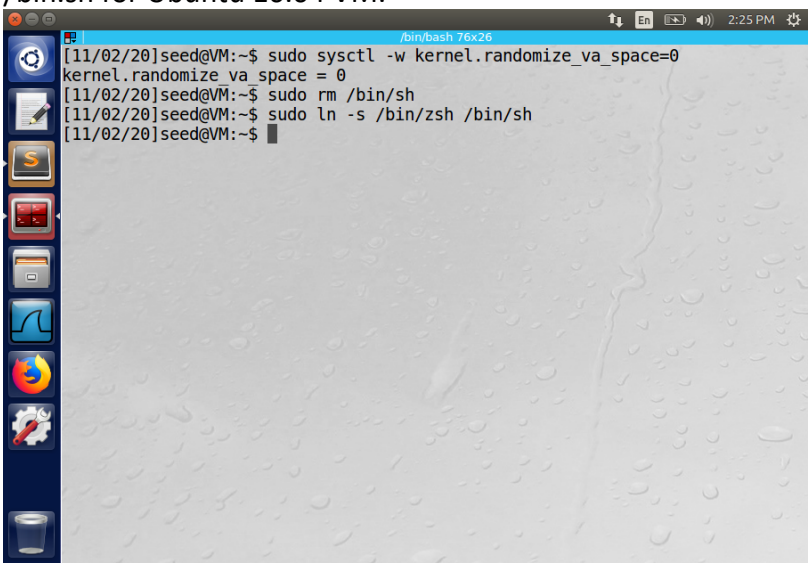
Man Tik Li
CS 377 - 001
November 2, 2020

