10

Real-World Analysis

No doubt you've had your fill of healthy, low-fat theory on packet dissection and header fields. How about bringing on some of the more interesting, tasty, real-world traffic? That is what we are about to embark on in this chapter. For you to understand the analysis that will be shown here, it was necessary to lay the groundwork in previous chapters first.

To refresh your memory of the intent of this section, we want to analyze traffic from many different viewpoints. We've evolved from bits and fields in previous chapters to inspecting one or more packets for their intent and explaining some actual events of interest that were captured by Shadow from sites.

The transition from understanding theory to actually explaining some traffic that you see is not necessarily an easy or intuitive one. It takes time and exposure to some interesting traffic before you gain the confidence and experience to make this transition. The examples shown in this chapter should help you get started.

You've Been Hacked!

The simplicity of this first real-world event belies its poignancy. In a former lifetime, I worked for a local military Computer Emergency Response Team (CERT). I worked an early shift beginning about 5:30 A.M. to avoid the brunt of the rush hour traffic from the suburbs of one of the nation's most awful commuting cities, Washington, DC. I walked into the office one morning, and the phone was already ringing—not a good sign unless it is Ed McMahon calling to tell me I'd won the Publisher's Clearinghouse Sweepstakes. Instead, the call was from one of our parent military CERTs informing us that we'd had a break-in over night.

As a bit of background, the parent CERT used a different set of tools to monitor our site than we did, and would sometimes call when it had an inquiry about traffic or to report something noteworthy, as in this case. The CERT supplied the date, approximate time, and source and destination IPs associated with the break-in, but could supply no more information than this when queried.

The destination IP of the alleged victim host was a DNS server at the site. This was probably one of the best maintained hosts on the site; it had the most recent patches of BIND, it had all ports closed except for secure shell (SSH) from specific source addresses and DNS queries, and it had been stripped of all unnecessary user accounts. It was not as if this was some legacy system sitting openly on a DMZ with no recent attention, superfluous ports open, and unrestricted access. Still, although my first reaction was skepticism, I wasn't naive enough to think that any host connected to the Internet was impervious to attack. After all, this was a DNS server, and the venerable BIND software has been plagued with a history of problems, including buffer overflow attacks that allowed remote root access.

A rational way to approach this early morning report was to use TCPdump records from Shadow to examine all traffic to and from our DNS server from the alleged attacker's IP address. Before showing you an excerpt of the results of that, let's just re-examine what an established TCP session looks like in terms of TCPdump.

Three-Way Handshake:

```
boulder.myplace.com.38060 > aspen.myplace.com.telnet: S 3774957990:
3774957990(0) win 8760 <mss 1460> (DF)
aspen.myplace.com.telnet > boulder.myplace.com.38060: S 2009600000:
2009600000(0) ack 3774957991 win 1024 <mss 1460>
boulder.myplace.com.38060 > aspen.myplace.com.telnet:. ack 1 win 8760 (DF)
```

Data Exchange:

```
boulder.myplace.com.38060 > aspen.myplace.com.telnet: P 1:28(27) ack 1 win
8760 (DF)
aspen.myplace.com.telnet > boulder.myplace.com.38060: P 1:14(13) ack 1 win
1024
aspen.myplace.com.telnet > boulder.myplace.com.38060: P 14:23(9) ack 28 win
1024
```

Session Termination:

```
aspen.myplace.com.telnet > boulder.myplace.com.38060: F 4289:4289(0) ack 92
win 1024
boulder.myplace.com.38060 > aspen.myplace.com.telnet: .ack 4290 win 8760 (DF)
boulder.myplace.com.38060 > aspen.myplace.com.telnet: F 92:92(0) ack 4290
win 8760(DF)
aspen.myplace.com.telnet > boulder.myplace.com.38060: .ack 93 win 1024
```

First, for two hosts to exchange some kind of data, they have to complete the three-way handshake. In this case, we have host `boulder.myplace.com` requesting to connect to host `aspen.myplace.com` on port telnet. Host `aspen.myplace.com` offers telnet service; and the two hosts synchronize sequence numbers and the connection is established.

Next, typically a client connects to a host for the purpose of exchanging some data. And in this case, we witness the exchange between both hosts as we see 27, 13, and 9 bytes of data sent respectively in the three PUSH packets displayed. More data was exchanged before the session was terminated, but that is not shown because it really adds no new insight into this discussion.

Finally, the two hosts gracefully sever the connection by exchanging and acknowledging FIN packets. That is what normal TCP sessions look like.

