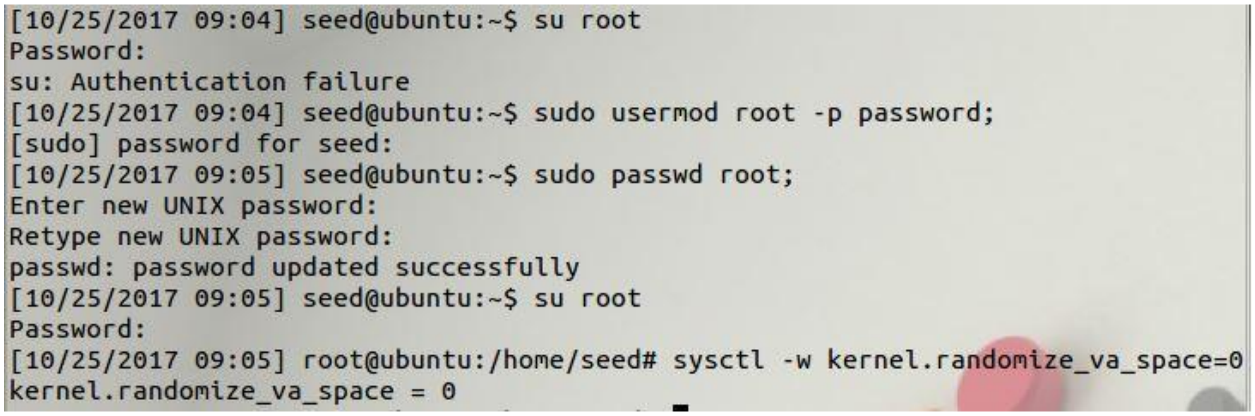
Assignment 5

In this assignment we are experimenting with the Buffer Overflow attack on the virtual machine. Address space randomization is disabled to make our attack easier. Address space randomization is a defense against the buffer overflows which makes guessing addresses in the heap and stack more difficult.

2.1 Initial Setup:

Address Space Randomization.

First, I tried to enter the root shell but as I don't know the root password it showed me the authentication failure. Then I used the below sudo commands to change the password of root. Then I logged into the root shell successfully. After that I disabled the Address space randomization by using the last command in Fig 1.

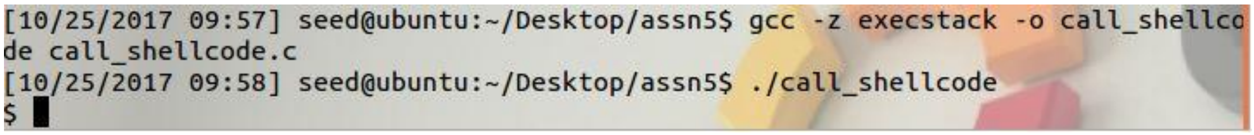


```
[10/25/2017 09:04] seed@ubuntu:~$ su root
Password:
su: Authentication failure
[10/25/2017 09:04] seed@ubuntu:~$ sudo usermod root -p password;
[sudo] password for seed:
[10/25/2017 09:05] seed@ubuntu:~$ sudo passwd root;
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
[10/25/2017 09:05] seed@ubuntu:~$ su root
Password:
[10/25/2017 09:05] root@ubuntu:/home/seed# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

Fig 1: Disabling address space randomization

2.2 Shellcode

A shellcode is needed to launch a shell before starting the attack. It must be loaded into the memory so that one can force the vulnerable program to jump to it. A new file, call_shellcode.c is created and pasted the given program into that file. The it is compiled using gcc and executed which started a shell that can be used to run programs.

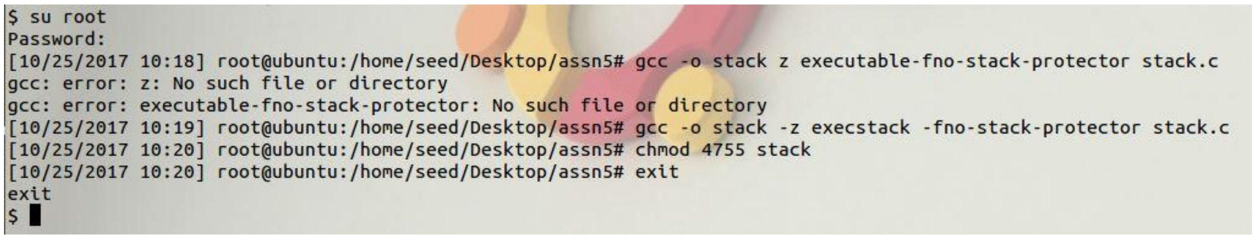


