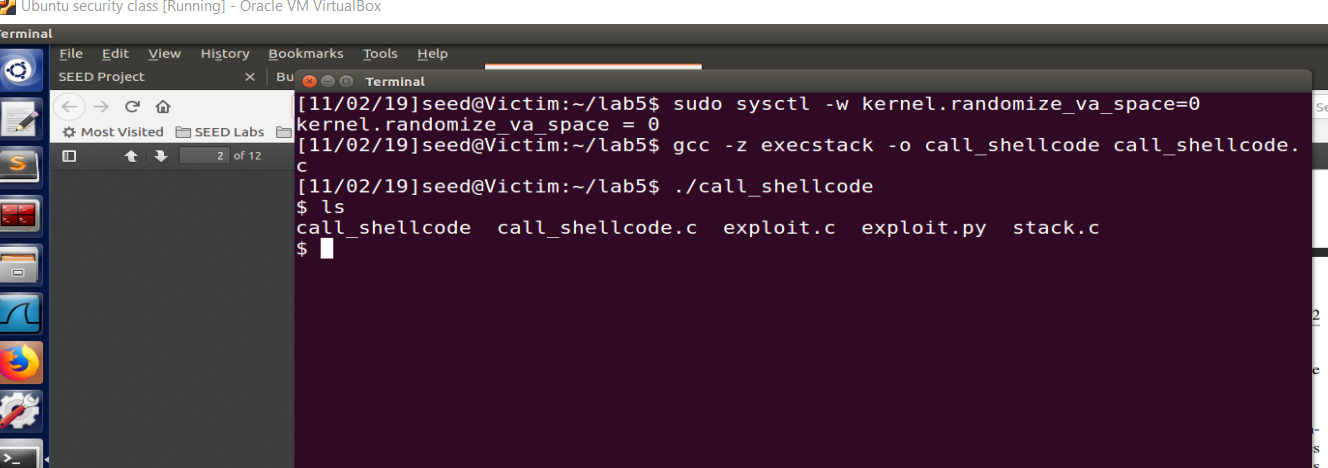
**Lab5: Buffer overflow attack Lab5**

**Pramod kumar, pjk5502@psu.edu**

**Task 1: ﻿Running Shellcode**

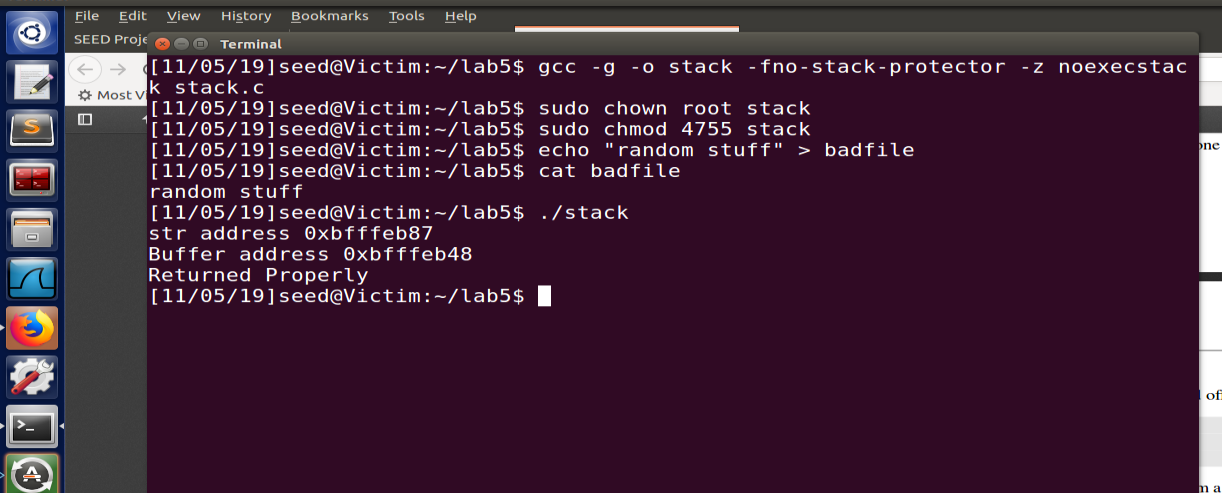
Turn off Address randomization and execute call\_shallcode to get shall exec



Observation : On running above simple program we able to get system shell.

