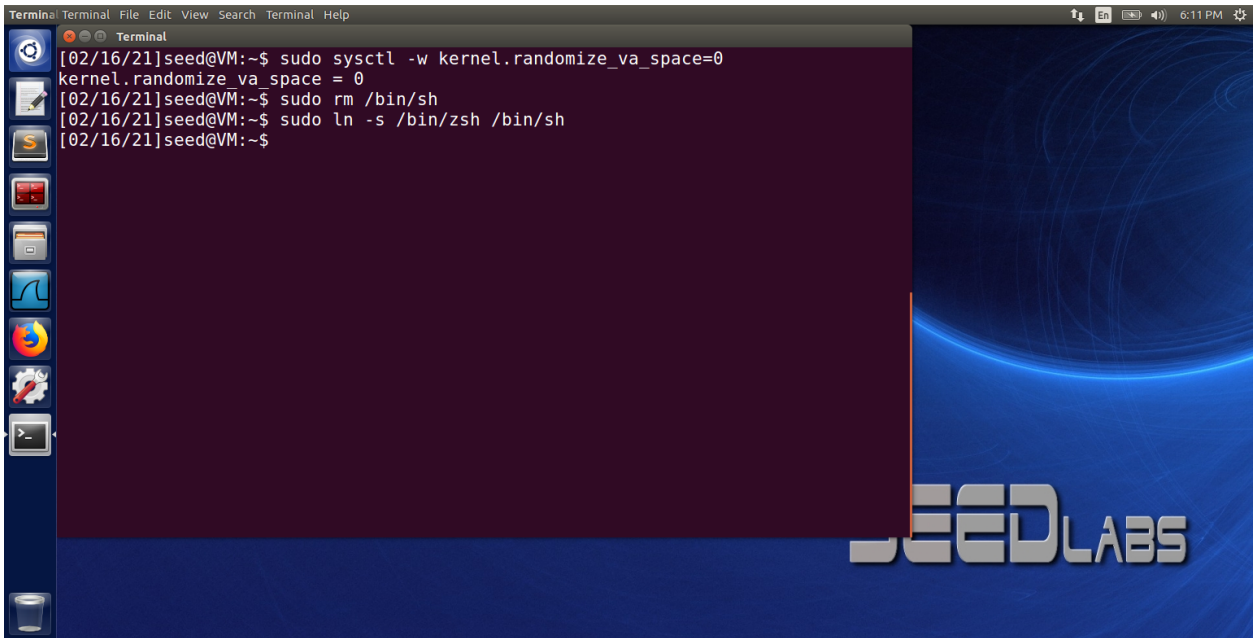
Lab 3: Buffer Overflow

Before we can start to or commence the Buffer Overflow attacks, we will need to disable some key settings, like the Address Space Layout Randomization feature. This is usually enabled to make it hard for the prediction of the stack's position in the memory. Thus, we will disable it (for now) to make it easier to understand these kinds of attacks by setting the value of the setting in the sysctl file to 0 (indicating that it is false).

To make sure that we avoid the countermeasure implemented in *bash* for any SETUID programs, we will change the default shell from *dash* to *zsh*. The compiler does come in with counteractive measures to mitigate the buffer overflow attack.



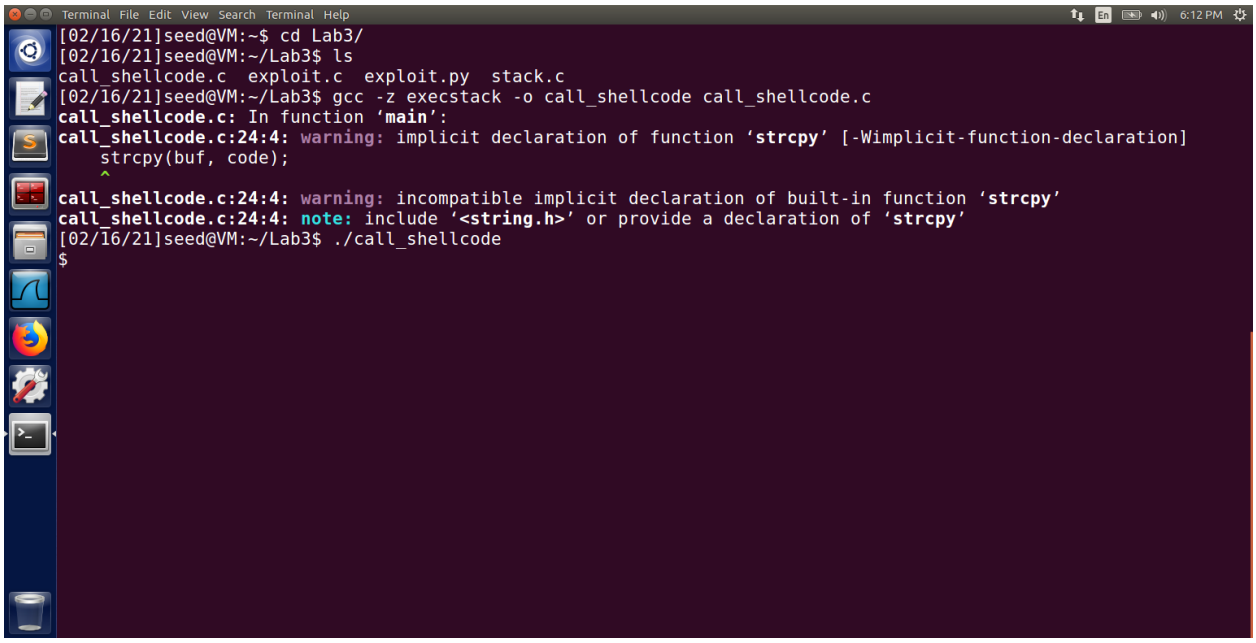
Task 1: Running Shellcode

Thus, to try to be able to exhibit this kind of attack, we will disable further settings during the program's compilation stage. The settings are as follows:

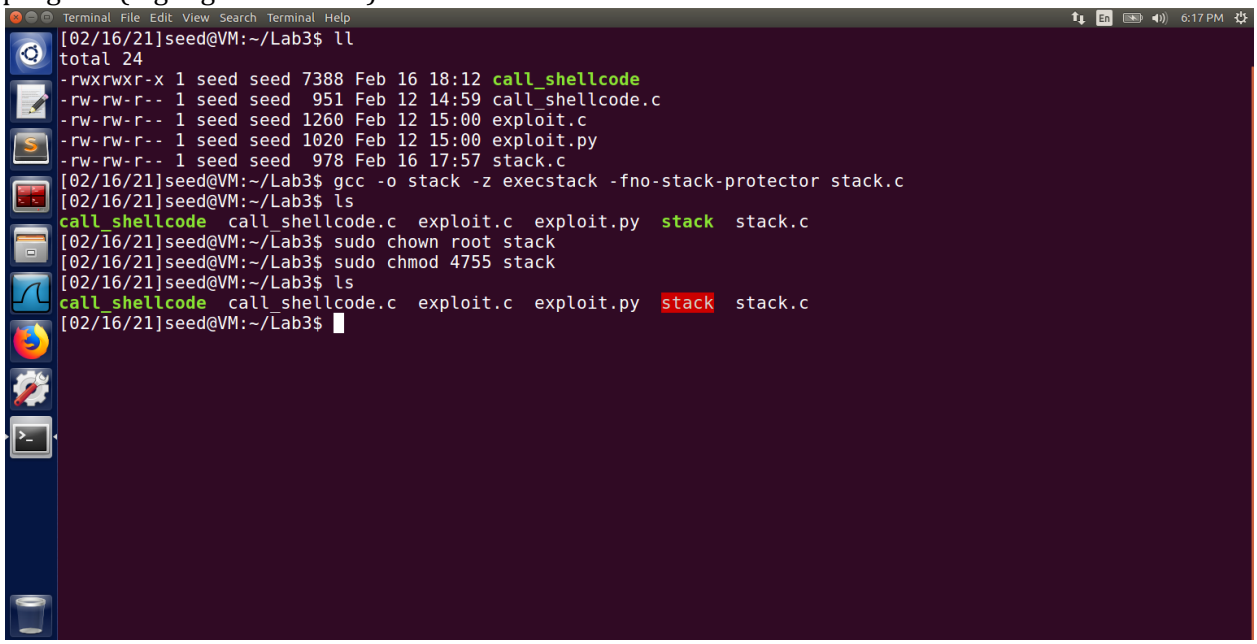
- Non-Executable stack: By default, the stack cannot be an executed or be an executable. The operating system usually identifies whether the stack is an executable or not by a binary bit that is set within the system, which can be manipulated by the compiler. Therefore, by providing *-z execstack,* the stack becomes executable and hence allows the code to be executed when present in stack.
- Stack-Guard Protection Scheme: The sole purpose of this scheme is to migitate the stack-based buffer overflow attacks, by some special mechanism in the code.