```
[10/25/2017 09:57] seed@ubuntu:~/Desktop/assn5$ gcc -z execstack -o call_shellco
de call_shellcode.c
[10/25/2017 09:58] seed@ubuntu:~/Desktop/assn5$ ./call_shellcode
$ 
```

Fig 2: Compiling and executing call_shellcode.c

2.3 The Vulnerable Program

A file is created and named it as a stack.c which is a vulnerable program. The given code is pasted into this file. It needs to be made set-root-uid. This can be done by compiling it in a root account. The set-root-uid exploits the buffer overflow to be able to gain access to a root shell. As the stack guard option is enabled by default, one needs to disable it while compiling. Executable stack option is also added to be able to run our shellcode from buffer. After that we chmod the permissions to 4755 to make the file executable.



```
$ su root
Password:
[10/25/2017 10:18] root@ubuntu:/home/seed/Desktop/assn5# gcc -o stack z executable-fno-stack-protector stack.c
gcc: error: z: No such file or directory
gcc: error: executable-fno-stack-protector: No such file or directory
[10/25/2017 10:19] root@ubuntu:/home/seed/Desktop/assn5# gcc -o stack -z execstack -fno-stack-protector stack.c
[10/25/2017 10:20] root@ubuntu:/home/seed/Desktop/assn5# chmod 4755 stack
[10/25/2017 10:20] root@ubuntu:/home/seed/Desktop/assn5# exit
exit
$ █
