CHAPTER 1

Introduction

INFORMATION IN THIS CHAPTER:

- Setting up a Development Environment for Python
- Introduction to the Python Programming Language
- An Explanation of Variables, Data types, Strings, Lists, Dictionaries, Functions
- Work with Networking, Iteration, Selection, Exception Handling and Modules
- Write Your First Python Program, a Dictionary Password Cracker
- Write Your Second Python Program, a Zipfile Brute-Force Cracker

To me, the extraordinary aspect of martial arts lies in its simplicity. The easy way is also the right way, and martial arts is nothing at all special; the closer to the true way of martial arts, the less wastage of expression there is.

– Master Bruce Lee, Founder, Jeet Kune Do

INTRODUCTION: A PENETRATION TEST WITH PYTHON

Recently, a friend of mine penetration tested a Fortune 500 company's computer security system. While the company had established and maintained an excellent security scheme, he eventually found a vulnerability in an unpatched server. Within a few minutes, he used open source tools to compromise the system and gained administrative access to it. He then scanned the remaining servers as well as the clients and did not discover any additional vulnerabilities. At this point his assessment ended and the true penetration test began.

CONTENTS

Violent Python. http://dx.doi.org/10.1016/B978-1-59-749957-6.00001-6

Opening the text editor of his choice, my friend wrote a Python script to test the credentials found on the vulnerable server against the remainder of the machines on the network. Literally, minutes later, he gained administrative access to over one thousand machines on the network. However, in doing so, he was subsequently presented with an unmanageable problem. He knew the system administrators would notice his attack and deny him access so he quickly used some triage with the exploited machines in order to find out where to install a persistent backdoor.

After examining his pentest engagement document, my friend realized that his client placed a high level of importance on securing the domain controller. Knowing the administrator logged onto the domain controller with a completely separate administrator account, my friend wrote a small script to check a thousand machines for logged on users. A little while later, my friend was notified when the domain administrator logged onto one of the machines. His triage essentially complete, my friend now knew where to continue his assault.

My friend's ability to quickly react and think creatively under pressure made him a penetration tester. He forged his own tools out of short scripts in order to successfully compromise the Fortune 500 Company. A small Python script granted him access to over one thousand workstations. Another small script allowed him to triage the one thousand workstations before an adept administrator disconnected his access. Forging your own weapons to solve your own problems makes you a true penetration tester.

Let us begin our journey of learning how to build our own tools, by installing our development environment.

SETTING UP YOUR DEVELOPMENT ENVIRONMENT

The Python download site (http://www.python.org/download/) provides a repository of Python installers for Windows, Mac OS X, and Linux Operating Systems. If you are running Mac OS X or Linux, odds are the Python interpreter is already installed on your system. Downloading an installer provides a programmer with the Python interpreter, the standard library, and several built-in modules. The Python standard library and built-in modules provide an extensive range of capabilities, including built-in data types, exception handling, numeric, and math modules, file-handling capabilities, cryptographic services, interoperability with the operating system, Internet data handling, and interaction with IP protocols, among many other useful modules. However, a programmer can easily install any third-party packages. A comprehensive list of third-party packages is available at http://pypi.python.org/pypi/.

Installing Third Party Libraries

In Chapter two, we will utilize the python-nmap package to handle parsing of nmap results. The following example depicts how to download and install the python-nmap package (or any package, really). Once we have saved the package to a local file, we uncompress the contents and change into the uncompressed directory. From that working directory, we issue the command *python setup.py install*, which installs the python-nmap package. Installing most third-party packages will follow the same steps of downloading, uncompressing, and then issuing the command *python setup.py install*.

```
programmer:~# wget http://xael.org/norman/python/python-nmap/python-
   nmap-0.2.4.tar.gz-On map.tar.gz
--2012-04-24 15:51:51--http://xael.org/norman/python/python-nmap/
   python-nmap-0.2.4.tar.gz
Resolving xael.org... 194.36.166.10
Connecting to xael.org|194.36.166.10|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 29620 (29K) [application/x-gzip]
Saving to: 'nmap.tar.gz'
100%[=====================================================
    =====================================================
    ==============>] 29,620 60.8K/s in 0.5s
2012-04-24 15:51:52 (60.8 KB/s) - 'nmap.tar.gz' saved [29620/29620]
programmer:~# tar -xzf nmap.tar.gz
programmer:~# cd python-nmap-0.2.4/
programmer:~/python-nmap-0.2.4# python setup.py install
running install
running build
running build_py
creating build
creating build/lib.linux-x86_64-2.6
creating build/lib.linux-x86_64-2.6/nmap
copying nmap/__init__.py -> build/lib.linux-x86_64-2.6/nmap
copying nmap/example.py -> build/lib.linux-x86_64-2.6/nmap
copying nmap/nmap.py -> build/lib.linux-x86_64-2.6/nmap
running install_lib
creating /usr/local/lib/python2.6/dist-packages/nmap
copying build/lib.linux-x86_64-2.6/nmap/__init__.py -> /usr/local/lib/
   python2.6/dist-packages/nmap
copying build/lib.linux-x86_64-2.6/nmap/example.py -> /usr/local/lib/
   python2.6/dist-packages/nmap
```

```
copying build/lib.linux-x86_64-2.6/nmap/nmap.py -> /usr/local/lib/
   python2.6/dist-packages/nmap
byte-compiling /usr/local/lib/python2.6/dist-packages/nmap/__init__.py
   to __init__.pyc
byte-compiling /usr/local/lib/python2.6/dist-packages/nmap/example.py
   to example.pyc
byte-compiling /usr/local/lib/python2.6/dist-packages/nmap/nmap.py to
   nmap.pyc
running install_egg_info
Writing /usr/local/lib/python2.6/dist-packages/python_nmap-0.2.4.egg-
   info
```