We first navigate to the lab folder and list out the contents just to make sure that we have the files present. We then compile the shellcode program (which is not a SETUID program) by passing the Non-Executable stack parameter to make the stack executable (this is done for the compiler to not throw the *segmentation fault* error message). The program is stored in an executable file

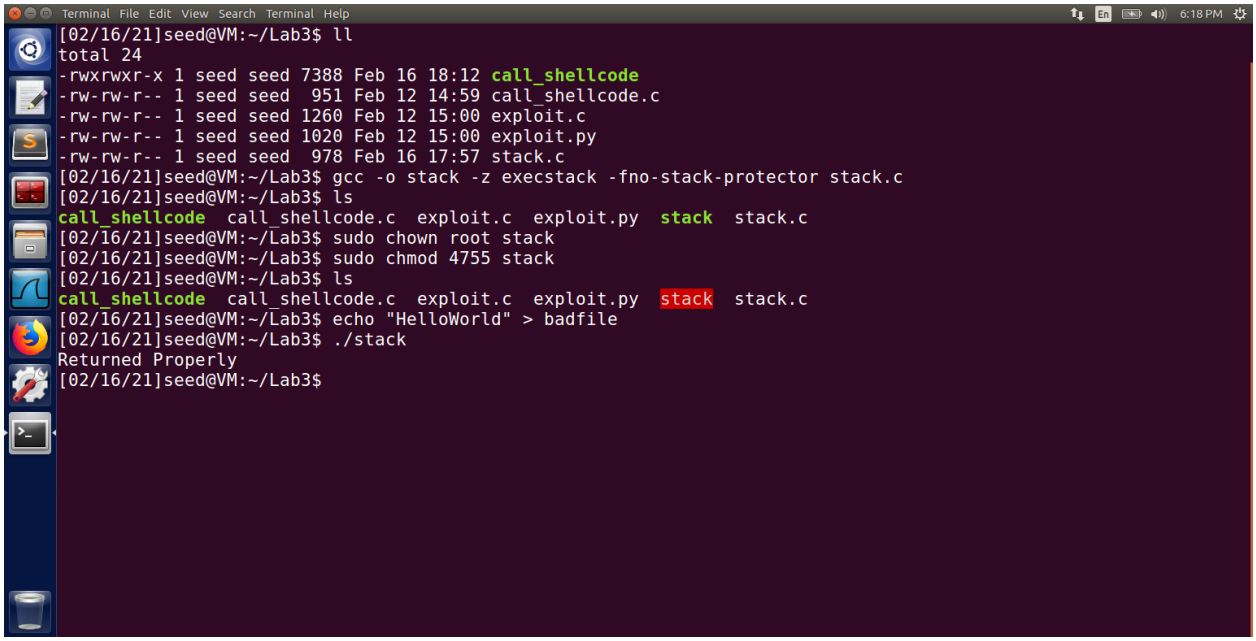
call_shellcode, which is then executed and then we enter the shell (indicated by the dollar ($) symbol), thus getting access to */bin/sh.*



we enter the shell We then compile the given stack program (*stack.c*) and during compilation, we will disable the StackGuard Protection Scheme. The compiled program is made into a SETUID root program (highlighted in **RED**).



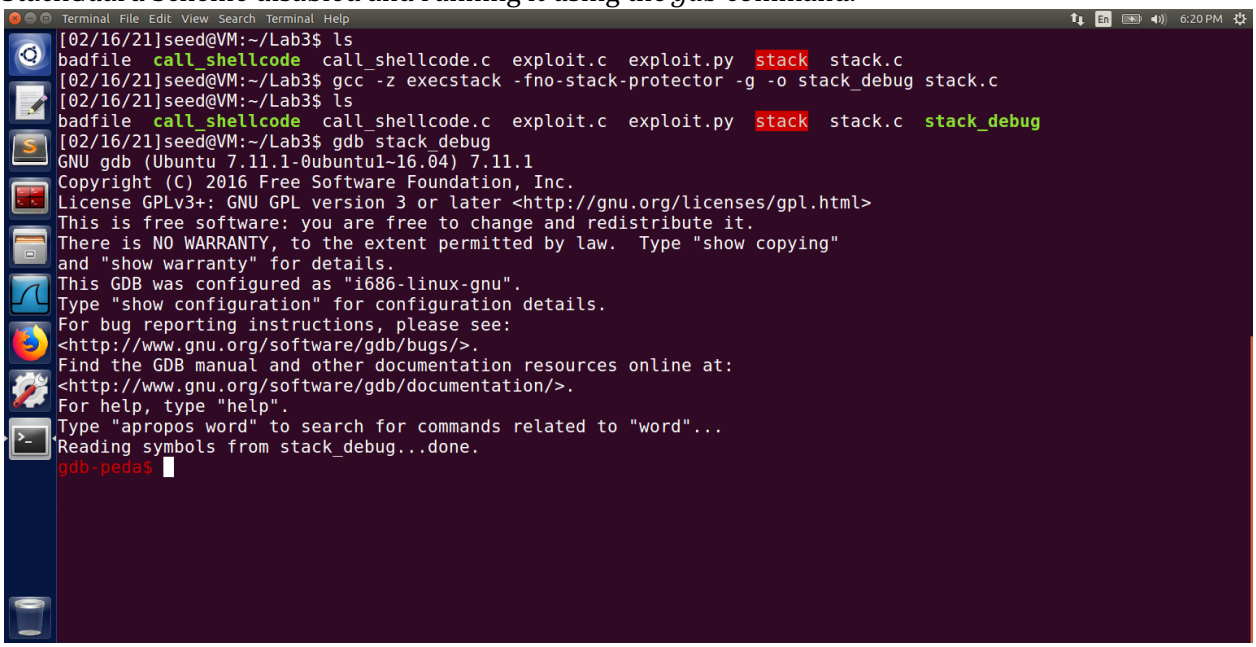
When the stack program is run normally, its functioning is shown below:



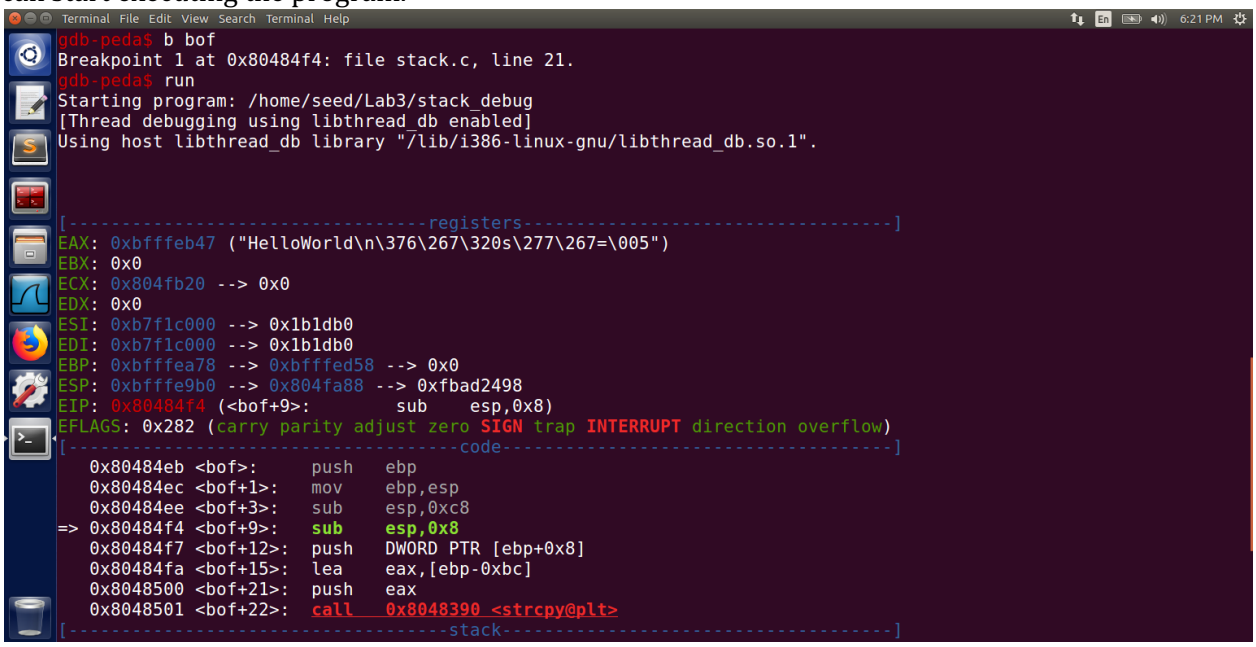
Task 2: Exploiting the Vulnerability

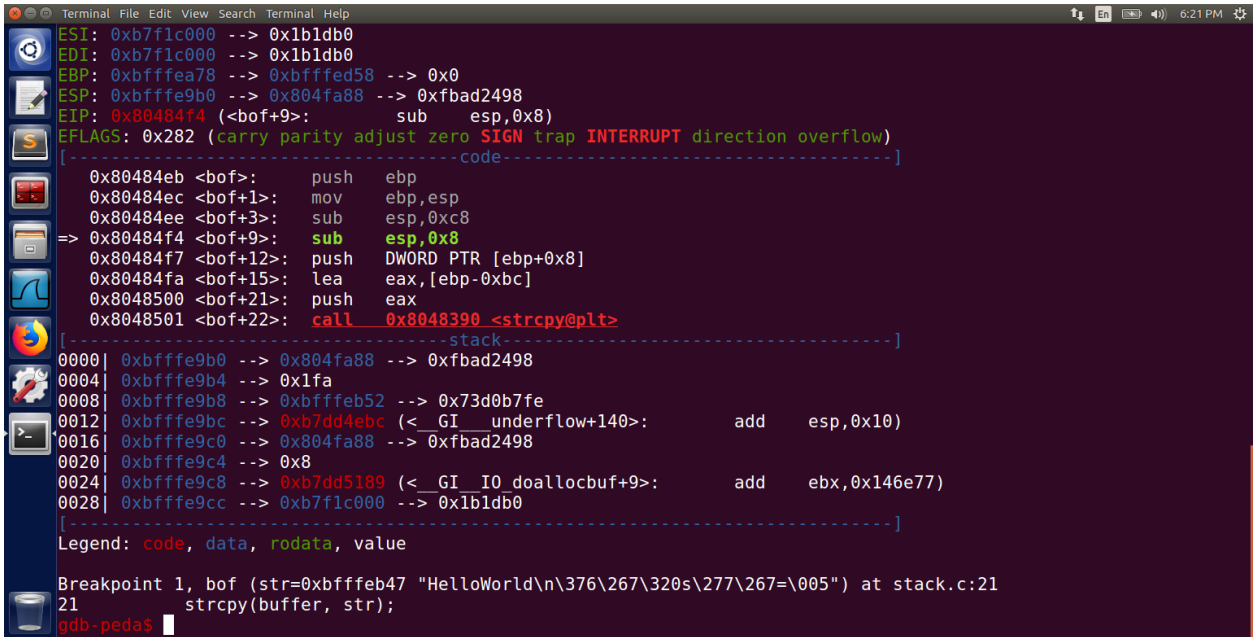
The process that we will run in this task will be stored in the same memory always in the stack as we have already disabled Address Space Layout Randomization. We will now compile the program in debug mode to find the address of the running program in the memory. We will be able to find the EBP and its offset by using the help of debugging so that we can use the right buffer payload to run our program.

The compile command contains the *-g* parameter and it indicates the debug mode, with the StackGuard Scheme disabled and running it using the *gdb* command.

