CSE: 5382-001: SECURE PROGRAMMING

ASSIGNMENT 4

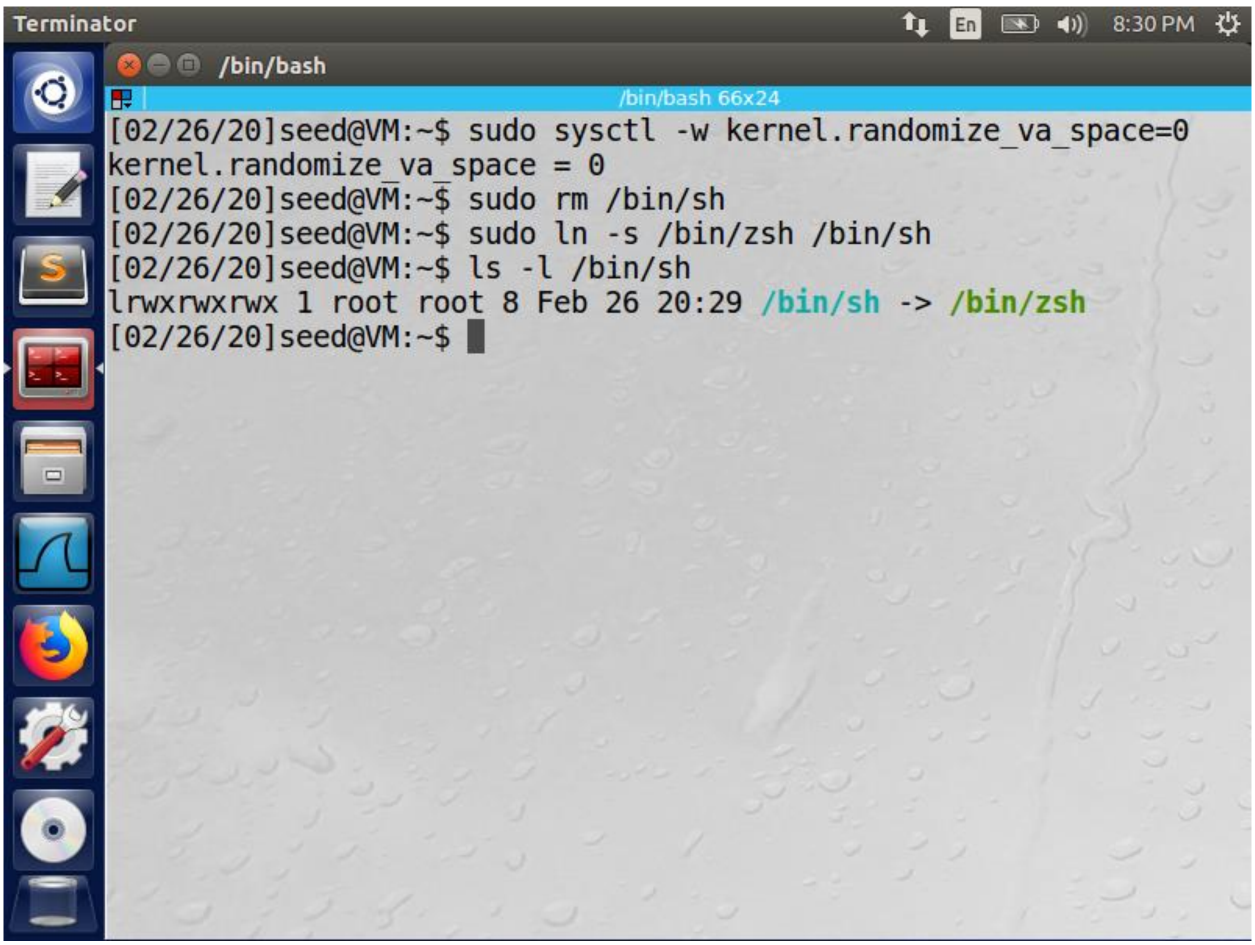
Tharoon T Thiagarajan

1001704601

2.1 Turning Off Countermeasures

Output:

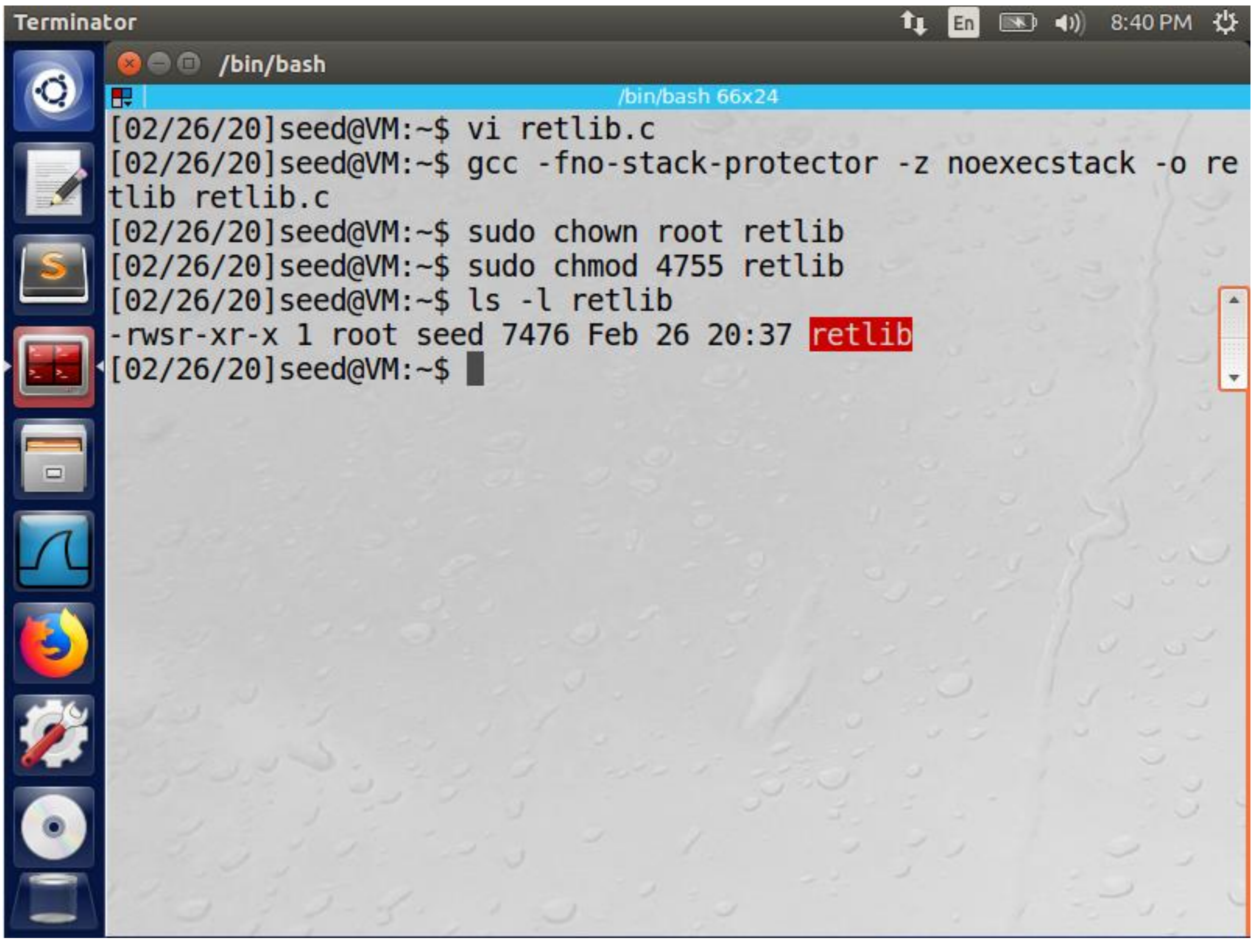
Before starting the assignment with Return-to-libc attacks we first disable the address space randomization of the linux system using the command sudo sysctl -w kernel.randomize_va_space=0. This does not the randomize the starting address of the heap and the stack. I also created the symbolic link to 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.



