10

DEPLOYING FWSNORT

With the theoretical discussion in Chapter 9 on the emulation of Snort rule options within iptables behind us, we'll talk in this chapter about how to get fwsnort to actually do something! Namely, we'll discuss the administration of fwsnort and illustrate how it can be used to instruct iptables to detect attacks that are associated with the Snort signature ruleset.

Installing fwsnort

Like psad, fwsnort comes bundled with its own installation program install.pl. This program handles all aspects of installation, including preserving configurations from a previous installation of fwsnort, the installation of two Perl modules (`Net::IPv4Addr` and `IPTables::Parse`), and the (optional) downloading of the latest Bleeding Snort signature set from http://www.bleedingsnort.com. You can also install fwsnort from the RPM if you are running an RPM-based Linux distribution.

NOTE *As of March 2005, the Snort signature ruleset is only available as part of a for-pay service. Before that date, the Snort rules were available for free from the Snort website (http://www.snort.org). Many security applications (including fwsnort) took advantage of the free rules by providing an automatic update feature to synchronize with the latest Snort rules. While automatically updating in this way is no longer possible, as of this writing the latest Snort rulesets distributed by the Bleeding Snort project are still available for (free) download.*

The fwsnort installer places the `Net::IPvAddr` and `IPTables::Parse` Perl modules within the directory /usr/lib/fwsnort so as to not clutter the system Perl library tree. (This is similar to the installation strategy implemented by psad, as discussed in Chapter 5.)

In order to use fwsnort, you will need to be able to use the iptables string-matching capability. If you are running kernel version 2.6.14 or later, string matching may already be compiled into your kernel.

An easy way to check to see if the running kernel supports the string-matching extension is to attempt to create a string-matching iptables rule against a nonexistent IP address (so that any real network communications are not disrupted), like so:

```
[iptablesfw]# iptables -D INPUT 1 -i lo -d 127.0.0.2 -m string --string
"testing " --algo bm -j ACCEPT
```

If the error `iptables: no chain/target/match by that name` is returned, then the extension is not available in the running kernel. This can be fixed by enabling the `CONFIG_NETFILTER_XT_MATCH_STRING` option in the kernel configuration file, recompiling, and then booting into the new kernel (see "Kernel Configuration" on page 14 for recommended iptables kernel compilation options). If the command above succeeds, then iptables string matching is compatible with your kernel, and you should delete the new rule:

```
[iptablesfw]# iptables -D INPUT 1
```

To install fwsnort-1.0, execute the following commands. (This installer output is somewhat abbreviated but shows the various files that partition the original Snort ruleset, such as backdoor.rules and web-cgi.rules.)

```
[iptablesfw]$ cd /usr/local/src
[iptablesfw]$ wget http://www.cipherdyne.org/fwsnort/download/fwsnort-1.0.tar.bz2
[iptablesfw]$ wget http://www.cipherdyne.org/fwsnort/download/fwsnort-
1.0.tar.bz2.md5
[iptablesfw]$ wget http://www.cipherdyne.org/fwsnort/download/fwsnort-
1.0.tar.bz2.asc
[iptablesfw]$ md5sum -c fwsnort-1.0.tar.bz2.md5
gpg --verify fwsnort-1.0.tar.bz2.asc
gpg: Signature made Sat 21 Apr 2007 09:29:02 AM EDT using DSA key ID A742839F
gpg: Good signature from "Michael Rash <mbr@cipherdyne.org>"
gpg:                 aka "Michael Rash <mbr@cipherdyne.com>"
```

```
fwsnort-1.0.tar.bz2: OK
[iptablesfw]$ tar xfj fwsnort-1.0.tar.bz2
[iptablesfw]$ su -
Password:
[iptablesfw]# cd /usr/local/src/fwsnort-1.0
[iptablesfw]# ./install.pl
[+] mkdir /etc/fwsnort
[+] mkdir /etc/fwsnort/snort_rules
[+] Installing the Net::IPv4Addr Perl module
[+] Installing the IPTables::Parse Perl module
[+] Would you like to download the latest Snort rules from
    http://www.bleedingsnort.com?
    ([y]/n)? y
--22:01:11--  http://www.bleedingsnort.com/bleeding-all.rules
           => `bleeding-all.rules'
Resolving www.bleedingsnort.com... 69.44.153.29
Connecting to www.bleedingsnort.com[69.44.153.29]:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 292,192 [text/plain]
100%[====================================>] 292,192      109.94K/s
22:01:17 (109.77 KB/s) - `bleeding-all.rules' saved [292,192/292,192]
[+] Copying all rules files to /etc/fwsnort/snort_rules
[+] Installing snmp.rules
[+] Installing finger.rules
[+] Installing info.rules
[+] Installing ddos.rules
[+] Installing virus.rules
[+] Installing icmp.rules
[+] Installing dns.rules
[+] Installing rpc.rules
[+] Installing backdoor.rules
[+] Installing scan.rules
[+] Installing shellcode.rules
[+] Installing web-client.rules
[+] Installing web-cgi.rules
[+] Installing exploit.rules
[+] Installing attack-responses.rules
[+] Installing web-attacks.rules
[+] Installing fwsnort.8 man page as /usr/share/man/man8/fwsnort.8
[+] Compressing manpage /usr/share/man/man8/fwsnort.8
[+] Copying fwsnort.conf -> /etc/fwsnort/fwsnort.conf
[+] Copying fwsnort -> /usr/sbin/fwsnort
[+] fwsnort will generate an iptables script located at:
    /etc/fwsnort/fwsnort.sh when executed.
[+] fwsnort has been successfully installed!
```