To make installing Python packages even easier, Python setuptools provides a Python module called easy_install. Running the easy installer module followed by the name of the package to install will search through Python repositories to find the package, download it if found, and install it automatically.

```
programmer:∼ # easy_install python-nmap
Searching for python-nmap
Readinghttp://pypi.python.org/simple/python-nmap/
Readinghttp://xael.org/norman/python/python-nmap/
Best match: python-nmap 0.2.4
Downloadinghttp://xael.org/norman/python/python-nmap/python-nmap-
   0.2.4.tar.gz
Processing python-nmap-0.2.4.tar.gz
Running python-nmap-0.2.4/setup.py -q bdist_egg --dist-dir /tmp/easy_
   install-rtyUSS/python-nmap-0.2.4/egg-dist-tmp-EOPENs
zip_safe flag not set; analyzing archive contents...
Adding python-nmap 0.2.4 to easy-install.pth file
Installed /usr/local/lib/python2.6/dist-packages/python_nmap-0.2.4-
   py2.6.egg
Processing dependencies for python-nmap
Finished processing dependencies for python-nmap
```

To rapidly establish a development environment, we suggest you download a copy of the latest BackTrack Linux Penetration Testing Distribution from http://www.backtrack-linux.org/downloads/. The distribution provides a wealth of tools for penetration testing, along with forensic, web, network analysis, and wireless attacks. Several of the following examples will rely on tools or libraries that are already a part of the BackTrack distribution. When an example in the book requires a third-party package outside of the standard library and built-in modules, the text will provide a download site.

When setting up a developmental environment, it may prove useful to download all of these third-party modules before beginning. On Backtrack, you can install the additional required libraries with easy_install by issuing the following command. This will install most of the required libraries for the examples under Linux.

```
programmer:~ # easy_install pyPdf python-nmap pygeoip mechanize
   BeautifulSoup4
```

Chapter five requires some specific Bluetooth libraries that are not available from easy_install. You can use the aptitude package manager to download and install these librariers.

```
attacker# apt-get install python-bluez bluetooth python-obexftp
Reading package lists... Done
Building dependency tree
Reading state information... Done
<..SNIPPED..>
Unpacking bluetooth (from .../bluetooth_4.60-0ubuntu8_all.deb)
Selecting previously deselected package python-bluez.
Unpacking python-bluez (from .../python-bluez_0.18-1_amd64.deb)
Setting up bluetooth (4.60-0ubuntu8) ...
Setting up python-bluez (0.18-1) ...
Processing triggers for python-central .
```

Additionally, a few examples in Chapter five and seven require a Windows installation of Python. For the latest Python Windows Installer, visit http://www.python.org/getit/.

In recent years, the source code for Python has forked into two stable branches-2.x, and 3.x. The original author of Python, Guido van Rossum, sought to clean up the code to make the language more consistent. This action intentionally broke backward compatibility with the Python 2.x release. For example, the author replaced the print statement in Python 2.x with a print() function that required arguments as parameters. The examples contained in the following chapter are meant for the 2.x branch. At the time of this book's publication, BackTrack 5 R2 offered Python 2.6.5 as the stable version of Python.

```
programmer# python -V
Python 2.6.5
```

Interpreted Python Versus Interactive Python

Similar to other scripting languages, Python is an interpreted language. At runtime an interpreter processes the code and executes it. To demonstrate the use of the Python interpreter, we write print "Hello World" to a file with a .py extension. To interpreter this new script, we invoke the Python interpreter followed by the name of the newly created script.

```
programmer# echo print \"Hello World\" > hello.py
programmer# python hello.py
Hello World
```

Additionally, Python provides interactive capability. A programmer can invoke the Python interpreter and interact with the interpreter directly. To start the interpreter, the programmer executes python with no arguments. Next, the interpreter presents the programmer with a >>> prompt, indicating it can accept a command. Here, the programmer again types *print "Hello World."* Upon hitting return, the Python interactive interpreter immediately executes the statement.

```
programmer# python
Python 2.6.5 (r265:79063, Apr 16 2010, 13:57:41)
[GCC 4.4.3] on linux2
>>>
>>> print "Hello World"
Hello World
```

To initially understand some of the semantics behind the language, this chapter occasionally utilizes the interactive capability of the Python interpreter. You can spot the interactive interpreter in usage by looking for the >>> prompt in the examples.

As we explain the Python examples in the following chapters, we will build our scripts out of several functional blocks of code known as methods or functions. As we finalize each script, we will show how to reassemble these methods and invoke them from the main() method. Trying to run a script that just contains the isolated function definitions without a call to invoke them will prove unhelpful. For the most part, you can spot the completed scripts because they will have a main() function defined. Before we start writing our first program though, we will illustrate several of the key components of the Python standard library.

THE PYTHON LANGUAGE

In the following pages, we will tackle the idea of variables, data types, strings, complex data structures, networking, selection, iteration, file handling, exception handling, and interoperability with the operating system. To illustrate this, we will build a simple vulnerability scanner that connects to a TCP socket, reads the banner from a service, and compares that banner against known vulnerable service versions. As an experienced programmer, you may find some of the initial code examples very ugly in design. In fact, hopefully you do. As we continue to develop our script in this section, the script will hopefully grow into an elegant design you can appreciate. Let's begin by starting with the bedrock of any programming language—variables.

