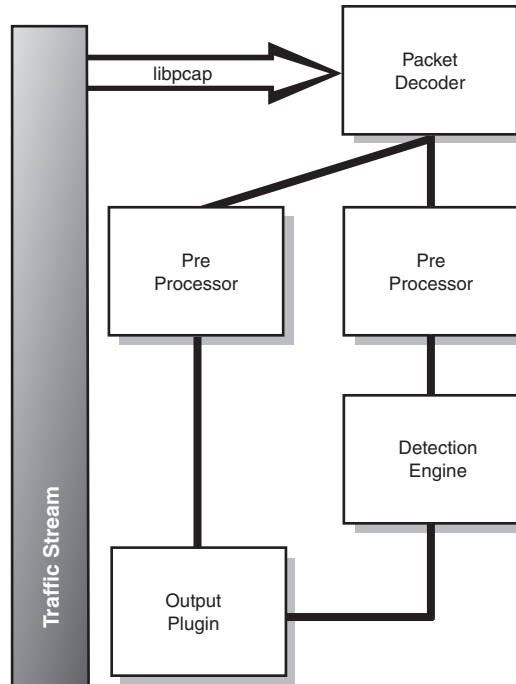


Figure 3.1 Snort component dataflow.

Feeding Snort Packets with Libpcap

To get packets into the preprocessors and then the main detection engine, some prior labor must first occur. Snort has no native packet capture facility yet; it requires an external packet sniffing library: libpcap. Libpcap was chosen for packet capture for its platform independence. It can be run on every popular combination of hardware and OS; there is even a Win32 port—winpcap. Because Snort utilizes the libpcap library to grab packets off the wire, it can leverage libpcap's platform portability and be installed almost anywhere. Using libpcap makes Snort a truly platform-independent application.

The responsibility for grabbing packets directly from the network interface card belongs to libpcap. It makes the capture facility for raw packets provided by the underlying operating system available to other applications.

A *raw packet* is a packet that is left in its original, unmodified form as it had traveled across the network from client to server. A raw packet has all its protocol header information left intact and unaltered by the operating system. Network applications typically do not process raw packets; they depend on the OS to read protocol information and properly forward payload data to them. Snort is unusual in this sense in that it requires

the opposite: it needs to have the packets in their raw state to function. Snort uses protocol header information that would have been stripped off by the operating system to detect some forms of attacks.

Using libpcap is not the most efficient way to acquire raw packets. It can process only one packet at a time, making it a bottleneck for high-bandwidth (1Gbps) monitoring situations. In the future Snort will likely implement packet capture libraries specific to an OS, or even hardware. There are several methods other than using libpcap for grabbing packets from a network interface card. Berkeley Packet Filter (BPF), Data Link Provider Interface (DLPI), and the SOCK_PACKET mechanism in the Linux kernel are other tools for grabbing raw packets.

Packet Decoder

As soon as packets have been gathered, Snort must decode the specific protocol elements for each packet. The packet decoder is actually a series of decoders that each decode specific protocol elements. It works up the Network stack, starting with lower level Data Link protocols, decoding each protocol as it moves up. A packet follows this data flow as it moves through the packet decoder (see Figure 3.2).

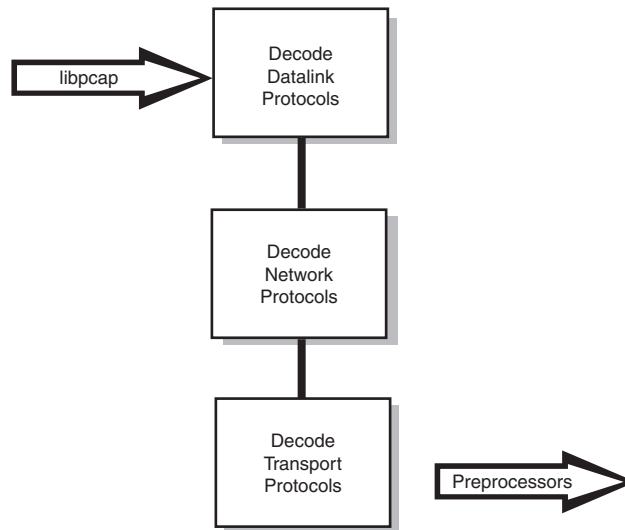


Figure 3.2 Decoder data flow.

As packets move through the various protocol decoders, a data structure is filled up with decoded packet data. As soon as packet data is stored in a data structure it is ready to be analyzed by the preprocessors and the detection engine.

Preprocessors

Snort's preprocessors fall into two categories. They can be used to either examine packets for suspicious activity or modify packets so that the detection engine can properly interpret them. A number of attacks cannot be detected by signature matching via the detection engine, so “examine” preprocessors step up to the plate and detect suspicious activity. These types of preprocessors are indispensable in discovering non-signature-based attacks. The other preprocessors are responsible for normalizing traffic so that the detection engine can accurately match signatures. These preprocessors defeat attacks that attempt to evade Snort's detection engine by manipulating traffic patterns.

Additionally, Snort cycles packets through every preprocessor to discover attacks that require more than one preprocessor to detect them. If Snort simply stopped checking for the suspicious attributes of a packet after it had set off an alert via a preprocessor, attackers could use this deficiency to hide traffic from Snort. Suppose a black hat intentionally encoded a malicious remote exploit attack in a manner that would set off a low priority alert from a preprocessor. If processing is assumed to be finished at this point and the packet is no longer cycled through the preprocessors, the remote exploit attack would register only an encoding alert. The remote exploit would go unnoticed by Snort, obscuring the true nature of the traffic.

Preprocessor parameters are configured and tuned via the `snort.conf` file. The same `snort.conf` file lets you add or remove preprocessors as you see fit.

`frag2`

The `frag2` preprocessor is Snort's weapon against IP fragmentation attacks. Fragmentation is a normally occurring phenomenon in IP networks. It is necessary to successfully send traffic over different types of network media. Different network-based protocols have divergent rules for the maximum allowable size or maximum transmission unit (MTU) for datagrams on their networks.

Fragmentation occurs normally when a packet's payload exceeds the MTU. On an ethernet network, any IP datagram larger than 1,500 bytes has to be fragmented. Fragmentation does not have to happen exclusively at the origination point; it can occur at an intermediate router.

After packets are fragmented, they must be reassembled at the target host. It is the reassembly process that hackers exploit to perpetrate attacks. The receiving host expects the sender to follow some rules when fragmenting. They are:

- The receiving host reassembles packets by associating a fragment with an identical fragment identification number, or fragment ID. The fragment ID is a copy of the IP identification number in the IP header.
- Each fragment must carry its position or offset in the original unfragmented packet (that is, the first fragment will have an offset of 0).
- Each fragment must display the amount of data carried in the corresponding fragment.

