**BUFFEROVERFLOW ATTACK SETUID**

Contents

[Environment Setup 3](#_Toc118408210)

[Task 1 3](#_Toc118408211)

[Task 2 6](#_Toc118408212)

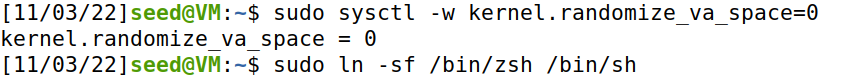
[Task 3 8](#_Toc118408213)

[Task 4 13](#_Toc118408214)

[Task 5 14](#_Toc118408215)

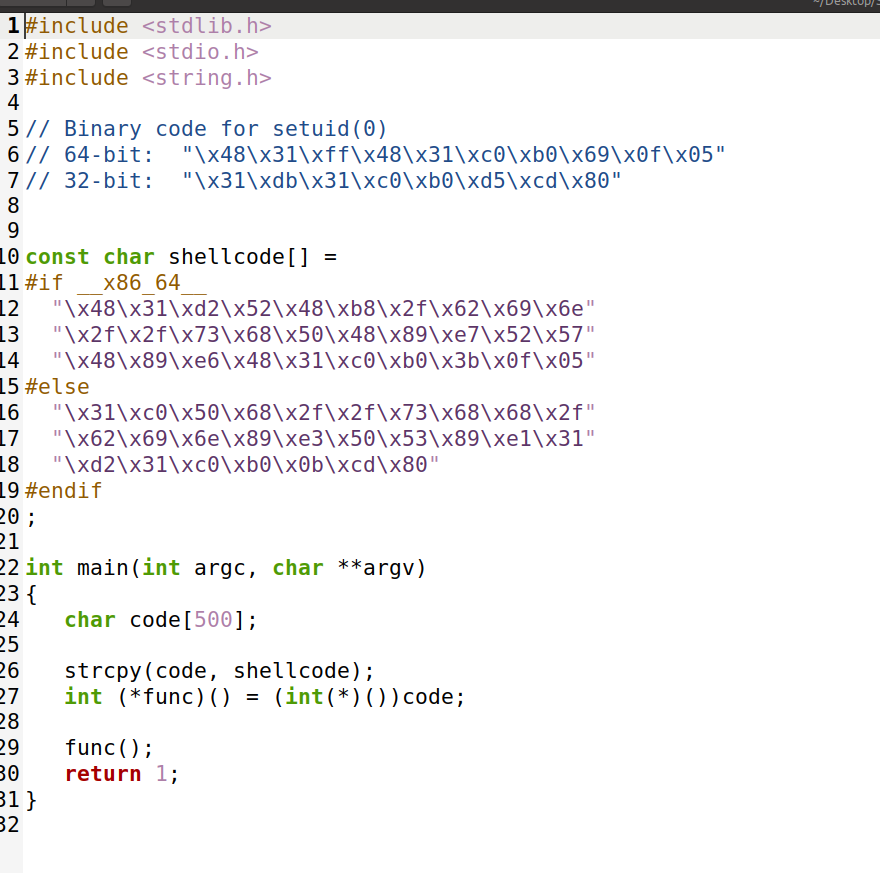
# Environment Setup

Turning off address space randomization and configuring /bin/sh

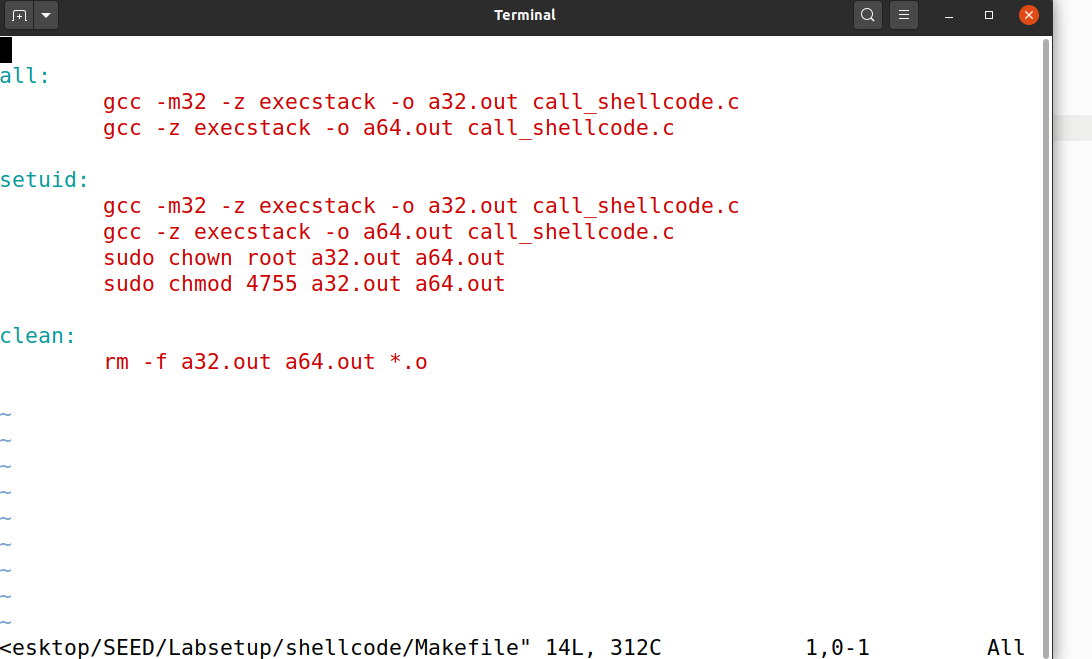


# Task 1

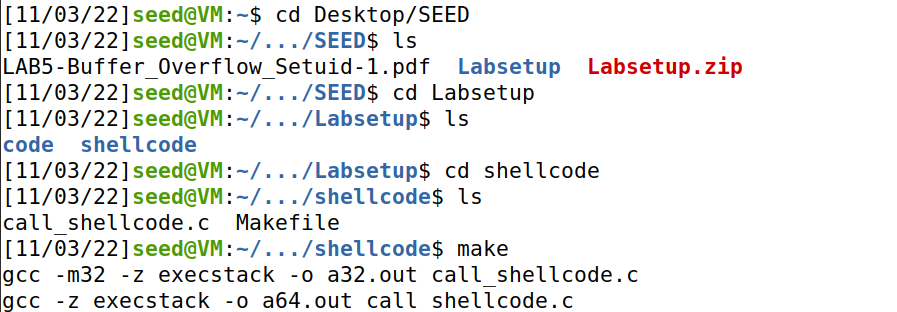
This is the Shellcode provided.