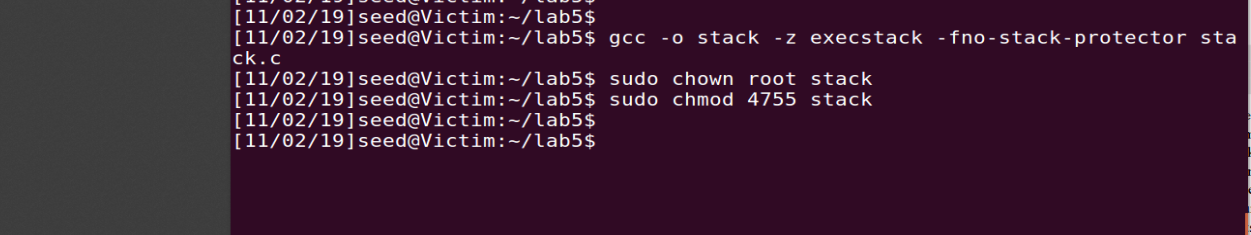
Stack was executable and address randomization was off.

Subtask 1.1: Compile and run the program. As program need “Badfile” , so create badfile with random string.



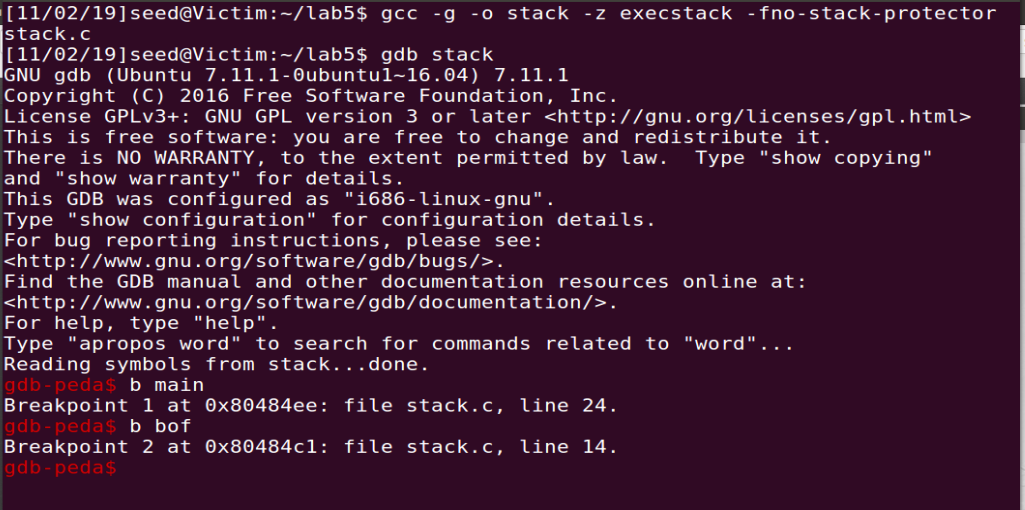
Observation: Program worked as expected, No buffer overflow, returned properly