Lab 3

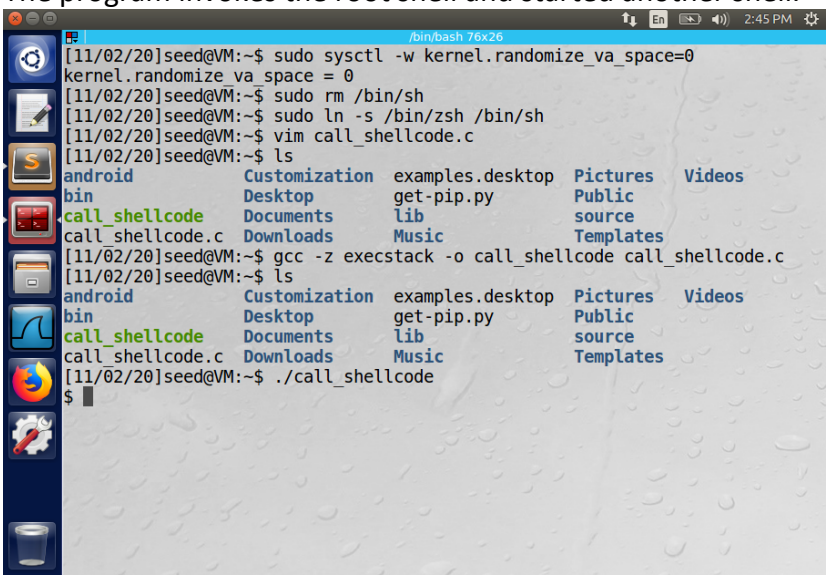
Task 1

2.1 Turning off Countermeasures – Disable address space randomization and configuring /bin.sh for Ubuntu 16.04 VM.



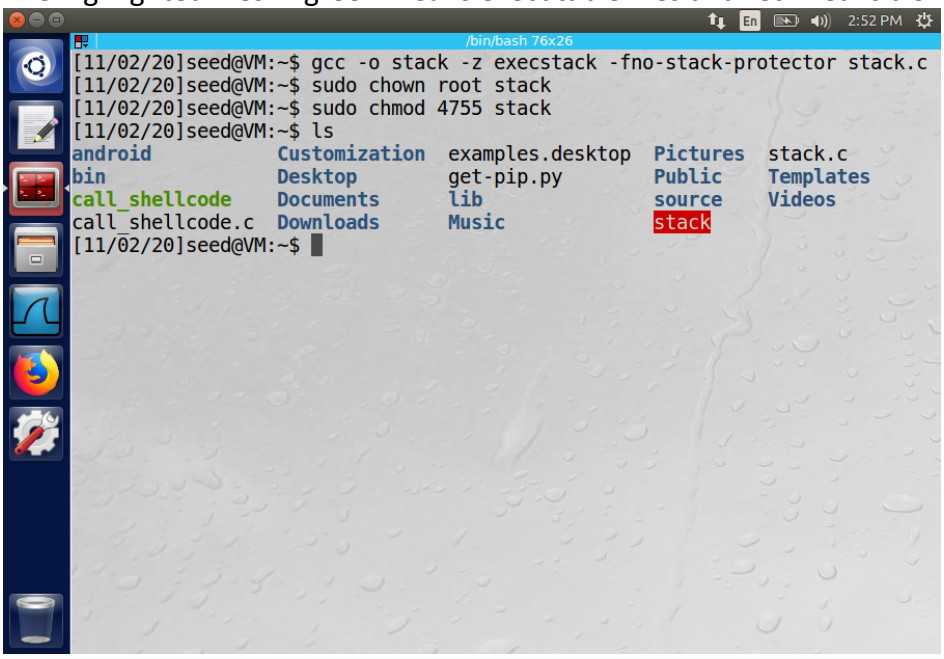
2.2 Running Shellcode

The program invokes the root shell and started another shell.

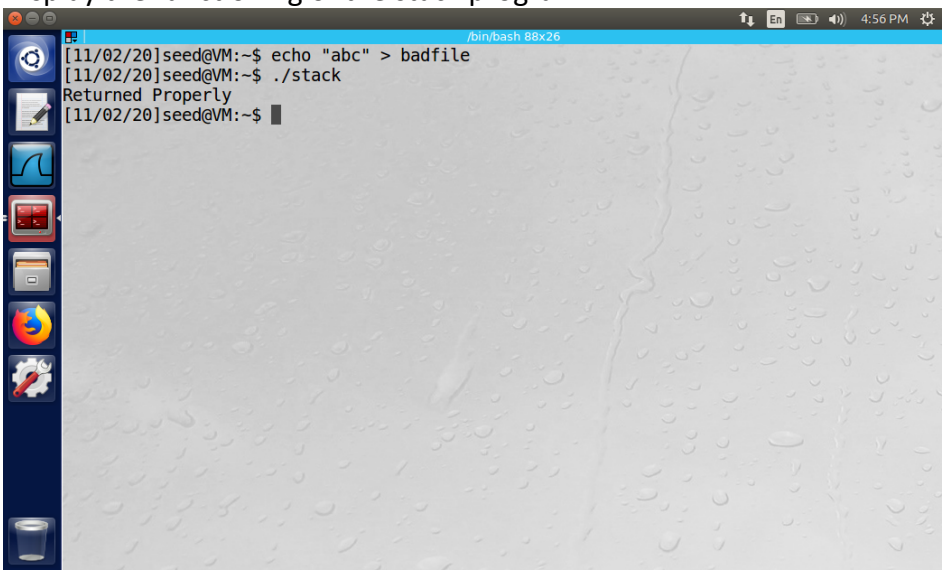


2.3 The Vulnerable Program

The highlighted files in green means executable files and red means a SET_UID program.



Display the functioning of the stack program.