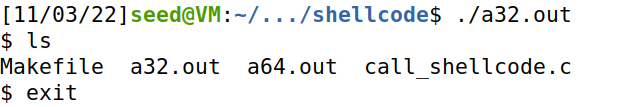
Since MakeFile is already provided we will simply compile the program.



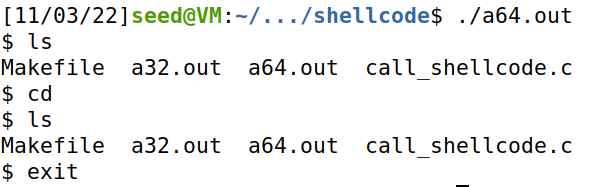
Using make command to compile files and run the output



And it can be seen that 32 bit shellcode working.

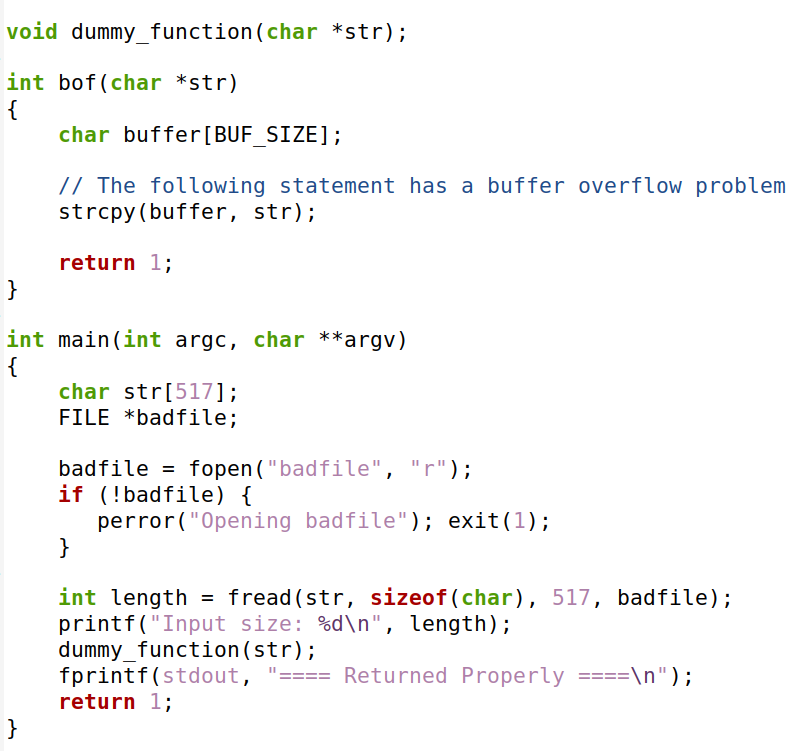


For 64-bit code it is also visible it is working but both codes are limited to the directory they were executed in.

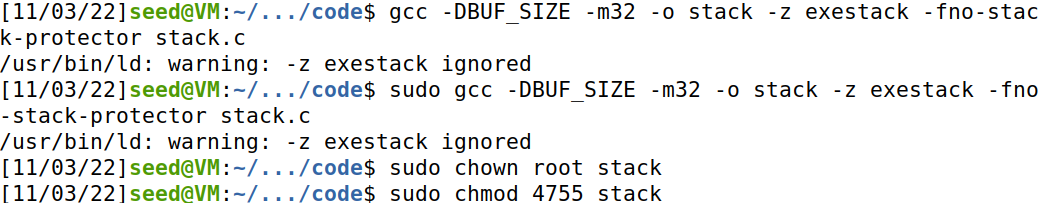


# Task 2

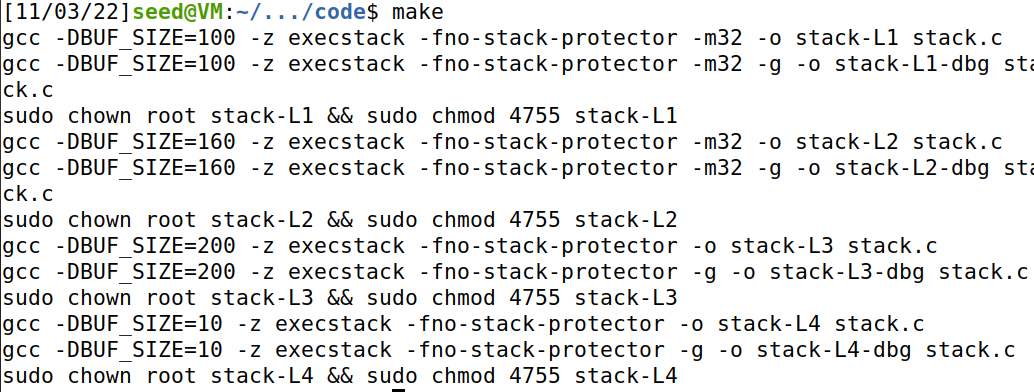
* Here we can notice that the maximum length is set to 517 while the default value is 100. The problem will be when the code is executed it will turn out to be an buffer overflow attack due to no boundary check on the statement strcpy(buffer, str) in the bof function.



* Given as the vulnerabilities are out in the clear, I will simply compile the code file around security policy by closing StackGuard and Non-Executable Stack. Further, I have set the owner to root and permissions to uid.



