4 Lifecycle of a Vulnerability

This chapter walks you through the process of providing Intrusion Detection System (IDS) coverage for a security vulnerability from start to finish, using practical examples and highlighting popular and useful open source tools. After the process is introduced, this chapter focuses on how to write Snort signatures for more complex vulnerabilities by using features such as flowbits, Perl-Compatible Regular Expressions (PCRE), and the relatively new shared object rules, which allow Snort to leverage all the power of the C programming language.

A Vulnerability Is Born

The vast majority of new software vulnerabilities are announced on public forums, such as the SecurityFocus Bugtraq mailing list (www.securityfocus.com/archive/1); official government sites, such as US-CERT (www.us-cert.gov/) or NIST's CVE database (http://cve.mitre.org/); or directly on vendor Web sites or mailing lists, such as Microsoft's monthly security bulletin releases. These sites tend to cross-reference each other and typically sync vulnerability references and information within a matter of a day or two. Thus, for all but the newest vulnerability announcements, just about any major information source suffices.

In many cases, the details associated with a vulnerability release are vague; any IDS analyst who's been in the field for more than a few months is familiar with statements like, "Sending malformed packets cause the application to crash, which creates a denial

of service condition." This lack of detail is usually the result of a conscious decision by the vendor to make one last stab at security through obscurity, essentially hoping that, by not releasing the exact mechanism of exploitation, it can save its users from having their software hijacked because no one takes the time to figure out how to use the newly announced security hole.

Although most users of a piece of affected software generally find this practice laudable, refusing to release technical details about a vulnerability usually only serves to annoy IDS analysts and challenge malicious adversaries. For the IDS analyst, the problem is obvious: Without details of how a successful exploit can be accomplished, it's impossible to determine what that exploit looks like coming across the network; thus, it is impossible to create an IDS signature to detect such behavior. For others, a newly announced vulnerability can be anything from a potential profit center (for example, for criminal crackers using it to further their nefarious goals) to a fascinating puzzle (see the excellent XKCD comic "Nerd Sniping" at http://xkcd.com/356/).

Given these pressures, a large number of vulnerabilities that are initially disclosed without details are soon followed by proof-of-concept code or an actual working exploit. (The difference between the two being that the latter actually injects shellcode into the vulnerable piece of software, which causes it to spawn a shell, open a TCP connection, or do something else that the cracker finds useful; the former simply crashes the application.) Generally speaking, the wider the installed base of the software, the more high-profile the vulnerability and the less complex the software, the more likely it is that a working exploit emerges soon after the initial announcement. For example, a buffer overflow in Microsoft Exchange almost certainly has a working exploit released within a day or less of the initial announcement; conversely, a format string bug in an obscure piece of proprietary software might never have one developed.

Luckily for IDS analysts, especially those without ties into the network security underground, several good publicly accessible Web sites publish working exploits, sometimes even before a vulnerability announcement is made. Sites such as Milw0rm, Packet Storm, and Metasploit serve as repositories for thousands of publicly available exploits, along with detailed information on how to break software, commonly used network security tools, and so on, all of which are extremely valuable to IDS analysts.

FLASHGET VULNERABILITY

Let's start out with a relatively simple buffer overflow exploit written in Perl, which is a language that about any Linux or BSD user has installed by default and that can be easily installed on Windows (www.milw0rm.com/exploits/6256). By selecting something that's easily readable for anyone who knows Perl and that takes no effort to get running, you

can focus on the necessary tools to do the analysis and get into the details about the actual analysis later. Note that the console listings are straight copy and pastes from the machines used to actually run this exploit—you see exactly what I saw while I ran it.

The vulnerable software is FlashGet, which is a popular download manager for numerous protocols, such as HTTP, FTP, and BitTorrent. As originally disclosed on August 13, 2008, FlashGet is prone to a buffer overflow when parsing the FTP command `PWD`, and successful attacks can lead to remote code execution (for example, an attacker can run whatever it wants on the victim machine).

First, download the exploit to a machine that can have clients connect to it on TCP port 21, the standard FTP port, and verify that you can run it properly. (Note that if you simply use a tool like wget to fetch the file, it comes down as HTML that you need to clean up; it's simpler to copy and paste the exploit manually into a file.)

```
alex@gateway: ~$ sudo perl flashget-overflow.pl
usage: flashget-overflow.pl [1,2,3]
  1 -> Windows XP SP1
  2 -> Windows XP SP2
  3 -> Windows XP SP3
```

This result isn't too surprising. It tells you that you need to choose the operating system (OS) that you want to exploit. This often becomes a choice between different service packs of Windows XP, because each service pack includes different security mechanisms and each has important kernel functions that the shellcode uses for actually executing code in different memory locations. If you have a vulnerable copy of this software, try running the exploit first for the incorrect XP service pack and then the right one—you see that the first attack fails and the second works, popping up calc.exe on the compromised machine.

Because there isn't actually a vulnerable client on the receiving end of the exploit—as an IDS analyst, it's often more hassle than it's worth to set up the actual vulnerable software when recording an exploit—I arbitrarily choose to do SP2, because that's the most common version of XP. Running the command again with 2 as the argument, no prompt returned. The program simply sat and waited. To verify that it is actually listening properly, I used another window on the machine and ran `netstat`, which is a standard Linux/BSD command that displays open sockets:

```
alex@gateway: ~$ netstat -tna
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address     Foreign Address  (state)
tcp        0      0  *.21              *.*              LISTEN
tcp        0    320  68.55.r.s       71.163.x.y     ESTABLISHED
tcp        0      0  *.22              *.*              LISTEN
```

The first line of actual data confirms that there is something listening to port 21. Because I am not running an FTP server on this machine, it's clearly the exploit.

COLLECTING A SAMPLE PACKET CAPTURE

Now that you are ready to test the exploit, let's pause for a moment to ensure that you can actually record the traffic that it sends across the wire; after all, an exploit is useless to an IDS analyst if he can't see what it actually looks like on the network. Although some people can directly read exploit code in Perl, C, Python, or whatever other language it's written in, to determine precisely what the network traffic looks like, this is only useful for the most basic of exploits. Even a veteran IDS analyst can easily miss something if he attempts to rely solely on reading the exploit. Having an actual packet capture in hand prevents human error and helps immensely when it comes time to test the IDS' new detection capability.

Two primary options exist for capturing a copy of an exploit as it crosses the wire: tcpdump and Wireshark (which is the packet-analysis tool formerly known as Ethereal). Because Wireshark is primarily useful when you capture packets on a local system and have access to a graphical user interface [GUI]—and because it's trivial to figure out how to capture packets properly with that interface—I use tcpdump for this exploit, which is handy on any system where an IDS analyst has command-line access.

With both tools, the user capturing the packets needs to have either root access or has been granted permissions to set the interface he wants to capture on into promiscuous mode. Although this feature sounds like some kind of bad joke thought up by a lonely programmer, its name is actually reasonable when you consider what it means: Promiscuous mode sends all packets that a given interface sees up the network stack to the application listening on it, instead of only sending packets destined specifically for that interface. Although promiscuous mode is usually not particularly useful when an IDS analyst attempts to record a specific attack, it has distinct network-watching benefits in other situations and is required for packet-sniffing tools to properly function.

With that said, here's the command line I used to capture the exploit and the initial output from tcpdump letting me know that it was recording:

```
alex@home: ~/pcaps$ sudo tcpdump -n -i eth0 -s0 -w flashget.pcap host
68.55.225.129 and port 21

tcpdump: listening on eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
```

Let's step through this piece by piece to ensure that you understand what you're looking at. The first option, `-n`, tells tcpdump not to do Domain Name System (DNS) lookups on the Internet Protocol (IP) addresses it outputs. You're free to leave this out if DNS helps you, but most of the time, an IDS analyst has no need for DNS when capturing

a sample exploit. Also, any kind of issues with the DNS setup might slow tcpdump to the point that it drops packets while waiting for the DNS lookup to finish.

The second option, `-i eth0`, specifies the network interface to use for recording packets. On Linux systems, this often defaults to `eth0`, which is the standard name for the first network interface on the system. (Although on some machines, it might default to `lo0`, the loopback operator, which doesn't help when you're sending an exploit across the network.) If you've got multiple interfaces on the system, or you're running a BSD system (where network cards are named after the driver used to power them, such as `x10` for 3Com, `nve0` for nVidia, and so on), it's worth specifying this explicitly. If nothing else, it's a habit worth getting into, in case you end up recording traffic on an unfamiliar system.

The third option, `-s0`, specifies the amount of data per packet that you want to capture, which the tcpdump manual calls the *snaplen*. A value of zero obviously doesn't mean that you want to record no data; instead, it specifies that tcpdump needs to capture the entire packet, regardless of its size. (OpenBSD users: `-s0` is invalid on that platform, so use an arbitrary large value, such as `-s2000`, to achieve the same effect.) It's particularly important to set this option, because if you don't, you end up with errors such as `[Packet size limited during capture]` in Wireshark or `IP Len field is 98 bytes bigger than captured length (ip.len: 180, cap.len: 82)` in Snort. These are signs that the data you want is simply not there.