Running fwsnort

With fwsnort installed on a system that offers string-match support in the kernel, we can now put fwsnort to work for us. Without further ado, we fire up fwsnort from the command line. Normally, fwsnort is executed as root

because by default it queries iptables in order to determine which extensions are available in the running kernel, and then it tailors the translation process accordingly[1] (some output below is abbreviated):

```
[iptablesfw]# fwsnort

    Snort Rules File          Success   Fail      Ipt_apply Total
[+] attack-responses.rules    15        2         0         17
[+] backdoor.rules            62        7         1         69
[+] bad-traffic.rules         10        3         0         13
[+] bleeding-all.rules        1076      573       5         1649
[+] exploit.rules             31        43        0         74
[+] web-cgi.rules             286       62        0         348
[+] web-client.rules          7         10        0         17
[+] web-coldfusion.rules      35        0         0         35
[+] web-frontpage.rules       34        1         0         35
[+] web-iis.rules             103       11        0         114
[+] web-misc.rules            265       61        0         326
[+] web-php.rules             78        48        0         126
[+] x11.rules                 2         0         0         2

                              2725      1761      91        4486
[+] Generated iptables rules for 2725 out of 4486 signatures: 60.74%
[+] Found 91 applicable snort rules to your current iptables policy.
[+] Logfile:         /var/log/fwsnort.log
[+] Iptables script: /etc/fwsnort/fwsnort.sh
```

One of the first things to notice about the fwsnort output is that for each Snort rules file, counters are printed for the number of successfully and unsuccessfully translated rules (`Success` and `Fail`), the number of rules that are applicable to the running iptables policy (`Ipt_apply`), and the total number of Snort rules in the rules file (`Total`).

At the end of the output above, fwsnort prints the total number of Snort rules that could be successfully translated (2,725 out of 4,486). The 60 percent translation rate is obtainable on any Linux system whose kernel has been compiled with support for the iptables `string`, `length`, `tos`, `ttl`, and `ipv4options` matches.

You'll also see printed at the end of the fwsnort output the sentence `Found 91 applicable snort rules to your current iptables policy`. This message indicates that fwsnort has parsed the iptables ruleset that is currently running on the system in order to throw away those Snort rules that iptables would not allow through in the first place. For example, if the iptables policy does not allow connections to an internal HTTP server, then it is of little use to translate Snort rules that deal with inbound HTTP connections initiated from the external network; hence, fwsnort omits such rules from the translation process.

[1] Note that any non-root user with the `CAP_NET_ADMIN` capability can also execute iptables commands.

NOTE *Because the policies constructed by iptables commands can be complex and tricky to parse, fwsnort may not always correctly determine whether an arbitrary type of traffic will be allowed through. You can use the `fwsnort --no-ipt-sync` command-line option to force the translation of as many Snort rules as possible without referencing the underlying iptables policy.*

Finally, the fwsnort output displays two file paths: /var/log/fwsnort.log and /etc/fwsnort/fwsnort.sh.

The fwsnort.log file contains information about the translation process and can be used to determine the reason for the unsuccessful translation of particular Snort rules. For example, the Snort rule identified by `SID 2003306` within the bleeding-all.rules file contains the Snort `pcre` option and is therefore incompatible with iptables. The incompatibility is noted in a log entry within the fwsnort.log file:

```
[-] SID: 2003306  Unsupported option: "pcre" at line: 120. Skipping rule.
```

NOTE *The fwsnort.sh script is the real "meat and potatoes" of fwsnort; it's a Bourne shell script generated by fwsnort that is responsible for implementing the necessary iptables commands to construct the equivalent iptables policy. The internals of this script are discussed in "Structure of fwsnort.sh" on page 179, and a complete fwsnort.sh script can be found in Appendix B.*

Configuration File for fwsnort

The main configuration file for fwsnort, /etc/fwsnort/fwsnort.conf, defines networks, port numbers, paths to system binaries (such as the path to iptables), and other key pieces of information needed for proper execution.

As with psad, the fwsnort.conf file follows a simple key/value format, and many of the keywords and semantics are identical to those found in Snort's own configuration file. For example, both the `HOME_NET` and `EXTERNAL_NET` keywords are defaulted to the wildcard value `any`, and lists of IP addresses and/or networks can be enclosed within braces. (Nearly all Snort rules use some combination of the `HOME_NET` and `EXTERNAL_NET` keywords.) The notion of variable resolution is also supported; that is, `HTTP_SERVERS` maps to `$HOME_NET`, which in turn maps to a specific network (or networks) or the wildcard value `any`, for example.

You'll find a complete example fwsnort.conf file below (and at http://www.cipherdyne.org/LinuxFirewalls), and all fwsnort usage examples in this book will reference this configuration file. In this case, the network protected by the iptables firewall on which fwsnort is deployed is the Class C network 192.168.10.0/24 (see Figure 1-2), so we set `HOME_NET` accordingly.

```
[iptablesfw]# cat /etc/fwsnort/fwsnort.conf
#  This is the configuration file for fwsnort. There are some similarities
#  between this file and the configuration file for Snort.
# $Id: fwsnort.conf 356 2007-03-20 01:31:28Z mbr $
```

```
### fwsnort treats all traffic directed to / originating from the local
### machine as going to / coming from the HOME_NET in Snort rule parlance.
### If there is only one interface on the local system, then there will be
### no rules processed via the FWSNORT_FORWARD chain because no traffic
### would make it into the iptables FORWARD chain.
HOME_NET                192.168.10.0/24;
EXTERNAL_NET            any;
### List of servers. fwsnort supports the same variable resolution as Snort.
HTTP_SERVERS            $HOME_NET;
SMTP_SERVERS            $HOME_NET;
DNS_SERVERS             $HOME_NET;
SQL_SERVERS             $HOME_NET;
TELNET_SERVERS          $HOME_NET;
### AOL AIM server nets
AIM_SERVERS        [64.12.24.0/24, 64.12.25.0/24, 64.12.26.14/24, 64.12.28.0/24,
64.12.29.0/24, 64.12.161.0/24, 64.12.163.0/24, 205.188.5.0/24, 205.188.9.0/24];
### Configurable port numbers
SSH_PORTS  22;
HTTP_PORTS          80;
SHELLCODE_PORTS     !80;
ORACLE_PORTS        1521;
### Define average packet lengths and maximum frame length. This is used
### for iptables length match emulation of the Snort dsize option.
❶ AVG_IP_HEADER_LEN       20;   ### IP options are not usually used.
AVG_TCP_HEADER_LEN      40;   ### Includes options
MAX_FRAME_LEN           1500;
### Use the WHITELIST variable to define a list of hosts/networks that
### should be completely ignored by fwsnort. For example, if you want
### to whitelist the IP address 192.168.10.1 and the network 10.1.1.0/24,
### you will use (note that you can also specify multiple WHITELIST
### variables, one per line):
#WHITELIST             192.168.10.1, 10.1.1.0/24;
❷ WHITELIST               NONE;
### Use the BLACKLIST variable to define a list of hosts/networks
### that for which fwsnort should DROP or REJECT all traffic. For
### example, to DROP all traffic from the 192.168.10.0/24 network,
### you can use:
###     BLACKLIST           192.168.10.0/24    DROP;
### To have fwsnort REJECT all traffic from 192.168.10.0/24,
### you would use:
###     BLACKLIST           192.168.10.0/24    REJECT;
BLACKLIST               NONE;
### Define the jump position in the built-in chains to jump to
### the fwsnort chains.
❸ FWSNORT_INPUT_JUMP      1;
FWSNORT_OUTPUT_JUMP     1;
FWSNORT_FORWARD_JUMP    1;
### iptables chains (these do not normally need to be changed)
FWSNORT_INPUT           FWSNORT_INPUT;
FWSNORT_INPUT_ESTAB     FWSNORT_INPUT_ESTAB;
FWSNORT_OUTPUT          FWSNORT_OUTPUT;
FWSNORT_OUTPUT_ESTAB    FWSNORT_OUTPUT_ESTAB;
FWSNORT_FORWARD         FWSNORT_FORWARD;
FWSNORT_FORWARD_ESTAB   FWSNORT_FORWARD_ESTAB;
### System binaries
```

```
shCmd           /bin/sh;
echoCmd         /bin/echo;
tarCmd          /bin/tar;
wgetCmd         /usr/bin/wget;
unameCmd        /usr/bin/uname;
ifconfigCmd     /sbin/ifconfig;
iptablesCmd     /sbin/iptables;
```