Now, examine some of the traffic from the alleged break-in:

```
whatsup.net.24997 > dns.myplace.com.sunrpc: S 2368718861:2368718861(0) win
512 <mss 1460>
whatsup.net.25002 > dns.myplace.com.139: S 4067302570:4067302570(0) win 512
<mss 1460>
whatsup.net.25075 > dns.myplace.com.ftp: S 1368714289:1368714289(0) win 512
<mss 1460>
dns.myplace.com.ftp > whatsup.net.25075: R 0:0(0) ack 1368714290 win 0 (DF)
whatsup.net.25177 > dns.myplace.com.1114: S 3231175487:3231175487(0) win 512
<mss 1460>
whatsup.net.25189 > dns.myplace.com.tcpmux: S 368146356:368146356(0) win 512
<mss 1460>
whatsup.net.25118 > dns.myplace.com.22: S 2035824356:2035824356(0)        win
512 <mss 1460>
```

The malicious host is `whatsup.net` and our DNS server is `dns.myplace.com`. We see a bunch of attempted SYN connections to various different ports staring with port 111, also known as sunrpc or portmapper, port 139, NetBIOS session manager, ftp, and so on. We see no response from the DNS server

except to return a RESET on the ftp query. We can surmise that the packet-filtering device blocked the other ports we see, yet not ftp. When the DNS server received the ftp SYN attempt, it responded with a RESET because it didn't listen at that port.

This is just an excerpt of the traffic seen, yet it all was similar except for the different destination ports attempted. The point is that there were no successful three-way handshakes, data exchange, or session terminations witnessed. Unless there was some kind of unknown backdoor into our network that was not monitored, it appears that this was a simple scan of the DNS server and not a break-in.

After analyzing this traffic, I called the person who had reported the break-in. I shared my results and asked what kind of evidence they had that there was a break-in. The person replied that one of their parent CERT organizations had reported this and was just passing the information on to our site. I got the contact information for the original person who reported the incident and called to inquire why he believed we had suffered an intrusion. The response was that he had reported it as a scan, and it had been mistakenly communicated to me as a break-in.

My mission had not been to determine culpability; it was to determine what kind of solid evidence anyone had to refute my belief that we had only had a scan. But, as it turned out, there really was no break-in after all. This incident brought home the necessity for having an audit trail of activity into and out of the network. Had we not had the TCPdump records of the scan, we would have had no evidence to refute the intrusion claim. We would have had to trust the caller and believe that we had an intrusion that none of our NIDS had detected.

We could have logged on to the DNS server. Yet, there would be an absence of any evidence, if we were lucky. There would be no changes in any of the Tripwire logs that maintained integrity audits of important files, there would be no rootkits, and there would be no changes to password files or inetd startup files. It would be impossible to know for certain that there had been no intrusion; there would be lingering doubt that we just were not seeing the manifestations of the break-in, perhaps because of installed rootkits and Trojaned software. In such a case where you are still uncertain about the health of the host, there are not a lot of options. You have to rebuild the system from the ground up—not a desirable task.

Prior to this event, I had been a proponent of Shadow and had been collecting TCPdump activity at monitored sites. This converted me to a die-hard Shadow user, and I now use Shadow for all sites that I monitor. Truthfully, it

doesn't matter if you use TCPdump or any other collection mechanism. What matters is that you have this historical capture of the traffic entering and leaving your network. And, you don't need to capture payload, just the header portions of the records, to understand the nature of the activity as was demonstrated in this incident. Indeed, it also can be helpful to capture payload if you have enough space, even if only to keep it a couple of days before archiving it.

Netbus Scan

In the next incident, we examine a scan to destination port 12345, which is typically associated with the netbus Trojan that affects Windows hosts. This particular scan was launched against a Class B subnet so that it set off all kinds of alarms. The network that was scanned had some high-numbered port access open through the packet-filtering devices.

