INFORMATION SECURITY LAB

LAB 3: Buffer Overflow Vulnerability Lab

Name: Ankitha P

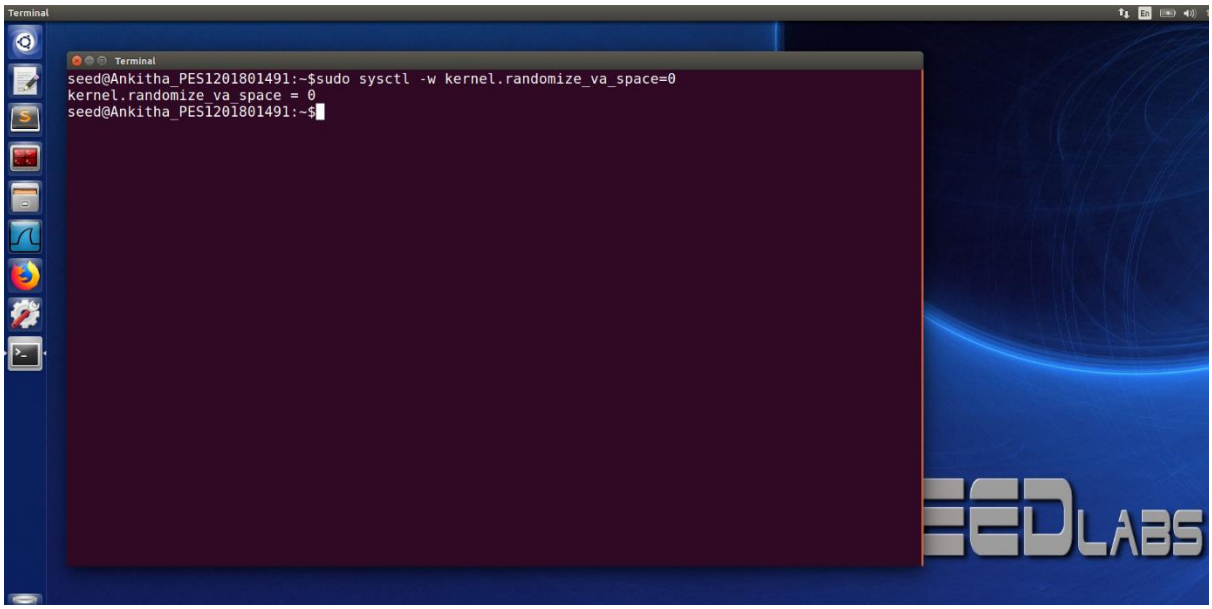
Class: 6 'D'

Date : 23/02/2021

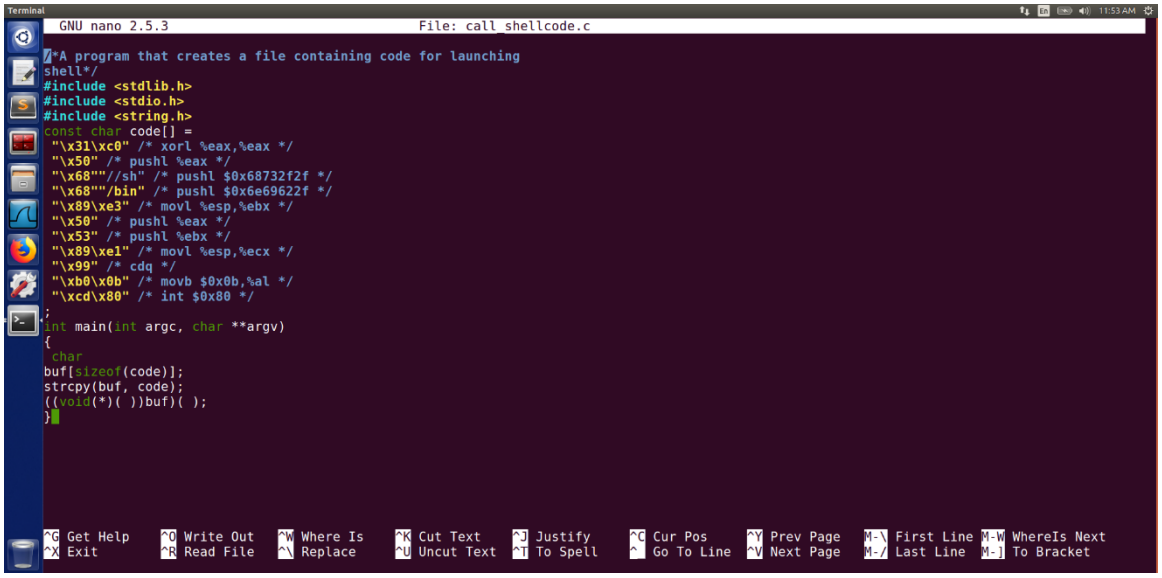
EXECUTION

TASK 1: Turning Off Countermeasures

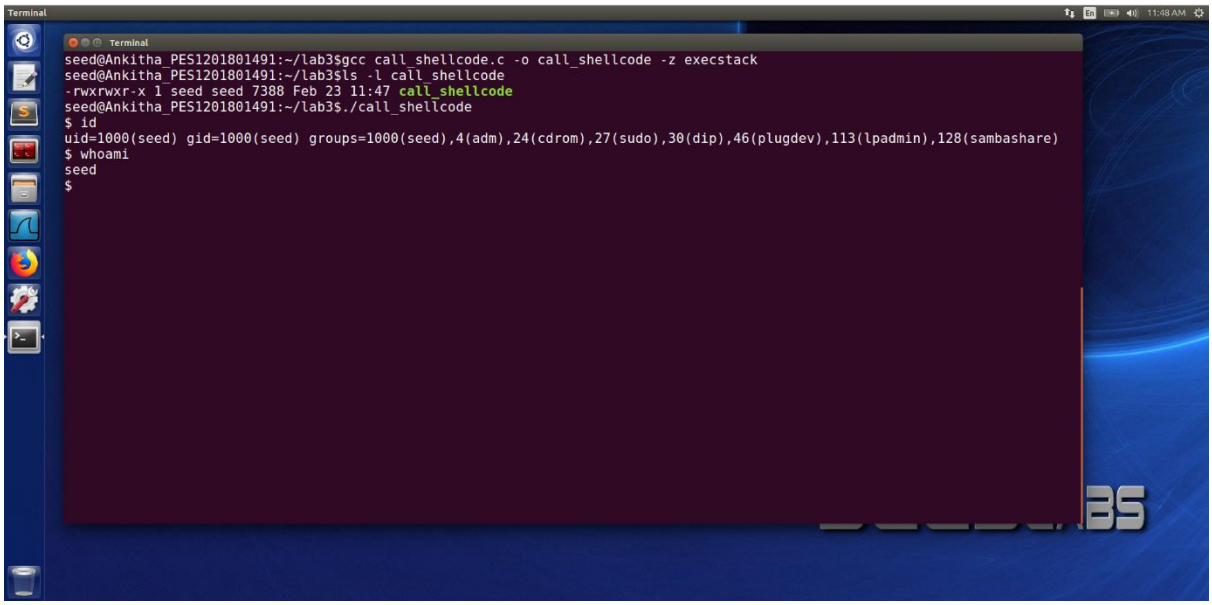
In order to perform the Buffer Overflow attack, first we disable the countermeasure in the form of Address Space Layout Randomization. If it is enabled then it would be hard to predict the position of stack in the memory. So, for simplicity, we disable this countermeasure by setting it to 0 (false) in the sysctl file, as follows:



Below screenshot shows call_shellcode.c file which has code to launch a shell:

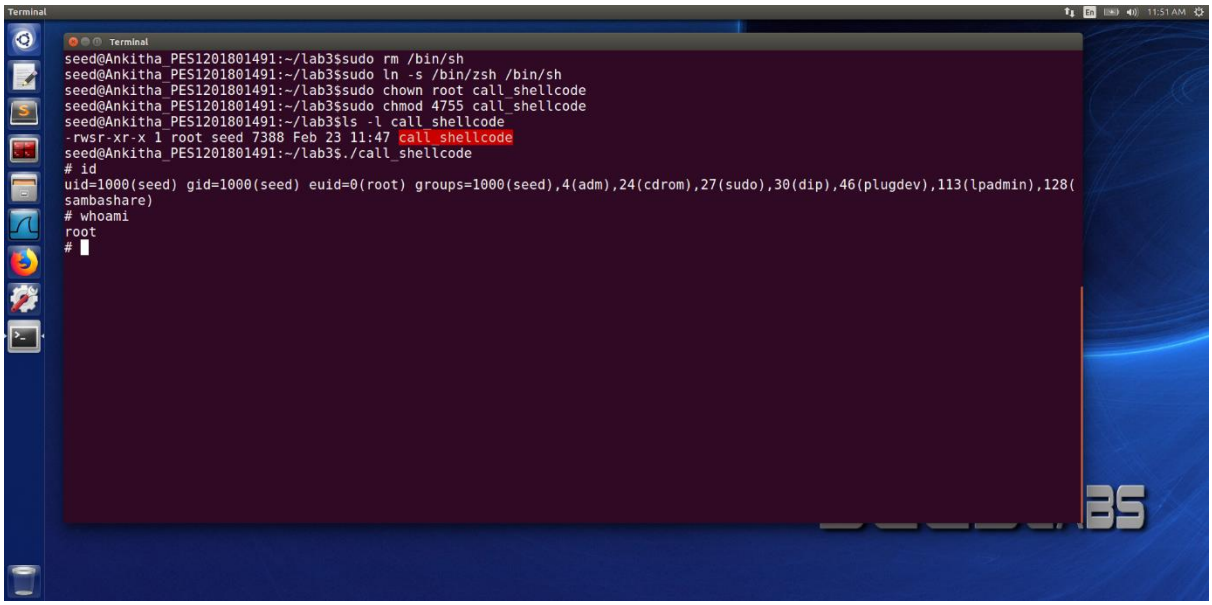


We compile this program by passing the parameter '-z execstack' to make the stack executable, in order to run our shellcode and not give us errors such as segmentation fault. The compiled program is stored in a file named 'call_shellcode.' Next, we execute this compiled program, and as seen, we enter the shell of our account (indicated by $). Since there were no errors, this proves that our program ran successfully, and we got access to '/bin/sh'. Here we observe that since it was not a SET-UID root program, nor we were in the root account, the terminal was of our account and not the root. This can be confirmed by the command id which shows user id to be that of the seed account and whoami command shows current account user as seed.



