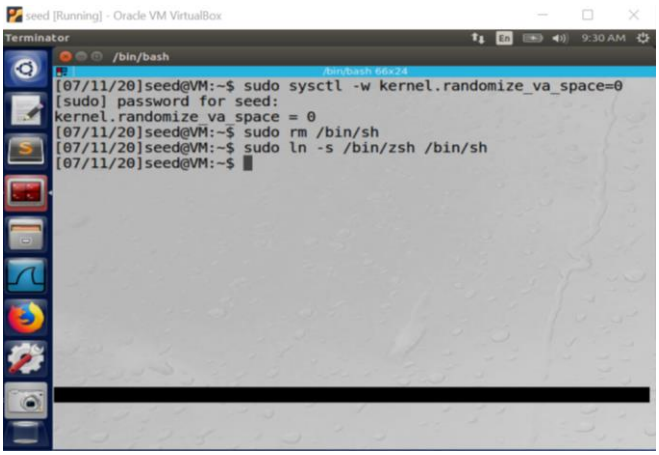
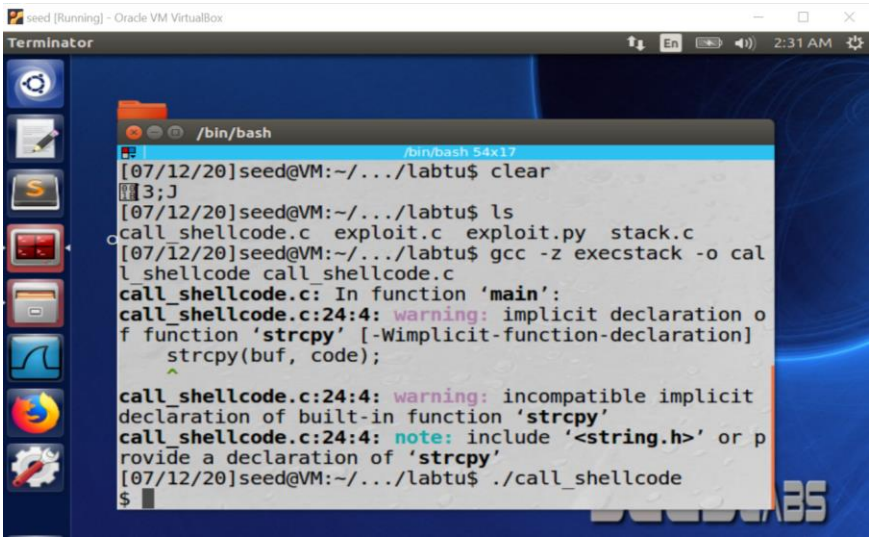


First we disable the countermeasures by setting its settings to 0 in the sysctl file.

We changed the default shell from dash to zsh to avoid the countermeasures in bash.

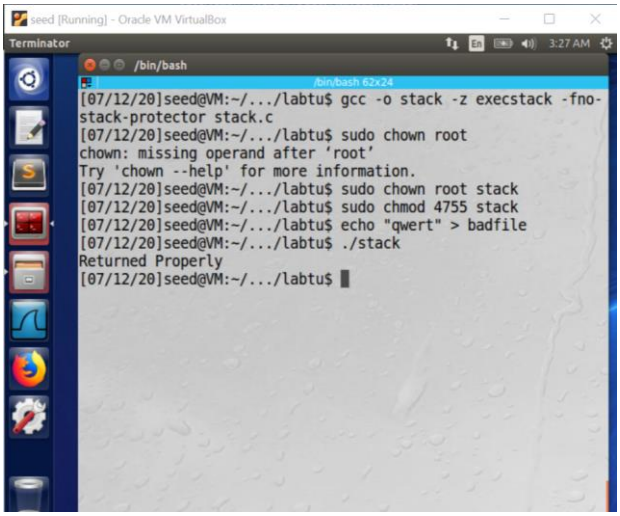
Then we compile the program with –z ececstack parameter. By default the gcc compiler has does not allow code in stack to be executable, this is a countermeasure set for the buffer overflow attack. This parameter makes the stack executable, allowing our vulnerable code to be executed.

Then we run the executable 'call_shellcode'. The process is shown in the following screenshot.



As we can see there were no errors and we entered the shell of our user account ($) this means the code ran successfully and we got access to /bin/sh.

