**Homework 4:- Return-to-libc Attack Lab**

**Name:- Aparna Krishna Bhat**

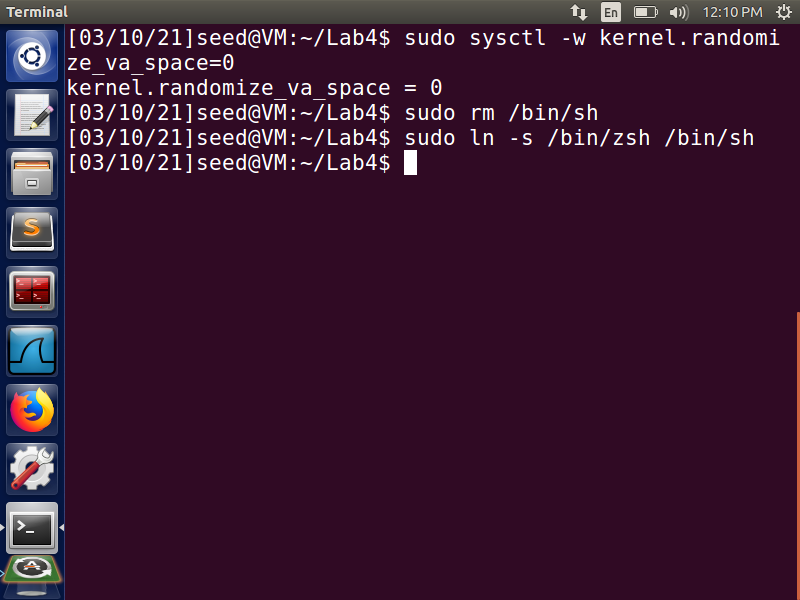
**ID:- 1001255079**

A ***return-to-libc*** attack is a computer security attack usually starting with a buffer overflow in which a subroutine return address on a call stack is replaced by an address of a subroutine that is already present in the process executable memory, bypassing the no-execute bit feature (if present) and ridding the attacker of the need to inject their own code.

**Turning off the countermeasures**

To carry out the Buffer Overflow attack, I disabled the countermeasure in the form of address space layout randomization. If it is enabled, then it would be difficult to predict the exact address of stack in the memory. Hence, I disable this countermeasure **by setting it to** **0 (false)** in the **sysctl file.** The compiler has certain countermeasures to the buffer overflow attack. To carry out the successful attack disable these countermeasures while compiling the program.

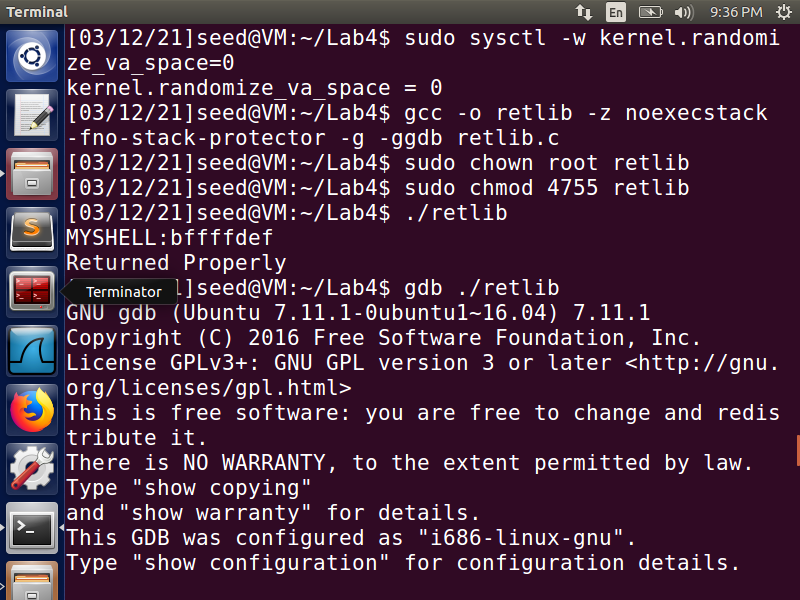
Also, changed the default shell from ***‘dash’*** to ‘***zsh’*** to avoid any countermeasures implemented in ‘bash’ for the SET-UID programs



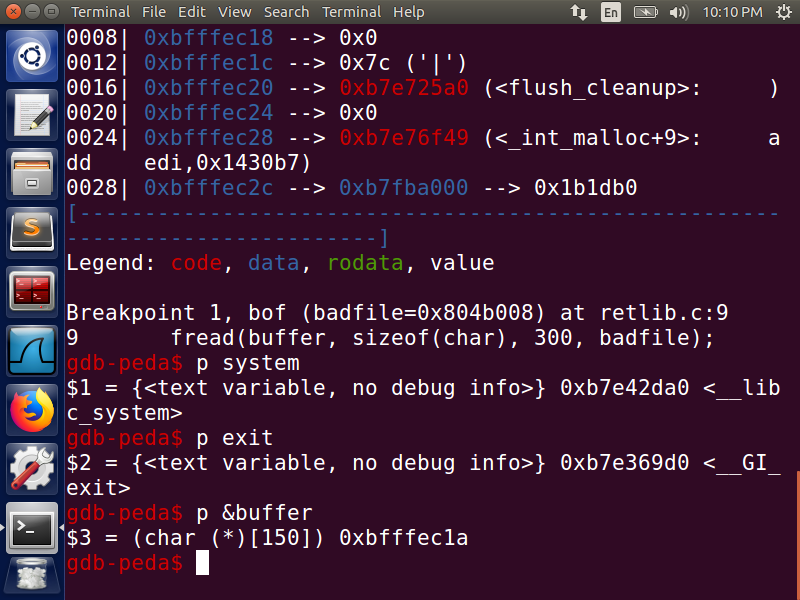
**Fig 1:- Countermeasures turned off**

**Task 1:- Finding out the addresses of libc functions**

Compile the vulnerable ***retlib.c*** program with executive stack option and turn off the Stack -Guard. Now make the vulnerable program a set UID program owned by root. These operations can be seen in the below screenshot.