The following records provide a very brief excerpt of the detected traffic. This scan attempted connections to more than 65,000 IPs in the target network. It is important to note that this traffic was collected on a sensor located behind (inside) the packet-filtering device. This is the traffic that actually got inside the network. Scans happen! In fact, they happen all the time on this particular network. It's not that this network is any more inviting than others; it is just a fact of life that scanning is inevitable and frequent. Knowing this, you cannot get too worked up when you see scans. However, this is inside the packet-filtering device making it more than a curiosity, as we will later see. Here are the records:

```
bigscan.net.1737 > 192.168.7.0.12345: S 2299794832:2299794832(0) win 32120
<mss 1380,sackOK,timestamp 120377100[|tcp]> (DF)

bigscan.net.1739 > 192.168.7.2.12345: S 2299202490:2299202490(0) win 32120
<mss 1380,sackOK,timestamp 120377100[|tcp]> (DF)

bigscan.net.1741 > 192.168.7.4.12345: S 2293163750:2293163750(0) win 32120
<mss 1380,sackOK,timestamp 120377100[|tcp]> (DF)

bigscan.net.1743 > 192.168.7.6.12345: S 2298524651:2298524651(0) win 32120
<mss 1380,sackOK,timestamp 120377100[|tcp]> (DF)

bigscan.net.1745 > 192.168.7.8.12345: S 2297131917:2297131917(0) win 32120
<mss 1380,sackOK,timestamp 120377100[|tcp]> (DF)

bigscan.net.1747 > 192.168.7.10.12345: S 2291750743:2291750743(0) win 32120
<mss 1380,sackOK,timestamp 120377100[|tcp]> (DF)

bigscan.net.1749 > 192.168.7.12.12345: S 2287868521:2287868521(0) win 32120
<mss 1380,sackOK,timestamp 120377100[|tcp]> (DF
```

We see the scanning host bigscan.net methodically moving through the 192.168.7 subnet with a unique scan search pattern of looking at the .0 address and even final octets thereafter.

Netbus Hijinks

Netbus is a tool that allows remote access and control of a Windows host. After a host is compromised, it behooves the attacker to have a means of future access to the host. Netbus is one of many backdoor Trojans that can be run to provide stealthy access. It predates another, more familiar backdoor Trojan, Back Orifice. Both Netbus and Back Orifice have user-friendly GUI interfaces to easily control the remote compromised host.

Not All That Runs on Port 12345 Is Malicious

The OfficeScan virus eradication package for the corporate enterprise listens on TCP port 12345 on the desktop host. The enterprise software accommodates central virus reporting, automatic update (apparently via port 12345 on the updated host), and remote management for ease of use to assist in monitoring and configuration.

If you ever see a host that listens on TCP port 12345, it is possible that it might be a helpful rather than harmful process. Given the range of possible listening ports 1 through 65535, I'd much prefer to see the white hats (good guys) select listening ports that don't share commonly used hacker ports.

Let's go for the jugular and see if there is any need to further investigate this scan. We want to examine the hosts in the internal network and see if they responded to the scan. The TCPdump filter to examine this would look for traffic from the internal network of 192.168 with a source port of 12345 and a TCP flag pair of SYN and ACK. This means that we have a listening host, which can be potentially very dangerous. Our filter could have used the IP number of the scanning host instead of or in conjunction with the internal subnet address.

The TCPdump command used to extract response records associated with the scan reads from the binary file of collected records for the site, and identifies this scan as one that involved the internal 192.168 subnet and port 12345. The TCPdump command is further refined by using a filter that looks at the 13th byte offset of the TCP header, where the TCP flag byte is located, and looks for the ACK flag and the SYN flag set simultaneously. Here is the TCPdump command and the output generated from it:

```
tcpdump -r tcpdumpfile 'net 192.168 and port 12345 and tcp[13] = 0x12'

mynet.edu.12345 > bigscan.net.3698: S 2633608519:2633608519(0) ack 2346088305
win 49152 <mss 1380,nop,nop,timestamp 2662730[|tcp]> (DF)
```

The good news is that only one host responded. The bad news is that one host responded! When it was discovered that there was a responding host, this incident was escalated to the highest priority because we believed we had a host offering the netbus Trojan, a natural conclusion. The scan and subsequent discovery that there was a responding host occurred by 7:00 A.M., meaning that most of the staff had not yet arrived at work. In the interim, the network group was contacted and told to disallow any inbound or outbound traffic to or from the responding host by blocking it at the packet-filtering device. Also, the local computer incident response team was mobilized to scan the host for vulnerabilities and track down the owner.

After some superficial probing, the incident response team discovered that the host was a Silicon Graphics, Inc. (SGI) running an older version of Irix (SGI's version of UNIX). As a veteran of incident response teams, I remembered that older versions of Irix used to come configured with an account of lp (line printer) with no password. Tragically, a telnet connection to the host allowed me access to the host, using the lp account and no password. This discovery pretty much ruled out that this was a netbus problem because the responding host ran a version of UNIX, but we did have a rogue port answering and a host that had little, if any, security.