The fourth option, `-w flashget.pcap`, is as simple as it gets: It specifies the file to which you want to write the packets, which generally has the extension .pcap (or, for people more accustomed to the old-school Windows eight/three notation, .cap). As obvious as this option might seem, it's worth highlighting because of the number of people who might simply redirect the output of tcpdump into a file, which produces an ASCII file with lines like this:

```
20:20:39.273666 IP 192.168.1.4 > 64.233.161.99: ICMP echo request, id
53591, seq 1, length 64

20:20:39.283251 IP 64.233.161.99 > 192.168.1.4: ICMP echo reply, id
53591, seq 1, length 64
```

A file like this is completely useless to a program like Wireshark or Snort, because it doesn't actually contain the packet's contents. By having tcpdump record directly to a file, however, a binary file is created with an exact copy of every single byte transmitted across the network. This enables analysis tools to dig as deeply as they need into the packet data.

The final piece in the sample command line, `host 68.55.225.129 and port 21`, isn't necessary and isn't something that's within the scope of this chapter. However, a brief example or two is illustrative. You are looking at a Berkeley Packet Filter (BPF) filter, which is

a virtual interface directly into the structure of packets. BPF filters provide a straightforward syntax to specify what portion of a packet you're interested in examining and to filter packets out. Simple examples include `host 192.168.1.1`, `port 25` or a combination of the two, such as `host 10.1.1.10 and port 80`. More complex options that allow for fine-grained control, such as selecting the type of Internet Control Message Protocol (ICMP) message, specific IP fragmentation flags, and so on, can be found in the tcpdump manual page.

Although an analyst can create a perfectly valid packet capture without using a BPF filter, using one makes it easier to identify the packets that actually contain an exploit or that you're interested in analyzing if capturing a sample of live traffic on an unknown entity. For example, capturing a simple visit to a site, such as www.google.com, without a BPF filter could easily collect ARP traffic, the DNS lookup for that address, and so on. If you're on a busy system that's doing a lot on the network, the capture might accidentally pick up things like a Secure Shell (SSH) session running in another window, an HTTP file transfer going on in the background, and so on. It's trivial to end up with tens to thousands of unwanted packets in your capture if you're not using a good BPF filter. This does nothing but frustrate you and any analysts with which you want to share the packet capture.

Now that that's out of the way, let's get to the interesting part: actually running the exploit. I used the simple built-in command-line FTP client on my Ubuntu Linux system to run the client side of things:

```
alex@home: ~$ ftp 68.55.225.129
Connected to 68.55.225.129.
220 Hello ;)
Name (68.55.225.129:alex):
331 pwd please
Password:
230 OK
Remote system type is CWD.
ftp> pwd
257 "AAAAAAAAAAA...<lots of nonprintable characters>" is current
directory.
ftp>
```

Switching over to the window with tcpdump running, I hit Control-C to stop the capture and got the following output:

```
24 packets captured
24 packets received by filter
0 packets dropped by kernel
```

This output indicates that the capture succeeded. If, for some reason, the number of packets captured is larger than the number received by the filter, they might have been lost before they could be run through the filter and recorded, if applicable, or more importantly, if the kernel had dropped packets. This indicates that there are missing packets and the data is coming in faster than the system can process it. This means that either there's too much traffic on the wire or something else is hogging a huge amount of system resources. Dropping or missing packets occurs when the kernel-level buffer for the holding packets received by the network card (before they're processed by tcpdump or similar application) does not have the resources necessary to keep up.

Now that the exploit is recorded, it's time to open it in Wireshark. Certainly, you can use tcpdump's `-r` option to play it back or use Snort's equivalent option to dump the data. However, Wireshark includes several excellent analytical tools and features that make an IDS analyst's life easier, so unless there's a good reason not to do so, always use Wireshark when you examine your packet captures. Doing so on my system, Wireshark shows the results (see Figure 4-1).

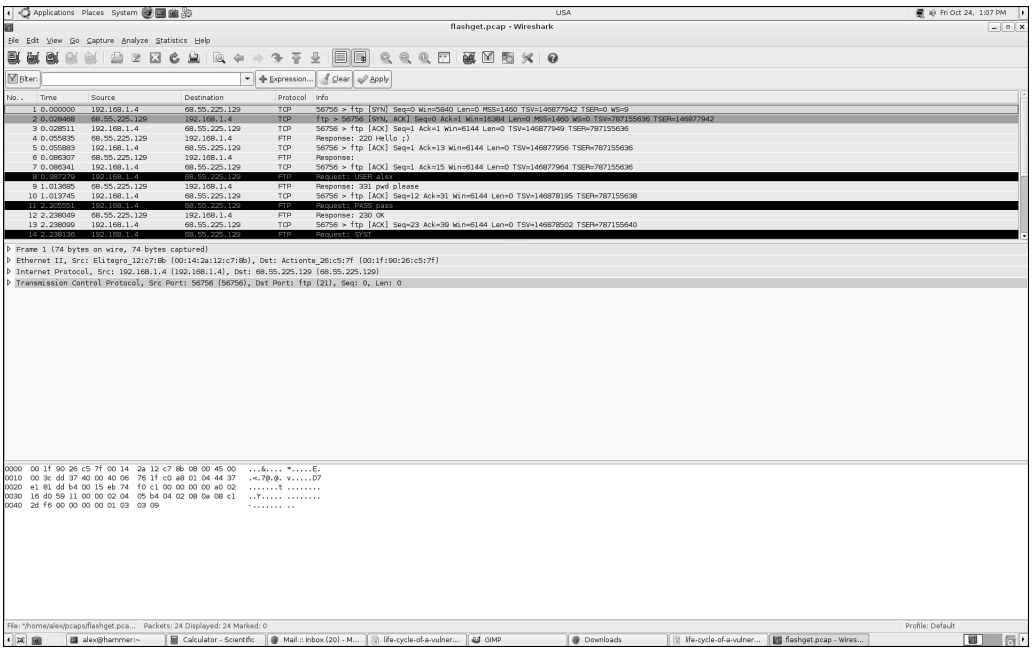

Figure 4-1 Wireshark results

One item that immediately sticks out is that several packets are highlighted in black with red letters. Looking at the first such packet more closely, you can see that Wireshark highlighted the TCP line in red. Expanding the drop-down menu reveals that the checksum line is also highlighted in red (see Figure 4-2). Reading it closely, you see the following error:

```
Checksum: 0xe796 [incorrect, should be 0x0635 (maybe caused by "TCP checksum
offload"?)]
```

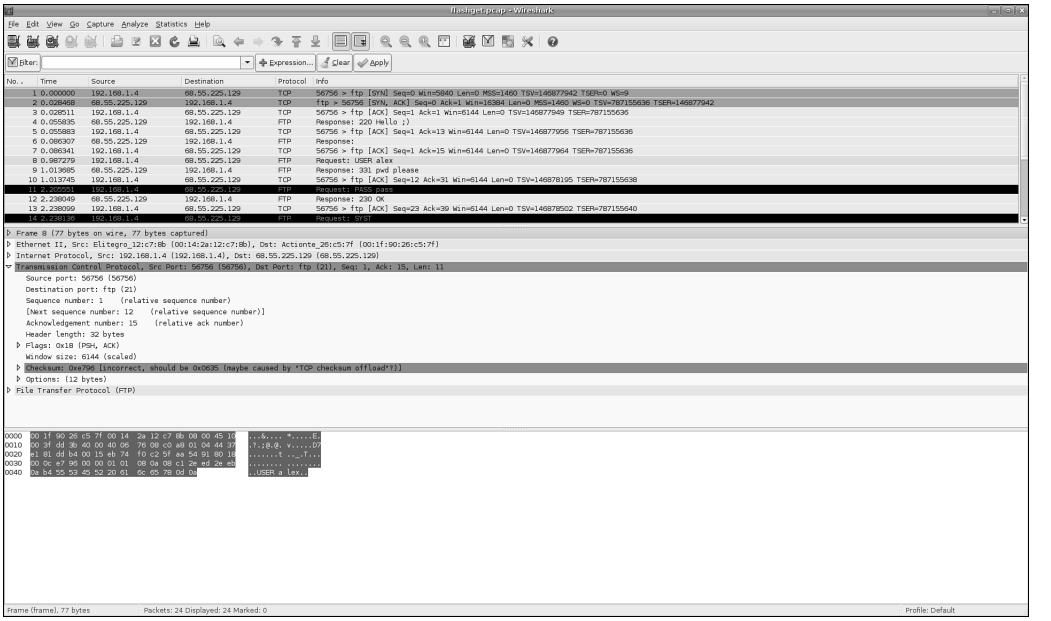

Figure 4-2 The TCP and checksum lines are highlighted.

