Homework 3:- Buffer Overflow Vulnerability Lab

Name:- Aparna Krishna Bhat

ID:- 1001255079

Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundaries of pre-allocated fixed length buffers. This vulnerability can be used by a malicious user to alter the flow control of the program, leading to the execution of malicious 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.

1) ***-fno-stack-protector:*** This option turns off the Stack-Guard Protection Scheme, which could defeat the stack-based buffer overflow. It detects buffer overflow by adding special data or checking mechanism in the code.
2) ***-z execstack***: By providing this parameter, the stack becomes executable which then would allow our code to execute when in stack. As a countermeasure by default the stack is non-executable, and the OS knows whether the stack is executable or not by a binary bit set in the system. This bit can be manipulated by the compiler, and the gcc compiler sets the stack as non-executable by default.

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

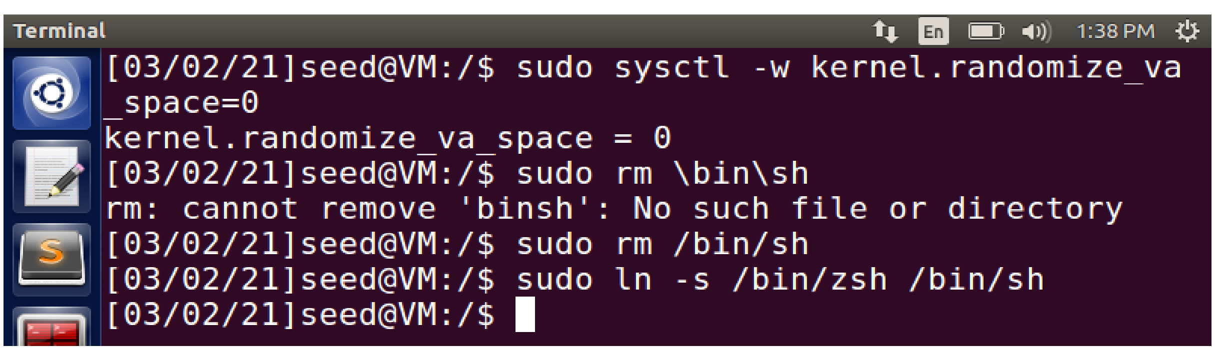


Fig 1:- Countermeasures turned off

<u>Task 1:- Running Shellcode</u>

The aim of this task is to learn how to launch a shell by executing a shellcode stored in a buffer.

I have created the files ***call_shellcode.c, dash_shell.c, stack.c*** in the folder named Lab3. Compiled the file ***call_shellcode.c*** by passing the parameter ***'-z execstack'*** to make the stack executable, to execute the shellcode and not give me any errors such as segmentation fault. The compiled program is stored in the output file named 'call_shellcode.' Next, I executed this compiled program file, and we can notice from the output that I have entered the shell of the account (indicated by $). Since there were no errors, it proves that the program got executed successfully, and I ***got access to '/bin/sh'***. Since it was neither a Set-UID root program nor I were in the root account hence, the terminal was my account and not the root.

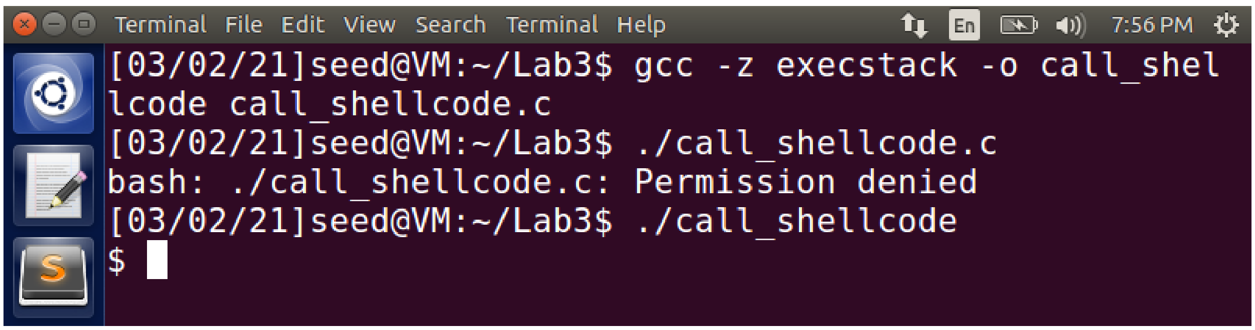
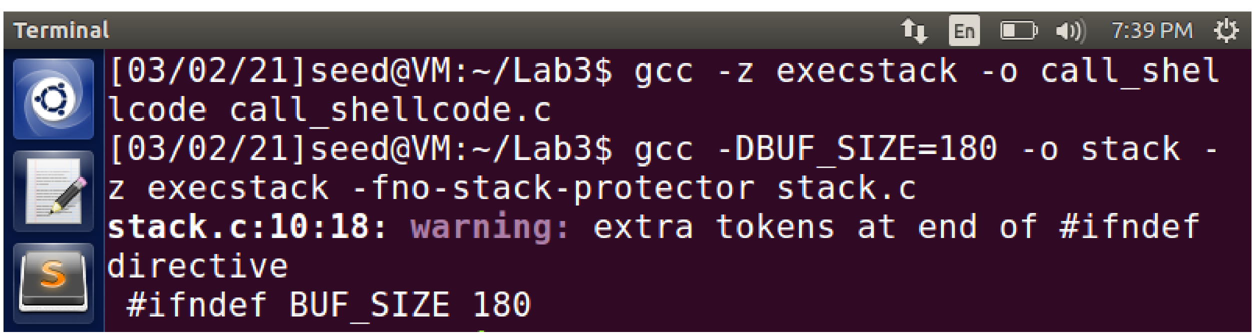


Fig 2

Next, compiled the given vulnerable program ***stack.c*** and while compiling ***disabled the Stack-Guard*** Protection mechanism and made the stack executable by passing the respective parameters to the command. Also, the compiled program is stored in the output file 'stack' which is then made a Set-UID root program. From the below screenshot, we can notice call_shellcode.c is highlighted in green color which means it is an executable file and stack.c is highlighted in red color which means it is a Set-UID program.