2.2 The Vulnerable Program

Output:

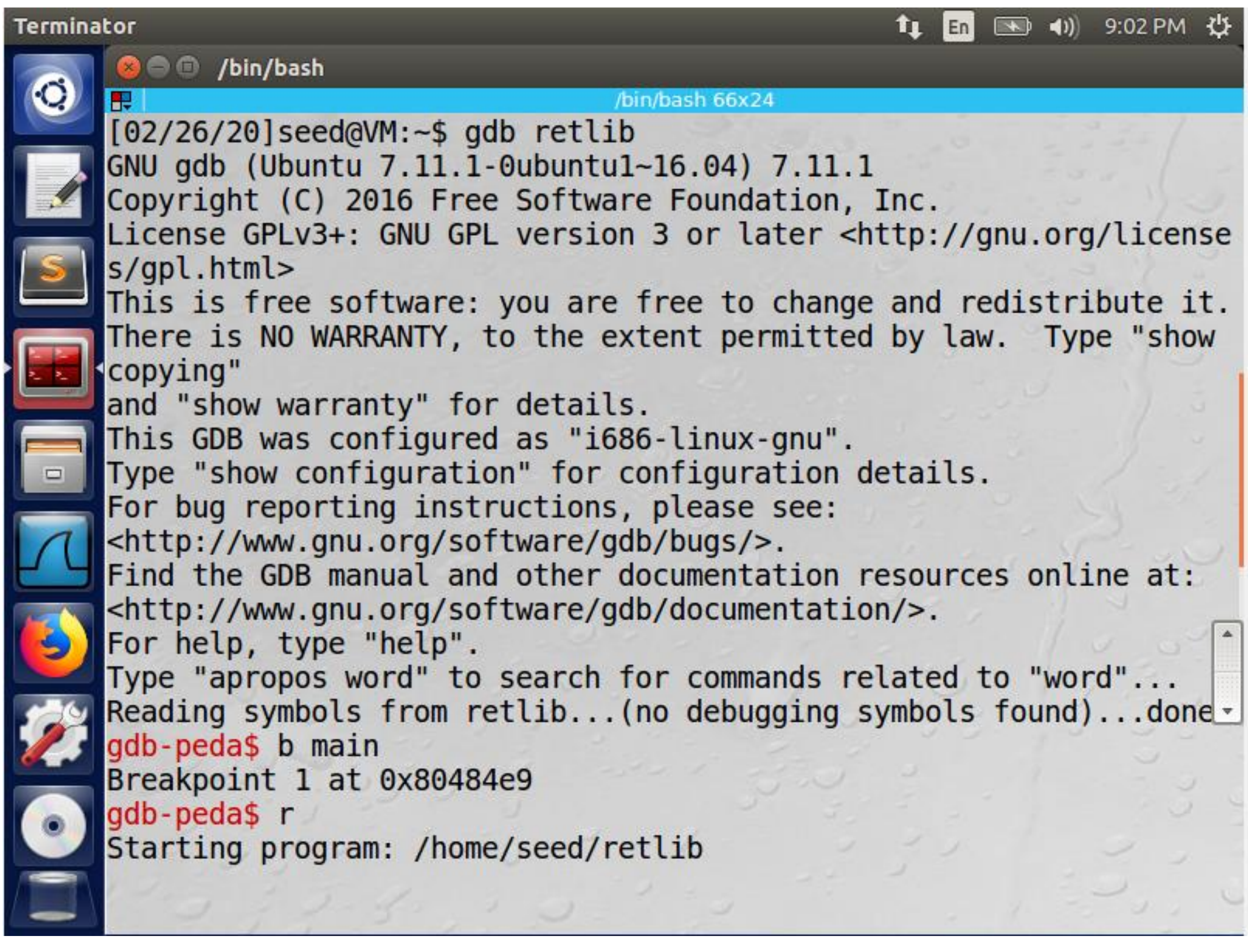
I now created the given retlib.c program and compiled using the gcc -fno-stack-protector -z noexecstack -o retlib retlib.c command. I have compiled the program with non-executable stack and stack guard disabled. I now changed the ownership of the compiled program to root and made the compiled program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