Now we compile the provided vulnerable program "stack.c". We should disable the StackGuard protection system by passing '–fno-stack-protector' parameter and also make the stack executable like before. After compiling we make the executable a set-uid root program. The process is shown in the following screenshot.

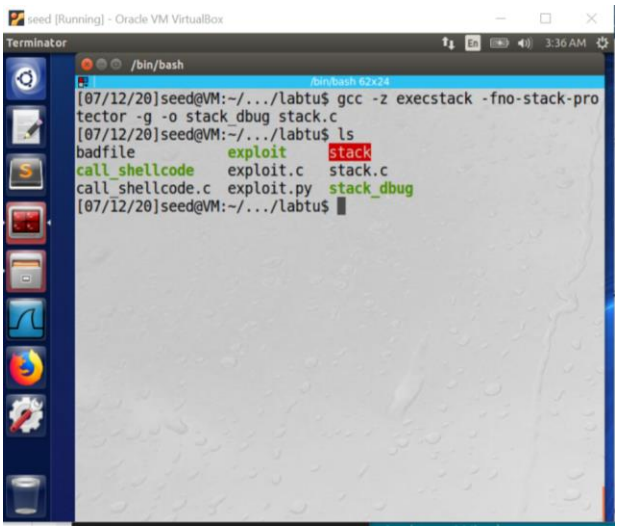


As we can see after compiling we run our executable file 'stack' which took its input from badfile. And as you can see we can run the program without any segmentation fault.

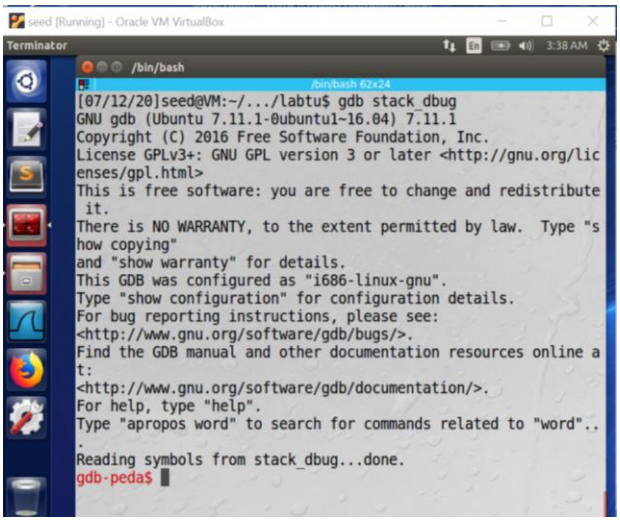
Task 2:

In this task we will exploit the vulnerable set-uid program to gain access.

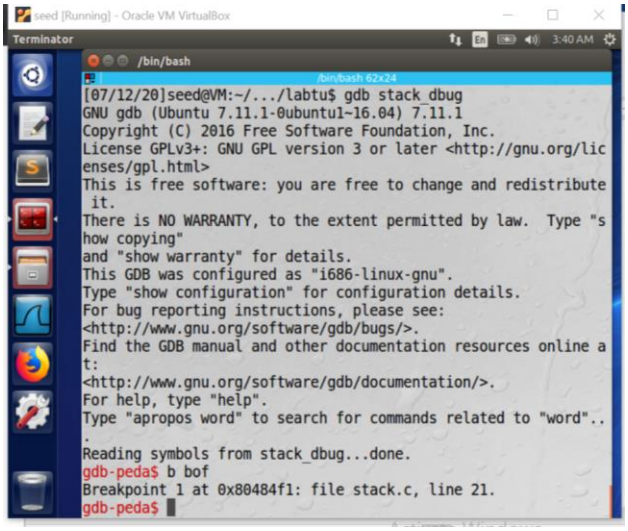
We compile the program in debug mode using the '-g' parameter. (make the stack executable and disable the StackGuard protection).



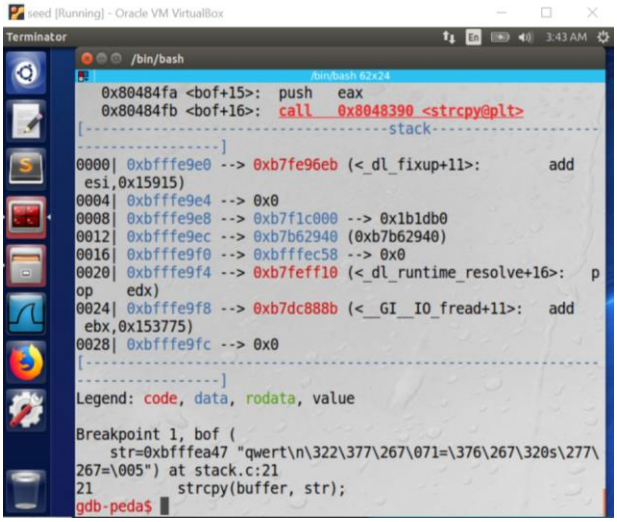
Now we run the stack_dbug program in debugging mode using 'gdb'.



