Jorge Avila

Professor Trey

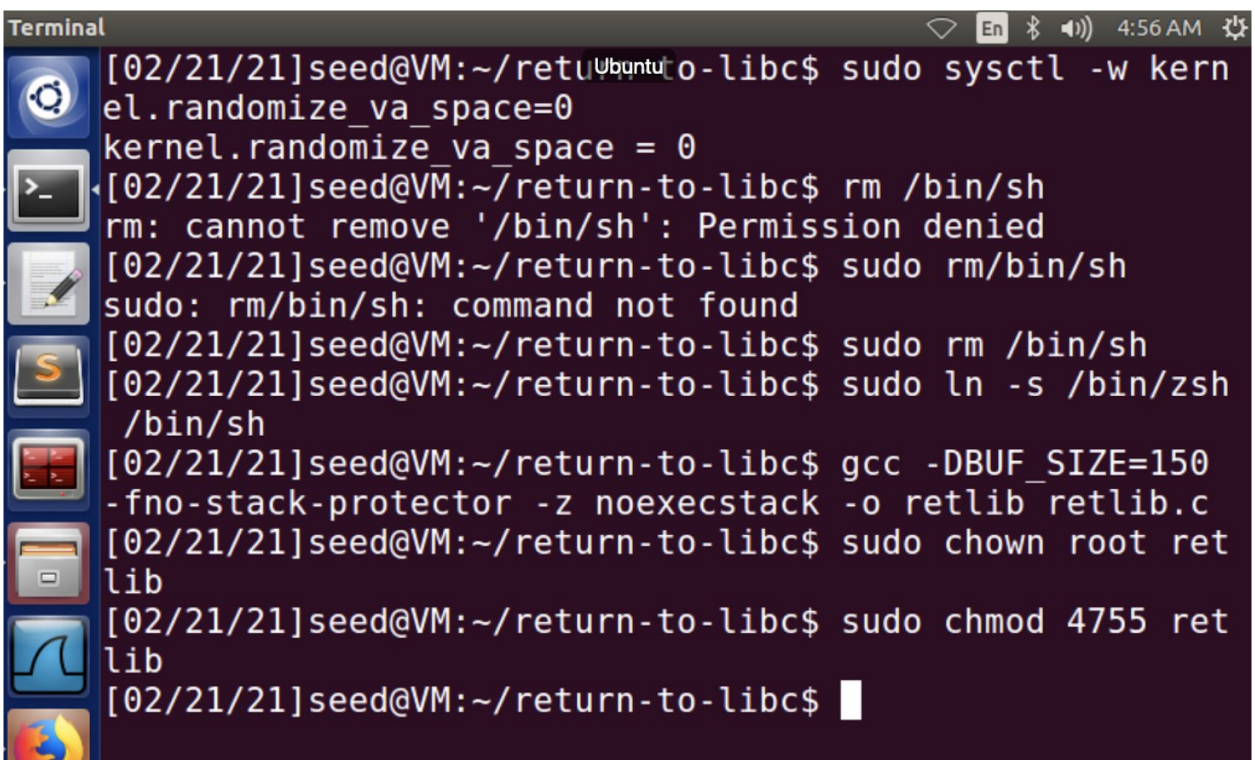
Secure Programming

12 March 2021

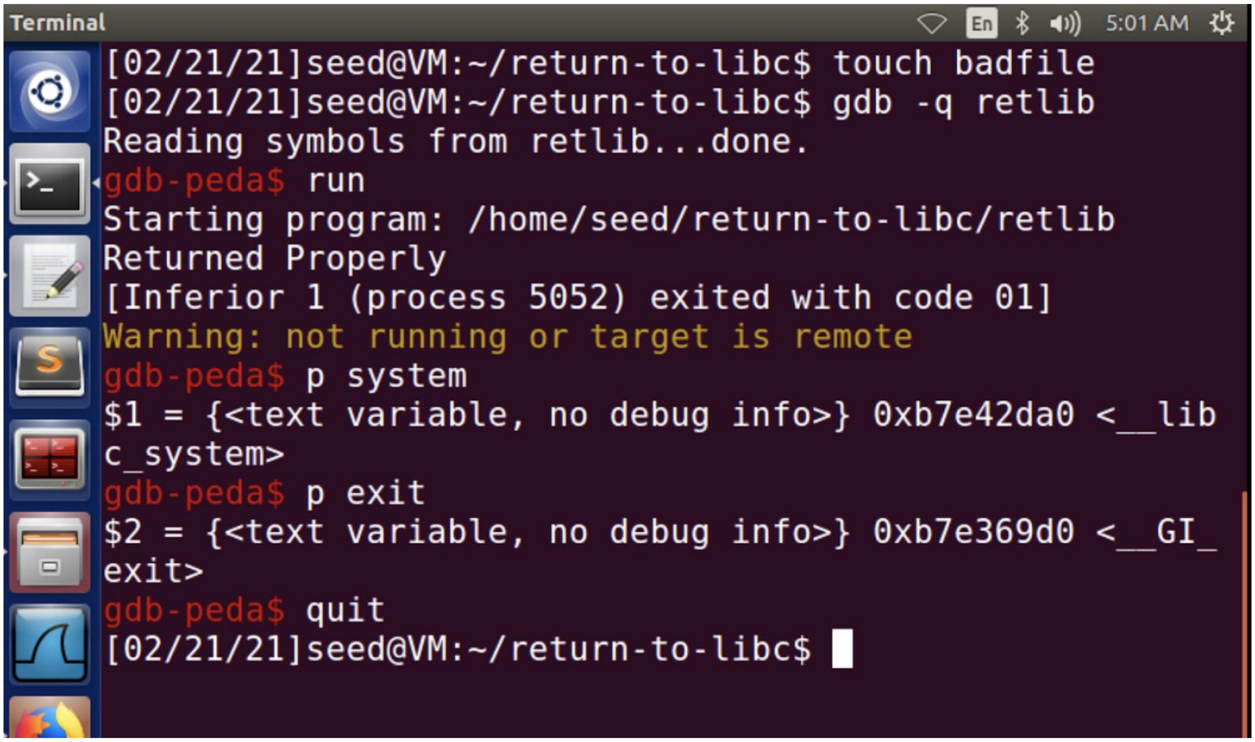
Return-to-libc Attack

<u>Task 1:</u>

In this task we are going to set the randomize equal to 0, link the shell to point to something that does not have the countermeasure, recompile