

Studying Normal Network Traffic, Part One

Karen Kent Frederick 2001-01-29

Studying Normal Network Traffic, Part One

by *Karen Kent Frederick*

last updated Jan. 29, 2001

Many intrusion detection analysts concentrate on identifying the characteristics of suspicious packets - illegal TCP flag combinations or reserved IP addresses, for example. However, it is also important to be familiar with what normal traffic looks like. A great way to learn what traffic should look like is to generate some normal traffic, capture the packets and examine them. In this article, I will discuss a tool for logging packets, and I will review some packet captures in depth. In future articles in this series, I will be examining normal traffic in greater depth, as well as reviewing some examples of abnormal traffic. Note that in order to understand this material, you should already know the fundamentals of TCP/IP.

There are many utilities that can be used to perform packet captures. Two of the most popular are [tcpdump](#), for Unix systems, and its Windows version, [WinDump](#). I ran the beta version of WinDump 2.1 on one of my home computers, a Windows 98 machine connected to a cable modem, to capture this traffic. Please note: if you are going to capture traffic, make sure that you have the proper authority to do so. For example, using a packet sniffer on your machine at work without authorization is a really bad idea!

WinDump Basics

WinDump is very simple to use, and the documentation at its web site should give you all the information you need to install and use it. The syntax that I used was `Windump -n -S`, `Windump -n -S -v` and `Windump -n -S -vv`. The `-n` option tells Windump to display IP addresses instead of the computers' names. The `-S` option indicates that the actual TCP/IP sequence numbers should be shown. If this option is omitted, relative numbers will be shown; for example, if the initial sequence number is 87334271 and the next number is one higher, a 1 will be displayed instead of 87334272. The `-v` and `-vv` options make the output more verbose and much more verbose respectively, adding fields such as time to live and IP ID number to the logs.

Before reviewing our example capture, let's look at how WinDump records various types of packets. Here's a TCP example, which shows a data packet with the PUSH and ACK flags set. First, we have the WinDump log entry for the packet. Immediately after it is the same entry, but with an explanation added for each field:

```
13:45:19.184932 sshserver.xx.yy.zz.22 > mypc.xx.yy.zz.3164: P 4138420250:4138420282
(32) ack
87334272 win 32120 (DF)
```

```
13:45:19.184932 [timestamp]      sshserver.xx.yy.zz.22 [source address and port]      >
mypc.xx.yy.zz.3164: [destination address and port]      P [TCP flags]
4138420250:4138420282
[sequence numbers]      (32) [bytes of data]      ack 87334272 [acknowledgment flag and
number]      win
32120 [window size]      (DF) [don't fragment flag is set]
```

The next example is UDP. It contains the same initial fields as the TCP example: timestamp, source address and port, destination address and port. After that, it lists the protocol as UDP and then gives the number of data bytes in the packet:

```
15:19:14.490029 208.148.96.68.23079 > mypc.xx.yy.zz.6976:  udp 401
```

ICMP packets are logged in a similar format. Note that the fields at the end, time to live and the IP ID number, were included because I was using windump's -v option:

```
18:33:45.649204 mypc.xx.yy.zz > 64.208.34.100: icmp: echo request (ttl 4, id 56693)
```

Finally, WinDump will also capture ARP requests and replies. Although this format is fairly self-explanatory, let's take a quick look at it. The first line is an ARP request. In this case, mypc has requested that the MAC address of 24.167.235.1 be sent to mypc.xx.yy.zz (mypc's IP address). The second line shows the ARP reply, which contains the MAC address of 24.167.235.1.

```
13:45:13.836036 arp who-has 24.167.235.1 tell mypc.xx.yy.zz
13:45:13.841823 arp reply 24.167.235.1 is-at 0:xx:xx:xx:xx:xx
```

Basic UDP and ICMP Examples

Now that we've reviewed the WinDump log format, we're ready to look at some packets. Let's start with an easy example. mypc uses DHCP in order to get its IP address. The DHCP lease needs to be renewed periodically. These two entries show the DHCP renewal request from mypc's port 68 to the DHCP server's port 67, and then the corresponding reply from the DHCP server to mypc.

```
18:47:02.667860 mypc.xx.yy.zz.68 > dnsserver.xx.yy.zz.67:  xid:0x8d716e0f C:mypc.xx.
yy.zz [|bootp]
18:47:03.509471 dnsserver.xx.yy.zz.67 > mypc.xx.yy.zz.68:  xid:0x8d716e0f C:mypc.xx.
yy.zz
Y:mypc.xx.yy.zz [|bootp]
```

One of the best features of WinDump is that it can automatically recognize certain protocols and log additional information about them. In this case, it can tell that this traffic is bootp, so it not only records the standard UDP entries, but it also records bootp-specific information: the values listed as xid, C and Y.

Now let's look at some ICMP traffic. This example is part of a traceroute that I performed from my Windows 98 machine using the tracert command. Windows traceroutes use ICMP echo requests to identify all the network hops between two systems. (Note that Unix-style traceroutes typically use UDP packets instead of ICMP.)