This common problem occurs when packets are recorded on the same system from which they're being sent. Many Ethernet cards can calculate checksums for packets they're sending out directly on the card itself, which saves the CPU of the sending machine the overhead of calculating the checksum. As noted in the error message, this is known as *TCP checksum offloading*. When a packet destined for an external host is created by an application, it travels through the network stack, having items like TCP and IP headers added along the way. When TCP checksum offloading is enabled, the OS simply inserts a fake checksum into the header (whose data is essentially random for this chapter's purpose), knowing that the Ethernet card calculates the actual value and places it at the appropriate location in the packet before sending it to the network. Because the

kernel-level packet buffer gets its copy of the packet before the real checksum is calculated, any capture tool reading out of this buffer gets the invalid checksums that the kernel generates.

Although this is a good system design and use of resources, it's frustrating for IDS analysts, because, by default, tools like Snort skip packets with bad checksums when they read them in. (This is logical, because a host receiving a packet with a bad checksum in the real world simply discards it.) The good news is that Brian Caswell wrote a free Perl script that fixes a packet's checksums (www.shmoo.com/~bmc/software/random/fix-cksum.pl). (You need the free Perl modules Net::PCAP and NetPacket, which are available from CPAN, to run this tool.) Running it is as simple as

```
alex@home: ~/pcaps$ perl fix-cksum.pl flashget.pcap flashget.pcap.fixed
```

Now, the errors are gone in Wireshark, and you can get to the interesting part: actually analyzing the packets and creating a Snort rule based on the nature of the vulnerability.

PACKET ANALYSIS AND SIGNATURE-WRITING

By scrolling through the packets in flashget.pcap, it's obvious which packet contains the exploit (see Figure 4-3).

Note that Wireshark shows the command line as [truncated], which is not surprising, because it's 1332 bytes long, obviously much too long to display as a single line onscreen. Having an exploit packet generate errors in Wireshark is common, because the malicious packets are typically malformed to achieve their intended effect. In fact, Wireshark has been vulnerable over the years to numerous exploits that might be triggered by reading back a malicious packet capture. Of course, it's not worth discarding it as an analytical tool just because of these errors; it often correctly renders 99.99 percent of a packet capture. As an IDS analyst, keep in mind that an error in Wireshark doesn't mean you've done something wrong.

At this point, the question that must be answered is simple: What makes a malicious server response distinct from a legitimate one? Given that the vulnerability is a buffer overflow—in which data is copied into a fixed-size buffer in memory without ensuring that the data is actually the size of, or smaller than, the buffer—the answer is: The size of the response is going to be problematic.

This, of course, leads to a follow-up question: Just how big of a response should the Snort signature look for? The answer is deceptively simple: exactly the number of bytes necessary in order to overflow the vulnerable buffer in FlashGet. To get that number, an analyst has several options: relying on published information about the vulnerability to tell him the size of the buffer in question; directly testing the vulnerable software to determine how much data must be sent to cause a crash; and using a binary analysis

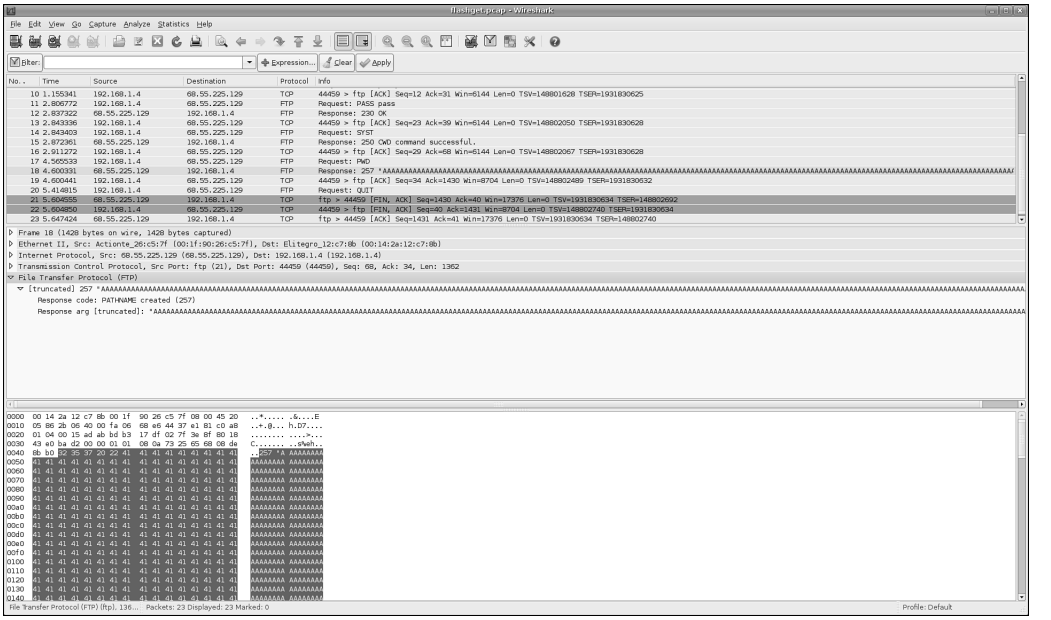

Figure 4-3 The packet containing the exploit

tool, like WinDBG or IDA Pro, to locate the vulnerable buffer in the program and determine its size by examining a disassembly of the program.

As an IDS analyst, chances are good that you're often going to be short on time, either because you've got a ton of analysis to do or because you're dealing with a newly released vulnerability that you must immediately create detection for. Given these constraints, when writing a Snort signature, an intelligent strategy is to start with the easiest possible methods and move on to more complex analysis later, if necessary. In this case, because a public exploit exists, the first thing to do is to check for publicly available information about the nature of the vulnerability. Because the Common Vulnerabilities and Exposures (CVE) database usually has the most comprehensive list of links to known information about any given vulnerability, it is a good place to start.

For this vulnerability, the URL to the appropriate CVE page is http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-4321. Digging through the available links, you quickly note that www.securityfocus.com/bid/30685/exploit has three distinct exploits available for this vulnerability: two in Perl and one in Python. Because you want to ensure that your signature detects all available exploits, go ahead and examine all three.

As it turns out, the last exploit available is a copy of the Milw0rm exploit. This makes things easier, because now there are only two more exploits to capture samples of in action.

After you run through the previous exercise to get samples of these exploits, it becomes immediately obvious that far fewer than the 1332 bytes worth of payload in the first exploit are actually necessary. In fact, the Python version of the exploit has only 356 bytes worth of payload. With no other information available in any of the public advisories regarding the size of the buffer required to perform the exploit, 356 is a reasonable size value to check for in the signature; it is a value that has been arrived at with a minimal amount of time spent examining the vulnerability.

Based on what the previous chapter covered (or your previous knowledge of Snort, of course) and what is known of the vulnerability and FTP, the basics of a signature should be fairly obvious. Malicious packets must be directed at TCP port 21 (used for FTP commands) and must come from a TCP server. All malicious packets contain the string 257, which is the response code sent along with a reply to a request for PWD, followed by at least 356 bytes that are not a newline (as a newline character terminates an FTP command). This makes for the following Snort signature:

```
alert tcp $EXTERNAL_NET 21 -> $HOME_NET any (msg:"FTP Flashget PWD response
buffer overflow attempt"; flow:established,to_client; content:"257"; nocase;
pcre:"/^257[^\n]{356,}/smi"; classtype:attempted-admin; reference:bugtraq,30685;
reference:cve,2008-4321;)
```

Before moving this signature to any sort of production system, however, you must first test it. The simplest way to do this is to put it into the local.rules file on a system where you've unpacked and compiled a current version of Snort. (Note that you don't need to have installed Snort anywhere; so long as all the paths in the snort.conf file are set appropriately, you can run any version of Snort out of any directory on a system, which is handy if you don't have administrative access on the machine on which you want to do the testing.) From there, manually call Snort with a command line similar to this:

```
alex@home: ~/downloads/snort-2.8.3$ src/snort -c etc/snort.conf -q -A
cmg -r ~/pcaps/flashget-python.pcap
```

Let's step through the arguments to ensure that you know exactly what's happening. First, note that the command is actually `src/snort`, and not just `snort`. This is because it's being executed out of the directory it was unpacked into, and you are specifying the

direct path to the Snort binary compiled in there instead of letting the system search through its path to find a binary matching the name "snort." This enables multiple versions of Snort to be running on the same system, because you simply specify which Snort you want to run each time you call it. The next option, `-c etc/snort.conf`, specifies the configuration file that Snort should use. Again, it's key that this is manually specified, not just so you know exactly what file it's using, but because directly specifying it like this allows for different versions of Snort to nicely coexist.

The second and third options, `-q` and `-A cmg`, aren't strictly necessary for analysis, but they make dealing with Snort's output much easier. The `-q` flag tells Snort to be quiet (for example, to not bother outputting its initiation messages, protocol statistics, and so on), which is usually just unnecessary crud on your screen when your goal is verifying a signature. The `-A cmg` option tells Snort to output all alerts to standard output (such as your screen) so that you don't have to dig through an alert file to see if the signature you're testing just fired. Finally, the `-r` flag tells Snort that it should read an existing packet capture instead of pulling packets from the network.

