INFORMATION SECURITY LAB

LAB 4: Return-to-libc Attack Lab

Name: Ankitha P

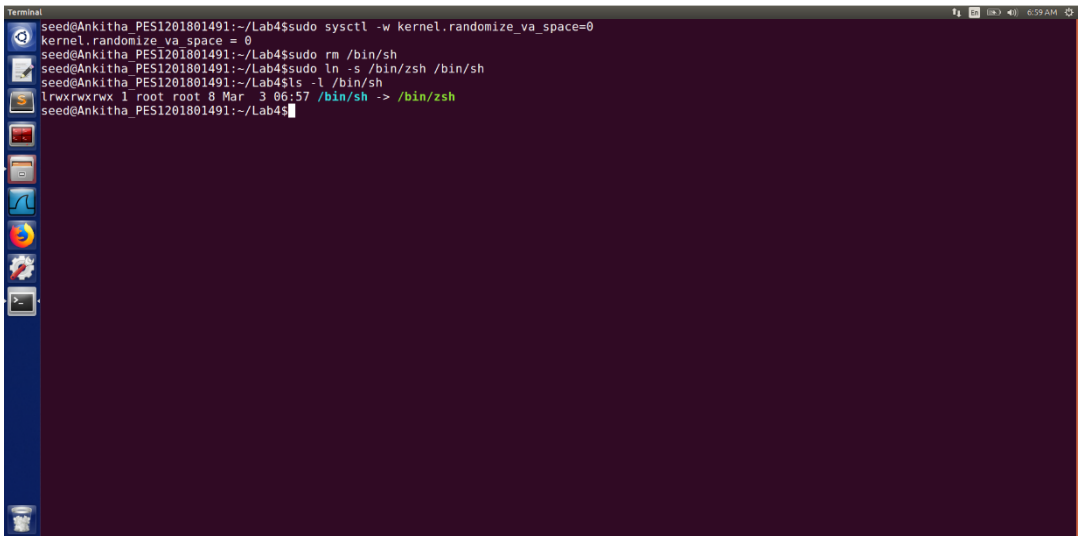
Class: 6 'D'

Date : 03/03/2021

EXECUTION

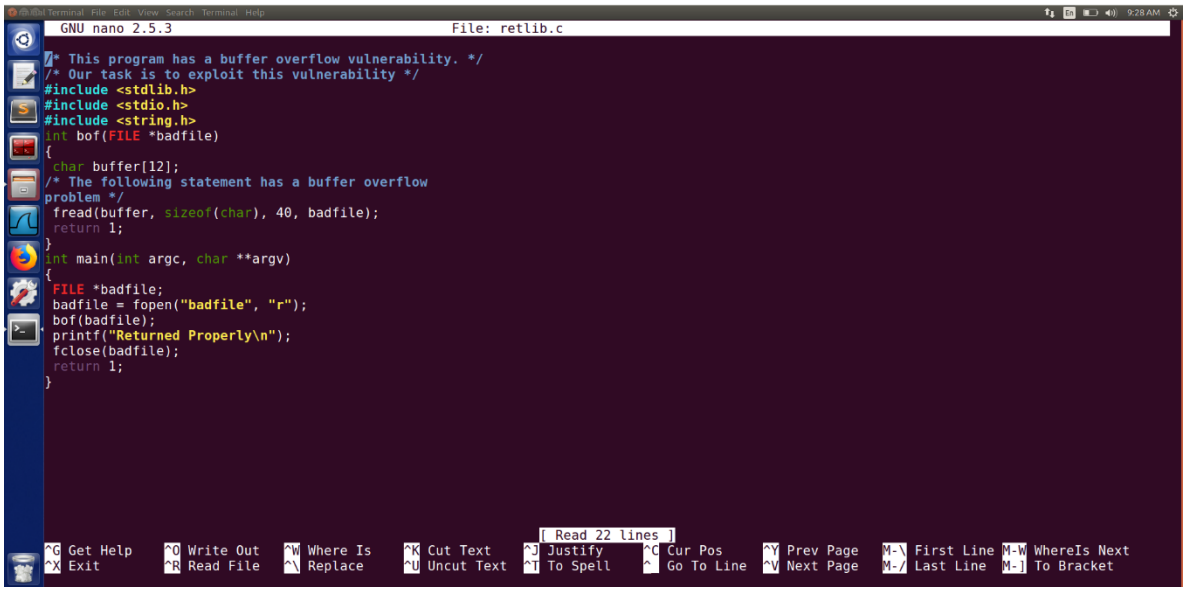
Task 1: Address Space Randomization

In order to perform the attack, first we disable the countermeasure in the form of Address Space Layout Randomization. If it is enabled then it would be hard to predict the position of stack in the memory. So, for simplicity, we disable this countermeasure by setting it to 0 (false) in the sysctl file. Since our victim program is a Set-UID program, and our attack relies on running /bin/sh, the countermeasure in /bin/dash makes our attack more difficult. Therefore, we will link /bin/sh to another shell that does not have such a countermeasure. Hence we have changed the default shell from 'dash' to 'zsh' to avoid the countermeasure implemented in 'bash' for the SET-UID programs.

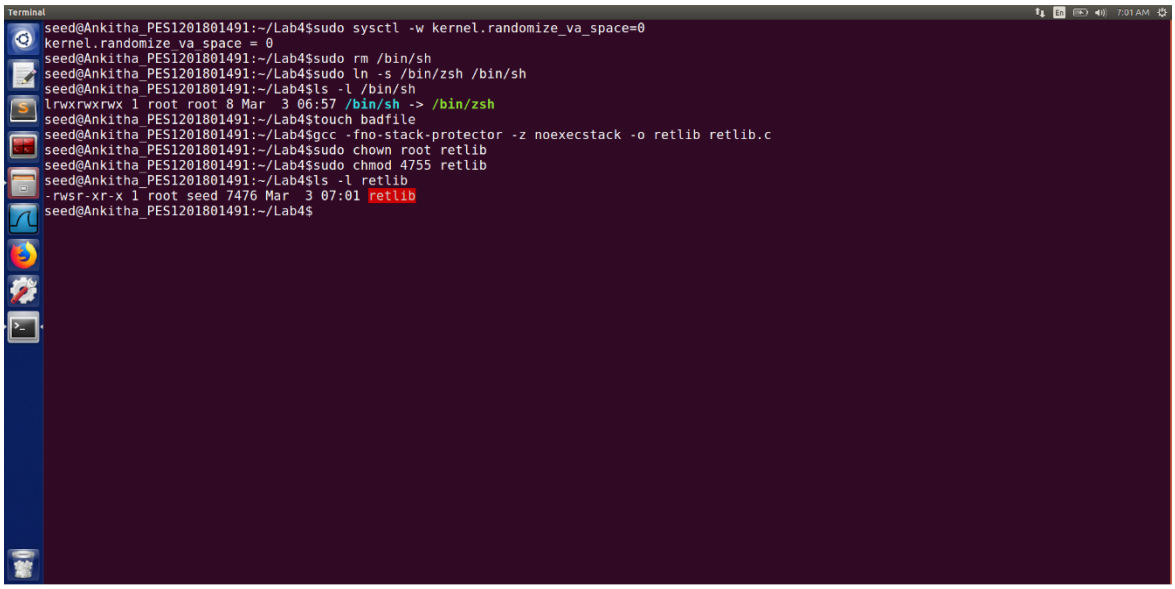


Next we create a vulnerable program named retlib.c. The below program has a buffer overflow vulnerability. It reads input from a file called badfile and passes the read data to a buffer via the function bof(). We declared the original input to have a maximum length of 40 bytes, but the buffer in

bof() function is only 12 bytes long. If our badfile has more than 12 bytes of data, there will be a buffer overflow.



We then compile the vulnerable program with executive stack option and turn off the stack guard. We use the -fno-stack-protector option (for turning off the StackGuard protection) and the "-z noexecstack" option (for turning on the non-executable stack protection)Now the vulnerable program is set as a set UID program owned by root.