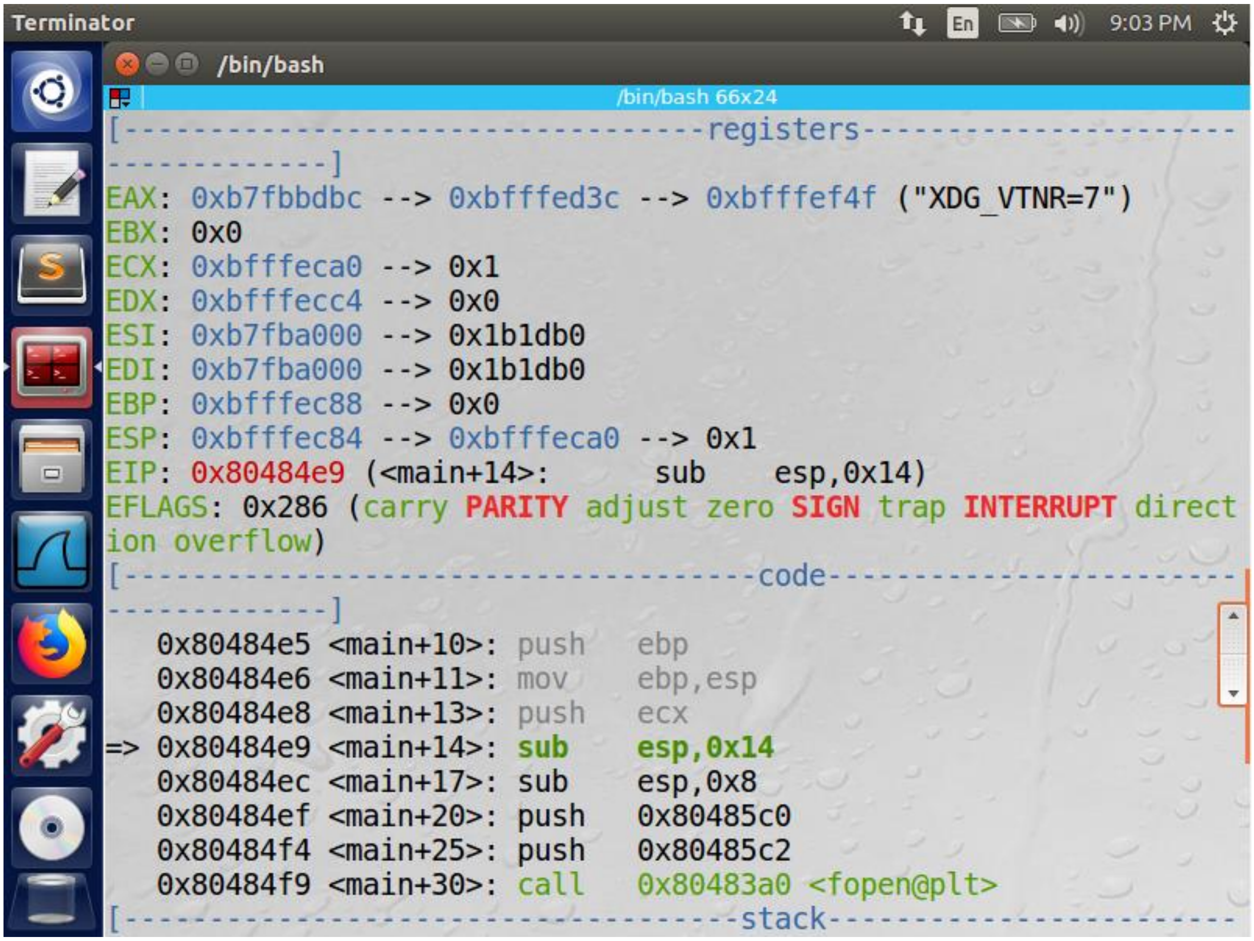


2.3 Task 1: Finding out the addresses of libc functions

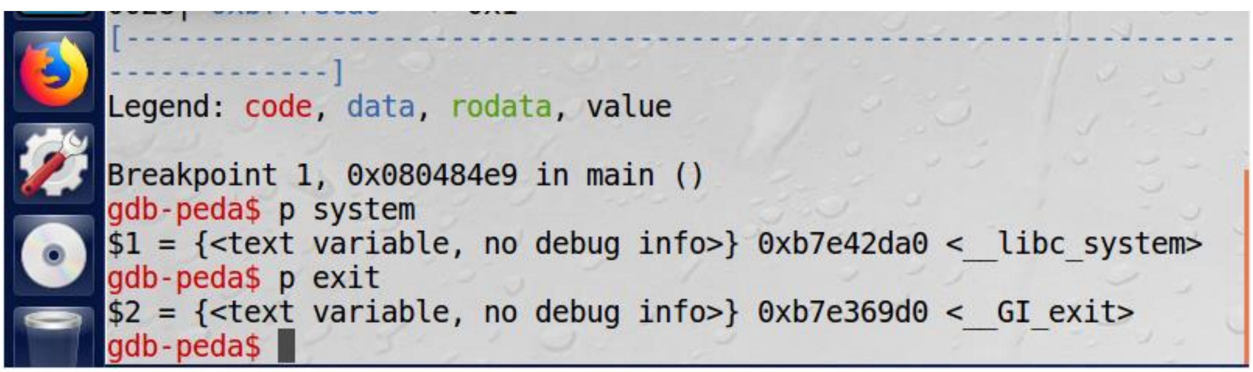
Output:

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.





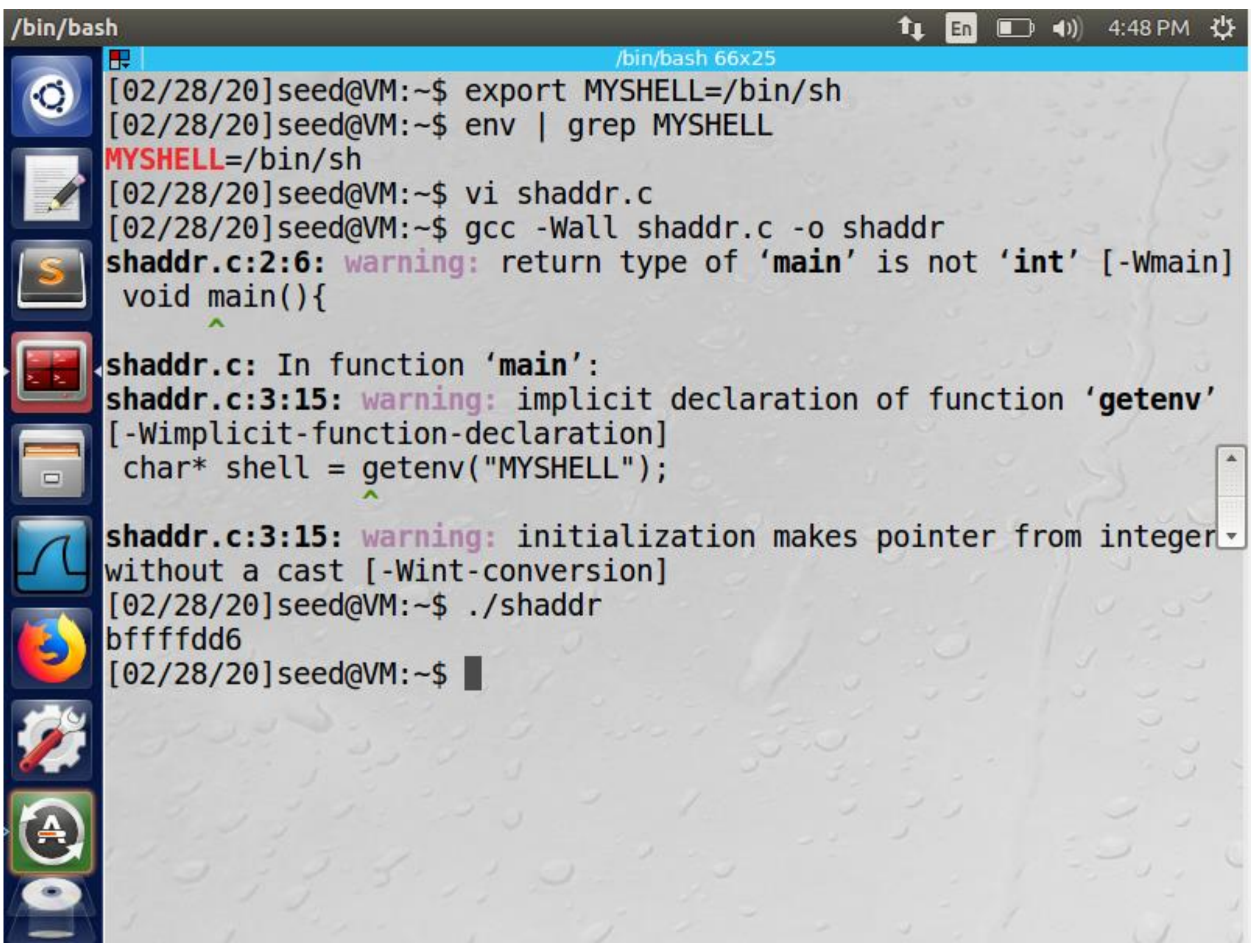
I got the address of the system() and exit() function using the p command in the GNU debugger.