**Task2: ﻿Exploiting the Vulnerability**

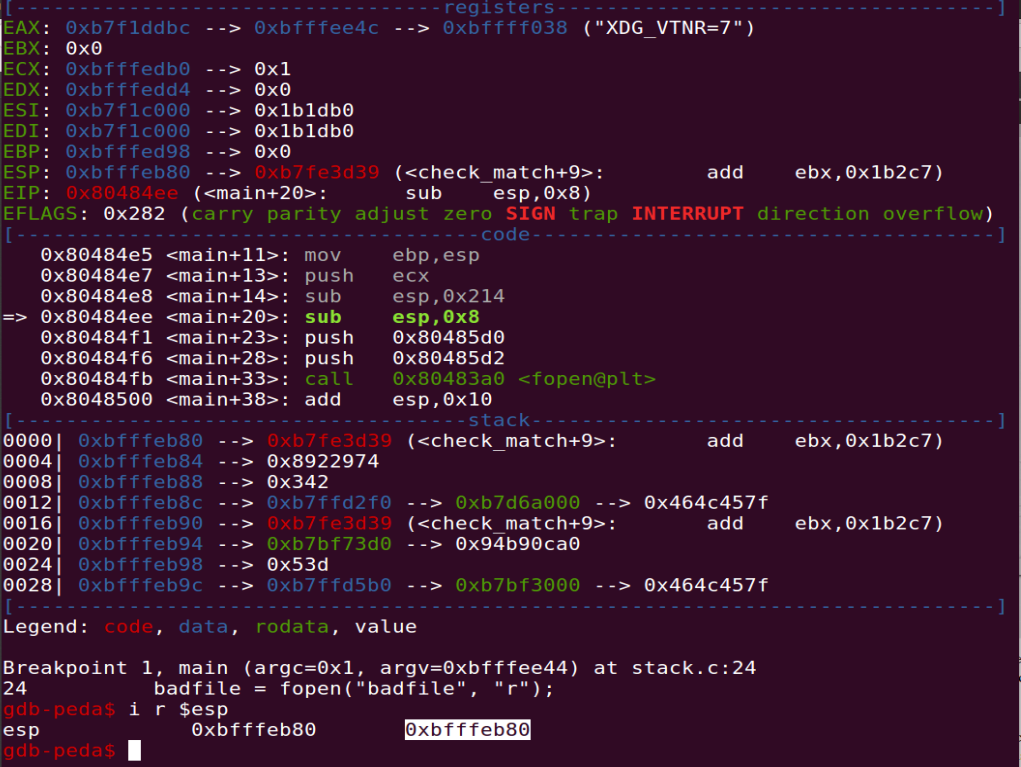


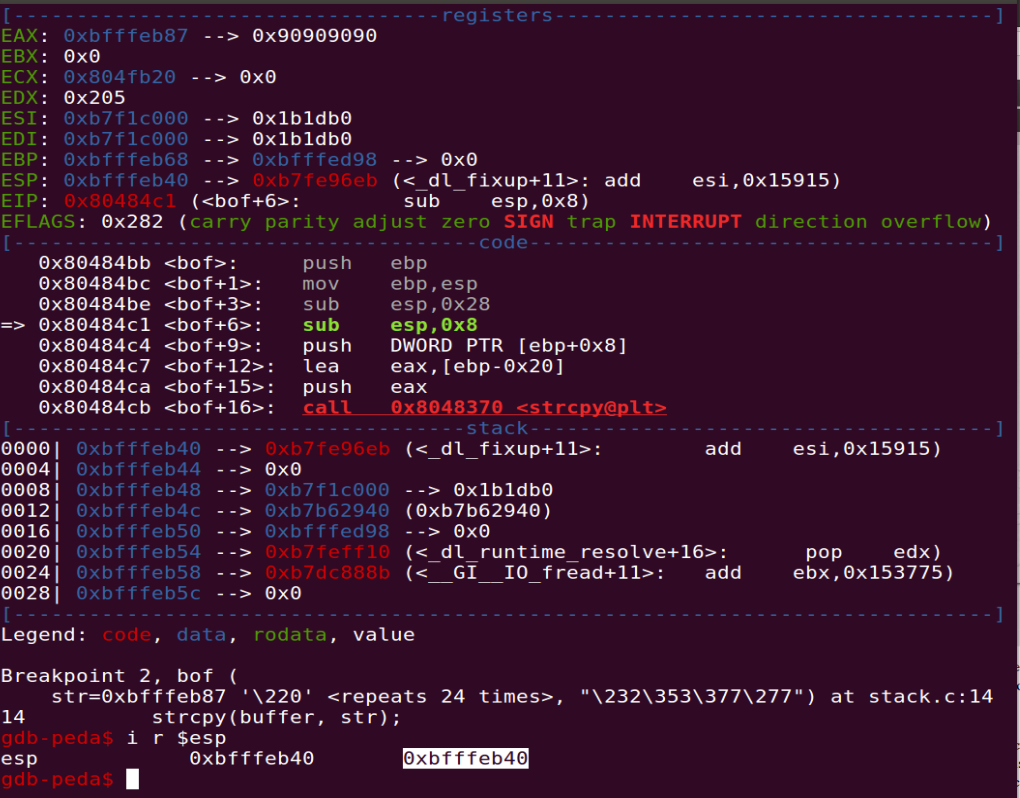
Run GDB(compile again with -g flag) to fetch address of stack pointer for “main” and “bof” function, which will tell the difference from “buffer” to the place where we can copy our shell code. And also that will be the address which we have to store at “Return address”.