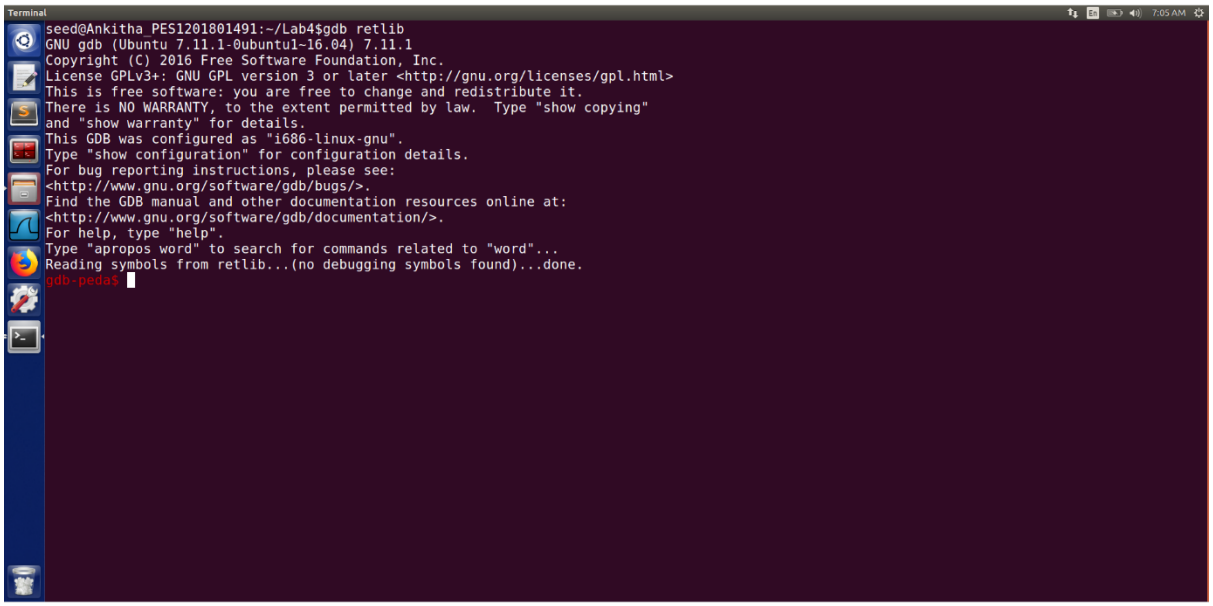


Since this program is a Set-UID program, if a normal user can exploit this buffer overflow vulnerability, the normal user might be able to get a root shell.

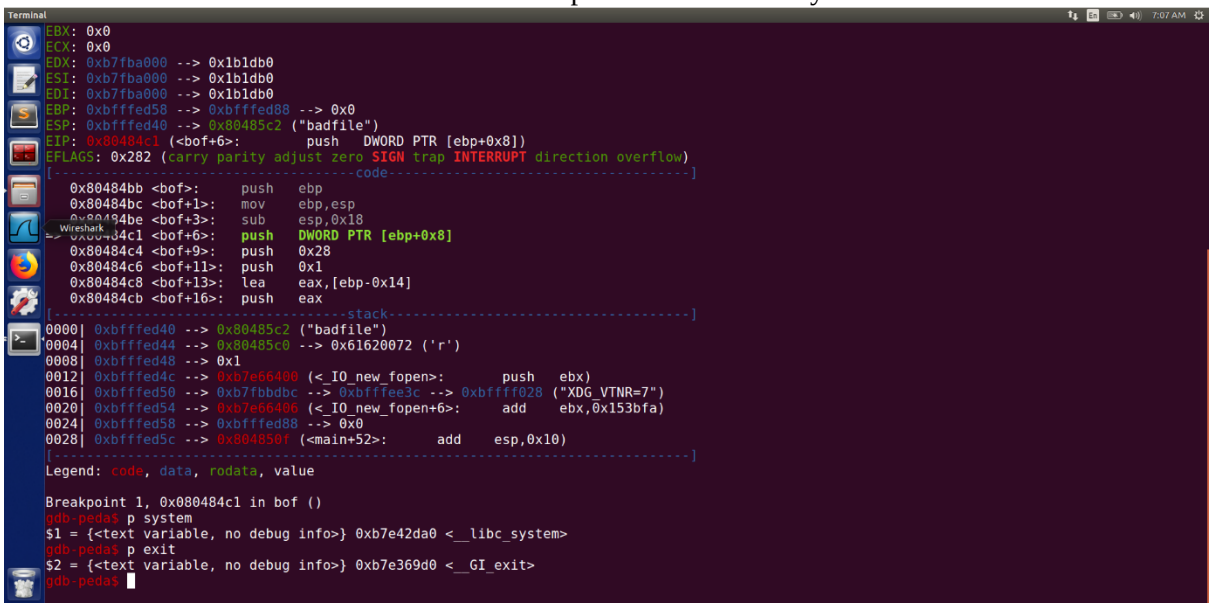
Task 2: Finding out the address of the lib function

In Linux, when a program runs, the libc library will be loaded into memory. When the memory address randomization is turned off, for the same program, the library is always loaded in the same. Therefore, we can easily find out the address of system() using a debugging tool such as gdb. Hence we debug the target program retlib which is a setUID program to obtain the address of system and exit. Inside gdb, we need to type the run command to execute the target program once, otherwise, the library code will

not be loaded. We use the p command (or print) to print out the address of the system() and exit() functions



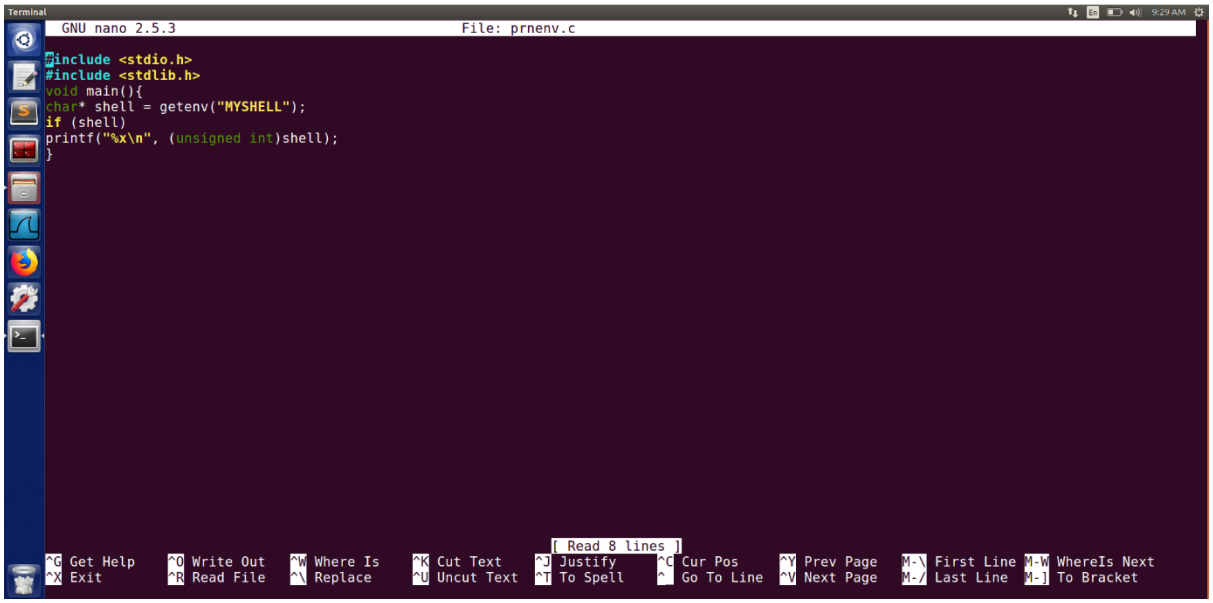
We can see below that the address of the system function is 0xb7e42da0 and the address of the exit function is 0xb7e369d0 which can be used to exploit the vulnerability.



Task 3 : Putting the shell string in the memory

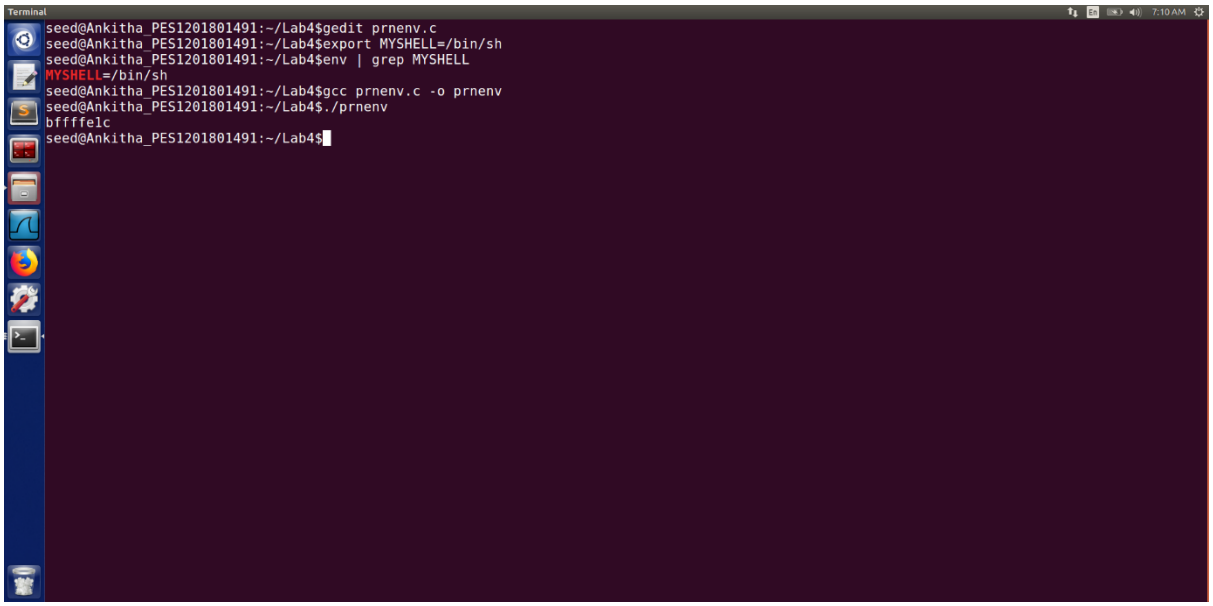
Our attack strategy is to jump to the system() function and get it to execute an arbitrary command. Since we would like to get a shell prompt, we want the system() function to execute the "/bin/sh" program. Therefore, the command string "/bin/sh" must be put in the memory first and we have to know its address.