our program of retlib.c with the following buffer size of 150 and make it a set-UID as the following:



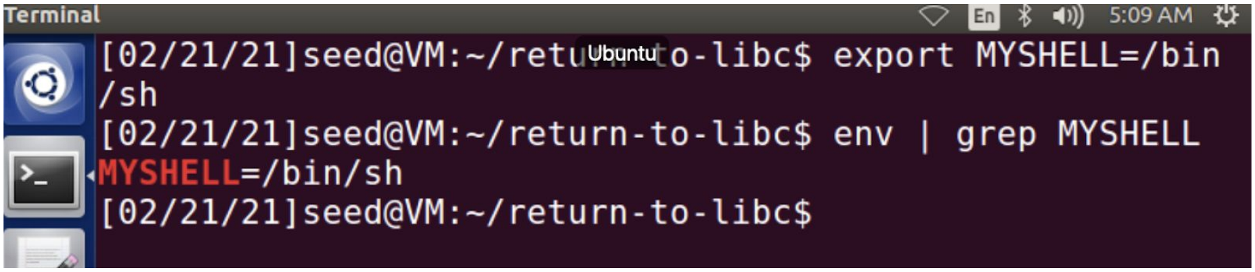
Once this is done, we are going to go into the debugger , touch a file named **badfile** and save the address of the **system()** and **exit()** as we will use them later.



