**ASSIGNMENT – 4**

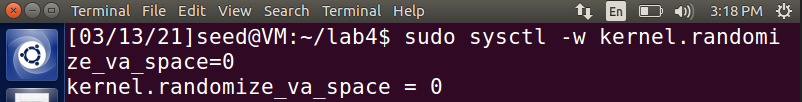
Name: **Sudharsan Srinivasan**

UTA ID: **1001755919**

**Countermeasures Turned off:**

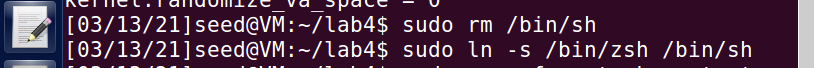
* Address Space Randomization is done to stop randomizing the starting address of heap and stack. This makes guessing the exact address difficult, thereby making the buffer-overflow attack also difficult.

The following image shows the command required for the same.



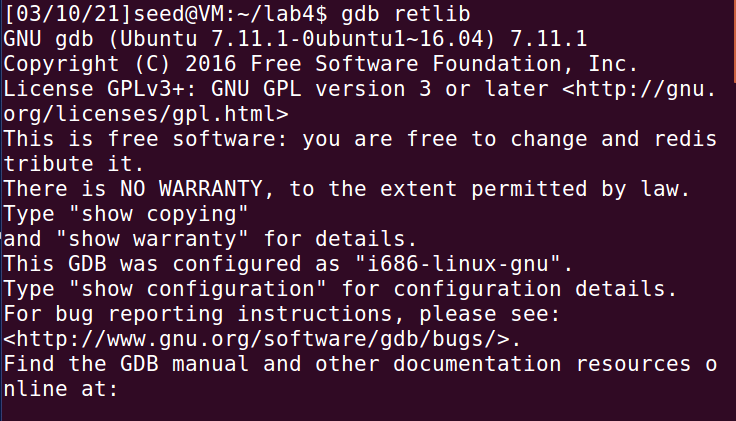
**Configuring /bin/sh**

* Here, the victim program is a Set-UID program and the countermeasure in /bin/sh makes our attack more difficult.

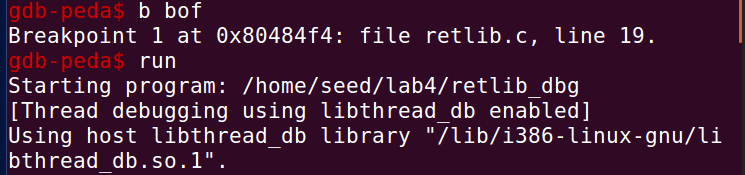


**Task 1 – Finding Addresses of libc functions:**

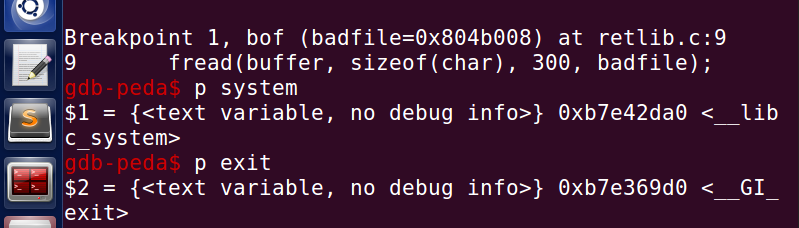
* For launching buffer-overflow attack on the vulnerable program, we first need to find the addresses of system(), exit() and ‘/bin/sh’ and their indexes to update into our exploit.c code that we have created to launch an attack. To do so, use the command **gdb retlib**,



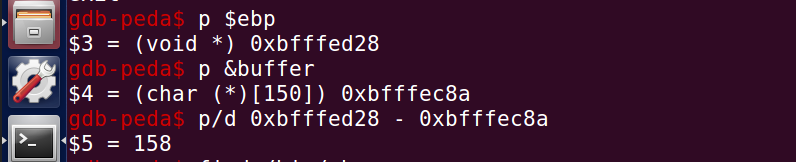
* After this, once inside debugger, insert a breakpoint at bof (function to create badfile) and run the program.