The below program is used to find and print the address of the environment variable called MYSHELL.



We use the method of environment variables to get the address of "/bin/sh" to put it into the memory by passing it to system(). When we execute a program from a shell prompt, the shell actually spawns a child process to execute the program, and all the exported shell variables become the environment variables of the child process. This creates an easy way for us to put some arbitrary string in the child process's memory.

We export a new environment variable called MYSHELL pointing to the /bin/sh shell. When we run the above program, we can get the address of "/bin/sh" in memory as 0xbffffe1c.

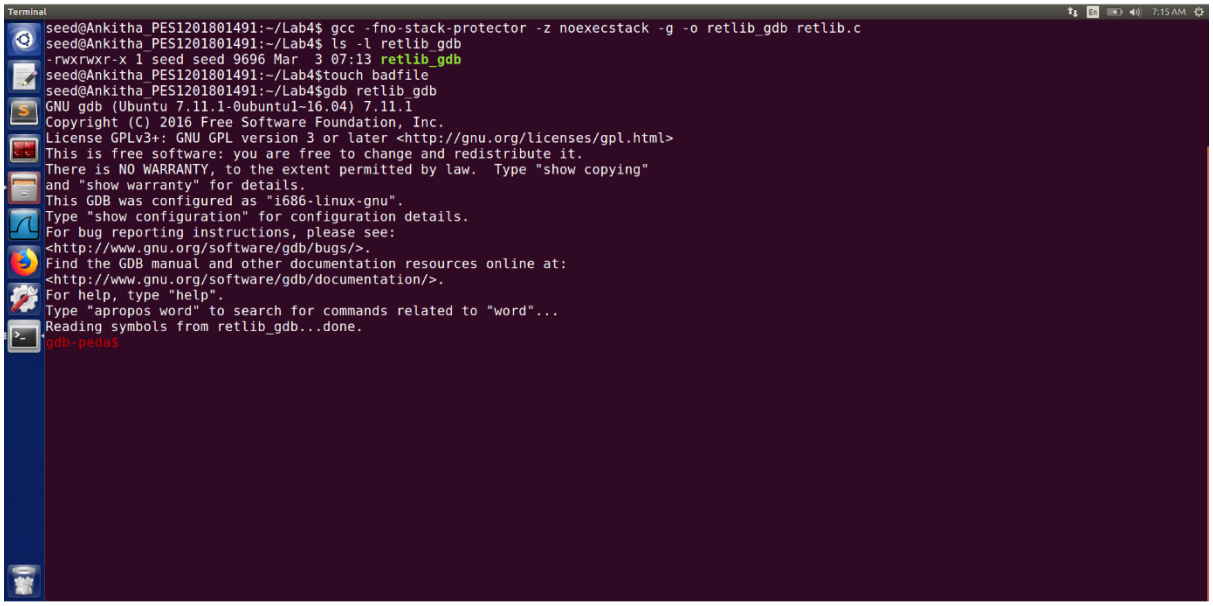


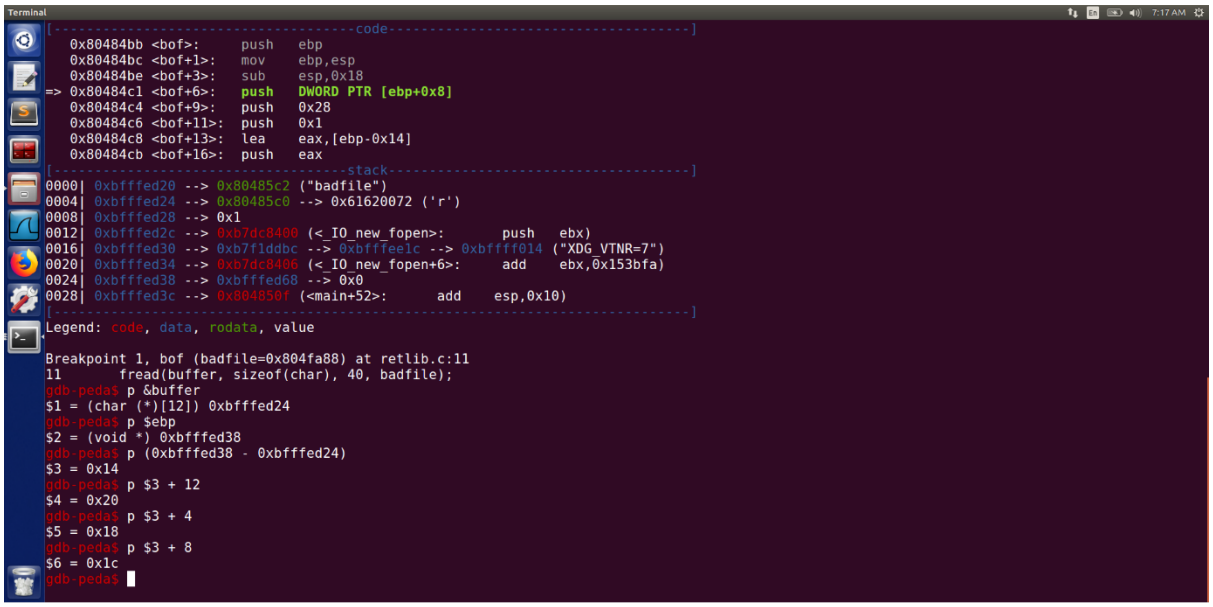
Now we have the addresses of the system call, exit call and the /bin/sh program but we need to know where to put these values so that we can exploit the buffer overflow vulnerability. For this reason, we again run the retlib program in debug mode(-g option), with the StackGuard countermeasure disabled and Stack executable and then run the program in debug mode using gdb. In gdb, we set a breakpoint on the bof function using b bof, and then start executing the program. The program stops inside the bof() function due to the breakpoint created. EBP stack frame value is used as a reference when accessing local variables and arguments of the bof function. We also print out the buffer value to find the start of the buffer and the difference of the two in order to find return address value's address to our malicious program.

Value of location of /bin/sh address = (ebp value - buffer value) + 12 (X)

Value of location of system call address = (ebp value - buffer value) + 4 (Y)

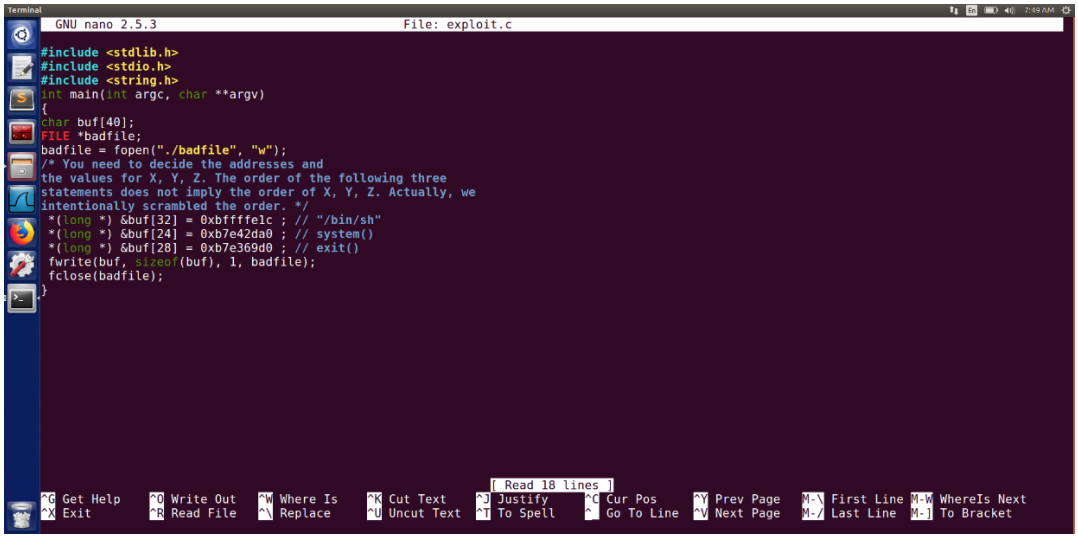
Value of location of exit call address = (ebp value - buffer value) + 8 (Z)