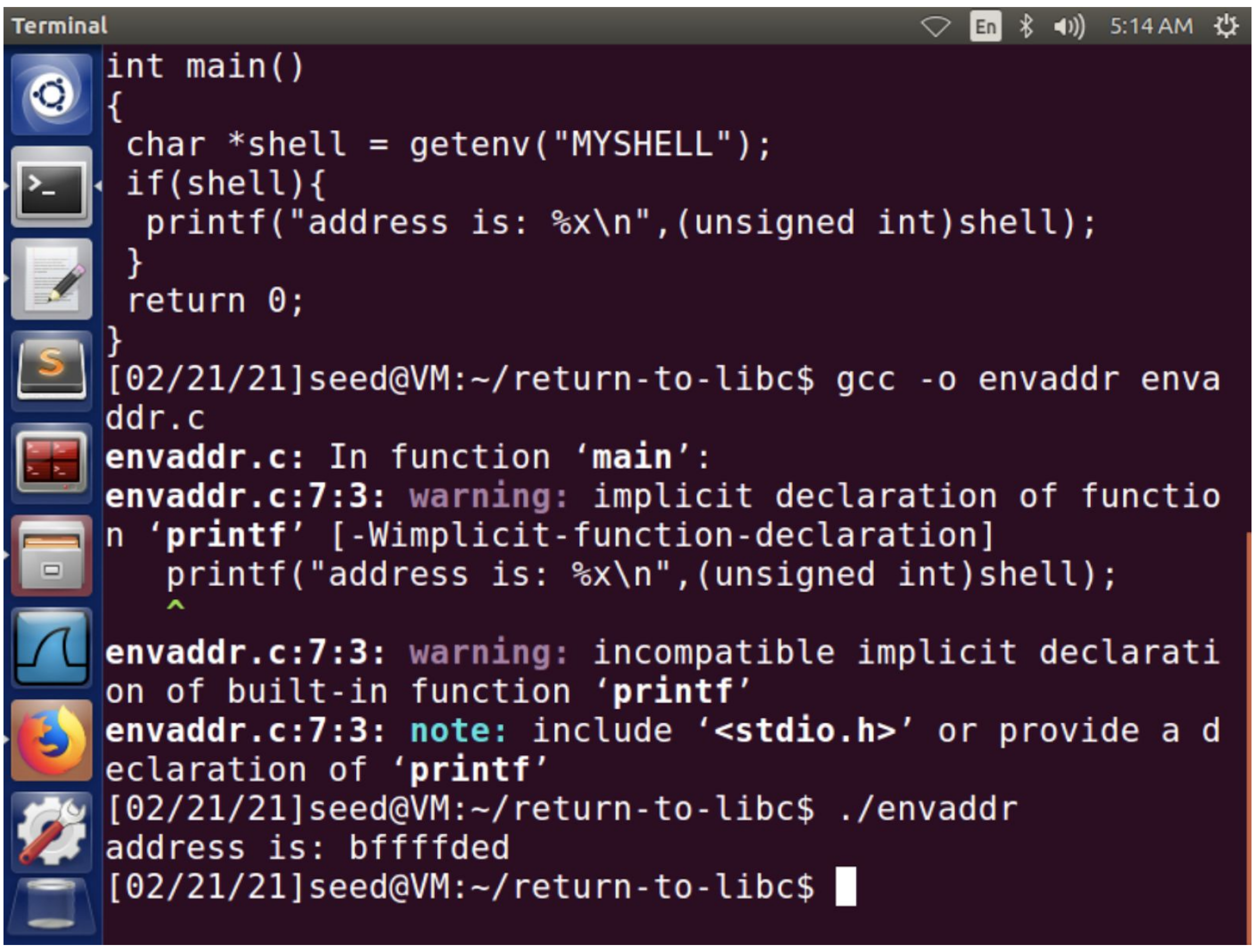
This concludes task 1 with us setting the program as a set-UID and finding the two important addresses for the usage of later on.

<u>Task 2:</u>

In this task, our goal is to jump to the **system()** function and get it to execute the /bin/sh command. We know we will be using **system()** to do this first, therefore we have created an environment variable and export it into memory before we begin our process. We know that when we create a program in a shell prompt, the shell creates a process for a child to execute this program and all the exported shell variables then become the environment variables of that of the child process. We can refer to this phenomena back in our first lab. As you can see below:



Then we will use the address of this variable as the argument into our **system()** function call. We are going to use the c code called envaddr.c to do this. The code and compilation are shown below:



We see that the address is 0xbffffded.