- If the fragment is not the last fragment to be received, it must flag the more fragments (MF) bit.

As you can see, any host that expected these rules to be followed at all times could be exploited by a malicious hacker. One simple attack devised to bypass firewalls and filtering devices uses fragmentation to overwrite TCP header data. The attack works by sending a fragmented TCP packet with header information that is allowed through the firewall. Subsequent fragments contain malicious data that would not otherwise be allowable through the firewall. If the fragmentation offset is small enough, though, the malicious packet overwrites the header information, allowing the malicious traffic through the firewall. An example is overwriting the header to change the destination port number. The attacker might change traffic from normal port 80 HTTP traffic to attack RPC on TCP port 111.

A good proportion of fragmentation attacks are DoS attacks. Host operating systems have been discovered to react unexpectedly to crafted fragmented packets. Probably the most famous type of these attacks was the Ping of Death attack. This attack used many small fragmented ICMP packets, which, when reassembled, exceeded the maximum allowable size for an IP datagram of 65,535 bytes. This caused most operating systems vulnerable to the Ping of Death to crash unexpectedly.

A hacker tool named `fragroute` uses fragmentation techniques to hide malicious traffic from Snort. `fragroute` combines many of the previously known fragmentation evasion tactics into one tool. The various techniques can be mixed and matched together, making reconstruction of the fragmented attacks difficult. Fortunately, `frag2` can now detect these types of attacks with some additional configuration options.

The `frag2` preprocessor can detect attack types that are related to fragmentation, whether they are IDS evasion techniques or malicious DoS attacks. The `frag2` preprocessor is important and should never be disabled. There are five options for configuring `frag2`. They are described in the following sections.

`timeout` *Seconds*

This option sets the number of seconds that a fragment is to be saved. If the fragment is not completed in the defined time allowance, it is dropped. Sixty seconds is the default, and should work well for most networks. If you are monitoring a network prone to IP fragmentation, such as a network with an abundance of NFS traffic, it is recommended that you reduce this threshold to avoid false positives. Most external monitoring sensors should not see much fragmented traffic. Some crafty black hats know that 60 seconds is the default setting for `frag2`, and use this information to evade `frag2`. It is recommended that you increase this threshold slightly to 65 seconds to combat this.

`memcap` *bytes*

The memory cap option limits the amount of memory the `frag2` preprocessor can utilize. The default is 4,194,304 bytes, which is suitable for most fragmentation situations. Once again, if you are monitoring a network with heavy fragmentation traffic, it would be prudent to increase this amount.

`detect_state_problems`

This simple configuration option enables detection of overlapping fragments. Overlapping fragments have an offset that overwrites a previous fragment. They are used in firewall bypassing attacks, described previously in this section.

`min_ttl number`

The `min_ttl` configuration option specifies the minimum time to live (TTL) that `frag2` will accept. Anything lower than `[number]` will be dropped. The default setting is 0. TTL is a death counter for packets. Every time a router forwards a packet, it decrements the TTL value and forwards it on. If the TTL reaches 0, the packet is dropped and no longer routed. TTLs were put in place to avoid situations where packets would get caught in endless loops, which could potentially create a Denial of Service situation.

Tools such as `fragroute` use small TTL values to set the expirations for datagrams before they reach the target host, but Snort still picks them up. This scenario is possible when a sensor lies at an intermediate point between the attacker and the target. If the target is two hops away from the sensor, the sensor and not the target interprets any packets with a TTL of 1. Packets with a TTL of 1 expire and are not forwarded on to the target. This can have the effect of inserting garbage data into an attack signature, making the signature not discernable by Snort. Snort is unable to detect the attack. Imagine that three packets make up an attack on an FTP server:

```
Packet 1 - QUOTE SITE - TTL = 55
Packet 2 - inserted_garbage data - TTL = 1
Packet 3 - EXEC EXEC echo toor::0:0:::/bin/sh >> /etc/passwd - TTL = 122
```

Snort sees the attack as the following:

```
QUOTE SITEinsert_garbage dataEXEC EXEC echo toor::0:0:::/bin/sh >> /etc/passwd
```

Because of the *insert garbage data*, Snort cannot match a rule to this signature. However, the host still sees the real attack because the packet containing the garbage data will expire and not be forwarded:

```
QUOTE SITE EXEC EXEC echo toor::0:0:::/bin/sh >> /etc/passwd
```

Dropping packets that have a very low TTL can avert this type of evasion. The recommended setting is 3.

`t1_limit number`

This option specifies the maximum difference in TTL values that fragmented packets with the same fragment ID can have. The default is 5. This option was added because tools like `Fragroute` utilize inconsistent values for fragmented packets to evade Snort. They are vastly different than what is found in normal Internet traffic. The average change in TTL values for normal Internet traffic is somewhere in the 5–7 range. Leaving

this set at the default of 5 creates some false positive noise; it is recommended to set `ttl_limit` to 8.

stream4

The `stream4` preprocessor is what Snort uses to maintain the state of TCP streams, which is used in detecting some types of information gathering attacks. Stateful inspection with `stream4` helps Snort better match attack signatures across multiple packets. This is important because attacks can be spread over many individual packets or exhibit anomalies in terms of TCP connections. The `stream4` preprocessor also lets you create rules that are state-aware via the *flow* option.

A simple example can be demonstrated with the WU-FTP exploit. A vulnerability was discovered for all versions of WU-FTP, which is included by default in most major Linux distributions. With this exploit, any user who can log into a vulnerable version of the WU-FTP server can execute arbitrary code remotely with root access. The vulnerability is due to a combination of coding errors, one located within the function responsible for the remote command globbing feature, which fails to properly signal an error to its caller under certain conditions. The globbing function does not properly handle the string "`~{`" as an illegal parameter.

To detect the first phase of this attack, a rule must be created to trigger any time a signature matched "`~{`". This works fine if the entire content of the attack is in the same packet. If the attacker causes the signature to be split up, so that `~` is in one packet and the `{` is in the next, the signature does not trigger. With `stream4`, Snort stores the `~` portion of the signature. When the attacker sends the final piece, `{`, in a subsequent packet, Snort completes the signature and an alert is generated. Without stream reassembly this would not be possible.

The `stream4` preprocessor can also detect attackers doing unusual things with TCP streams in an attempt to hide information-gathering traffic. One such method of reconnaissance is a TCP stealth or half-open scan. In this method, the TCP three-way handshake is never fully completed. The attacker first sends a SYN packet as normal. When a SYN-ACK is received from the host, the service is assumed to be listening. The attacker never completes the handshake by responding with the expected ACK packet. This type of scanning could trick an IDS that is not monitoring state.