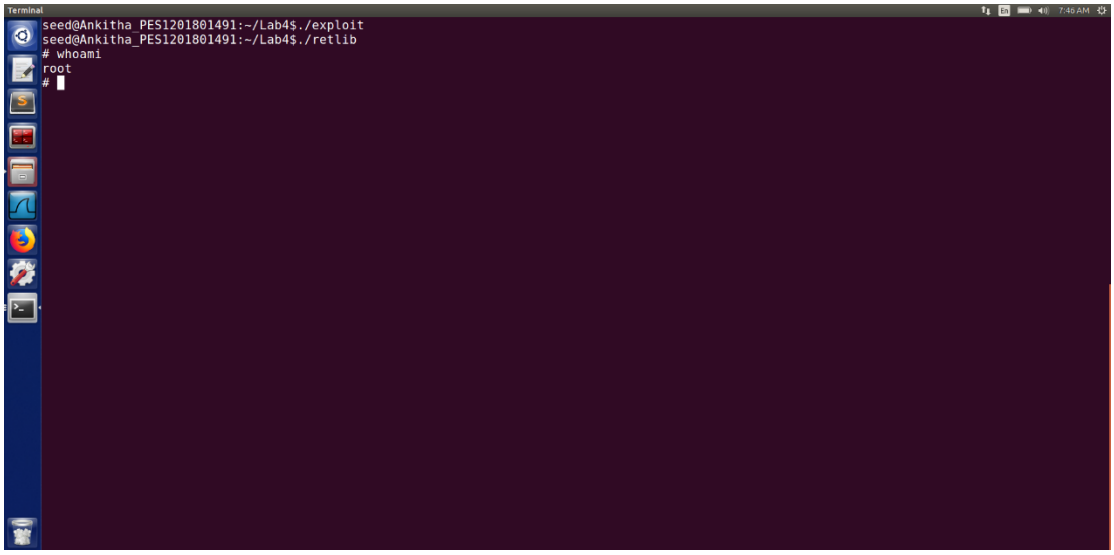


We store the addresses of the /bin/sh, system call and exit call in the locations 32, 24 and 28 respectively.The addresses are picked as such because when the function bof() returns, it will return to the system() function, and execute system("/bin/sh") and after the attack finishes we call exit to leave the shell.

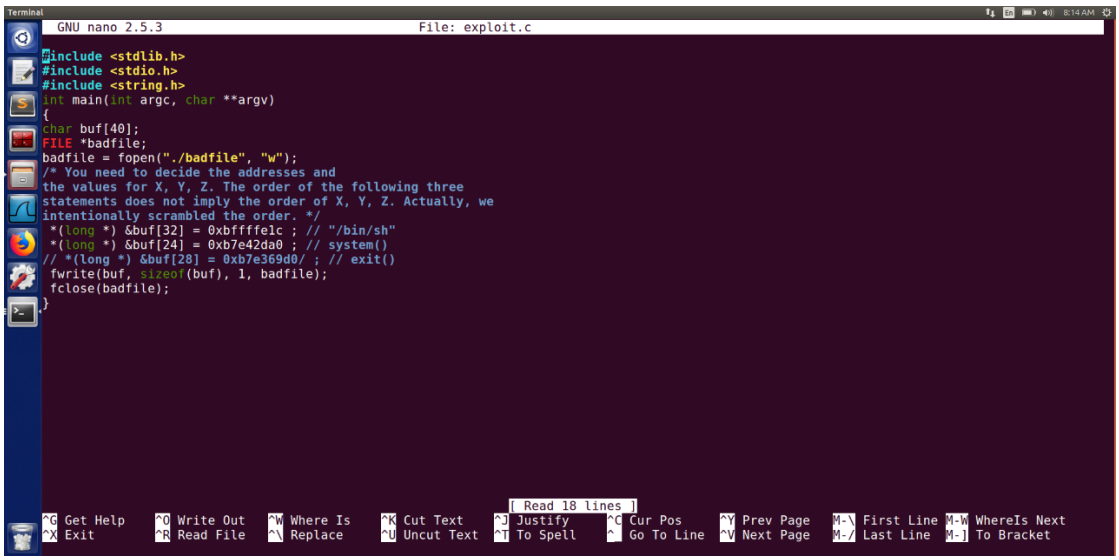
The below program is our exploit.c file in which we enter the address and locations of /bin/sh, system(), exit(). This file is used to exploit the vulnerability.



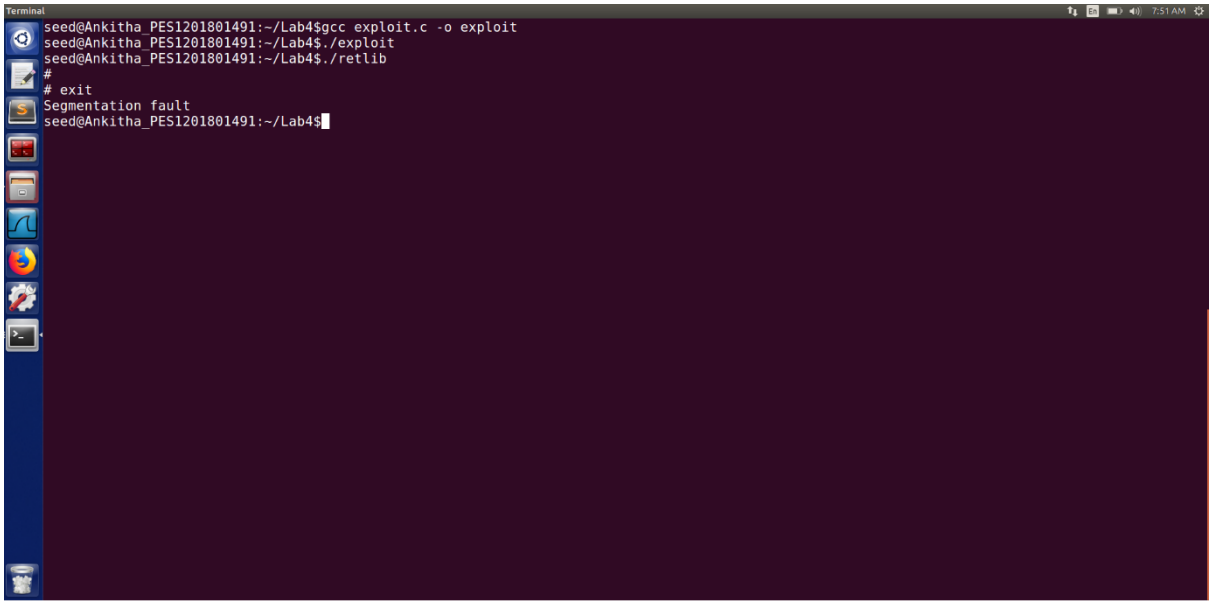
Now when we compile and run this exploit.c file, we generate the badfile with these addresses stored in it. On running the retlib file, we get the root shell as shown below:



On commenting the exit() call.

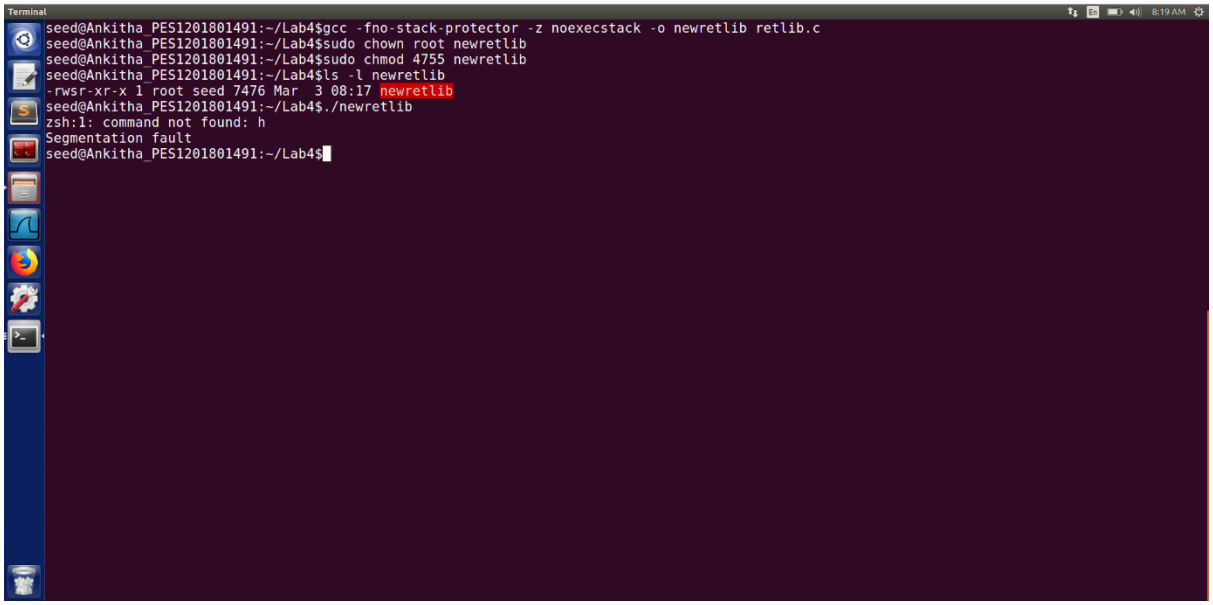


The exit() function is not very necessary for this attack; however, When we run the program without exit call, the program crashes after getting the root shell and exiting, causing suspicions.