* Now I compiled the files using MakeFile already provided in LabSetup

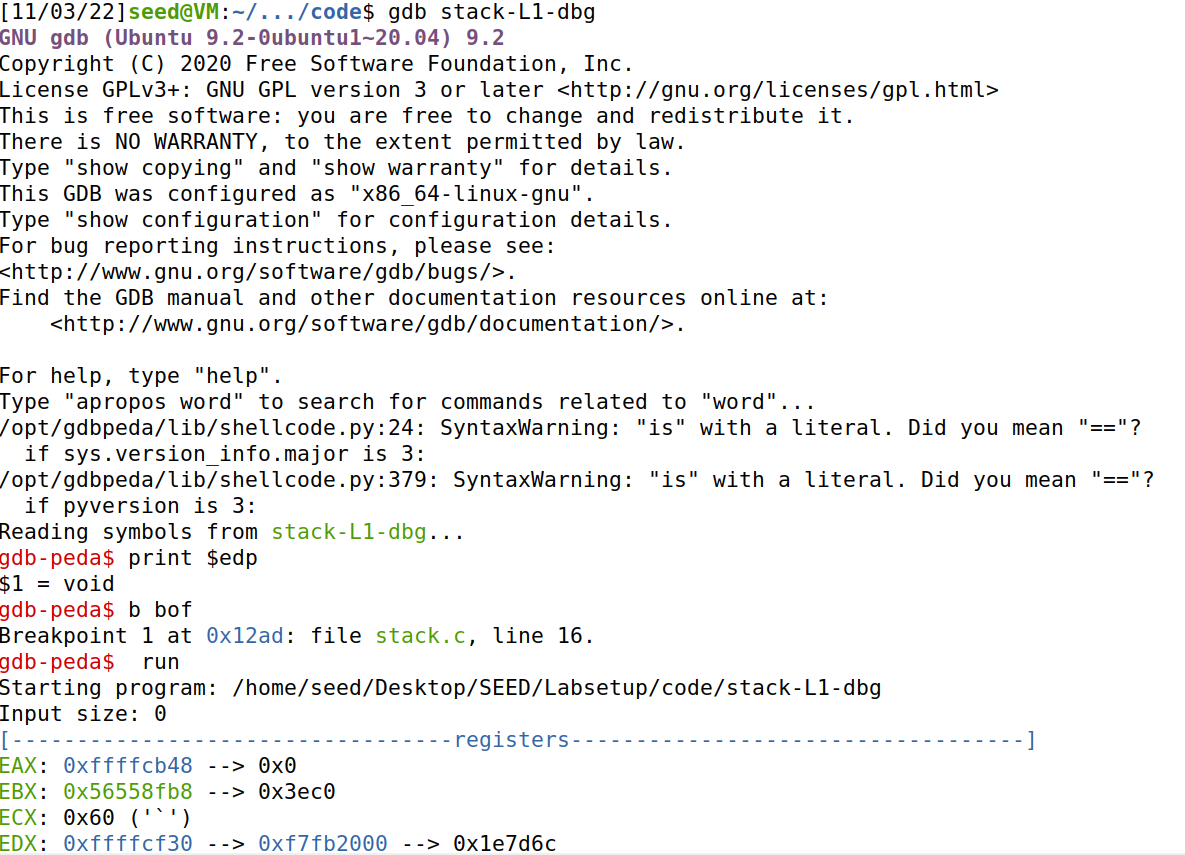


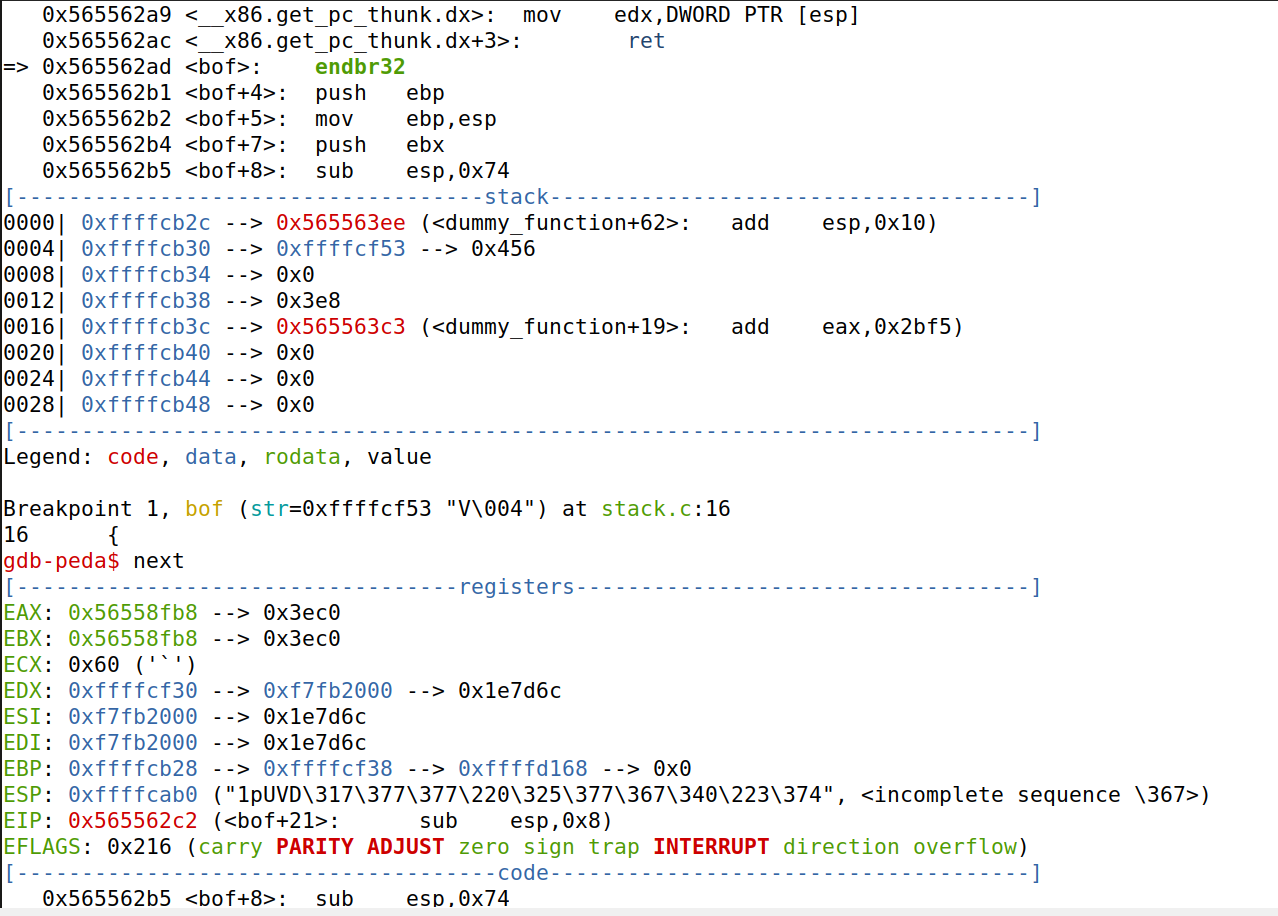