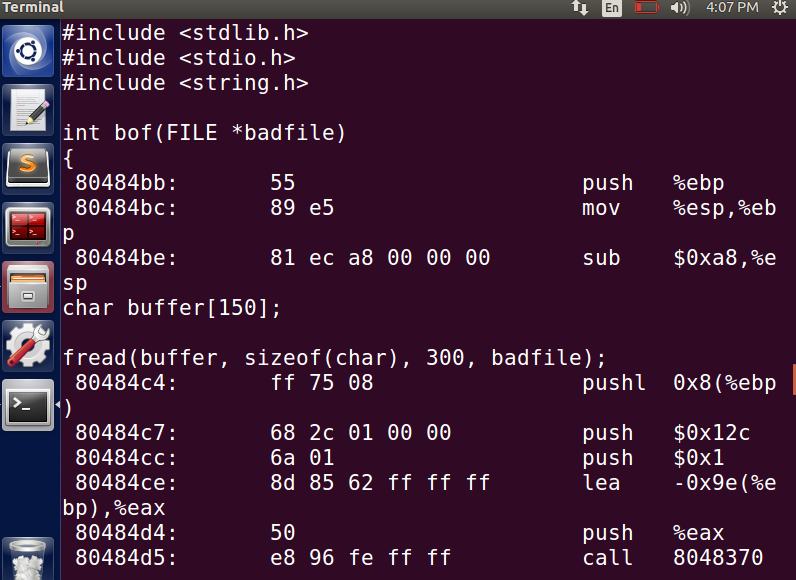
* Once it stops in bof function, use the commands **p system** and **p exit** to find the addresses of system(), exit().



* To find out the index/offset values for the system, exit and bin/sh, use the commands ‘**p $ebp**’ and ‘**p &buffer**’, the difference of these two addresses gives the offset value, as shown in the below image, 158 + 4 = 162 is where is the system offset starts.



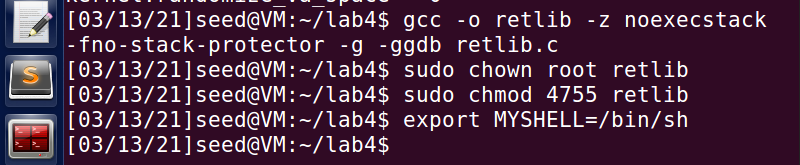
* Alternatively, we can use the command ‘**objdump –source retlib**’ to get to debugger of retlib and scroll to find the bof function as seen from the below pic.



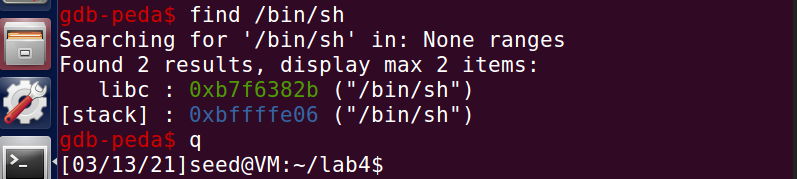
* Here, **lea** stores the address of the buffer head, therefore, we use the hex value **0x9E** and find its corresponding decimal value, which is 158 +4 = 162. Either way, we find out that the system head start at offset 162.

**Task 2 – Shell String in the memory:**

* This task involves finding out the memory address of /bin/sh. To do so, we compile retlib after modifying its buffer value to the one recommended for this assignment (i.e., 150)
* Then, export MYSHELL command as follows and then once inside debugger of retlib, we use **find /bin/sh** command to find its address.