With that done, the output from an initial run of Snort with the previous rule pasted in is as follows:

```
ERROR: etc/../rules/local.rules(1) => Each rule must contain a Rule-sid
Fatal Error, Quitting..
```

Although this might seem like a simple error to fix—adding `sid:99999;` to the end of the signature does the trick—it's worth pointing out, because it gives us a chance to briefly discuss Snort IDs (SIDs). Snort uses SIDs to keep track of rules, and they are included in Snort's output so users can tell what signature generated a given alert (and thus how legitimate it is and what service might have been exploited). What is less obvious is the debugging nightmare that can arise if you accidentally use an identical SID for two different rules. This is a mistake that most longtime IDS analysts who regularly use Snort have made. Because Snort silently overwrites the first rule with the second when it encounters an identical SID, it's possible to spend hours trying to figure out why a perfectly good signature isn't firing if you've made this mistake. Because no major public signature sets currently use the SID space between 10000 and 99999, SIDs in that range are easy to use for testing purposes.

After modifying the rule to include a SID, Snort's output is much happier:

```
10/24-14:23:26.067596  [**] [1:99999:0] FTP Flashget PWD response
buffer overflow attempt [**] [Classification: Attempted Administrator
Privilege Gain] [Priority: 1] {TCP} 68.55.225.129:21 ->
192.168.1.4:57564
```

```
10/24-14:23:26.067596 0:1F:90:26:C5:7F -> 0:14:2A:12:C7:8B type:0x800
len:0x1AD

68.55.225.129:21 -> 192.168.1.4:57564 TCP TTL:250 TOS:0x20 ID:47844
IpLen:20 DgmLen:415 DF

***AP*** Seq: 0xA9E10DAB  Ack: 0x553E2909  Win: 0x43E0  TcpLen: 32

TCP Options (3) => NOP NOP TS: 104617918 149029958

32 35 37 20 22 41 41 41 41 41 41 41 41 41 41 41  257 "AAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
41 2F 22 20 69 73 20 63 75 72 72 65 6E 74 20 64  A/" is current d
69 72 65 63 74 6F 72 79 2E 0D 0A                 irectory...

=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=
```

This alert contains a wealth of information, most of which can be safely ignored during the regular course of packet analysis. The most important piece is in the first line: `[1:99999:0]` tells you that SID 99999 generated the alert, meaning that Snort behaved as expected; the leading `1` specifies that the alert came from generator ID 1 (reserved for plain Snort rules, as opposed to 3 for shared object rules); and the trailing `0` tells you that the SID had no associated revision number. The other relevant piece is the payload, which is represented as hexadecimal digits on the left and ASCII characters on the right. It tells you that the alert was generated on the appropriate packet.

At this point, you have successfully written a Snort signature to detect this vulnerability. That said, the analysis process is only partially complete; there's still signature tuning to do, either immediately or in the future after the rule is deployed on a production system.

SIGNATURE TUNING

Signature tuning generally falls into two categories: detection tuning and performance tuning. Although the two categories are heavily intertwined—doing proper detection for some vulnerabilities can be highly performance-intensive and require IDS analysts to make best-guess tradeoffs between more accurate detection and better performance—it's worth discussing them separately so that the steps involved in each are fully outlined.

DETECTION TUNING

The process of detection tuning is better defined and more straightforward than performance tuning. Only two real pieces are involved in detection tuning: false negatives (when a signature fails to alert on a valid exploit) and false positives (when a signature alerts on valid and/or unrelated traffic).

Continuing with the example of FlashGet, assume that another exploit is released that has a payload of only 300 bytes long. Clearly, the previously signature written would have a false negative, because this new 300-byte exploit does not meet the minimum requirement of 356 bytes necessary to trigger an alert. A simple fix is to alter the rule to look for 300 or more bytes:

```
alert tcp $EXTERNAL_NET 21 -> $HOME_NET any (msg:"FTP Flashget PWD response
buffer overflow attempt"; flow:established,to_client; content:"257"; nocase;
pcre:"/^257[^\n]{300,}/smi"; classtype:attempted-admin; reference:bugtraq,30685;
reference:cve,2008-4321; sid:99999; rev:2;)
```

Faced with this new piece of information, many analysts might decide to drop the minimum size requirement even further—say, to 200 bytes—just to ensure that their systems were covered in case a newer, smaller exploit was released later. Unfortunately, this might lead to false positives when working on systems with multiple layers of nested directories, because a legitimate long path name might trigger an alert. If these types of false positives were rare—no more than one or two a day—were coming in IDS mode, and not fouling up the network in IPS mode, it is advisable to simply ignore them as they come in, because it's better to be safe than sorry. If they became common, however, it is worth revisiting the signature and at least applying some programming common sense

to the number chosen for the overflow size: 256 is often a reasonable number, because it's a common buffer size that C programmers use and a size associated with a large number of buffer overflows.

PERFORMANCE TUNING

Performance tuning is a considerably less intuitive business. First of all, few people are familiar enough with the internals of Snort or any other IDS to know what tweaks impact performance. A more important consideration, however, is the fact that there is no hard-and-fast set of rules that can be applied to signature writing to speed performance across all environments. Although some things can be done that always improve performance, the nature of the network traffic that an IDS examines so heavily influences its performance that it's impossible to predict performance without detailed knowledge of the traffic. Thus, as you attempt to tune an IDS signature for performance reasons, keep in mind that testing it with someone else's network traffic has limited value, if there is any at all.

In the signature discussed so far, I have intentionally left room for performance so that I can introduce some of the most common performance optimizations with it. As discussed in the previous chapter, the signature already has numerous optimizations in it, simply by virtue of the fact that it's looking only for packets coming from TCP port 21 outside your network to some location inside your network and that it's constrained to established TCP sessions, and from there, only to packets being sent to the TCP client. Again, although these might seem like trivial optimizations, being specific about the nature of the involved TCP flow is one of the most important things you can do to improve a signature's performance.

In terms of the actual content of the packet(s) in question, one of the first things that you need to look for when attempting to speed up a rule is whether you can restrict the amount of data that the signature wades through to find what it's looking for. This is particularly true if the IDS searches for any fixed strings that are small, because, in essence, the smaller the fixed string, the more iterations the IDS must make through the packet to attempt to match the string. Because the signature you are looking to speed up has only one string that it must match in the packet, specified by `content:"257"; nocase;`, it is particularly slow. However, you know that, as an FTP response code, `"257"` must come at the beginning of the packet. By adding the option `depth:3;` after the `nocase;` option, you tell Snort to only search through the first three bytes of the packet for the string in question. This suddenly reduces the number of necessary string match operations from hundreds (or possibly even thousands) to one.

To illustrate this point, take a brief detour into a discussion of one of Snort's lesser known features: rule profiling (available as a configuration option and fully documented in the README.PerfProfiling file within the doc/ directory of Snort). By simply adding config profile_rules to your snort.conf, you get basic rule-performance data to print at the end of each run to Snort. For example, here is the output from running the original signature without the `depth` clause against the Milw0rm exploit for FlashGet:

```
Rule Profile Statistics (all rules)
=========================================================
SID   Chcks Matches Alerts Microsecs  Avg/Check  Avg/Match Avg/Nonmatch
===   ====  ======= ======  =====      =========  ========= ===========
99999    2       2    1     563         281.8        0.5          0.0
```

The first and third columns (Num and GID, respectively) have been redacted to ease reading and properly display the the columns in print. They are just information on which signature the statistics are being reported. The second column in our print version, Checks, provides information about the number of signature-matching operations performed on the payload in question; each item, such as content and pcre, constitute a distinct check. (Often, the number of checks won't line up exactly to the number of rule operations and potentially matching packets, because Snort sometimes performs a given check more than once on a single packet. Also, issues such as reassembled packets out of a TCP stream often come into play.) The third column, Matches, is closely correlated: For each performed check, the operation can result in either a match (meaning that the check succeeded because the data being sought by the rule is present) or a nonmatch (failure). These two columns, in combination, provide a useful piece of information about a given signature. If the number of checks is considerably higher than the number of matches, some portion of the signature is being found in a large number of packets that are eventually determined to be uninteresting (because they don't contain all the criteria necessary for an exploit). Because this scenario causes Snort to do much unnecessary work, if at all possible, make the option that is matching frequently have more specific results in a faster signature; this cuts the number of packets that Snort evaluates as it looks for an exploit.

The four options grouped together (Microsecs, Avg/Check, Avg/Match, and Avg/Nonmatch) provide information about the amount of CPU time used while a given signature is processed, sliced into several different measures. (Note that CPU time can differ from wall-clock time if the system you're running the tests on is under heavy load and has many other processes competing for CPU time.) Although these options might seem like the most useful pieces of available information, a large number of factors limit their usefulness unless examined properly. For example, the number of microseconds spent evaluating a signature varies wildly between machines, because

different CPU speeds result in considerably faster or slower signature processing. Additionally, other uncontrollable factors, such as hard drive caching of a packet capture, the load placed on the system by other processes, potential memory swapping, and so on, can equally dramatically impact the speed of signature evaluation, even on a single system.