```

Fig 3: Compiling and chmod the permissions to stack.

2.4 Task 1: Exploiting the Vulnerability

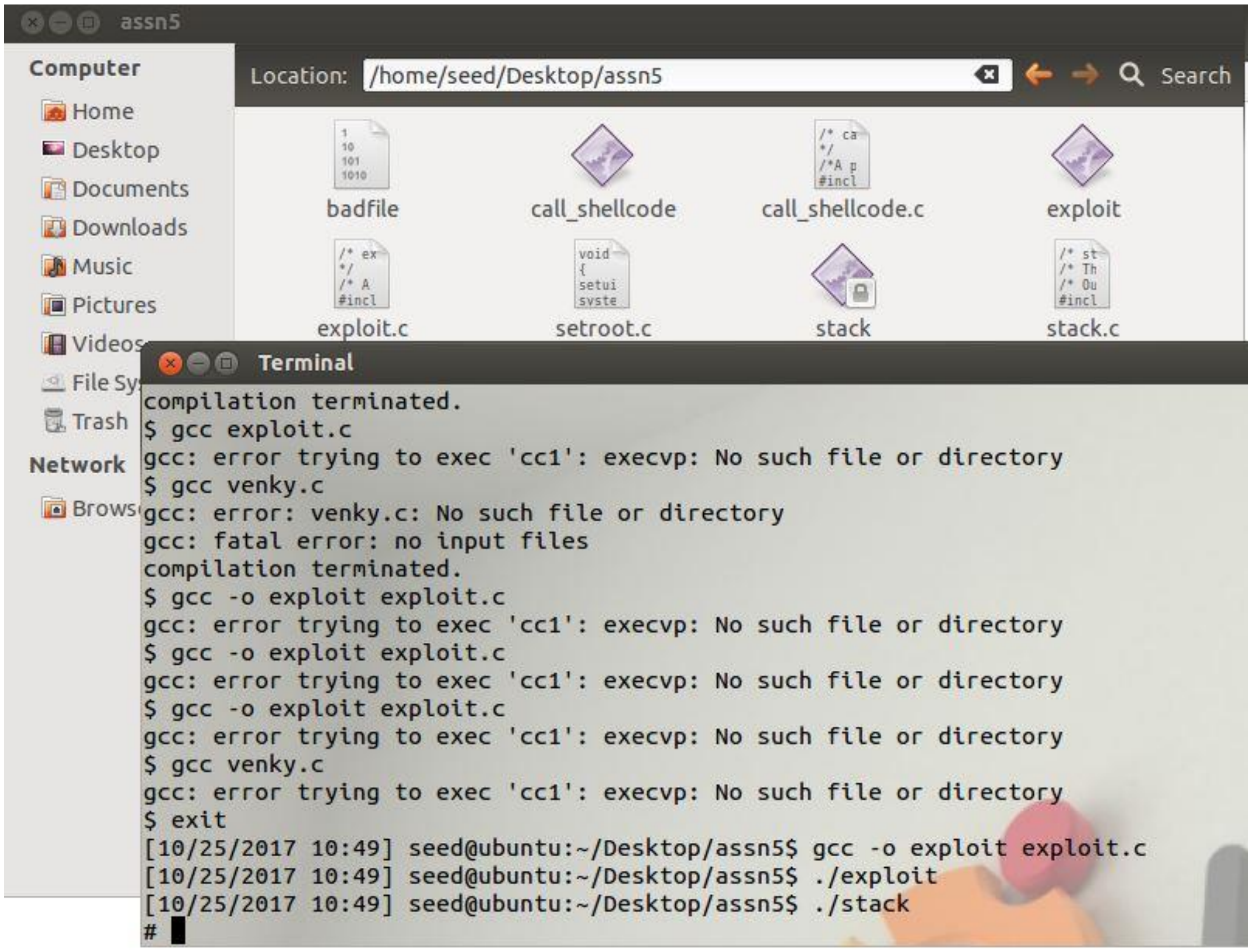
A new file is created and christened it as exploit.c. The given code is pasted in this file. Now we need to create the **badfile** which can be used by the vulnerable program(stack.c) to get attacked (Buffer overflow attack) by storing this in the buffer that will be overflown**.** I filled the buffer with appropriate contents in the code and compiled.

Observations:

The exploit.c file is compiled normally and executed the executable file which is created after the compilation. The exploit file evaluates the stack pointer and creates a buffer which is then saved into a badfile. Now, the vulnerable program stack is executed which reads the badfile and loads the buffer. Now, the size of buffer in bof() function is less than str (517 bytes) .Now, the program got attacked by the buffer overflow attack and starts executing the shell code, thus giving the root shell.

Others:

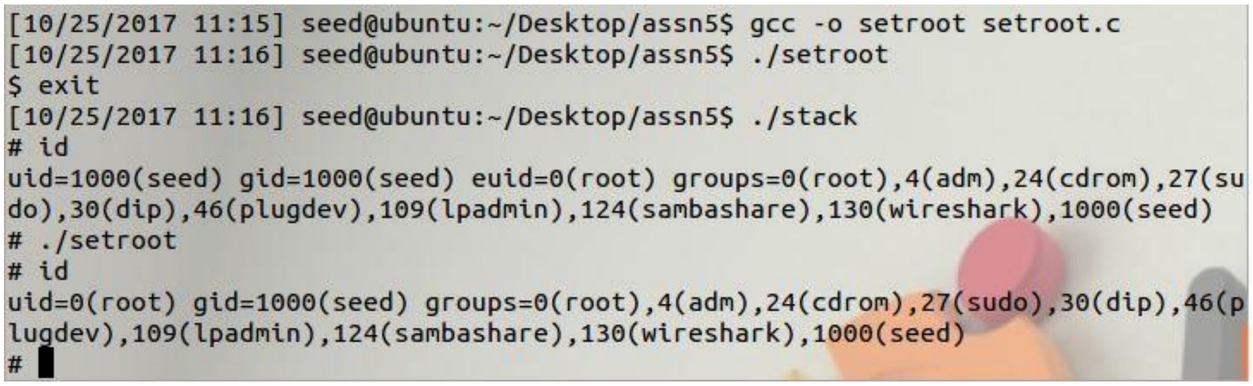
First, I created a bad file with shellcode which can be seen in the exploit.c program. Next, the bad file is created with eip overwritten with some content. Later, the bad file is created with the appropriate address as eip.



