ASSIGNMENT – 3

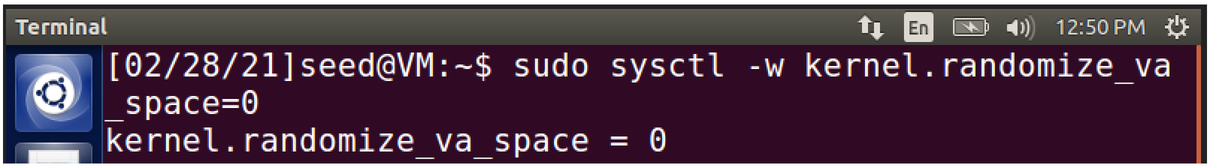
Name: **Sudharsan Srinivasan**

UTA ID: **1001755919**

Turning off countermeasures:

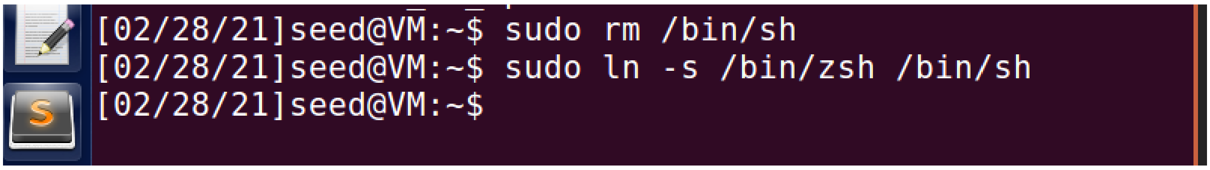
- Address Space Randomization is done to stop randomizing the starting address of heap and stack. This makes guessing the exact address difficult, thereby making the buffer-overflow attack also difficult.

The following image shows the command required for the same.



Configuring /bin/sh

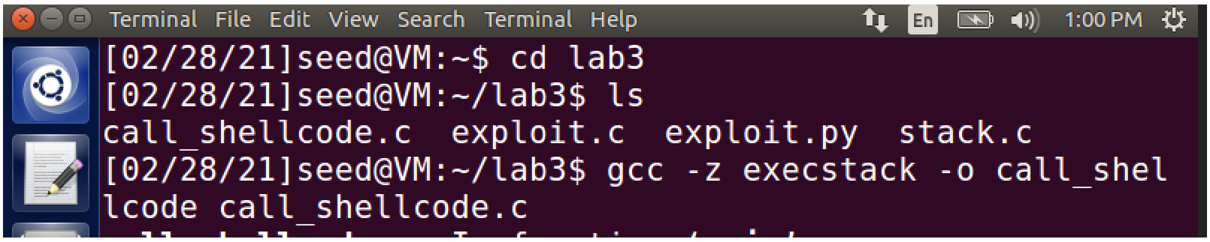
- Here, the victim program is a Set-UID program and the countermeasure in /bin/sh makes our attack more difficult.



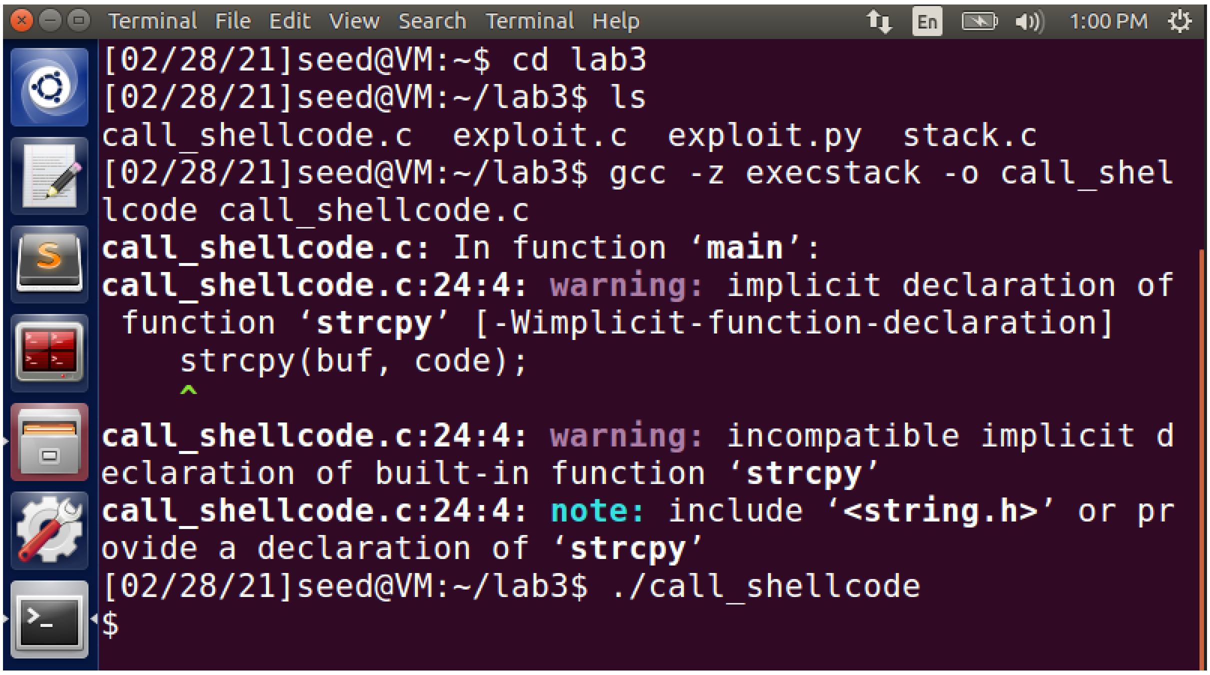
- There we change our shell from **'dash'** to **'zsh'**, linking **/bin/sh** to another corresponding shell that does not have this countermeasure.

Task 1 – Running Shellcode:

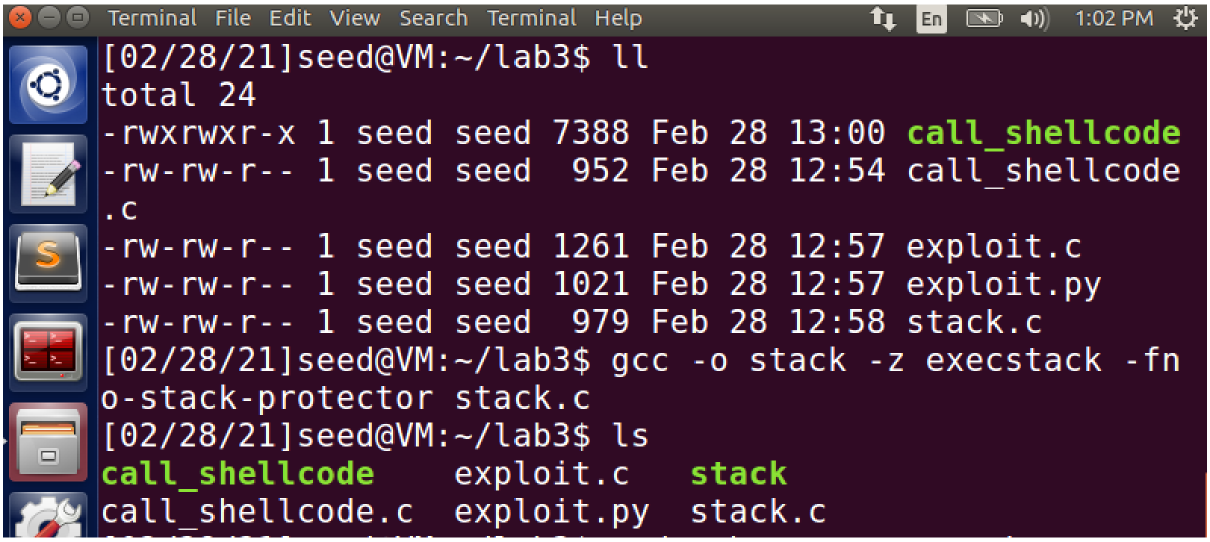
- In this task, we create a folder lab3 and store the files **stack.c, exploit.c, call_shellcode.c** and **exploit.py** (files provided for the task) in that folder.
- Then, we use **'-z execstack'** to compile the call_shellcode program into a file, 'call_shellcode'



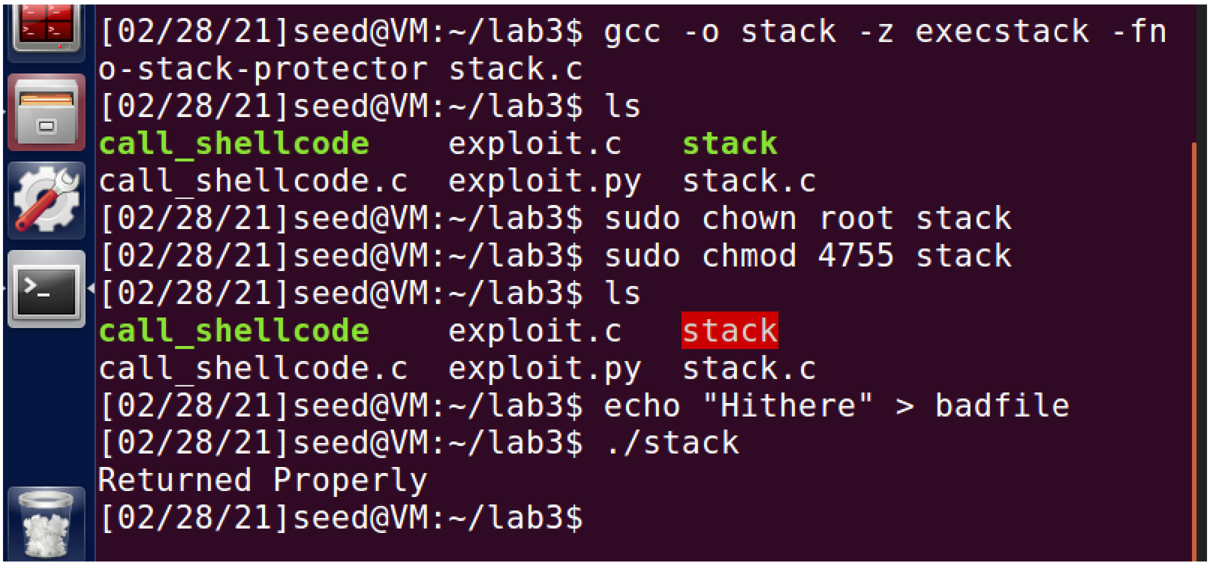
- As seen above, we use **execstack** to make sure that our stack is executable and to run the call_shellcode program. Then, we run the call_shellcode program.



- From the above screenshot, we can see that we have accessed ‘/bin/sh’ and entered the shell of the account, which is indicated by the ‘**$**’ sign.
- Next, we compile the vulnerable stack.c program by turning off Stack-Guard Protection measures and by using execstack option, to make the stack executable, as shown below.