Given this potential for unreliable output, an IDS analyst can do two things to pull useful information out of Snort's signature-profiling feature. First, running the same test a large number of times (preferably 10 or more) and pulling median values from that data set help eliminate some of the noise inherent in the generated statistics. The second is to look at a group of signatures all running simultaneously and look for those that require the longest amount of time to evaluate. Because a given group of signatures is being run simultaneously, their performance relative to each other—particularly if it holds true over repeated tests—is a reasonably valid measure of which signatures might need performance improvements.

With this in mind, the fact that the signature profiling output from ten runs each of the original rule and the updated rule (with the `depth` clause included) against the sample packet capture yielded a median value of 598 microseconds in the first case and 300 microseconds in the second case validates the fact that constraining the amount of packet data to search through sped up the signature. Although that conclusion should have been obvious, confirming it with Snort's signature-profiling feature highlights that feature in an easy-to-deal-with way. This is helpful if you ever choose to use it to debug more complex signature-performance issues.

Another optimization that can be added to the signature is an isdataat clause, which is integrated into the signature as follows:

```
alert tcp $EXTERNAL_NET 21 -> $HOME_NET any (msg:"FTP Flashget PWD response
buffer overflow attempt"; flow:established,to_client; content:"257"; nocase; depth:3;
isdataat:300,relative; pcre:"/^257[^\n]{300,}/smi"; classtype:attempted-admin;
reference:bugtraq,30685; reference:cve,2008-4321; sid:99999; rev:2;)
```

The concept behind the isdataat keyword is simple: It checks to ensure that the number of bytes specified is actually present in a given packet. By specifying the relative modifier, isdataat checks for the required number of bytes following the current location in the packet (for example, after all previous content operations are performed), instead of simply checking for packets of at least the required size. As it applies to the FlashGet vulnerability, the value of isdataat is obvious: If 300 bytes are required for a buffer overflow to occur, ensuring that there are at least that many bytes remaining in the packet after the string "257" enables Snort to rapidly skip packets that cannot possibly contain an exploit. The advantage of using isdataat becomes more obvious when you consider that it

boils down to a simple piece of math on an existing packet structure, instead of the CPU-intensive process of calling into the PCRE library and performing a search that has to check each character as it moves along.

The final optimization that can be done on this signature is the most subtle. It is actually a feature of PCRE itself and not Snort specifically. By changing the expression from

```
/^257[^\n]{300,}/smi
```

to

```
/^257[^\n]{300}/smi
```

You can eliminate unnecessary work by the PCRE library on packets that contain more than 300 bytes of non-newline data following the string 257. By removing the comma from the repetition quantifier, you allow PCRE to end the process of matching as soon as it finds the 300 bytes necessary for an exploit to occur, instead of letting it continue to match until it runs out of data at the end of the packet, wasting CPU cycles as it goes.

ADVANCED EXAMPLES

Thus far, this chapter highlighted a simple vulnerability to keep the focus on the process of taking a known vulnerability and bringing it through the process of IDS analysis to create a Snort signature. Now that you understand at least the fundamentals of this process, let's get into some more advanced examples, highlighting additional useful tools and more advanced features of the Snort signature language. For ease of reading, this section focuses on a single vulnerability at a time and provides a brief header at the start of each analysis.

CITECTSCADA ODBC SERVER BUFFER OVERFLOW: METASPLOIT

Given the increase in concern about the security of Supervisory Data Acquisition and Control (SCADA) systems, which monitor critical infrastructure, such as power grids or water distribution networks, this vulnerability is a natural choice, both for attackers interested in testing those systems and IDS analysts concerned with protecting them. The fact that the exploit comes in a handy Metasploit module provides us with a chance to go over that framework.

The initial disclosure of this vulnerability came via a Core Security Technologies advisory on June 11, 2008 (www.coresecurity.com/content/citect-scada-odbc-service-vulnerability). By reading the technical details of the advisory, it's immediately noted that the signature is listening on TCP port 20222 and that there is a 5-byte content match at the start of the packet. It's also clear that some sort of size check is happening. However, the initial advisory lacks critical details; without knowing the nature of the fixed 5-byte header, for example, you can't search for it. Without sample packet captures for this application, or the time and desire to reverse-engineer a vulnerable copy of the application, no signature could be created.

All that changed when the Metasploit module was released for this vulnerability on September 5, 2008. With a functioning exploit in hand, an IDS analyst looking to create a signature for this could suddenly determine what the fixed header string is, get an idea of how large the buffer must be for the overflow, and test detection against a packet capture of a valid attack.

After downloading the latest version of the Metasploit framework and extracting it to a handy directory, you can run any of its included exploits with ease (assuming that you already have a copy of the Ruby programming language installed on your system, because the current version of Metasploit is written in Ruby). The process of running this exploit is walked through here for readers unfamiliar with operating Metasploit by using the `msfconsole` version of the system (which is by far the simplest way to run Metasploit if you're just starting out or are unfamiliar with a given exploit). On load, you're greeted with a friendly piece of ASCII art, some summary information, and a UNIX-style prompt:

```
alex@home: ~/packages/framework-3.1$ ./msfconsole


 _____________
< metasploit >
 -------------
       \   ,__,
        \  (oo)____
           (__)    )\
              ||--|| *


        =[ msf v3.1-release
+ -- --=[ 262 exploits - 117 payloads
+ -- --=[ 17 encoders - 6 nops
        =[ 46 aux

msf >
```

Use the `show exploits` command to look for the name of the exploit you want to run. If the exploit you're looking to run is not included—which is often the case, particularly with a fresh exploit—it's trivial enough to add it to the system. For example, by October 31, 2008, the CitectSCADA module was still not included in the base distribution available on the web. All Metasploit modules include a comment near the top of the file that gives the path and name of the module file. This exploit was published to Milw0rm (www.milw0rm.com/exploits/6387), and the path is exploit/windows/misc/citect_scada_odbc. By creating the file modules/exploits/windows/misc/citect_scada_odbc.rb with the contents of what's on Milw0rm and reloading Metasploit, the module is loaded into the system. With the string windows/misc/citect_scada_odbc appearing as the name of the module, you can easily select it for execution:

```
msf > use windows/misc/citect_scada_odbc
msf exploit(citect_scada_odbc) >
```

The first time you run a Metasploit module, the first thing you'll want to do after selecting it is see what options you need to specify to run it:

```
msf exploit(citect_scada_odbc) > show options


Module options:
   Name   Current Setting  Required  Description
   ——   ———————·  ————  ——————·
   RHOST                   yes       The target address
   RPORT  20222            yes       The target port
```

Each option can be set by a simple statement, such as `set RHOST 68.55.225.129` (for readers not familiar with Metasploit `RHOST` means remote host, whereas, `LHOST` means local host – `RPORT` and `LPORT` follow suit. Finally, before sending your exploit, you need to select a payload, which controls what the remote system does on a successful exploit. (In some cases, you also need to specify a target, which specifies the OS on which the vulnerable software is running.) Available options can be shown with `show payloads` (and `show targets`, if necessary) and is set with the same method as any of the other configuration options. From there, after you set up the remote system to listen properly and get a packet capture started with tcpdump, you simply type `exploit` and let Metasploit do its magic:

```
msf exploit(citect_scada_odbc) > exploit
[*] Started bind handler
[*] Trying target CiExceptionMailer.dll on XP Sp2 or SP3 5.42...
[*] Space: 329
[*] Using Windows XP Target
```

```
[*] Sent malicious ODBC packet..
```

With that done, pull up the packet capture and look at the payload. As expected, after reading the Core advisory, there are two packets with data from the client: one with 4 bytes, which you know to be the length of the following packet, and another with the actual exploit in it (see Figure 4-4). The aforementioned attack packet with the 4 bytes of data is the fourth packet in the packet capture.

