

# Passive Network Traffic Analysis: Understanding a Network Through Passive Monitoring

*Kevin Timm* 2003-05-21

Network IDS devices use passive network monitoring extensively to detect possible threats. Through passive monitoring, a security admin can gain a thorough understanding of the network's topology: what services are available, what operating systems are in use, and what vulnerabilities may be exposed on the network. Much of this data can be gathered in an automated, non-intrusive manner through the use of standard tools, which will be discussed later in this article. While the concepts presented here are not difficult to understand, the reader should have at least an intermediate understanding of IP and a base-level familiarity with the operation of network sniffers.

## IP and TCP Headers

Since it is assumed the reader has an intermediate level understanding of the Internet Protocol suite (IP), we will take only a cursory look at the IP and TCP headers, highlighting and giving a brief description of the fields of interest. A very detailed reference to the specific IP protocols is available at in the [RFC Sourcebook](#). In the sample header below, we are primarily interested in the fields that are highlighted in blue and red: the blue fields require mostly manual interpretation while the red fields have tools that automate much of the analysis.

IP Header Format:

	4	8	16	32 bits
Version	IHL	Type of Service	Total Length	
Identification			Flags	Fragment Offset
TTL	Protocol		Header Checksum	
Source Address				
Destination Address				
Options + Padding				
Data				

TCP Header Format:

4		8		16		32 bits	
Source Port				Destination Port			
Sequence Number							
Acknowledgement Number							
Data Off	Res	ECN	Flags	Window			
CheckSum				Urgent Pointer			
Options and Padding							
Data							

From the above headers, a table can be created that details the fields of interest and the value of that field during analysis. Much of the data needed is only available in the TCP header options, which means that TCP SYN packets are preferred for optimum fidelity. However, Even though TCP SYN packets are preferred, UDP and ICMP can be used with reduced fidelity. The following table provides a list of the fields, their location within the header, whether automated tools exist and what value that field provides (for example, whether it helps determine topology, host type, or what the host may be doing).

Field	Location	Automated Tools	Value
TTL	IP Header	Yes	OS & Topology
Fragmentation Flags	IP Header	Yes	OS
IHL	IP Header	Yes	OS
TOS	IP Header	No	OS
IP Identification	IP Header	No	OS & Traffic
Source Port	TCP Header	No	OS & Traffic
Window	TCP Header / Options	Yes	OS
Max Segment Size	TCP Header / Options	Yes	OS
Sack OK	TCP Header / Options	Yes	OS
Nop Flag	TCP Header / Options	Yes	OS

The only tool necessary tool to perform passive analysis is a packet sniffer that has network

access or historical logs. Commonly used sniffers are [tcpdump](#), Snoop and [Ethereal](#). Ethereal has a nice GUI front end while tcpdump and Snoop are command line utilities. An explanation of the sniffers use is beyond the scope of this article; however, a nice Snoop tutorial is available at <http://www.var-log.com/papers.snoop.html>. Even though all of the analysis can be done with only the output of a sniffer, large amounts of data will be difficult to cull through without the assistance of other specialized tools. These invaluable tools will be discussed through the course of this article.

As stated previously, passive monitoring can be used to gain a general understanding of network topology and the services being offered therein. By analyzing packet traces, a host's logical location can be determined through analysis of the TTL field of the IP header. Operating systems normally use an initial TTL value of either 32, 64, 128 or 255. This value is decreased incrementally each time the packet passes through a router. Without going into unnecessary detail, most of the time a host's TTL, as seen in a trace, will be its base TTL minus the number of hops it is away from the originating network. There are some exceptions, most notably [traceroute](#); however, traceroute is easily identified. In the following trace, the example inside network is 10.10.10.X. The TTL field is highlighted in blue.

```
09:37:56.045881 10.10.10.3.9082 > 64.154.80.49.80: P [bad tcp cksum a34f!]
    1560784165:1560785134(969) ack 4104043405 win 8280 (DF)
    (ttl 127, id 39129, len 1009)
10:19:16.695881 170.208.15.77.31386 > 10.10.10.22.6588: S [tcp sum ok]
    1804289383:1804289383(0) win 16384 [tos 0x10]
    (ttl 243, id 63442, len 40)
10:19:16.695881 10.10.10.22.6588 > 170.208.15.77.31386: R [tcp sum ok]
    0:0(0) ack 1804289384 win 0 (DF)
    (ttl 64, id 53077, len 40)
11:10:20.895881 10.10.10.5.445 > 68.157.138.20.4640: R
    [tcp sum ok] 0:0(0) ack 2374038867 win 0 (DF)
    (ttl 255, id 0, len 40)
11:10:21.185881 68.157.138.20.4292 > 10.10.10.2.445: S [tcp sum ok]
    2358041082:2358041082(0) win 16384 (DF)
    (ttl 112, id 51954, len 48)
14:29:17.035881 10.10.10.11.80 > 139.223.175.6.2535: R [tcp sum ok]
    0:0(0) ack 3206335729 win 0 (DF)
    (ttl 255, id 0, len 40)
```

```
14:29:17.735881 139.223.175.6.2535 > 10.10.10.11.80: S [tcp sum ok]
      3206335728:3206335728(0) win 16384 (DF)
      (ttl 105, id 57821, len 48)
```