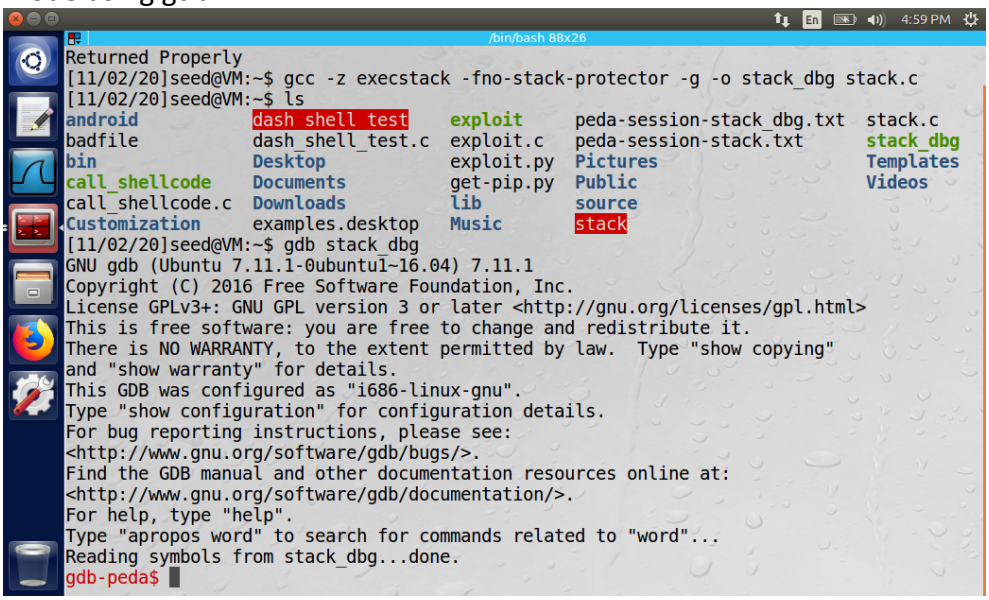


Display the functioning of the stack program.