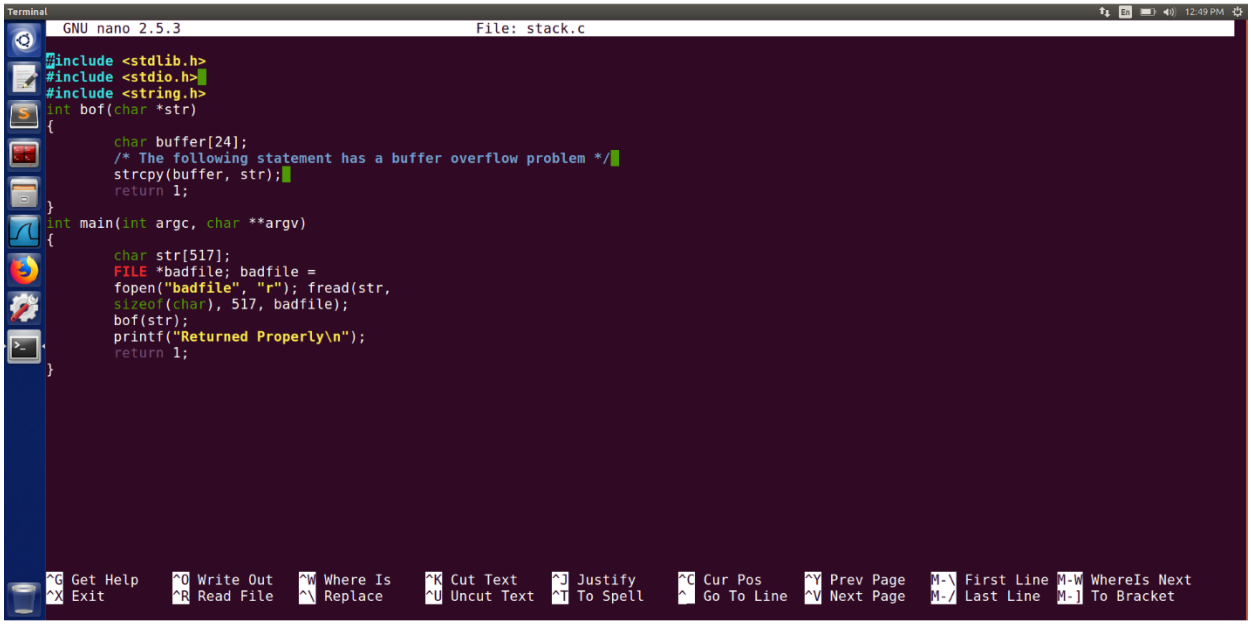
Since our victim program is a Set-UID program, and our attack relies on running /bin/sh, the countermeasure in /bin/dash makes our attack more difficult. Therefore, we will link /bin/sh to another shell that does not have such a countermeasure. Hence we have changed the default shell from 'dash' to 'zsh' to avoid the countermeasure implemented in 'bash' for the SET-UID programs.

Then we make the call_shellcode program a set-UID program by giving ownership to root and setting execution permission mode to 4755. On, running the compiled program, we successfully obtain the root shell as shown below :

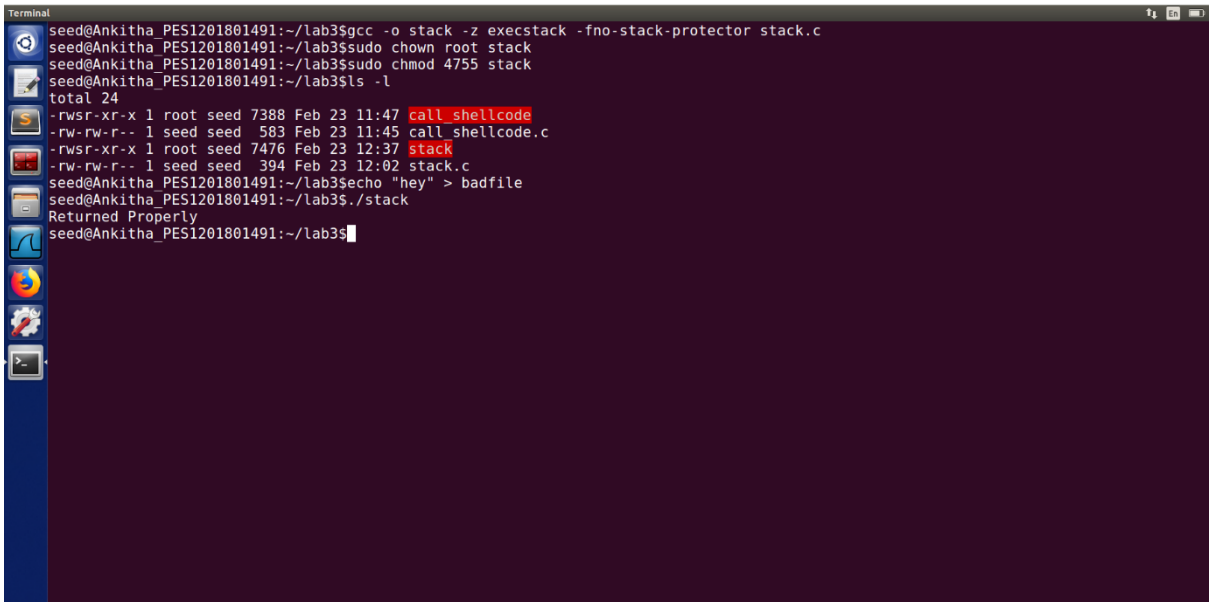


TASK 2: Vulnerable Program

The below program has a buffer overflow vulnerability. It reads input from a file called badfile and passes the read data to a buffer via the function bof(). We declared the original input to have a maximum length of 517 bytes, but the buffer in bof() function is only 24 bytes long. If our badfile has more than 24 bytes worth of data, there will be a buffer overflow on the strcpy function between buffers which doesn't check lengths.

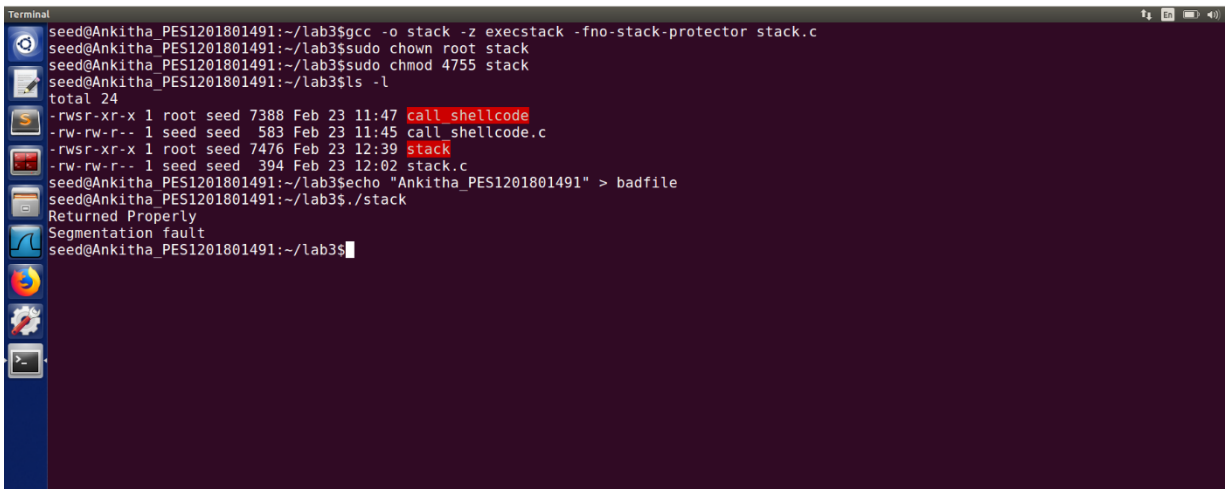


Next, we compile the given vulnerable program stack.c and while compiling, we disable the StackGuard Protection mechanism with the help of -fno-stack-protector tag and make the stack executable by adding -z execstack tag. Also, the compiled program, stored in ‘stack’, is then made a SETUID root program.



In the above screenshot, we pass a very small input which is less than 24 bytes and hence the stack program functions properly.

Now when the content of badfile is greater than the buffer in bof() which is 24 bytes , there is a segmentation fault due to buffer overflow.