Variables

In Python, a variable points to data stored in a memory location. This memory location can store different values such as integers, real numbers, Booleans, strings, or more complex data such as lists or dictionaries. In the following code, we define a variable *port* that stores an integer and *banner* that stores a string. To combine the two variables together into one string, we must explicitly cast the port as a string using the str() function.

```
>>> port = 21
>>> banner = "FreeFloat FTP Server"
>>> print "[+] Checking for "+banner+" on port "+str(port)
[+] Checking for FreeFloat FTP Server on port 21
```

Python reserves memory space for variables when the programmer declares them. The programmer does not have to explicitly declare the type of variable; rather, the Python interpreter decides the type of the variable and how much space in the memory to reserve. Considering the following example, we declare a string, an integer, a list, and a Boolean, and the interpreter correctly automatically types each variable.

```
>>> banner = "FreeFloat FTP Server" # A string
>>> type(banner)
<type 'str'>
>>> port = 21                       # An integer
>>> type(port)
<type 'int'>
>>> portList=[21,22,80,110]         # A list
>>> type(portList)
<type 'list'>
>>> portOpen = True                 # A boolean
>>> type(portOpen)
<type 'bool'>
```

Strings

The Python string module provides a very robust series of methods for strings. Read the Python documentation at http://docs.python.org/library/string.html for the entire list of available methods. Let's examine a few useful methods.

Consider the use of the following methods: upper(), lower(), replace(), and find(). Upper() converts a string to its uppercase variant. Lower() converts a string to its lowercase variant. Replace(old,new) replaces the old occurrence of the substring old with the substring new. Find() reports the offset where the first occurrence of the substring occurs.

```
>>> banner = "FreeFloat FTP Server"
>>> print banner.upper()
FREEFLOAT FTP SERVER
>>> print banner.lower()
freefloat ftp server
>>> print banner.replace('FreeFloat','Ability')
Ability FTP Server
>>> print banner.find('FTP')
10
```

Lists

The list data structure in Python provides an excellent method for storing arrays of objects in Python. A programmer can construct lists of any data type. Furthermore, built-in methods exist for performing actions such as appending, inserting, removing, popping, indexing, counting, sorting, and reversing lists. Consider the following example: a programmer can construct a list by appending items using the append() method, print the items, and then sort them before printing again. The programmer can find the index of a particular item (the integer 80 in this example). Furthermore, specific items can be removed (the integer 443 in this example).

```
>>> portList = []
>>> portList.append(21)
>>> portList.append(80)
>>> portList.append(443)
>>> portList.append(25)
>>> print portList
[21, 80, 443, 25]
>>> portList.sort()
>>> print portList
[21, 25, 80, 443]
>>> pos = portList.index(80)
>>> print "[+] There are "+str(pos)+" ports to scan before 80."
[+] There are 2 ports to scan before 80.
```

```
>>> portList.remove(443)
>>> print portList
[21, 25, 80]
>>> cnt = len(portList)
>>> print "[+] Scanning "+str(cnt)+" Total Ports."
[+] Scanning 3 Total Ports.
```

Dictionaries

The Python dictionary data structure provides a hash table that can store any number of Python objects. The dictionary consists of pairs of items that contain a key and value. Let's continue with our example of a vulnerability scanner to illustrate a Python dictionary. When scanning specific TCP ports, it may prove useful to have a dictionary that contains the common service names for each port. Creating a dictionary, we can lookup a key like *ftp* and return the associated value *21* for that port.

When constructing a dictionary, each key is separated from its value by a colon, and we separate items by commas. Notice that the method .keys() will return a list of all keys in the dictionary and that the method .items() will return an entire list of items in the dictionary. Next, we verify that the dictionary contains a specific key (ftp). Referencing this key returns the value 21.

```
>>> services = {'ftp':21,'ssh':22,'smtp':25,'http':80}
>>> services.keys()
['ftp', 'smtp', 'ssh', 'http']
>>> services.items()
[('ftp', 21), ('smtp', 25), ('ssh', 22), ('http', 80)]
>>> services.has_key('ftp')
True
>>> services['ftp']
21
>>> print "[+] Found vuln with FTP on port "+str(services['ftp'])
[+] Found vuln with FTP on port 21
```

Networking

The socket module provides a library for making network connections using Python. Let's quickly write a banner-grabbing script. Our script will print the banner after connecting to a specific IP address and TCP port. After importing the socket module, we instantiate a new variable s from the class socket class. Next, we use the connect() method to make a network connection to the IP address and port. Once successfully connected, we can read and write from the socket.

The recv(1024) method will read the next 1024 bytes on the socket. We store the result of this method in a variable and then print the results to the server.

```
>>> import socket
>>> socket.setdefaulttimeout(2)
>>> s = socket.socket()
>>> s.connect(("192.168.95.148",21))
>>> ans = s.recv(1024)
>>> print ans
220 FreeFloat Ftp Server (Version 1.00).
```

Selection

Like most programming languages, Python provides a method for conditional select statements. The IF statement evaluates a logical expression in order to make a decision based on the result of the evaluation. Continuing with our banner-grabbing script, we would like to know if the specific FTP server is vulnerable to attack. To do this, we will compare our results against some known vulnerable FTP server versions.

```
>>> import socket
>>> socket.setdefaulttimeout(2)
>>> s = socket.socket()
>>> s.connect(("192.168.95.148",21))
>>> ans = s.recv(1024)
>>> if ("FreeFloat Ftp Server (Version 1.00)" in ans):
...     print "[+] FreeFloat FTP Server is vulnerable."
...elif ("3Com 3CDaemon FTP Server Version 2.0" in banner):
...     print "[+] 3CDaemon FTP Server is vulnerable."
... elif ("Ability Server 2.34" in banner):
...     print "[+] Ability FTP Server is vulnerable."
... elif ("Sami FTP Server 2.0.2" in banner):
...     print "[+] Sami FTP Server is vulnerable."
... else:
...     print "[-] FTP Server is not vulnerable."
...
[+] FreeFloat FTP Server is vulnerable."
```

Exception Handling

Even when a programmer writes a syntactically correct program, the program may still error at runtime or execution. Consider the classic runtime error—division by zero. Because zero cannot divide a number, the Python interpreter displays a message informing the programmer of the error message. This error ceases program execution.

```
>>> print 1337/0
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

What happens if we just wanted to handle the error within the context of the running program or script? The Python language provides exception-handling capability to do just this. Let's update the previous example. We use try/except statements to provide exception handling. Now, the program tries to execute the division by zero. When the error occurs, our exception handling catches the error and prints a message to the screen.

