

## Chapter 5

# Network Inline Data Modification

### Solutions in this Chapter:

- Introduction
- Snort\_inline
- Netfilter Data Replacement Patch
- Attack Mitigation and Nullification

**Related Chapters: Four Layers of Intrusion Prevention**

- ☑ Summary
- ☑ Solutions Fast Track
- ☑ Frequently Asked Questions

## Introduction

This chapter explores the concept and implementation of inline Application-layer data modification, and provides several motivating examples for why this technique provides an effective method to augment the security arsenal available to any security administrator. Many Intrusion Detection Systems (IDS') and Intrusion Prevention Services (IPS') offer the capability of taking *some* action against an Internet Protocol (IP) address from which an attack has been detected. Even though IDS' are generally passive in terms of the network traffic they monitor, many offer active response capabilities such as the ability to spoof Transmission Control Protocol (TCP) reset packets and interact with firewall software to implement Network-layer blocking rules against offending IP addresses. This chapter discusses the notion of active response implemented at the highest layer in the protocol stack: the Application layer. This technique involves the direct alteration of the application portion of IP packets that are associated with an attack as they traverse a network, in an effort to nullify the attack. Performing this operation requires direct access to packet data structures as they flow across a network, and hence can only realistically be performed by an inline device such as a firewall, router, or specialized Ethernet bridge.

As discussed in Chapter 4, active response and/or intrusion prevention actions can be implemented at any layer of the stack above Layer 1 (Physical Media). However, Application-layer data modification provides the most stealthy active response method. To illustrate this, let's examine the implementation of active response techniques at the Data Link, Network, and Transport layers.

For the Data Link layer, if a response is made to administratively take down the switch interface to which the attacker's machine is connected, the attacker is immediately aware of the action, since all connectivity (even within the immediate Local Area Network [LAN]) is severed. For the Network layer, if a firewall or router access control list (ACL) is modified to disallow the source IP address used by the attacker, most likely the attacker is still free to connect to other systems on the Internet but will find the loss of all connectivity to the specific target system and/or network difficult to miss. Such a disruption in

connectivity would provide the attacker with ample evidence that the target has employed the services of an intrusion prevention or active response system. Lastly, if Transport layer controls are implemented with Internet Control Message Protocol (ICMP) “port unreachable” messages or TCP resets being generated in response to attacks that leverage the User Datagram Protocol (UDP) or TCP protocols respectively, the attacker would notice as connectivity to the target temporarily appears to fade on a request or session basis. Connectivity would subsequently be restored, but nonetheless, intermittent Transport-layer connectivity in response to specific attacks raises a red flag and makes it clear that an IPS is deployed on the target network.

Transport layer active response, which takes place exclusively for individual UDP requests or TCP sessions, requires an automated mechanism for generating the appropriate response traffic, whereas firewall or router ACLs can be modified manually after the attacks have triggered sufficient alarms from an IDS. Note that many firewalls possess the ability to include rules that generate ICMP port or unreachable messages, or TCP reset packets for specific IP addresses and port numbers. However, such rules do not instruct the firewall to generate such session-busting traffic based on characteristics of Application-layer data. Iptables running FWSnort (see [www.cipherdyne.org/fwsnort](http://www.cipherdyne.org/fwsnort)) is a notable exception, because sessions can be torn down that contain Application-layer strings that match content fields in Snort rules. Commercial offerings such as Firewall-1 from Check Point also have the ability to disallow sessions or traffic that contain particular Application-layer signatures.

Hence, if a target system selectively tears individual sessions down, it provides excellent evidence that not only is an IPS or active response system lying in wait on the target network, but that this system is completely automated and requires no human intervention. Once the attacker discovers that an IPS is providing additional protection to a target system, the IPS itself may become the new target, since it may be possible to fool the IPS into blocking or otherwise interfering with network traffic from critical systems such a Domain Name System (DNS) server or an upstream router.

If implemented improperly, any technology that alters Application-layer data can exact a heavy toll on the stability and integrity of a network and the

systems contained therein. The Application layer is the entire reason why the Network layer exists; therefore, changing Application-layer data en route as it enters or exits a network must be done with the up-most care. For this discussion, assume that the alteration of Application-layer data is performed by an inline device such as a firewall or Ethernet bridge that is in the direct path of the packets as they traverse from one hop to the next. Note that packets can also be altered at the endpoint host independently of the application, by intercepting packets from within the kernel before they are placed on the wire. Normally, the kernel takes the buffers handed it by the application and wraps the appropriate Transport, Network, and Data Link layer headers around the data as it is sent down through the transport and network stacks, out through the network device driver, and onto the physical media. The data generated by the application is left relatively unchanged.

This chapter discusses two pieces of software—Snort\_inline and a kernel patch to the Netfilter string match extension—both of which are capable of functioning within an inline device and altering Application-layer data in packets as they flow through the device. The modifications made to Application-layer data are characterized by a set of rules that describe exactly which portion of the data should be altered and how. Lastly, this chapter shows both Snort\_inline and the data replacement patch in action and illustrates how each can be utilized to thwart example attacks sent across a network. In each of these examples, the total length of each packet remains intact, and only specific bytes within the packet payload are changed.

## Application Layer Data Modification and Protocol Breakage

Any piece of software that possesses the ability to alter Application-layer data in packets that traverse a network must take special care not to break packet header information in the process of altering the data. RFC 793, *The Transmission Control Protocol*, requires a 16-bit checksum to be calculated by each endpoint node. This checksum is calculated as the 16-bit one's complement of the one's complement sum of all 16-bit words in the packet headers

and packet data. Hence, any device that tries to alter Application-layer data must recalculate the checksum by applying the same algorithm to the packet headers and altered data bytes and replace the original checksum with the new. According to RFC 793, the calculation of checksums is a mandatory operation for the TCP protocol, because TCP provides reliable transport of data across an unreliable network. According to RFC 768, *The User Datagram Protocol*, checksum calculation is optional for UDP and is only calculated depending on whether the checksum had been previously calculated by an endpoint node.

Re-calculation of the Transport-layer checksum as an inline device alters Application-layer data and maintains the validity of packets in terms of the requirements imposed on communication utilizing TCP/IP. However, this does nothing to maintain any requirements imposed by the application that is actually initiating the communication. For example, suppose the Hypertext Transfer Protocol (HTTP)/1.1 portion of a client Web request is replaced with “0000/1.1,” or the transaction ID associated with a DNS server response to a client is replaced with garbage data, or the active port number returned by a File Transfer Protocol (FTP) server is altered by subtracting 1 from its value? All of these actions are possible with an inline device that can modify Application-layer data. As long as the Transport-layer checksum is recalculated, such modifications will be passed on to each respective application with potentially damaging results. The packet trace in Figure 5.1 was generated by connecting to *www.google.com* through an inline device that re-wrote the outbound HTTP/1.1 portion of the Web request to 0000/1.1. The result was that the Google Web server immediately sent a TCP reset to tear down the session since the Application-layer packet data did not contain a valid HTTP request.

**Figure 5.1** Broken HTTP Request

```
# tcpdump -i eth0 -s 0 -l -nn -X port 80
20:32:00.739130 IP 68.x.x.x.13829 > 216.239.39.104.80: S
328538606:328538606(0) win 5840 <mss 1460,sackOK,timestamp 93322223
0,nop,wscale 0>
```

**138 Chapter 5 • Network Inline Data Modification**

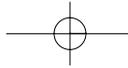
```

20:32:00.755460 IP 216.239.39.104.80 > 68.x.x.x.13829: S
2192939124:2192939124(0) ack 328538607 win 8190 <mss 1460>
20:32:00.758188 IP 68.x.x.x.13829 > 216.239.39.104.80: . ack 1 win 5840
20:32:00.760502 IP 68.x.x.x.13829 > 216.239.39.104.80: P 1:802(801) ack 1
win 5840
    0x0000:  4500 0349 e203 4000 3e06 c0a6 0202 0202  E..I..@.>.....
    0x0010:  d8ef 2768 3605 0050 1395 19ef 82b5 9875  ..'h6..P.....u
    0x0020:  5018 16d0 0450 0000 4745 5420 2f73 6561  P...P..GET./sea
    0x0030:  7263 683f 713d 7073 6164 2b49 4453 2662  rch?q=psad+IDS&b
    0x0040:  746e 473d 5365 6172 6368 2668 6c3d 656e  tnG=Search&hl=en
    0x0050:  266c 723d 2663 3263 6f66 663d 3126 636c  &lr=&c2coff=1&c1
    0x0060:  6965 6e74 3d66 6972 6566 6f78 2672 6c73  ient=firefox&rls
    0x0070:  3d6f 7267 2e6d 6f7a 696c 6c61 2533 4165  =org.mozilla%3Ae
    0x0080:  6e2d 5553 2533 4175 6e6f 6666 6963 6961  n-US%3Aunofficia
    0x0090:  6c20 3030 3030 2f31 2e31 0d0a 486f 7374  l.0000/1.1..Host
    0x00a0:  3a20 7777 772e 676f 6f67 6c65 2e63 6f6d  :.www.google.com
    0x00b0:  0d0a 5573 6572 2d41 6765 6e74 3a20 4d6f  ..User-Agent:.Mo
    0x00c0:  7a69 6c6c 612f 352e 3020 2858 3131 3b20  zilla/5.0.(X11;.
    0x00d0:  553b 204c 696e 7578 2069 3638 363b 2072  U;.Linux.i686;.r
    0x00e0:  763a 312e 372e 3329 2047 6563 6b6f 2f32  v:1.7.3).Gecko/2
    0x00f0:  3030 3431 3032 3520 4669 7265 666f 782f  0041025.Firefox/
    0x0100:  302e 3130 2e31 0d0a 4163 6365 7074 3a20  0.10.1..Accept:.
20:32:00.783623 IP 216.239.39.104.80 > 68.x.x.x.13829: R
2192939125:2192939125(0) win 9814

```

## Snort\_inline

The well-known IPS from the open source community has been implemented as a patch to the venerable Snort IDS. This patch gives strong IPS capabilities to Snort, including the ability to run on a Linux system that is configured as a bridge between two Ethernet segments. Snort\_inline was originally developed by Jed Haile, and is currently maintained by William Metcalf, Rob McMillen, and Victor Julien. The Snort IDS developers have taken notice of the Snort\_inline patch, and have decided to include the patch



in the main Snort codebase. As of this writing, the Snort\_inline patch has been integrated with Snort by Dan Roelker, and is available for download with Snort-2.3.0RC1 (Release Candidate 1) from [www.snort.org](http://www.snort.org). We will reference this release for the remainder of this discussion.

Prior to the 2.3.0RC1 release of the Snort IDS, the method chosen by Snort to extract packets off the wire was to utilize functions provided by the packet capture library *libpcap* ([www.tcpdump.org](http://www.tcpdump.org)). Snort\_inline chose an entirely different method of interacting with packets by allowing them to make use of *libipq* ([www.netfilter.org](http://www.netfilter.org)), which was developed by the Netfilter Project. Libipq is a library that allows packets to be queued from a Netfilter firewall running on a Linux system from kernel memory to a userspace application. The application then makes a decision about what the Netfilter should do with the packet (i.e., drop it or allow it to pass) and hand the packet back to the kernel along with the verdict. Netfilter then processes the packet accordingly. One of the most interesting features of libipq is that the userspace application may modify the packet before handing it back to the kernel for further processing. What libipq allows a Linux system to achieve is that if normal IP forwarding is turned off, packets can be sent through a userspace application where more complex analysis can be performed, and packets can be routed, blocked, or altered accordingly.

Some of the most important additions that Snort\_inline has made to the Snort IDS include three new rule actions—*drop*, *reject*, and *sdrop*—which allow (respectively) Snort to drop packets and log them via the usual Snort logging mechanism, generate appropriate protocol responses (such as TCP resets or ICMP port unreachable messages) to tear down sessions, and silently drop packets with no logging entries generated. Snort\_inline also has the ability to Snort replacing Application-layer data with new bytes specified within a Snort rule (via the *replace* keyword). The alteration of Application-layer data is accomplished with the new Snort rule keyword *replace*.

## Installation

The installation of Snort with the ability to run an inline mode involves several steps including the recompilation of the Linux kernel to enable bridging

**140 Chapter 5 • Network Inline Data Modification**

support. Although a complete discussion of the individual steps necessary to change compilation options, recompile, and install the Linux kernel is beyond the scope of this book, we provide an outline of the required steps. More information about the Linux kernel compilation and installation process can be found in any book on Linux system administration.

The most effective way to deploy Snort running in inline mode is to configure a dedicated Linux system to function as an Ethernet bridge between two Ethernet segments, and pass all traffic through this bridge before it can reach any systems on an internal network (see Figure 5.3). By default, the Linux kernel does not have Ethernet bridging support enabled, so we need to enable it and recompile. Also, since we require the ability to queue packets via `libipq` to Snort, we need to apply the `ebtables` patch to the Linux kernel (if installing on a 2.4 series kernel), because the kernel cannot function as an Ethernet bridge *and* filter traffic that traverses interfaces associated with a bridge. We need to install `bridge-utils`, a pre-1.1 version of `libnet` (versions 1.1 and later are not supported by Snort-2.3.0RC1), and `libipq`, which is part of the `iptables` tarball distributed by the Netfilter Project. Finally, we need to download and compile Snort-2.3.0RC1 with inline mode enabled and install it on our Linux system. All of these steps, including downloading the 2.4.27 kernel, installing the `ebtables` patch, installing `libipq`, and installing Snort-2.3.0RC1, are outlined in Figure 5.2 (some of the command output has been removed for brevity). It should be noted that there is a similar series of steps for running Snort in inline mode on a 2.6.x series kernel.

