Created by: **Eduardo Argueta Cantizzano**
Vulnerability: **Buffer Overflow**
Date: **03/16/2020**

Buffer Overflow Vulnerability

BUFFER OVERFLOW:

A buffer overflow occurs when a program writes more data to a buffer than it can hold. Buffers are areas of memory allocated to store data temporarily. When data exceeds the buffer's capacity, it overwrites adjacent memory, which can lead to unpredictable behavior, crashes, or security vulnerabilities.

How Buffer Overflows Occur

1. **Fixed-Size Buffer**: A buffer of fixed size is allocated in the memory to hold data.
2. **Excess Data**: When the program writes data to the buffer without checking if it fits, the excess data spills over into adjacent memory locations.
3. **Overwriting Memory**: The overflow can overwrite other important data, including program variables, return addresses, or function pointers.

Address Space Randomization

The address space randomization is turned off so that our program has the same addresses each time it is executed.

StackGuard Protection

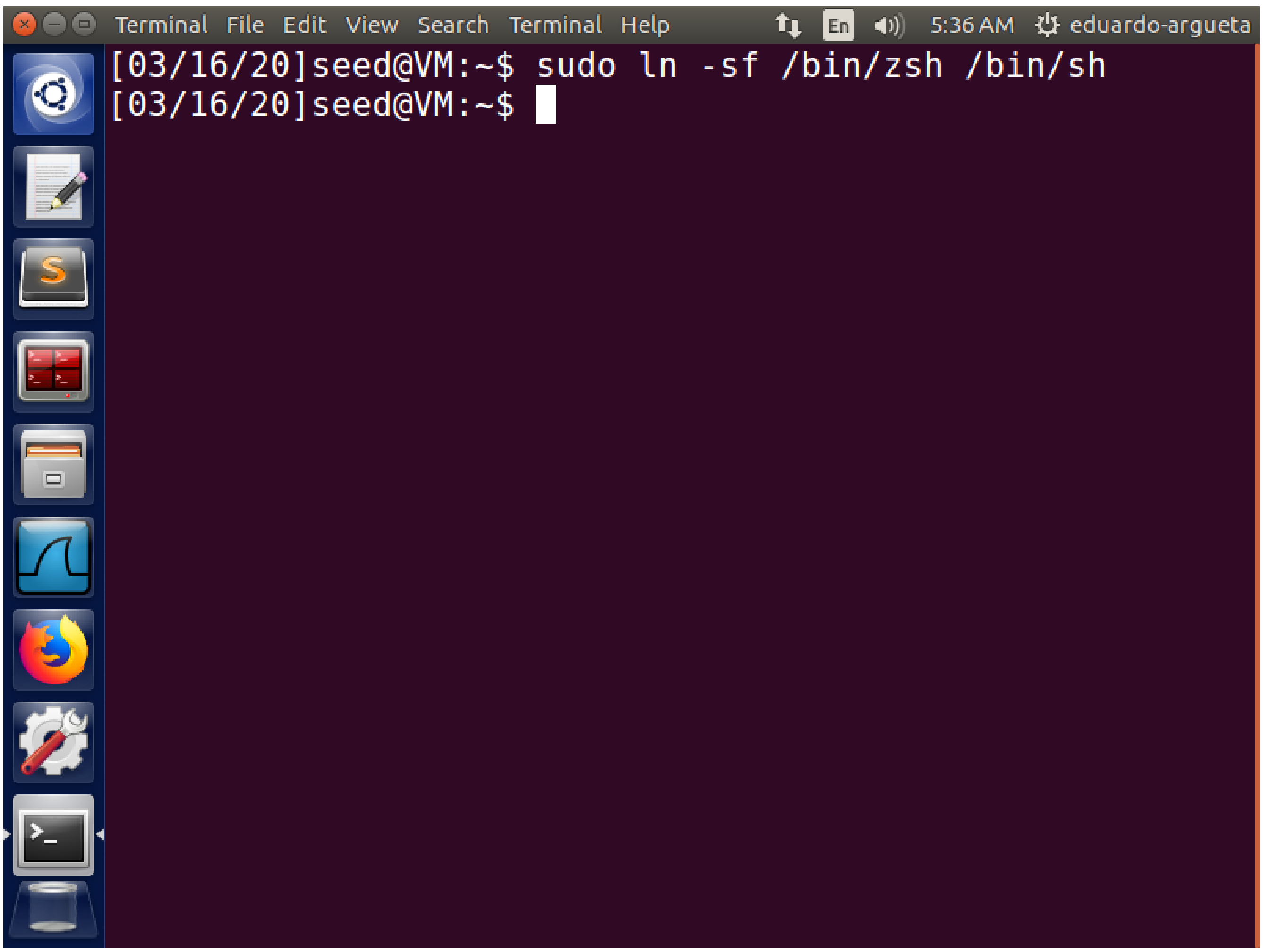
The vulnerable program is compiled with –fno-stack-protector argument so that protection against buffer overflow is disabled.

Non-Execution Stack

The vulnerable program is compiled with -z execstack argument so that the stack is executable.

Configuring /bin/sh

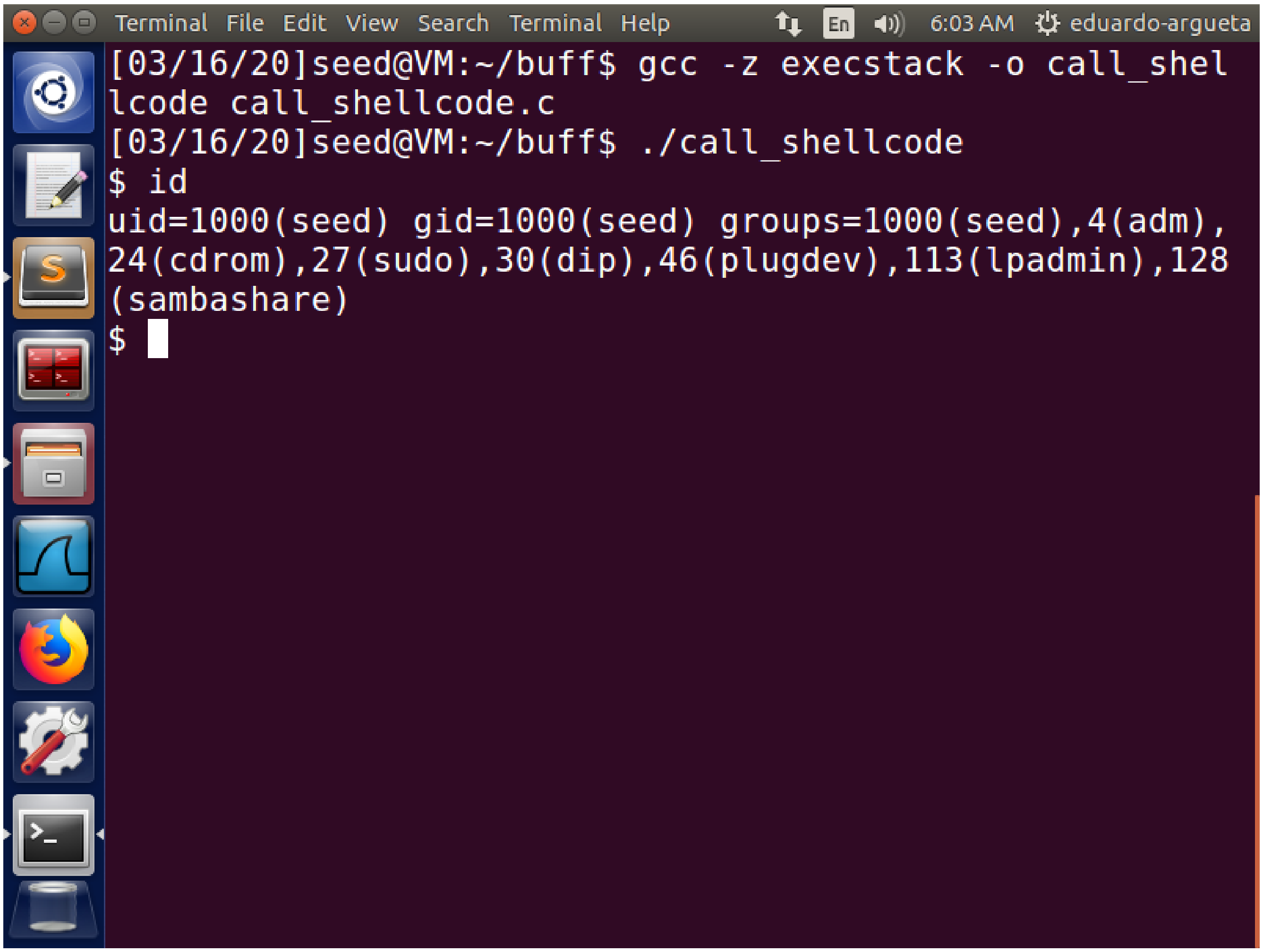
/bin/sh to another shell which does not have countermeasure that prevents it from being executed in set-UID process. For this, it is linked to zsh