The `stream4` preprocessor is the next evolutionary step up from the `stream` preprocessor. It had to be expanded to deal with a type of stateless attack specifically created to create a Denial of Service condition with Snort. Snot attacks take a Snort's own rule-set and generate a flood of randomly chosen attack signatures. When this flood is aimed at Snort, it creates a massive amount of false positives. A Snot attack can be used to bury a legitimate attack in the noise. When a Snot attack is run, it completes only the first phase of the three-way handshake, similar to the stealth scan. You can configure `stream4` in conjunction with a command-line option to defeat Snot attacks. With the `-z` command-line option specified with the "`est`" argument, Snort ignores TCP streams that do not demonstrate normal bidirectional activity.

The `stream4` preprocessor was also rewritten to incorporate higher bandwidth conditions. With modern hardware, `stream4` can handle stateful inspection for up to 64,000 simultaneous connections. `stream4` supports 10 different configurable options, described in the following sections.

`detect_scans`

This option simply enables `stream4` for portscan detection. This option detects normal TCP connect scans and stealthy scans, such as the Xmas Tree, Half Open, and SYN-FIN scans. It can detect a variety of other attempts at stealthy scanning.

`detect_state_problems`

With this option enabled, `stream4` alerts on various state inconsistencies that black hats have in their arsenals. One of the many possible attacks this configuration option detects is a TCP reset packet evasion attempt. This works by sending forged TCP reset packets each time a TCP connection is made. If this option is not enabled, Snort might perform stream reassembly incorrectly and assume that the connection has been terminated. Snort would then think that the session had been torn down and not log future data relating to this stream. Malicious traffic could be hidden from Snort in this manner.

`disable_evasion_alerts`

This configuration option disables some of the alerting that Snort does for aged, less widely used IDS evasion attempts. TCP overlapping is one of these methods. It works by creating connections with overlapping but incongruous data. This may cause older intrusion detection systems to misinterpret the intent of the connection. TCP overlapping was used to create both false positives and negatives in older IDSs. Recent TCP implementations of Windows 2000 and XP send status information in this manner, which results in a noticeable number of false positives. For this reason, it may be advisable to enable this option if the number of false positives becomes too much to bear. With any of the options beginning with “disable” in Snort, it is always best to leave them enabled unless they present an unmanageable number of false positives.

`min_ttl number`

Nearly the same as the option in `frag2`. This option drops packets with a TTL less than *number*. A short TTL attack has a short enough TTL set for some packets in a stream. A router may then cause some of the packets with TTLs of zero to expire before they reach the target. If Snort is only one hop away from the target and does not check the TTLs, it will reassemble the TCP stream badly. The recommended setting for *number* is 3.

`ttl_limit number`

This option specifies the maximum difference in TTL values that fragmented packets with the same fragment ID can have. The default is 5. Fragroute utilized inconstant TTL values for fragmented packets that were way out of the range of normal Internet traffic. The average change in TTL values for normal Internet traffic is somewhere in the

5–7 range. Leaving this set at the default of 5 creates some false positive noise; it is recommended to set *number* equal to 8.

`noinspect`

This option disables stateful inspection of TCP streams. The `noinspect` command should be utilized only in special situations, such as testing in very high bandwidth or limited hardware.

`keepstats machine or binary`

This option logs session summary information to `session.log` in Snort's log directory. The `machine` switch logs in a flat file; the `binary` switch logs in unified binary output format. This log is useful for additional event correlation and forensics.

`timeout seconds`

This option sets the number of seconds for a which a session is to be saved. If the session is not completed in the defined time allowance, it is flushed. Thirty seconds is the default. If you are monitoring a network with heavy session traffic, it is recommended to increase this threshold to avoid false negatives. Some crafty black hats know that 30 seconds is the default setting for `stream4`, and use this to evade `stream4`. It is recommended that you increase this threshold slightly to 35 seconds to combat this.

`memcap bytes`

The memory cap option limits the amount of memory the `stream4` preprocessor can utilize. The default is 8,388,608 bytes, which is suitable for normal networks. Once again, if you are monitoring a network with heavy session traffic, it would be prudent to increase this amount.

`log_flushed_streams`

This option is used to log complete streams to disk that contain a packet that matches a signature. Without this option, Snort logs only the specific packet(s) that match the signature. This option works only when Snort is run in pcap mode.

`stream4_reassemble`

The `stream4_reassemble` preprocessor is closely related to `stream4`. It performs TCP stream reassembly as described earlier. Malformed packet injection is one of the many reassembly attacks. Malicious hackers attack the TCP stream reassembly process by creating packets with some valid header data such as sequence numbers, ACK numbers, source ports, and destination ports. The forged packets have a bogus payload and a checksum that fails. These crafted packets are inserted between totally valid packets. The target drops the injected crafted packets, and IDSs that do not do TCP checksum verification when reassembling packets miss the attack. Fortunately, Snort has `stream4_reassemble` to catch these attacks.

The `stream4_reassemble` preprocessor has five different configuration options, described in the next few sections.

`clientonly`

The default setting for `stream4_reassemble` is `clientonly`. This setting reassembles sessions only if they originate from the client side. The client network is defined elsewhere in the `snort.conf` file. Chapter 6 examines the definition of clients, servers, and other important hosts.

`serveronly`

The inverse of the `clientonly` option is `serveronly`. The `stream4_reassemble` preprocessor reassembles only sessions originating from the server side.

`both`

This reassembles from both client and server directions.

`ports list`

The `ports` option is used to specify which TCP destination port traffic is reassembled. The default is set for ports 21/FTP, 23/Telnet, 25/SMTP, 53/DNS, 80/HTTP, 143/IMAP, 110/POP, 111/RPC, and 513/rlogin. You can enable this port list by specifying `default` for the `[list]` parameter. You can enable reassembly for every port by specifying `all`. If you have a custom list of TCP ports on which you would like reassembly, you can list them, separated by spaces.

`noalerts`

This option disables alerting for reassembly attacks and evasions. If you are generating an unhealthy number of false positives related to TCP session reassembly, this should be your last resort. It is recommended that you attempt to narrow down the traffic with the `ports`, `serveronly`, and `clientonly` configuration options before disabling alerts altogether.

`HTTP_decode`

The `HTTP_decode` preprocessor is responsible for detecting abnormal HTTP traffic and normalizing it so that the detection engine can properly interpret it. Normalizing traffic is the process of translating an obscure character set, such as Unicode or hex, to a character set that Snort can recognize. This is necessary for Snort to be able to match signatures to malicious content. `HTTP_decode` works specifically with the URI string of an HTTP request. It generates an alert if it encounters traffic that requires decoding.