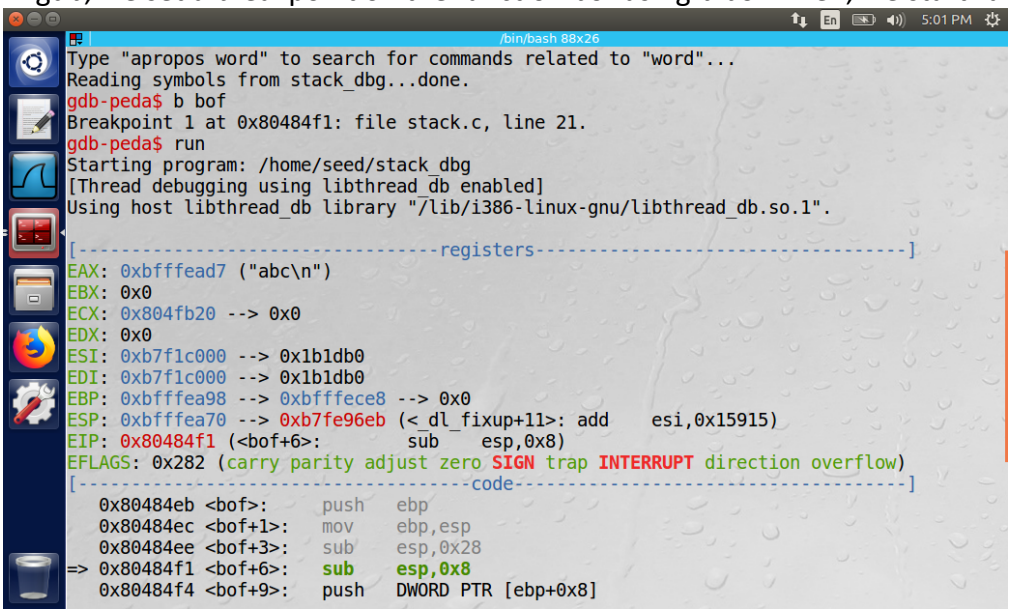
Task 2

2.4 Exploiting the Vulnerability

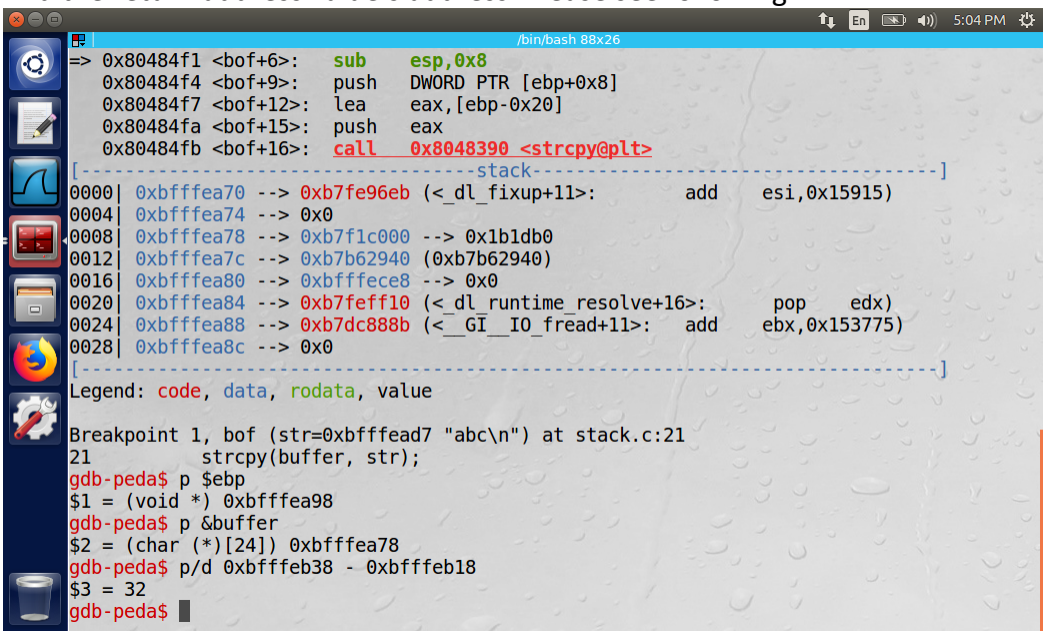
First, compile the program in the debug mode with -g option. We run the program in debug mode using gdb:



In gdb, we set a breakpoint on the function bof using b bof. Then, we start run the program:



The program stops inside the function bod due to the breakpoint we declared. The stack frame values for this function will be used to construct badfile contents. First, we print the ebp and buffer values. We also find the difference between the ebp and start of the buffer in order to find the return address value's address. Please see following:



We can see 0xbfffeb38 is our frame pointer. Hence the return address must be stored at 0xbfffeb38 + 4. The first address is 0xbfffeb38 + 8. The location that stores the return address was the difference between the return address and the buffer start address. Therefore, the input is copied to the buffer from the start. The output showed the difference between ebp and buffer start. We can find the return address by the stack layout, which is 4 bytes above where the ebp points. Finally, the distance between the return address and the start of the buffer is 36. The return address of badfile is 36.

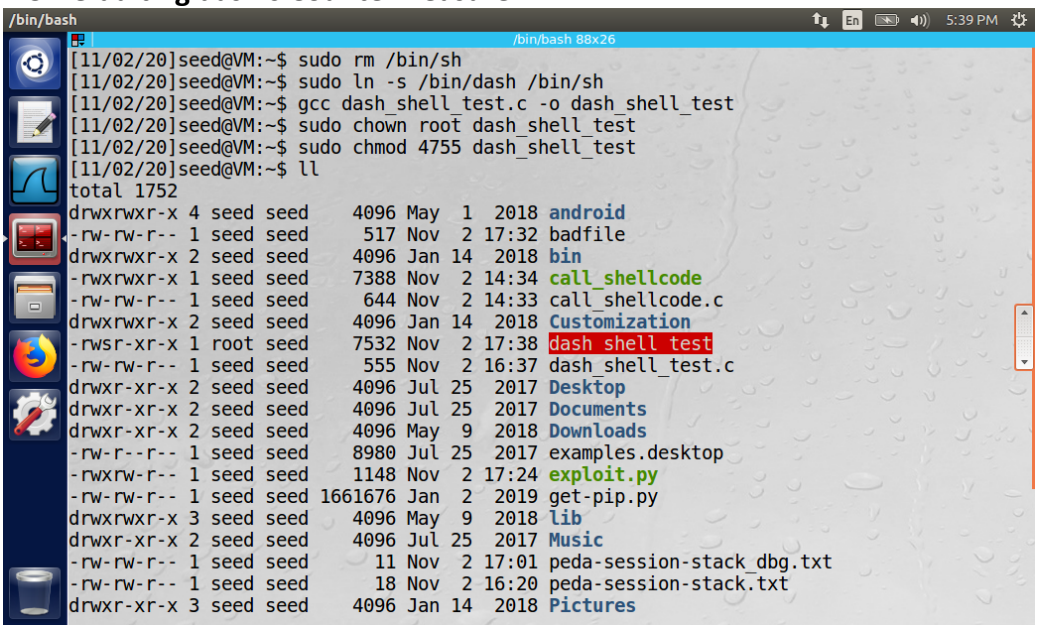
exploit.py:

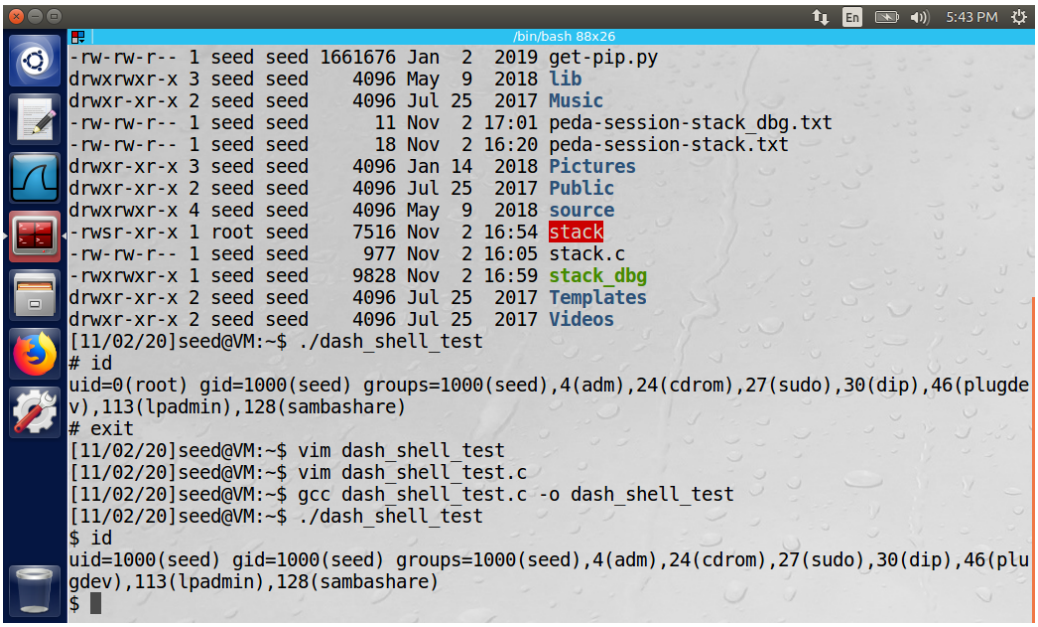

```
[11/02/20]seed@VM:~$ vim exploit.py
[11/02/20]seed@VM:~$ chmod u+x exploit.py
[11/02/20]seed@VM:~$ ls
android  call_shellcode    dash_shell_test  Documents         exploit.py  Music                      Pictures  stack      Templates
badfile  call_shellcode.c  dash_shell_test.c  Downloads       get-pip.py  peda-session-stack_dbg.txt  Public    stack.c    Videos
bin      Customization     Desktop          examples.desktop  lib         peda-session-stack.txt     source    stack_dbg
[11/02/20]seed@VM:~$ rm badfile
[11/02/20]seed@VM:~$ exploit.py
[11/02/20]seed@VM:~$ ./stack
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$
```

We have successfully performed the buffer overflow attack and gained root privileges.