2.4 Task 2: Putting the shell string in the memory

Output:

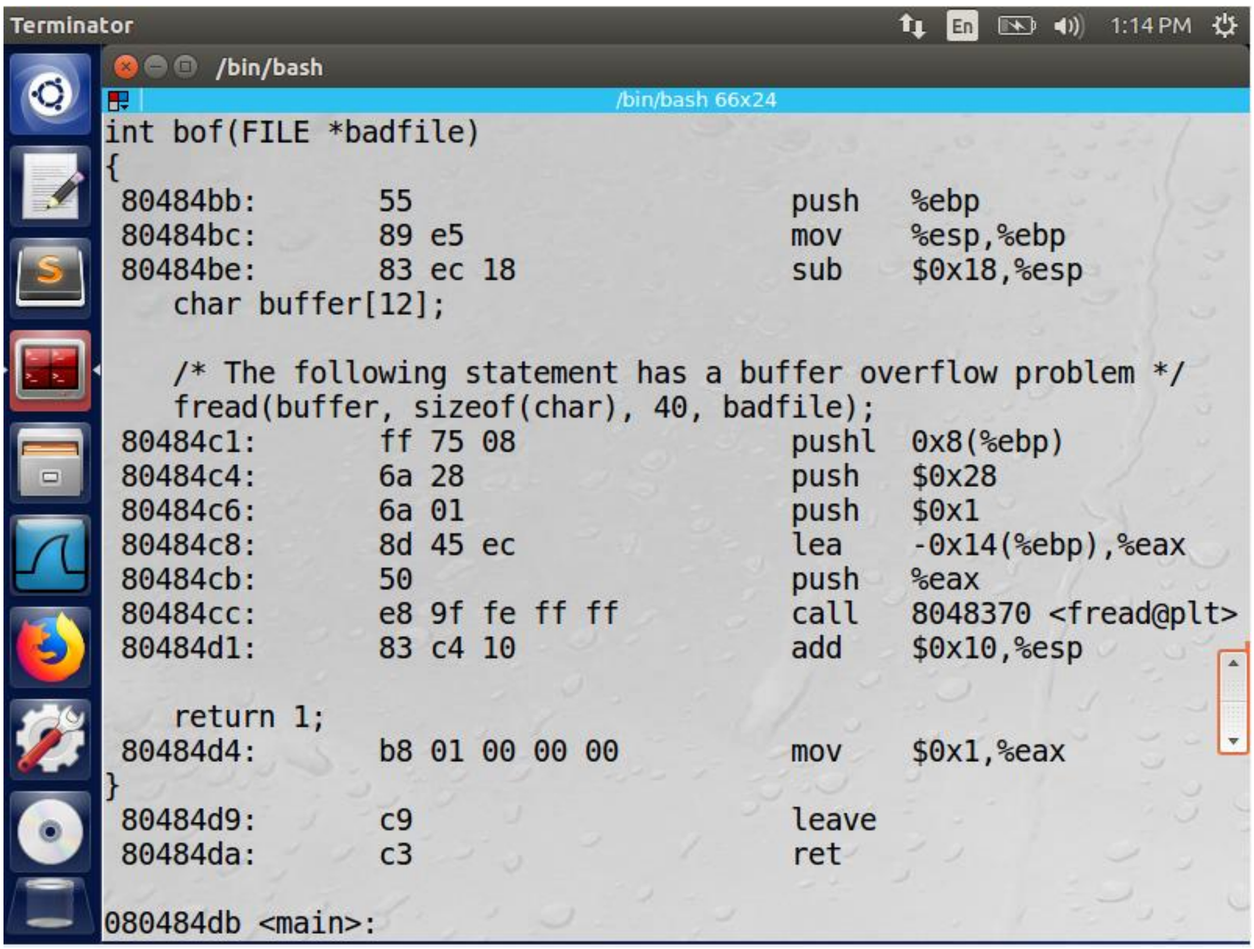
To execute an arbitrary command by the system(), we need the system() function execute the /bin/sh program. Using environment variables I put the /bin/sh command into memory so that we can the pass the address of the /bin/sh command to the system() function. I used the export function create a new shell 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 shadrr.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.



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().



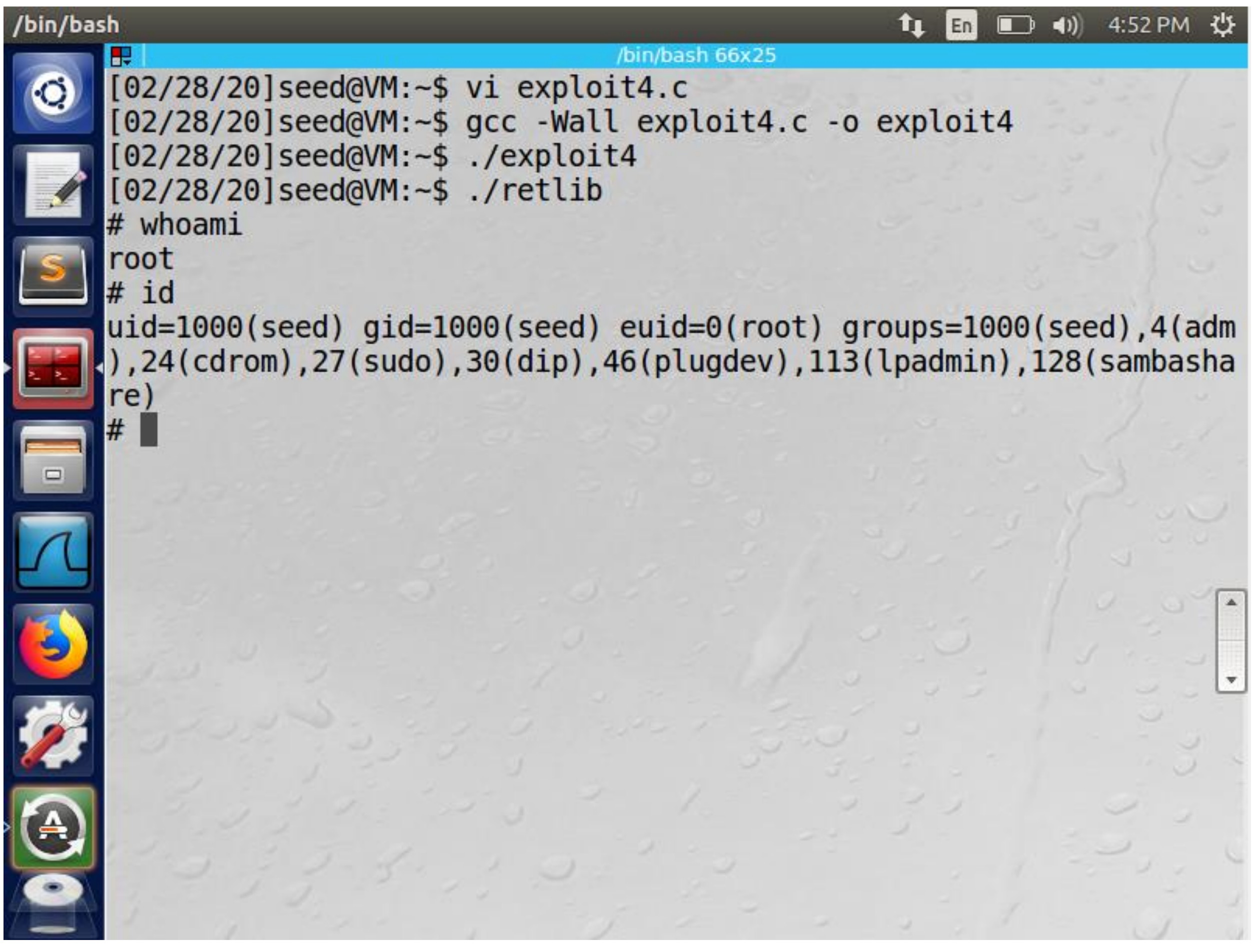
From getting the offset of system as 24, we store the address of the system into the buffer. We then get the offset of the exit() from the system 4 bytes higher from the system, thus the offset of the exit is 28. I then store the address of the exit into buffer. The offset if /bin/sh is 32 which again 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.