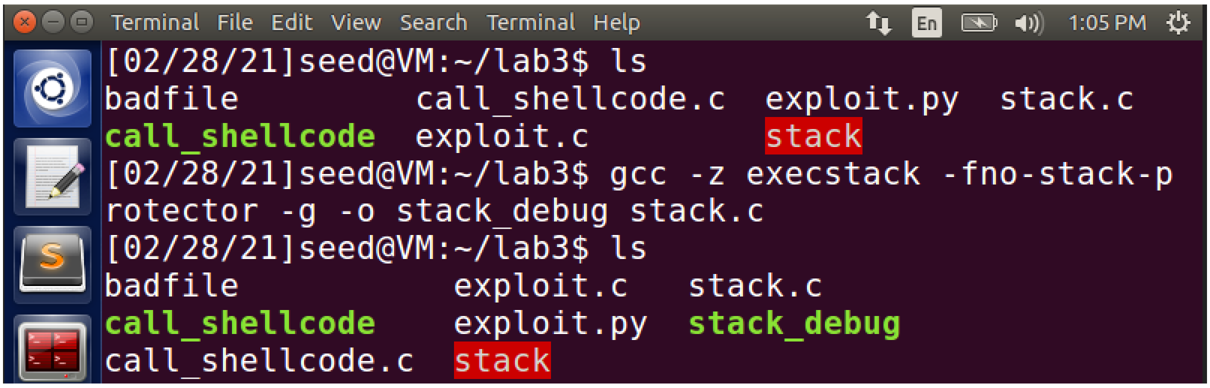


- Then, we provide root access to this executable stack file (indicated in green). On running the After providing root access, the file changes to color red, indicating that it has root access. On running the stack program, we can see that the vulnerable program has been run successfully and the output is displayed.

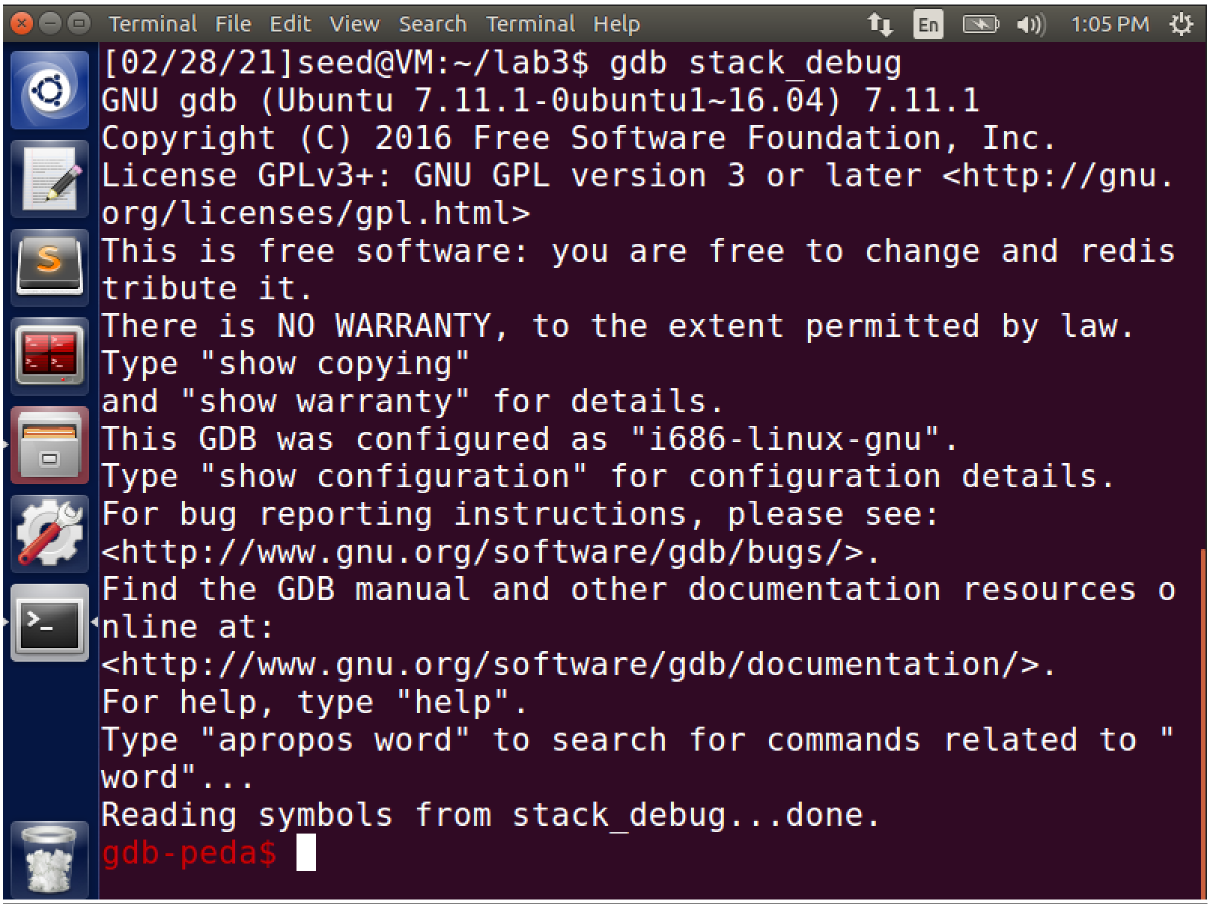


Task 2 – Exploiting the vulnerability:

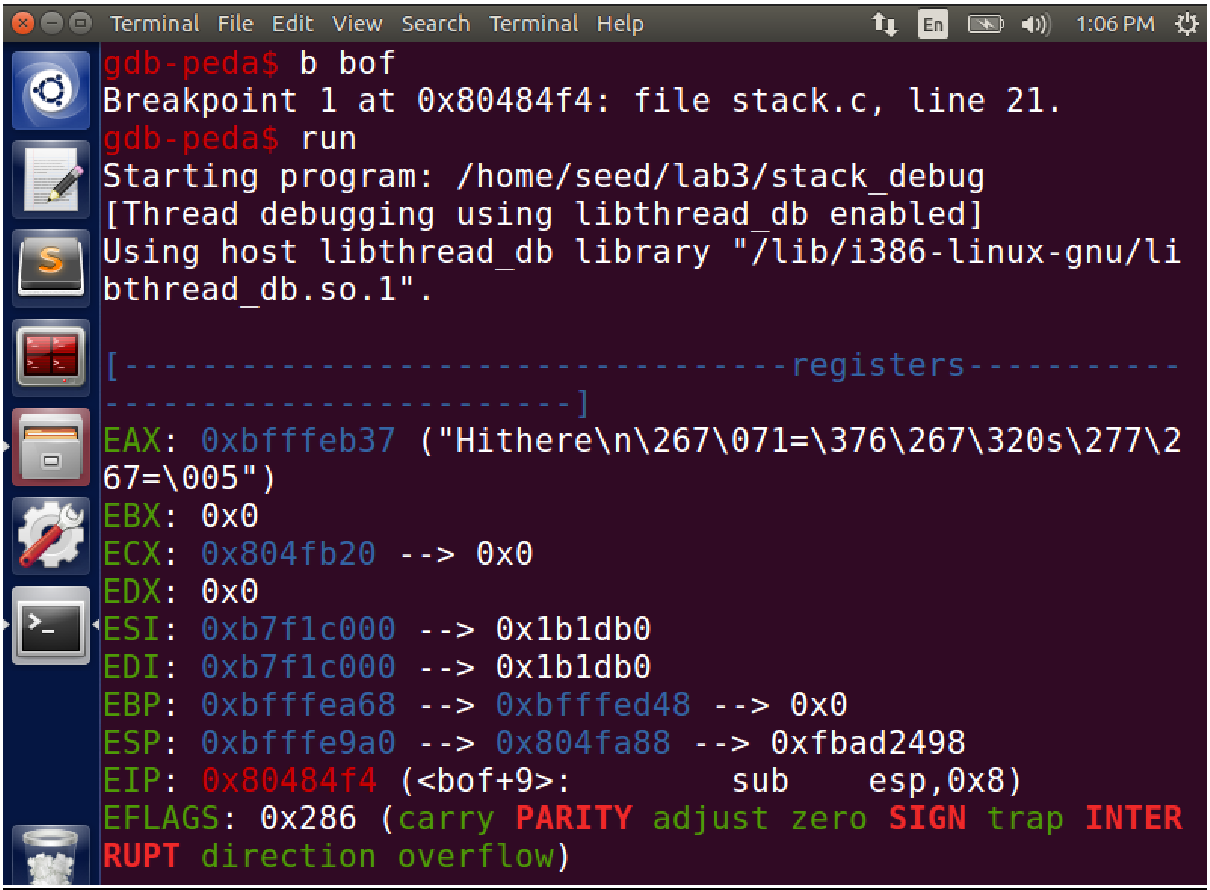
- The aim of this task is to construct contents of badfile to exploit the vulnerability. For that, we will have to find the program address in the memory. So, we first compile the program in debug mode, by using **execstack** to make the stack executable and **-g** command, also disabling the Stack Guard protector as shown in the following screenshot. The stack.c program is run using these commands to support it and stored into **stack_debug** file



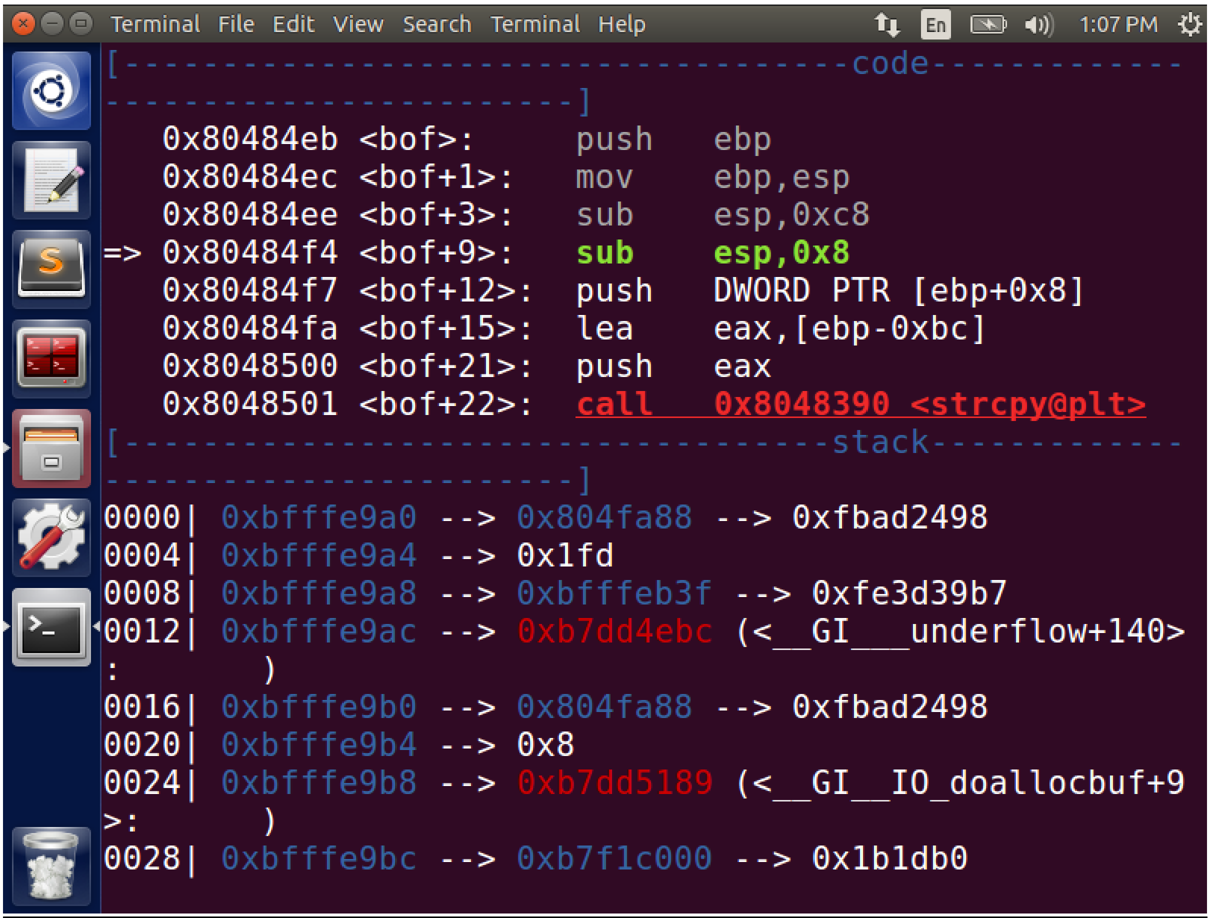
- We run this stack_debug in gdb mode to see what is going on inside this program during execution.