At ❶ above, the fwsnort.conf file sets the average length for the IP and TCP headers. This is necessary because the iptables length match begins at the IP header, whereas the Snort `dsize` option applies only the application layer data associated with a packet. By specifying the average header lengths, fwsnort can approximate the `dsize` option to assist in the translation process.

At ❷ we can add a whitelist and a blacklist; see "Setting Up Whitelists and Blacklists" on page 191.

At ❸ the position of the jump rule into the fwsnort chains within each of the built-in chains is defined. By default the jump rule position is the very first rule within each of these chains, but you can alter this to your liking by changing these variables around. This is not usually necessary unless you have an iptables policy that has inspection or filtering requirements that must be met before fwsnort has a chance to inspect packets.

Structure of fwsnort.sh

The Bourne shell script /etc/fwsnort/fwsnort.sh generated by fwsnort is divided into five sections. The first section is a header constructed out of comments that includes a short blurb about the purpose of the fwsnort.sh script, the command-line arguments given to fwsnort to generate fwsnort.sh, and the version of fwsnort:

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

# File:  /etc/fwsnort/fwsnort.sh

# Purpose:  This script was auto-generated by fwsnort and implements an
#           iptables ruleset based upon Snort rules. For more information,
#           see the fwsnort man page or the documentation available at
#           http://www.cipherdyne.org/fwsnort.

# Generated with:     fwsnort -no-ipt-sync
# Generated on host:  iptablesfw
# Generated at:       Sun Jul 15 23:12:43 2007

# Author:  Michael Rash <mbr@cipherdyne.org>

# Version: 1.0 (file revision: 381)
```

The second section of the fwsnort.sh script defines paths to the iptables and echo system binaries. These paths are inherited from the `iptablesCmd` and `echoCmd` keywords in the fwsnort.conf configuration file, and fwsnort checks to be sure

that the paths make sense before building fwsnort.sh. However, the fwsnort.sh script does not necessarily have to be executed on the same system where fwsnort is installed. In fact, from a security perspective, it is better not to have Perl or any other highly capable interpreter or compiler installed on a dedicated firewall device that is not strictly necessary from an operations perspective.[2]

The configuration section allows the paths to be tweaked easily for the eventual system on which fwsnort.sh is deployed:

```
ECHO=/bin/echo
IPTABLES=/sbin/iptables
```

The third section in fwsnort.sh is responsible for building dedicated iptables chains for fwsnort rules. All fwsnort rules, with the exception of the jump rules discussed below, are added to these custom chains to maintain strict separation from any existing iptables policy.

The names given to fwsnort chains broadly describe the type of traffic inspection that is performed within each chain. For example, the `FWSNORT_INPUT` chain is for the inspection of traffic that is directed at the local system and is therefore governed by the iptables `INPUT` chain. Similarly, the `FWSNORT_OUTPUT` chain only applies to packets that originate from the firewall system itself (via the `OUTPUT` chain), and the `FWSNORT_FORWARD` chain governs packets that are destined to be forwarded through the local system (via the `FORWARD` chain).

TCP Connection States and fwsnort Chains

Because of the relative importance of applying Snort rules to established TCP sessions through the use of the Snort `flow: established` option, fwsnort creates special chains for such rules. The names for these chains simply append the string `_ESTAB` to each of the fwsnort chains mentioned previously. Once all of the fwsnort chains have been created, jump rules are added that use the iptables state match to send TCP packets that are part of established sessions to the appropriate `_ESTAB` chain. For example, packets in the `FWSNORT_INPUT` chain are jumped to the `FWSNORT_INPUT_ESTAB` chain, as shown here:

```
############ Create fwsnort iptables chains. ############
$IPTABLES -N FWSNORT_INPUT 2> /dev/null
$IPTABLES -F FWSNORT_INPUT
$IPTABLES -N FWSNORT_INPUT_ESTAB 2> /dev/null
$IPTABLES -F FWSNORT_INPUT_ESTAB
$IPTABLES -N FWSNORT_OUTPUT 2> /dev/null
$IPTABLES -F FWSNORT_OUTPUT
$IPTABLES -N FWSNORT_OUTPUT_ESTAB 2> /dev/null
$IPTABLES -F FWSNORT_OUTPUT_ESTAB
$IPTABLES -N FWSNORT_FORWARD 2> /dev/null
$IPTABLES -F FWSNORT_FORWARD
$IPTABLES -N FWSNORT_FORWARD_ESTAB 2> /dev/null
$IPTABLES -F FWSNORT_FORWARD_ESTAB
############ Inspect ESTABLISHED tcp connections. ############
```

[2] For more information on host security issues and hardening strategies, Bastille Linux (http://www.bastille-linux.org) provides lots of great educational information, along with the ability to automatically harden various Linux distributions.