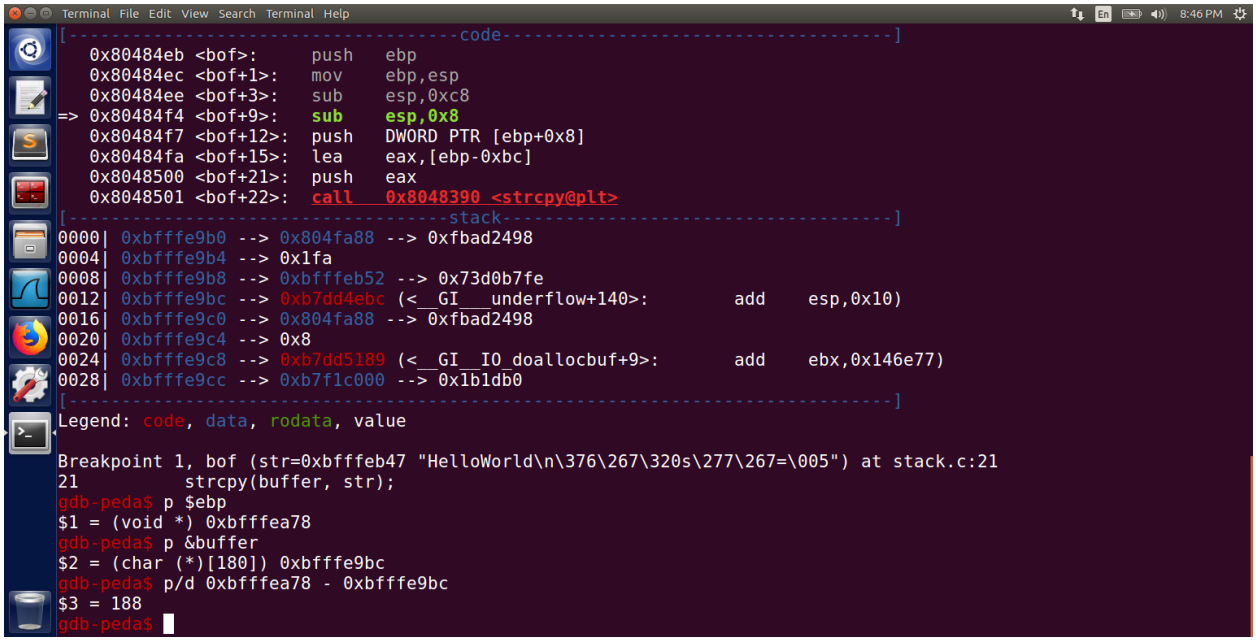


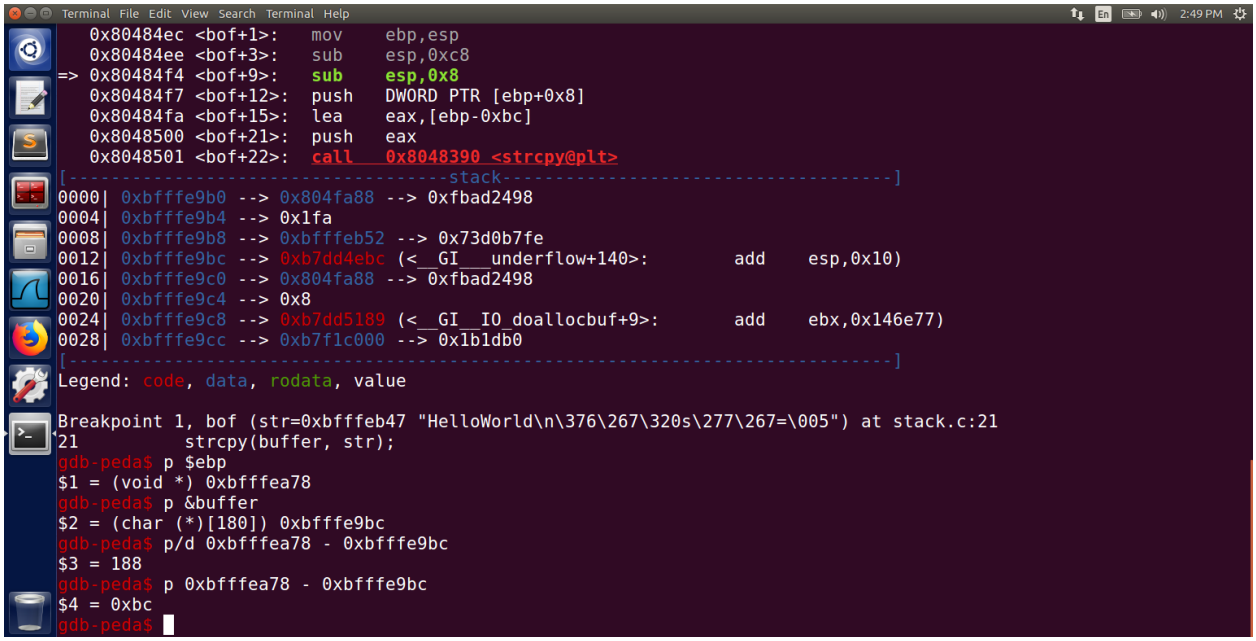
Inside of GDB, we using the *b bof* function to set a breakpoint on the *bof* function, and only then we can start executing the program.





The program stops inside the *bof* function due to the breakpoint that is created. The only content that will matter to us is the stack frame values (ebp) as that is what is used to construct the badfile contents in the later phases. In the following screenshot, we print the values of the EBP and buffer and the difference between the EPB and start of buffer, thus getting the return address value.