TASK 1

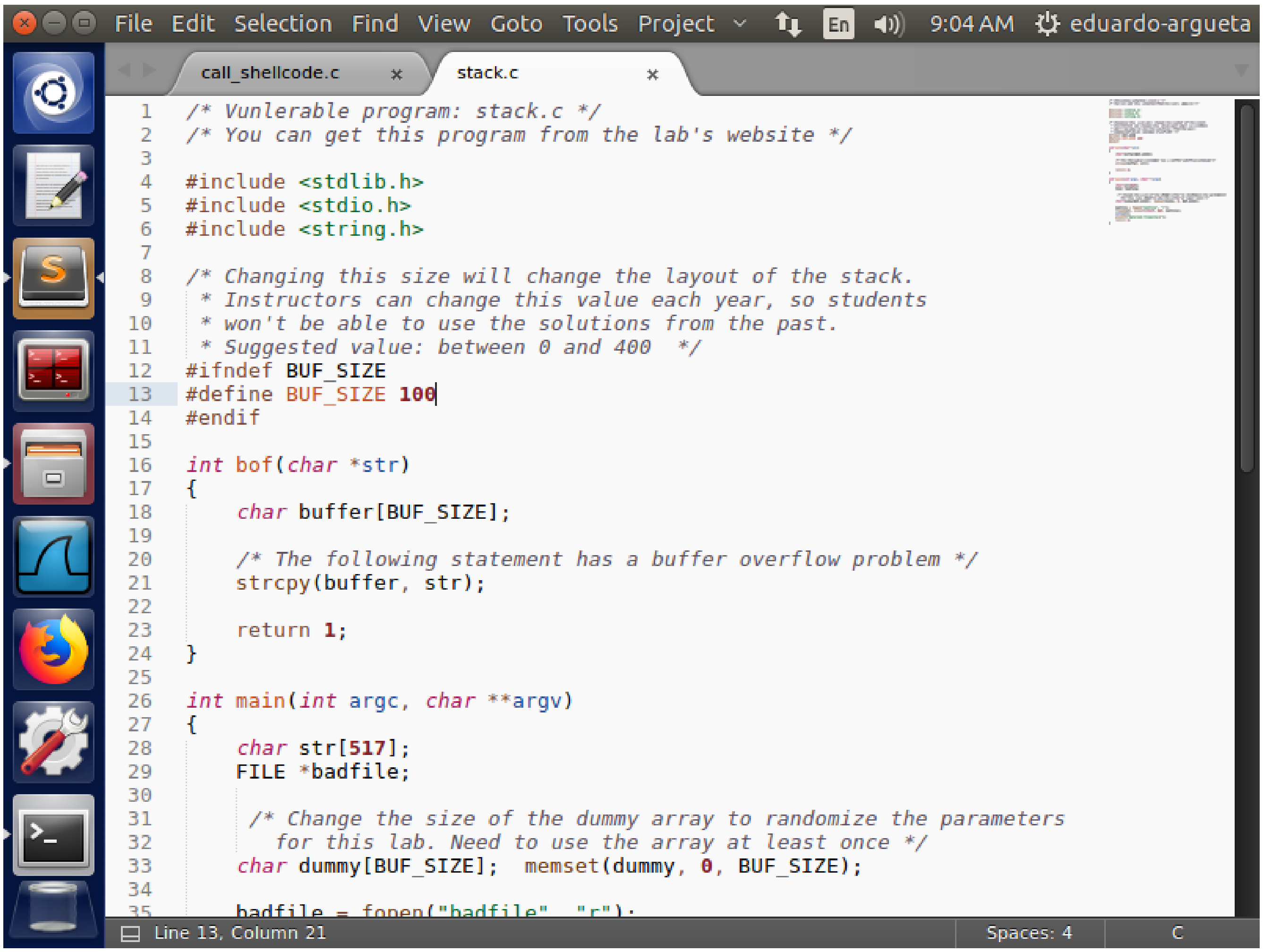
Running the shellcode

The program call_shellcode.c is compiled and executed to check if a shell is invoked. As seen in the screenshot the shell is invoked with uid=1000