```
>>> try:
...     print "[+] 1337/0 = "+str(1337/0)
... except:
...     print "[-] Error. "
...
[-] Error
>>>
```

Unfortunately, this gives us very little information about the exact exception that caused the error. It might be useful to provide the user with an error message about the specific error that occurred. To do this, we will store the exception in a variable e to print the exception, then explicitly cast the variable e as a string.

```
>>> try:
...     print "[+] 1337/0 = "+str(1337/0)
... except Exception, e:
...     print "[-] Error = "+str(e)
...
[-] Error = integer division or modulo by zero
>>>
```

Let's now use exception handling to update our banner-grabbing script. We will wrap the network connection code with exception handling. Next, we try to connect to a machine that is not running a FTP Server on TCP port 21. If we wait for the connection timeout, we see a message indicating the network connection operation timed out. Our program can now continue.

```
>>> import socket
>>> socket.setdefaulttimeout(2)
>>> s = socket.socket()
>>> try:
...     s.connect(("192.168.95.149",21))
... except Exception, e:
...     print "[-] Error = "+str(e)
...
[-] Error = Operation timed out
```

Let us provide you one caveat about exception handling in this book. In order to cleanly illustrate the wide variety of concepts in the following pages, we have put minimal exception handling into the scripts in this book. Feel free to update the scripts included on the companion website to add more robust exception handling.

Functions

In Python, functions provide organized blocks of reusable code. Typically, this allows a programmer to write a block of code to perform a single, related action. While Python provides many built-in functions, a programmer can create user-defined functions. The keyword def() begins a function. The programmer can place any variables inside the parenthesis. These variables are then passed by reference, meaning that any changes to these variables inside the function will affect their value from the calling function. Continuing with the previous FTP vulnerability-scanning example, let's create a function to perform just the action of connecting to the FTP server and returning the banner.

```
import socket
def retBanner(ip, port):
    try:
        socket.setdefaulttimeout(2)
        s = socket.socket()
        s.connect((ip, port))
        banner = s.recv(1024)
        return banner
    except:
        return
def main():
    ip1 = '192.168.95.148'
    ip2 = '192.168.95.149'
    port = 21
```

```
   banner1 = retBanner(ip1, port)
   if banner1:
      print '[+] ' + ip1 + ': ' + banner1
   banner2 = retBanner(ip2, port)
   if banner2:
       print '[+] ' + ip2 + ': ' + banner2
if __name__ == '__main__':
   main()
```

After returning the banner, our script needs to check this banner against some known vulnerable programs. This also reflects a single, related function. The function checkVulns() takes the variable banner as a parameter and then uses it to make a determination of the vulnerability of the server.

```
import socket
def retBanner(ip, port):
    try:
        socket.setdefaulttimeout(2)
        s = socket.socket()
        s.connect((ip, port))
        banner = s.recv(1024)
        return banner
    except:
        return
def checkVulns(banner):
   if 'FreeFloat Ftp Server (Version 1.00)' in banner:
       print '[+] FreeFloat FTP Server is vulnerable.'
   elif '3Com 3CDaemon FTP Server Version 2.0' in banner:
       print '[+] 3CDaemon FTP Server is vulnerable.'
   elif 'Ability Server 2.34' in banner:
       print '[+] Ability FTP Server is vulnerable.'
   elif 'Sami FTP Server 2.0.2' in banner:
       print '[+] Sami FTP Server is vulnerable.'
   else:
       print '[-] FTP Server is not vulnerable.'
   return
def main():
    ip1 = '192.168.95.148'
    ip2 = '192.168.95.149'
    ip3 = '192.168.95.150'
```

```
     port = 21
    banner1 = retBanner(ip1, port)
    if banner1:
        print '[+] ' + ip1 + ': ' + banner1.strip('\n')
        checkVulns(banner1)
    banner2 = retBanner(ip2, port)
    if banner2:
        print '[+] ' + ip2 + ': ' + banner2.strip('\n')
        checkVulns(banner2)
    banner3 = retBanner(ip3, port)
    if banner3:
        print '[+] ' + ip3 + ': ' + banner3.strip('\n')
        checkVulns(banner3)
if __name__ == '__main__':
    main()
```

Iteration

During the last section, you might have found it repetitive to write almost the same exact code three times to check the three different IP addresses. Instead of writing the same thing three times, we might find it easier to use a for-loop to iterate through multiple elements. Consider, for example: if we wanted to iterate through the entire /24 subnet of IP addresses for 192.168.95.1 through 192.168.95.254, using a for-loop with the range from 1 to 255 allows us to print out the entire subnet.

```
>>> for x in range(1,255):
...     print "192.168.95."+str(x)
...
192.168.95.1
192.168.95.2
192.168.95.3
192.168.95.4
192.168.95.5
192.168.95.6
... <SNIPPED> ...
192.168.95.253
192.168.95.254
```

Similarly, we may want to iterate through a known list of ports to check for vulnerabilities. Instead of iterating through a range of numbers, we can iterate through an entire list of elements.

```
>>> portList = [21,22,25,80,110]
>>> for port in portList:
...     print port
...
21
22
25
80
110
```

Nesting our two for-loops, we can now print out each IP address and the ports for each address.