# Task 3

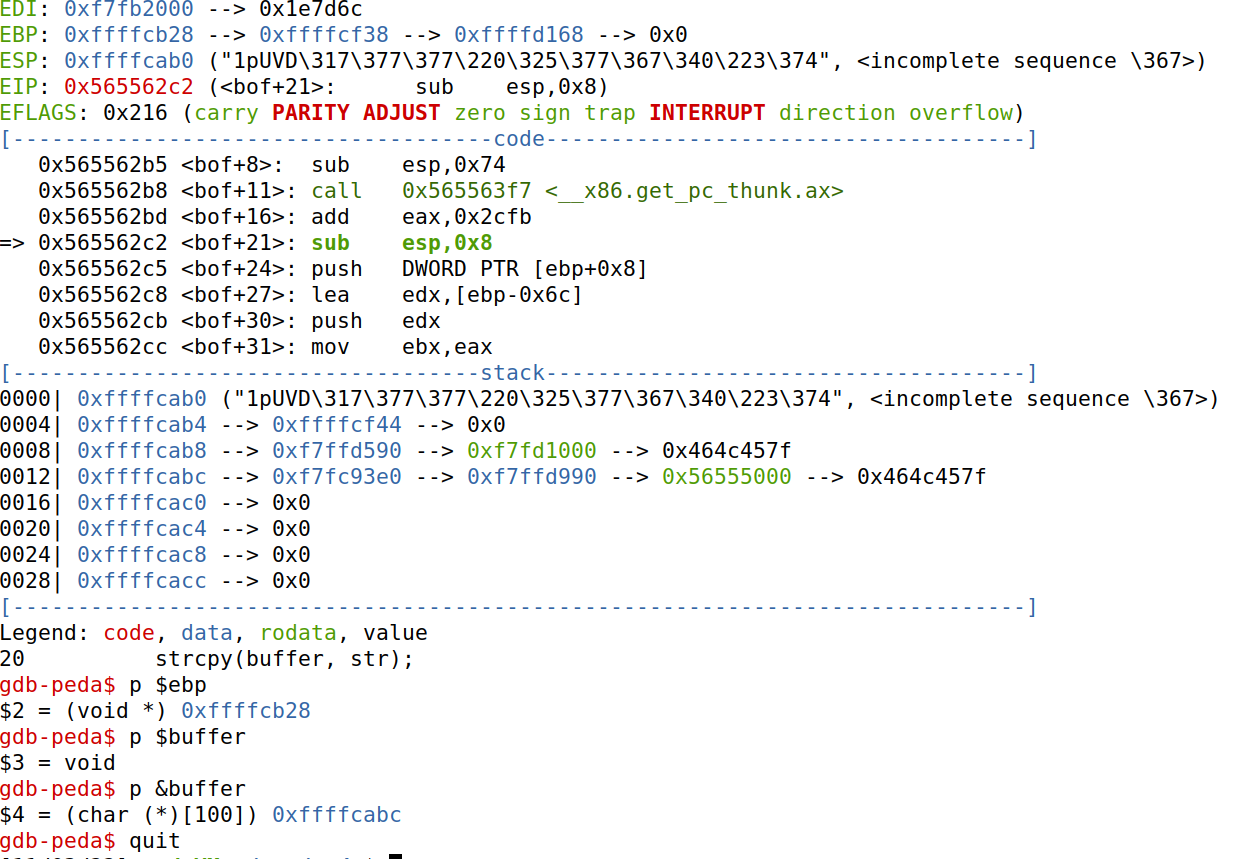
* Create a badfile.



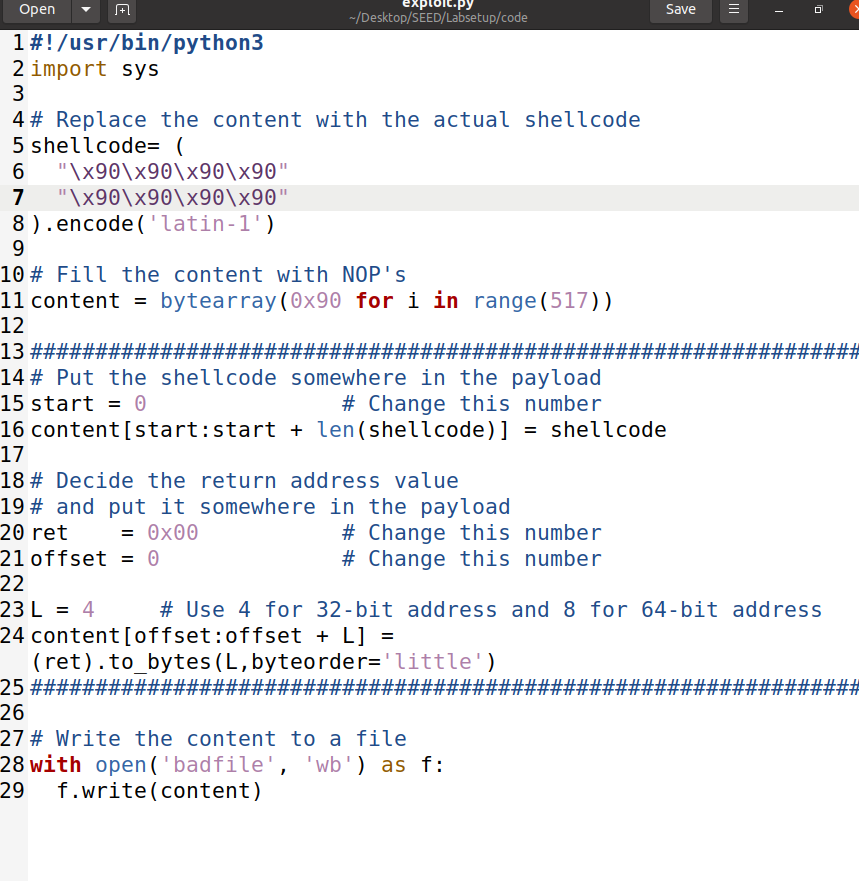
* Now I went for the addresses of bof function and buffer.