**Figure 5.2 Installation of Snort-2.3.0RC1 in Inline Mode**

```
$ cd /usr/src
$ wget http://www.kernel.org/pub/linux/kernel/v2.4/linux-2.4.27.tar.bz2
$ tar xvj linux-2.4.27.tar.bz2
$ cd linux-2.4.27

$ wget http://aleron.dl.sourceforge.net/sourceforge/ebtables/ebtables-brnf-7_vs_2.4.27.diff.gz
$ gunzip ebtables-brnf-7_vs_2.4.27.diff.gz
$ patch -p1 < ebtables-brnf-7_vs_2.4.27.diff
```



```
$ make menuconfig
< Go to Networking options, enable Network Packet Filtering, enable 802.1d
Ethernet Bridging and ebttables Bridging, save and exit >
# make dep && make clean && make bzImage && make modules && make
modules_install
< Install the kernel within /boot and edit boot loader config file
appropriately >
$ cd /usr/local/src

$ wget http://www.packetfactory.net/libnet/dist/deprecated/libnet-
1.0.2a.tar.gz
$ tar xzf libnet-1.0.2a.tar.gz
$ cd Libnet-1.0.2a
$ ./configure --prefix=/usr && make
# make install
$ cd ..

$ wget http://voxel.dl.sourceforge.net/sourceforge/bridge/bridge-utils-
1.0.4.tar.gz
$ tar xzf bridge-utils-1.0.4.tar.gz
$ cd bridge-utils-1.0.4
$ ./configure --prefix=/usr && make
# make install
$ cd ..

$ wget http://www.netfilter.org/files/iptables-1.2.11.tar.bz2
$ tar xjf iptables-1.2.11.tar.bz2
$ cd iptables-1.2.11
$ make KERNEL_DIR=/usr/src/linux-2.4.27 BINDIR=/sbin LIBDIR=/lib
# make install KERNEL_DIR=/usr/src/linux-2.4.27 BINDIR=/sbin LIBDIR=/lib
# make install-devel
$ cd ..

$ wget http://www.snort.org/dl/snort-2.3.0RC1.tar.gz
$ tar xzf snort-2.3.0RC1
```

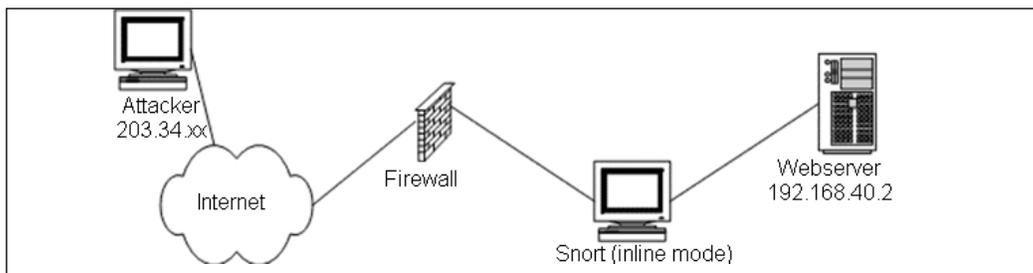
**142 Chapter 5 • Network Inline Data Modification**

```
$ cd snort-2.3.0RC1
$ ./configure --prefix=/usr --enable-inline && make
# make install
```

## Operation

Now that Snort-2.3.0RC1 is installed, it is time to configure and test it. The examples in this chapter surround the concept of Application-layer data replacement, so we are most interested in verifying that Snort can provide the means to accomplish this. To this end, we set up a Linux bridge on a dedicated system in the network configuration show in Figure 5.3. All connectivity between the external and internal networks must qualify against both the firewall policy deployed on the firewall host *and* the Application layer inspection provided by Snort running in inline mode. For the remainder of the discussion, we assume that all physical cabling has been finished for the network diagram in Figure 5.3, so that we may concentrate on the operating system (OS) and application administration requirements.

**Figure 5.3** Network Diagram for Inline-mode Snort



First, we must use the *brctl* program provided by *bridge-utils* to construct a bridging interface on the Snort box. This interface looks like a normal Ethernet or loopback interface under the output of *ifconfig*, and can be controlled in the same way. An Ethernet bridge is strictly a Data Link-layer device that connects two physical Ethernet segments; there is no notion of an IP address assigned to any interface that connects the two segments. In this regard, an Ethernet bridge is similar to an Ethernet switch. To assist in the

configuration of Snort and allow us to achieve reproducible results, we use the script in Figure 5.4. This script was adapted from *rc.firewall* script written by Rob Millen (see [www.honeynet.org/tools/dcontrol/rc.firewall](http://www.honeynet.org/tools/dcontrol/rc.firewall)) and configures the *eth1* and *eth2* interfaces attached to the Linux system to form an Ethernet bridge called *br0*. Any existing iptables rules are flushed from kernel memory (*iptables -F*), all packets processed by the *FORWARD* chain (see Figure 5.8) are queued via *libipq* to userspace, and all normal forwarding is turned off. The only packets that will be allowed to traverse the *br0* bridge are those that Snort does not instruct Netfilter to drop after running the packet data structures through its detection engine.

**Figure 5.4 Linux Bridge Script**

```
#!/bin/sh

BRIDGE=/usr/sbin/brctl
IFCONFIG=/sbin/ifconfig
INET_IFACE=eth1
LAN_IFACE=eth2
ECHO=/bin/echo
IPTABLES=/sbin/iptables

$BRIDGE delif br0 eth1
$BRIDGE delif br0 eth2
$IFCONFIG br0 down
$BRIDGE delbr br0

#####
# Make sure our interfaces don't have ip information
#
$IFCONFIG $INET_IFACE 0.0.0.0 up -arp
$IFCONFIG $LAN_IFACE 0.0.0.0 up -arp

#####
# Start the bridge
```

**144 Chapter 5 • Network Inline Data Modification**

```
#
$BRIDGE addbr br0
$BRIDGE addif br0 $LAN_IFACE
$BRIDGE addif br0 $INET_IFACE

#####
# Make sure our bridge is not sending out
# BPDUs (part of the spanning tree protocol).
#
$BRIDGE stp br0 off

#####
# Bring the bridge interface up
#
$IIFCONFIG br0 0.0.0.0 up -arp

#####
# Flush iptables rules, and then force all packets in the
# FORWARD chain to be queued to userspace.
#
$IPTABLES -F
$IPTABLES -A FORWARD -j QUEUE

#####
# Turn normal forwarding off!!! The only packets that will
# be forwarded are those that Snort says are ok.
#
$ECHO 0 > /proc/sys/net/ipv4/ip_forward

### EOF ###
```

---

Let's illustrate the usage of the script in Figure 5.4, and show how Snort running in inline mode can alter Application-layer data with the *replace* key-

word. First, we create a basic Snort rule that uses the *replace* keyword to alter a Web request, and place this rule in rules file */etc/snort/rules/inline.rules*:

```
alert tcp any any -> any 80 (msg:"nofile.html -> index1.html";
content:"nofile.html"; replace:"index1.html"; classtype:attempted-recon;
sid:9000; rev:1;)
```

Next, we edit */etc/snort/snort.conf* to reference the *inline.rules* file. We are now ready to test the data-mangling capability of Snort. On the Web server in Figure 5.3, we create the basic Hypertext Markup Language (HTML) file */webroot/htdocs/index1.html* and start the Web server like so:

```
# cat > /webroot/htdocs/index1.html
<html>
<body>
this is a test
</body>
</html>
# /etc/init.d/httpd start
Starting httpd: [ OK ]
```

On the bridging Linux system, we activate the bridge and execute Snort with the *-Q* command line option, which instructs Snort to read packets from *libipq* instead of from *libpcap* (note the inclusion of the note about initializing in inline mode):

```
# /root/bin/bridge.sh
# snort -Q -c /etc/snort/snort.conf
+++++
Initializing rule chains...
Initializing Inline mode
----- Initialization Complete -----

,,_    -*> Snort! <*-
o"   )~  Version 2.3.0RC1 (Build 8)
    ''''  By Martin Roesch & The Snort Team:
http://www.snort.org/team.html
        (C) Copyright 1998-2004 Sourcefire Inc, et al.
```

## 146 Chapter 5 • Network Inline Data Modification

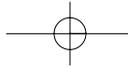
A legitimate Web request for the *nofile.html* file produces the “this is a test” string in the client browser. (Note that the *nofile.html* file does not exist on the Web server.) Finally, due to the fact that we added the *alert* rule action to our test rule in *inline.rules*, Snort produces the following log message in response to our *nofile.html* request:

```
[**] nofile.html -> index1.html [**]
11/23-21:23:11.750470 203.34.x.x:58737 -> 192.168.40.2:80
TCP TTL:64 TOS:0x0 ID:58313 IpLen:20 DgmLen:1280 DF
***AP*** Seq: 0xB0560178 Ack: 0xB1C15E60 Win: 0x16D0 TcpLen: 32
TCP Options (3) => NOP NOP TS: 51110577 47322
=====
```

It should be noted that the emphasis in this section is on the raw technology surrounding Snort\_inline instead of system administration and Best Practices. A serious deployment of Snort running in inline mode must include the proper boot time initialization script that is compatible with the style of initialization scripts on your particular Linux distribution. Also, the importance of tuning inline-enabled Snort cannot be emphasized enough, since false positives become doubly important when Snort is given the ability to alter network traffic (see Chapter 3).

## Netfilter Data Replacement Patch

The Netfilter firewall in the Linux kernel is an extremely powerful tool for enhancing the security stance of any network. It is stateful, has a complete Network Address Translation (NAT) implementation, generates rich *syslog* messages, and can search for strings in Application-layer data using the Netfilter string match extension. The official project name, Netfilter, refers to the set of kernel-level hooks implemented within the Linux kernel, which perform the heavy lifting in terms of enforcing a security policy. The userspace program that provides an environment to interact with Netfilter is called iptables and is the program that is most familiar to Linux administrators. Since Linux (and Netfilter with it) are open source, anyone can provide a patch for the original code in an effort to fix bugs or provide new functionality. We take advantage of this fact by illustrating a patch to the Netfilter



string match module that can be used to alter Application-layer data in a manner similar to the *replace* keyword in Snort.

The data replacement functionality provided by the patch to the string match extension is implemented somewhat differently than the *replace* keyword in Snort. Instead of requiring that the length of the replacement content be exactly equal to the length of the matching data in the Application layer of a packet, the string extension patch allows the length of the replacement data to be less than or equal to the length of the data to be overwritten. If a replacement string is shorter than the string used to detect an attack, as many bytes as possible are overwritten starting at the beginning of the matching string, but remaining bytes are left untouched in the packet data. For example, suppose Netfilter has been instructed to use this technique to replace the string “*long string*” (spaces are allowed) with the string “*short;*” the result of a successful match will be that a packet will contain the new string “*shortstring.*”

## Installation

Netfilter extensions are maintained as a set of patches to the Linux kernel and also to the iptables userland program. These patches can easily be installed by using a specialized installation package called *patch-o-matic*, which is available at [www.netfilter.org](http://www.netfilter.org). We use the “next generation” version of this program, *patch-o-match-ng*, to install the string match extension. After testing that we can use the functionality provided by the string extension to instruct Netfilter to match on Application-layer data. We will apply the data replacement patch to the string extension and show how Netfilter can then alter packet data as it flows to and/or through a Linux system.

The data replacement patch contains two main parts: a series of modifications to the userland iptables program and to the string match kernel module. The Netfilter project maintains a collection of modules that extend the functionality of the Netfilter firewall. These modules are maintained within one of three categories: *base*, *pending*, and *extra*. The *base* category is the core set of extensions that are considered stable and provide important functionality. The *pending* category includes extensions that are generally stable but are waiting to be included within the *base* category pending additional testing, inclusion

## 148 Chapter 5 • Network Inline Data Modification

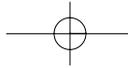
within the Linux kernel, or wider acceptance by the Netfilter community. (Note that most *base* extensions are not included within the stock Linux kernel and must be applied as a patch.) Finally, the *extra* category is reserved for modules that should not necessarily be included within the *base* category and/or are relatively unstable. The Netfilter string match extension falls into the *extra* category (see Figure 5.5). Note that these steps assume that the Linux kernel sources have been installed on the local system at `/usr/src/linux`, the latest iptables code (1.2.11 as of this writing) has been installed at `/usr/local/src/iptables-1.2.11`, and each command must be executed as root. As of this writing, the string match extension has not been ported to the 2.6 kernel, so we assume that the latest version of the 2.4 kernel sources (2.4.27) has been installed.


**Figure 5.5** Netfilter String Match Extension Installation

```
$ cd /usr/local/src
$ wget http://www.netfilter.org/files/patch-o-matic-ng-20040621.tar.bz2
$ tar xvj patch-o-matic-ng-20040621.tar.bz2
$ cd patch-o-matic-ng-20040621
# KERNEL_DIR=/usr/src/linux IPTABLES_DIR=/usr/local/src/iptables-1.2.11
./runme string
Welcome to Patch-o-matic (1.17)!

Kernel: 2.4.27, /usr/src/linux-2.4.27
Iptables: 1.2.11, /usr/local/src/iptables-1.2.11
Each patch is a new feature: many have minimal impact, some do not.
Almost every one has bugs, so don't apply what you don't need!
-----
Already applied:

Testing string... not applied
The 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/?] y

Excellent! Source trees are ready for compilation.

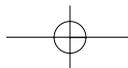
At this point, we have the string match extension installed in source form on the system. We have yet to patch and compile it for actual use on the system (see Figure 5.6). The string replacement patch is in two pieces: one patches the userland iptables library *libipt\_string.c*, which is responsible for interfacing with the string match kernel code, and the other patches the kernel portion of the string match extension.



**Figure 5.6** Applying and Installing the String Replacement Patch

```
$ cd /usr/local/src/iptables-1.2.11
$ wget http://www.cipherdyne.org/fwsnort/string_replace_iptables.patch
$ patch -p0 < string_replace_iptables.patch
patching file extensions/libipt_string.c
$ make KERNEL_DIR=/usr/src/linux
# make install KERNEL_DIR=/usr/src/linux
# cd /usr/src/linux

$ wget http://www.cipherdyne.org/fwsnort/string_replace_kernel.patch
$ patch -p1 < string_replace_kernel.patch
patching file include/linux/netfilter_ipv4/ipt_string.h
patching file net/ipv4/netfilter/ipt_string.c
$ make menuconfig
<turn on string matching under Netfilter configuration in Networking Options>
```



**150 Chapter 5 • Network Inline Data Modification**

```
# make dep && make clean && make bzImage && make modules && make  
modules_install
```

The last two steps in Figure 5.6 complete the process of configuring the kernel to compile in the string match extension by setting `CONFIG_IP_NF_MATCH_STRING=y` in the kernel configuration file (`/usr/src/linux/.config`) and then compiling the kernel. A complete treatment of the steps required to compile and install the Linux kernel are beyond the scope of this book; any book on Linux system administration will provide more information. After the above steps are completed, the resulting kernel binary (`/usr/src/linux/arch/i386/boot/bzImage`) is copied into place in the `/boot` partition, and the boot loader configuration file is altered to include the ability to boot into the new kernel.

### Notes from the Underground...

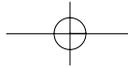
#### Data Replacement Patch Code Motivation

The original Netfilter string match extension, written by Emmanuel Roger, uses the *Boyer-Moore Fast String Searching Algorithm* (see [www.cs.utexas.edu/users/moore/best-ideas/string-searching/](http://www.cs.utexas.edu/users/moore/best-ideas/string-searching/)) in an effort to allow Netfilter to maintain the highest possible throughput even when searching for strings in Application-layer data. The function prototype for the entry point into the search algorithm is as follows:

```
char * search(char *needle, char *haystack, int nlen, int hlen)
```

Note that the above prototype returns a pointer to a character. Given a search string, this pointer provides the address in kernel memory where the matching string is found in packet data, or NULL, if no match is found. This provided the motivation for writing the data replacement patch. From there it was relatively easy to modify the string match extension to also allow the matching strings to be altered at the request of the system administrator. The meat of the code modification that allows packet data to be mod-

Continued



ified after a successful match is found appears below. This code is placed in the file. Note the mandatory recalculation of the TCP checksum and the optional recalculation of the UDP checksum after the data modification is performed. The UDP checksum is recalculated only if the original packet contains a non-zero checksum indicating that the originated host calculated it first.