Concurrently, the search for the host's system administrators continued. Ownership records were dated and the host had been tossed from administrator to administrator as people moved in and out of the organization and assignments changed. This particular host had a rich history of neglect because the user-administrators were scientists or engineers who were never really trained in administration, let alone security. This is a common paradigm of neglect because many research departments do not have the budget to hire trained administrators. The users are usually overburdened workers who just need to keep the host running.

The system administrators of the SGI hosts finally arrived at work. As suspected, they had no idea what was listening on port 12345. It was also quite apparent that they and their users had little concern or appreciation for security. We told them it was necessary to disconnect the host from the network and begin backups for forensic purposes. An argument ensued when one of the users became indignant about needing to have the host up and accessible on the network. The leader of the incident response team politely told him that he had two options: first, to cooperate and willingly cede control, or second, to have the network connection unceremoniously severed by wire cutters. It seems the light bulb went on at that point, and they agreed to cooperate.

When we finally got access to the system, we wanted to make sure that the host was listening on port 12345. The process of making backups on this host was long and cumbersome, so we didn't want to make them do anything unnecessary. At the same time, we didn't want to ruin any forensic evidence by poking around too much. Only one command was attempted—the **netstat –a** command to make sure that port 12345 was running.

Can you see the flaw of executing the **netstat** command? In hindsight, it seems this was really not such a wise move. Had the **netstat** command reported that port 12345 was not listening, this would have been extremely suspicious and more indicative of a Trojaned or rootkit netstat program running on the host that was altered to not report that port 12345 was listening. But, this was not the case; port 12345 was listening.

System backups were started to preserve any forensic evidence in case some kind of prosecution ever had to be done. Finally, when the backups were completed, we had an opportunity to examine the system. We didn't want to disturb it in any way prior to the backups.

A very handy command in this situation is the **fuser** command. This is not available for all UNIX operating systems, but it is available on Irix and Linux:

```
[root@irix]#   fuser 12345/tcp
12345/tcp:              490
```

The command was issued to find the process number associated with port 12345 on TCP. By looking at the netstat output, you don't know the process that is running the service on port 12345. The **fuser** command returns the process number of the software running on that port.

Next, you have to find what that particular process number is running. That can be done using the **ps** command and then examining the output for the process number, in this case 490:

```
[root@irix]#   ps -ef | grep 490
root       490   483  0 Sep19 ? 00:02:17 /usr/local/bin/license_manager
```

You see that there is a license manager running. When this appeared on the console with the system administrator watching, he remarked that he had recently installed a license manager. He had no idea what port it listened on. The mystery was solved! This was the best possible resolution considering the alternatives. But, give me a break—what reputable license manager software maker would use a default listening port of 12345?

Before this host was allowed back on the network, it was cleaned up with the assistance of a savvy UNIX administrator. An initial vulnerability scan of the host produced about twenty pages of high- and medium-range security problems. It was scanned again after the changes and upgrades to make sure that no known vulnerabilities existed.

Other Commands to Display Programs Associated with Ports

The UNIX command **lsof** can be used, as well, to list information about files opened by processes. This comes with many UNIX operating systems, but can be downloaded and added if it is not available. To find the process ID associated with the service listening on port 901 using **lsof**, execute the following:

```
lsof -i TCP:901

COMMAND PID USER   FD   TYPE DEVICE SIZE NODE NAME
inetd   387 root   9u   IPv4 369    TCP *:swat (LISTEN)
```

You see that port 901 is associated with the inetd process. This is the Internet daemon that starts most of the listening services. Some additional information is displayed in the last column; port 901 is associated with Samba Web Administration Tool (swat). You should find this started in the file /etc/inetd.conf:

```
grep swat /etc/inetd.conf
swat      stream  tcp     nowait.400      root /usr/sbin/swat swat
```

A Windows tool known as fport (available with a tool search on www.securityfocus.com) can be used to associate processes with ports on which they run. Here is a sample output from running fport on a Windows 2000 host:

```
FPort v1.31 - TCP/IP Process to Port Mapper
Copyright 2000 by Foundstone, Inc.
http://www.foundstone.com
Securing the dot com world

Pid   Process            Port  Proto Path
384   svchost        ->  135   TCP   C:\WINNT\system32\svchost.exe
8     System         ->  445   TCP
496   MSTask         ->  1025  TCP   C:\WINNT\system32\MSTask.exe
8     System         ->  1027  TCP
1692  SshClient      ->  3705  TCP   C:\Program Files\SSH Communications
Security\SSH Secure Shell\SshClient.exe
1892  OUTLOOK        ->  4040  TCP   C:\Program Files\Microsoft
Office\Office\OUTLOOK.EXE

384   svchost        ->  135   UDP   C:\WINNT\system32\svchost.exe
8     System         ->  445   UDP
220   services       ->  1026  UDP   C:\WINNT\system32\services.exe
916   iexplore       ->  1341  UDP   C:\Program Files\Internet
Explorer\iexplore.exe
1892  OUTLOOK        ->  4024  UDP   C:\Program Files\Microsoft
Office\Office\OUTLOOK.EXE
```