Encoding or obfuscating HTTP traffic is a method that hackers can use to disguise an attack from an IDS or even the human eye. Without `HTTP_decode`, an attack that Snort would normally catch can be obfuscated in a manner that does not match a signature, but that the target Web server still accepts as a valid URL string. Using Microsoft's IIS %u encoding, an .ida attack that would normally match a signature can be easily hidden.

Suppose you have a Snort rule designed to trigger on any URI content that matches `.ida`. An attacker could use `%u` encoding to hide the `a` and evade Snort with the following URL request:

```
GET /vulnerable.id%u0061 HTTP/1.0
```

After the request evaded Snort, IIS would translate this to

```
GET /vulnerable.ida HTTP/1.0
```

Encoding URLs in this manner can be used to obfuscate any type of malicious HTTP URI request. Encoded URLs are also used to coax unsophisticated Internet users into clicking on malicious hyperlinks. Cross-site scripting attacks sometime require a user to click on a link for an authentication token to be delivered to a host controlled by the attacker. These links are often encoded in hex to increase the chances of success. The `HTTP_decode` preprocessor can detect users clicking on hex-encoded links.

Moving beyond obfuscation, some malicious attacks utilize encoded strings. Programmers implementing application-level access controls and bug fixes often forget about encoded URI strings. The Unicode directory traversal sample attack in Chapter 1 is a good example of a coding error that resulted in a security exposure. Even though directory traversal attacks had been patched, they were still vulnerable when the `/` symbol was encoded in Unicode.

`HTTP_decode` has five different configuration options that relate to different encoding strategies attackers employ.

port list

This option is the list of source ports where `HTTP_decode` is to look for encoded HTTP URI requests. The default is 80. Any Web server ports should be listed here, delimited by a space.

unicode

Specifying the `unicode` option normalizes all Unicode strings to ASCII. Leave this option off if you do not want alerts for Unicode URI traffic.

iis_alt_unicode

Enabling this option allows `HTTP_decode` to detect `%u` encoding. Microsoft's IIS supports a little-known, proprietary method of HTTP URI encoding known as `%u` encoding. The legitimate purpose of this `%u` encoding was to represent true Unicode character strings. Because `%u` encoding is proprietary and not an IETF standard, most IDS systems are unaware that `%u` encoding existed. Attackers can use `%u` encoding to hide IIS attacks that would have otherwise triggered an alert. Not including this configuration option causes Snort to not generate alerts for `%u` encoding.

double_encode

This option detects possible double encoding attempts and normalizes double decoding attempts. In a double encoding attack, the cracker forwards an encoded HTTP URI

request that has been through two rounds of encoding. The IDS detects the first round of encoding and normalizes it. The attack still evades the IDS, however, because the result of the first normalization outputs an encoded string that does not match a signature. Without this option enabled, Snort decodes a string only once and assumes there is no more nested encoding. For example, if an attacker wanted to pass in the infamous `\` character with double encoding, she would encode the character in hex as `%5c`. To double encode, the attacker hex encodes the `%5c` string again. The attacker hex encodes `%` to `%25`, `5` to `%35`, and `c` to `%63`, resulting in `%25%35%63`. Thus, the cracker has double encoded the `\` character as `%25%35%63`.

`iis_flip_slash`

This option detects and decodes directory separators that are obscured through the use of a Microsoft proprietary directory separator. Microsoft OSs separate directories using a `\` instead of the standard `/`. A HTTP request requires that the direction separator be a `/`. This means that IIS, as well as all other Win32-based Web servers, must convert the directory separator to a `/`. IIS still allows the use of `\` in HTTP requests, because it is considered a valid directory separator. With IIS, directory separators can be encoded as `/scripts\admin.asp`, which does not match a typical `/scripts/admin.asp` signature.

`full_whitespace`

This option generates alerts and normalizes strings that are obscured by the use of the `/t` method for formatting HTTP requests to an Apache Web server. A standard HTTP request uses spaces to separate different portions of the request. It looks like this:

```
method <whitespace> URI <whitespace> HTTP/ Version
```

The `<whitespace>`s are used as delimiters for extracting specific portions of the HTTP request, such as the URI string. Snort utilizes this method for extracting URI strings to match to signatures.

Apache 1.3.6 and newer permits a different nonstandard request:

```
method /t URI /t HTTP/ Version
```

This request evades any IDS that expects an HTTP request to be separated by `<whitespace>` rather than `/t` or a `<tab>`.

`RPC_decode`

This preprocessor has a similar function as `HTTP_decode`, but with another protocol: RPC. RPC can be used by black hats for both reconnaissance and remote exploit attacks. Attackers can use port mapping applications, such as `rpcbind` and `portmapper`, that make dynamic binding of remote services possible. The attacker can use information gathered from `rpcbind` to find additional targets for buffer overflows, or the attacker can attack the RPC service itself.

Malicious hackers desiring to hide RPC traffic can break up the RPC signature. The RPC signature, 0186A0, can be split up to obscure an RPC request. If the signature is spread out over many packets, Snort cannot match a valid signature. For this reason, RPC requests are normalized with the `RPC_decode` preprocessor. `RPC_decode` has one configuration option: `[ports list]` is the list of RPC ports to be normalized by `RPC_decode`. The default is 111 and 32771.

BO

The `BO` preprocessor detects Back Orifice. Back Orifice is a remote control Trojan for Windows systems. Back Orifice was a very popular Trojan when it was released, but has recently been replaced by other more advanced Trojans. Back Orifice still remains a considerable threat; it can be used by script kiddies to take total control of a remote system.

This preprocessor detects Back Orifice UDP traffic when commands are issued to the Trojan. It works by detecting the “magic cookie” that Back Orifice servers and clients require to communicate with each other. A fairly weak encryption algorithm is used to encrypt `BO` traffic. The entire keyspace of the Trojan’s encrypted communication protocol can be brute forced with little effort. The `BO` preprocessor supports the brute forcing of Back Orifice’s entire keyspace. It can also be configured to search for packets that use the default installation encryption key, 31337, to search for packets. Searching solely for the default key improves the performance of `BO` and Snort in general, but allows Back Orifice communication to slip by unnoticed if a key other than the default is used for encryption. `BO` supports two configuration options.

number

This option is used to set the default key used to decode Back Orifice–encrypted communication to something other than the default of 31337.

`-nobrute`

As explained previously, this option disables the brute forcing of the Back Orifice key-space for a valid encryption key. This is the default setting. This setting should not be enabled unless `BO` is causing a significant performance impact on Snort.