new 1 | retlib.c | exploit.c | new 2

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

```

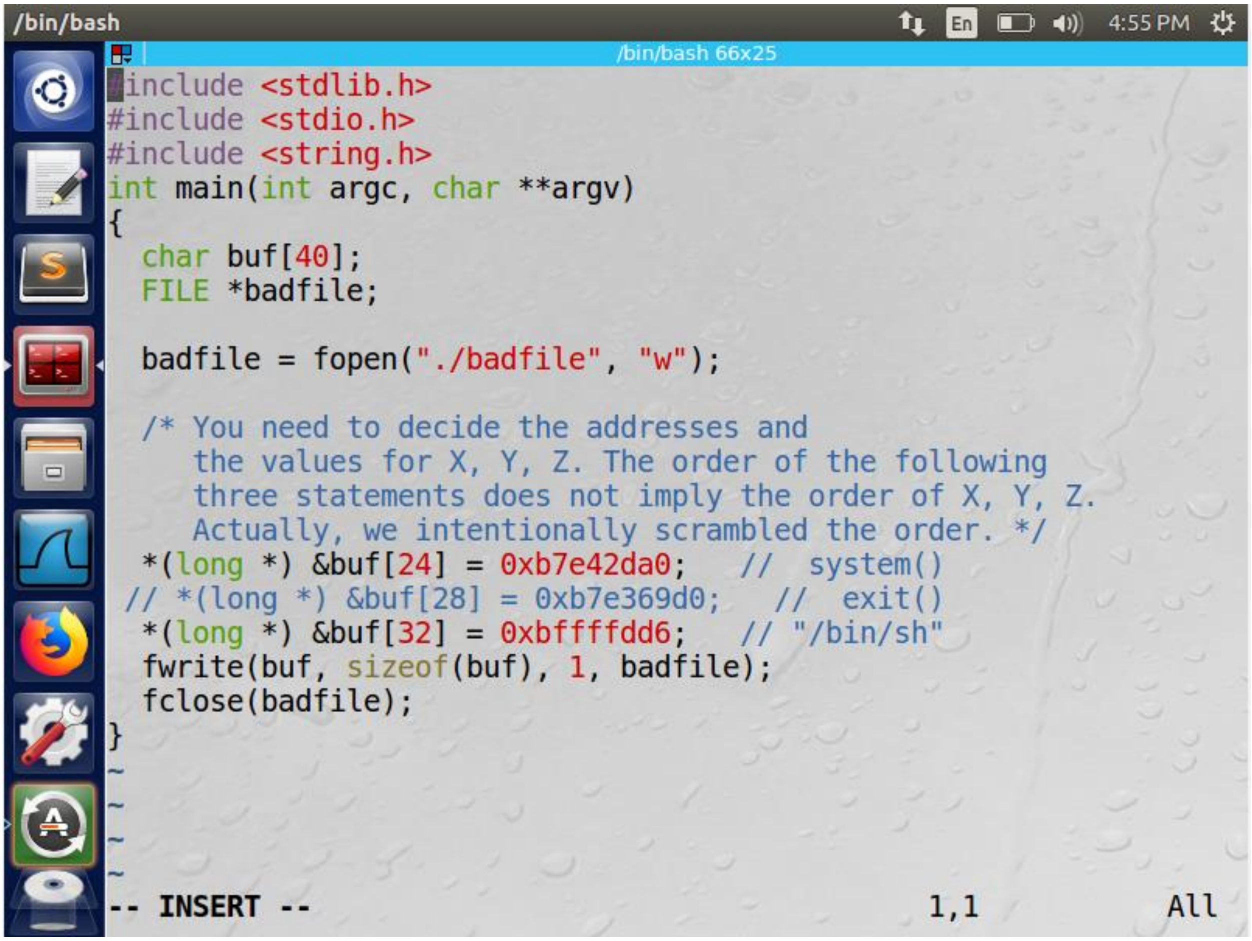
C source file | length : 589 lines : 21

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().