```

if (repl_ptr != NULL && rlen > 0) {
    /* if we change the data portion of the packet we
       recalculate the transport layer checksum (mandatory
       for TCP). */
    if (skb->nh.iph->protocol == IPPROTO_TCP) {
        /* repl_ptr points to the start of the needle
           * in the packet, and we know the entire needle
           * is there so we can just replace. */
        for (rctr=0; rctr < rlen; rctr++)
            repl_ptr[rctr] = repl_str[rctr];

        tcph = (struct tcphdr *)((u_int32_t*)skb->nh.iph +
                                skb->nh.iph->ihl);
        unsigned int tcplen = skb->len -
                               (skb->nh.iph->ihl<<2);
        tcph->check = 0;
        tcph->check = tcp_v4_check(tcph, tcplen,
                                   skb->nh.iph->saddr,
                                   skb->nh.iph->daddr,
                                   csum_partial((char *)tcph, tcplen, 0));
    } else if (skb->nh.iph->protocol == IPPROTO_UDP) {
        /* repl_ptr points to the start of the needle
           * in the packet, and we know the entire needle
           * is there so we can just replace. */
        for (rctr=0; rctr < rlen; rctr++)
            repl_ptr[rctr] = repl_str[rctr];
        /* recalculate UDP checksum only if it was
           previously calculated */
    }
}

```

Continued

[www.syngress.com](http://www.syngress.com)

```
    udph = (struct udphdr *)((char *)skb->nh.iph +
                            (skb->nh.iph->ihl<<2));
    unsigned int udplen = skb->len -
                            (skb->nh.iph->ihl<<2);
    if (udph->check) {
        udph->check = 0;
        udph->check =
            csum_tcpudp_magic(skb->nh.iph->saddr,
                              skb->nh.iph->daddr,
                              udplen, IPPROTO_UDP,
                              csum_partial((char *)udph, udplen, 0));
    }
}
```

## Netfilter String Match Operation

The *iptables* userland program (located at */sbin/iptables* or */usr/sbin/iptables*) is the main interface provided to the administrator for modifying and viewing the Netfilter policy that a running Linux kernel is executing. If the *iptables* program is invoked to add a rule into or delete a rule from a running Netfilter policy, its operation is restricted to the modification of an individual rule; multiple rules cannot be added or deleted with a single execution of the *iptables* binary from the command line. This implies that for a firewall that has tens (or even hundreds) of rules, restoring a Netfilter policy upon reboot can be a relatively expensive operation. The kernel data structures associated with any running Netfilter policy are fundamentally communicated from userland; therefore, this same communication must take place at system boot time. Netfilter cannot maintain a policy statically within kernel space across a reboot. Fortunately, the *iptables-save* command can save a running Netfilter policy into a file in a format that can be read in a single invocation of the *iptables-restore* command. This allows an entire Netfilter policy to be saved or restored without having to resort to multiple executions of the *iptables* program. Rules that are constructed with the string match extension (and with

the data replacement patch) are compatible with the iptables-save and iptables-restore commands.

Next, we illustrate how to use iptables to interface with the string match extension. Iptables maintains a notion of a *table*, *chain*, and *target* on a per-rule basis. These three programmatic constructs taken together inform the kernel what action should be taken in response to a packet that matches the specific criteria spelled out within the rule. For this discussion, we apply string-matching rules to the INPUT, OUTPUT, and FORWARD chains, and make use of the DROP, REJECT, LOG, and ACCEPT targets. It should be noted, however, that rules built with the string match extension are compatible with additional iptables chains and targets not mentioned in this discussion. A complete treatment of the vagaries of iptables options, configuration, and operation is beyond the scope of this book; more information can be found in the iptables *man page* on any Linux system. There is also a good deal of documentation available at [www.netfilter.org](http://www.netfilter.org).

Iptables chains are as follows:

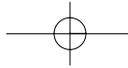
- **INPUT** Processes packets that are destined for the local system on which Netfilter is deployed.
- **OUTPUT** Processes packets that are generated by the local system on which Netfilter is deployed.
- **FORWARD** Processes packets that are traveling through the system on which Netfilter is deployed. This requires multiple interfaces to be installed on the system since otherwise forwarding traffic has no meaning.
- **PREROUTING** Gives Netfilter the opportunity to process packets within the kernel before a routing decision is made (i.e., before the kernel knows whether packets are destined for the local system or are to be forwarded to a different system). This is the earliest chance Netfilter has for processing packets as they arrive off the wire and are placed within kernel memory.
- **POSTROUTING** Gives Netfilter the opportunity to process packets within the kernel after the routing decision has been made.

## 154 Chapter 5 • Network Inline Data Modification

This is the last chance that Netfilter has to process packets before they are forwarded to other destination systems.

Iptables targets are as follows:

- **DROP** Drops matching packets on the floor without allowing any response traffic to be generated. For example, the rule `iptables -A INPUT -p tcp -dport 22 -s 192.168.10.1 -j DROP` will drop all TCP packets destined for the local system that have a destination port of 22 (Secure Shell [SSH]) from IP address 192.168.10.1.
- **LOG** Logs matching packets via syslog to `/var/log/messages` by default, but the destination file for such syslog messages can be changing the file associated with `kern.info` messages in the syslog configuration file. (The specifics of this step depend on which syslog daemon is in use; the `syslog-ng` configuration file is much different from the traditional `sysklogd` configuration file.) For example, the rule `iptables -A INPUT -p tcp -dport 22 -s 192.168.10.1 -syn -j LOG` will log all connection setup requests from arbitrary IP addresses to TCP port 22 on the local system.
- **ACCEPT** Accepts matching packets. The main use for this target is to accept traffic that should be allowed to or through the firewall. For example, the rule `iptables -A INPUT -p tcp -dport 443 -syn -j ACCEPT` allows arbitrary IP addresses to initiate a TCP session with the local system over port 443 (Hypertext Transfer Protocol Secure sockets [HTTPS]).
- **REJECT** Rejects matching packets. This target can be combined with the `-reject-with` command line argument to allow Netfilter to generate TCP reset packets for TCP traffic, or ICMP port-unreachable packets for UDP traffic. For example, the rule `iptables -A INPUT -p tcp -dport 22 -j REJECT --reject-with tcp-reset` instructs Netfilter to generate a TCP reset packet in response to any TCP packet destined for the local system on port 22 (SSH).
- **RETURN** Allows the processing of rules contained within user-defined chains to be short-circuited. This target is usually used to minimize the number of rules that Netfilter must process packets

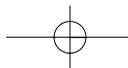
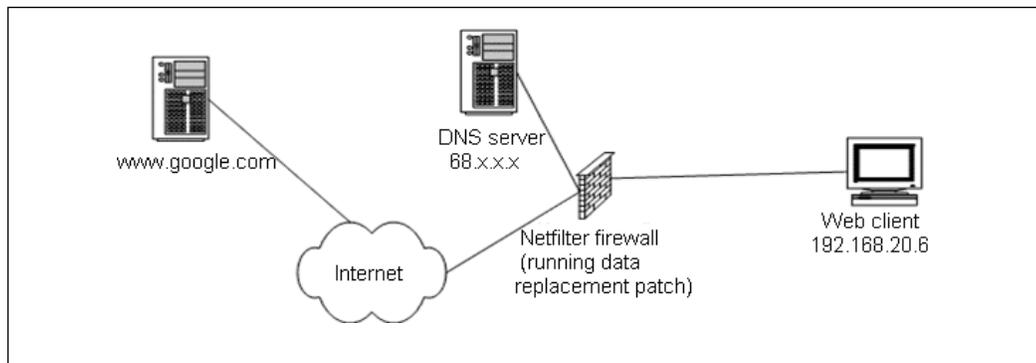


against, and can be an important tool for optimizing Netfilter policies that contain large numbers of rules in chains that are defined by the user.

- **QUEUE** Instructs Netfilter to pass matching packets to the userland application for further processing. `Snort_inline` makes use of this target to queue packets from kernel space instead of appealing to the traditional `libpcap` packet capture library used by the Snort IDS.

Now that you have a basic understanding of some of the important ways that Netfilter rules can be constructed, let's turn to the operation of the string match extension. We present a few examples that show how the string match extension can be used to add application-level processing to Netfilter rulesets. We also illustrate how the data replacement patch can be used to alter Application-layer data, and show the corresponding packet traces that prove that it works. These examples do not represent an exhaustive treatment of the usage of the string match extension; they merely serve as illustrations. The  *FWSnort* project ([www.cipherdyne.org/fwsnort](http://www.cipherdyne.org/fwsnort)) makes heavy use of the string match extension to generate logs for (and optionally block) packets based on Netfilter rules that are derived from Snort signatures. (See Chapter 8 for more information on FWSnort.)

**Figure 5.7** Linux Firewall Network Diagram



## ASCII String Matching and Data Replacement

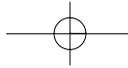
One of the most important Netfilter targets is the LOG target. The logging ability of Netfilter is extremely complete and allows almost every interesting field of the Network- and Transport-layer headers to be efficiently logged via syslog. Suppose we want Netfilter to inform us when someone has made a search request to *www.google.com* that includes the keyword *site*: to restrict the search to the *www.syngress.com* Web site. We would execute the following command on the Linux firewall where the string match extension is deployed (see Figure 5.7):

```
# iptables -I FORWARD 1 -p tcp --dport 80 -d www.google.com -m string --
string "site%3Awww.syngress.com" -j LOG --log-prefix "SYNGRESS SITE SEARCH "
```

We apply the above rule to the FORWARD chain because any packets coming from an internal client system such as *linuxclient* (see Figure 5.7) that are destined for a machine external to the local network (such as any external Google Web server) will be processed by this Netfilter chain in the Linux firewall before being allowed out. Now that the string match rule is in place, whenever a Web query is issued to Google that contains *site%3Awww.syngress.com* (note the URL encoding of the “:” character as “%3A” which is done by the Web browser), the following log message is sent by Netfilter to syslog:

```
Nov 13 11:07:50 orthanc kernel: SYNGRESS SITE SEARCH IN=eth1 OUT=eth0
SRC=192.168.10.2 DST=216.239.39.99 LEN=781 TOS=0x00 PREC=0x00 TTL=62
ID=19868 DF PROTO=TCP SPT=13509 DPT=80 WINDOW=32767 RES=0x00 ACK PSH URGP=0
```

The specific search string typed into the Web browser was *Snort site:www.syngress.com*. In the syslog message above the string, *SYNGRESS SITE SEARCH* is clearly displayed in **bold** characters. This string was generated by Netfilter due to the use of the *—log-prefix* option supplied to the command-line invocation of iptables. This option makes it easy for Netfilter to generate descriptive tags associated with syslog messages as specific rules are matched. Also, for completeness, a portion of a packet trace taken on the external interface of the firewall is displayed in Figure 5.8. Clearly displayed in **bold** characters is the search string *Snort+site%3Awww.syngress.com* supplied to



Google in the Web search request, and a packet containing an *HTTP/1.1 200 OK* response from the Google Web server indicating that the Web search request was well formed. The iptables rule and corresponding syslog message were generated without modifying any Application-layer data, and the packet trace in Figure 5.8 was taken on the egress interface.

**Figure 5.8** Packet Trace of Google “site:www.syngress.com” Search

```
# tcpdump -i eth0 -nn -s 0 -l -X port 80
13:16:37.333887 IP 68.x.x.x.13591 > 216.239.39.99.80: S
4009791563:4009791563(0) win 5840 <mss 1460,sackOK,timestamp 67194990
0,nop,wscale 0>
13:16:37.352403 IP 216.239.39.99.80 > 68.x.x.x.13591: S
3248864028:3248864028(0) ack 4009791564 win 8190 <mss 1460>
13:16:37.357594 IP 68.x.x.x.13591 > 216.239.39.99.80: . ack 1 win 5840
13:16:37.362079 IP 68.x.x.x.13591 > 216.239.39.99.80: P 1:742(741) ack 1
win 5840
    0x0000:  4500 030d c027 4000 3e06 e2c3 0202 0202  E....'@.>.....
    0x0010:  d8ef 2763 3517 0050 ef00 904c c1a5 bb1d  ..'c5..P...L....
    0x0020:  5018 16d0 e880 0000 4745 5420 2f73 6561  P.....GET./sea
    0x0030:  7263 683f 686c 3d65 6e26 6c72 3d26 6332  rch?hl=en&lr=&c2
    0x0040:  636f 6666 3d31 2671  3d53 6e6f 7274 2b73  coff=1&q=Snort+s
    0x0050:  6974 6525 3341 7777 772e 7379 6e67 7265  ite%3Awww.syngre
    0x0060:  7373 2e63 6f6d 2662 746e 473d 5365 6172  ss.com&btnG=Sear
    0x0070:  6368 2048 5454 502f 312e 310d 0a48 6f73  ch.HTTP/1.1..Hos
    0x0080:  743a 2077 7777 2e67 6f6f 676c 652e 636f  t:.www.google.co
    0x0090:  6d0d 0a55 7365 722d 4167 656e 743a 204d  m..User-Agent:.M
    0x00a0:  6f7a 696c 6c61 2f35 2e30 2028 5831 313b  ozilla/5.0.(X11;
    0x00b0:  2055 3b20 4c69 6e75 7820 6936 3836 3b20  .U;.Linux.i686;.
    0x00c0:  7276 3a31 2e37 2e33 2920 4765 636b 6f2f  rv:1.7.3).Gecko/
    0x00d0:  3230 3034 3130 3235 2046 6972 6566 6f78  20041025.Firefox
13:16:37.392899 IP 216.239.39.99.80 > 68.x.x.x.13591: P 1431:1677(246) ack
742 win 31460
13:16:37.394483 IP 216.239.39.99.80 > 68.x.x.x.13591: . 1:1431(1430) ack
742 win 31460
    0x0000:  4500 05be 5f11 0000 3206 8d29 d8ef 2763  E..._...2..)..'c
```

## 158 Chapter 5 • Network Inline Data Modification