We will use the system() and exit() functions in the libc library in our attack, so we need to know their addresses. Using the GNU ***gdb debugger*** is one of the easiest ways to find the addresses. From the ***gdb commands***, we can notice that the address for the ***system()*** function that we have obtained is ***0xb7e42da0***, and the address for the ***exit()*** function obtained is ***0xb7e369d0***.

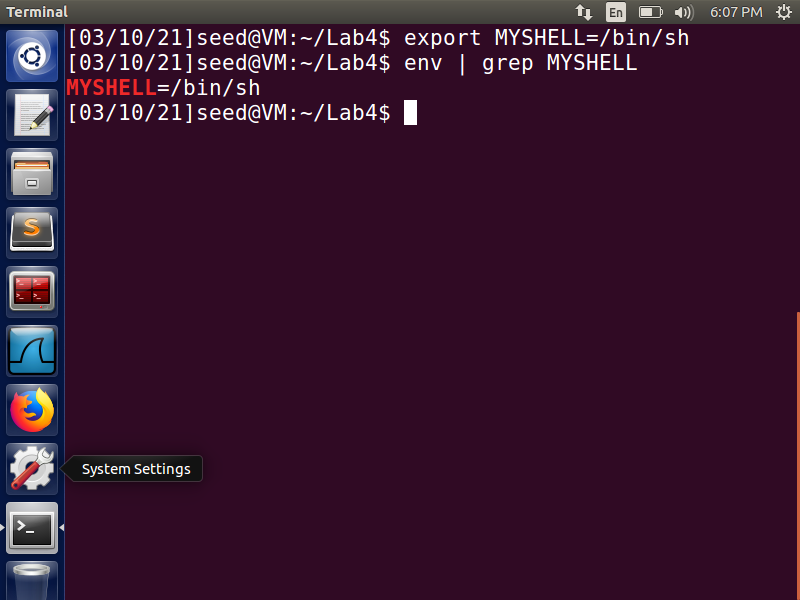


**Fig 2:- System and exit address**

We also find the address of the buffer which helps us to determine where our return address will be located.

**Task 2:- Putting the shell string in the memory**

Our attack strategy is to get the system() function to execute an arbitrary command. Since we would like to get to the root shell, we want the system() function to execute the “/bin/sh” program. Create and export a new shell environment variable called MYSHELL containing /bin/sh which will be passes as parameter to system function. Next, find the address of the location where this variable is stored. When we execute a program from a shell prompt, the shell spawns a child process to execute the program. The environment variable of the child process will contain all the exported shell variables. This makes it simple for us to put some arbitrary string in the child process’s memory.



We will use the address of MYSHELL variable as an argument to system() call. Next, we find the address of the location where this variable is stored by creating an envvar.c program and executing it. We obtain the ***memory location*** for the variable as ***bffffdef.***

**C- code**

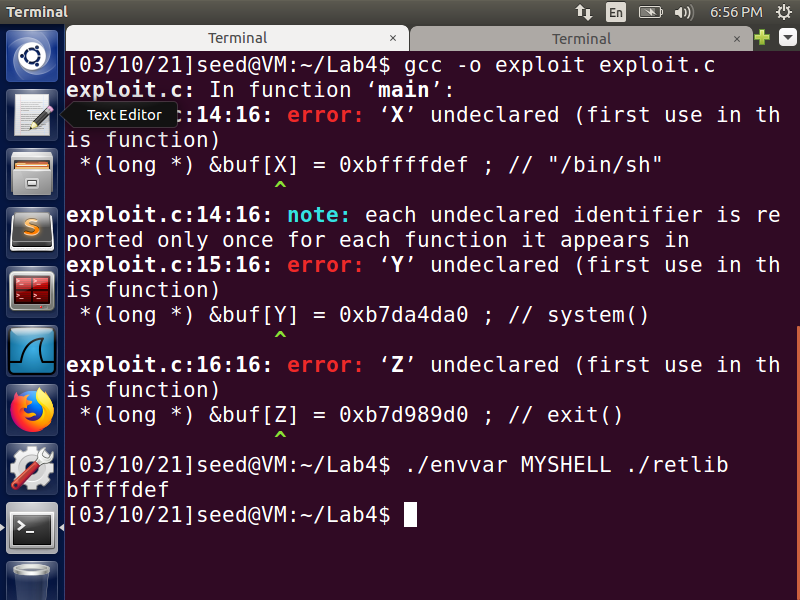
void main(){

char\* shell = getenv("MYSHELL");

if (shell)

printf("%x\n", (unsigned int)shell);