Although this turned out to be a non-incident in terms of intrusions, it does illustrate a very noteworthy point. It is extremely helpful to be able to do a quick assessment of potential reconnaissance or potential damage from scan activity of your network. Most NIDS report about scans, notifying you that

they have occurred. But, the more relevant knowledge is this: did any host respond to the scans? That is where TCPdump recorded activity is once again invaluable.

How Slow Can you Go?

This event concerns a remotely monitored site that had poor response time on a good day. One day while attempting to examine activity, the response time became painfully slow. It was so slow, you could type in one character and it would take about 30 seconds to see it echoed back on the screen. This was pretty annoying, but signaled that the site had some issues other than normal poor response time.

Although this was occurring, we were collecting a copy of their Shadow sensor data at our site. In an attempt to explain the poor response time, the site's Shadow events of interest were examined. It showed that they were getting a lot of fragmented activity directed at their network address of 192.168.133.0 (this is a translated address for anonymity purposes). Upon further examination, it was discovered that this had been going on for many hours. Here is a sample of the records that they were getting:

```
12:01:12.150572 dos.com > 192.168.133.0: (frag 54050:1480@4440+)
12:01:17.560572 dos.com > 192.168.133.0: (frag 54050:1480@2960+)
12:01:17.570572 dos.com > 192.168.133.0: (frag 54050:1480@4440+)
12:01:22.200572 dos.com > 192.168.133.0: (frag 54050:1480@1480+)
12:01:22.210572 dos.com > 192.168.133.0: (frag 54050:1480@2960+)
12:01:22.220572 dos.com > 192.168.133.0: (frag 54050:1480@4440+)
12:01:22.230572 dos.com > 192.168.133.0: (frag 54050:1480@5920+)
12:01:27.240572 dos.com > 192.168.133.0: (frag 54050:1480@2960+)
12:01:27.250572 dos.com > 192.168.133.0: (frag 54050:1480@5920+)
12:01:37.230572 dos.com > 192.168.133.0: (frag 54050:1480@1480+)
12:01:37.240572 dos.com > 192.168.133.0: (frag 54050:1480@2960+)
12:01:37.240572 dos.com > 192.168.133.0: (frag 54050:1480@4440+)
12:01:37.250572 dos.com > 192.168.133.0: (frag 54050:1480@5920+)
12:01:42.300572 dos.com > 192.168.133.0: (frag 54050:1480@1480+)
```

You see dos.com sending fragmented packets to the network address. As mentioned, this activity had been happening for several hours. There are a couple of problems with the traffic that need to be examined. See if you can find the three problems associated with fragmentation in the previous TCPdump output.

First, a normal fragmented packet train usually has two or more parts:

- There is an initial fragment that has an offset of 0 and the More Fragments flag set (+):

```
frag 54050:1480@0+
```

Recall that the fragment format is as follows:

```
frag FRAG-ID:BYTES-IN-CURRENT-FRAGMENT@OFFSET-INTO-FRAGMENT-DATA [+]
```

- There might be intermediate fragments that are neither the first nor last fragments. An intermediate fragment has a non-zero offset and the More Fragments flag set. The + flag indicates that the more fragments bit is set or there is another fragment following the one being sent. The More Fragments flag is set in the first and intermediate fragments.
- There is a final fragment, one in which the more fragments bit is not set: no + flag.

This activity appeared on Shadow's hourly wrap-up from the default because both the fragmentation and the destination address having a final octet of 0 (the network address 192.168.133.0).

The fragmentation that is seen in this log has some definite problems:

1. There is no first fragment—one that has an offset of 0.
2. You see repeated offsets for fragments that are in the same fragment train with the fragment ID of 54050. For instance, the fragment offset 4440 is repeated several times.
3. There is no final fragment—one that doesn't have the More Fragments flag (+) set.

It is possible that the offset values are not chronological because the fragments don't necessarily arrive in the order in which they were sent.