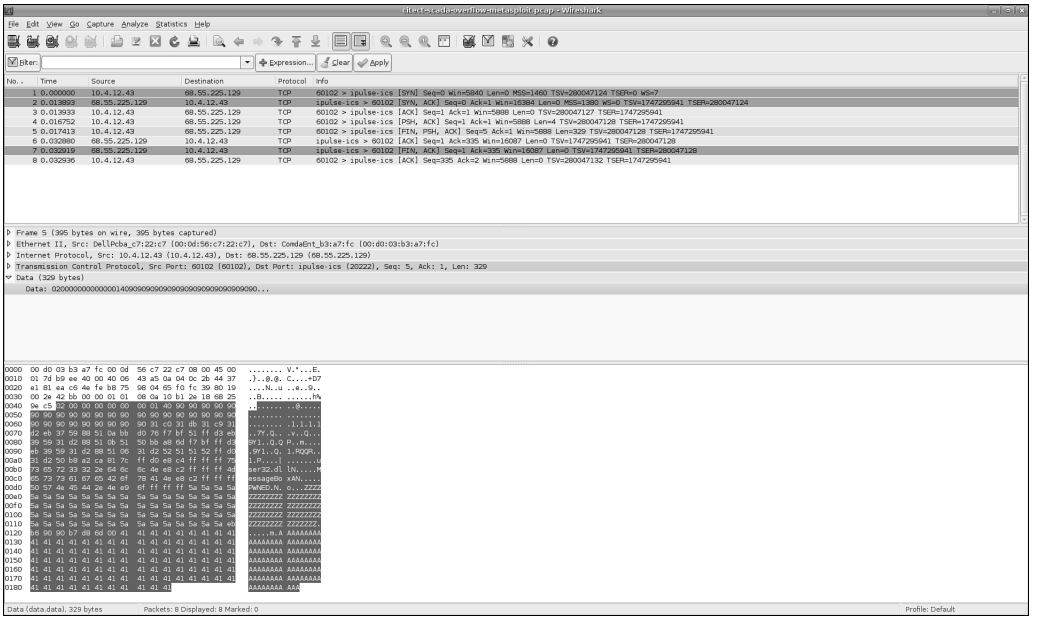

Figure 4-4 The CitectSCADA packet capture

The first question necessary to write a signature is easily answered: The fixed header is the sequence of hexadecimal bytes 02 00 00 00 00. You also have a decent idea of the sort of size you're looking for to cause an overflow; with 329 bytes of payload data, a reasonable default is probably 256 bytes or more. Because the size is declared in the data stream sent to the vulnerable system, check that size instead of relying on the size of the payload seen by Snort, because most software simply discards data that goes beyond the declared size and because, in some cases, software copies the amount of data specified in the packet even if it's not present on the wire, reading random bytes out of memory, and so on.

Had Core's advisory told you from where the size for the call to `memcpy` was coming, you might refine the signature to check for the appropriate behavior; unfortunately, the disassembly left out that argument:

```
.text:0051BC33 loc_51BC33:
.text:0051BC33 lea     ecx, [ebp+pDestBuffer]
.text:0051BC39 push    ecx     ; stack based buffer
.text:0051BC3A mov     edx, [ebp+arg_0]
.text:0051BC3D push    edx     ; class that contains packet
.text:0051BC3E call    sub_52125A ; memcpy
```

Arguments to a function are pushed on the stack in reverse order as compared to the way they're specified in a call in C. Because the specification for `memcpy` is

```
void *memcpy(void *dest, const void *src, size_t n);
```

You only have two calls to push in the assembly provided before the call to `memcpy` is made, and you have no information on from where the size used in the copy comes.

Given the choice to use the size specified in the packet, an immediate problem appears: The specified size and the fixed string that you are using to pick out packets of this particular type are in different packets, and worse, the header string comes after the size value. Because the packet with the size value in it has passed through the detection engine and out of Snort's memory by the time the header is identified, there's no way to check that value.

As it turns out, this roadblock is not permanent. Looking at the payload packet for clues about what else might be a triggering condition, the 4 bytes after the header string jump out as a possibility: 00 00 01 40. What makes these bytes special? Looking at the 4-byte size packet, the value contained is 00 00 01 49, which is the hexadecimal representation of 329, the payload packet's size. Because 0x140 is close to 0x149, it stands to reason that it might be another length value; subtracting the 5 bytes worth of fixed header, and the 4 bytes of potential size, you're left with 0x140 bytes of remaining payload. This information is confirmed by line 157 of the exploit:

```
wakeup = [0x0000000002].pack('Q')[0..4] + [mal.length].pack("N") + mal
```

With this new information in hand, writing a rule becomes trivial: Check for the header at the start of the packet and then check the size value contained in the next four packets for values above 256 bytes:

```
alert tcp $EXTERNAL_NET ANY -> $HOME_NET 20222 (msg:"MISC CitectSCADA buffer
overflow attempt"; flow:established,to_server; content:"¦02 00 00 00 00¦";
depth:5; byte_test:4,>,256,0,relative; reference:bugtraq,29634;
reference:cve,2008-2639; classtype:attempted-admin; sid:99999;)
```

After confirming that this generates an alert when run against the Metasploit packet capture, the signature is ready for production. There's no further specificity that can be added to it, there are no known false negatives or false positives, and no performance tricks can be employed to increase its speed.

FASTSTONE IMAGE VIEWER BITMAP PARSING

Along with several other popular Windows image viewers, the FastStone Image Viewer system was discovered to be vulnerable to integer overflow bugs when processing bitmap image headers in April 2007 (documented as Security Focus Bugtraq ID 23312). Although this might seem like an older vulnerability, a new script was released to exploit this problem on Milw0rm on October 5, 2008. Because this vulnerability exists in a client-side piece of software, and because many home users (or corporate desktop users) do not regularly update their software, it makes sense that it is still being exploited in the wild.

The Milw0rm script actually produces a malicious bitmap file. Because hosting a malicious image on a Web site is one of the simplest ways to exploit vulnerable users—there are a myriad of tricks for getting people to visit Web sites that an attacker controls—detecting a malicious file transfer over HTTP is likely to be one of the most effective means of mitigation an IDS can provide for a vulnerability like this. Thus, after generating the file based on the Milw0rm code, the first step in your research is to host the file on a Web server that you control and then download it, grabbing a packet capture as you go. (Of course, remove the image after you have the packet capture.) Figure 4-5 shows the packet capture.

Looking at the actual bitmap data, it's not immediately obvious where the problem is. So, more research is necessary into the nature of the vulnerability. In the References tab of the Security Focus writeup, a link exists to a blog entry written by Ivan Fratric (http://ifsec.blogspot.com/2007/04/several-windows-image-viewers.html), who discovered the vulnerability. In his section, "Experimental Results," you see that the files listed as wh3intof.bmp and wh4intof.bmp in his writeup caused FastStone Image Viewer to crash. Reading the descriptions of those files, the nature of the vulnerability becomes clear: If image width * image height * 3 is greater than the maximum size of a 32-bit integer (0xFFFFFFFF, or 4,294,967,295), an integer overflow occurs. More specifically, if a total value of, say, 0x100000001 (4,294,967,295) were arrived at by this calculation, a 32-bit system would actually store the result as 2, or the actual size minus the maximum size of an integer on that platform. Because memory is allocated to store the actual bitmap data based on that size calculation, causing this sort of an overflow might lead to the dynamically allocated memory buffer being overflowed, and thus causing a potential remote code execution.

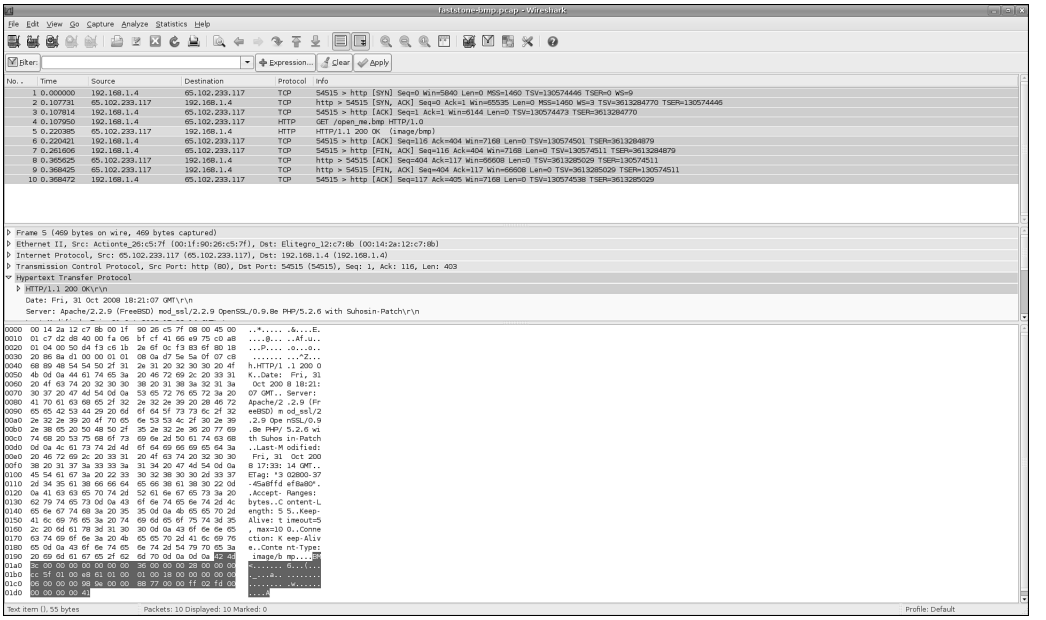

Figure 4-5 The FastStone Image Viewer packet capture

Because Wireshark doesn't parse out bitmap images for you, you are forced to go look up the bitmap specification to determine where in a bitmap file the width and the height are stored. The good news is that the Windows Bitmap Official Specification is now available online (www.fileformat.info/format/bmp/spec/e27073c25463436f8a64fa789c886d9c/view.htm). Some image or file formats do not have public documentation, and therefore require painstaking reverse-engineering to understand properly.