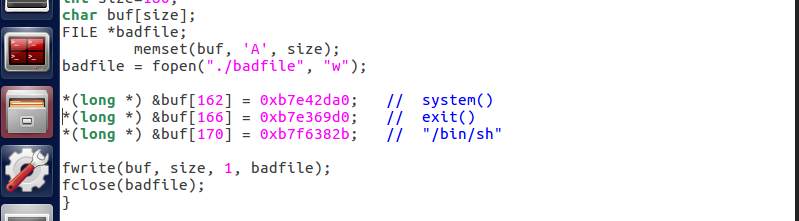
* Once, inside retlib, we use find /bin/sh we see that it has 2 different addresses, since we have exported the default MYSHELL stack address. Since this involves libc attack,we choose that address value for the attack.



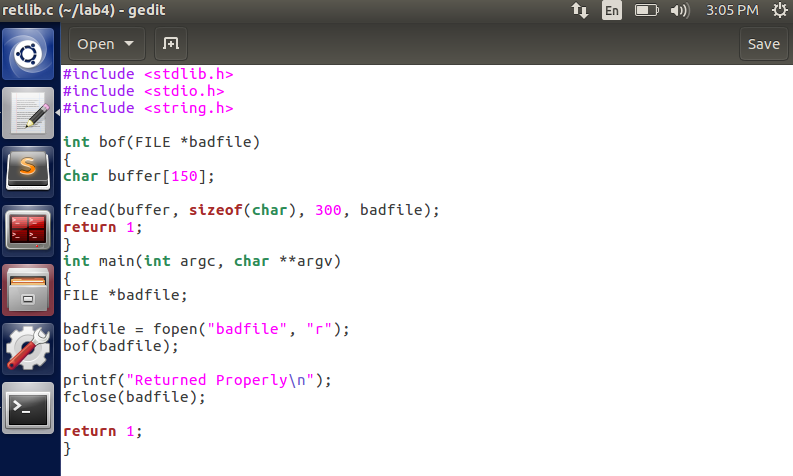
* From the above addresses, we select libc address **0xb7f6382b.**

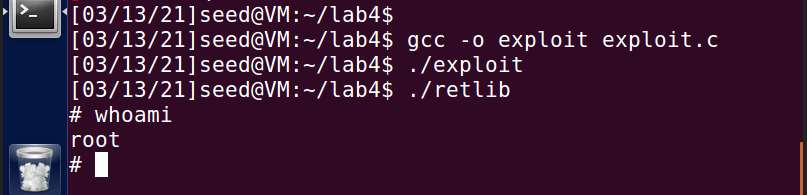
**Task 3 – Buffer Overflow Vulnerability exploitation:**

* Now, after finding out the address values of system, exit and bin along with their offset values, the contents of exploit program are ready to be updated and to launch the attack.
* Modify the contents of exploit.c and save them.



* It is also important to modify the buffer size of retlib to 150 for this assignment. Now that we have modified the contents, we compile the exploit.c program and then run the vulnerable retlib program.

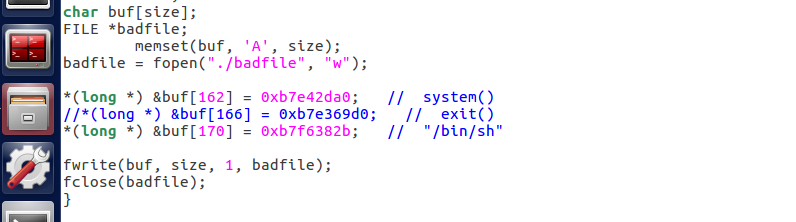


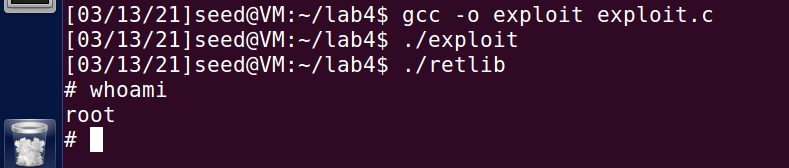


* As evident from the above image, we have gained access to the root shell through the vulnerable program indicating that the attack is successful.