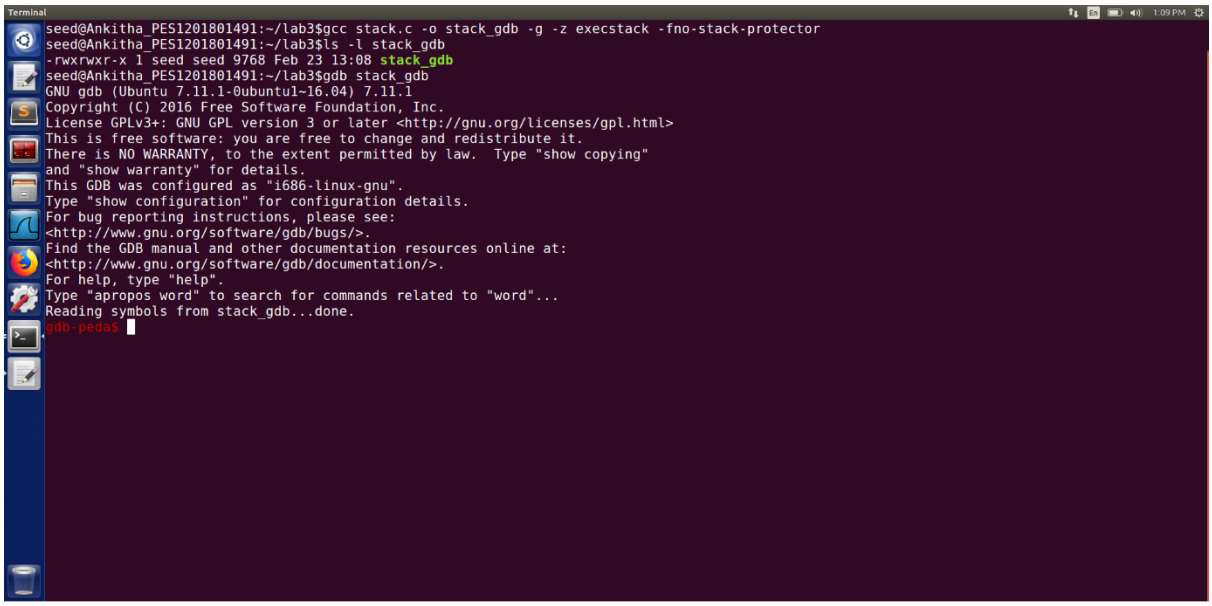
Since this program is a Set-UID program, if a normal user can exploit this buffer overflow vulnerability, the normal user might be able to get a root shell.

Task 3: Exploiting the Vulnerability

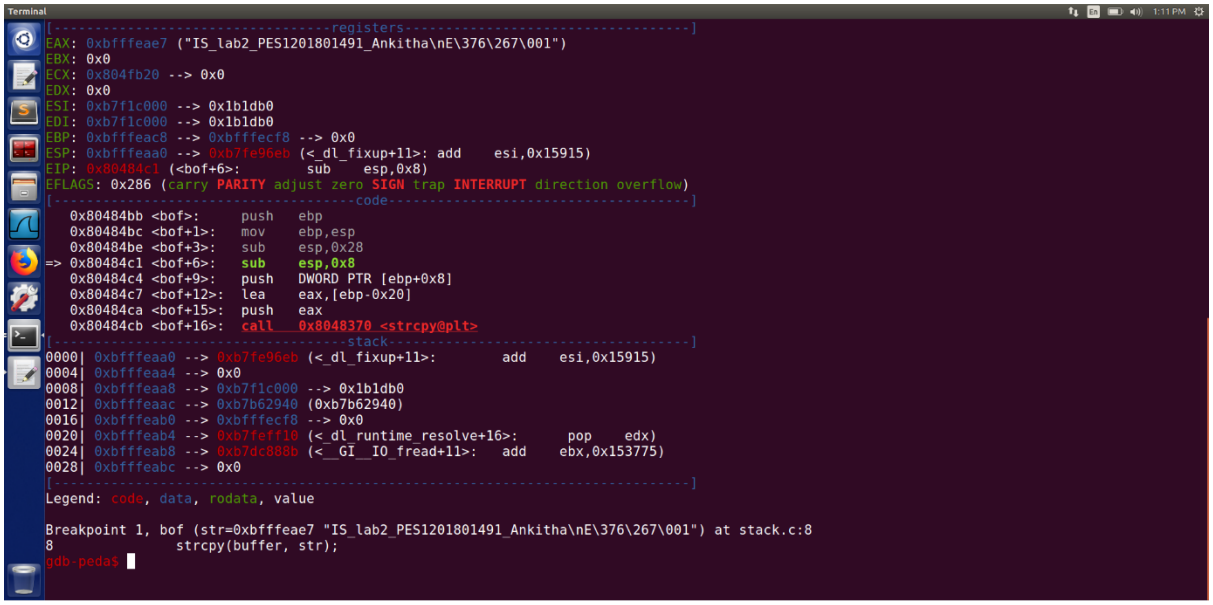
We now use the above vulnerable SET-UID root program to gain access to the root shell. Since, we have disabled Address Space Layout Randomization, we know that our process

will be stored in around the same memory always in the stack. So, in order to find the address of the running program in the memory, we compile the program in debug mode. Debugging will help us to find the ebp and the offset, so that we can construct the right buffer payload that will help us to run our desired program.

So, we first compile the program in the debug mode (-g option), with the StackGuard countermeasure disabled and Stack executable and then run the program in debug mode using gdb:

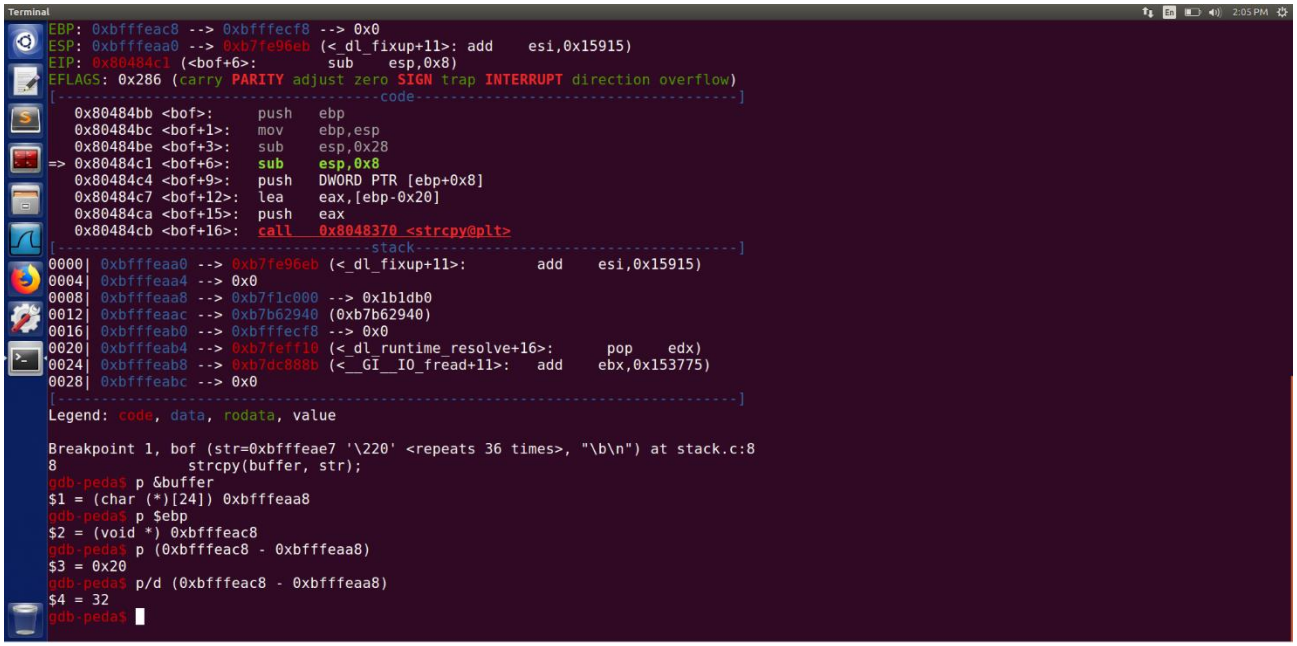


In gdb, we set a breakpoint on the bof function using b bof, and then start executing the program:



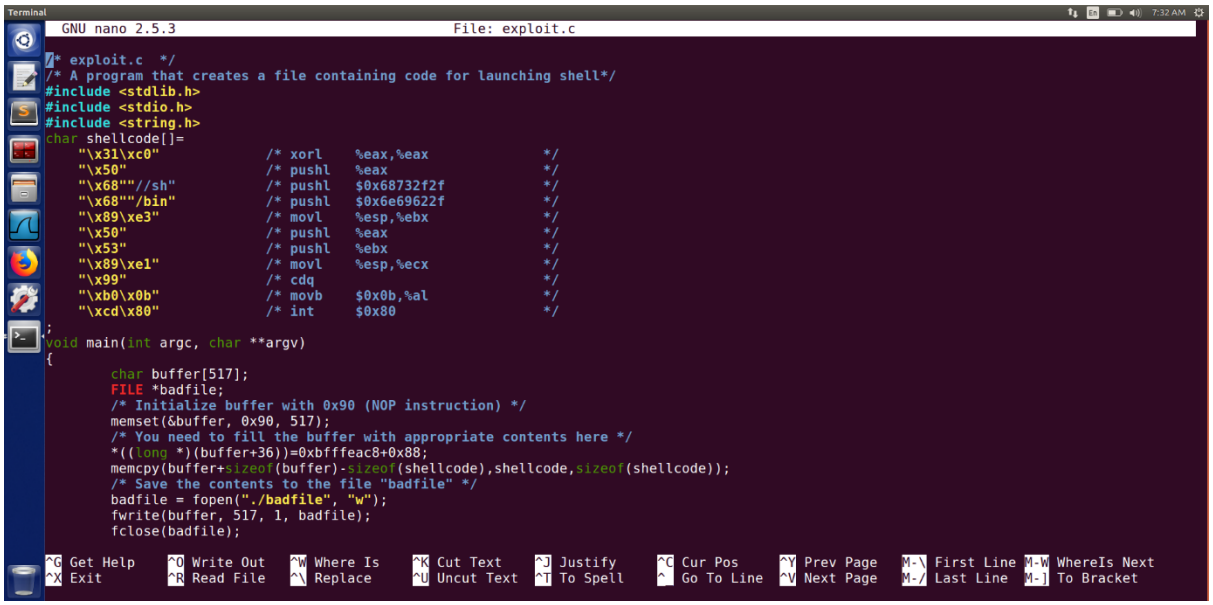
The program stops inside the bof() function due to the breakpoint created. EBP stack frame value is used as a reference when accessing local variables and arguments of the bof function. We also print out the buffer value to find the start of the buffer and the difference of the two in order to find return address value's address to our malicious program.