Windows performs a traceroute by sending 3 ICMP packets to the target system; however, it sets their time to live value to 1. This means that when the packets reach the first hop, the time to live decrements to 0. At that point, an ICMP time exceeded error message is returned by the system at the first hop to the sender. Windows then sends 3 more ICMP packets to the target system, setting their time to live value to 2. This time, the packets get to the second hop before the TTL expires; the system at the second hop then sends ICMP time exceeded messages to the sender. This continues until the packets reach the target system, at which point the traceroute is complete, or until

an intermediate system drops or rejects this traffic.

Here's part of the capture of a Windows-style traceroute. At this point, ICMP packets with their time-to-live values set to 1, 2 and 3 have already been sent and expired. Since the final destination has not yet been reached, Windows will now send out 3 ICMP packets with the time to live set to 4. Here's the first packet and the response. Note that although the destination of the first packet is 64.208.34.100, the reply comes back from 24.95.80.133, which is one of the intermediate network devices between mypc and 64.208.34.100.

```
18:33:45.649204 mypc.xx.yy.zz > 64.208.34.100: icmp: echo request (ttl 4, id 56693)
18:33:45.668638 24.95.80.133 > mypc.xx.yy.zz: icmp: time exceeded in-transit (ttl
252, id 0)
```

About a thousandth of a second after the ICMP time exceeded error message is received, mypc sends the second ICMP packet. As soon as the error message for that packet is received, the third ICMP packet is then sent out:

```
18:33:45.669968 mypc.xx.yy.zz > 64.208.34.100: icmp: echo request (ttl 4, id 56949)
18:33:45.690719 24.95.80.133 > mypc.xx.yy.zz: icmp: time exceeded in-transit (ttl
252, id 0)
18:33:45.691863 mypc.xx.yy.zz > 64.208.34.100: icmp: echo request (ttl 4, id 57205)
18:33:45.710787 24.95.80.133 > mypc.xx.yy.zz: icmp: time exceeded in-transit (ttl
252, id 0)
```

Note that these entries are nearly identical to the first set, except that the IP ID number is different for each ICMP echo request. This is expected behavior; we should be suspicious of traffic with all the IP ID numbers set to the same value.

Examining an SSH Session

Here's a more complicated example of typical traffic: an SSH session. I have an SSH client installed on my workstation, and I initiated a connection to an external server where I have a user account. Just making that connection takes several steps, which occur in less than a fourth of a second. We'll go through these steps one at a time.

I have told my SSH client software that I want to contact sshserver. My computer does not know what the IP address for sshserver is, so it needs to issue a DNS query to resolve the address. Unfortunately, my machine can't do a DNS query at this time. It has been idle for a while, so the cached ARP entry for its default gateway has expired. In order to make the DNS query, it must first use ARP to ask for the MAC address of the default gateway:

```
13:45:13.836036 arp who-has gateway.xx.yy.zz tell mypc.xx.yy.zz
13:45:13.841823 arp reply gateway.xx.yy.zz is-at 0:xx:xx:xx:xx:xx
```

Now that it knows how to reach its default gateway, mypc can make a DNS query, which is shown below. Notice that mypc uses a port number higher than 1023, as we would expect it to. It directs its request to port 53 on the DNS server. This request is made using UDP, as we would expect with a DNS query:

```
13:45:13.841920 mypc.xx.yy.zz.3163 > dnsserver.xx.yy.zz.53: 1+ A? sshserver. (32)
```

The DNS query is represented by "1+ A? sshserver." We can tell that this is a query for an IP address because of the A?, and the + indicates that we have asked for the query to be recursive. The 1 is the DNS query number, which is used to match a DNS query with its reply. Finally, 'sshserver' is the name that we want to resolve.

Next, the DNS server returns a reply to our query:

```
13:45:13.947208 dnsserver.xx.yy.zz.53 > mypc.xx.yy.zz.3163: 1 q: sshserver. 3/4/6
sshserver. CNAME
ssh2server., ssh2server. CNAME ssh3server., ssh3server. A sshserver.xx.yy.zz (283)
```

The properties of the reply are designated by "1 q: sshserver. 3/4/6". The 1 is the DNS query number, which matches the value used in the request. The "q: sshserver." portion tells us what name we queried for. The 3/4/6 indicates that 3 answers were returned and that there were 4 authority records and 6 additional records. The actual IP address associated with sshserver is listed after A.

Now that we know the IP address of the SSH server, we can attempt to connect to it. mypc initiates the TCP three-way handshake with the server:

```
13:45:13.956853 mypc.xx.yy.zz.3164 > sshserver.xx.yy.zz.22: S 87334271:87334271(0)
win 65535 (DF)
13:45:14.059243 sshserver.xx.yy.zz.22 > mypc.xx.yy.zz.3164: S 4138420249:4138420249
(0) ack 87334272
win 32120 (DF)
13:45:14.059475 mypc.xx.yy.zz.3164 > sshserver.xx.yy.zz.22: . 87334272:87334272(0)
ack 4138420250
win 65535 (DF)
```