After doing some research about the topology of the remote site, we discovered that our sensor was located behind (inside) a packet-filtering device that blocked inbound ICMP echo requests. That is the reason we believe that the initial fragment was never seen. Keep in mind that only the first fragment in the fragment train carries the embedded protocol header, such as the information to say that these packets were associated with ICMP echo requests. We can only surmise that the fragmented activity was associated with the dropped ICMP echo requests.

The packet-filtering device that blocked this activity was a router that did not keep track of state. Therefore, it blocked the first fragment of the fragment train because it was the one that contained the information that this was an ICMP echo request. The router had no means of associating the first fragment with subsequent ones. It appears obvious to us that the subsequent packets all share the same fragment ID and are assumed to be associated with the blocked one. Yet, this router did not maintain that information and allowed the subsequent fragments into the network.

However, this doesn't explain why no final fragment was observed. This should have nothing to do with a router that is incapable of keeping track of state. The only explanation for not receiving a final fragment is that is was intentionally omitted.

Normally, fragments are reassembled by the destination host only and not by intermediate routers through which they travel. However, in this case, the router attempts to reassemble the fragmented packets because they are sent to the network address 192.168.133.0 on which the router resides. This particular router has an old Berkeley Software Distribution (BSD) TCP/IP style stack that responds to this "broadcast" so that it attempts to reassemble the fragments.

The router has limited cache for reassembly. The combination of the repetition of the same fragment ID, the more fragments bit set in every fragment, and the missing first and last fragments severely encumbered the router so that it couldn't do routing work. The router would never time out on reassembly of these packets because it kept seeing evidence that more fragments were coming. This was a successful denial of service (DoS) against the router. When the hostile IP was blocked on an external router, the response time returned to normal.

Why didn't this router expire the incomplete set of fragments with an ICMP "IP reassembly time exceeded" message? After all, isn't this a prime candidate for resource exhaustion, waiting for a fragment or fragments that are never sent? The problem is that for the "IP reassembly time exceeded" message to be delivered and for the receiving host to expire the fragments, the first fragment must be received. Because the outermost router blocked these, the first fragment never arrived, and others could not be expired.

Although some routers block incoming ICMP echo requests, denial of service attacks against the router should not occur for "normal" traffic. The DoS attack succeeded against this particular router because of the broadcast address, the repeated fragment ID, and the missing fragments. After the problem was discovered, the activity was blocked from the hostile source IP address. This blocked all inbound traffic including fragments because the IP address is repeated in each of the IP headers of every fragment.

This was successful, and the response time returned to its normal slow (but not painful) state. The attackers must have sensed this; chances are that the monitored site must have foolishly sent ICMP errors that indicated that their activity was blocked. The attackers responded by attempting the same attack with a different source IP address on the same subnet.

Explanation Acknowledgement and Additional Reference

Many thanks and much credit to Vicki Irwin of the SANS Institute for her assistance in figuring out the router DoS. She referenced the following for a discussion of this and similar exploits:

`www.cisco.com/warp/public/770/nifrag.shtml`

RingZero Worm

Let's wrap up our foray into real-world analysis by examining the RingZero Worm. This worm would probably be considered ancient in Internet time because it was discovered in the latter part of 1999. Plenty has transpired concerning malicious code since that time, yet some of the concepts that can be learned from examination of the worm activity are timeless. This presents a good transition into the next and final chapter of this section that delves more deeply into forensics.

The first indication that the monitored site had some new and unusual activity was that Shadow reported many different attempts to connect to TCP port 3128, the squid web proxy server. These attempted connections occurred many times an hour and were from source hosts from all over the world. Although it has become rather commonplace today with malicious code such as Code Red and nimda to see many different source IPs scanning many different destination IPs, in late 1999, it was a rarity. Here is an excerpt of the kind of activity seen for one hour at the monitored site:

```
12:29:48.230000 4.3.2.1.1049 > 172.16.54.171.3128: S 9779697:9779697(0) win
8192 <mss 1460> (DF)
12:29:58.070000 4.3.2.1.1049 > 172.16.54.171.3128: S 9779697:9779697(0) win
8192 <mss 1460> (DF)
12:30:10.960000 4.3.2.1.1049 > 172.16.54.171.3128: S 9779697:9779697(0) win
8192 <mss 1460> (DF)

12:44:54.960000 1.2.3.4.3243 > 172.16.187.212.3128: S 356330349:356330349(0)
win 8192 <mss 1460> (DF)
12:44:57.930000 1.2.3.4.3243 > 172.16.187.212.3128: S 356330349:356330349(0)
win 8192 <mss 1460> (DF)
12:45:03.930000 1.2.3.4.3243 > 172.16.187.212.3128: S 356330349:356330349(0)
win 8192 <mss 1460> (DF)
12:45:15.930000 1.2.3.4.3243 > 172.16.187.212.3128: S 356330349:356330349(0)
win 8192 <mss 1460> (DF)

12:46:13.070000 1.1.1.1.2262 > 172.16.99.110.3128: S 20315949:20315949(0) win
8192 <mss 1460,nop,nop,sackOK> (DF)
12:46:16.080000 1.1.1.1.2262 > 172.16.99.110.3128: S 20315949:20315949(0) win
8192 <mss 1460,nop,nop,sackOK> (DF)
12:46:22.070000 1.1.1.1.2262 > 172.16.99.110.3128: S 20315949:20315949(0) win
8192 <mss 1460,nop,nop,sackOK> (DF)
```