Telnet_decode

This is another one of the family of decoding preprocessors. This preprocessor specifically relates to Telnet and FTP protocols. `Telnet_decode` decodes or removes arbitrarily inserted binary Telnet control codes in a Telnet or FTP stream. Malicious hackers insert control codes into communication in an attempt to evade Snort’s watchful eye. Control code insertion is often performed with the `SITE EXEC FTP` command and its associated vulnerabilities. The FTP command `SITE EXEC` is used by attackers to execute system commands via an FTP connection. An attacker could enter an FTP command like so to retrieve a passwd file:

```
QUOTE SITE EXEC EXEC echo toor::0:0:./:/bin/sh >> /etc/passwd
```

More sophisticated hackers intersperse Telnet control codes into the command hoping to evade Snort. `Telnet_decode` prevents this by removing or decoding the malicious control codes. `Telnet_decode` has no configuration options.

ARPspooF

`ARPspooF` is a preprocessor designed to detect malicious Address Resolution Protocol (ARP) traffic. ARP is used on ethernet networks to map an IP address to a hardware MAC address. To reduce the number of ARP broadcasts on modern networks, operating systems of connected devices store a cache of ARP mappings. When a device receives an ARP reply, it updates its ARP cache with the new IP-to-MAC address mapping whether or not the device sent the ARP request.

Various attacks involve ARP; the basis of them is ARP spoofing. ARP spoofing is accomplished by crafting ARP request and reply packets. Forged ARP reply packets are stored in the ARP cache of the receiving device even if the device did not send the request.

ARP spoofing can be used to misdirect traffic, making sniffing possible on switched networks. Forged ARP replies can trick a target device into misdirecting ethernet frames intended for a legitimate device to the attacker's choice of device. The legitimate device is not aware that misdirection had taken place. ARP spoofing can also be used to perform a simple yet effective DoS. The DoS happens when spoofed ARP replies that contain invalid IP-to-MAC mappings flood a device's ARP cache. This causes the target device to send traffic to non-existent devices.

The `ARPspooF` preprocessor can detect some of these types of attacks and ARP spoofing attempts. `ARPspooF` detects spoofing attempts by determining that an ethernet source address is different from the one included in the ARP message, indicating a spoofed ARP reply. Conversely, `ARPspooF` detects a destination address that differs from the one in the ARP message, indicating a spoofed ARP request.

Another type of attack `ARPspooF` detects is the ARP cache overwrite attack. The attack works by sending ARP packets received by the device for the device's own interface address but a different MAC address. This overwrites the device's own MAC address in the ARP cache with the malicious ARP request. This causes the device to be unable to send and receive any ARP packet. In turn, this causes the device and any other devices that depend on it for communication to be unable to send packets to each other.

Because ARP is a Layer 2 protocol, `ARPspooF` detects only those attacks occurring on the same physical segment as the Snort sensor. `ARPspooF` has two configuration options.

host IP address host MAC address

Each device that you wish to monitor with `ARPspooF` must be specified with its own IP-to-MAC address mapping. Each one of the devices is listed on a new line in the `snort.conf` file. Any time the mapping changes, you must reconfigure the file. Devices that obtain their IP address via DHCP should be converted to static IPs before `ARPspooF` is enabled.

`-unicast`

This option will enable detection of ARP `unicast` attacks. Most valid ARP requests are sent to broadcast addresses. ARP requests that are sent to a Unicast address are often the sign of an attack designed to modify ARP caches. This option is disabled by default, but should be enabled for serious ARP misuse monitoring.

`ASN1_decode`

`ASN1_decode` is a preprocessor designed to detect various inconsistencies in ASN.1 that may indicate malicious behavior. ASN.1 or Abstract Syntax Notation One is an international standard for coding and transmitting complex data structures. ASN.1 protocol is used by a diverse set of higher-level protocols including LDAP, SNMP, SSL, and X.509. ASN.1 is also used for communication in proprietary government and industrial applications. Portions of the U.S. power grid are controlled remotely by the ASN.1 protocol. The SS7 network that controls telephone call routing, parcel delivery, and credit card verification systems all use ASN.1. Vulnerabilities associated with ASN.1-based networks are a matter of national security and are a major concern.

A Finnish group of researchers at Oulu University discovered a wide range of ASN.1-related vulnerabilities. They were able to exploit poor coding of ASN.1 on almost every SNMP-enabled device. The exploits operate by deliberately violating the ASN.1 standard in thousands of different ways. Most of these violations take the format of extreme oversizing or undersizing of ASN.1 messages. These cause the higher-level protocol dependent on ASN.1 to crash or create a situation where a buffer overflow occurs. This may have the effect of creating a Denial of Service condition or remotely controlling the attacked device.

The `ASN1_decode` preprocessor detects various attempts to attack ASN.1-enabled hosts by breaking the rules of the ASN.1 protocol. The OpenSSL ASN.1 DoS is an example of the type of attack `ASN1_decode` is designed to detect. OpenSSL had a vulnerability in its implementation of the ASN.1 library. This vulnerability, due to parsing errors, affects SSL, TLS, S/MIME, PKCS#7, and certificate creation routines. The certificate encodings can cause a Denial of Service to server and client implementations that depend on OpenSSL-to-secure communication over public networks.

`ASN1_decode` is an experimental beta preprocessor that may generate more false positives than other preprocessors. It has no configuration options at this time.

`fnord`

The `fnord` preprocessor is designed to detect and defeat polymorphic shell code evasion attempts. Polymorphic shell code is a new twist on an old idea. Virus creators have been designing polymorphic viruses for years. Polymorphic viruses change their signature randomly to avoid detection by static signature-based Antivirus applications. A similar method of randomizing remote exploit shell code was developed recently to evade IDSs, and released with the ADMutate tool.

ADMutate evades Snort or any other IDS by randomly polymorphing the buffer overflow signature by choosing random no-op instructions and encrypting the shellcode portion of the exploit. A buffer overflow is divided into two sections: a no-op pad or no-op sled and the shellcode payload. The no-op pad is a large section of no operation codes (that is 0x90 for Intel) specific to the target architecture. The shellcode payload is a portion of the buffer overflow where the shell gets executed and bound to a TCP port. Snort and other IDSs use the no-ops and the shellcode signature to detect buffer overflow exploits.