- Now, we add a breakpoint at Buffer Overflow function by using '**b bof**' command and run the program again.



Terminal File Edit View Search Terminal Help
1:06 PM
gdb-peda$ b bof
Breakpoint 1 at 0x80484f4: file stack.c, line 21.
gdb-peda$ run
Starting program: /home/seed/lab3/stack_debug
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".
[----------------------------------registers-----------------------------------]
EAX: 0xbfffeb37 ("Hithere\n\267\071=\376\267\320s\277\267=\005")
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffea68 --> 0xbfffed48 --> 0x0
ESP: 0xbfffe9a0 --> 0x804fa88 --> 0xfbad2498
EIP: 0x80484f4 (<bof+9>: sub esp,0x8)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)

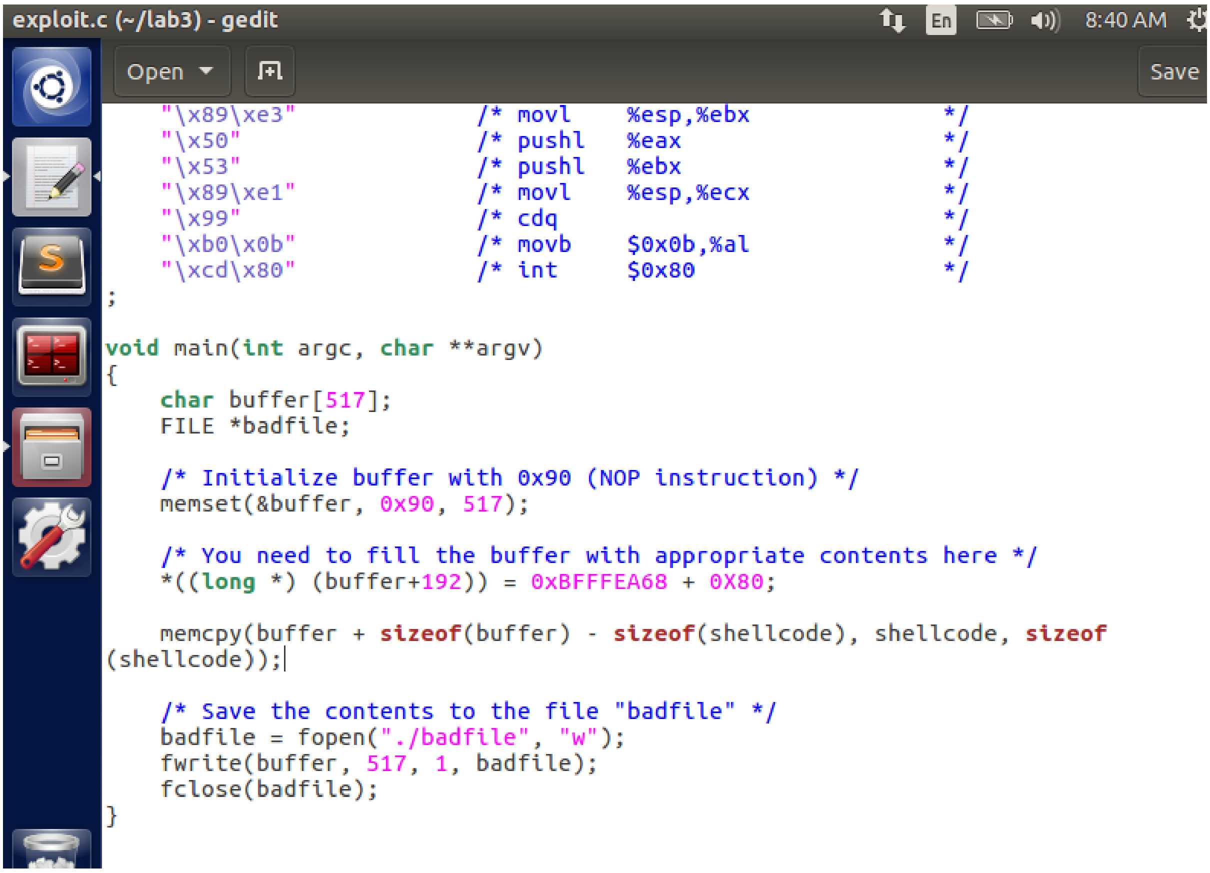


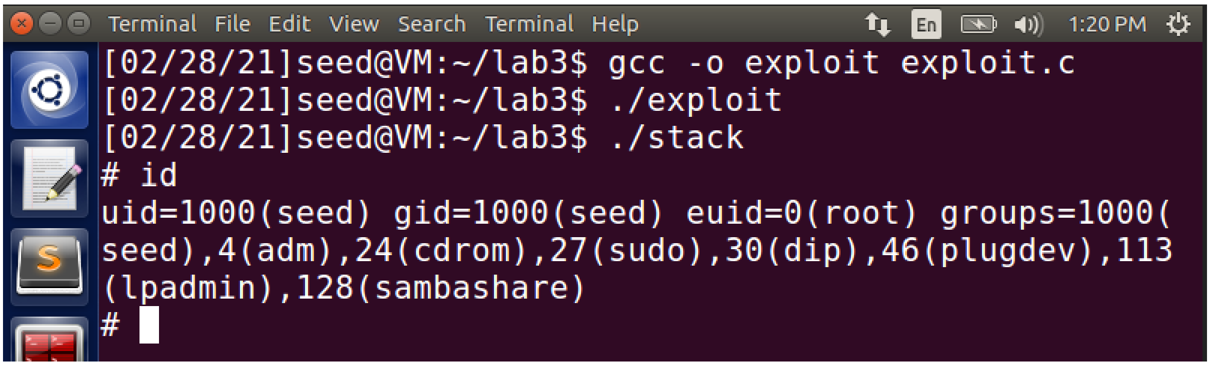
Terminal File Edit View Search Terminal Help
1:07 PM
[-------------------------------------code-------------------------------------]
0x80484eb <bof>: push ebp
0x80484ec <bof+1>: mov ebp,esp
0x80484ee <bof+3>: sub esp,0xc8
=> 0x80484f4 <bof+9>: sub esp,0x8
0x80484f7 <bof+12>: push DWORD PTR [ebp+0x8]
0x80484fa <bof+15>: lea eax,[ebp-0xbc]
0x8048500 <bof+21>: push eax
0x8048501 <bof+22>: call 0x8048390 <strcpy@plt>
[------------------------------------stack-------------------------------------]
0000| 0xbfffe9a0 --> 0x804fa88 --> 0xfbad2498
0004| 0xbfffe9a4 --> 0x1fd
0008| 0xbfffe9a8 --> 0xbfffeb3f --> 0xfe3d39b7
0012| 0xbfffe9ac --> 0xb7dd4ebc (<__GI___underflow+140>:)
0016| 0xbfffe9b0 --> 0x804fa88 --> 0xfbad2498
0020| 0xbfffe9b4 --> 0x8
0024| 0xbfffe9b8 --> 0xb7dd5189 (<__GI__IO_doallocbuf+9>:)
0028| 0xbfffe9bc --> 0xb7f1c000 --> 0x1b1db0

- Now that the program is inside bof function, we go ahead and find the address of **ebp** and **buffer**.
- **ebp** refers to the **base pointer of the current active instruction of the stack.**

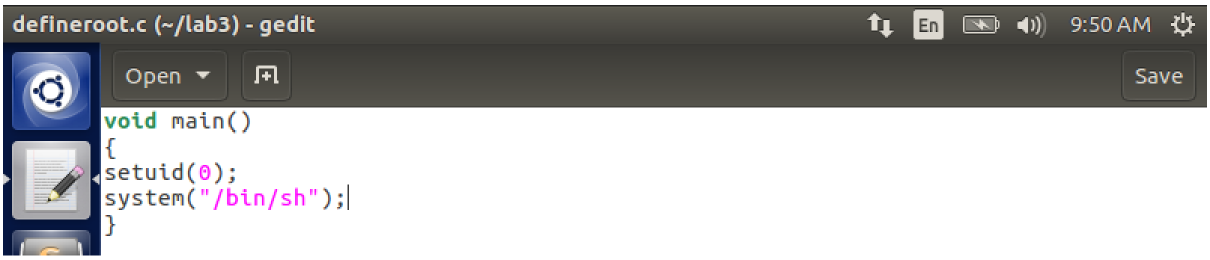
- The difference of ebp and buffer is found to get the address of the return value. We get the address value **0XBFFFEA68** and add 4 to the offset value **188**, which makes it **192**. Then,

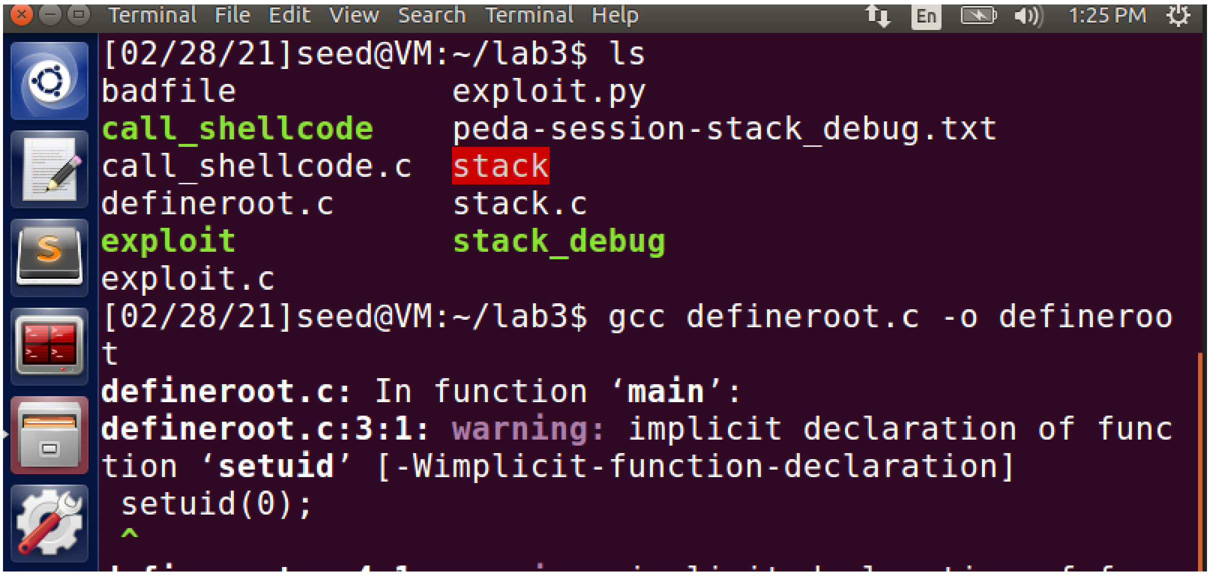
memcpy function is used, to copy bytes from source memory to destination memory. We update these details in the exploit.c program and try running the vulnerable stack.c program again.