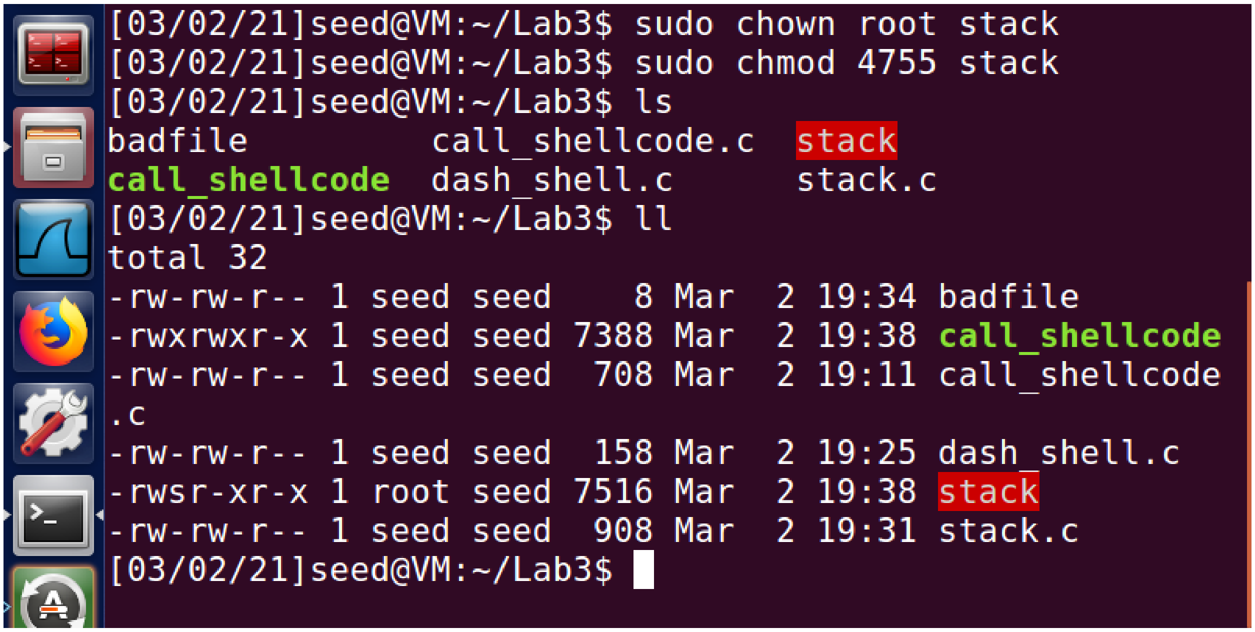


Fig 3

The below screenshot shows the normal functioning of the stack program

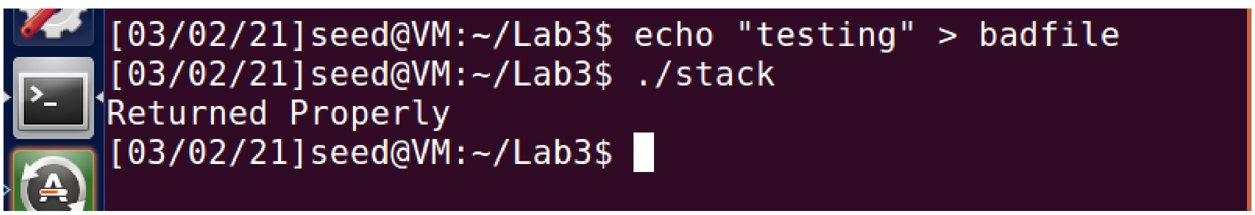


Fig 4

Task 2:- Exploiting the Vulnerability [The BUF_SIZE value used is 180]

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 build the right buffer payload that will help us to execute our desired program. Hence, we first compile the program in the debug mode (-g option), with the Stack-Guard countermeasure disabled and Stack executable and then execute the program in debug mode using ***gdb.***

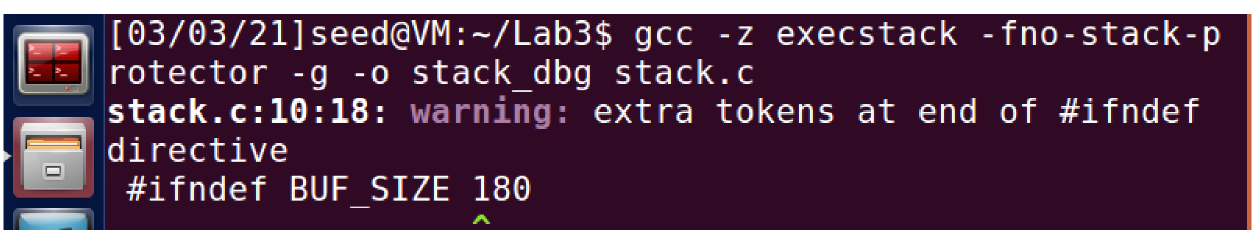


Fig 5

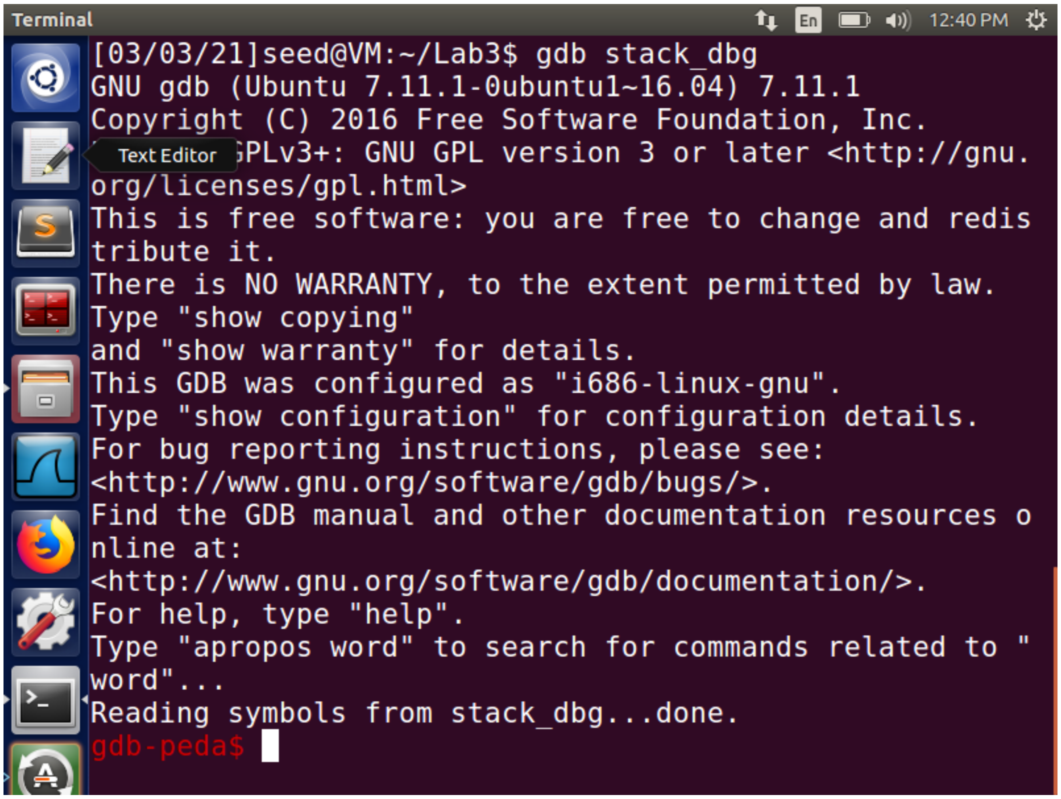


Fig 6

In gdb, we set a ***breakpoint on the bof function using b bof***, and then run the program