**Variation 1:**

* Now, we **comment out the exit() line from exploit.c program** and try running the vulnerable program again.

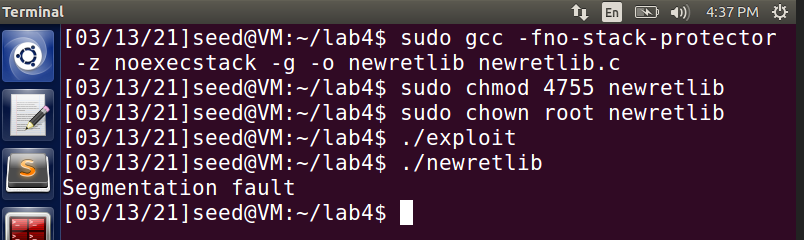




* We are able to gain root access even commenting out the exit() from the exploit program, showing that our attack was successful.

**Variation 2:**

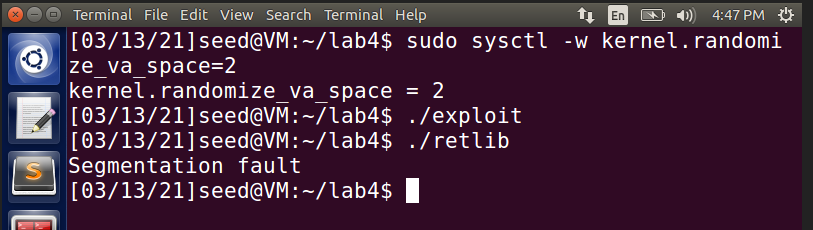
* Now, we modify the name of retlib file to a different file name newretlib. The program is recompiled and run to see if we still get root access.



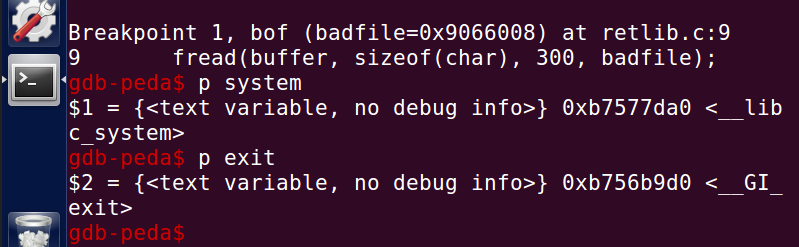
* After changing the name of the file to newretlib, we are not able to get root access. This could possibly be because of the fact that the length of the shell address program is not the same as the vulnerable program length, hence resulting in the attack failing.

**Task 4 – Address Randomization Turned on:**

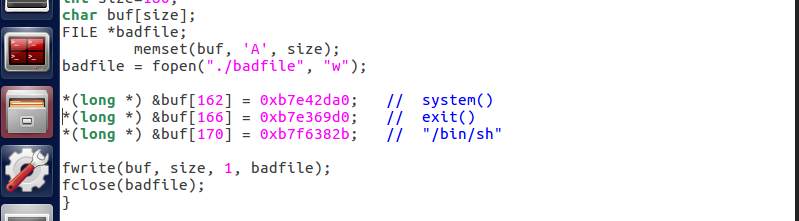
* Space randomization, which was earlier set to 0 to avoid changing of address randomly for heap and stack is now again set to a random value, say 2. Task 3 is run again to see if we get root access.



* This is because, the address is randomly changed during run time, which does not match with the set values inputted in exploit.c program as shown in the below image.



* The current address in system and exit shown in the above picture as compared to the set address below in exploit.c program.

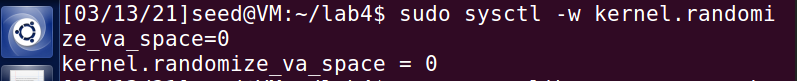


* Since both the address values are different, we are not able to gain access to the root shell thus failing the attack.