Set break point on both function



Main 0xbfffeb80



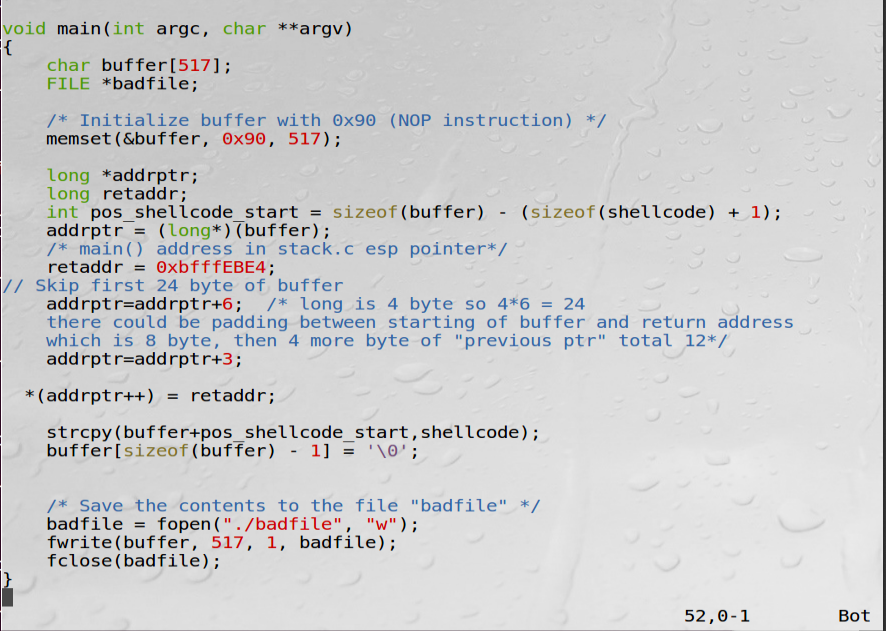


Difference between both stack base pointer is 64 ie (0xbfffeb80 - 0xbfffeb40).

It means our shellcode can be places after 24(array size in bof)+64(difference) = 88.

Main function esp: 0xbfffeb80 + 0x64(dec 100) will be destination address for shellcode. **=> 0xbfffEBE4**