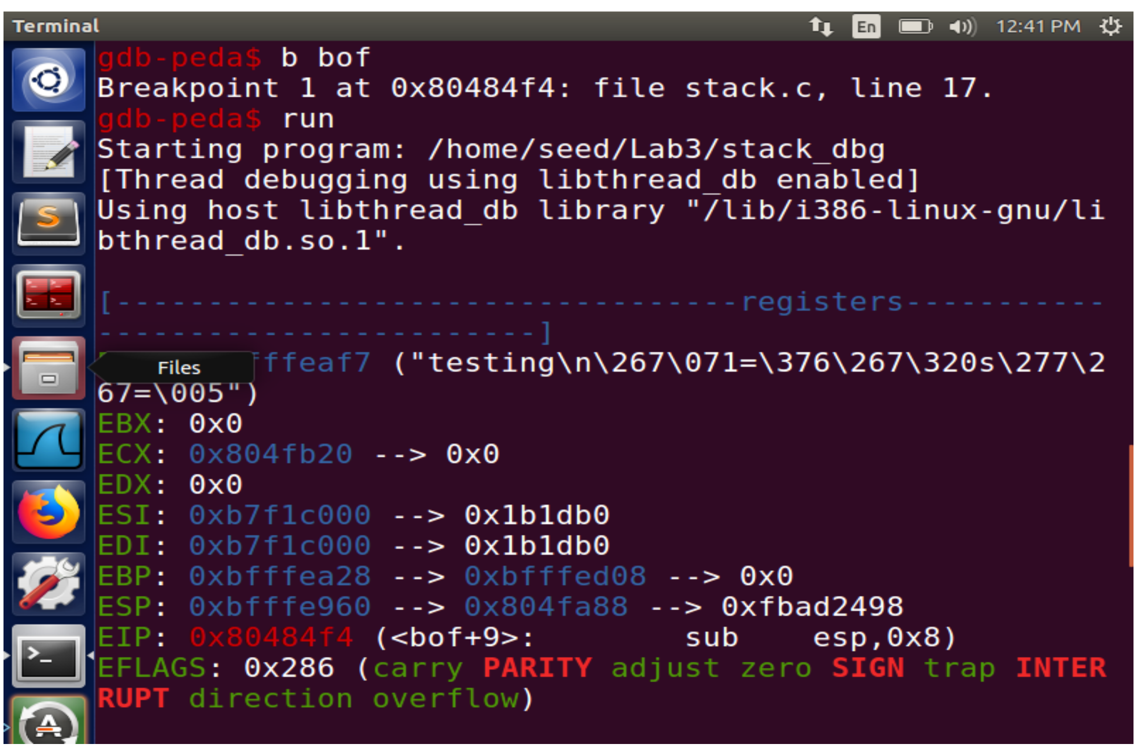


Fig 7

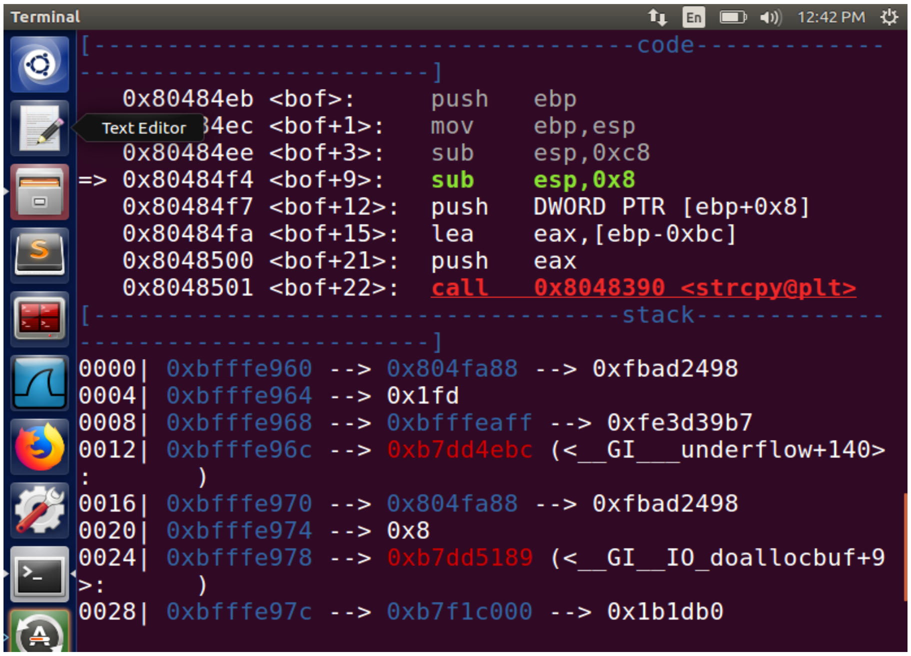


Fig 8

The program execution stops inside the bof function because of the created breakpoint. The stack frame values for this function are important and will be used to build the badfile contents. Here, we print out the ebp and buffer values, and find the difference between the ebp and start of the buffer to find the return address value's address.

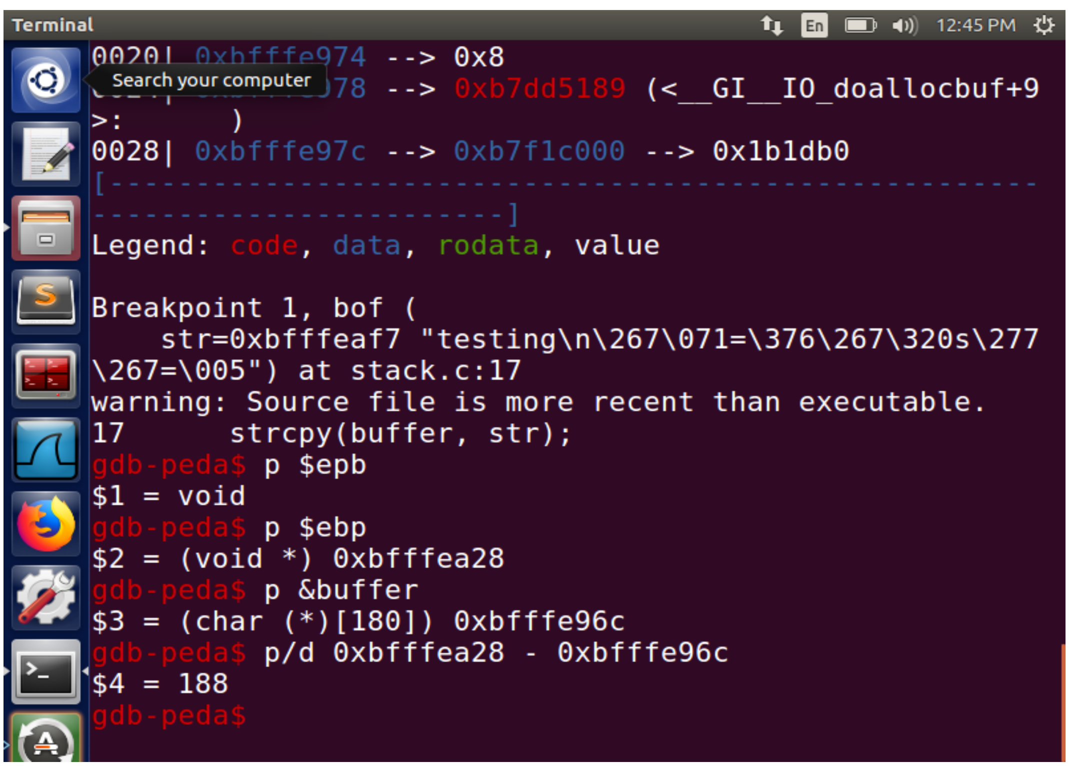


Fig 9

Now to find the return address we need to find the address of ebp,as from the above screenshot we can notice that return address is at ebp +4 byte (in 32-bit OS). After finding the address of ebp we need to find the starting address of buffer[180] which is buffer[0] so that we can calculate offset. As we know that the return address is at ebp + 4 byte as the size of frame pointer is 4 bytes, so the return address is at 188+4=192 bytes from the buffer. After determining the return address, we replace the return address by ebp + offset where offset can be any value that will map it to the address containing NOP instructions that will eventually lead to the address containing shell code.

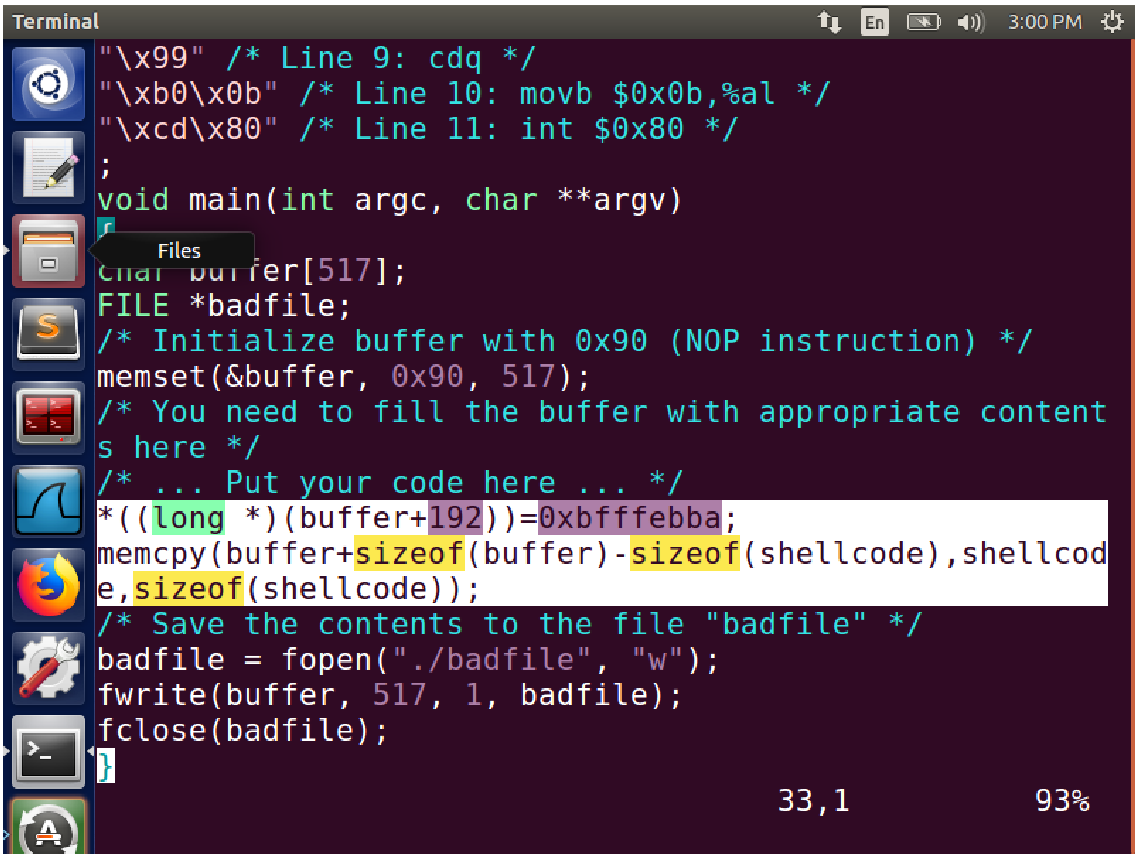
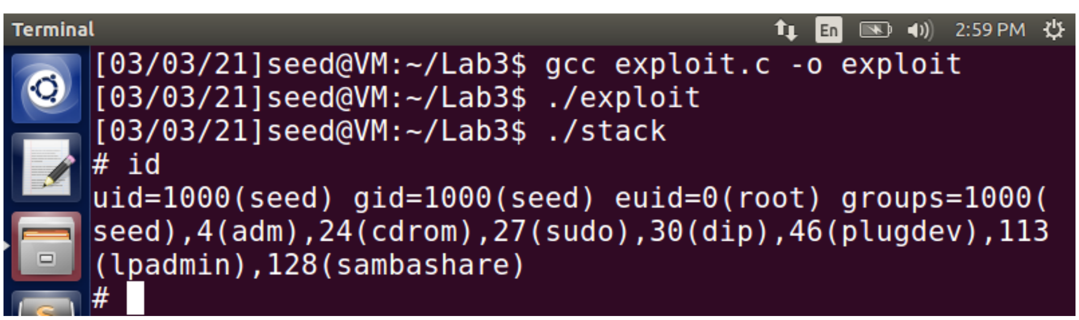