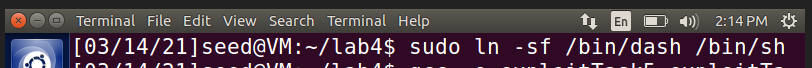
**Task 5 – Defeat countermeasure of shell:**

* To do this, address space randomization is turned off. Address Space Randomization is done to stop randomizing the starting address of heap and stack. This makes guessing the exact address difficult, thereby making the buffer-overflow attack also difficult.

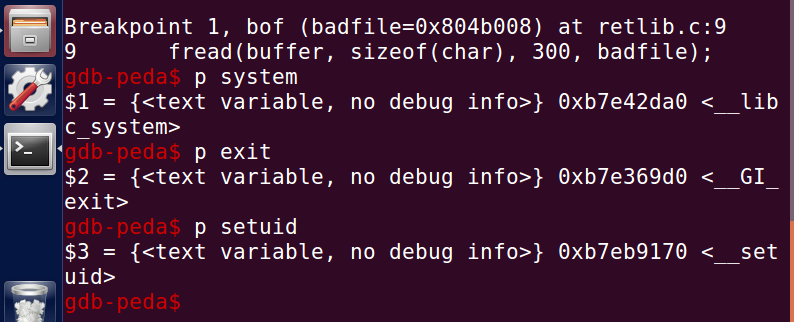
The following image shows the command required for the same.



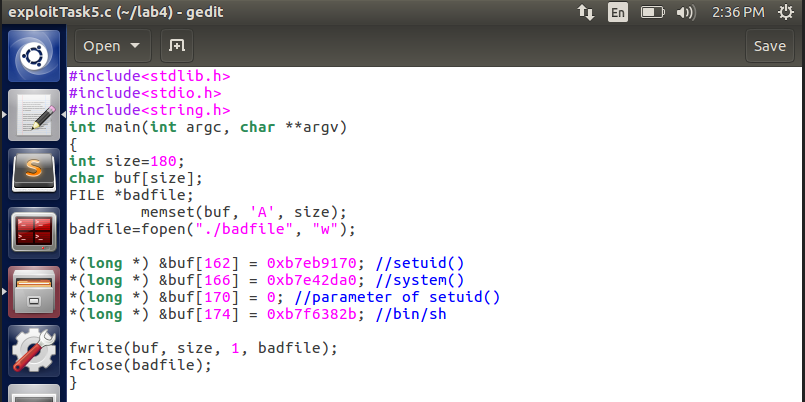
* For achieving this, modification to the exploit vulnerable program is needed because with the current values, we will not be able to gain root access since the countermeasures are active. Also, we make /bin/sh point to /bin/dash as shown below.



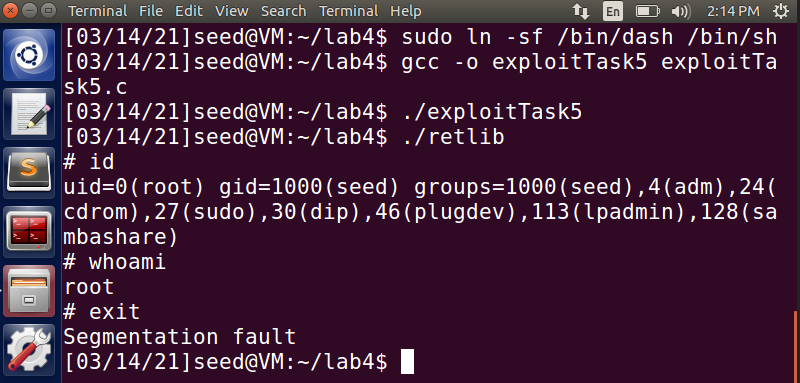
* This stops us from coming back to the system(), since pointing to dash results in privileges being dropped. One approach that will be followed in this case is to invoke setuid(0) before we invoke system().
* For this, first we obtain the address of setuid from gdb retlib as seen from the below pic.



* The above setuid address is updated inside the exploit program. The offset value for setuid is set such that it is invoked before invoking the system() function.