**Using this create badfile from exploit.c**



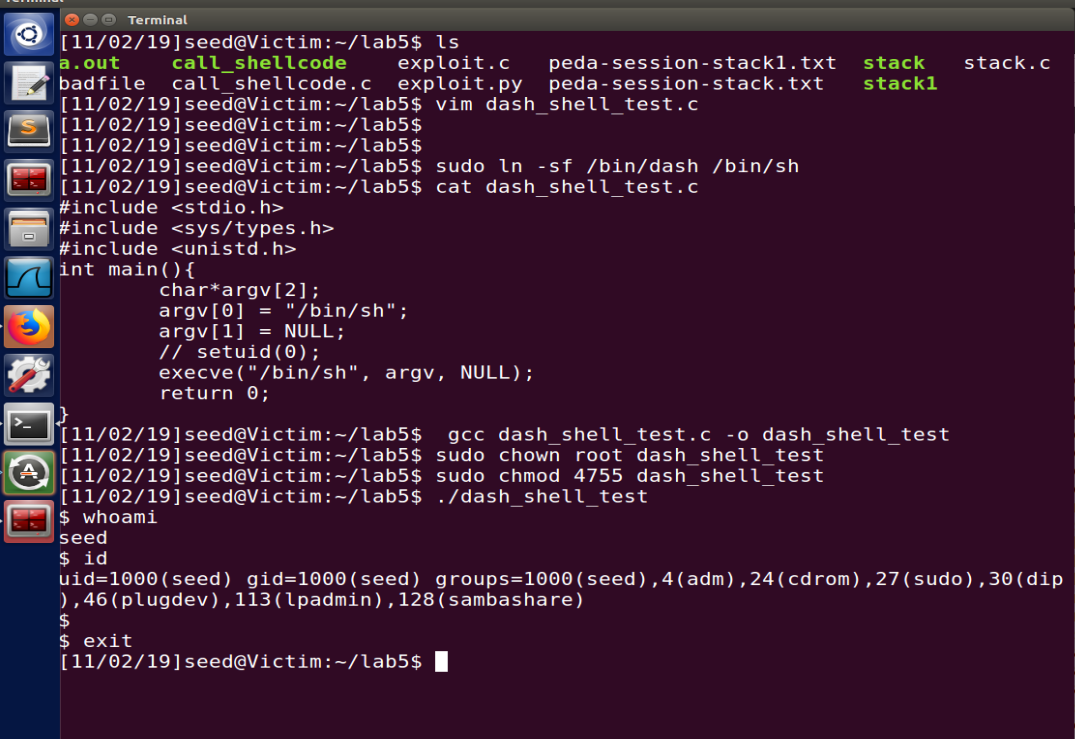


**Task3: Defeating dash’s Countermeasure**

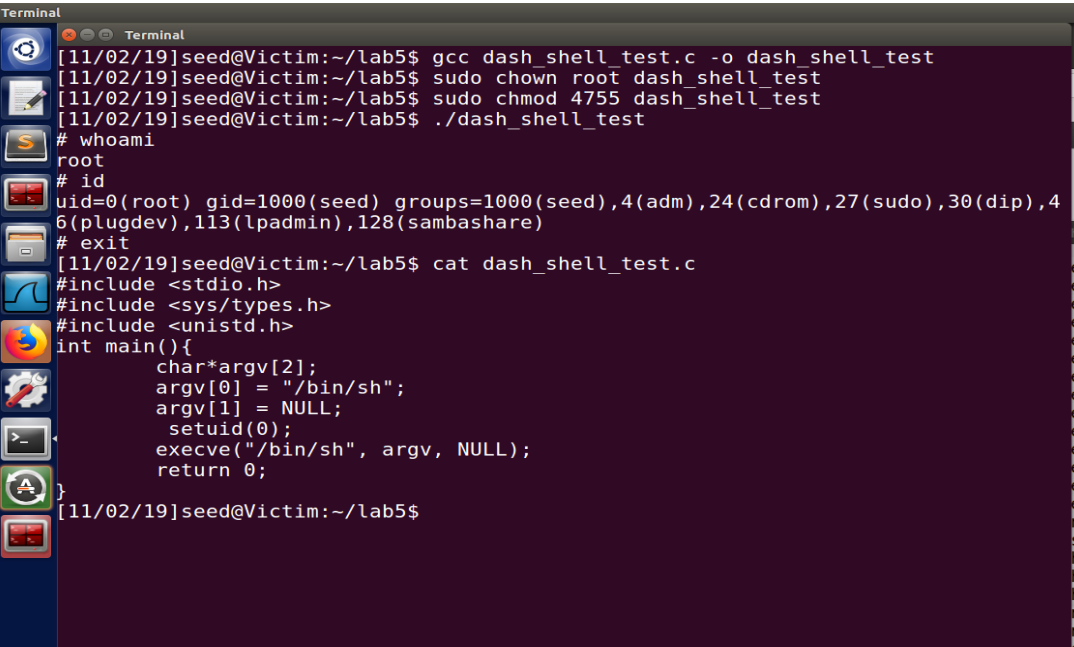
Lets run dash\_shell\_test program with setuid(0) commented and uncommented.

**Observation**: with setuid(0) commented, we didn’t get root shell despite file was owned by root as dash has counter measure to reduce the privilege.