Three different source IPs—4.3.2.1, 1.2.3.4, and 1.1.1.1—are attempting connections to three different internal destination IP addresses. Because many of the scanned destination IPs in our network were not active, there appeared to be no prior reconnaissance that would target live hosts only. Each source host retries (source ports and TCP sequence numbers do not change) the connection several times because the destination hosts do not respond, and no ICMP error message is returned to indicate that the destination hosts are unreachable. Looking at the timestamps, you can see that these connection attempts occurred at different times during the 12:00 hour.

Our site was not the only one that witnessed this activity; the Naval Surface Warfare Center was also seeing these scans as well as ones to destination port 80 and 8080. Other sites witnessed this activity, and soon, it became apparent that this activity was very widespread.

The initial assessment of the activity was someone attempting to find open web proxy servers. Open proxy servers sometimes offer a "tunnel" through which a hacker can gain access and assume the source IP of the proxy to hide his tracks. At this point, because the traffic was coming from all over the world, one theory was that the source IPs had been spoofed and it was just a handful of hosts involved. Again, this attack pre-dates the notion of distributed denial of service (DDoS) attacks, so we were unaccustomed to dealing with many source hosts to many destination host attacks.

The verbose option (**-vv**) of TCPdump might provide some assistance in determining whether or not the source IPs were spoofed. The same TCPdump records are examined again, but this time using the verbose option:

```
12:29:48.230000 4.3.2.1.1049 > 172.16.54.171.3128: S 9779697:9779697(0) win
8192 <mss 1460> (DF) (ttl 19, id 9072)
12:29:58.070000 4.3.2.1.1049 > 172.16.54.171.3128: S 9779697:9779697(0) win
8192 <mss 1460> (DF) (ttl 19, id 29552)
12:30:10.960000 4.3.2.1.1049 > 172.16.54.171.3128: S 9779697:9779697(0) win
8192 <mss 1460> (DF) (ttl 19, id 39792)

12:44:54.960000 1.2.3.4.3243 > 172.16.187.212.3128: S 356330349:356330349(0)
win 8192 <mss 1460> (DF) (ttl 19, id 962)
12:44:57.930000 1.2.3.4.3243 > 172.16.187.212.3128: S 356330349:356330349(0)
win 8192 <mss 1460> (DF) (ttl 19, id 11714)
12:45:03.930000 1.2.3.4.3243 > 172.16.187.212.3128: S 356330349:356330349(0)
win 8192 <mss 1460> (DF) (ttl 19, id 22466)
12:45:15.930000 1.2.3.4.3243 > 172.16.187.212.3128: S 356330349:356330349(0)
win 8192 <mss 1460> (DF) (ttl 19, id 33218)

12:46:13.070000 1.1.1.1.2262 > 172.16.99.110.3128: S 20315949:20315949(0) win
8192 <mss 1460,nop,nop,sackOK> (DF) (ttl 116, id 35676)
12:46:16.080000 1.1.1.1.2262 > 172.16.99.110.3128: S 20315949:20315949(0) win
8192 <mss 1460,nop,nop,sackOK> (DF) (ttl 116, id 46428)
```

```
12:46:22.070000 1.1.1.1.2262 > 172.16.99.110.3128: S 20315949:20315949(0) win
8192 <mss 1460,nop,nop,sackOK> (DF) (ttl 116, id 57180)
12:46:34.080000 1.1.1.1.2262 > 172.16.99.110.3128: S 20315949:20315949(0) win
8192 <mss 1460,nop,nop,sackOK> (DF) (ttl 116, id 2397)
```

Let's scrutinize these records, but this time in terms of source IP spoofing. The salient advice to remember when looking for spoofed source IPs is to look for similarities in the fields or characteristics of packets. More likely than not, an attacker will not take time to "craft" in differences in the packets, and there will be some traces of unlikely similarities. Conversely, when distinct source IPs truly represent different source hosts, differences in packet characteristics should be apparent. Given this knowledge, what differences can you find among the three different source IPs of the previously shown traffic?