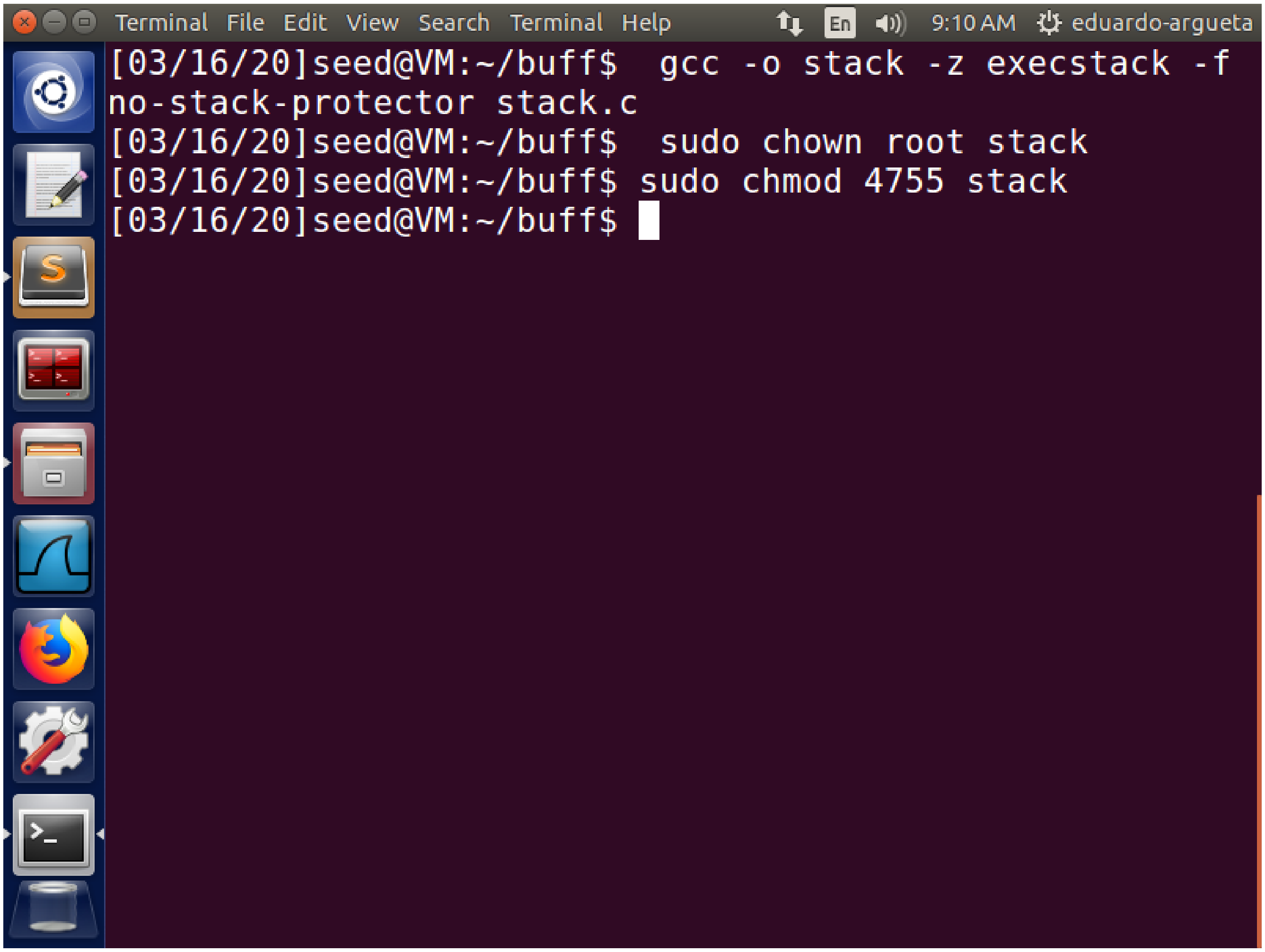


Compiling the vulnerable program

The program stack.c is compiled with the countermeasures turned off and BUFF_SIZE changed to 100



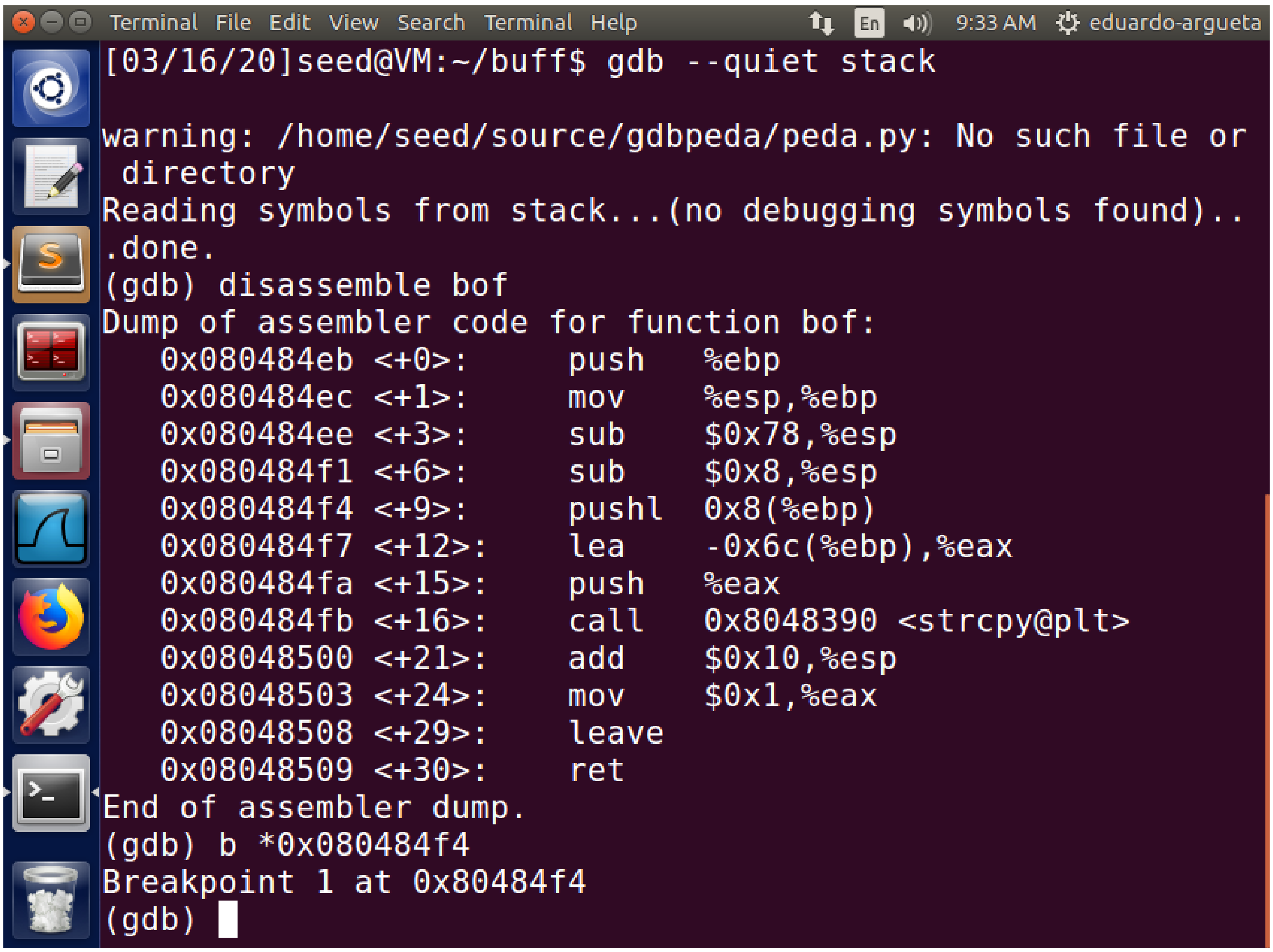
The BUF_SIZE is changed to 100 as stated in the instructions.