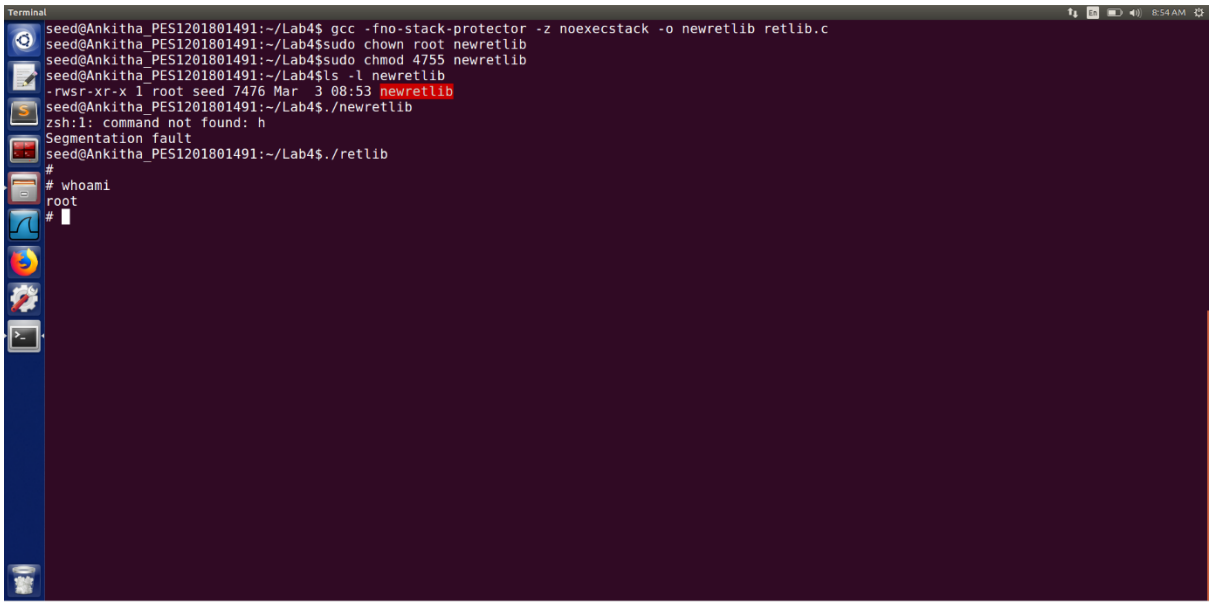
Task 4: Changing length of the file name

we try performing the attack with a new name for the vulnerable program as newretlib. We make it a Set-UID program and execute it but we are unsuccessful in exploiting the vulnerability.



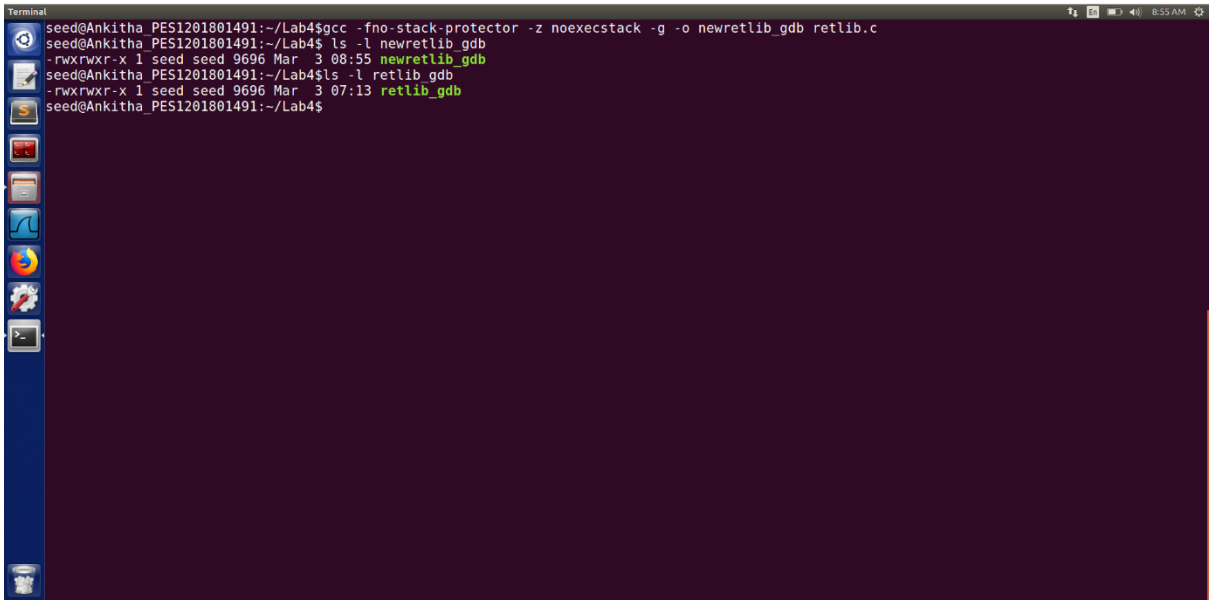
The attack no longer works with the new executable file. This is because the length of file name has changed the address of the environment variable(MYSHELL) in the process address space. The error message also makes it evident that the address has been changed from myshell, as the system() was now looking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

Performing the attack with old executable file.

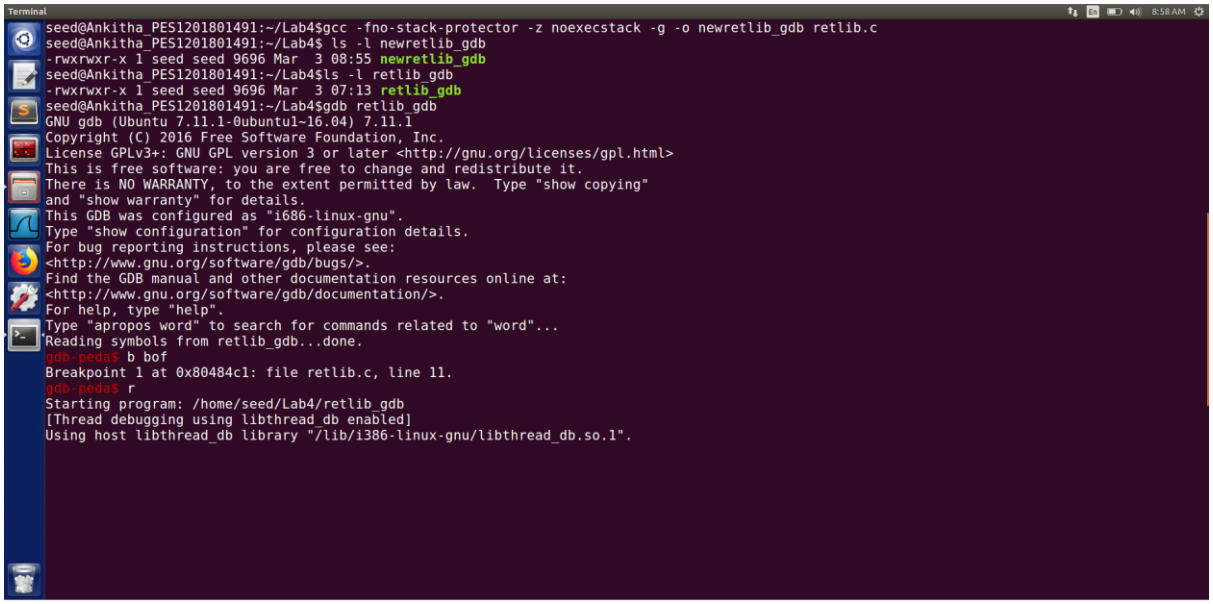


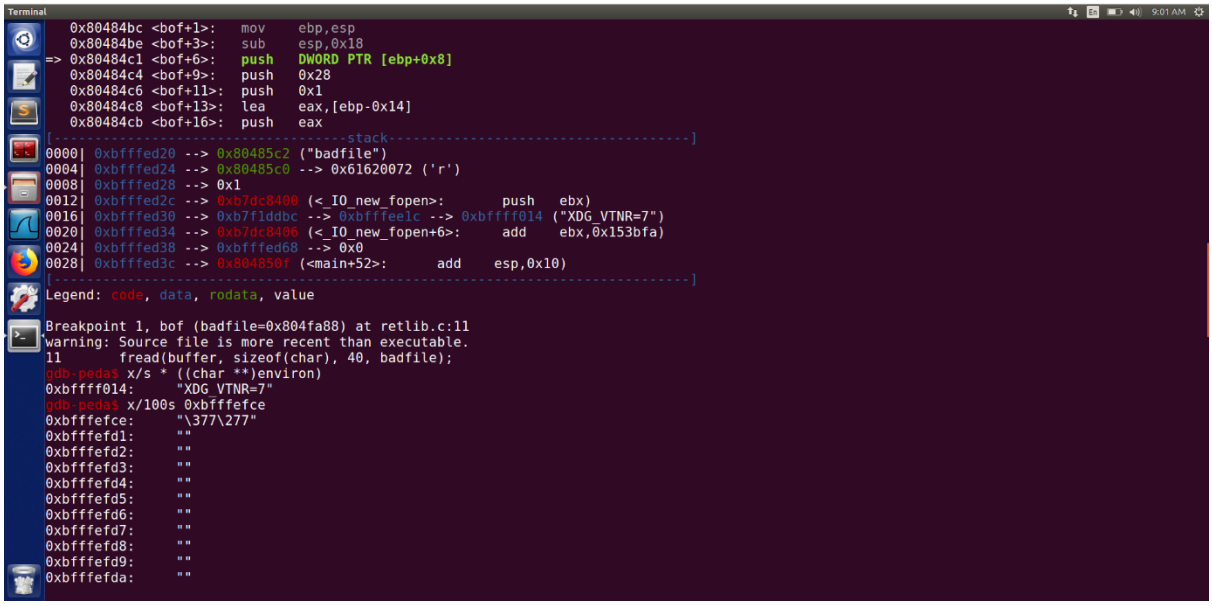
As we can observe from the screen shot the attack no longer works with the new executable file but still works with the old executable file, using the same content of badfile..