```
compilation terminated.
$ gcc exploit.c
gcc: error trying to exec 'cc1': execvp: No such file or directory
$ gcc venky.c
gcc: error: venky.c: No such file or directory
gcc: fatal error: no input files
compilation terminated.
$ gcc -o exploit exploit.c
gcc: error trying to exec 'cc1': execvp: No such file or directory
$ gcc -o exploit exploit.c
gcc: error trying to exec 'cc1': execvp: No such file or directory
$ gcc -o exploit exploit.c
gcc: error trying to exec 'cc1': execvp: No such file or directory
$ gcc venky.c
gcc: error trying to exec 'cc1': execvp: No such file or directory
$ exit
[10/25/2017 10:49] seed@ubuntu:~/Desktop/assn5$ gcc -o exploit exploit.c
[10/25/2017 10:49] seed@ubuntu:~/Desktop/assn5$ ./exploit
[10/25/2017 10:49] seed@ubuntu:~/Desktop/assn5$ ./stack
# 
```

Fig 4: Buffer overflow attack

The root shell that we opened is still using our used ID. This can be checked by running **id** command in the root shell. Fig 5 clearly demonstrates the id command. First time when I type the id it is showing uid = 1000 and after executing the setroot the uid is changed to 0. setroot is a program which turns the real user id to root.



```
[10/25/2017 11:15] seed@ubuntu:~/Desktop/assn5$ gcc -o setroot setroot.c
[10/25/2017 11:16] seed@ubuntu:~/Desktop/assn5$ ./setroot
$ exit
[10/25/2017 11:16] seed@ubuntu:~/Desktop/assn5$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root),4(adm),24(cdrom),27(su
do),30(dip),46(plugdev),109(lpadmin),124(sambashare),130(wireshark),1000(seed)
# ./setroot
# id
uid=0(root) gid=1000(seed) groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(p
lugdev),109(lpadmin),124(sambashare),130(wireshark),1000(seed)
# 
```

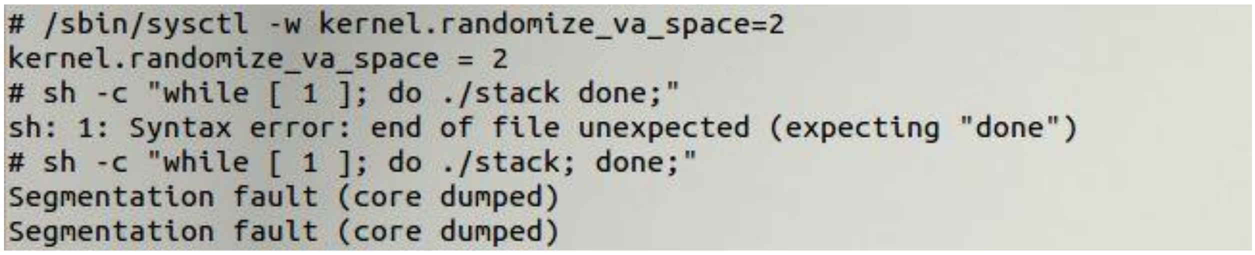
Fig 5: Setting real user id to root

2.5 Task 2: Address Randomization

Address Randomization is disabled in the first task to make the attack easier. Now, the address randomization is enabled by using the command executed in the below figure.

Observations:

Running the stack in a loop gives the segmentation fault (core dumped) error because the address is randomized every time the program is executed. So, the stack pointer is different and the exploit.c program will not set the address properly anymore for the buffer flow to run the shellcode. After some time, the shell is opened as at some time the stack address matches with the address mentioned in the program.