The below screenshot shows the commands:

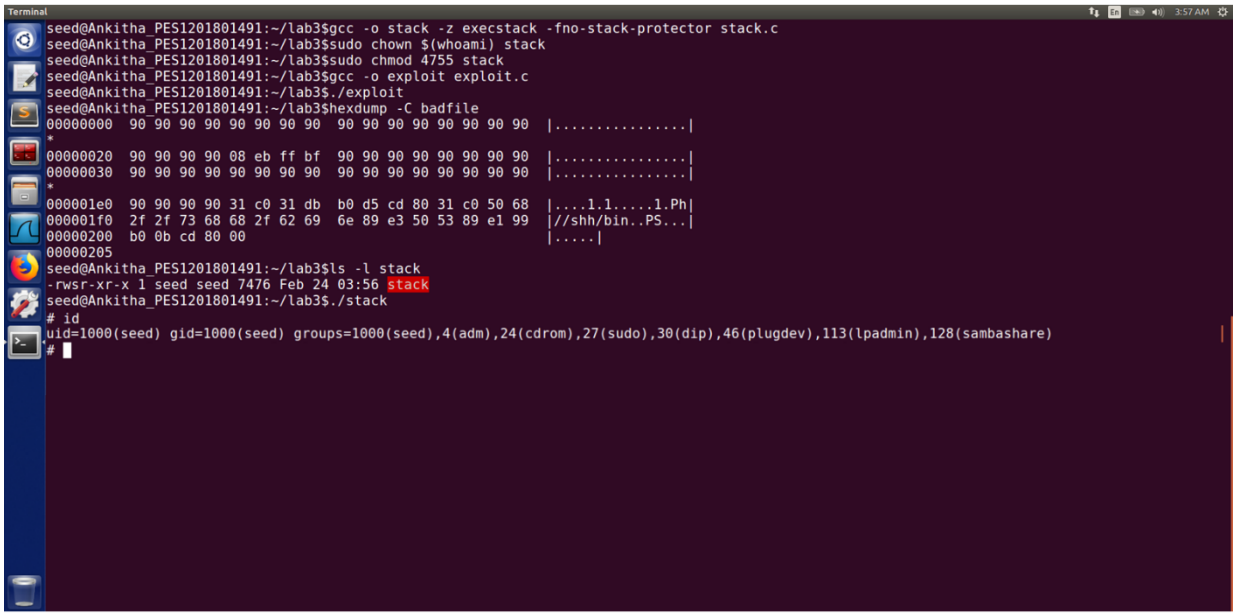


The difference between ebp and buffer start can be seen in the output, and by the layout of the stack, we know that return address will be 4 bytes above where the ebp points. Hence, the distance between the return address and the start of the buffer is 36, and so the return address should be stored in the badfile at an offset of 36. Here we have added 0x88 (128) into the ebp address . As the exploit has buffer size of 517 among which 36 bytes are used by the stack and 25 bytes are used by shell code so I added a value in between it like 0x80 that will result in replacement of current return address by address containing the NOP codes.

The exploit.c file is modified as below:

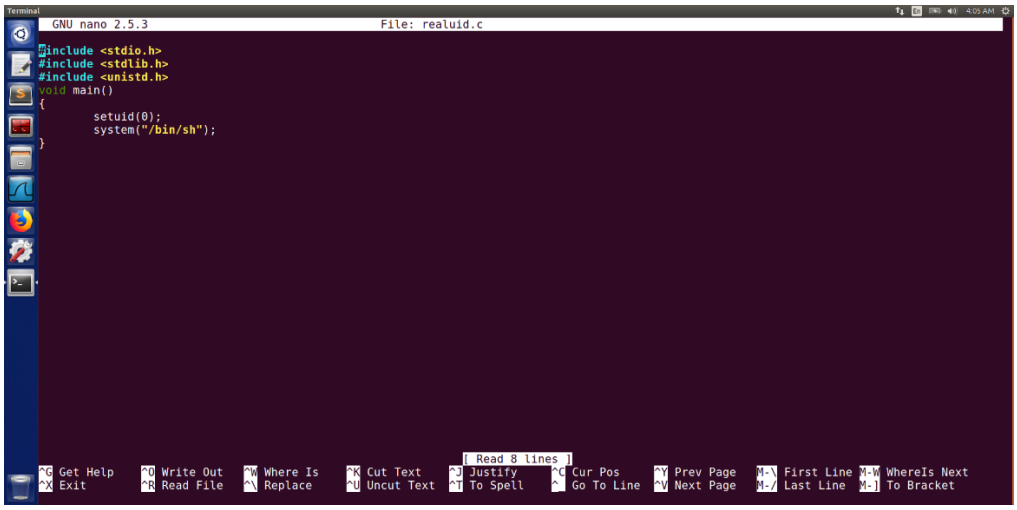


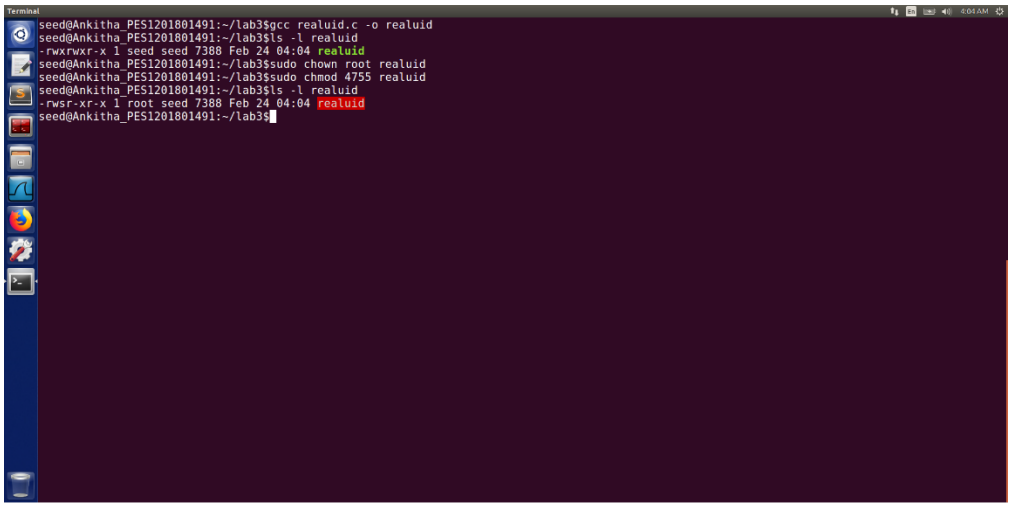
We remove previously created badfiles and compile the exploit.c code. On running the exploit.c code, a badfile is created. Next, we run the vulnerable SET-UID program that uses this badfile as input and copies the contents of the file in the stack, resulting in a buffer overflow. The # sign indicates that we have successfully obtained the root privilege by entering into the root shell. The real user id is still seed, whereas the effective uid is now root.



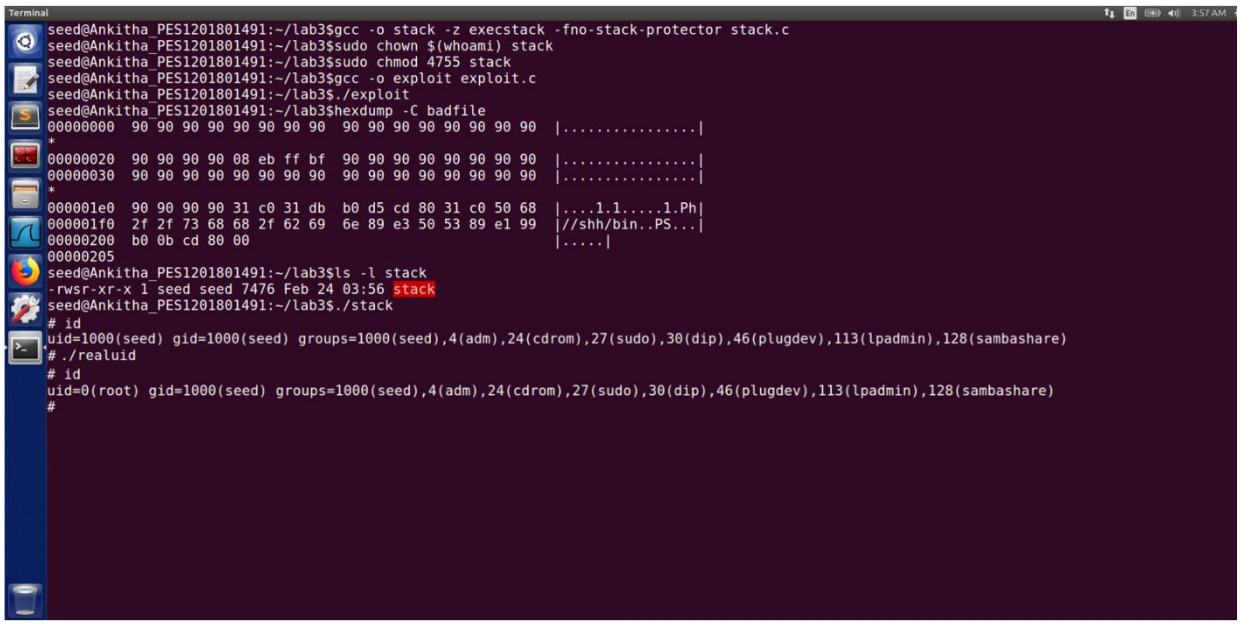
Hence, we have successfully performed the buffer overflow attack and gained root privileges.