Task 3

2.5 Defaulting dash's Countermeasure





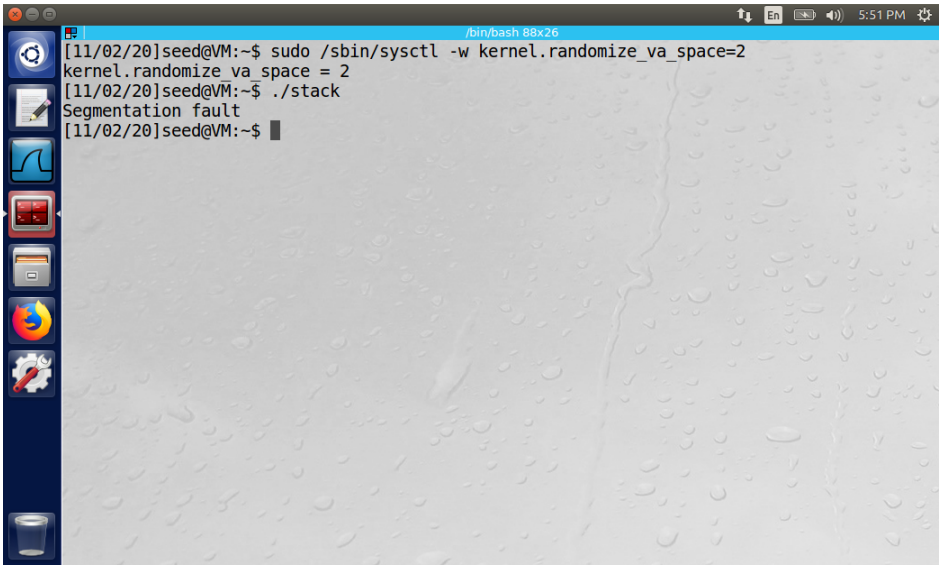
When comment and uncomment the setuid, the user ID is different. User will be normal user when comment the setuid(0) and became root when uncomment. The cause is because the bash program drops the provileges of the setuid program since the effective user id and the actual user are not the same. Therefore, it executed the program with normal priveleges and not root. When having the setuid in the program, it changed because the actual user id is set to that of root, and the effective user id is 0 due to the setuid program.

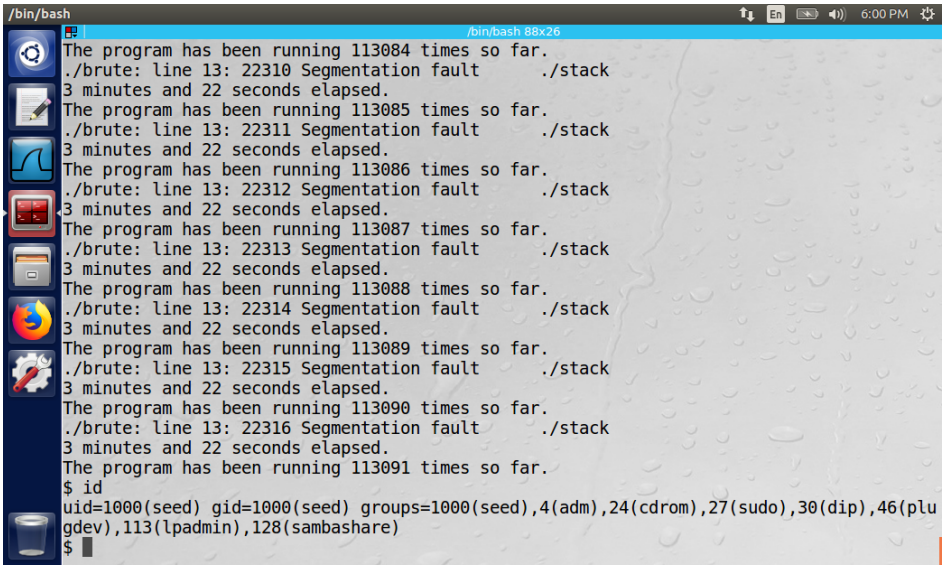
When perform buffer overflow attack, we did in task 2, but with /bin/dash countermeasure for setuid programs is appear due to the symbolic link from /bin/sh to /bin/dash.

Task 4

2.6 Defaulting Address Randomization

We enable address randomization for both stack and heap by setting the value to 2. When we try on running the same attack as in task, we receive segmentation fault, which shows that the attack was not successful.