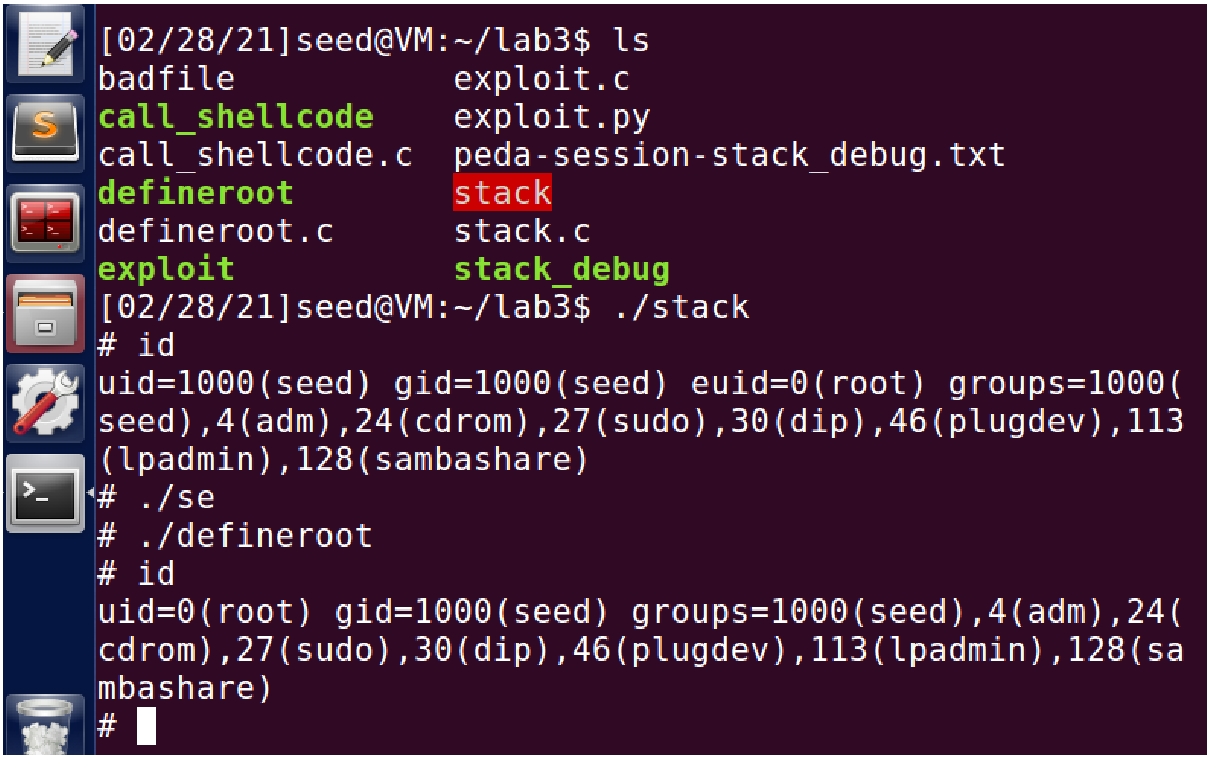


- On running the stack program, we see that we are able to access the stack, but don’t have root access on using ‘**id**’ command, which can be seen from **uid=1000(seed)**. To overcome this, we create our own root program **defineroot.c** with system() to try and access the root of the vulnerable stack program using this root.





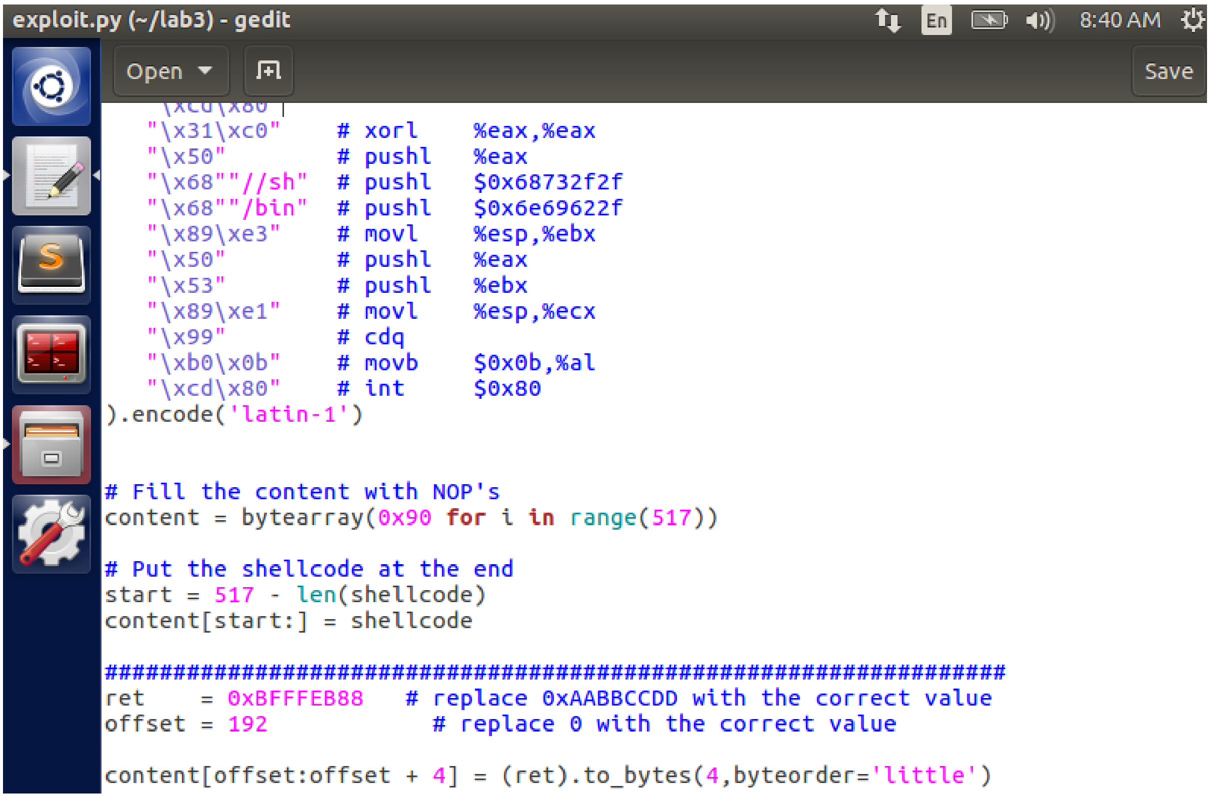
- Compiled the **defineroot.c** program and stored it in **defineroot** file.



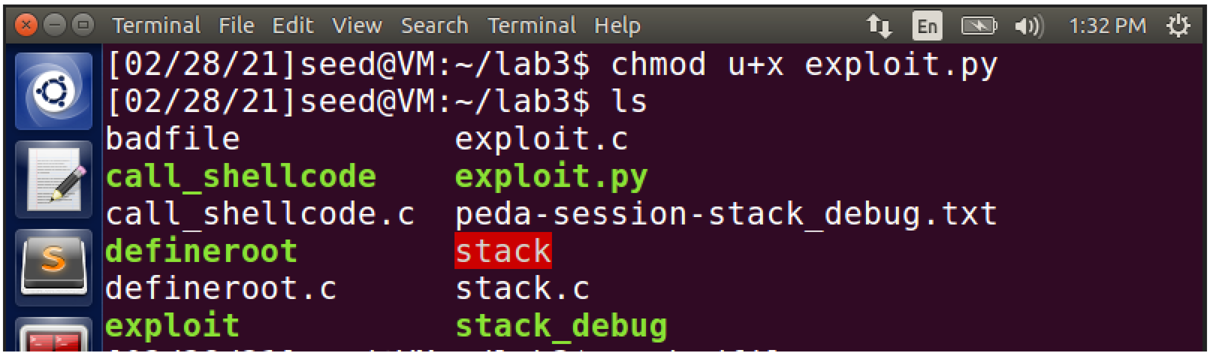
- On running the stack program and using **'./defineroot'** in the root terminal, we can see that we are able to gain access to root, which is shown by **uid=0(root)**

<u>Bonus – Using Python file exploit.py:</u>

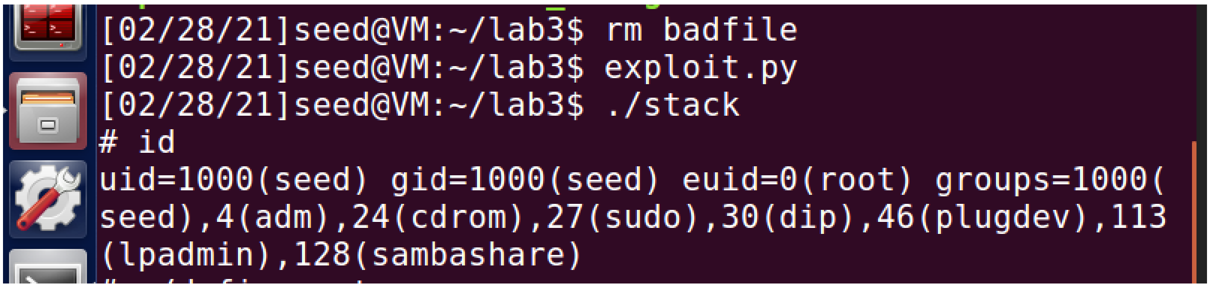
- We try to implement the same concept done using exploit.c but with exploit.y. Since we already know the buffer address and the offset, we update these values as shown in the below image.



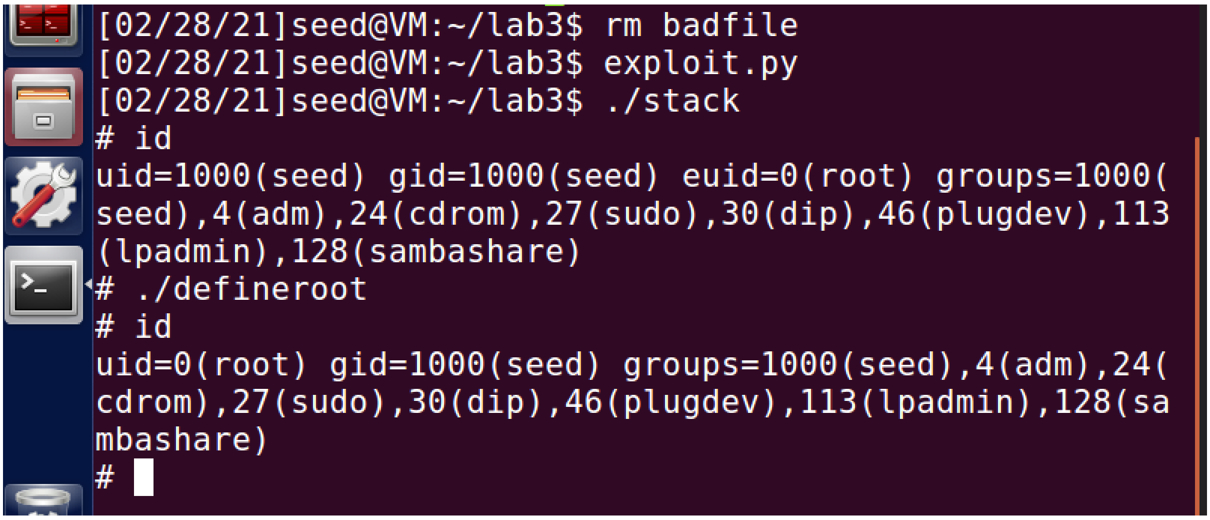
- We execute the exploit.py program using **'chmod u+x'** command. In that command, **chmod** is used to provide certain permissions, in this case u+x where **'u'** indicates the **user** and **'x'** indicates execute permission.