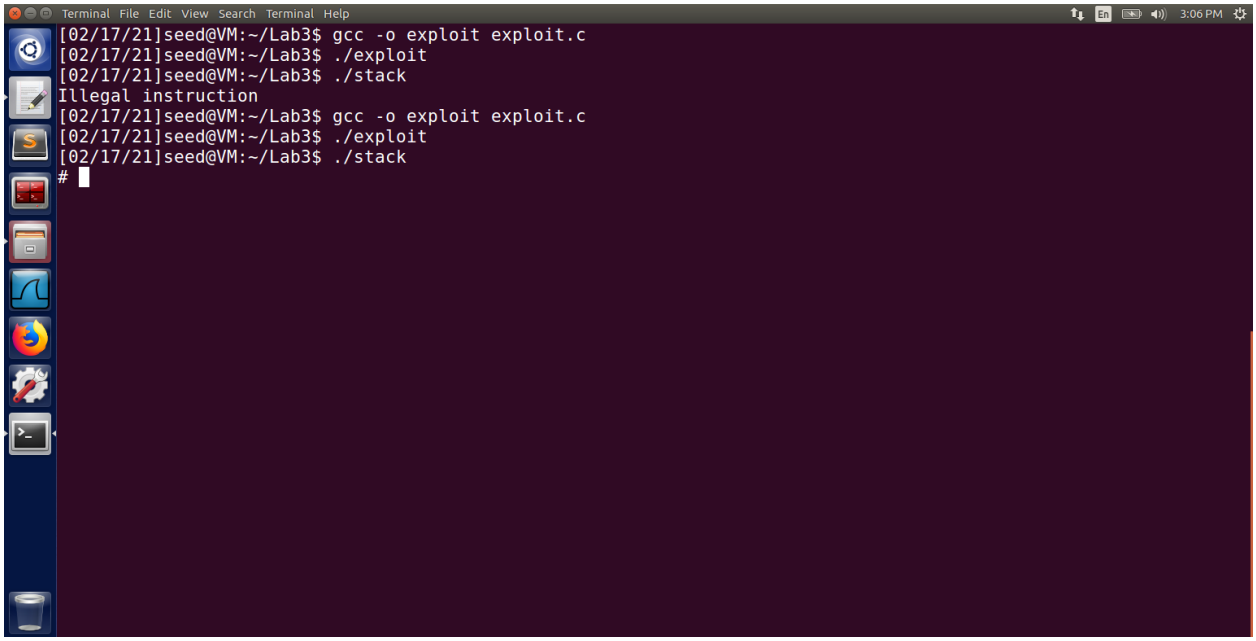


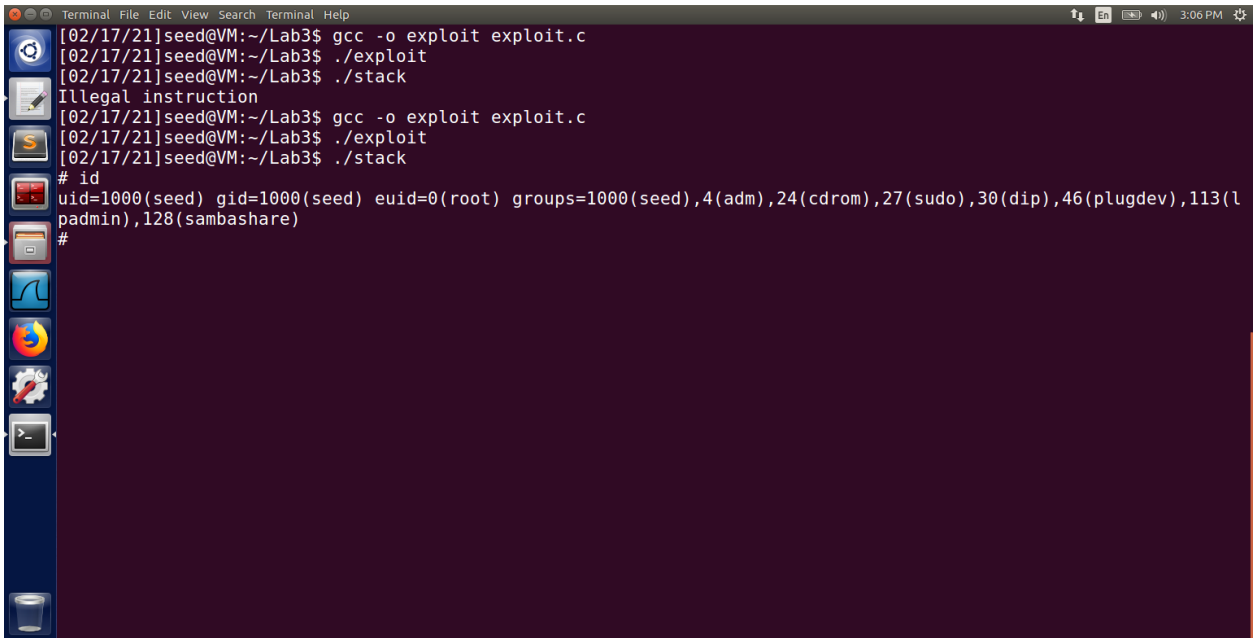


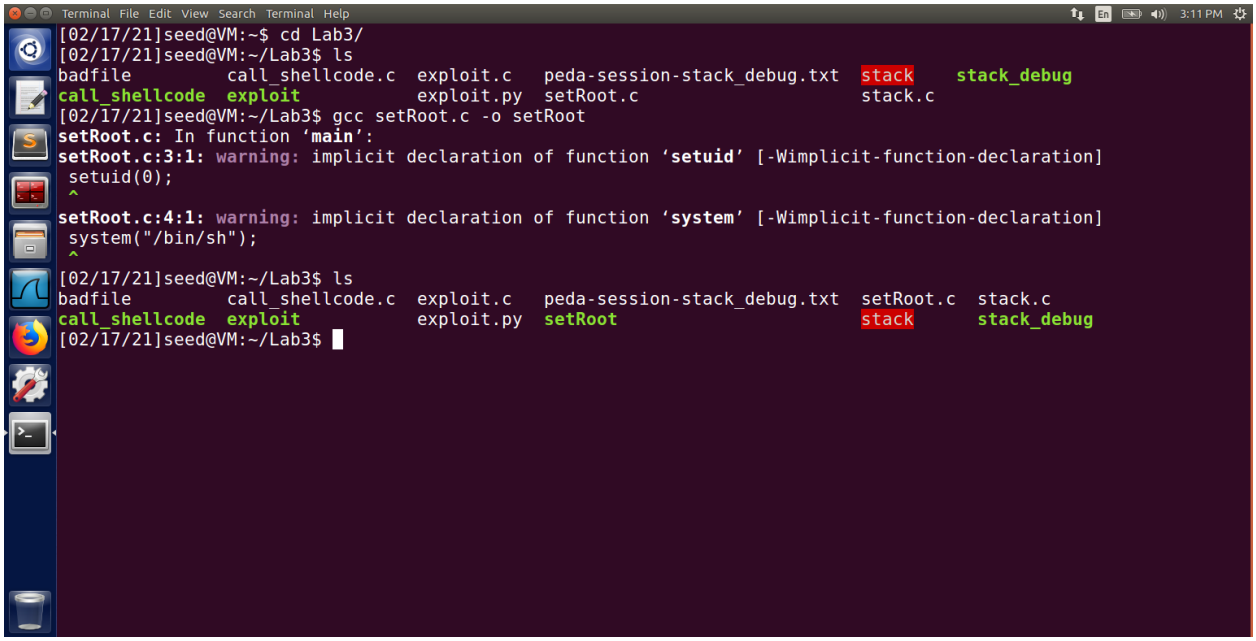
In the above screenshot, we can observe that the frame point is *0xbfffea78*. Therefore, the return address must be stored at *0xbfffea78 + 4*. The first address that we will be able to go to would be *0xbfffea78 + 8*. But in order to get to our code where the return address points to it, we will need to know the location to store the return address inside the input, as it is stored in the return address field inside the stack. Since we already know that return address will be 4 bytes above the EBP, the distance between the return address and the start of the buffer is going to be 188 + 4 = 192. The return address will be stored in the *badfile* at an offset value of 192.

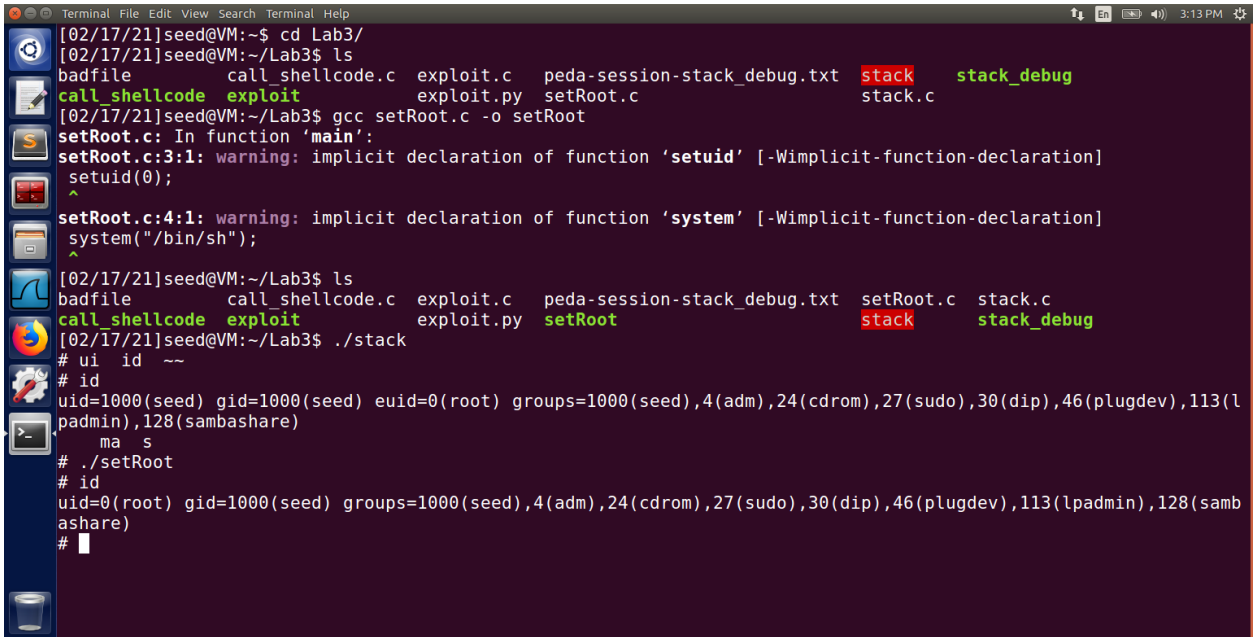
In the next steps, we will modify both the C and python programs of *exploit*. One thing that we may fail to realize or miss is that, the stack might actually be deeper than we think, as it was used in debug mode. Therefore, we will give a much larger value to the EPB, by adding around 80 to it. Next

we compile the program and run the *exploit.c* program to generate the *badfile*. Then we run the *stack* program (the vulnerable one!) which will use this *badfile* as the input and will start to copy the content of the file into the stack, thus resulting in buffer overflow. The # indicates that we have gotten access to the root privileges by entering into the root shell. Next, we will run the program to transform our *seed* user to *root* as well.

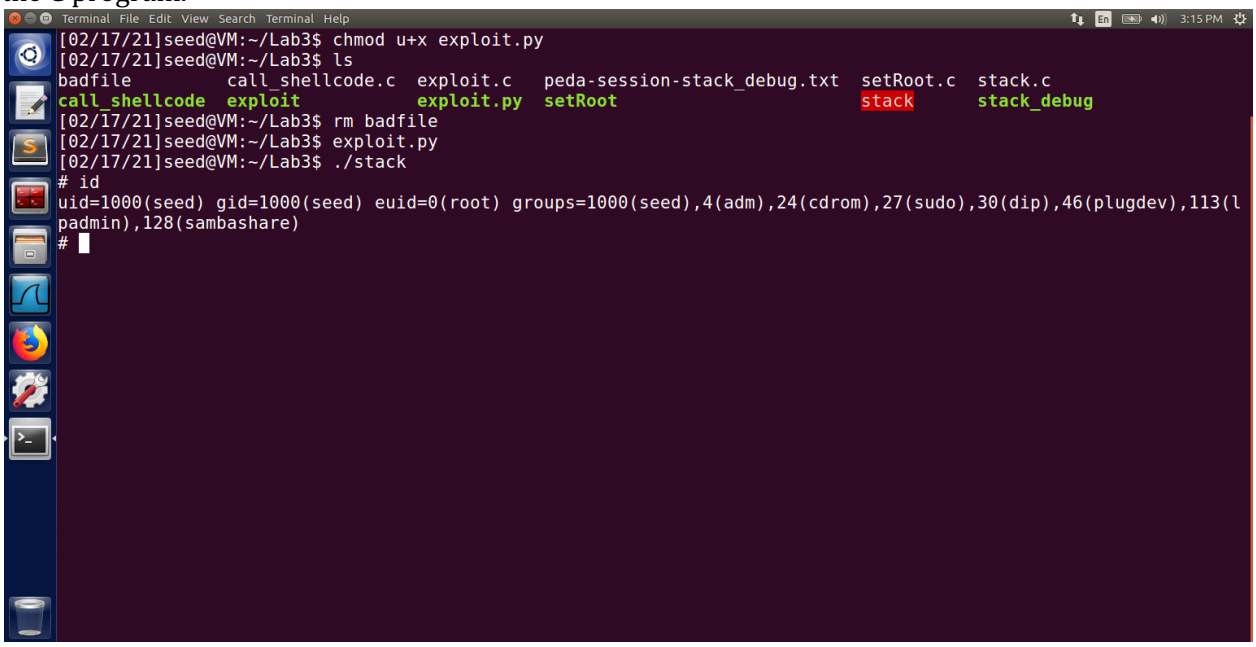





Terminal File Edit View Search Terminal Help
3:11 PM
[02/17/21]seed@VM:~$ cd Lab3/
[02/17/21]seed@VM:~/Lab3$ ls
badfile call_shellcode.c exploit.c peda-session-stack_debug.txt stack stack_debug
call_shellcode exploit exploit.py setRoot.c stack.c
[02/17/21]seed@VM:~/Lab3$ gcc setRoot.c -o setRoot
setRoot.c: In function 'main':
setRoot.c:3:1: warning: implicit declaration of function 'setuid' [-Wimplicit-function-declaration]
setuid(0);
^
setRoot.c:4:1: warning: implicit declaration of function 'system' [-Wimplicit-function-declaration]
system("/bin/sh");
^
[02/17/21]seed@VM:~/Lab3$ ls
badfile call_shellcode.c exploit.c peda-session-stack_debug.txt setRoot.c stack.c
call_shellcode exploit exploit.py setRoot stack stack_debug
[02/17/21]seed@VM:~/Lab3$