Now, still our user id (uid) is not equal to the effective user id (euid). Therefore, in the next step we run our program to turn our real user id to root as well. We compile the following program realuid.c that changes the uid of the account to 0, which is of the root:



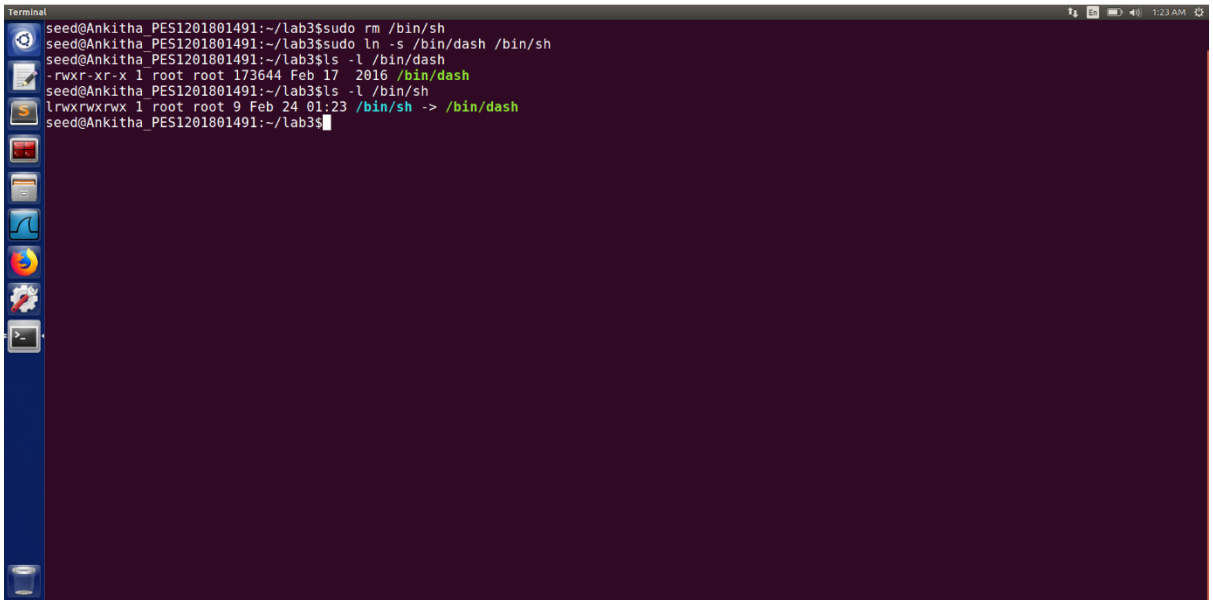


We run the compiled program realuid in the root shell to set the uid as 0. Since, we have already obtained root privileges because of the buffer overflow attack, we are able to change the user id to 0 without errors as shown below:

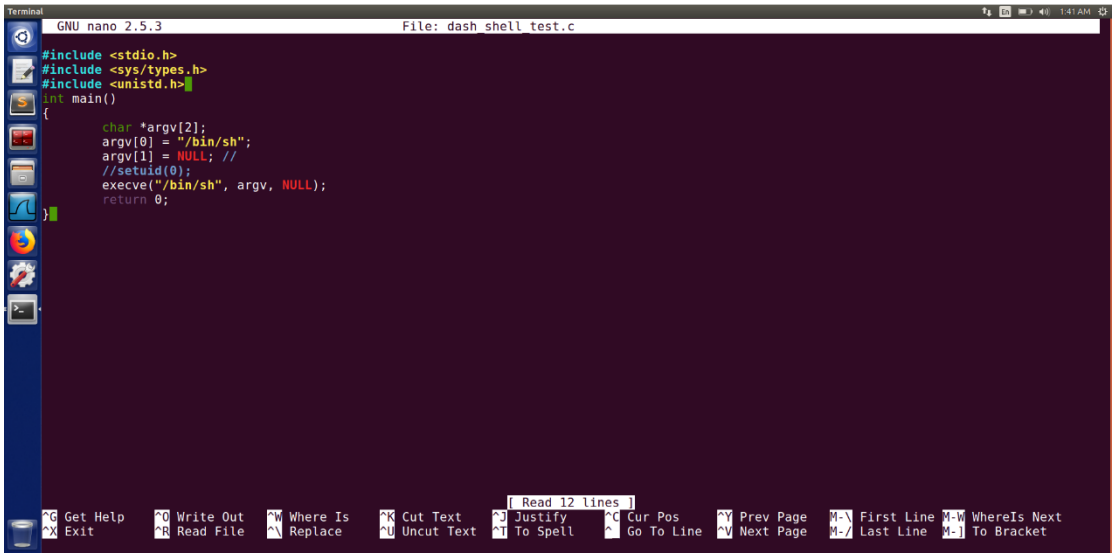


Task 4: Defeating dash's Countermeasure

In order to defeat the dash's countermeasure, we first change the /bin/sh symbolic link to point it back to /bin/dash again.

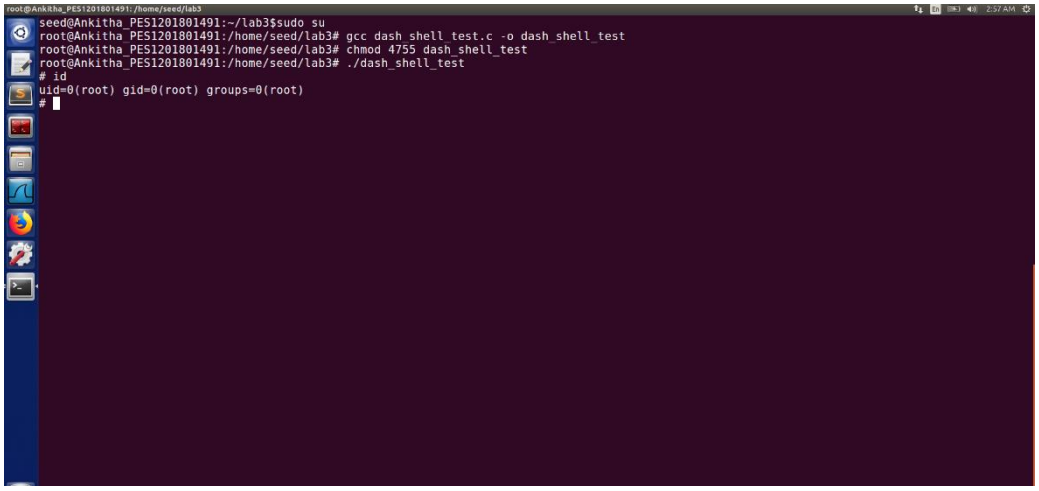


We compile the below dash_shell_test.c file which is used to spawn a shell. The setuid(0) line is commented.

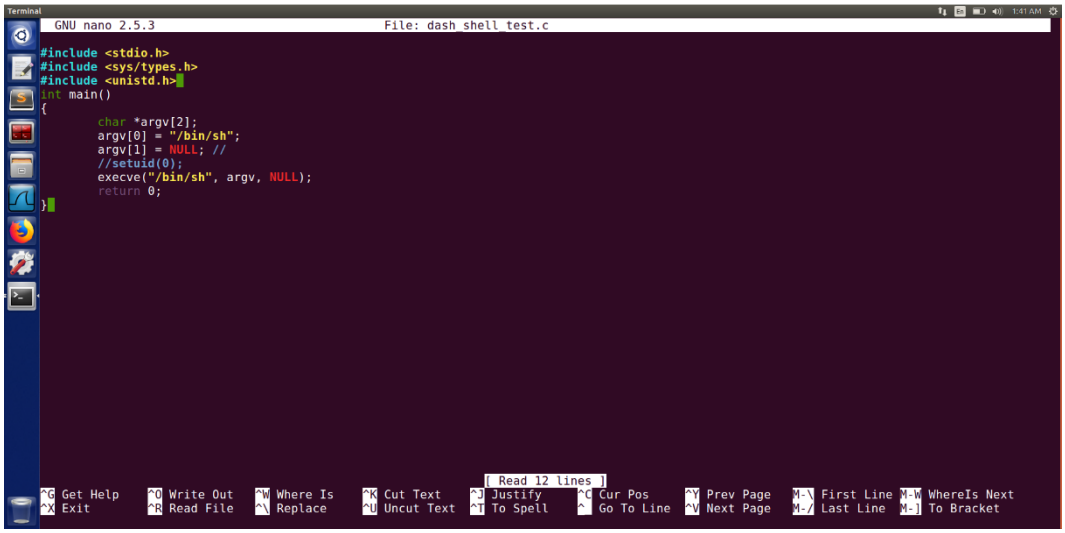


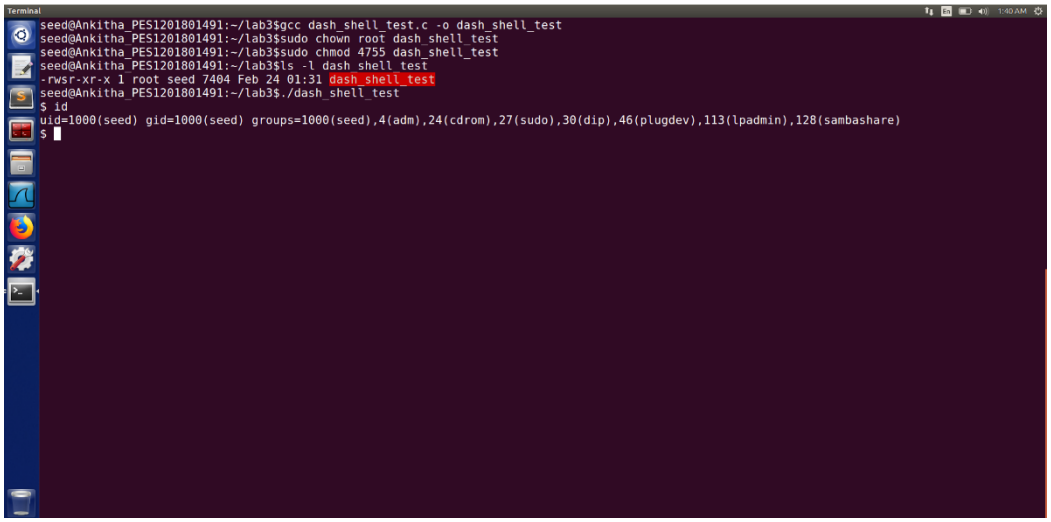
We then compile the dash_shell_test.c file and make it a SET-UID root program:

Running it on Root VM:



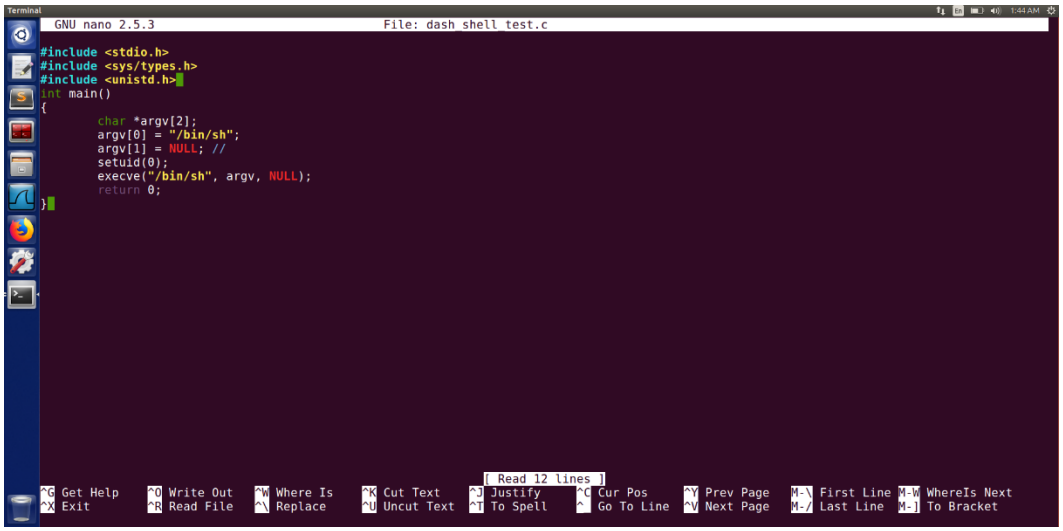
Setuid(0) line is commented:



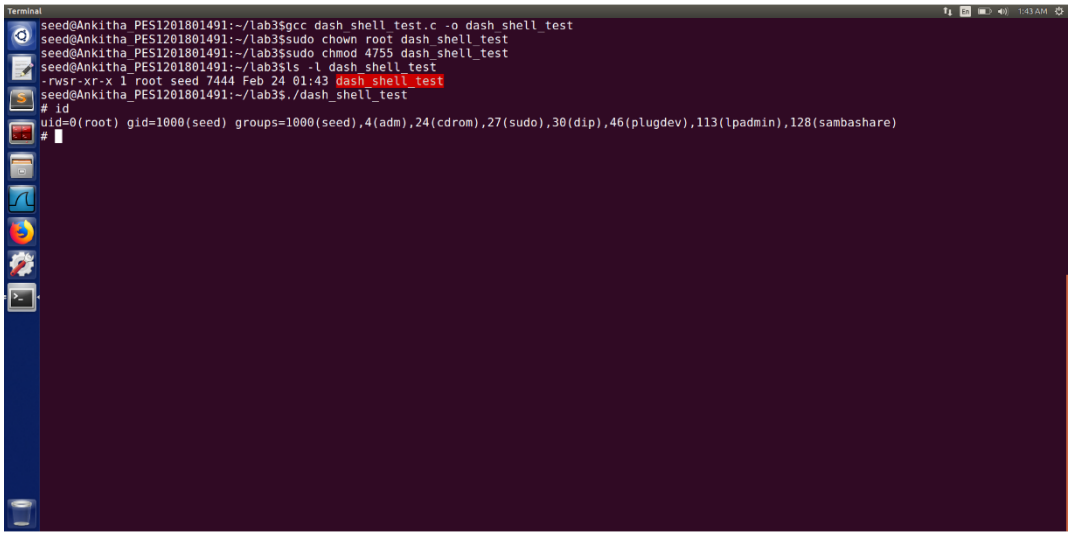


On running this program, we see that we enter our own account shell and the program's user id is that of the seed.

Uncommenting the setuid(0) line:



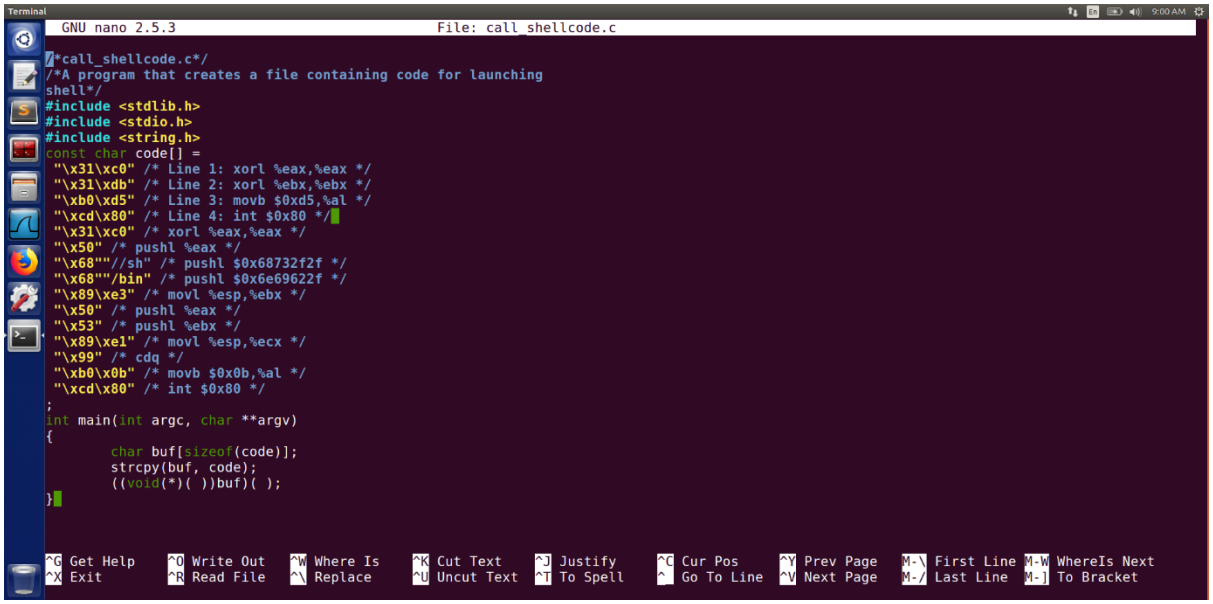
After removing the comment of setting the user id to 0, and running the program, we get:



As seen, we enter the root shell and on checking for the user ID, it is that of the root.

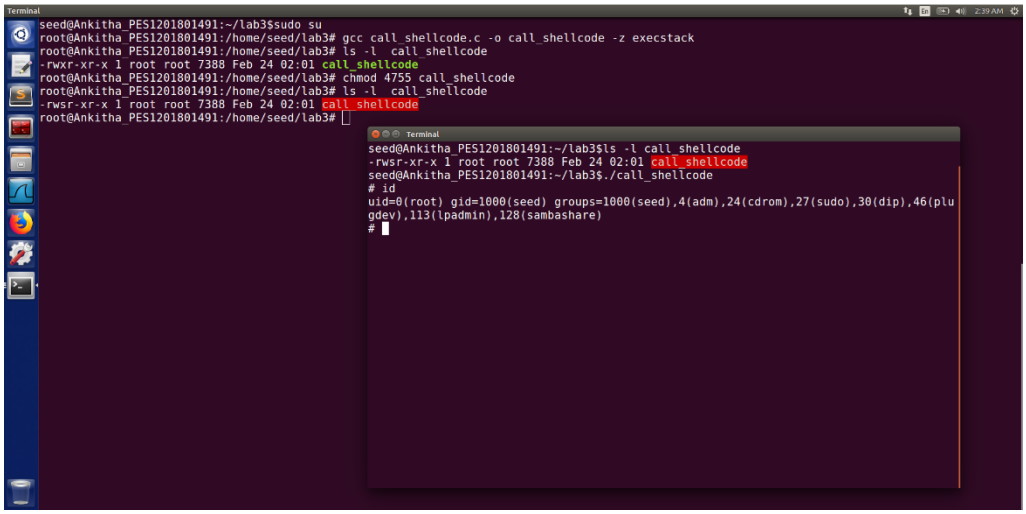
So, we see that both the times we get access to the shell, but in the first one it is not of the root because the bash program drops the privileges of the SET-UID program since the effective user id and the actual user id are not the same. Hence, it is executed as a program with normal priveleges and not root. But by having the setuid command in the program, it makes a difference because the actual user id is set to that of root, and the effective user id is 0 as well because of the SET-UID program, and hence the dash does not drops any privileges here, and the root shell is run. This command, therefore, can defeat the dash's countermeasure by setting the uid to that of the root for SET-UID root programs, providing with root's terminal access.