Fig 10:- exploit.c file

compile the exploit.c file then we run the file by ./exploit and ./stack. Once we run ./stack we get the root access. The below screenshot shows these operations.



For python program first we make the python program executable and run the exploit.py file to generate the badfile. 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 the root shell. The EUID is seen to be that of the root (0):

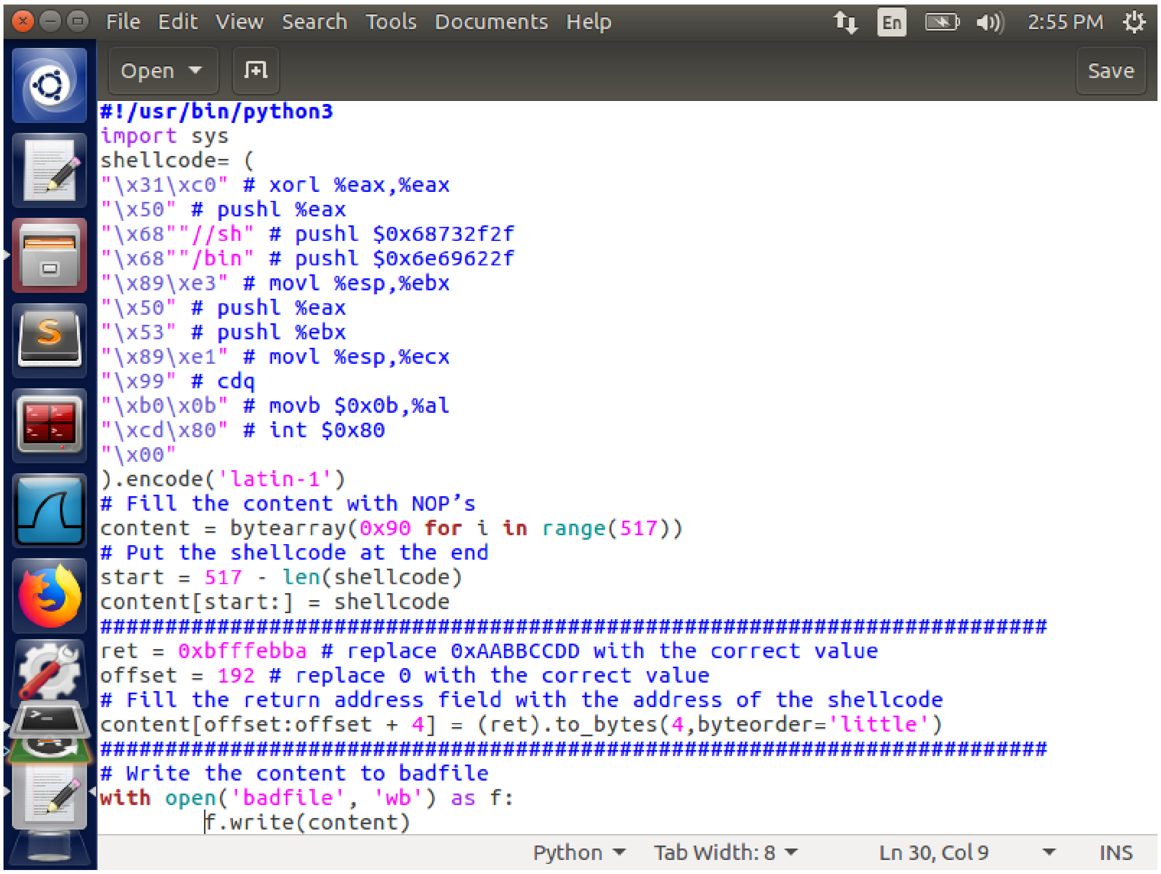


Fig 11:- exploit.py modified file

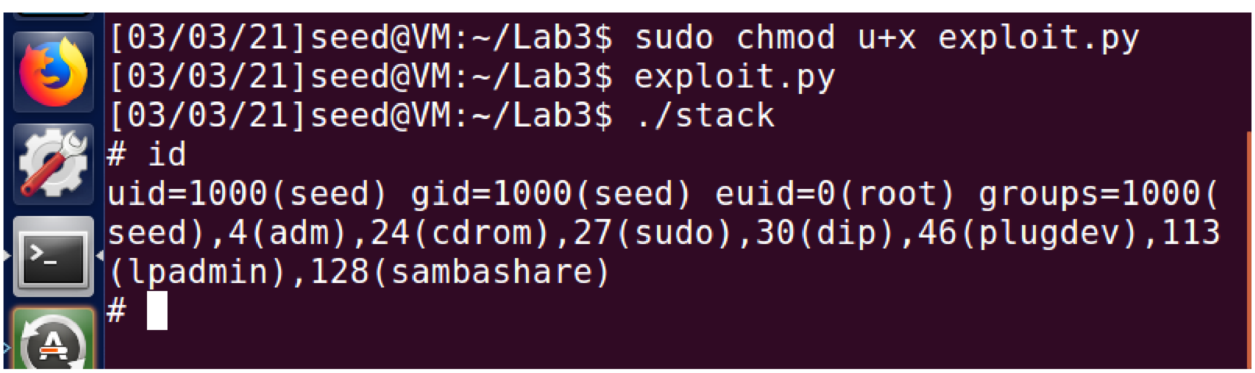
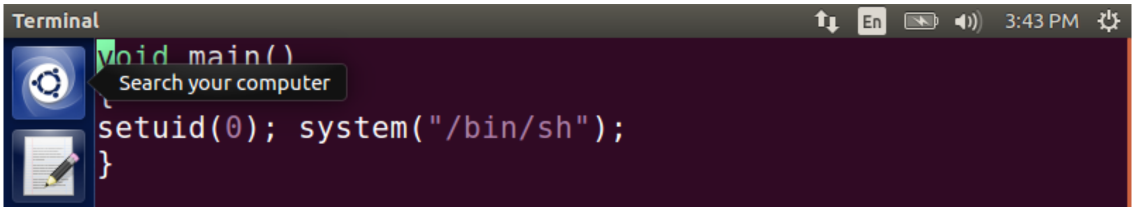


Fig 12:- Gained root access

From the above screenshot we can notice that 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). So, we run our program to turn our real UID to root as well. We create and compile the make_root.c program that changes the UID of the account to 0, which is of the root.