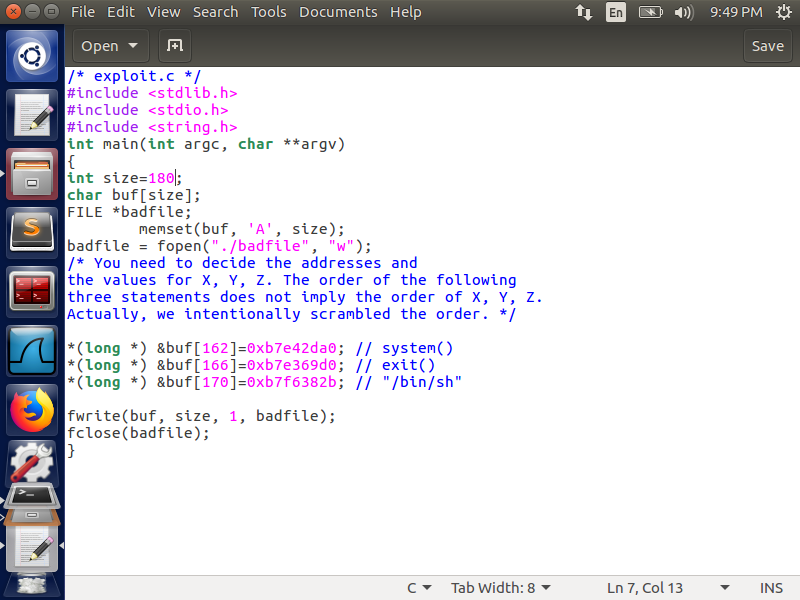
}



**Fig 3:- \bin\sh location obtained**

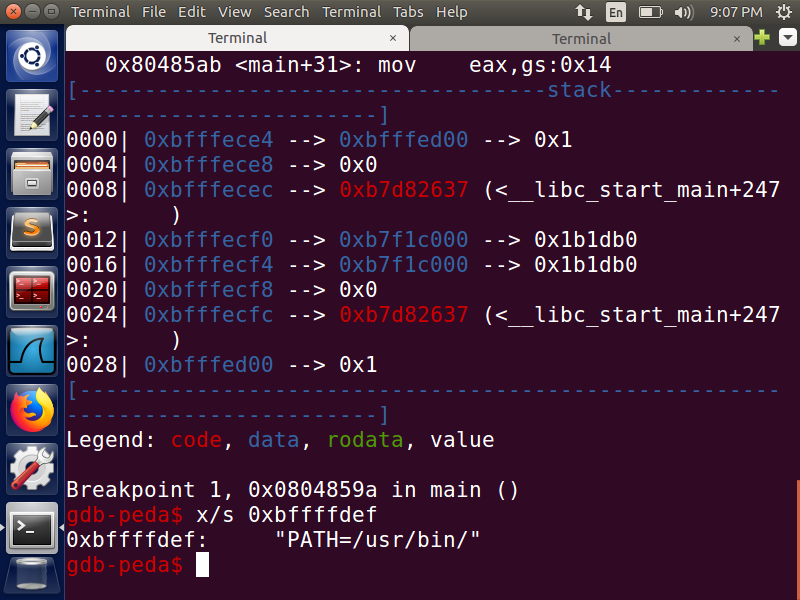
**Task 3: Exploiting the buffer-overflow vulnerability**

Create and compile the exploit program exploit.c which creates the badfile.

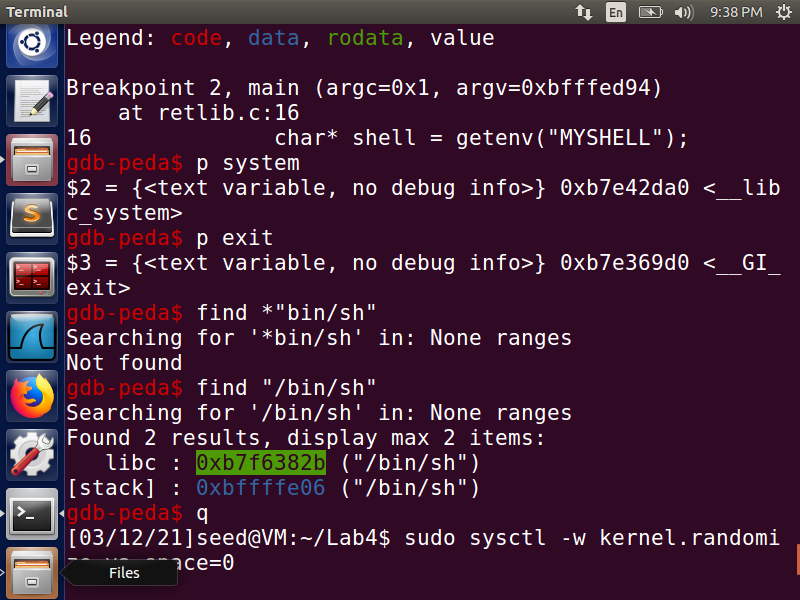


**Fig 4:- exploit.c file**

The obtained address 0xbffffdef obtained for MYSHELL variable is further verified using gdb which in not the correct location.



Using find “\bin\sh” in the gdb we obtain the correct location for”\bin\sh” as ***0xb7f6382b*** which will be used in exploit.c file. The address of stack frame pointer when running the code in gdb is different from running it normally. So, you may corrupt the return address right in gdb mode, but it may not be right when running in normal mode.



**Fig 5:- Correct address of libc “/bin/sh”**



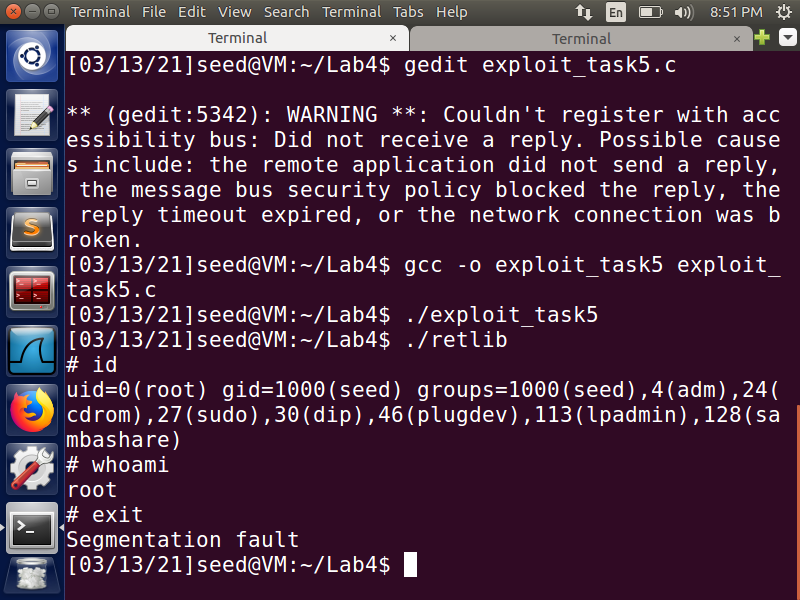
The given retlib.c program has a buffer overflow vulnerability. We are trying to read 300 bytes from the file into a buffer of size 150 using ***fread*** which is an unsafe function. As a result, we can exploit the program by manipulating the input to the program.