Now we set the breakpoint. Using the b bof.

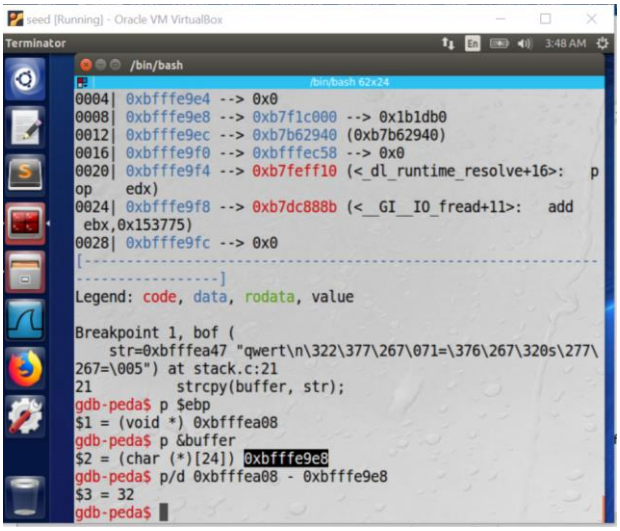


Now we execute the program using ‘run’.

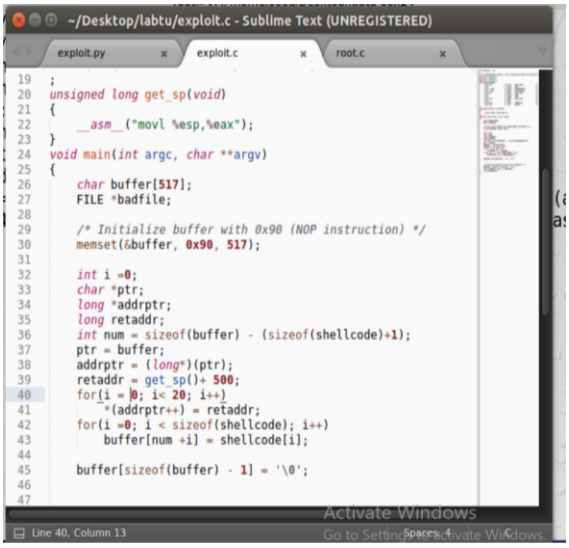


The program stopped at the breakpoint that we had created (in bof function).

Now we print the values of ebp and buffer. And we find the return address value's address by finding the difference between the ebp and the start of the buffer. The process is shown in the following screenshot

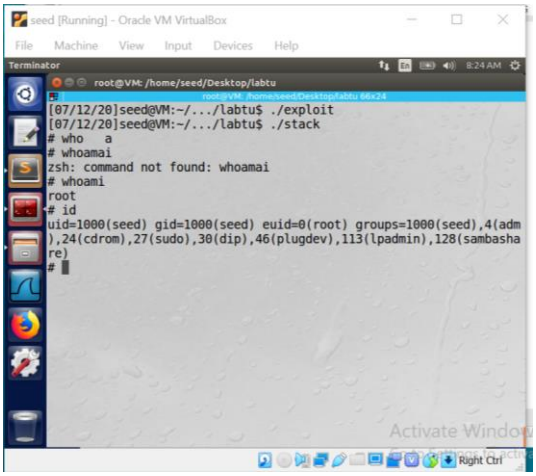


Now we modify the exploit file. The exploit code writes the buffer and overflows it wit NOP, which then allows for the execution of the next line of the command. The exploit.c d code has this addition toallow for the overflow of the stack and also to the point the code to execute the malicious code.



After executing the executable exploit and then executing the stack we enter to shell as root.