```
# /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
# sh -c "while [ 1 ]; do ./stack done;"
sh: 1: Syntax error: end of file unexpected (expecting "done")
# sh -c "while [ 1 ]; do ./stack; done;"
Segmentation fault (core dumped)
Segmentation fault (core dumped)
```

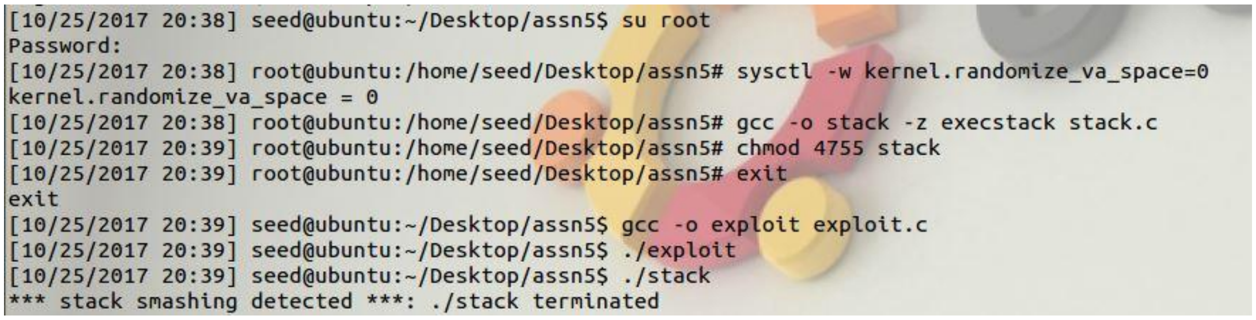
Fig 6: Address Randomization

2.6 Task 3: Stack Guard

Now the stack file is compiled without the stack guard protection mechanism. To perform this task the address randomization is disabled first because we don't know which protection helps achieve the protection. Next, the stack.c file is compiled by disabling the stack guard protection.

Observations:

gcc detected the stack smashing while executing the stack and terminated the program and prevented the attack.



```
[10/25/2017 20:38] seed@ubuntu:~/Desktop/assn5$ su root
Password:
[10/25/2017 20:38] root@ubuntu:/home/seed/Desktop/assn5# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/25/2017 20:38] root@ubuntu:/home/seed/Desktop/assn5# gcc -o stack -z execstack stack.c
[10/25/2017 20:39] root@ubuntu:/home/seed/Desktop/assn5# chmod 4755 stack
[10/25/2017 20:39] root@ubuntu:/home/seed/Desktop/assn5# exit
exit
[10/25/2017 20:39] seed@ubuntu:~/Desktop/assn5$ gcc -o exploit exploit.c
[10/25/2017 20:39] seed@ubuntu:~/Desktop/assn5$ ./exploit
[10/25/2017 20:39] seed@ubuntu:~/Desktop/assn5$ ./stack
*** stack smashing detected ***: ./stack terminated
```

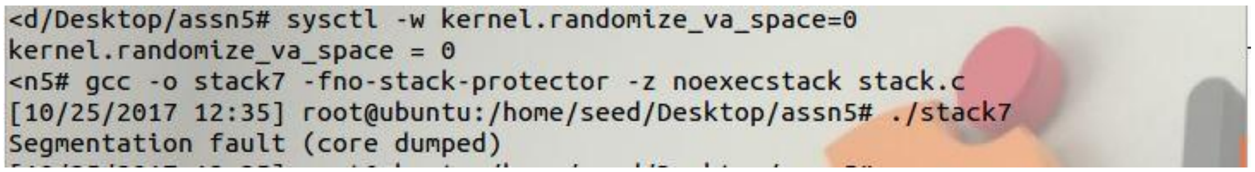
Fig 7: Program executed without stack guard protection

2.7 Task 4: Non-executable Stack

Now the stack file is compiled with a non-executable stack. To perform this task the address randomization is disabled first because we don't know which protection helps achieve the protection. Next, stack.c file is compiled by using the non-executable stack. After executing the stack

Observations:

I got a segmentation fault (core dumped) error which means that the buffer is overflown but the shell was not opened as it is a non-executable stack (Shell code is not executed).



```
<d/Desktop/assn5# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
<n5# gcc -o stack7 -fno-stack-protector -z noexecstack stack.c
[10/25/2017 12:35] root@ubuntu:/home/seed/Desktop/assn5# ./stack7
Segmentation fault (core dumped)
```

Fig 8: Program executed with a non-executable stack.