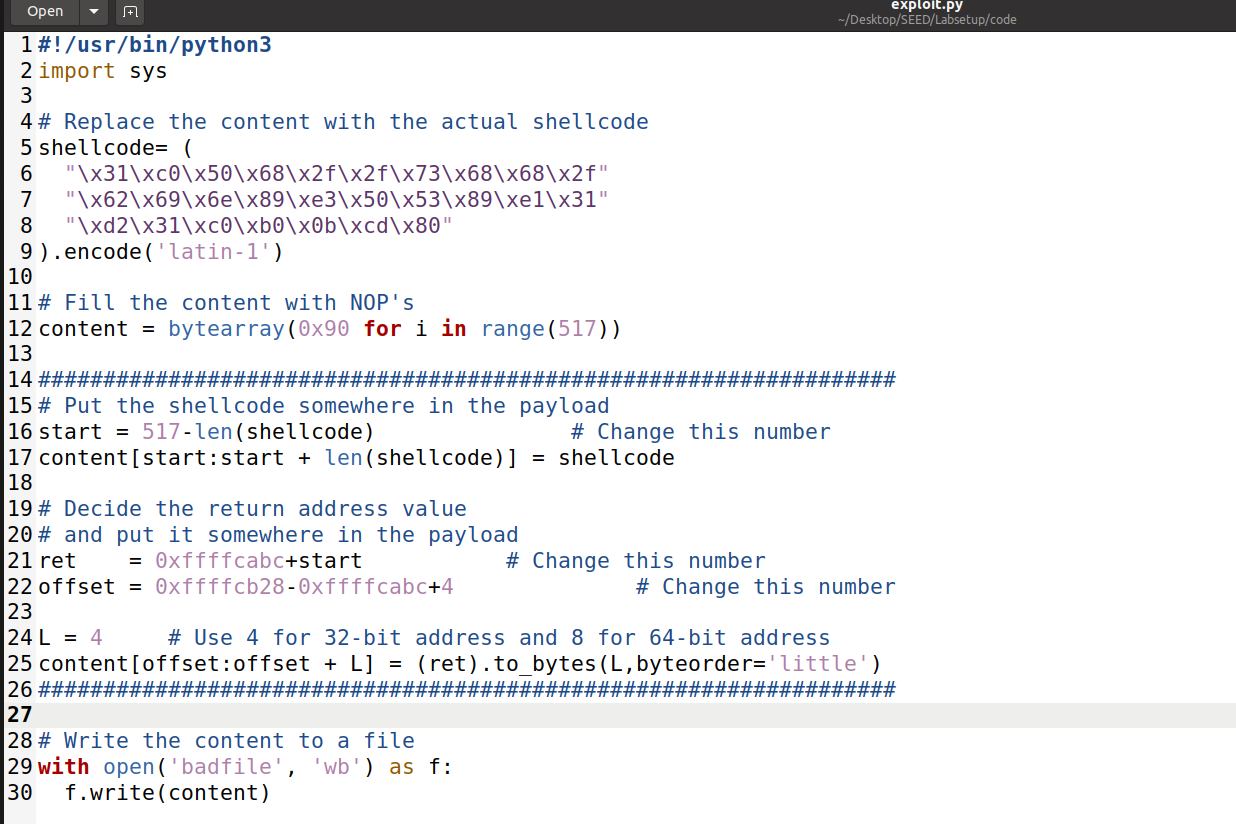
* This is the provided exploit.py file



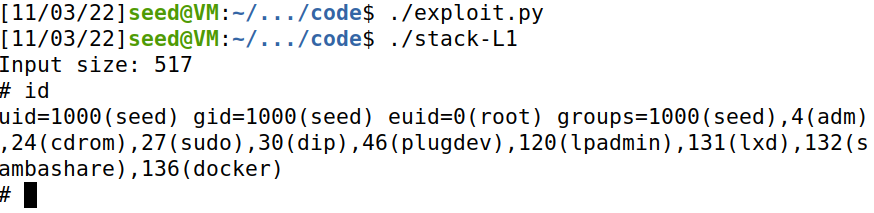
* Moreover, we will take the 32-bit condition from the call\_shellcode.c and paste in exploit.py



* I modified the file to what it should be to launch a successful buffer overflow attack.