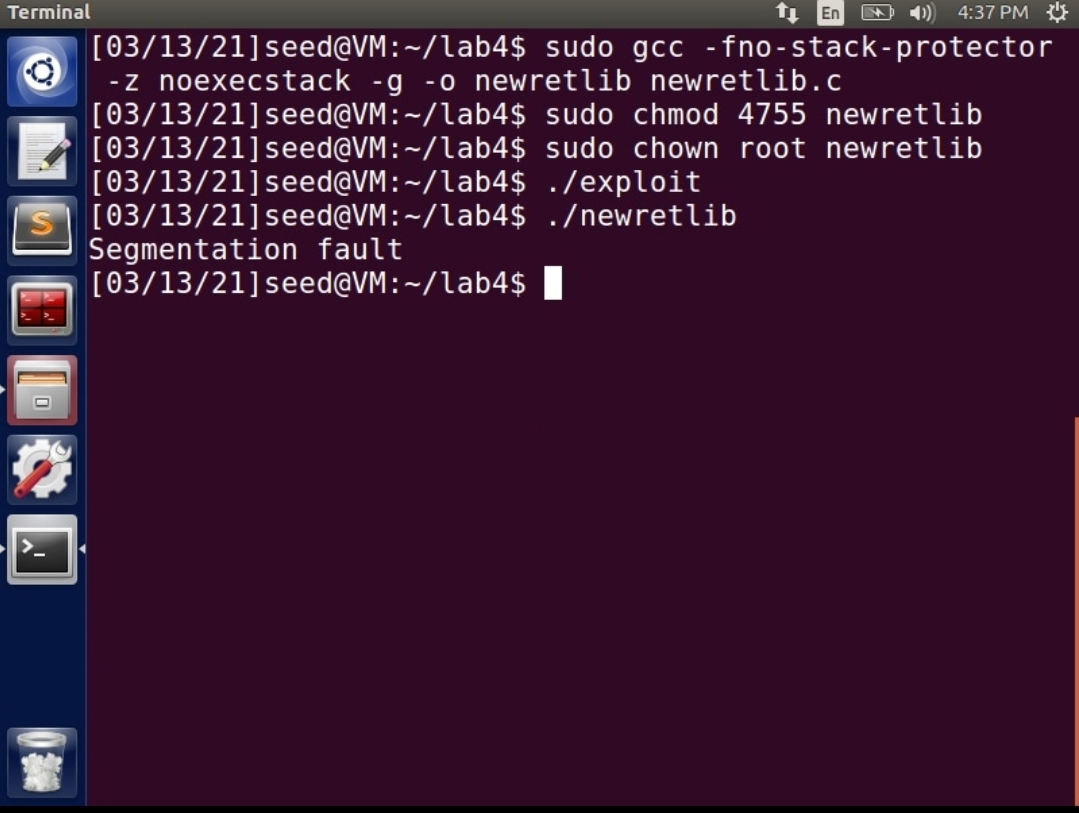
To find the offsets X, Y, Z in the exploit.c program, we analyze the stack of the retlib.c program.

1. The initial ***158*** bytes of buf[] in exploit.c are kept empty.
2. The next 4 bytes contain the return address after to which the control will point to after execution of bof(). Hence, we can overwrite this to execute system() function. Thus, at an offset 162 i.e., ***buf[162***], we place the address of system() function : ***0xb7e42da0***.
3. The control is transferred to next 4 bytes after the execution of system(). Hence, we place the address of exit() at offset 166. i.e., ***buf[166],*** we place the address of exit() function: ***0xb7e369d0***.
4. Next, set of 4 bytes will contain the address of ***“/bin/sh”*** as during the execution of system(), it will look for its parameters here. So, we place the address of ***“/bin/sh***” at offset 170. i.e., ***buf[170]*** =***0xb7f6382b***.

**Attack variation 1: Is the exit() function necessary?**

Answer:- No, it is not necessary even after commenting exit() in the exploit.c file and trying the attack again I am able to get to the root shell. however, without this function when system() returns, the program might crash, causing suspicions. When we exit will get segmentation fault.