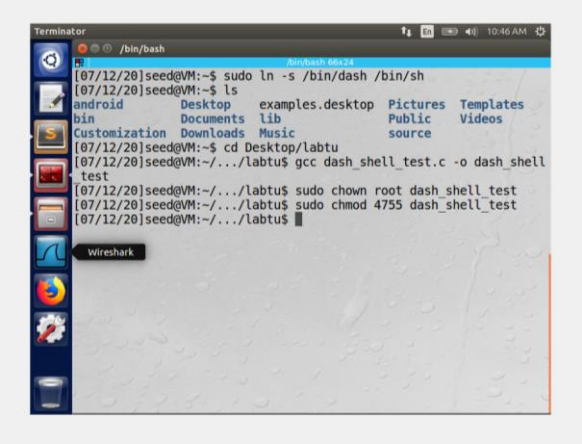
As can been seen the euid is 0, and upon entering the ‘whoami’ command the result is root.



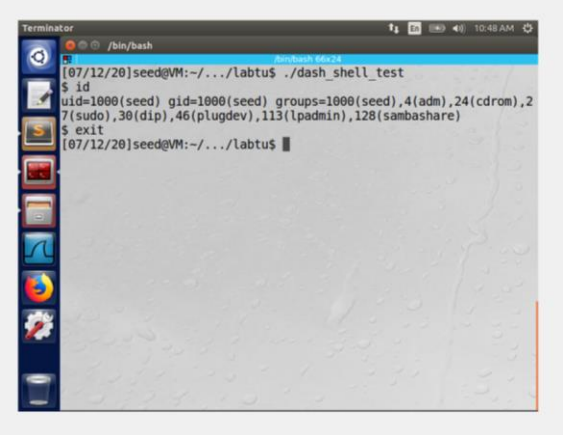
<u>Task 3: defeating dash’s countermeasures:</u>

First we change change the /bin/sh symbolic link to point it back to /bin/dash again.

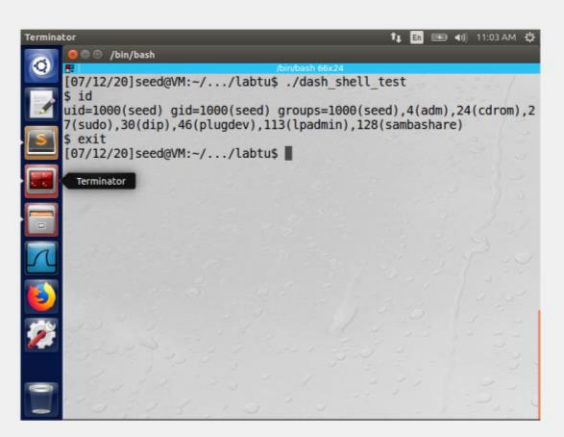
Next we compile the dash_shell_test.c file and then make the executable a setuid program.



Now we run the program.

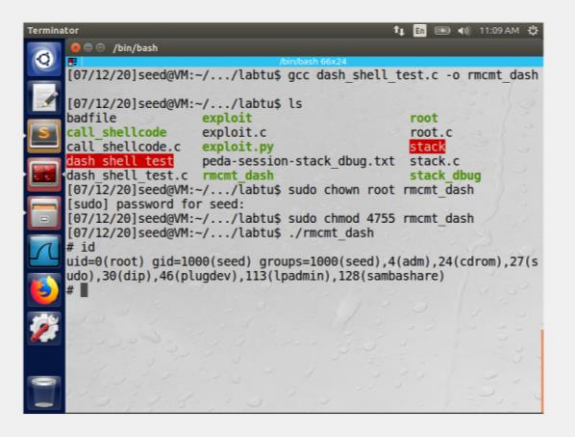


Now we run the program and see the uid.



As we can see the user id that we got is that of user seed.

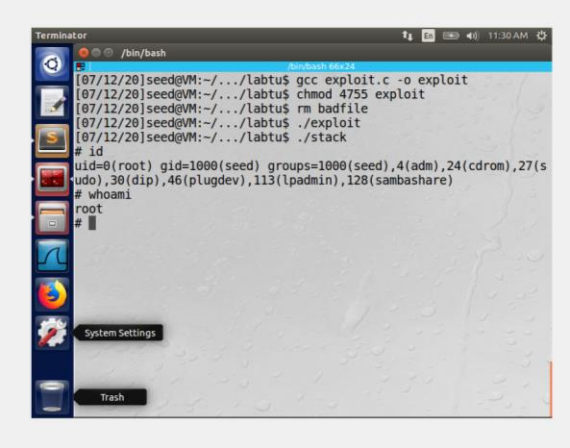
Now we uncomment the setuid(0) and compile the file to an executable to rmcmt_dash. And then run the executable, we then check the id. As is seen in the screenshot below the uid is 0(root).