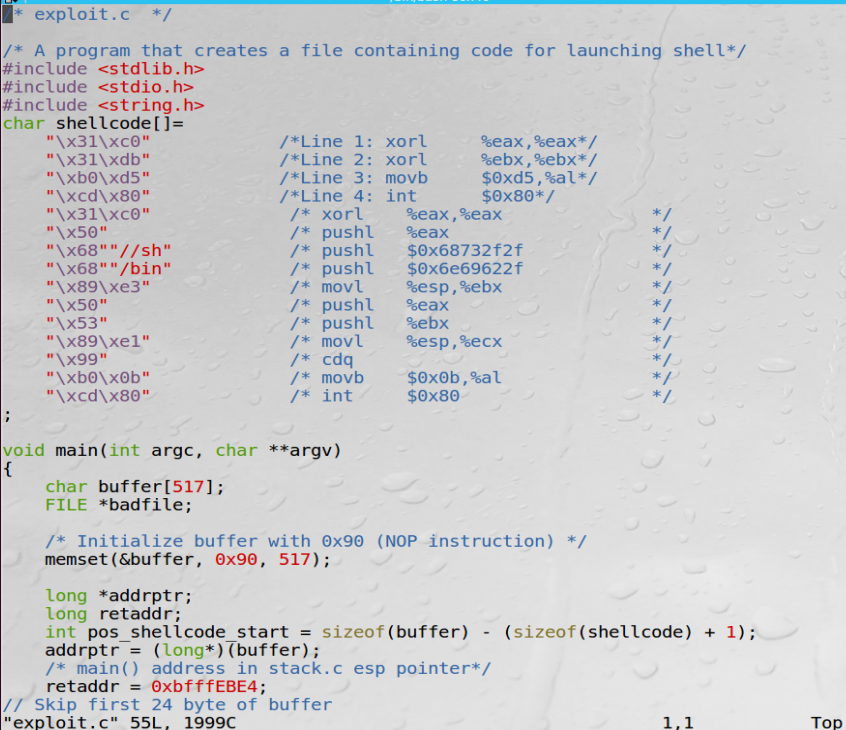
When we do setuid(0), we could get the shell with root privilege(ID 0).