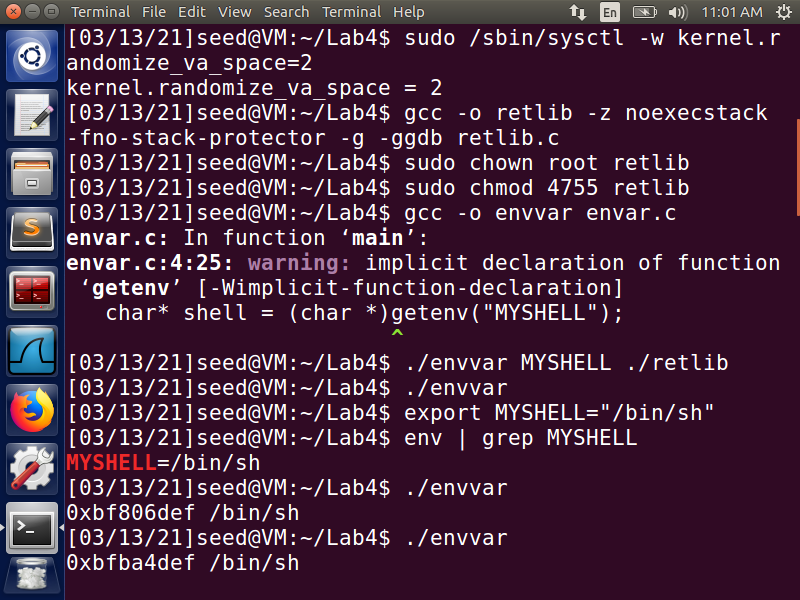




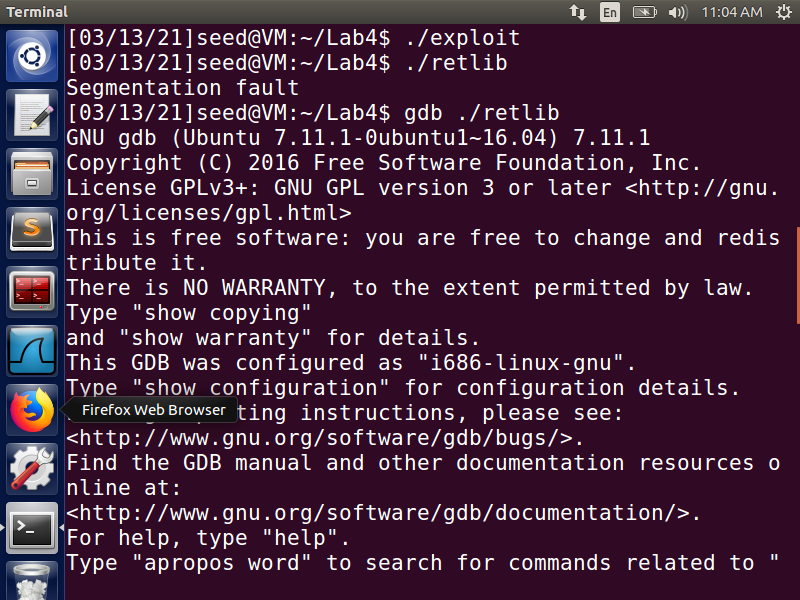
When we change the name of the file, the attack is not successful. This happens since the name of the executable that generates the environment variable is stored as part of the environment variable. Thus, both the environment variable creating executable and the vulnerable program executable should have the same length of characters in their names. We can see that the attack failed because the filename was changed because the position of the return address does not fit when the number of characters varies.

**Task 4: Turning on address randomization**

We compile the vulnerable program retlib.c with the address randomization turned on and perform the same attack as above. We get a segmentation fault error. These operations are shown in the below screenshots Fig 6 and Fig 7.

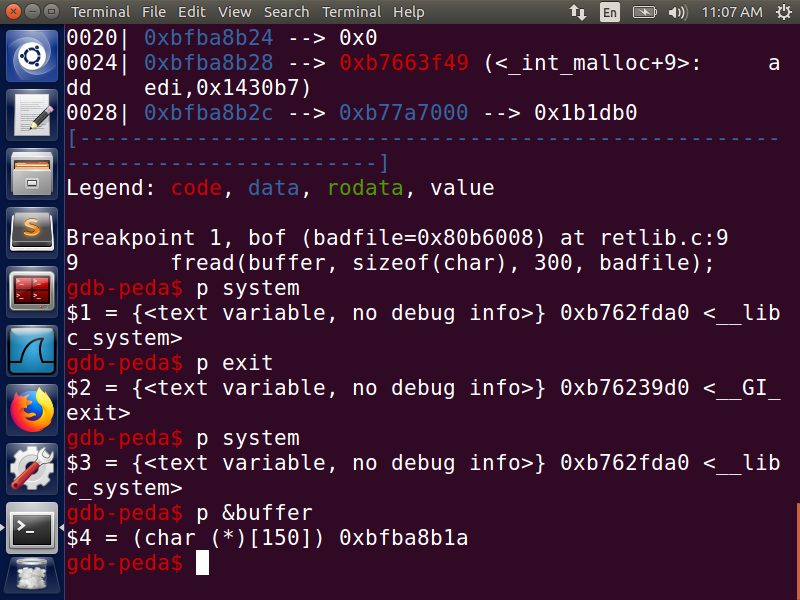


**Fig 6:- Address randomization turned ON.**



**Fig 7:- Segmentation error**

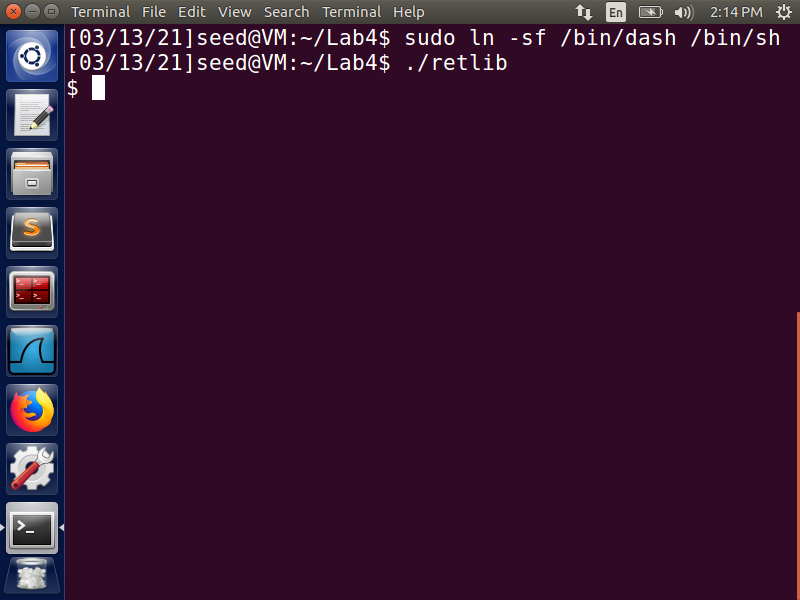
As the address randomization has been turned on, the address of the environment variable, system function location and the exit function location keep changing randomly as it can be noticed from Fig 6 and Fig 8. Address randomization is the process where the addresses of stack or heap in address space of any process is randomized. It makes it impossible for the attackers to guess the addresses on the stack. Hence, the probability of exploiting the vulnerability becomes very less. This serves as a strong protection mechanism against buffer overflow vulnerability.