```
$IPTABLES -A FWSNORT_INPUT -p tcp -m state --state ESTABLISHED -j
FWSNORT_INPUT_ESTAB
$IPTABLES -A FWSNORT_OUTPUT -p tcp -m state --state ESTABLISHED -j
FWSNORT_OUTPUT_ESTAB
$IPTABLES -A FWSNORT_FORWARD -p tcp -m state --state ESTABLISHED -j
FWSNORT_FORWARD_ESTAB
```

Signature Inspection and Log Generation

The fourth section of fwsnort.sh is where the heavyweight packet inspection takes place. All of the rules within this section are added to one of the fwsnort chains mentioned above. Each rule contains elements from the Snort rule header and rule options such as source and destination IP addresses and port numbers, and content strings, `length`, `ttl`, or `tos` matches, and so on.

By default, every Snort rule translated by fwsnort results in an iptables command that uses the `LOG` target along with a logging prefix that is designed to communicate signature specifics to the user. The logging prefixes built by fwsnort contain the rule number within the fwsnort chain and the Snort signature ID value, and they indicate whether the signature is logged from an established TCP connection.

For example, the first rule in the `FWSNORT_FORWARD_ESTAB` chain contains a logging prefix that is built up from the Volume Serial Number signature (Snort ID 1292) and looks like this: `[1] SID1292 ESTAB`.

By default each iptables `LOG` rule makes use of the comment match to annotate the rule with the Snort `sid`, `msg`, `classtype`, `rev`, and `reference` fields, and the fwsnort version number. For example, for Snort rule ID 1292, the associated comment is:

```
sid:1292; msg:ATTACK-RESPONSES directory listing; classtype: bad-unknown; rev: 9;
FWS:1.0
```

Below is the signature section of the fwsnort.sh script. (Note that the iptables rules are organized by the corresponding Snort rules file.)

```
############ attack-responses.rules ############

$ECHO "[+] Adding attack-responses rules."
### alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"ATTACK-RESPONSES
directory listing"; flow:established; content:"Volume Serial Number";
classtype:bad-unknown; sid:1292; rev:9;)
$IPTABLES -A FWSNORT_FORWARD_ESTAB -s 192.168.10.0/24 -p tcp -m string --string
"Volume Serial Number" --algo bm -m comment --comment "sid:1292; msg:
ATTACK-RESPONSES directory listing; classtype: bad-unknown; rev: 9; FWS:1.0;"
-j LOG --log-ip-options --log-tcp-options --log-prefix "[1] SID1292 ESTAB "
$IPTABLES -A FWSNORT_OUTPUT_ESTAB -p tcp -m string --string "Volume Serial
Number" --algo bm -m comment --comment "sid:1291; msg: ATTACK-RESPONSES
directory listing; classtype: bad-unknown; rev: 9; FWS:1.0;" -j LOG
--log-ip-options --log-tcp-options --log-prefix "[1] SID1292 ESTAB "
### alert tcp $HTTP_SERVERS $HTTP_PORTS -> $EXTERNAL_NET any (msg:"ATTACK-
RESPONSES command completed"; flow:established; content:"Command completed";
nocase; reference:bugtraq,1806; classtype:bad-unknown; sid:494; rev:10;)
```

```
$IPTABLES -A FWSNORT_FORWARD_ESTAB -s 192.168.10.0/24 -p tcp --sport 80 -m
string --string "Command completed" --algo bm -m comment --comment "sid:494;
msg: ATTACK-RESPONSES command completed; classtype: bad-unknown; reference:
bugtraq,1806; rev: 10; FWS:1.0;" -j LOG --log-ip-options --log-tcp-options
--log-prefix "[2] SID494 ESTAB "
$IPTABLES -A FWSNORT_OUTPUT_ESTAB -p tcp --sport 80 -m string --string "Command
completed" --algo bm -m comment --comment "sid:494; msg: ATTACK-RESPONSES
command completed; classtype: bad-unknown; reference: bugtraq,1806; rev: 10;
FWS:1.0;" -j LOG --log-ip-options --log-tcp-options --log-prefix "[2] SID494
ESTAB "
```

Activating the fwsnort Chains with Jump Rules

The final section in fwsnort.sh makes the whole ruleset active within the kernel by directing iptables to send traffic through these rules. All of the iptables commands executed by fwsnort.sh up until this point simply load the fwsnort policy into the running kernel.

Because there are not yet any jump rules to send packets from the built-in iptables chains into the fwsnort chains, we have utilized only kernel memory so far; none of the rules can yet interact with packets as they flow within the kernel. This changes with the final six commands, which first delete any existing fwsnort jump rule[3] and then make the very first rule in each of the `INPUT`, `OUTPUT`, and `FORWARD` chains jump all packets to the respective fwsnort chain. (The jump rules are the only rules added by fwsnort to any of the built-in iptables chains.)