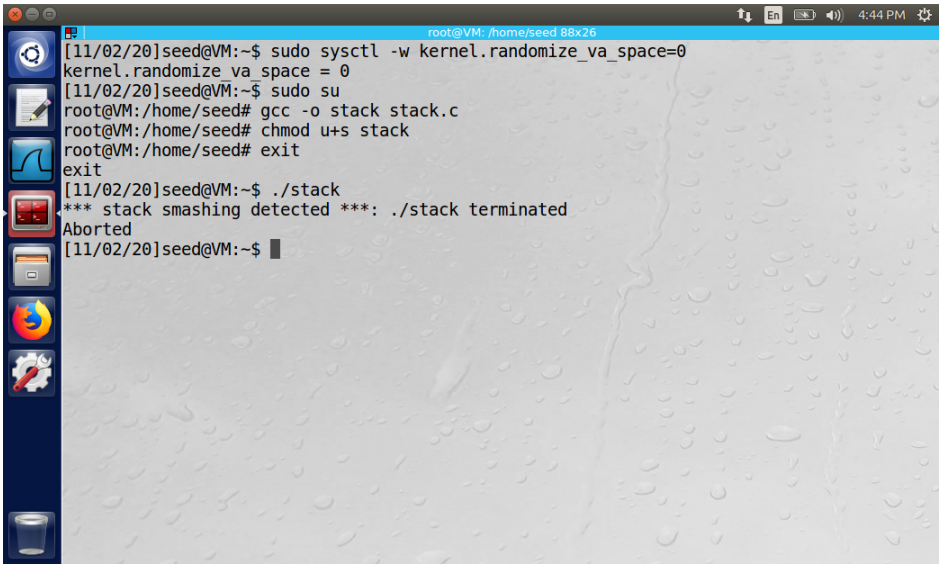




The output shows the time taken and the attempts taken to perform the attackwith address Randomization and brue-force approach. The stack frame always started from the sasme memory point for each program for simplicy purpose.

Task 5

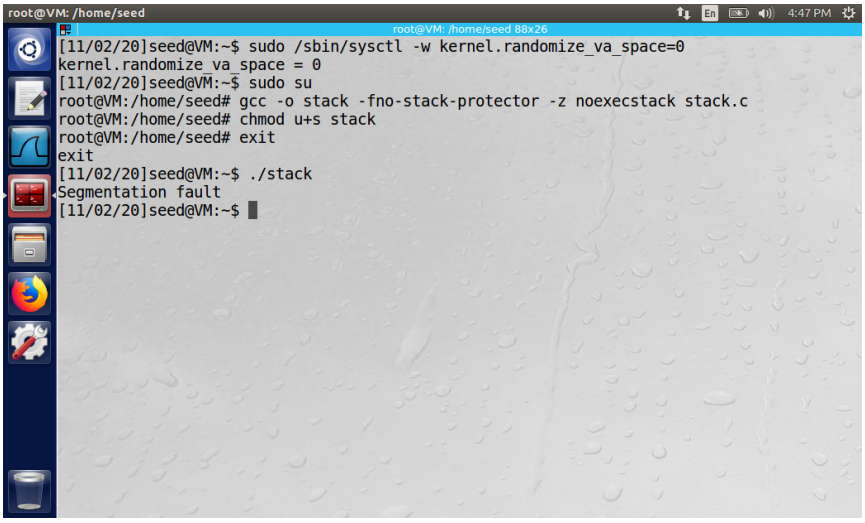
2.7 – Turn on the StackGuard Protection



Due to the StackGuard Protection mechanism, Buffer Overflow attack can be detected and being blocked.

Task 6

2.8 – Turn on the Non-executale Stack Protection



The stack causes the segmentation fault; it is no more executable. When we attempt to attack, we can run a program quickly with root access. The program stored in stack, and when we try to enter a return address that points to that malicious program. The stack stackmemory layout specifies the local variables and arguments that stores in it. But all of these values will not require any execution; hence there is not necessary to have the stack as executable. The normal user can still run the program without side effects, but the malicious code will also be considered data rather than the program itself. Our attack fails because the stack was not executable.