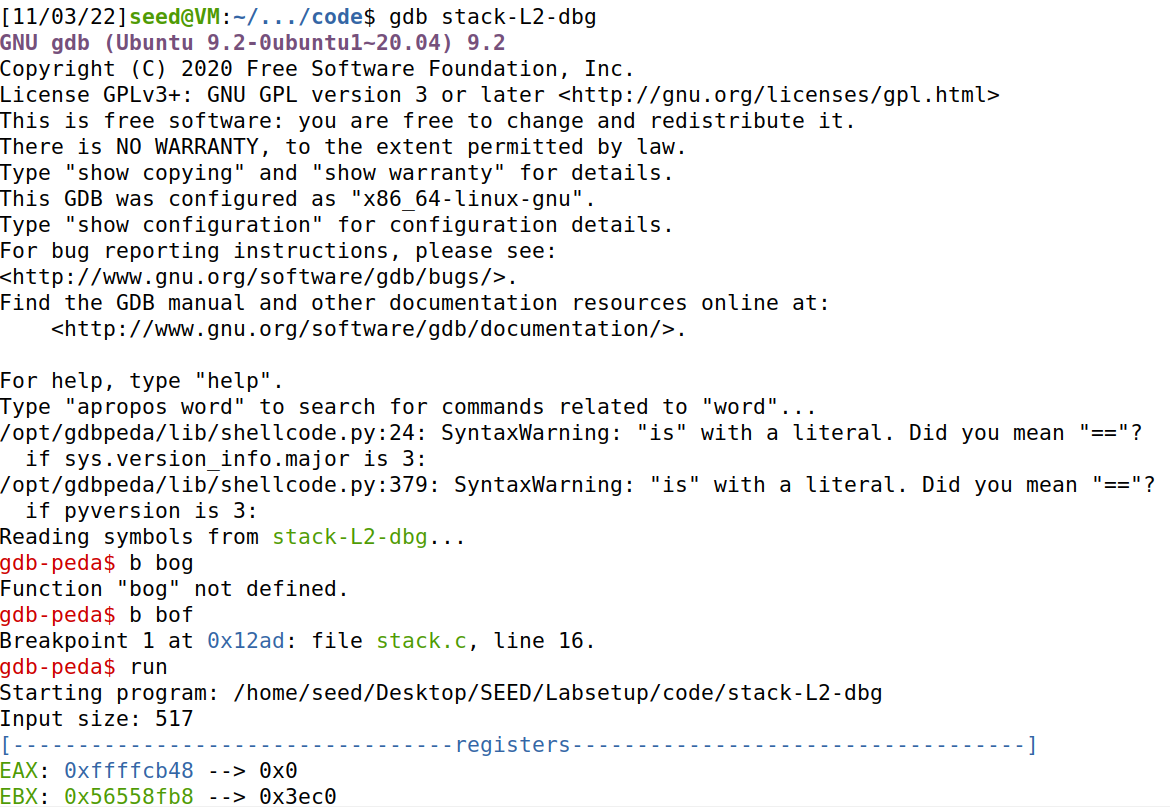


* Generating payload and launching the attack where we can see euid set to root and the attack being a success.

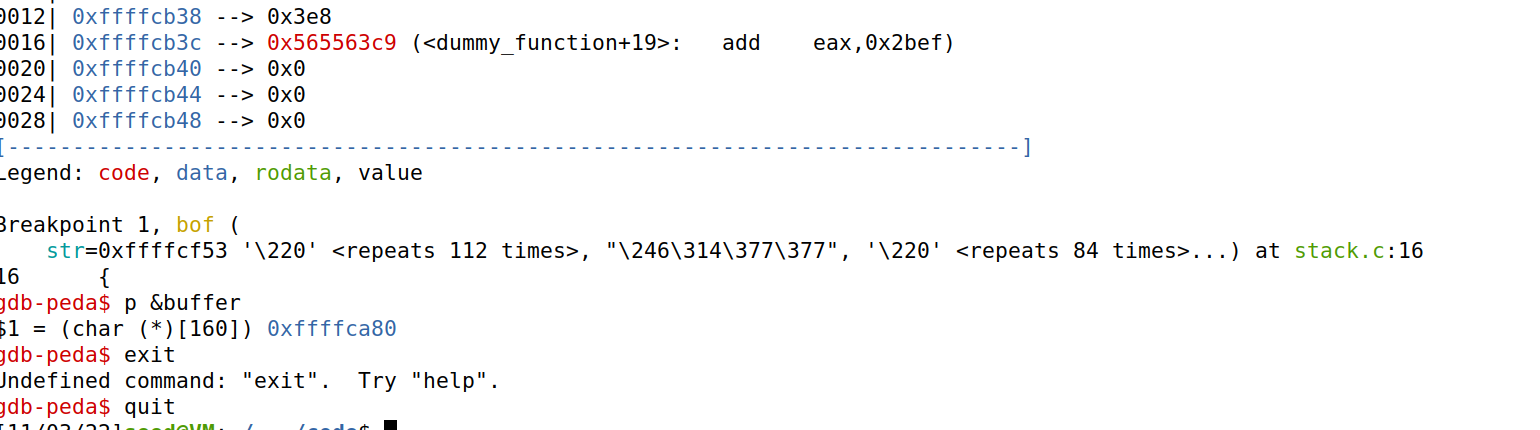


# Task 4

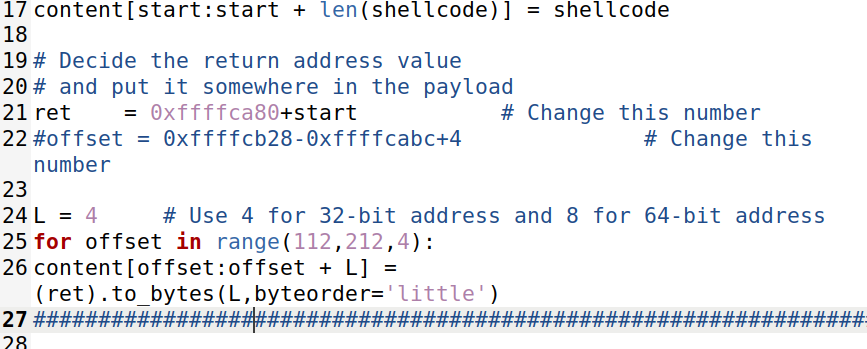
* Now checking the stack-L2-dbg for the buffer address.



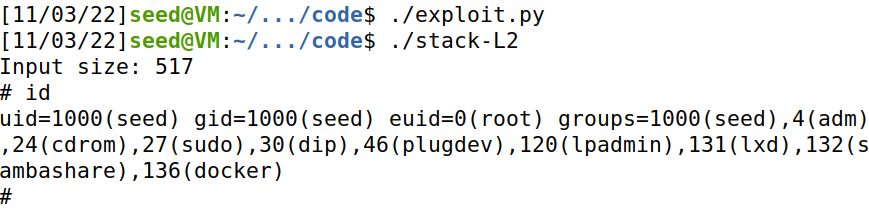
* And I found it.