- On running the vulnerable stack file after executing the exploit.py file, we are able to get access to the stack as indicated by **'#'**, on using **'id'** command we can see that the **uid=1000** is still in seed.



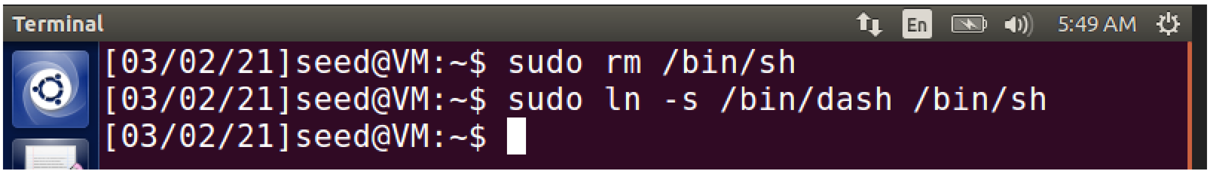
- Now, we try to run our own defineroot program which we have already compiled for the previous program. On using that, we are able to gain root access, which can be seen by uid=0(root) seen below.



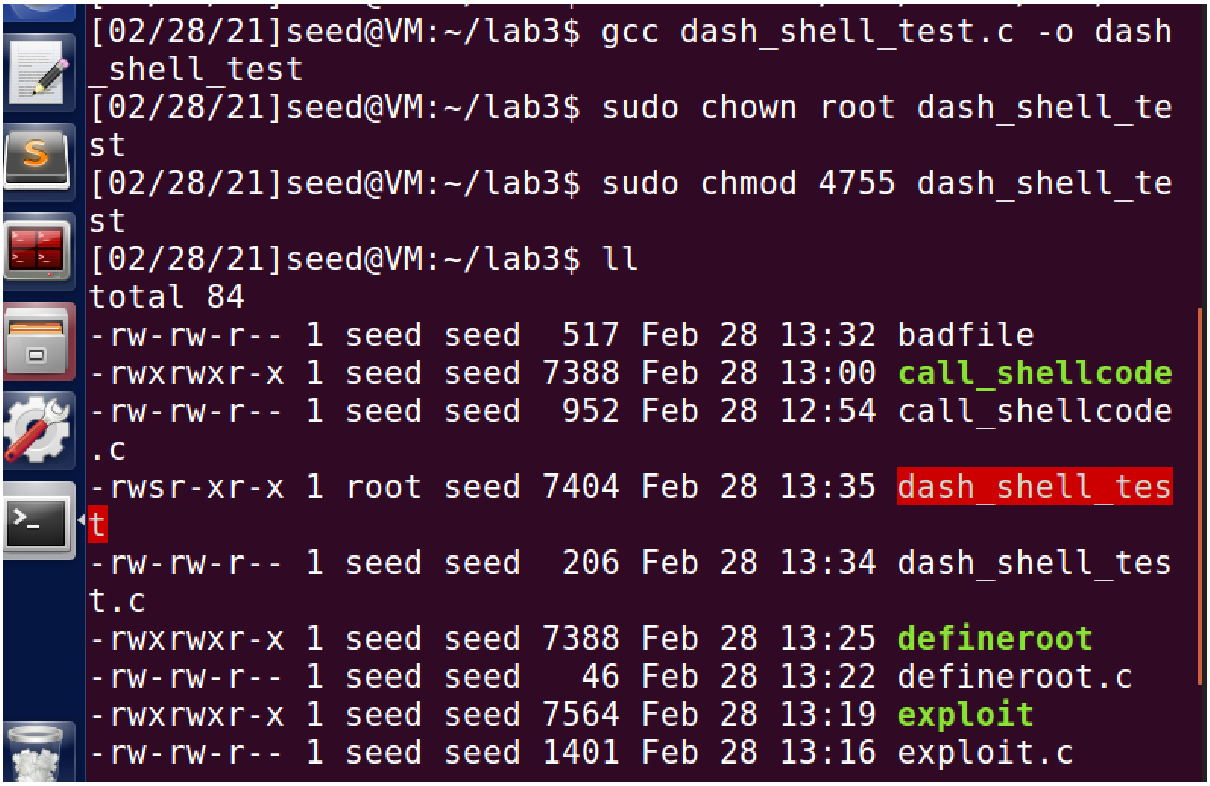
- Thus, we are able to exploit the vulnerability due to the successful buffer overflow attack.

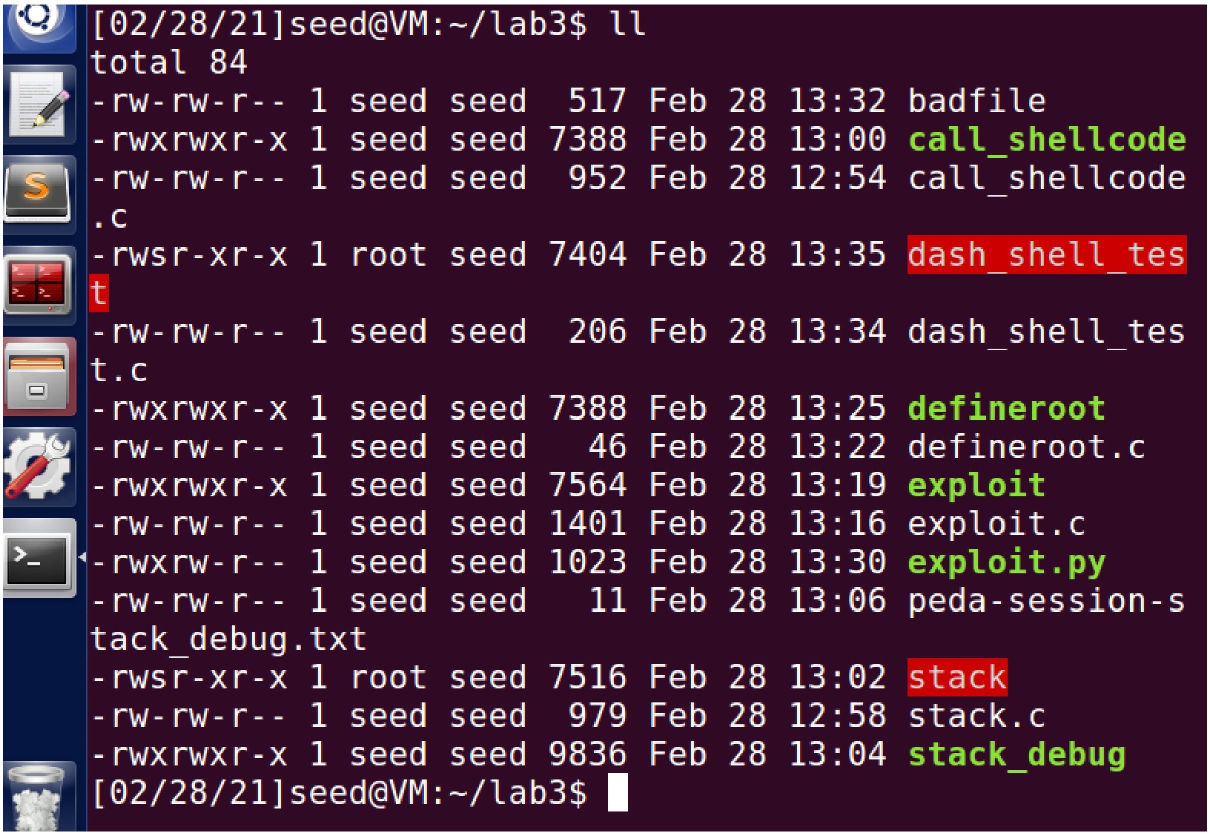
<u>Task 3 – Defeating dash's countermeasure:</u>

- In this task, we try to defeat the countermeasure implemented in dash by invoking another shell program using the system call **setuid(0)**.
- We first change bin symbolic link (sh) to make it to point to dash.

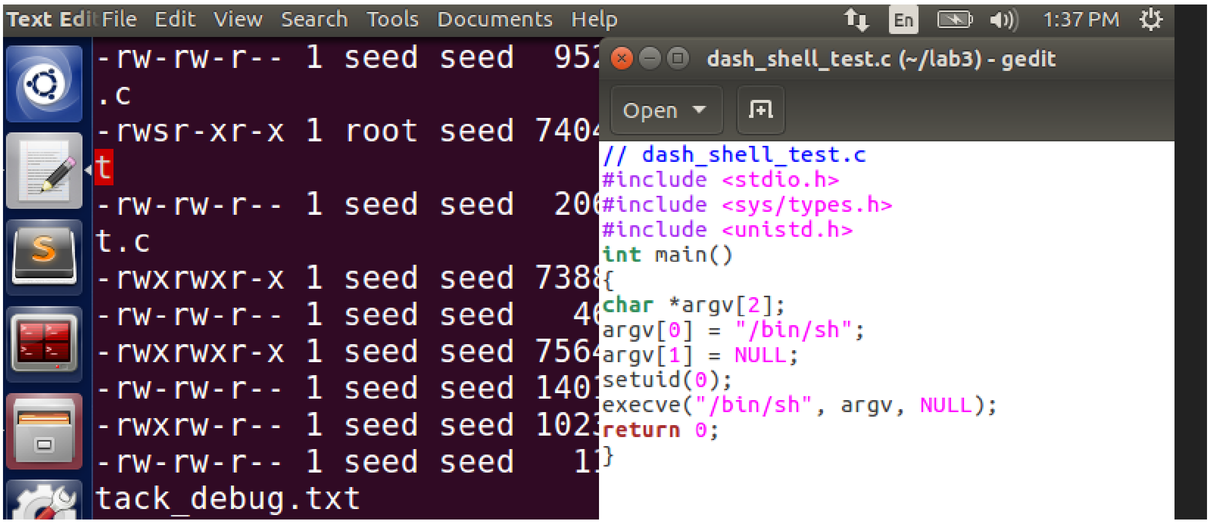


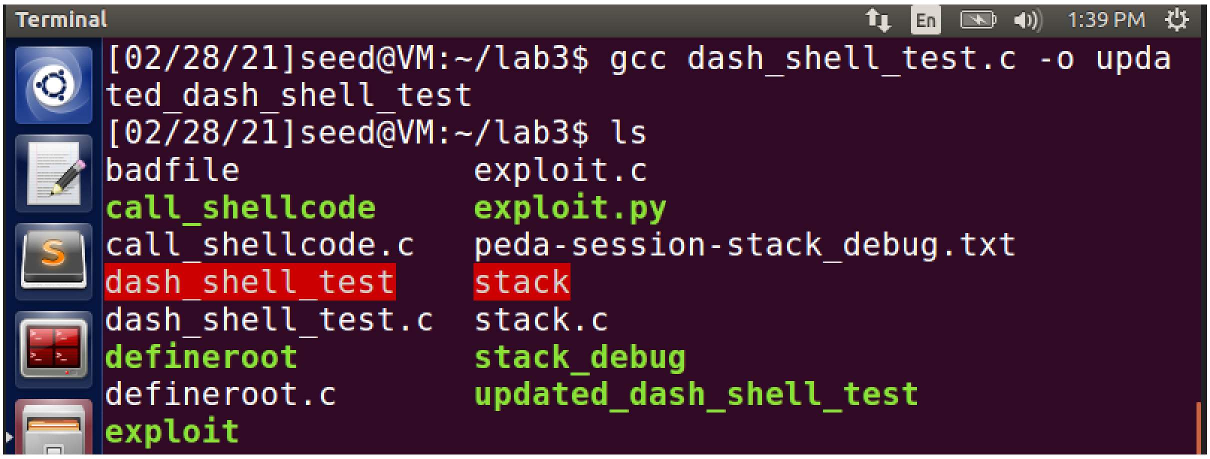
- Then, the **dash_shell_test.c** program is compiled and stored in **dash_shell_test** file. Root access is given to the **dash_shell_test** file to access it, by using **chown** and **chmod** commands.