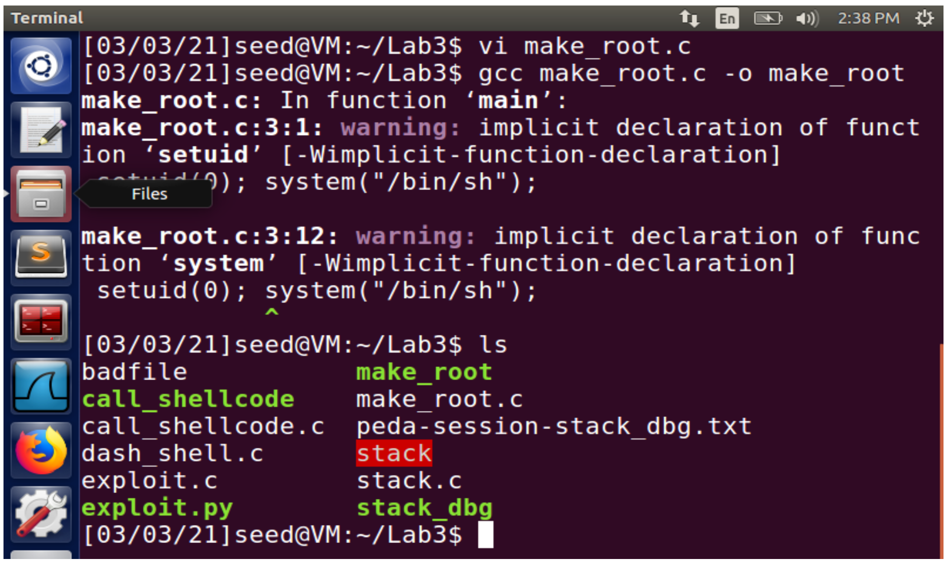
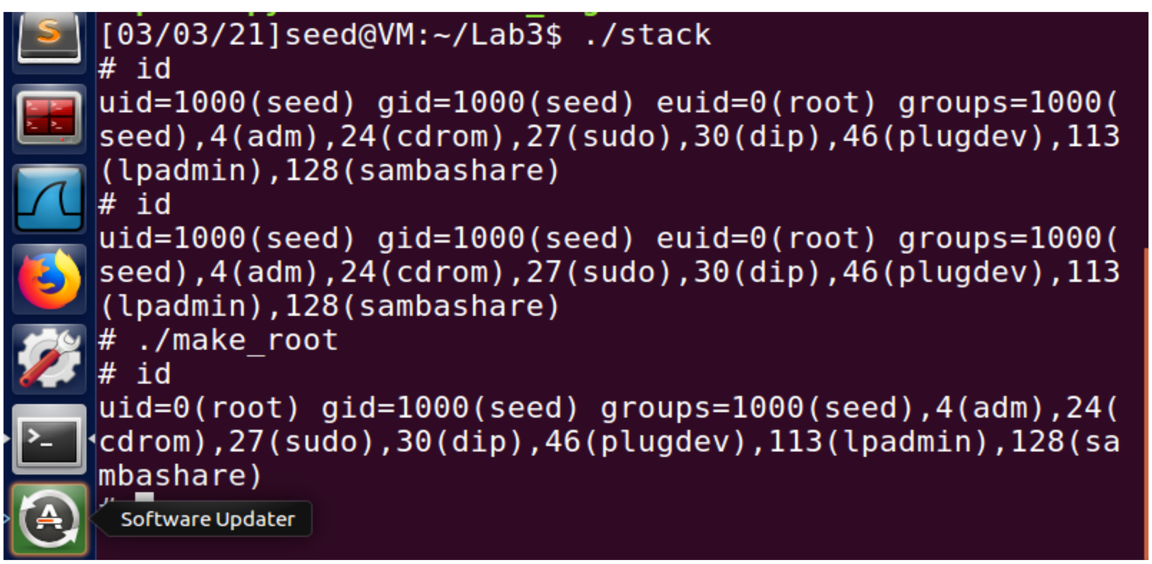


Fig 13

Note that make_root is not a Set-UID root program. Next, we run this program in the root terminal to set the UID as 0 (from the program). Since, we have the root privileges already due to the successful buffer overflow attack, we can change the UID to 0 without any issues.



Task 3:- Defeating dash's Countermeasure

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

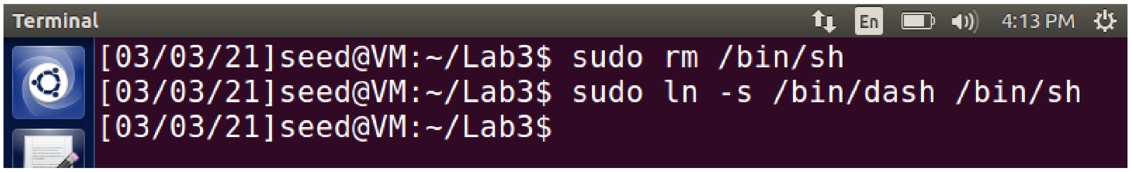


Fig 14

Compile the ***das_shell_test.c*** file and make it a Set-UID root program Fig 15 screenshot shows these operations. On executing this program, we can notice from Fig 16 that we have entered our own account shell and the program's user ID is that of the seed.

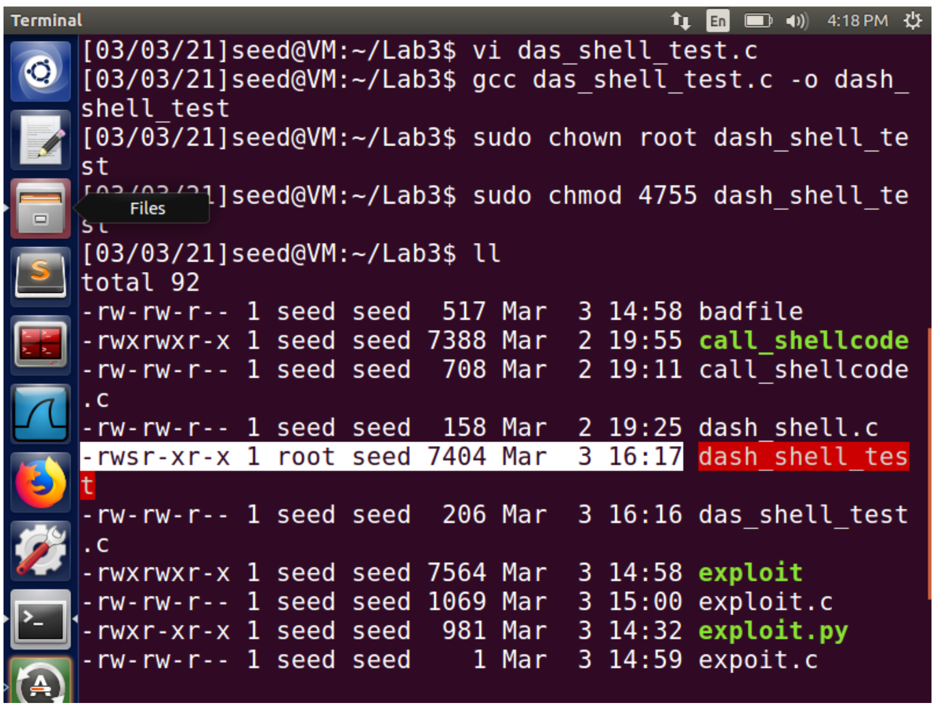


Fig 15

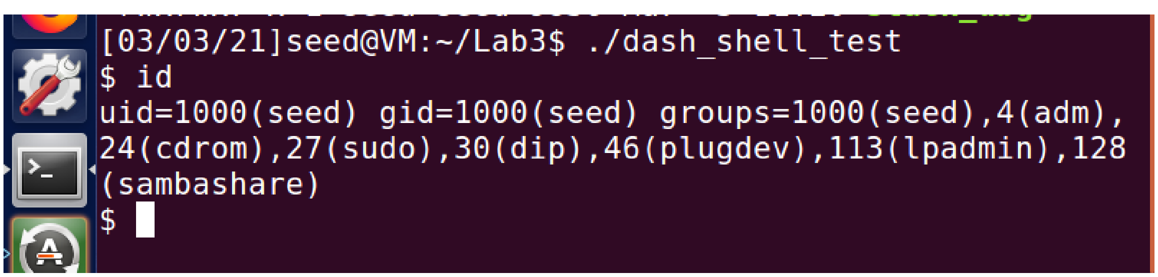


Fig 16

After uncommenting setuid(0) in the das_shell_test.c program we can notice that it is a non-Set-UID program. Next, making it a Set-UID program and executing it. we can enter the root shell and on checking for the user ID, we can notice that it is that of the root.