```
$IPTABLES -D FORWARD -i ! lo -j FWSNORT_FORWARD 2> /dev/null
$IPTABLES -I FORWARD 1 -i ! lo -j FWSNORT_FORWARD
$IPTABLES -D INPUT -i ! lo -j FWSNORT_INPUT 2> /dev/null
$IPTABLES -I INPUT 1 -i ! lo -j FWSNORT_INPUT
$IPTABLES -D OUTPUT -o ! lo -j FWSNORT_OUTPUT 2> /dev/null
$IPTABLES -I OUTPUT 1 -o ! lo -j FWSNORT_OUTPUT
```

NOTE *See Appendix B for an example fwsnort.sh script that translates the web-attacks Snort rules file into an equivalent iptables policy.*

Command-Line Options for fwsnort

There are many command-line options for fwsnort that you can use to influence its execution, and we'll cover some of the more commonly used ones here. (You'll find an exhaustive treatment of all command-line arguments in the fwsnort(8) man page.)

`--ipt-drop` This option instructs fwsnort to drop packets before they are forwarded to their intended target, in addition to logging them. (By default, fwsnort only logs malicious packets.) This grants fwsnort the authority to actively respond to network attacks.

[3] This makes it possible to execute the fwsnort.sh script multiple times and maintain a clean interface with an existing iptables policy since only one fwsnort jump rule can exist for each built-in chain. Versions of fwsnort prior to 1.0 had a bug where additional jump rules were added if the fwsnort.sh script was executed multiple times.

--ipt-reject This option instructs fwsnort to build an iptables policy that utilizes the REJECT target to tear down malicious TCP connections with TCP Reset packets, and to respond against malicious UDP traffic with an ICMP Port Unreachable message.

--snort-conf *path* This option instructs fwsnort to read variables such as HOME_NET, EXTERNAL_NET, HTTP_SERVERS, and so on directly from an existing Snort configuration file (usually located at /etc/snort/snort.conf). There is nothing to prevent Snort and fwsnort from running on the same system. This remains true even when Snort is running in inline mode, because fwsnort rules are sectioned off within their own chains; packets can be jumped to these chains before hitting a QUEUE rule within the iptables policy.

--snort-sid *sids* This option allows the translation efforts of fwsnort to be restricted to a specific Snort ID or a list of Snort IDs. This is most useful when a new vulnerability is announced in a piece of software that is protected by an iptables firewall and a new signature is released by the Snort community to detect an attack that exploits this vulnerability. By using fwsnort with the --snort-sid option, we can quickly deploy a new policy to log and/or drop malicious packets that are associated with this new attack.

--include-type *type* This option instructs fwsnort to translate only Snort rules that are contained within a single rules file. For example, to translate the rules from the backdoor.rules file, one would use --include-type backdoor on the fwsnort command line. A comma-separated list of types is also supported, such as --include-type ftp,mysql.

--ipt-list This option displays all active rules in the various fwsnort chains. These include FWSNORT_INPUT, FWSNORT_INPUT_ESTAB, FWSNORT_OUTPUT, FWSNORT_OUTPUT_ESTAB, FWSNORT_FORWARD, and FWSNORT_FORWARD_ESTAB.

--ipt-flush This option flushes all active rules in the fwsnort chains. This is useful for quickly removing fwsnort rules without removing other iptables rules associated with an existing policy.

--no-addresses This option forces fwsnort to not reference IP addresses associated with any interfaces on the firewall system. This option is most useful if fwsnort is deployed on a bridging firewall that has no IP addresses assigned to its interfaces.

--no-ipt-sync This option instructs fwsnort to disable all compatibility checks that are normally run against the local iptables policy. The resulting fwsnort policy will not skip any rules that detect traffic that the firewall is configured to not accept in the first place.

--restrict-intf *intf* This option restricts fwsnort rules to the specified interface (or interfaces). By default, fwsnort does not inspect traffic over the loopback interface but inspects traffic on all other interfaces. To have fwsnort inspect traffic over, say, the eth0 and eth1 interfaces only, you would use --restrict-intf eth0,eth1.

Observing fwsnort in Action

Illustrating fwsnort operations with specific example attacks is a practical way to see how fwsnort functions and how to put it to good use. In this section we'll cover a set of attacks derived from the Snort ruleset, and we'll see how fwsnort detects and (optionally) reacts to these attacks. By default, a policy built by fwsnort behaves like an intrusion detection system in the sense that attacks are only logged via the `LOG` target; no attempt is made to drop packets, reset TCP connections, or generate ICMP error code packets. However, we can quickly turn this passive stance into an active one by using the `--ipt-reject` or `--ipt-drop` command-line arguments to fwsnort, as we'll see in the following examples.

Detecting the Trin00 DDoS Tool

Trin00 is a classic tool for mounting a Distributed Denial of Service (DDoS) attack by sending large quantities of UDP packets against a target in a simultaneous flood from multiple attack nodes. Trin00 implements its own methods for coordinating the efforts of the attack nodes, and the Snort signature set devotes several signatures to detecting Trin00 administrative communications. For example, Snort ID 237 looks for the string `l44adsl` contained within a UDP packet destined for port 27444 on the home network. This string is the default password that a Trin00 control node uses to authenticate to an endpoint node in order to instruct it to perform particular operations, and is included within Snort rule ID 237:

```
alert udp $EXTERNAL_NET any -> $HOME_NET 27444 (msg:"DDOS Trin00 Master to
Daemon default password attempt"; content:"l44adsl"; reference:arachnids,197;
classtype:attempted-dos; sid:237; rev:2;)
```

Using fwsnort, we recast the Snort rule into equivalent iptables rules:

```
[iptablesfw]# fwsnort --snort-sid 237
[+] Parsing Snort rules files...
[+] Found sid: 237 in ddos.rules
    Successful translation.
```

Here is the resulting iptables rule in the `FWSNORT_FORWARD` chain.

```
$IPTABLES -A FWSNORT_FORWARD -d 192.168.10.0/24 -p udp --dport 27444 -m string
--string "l44adsl" --algo bm -m comment --comment "sid:237; msg: DDOS Trin00
Master to Daemon default password attempt; classtype: attempted-dos; reference:
arachnids,197; rev: 2; FWS:1.0;" -j LOG --log-ip-options --log-prefix "[1]
SID237 "
```

Because this is a UDP signature, there is no notion of an established connection, and hence the signature belongs in the `FWSNORT_FORWARD` chain instead of the `FWSNORT_FORWARD_ESTAB` chain. In addition, even though the default policy in this book (see "Default iptables Policy" on page 20) does not accept

UDP packets destined for port 27444, fwsnort can still detect packets that match the Trin00 signature because a connection does not have to be established before data can be sent (as in the case of TCP signatures). That is, we don't need an ACCEPT rule before data can be sent over the UDP socket from the client. This is a fundamental difference between TCP and UDP sockets.

Now, from the ext_scanner system, we execute the following command to see if the signature triggers:

```
[ext_scanner]$ echo "l44adsl" | nc -u 71.157.X.X 27444
```

The iptables log faithfully reports the signature match:

```
[iptablesfw]# grep SID237 /var/log/messages | tail -n 1
Jul 19 22:18:24 iptablesfw kernel: [1] SID237 IN=eth0 OUT=
MAC=00:13:d3:38:b6:e4:00:30:48:80:4e:37:08:00 SRC=144.202.X.X DST=71.157.X.X
LEN=36 TOS=0x00 PREC=0x00 TTL=64 ID=42386 DF PROTO=UDP SPT=54494 DPT=27444
LEN=16
```

In bold above is the iptables log prefix [1] SID237 from the ext_scanner system—indeed, fwsnort has detected the (simulated) attack.

Detecting Linux Shellcode Traffic

Because exploit developers sometimes share some of the same shellcode, the shellcode.rules file in the Snort signature set looks for this common base of bytes in network traffic. The content field in the following signature shows a smattering of common shellcode used against Linux systems:

```
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any (msg:"SHELLCODE Linux
shellcode"; content:"|90 90 90 E8 C0 FF FF FF|/bin/sh";
reference:arachnids,343; classtype:shellcode-detect; sid:652; rev:9;)
```

Translating this signature with fwsnort --snort-sid 652 builds the iptables command below. While the original Snort rule applies to all IP traffic, the destination port requirement forces iptables to match only on TCP or UDP packets.

Here is the translated Snort rule applied to TCP traffic:

```
$IPTABLES -A FWSNORT_FORWARD -d 192.168.10.0/24 -p tcp --sport ! 80 -m string
--hex-string "|90 90 90 E8 C0 FF FF FF|/bin/sh" --algo bm -m comment --comment
"sid:652; msg: SHELLCODE Linux shellcode; classtype: shellcode-detect;
reference: arachnids,343; rev: 9; FWS:1.0;" -j LOG --log-ip-options
--log-tcp-options --log-prefix "[1] SID652 "
```

To trigger the signature match within iptables, first execute the fwsnort.sh script on the iptablesfw system, and then execute the Perl command below from the ext_scanner system. As required by the signature, the source port of

the TCP session built by Netcat is not port 80, since it chooses a random high port above 1024 according to how the local TCP stack instantiates a client TCP socket:

```
[iptablesfw]# /etc/fwsnort/fwsnort.sh
[+] Adding shellcode rules.
    Rules added: 2
[ext_scanner]$ perl -e 'print "\x90\x90\x90\xE8\xC0\xFF\xFF\xFF/bin/sh"' | nc
71.157.X.X 80
```

The simulated attack is caught by iptables, and this log message appears:

```
[iptablesfw]# grep SID652 /var/log/messages | tail -n 1
Jul 19 23:48:18 iptablesfw kernel: [1] SID652 IN=eth0 OUT=eth1 SRC=144.202.X.X
DST=192.168.10.3 LEN=67 TOS=0x00 PREC=0x00 TTL=63 ID=570 DF PROTO=TCP SPT=54629
DPT=80 WINDOW=92 RES=0x00 ACK PSH URGP=0 OPT (0101080A2B3139EFAD325718)
```

This shows that fwsnort, with guidance from the Snort signature set, is effective at detecting the simulated attack.

Detecting and Reacting to the Dumador Trojan

In recent years, malware authors have elevated the stakes in computer security. With a rich target environment provided primarily by unpatched Windows systems with broadband connectivity to the Internet, the damaging effects of malware designed specifically to gather financial and other personal data can be enormous.

The Dumador trojan is malware that contains both a keylogger (for collecting and transmitting sensitive information typed on a keyboard back to an attacker), and a backdoor server that listens on ports 9125 and 64972. The Bleeding Snort ruleset contains a signature designed to detect when the Dumador trojan attempts to send information back to an attacker via a web session, as shown here:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"BLEEDING-EDGE
TROJAN Dumador Reporting User Activity"; flow:established,to_server;
uricontent:".php?p="; nocase; uricontent:"?machineid="; nocase;
uricontent:"&connection="; nocase; uricontent:"&iplan="; nocase;
classtype:trojan-activity; reference:url,www.norman.com/Virus/
Virus_descriptions/24279/; sid:2002763; rev:2;)
```

This signature is particularly interesting in the context of fwsnort because it requires multiple application layer content matches. In order to translate the signature, we execute the following:

```
[iptablesfw]# fwsnort --snort-sid 2002763
[+] Parsing Snort rules files...
[+] Found sid: 2002763 in bleeding-all.rules
    Successful translation.
```

This results in the lengthy iptables command you see below, which searches for each of the strings required by the original Bleeding Snort rule by using the iptables string match four times (as shown in bold):

```
$IPTABLES -A FWSNORT_FORWARD_ESTAB -s 192.168.10.0/24 -p tcp --dport 80 -m
string --string ".php?p=" --algo bm -m string --string "?machineid=" --algo
bm -m string --string "&connection=" --algo bm -m string --string "&iplan="
--algo bm -m comment --comment "sid:2002763; msg: BLEEDING-EDGE TROJAN
Dumador Reporting User Activity; classtype: trojan-activity; reference:
url,www.norman.com/Virus/Virus_descriptions/24279/; rev: 2; FWS:1.0;" -j LOG
--log-ip-options --log-tcp-options --log-prefix "[1] SID2002763 ESTAB "
```

Now we make the signature active in the Linux kernel by executing the fwsnort.sh script:

```
[iptablesfw]# /etc/fwsnort/fwsnort.sh
[+] Adding bleeding-all rules.
    Rules added: 2
```

With the signature active, it is time to test it, and for this we refer to the network diagram in Figure 1-2. On the system labeled lan_client, we execute the following Perl command (the usage of the `A` character is optional and just provides filler data between the separate match criteria) and pipe the output through Netcat to direct it to the webserver labeled ext_web:

```
[lan_client]$ perl -e 'print
".php?p=AAAAA?machineid=AAAAA&connection=AAAAA&iplan="' | nc 12.34.X.X 80
```

On the firewall system, iptables catches the activity and outputs this succinct log message:

```
[iptablesfw]# grep SID2002763 /var/log/messages | tail -n 1
Jul 20 01:12:53 iptablesfw kernel: [1] SID2002763 ESTAB IN=eth1 OUT=eth0
SRC=192.168.10.3 DST=12.34.X.X LEN=104 TOS=0x00 PREC=0x00 TTL=63 ID=17247 DF
PROTO=TCP SPT=55040 DPT=80 WINDOW=1460 RES=0x00 ACK PSH URGP=0 OPT
(0101080AAD7FC90A2B44969B)
```

With a rule in place to detect when the Dumador trojan attempts to call home with a juicy payload of information, fwsnort can refuse to play nicely by forcing Dumador's TCP session to close by using the `--ipt-reject` command-line argument:

```
[iptablesfw]#  fwsnort --snort-sid 2002763 --ipt-reject
[+] Parsing Snort rules files...
[+] Found sid: 2002763 in bleeding-all.rules
    Successful translation.
[iptablesfw]# /etc/fwsnort.fwsnort.sh
[+] Adding bleeding-all rules.
    Rules added: 4
```