From the specification, the high-level layout of a bitmap file is as follows:

```
BITMAPFILEHEADER bmfh;
BITMAPINFOHEADER bmih;
RGBQUAD          aColors[];
BYTE             aBitmapBits[];
```

Because chances are good that something like image width and height are defined in a header, look to the definition of the bitmap file header and the bitmap info header:

```
typedef struct tagBITMAPFILEHEADER {    /* bmfh */
    UINT    bfType;
    DWORD   bfSize;
    UINT    bfReserved1;
```

```
    UINT    bfReserved2;
    DWORD   bfOffBits;
} BITMAPFILEHEADER;

typedef struct tagBITMAPINFOHEADER {    /* bmih */
    DWORD   biSize;
    LONG    biWidth;
    LONG    biHeight;
    WORD    biPlanes;
    WORD    biBitCount;
    DWORD   biCompression;
    DWORD   biSizeImage;
    LONG    biXPelsPerMeter;
    LONG    biYPelsPerMeter;
    DWORD   biClrUsed;
    DWORD   biClrImportant;
} BITMAPINFOHEADER;
```

These structures tell you everything you need to know to find the bitmap's width and height within a given file, because all the size specifications used for different structure members are fixed, and you know that they're the first two structures present in a bitmap file. Looking more closely at the definition of the bitmap file header, you see that the `bfType` item must always be set to BM, which makes for a handy string to use during detection.

Armed with that information, you are ready to write a signature. Unfortunately, a problem immediately arises: There's no feature built into the Snort rules language to do math, even simple arithmetic. Thus, the best possible detection that can be done is to look for unreasonably large values for the image width or the image height. By examining the Milw0rm exploit again, you see that those values are set to 0x15FCC (90,060) pixels and 0x161E8 (90,600) pixels, respectively. (Keep in mind that Microsoft typically records multibyte integers in little-endian order—with the least significant bytes first.) Because the total number is 0x5B30556A0 (24,478,308,000), it's obviously large enough to cause an integer overflow and so large that it's not particularly useful in terms of setting a minimum size for which you can look. Instead, some simple math helps provide a reasonable number: If you take 0xFFFFFFFF, divide it by 3, and take the square root of that number (because you want the smallest possible number that could appear as both the width and the height and still trigger an overflow), you end up with 0x93CD (37,837). Because no computer screen is anywhere close to being that large, you can safely set the size values that you're looking for to a number even smaller than that (say 35,000) without the likelihood of a large number of false positives.

With that conquered, the next problem is the small size of the fixed string you can search for in the signature: BM might easily appear in all sorts of HTTP traffic that have nothing to do with bitmaps. Even requiring that it appear at the start of a line (which you can do, because HTTP headers are delimited by a carriage-return new line sequence) does not help in terms of eliminating false positives. Knowing this, you need to examine the rest of the sample packet capture for anything useful (see Figure 4-6).

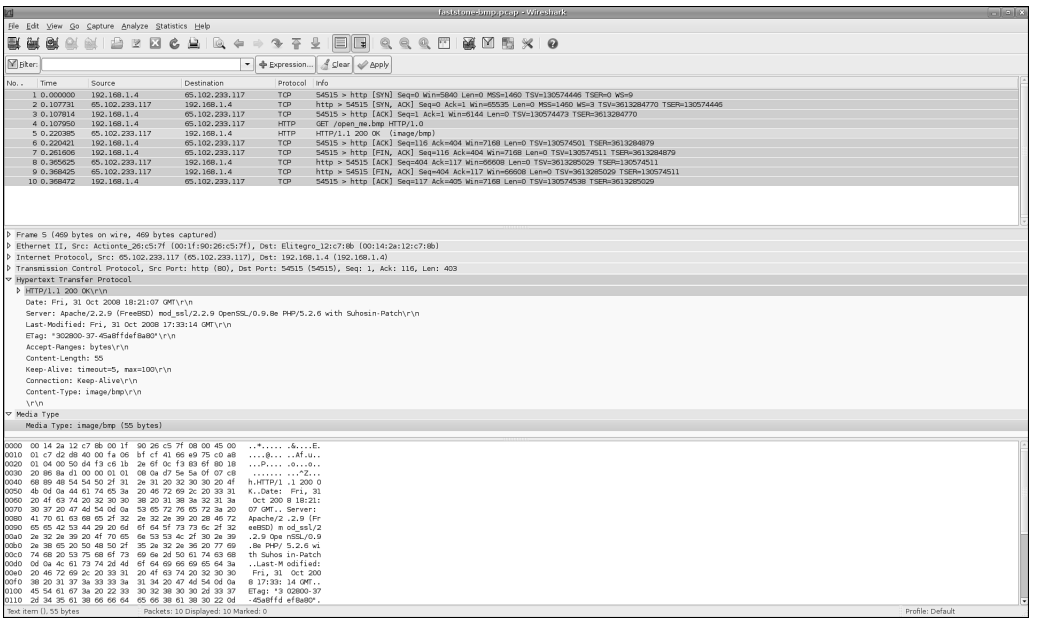

Figure 4-6 Remainder of the sample packet capture

The good news is that the HTTP headers actually provide a useful clue: the `Content-Type` header, which is set to image/bmp in the sample capture. Because all Web servers supply this header set to this value (even a malicious, custom-written server has incentive to do so, because such a header might trigger a helper application to view the malicious image), you can use it as a long content string for finding bitmap images. The resulting Snort signatures look like this:

```
alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any (msg:"WEB-CLIENT FastStone
Image Viewer integer overflow attempt - oversized width"; flow:established,to_client;
content:"Content-Type¦3A¦"; nocase;
content:"image/bmp"; nocase; distance:0;
```

```
pcre:"/^Content-Type\x3A\s*image\x2Fbmp/smi"; content:"BM"; distance:0;
byte_test:4,>,35000,16,relative,little; reference:bugtraq,23312;
reference: cve,2007-1942; classtype:attempted-user; sid:99999;)

alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any (msg:"WEB-CLIENT FastStone
Image Viewer integer overflow attempt – oversized height"; flow:established,to_client;
content:"Content-Type¦3A¦"; nocase; content:"image/bmp"; nocase; distance:0;
pcre:"/^Content-Type\x3A\s*image\x2Fbmp/smi"; content:"BM"; distance:0;
byte_test:4,>,35000,20,relative,little; reference:bugtraq,23312;
reference: cve,2007-1942; classtype:attempted-user; sid:99998;)
```

Both signatures fire on the Milw0rm packet capture as expected. However, these are not the final rules that need to go on a production system. Some Web servers send all the HTTP response headers in one packet and then begin data transmission in the next. This separates `Content-Type: image/bmp` from the actual bitmap data. Because Snort's stream-reassembly mechanism is not guaranteed to put these two packets into a single stream buffer and flush them through the detection engine, proper detection requires the use of the flowbits mechanism in Snort, which tags TCP streams so that a signature looking at a packet can tell if a necessary precondition has occurred in the stream. To do this, begin by setting a flowbit when the `Content-Type` header is seen:

```
alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any (msg:"WEB-CLIENT bitmap
transfer"; flow:established,to_client; content:"Content-Type¦3A¦"; nocase;
content:"image/bmp"; nocase; distance:0;
pcre:"/^Content-Type\x3A\s*image\x2Fbmp/smi"; flowbits:set,http.bitmap;
flowbits:noalert; sid:99997;)
```

Here, the rule is virtually identical to the start of the previously written rules; the only difference, besides the message string, are the keywords `flowbits:set,http.bitmap; flowbits:noalert;`. The first of these creates a flowbit on the current TCP session named `http.bitmap` (essentially any arbitrary name, composed of letters, digits, and/or periods, is valid); the second tells Snort that even when the signature detects everything necessary to generate an alert, it should not actually log that alert (which is particularly useful in a case like this, because generating an alert for every single bitmap downloaded over HTTP on a given network generates a huge volume of alerts). With this done, you can now tweak the other two rules to look for that flowbit:

```
alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any (msg:"WEB-CLIENT FastStone
Image Viewer multipacket integer overflow attempt – oversized width";
flow:established,to_client; flowbits:isset,http.bitmap; content:"BM"; distance:0;
byte_test:4,>,35000,16,relative,little; reference:bugtraq,23312;
reference: cve,2007-1942; classtype:attempted-user; sid:99999;)

alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any (msg:"WEB-CLIENT FastStone
Image Viewer multipacket integer overflow attempt – oversized height";
flow:established,to_client; flowbits:isset,http.bitmap; content:"BM";
distance:0; byte_test:4,>,35000,20,relative,little; reference:bugtraq,23312;
reference: cve,2007-1942; classtype:attempted-user; sid:99998;)
```

Finally, because alerts are not reliably generated when a flowbit is set and then checked successfully on the same packet, you need to bring back the original two rules, only with different SIDs, and possibly an updated message string that reflects that the attack occurred in a single packet.

LIBSPF2 DNS TXT RECORD SIZE MISMATCH

The final vulnerability this chapter discusses is a DNS bug released by Dan Kaminski in October 2008, a buffer overflow in libspf when parsing DNS TXT records. (I'm specifically avoiding the DNS cache poisoning vulnerability he released earlier that year because that vulnerability has already been widely discussed.) Because DNS often contains multiple records and does not have clear string matches that allow Snort to find the start of each record, this vulnerability excellently showcases the power of Snort's new shared object rules, which are written in C and can do anything you can do in a C program.