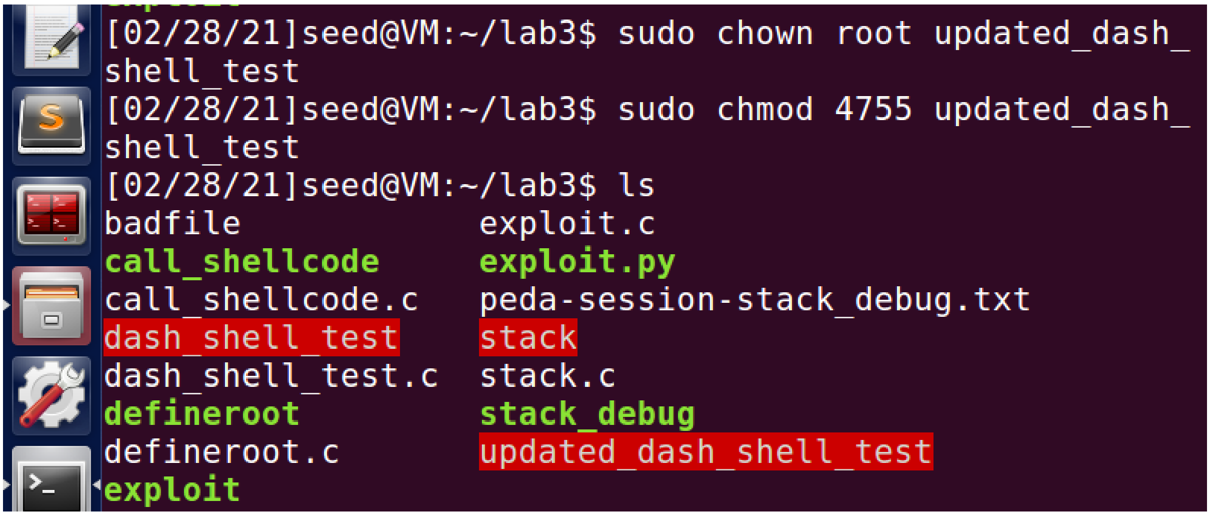


- Then, we try to execute dash_shell_test file after giving root access. But on execution, we see that **uid=1000(seed)** indicating that we do not have root access.
- Now, we uncomment the **setuid(0)** command and re-compile the dash_shell_test program and store it in a different file.

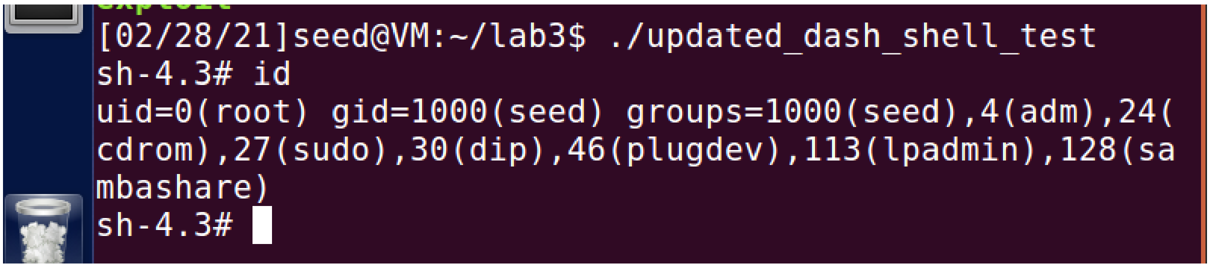




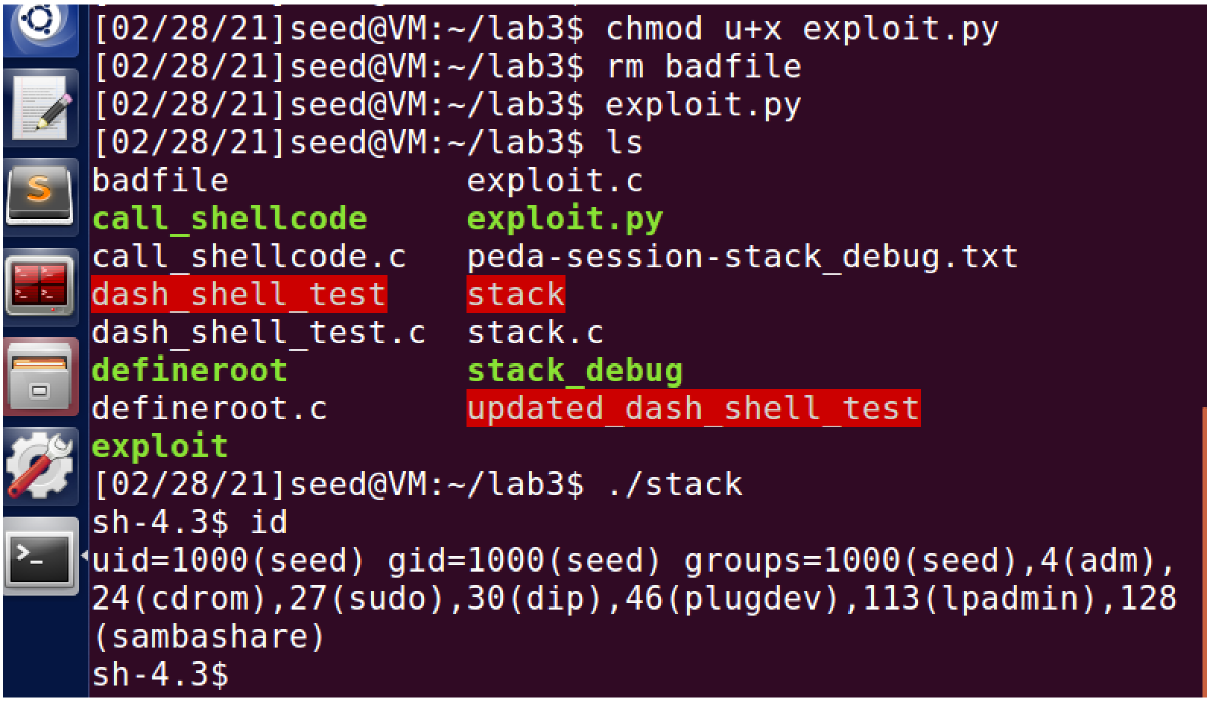
- Root access is given to the updated compiled file using **chown** and **chmod** commands.



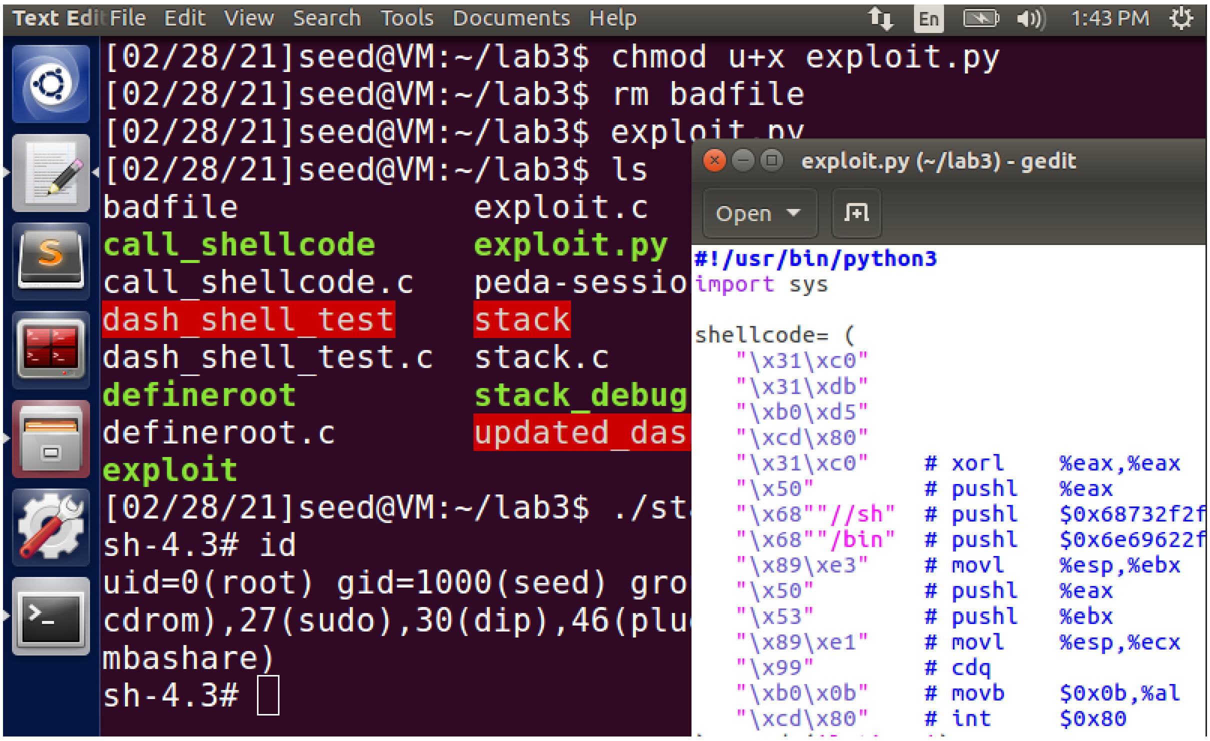
- The updated file has root access, indicated by color red. And later, we run the updated file.



- As seen above, we are able to access the shell, indicated by '#' and we see that we are able to gain root access shown by **uid=0(root).**
- In both the cases, we were able to access the shell, but in the first case, because the bash program dropped root privileges, we were not able to get root access privileges when setuid(0) was commented out. In this case, **Set-UID was different from effective-UID,** thus preventing access.
- In the second case, since we uncomment setuid(0), **Set-UID is same as that of effective-UID** and therefore bash program did not have to drop any privileges and we gain root access. **This system command setuid() is used here to defeat dash's countermeasure and successfully perform the attack.**
- We now replicate the same processes, but with exploit.py file as shown below.