```

0x0010: 0202 0202 0050 3517 c1a5 bb1d ef00 9331 .....P5.....1
0x0020: 5010 7ae4 3d5e 0000 4854 5450 2f31 2e31 P.z.=^..HTTP/1.1
0x0030: 2032 3030 204f 4b0d 0a43 6163 6865 2d43 .200.OK..Cache-C
0x0040: 6f6e 7472 6f6c 3a20 7072 6976 6174 650d ontrol:.private.
0x0050: 0a43 6f6e 7465 6e74 2d54 7970 653a 2074 .Content-Type:.t
0x0060: 6578 742f 6874 6d6c 0d0a 5365 7276 6572 ext/html..Server
0x0070: 3a20 4757 532f 322e 310d 0a54 7261 6e73 :.GWS/2.1..Trans
0x0080: 6665 722d 456e 636f 6469 6e67 3a20 6368 fer-Encoding:.ch
0x0090: 756e 6b65 640d 0a43 6f6e 7465 6e74 2d45 unked..Content-E
0x00a0: 6e63 6f64 696e 673a 2067 7a69 700d 0a44 ncoding:.gzip..D
0x00b0: 6174 653a 2053 6174 2c20 3133 204e 6f76 ate:.Sat,.13.Nov
0x00c0: 2032 3030 3420 3138 3a31 373a 3035 2047 .2004.18:17:05.G
0x0200: a89e d094 6ab0 296f aceb 2675 bd0a 4974 ....j.)o..&u..It
13:16:37.403997 IP 68.x.x.x.13591 > 216.239.39.99.80: . ack 1 win 5840
13:16:37.407477 IP 68.x.x.x.13591 > 216.239.39.99.80: . ack 1677 win 8580
13:16:37.718527 IP 216.239.39.99.80 > 68.x.x.x.13591: . 1677:3107(1430) ack
742 win 31460
13:16:37.718634 IP 216.239.39.99.80 > 68.x.x.x.13591: P 4537:4591(54) ack
742 win 31460
13:16:37.718703 IP 216.239.39.99.80 > 68.x.x.x.13591: P 4591:4596(5) ack
742 win 31460
13:16:37.719910 IP 216.239.39.99.80 > 68.x.x.x.13591: . 3107:4537(1430) ack
742 win 31460
13:16:37.725626 IP 68.x.x.x.13591 > 216.239.39.99.80: . ack 3107 win 11440
13:16:37.729286 IP 68.x.x.x.13591 > 216.239.39.99.80: . ack 3107 win 11440
13:16:37.732455 IP 68.x.x.x.13591 > 216.239.39.99.80: . ack 3107 win 11440
13:16:37.732712 IP 68.x.x.x.13591 > 216.239.39.99.80: . ack 4596 win 14300

```

Now we turn to an illustration of the data replacement patch in action. For this, we use the same Google Web search request as above, but modify the iptables *SYNGRESS SITE SEARCH* rule so that instead of simply generating a syslog message for a matching Web request, we will modify the outbound request inline to replace the search criteria *site%3Awww.syngress.com* with the string *site%3Awww.nonexist.com*. Note the addition of the command

[www.syngress.com](http://www.syngress.com)

line argument —*replace-string* in the iptables command below, and also note that the number of characters in the search and replace strings is identical (23 characters):

```
# iptables -I FORWARD 1 -p tcp --dport 80 -d www.google.com -m string --
string "site%3Awww.syngress.com" --replace-string "site%3Awww.nonexist.com"
-j LOG --log-prefix "SYNGRESS -> NONEXIST SEARCH "
```

As before, the specific Web search request we sent to Google (via the Firefox Web browser) was *Snort site:www.syngress.com*. However, this time the data replacement code in the string match extension modified the outgoing Web request so that the search string no longer contained *site%3Awww.syngress.com* and instead contained the string *site%3Awww.nonexist.com*. Netfilter generated a new syslog message, but this time the logging prefix reflects the match on the new —*replace-string* rule:

```
Nov 13 14:31:24 orthanc kernel: SYNGRESS -> NONEXIST SEARCH IN=eth1
OUT=eth0 SRC=192.168.10.2 DST=64.233.161.104 LEN=781 TOS=0x00 PREC=0x00
TTL=62 ID=7814 DF PROTO=TCP SPT=13602 DPT=80 WINDOW=5840 RES=0x00 ACK PSH
URGP=0
```

The packet trace in Figure 5.9 taken on the egress interface of the firewall shows the modified search request in **bold** after the TCP connection establishment.

**Figure 5.9** Packet Trace of Altered Google Web Search

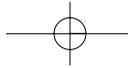
```
# tcpdump -i eth0 -nn -s 0 -l -X port 80
14:31:24.531454 IP 68.x.x.x.13602 > 64.233.161.104.80: S
4259894695:4259894695(0) win 5840 <mss 1460,sackOK,timestamp 71682851
0,nop,wscale 0>
14:31:24.548918 IP 64.233.161.104.80 > 68.x.x.x.13602: S
3597102845:3597102845(0) ack 4259894696 win 8190 <mss 1460>
14:31:24.551624 IP 68.x.x.x.13602 > 64.233.161.104.80: . ack 1 win 5840
14:31:24.553734 IP 68.x.x.x.13602 > 64.233.161.104.80: P 1:742(741) ack 1
win 5840
0x0000: 4500 030d 1e86 4000 3e06 a266 0202 0202 E.....@.>..f....
0x0010: 40e9 a168 3522 0050 fde8 d5a8 d667 6efe @..h5".P.....gn.
0x0020: 5018 16d0 08b7 0000 4745 5420 2f73 6561 P.....GET./sea
```

## 160 Chapter 5 • Network Inline Data Modification

```

0x0030: 7263 683f 686c 3d65 6e26 6c72 3d26 6332  rch?hl=en&lr=&c2
0x0040: 636f 6666 3d31 2671 3d53 6e6f 7274 2b73  coff=1&q=Snort+s
0x0050: 6974 6525 3341 7777 772e 6e6f 6e65 7869  ite%3Awww.nonexi
0x0060: 7374 2e63 6f6d 2662 746e 473d 5365 6172  st.com&btnG=Sear
0x0070: 6368 2048 5454 502f 312e 310d 0a48 6f73  ch.HTTP/1.1..Hos
0x0080: 743a 2077 7777 2e67 6f6f 676c 652e 636f  t:.www.google.co
0x0090: 6d0d 0a55 7365 722d 4167 656e 743a 204d  m..User-Agent:.M
0x00a0: 6f7a 696c 6c61 2f35 2e30 2028 5831 313b  ozilla/5.0.(X11;
0x00b0: 2055 3b20 4c69 6e75 7820 6936 3836 3b20  .U;.Linux.i686;.
0x00c0: 7276 3a31 2e37 2e33 2920 4765 636b 6f2f  rv:1.7.3).Gecko/
0x00d0: 3230 3034 3130 3235 2046 6972 6566 6f78  20041025.Firefox
0x00e0: 2f30 2e31 302e 310d 0a41 6363 6570 743a  /0.10.1..Accept:
14:31:24.586655 IP 64.233.161.104.80 > 68.x.x.x.13602: P 1431:1677(246) ack
742 win 9228
14:31:24.588033 IP 64.233.161.104.80 > 68.x.x.x.13602: . 1:1431(1430) ack
742 win 9228
0x0000: 4500 05be 6be7 0000 3306 9d54 40e9 a168  E...k...3..T@..h
0x0010: 0202 0202 0050 3522 d667 6efe fde8 d88d  ....P5".gn.....
0x0020: 5010 240c 6b70 0000 4854 5450 2f31 2e31  P.$..kp..HTTP/1.1
0x0030: 2032 3030 204f 4b0d 0a43 6163 6865 2d43  .200.OK..Cache-C
0x0040: 6f6e 7472 6f6c 3a20 7072 6976 6174 650d  ontrol:.private.
0x0050: 0a43 6f6e 7465 6e74 2d54 7970 653a 2074  .Content-Type:.t
0x0060: 6578 742f 6874 6d6c 0d0a 5365 7276 6572  ext/html..Server
0x0070: 3a20 4757 532f 322e 310d 0a54 7261 6e73  :.GWS/2.1..Trans
0x0080: 6665 722d 456e 636f 6469 6e67 3a20 6368  fer-Encoding:.ch
0x0090: 756e 6b65 640d 0a43 6f6e 7465 6e74 2d45  unked..Content-E
0x00a0: 6e63 6f64 696e 673a 2067 7a69 700d 0a44  ncoding:.gzip..D
0x00b0: 6174 653a 2053 6174 2c20 3133 204e 6f76  ate:.Sat,.13.Nov
0x00c0: 2032 3030 3420 3139 3a33 313a 3533 2047  .2004.19:31:53.G
14:31:24.590598 IP 68.x.x.x.13602 > 64.233.161.104.80: . ack 1 win 5840
14:31:24.594328 IP 68.x.x.x.13602 > 64.233.161.104.80: . ack 1677 win 8580
14:31:24.734578 IP 64.233.161.104.80 > 68.x.x.x.13602: P 3107:3233(126) ack
742 win 9228
14:31:24.736286 IP 64.233.161.104.80 > 68.x.x.x.13602: . 1677:3107(1430)
ack 742 win 9228

```



```
14:31:24.736355 IP 64.233.161.104.80 > 68.x.x.x.13602: P 3233:3238(5) ack
742 win 9228
14:31:24.739226 IP 68.x.x.x.13602 > 64.233.161.104.80: . ack 1677 win 8580
14:31:24.742933 IP 68.x.x.x.13602 > 64.233.161.104.80: . ack 3233 win 11440
14:31:24.744033 IP 68.x.x.x.13602 > 64.233.161.104.80: . ack 3238 win 11440
```

---

The data replacement code recalculated the TCP checksum in the Transport-layer header to include the new value derived from the altered Application-layer data. Otherwise, the TCP session would be unable to maintain reliable delivery of data and would hence be corrupted. Also, the specific alteration of the Application-layer data maintained the validity of the HTTP protocol. The two facts combined are reflected in the *HTTP/1.1 200 OK* response (displayed in **bold** in the server response packets from the packet trace in Figure 5.9) from the Google Web server, and the Web browser that initiated the search request is completely unaware of the change and displays whatever is returned by Google. Not surprisingly, there are no matching search results for the string *Snort* at the site *www.nonexist.com* (which really does not exist), and so the results from Google are the standard “nothing found” message in Figure 5.10.

### Figure 5.10 Google Search Results

---

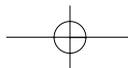
```
Your search - Snort site:www.nonexist.com - did not match any documents.
No pages were found containing "site:www.nonexist.com".
```

Suggestions:

- Make sure all words are spelled correctly.
- Try different keywords.
- Try more general keywords.
- Try fewer keywords.

Also, you can try Google Answers for expert help with your search.

---



**NOTE**

In the iptables commands that specify the destination *www.google.com* with the *-d* command line argument, it should be noted that because Google rotates the IP addresses that are returned by DNS for load-balancing purposes, more general iptables rules would need to be added to catch *all* Web search requests sent to *www.google.com*. When a hostname is supplied to iptables on the command line instead of a specific IP address in the standard dotted-quad notation, iptables issues a DNS request against the hostname and puts the resulting IP address (if returned by DNS) into the rule in question before adding it to the policy. This will be a single IP address associated with *www.google.com* and, therefore, will not include the next IP returned by the DNS rotation. Netfilter cannot enumerate all IP addresses that may be returned in a given DNS lookup, and it cannot take a hostname and use it directly within the running policy. You do not want to burden Netfilter by forcing it to issue a DNS request against every packet that traverses the firewall interfaces.

## Binary Data Matching and Replacement

The contents of packets generated by applications that use American Standard Code for Information Interchange (ASCII) protocols such as HTTP and Simple Mail Transfer Protocol (SMTP), are easy to visually inspect in the output of an Ethernet sniffer, but there are also many binary protocols (such as DNS) that heavily utilize non-ASCII printable characters in packet contents. So far, we have only performed Netfilter string matches against ASCII Application-layer data, but the string match extension also supports the inspection and replacement of hexadecimal codes in packet data through the use of the *—hex-string* and *—replace-hex-string* command-line arguments, respectively.

Suppose we want Netfilter to inform us whenever any machine on the internal network makes a *host address* or *A-record* DNS request for the IP address associated with *www.google.com* (see Figure 5.11). DNS requests are constructed by separating the name, or octet, components of a request with the length of the name or octet component, and then appending a request

type. For example, a *host address* request for the IP address associated with the hostname *www.google.com* will be converted to the number 3, followed by *www*, followed by the number 6, followed by *google*, and so forth, and finally a NULL character and a type code of 1 is appended to the query (see *RFC 1035* for a complete listing of DNS request types). Putting this all together, in hexadecimal notation our request becomes *0377 7777 0667 6f6f 676c 6503 636f 6d00 0001* and can be confirmed in the packet trace shown in **bold** characters in Figure 5.11. Armed with the knowledge of how a DNS request is constructed, the following Netfilter rule was written to detect a *host address* request for *www.google.com*:

```
iptables -I FORWARD 1 -p udp --dport 53 -m string --hex-string
"|03|www|06|google|03|com|00 00 01|" -j LOG --log-prefix "DNS www.google.com
"
```

If a DNS lookup is issued against *www.google.com* on any client system in the internal network in Figure 5.7, the following syslog message will be generated by Netfilter:

```
Nov 14 13:33:45 orthanc kernel: DNS www.google.com IN=eth1 OUT=eth0
SRC=192.168.10.2 DST=68.48.0.12 LEN=60 TOS=0x00 PREC=0x00 TTL=62 ID=0 DF
PROTO=UDP SPT=14798 DPT=53 LEN=40
```

This indicates that the string match extension has indeed detected the *A record* DNS lookup. If any other type of DNS request is made, such as for an *MX record*, no syslog message will be generated because we have restricted our search for DNS requests that include the *01* type (*host address*).

### Figure 5.11 Packet Trace for www.google.com Host Address Lookup

```
# tcpdump -i eth0 -nn -s 0 -l -X port 53
13:33:45.064331 IP 68.x.x.x.14798 > 68.48.0.12.53: 48167+ A?
www.google.com. (32)
    0x0000: 4500 003c 0000 4000 3e11 61c8 0202 0202  E..<..@.>.a.....
    0x0010: 0202 0202 39ce 0035 0028 9fe2 bc27 0100  ....9..5.(...'.
    0x0020: 0001 0000 0000 0000 0377 7777 0667 6f6f  ....www.goo
    0x0030: 676c 6503 636f 6d00 0001 0001      gle.com.....
```

## 164 Chapter 5 • Network Inline Data Modification

```

13:33:45.078956 IP 68.48.0.12.53 > 68.x.x.x.14798: 48167 3/11/11 CNAME
www.google.akadns.net., A 216.239.39.99, A 216.239.39.104 (487)
0x0000: 4500 0203 aa35 4000 f911 faca 0202 0202 E....5@.....
0x0010: 0202 0202 0035 39ce 01ef 6e73 bc27 8180 .....59...ns.'..
0x0020: 0001 0003 000b 000b 0377 7777 0667 6f6f .....www.goo
0x0030: 676c 6503 636f 6d00 0001 0001 c00c 0005 gle.com.....
0x0040: 0001 0000 01d0 0017 0377 7777 0667 6f6f .....www.goo
0x0050: 676c 6506 616b 6164 6e73 036e 6574 00c0 gle.akadns.net..
0x0060: 2c00 0100 0100 0000 7d00 04d8 ef27 63c0 ,.....}....'c.
0x0070: 2c00 0100 0100 0000 7d00 04d8 ef27 68c0 ,.....}....'h.
0x0080: 3700 0200 0100 01f6 bf00 0805 6173 6961 7.....asia
0x0090: 33c0 37c0 3700 0200 0100 01f6 bf00 0704 3.7.7.....
0x00a0: 6575 7233 c037 c037 0002 0001 0001 f6bf eur3.7.7.....
0x00b0: 0007 0475 7365 32c0 37c0 3700 0200 0100 ...use2.7.7.....
0x00c0: 01f6 bf00 0704 7573 6534 c037 c037 0002 .....use4.7.7..
0x00d0: 0001 0001 f6bf 0007 0475 7377 35c0 37c0 .....usw5.7.
0x00e0: 3700 0200 0100 01f6 bf00 0704 7573 7736 7.....usw6
0x00f0: c037 c037 0002 0001 0001 f6bf 0007 0475 .7.7.....u
0x0100: 7377 37c0 37c0 3700 0200 0100 01f6 bf00 sw7.7.7.....
0x0110: 0f02 7a61 0661 6b61 646e 7303 6f72 6700 ..za.akadns.org.

```

Lastly, we illustrate how the above Netfilter rule can be combined with the `--replace-hex-string` command line option to intercept and alter the *A record* DNS request for *www.google.com*. Again, it should be noted that this example is included to show, from a technical perspective, what is possible in terms of modifying Application layer-data as it passes through an inline device. Motivating examples for why this technology is important from an intrusion prevention standpoint are included in the “Attack Mitigation and Nullification” section later in this chapter. For this data modification example, we will intercept the *www.google.com* request and replace the hostname with the string *www.badreq.com*; the *host address* type field is left unchanged. Hence, the Netfilter rule becomes:

```

iptables -I FORWARD 1 -p udp --dport 53 -m string --hex-string
"|03|www|06|google|03|com|00 00 01|" --replace-hex-string

```

```
"|03|www|06|badreq|03|com|00 00 01|" -j LOG --log-prefix "DNS REPLACE
www.badreq.com "
```

Issuing an *A record* lookup for *www.google.com* from any internal system will result in the domain not being found. See the packet trace in Figure 5.12, and also the corresponding syslog message generated by Netfilter below:

```
Nov 14 13:38:13 orthanc kernel: DNS REPLACE www.badreq.com IN=eth1 OUT=eth0
SRC=192.168.10.2 DST=68.x.x.x LEN=60 TOS=0x00 PREC=0x00 TTL=62 ID=0 DF
PROTO=UDP SPT=14806 DPT=53 LEN=40
```

### Figure 5.12 Packet Trace of Modified www.google.com Host Address Lookup

```
# tcpdump -i eth0 -nn -s 0 -l -X port 53
13:38:13.040026 IP 68.x.x.x.14806 > 68.x.x.x.53: 13172+ A? www.badreq.com.
(32)
    0x0000:  4500 003c 0000 4000 3e11 61ce 0202 0202  E...<..@.>.a.....
    0x0010:  0202 0202 39d6 0035 0028 1fab 3374 0100  ....9..5(..3t..
    0x0020:  0001 0000 0000 0000 0377 7777 0662 6164  ....www.bad
    0x0030:  7265 7103 636f 6d00 0001 0001          req.com.....
13:38:13.565758 IP 68.x.x.x.53 > 68.x.x.x.14806: 13172 NXDomain 0/1/0
(105)
    0x0000:  4500 0085 1d93 4000 f911 88f1 0202 0202  E.....@.....
    0x0010:  0202 0202 0035 39d6 0071 e8d9 3374 8183  ....59..q..3t..
    0x0020:  0001 0000 0001 0000 0377 7777 0662 6164  ....www.bad
    0x0030:  7265 7103 636f 6d00 0001 0001 c017 0006  req.com.....
    0x0040:  0001 0000 1db7 003d 0161 0c67 746c 642d  ....=.a.gtld-
    0x0050:  7365 7276 6572 7303 6e65 7400 056e 7374  servers.net..nst
    0x0060:  6c64 0c76 6572 6973 6967 6e2d 6772 73c0  ld.verisign-grs.
    0x0070:  1741 9799 9500 0007 0800 0003 8400 093a  .A.....:
    0x0080:  8000 0003 84                               .....
```

## Tools & Traps...

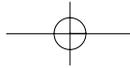
### The Packet Purgatory Library

At this point, we have presented material on both `Snort_inline` and the data replacement patch for the Netfilter string match extension. Each of these tools operates on an inline device and provides the ability to alter pre-defined strings at the Application layer as packets are intercepted and forwarded. However, what happens if a need arises to alter Application-layer data based on a more complex search criteria than either Netfilter or `Snort_inline` is able to provide? The *Packet Purgatory* library, written by Todd MacDermid and available at [www.synacklabs.net/projects/packetp/](http://www.synacklabs.net/projects/packetp/), offers a programmatic interface to packet alteration. The Packet Purgatory library is meant to be deployed on an endpoint host, and functions are provided to modify either inbound or outbound traffic from the host. The logic used to search and replace Application-layer data is limited only by the programmer's imagination and ingenuity. The Packet Purgatory library can also be used to alter other portions of IP packets such as the Network- or Transport-layer headers. Neither the Netfilter data replacement patch nor `Snort_inline` offer this capability and can only modify Application-layer data. The Packet Purgatory Web site includes the following motivating example for why this capability is interesting:

Suppose your ISP is using the Type of Service (TOS) field of the IP header to route traffic based on the priority contained therein? With Packet Purgatory it is possible to artificially increase the priority of the traffic generated by your system by elevating the TOS field value.

## Application-Layer Byte Replacement

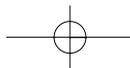
So far, we have presented two pieces of software—`Snort_inline` and the data replacement patch for the Netfilter string match extension—that are capable



of modifying Application-layer data as packets traverse the interfaces of an inline device. We have yet to provide any motivation for why this might be a good technology from the standpoint of intrusion prevention. In this section, we show how altering Application-layer data associated with network-based attacks can provide an effective method for nullifying such attacks while at the same time reducing the chances that an attacker will discover that an inline IPS is deployed on the target network.

As we discussed in Chapter 1, when a vulnerability is announced for an application that is deployed on a network, the vendor provides a patch or update to fix the offending code, or releases an entirely new version of the application in question that fixes the vulnerability. If such an application is 1) critical to your business and 2) run as a server that is meant to be globally accessible, it is not feasible to simply restrict access to the server with a firewall or router ACL. Hence, before the vendor releases a patch or other fix for the application, the only way to guard against an exploit for the vulnerability is to deploy a preventative system (either host-based or network-based) that can allow legitimate application functionality to continue, but stop the specific exploit in its tracks. Good examples of such applications are Web and mail servers, which must (usually) remain globally accessible.

In some cases, even if the vendor releases a patch for a vulnerable application, it may be difficult to deploy or cause more problems than it is designed to solve. For example, many companies delayed the deployment of Microsoft's Service Pack 2 (SP2) for Windows XP, which was over 270MB large and contained fixes for several high-profile vulnerabilities, so that it could be evaluated for inclusion in their corporate networks. Companies could have installed individual security updates that were available from Microsoft for the vulnerabilities fixed by SP2, but considering that SP2 fixed over 50 vulnerabilities, deploying a security update for each would involve a significant time investment. However, as of this writing, there is an important vulnerability in Microsoft's Internet Explorer called *IFRAME* for which there is no separate patch available. The *IFRAME* vulnerability affects all versions of Internet Explorer except those deployed on Windows XP systems with SP2 installed, and there is now an entire family of worms called the *BOFRA* family that are



**168 Chapter 5 • Network Inline Data Modification**

designed to exploit this vulnerability (see [www.sophos.com/virusinfo/articles/how-bofrawork.html](http://www.sophos.com/virusinfo/articles/how-bofrawork.html)). Without question, the best strategy for protecting any software against arbitrary attacks is to fix all vulnerabilities contained therein. However, with an IPS, it is possible to buy some time against specific attacks for known vulnerabilities while the vendor develops a security patch that can be easily applied.

Network intrusion prevention at the Application layer via inline modification of application data offers several advantages. First, stopping attacks via Application-layer data modification allows the inline IPS to reduce the chances that the attacker will discover that an inline IPS is deployed on the target network. Even after expending substantial effort in reconnaissance against a target system to find vulnerable applications on the target, an attacker cannot be absolutely confident that an exploit will work.

For example, if the targeted application suffers from a buffer-overflow vulnerability, the attacker may have to iterate many possible offsets in order to successfully exploit the vulnerability. An inline IPS can take advantage of this fact by tweaking the exploit packets in an effort to render them harmless before they reach the target, and because the attacker cannot always expect success anyway, the IPS remains hidden to the attacker while providing protection at the same time. On the other hand, an attacker would immediately notice if an intrusion prevention mechanism were to generate TCP reset packets or add blocking rules to a local firewall in response to an exploit on the network sent to the target system.

Altering Application-layer data also allows application error codes to be preserved. For example, a Web server already has a well-defined mechanism for returning error codes for various invalid requests such as a *404 File Not Found* for a request for a non-existent URL, or a *401 Permission Denied* in response to a request for a URL to which the requestor does not have permission to access and/or execute. If an attack is embodied in a request to a Web server, an inline IPS can alter the attack such that the Web server returns a legitimate error code. This may have the effect of convincing the attacker that the Web server is not vulnerable to the attack while at the same time not revealing the fact that an inline IPS is in use on the target network.

There are also some disadvantages to implementing intrusion prevention via Application-layer data modification. First, altering Application-layer data requires extremely detailed knowledge of application protocols, what specific bytes signify, and how they are used. There is a difference between data used by an application and data that forms an application protocol. A good example can be derived from the standard of Web servers; text in a Web page is data used by the application (in this case a Web browser) vs. HTTP header information (used to construct a valid HTTP session by which Web pages are transmitted). An inline device can alter *any* Application-layer bytes, but without knowing which bytes may legitimately be changed. To allow the application to continue to function is asking for trouble. Second, IPS' fundamentally rely on techniques implemented by IDS' to detect attacks. In a signature-based IDS (such as Snort), there are signatures that allow attacks to be detected but the specific characteristics of the attack that are searched for by the signature are not part of the damaging portion of the attack. Hence, simply altering the bytes associated with such a signature would allow the attack through unimpeded. An analogy can be made between this situation and having an inline device alter the *SSH-2.0-OpenSSH\_3.8.1p1* banner provided by an OpenSSH server to read *SSH-2.0-OpenSSH\_3.7.0p0*, as a client connects to the SSH server. The server is still version 3.8.1p1, even though the banner communicated to the client seems to indicate that the server is actually version 3.7.0p0. For a concrete example that demonstrates the ineffectiveness of stopping attacks that are detected by certain signatures via Application-layer byte modification, consider the Snort rule *gobbles SSH exploit attempt (sid 1812)* below:

```
exploit.rules:alert tcp $EXTERNAL_NET any -> $HOME_NET 22 (msg:"EXPLOIT
gobbles SSH exploit attempt"; flow:to_server,established; content:"GOBBLES";
reference:bugtraq,5093; reference:cve,2002-0390; reference:cve,2002-0639;
classtype:misc-attack; sid:1812; rev:5;)
```

This rule looks for the string “GOBBLES” over TCP port 22 directed at an SSH server on the internal network. Simply replacing this string with any other string of the same length will not stop the attack from succeeding against a vulnerable SSH server. However, if an inline IPS implemented active

**170 Chapter 5 • Network Inline Data Modification**

response at the Transport layer instead of the Application layer by issuing a TCP reset packet to tear the entire session down after detecting the GOB-BLES string, the attack would (most likely) have been stopped in its tracks. In this instance, trying to implement stealthy intrusion prevention at the Application layer is not possible given the nature of the signature designed to detect the attack, and hence requires an approach based more on brute force at the Transport layer or below.

Now, we turn to some specific exploit examples and show how Snort and the Netfilter data replacement patch each thwart the attacks via Application-layer data modification. We illustrate three attacks and display packet traces for each as they are sent across the network. The first is a simulated buffer overflow exploit against a DNS server, the second is an instance of a Frontpage extensions attack, and the third is an automated exploit from the Metasploit Project (see [www.metasploit.com](http://www.metasploit.com)) for a vulnerability in the Microsoft Local Security Authority Subsystem Service (LSASS). The LSASS vulnerability was discovered by eEye Digital Security (see [www.eeye.com](http://www.eeye.com)) and was assigned the Microsoft Security Bulletin number MS04-011. This vulnerability was exploited by the *Sasser* worm (see [www.lurhq.com/sasser.html](http://www.lurhq.com/sasser.html)) in mid-2004, and requires a fairly complicated Snort rule to detect. Both the DNS and the Frontpage attacks can be stopped by the Netfilter data replacement patch, but for the LSASS exploit, simple string matching against packet contents is not powerful enough to effectively detect the attack and hence requires the highly descriptive Snort rules language.



## DNS Exploit: x86 Linux Overflow

In this example, we simulate an exploit that corresponds to Snort rule *sid 264*. As of Snort-2.3.0RC1, this particular rule has no reference keywords defined that tie the rule to a specific Bugtraq or CVE (Common Vulnerabilities and Exposures) ID, or to a specific exploit. There is little additional information in the *snort/docs/signatures/264.txt* file, so we have no clear direction in which to search for sample exploit code or specific versions of DNS servers that are vulnerable to attack. Therefore, we must resort to attack simulations. However, the fact that a Snort rule exists and appears to generi-

cally detect an overflow exploit attempt against a DNS server, is a pretty good indication that if you see a corresponding sequence of bytes against a DNS server on your network you should be concerned. The Snort rule *sid 264* appears below:

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 53 (msg:"DNS EXPLOIT x86 Linux
overflow attempt"; flow:to_server,established; content:"1|C0 B0 02 CD 80 85
C0|uL|EB|L^|B0|"; classtype:attempted-admin; sid:264; rev:6;)
```

A quick examination of the above rule shows that to simulate the attack we must establish a TCP session over port 53 (DNS) to a system attached to the internal network, and then, within this established session, the sequence of bytes `1|C0 B0 02 CD 80 85 C0|uL|EB|L^|B0|` must be sent from the client to the server. This leads us to write the following Snort and Netfilter rules to foil such an attack (note the replacement of the bytes in the original Snort rule with the meaningless sequence of fourteen “e” characters in **bold**):

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 53 (msg:"DNS EXPLOIT x86 Linux
overflow attempt"; flow:to_server,established; content:"1|C0 B0 02 CD 80 85
C0|uL|EB|L^|B0|"; replace:"eeeeeeeeeeeeee"; classtype:attempted-admin;
sid:264; rev:6;)
# iptables -I FORWARD 1 -p tcp --dport 53 -m string --hex-string "1|C0 B0
02 CD 80 85 C0|uL|EB|L^|B0" -replace-string "eeeeeeeeeeeeee" -j LOG --log-
prefix "NULLIFY SID264 "
```



For this example, we implement the above Netfilter rule into the firewall system displayed in Figure 5.9 (implementing this Snort rule in the Snort system in Figure 5.7 would have a similar effect on network traffic). To simulate the attack, we use simple command-line invocations *perl* and *netcat* together (see [www.securityfocus.com/data/tools/137](http://www.securityfocus.com/data/tools/137)) to send the hexadecimal codes `1|C0 B0 02 CD 80 85 C0|uL|EB|L^|B0` across the network to the DNS server:

```
perl -e 'print "1\xc0\xb0\x02\xcd\x80\x85\xc0uL\xebL^\xb0"' |nc 68.x.x.x 53
```

This results in the network trace in Figure 5.13 being generated on the egress interface of the firewall. The altered bytes in the TCP stream are displayed in **bold** characters. Note that when the packet data hits the DNS

## 172 Chapter 5 • Network Inline Data Modification

server it does not correspond to any legitimate DNS request; therefore, the server responds with the application error “Server failure.”

**Figure 5.13** Netfilter Data Replacement of Snort Rule SID 264

---

```

00:38:01.436452 IP 68.x.x.x.47375 > 68.x.x.x.53: S 3294500039:3294500039(0)
win 5840 <mss 1460,sackOK,timestamp 86999640 0,nop,wscale 0>
00:38:01.448622 IP 68.x.x.x.53 > 68.x.x.x.47375: S 1976415119:1976415119(0)
ack 3294500040 win 10136 <nop,nop,timestamp 2263379902 86999640,nop,wscale
0,nop,nop,sackOK,mss 1460>
00:38:01.448680 IP 68.x.x.x.47375 > 68.x.x.x.53: . ack 1 win 5840
<nop,nop,timestamp 86999641 2263379902>
00:38:01.448765 IP 68.x.x.x.47375 > 68.x.x.x.53: P 1:15(14) ack 1 win 5840
<nop,nop,timestamp 86999641 2263379902> 25957 updateM+ [b2&3=0x6565]
[25957a] [25957q] [25957n] [25957au][|domain]
    0x0000: 4500 0042 96a1 4000 4006 c931 0202 0202 E..B..@.@..1....
    0x0010: 0202 0202 b90f 0035 c45e 14c8 75cd b390 .....5.^..u...
    0x0020: 8018 16d0 8535 0000 0101 080a 052f 8259 .....5...../.Y
    0x0030: 86e8 6fbe 6565 6565 6565 6565 6565 6565 ..o.eeeeeeeeeee
    0x0040: 6565 ee
00:38:01.460466 IP 68.x.x.x.53 > 68.x.x.x.47375: . ack 15 win 10122
<nop,nop,timestamp 2263379903 86999641>
00:38:02.676420 IP 68.x.x.x.47375 > 68.x.x.x.53: F 15:15(0) ack 1 win 5840
<nop,nop,timestamp 86999764 2263379903>
00:38:02.689452 IP 68.x.x.x.53 > 68.x.x.x.47375: . ack 16 win 10136
<nop,nop,timestamp 2263380026 86999764>
00:38:02.690143 IP 68.x.x.x.53 > 68.x.x.x.47375: P 1:15(14) ack 16 win
10136 <nop,nop,timestamp 2263380026 86999764> 25957 updateM ServFail*$ [0q]
0/0/0 (12)
    0x0000: 4500 0042 5d70 4000 f906 4962 0202 0202 E..B]p@...Ib....
    0x0010: 0202 0202 0035 b90f 75cd b390 c45e 14d7 .....5..u....^..
    0x0020: 8018 2798 edd8 0000 0101 080a 86e8 703a ..'.....p:
    0x0030: 052f 82d4 000c 6565 e5e2 0000 0000 0000
./.....ee.....
    0x0040: 0000 ..

```

---

## Notes from the Underground...

### IPS Evasion

Like most attacks detected by the Snort IDS, the previous DNS overflow exploit example was detected by searching for specific sequences of bytes in the Application portion of packets as they traverse a network. Both the Snort and Netfilter data replacement patch have ample machinery to detect and nullify such an attack if the byte sequence is contained in a single packet. But what happens if we split the sequence across multiple packets? The Snort *stream4* preprocessor can easily deal with this possibility, because it searches for byte sequences across a reassembled stream; however, it does not function together with Snort running in inline mode. Hence, splitting the attack across two or more packets will defeat the detection mechanism in Snort.

The same is also true of the Netfilter string match extension (and the data replacement patch with it), due to the fact that it is strictly limited to searching for strings in individual packets. The Snort\_inline project has not neglected this detail, and in an upcoming release a new version of the stream4 preprocessor will be available that is compatible with Snort running in inline mode. Most of the work accomplished so far on the inline-aware version of stream4, has been done by Victor Julien. Note that there are many IDS evasion techniques; TCP session splicing is just one example. Another important technique is Network-layer packet fragmentation as implemented by Dug Song's *fragroute* (see [www.monkey.org/~dugsong/fragroute](http://www.monkey.org/~dugsong/fragroute)). One of the best references for more information about IDS evasion techniques is the paper "Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection" by Tomas H. Ptacek and Timothy N. Newsham (see [www.insecure.org/stf/secnet\\_ids/secnet\\_ids.html](http://www.insecure.org/stf/secnet_ids/secnet_ids.html)). To illustrate the evasion of either Snort or the Netfilter string match extension, we appeal to the simple perl script below which splits the DNS Linux overflow attack across two packets:

Continued

[www.syngress.com](http://www.syngress.com)

## 174 Chapter 5 • Network Inline Data Modification



```
#!/usr/bin/perl -w

use IO::Socket;
use Time::HiRes 'usleep';
use strict;

my $sock = new IO::Socket::INET(
    PeerAddr => '192.168.40.2',
    PeerPort => 53,
    Proto    => 'tcp',
    Timeout  => 7
);

die "[*] Could not open socket: $!" unless defined $sock;

### use usleep() to introduce a brief time delay between
### printing data to the socket
print $sock "1\xc0\xb0\x02\xcd\x80\x85";
usleep 10;
print $sock "\xc0uL\xebL^\xb0";
close $sock;

exit 0;
```

## Microsoft Frontpage Server Extensions Attack

This example illustrates how an inline IPS can thwart an attack against a Microsoft Internet Information Server (IIS) Web server that is running Frontpage server extensions. The attack is known as the *Microsoft Frontpage Server Extensions Path Disclosure Vulnerability*, and is associated with Bugtraq ID 1174 and CVE ID CAN-2000-0413. This vulnerability affects Frontpage versions 1.1 and earlier, and allows an attacker to extract the full path to the Web root directory on the underlying server by issuing a normal Web request against a non-existent file. For example, issuing a Web request for URL

`http://targetsystem/_vti_bin/shtml.dll/badfile.html` will cause a vulnerable IIS server to respond with an error page that includes the full path to the local Web root directory, which would look something like `C:\localhost\wwwroot\badfile.html`. The `shtml.dll` file may be installed at `shtml.exe`, depending on which platform IIS is installed.

There have been multiple vulnerabilities associated with Microsoft Frontpage extensions; see Matt Shannon's DEFCON 11 presentation `_vti_fpx-exploitation` ([www.defcon.org](http://www.defcon.org)). The Snort community even has a rules file, `web-frontpage.rules`, that is dedicated to detecting attacks against IIS servers that are running Frontpage extensions. This file contains 35 signature rules, and Snort ID (or SID) 940 specifically detects when someone is trying to exploit the path disclosure vulnerability mentioned above. The normal Snort rule 940 appears in Figure 5.14, along with a modified version that replaces the string `/_vti_bin/shtml.dll` with the harmless (and non-existent) URL string `/Ovti0bin/shtml.dab` with the `replace` keyword in **bold** characters. In Figure 5.15, we illustrate how a Netfilter rule can be written that is equivalent to Snort rule 940, and also a modified Netfilter rule that performs the same data replacement as the example Snort\_inline rule in Figure 5.14.

**Figure 5.14** Snort Rule sid 940 (Normal and Modified)

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"WEB-
FRONTPAGE shtml.dll access"; flow:to_server,established;
uricontent:"/_vti_bin/shtml.dll"; nocase; reference:arachnids,292;
reference:bugtraq,1174; reference:bugtraq,1594; reference:bugtraq,1595;
reference:cve,2000-0413; reference:cve,2000-0746; reference:nessus,11395;
reference:url,www.microsoft.com/technet/security/bulletin/ms00-060.mspx;
classtype:web-application-activity; sid:940; rev:15;)
```

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"WEB-
FRONTPAGE shtml.dll access"; flow:to_server,established;
uricontent:"/_vti_bin/shtml.dll"; replace:"/Ovti0bin/shtml.dab; nocase;
reference:arachnids,292; reference:bugtraq,1174; reference:bugtraq,1594;
reference:bugtraq,1595; reference:cve,2000-0413; reference:cve,2000-0746;
reference:nessus,11395;
```

## 176 Chapter 5 • Network Inline Data Modification

```
reference:url,www.microsoft.com/technet/security/bulletin/ms00-060.msp;
classtype:web-application-activity; sid:940; rev:15;)
```

**Figure 5.15** Netfilter Detection Rule for Snort Rule sid 940 (Normal and Modified)

```
# iptables -I FORWARD 1 -p tcp --dport 80 -m string --string
"/_vti_bin/shtml.dll" -j LOG --log-prefix "SID940 "

# iptables -I FORWARD 1 -p tcp --dport 80 -m string --string
"/_vti_bin/shtml.dll" --replace-string "/OvtiObin/shtml.dab" -j LOG --log-
prefix "NULLIFY SID940 "
```

Now we will demonstrate the attack and show how replacing the string `/_vti_bin/shtml.dll` with `/OvtiObin/shtml.dab` thwarts the attack. For this we use the `wget` Web client program installed on any Linux system, and issue a request against an IIS 5.0 Web server. The `wget` command and corresponding output appears below as it is executed on the Linux client system in Figure 5.7:

```
$ wget http://209.x.x.x/_vti_bin/shtml.dll/badfile.html
Connecting to 209.x.x.x:80... connected.
HTTP request sent, awaiting response... 404 Object Not Found
14:27:00 ERROR 404: Object Not Found.
```

**Figure 5.16** Packet Trace of Nullified Frontpage Attack (Bugtraq ID 1174)

```
17:04:35.836713 IP 192.168.20.6.33283 > 209.x.x.x.80: S
3884145455:3884145455(0) win 5840 <mss 1460,sackOK,timestamp 23203340
0,nop,wscale 0> 17:04:35.861179 IP 209.x.x.x.80 > 192.168.20.6.33283: S
1771239932:1771239932(0) ack 3884145456 win 17520 <mss 1460,nop,wscale
0,nop,nop,timestamp 0 0,nop,nop,sackOK>
17:04:35.861236 IP 192.168.20.6.33283 > 209.x.x.x.80: . ack 1 win 5840
<nop,nop,timestamp 23203367 0>
17:04:35.861532 IP 192.168.20.6.33283 > 209.x.x.x.80: P 1:515(514) ack 1
win 5840 <nop,nop,timestamp 23203367 0>
0x0000: 4500 0236 5525 4000 4006 1331 c0a8 1406 E..6U%@.@..1....
0x0010: 0202 0202 8203 0050 e783 5b30 6992 f9fd ..*6...P..[0i...
0x0020: 8018 16d0 e0f8 0000 0101 080a 0162 0e27 .....b.'
```