* Now modifying the file with buffer address on line 21, commenting line 22 and adding a loop on line 25.

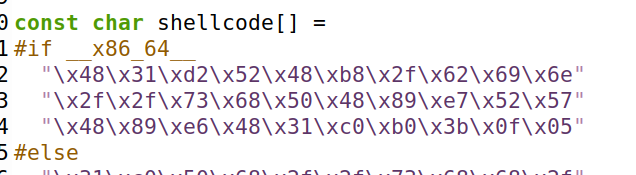


Now launching the attack which is again a success.

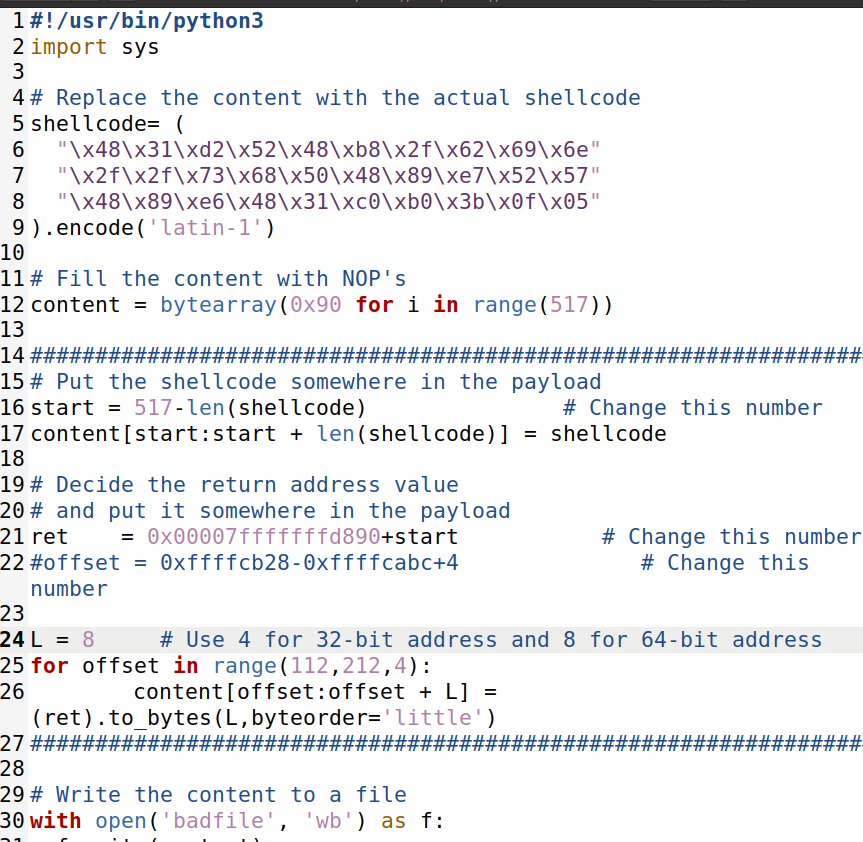


# Task 5

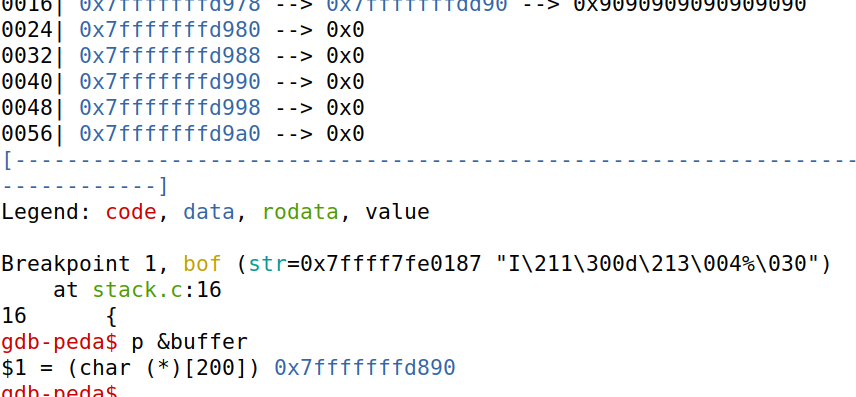
* Changing shellcode to that of 64-bit from the file call\_shellcode.c



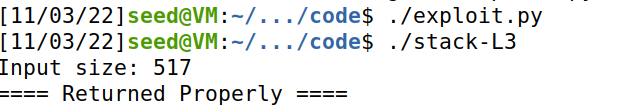
* Now changing the buffer address on line 21, L = 8 on line 24 and shellcode to as shown in the above file.



* Checking the buffer address in stack-L3-dbg



* The result is returned even after some configurations.