For starters, you pretty much have to do fourth-down-and-punt with the IP identification numbers. The time gaps between when each set of initial connections received is too great to see real trends in IP identification numbers. Ten minutes pass between the first and second set of connections, which is enough time for the IP identification numbers of a busy host to go through all 65,535 numbers and wrap. You would ordinarily look for a chronology of very close IP identification numbers, which would indicate source IP spoofing. But, this can only be done if the time changes are insignificant.

What about the arriving TTL values? They look promising for spoofing because both the first two sets of connections involving source IPs of 1.2.3.4 and 4.3.2.1 have an arriving TTL value of 19. However, the third set from 1.1.1.1 has an arriving TTL value of 116.

Are there any other differences? Look at the TCP options for the connections. The first two source IPs share the same TCP options, a maximum segment size (mss) of 1460. Yet, the third source IP also has a selective acknowledgement (sackOK) that must be padded with two noop's to fall on a 4-byte boundary.

Finally, look at the number of retries per attempted connection and the backoff time between initial tries and retries and between subsequent retries. The first source IP 4.3.2.1 has an initial try and two retries. The backoff time between retries is approximately 10 seconds. Next, IP 1.2.3.4 has one initial try and three retries with the retry attempts doubling in the amount of time before subsequent ones. Finally, the source IP 1.1.1.1 behaves much like 1.2.3.4 as far as retries in that it has three retries with a doubling of the backoff time. From all the forensics from the preceding dump, we can pretty much conclude that these are actual separate source IPs.

When the traffic was observed, we took the TTL values, estimated the initial TTL values, and subtracted the arriving from the initial values. This gave us

the number of hop counts that the datagram took to arrive on the sensor network. Then, we executed a traceroute back to the source IP to see if the expected hop count was close to the actual hop count.

About a dozen traceroutes were attempted; most had a hop count credibly close to the actual hop count. Also, all the targeted IPs were alive, which might not be the case had random IPs been chosen for spoofing. It would be rare if someone were doing mass amounts of spoofing using hand picked live IP numbers only. Usually, it is a far more random selection of spoofed source IP numbers.

This kind of widespread scan was difficult to explain examining one site. Before the days of `www.incidents.org`, Stephen Northcutt asked SANS members to look at traffic at their individual sites and see if they could provide any explanations about the activity. Hundreds of sites reported similar activity.

A couple of sites were able to see the HTTP request that was executed, and it appeared to implicate a host `www.rusftpsearch.net`. The site was available for a few days and it appeared to be collecting IPs of any open proxy servers found.

Ron Marcum of Vanderbilt University discovered a PC on his network that was scanning hosts on other networks looking for ports 80, 8080, and 3128. He discovered a Trojan called RingZero that appeared to be the culprit. At a SANS conference in 1999, conference members and instructors installed the program that was discovered on the Vanderbilt host and examined what it did. They were able to recreate that this Trojan would scan other hosts on web ports.

The suspected infection means is via email or mp3 sharing. But, this seminal malicious code is one of the first that infected hosts and gathered some valuable information from the hosts, and then used the infected hosts to scan other hosts. This is the same model used for scans and attacks today, albeit quite a bit more sophisticated.

Summary

Without unnecessarily belaboring the point, the events described in this chapter have demonstrated the added value of having TCPdump or Shadow running at a site to capture the background traffic. The first incident of a non-intrusion showed how TCPdump can be invaluable because its purpose is not exclusively to show alerts of events of interest, but to capture all traffic. It can provide an audit trail of activity that occurred, or more descriptively in this case, of activity that did not occur.

In addition, TCPdump was used in the scan incident to assess the reaction of hosts on the monitored network to the scan. Scans can be harmless distractions when there is no response by the scanned hosts, or in this case, they can be a reason for concern. Although most NIDS will inform you of scans, none will automatically alert you of responding hosts.

In the third and final events, TCPdump was used to get very specific information about the fragments or packets in order to make more accurate evaluations of the nature of the attack. You can even begin to do forensic investigation about the type of hosts that are conducting the hostile activity. You will see a more thorough discussion of passive analysis of hostile traffic in the next chapter.