**Fig 8:- Different address obtained for system() and exit()**

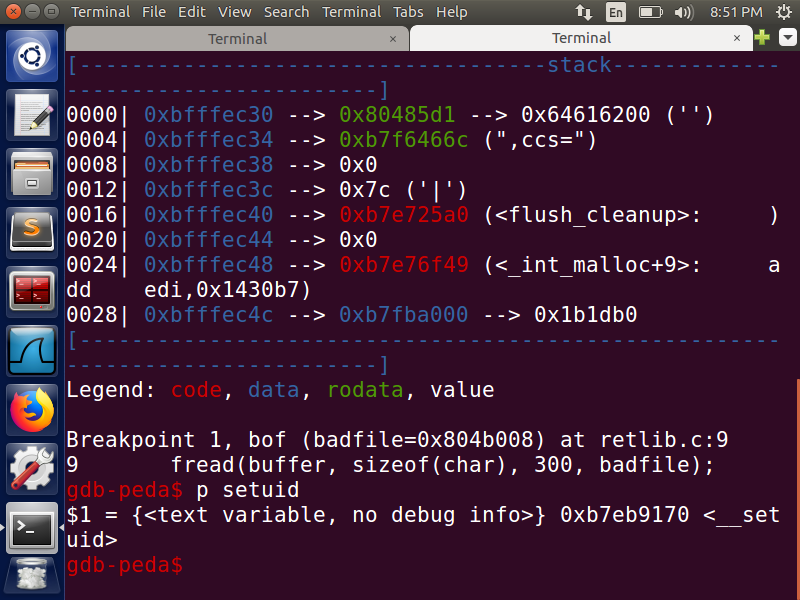
**Task 5:- Defeat Shell’s countermeasure**

For this task, we must first turn off the Address Randomization. Next, link /bin/sh to /bin/dash using command ***sudo ln -sf /bin/dash /bin/sh.*** On executing the retlib.c program we get a shell, but it is not the root shell. These operations are shown in the below Fig 9 screenshot.



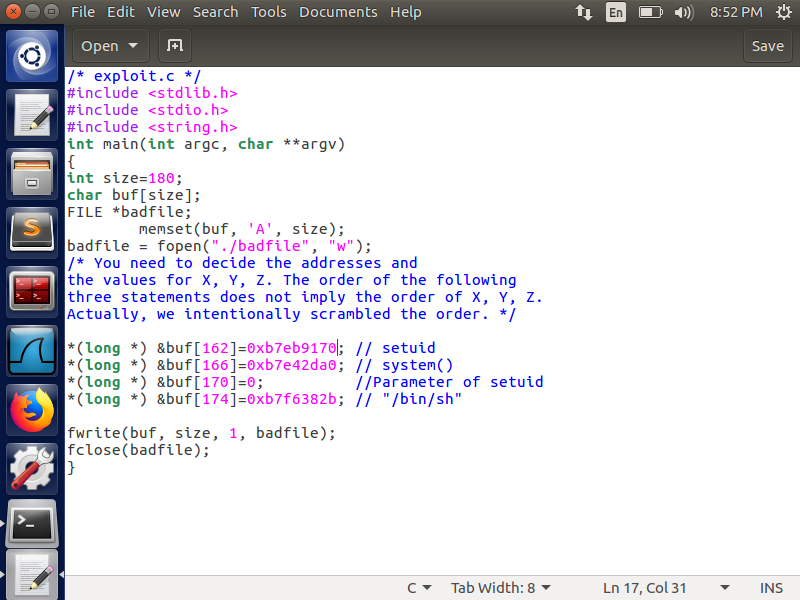
**Fig 9**

To defeat shell’s protection mechanism again malicious activities we need to need to call ***setuid(0)*** before calling the system() function. We can get the address of setuid() from gdb just like how we got the address of system() and exit().



**Fig 10:- Address of setuid() on gdb**

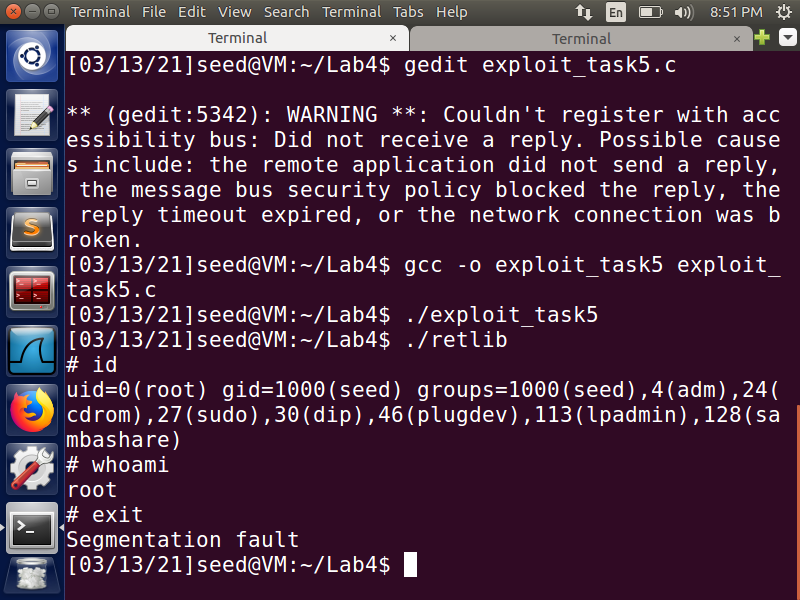
The parameter of setuid() should be at address ebp + 8. From the previous tasks we know that the ebp is pointing to 158, parameter, 0 should be at 166, 167, 168 and, 169 address making buf[170]= 0. We are considering 0 as parameter because uid=0 represents the root user.



**Fig 11:- exploit\_task5.c file**

At buf[162], setuid() gets called, its parameter is at offset of 8, i.e., at buf[170]. Then, the return address becomes buf[166] where we can call system() whose parameter would be at an offset of 8, i.e., at buf[174].

On compiling and executing the exploit\_task5.c program with ./retlib we get the root privileges. Fig 12 screenshot shows these operations.