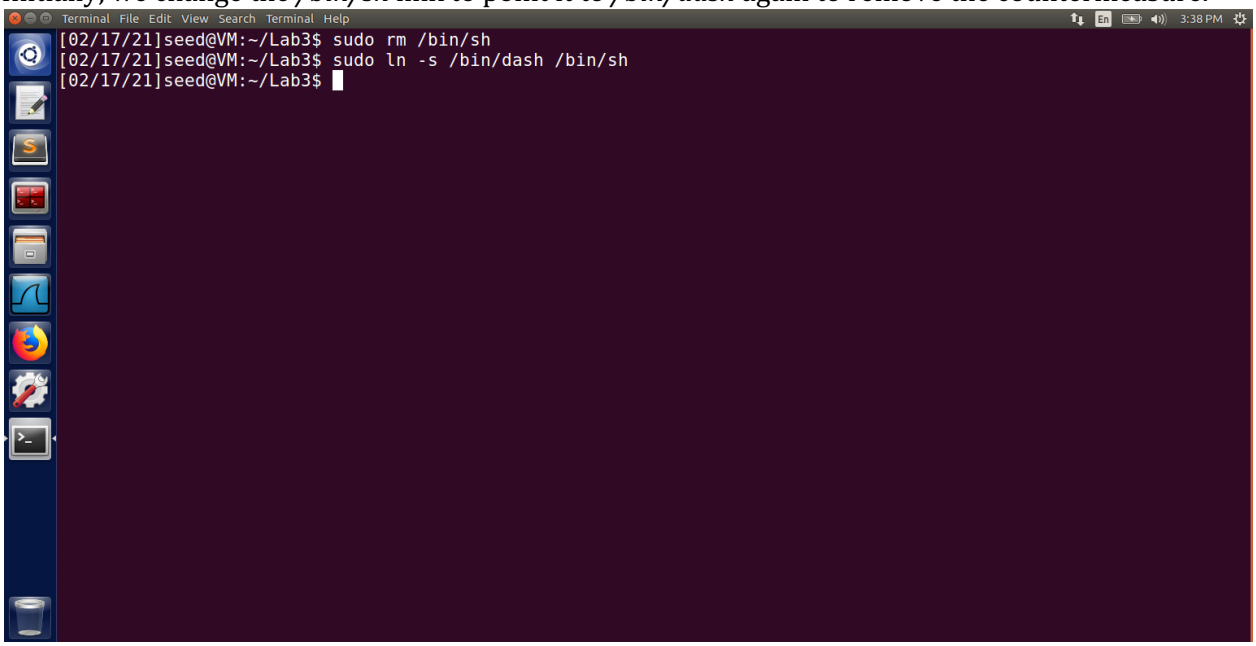

Terminal File Edit View Search Terminal Help
3:13 PM
[02/17/21]seed@VM:~$ cd Lab3/
[02/17/21]seed@VM:~/Lab3$ ls
badfile call_shellcode.c exploit.c peda-session-stack_debug.txt stack stack_debug
call_shellcode exploit exploit.py setRoot.c stack.c
[02/17/21]seed@VM:~/Lab3$ gcc setRoot.c -o setRoot
setRoot.c: In function 'main':
setRoot.c:3:1: warning: implicit declaration of function 'setuid' [-Wimplicit-function-declaration]
setuid(0);
^
setRoot.c:4:1: warning: implicit declaration of function 'system' [-Wimplicit-function-declaration]
system("/bin/sh");
^
[02/17/21]seed@VM:~/Lab3$ ls
badfile call_shellcode.c exploit.c peda-session-stack_debug.txt setRoot.c stack.c
call_shellcode exploit exploit.py setRoot stack stack_debug
[02/17/21]seed@VM:~/Lab3$ ./stack
# ui id ~~
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
ma s
# ./setRoot
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#

Next, we do the same with the python version and we are able to observe the same conditions as in the C program.

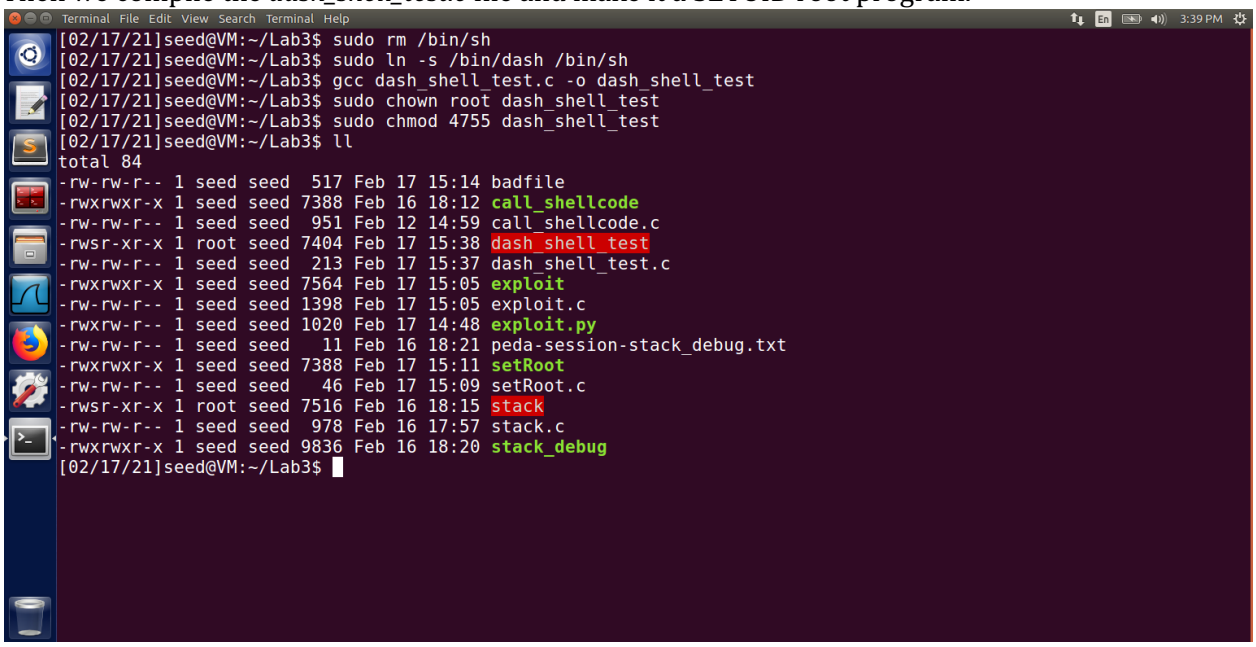


Task 3: Defeating Dash's Countermeasure

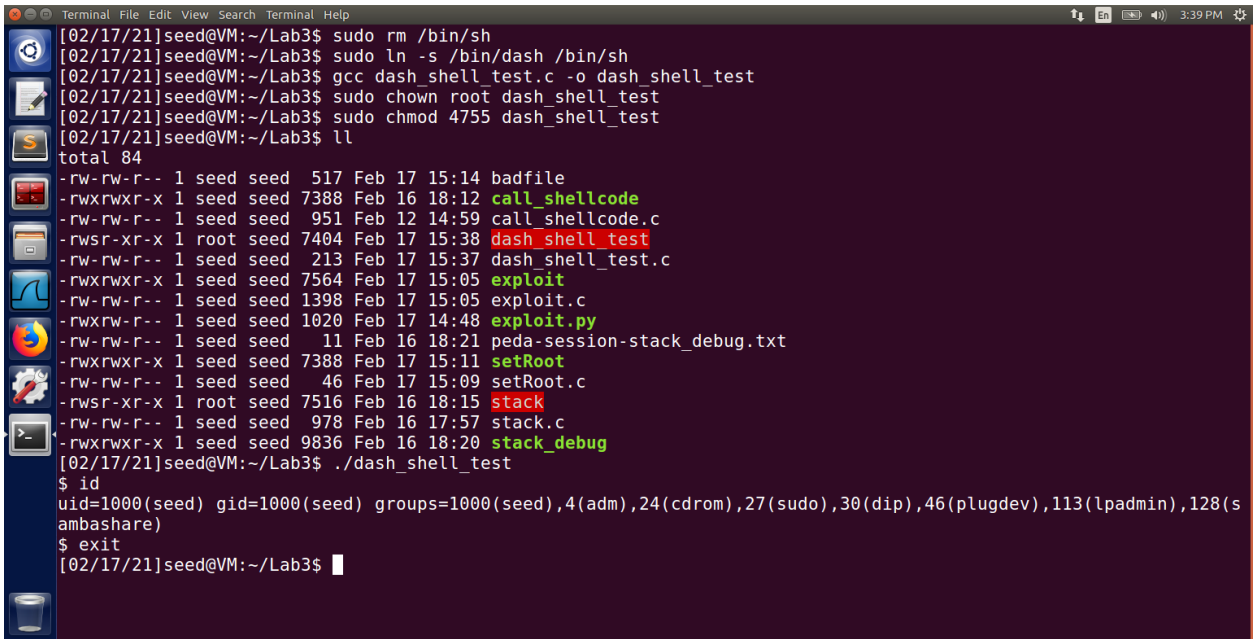
Initially, we change the */bin/sh* link to point it to */bin/dash* again to remove the countermeasure.

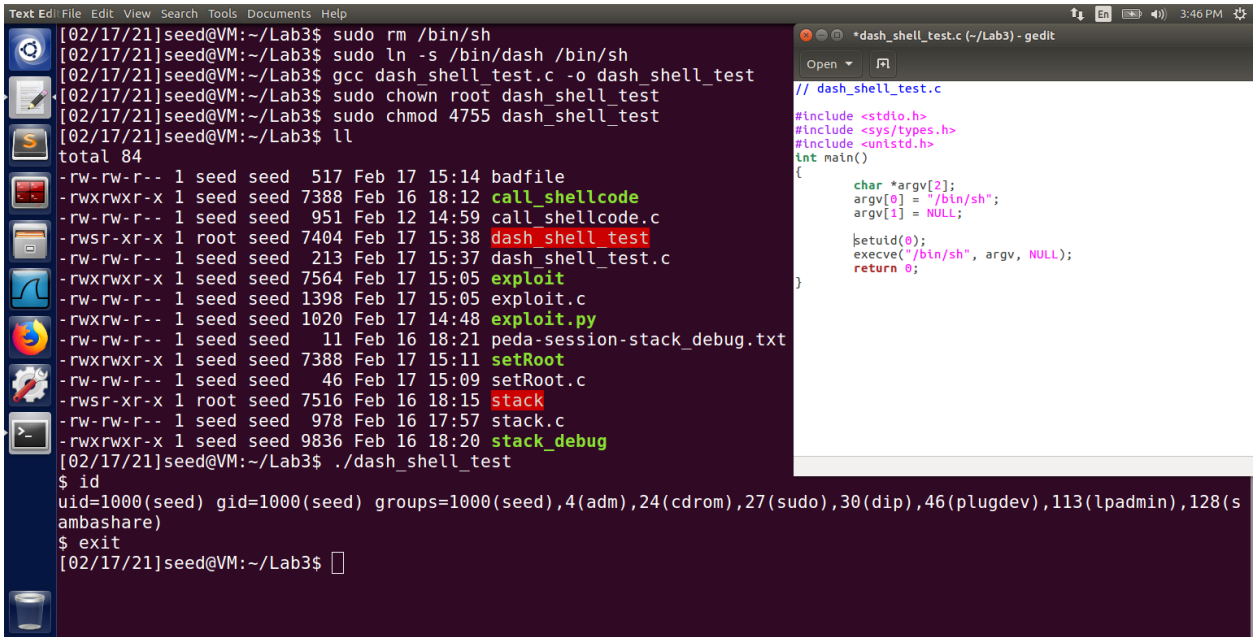


Then we compile the *dash_shell_test.c* file and make it a SETUID root program.