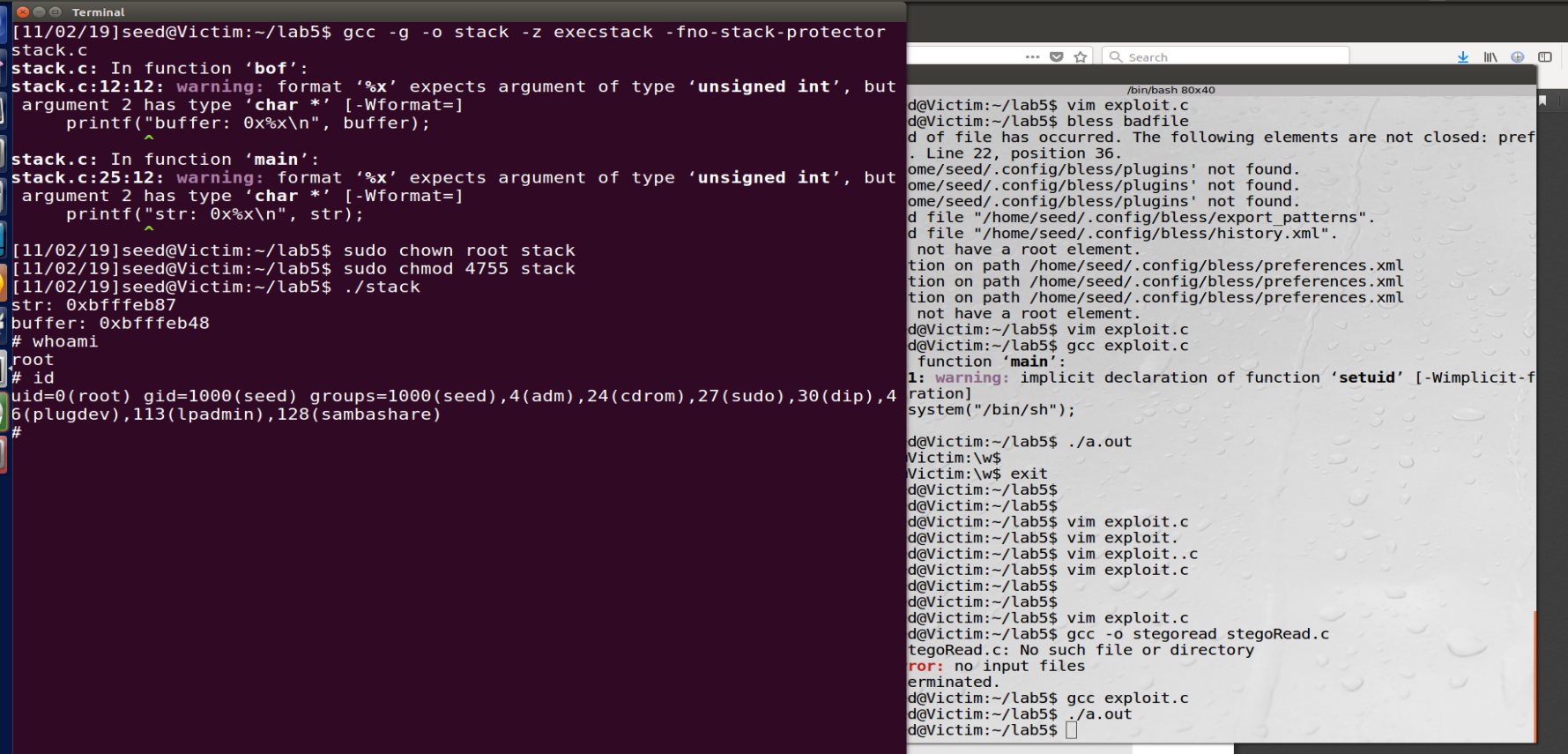
After enabling setid



**Add setuid code in the exploit**

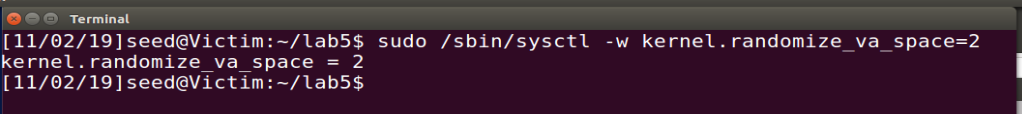


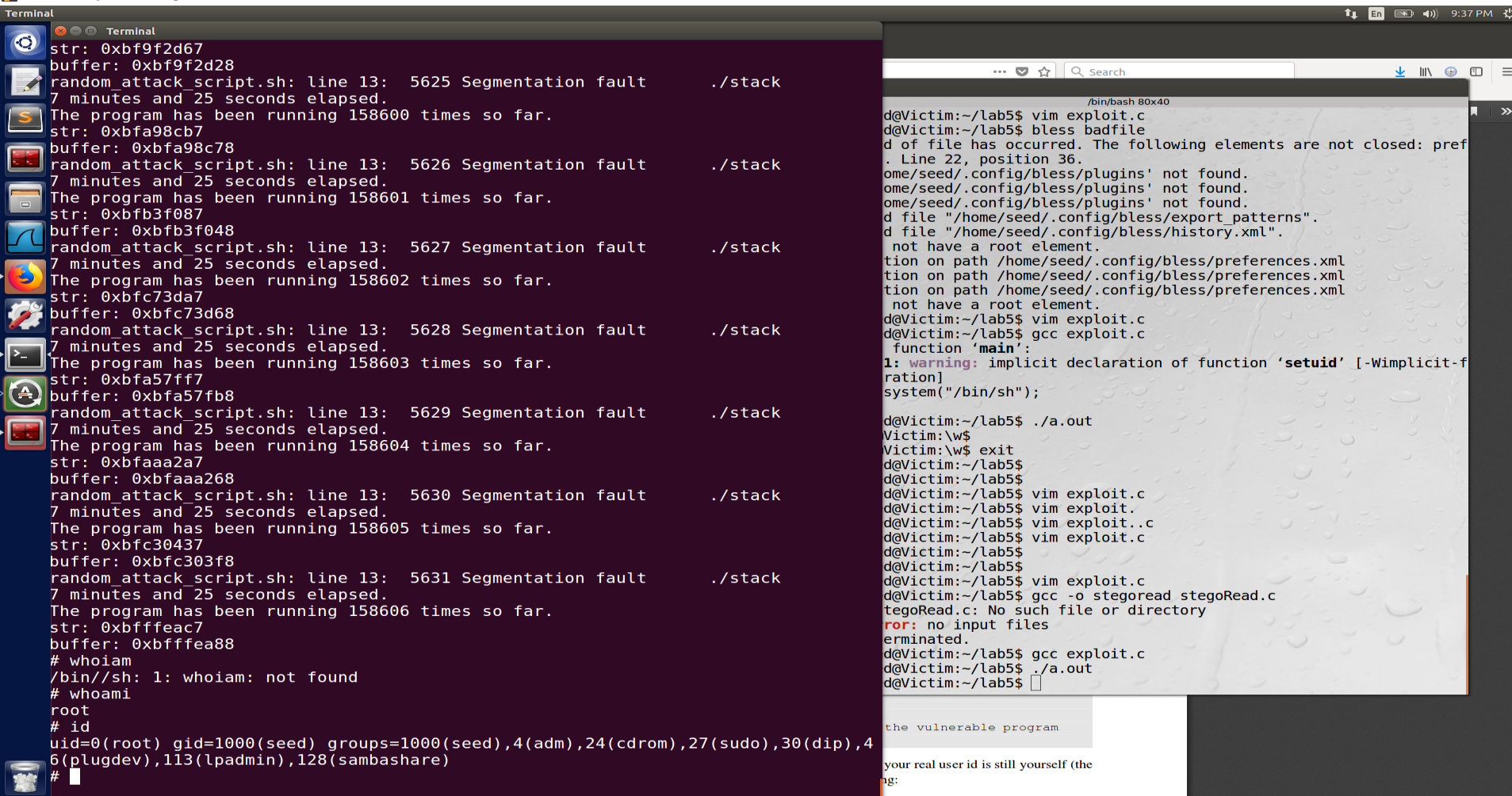
Build the badfile and execute stack program. As we can see in below screenshot, **privilege level is changed.**