Observation: as we saw in both cases we got to the shell. But in the first example the shell is of a normal user and not of the root and not with root privileges. This is because the effective uid and actual uid of the program are different. But in the second program we set the setuid to 0, and the effective uid is also 0. Since the effective and actual uid are same, the dash gives privileges and the root shell is access is given. Hence defeating the countermeasure of the dash.

The next part of this task is to do the buffer overflow attack when /bin/sh is linked to /bin/dash.

First we added the given assembly code to our exploit.c program we compiled it, made it a setuid program. And then ran it.

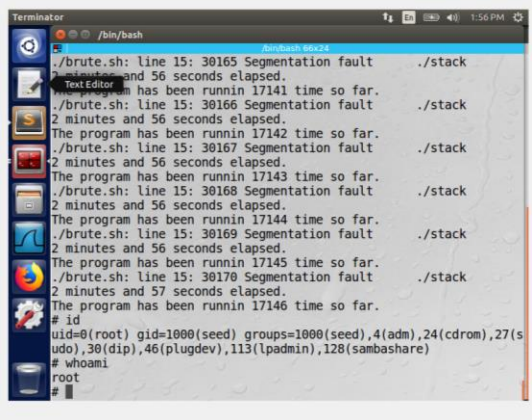


Upon running the exploit program then running the stack program we were able to get access to root shell as is shown in the above screenshot.

Task 4: Defeating the address Randomization

First we turn ubuntu's address randomization using the sudo /sbin/sysctl –w kernel.randomize_va_space=2.

Then we saved the shell script into a brute.sh file. Next we made it a setuid program.

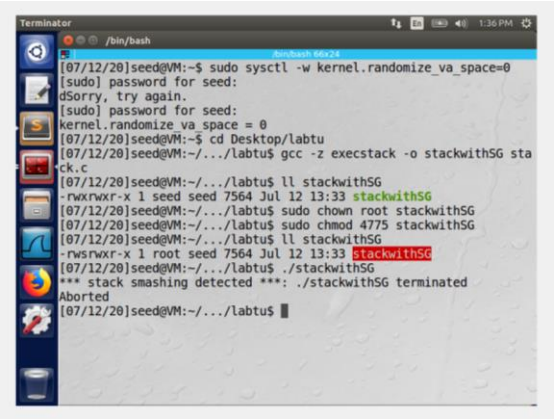


After leaving it a while it should give open root shell.

The idea is that if since in 32-bit linux the stack base address is not very large (524288) it can be brute forced. If running the vulnerable program (stack) once does not get you to the root shell, we keep running it again in an infinite loop, and we should be able to get into the root shell, that is if the stack program is written well. And by running the given shell script we will eventually get access to root shell.

Task 5: Turn on the StackGuar Protection

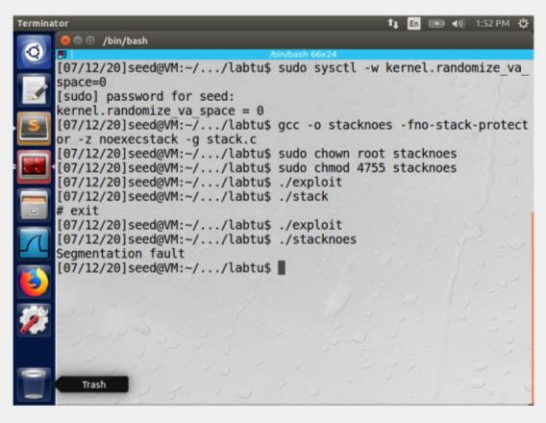
In this task we first disabled the address randomization countermeasure. Next we compile the stack.c file int stackwithSG executable. After that we make it a set-uid program. Then we ran the program. These steps are shown in the screenshot below



As is seen the vulnerable program was terminated and we weren't able to do the buffer overflow attack. Hence proving that the StackGuard Protection Mechanism is an effective mechanism for detecting and preventing the buffer overflow attack.

Task 6: Non-execuatable Stack

In this task we first turn off the address randomization then compile the stack program to an executable stacknoes with –noexecstack parameters (making the stack non executable). We then run the exploit file and then we first rant the previous stack file to show that we get root shell access. Then we run our new program stacnoes (with non-executable stack).



As we can see in the screenshot we get segmentation fault. This is because of the non-executable stack protection mechanism which avoids code from stack to be executed.