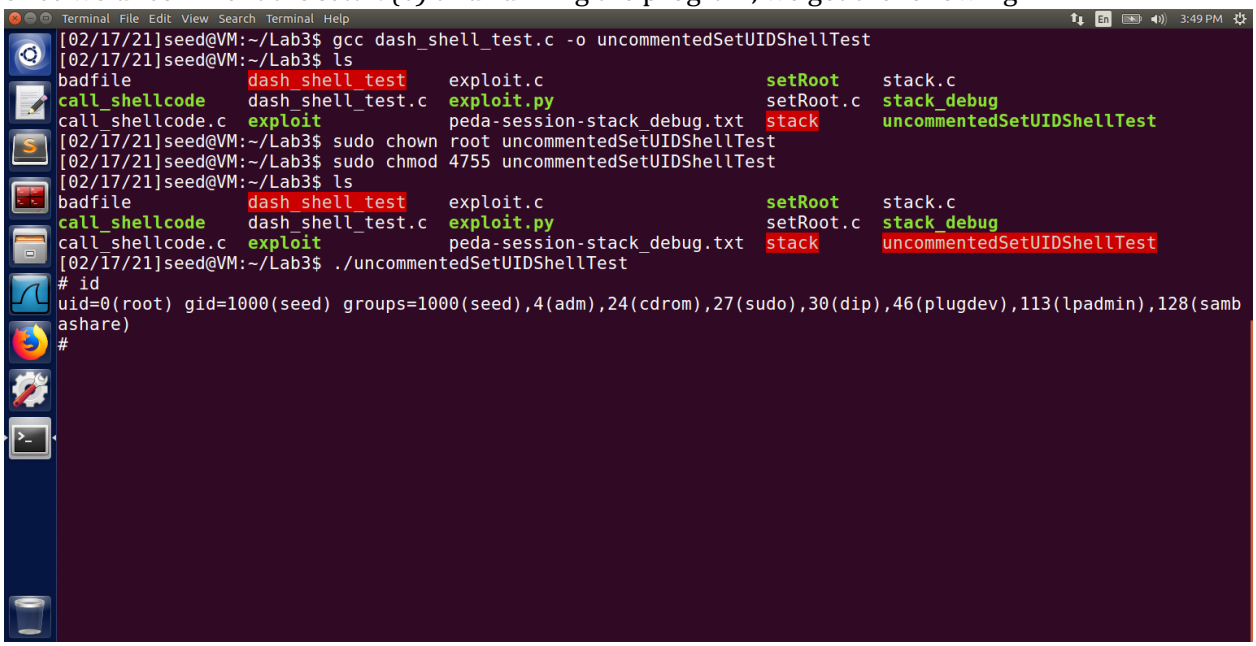


We can observe that we entered our own account's shell and Terminal is pointing to the User ID as *seed*.

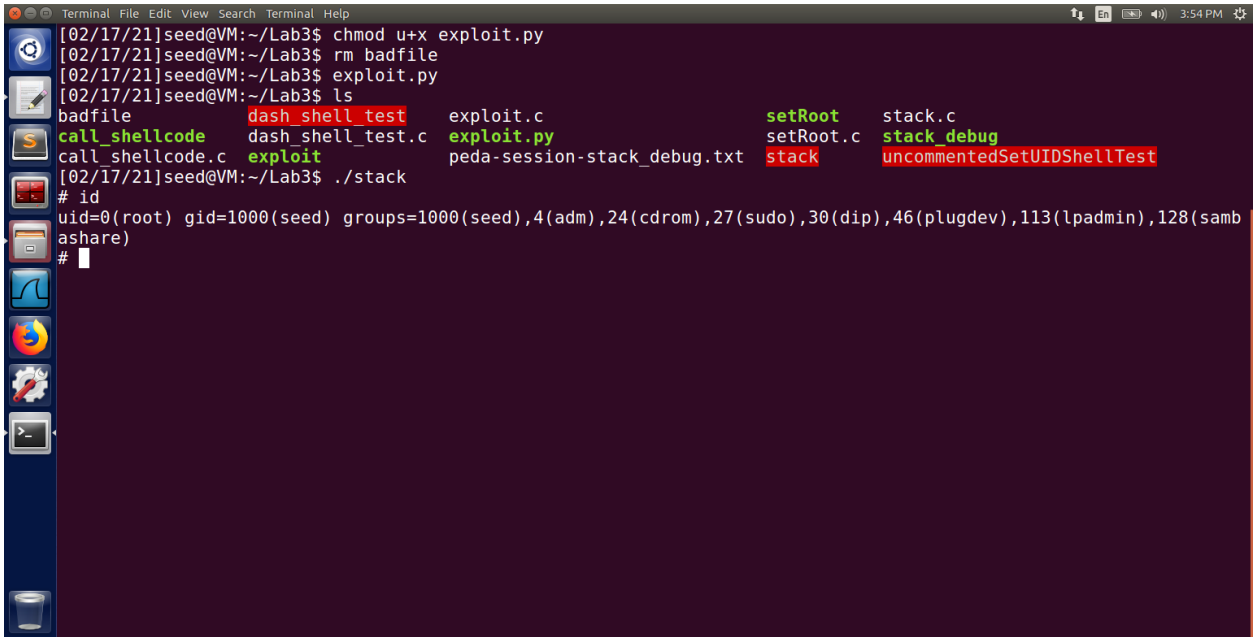


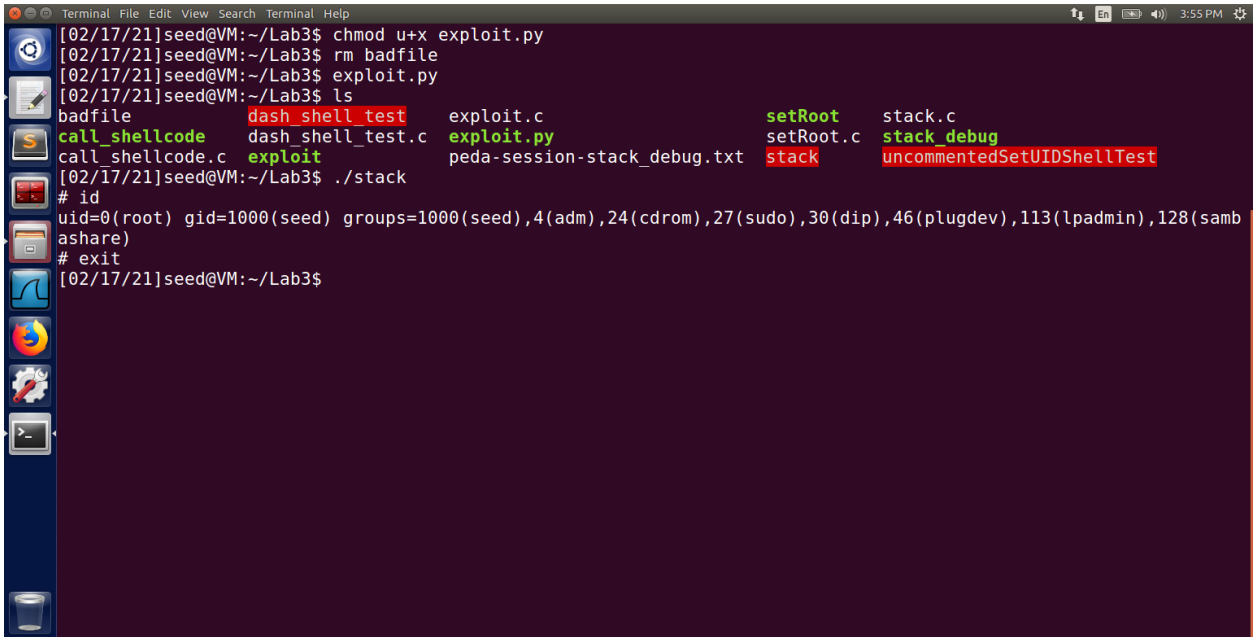


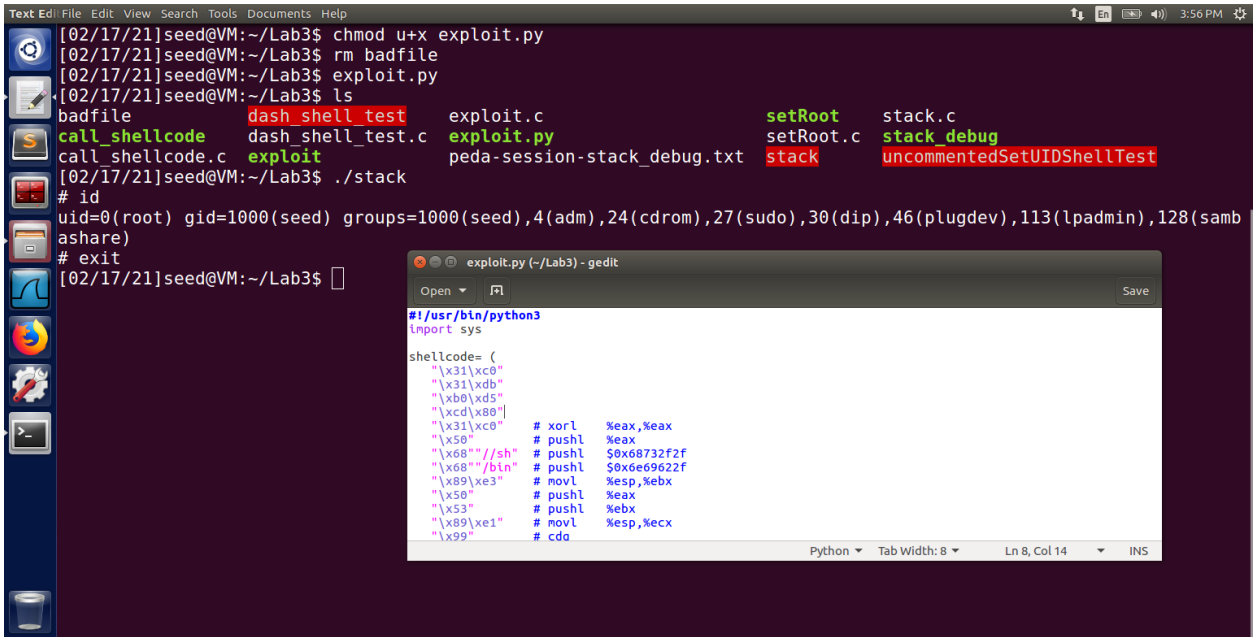
Once we uncomment the *setuid(0)* and running the program, we get the following:



Based on the above screenshots, we can observe that the *setuid(0)* command will remove the dash's countermeasure by setting the UID to root for SETUID programs. We will now do the same buffer overflow attack like we did in Task 2, but with the difference being */bin/dash* for SETUID programs is still present due to the link we did in the previous stages. We will add the assembly code to perform the setuid operation in *exploit.py* file before running the *execve()* command. Once we run it, we can build the *badfile* with the updated code and then run the *stack* program as well. We can observe in the screenshots that the attack was successful as the user id is that of the root user.

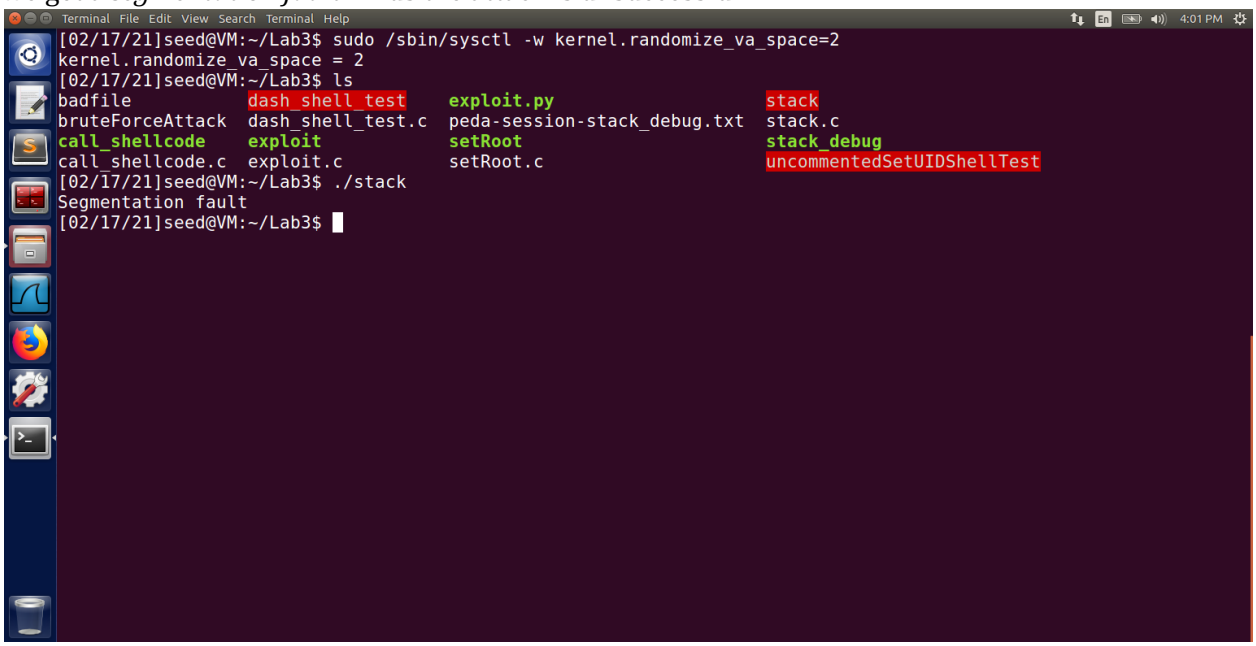

Terminal File Edit View Search Terminal Help
3:54 PM
[02/17/21]seed@VM:~/Lab3$ chmod u+x exploit.py
[02/17/21]seed@VM:~/Lab3$ rm badfile
[02/17/21]seed@VM:~/Lab3$ exploit.py
[02/17/21]seed@VM:~/Lab3$ ls
badfile dash_shell_test exploit.c setRoot stack.c
call_shellcode dash_shell_test.c exploit.py setRoot.c stack_debug
call_shellcode.c exploit peda-session-stack_debug.txt stack uncommentedSetUIDShellTest
[02/17/21]seed@VM:~/Lab3$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(samb
ashare)
#


Terminal File Edit View Search Terminal Help
3:55 PM
[02/17/21]seed@VM:~/Lab3$ chmod u+x exploit.py
[02/17/21]seed@VM:~/Lab3$ rm badfile
[02/17/21]seed@VM:~/Lab3$ exploit.py
[02/17/21]seed@VM:~/Lab3$ ls
badfile dash_shell_test exploit.c setRoot stack.c
call_shellcode dash_shell_test.c exploit.py setRoot.c stack_debug
call_shellcode.c exploit peda-session-stack_debug.txt stack uncommentedSetUIDShellTest
[02/17/21]seed@VM:~/Lab3$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(samb
ashare)
exit
[02/17/21]seed@VM:~/Lab3$

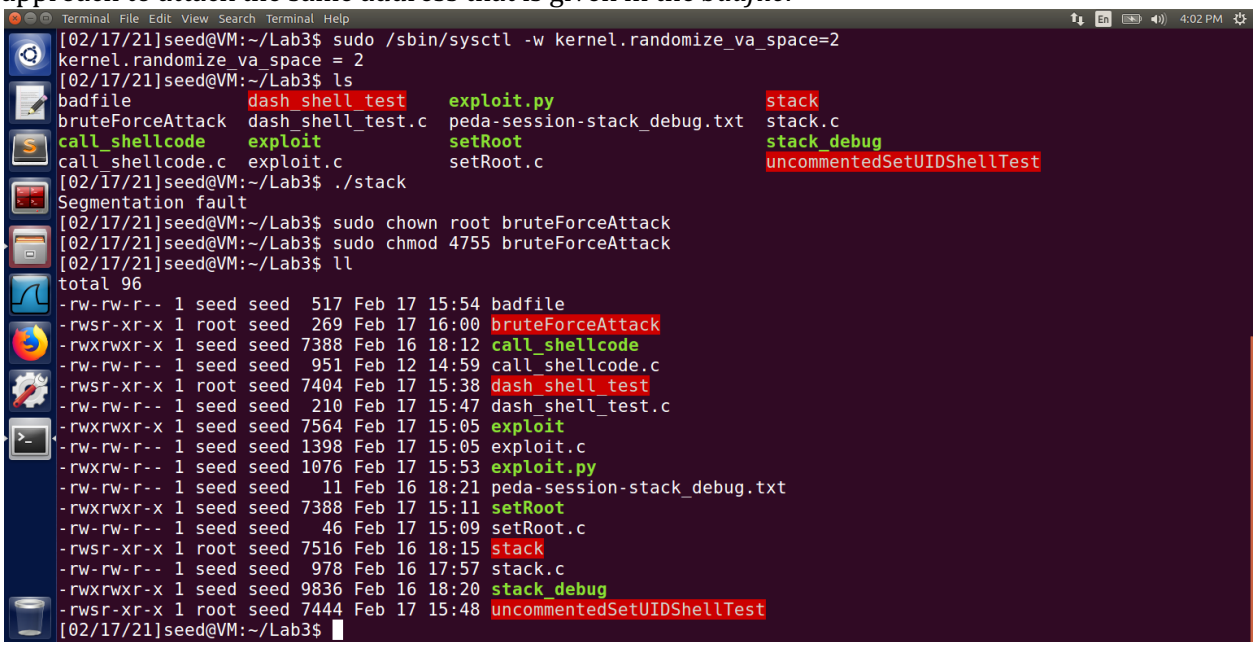


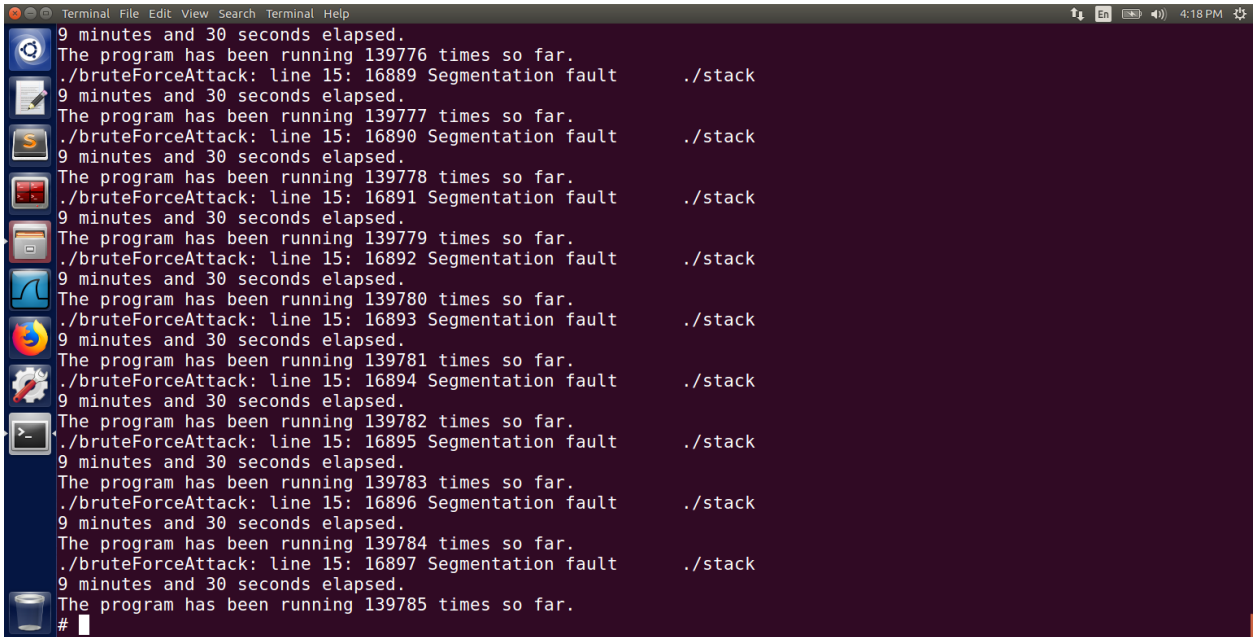
Task 4: Defeating Address Randomization

We first enable the Address Space Randomization for both the stack and the heap by setting the value to 2. The value 1 only denotes the stack address being randomized. When we try the attack, we get a *segmentation fault*. Thus the attack is unsuccessful!