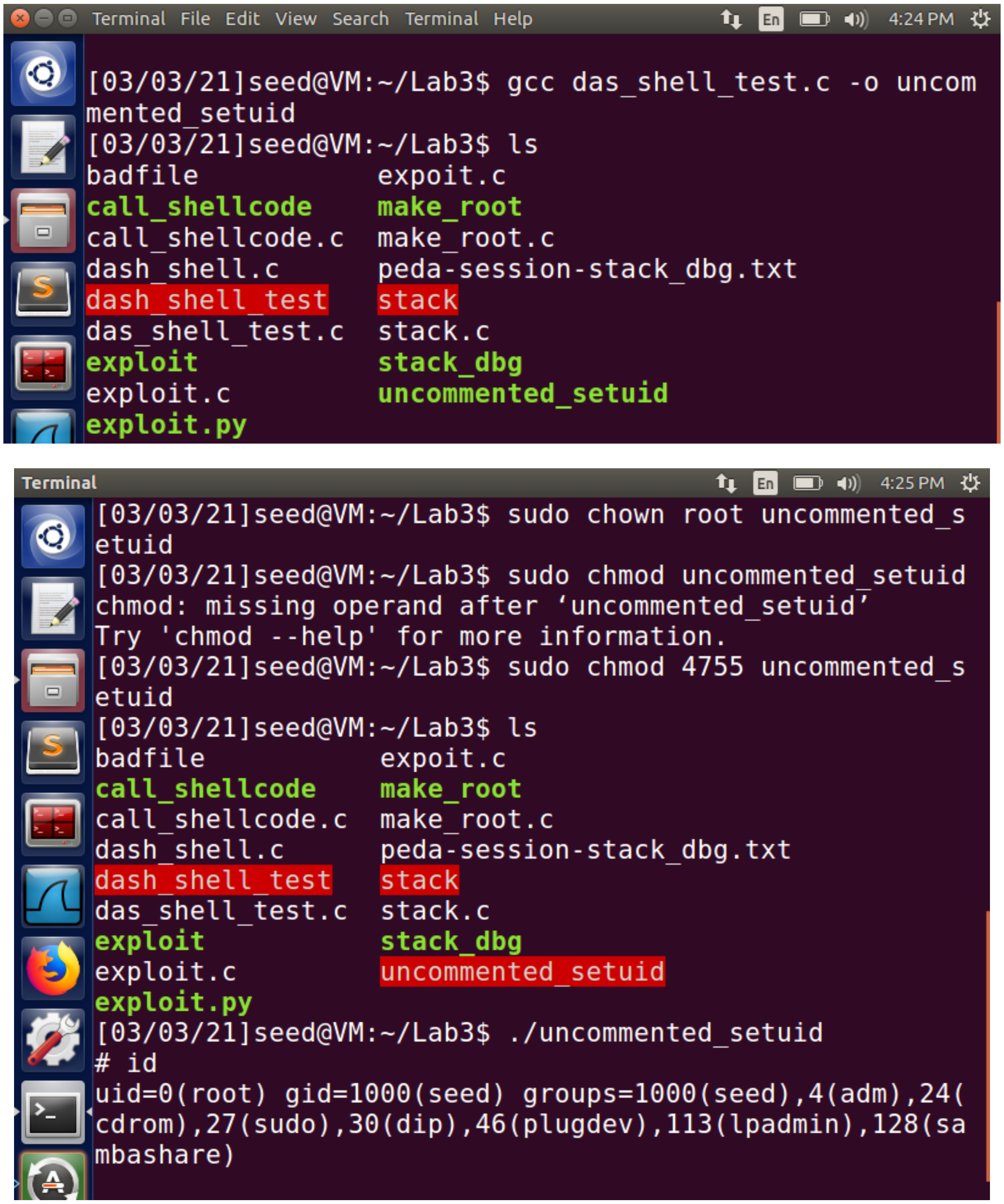
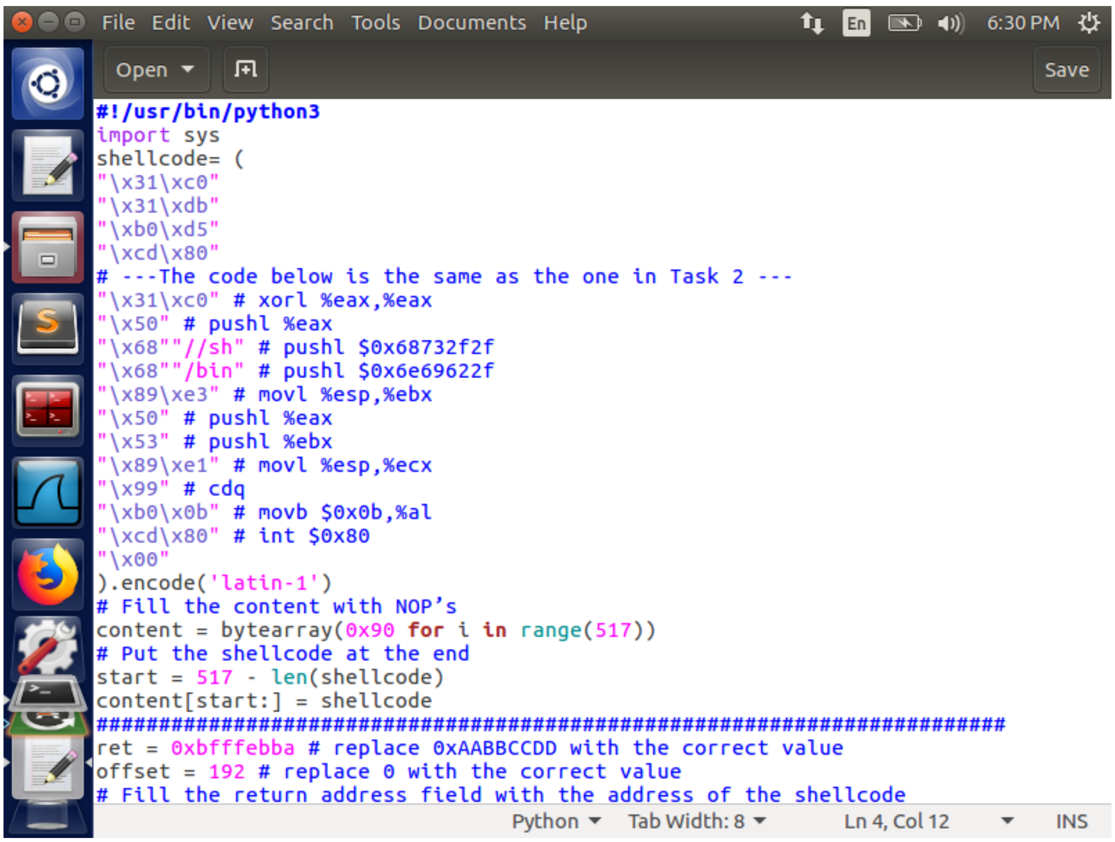


Fig 17

We can notice from the Figs 16 and 17 that both the times we got access to the shell, but in the first one it is not of the root. Since the effective user id and the actual user id are different bash program drops the privileges of the Set-UID program. So, it is executed as a program with normal privileges and not root. For the second one by uncommenting the setuid(0) command in the das_shell_test.c program, it made a difference. Since it is a Set-UID program the actual user id is set to that of root, and the effective user id is 0 , and hence the dash does not drop any privileges here, and the root shell is run. This command, hence, can defeat the dash's countermeasure by setting the uid to that of the root for Set-UID root programs, and providing with root's terminal access.