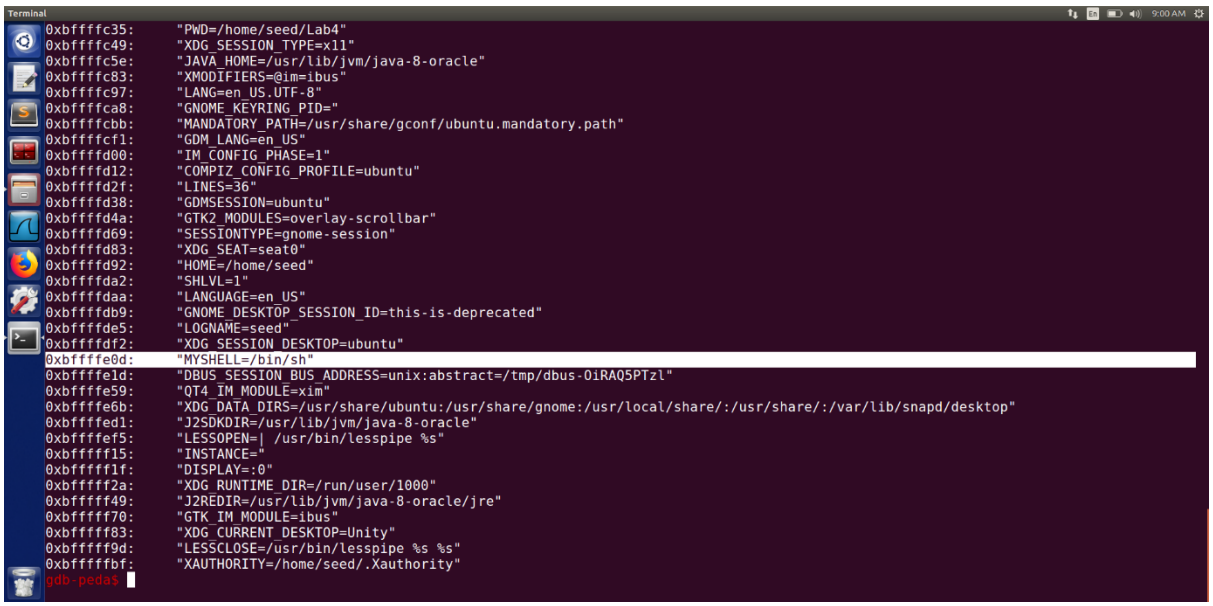
Now we compile the retlib program in the debug mode (-g option), with the StackGuard countermeasure disabled and Stack executable.



To show that the relative location of the MYSHELL environment variable changes on changing the program's name, we run the retlib program in debug mode and retrieve addresses to find out where exactly our MYSHELL environment variable resides. This is shown below:

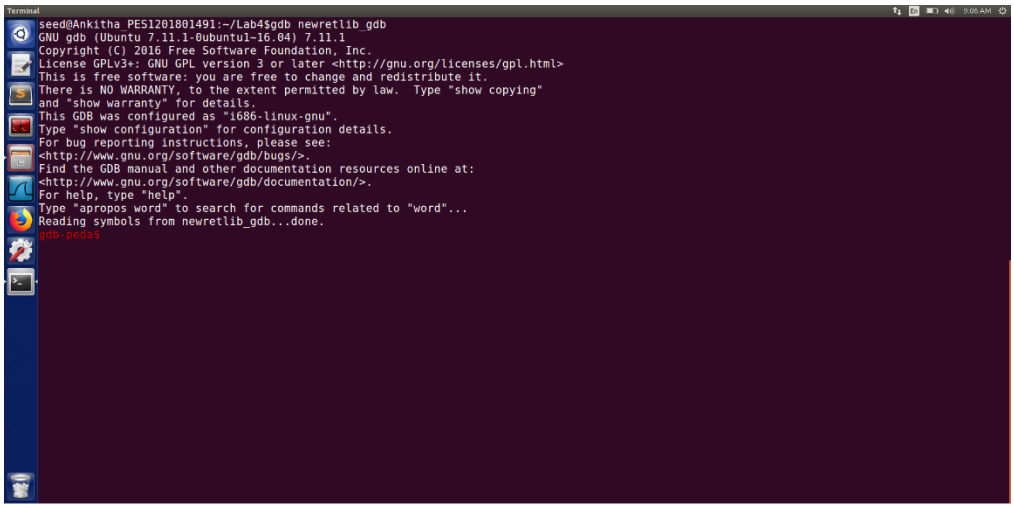

Terminal
8:58 AM
seed@Ankitha_PES1201801491:~/Lab4$gcc -fno-stack-protector -z noexecstack -g -o newretlib_gdb retlib.c
seed@Ankitha_PES1201801491:~/Lab4$ ls -l newretlib_gdb
-rwxrwxr-x 1 seed seed 9696 Mar 3 08:55 newretlib_gdb
seed@Ankitha_PES1201801491:~/Lab4$ls -l retlib_gdb
-rwxrwxr-x 1 seed seed 9696 Mar 3 07:13 retlib_gdb
seed@Ankitha_PES1201801491:~/Lab4$gdb retlib_gdb
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_gdb...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484c1: file retlib.c, line 11.
gdb-peda$ r
Starting program: /home/seed/Lab4/retlib_gdb
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

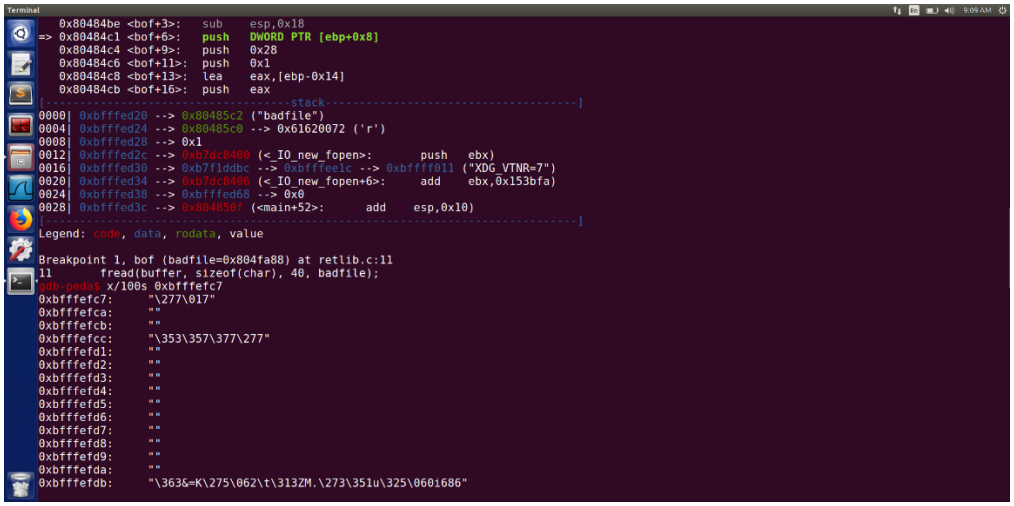

Terminal
9:01 AM
0x80484bc <bof+1>: mov ebp,esp
0x80484be <bof+3>: sub esp,0x18
=> 0x80484c1 <bof+6>: push DWORD PTR [ebp+0x8]
0x80484c4 <bof+9>: push 0x28
0x80484c6 <bof+11>: push 0x1
0x80484c8 <bof+13>: lea eax,[ebp-0x14]
0x80484cb <bof+16>: push eax
[----------stack----------]
0000| 0xbfffed20 --> 0x80485c2 ("badfile")
0004| 0xbfffed24 --> 0x80485c0 --> 0x61620072 ('r')
0008| 0xbfffed28 --> 0x1
0012| 0xbfffed2c --> 0xb7dc8400 (<_IO_new_fopen>: push ebx)
0016| 0xbfffed30 --> 0xb7f1ddbc --> 0xbfffee1c --> 0xbffff014 ("XDG_VTNR=7")
0020| 0xbfffed34 --> 0xb7dc8406 (<_IO_new_fopen+6>: add ebx,0x153bfa)
0024| 0xbfffed38 --> 0xbfffed68 --> 0x0
0028| 0xbfffed3c --> 0x804850f (<main+52>: add esp,0x10)
[----------]
Legend: code, data, rodata, value
Breakpoint 1, bof (badfile=0x804fa88) at retlib.c:11
warning: Source file is more recent than executable.
11 fread(buffer, sizeof(char), 40, badfile);
gdb-peda$ x/s * ((char **)environ)
0xbffff014: "XDG_VTNR=7"
gdb-peda$ x/100s 0xbfffefce
0xbfffefce: "\377\277"
0xbfffefd1: ""
0xbfffefd2: ""
0xbfffefd3: ""
0xbfffefd4: ""
0xbfffefd5: ""
0xbfffefd6: ""
0xbfffefd7: ""
0xbfffefd8: ""
0xbfffefd9: ""
0xbfffefda: ""