Adding the assembly code for invoking the system call at the beginning of our shellcode, before we invoke execve(). The call_shellcode.c is modified as follows:

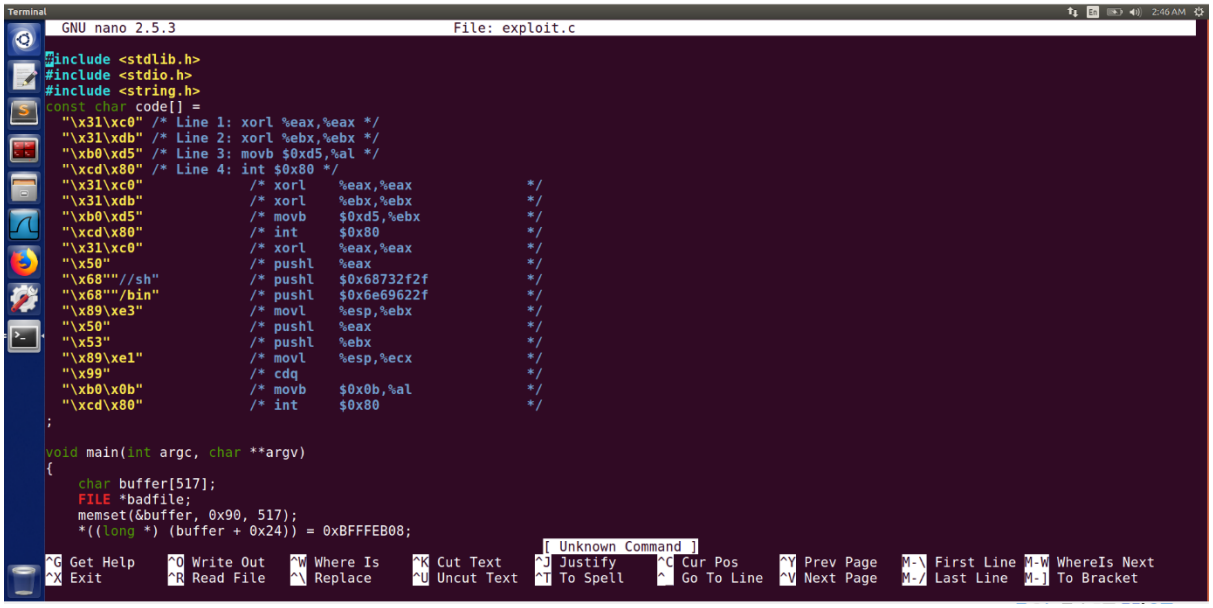


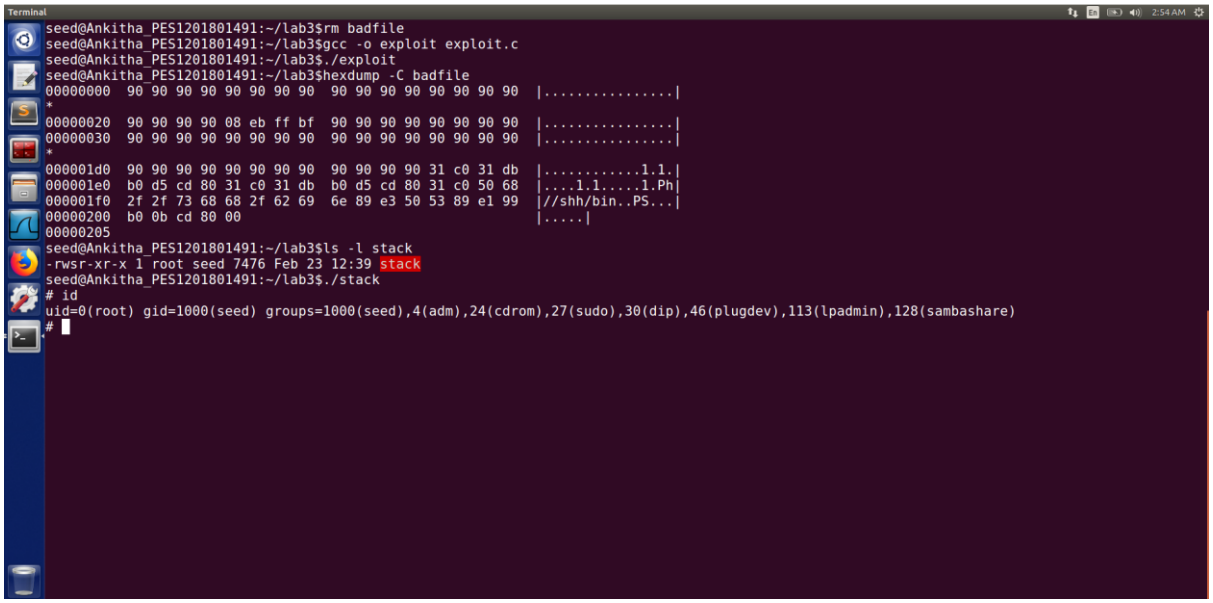
The updated shellcode adds 4 instructions: (1) set ebx to zero in Line 2, (2) set eax to 0xd5 via Line 1 and 3 (0xd5 is setuid()'s system call number), and (3) execute the system call in Line 4. Using this shellcode, we can attempt the attack on the vulnerable program when /bin/sh is linked to /bin/dash.

On compiling the call_shellcode.c we get:



Next, we try to perform the buffer overflow attack, in the same way we did it in task 2, but now the /bin/dash countermeasure for SET-UID programs is present due to the symbolic link from /bin/sh to /bin/dash. We add the assembly code to perform the system call of setuid at the beginning of the shellcode in the exploit.c, even before we invoke execve(). On running this exploit.c, we construct the badfile with updated code to be executed in the Stack, and then run the stack SETUID root program..

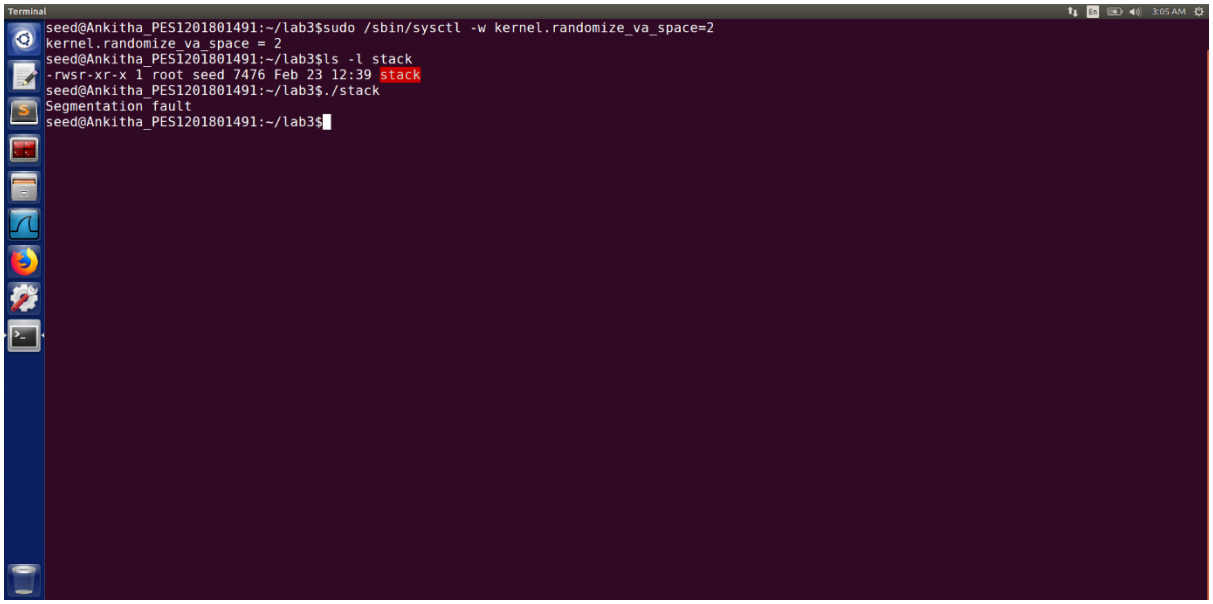




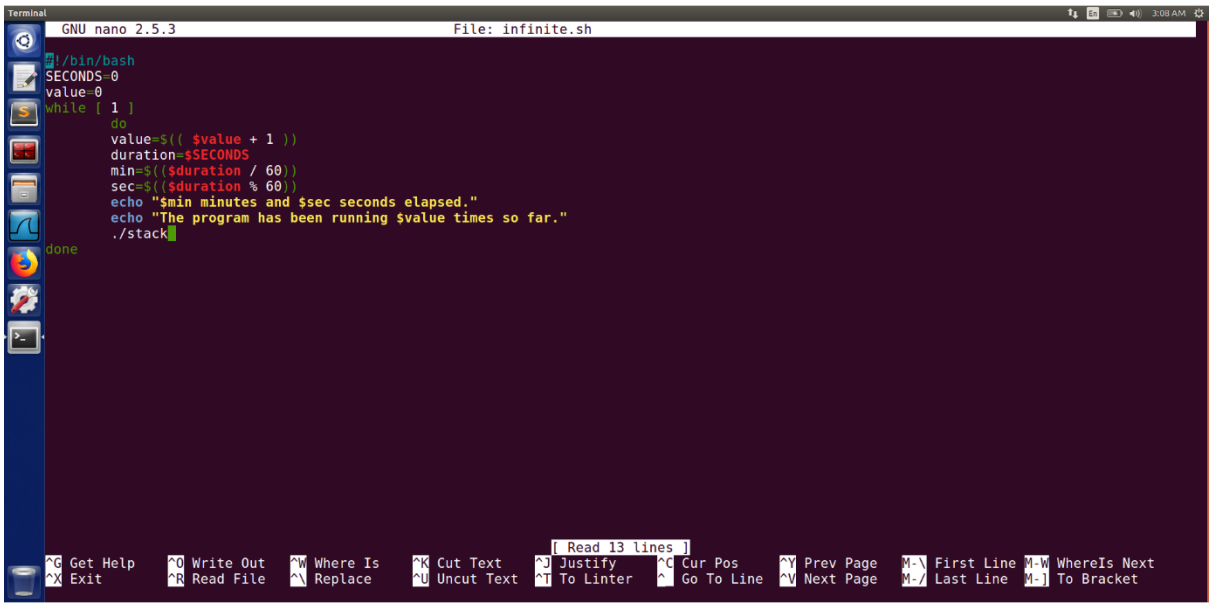
The results show that we were able to get access to the root's terminal and on checking for the id, we see that the user id (uid) is that of the root. Hence, the attack was successfully performed and we were able to overcome the dash countermeasure by using setuid() system call.