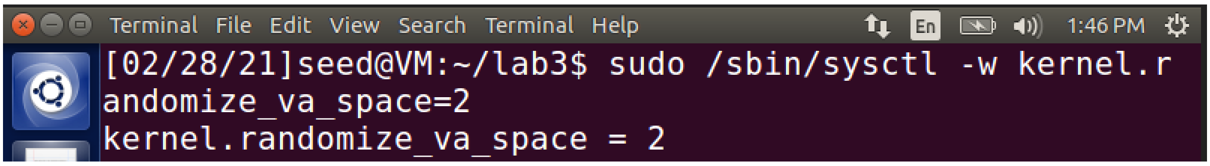
- We don't have root access as seen above because there were no necessary changes made to the exploit.py file for the bash program to retain privileges. So, we make the necessary updates to the exploit.py file as given in the below image.



- We add the assembly code in the python program before invoking execve() to perform the same function as setuid(0) and construct the badfile. On running the stack program, we can see above that we were able to get to root, indicated by **uid-0(root),** thus overcoming dash's countermeasure.

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

- Address Space Randomization is done to stop randomizing the starting address of heap and stack. This makes guessing the exact address difficult, thereby making the buffer-overflow attack also difficult. Here, it is set to 2 as seen from the below image, meaning both Stack and Heap starting addresses are randomized.



- Now, we try to run the vulnerable stack file developed in Task 2, seen below.

```
[02/28/21]seed@VM:~/lab3$ ls
badfile              exploit.c
call_shellcode       exploit.py
call_shellcode.c     peda-session-stack_debug.txt
dash_shell_test      stack
dash_shell_test.c    stack.c
defineroot           stack_debug
defineroot.c         updated_dash_shell_test
exploit
[02/28/21]seed@VM:~/lab3$ ./stack
Segmentation fault
[02/28/21]seed@VM:~/lab3$
```

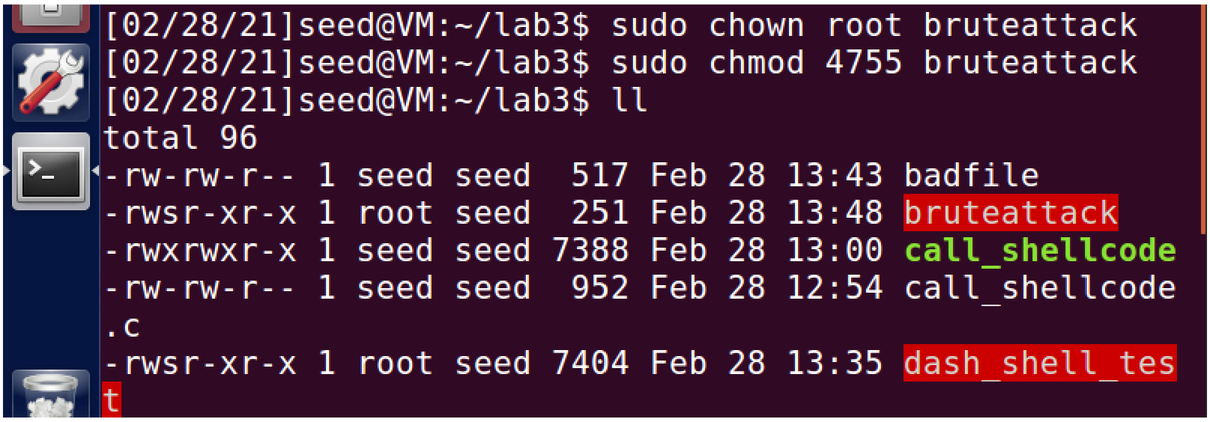
- As seen above, we get **Segmentation Fault** error, showing that the attack is not successful. To access the stack and launch an attack, we use bruteforce attack approach. The below shell script code is used to run the vulnerable program, in an infinite loop.

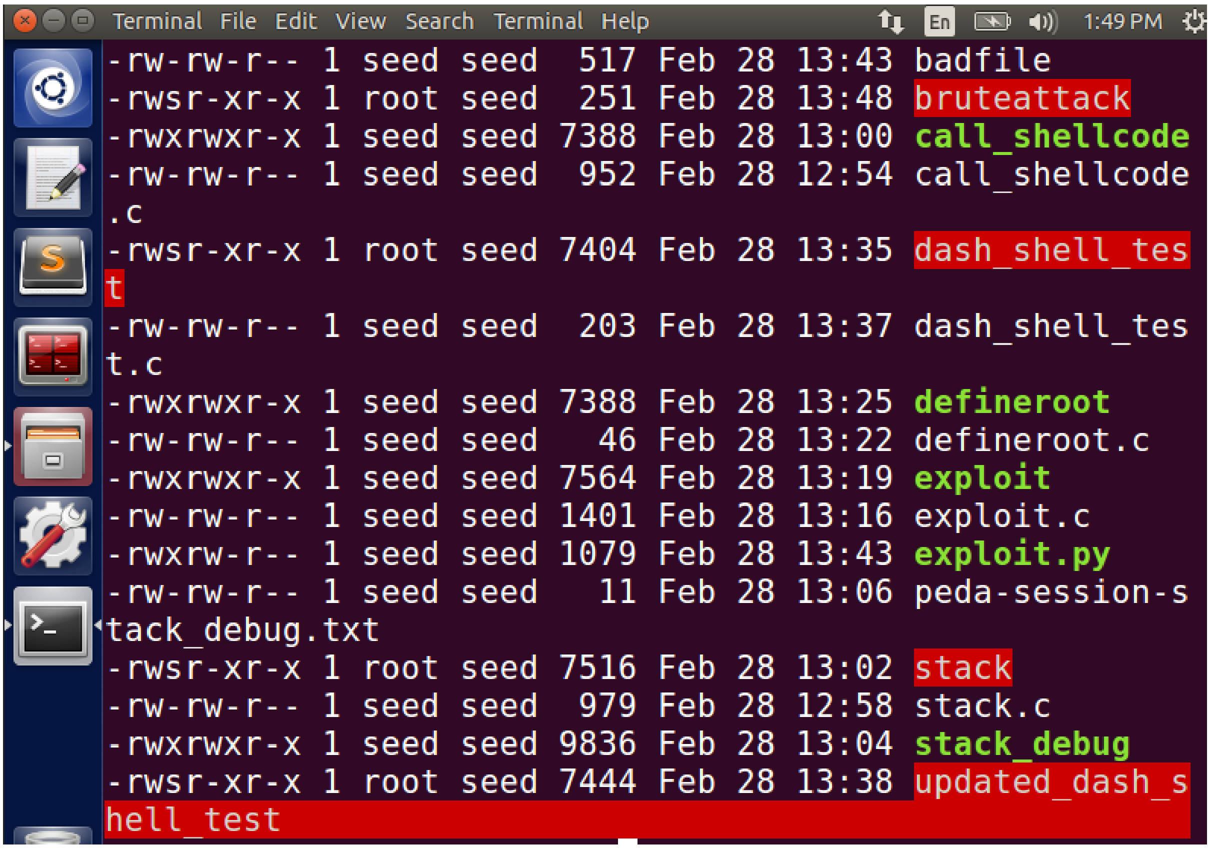
```
#!/bin/bash
SECONDS=0
value=0
while [ 1 ]
do
value=$(( $value + 1 ))
duration=$SECONDS
min=$(($duration / 60))
sec=$(($duration % 60))
echo "$min minutes and $sec seconds elapsed."
echo "The program has been running $value times so far."
./stack
done
```

- The above shell code is stored in a file named **bruteattack**, which will be used to try and see if we are able to access the stack.

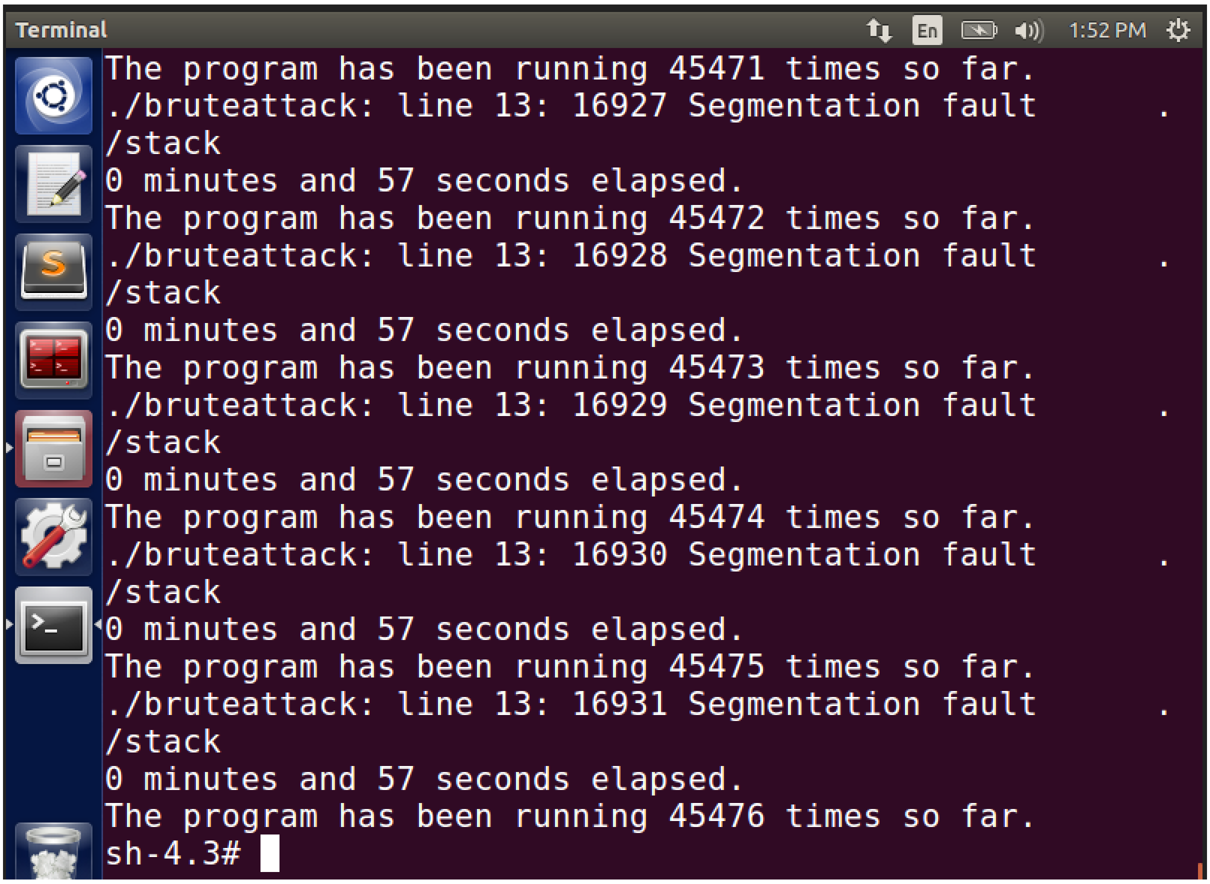
```
[02/28/21]seed@VM:~/lab3$ ls
badfile              exploit
bruteattack          exploit.c
call_shellcode       exploit.py
call_shellcode.c     peda-session-stack_debug.txt
dash_shell_test      stack
dash_shell_test.c    stack.c
defineroot           stack_debug
defineroot.c         updated_dash_shell_test
```

- Root access is then provided to the **bruteattack** file to see if we are able to access the stack by running the program in a loop.