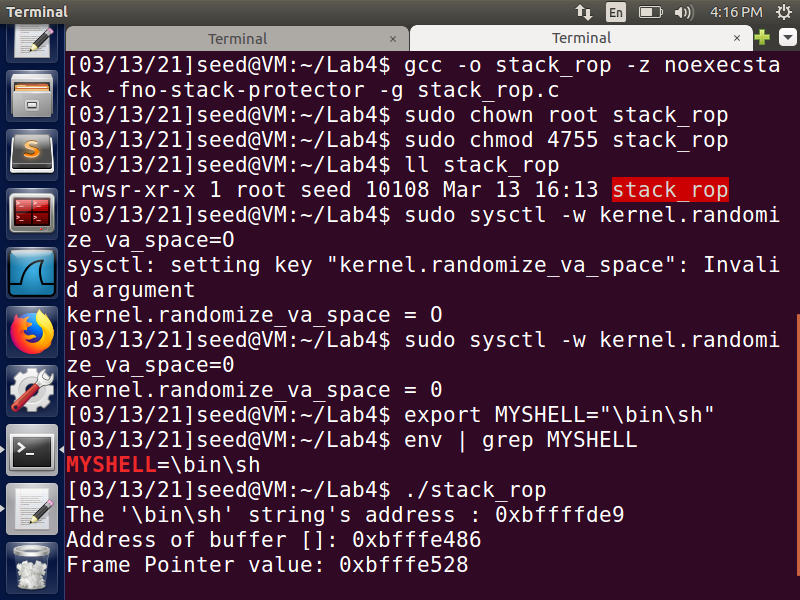
**Fig 12:- Defeated the Shell’s countermeasure**

We can notice that we are getting a segmentation fault when we exit the root shell. This is happening because the return address of system(), buf[170] is replaced by 0, parameter of setuid(), and there is no place for exit(). Thus, by invoking setuid() before system() we can gain root access as setuid(0) function is calling both effective uid and real uid and is setting both to 0 making it a non-SetUID program even though it still has root privileges.

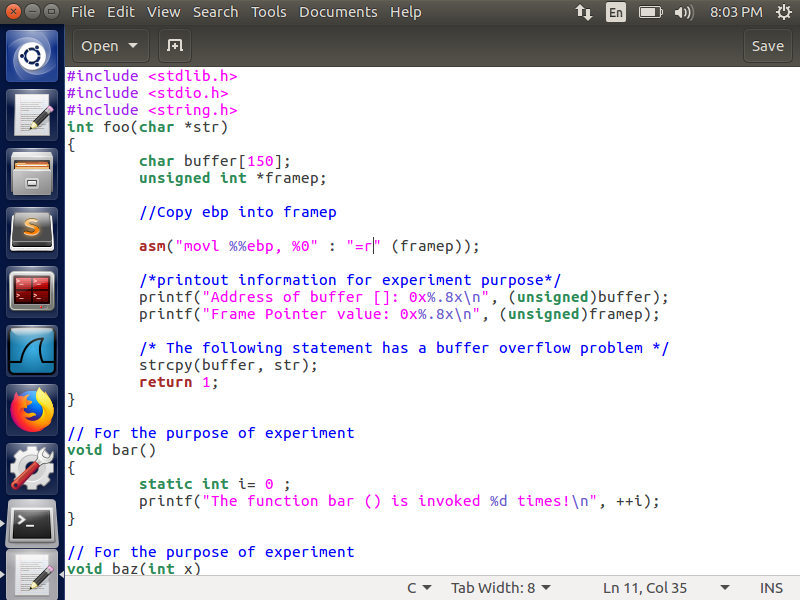
**Task 6:- Defeat Shell’s countermeasure without putting zeros in input**

The aim of this task to develop a technique to defeat shell’s protection mechanism without adding zeros in the input as it was in the previous case. In this task we will make use of Return Oriented Programming(ROP) to chain multiple functions and to change non-zero values to 0 internally when the program is executed making use of sprintf() function.

First, we create ***stack\_rop.c*** program, compile it and make it root owned Set-UID program. we save the value of the ebp register (the frame pointer) to a variable called framep in the stack\_rop.c program, so we can print out the frame pointer address. Before executing the program, we set an environment variable MYSHELL that contains a string “/bin/sh”. These operations are shown in the below Fig 13 screenshot.