Task 5: Defeating Address Randomization

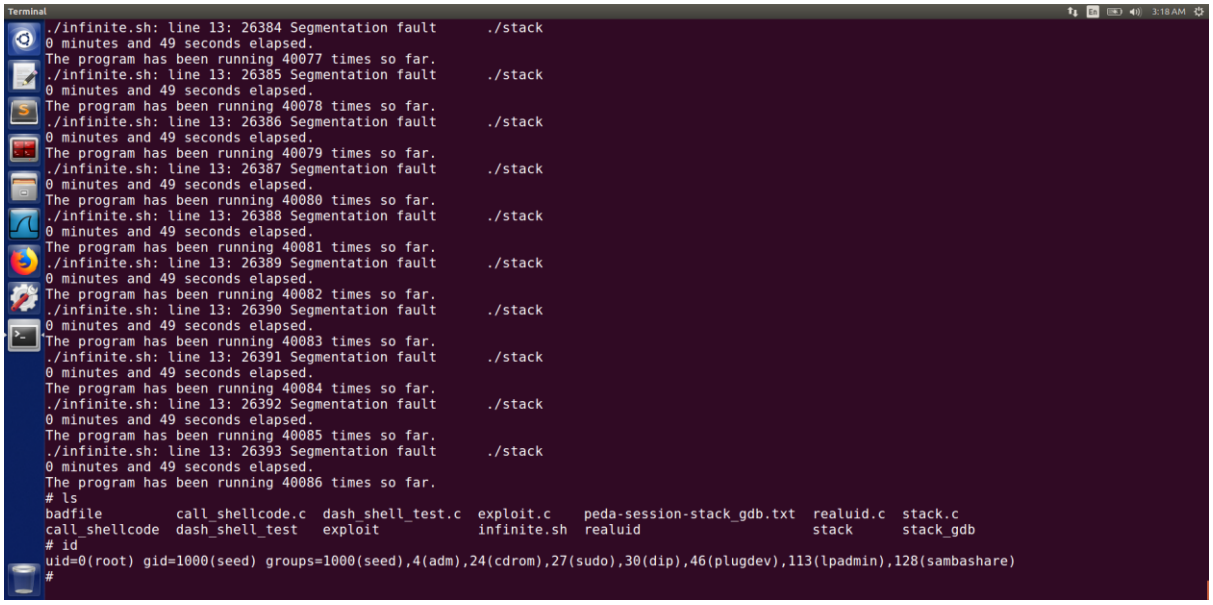
We enable address randomization for both stack and heap by setting the value to 2. On running the stack set-UID program from task 2, we get a segmentation fault as shown below:



The below shell script infinite.sh uses brute force by running the vulnerable program in a loop. This is basically done to hit the same address as the one we put in the badfile. The output shows the time taken and the number of times it has run the stack program.



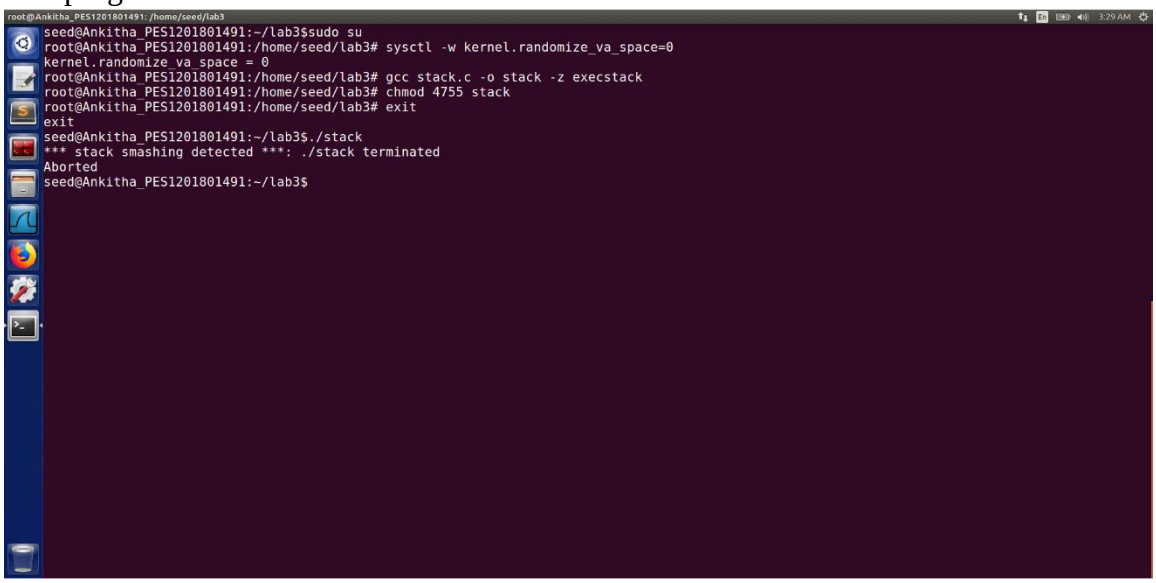
The output shows the time taken and the attempts taken to perform this attack with Address Randomization and Brute-Force Approach. It leads to a successful buffer overflow attack:



When Address Space Layout Randomization countermeasure is on, then the stack frame's starting point is always randomized and different. So, we can't correctly find the starting point or the offset to perform the overflow. The only option left is to try as many numbers of time as possible, unless we hit the address that we specify in our vulnerable code. On running the brute force program, the program ran until it hit the address that allowed the shell program to run. As seen, we get the root terminal (as it is a SET-UID root program), indicated by #.

Task 6: Turn on the StackGuard Protection

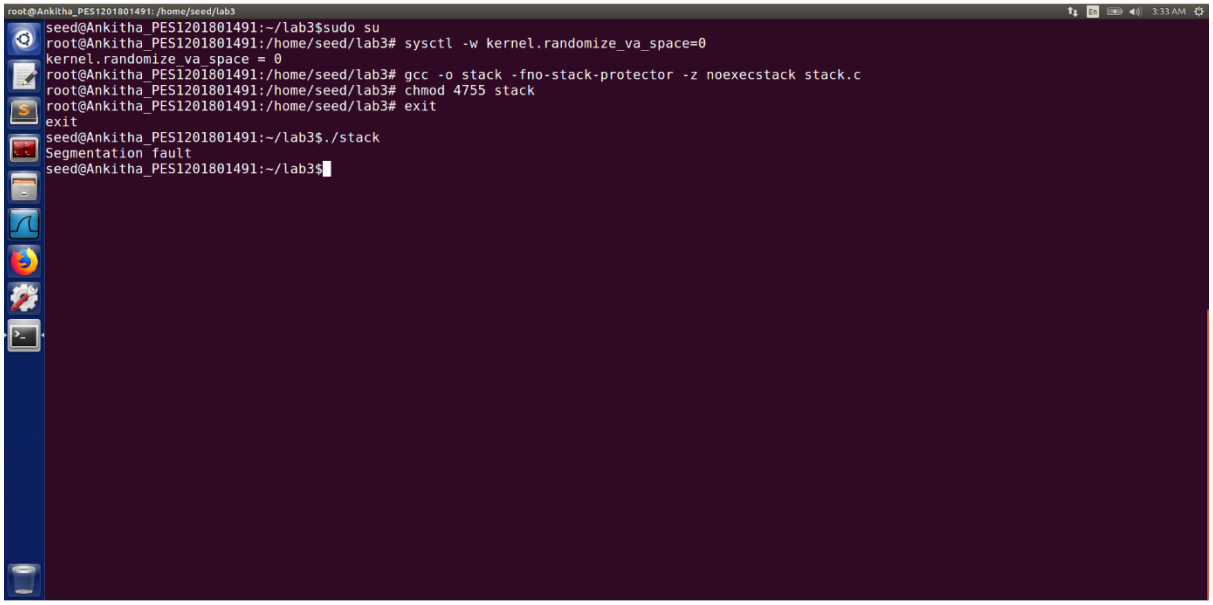
First, we disable the address randomization countermeasure. Then we compile the program 'stack.c' with StackGuard Protection (by not providing -fno-stack-protector) and executable stack (by providing -z execstack). Then we convert this compiled program into a SET-UID root program.



On running the vulnerable stack program we see that the buffer overflow attempt fails because of the above error, and the process is aborted. This proves that with StackGuard Protection mechanism, Buffer Overflow attack can be detected and prevented.

Task 7: Turn on the Non-executable Stack Protection

First, we disable the address randomization countermeasure. We then compile the program with StackGuard Protection off (due to -fno-stack-protector) and nonexecutable stack (by adding -z noexecstack). Then we make this program a SET-UID root program. On running this compiled program, we get the error of segmentation fault. This shows that the buffer overflow attack did not succeed, and the program crashed.



In the case of a non executable stack, a program would function normally but we are not able to run the malicious code which will be considered as read-only data rather than executable code. Therefore, the attack failed because of the stack being non executable.