ADMutate performs two actions on the buffer overflow exploit to disguise it. It randomly replaces the no-op instructions with other no-effect instructions that are functionally similar to a no-op. It then encrypts the shellcode with a fairly simple encryption algorithm (usually xor or a double xor). The purpose of encrypting is not to protect the shellcode, but rather to obscure it in a random fashion. ADMutate has to include the decryption routine with the newly formulated exploit. The decryption routine is randomly created through the use of methods similar to those employed by polymorphic viruses. This process makes the no-ops, the shellcode, and the newly added decryption engine impossible to detect for normal signature-based IDSs. Any attacker using ADMutate on previously existing exploits can happily hack away without Snort noticing.

To defeat polymorphic shellcode attacks, the `fnord` preprocessor was created. It detects a large amount of no-effect instructions grouped together for Intel, Sparc, or HP hardware architectures. It is effective at detecting polymorphed exploits, but generates some false positives.

The `fnord` preprocessor is relatively new and is potentially computationally expensive. It has no configuration options.

`conversation`

The `conversation` preprocessor is similar to the `stream4` and `stream4_reassemble` preprocessors. It is designed to extend session tracking beyond TCP. It tracks pseudo-sessions or conversations for any other IP protocol (such as ICMP). This is useful for detecting where communication has originated. It can also detect when new communication commences. The `conversation` preprocessor must be enabled for the `portscan2` preprocessor to function.

As of now, `conversation` is used only in conjunction with `portscan2` to detect information gathering attempts. It has three configuration options.

`allowed_protocols port list or any`

This is the list of IP protocols for which Conversation will track sessions. It can be configured with either a list of IP protocols or the `any` option. The number list corresponds to the IP protocol number defined in RFC 791. Some of the most common are 1/ICMP, 6/TCP, and 17/UDP. The `any` option tracks all protocols.

`timeout`

This option sets the number of seconds that a conversation will be saved. If the conversation is not completed in the defined time allowance, it is dropped. The default is 60 seconds. Some crafty black hats know that 60 seconds is the default setting for conversation and use this as an evasion technique. It is recommended to increase this threshold slightly to 65 seconds to combat this attack.

`max_conversations numbers`

This is the maximum conversations that `conversation` will support simultaneously. The default is set to 65,335, which is the recommended value.

`portscan2`

The `portscan2` preprocessor is the successor to the original anomaly detection `portscan` preprocessor. It functions by anomaly detection as well. The original `portscan` detected reconnaissance attempts by alerting whenever packets were seen going to four different ports in less than three seconds. This set off a lot of false positives. Even though you could specify which IP addresses to ignore, a fair amount of Snort regulars would disable the preprocessor altogether. It was not a bad preprocessor by any means; it just needed some evolutionary improvements. Fortunately, `portscan2` is here to avenge the death of his father.

The `portscan2` preprocessor works in concurrence with `conversation` to track sessions for IP protocols. It has configurable options that enable it to detect different methods of scanning that are anomalous for most IP networks. It can detect vertical portscans, or scans that sweep the entire range of ports for one host. It can also detect horizontal portscans, which scan one port across many hosts. The `portscan2` preprocessor is not as prone to false positives because it has more configuration options. It utilizes `conversation`, and is therefore aware of state. It has five different configuration options.

`scanners_max number`

This option sets the maximum number of possible conversations to track for reconnaissance attempts. The default setting is 3,200.

`targets_max number`

This is the maximum number of target hosts to be monitored. The default setting is 5,000. Even if you do not have this many possible targets, it is a good idea to keep this setting high. The attacker may be blindly scanning for IP addresses that have no host bound to them, and you would not want to miss this type of information gathering attempt.

`target_limit number`

This configuration option is the horizontal portscan threshold. It is the maximum number of unique hosts the conversation can attempt to access before generating an alert. The default is 5, and is the recommended setting.

port_limit number

This is the vertical portscan threshold. It is the maximum number of unique ports the conversation can attempt to access before generating an alert. The ports do not have to be on the same host to generate the alert. The default setting is 20, and is recommended.

timeout seconds

This configuration option is the length of time to hold a conversation in memory before flushing it. This is the amount of time you are giving `portscan2` to generate an alert. The default setting is 60, and is recommended.

SPADE

SPADE (Statistical Packet Anomaly Detection Engine) is a preprocessor that detects suspicious traffic via anomaly detection methods. SPADE determines whether traffic is anomalous by building a table to describe normal traffic patterns. When traffic does not match data stored in the table, SPADE outputs an alert.

SPADE functions by assigning all incoming packets an anomaly score. High anomaly scores are assigned to specific packets that are rare in occurrence. SPADE defines a unique packet by combining packet IP and port information. For instance, a packet with a destination IP of 192.168.1.1 and destination port of 443 is defined as one packet.

A table is constructed that reflects the occurrences of different kinds of packets over time. Recent packets are given precedent over aged packets. Over time, an alerting threshold is defined for SPADE. For each packet that exceeds this threshold, an alert is generated.

SPADE has five configuration options.

anom-report-thresh

This is the threshold at which packets are reported when the Snort process starts. If a packet has an anomaly score that exceeds this threshold, an alert is generated. If configured with `-1`, SPADE starts with no default alerting threshold. The default is `-1`.

filename

This is the filename to which alerts will be written. If SPADE is used in conjunction with an output plugin, alerts are sent as expected through the plugins.

statefile

This file is used to store the state of SPADE's anomaly table. The statefile is used to keep the anomaly table intact when the Snort process is refreshed or restarted. You can set this option to 0 if you do not wish to store a statefile.

checkpoint-freq

The option to configure how often the statefile is refreshed. *Checkpoint-freq* is the number of packets to accept before refreshing the table.

probability-mode

This is a configurable list of probability modes to use in detecting anomalous behavior. The four different modes define a SPADE “packet” differently. You configure this by choosing one of the related numbers.

- 0: A Bayes network approximation of the source IP, source port, destination IP, and destination port.
- 1: Source IP, source port, destination IP, and destination port.
- 2: Source IP, destination IP, and destination port.
- 3: Destination IP and destination port.

The default setting is 3.

The Detection Engine

The *detection engine* is the primary Snort component. It has two major functions: rules parsing and signature detection. The detection engine builds attack signatures by parsing Snort rules. Snort rules are read line by line, and are loaded into an internal data structure. The rules are loaded only when the Snort service is started, meaning that to modify, add, or delete a rule you must refresh the Snort daemon.

The detection engine runs traffic through the now loaded rule set in the order that it loads them into memory. You can dictate which rules are run first by prioritizing and then organizing in the manner you see fit. Rules are split into two functional sections: the rule header (rule tree node) and the rule option (option tree node). The rule header contains information about the conditions for applying the signature. You can specify the protocol, source, and destination IP address ranges, the port, and the log type in the rule header. The rule header for the OpenSSH CRC32 remote exploit is:

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 22
```

The rule option for the same exploit begins and ends with a parenthetical. The rule option contains the actual signature, the priority level, and some documentation about the attack.