Now, rerunning our simulation results in a different iptables log message. (The logging prefix `[1] REJ SID2002763` indicates that fwsnort took action against the web session by generating a RST.)

```
[iptablesfw]# grep SID2002763 /var/log/messages | tail -n 1
Jul 20 01:16:41 iptablesfw kernel: [1] REJ SID2002763 ESTAB IN=eth1 OUT=eth0
SRC=192.168.10.3 DST=12.34.X.X LEN=104 TOS=0x00 PREC=0x00 TTL=63 ID=17507 DF
PROTO=TCP SPT=39786 DPT=80 WINDOW=1460 RES=0x00 ACK PSH URGP=0 OPT
(0101080AAD8346092B4575DD)
```

In this particular case, if you are running a network of Windows systems as a part of a financial institution (for example), it might make good sense to take punitive action like the above against network traffic that matches the Dumador signature. The risk of tearing down legitimate connections might be less than the risk of losing important financial data.

Detecting and Reacting to a DNS Cache-Poisoning Attack

In February 2005, it was discovered that the default configuration of Windows NT 4 and 2000 DNS servers and some Symantec Gateway products left them open to a DNS cache-poisoning attack.[4] This vulnerability was exploited on the Internet by an attack in which a set of rogue DNS servers was used to advertise false DNS records to vulnerable downstream DNS servers so that legitimate user requests for some domains could be directed to IP addresses of the attacker's choosing.

To make an arbitrary DNS server "downstream" from one of the rogue DNS servers, the attacker just needed to get the targeted server to issue a DNS request to the rogue server. This could be accomplished in a variety of ways, such as sending an email to a bogus user, thus eliciting a non-delivery report (NDR) to the source domain—this requires a mail server to be running on the targeted network, or by issuing a request to the malicious server from a previously installed piece of spyware.

In the bleeding-all.rules file provided by http://www.bleedingsnort.com, Snort ID 2001842 detects when a system that is part of the internal network issues a DNS request for one of the malicious domains that took part in the DNS cache-poisoning attack, 7sir7.com. We can have fwsnort alert us to this fact by translating the rule into an iptables policy and executing the resulting fwsnort.sh script:

```
[iptablesfw]# fwsnort --snort-sids 2001842
[+] Parsing Snort rules files...
[+] Found sid: 2001842 in bleeding-all.rules
    Successful translation.
[iptablesfw]# /etc/fwsnort/fwsnort.sh
[+] Adding bleeding-all rules.
    Rules added: 2
```

[4] See http://isc.sans.org/presentations/dnspoisoning.php for a comprehensive write-up of the DNS cache-poisoning attack and the strategy used by the attackers.

The original Snort rule identified by SID 2001842 and its iptables equivalent appear in the FWSNORT_FORWARD chain to which packets are jumped from the built-in FORWARD chain:

```
alert udp $HOME_NET any -> any 53 (msg: "BLEEDING-EDGE Possible DNS Lookup for
DNS Poisoning Domain 7sir7.com"; content:"|05|7sir7|03|com"; nocase;
reference:url,isc.sans.org/diary.php?date=2005-04-07; classtype: misc-
activity; sid:2001842; rev:3;)

$IPTABLES -A FWSNORT_FORWARD -p udp --dport 53 -m string --hex-string " 05|
7sir7|03|com" --algo bm -m comment --comment "sid:2001842; msg:BLEEDING-EDGE
Possible DNS Lookup for DNS Poisoning Domain 7sir7.com; classtype:misc-
activity; reference:url,isc.sans.org/diary.php?date=2005-04-07; rev:3;
FWS:1.0;" -j LOG --log-ip-options --log-prefix "[1] SID2001842 "
```

In order to show that the fwsnort rule actually works, we simulate the traffic needed to cause a signature match from an internal host. Again, we use the network diagram in Figure 1-2 to help illustrate this example.

The dnsserver host simulates a request as if it does not yet have an "A" record mapping www.7sir7.com to an IP address, and so it must issue a request that will eventually query the authoritative (malicious) DNS server for the 7sir7.com domain. We don't need (or want!) an internal system that is actually vulnerable to the cache-poisoning attack in order to test whether our fwsnort ruleset works; it is sufficient to manufacture a UDP packet that contains the consecutive bytes |05|7sir7|03|com from any system on the internal network to any external IP address with a destination port of 53.

We can easily craft this packet by using the single Perl command shown below on the dnsserver system and piping the output to Netcat to send it over the network to an IP address that represents a malicious DNS server:

```
[dnsserver]$ perl -e 'print "\x057sir7\x03com"' | nc -u 234.50.X.X 53
```

On the iptablesfw firewall system, we see that, indeed, iptables has detected the suspicious packet and has created the following log message in /var/log/messages (note the [1] SID2001842 logging prefix):

```
[iptablesfw]# grep SID2001842 /var/log/messages | tail -n 1
Jul  7 22:31:43 iptablesfw kernel: [1] SID2001842 IN=eth1 OUT=eth0
SRC=192.168.10.4 DST=234.50.X.X LEN=38 TOS=0x00 PREC=0x00 TTL=62 ID=36070 DF
PROTO=UDP SPT=16408 DPT=53 LEN=18
```

Because we did not supply either the --ipt-drop or --ipt-reject command-line arguments to fwsnort when we translated the cache-poisoning signature, iptables made no effort to prevent the suspicious packet from exiting the network. We can confirm this by running a packet trace on the external interface of the firewall and executing the same Perl command above:

```
[iptablesfw]# tcpdump -i eth0 -l -nn port 53 and host 234.50.X.X -s 0 -X
tcpdump: verbose output suppressed, use -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
22:41:22.683862 IP 71.157.X.X.16414 > 234.50.X.X.53: [|domain]
```

```
0x0000:  4500 0026 64fc 4000 3e11 fce1 0000 0000  E..&d.@.>.......
0x0010:  0000 0000 401e 0035 0012 86e5 0537 7369  D0..@..5.....7si
0x0020:  7237 0363 6f6d                           r7.com \
```

In the tcpdump output shown in bold above are the hex codes that show the exact application layer data associated with the cache-poisoning signature. This proves the packet is forwarded through the iptables firewall.