**Task 4: Defeating Address Randomization**

Now lets enable address randomization. In this case our previous attack MAY not work as base address is changed. So we need to run script to execute attack again and again, may be we get lucky in few minute.





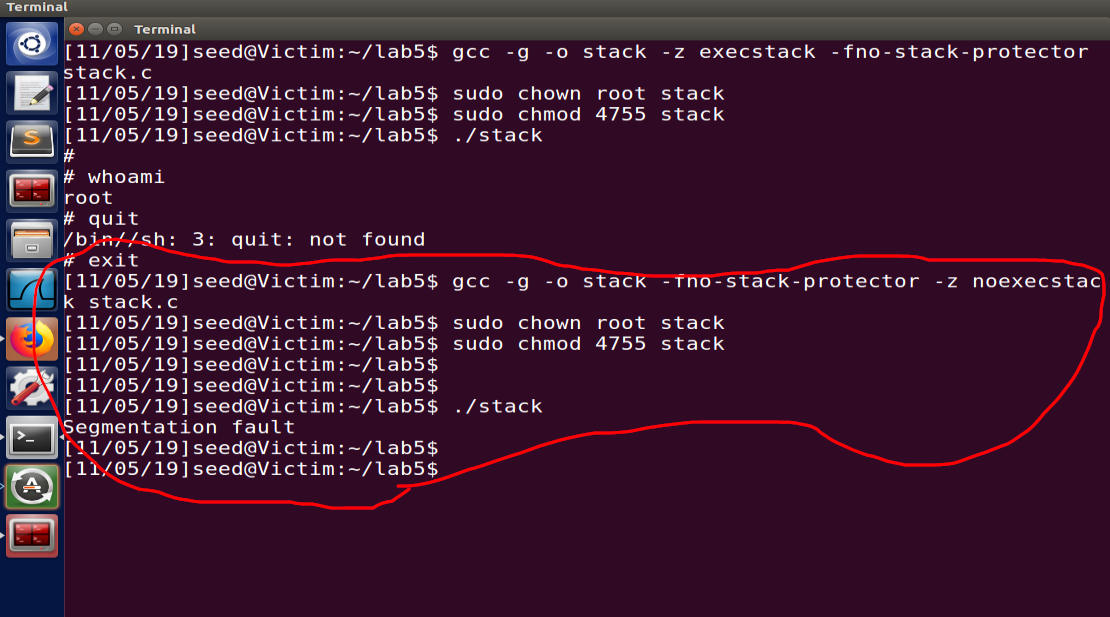
**Task 5: ﻿Turn on the StackGuard Protection**

Compile both program from task1 call\_shellcode.c and stack.c with stackgaurd off.



Observation: call\_shellcode is running properly as we didn’t modify stack while stack.c safeguard detected the stack tempering, hence it aborted

**Task 6:** ﻿**Turn on the Non-executable Stack Protection**



Above screen shot shows both execution of “executable and non-executable” stack.

﻿

Can you get a shell? If not, what is the problem?

Answer: No, we didn’t get the shell. Buffer overflow did happen but the address it was pointing was non executable hence we got segfault.

﻿How does this protection scheme make your attacks difficult?

Answer: we wrote Return address to our shell code, but that location in stack is not executable. Now to achieve that we need to write return address of some other library in system like libc. And also, we need to inject code in that library, which is more difficult. Another very minute reason could be that each machine runs a different version of library so offset could be different, Hence make more difficult to exploit.