A cursory analysis reveals that our host 10.10.10.3 has a TTL of 127 and is one hop away from us on the network. Furthermore, internal hosts 10.10.10.22, 10.10.10.5 and 10.10.10.11 both have their TTL set to either 64 or 255, indicating that they are most likely on the broadcast domain from where the sniffing is taking place. This topology can be extended to external hosts such as 170.208.15.77 (TTL 243), which we assume is approximately twelve hops away (base TTL 255 - TTL = 12). The use of the TTL field to determine internal logical segmentation and layout would be of greater benefit on larger networks than in our example.

Delving further into sample traces it is easy to use a combination of common Unix tools such as tcpdump, sort and uniq to determine what services are being offered on the network and how often they are being accessed. This is easily accomplished by creating a filter that will look for SYN/ACK packets leaving the protected network. Similarly filters can be created that sort UDP and ICMP traffic, although traffic direction and initiation are harder to establish without manual interpretation or payload analysis.

```
tcpdump -r 0307\@09-snort.log -n src net 10.10.10 and 'tcp[13] ==18' -
tt |awk '{print $2}' |sort |uniq -c |sort -rn |more
```

```
2059 10.10.10.9.http
      206 10.10.10.1.4065
      184 10.10.10.1.h323hostcall
      79 10.10.10.1.2065
      45 10.10.10.1.6065
      13 10.10.10.1.9065
      3 10.10.10.7.http
      1 10.10.10.8.http
      1 10.10.10.5.telnet
      1 10.10.10.12.telnet
      1 10.10.10.11.telnet
```

From this output it can be determined that HTTP and Telnet are offered by several hosts on the

network. Meanwhile, host 10.10.10.1 appears to be offering services on several non-standard ports.

In a similar fashion, the same logic can be applied to determine if the network has been probed for services that were not open. For this determination, constructing a filter to look for TCP reset packets from the protected network would yield results where someone sent a request to a port that was closed but not filtered. This is an important distinction: if this data is gathered from hosts that are protected from a firewall this would then indicate services that the firewall is allowing through but are not offered. This is a simple way to verify correct implementation of policy.

```
tcpdump -r 0220\@00-snort.log -n src net 10.10.10 and 'tcp[13] == 4'
08:22:34.200240 10.10.10.20.https > 67.64.67.97.14448: R
360205510:360205510(0) win 0
14:02:45.760240 10.10.10.9.smtp > 218.144.104.100.4284: R
338876750:338876750(0) win 0
```

In the above trace 10.10.10.20 was probed for https while 10.10.10.9 was probed for SMTP. The ports were closed for both hosts.

## Passive Operating System Identification

Passive Operating System identification relies on the fact that each OS's IP implementation differs slightly, depending on RFC interpretation. This is most accurately done by analyzing TCP SYN packets, as these have the greatest number of unique identifiers. The fields used to make this determination are Initial TTL, Don't Fragment flag, Initial SYN Packet Size, IP ID, TOS Field, Source Port and TCP Options, such as Window Size, MSS Size, Nop Option, Window Scaling Option and SackOK Option.