```
>>> for x in range(1,255):
...     for port in portList:
...             print "[+] Checking 192.168.95."\
                    +str(x)+": "+str(port)
...
[+] Checking 192.168.95.1:21
[+] Checking 192.168.95.1:22
[+] Checking 192.168.95.1:25
[+] Checking 192.168.95.1:80
[+] Checking 192.168.95.1:110
[+] Checking 192.168.95.2:21
[+] Checking 192.168.95.2:22
[+] Checking 192.168.95.2:25
[+] Checking 192.168.95.2:80
[+] Checking 192.168.95.2:110
<... SNIPPED ...>
```

With the ability to iterate through IP addresses and ports, we will update our vulnerability-checking script. Now our script will test all 254 IP addresses on the 192.168.95.0/24 subnet with the ports offering telnet, SSH, smtp, http, imap, and https services.

```
import socket
def retBanner(ip, port):
    try:
        socket.setdefaulttimeout(2)
        s = socket.socket()
```

```
            s.connect((ip, port))
            banner = s.recv(1024)
            return banner
    except:
        return
def checkVulns(banner):
    if 'FreeFloat Ftp Server (Version 1.00)' in banner:
        print '[+] FreeFloat FTP Server is vulnerable.'
    elif '3Com 3CDaemon FTP Server Version 2.0' in banner:
        print '[+] 3CDaemon FTP Server is vulnerable.'
    elif 'Ability Server 2.34' in banner:
        print '[+] Ability FTP Server is vulnerable.'
    elif 'Sami FTP Server 2.0.2' in banner:
        print '[+] Sami FTP Server is vulnerable.'
    else:
        print '[-] FTP Server is not vulnerable.'
    return
def main():
    portList = [21,22,25,80,110,443]
    for x in range(1, 255):
        ip = '192.168.95.' + str(x)
        for port in portList:
            banner = retBanner(ip, port)
            if banner:
                print '[+] ' + ip + ': ' + banner
                checkVulns(banner)
if __name__ == '__main__':
    main()
```

File I/O

While our script has an IF statement that checks a few vulnerable banners, it would be nice to occasionally add a new list of vulnerable banners. For this example, let's assume we have a text file called vuln_banners.txt. Each line in this file lists a specific service version with a previous vulnerability. Instead of constructing a huge IF statement, let's read in this text file and use it to make decisions if our banner is vulnerable.

```
programmer$ cat vuln_banners.txt
3Com 3CDaemon FTP Server Version 2.0
Ability Server 2.34
```

```
CCProxy Telnet Service Ready
ESMTP TABS Mail Server for Windows NT
FreeFloat Ftp Server (Version 1.00)
IMAP4rev1 MDaemon 9.6.4 ready
MailEnable Service, Version: 0-1.54
NetDecision-HTTP-Server 1.0
PSO Proxy 0.9
SAMBAR
Sami FTP Server 2.0.2
Spipe 1.0
TelSrv 1.5
WDaemon 6.8.5
WinGate 6.1.1
Xitami
YahooPOPs! Simple Mail Transfer Service Ready
```

We will place our updated code in the checkVulns function. Here, we will open the text file in read-only mode ('r'). We iterate through each line in the file using the method .readlines(). For each line, we compare it against our banner. Notice that we must strip out the carriage return from each line using the method .strip('\r'). If we detect a match, we print the vulnerable service banner.

```
def checkVulns(banner):
    f = open("vuln_banners.txt",'r')
    for line in f.readlines():
        if line.strip('\n') in banner:
           print "[+] Server is vulnerable: "+banner.strip('\n')
```

Sys Module

The built-in sys module provides access to objects used or maintained by the Python interpreter. This includes flags, version, max sizes of integers, available modules, path hooks, location of standard error/in/out, and command line arguments called by the interpreter. You can find more information on the Python online module documents available from http://docs.python.org/library/sys. Interacting with the sys module can prove very helpful in creating Python scripts. We may, for example, want to parse command line arguments at runtime. Consider our vulnerability scanner: what if we wanted to pass the name of a text file as a command line argument? The list sys.argv contains all the command line arguments. The first index sys.argv[0] contains the name of the interpreter Python script. The remaining items in the list contain all the following command line arguments. Thus, if we are only passing one additional argument, sys.argv should contain two items.

```
import sys
if len(sys.argv)==2:
        filename = sys.argv[1]
        print "[+] Reading Vulnerabilities From: "+filename
```

Running our code snippet, we see that the code successfully parses the command line argument and prints it to the screen. Take the time to examine the entire sys module for the wealth of capabilities it provides to the programmer.

```
programmer$ python vuln-scanner.py vuln-banners.txt
[+] Reading Vulnerabilities From: vuln-banners.txt
```

OS Module

The built-in OS module provides a wealth of OS routines for Mac, NT, or Posix operating systems. This module allows the program to independently interact with the OS environment, file-system, user database, and permissions. Consider, for example, the last section, where the user passed the name of a text file as a command line argument. It might prove valuable to check to see if that file exists and the current user has read permissions to that file. If either condition fails, it would be useful to display an appropriate error message to the user.

```
import sys
import os
if len(sys.argv) == 2:
   filename = sys.argv[1]
   if not os.path.isfile(filename):
       print '[-] ' + filename + ' does not exist.'
       exit(0)
   if not os.access(filename, os.R_OK):
       print '[-] ' + filename + ' access denied.'
       exit(0)
    print '[+] Reading Vulnerabilities From: ' + filename
```

To verify our code, we initially try to read a file that does not exist, which causes our script to print an error. Next, we create the specific filename and successfully read it. Finally, we restrict permission and see that our script correctly prints the access-denied message.