Next, we try to perform the buffer overflow attack, in the same manner as we did it for task 2. 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.py, even before we invoke execve(). 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 syscall number), and (3) execute the system call in Line 4.




```
#!/usr/bin/python3
import sys
shellcode= (
"\x31\xc0"
"\x31\xdb"
"\xb0\xd5"
"\xcd\x80"
# ---The code below is the same as the one in Task 2 ---
"\x31\xc0" # xorl %eax,%eax
"\x50" # pushl %eax
"\x68""//sh" # pushl $0x68732f2f
"\x68""/bin" # pushl $0x6e69622f
"\x89\xe3" # movl %esp,%ebx
"\x50" # pushl %eax
"\x53" # pushl %ebx
"\x89\xe1" # movl %esp,%ecx
"\x99" # cdq
"\xb0\x0b" # movb $0x0b,%al
"\xcd\x80" # int $0x80
"\x00"
).encode('latin-1')
# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))
# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode
##########################################################################
ret = 0xbfffebba # replace 0xAABBCCDD with the correct value
offset = 192 # replace 0 with the correct value
# Fill the return address field with the address of the shellcode
```

Fig 18:- Updated exploit.py

On executing this exploit.py, we build the badfile with updated code to be executed in the stack, and then execute the stack Set-UID root program. From the Fig 19 we can notice that we are able to get access to the root's terminal and we can also notice that the user id (uid) is same as that of the root. Therefore, the attack was performed successfully, and we were able to overcome the dash's countermeasure by using setuid() system call.

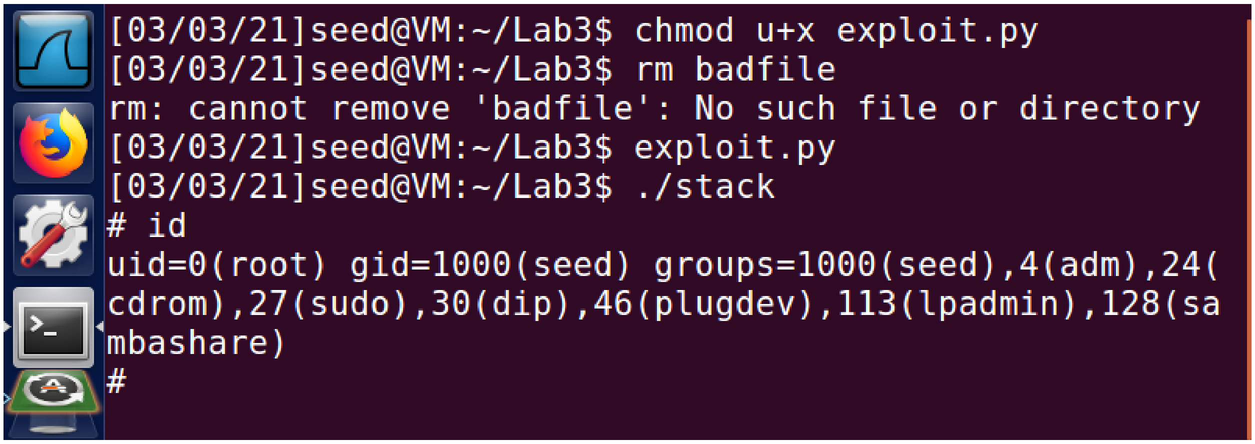


Fig 19

<u>Task 4:- Defeating Address Randomization</u>

First, we turn on the address randomization for both stack and heap by setting the value to 2. Then on executing the same attack as that in Task2, we get segmentation fault. This demonstrates that the attack was not successful. The below screenshot shows the above-mentioned operations.

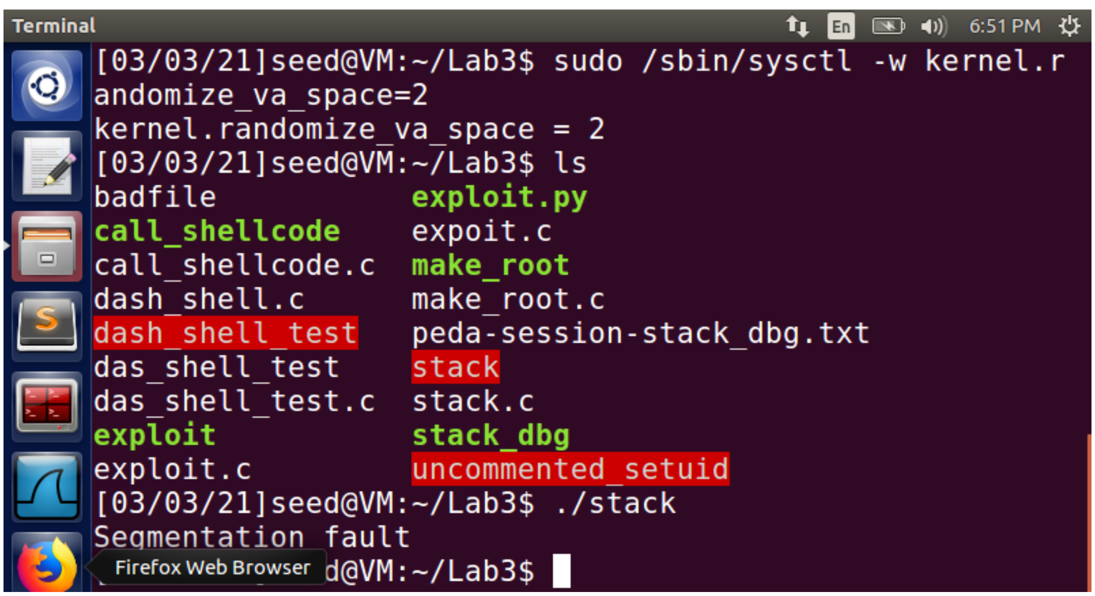


Fig 20

Next, we execute the shell script provided to us to execute the vulnerable program in loop. This is basically a brute-force approach to hit the same address as the one we put in the badfile. The shell script is stored in the brute_attack file and is made a Set-UID root program.

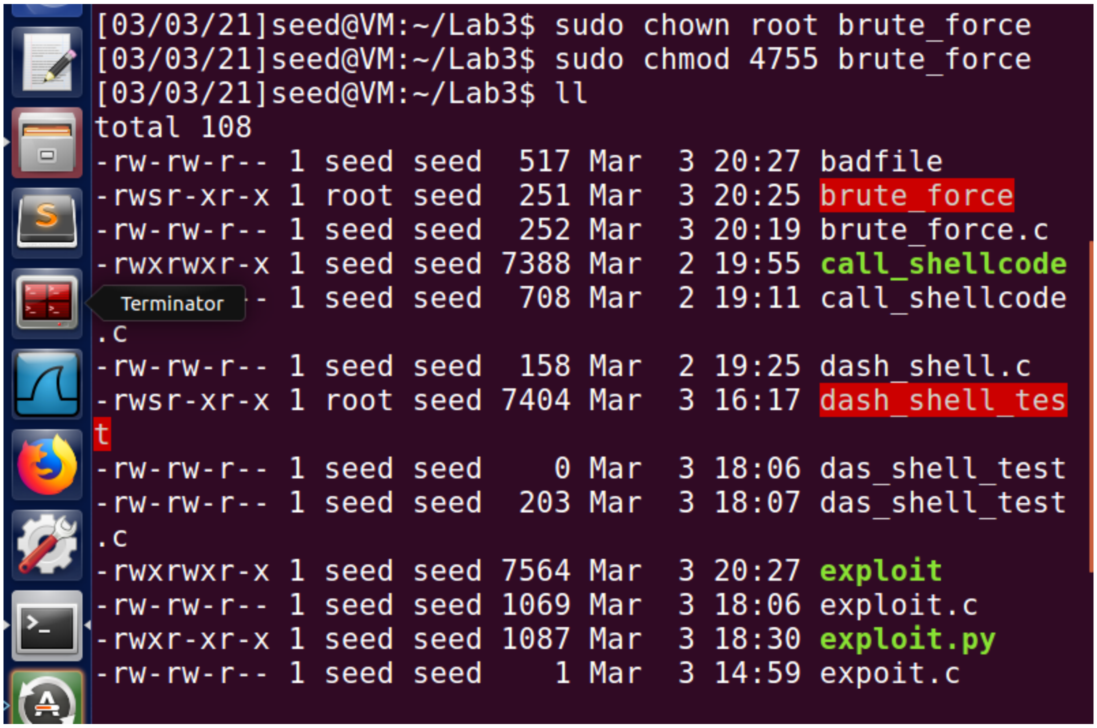


Fig 21:- brute_force file made a Set-UID program

The below screenshot 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.