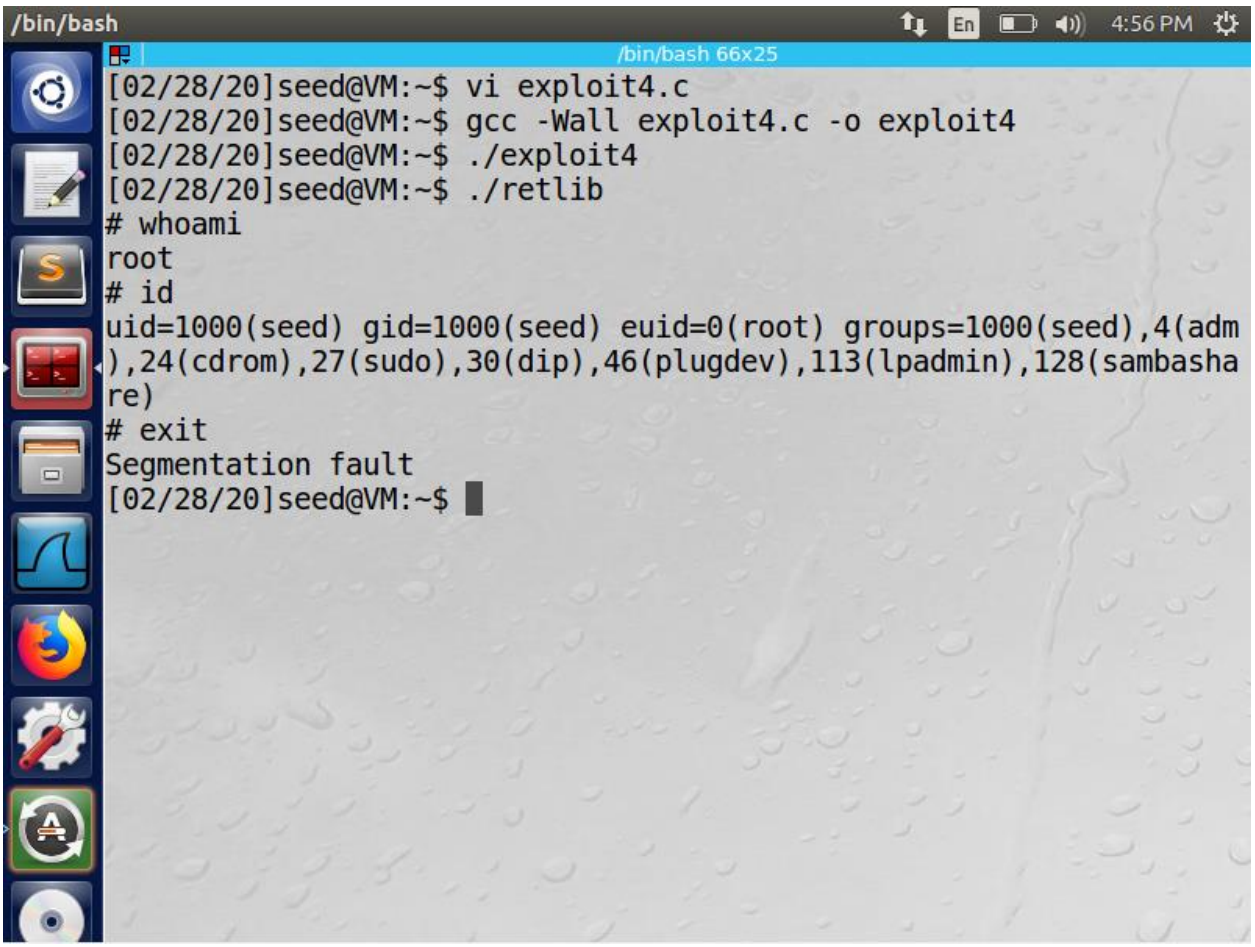
Attack variation 1:

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



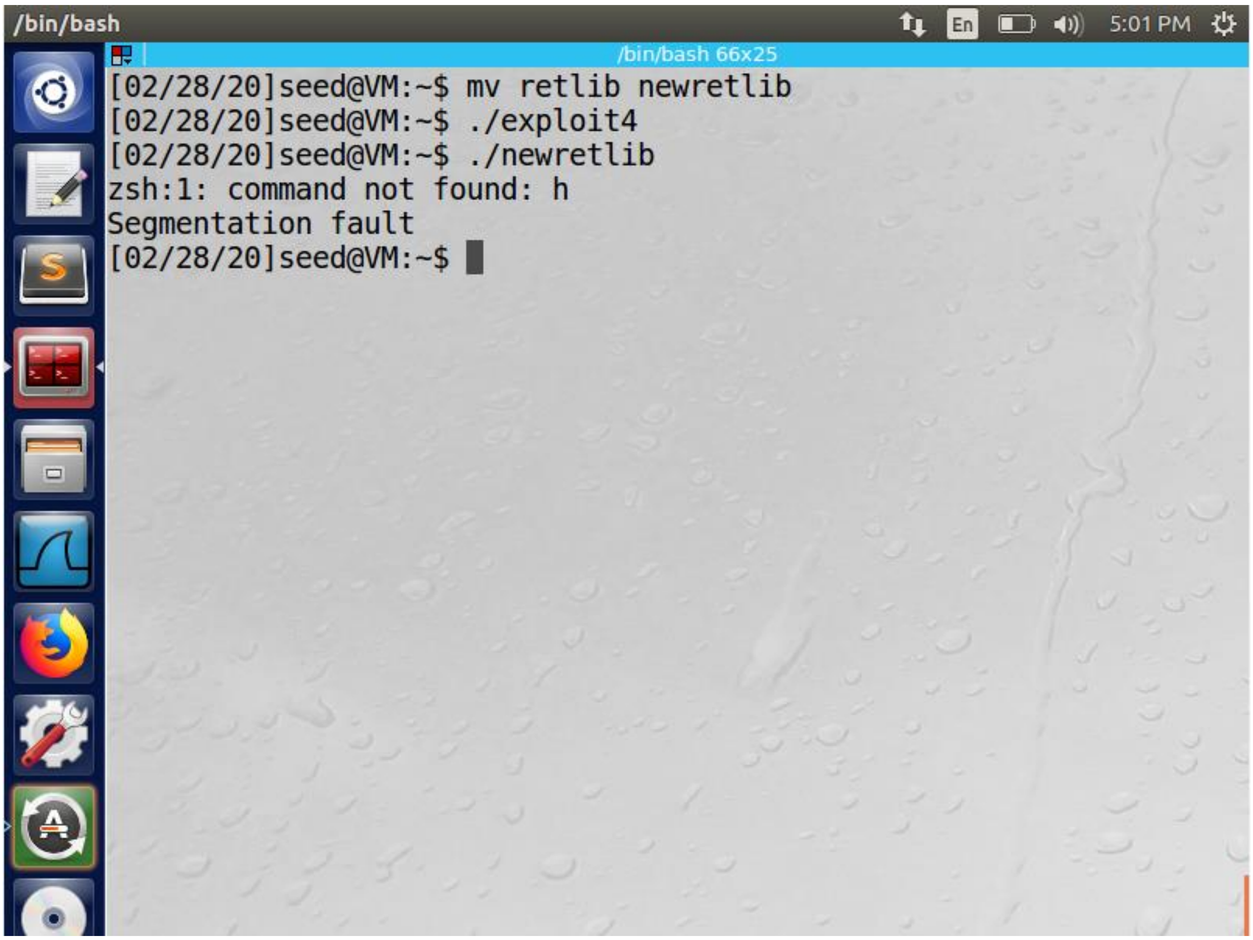
```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
  char buf[40];
  FILE *badfile;