```
(msg:"EXPLOIT ssh CRC32 overflow /bin/sh"; flow:to_server,established;  
content:"/bin/sh"; reference:bugtraq,2347; reference:cve,CVE-2001-0144;  
classtype:shellcode-detect; sid:1324; rev:3;)
```

The detection engine processes rule headers and rule options differently. The detection engine builds a linked list decision tree. The nodes of the tree test each incoming packet for increasingly precise signature elements. A packet is tested to see whether it is TCP; if so, it is passed to the portion of the tree that has rules for TCP. The packet is then tested to see whether it matches a source address in a rule; if so, it passes down the corresponding rule chains. This process happens until the packet either matches an attack signature or tests clean and is dropped. The important thing to remember is that Snort

commences testing a packet after it has found a signature to match to the packet. Even if the packet could possibly match another signature, the detection engine moves on to the next packet. This is why it is valuable to organize rules so that the most malicious signatures are loaded first. Look for this to change in the near future; Snort's developers are hard at work implementing a last-exit strategy.

Output Plugins

Snort's output plugins are the means Snort has to get intrusion data to you. The purpose of the output plugins is to dump alerting data to another resource or file. Multiple outputting plugins can be activated to perform different functions. Loads of external applications—some even built exclusively for Snort—are designed to read Snort's output and manage intrusion data. Chapters 6 and 11 examine some of these applications.

Output plugins can be a major bottleneck for Snort. Snort can read and process packets quickly, but bogs down when trying to write to a slow database or over a network. Database output plugins are not used in high-bandwidth environments. It is recommended to configure Snort to spool to unified format and let Snort's unified log application, Barnyard, take over. Snort has 12 output plugins that push out data in different formats.

Alert_fast

`Alert_fast` is the quick and dirty outputting mechanism for Snort. It spits out alerts in a one-line file as fast as the detection engine can spawn them. With `Alert_fast` Snort does not write packet headers, making it a fast but brief method of logging. `Alert_fast` takes one configuration option.

[filename] is simply the name of the file for `Alert_fast` to log to. The log is created in the default logging directory (`/var/log/snort`) or the directory specified.

Alert_full

This is a somewhat antiquated logging facility for Snort, but still useful for low-bandwidth networks. `Alert_full` creates a directory for each IP that generates an alert and fills it with decoded packet dumps. It includes the packet headers in the dumps, unlike `Alert_fast`. It has one configuration option.

[filename] is simply the name of the file for `Alert_full` to log to. The log is created in the default logging directory (`/var/log/snort`) or the directory specified.

Alert_smb

`Alert_smb` is another antiquated output plugin. It sends windows SMB requests to Windows machines. If you take perverse pleasure in clearing message boxes hour after hour, this plugin is for you. `Alert_smb` sends alerting information in the clear, and executes an external binary with root privileges, making it a security risk in and of itself. `Alert_smb` has one configuration option.

[alertworkstationlist] is a file listing workstation to be notified. The format of the workstation file is a list of the NETBIOS names of the hosts, one per line.

Alert_unixsock

This plugin sets up a Unix domain socket and sends alerts to it. External programs/processes can listen in on this socket and receive Snort alert and packet data in real time. This plugin does not work with Windows installations, for obvious reasons. `Alert_unixsock` has no configuration options.

Log_tcpdump

`Log_tcpdump` logs packets to the famous tcpdump file format. There exists a wide assortment of applications and tools designed to read tcpdump output. This module enables you to use them in conjunction with Snort. `Log_tcpdump` has one configuration option.

[filename] is the name of the output file. The *[filename]* will have the `<month><date>@<time>` prepended to it. This is to keep data from separate Snort runs distinct.

CSV

The CSV plugin outputs to a comma delimited file. CSV files are easily imported into other databases and spreadsheets. The data separated by commas can be organized in any manner required. You can use CSV to write as few as one field or all 24. CSV takes two configuration options.

[filename]

This is the name of the output file.

[default | field list]

Here you can list out the fields in your desired order. The possible fields are:

timestamp	ethdst	tos
msg	ethlen	id
proto	tcpflags	dgmlen
src	tcpseq	iplen
srcport	tcpack	icmptype
dst	tcplen	icmpcode
dstport	tcpwindow	icmpid
ethsrc	ttd	icmpseq

You can modify this list to output the fields you want in the order you desire. If you specify default, CSV outputs a comma-separated file in the preceding order.

XML

The XML plugin allows you to log to Simple Network Markup Language (SNML). SNML can be used to collect data from many different sensors in a single management database. The XML plugin logs via encrypted or plain text HTTP sessions. XML has a unique sanitization feature. With the sanitization feature enabled, all IP addresses will be outputted with the mask, XXX.XXX.XXX.XXX. You can then anonymously submit your intrusion data for review by other entities, such as the CERT/CC (check www.cert.org for more information). CERT/CC can use this data to predict trends and monitor malicious activity on a global scale. The XML plugin has a good number of configuration options.

log or alert

This is used to specify whether to attach the XML output plugin to the log or alert facility. Certain rules are set to “alert” while others are set to “log”; this option outputs the chosen rule type.

parameter list

This option is a monster list of possible parameters for configuring XML to function in the way you want it to. Parameters are set in the familiar *name=value* pair. A list of names and descriptions follows:

- *file*—Filename for XML output. If this is the only parameter entered, it writes to disk, to a local file.
- *protocol*—Used to specify the protocol for logging to a remote host. Possible values are
 - *http*—Send output via HTTP posts; requires *file* parameter
 - *https*—Send via encrypted HTTPS posts; requires *file*, *cert*, and *key* parameters
 - *tcp*—Send via a TCP port; requires an external application to listen to the TCP port
- *host*—The remote host to send logs to.
- *cert*—The client certificate to be used for HTTPS communication.
- *sanitize*—An IP address range with netmask combination. The range specified will be sanitized. You can use this option several times to sanitize multiple ranges. With any alerts that are sanitized, packet payloads are not logged.
- *hex*—Include this name with no value to store binary data in hex.
- *base64*—Include this name with no value to store binary data in base64 encoding.
- *ascii*—Include this name with no value to store binary data in human-readable ASCII encoding.

- *detail*—Can be either full or fast. With full you log all details of the packet that Snort has available. Fast logs only timestamp, signature, source and destination IPs, source and destination ports, TCP flags, and protocol.