Terminal
9:00 AM
0xbffffc35: "PWD=/home/seed/Lab4"
0xbffffc49: "XDG_SESSION_TYPE=x11"
0xbffffc5e: "JAVA_HOME=/usr/lib/jvm/java-8-oracle"
0xbffffc83: "XMODIFIERS=@im=ibus"
0xbffffc97: "LANG=en_US.UTF-8"
0xbffffca8: "GNOME_KEYRING_PID="
0xbffffcbb: "MANDATORY_PATH=/usr/share/gconf/ubuntu.mandatory.path"
0xbffffcf1: "GDM_LANG=en_US"
0xbffffd00: "IM_CONFIG_PHASE=1"
0xbffffd12: "COMPIZ_CONFIG_PROFILE=ubuntu"
0xbffffd2f: "LINES=36"
0xbffffd38: "GDMSESSION=ubuntu"
0xbffffd4a: "GTK2_MODULES=overlay-scrollbar"
0xbffffd69: "SESSIONTYPE=gnome-session"
0xbffffd83: "XDG_SEAT=seat0"
0xbffffd92: "HOME=/home/seed"
0xbffffda2: "SHLVL=1"
0xbffffdaa: "LANGUAGE=en_US"
0xbffffdb9: "GNOME_DESKTOP_SESSION_ID=this-is-deprecated"
0xbffffde5: "LOGNAME=seed"
0xbffffdf2: "XDG_SESSION_DESKTOP=ubuntu"
0xbffffe0d: "MYSHELL=/bin/sh"
0xbffffe1d: "DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-OiRAQ5PTzl"
0xbffffe59: "QT4_IM_MODULE=xim"
0xbffffe6b: "XDG_DATA_DIRS=/usr/share/ubuntu:/usr/share/gnome:/usr/local/share/:/usr/share/:/var/lib/snapd/desktop"
0xbffffed1: "J2SDKDIR=/usr/lib/jvm/java-8-oracle"
0xbffffef5: "LESSOPEN=| /usr/bin/lesspipe %s"
0xbfffff15: "INSTANCE="
0xbfffff1f: "DISPLAY=:0"
0xbfffff2a: "XDG_RUNTIME_DIR=/run/user/1000"
0xbfffff49: "J2REDIR=/usr/lib/jvm/java-8-oracle/jre"
0xbfffff70: "GTK_IM_MODULE=ibus"
0xbfffff83: "XDG_CURRENT_DESKTOP=Unity"
0xbfffff9d: "LESSCLOSE=/usr/bin/lesspipe %s %s"
0xbfffffbf: "XAUTHORITY=/home/seed/.Xauthority"
gdb-peda$

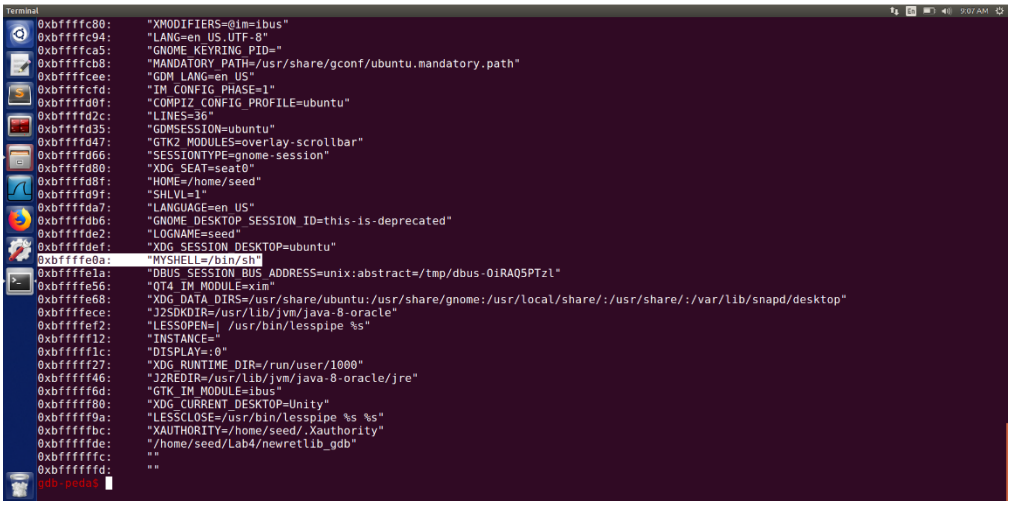
We are able to find the location of the MYSHELL environment variable and hence the retlib program on execution is able to call the /bin/sh shell.

We also run the newretlib program in debug mode and try to locate the MYSHELL environment variable.





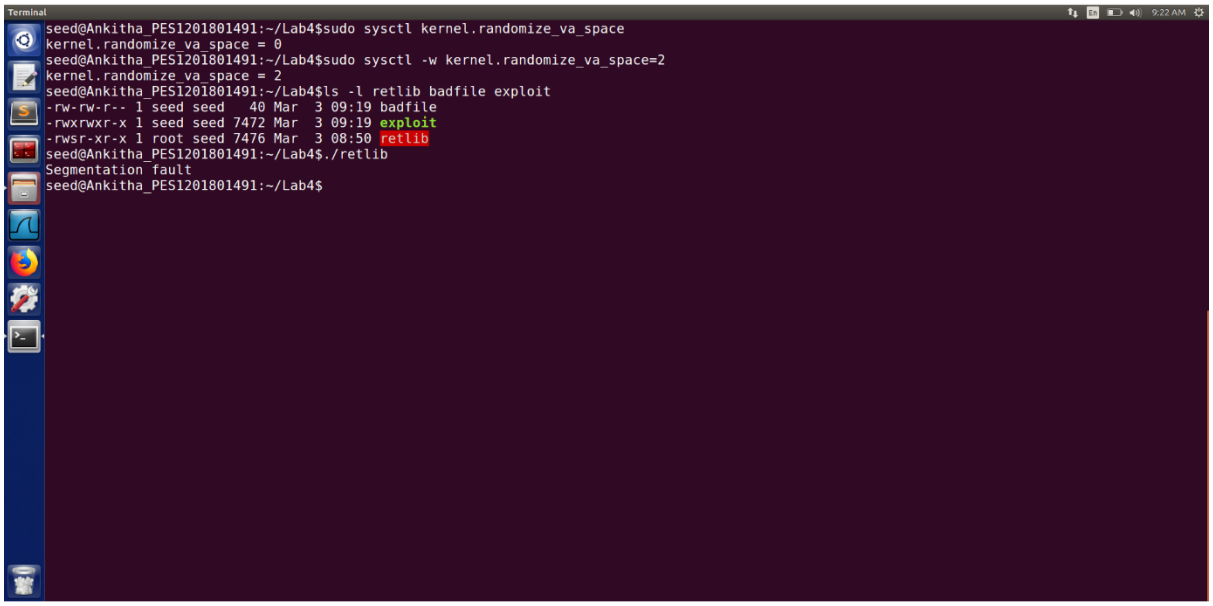
We see that the address of the MYSHELL variable is at some location different from the one written in the exploit.c code and this is why the attack fails when run by newretlib.



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

Task 5: Address Randomization

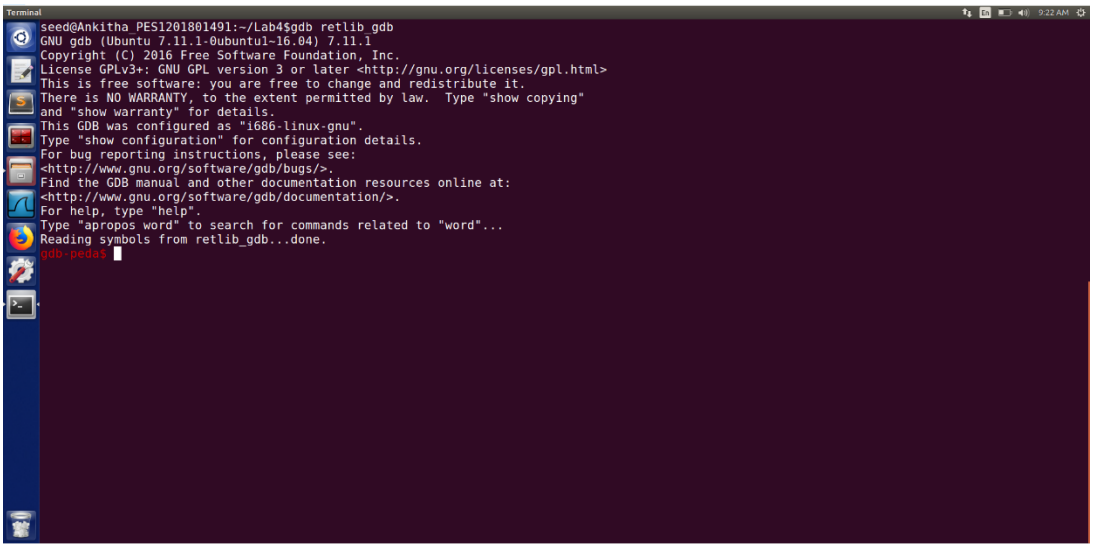
We compile the vulnerable program retlib.c with the address randomization turned on. We enable address randomization for both stack and heap by setting the value to 2. We get a segmentation fault on performing the attack.