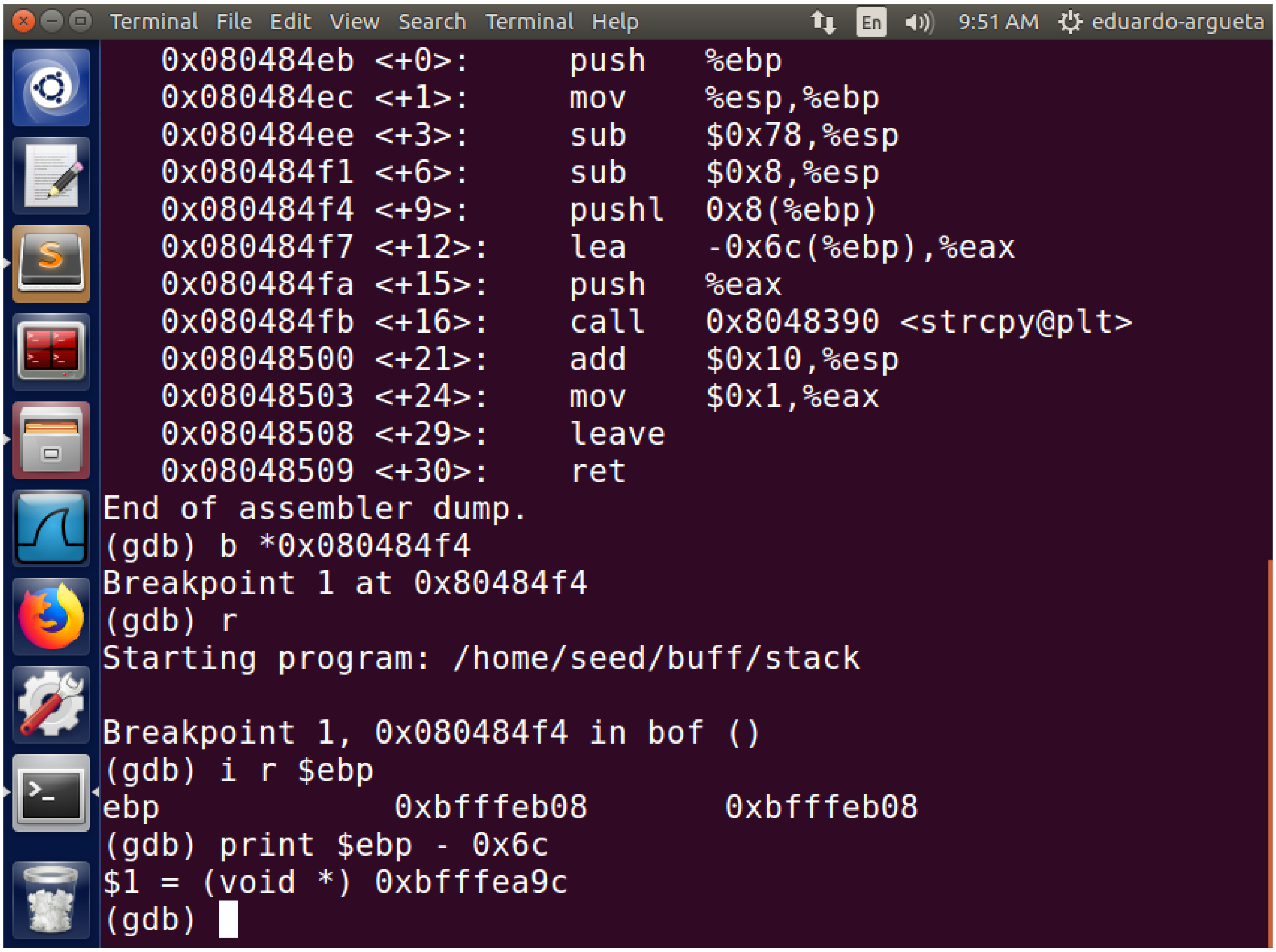


The ownership of stack program is changed to root and set-UID bit is turned on.

Finding the base pointer address and changing exploit.c

The stack.c program is examined with gdb and the address of base pointer is located by examining the disassembly of bof function. The dissembly shows that the buffer address is located -0x6c bytes below the base pointer address. Since the return address is right next to base pointer address (4 bytes below), this means that the return address is 108 (decimal of 0x6c) + 4 bytes = 112 bytes above the starting address of buffer.

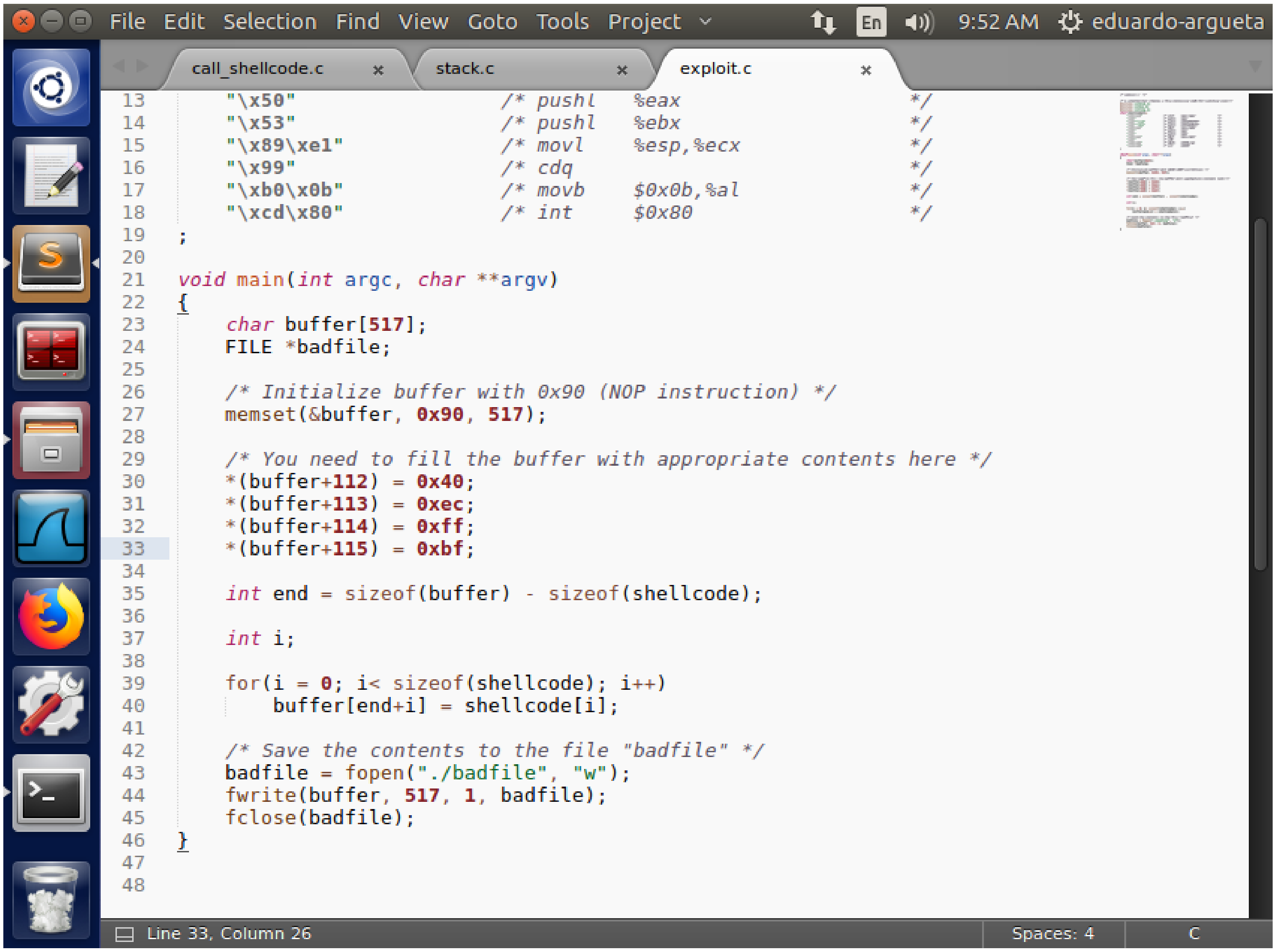




The base pointer address is 0xbfffeb08, we now we choose an address that will bring the execution to one of the NOPs inside the badfile from where it will go to the shellcode we want to execute. From this we go to 0xbfffec40 which is 312 bytes above the base pointer address and will certainly be inside the NOP instructions. This is done because addresses shown in gdb are slightly off from the addresses during execution.

TASK 2:

Exploiting the Vulnerability



The address 0xbfffec40 is placed at 112 bytes from the start of buffer as this was calculated to be the address of the place where return address is stored. So now, the return address will contain 0xbfffec40 and the program will jump to one of the NOP instructions from where it will go to our shellcode.