As discussed in Dan's writeup (www.doxpara.com/?page_id=1256), the nature of this vulnerability is straightforward. One of the types of DNS resource records is TXT, which contains an ASCII text string. All DNS resource records have a 2-byte data length field immediately preceding the actual payload. Because TXT records are of type character-string, according to RFC 1035, they have a single-byte length field, followed by the actual string itself. The problem is that many DNS implementations do not perform a sanity check to ensure that the two values work together. In some cases, such as libspf, the 2-byte length field allocates a buffer in memory and the single-byte field is the size of the memcpy, which leads to a buffer overflow if that value is larger than the 2-byte value. For this signature, assume that any time the character-string length is not equal to the resource record length minus one (to account for the length byte), something suspicious is going on and you should generate an alert. Additionally, because Dan's example uses a TXT record in the answers section, concentrate your search there and skip the possibility

that such a record might be present in an authority or additional resource record (for simplicity's sake, especially because you should have the knowledge necessary to check those other sections if you choose after reading the rest of this chapter).

After downloading Dan's tool and generating a sample packet (see Figure 4-7), which can easily be done by running `dig @ <ip of host running script> www.google.com TXT`, several things present themselves as obvious criteria for detecting this vulnerability.

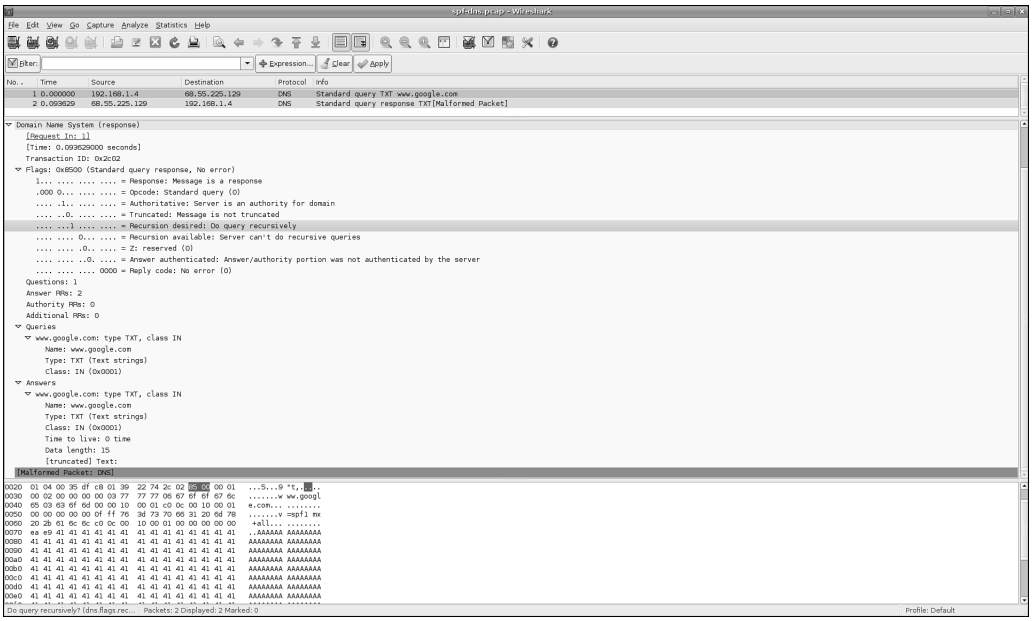

Figure 4-7 The Libspf2 packet capture

First, the packet must be inbound to your network from UDP port 53. Second, it must have the DNS response flag set, because you're looking for TXT records in answers, and answers are only present in DNS replies. Third, it must have more than zero answers listed in the header. Finally, at least one answer must contain a TXT record with a mismatched size.

Before venturing into the sometimes complex world of shared object rules, it's in your best interest to start with a standard Snort rule, add in as much detection as you can with the regular rules language, and ensure that the regular rule generates an alert on your sample packet capture. This helps you avoid dumb mistakes and frustration attempting to track down any problems in your build environment when the nature of the signature itself is the problem.

Here, check everything but the size mismatch portion in a regular Snort signature:

```
alert udp $EXTERNAL_NET 53 -> $HOME_NET any (msg:"DNS mismatched txt string
size"; byte_test:1,&,128,2; byte_test:2,>,0,6; reference:bugtraq,31881;
reference: cve,2008-2469; classtype:attempted-admin; sid:99999;)
```

The first `byte_test` performs a C-style bit masking operation to check only the single byte of the DNS packet that specifies that it's a response. The second `byte_test` ensures that the value that gives the number of answers is greater than zero. Note that, had there been a content clause in the signature before that point, it would actually have been faster to use `content:!"¦00 00¦";` to ensure that the value was not zero; however, Snort considers it a syntax error to have the only content clause in a signature be a negated content match, so you are forced to resort to the `byte_test`.

After you validate that the regular Snort rule alerts, move on to the actual shared object rule. Your best bet to properly create it is to take an existing shared object rule from the Sourcefire Certified rule set, copy it, and then tweak its data structures as necessary to use the features you want, including your references. Because this chapter is primarily concerned with the actual process of detecting vulnerabilities, it's assumed that you already know how to actually compile your shared object rule or that you can use one of the many good resources on the Internet to figure it out if you don't already know. It's also assumed that you have at least a working knowledge of C's fundamentals, because those are beyond this chapter's scope. Finally, note that the line numbers referenced correspond to those listed in the full source of this program (available at www.informit.com/store/product.aspx?isbn=0321591801).

After some initial standard setup that ensures that Snort actually gave the shared object rule a packet to work with, the first custom piece of this signature comes on line 139, where you check to ensure that at least 12 bytes of payload data are in the packet (the size of a DNS header) before doing anything else. This, along with all the other size checks performed throughout the rest of the rule, might seem pedantic; however, they're extremely critical, because if Snort, for some reason, passed in a malformed packet or, for some other reason, the shared object rule attempted to read data beyond the end of the packet, it is trivial to crash Snort (or even possibly cause exploitable memory corruption, if a skilled attacker had the source to a properly broken shared object rule).

The next two pieces of the rule call into the shared object API to run the `byte_tests`; if either test fails, you're not interested in the packet, so you return out of the rule without generating an alert. Continuing on, the rule sets the variable `cursor_raw` to point at 4 bytes from the start of the packet's payload, and then pulls out the number of queries and the number of answers declared to be in the DNS payload by using some simple C-bit shifting. Because this rule is not concerned with the other types of records that could be present, it simply skips those two numbers, sets the `end_of_payload` variable so that it can check itself before skipping anywhere within the DNS payload, and then moves on to the start of the payload.

Parsing the DNS queries is a fairly simple business. According to RFC 1035, each query is composed of a variable-length name, a 2-byte type, and a 2-byte class. Because the name portion of the query is terminated by a null byte, the easiest way to skip over each query is to loop, byte-by-byte, over the query until a null byte is encountered. (Each step along the way, check that you've not hit the end of the payload.) After this is done, the rule can skip over the next 5 bytes (the null byte itself and then the 4 bytes of data you're not interested in), and then proceed either to the next query if more are present or to the answer section of the payload.

Now you reached the interesting piece: the section where the vulnerability might be present. Looping over each of the answers noted as present in the packet, start by skipping over the name, because it's irrelevant to detection. For simplicity's sake, and based on all the DNS packets I've seen in my time as an IDS analyst, I've chosen to assume that the name is compressed using the mechanism outlined in section 4.1.4 of RFC 1035, and thus is always 2 bytes long. Next, check the 2-byte type field to ensure that it's 0x0010, which signifies a TXT record. Because the rule must skip the Time to Live (TTL) and class values of the record, from there, extract the record data size, irrespective of what type of record it is (because we'd need that value to skip over the actual data of non-TXT records). The rule simply sets a flag if the record is not of type TXT, so that it can easily be checked later. Finally, after the rule finds a TXT record, it compares the single-byte size value supplied in the packet to the data size in the previous 2 bytes and generates an alert if the values do not line up as expected.

SUMMARY

This chapter walked you through the natural evolution of a vulnerability, from discovering the vulnerability, capturing the packet stream, analyzing the malicious content within the packet, and writing an efficient Snort signature to provide an alert for it. Simultaneously, you were exposed to a small subset of necessary tools and Web sites to help you, including tcpdump, Wireshark, Metasploit, CVE, and milw0rm. The examples escalated in complexity and were specifically chosen because they were all identified

within the past few months. (This shows you the unavoidable lag time for publishing a book.) For newcomers, packet analysis might appear overwhelming and tedious, but if you segment it and step through the packet capture packet by packet, the process falls into place. Unfortunately, the process can be tedious and frustrating, but be honest—that is the challenge and enjoyment of finally comprehending it. For already skilled signature writers, hopefully the advanced examples that used flowbits, PCRE, and newly shared object rules shed some light on the thought process and technique used by the Sourcefire VRT team.