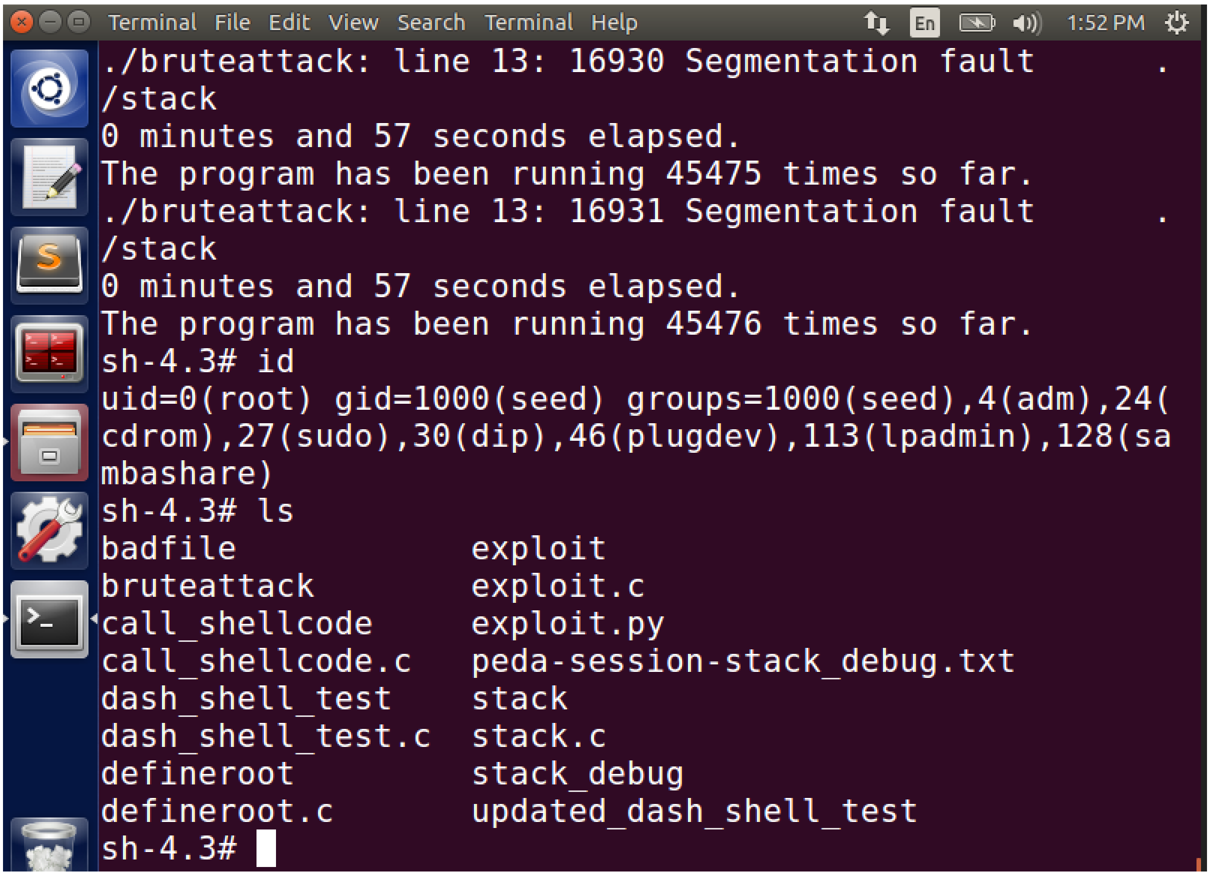




- The program runs infinitely until the point it is able to access the stack. Finally, it accesses the stack, which is indicated by the ‘#’ symbol.



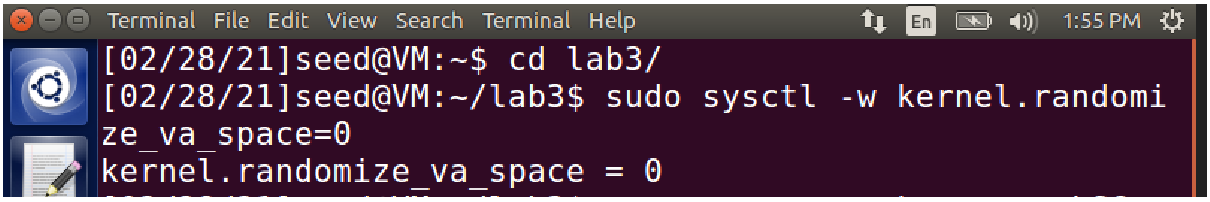
- On running the '**id**' command, we see that we have gained root access, as shown by '**uid=0(root)**' indicating that the attack is successful in this case.



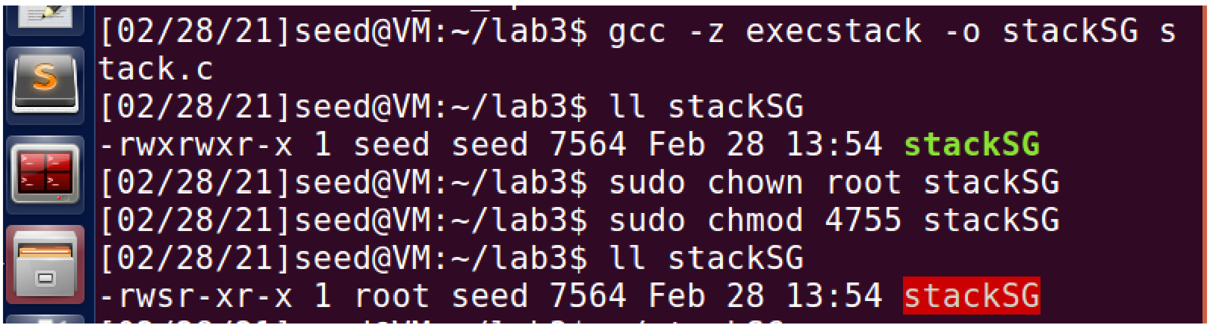
- This is because, since Address Space Randomization was done at the beginning of the task, address would be randomized for the stack and therefore it is not possible to guess the address of the stack. The only way is to run a brute force attack and repeatedly try to find out the address by launching the attack in loop, until the point where the program gets access to the stack.

<u>Task 5 – Turn on the Stack Guard Protection:</u>

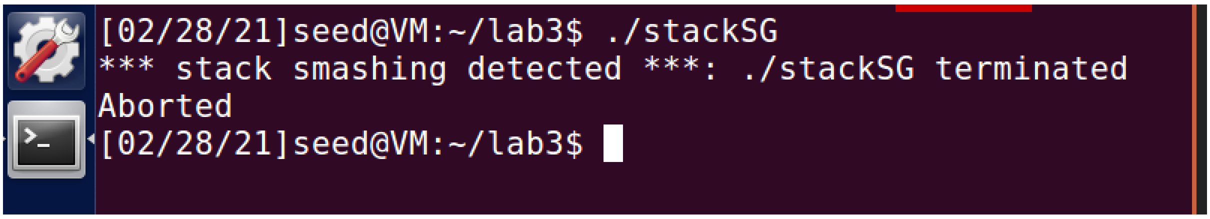
- The address space randomization is turned off initially for this task, using the command below;



- Task 1 is run again, where the vulnerable stack.c code is recompiled with GCC StackGuard. Root access (indicated in red) is given to that compiled file to have access to run the compiled file.



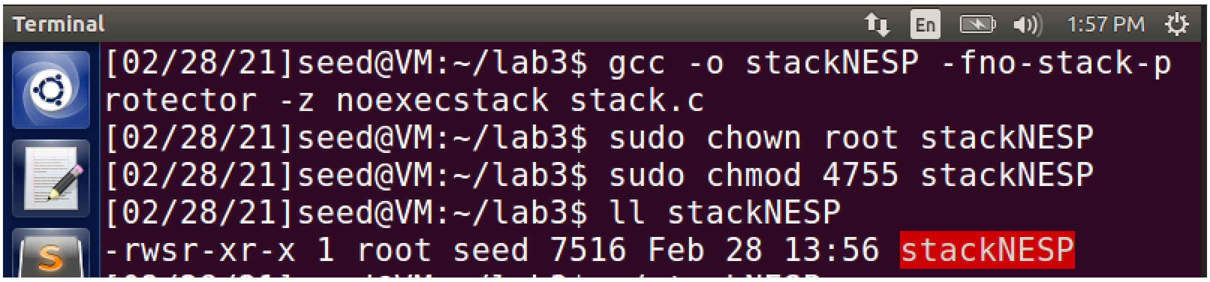
- On running the stacks program, we see that the program execution gets terminated and "**Aborted**" is displayed.



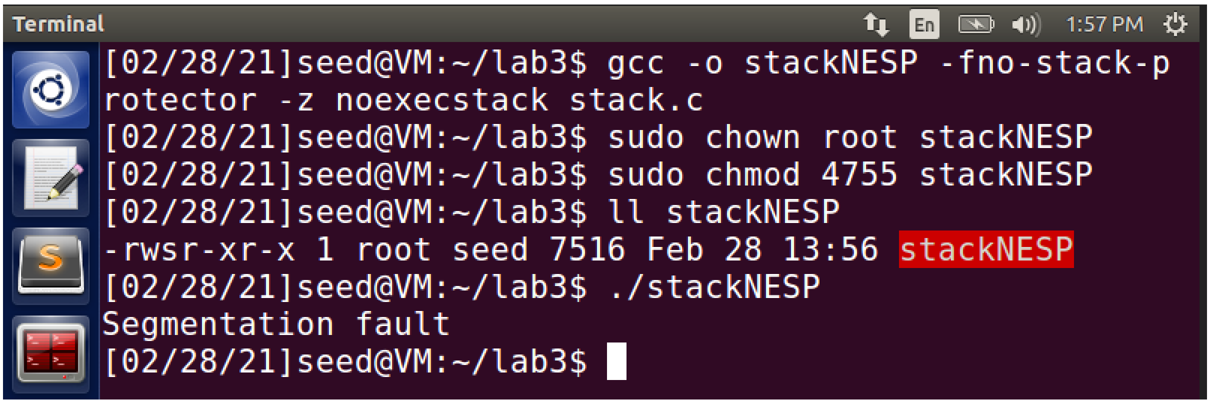
- This is because, since we have Stack Guard protector on for this program, it detects that there is a Buffer Overflow issue with the vulnerable program and prevents the execution of it by aborting the execution. This proves that, **with Stack Guard protector we can detect and prevent Buffer Overflow issues**.

<u>Task 6 – Turn on the Non-Executable Stack Protection:</u>

- In the previous task, address space randomization was already turned off. It is kept off for this task as well. The same stack.c vulnerable program is compiled again by using '**-z nonexecstack'** (non-executable stack)



- It is compiled and stored in a file names **stackNESP.** Root access is provided to this file to later run the file, which is indicated by RED.
- Then the file is run. On running the file, we get **Segmentation Fault error.**



- When trying to launch a buffer overflow attack with a vulnerable program, which could potentially get us root access, **it throws an error because the stack is made non-executable, using -z noexecstack.** Therefore, the attack fails in this case, unlike earlier where we were able to access the stack.