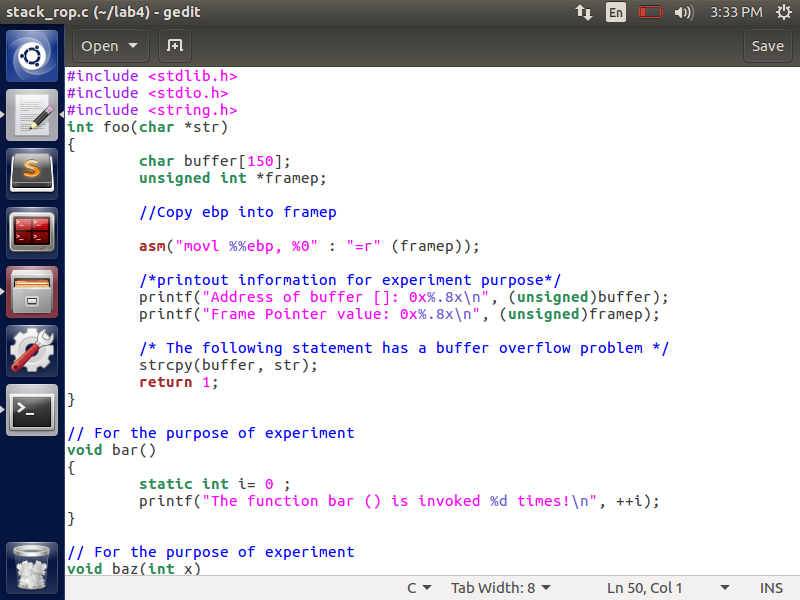
* This updated exploit program is compiled and run.



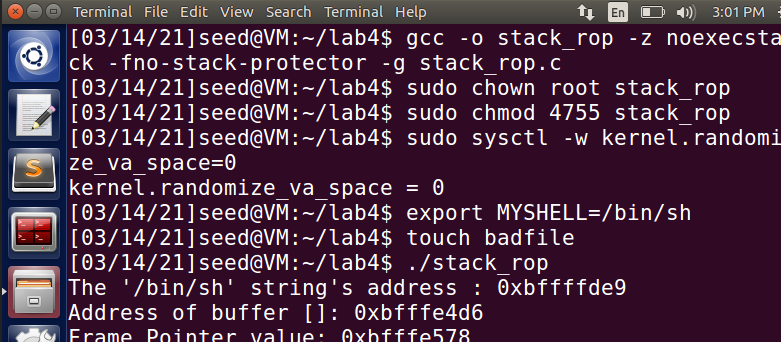
* As seen from above, we are able to gain root access by invoking the setuid() before system(). This is possible because the setuid(0) function calls both real user ID and effective user ID and sets them both to 0, making it a non-setUID program even though it still has root privileges.

**Task 6 – Defeat countermeasure of shell without putting zeros in input:**

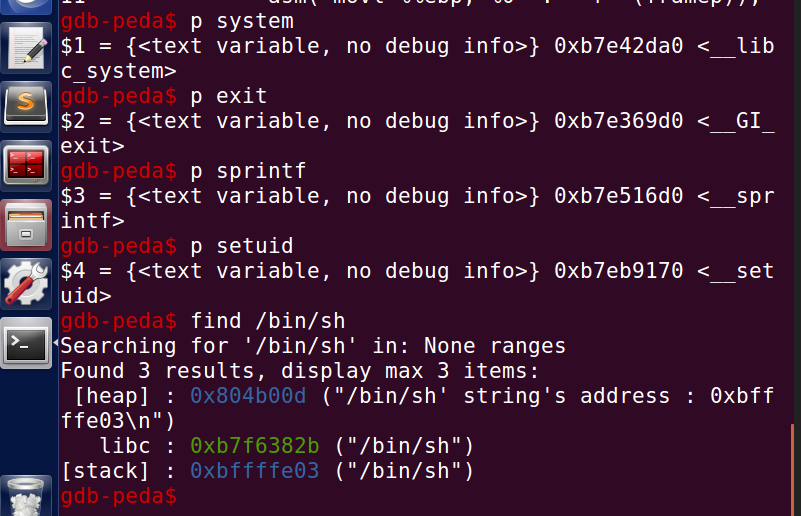
* In this task, we try to defeat shell’s countermeasure without adding zeros in input as in the case of previous task. We use ROP (Return Oriented Programming) to chain multiple functions and to change non-zero values to zero internally when the program is run, using function such as sprint(), which is called before setuid().
* We first create stack\_rop.c program which will be used to launch an attack to see if we can gain root access.