  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 --                                        1,1           All
```

I have compiled the exploit program with commenting the address of the exit() function. I complied 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.



Attack variation 2:

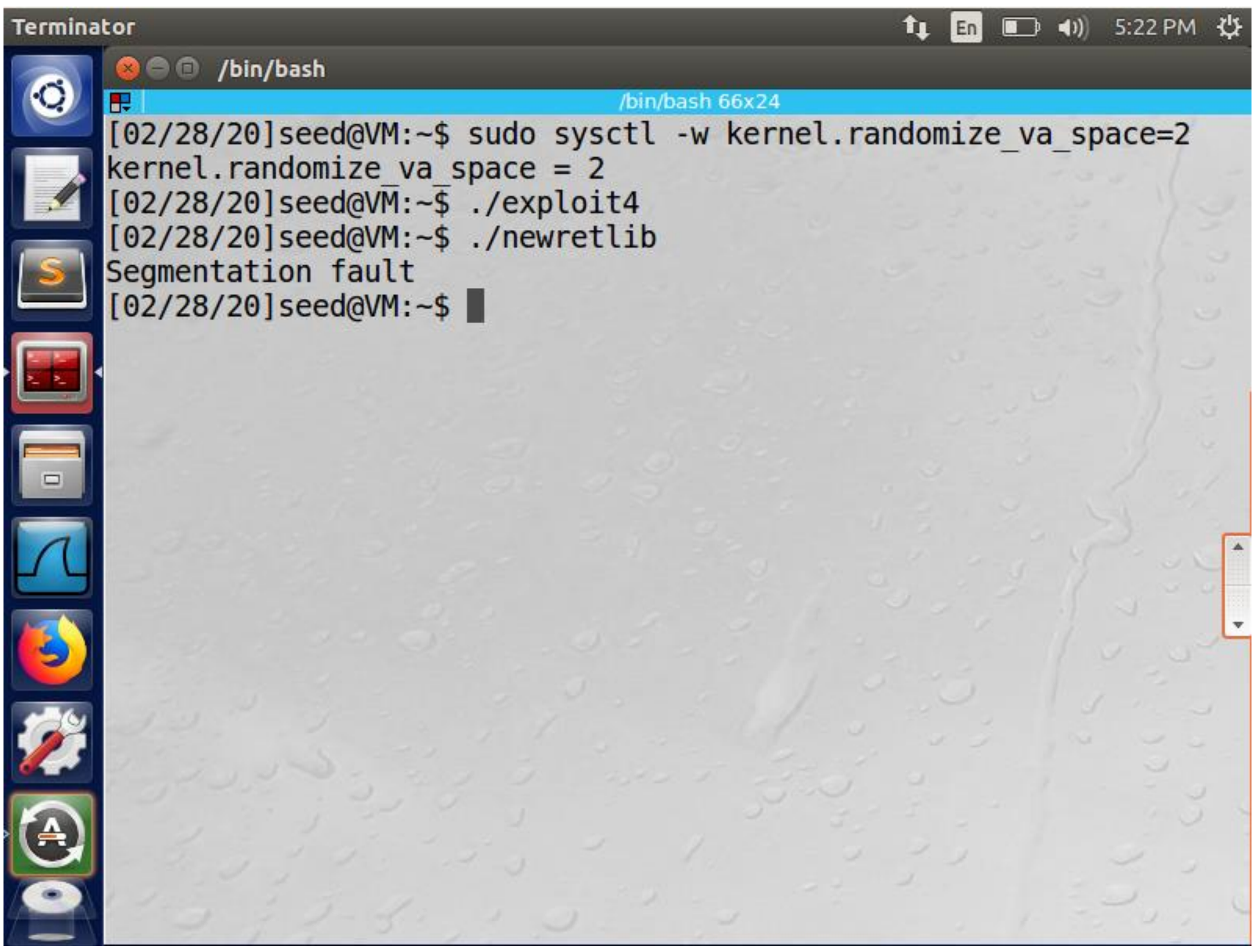
In this attack variation I have the changed the file name to newretlib from retlib using the mv command. I then ran the exploit program and then ran the newretlib program which was renamed. I was able to see segmentation fault, this is because I have changed the file name from retlib to newretlib without 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.



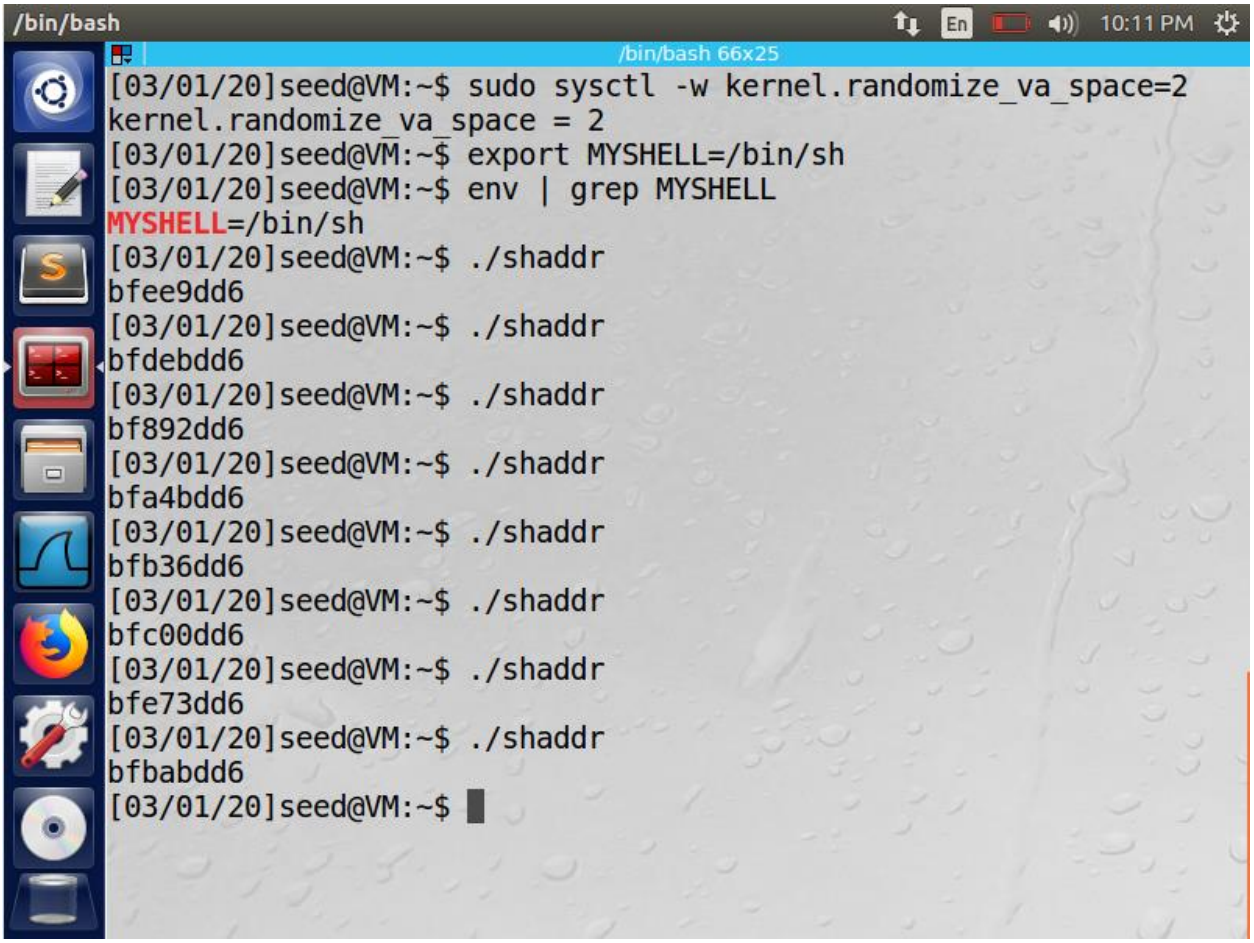
2.6 Task 4: Turning on Address Randomization

Output:

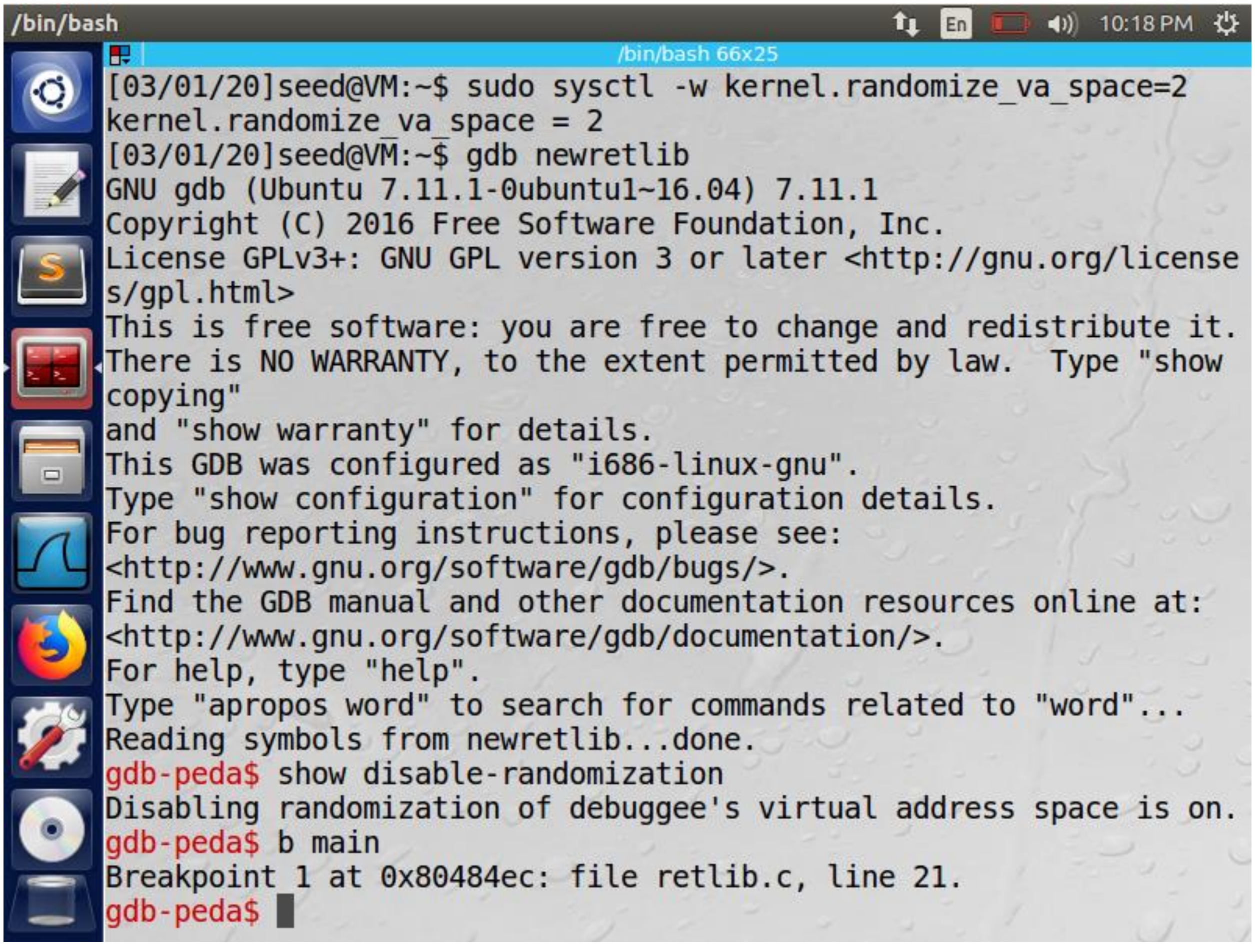
Before doing the task I have enabled the address randomization using the command sudo sysctl -w kernel.randomize_va_space=2. By enabling the address randomization, the memory addresses of the stack and heap gets changing randomly. By doing this it becomes difficult to guess the address of the stack and 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.