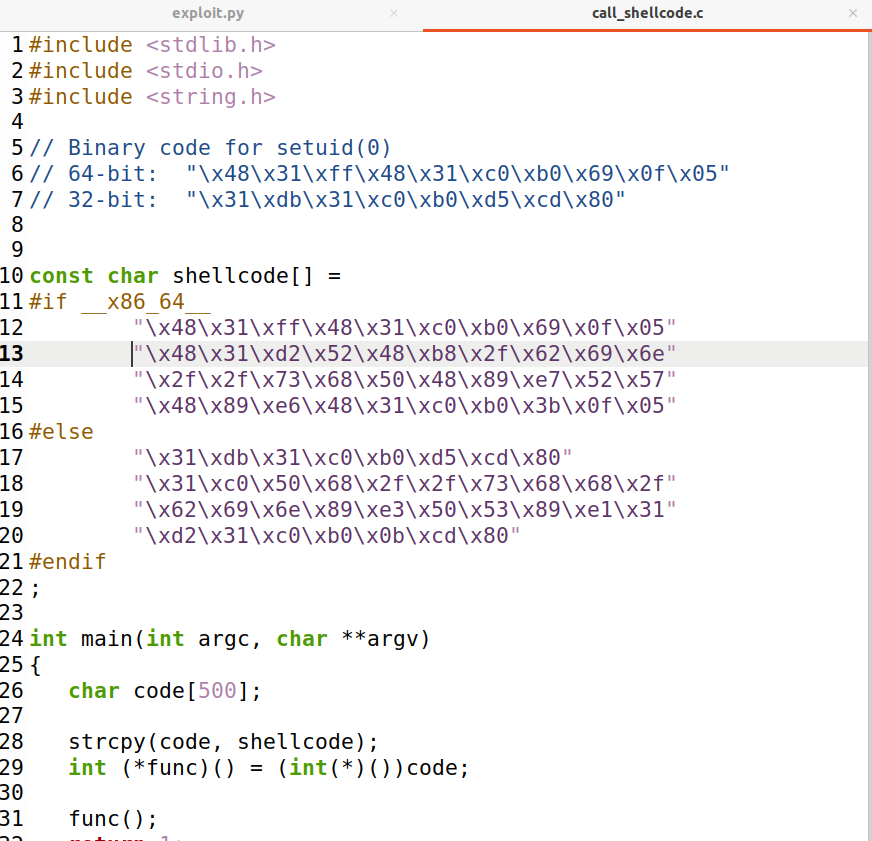


# Task 7

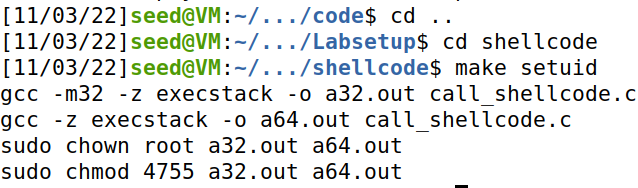
* Configuring /bin/sh again to meet the task requirements.



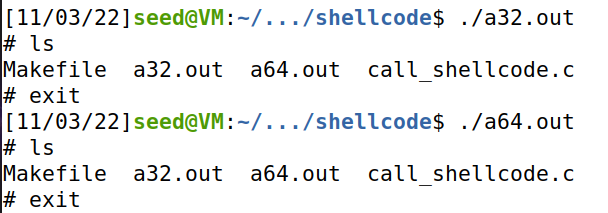
* Changed the code to setuid value 0 by some minor changes in the code.



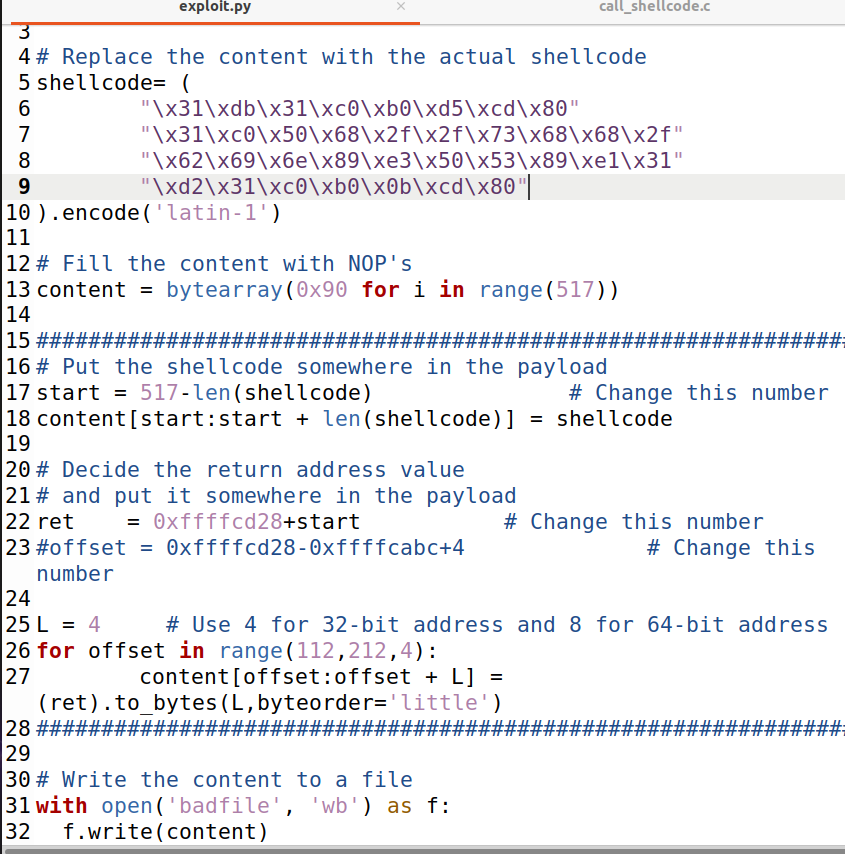
* Compiling the shellcode.



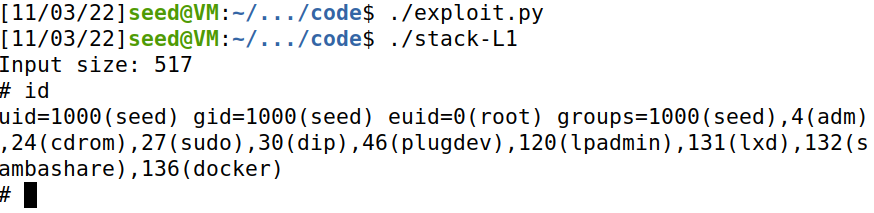
* After executing the shellcode the shell is accessible.

****

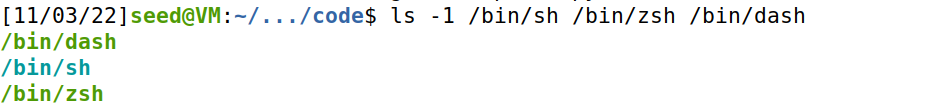
* Made changed to exploit.py file



* Now when we try running the attack we can notice that there are errors. As inevitable that the attack won’t work after the configurations made to the /bin/sh.



* And running the final command to see if the attack worked which it obviously did.



# Task 8

* Turning on Address Randomization.



* Now launching the attack



* The brute force the attack was a success.