```
programmer$ python test.py vuln-banners.txt
[-] vuln-banners.txt does not exist.
programmer$ touch vuln-banners.txt
programmer$ python test.py vuln-banners.txt
[+] Reading Vulnerabilities From: vuln-banners.txt
programmer$ chmod 000 vuln-banners.txt
programmer$ python test.py vuln-banners.txt
[-] vuln-banners.txt access denied.
```

We can now reassemble all the various pieces and parts of our Python vulnerability-scanning script. Do not worry if it appears pseudo-complete, lacking the ability to use threads of execution or better command line option parsing. We will continue to build upon this script in the following chapter.

```
Import socket
import os
import sys
def retBanner(ip, port):
    try:
        socket.setdefaulttimeout(2)
        s = socket.socket()
        s.connect((ip, port))
        banner = s.recv(1024)
        return banner
    except:
        return
def checkVulns(banner, filename):
    f = open(filename, 'r')
    for line in f.readlines():
        if line.strip('\n') in banner:
            print '[+] Server is vulnerable: ' +\
                banner.strip('\n')
def main():
   if len(sys.argv) == 2:
       filename = sys.argv[1]
       if not os.path.isfile(filename):
           print '[-] ' + filename +\
                ' does not exist.'
              exit(0)
              if not os.access(filename, os.R_OK):
                  print '[-] ' + filename +\
                      ' access denied.'
                  exit(0)
      else:
          print '[-] Usage: ' + str(sys.argv[0]) +\
          ' <vuln filename>'
          exit(0)
          portList = [21,22,25,80,110,443]
          for x in range(147, 150):
              ip = '192.168.95.' + str(x)
              for port in portList:
                  banner = retBanner(ip, port)
                  if banner:
                      print '[+] ' + ip + ': ' + banner
                      checkVulns(banner, filename)
if __name__ == '__main__':
    main()
```

YOUR FIRST PYTHON PROGRAMS

With an understanding how to build Python scripts, let us begin writing our first two programs. As we move forward, we will describe a few anecdotal stories that emphasize the need for our scripts.

Setting the Stage for Your First Python Program: *The Cuckoo's Egg*

A system administrator at Lawrence Berkley National Labs, Clifford Stoll, documented his personal hunt for a hacker (and KGB informant) who broke into various United States national research laboratories, army bases, defense contractors, and academic institutions in *The Cuckoo's Egg: Tracking a Spy Through the Maze of Computer Espionage* (Stoll, 1989). He also published a May 1988 article in *Communications of the ACM* describing the in-depth technical details of the attack and hunt (Stoll, 1988).

Fascinated by the attacker's methodology and actions, Stoll connected a printer to a compromised server and logged every keystroke the attacker made. During one recording, Stoll noticed something interesting (at least in 1988).

Almost immediately after compromising a victim, the attacker downloaded the encrypted password file. What use was this to the attacker? After all, the victim systems encrypted the user passwords using the UNIX crypt algorithm. However, within a week of stealing the encrypted password files, Stoll saw the attacker log on with the stolen accounts. Confronting some of the victim users, he learned they had used common words from the dictionary as passwords (Stoll, 1989).

Upon learning this, Stoll realized that the hacker had used a dictionary attack to decrypt the encrypted passwords. The hacker enumerated through all the words in a dictionary and encrypted them using the Unix Crypt() function. After encrypting each password, the hacker compared it with the stolen encrypted password. The match translated to a successful password crack.

Consider the following encrypted password file. The victim used a plaintext password *egg* and salt equal to the first two bytes or *HX*. The UNIX Crypt function calculates the encrypted password with *crypt('egg','HX') = HX9LLTdc/jiDE*.

```
attacker$ cat /etc/passwd
victim: HX9LLTdc/jiDE: 503:100:Iama Victim:/home/victim:/bin/sh
root: DFNFxgW7C05fo: 504:100: Markus Hess:/root:/bin/bash
```

Let's use this encrypted password file as an opportunity to write our first Python script, a UNIX password cracker.

Your First Program, a UNIX Password Cracker

The real strength of the Python programming language lies in the wide array of standard and third-party libraries. To write our UNIX password cracker, we will need to use the crypt() algorithm that hashes UNIX passwords. Firing up the Python interpreter, we see that the crypt library already exists in the Python standard library. To calculate an encrypted UNIX password hash, we simply call the function crypt.crypt() and pass it the password and salt as parameters. This function returns the hashed password as a string.

```
Programmer$ python
>>> help('crypt')
Help on module crypt:
NAME
    crypt
 FILE
    /System/Library/Frameworks/Python.framework/Versions/2.7/lib/
    python2.7/lib-dynload/crypt.so
MODULE DOCS
    http://docs.python.org/library/crypt
```

```
FUNCTIONS
   crypt(...)
       crypt(word, salt) -> string
       word will usually be a user's password. salt is a 2-character string
       which will be used to select one of 4096 variations of DES. The
       characters in salt must be either ".", "/", or an alphanumeric
       character. Returns the hashed password as a string, which will be
       composed of characters from the same alphabet as the salt.
```

Let's quickly try hashing a password using the crypt() function. After importing the library, we pass the password "egg" and the salt "HX" to the function. The function returns the hashed password value "HX9LLTdc/jiDE" as a string. Success! Now we can write a program to iterate through an entire dictionary, trying each word with the custom salt for the hashed password.