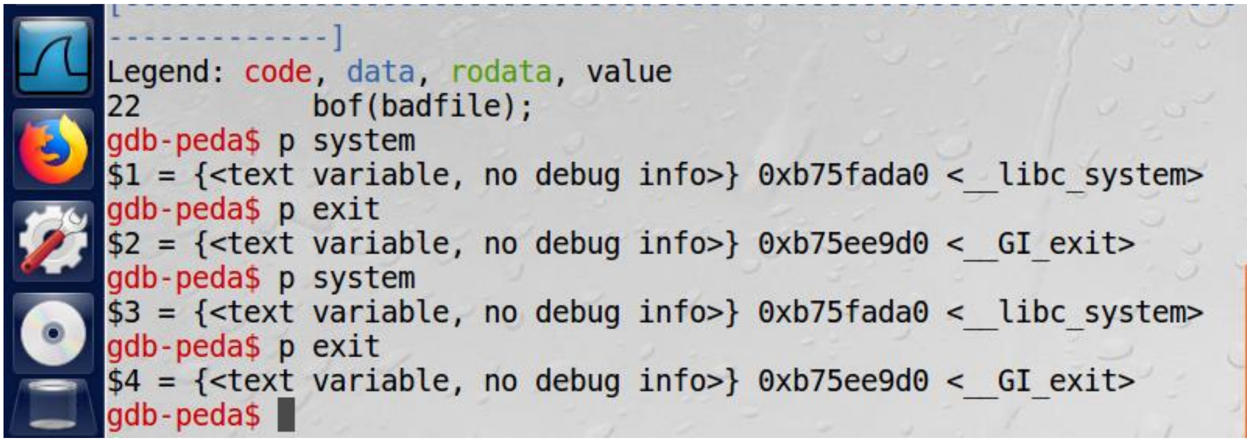
After enabling the address randomization, we export the shell variable called MYSHELL that holds the command /bin/sh, and I used the grep command to check if the shell variable has been exported. Then I ran the program from task2 which prints the address of the /bin/sh. I was able to see that the address 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.



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.

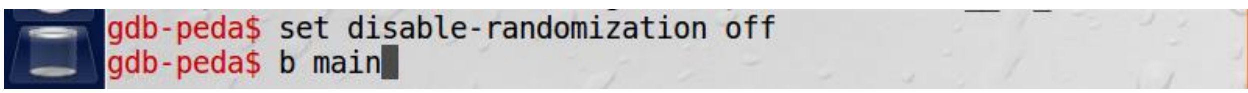


Now I check the addresses of the system() and the exit(), I was able to get the addresses for both system and exit when I ran the command to see the address.



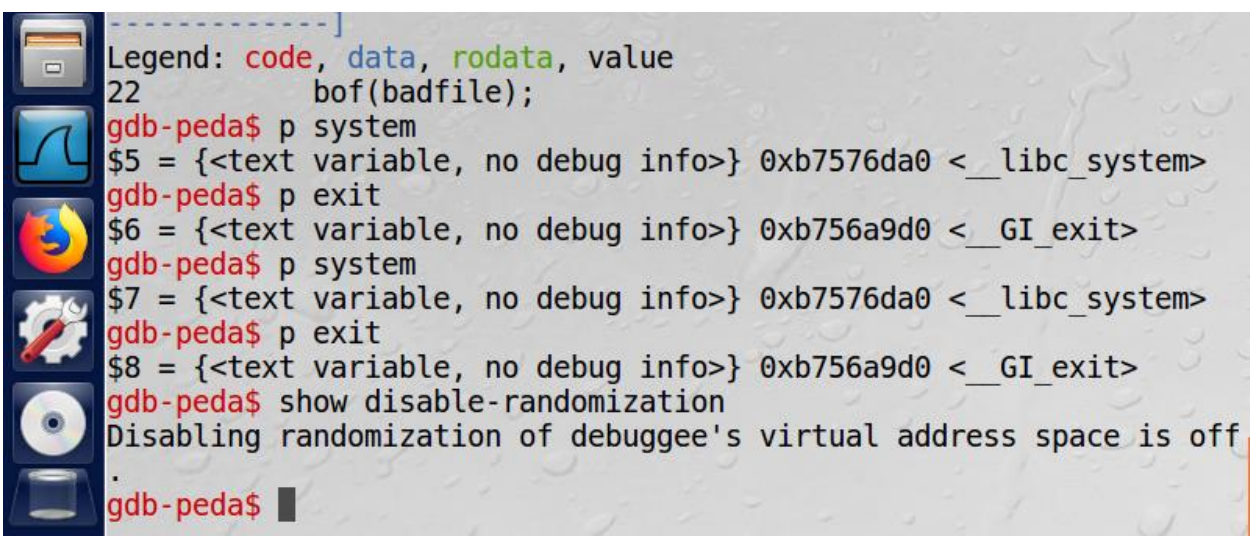
```
-------------]
Legend: code, data, rodata, value
22          bof(badfile);
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb75fada0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb75ee9d0 <__GI_exit>
gdb-peda$ p system
$3 = {<text variable, no debug info>} 0xb75fada0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