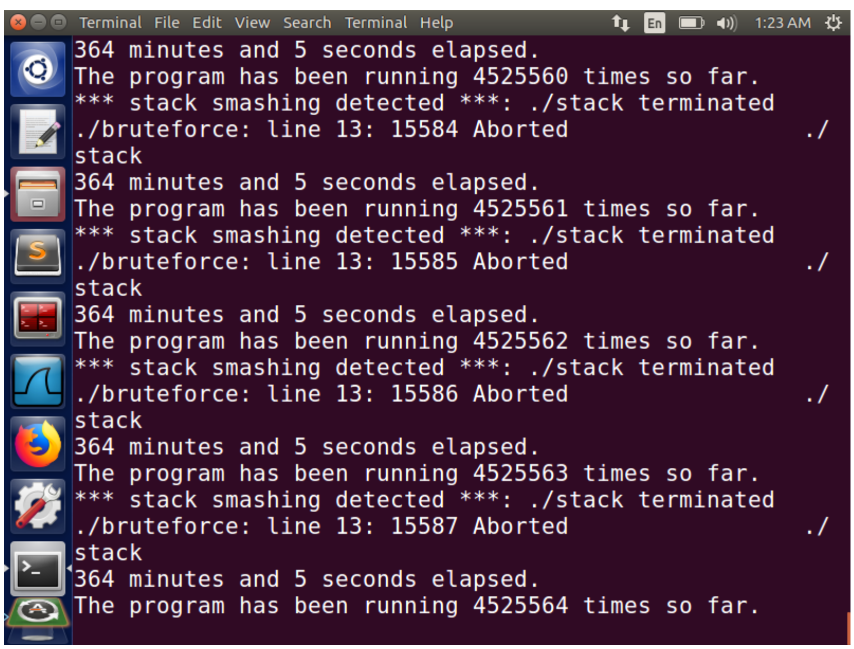


Fig 22:- Brute-Force Approach

Observations to be made here are when the address space layout randomization countermeasure was disabled, the stack frame always started from the same memory point for each program for the sake of simplicity. This made it simple for us to guess or find the offset, which is the difference between the return address and the start of the buffer, where we could place our malicious code and return address in the program. The stack frame's starting point is always randomized and different when the address space layout randomization countermeasure is enabled. To perform the overflow, we are unable to accurately determine the exact starting position or offset. Unless we hit the address that we specify in our vulnerable code, the only option left is to try as many times as possible. When the brute force program is run, it continuous to run until it reaches the address that allows the shell program to run. Eventually we get to the root terminal (because this is a Set-UID root program), which is indicated by #.

The probability of the attack succeeding is (1/2)^32 for a 32-bit machine and this mechanism is not the safest way to stop the execution of a malicious code from the buffer as this probability is not very small for the computer.

Task 5:- Turn on the Stack Guard Protection

We begin by turning off the Address Randomization countermeasure. The program 'stack.c' is then compiled with Stack-Guard protection (by omitting -fno-stack-protector) and an executable stack (by providing -z execstack). The compiled program is then converted into a Set-UID root program. The below screenshot shows these operations.

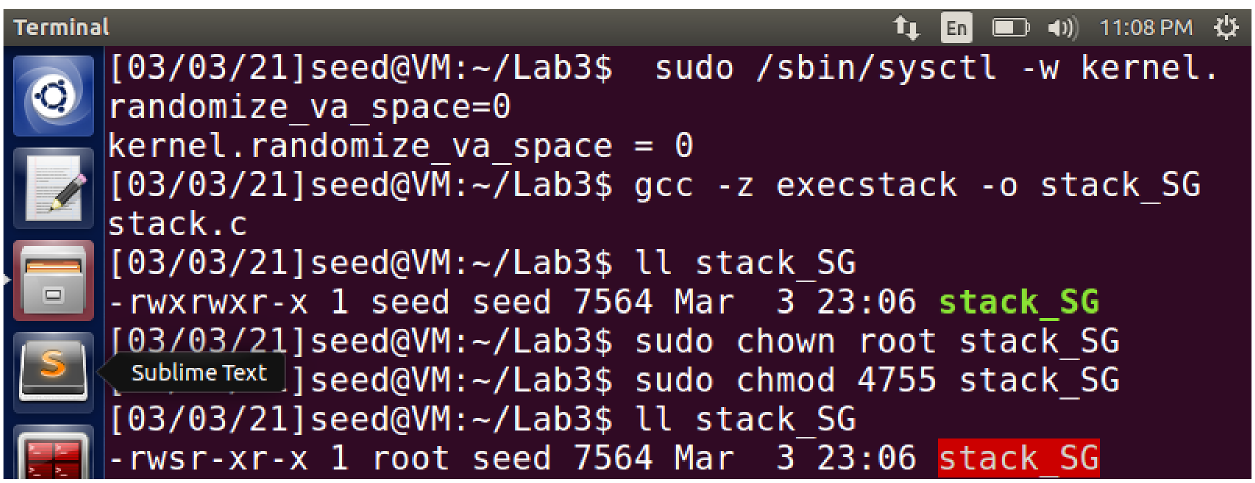


Fig 23

After that, we run this vulnerable stack program and notice that the buffer overflow attempt fails because of the error below, and the process gets aborted. This demonstrates that a Buffer overflow attack can be detected and prevented using the Stack-Guard protection mechanism.

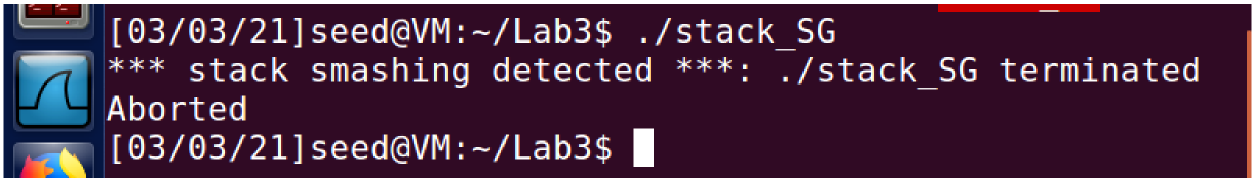


Fig 24

Task 6:- Turn on the Non-executable Stack Protection

In this task we will build the stack program and disable address randomization. We compile the program with no stack protection and make the stack a non-executable stack. We then make the program a root-owned Set-UID program. Next, compile and execute the vulnerable program which creates the badfile. A segmentation error occurs when we execute the output file stack_NE, the program gets terminated. The screenshot below demonstrates the buffer overflow attack that failed and crashed the program.

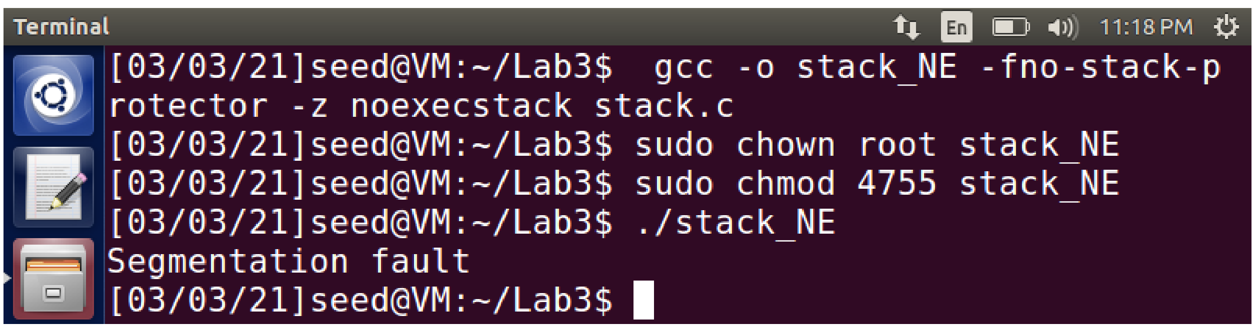


Fig 25

The stack is no longer executable, which simply causes this error. When we use a buffer overflow attack, we are attempting to run a program that could easily give us root access and thus be extremely malicious. However, since this program is typically stored in the stack, we attempt to enter a return address that points to the malicious program. Only local variables and arguments, as well as return addresses and ebp values, are stored in the stack memory layout. However, since none of these values need execution, the stack does not need to be made executable. As a result, by disabling the executable functionality, normal programs will continue to run normally with no side effects, but malicious code will be treated as data rather than code. It is viewed as read-only data rather than a program. As a result, our attack fails, as opposed to before, when our attacks were successful due to the stack being executable.