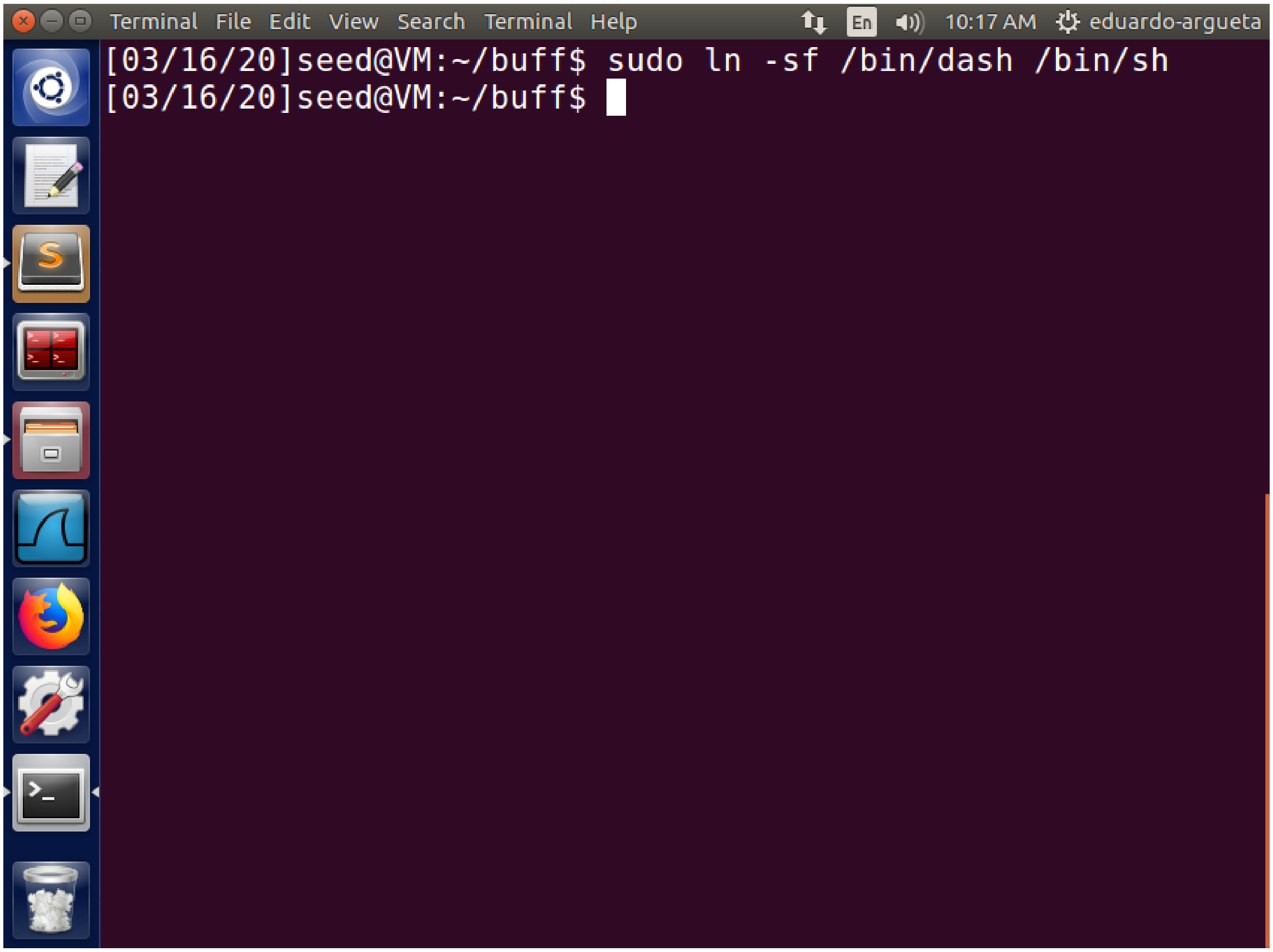
Executing the vulnerable program to obtain shell

After compiling and executing both exploit,c and then executing stack, a root shell is successfully obtained. The shell has effective UID = 0 (root)

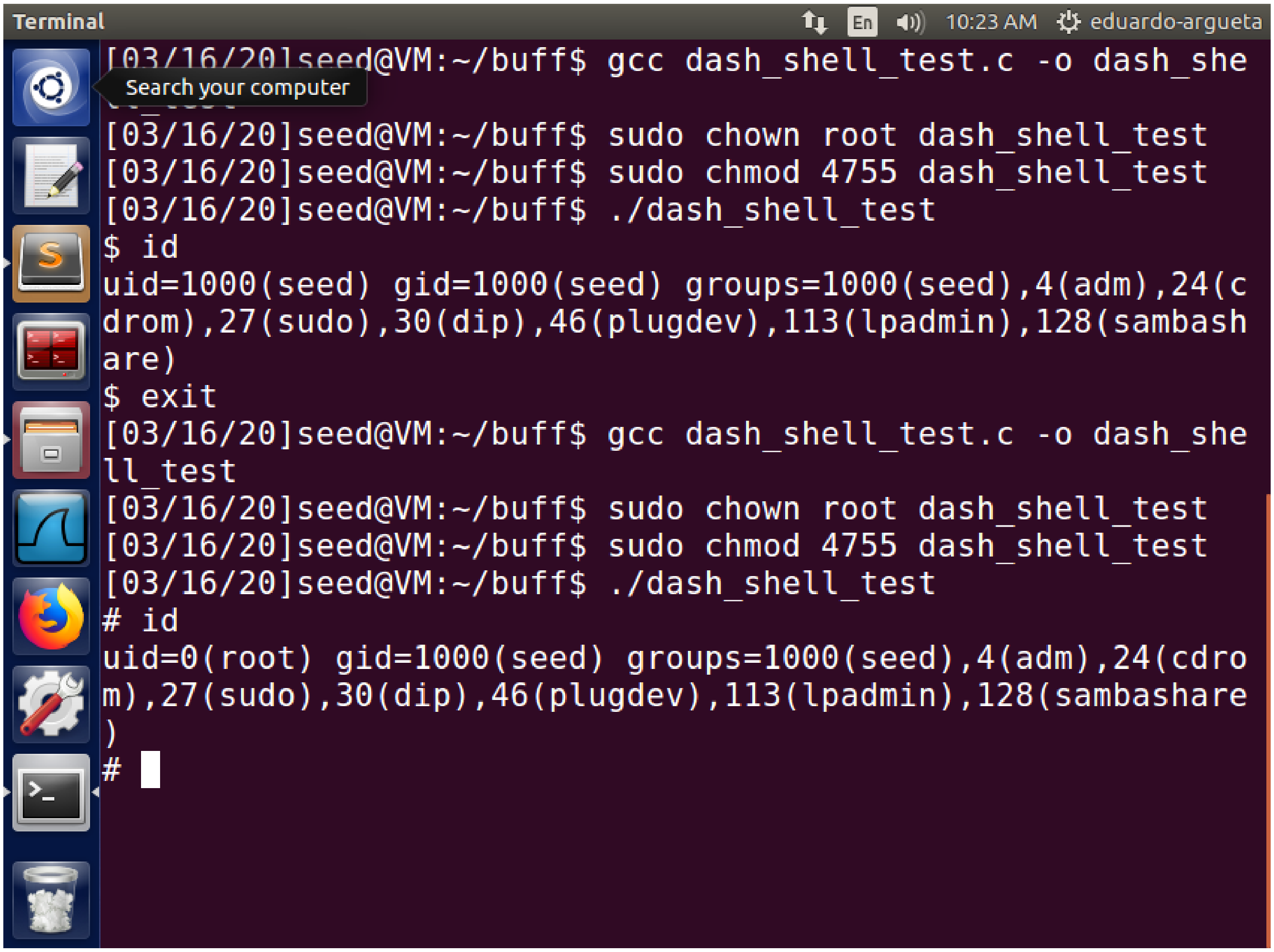
TASK 3:

Defeating dash's countermeasure

First we revert the changed we did to the symbolic link of /bin/sh, so that now it points back to /bin/dash

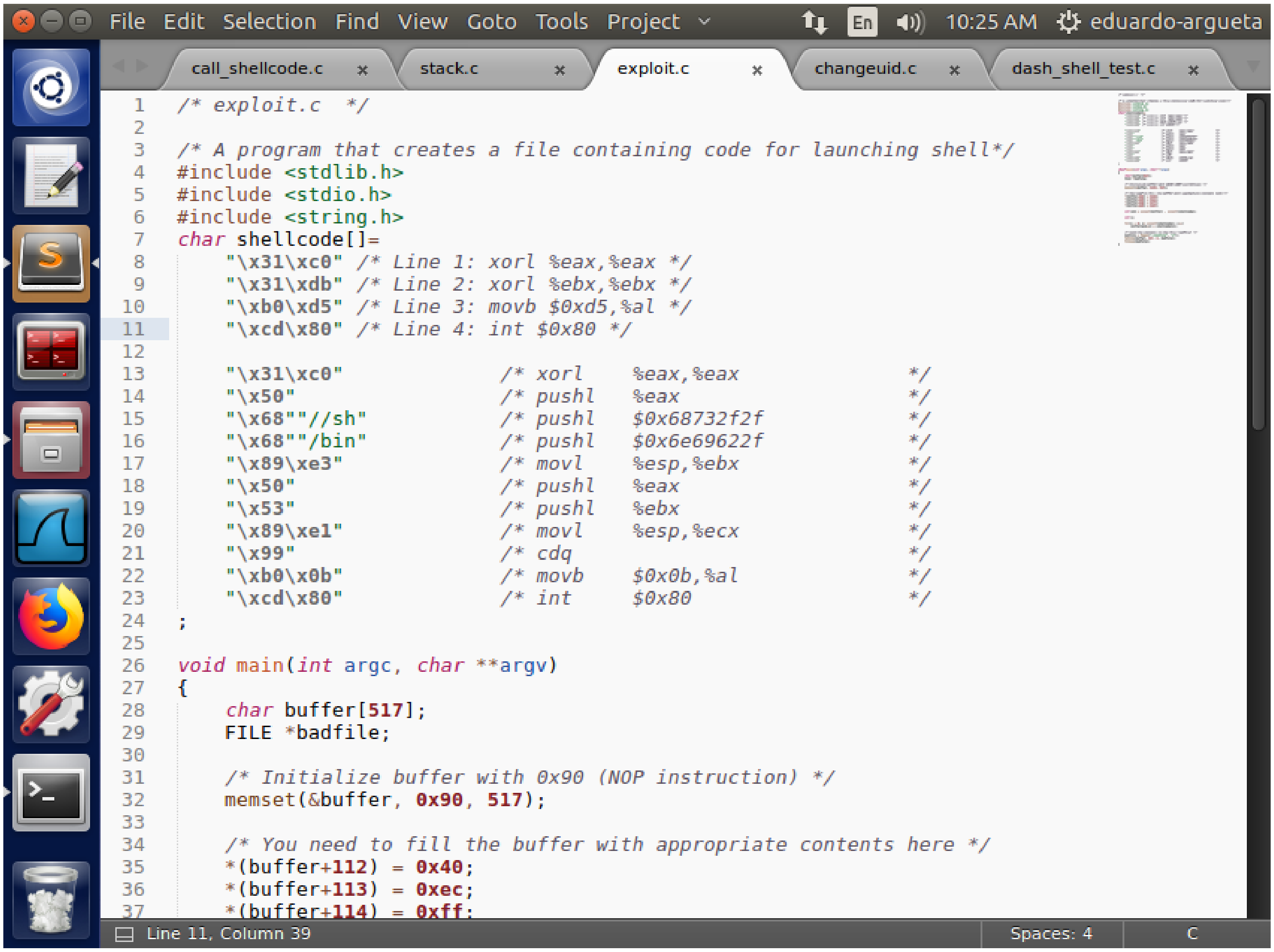


Executing dash_shell_test.c



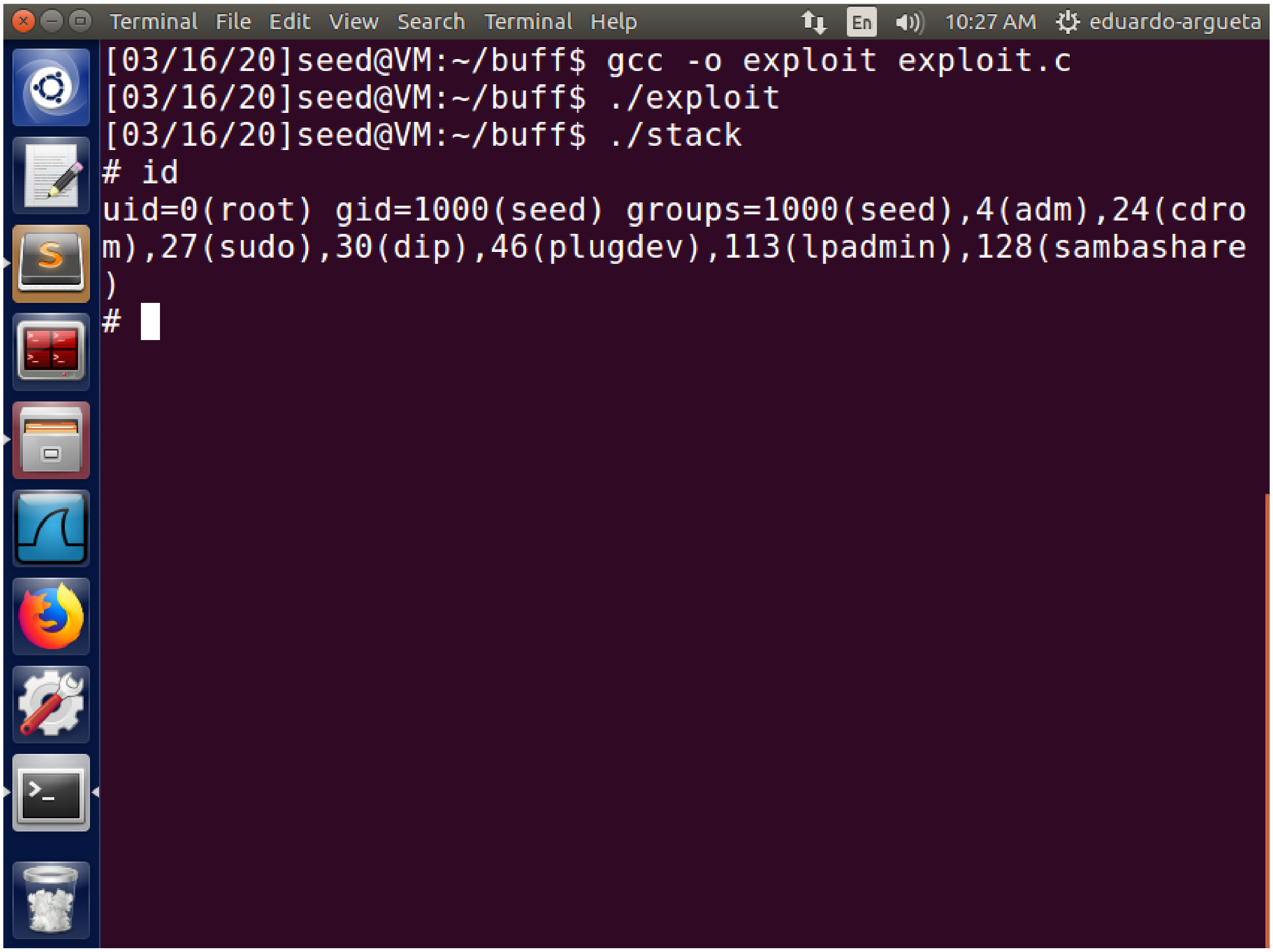
Without the setuid(0) call the shell has uid = 1000 and is not a root shell. With setuid(0) call the program has uid=0 and is a root shell.