Alert_syslog

`Alert_syslog` writes to the syslog facility. A syslog server can be used to collect logging information from a variety of different devices not related to Snort, such as routers, firewalls, Web servers, and so on. Inputting Snort intrusion data into a syslog server can vastly aid in event correlation and problem identification. Writing alerting data to syslog can be used in conjunction with other tools to perform real-time alerting and notification. `Alert_syslog` is one of the most popular outputting plugins.

This plugin cannot be used to log to a remote syslog server if the sensor is installed on a Windows machine. You must use the command-line option for syslog (`-s`) to write to a remote syslog server. `Alert_syslog` supports the standard three syslog configuration options.

facility

This is the facility parameter that will be assigned to `Alert_syslog` output in the syslog server. The default is `LOG_AUTH`. You may want to use `LOG_AUTHPRIV` if others will be working with the syslog server but should not have access to intrusion data.

priority

This is the priority parameter that will be assigned to `Alert_syslog` output in the syslog server. The default is `LOG_ALERT`. This setting is a good fit for alerts; it is included with other critical events.

options

This is the option parameter that will be assigned to `Alert_syslog` output in the syslog server. The default is not assigned. You can use `options` to write directly to the console, or to log Snort's process ID with each alert. The `options` parameter can be used to identify multiple Snort processes on a single machine.

Database

The database output plugin logs directly to a relational database of your choice. It supports MySQL, PostgreSQL, Oracle, and UnixODBC-compliant databases such as SAPdb. Outputting to a relational database makes large amounts of intrusion data accessible. When the database plugin is placed into a database, alerts can be sorted, searched for, and prioritized in an organized manner. There are several applications that use the intrusion data in a relational database to create a management GUI. Snort data is written to the database in the table structure shown in Figure 3.3.

Snort Database ER Diagram (version 1.03): snort 1.8

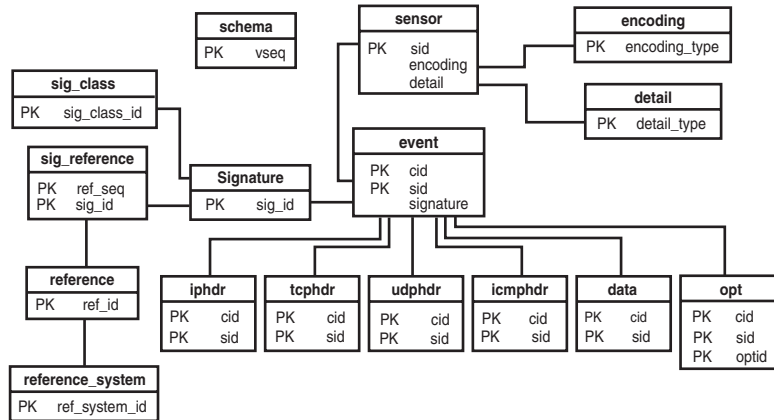


Figure 3.3 Snort database structure.

The database plugin can serve as a bottleneck in even moderately saturated networks. The plugin itself is not a bottleneck; rather, the database being written to is. When the database plugin writes to the chosen database, it must wait until the database is ready for another write. Writing to a remote database over a network can exacerbate the problem.

The database plugin has three options.

log or alert

This option lets you write alert or log data to the selected database. Remember, you can enable as many output plugins as you wish. To write both alert and log data to the same database, activate the database plugin twice, once with *alert* and once with *log*.

mysql or postgresql or unixodbc or mssql

This is the type of database you are going to write to. You can write to MySQL, PostgreSQL, Oracle, Microsoft's SQL Server, and UnixODBC-compliant databases.

parameter list

This list contains the *name=value* pairs required to successfully write to a database. The names and descriptions are as follows:

- *host*—This is the host where the database resides. Set to *localhost* if connecting to a database local to Snort.
- *port*—The port number to connect at the remote database host.
- *dbname*—The database username required to log in to the database.
- *password*—The password required to authenticate to the database.

- `sensor_name`—The name you want given to this sensor or Snort process in the database.
- `hex`—Include this name with no value to store binary data in hex. This is the default.
- `base64`—Include this name with no value to store binary data in base64 encoding.
- `ascii`—Include this name with no value to store binary data in human-readable ASCII encoding.
- `detail`—Can be either full or fast. With full you log all details of the packet that Snort has available. Fast logs only timestamp, signature, source and destination IPs, source and destination ports, TCP flags, and protocol.

Unified

The unified output plugin is designed specifically for speed. It is the fastest possible method of outputting Snort intrusion data. Unified writes intrusion data to its own binary format. It outputs two files: an alert file and a packet log file. The alert file contains a summary of the alert, including only the source and destination IP addresses, the protocol, the source and destination ports, and the alert message id. The log file contains the full packet information.

The purpose of the unified plugin is to allow data to be pushed out of Snort as fast as possible and to outsource plugins to a dedicated application. The application, Barnyard, reads unified output and sends data to other plugins, namely database output plugins. This gives Snort the luxury of not having to wait for a slower database to be ready to accept more input.

Unlike other plugins, the unified plugin is enabled with two different commands. The `alert_unified` command outputs alert data and the `log_unified` command outputs log data. Each produces a file that has a time signature (`monthday@hourminute-`) prepended to it. They both have one configuration option.

limit [maximum size]

The maximum size to which a file is allowed to grow. Default is 128MB.

Summary

This chapter delved deep into the inner workings of Snort. Snort acquires raw packets directly from a network interface card. The acquisition of packets is performed by the libpcap, which is external to Snort. Libpcap is portable to every popular computing platform, making Snort a truly platform-independent application.

The packet decoder is the first internal component of Snort that a sniffed packet encounters. Its purpose is to strip off the various headers. It works by decoding up the TCP/IP stack, and placing the packet in a data structure. Packets are then routed to the preprocessors.

Snort's preprocessors perform two fundamental functions. They either manipulate packets so the detection engine can properly analyze them, or they examine traffic for suspicious use that cannot be discovered by signature detection alone. Snort has a variety of preprocessors, most of which have been added to combat new methods of IDS evasion. Everything from polymorphic shellcode to fragmented packets can be detected with the aid of Snort's preprocessors. After traffic is run through the preprocessors, it is sent on to the detection engine.

The detection engine is responsible for the actual signature detection. Snort rules are loaded into the detection engine and are categorized in a tree-like data structure. This tree structure is implemented to be more efficient by minimizing the number of tests the detection engine has to perform to discover malicious activity. After malicious activity has been discovered, Snort writes intrusion data to any number of output plugins.

The output plugins are the means Snort has to get data from the detection engine to you. Snort can be configured with multiple output plugins to better facilitate intrusion data management. Output plugins can range from simple comma-delimited output to complex relational database output. An output format has been specifically designed for Snort to outsource the writing to databases, which has traditionally been a bottleneck.