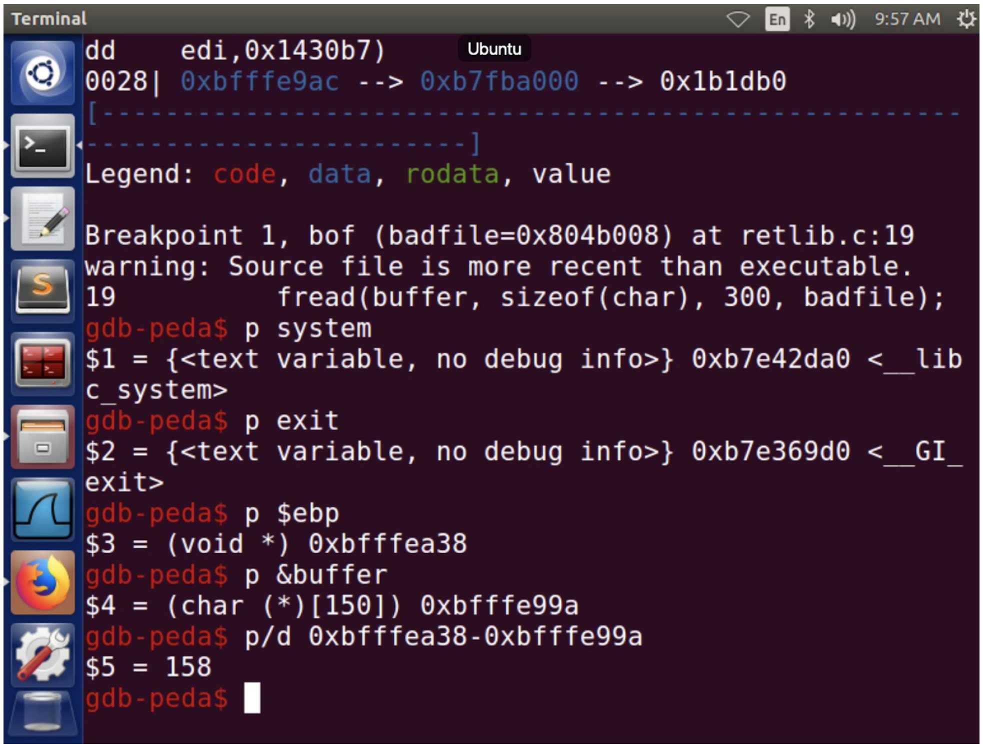
<u>Task 3:</u>

In this task we are supposed to figure out X,Y and Z. As well as the following addresses of the system(), exit() and /bin/sh of our exported variable MYSHELL.

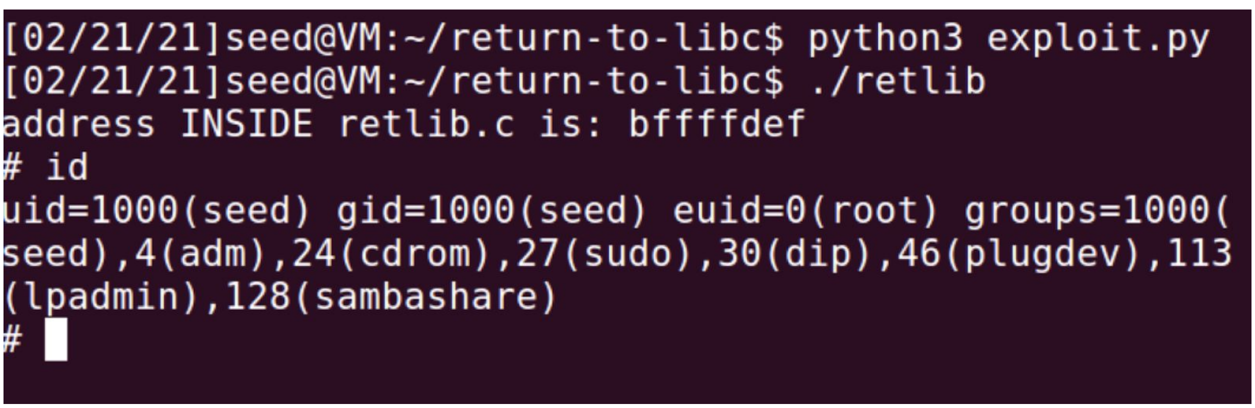
*task1.txt × | exploit.py × | retlib.c × | *exploit.c × | envaddr.c