Adding setuid(0) to our shellcode



```c
/* 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" /* Line 1: xorl %eax,%eax */
    "\x31\xdb" /* Line 2: xorl %ebx,%ebx */
    "\xb0\xd5" /* Line 3: movb $0xd5,%al */
    "\xcd\x80" /* Line 4: int $0x80 */

    "\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 */
    *(buffer+112) = 0x40;
    *(buffer+113) = 0xec;
    *(buffer+114) = 0xff;
```

The top 4 lines are added to execute setuid(0) before the rest of the program is executed ((0xd5 is setuid() system call number)

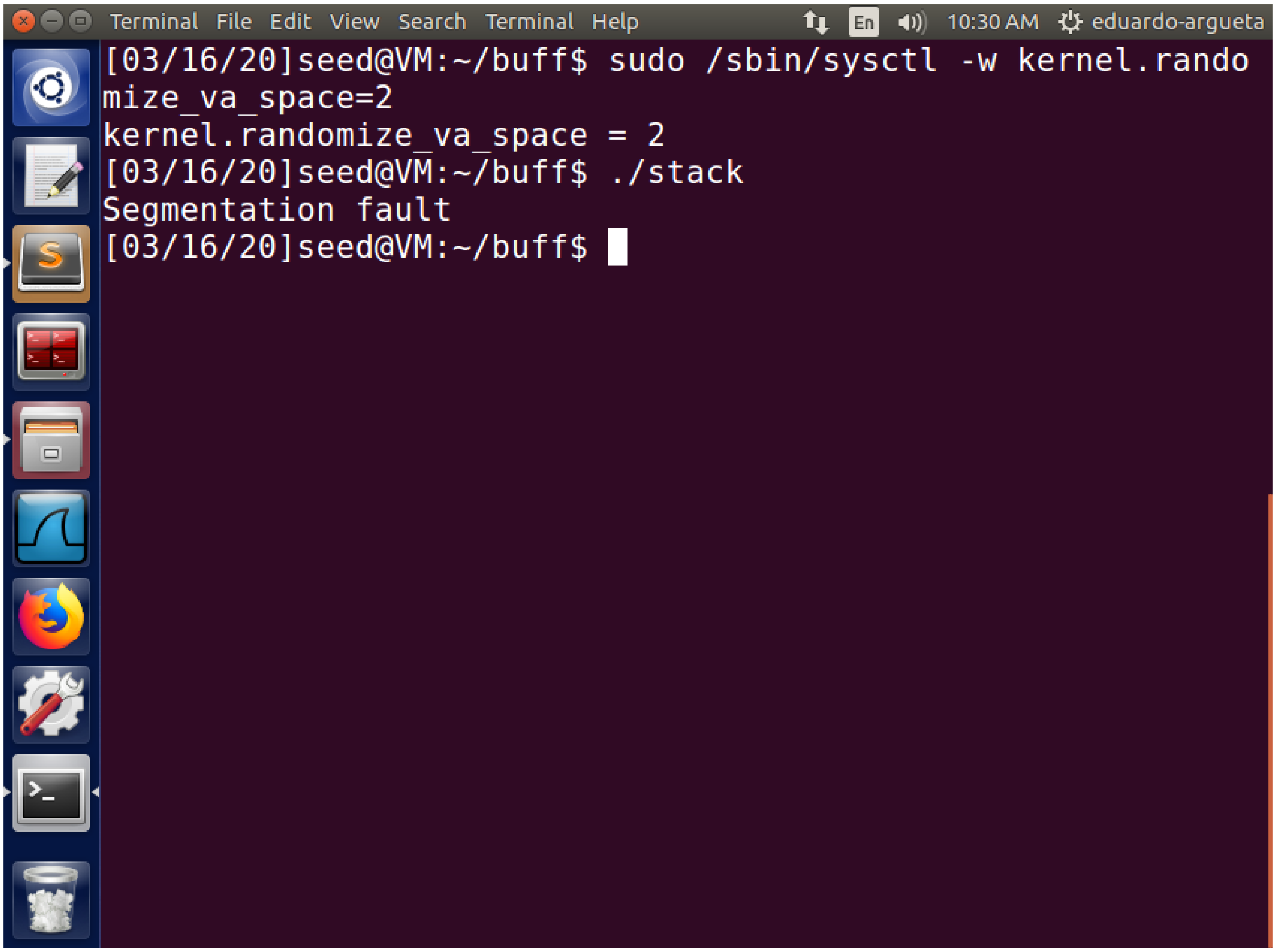


After changing the shellcode to include setuid(0), the uid of the resulting shell is 0 (root) as compared to 1000 (seed). The program now has real user id of root

TASK 4:

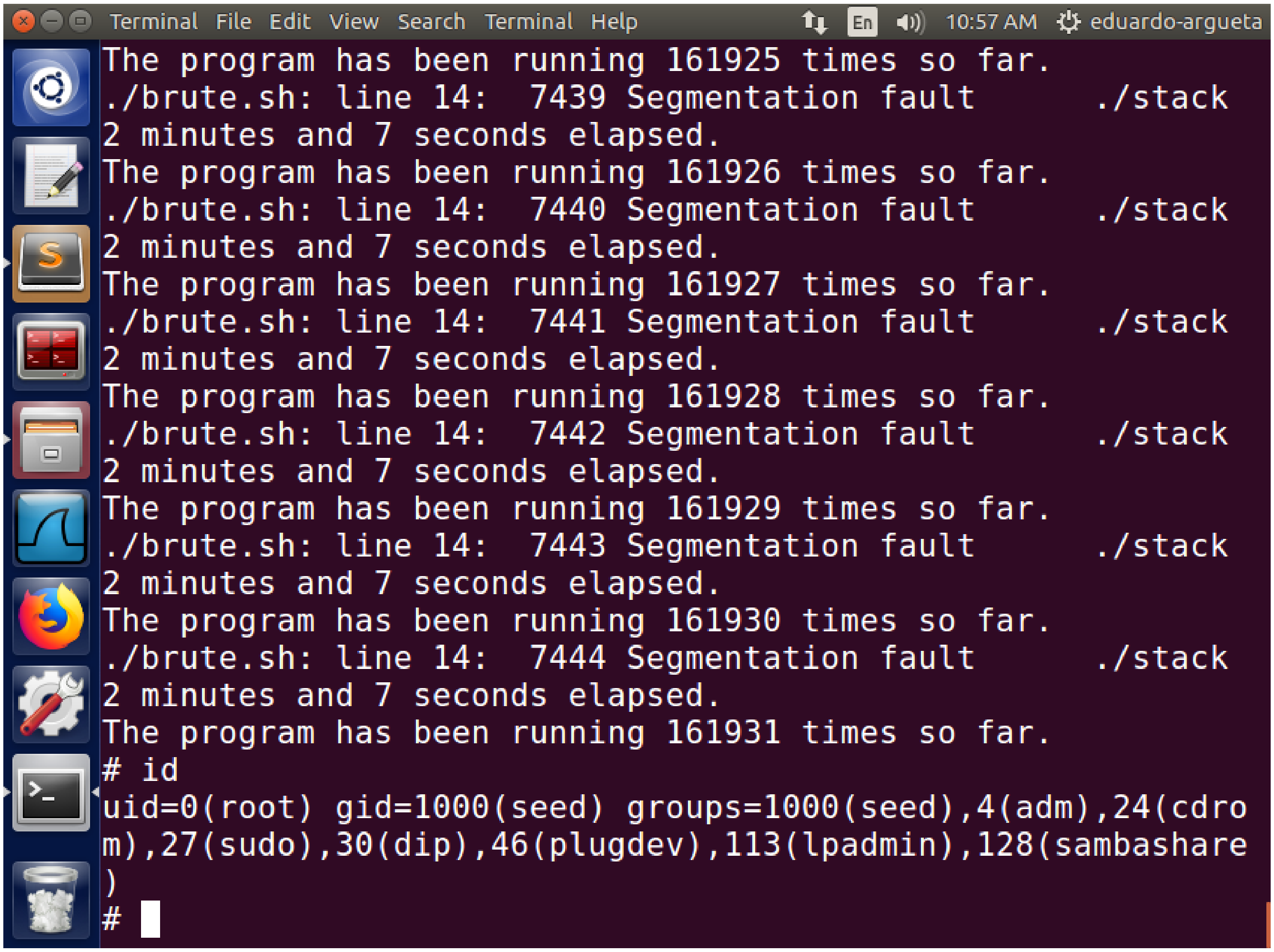
Defeating Address Randomization

The Address randomization is turned on for this.



Executing the stack program after this results in Segmentation fault because the address of the program stack has changed and the addresses provided in the badfile do not contain a valid execution address.

Using Bruteforce

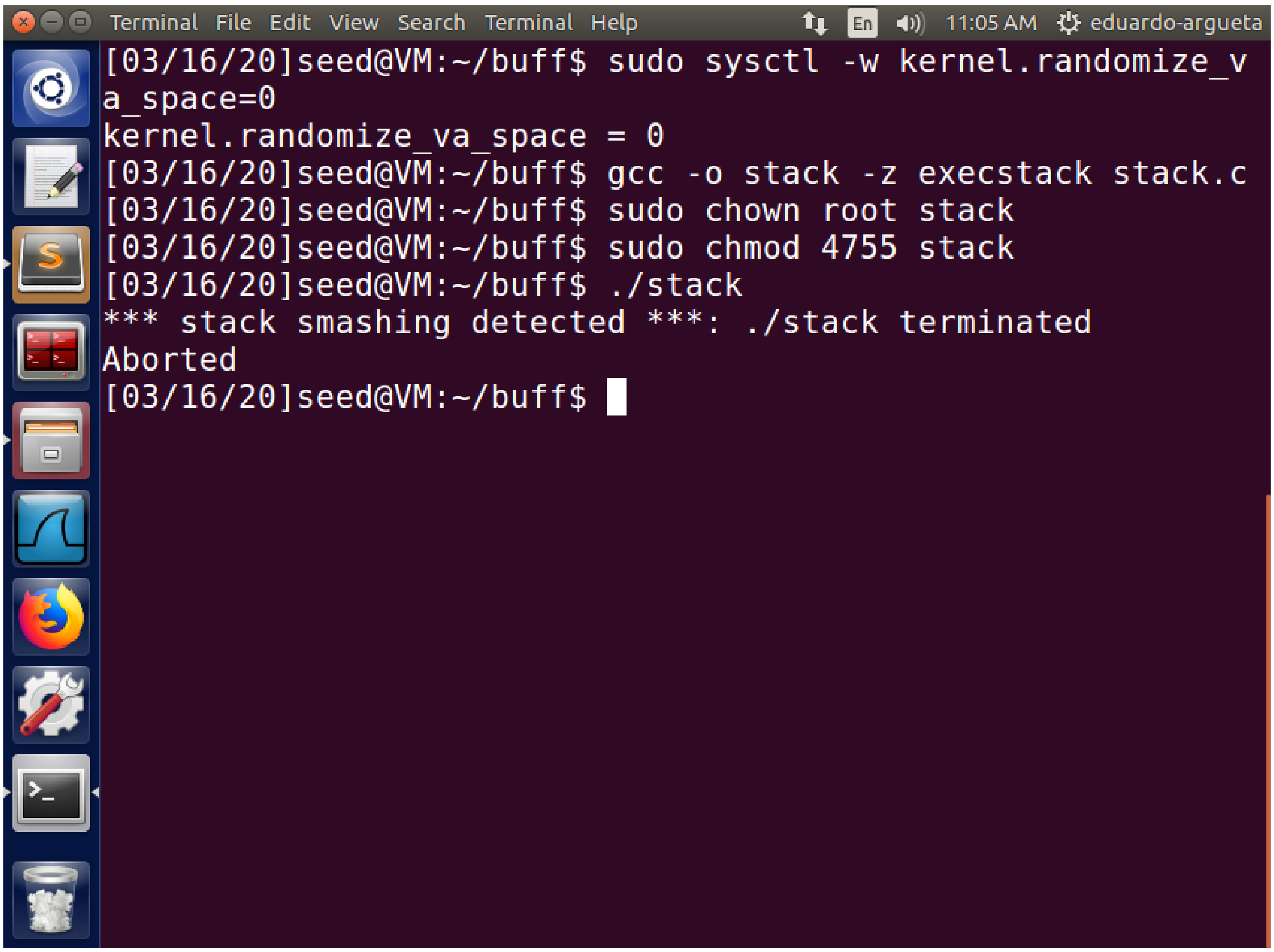


If keep executing the program with Address randomization turned on, we will eventually be able to land on an address which we intend to. The program is executed in a while(1) loop and after running for 161931 times the shell was invoked. So it can be seen that the Address randomization is not very effective.

TASK 5:

Turning on StackGuard Protection

Recompiling and executing the program with stackGuard turned on.



The program detects that there was an attempt to smash the stack (overwrite the values written in stack) and terminates the program. The stackGuard protection places a random integer in the stack and at any time during execution if that integer is overwritten, it detects it and aborts the program.

TASK 6:

Turning on the Non-executable Stack Protection

The program is recompiled and executed with non-executable stack turned on

Running the program results in Segmentation fault since the shellcode sitting at the return address is executable and stack execution has been disabled. This measure does not prevent buffer overflow as we are still able to go to the address we wrote inside the stack but cannot execute it. If we write an address outside the stack, then it will be executable.

Explanation on how exploit.c and the modification of the values to get the correct return addres.

As I already went thru on the section of TASK 1 modify the stack.c

The stack.c program is examined with gdb and the address of base pointer is located by examining the disassembly of bof function. The dissembly shows that the buffer address is located -0x6c bytes below the base pointer address. Since the return address is right next to base pointer address (4 bytes below), this means that the return address is 108 (decimal of 0x6c) + 4 bytes = 112 bytes above the starting address of buffer.

The base pointer address is 0xbfffeb08, we now we choose an address that will bring the execution to one of the NOPs inside the badfile from where it will go to the shellcode we want to execute. From this we go to 0xbfffec40 which is 312 bytes above the base pointer address and will certainly be inside the NOP instructions. This is done because addresses shown in gdb are slightly off from the addresses during execution.

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

   The program gives "Segmentation Fault" error because the program is directed to an address where execution code is placed and by default stack is non-executable

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

   When ASLR is enabled, the program is loaded to a different address every time it is executed. Hence the return address will be changed and the program will not jump to the address we want it to.

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

   The address of buffer[] in memory is slightly different when using GDB as compared to executing it like ./stack. This is because GDB alters the behavior of the program slightly for its own functioning.