```
programmer$ python
>>> import crypt
>>> crypt.crypt("egg","HX")
'HX9LLTdc/jiDE'
```

To write our program, we will create two functions-main and testpass. It proves a good programming practice to separate your program into separate functions, each with a specific purpose. In the end, this allows us to reuse code and makes the program easier to read. Our main function opens the encrypted password file "passwords.txt" and reads the contents of each line in the password file. For each line, it splits out the username and the hashed password. For each individual hashed password, the main function calls the testPass() function that tests passwords against a dictionary file.

This function, testPass(), takes the encrypted password as a parameter and returns either after finding the password or exhausting the words in the dictionary. Notice that the function first strips out the salt from the first two characters of the encrypted password hash. Next, it opens the dictionary and iterates through each word in the dictionary, creating an encrypted password hash from the dictionary word and the salt. If the result matches our encrypted password hash, the function prints a message indicating the found password and returns. Otherwise, it continues to test every word in the dictionary.

```
import crypt
def testPass(cryptPass):
     salt = cryptPass[0:2]
```

```
    dictFile = open('dictionary.txt','r')
    for word in dictFile.readlines():
         word = word.strip('\n')
         cryptWord = crypt.crypt(word,salt)
         if (cryptWord == cryptPass):
               print "[+] Found Password: "+word+"\n"
               return
    print "[-] Password Not Found.\n"
    return
def main():
    passFile = open('passwords.txt')
    for line in passFile.readlines():
        if ":" in line:
               user = line.split(':')[0]
               cryptPass = line.split(':')[1].strip(' ')
               print "[*] Cracking Password For: "+user
               testPass(cryptPass)
if __name__ == "__main__":
   main()
```

Running our first program, we see that it successfully cracks the password for victim but does not crack the password for root. Thus, we know the system administrator (root) must be using a word not in our dictionary. No need to worry, we'll cover several other ways in this book to gain root access.

```
programmer$ python crack.py
[*] Cracking Password For: victim
[+] Found Password: egg
[*] Cracking Password For: root
[-] Password Not Found.
```

On modern *Nix based operating systems, the /etc/shadow file stores the hashed password and provides the ability to use more secure hashing algorithms. The following example uses the SHA-512 hashing algorithm. SHA-512 functionality is provided by the Python hashlib library. Can you update the script to crack SHA-512 hashes?

```
cat /etc/shadow | grep root
root:$6$ms32yIGN$NyXjOYofkK14MpRwFHvXQWOyvUid.slJtgxHE2EuQqgD74S/
   GaGGs5VCnqeC.bSOMzTf/EFS3uspQMNeepIAc.:15503:0:99999:7:::
```

Setting the Stage for Your Second Program: Using Evil for Good

Announcing his Cyber Fast Track program at ShmooCon 2012, Peiter "Mudge" Zatko, the legendary l0pht hacker turned DARPA employee, explained that there are really no offensive or defensive tools-instead there are simply tools (Zatko, 2012). Throughout this book, you may initially find several of the example scripts somewhat offensive in nature. For example, take our last program that cracked passwords on Unix systems. An adversary *could* use the tool to gain unauthorized access to a system; however, could a programmer use this for good as well as evil? Certainly—let's expand.

Fast-forward nineteen years from Clifford Stoll's discovery of the dictionary attack. In early 2007, the Brownsville, TX Fire Department received an anonymous tip that fifty-year-old John Craig Zimmerman browsed child pornography using department resources (Floyd, 2007). Almost immediately, the Brownsville Fire Department granted Brownsville police investigators access to Zimmerman's work computer and external hard drive (Floyd, 2007). The police department brought in city programmer Albert Castillo to search the contents of Zimmerman's computer (McCullagh, 2008). Castillo's initial investigation found several adult pornographic images but no child pornography.

Continuing to browse through the files, Castillo found some suspect files, including a password-protected ZIP file titled "Cindy 5." Relying on the technique invented nearly two decades earlier, Castillo used a dictionary attack to decrypt the contents of the password-protected file. The resulting decrypted files showed a partially naked minor (McCullagh, 2008). With this information, a judge granted investigators a warrant to search Zimmerman's home, where they discovered several additional pornographic images of children (McCullagh, 2008). On April 3, 2007, a federal grand jury returned an indictment, charging John Craig Zimmerman with four counts of possession and production of child pornography (Floyd, 2007).

Let's use the technique of brute-forcing a password learned in the last example program but apply it to zip files. We will also use this example to expand upon some fundamental concepts of building our programmers.

Your Second Program, a Zip-File Password Cracker

Let's begin writing our zip-file password cracker by examining the zipfile library. Opening the Python interpreter, we issue the command help('zipfile') to learn more about the library and see the class ZipFile with a method extractall(). This class and method will prove useful in writing our program to crack password-protected zip files. Note how the method extractall() has an optional parameter to specify a password.

```
programmer$ python
Python 2.7.1 (r271:86832, Jun 16 2011, 16:59:05)
Type "help", "copyright", "credits" or "license" for more information.
>>> help('zipfile')
<..SNIPPED..>
    class ZipFile
         | Class with methods to open, read, write, close, list zip
   files.
         |
         | z = ZipFile(file, mode="r", compression=ZIP_STORED,
         allowZip64=False)
<..SNIPPED..>
         |    extractall(self, path=None, members=None, pwd=None)
         |        Extract all members from the archive to the current
                  working
         |        directory. 'path' specifies a different directory to
                  extract to.
         |       'members' is optional and must be a subset of the list
                  returned
```

Let's write a quick script to test the use of the zipfile library. After importing the library, we instantiate a new ZipFile class by specifying the filename of the password-protected zip file. To extract the zip file, we utilize the extractall() method and specify the optional parameter for the password.

```
import zipfile
zFile = zipfile.ZipFile("evil.zip")
zFile.extractall(pwd="secret")
```

Next, we execute our script to ensure it works properly. Notice that prior to execution, only the script and the zip file exist in our current working directory. We execute our script, which extracts the contents of evil.zip to a newly created directory called evil/. This directory contains the files from the previously password-protected zip file.