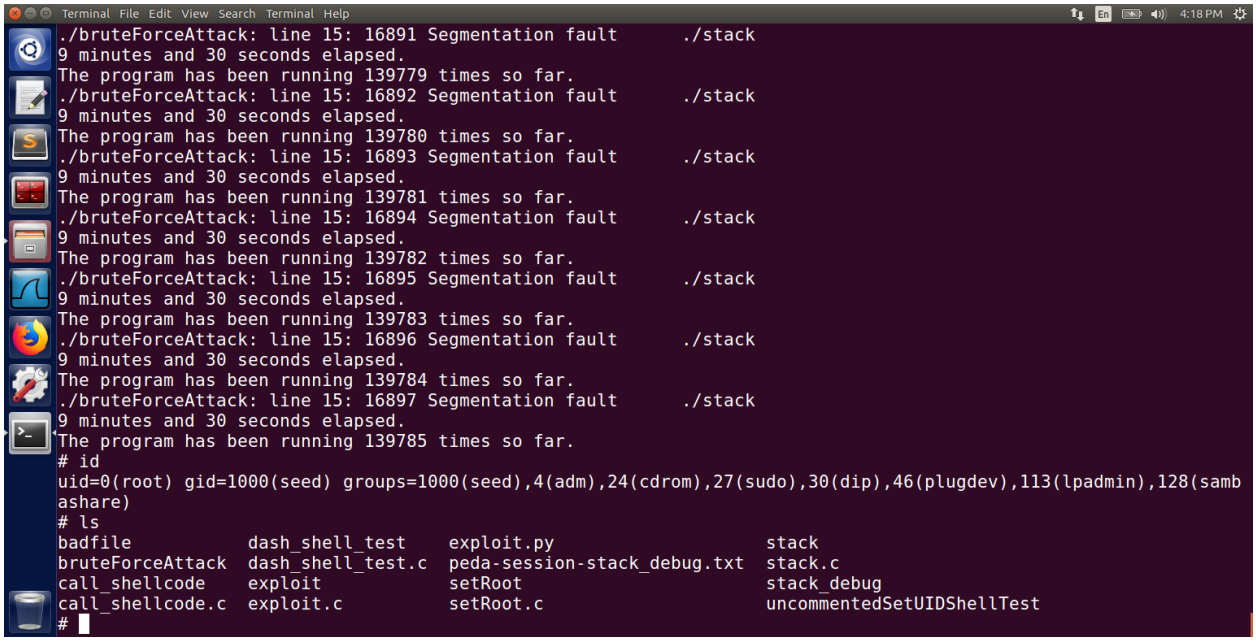


We then run the given script to make the program go on a loop. This is nothing but a brute-force approach to attack the same address that is given in the *badfile*.



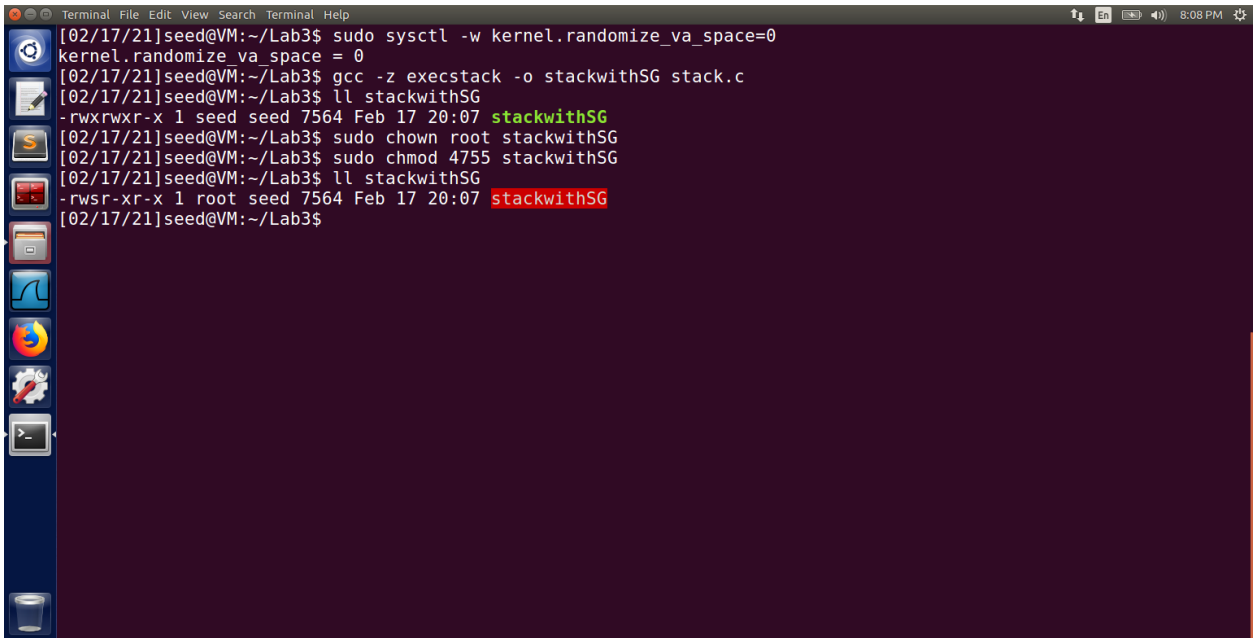


The output shows the time taken and the attempts taken to perform the attack with both Address Space Randomization and the Brute-Force Approach. This shows that the attack is successful! The main explanation here is that when Address Space Randomization was switched off, the stack always initialized from the same memory point for each program. This made it easy to find the offset (difference between return address and start of buffer) to place the bad code and the return address inside the program. But when Address Space Randomization is switched on, the stack's initialization always occurs in a randomized fashion. The only other way is the brute force approach where we try many number of times.

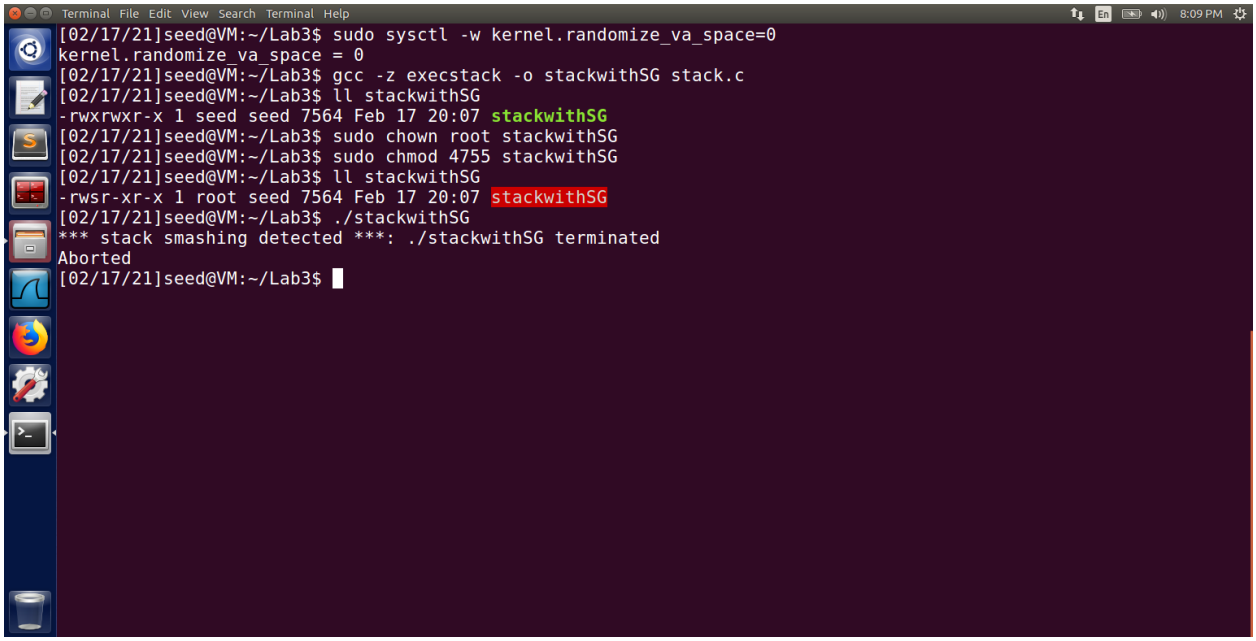


Task 5: Turn on the StackGuard Protection

First up, we disable the Address Space Randomization and we compile the program with StackGuard Protection Scheme and the Executable Stack. We also make this into a SETUID program.



When we run this vulnerable program, we can observes that the buffer overflow attack fails:



Task 6: Turn on the Non-Executable Stack Protection

We compile the program with StackGuard Scheme switched off and the non-executable stack as well. We convert this into a SETUID program. When we run it, we get the *segmentation fault* error as the buffer overflow attack did not work as intended and the program crashed.

This is because the stack is obviously no longer executable. Whenever we perform this kind of attack, we run it with root access and hence, the code can be very dangerous. By removing the executable feature, the normal program will be able to run the same but without any side effects. Here, the malicious code is considered as data rather than code and hence it is not treated as a program (just as a read-only data, maybe). Therefore the attack will fail unlike before as the stack was executable.