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

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

X = 158+12 #170
sh_addr = 0xbffffdef        # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 158+4 #162
system_addr = 0xb7e42da0   # The address of system()
#system_addr = 0xb7e42da0   # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 158 + 8 #166
exit_addr = 0xb7e369d0     # The address of exit()
#0xb7db39d0 , 0xb7e369d0
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)
```

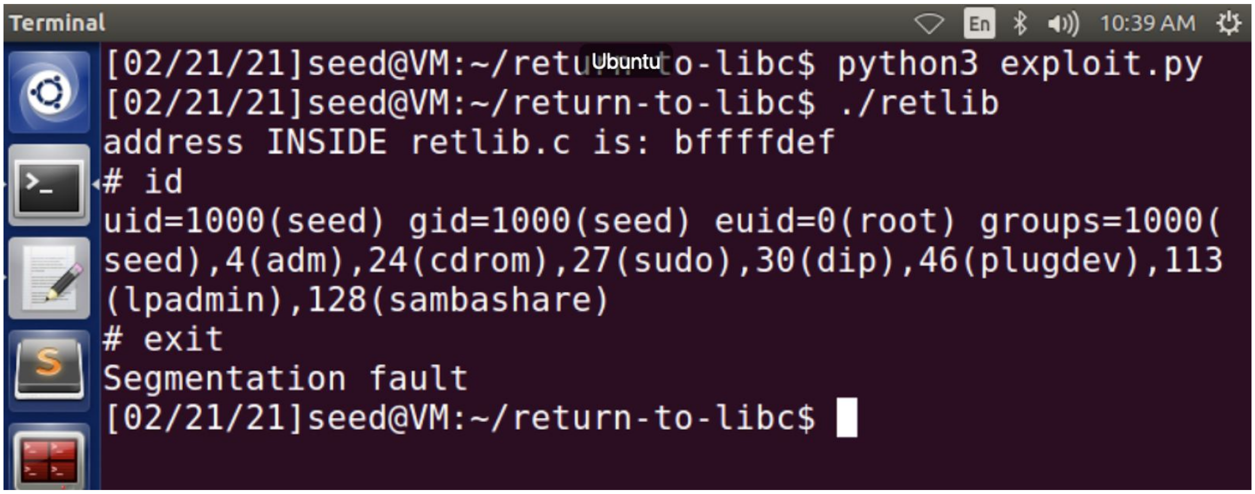
So the way I found these values were by going into the debugger, printing out system(), exit() and then computing the value in the debugger by $ebp-&buffer which got me the value of **158**. The following below is a screenshot on how I was doing it. Once we got 158, we know that plus 4 is the system's address, another 4 is the exit's address and another 4 is the exported variable that we first created back in task 1.