The three-way handshake has been successfully completed. Remember that at this point, although the two computers have established a connection on the SSH port, I am not logged in to the SSH server. No data is passed between the two computers until after the three-way handshake has been completed. Then the SSH client and server software communicate with each other to establish the actual SSH session, using packets like these:

```
13:45:19.184932 sshserver.xx.yy.zz.22 > mypc.xx.yy.zz.3164: P 4138420250:4138420282
(32) ack
87334272 win 32120 (DF)
13:45:19.201814 mypc.xx.yy.zz.3164 > sshserver.xx.yy.zz.22: P 87334272:87334314(42)
ack 4138420282
win 65503 (DF)
13:45:19.300401 sshserver.xx.yy.zz.22 > mypc.xx.yy.zz.3164: . 4138420282:4138420282
(0) ack 87334314
win 32120 (DF)
13:45:19.300616 mypc.xx.yy.zz.3164 > sshserver.xx.yy.zz.22: P 87334314:87334690(376)
ack 4138420282
win 65503 (DF)
13:45:19.303977 sshserver.xx.yy.zz.22 > mypc.xx.yy.zz.3164: P 4138420282:4138421210
(928) ack
87334314 win 32120 (DF)
13:45:19.422141 sshserver.xx.yy.zz.22 > mypc.xx.yy.zz.3164: . 4138421210:4138421210
(0) ack 87334690
```

```

win 32120 (DF)
13:45:19.488282 mypc.xx.yy.zz.3164 > sshserver.xx.yy.zz.22: . 87334690:87334690(0)
ack 4138421210
win 64575 (DF)

```

Eventually, I was prompted to enter my password, and I was authenticated as a user of the server. All of that traffic, and the traffic generated by my use of the SSH server, looked very similar to this, with PSH/ACK and ACK packets sent between mypc's port 3164 and sshserver's port 22.

Once I had completed my work, I logged off sshserver. More packets like these were passed between the client and the server; then my client program received an acknowledgment of its final data packet being received by the SSH server. Here is the last packet with data and its acknowledgement:

```

13:45:33.791528 mypc.xx.yy.zz.3164 > sshserver.xx.yy.zz.22: P 87335442:87335474(32)
ack 4138426586
win 64359 (DF)
13:45:33.902690 sshserver.xx.yy.zz.22 > mypc.xx.yy.zz.3164: . 4138426586:4138426586
(0) ack 87335474
win 32120 (DF)

```

Immediately after this was received, the SSH client software initiated a graceful termination of its half of the connection. The SSH server responded appropriately, acknowledging that termination and initiating the teardown of its half of the connection:

```

13:45:33.902909 mypc.xx.yy.zz.3164 > sshserver.xx.yy.zz.22: F 87335474:87335474(0)
ack 4138426586
win 64359 (DF)
13:45:34.002179 sshserver.xx.yy.zz.22 > mypc.xx.yy.zz.3164: . 4138426586:4138426586
(0) ack 87335475
win 32120 (DF)
13:45:34.003336 sshserver.xx.yy.zz.22 > mypc.xx.yy.zz.3164: F 4138426586:4138426586
(0) ack 87335475
win 32120 (DF)
13:45:34.003492 mypc.xx.yy.zz.3164 > sshserver.xx.yy.zz.22: . 87335475:87335475(0)
ack 4138426587
win 64359 (DF)

```

So we can think of an SSH connection as having five steps:

1. ARP request to determine the MAC address of mypc's default gateway
2. DNS request to find the IP address of the SSH server
3. TCP three-way handshake between a high port on mypc and the SSH port on sshserver
4. Communications between mypc and sshserver, which included the establishment of an SSH connection, user authentication, and data transfer
5. TCP connection teardown between mypc and myserver

In this example, the SSH client used a high port number (over 1023). In most client-server connections, the client uses a high port number and the server uses a low port number. It is important to note that some SSH clients actually use a low port number, so don't let that confuse you if your traces look a little different from this one.

There are many other protocols, such as telnet, that follow the same pattern as SSH connections.

Conclusion

Studying routine network traffic is a great way to learn more about TCP/IP and your own environment. Every time I do a capture and study the packets, I learn something new. If you're interested in this topic, I strongly encourage you to generate and capture your own traffic (with the proper authority, of course!) and to learn as much as you can from it. In this article, I have barely scratched the surface of what you can learn from doing this. The next article in this series will contain more examples of traffic, particularly FTP and HTTP, and we will dig deeper into the packets to better understand what they can tell us.

To read **Studying Normal Traffic, Part 2: Studying FTP Traffic**, click [here](#).

Karen Kent Frederick is a senior security engineer for NFR Security. Karen has a B.S. in Computer Science and is completing her Master's thesis in intrusion detection through the University of Idaho's Engineering Outreach program. She holds several certifications, including Microsoft Certified Systems Engineer + Internet, Check Point Certified Security Administrator, and SANS GIAC Certified Intrusion Analyst. Karen is one of the authors and editors of "Intrusion Signatures and Analysis", a new book on intrusion detection that was published in January 2001.

Relevant Links

[tcpdump](#)

Tcpdump Group

[WinDump](#)

NetGroup

[Privacy Statement](#)

Copyright 2006, SecurityFocus