Previously, excellent papers on this topic have been published by both [Lance Spitzner](#) and [Toby Miller](#). Both of these papers are excellent; however, they present slightly varied techniques. Currently, there exist two automated tools for doing such analysis: [Siphon](#) (which doesn't appear to have been updated in a while) and [pOf](#) (which was recently updated). Of these two tools, pOf relies mainly on a combination of IP fields and TCP SYN packet options, while Siphon uses mainly IP-level information such as TTL and Don't Fragment bit. Automating analysis is quite easy with pOf . POF even detects some not so OS-dependent nuances such as [Nmap](#)

scans, as follows:

```
/p0f -s 0307\@09-snort.log -f p0f.fp |sort |uniq -c |sort -rn >
outfile-fp
.
2059 24.248.226.68 [1 hops]: RedHat 8.0 (2.4.18.14)
 1086 10.10.10.21: UNKNOWN [16384:64:1460:1:0:1:1:64].
   717 219.60.4.30: UNKNOWN [54020:107:1460:1:0:1:1:52].
   610 61.241.160.195 [21 hops]: Windows 2000 (9)
     7 200.74.27.228 [15 hops]: NMAP scan (distance inaccurate) (1)
     6 38.144.36.16: UNKNOWN [0:243:0:0:-1:0:0:40].
     6 216.232.104.234 [17 hops]: Windows 2000 (9)
     5 10.10.10.16: UNKNOWN [3072:64:1460:0:0:0:1:60].
     4 24.249.226.68 [2 hops]: RedHat 8.0 (2.4.18.14)
     3 67.35.14.5: UNKNOWN [65535:109:1414:1:0:1:1:64].
     3 65.30.164.157 [13 hops]: Windows 2000 (9)
     3 63.178.85.140 [18 hops]: Windows NT 5.0 (2)
     3 62.123.112.135 [21 hops]: Windows 2000 (8)
     3 61.145.232.215: UNKNOWN [58944:37:1452:1:2:1:1:52]
```

It is relatively simple to automate this analysis on raw logs; however, readers should be aware that it is possible to change the default OS behavior on several operating systems. With Linux you can change IP behavior by using `sysctl` or by changing values in the `/proc/sys/net/ipv4` directory. Various options can be set this way, including default ttl, window scaling, as well as rate limitation, and other OS-specific operational information.

There is another specific option that is controlled by `sysctl` (`ip_local_port_range`) that helps to identify various Linux hosts; this option requires manual interpretation. For example, on a 2.4.18-14 Kernel, it is set between 32,768 and 61,000, whereas on a 2.4.7-10 Kernel, it is set to a value between 1024 and 4999. An excellent resource for understanding what values can be tuned using `sysctl` is available at <http://www.linuxdocs.org/HOWTOs/Adv-Routing-HOWTO-12.html>.

## Analyzing Packet Payloads

To understand more than just what applications are offering services or what operating systems are present on the network, packet payloads need to be analyzed. This can be a tedious task to automate; fortunately `ngrep` makes this chore much easier. `ngrep` can do regular expression-based analysis of network traffic. While this functionally may not seem to differ much from the traditional IDS, `ngrep` offers a quick, powerful interface to network level data with added flexibility. `ngrep` can be used for further examination of network traffic by helping to identify the types of services available on a network and if they are potentially vulnerable. `ngrep` is syntactically very similar to `tcpdump`. A simple example of what can be done would be to use `ngrep` to parse through previous acquired logs for specific information such as identifying which types of Web servers are available on the network.

```
ngrep -qt -I 0220\@00-snort.log -s 200 -i "200 OK" src port 80 |more
input: 0220@00-snort.log
```

```
T 2003/02/20 01:45:15.150240 10.10.10.17:80 -> 211.137.71.1:3774 [A]
  HTTP/1.1 200 OK..Server: Apache/1.3.12 (Unix) (Red Hat/Linux)
mod_ssl/2.6.6 OpenSSL/0.9.5a PHP/4.0.4pl1 mod_perl/1.24' ..Date:
```

```
T 2003/02/20 01:45:19.320240 10.10.10.17:80 -> 211.137.71.1:3777 [A]
  HTTP/1.1 200 OK..Server: Apache/1.3.12 (Unix) (Red Hat/Linux)
mod_ssl/2.6.6 OpenSSL/0.9.5a PHP/4.0.4pl1 mod_perl/1.24' ..Date:
```

```
T 2003/02/20 01:45:20.360240 10.10.10.17:80 -> 211.137.71.1:3784 [A]
  HTTP/1.1 200 OK..Server: Apache/1.3.12 (Unix) (Red Hat/Linux)
mod_ssl/2.6.6 OpenSSL/0.9.5a PHP/4.0.4pl1 mod_perl/1.24' ..Date:
```