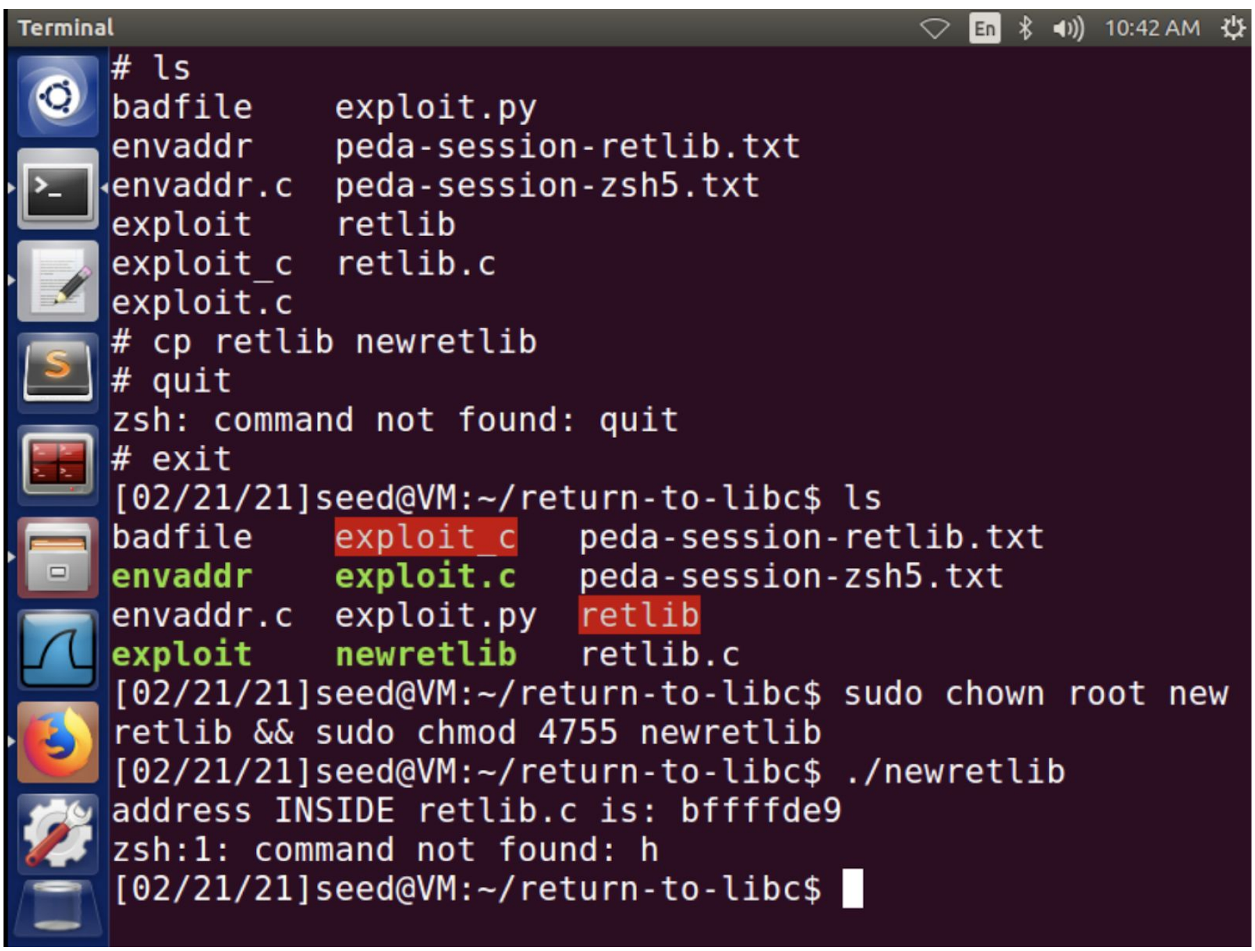
Once everything is into place we run the code as shown below and the we get a root shell!



Attack Variation 1: The exit() function is not really needed, it is only so it can transition smoothly, if we try it without the address of exit() this is the following.



Attack Variation 2: After we change and create a new file, we see that it is not successful as shown below:

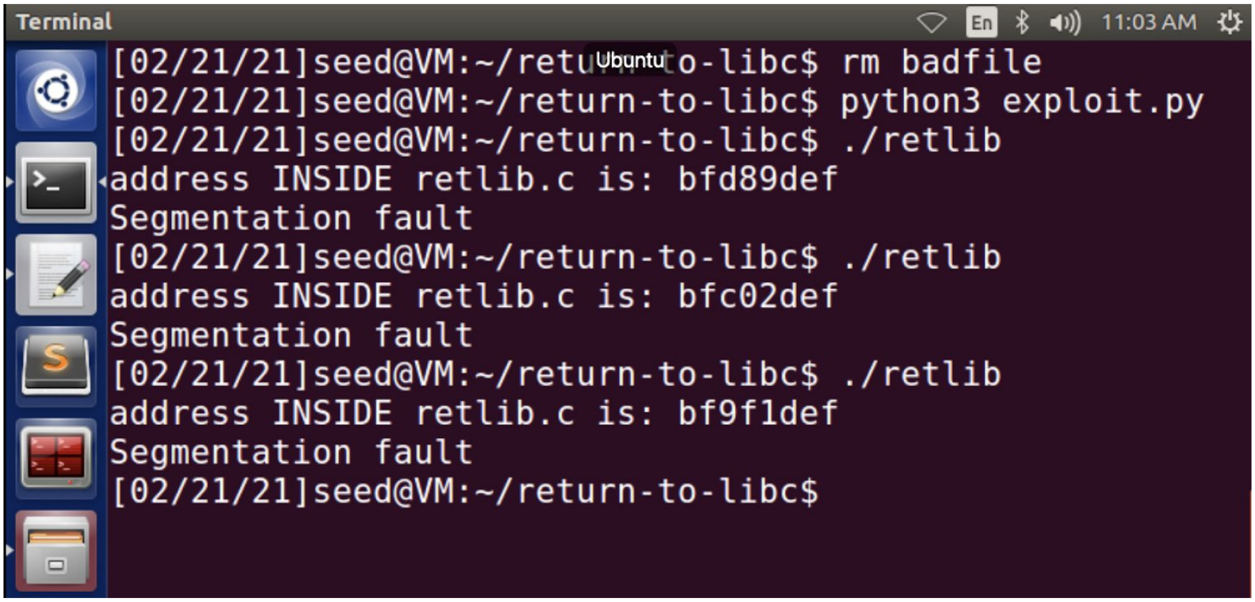


This is because the address of the /bin/sh is changing depending on the file name and size of the file you have. This is because it is in the stack (*ENV VARS*) and created before anything else. So it is crucial you find the address beforehand and make sure it is not changing.

Task4:

In this task we are turning back on the Address Space Randomization to back equal to 2. The following screenshot proves it:

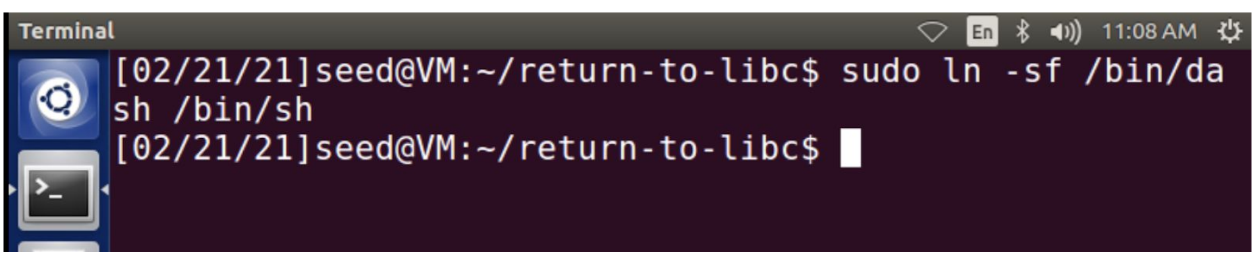
Now we do the same procedures that we did in task 2 and see the observations as below:



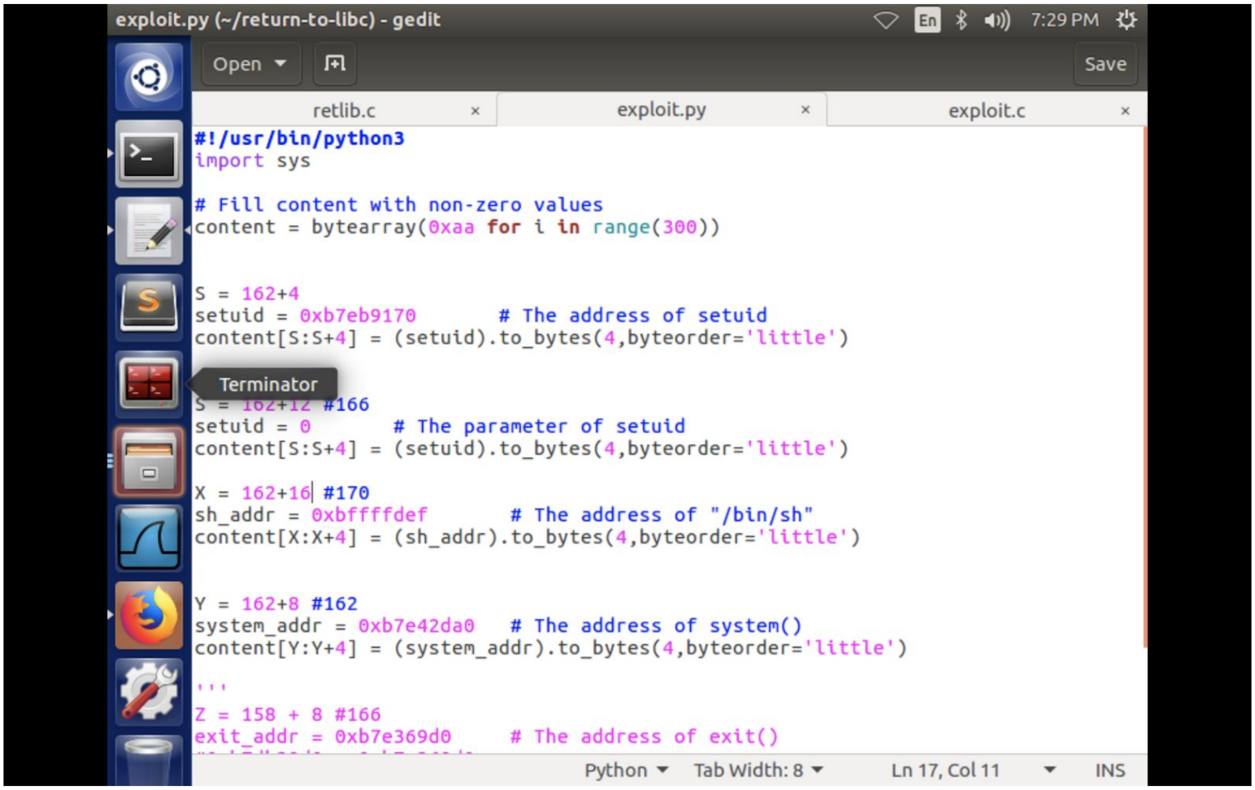
Since both the heap and the stack were now both being randomized, this made it harder for us to use the attack. As you can see the address is constantly changing and we are trying to access memory that is not ours. This is due to the */bin/sh, exit() and system()* functions to keep changing randomly throughout each compilation.

Task 5:

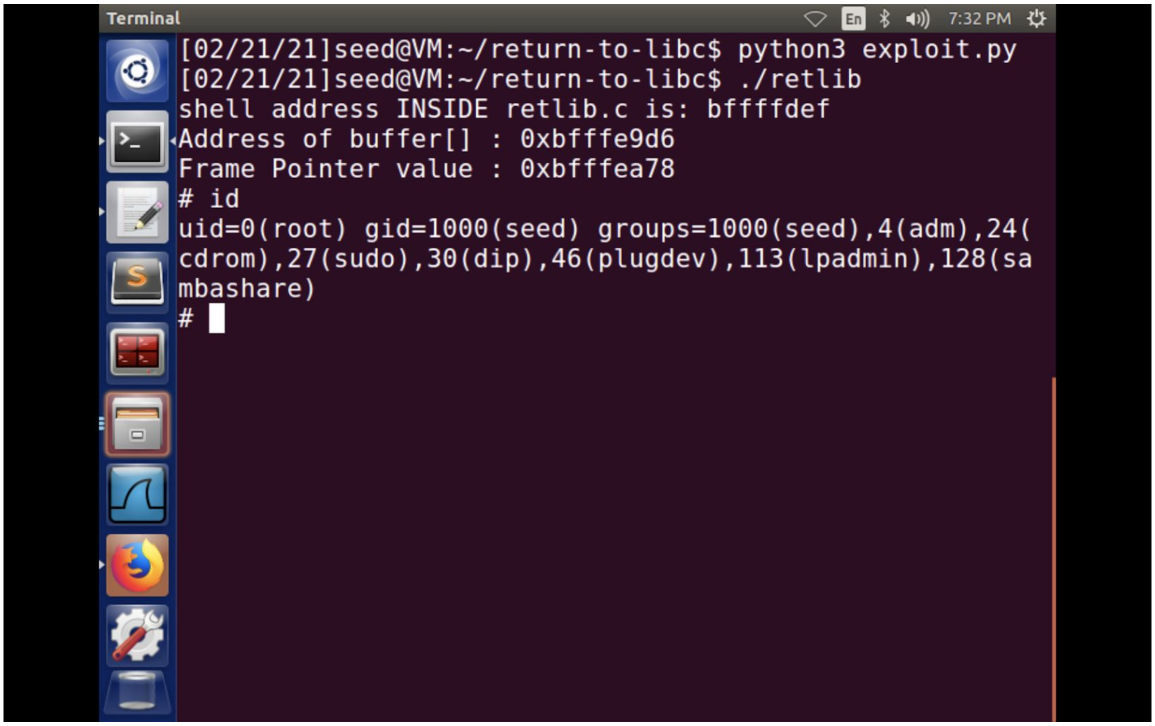
In this task we are going to do what the lab reports asks and change the symbolic link back to point to dash, as shown below:



The goal is to add a setuid(0) so that RUID is the same as the EUID. Which means we can use 0s in the argument because the code uses fread for the file, which doesn't terminate when it sees a 0 unlike the string copy function. If you look at the code now:



I modified the code for exploit.py, in this case $ebp-&buffer changed to 162. 162+4 is the address of setuid. 4 bytes more and you get the address of the system, 4 more bytes after that you get the parameter for setuid which is set to 0. Finally four bytes after that you get the address of the exported environment variable. This allowed me to get a root shell and defeat the countermeasure as shown below.


Terminal
7:32 PM
[02/21/21]seed@VM:~/return-to-libc$ python3 exploit.py
[02/21/21]seed@VM:~/return-to-libc$ ./retlib
shell address INSIDE retlib.c is: bffffdef
Address of buffer[] : 0xbfffe9d6
Frame Pointer value : 0xbfffea78
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#