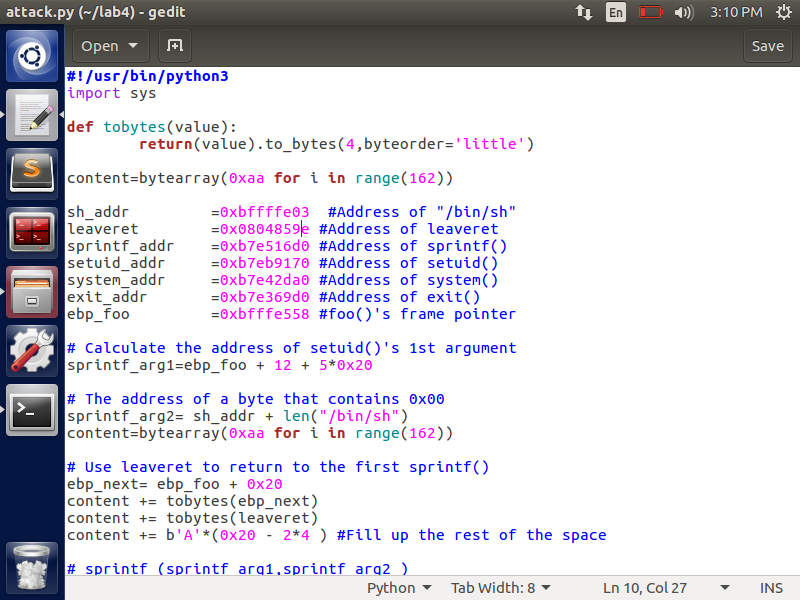
* After this, we compile stack\_rop program using non-executable stack and grant root privileges to it, also setting turning off address space randomization.



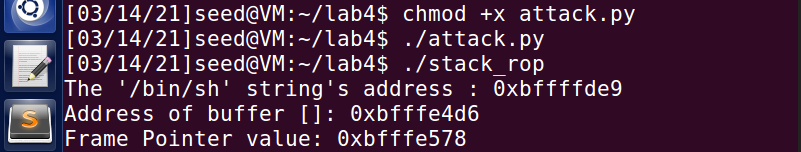
* We need to find the address values of system(), exit(), sprint(), setuid(), frame pointer and leaveret. For doing so, we go inside debugger of stack\_rop, whereas frame pointer address is printed while running stack\_rop code itself.



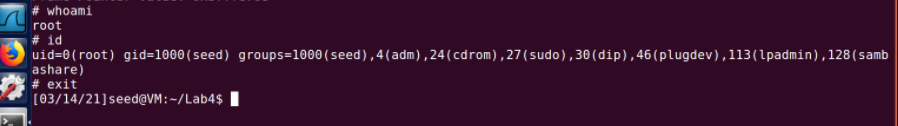
* These values are updated inside the attack.py file we create to launch an attack. The sprint() function is dynamically called to convert arguments to zero without us passing them externally.



* It is also important to note that setuid() is called before invoking system() to drop privileges and defeat the countermeasure as done earlier in Task 5.



* On running the attack.py file, we see below that we are able to gain root access to the shell.



* This is achieved by ROP chaining method where it overwrites the address at run time, by providing ROP payload which modifies the arguments into zero, thus enabling access to the root shell.