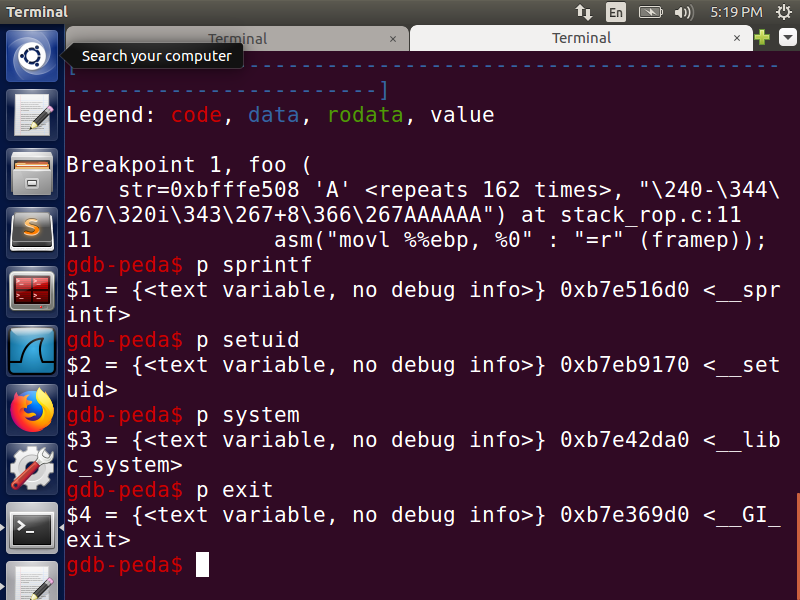


**Fig 13**



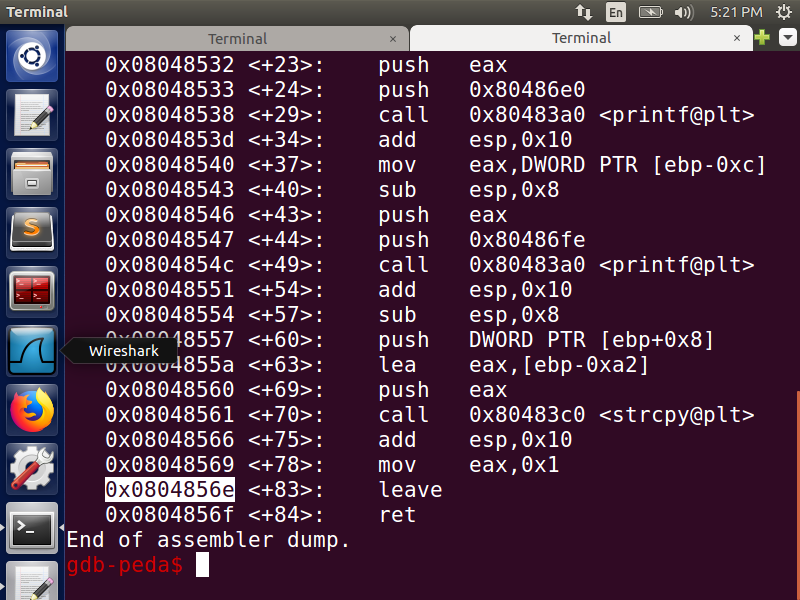
**Fig 14:- stack\_rop.c file**

We need to find the addresses of system(), exit(), leaveret, setuid(), sprint(). To obtain these values run debugger of ./stack\_rop.



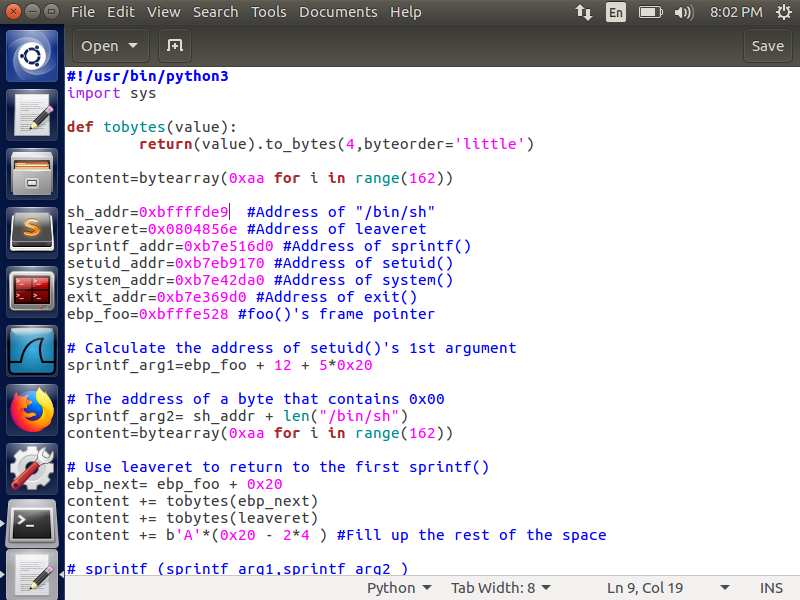
**Fig 15:- Address locations from gdb**

Run the command ***disassemble foo*** in gdb to get the address of ***leaveret.***



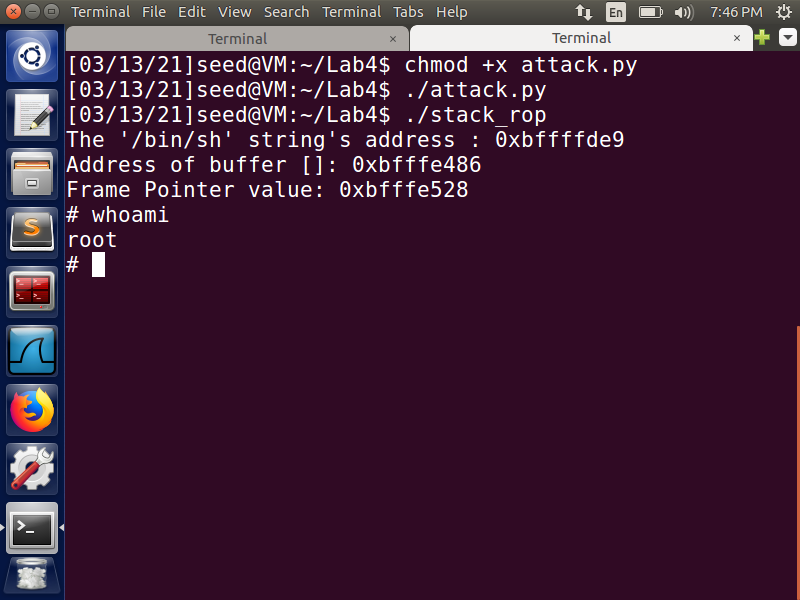
**Fig 16:-leaveret address**

Create a attack.py file and add the respective values from the above done steps. The idea of ROP is rather than use a single (libc) function to run your shellcode, string together pieces of existing code, called gadgets, to do it instead. Gadgets are instruction groups that end with ret. Using this chaining technique to defeat the countermeasure implemented by /bin/sh, so we can get a root shell using the system() function. In our construction, we place each function call's stack frame 0 x20 bytes apart, so if foo( )'s stack frame is at X(i.e., the frame pointer's value is X), the stack frame of the first function (i.e., the first sprintf()) will be at X + 4 + 0x20, the second function will be at X + 4 + 0x40, and so on. The setuid() function is the fifth on the call chain, so its stack frame will be at X + 4 + 5 \* 0x20. Since the first argument of a function is always at ebp + 8, thus the address of the setuid()'s argument will be at X + 12 + 5\*0x20.



**Fig 17:- attack.py file**

The sprint() function is dynamically called to convert arguments to zeroes without we passing them explicitly. Also, note that setuid() is called before invoking system() to drop the privileges. On compiling and executing the attack.py file we can generate the input, and then feed the input to the vulnerable program stack\_rop.c. able gain root access to the shell. These steps can be seen in the below Fig 18 screenshot. The following execution results show that we have defeated bash's countermeasure and have successfully obtained the root shell.



**Fig 18:- Got the root access**