But fwsnort does not need to remain complacent and just log the DNS cache-poisoning attack above. In this example, we instruct it to drop the DNS request to the cache-poisoning domain, redeploy the resulting iptables policy, simulate the request from the dnsserver system once again, and examine the iptables log:

```
[iptablesfw]# fwsnort --snort-sids 2001842 --ipt-drop
[+] Parsing Snort rules files...
[+] Found sid: 2001842 in bleeding-all.rules
    Successful translation.
[iptablesfw]# /etc/fwsnort/fwsnort.sh
[+] Adding bleeding-all rules.
    Rules added: 2
[dnsserver]$ perl -e 'print "\x05sir7\x03com"' | nc -u 234.50.X.X 53
[iptablesfw]# grep SID2001842 /var/log/messages |tail -n 1
Jul  7 22:33:42 fw kernel: [1] DRP SID2001842 IN=eth1 OUT=eth0 SRC=192.168.10.4
DST=234.50.X.X LEN=38 TOS=0x00 PREC=0x00 TTL=62 ID=36070 DF PROTO=UDP SPT=16408
DPT=53 LEN=18
```

This time, the logging prefix has changed. Instead of just

```
[1] SID2001842
```

we now have

```
[1] DRP SID2001842
```

The `DRP` string indicates that iptables has dropped the DNS request in addition to logging it. This is confirmed by once again running a packet trace on the external firewall interface and seeing that the request never makes it through.

NOTE *Instead of `DROP` and `REJECT`, fwsnort uses `DRP` and `REJ` because there is a 29-character limit imposed by the iptables `LOG` match for logging prefixes. You'll find additional information about what is going on behind the scenes with the `--ipt-drop` and `--ipt-reject` options in Chapter 11.*

Setting Up Whitelists and Blacklists

Any software that can block network communications based on application layer data should also be able to exclude certain networks or IP addresses from any blocking actions based on a *whitelist*. At the same time, it should be able to force all packets to or from certain networks or IP addresses to be dropped according to a *blacklist*.

Whitelists and blacklists are supported by fwsnort with the `WHITELIST` and `BLACKLIST` variables in the /etc/fwsnort/fwsnort.conf file. For example, to ensure that fwsnort never takes action against communications that originate from or are destined for the webserver (IP address 192.168.10.3 in Figure 1-2), and to `DROP` all packets to or from the IP address 192.168.10.200,[5] include the following lines in fwsnort.conf:

```
WHITELIST       192.168.10.3;
BLACKLIST       192.168.10.200;
```

When you use fwsnort to build the fwsnort.sh script, two new sections are added:

```
############ Add IP/network WHITELIST rules ############
$IPTABLES -A FWSNORT_FORWARD -s 192.168.10.3 -j RETURN
$IPTABLES -A FWSNORT_FORWARD -d 192.168.10.3 -j RETURN
$IPTABLES -A FWSNORT_INPUT -s 192.168.10.3 -j RETURN
$IPTABLES -A FWSNORT_OUTPUT -d 192.168.10.3 -j RETURN

############ Add IP/network BLACKLIST rules ############
$IPTABLES -A FWSNORT_FORWARD -s 192.168.10.200 -j DROP
$IPTABLES -A FWSNORT_FORWARD -d 192.168.10.200 -j DROP
$IPTABLES -A FWSNORT_INPUT -s 192.168.10.200 -j DROP
$IPTABLES -A FWSNORT_OUTPUT -d 192.168.10.200 -j DROP
```

The use of the `RETURN` target from each of the fwsnort chains in the whitelist short-circuits the signature comparison process as early as possible in order to minimize CPU resources that are devoted to heavyweight packet inspection; these rules are added to the fwsnort chains before the signature rules are added. Similarly, the `DROP` target for the blacklist rules drops matching packets on the floor before any additional processing is performed.

A summary of packet flow through the built-in `FORWARD` chain and fwsnort chains appears in Figure 10-1.

[5] This IP address is on the internal network, but sometimes certain systems function as dedicated resources for internal networks and should never communicate with networks outside the firewall. In this case, blacklist rules can enforce zero communications with external networks. Another scenario where blacklist rules would make sense is if the internal system has been compromised and its communications must therefore be severely curtailed until it can be cleaned.

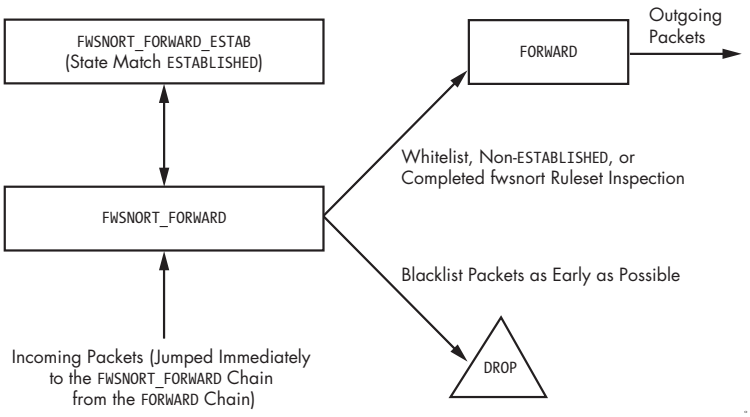

Figure 10-1: The path through the FORWARD chain and the fwsnort chains

Concluding Thoughts

The Snort community has lit the path toward an effective language for detecting network attacks, and so it is logical for fwsnort to use the Snort signature set as its source of attack descriptions. But, iptables is a firewall, and firewalls are all about *control*. Consider the scenario where a vulnerability is found within a piece of mission-critical server software that you are running on a Linux system. Until an outage window can be scheduled for this server to be patched, the system is vulnerable to attack. By leveraging the power of the Snort community, once a signature is developed and released, fwsnort can tell your Linux kernel how to discard packets that appear to exploit the vulnerability before they can do any real harm.

Although fwsnort can build iptables rulesets that discard packets, such a response does not dynamically implement persistent blocking rules against malicious IP addresses—a userland process is needed for this. We'll see in Chapter 11 that fwsnort combined with psad can build time-out–based blocking rules for application layer attacks.