```
programmer$ ls
evil.zip unzip.py
programmer$ python unzip.py
programmer$ ls
evil.zip unzip.py evil
programmer$ cd evil/
programmer$ ls
note_to_adam.txt apple.bmp
```

However, what happens if we execute the script with an incorrect password? Let's add some exception handling to catch and display the error message from the script.

```
import zipfile
zFile = zipfile.ZipFile("evil.zip")
try:
        zFile.extractall(pwd="oranges")
except Exception, e:
       print e
```

Executing our script with an incorrect password, we see that it prints an error message, indicating that the user specified an incorrect password to decrypt the contents of the password-protected zip file.

```
programmer$ python unzip.py
('Bad password for file', <zipfile.ZipInfo object at 0x10a859500>)
```

We can use the fact that an incorrect password throws an exception to test our zip file against a dictionary file. After instantiating a ZipFile class, we open a dictionary file and iterate through and test each word in the dictionary. If the method extractall() executes without error, we print a message indicating the working password. However, if extractall() throws a bad password exception, we ignore the exception and continue trying passwords in the dictionary.

```
import zipfile
zFile = zipfile.ZipFile('evil.zip')
passFile = open('dictionary.txt')
for line in passFile.readlines():
  password = line.strip('\n')
  try:
       zFile.extractall(pwd=password)
       print '[+] Password = ' + password + '\n'
       exit(0)
   except Exception, e:
        pass
```

Executing our script, we see that it correctly identifies the password for the password-protected zip file.

```
programmer$ python unzip.py
[+] Password = secret
```

Let's clean up our code a little bit at this point. Instead of having a linear program, we will modularize our script with functions.

```
import zipfile
def extractFile(zFile, password):
   try:
        zFile.extractall(pwd=password)
        return password
   except:
        return
def main():
  zFile = zipfile.ZipFile('evil.zip')
  passFile = open('dictionary.txt')
  for line in passFile.readlines():
        password = line.strip('\n')
        guess = extractFile(zFile, password)
        if guess:
             print '[+] Password = ' + password + '\n'
             exit(0)
if __name__ == '__main__':
    main()
```

With our program modularized into separate functions, we can now increase our performance. Instead of trying each word in the dictionary one at a time, we will utilize threads of execution to allow simultaneous testing of multiple passwords. For each word in the dictionary, we will spawn a new thread of execution.

```
import zipfile
from threading import Thread
def extractFile(zFile, password):
   try:
        zFile.extractall(pwd=password)
        print '[+] Found password ' + password + '\n'
   except:
      pass
def main():
    zFile = zipfile.ZipFile('evil.zip')
    passFile = open('dictionary.txt')
   for line in passFile.readlines():
       password = line.strip('\n')
       t = Thread(target=extractFile, args=(zFile, password))
       t.start()
if __name__ == '__main__':
      main()
```

Now let's modify our script to allow the user to specify the name of the zip file to crack and the name of the dictionary file. To do this, we will import the optparse library. We will describe this library better in the next chapter. For the purposes of our script here, we only need to know that it parses flags and optional parameters following our script. For our zip-file-cracker script, we will add two mandatory flags—zip file name and dictionary name.

```
import zipfile
import optparse
from threading import Thread
def extractFile(zFile, password):
   try:
       zFile.extractall(pwd=password)
       print '[+] Found password ' + password + '\n'
   except:
       pass
def main():
    parser = optparse.OptionParser("usage%prog "+\
    "-f <zipfile> -d <dictionary>")
    parser.add_option('-f', dest='zname', type='string',\
    help='specify zip file')
    parser.add_option('-d', dest='dname', type='string',\
    help='specify dictionary file')
    (options, args) = parser.parse_args()
    if (options.zname == None) | (options.dname == None):
       print parser.usage
       exit(0)
    else:
       zname = options.zname
       dname = options.dname
    zFile = zipfile.ZipFile(zname)
    passFile = open(dname)
    for line in passFile.readlines():
        password = line.strip('\n')
        t = Thread(target=extractFile, args=(zFile, password))
        t.start()
if __name__ == '__main__':
    main()
```

Finally, we test our completed password-protected zip-file-cracker script to ensure it works. Success with a thirty-five-line script!

```
programmer$ python unzip.py -f evil.zip -d dictionary.txt
[+] Found password secret
```

CHAPTER WRAP-UP

In this chapter, we briefly examined the standard library and a few built-in modules in Python by writing a simple vulnerability scanner. Next, we moved on and wrote our first two Python programs—a twenty-year-old UNIX password cracker and a zip-file brute-force password cracker. You now have the initial skills to write your own scripts. Hopefully, the following chapters will prove as exciting to read as they were to write. We will begin this journey by examining how to use Python to attack systems during a penetration test.

References

Floyd, J. (2007). Federal grand jury indicts fireman for production and possession of child pornography. John T. Floyd Law Firm Web site. Retrieved from <http://www.houston-federal-criminal-lawyer.com/news/april07/03a.htm>, April 3.

McCullagh, D. (2008). Child porn defendant locked up after ZIP file encryption broken. *CNET News*. Retrieved April 7, 2012, from <http://news.cnet.com/8301-13578_3-9851844-38.html>, January 16.

Stoll, C. (1989). *The cuckoo's egg: Tracking a spy through the maze of computer espionage*. New York: Doubleday.

Stoll, C. (1988). Stalking the Wily Hacker. *Communications of the ACM*, *31*(5), 484–500.

Zatko, P. (2012). Cyber fast track. ShmooCon 2012. Retrieved June 13, 2012. from <www.shmoocon.org/2012/videos/Mudge-CyberFastTrack.m4v>, January 27.