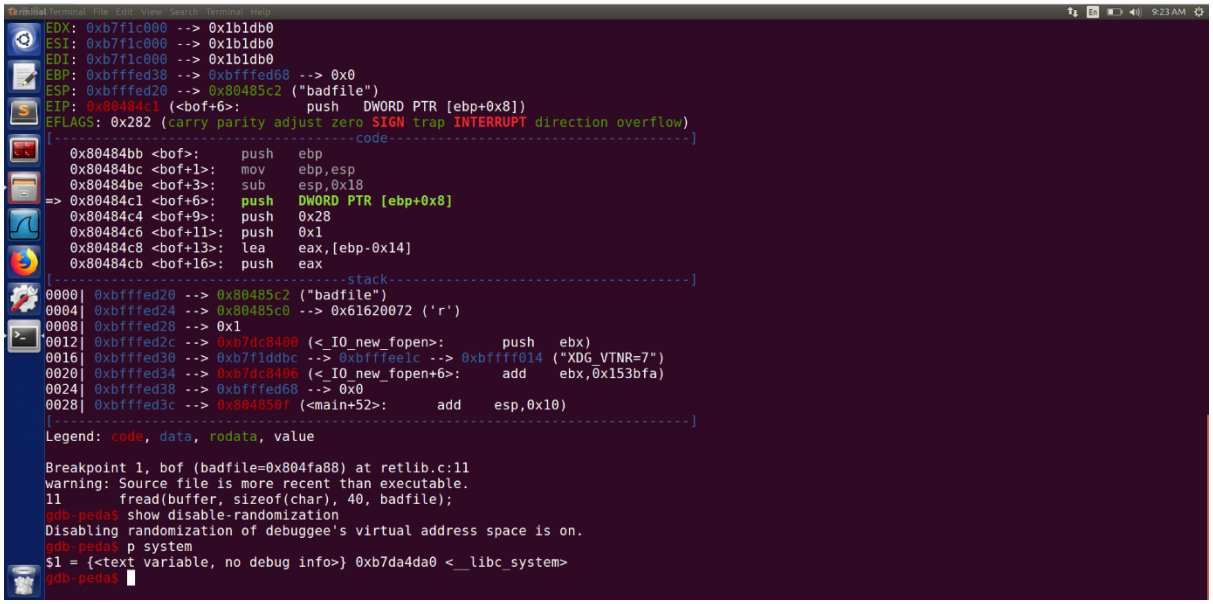
Since address randomization is turned on, the address of the environment variable, system function location and the exit function location keep changing randomly. So, the probability of exploiting the vulnerability becomes very less as we can't guess the addresses. This acts as a good protection mechanism against buffer overflow vulnerability.

Now we run the retlib_gdb program in debug mode with different breakpoints- one at the bof function and one at the main function.

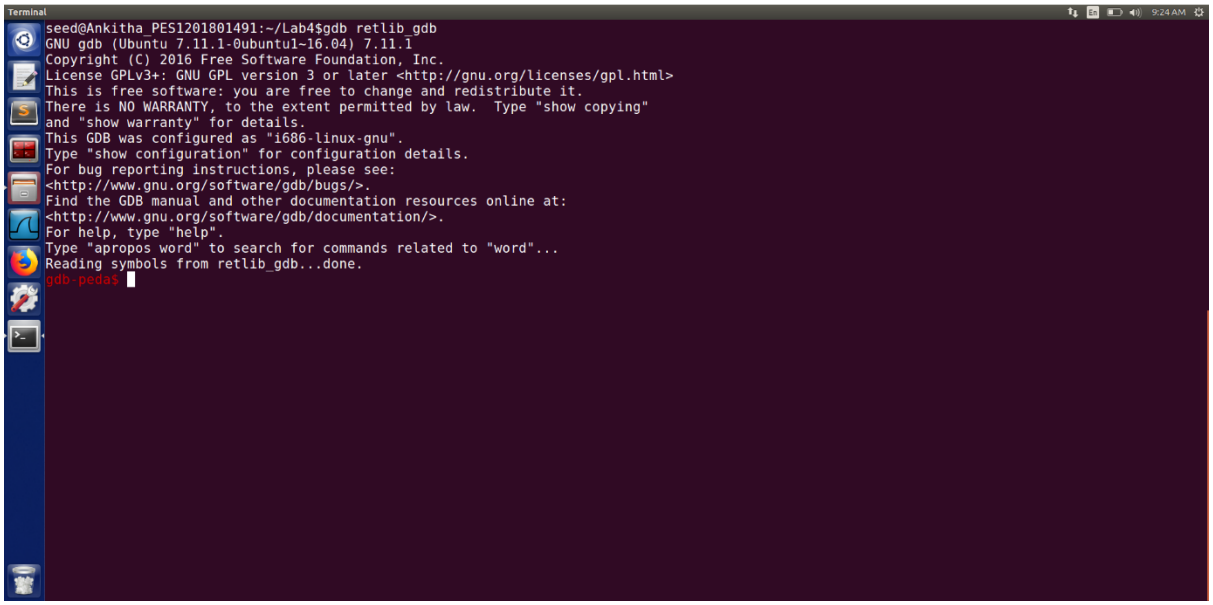
The debugging process with a breakpoint at the bof:



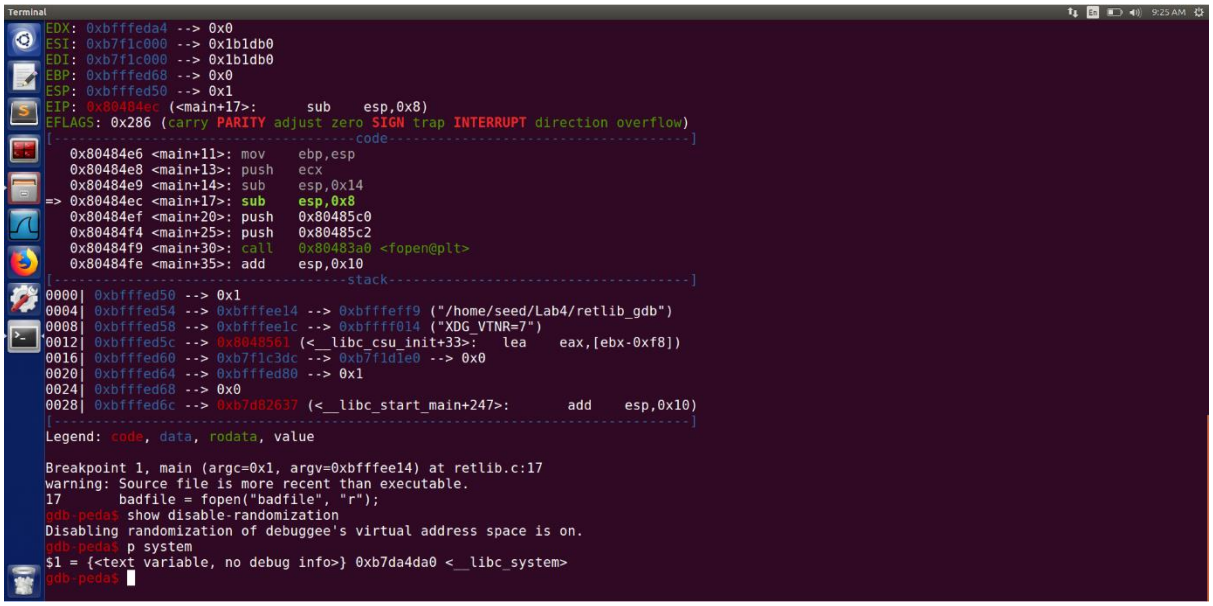
Inside the gdb debugger, we run "show disable-randomization" to see whether the randomization is turned off or not. We can see that the gdb debugger has disabled address randomization.



The debugging process with a breakpoint at the main:



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.



We observe that address of the system call is same for both the cases. This is because address randomization is disabled in the debugger. So even though the randomisation is enabled globally, gdb disables randomisation by default. If the address randomization is enabled in the debugger using “set disable-randomization on”, the addresses obtained in both the runs would have been different.