```

0x0030: 0000 0000 4745 5420 2f30 7674 6930 6269 ....GET./0vti0bi
0x0040: 6e2f 7368 746d 6c2e 6461 622f 6261 6466 n/shtml.dab/badf
0x0050: 696c 652e 6874 6d6c 2048 5454 502f 312e ile.html.HTTP/1.
0x0060: 310d 0a48 6f73 743a 2077 7777 2e75 7369 1..Host:.www.usi
0x0070: 2e6e 6574 0d0a 5573 6572 2d41 6765 6e74 .net..User-Agent
0x0080: 3a20 4d6f 7a69 6c6c 612f 352e 3020 2858 :.Mozilla/5.0.(X
0x0090: 3131 3b20 553b 204c 696e 7578 2069 3638 11;.U;.Linux.i68
0x00a0: 363b 2072 763a 312e 372e 3329 2047 6563 6;.rv:1.7.3).Gec
0x00b0: 6b6f 2f32 3030 3431 3032 3520 4669 7265 ko/20041025.Fire
0x00c0: 666f 782f 302e 3130 2e31 0d0a 4163 6365 fox/0.10.1..Acce
17:04:35.889787 IP 209.x.x.x.80 > 192.168.20.6.33283: . ack 515
win 17006 <nop,nop,timestamp 131132416 23203367>
17:04:35.894965 IP 209.x.x.x.80 > 192.168.20.6.33283: P 1:164(163) ack 515
win 17006 <nop,nop,timestamp 131132416 23203367>
0x0000: 4500 00d7 edb9 0000 7006 8bfb 0202 0202 E.....p.....*6
0x0010: c0a8 1406 0050 8203 6992 f9fd e783 5d32 .....P..i.....]2
0x0020: 8018 426e d270 0000 0101 080a 07d0 ec00 ..Bn.p.....
0x0030: 0162 0e27 4854 5450 2f31 2e31 2034 3034 .b.'HTTP/1.1.404
0x0040: 204f 626a 6563 7420 4e6f 7420 466f 756e .Object.Not.Foun
0x0050: 640d 0a53 6572 7665 723a 204d 6963 726f d..Server:.Micro
0x0060: 736f 6674 2d49 4953 2f35 2e30 0d0a 4461 soft-IIS/5.0..Da
0x0070: 7465 3a20 5361 742c 2032 3020 4e6f 7620 te:.Sat,.20.Nov.
0x0080: 3230 3034 2032 313a 3437 3a33 3220 474d 2004.21:47:32.GM
0x0090: 540d 0a43 6f6e 6e65 6374 696f 6e3a 2063 T..Connection:.c
0x00a0: 6c6f 7365 0d0a 436f 6e74 656e 742d 5479 lose..Content-Ty
0x00b0: 7065 3a20 7465 7874 2f68 746d 6c0d 0a43 pe:.text/html..C
0x00c0: 6f6e 7465 6e74 2d4c 656e 6774 683a 2034 ontent-Length:.4
17:04:35.895738 IP 192.168.20.6.33283 > 209.x.x.x.80: . ack 164
win 6432 <nop,nop,timestamp 23203402 131132416>
17:04:35.901527 IP 209.x.x.x.80 > 192.168.20.6.33283: . 164:1612(1448) ack
515 win 17006 <nop,nop,timestamp 131132416 23203367>
0x0000: 4500 05dc edba 0000 7006 86f5 0202 0202 E.....p.....*6
0x0010: c0a8 1406 0050 8203 6992 faa0 e783 5d32 .....P..i.....]2
0x0020: 8010 426e 7c3b 0000 0101 080a 07d0 ec00 ..Bn|;.....
0x0030: 0162 0e27 3c21 444f 4354 5950 4520 4854 .b.'<!DOCTYPE.HT

```

## 178 Chapter 5 • Network Inline Data Modification

```

0x0040: 4d4c 2050 5542 4c49 4320 222d 2f2f 5733 ML.PUBLIC."-//W3
0x0050: 432f 2f44 5444 2048 544d 4c20 332e 3220 C//DTD.HTML.3.2.
0x0060: 4669 6e61 6c2f 2f45 4e22 3e0d 0a3c 6874 Final//EN">..<ht
0x0070: 6d6c 2064 6972 3d6c 7472 3e0d 0a0d 0a3c ml.dir=ltr>....<

```

```

17:04:35.901553 IP 192.168.20.6.33283 > 209.x.x.x.80: . ack 1612 win 8688
<nop,nop,timestamp 23203407 131132416>

```

```

17:04:35.903521 IP 192.168.20.6.33283 > 209.x.x.x.80: . ack 2921 win 11584
<nop,nop,timestamp 23203409 131132416>

```

```

17:04:35.931245 IP 209.x.x.x.80 > 192.168.20.6.33283: FP 2921:4204(1283)
ack 515 win 17006 <nop,nop,timestamp 131132416 23203402>

```

```

0x0000: 4500 0537 edbc 0000 7006 8798 0202 0202 E..7....p.....*6
0x0010: c0a8 1406 0050 8203 6993 0565 e783 5d32 .....P..i..e..]2
0x0020: 8019 426e 569b 0000 0101 080a 07d0 ec00 ..BnV.....
0x0030: 0162 0e4a 2063 6861 6e67 6564 2c20 6f72 .b.J.changed,.or
0x0040: 2069 7320 7465 6d70 6f72 6172 696c 7920 .is.temporarily.
0x0050: 756e 6176 6169 6c61 626c 652e 3c2f 666f unavailable.</fo
0x0060: 6e74 3e3c 2f74 643e 0d0a 2020 3c2f 7472 nt></td>....</tr
0x0070: 3e0d 0a20 200d 0a20 203c 7472 3e0d 0a20 >.....<tr>...
0x0080: 2020 203c 7464 2077 6964 7468 3d22 3430 ...<td.width="40
0x0090: 3022 2063 6f6c 7370 616e 3d22 3222 3e0d 0".colspan="2">.
0x00a0: 0a09 3c66 6f6e 7420 7374 796c 653d 2243 ..<font.style="C
0x00b0: 4f4c 4f52 3a30 3030 3030 303b 2046 4f4e OLOR:000000;.FON
0x00c0: 543a 2038 7074 2f31 3170 7420 7665 7264 T:.8pt/11pt.verd
0x00d0: 616e 6122 3e0d 0a0d 0a09 3c68 7220 636f ana">....<hr.co
0x00e0: 6c6f 723d 2223 4330 4330 4330 2220 6e6f lor="#C0C0C0".no
0x00f0: 7368 6164 653e 0d0a 090d 0a20 2020 203c shade>.....<
0x0100: 703e 506c 6561 7365 2074 7279 2074 6865 p>Please.try.the
0x0110: 2066 6f6c 6c6f 7769 6e67 3a3c 2f70 3e0d .following:</p>.
0x0120: 0a0d 0a09 3c75 6c3e 0d0a 2020 2020 2020 ....<ul>.....
0x0130: 3c6c 693e 4966 2079 6f75 2074 7970 6564 <li>If.you.typed
0x0140: 2074 6865 2070 6167 6520 6164 6472 6573 .the.page.address
0x0150: 7320 696e 2074 6865 2041 6464 7265 7373 s.in.the.Address
0x0160: 2062 6172 2c20 6d61 6b65 2073 7572 6520 .bar,.make.sure.
0x0170: 7468 6174 2069 7420 6973 2073 7065 6c6c that.it.is.spell

```

```
0x0180: 6564 2063 6f72 7265 6374 6c79 2e3c 6272 ed.correctly.<br
17:04:35.931502 IP 192.168.20.6.33283 > 209.x.x.x.80: F 515:515(0) ack 4205
win 14480 <nop,nop,timestamp 23203437 131132416>
```

## Metasploit LSASS Exploit

For the last example, we turn to *Metasploit* (see [www.metasploit.com](http://www.metasploit.com)). Metasploit was developed by Spoonm and H. D. Moore as a framework for assisting penetration testing and exploit development. As of this writing, Metasploit contains over 30 different exploits for various software running on several different OS', from Windows XP to Linux. Metasploit is free and open source software and is released under the terms of two licenses: the GNU Public License (GPL) and the Perl Artistic License. In this example, we use the Metasploit implementation of an exploit for the well-known LSASS vulnerability:

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 139 (msg:"NETBIOS SMB DCERPC LSASS
DsRolerUpgradeDownlevelServer exploit attempt"; flow:to_server,established;
flowbits:isset,netbios.lsass.bind.attempt; content:"|FF|SMB"; depth:4;
offset:4; nocase; content:"|05|"; distance:59; content:"|00|"; within:1;
distance:1; content:"|09 00|"; within:2; distance:19;
reference:bugtraq,10108; reference:cve,2003-0533;
reference:url,www.microsoft.com/technet/security/bulletin/MS04-011.mspx;
classtype:attempted-admin; sid:2511; rev:9;)
```

As you can see, sid 2511 is fairly complex and requires the use of multiple content fields along with the use of the *distance*, *depth*, and *within* keywords. This complexity places the detection of the LSASS vulnerability out of reach of the Netfilter string match extension; hence, effectively detecting an attack that is based around this vulnerability requires the use of Snort. With the *replace* keyword, we are able to instruct Snort to not only detect the attack, but to thwart it at the same time (see the modified sid 2511 rule below):

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 139 (msg:"NETBIOS SMB DCERPC LSASS
DsRolerUpgradeDownlevelServer exploit attempt"; flow:to_server,established;
flowbits:isset,netbios.lsass.bind.attempt; content:"|FF|SMB"; depth:4;
offset:4; nocase; content:"|05|"; distance:59; content:"|00|"; within:1;
distance:1; content:"|09 00|"; within:2; distance:19; replace:"|ee ee|";
reference:bugtraq,10108; reference:cve,2003-0533;
```

## 180 Chapter 5 • Network Inline Data Modification

```
reference:url,www.microsoft.com/technet/security/bulletin/MS04-011.msp;
classtype:attempted-admin; sid:2511; rev:9;)
```

Next, we start up the Metasploit framework and attack the Windows 2000 host in Figure 5.9. The first attack is sent across the network to the host without configuring Snort to respond in any way, to illustrate what happens without running any IPS. The sequence of commands given to Metasploit along with their corresponding output is displayed below as Metasploit successfully “roots” the Windows 2000 host. We select the exploit for the LSASS vulnerability, set the payload to *win32\_bind* (which binds a command shell to a port and waits for a connection), set the target OS (Windows 2000) and IP address, and launch the exploit:

```
# ./msfconsole

      _____
     /         \ |_____  _____| |  ____|_|/  |
    /   \_/  _ \  _\___ \ /  ___/\___ \ | | /  _ \ | \  _\
 |  Y Y  \  ___/ | | /  _ \___ \ | |> > |_( <_> ) || |
 |__|_|  /\___ >_| (___ /___ >|  _/|___/\___/|_|_|
          \ /      \ / v2.2      \ /      \ / |__|

+ -- ---[ msfconsole v2.2 [33 exploits - 33 payloads]

msf > show exploits

Metasploit Framework Loaded Exploits
=====

Credits                                     Metasploit Framework Credits
afp_loginext                               AppleFileServer LoginExt PathName Buffer
Overflow
apache_chunked_win32                       Apache Win32 Chunked Encoding
blackice_pam_icq                           ISS PAM.dll ICQ Parser Buffer Overflow
distcc_exec                                 DistCC Daemon Command Execution
```

|                                |   |
|--------------------------------|---|
| exchange2000_xexch50           | Exchange 2000 MS03-46 Heap Overflow         |
| frontpage_fp30reg_chunked      | Frontpage fp30reg.dll Chunked Encoding      |
| ia_webmail                     | IA WebMail 3.x Buffer Overflow              |
| icecast_header                 | Icecast (<= 2.0.1) Header Overwrite (win32) |
| iis50_nsiislog_post            | IIS 5.0 nsiislog.dll POST Overflow          |
| iis50_printer_overflow         | IIS 5.0 Printer Buffer Overflow             |
| iis50_webdav_ntdll             | IIS 5.0 WebDAV ntdll.dll Overflow           |
| imail_ldap                     | IMail LDAP Service Buffer Overflow          |
| lsass_ms04_011                 | Microsoft LSASS MS04-011 Overflow           |
| mercantec_softcart             | Mercantec SoftCart CGI overflow             |
| msrpc_dcom_ms03_026            | Microsoft RPC DCOM MS03-026                 |
| mssql2000_preauthentication    | Microsoft SQL Server Hello Buffer Overflow  |
| mssql2000_resolution           | MSSQL 2000 Resolution Overflow              |
| openview_omniback              | HP OpenView Omniback II Command Execution   |
| poptop_negative_read           | Poptop Negative Read Overflow               |
| realserver_describe_linux      | RealServer Describe Buffer Overflow         |
| samba_nttrans                  | Samba Fragment Reassembly Overflow          |
| samba_trans2open               | Samba trans2open Overflow                   |
| sambar6_search_results         | Sambar 6 Search Results Buffer Overflow     |
| servu_mdtm_overflow            | Serv-U FTPD MDTM Overflow                   |
| smb_sniffer                    | SMB Password Capture Service                |
| solaris_sadmind_exec           | Solaris sadmind Command Execution           |
| squid_ntlm_authenticate        | Squid NTLM Authenticate Overflow            |
| svnserve_date                  | Subversion Date Svnserve                    |
| ut2004_secure_linux<br>(Linux) | Unreal Tournament 2004 "secure" Overflow    |
| ut2004_secure_win32<br>(Win32) | Unreal Tournament 2004 "secure" Overflow    |
| warftpd_165_pass               | War-FTPD 1.65 PASS Overflow                 |
| windows_ssl_pct                | Windows SSL PCT Overflow                    |

```

msf > use lsass_ms04_011
msf lsass_ms04_011 > show payloads

```

Metasploit Framework Usable Payloads

**182 Chapter 5 • Network Inline Data Modification**

```
=====

win32_adduser           Windows Execute net user /ADD
win32_bind              Windows Bind Shell
win32_bind_dllinject   Windows Bind DLL Inject
win32_bind_stg         Windows Staged Bind Shell
win32_bind_stg_upexec  Windows Staged Bind Upload/Execute
win32_bind_vncinject   Windows Bind VNC Server DLL Inject
win32_exec             Windows Execute Command
win32_reverse          Windows Reverse Shell
win32_reverse_dllinject Windows Reverse DLL Inject
win32_reverse_stg     Windows Staged Reverse Shell
win32_reverse_stg_ie  Windows Reverse InlineEgg Stager
win32_reverse_stg_upexec Windows Staged Reverse Upload/Execute
win32_reverse_vncinject Windows Reverse VNC Server DLL Inject

msf lsass_ms04_011 > set PAYLOAD win32_bind
PAYLOAD -> win32_bind
msf lsass_ms04_011(win32_bind) > show targets