Once again, this could be sorted through the use of Unix commands such as `awk`, `sort` and `uniq` to provide an output as such

```
root@snort Feb_20]# ngrep -qt -I 0220\@00-snort.log -s 200 -i "200
OK" src port 80 |grep HTTP |awk '{print $4}' |sort | uniq -c |sort -
rn
  422 Apache/1.3.27
  397 Netscape-Enterprise/3.6
  370 Apache/1.3.12
```

301 Microsoft-IIS/5.0

ngrep also allows a certain amount of packets to be displayed after each match. This is useful for determining if a probed host may have been vulnerable to an attack. To illustrate this, we will use a Nimda trace, since they are so prevalent and well understood. We must first establish that the network has been probed:

```
ngrep -qt -I 0313\@23-snort.log "cmd.exe" port 80 |wc -l
2356
```

From there, the use of the "-A 2" flag will catch the following two packets (Ack and the r response) and then a simple `grep` looking for a "200 OK" response code should determine if in fact a vulnerable host existed.

```
ngrep -qt -I 0313\@23-snort.log "cmd.exe" port 80 -A 2 |grep "200
OK"
[root@snort Mar_13]#
```

No vulnerable hosts. Great!!!

## Raw Sockets

When examining traffic in this manner it is essential to have a basic understanding of how most IP stacks work and what is considered normal traffic and what is not. Understanding the ability of raw sockets and how the use of these could impact traffic is key. The normal socket API handles most of the details about interface selection, session establishment, port selection, Sequence & Ack numbers, flags set, TTL values, options, and more. The only way to easily manipulate these is through the use of `sysctl` or `sysctl`-like operating system functions, which can control a few of these options, or through the use of a tool that can craft packets using the raw socket API.

The raw sockets API allows for manipulation of essentially any header field; however, to access these requires some network programming. Libraries such as [Libnet](#) have made this easier and other tools such as [hping](#) require no programming. With raw socket packet creation, all header fields will need to be filled in by the program; this will most likely create inconsistencies compared to packets generated using the standard sockets API. There are circumstances,

though, where only raw sockets can be used to intentionally create certain types of packets. Using Raw sockets is powerful; however, it is much more difficult to maintain a session than with traditional sockets API because the normal operating system responses must often be suppressed (since the packets will seem unsolicited to the Operating System kernel) and the programmer must have some way to use the response from the one packet to create further session related packets. This somewhat limits the use of raw sockets.

That said, understanding the abilities of raw sockets and what can easily be accomplished with them (or not) is essential to understanding the legitimacy of certain traffic. Armed with this knowledge, this final trace is a good example of a crafted packet becoming a part of the stream. The final "reset" packet is most likely crafted.

```
10:01:02.323906 10.10.10.123.39442 > 172.16.61.162.ftp: S [tcp sum
ok] 2147534086:2147534086(0) win 8192 <mss 524,nop,wscale
0,nop,nop,timestamp 2774130191 0> (ttl 58, id 13436, len 60)
10:01:02.324153 172.16.61.162.ftp > 10.10.10.123.39442: S [tcp sum
ok] 1226704774:1226704774(0) ack 2147534087 win 4096 <mss 1024>
(ttl 255, id 59526, len 44)
10:01:02.360296 10.10.10.123.39442 > 172.16.61.162.ftp: . [tcp sum
ok] 1:1(0) ack 1 win 8192 (ttl 58, id 13437, len 40)
10:01:02.360501 172.16.61.162.ftp > 10.10.10.123.39442: P [tcp sum
ok] 1:4(3) ack 1 win 4096 (ttl 255, id 59555, len 43)
10:01:02.404393 10.10.10.123.39442 > 172.16.61.162.ftp: R [tcp sum
ok] 2147534087:2147534087(0) win 0 (ttl 56, id 61581, len 40)
```

Looking only at the TTL and IP id section of the packets from the host that initiated the session, this doesn't appear to be the same host. Most likely, the final packet was solicited from a network security device somewhere along the line and that device spoofed the IP intentionally to tear down the session with a "reset".

```
(ttl 58, id 13436, len 60)
(ttl 58, id 13437, len 40)
(ttl 56, id 61581, len 40)
```

## Conclusion

There is a wealth of data available from network traffic that can aid in understanding the network, its devices, services and vulnerabilities. Some analysis of this data can be automated through simple scripting. This data can be used for determining how well an access policy is implemented and if vulnerabilities exist on the network. Becoming aware of these methods will assist in accurate analysis of network traces as well as understanding the network itself.

[Privacy Statement](#)

Copyright 2006, SecurityFocus