Supported Exploit Targets
=====

0 Automatic
1 Windows 2000
2 Windows XP
msf lsass_ms04_011(win32_bind) > set TARGET 1
TARGET -> 1
msf lsass_ms04_011(win32_bind) > set RHOST 192.168.40.2
RHOST -> 192.168.40.2
msf lsass_ms04_011(win32_bind) > set
PAYLOAD: win32_bind
RHOST: 192.168.40.2
TARGET: 1
msf lsass_ms04_011(win32_bind) > exploit
```



```
[*] Starting Bind Handler.  
[*] Sending 8 DCE request fragments...  
[*] Sending the final DCE fragment  
[*] Got connection from 192.168.40.2:4444
```

```
Microsoft Windows 2000 [Version 5.00.2195]  
(C) Copyright 1985-1999 Microsoft Corp.
```

```
C:\WINNT\system32>  
C:\WINNT\system32>hostname  
hostname  
mrashw2k
```

The *hostname* command was executed on the target OS after Metasploit successfully compromised the Windows 2000 host. Now we let the modified Snort sid 2511 loose on the inline Snort system and re-run the attack sequence above. Instead of the nice greeting of the command shell, we are unable to compromise the remote system and are left with the following output from Metasploit:

```
msf lsass_ms04_011(win32_bind) > exploit  
[*] Starting Bind Handler.  
[*] Sending 8 DCE request fragments...  
[*] Sending the final DCE fragment  
[*] Exiting Bind Handler.
```

As always, deploying any device that alters Application-layer data must be done with the greatest care in order to avoid potentially serious unintended consequences. The above strategy for altering packet data in response to detecting the LSASS exploit assumes that the Snort rule sid 2511 has extremely low rates of false positives. *This must be tested for your particular network before deployment.*

## Checklist

- Keep on the lookout for the official 2.3.0 release of the Snort IDS, which includes inline IPS functionality provided by Snort\_inline.
- Continually vet the false positives generated by any IDS deployed on your network before deploying an inline IPS. Understand under what conditions false positives are generated, and determine which signature rules may be safely deployed in an inline IPS.
- The best way to secure software is to not run vulnerable software in the first place. Know all of the software deployed on your network, watch for security updates from each of the respective vendors, and patch, patch, patch.
- Application-layer data replacement can be used as an alternate way of implementing intrusion prevention, but requires extremely detailed knowledge of exactly which bytes may be tweaked as applications communicate across the network.

## Summary

This chapter discussed the concept of intrusion prevention from the unorthodox standpoint of tweaking Application-layer bytes instead of altering Data Link-, Network-, or Transport-layer connectivity in response to monitoring a network-based attack. Two pieces of software—Snort\_inline and a patch to the Netfilter string match extension—have been presented that can inspect and replace specific sequences of bytes in the Application-layer of IP packets.

Snort\_inline is the open source community's answer to the need for an inline IPS, and is implemented as a patch to the Snort IDS. Snort\_inline changes the way packets are processed on a Linux system, from using the standard libpcap packet capture library to using the libipq library developed as a part of the Netfilter Project. Libipq allows packets to be queued from kernel memory into the process address space associated with a userspace application, which then may process packets for suspicious characteristics and make a verdict about whether the packet should be allowed to continue through the machine. Snort\_inline is meant to be deployed on a Linux system that has been configured to form an Ethernet bridge between two Ethernet segments, and to monitor all traffic passed between the two segments. No packet is allowed to pass through the Snort\_inline system without being passed through the Snort detection engine and inspected against the Snort signature ruleset. Snort\_inline was not immediately accepted into the main Snort code base, but as of November 2004, Snort\_inline has been integrated and will be available in version 2.3.0 of the Snort IDS. As of this writing, Release Candidate 1 of Snort-2.3.0 is available for download from [www.snort.org](http://www.snort.org).

The Netfilter string match extension endows the Netfilter firewall running in the Linux kernel with the ability to search the application portion of IP packets for strings specified in ASCII or hexadecimal codes. A patch implemented against this extension allows matching strings to be replaced with strings specified by the administrator in a manner similar to the *replace* keyword provided by Snort\_inline. There are tradeoffs between using the functionality provided by this patch and deploying Snort\_inline. The data replacement patch

accomplishes both detection and data replacement entirely within kernel space and thus can outperform `Snort_inline` for basic attacks because packets never have to be copied across the kernel/user memory boundary. However, there are many attacks that cannot be detected with simple string matching, and require the extremely complete and functional rules language provide by `Snort` to be detected with low rates of false positives. For such attacks, running the string match extension is not sufficient to provide any protection. Also, standard IDS evasion techniques such as packet fragmentation and TCP session splicing can defeat the string match extension, so it is mainly useful for those attacks that do not make use of such techniques.

Implementing Application-layer nullification of attacks instead of the more brute-force methods of tearing down TCP sessions with TCP resets or by implementing firewall or router ACLs in response to an attack, can provide a measure of stealth to an inline IPS due to the fact that attackers cannot be sure that many exploits will compromise security on the first try. In addition, Application-layer error codes can legitimately be generated in response to an altered attack, which may help convince the attacker that other lower-hanging fruit may be easier to chase.

This chapter closed with three separate exploit examples: a DNS server buffer overflow, a path disclosure attack against Microsoft's IIS Web server with Frontpage extensions, and an automated exploit provided by the Metasploit Project for Microsoft's LSASS service. In each case, an inline IPS used the data replacement technique to stop the exploits in their tracks.

## Solutions Fast Track

### `Snort_inline`

- ☑ Provides strong intrusion prevention functionality to the `Snort` IDS. `Snort` is now not only an IDS, but also an IPS. `Snort_inline` is most effective when deployed on a Linux system that is configured as a bridge between two Ethernet segments. All packets flowing through such a system are sent through the `Snort` detection engine before

they are allowed to exit the egress interface of the bridge.

- ☑ Adds three new actions—*drop*, *reject*, and *sdrop*—to the standard Snort rule actions. Snort\_inline also adds the capability of replacing bytes at the Application layer with new bytes specified within Snort rules. This allows attacks to be altered in subtle ways to be rendered benign as they are sent across a network to a target system.
- ☑ Snort\_inline has been integrated with the main Snort code base. As of this writing, the first release candidate for Snort-2.3.0 contains Snort\_inline capabilities. Snort-2.3.0RC1 is available for download from [www.snort.org](http://www.snort.org).

## Netfilter Data Replacement Patch

- ☑ The Netfilter string match extension allows a Linux firewall to search for specific strings specified in normal ASCII or as hexadecimal codes within Application-layer packet data.
- ☑ The data replacement patch (available from [www.cipherdyne.org/fwsnort](http://www.cipherdyne.org/fwsnort)) emulates the *replace* keyword implemented by Snort\_inline. It is slightly more flexible, however, since strings of lesser or equal length may be specified.
- ☑ The Netfilter string match extension performs simple string match operations against packets and can be evaded by any number of IDS evasion techniques. However, for attacks that make no use of these techniques, the string match extension can provide an effective tool against such attacks because packet data remains completely within kernel memory and hence is quite fast.

## Application Layer Byte Replacement

- ☑ Attacks are rendered harmless through modification of Application-layer data. For example, if a target system is vulnerable to an exploit that contains the bytes *abcde*, an inline IPS can replace these bytes with the harmless bytes *aaaaa* and allow the now harmless attack to continue on to the target system.

- ☑ Implementing network intrusion prevention at the Application layer allows application error codes to be preserved.
- ☑ Altering Application-layer data requires detailed knowledge of the shape of the data used by the application as well as the Application-layer protocol, so as to not introduce high levels of instability into the network. For example, Web page text is different from HTTP header information even though they both appear in the application portion of TCP packets as they traverse a network.
- ☑ As with all IPS' and techniques, false positives remain a problem because the detection mechanisms of an IPS is derived from those implemented in IDS'.

## Links to Sites

- ☑ [www.snort.org](http://www.snort.org) Home of everything Snort related.
- ☑ <http://snort-inline.sourceforge.net> The original home of the Snort\_inline project.
- ☑ [www.netfilter.org](http://www.netfilter.org) The main site for Netfilter, including all of the latest extensions and versions of the iptables userspace binary.
- ☑ [www.honeynet.org](http://www.honeynet.org) Home of the Honeynet Project. This project probably makes more use of Snort running in inline mode than any other project.
- ☑ [www.tcpdump.org](http://www.tcpdump.org) The main site for tcpdump (which was used to create all packet traces in this chapter) and of libpcap on which many IDS' depend.
- ☑ [www.packetfactory.net/libnet](http://www.packetfactory.net/libnet) Home of the libnet packet creation library. Snort\_inline depends on this library.
- ☑ <http://ebtables.sourceforge.net> Home of the ebtables project that allows the Linux kernel to apply Netfilter filtering rules to interfaces that form an Ethernet bridge.
- ☑ [www.eeye.com](http://www.eeye.com) Home of eEye Digital Security, which has discovered and announced some extremely cutting edge security

flaws in widely used software including the LSASS vulnerability in Windows systems.

- ☑ [www.metasploit.com](http://www.metasploit.com) The main site for the Metasploit exploit framework. Everyone interested in computer security should familiarize themselves with this tool.
- ☑ [www.cipherdyne.org](http://www.cipherdyne.org) The main site for FWSnort, the Netfilter data replacement patch, and Passive Search and Detection (PSAD) (covered in Chapter 8).
- ☑ [www.monkey.org/~dugsong/fragroute/](http://www.monkey.org/~dugsong/fragroute/) Home of the packet-fragmenting IDS evasion tool fragroute.
- ☑ [www.insecure.org/stf/secnet\\_ids/secnet\\_ids.html](http://www.insecure.org/stf/secnet_ids/secnet_ids.html) Excellent reference for various IDS evasion techniques.
- ☑ [www.lurhq.com/sasser.html](http://www.lurhq.com/sasser.html) A good analysis of the Sasser worm.
- ☑ [www.synacklabs.net/projects/packetp/](http://www.synacklabs.net/projects/packetp/) Home of the Packet Purgatory library, which facilitates arbitrary header and data tweaking of IP packets.
- ☑ [www.cs.utexas.edu/users/moore/best-ideas/string-searching/](http://www.cs.utexas.edu/users/moore/best-ideas/string-searching/) Home of the Boyer-Moore Fast String Searching Algorithm.
- ☑ [www.sophos.com/virusinfo/articles/howbofrawork.html](http://www.sophos.com/virusinfo/articles/howbofrawork.html) A good description of how the Bofra family of worms compromises and spreads to vulnerable Windows systems.

## Mailing Lists

- ☑ [focus-ids](#) (see ) A good list for lots of IDS and IPS-related threads. The original authors of the Snort IDS (Marty Roesch) and of the Dragon IDS (Ron Gula) are relatively frequent posters to this list.
- ☑ [Various Snort Lists](#) See [snort-users](#), [snort-sigs](#), and [snort-devel](#) for questions about Snort such as how to write Snort rules and help with detailed development questions.

- ☑ Netfilter Lists See netfilter-users and netfilter-devel for in-depth technical discussions about Netfilter. Some hardcore kernel developer-types hang out on the netfilter-devel list.

## 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](http://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:** Will Snort\_inline ever be included within the main Snort IDS instead of being distributed as a code patch?

**A:** Yes. The Snort\_inline code has been added into the main Snort code base, and will be officially included within the Snort-2.3 release.

**Q:** What is the main advantage to preventing an attack via Application-layer data modification?

**A:** Altering Application-layer data can provide a means of thwarting attacks in the least detectable way possible. Any technique implemented by a network IPS that actively responds at the Data Link, Network, or Transport layers to an attack, leaves a heavier footprint on network connectivity than what is possible with subtle Application-layer byte tweaking. Applications themselves can continue to return valid error codes for attacks that have been nullified by an inline IPS.

**Q:** Do false positives become a more critical issue for network IPS' that implement Application-layer data modification than for those that implement traditional responses via TCP resets or adding blocking rules to firewall or router ACL's?

**A:** Altering Application-layer data can potentially be more damaging to the integrity of systems running on a network than simply tearing down TCP sessions or adding blocking rules to firewall policies. For example, suppose a poorly written Web application uses unencrypted cookies for communi-

cating a user ID to a Web server over normal (unencrypted) HTTP. Further, suppose that an inline device mistakenly detects an attack coming from a client browser that is making use of this application, and in the process of attempting to nullify the attack proceeds to change the user ID from the real user to that of a completely different user. This by itself may or may not cause a huge problem until money is on the line and the transactions of one user become confused with another user. Altering Application-layer data can introduce serious integrity problems into the applications running on a network, if implemented without detailed knowledge of the exact effects that altering such data can cause. In addition, since network IPS' are fundamentally built on detection mechanisms borrowed from network IDS', if there is no reliable means of detecting an attack with low rates of false positives, it is unwise to deploy any response mechanism. There is no substitute for rigorous and thorough testing. The best way to maintain security is to not run vulnerable applications; however, if this is not possible, altering Application-layer data associated with specific attacks can be an effective method of increasing security.

- Q:** Does inline data replacement cause a large amount of overhead vs. other forms of active response implemented by an inline device?
- A:** No. If having an inline device is acceptable, the overhead introduced by replacing sequences of Application-layer bytes and recalculating checksums is minimal. The method of sending packets into the detection algorithms (e.g., copying from kernel memory into user space vs. keeping packets within kernel memory) and the specific detection algorithms themselves accrue a much larger performance penalty.
- Q:** I'm running a 2.4 series Linux kernel and I've compiled an Ethernet bridging support, but I can't seem to use Netfilter to filter packets exchanged between interfaces that participate in a bridge. What's wrong?
- A:** You need to apply the ebttables patch available at <http://ebtables.sourceforge.net>. A stock Linux 2.4 series kernel can act as an Ethernet bridge or as a firewall, but cannot act as both at the same time on interfaces which are configured to form a bridge without the ebttables patch.

