

# Lab5 Report

唐亚周 519021910804

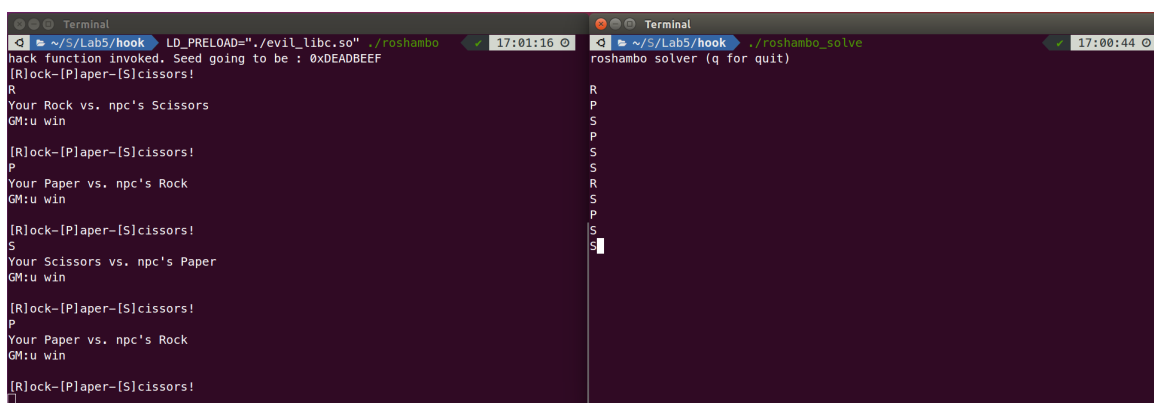
## 1 Hook

Q1: 在Hook的条件下, 设法赢得游戏 (生成一个可以赢得游戏的输入)

答: Hook 操作实际上是将标准库中的 `srand` 函数替换为了自定义的 `srand` 函数, 从而将输入的 seed 固定为 `0xdeadbeef`。这样的话, `rand` 函数的返回值也固定了。然后我们就可以根据其返回值来进行输入, 从而保证游戏胜利。我写了一个 `roshambo_solve.c`, 其主要代码如下。

```
1 char play()
2 {
3     unsigned char choice = -1;
4     switch (rand()%3)
5     {
6         case 0:
7             choice = 'P';
8             break;
9         case 1:
10            choice = 'S';
11            break;
12        case 2:
13            choice = 'R';
14            break;
15    }
16    return choice;
17 }
```

运行结果如下, 可以从右边的程序中不断获得下一步的选择, 再将其输入到左边的程序中。



```
Terminal
~/S/Lab5/hook LD_PRELOAD="./evil_libc.so" ./roshambo 17:01:16
hack function invoked. Seed going to be : 0xDEADBEEF
[R]ock-[P]aper-[S]cissors!
R
Your Rock vs. npc's Scissors
GM:u win

[R]ock-[P]aper-[S]cissors!
P
Your Paper vs. npc's Rock
GM:u win

[R]ock-[P]aper-[S]cissors!
S
Your Scissors vs. npc's Paper
GM:u win

[R]ock-[P]aper-[S]cissors!
P
Your Paper vs. npc's Rock
GM:u win

[R]ock-[P]aper-[S]cissors!

Terminal
~/S/Lab5/hook ./roshambo_solve 17:00:44
roshambo solver (q for quit)
R
P
S
P
S
R
S
P
S
S
```

Q2: Linux下动态链接与静态链接的区别?

答: 静态链接会在将所有使用到的库模块与程序编译产生的目标文件进行链接后, 产生可执行文件。而动态链接会在程序运行时同时将不包含库模块的可执行文件和用到的动态链接库加载到内存中。

Q3: LD\_PRELOAD的作用?

答: `LD_PRELOAD` 允许用户定义在程序运行前优先加载的动态链接库。这个功能主要就是用来有选择性的载入不同动态链接库中的相同函数。通过这个环境变量, 我们可以在主程序和其动态链接库的中间加载别的动态链接库, 甚至覆盖正常的函数库。一方面, 我们可以以此功能来使用自己的或是更好的函数(无需别人的源码), 而另一方面, 我们也可以以向别人的程序注入程序, 从而达到特定的目的。<sup>1</sup>

Q4: 试分析延时绑定(Lazy Bind)的过程 (Tips, `_dl_runtime_resolve`)

答<sup>2</sup>: 延迟绑定是当函数第一次被调用的时候才进行绑定(包括符号查找、重定位等), 如果函数不被调用就不进行绑定。延迟绑定机制可以大大加快程序的启动速度, 特别有利于一些引用了大量函数的程序。首先需要介绍 GOT 表和 PLT 表的概念。

GOT (Global Offset Table, 全局偏移表)

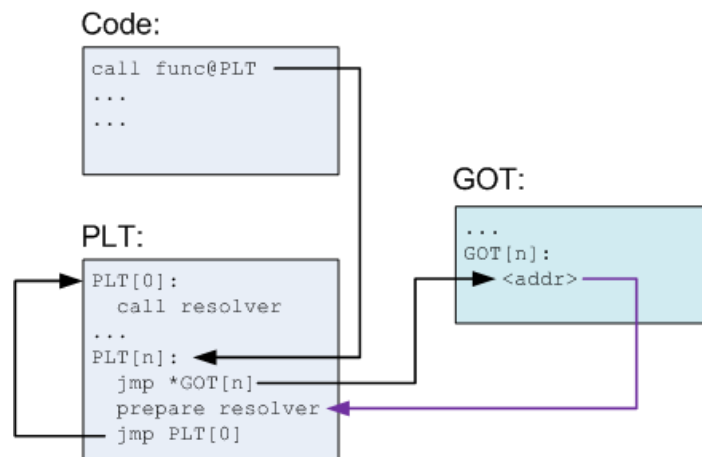
GOT 是数据段用于地址无关代码的 Linux ELF 文件中确定全局变量和外部函数地址的表。ELF 中有 `.got` 和 `.plt.got` 两个 GOT 表, `.got` 表用于全局变量的引用地址, `.got.plt` 用于保存函数引用的地址。

PLT (Procedure Linkage Table, 程序链接表)

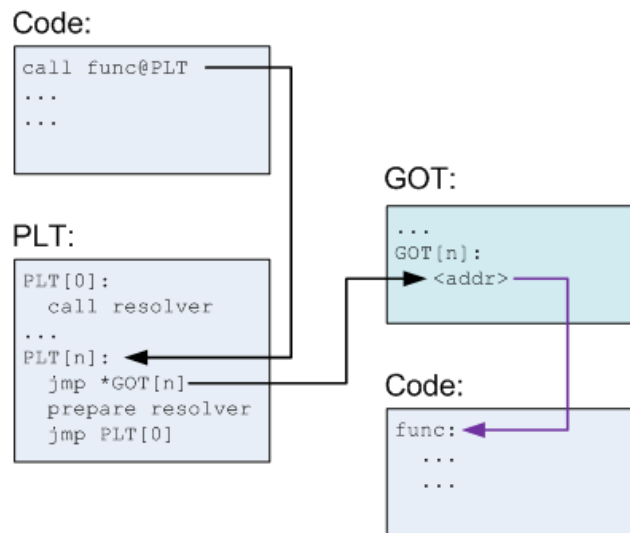
PLT 是 Linux ELF 文件中用于延迟绑定的表。

假设程序中调用 `func` 函数, 该函数在 `.plt` 段中相应的项为 `func@plt`, 在 `.got.plt` 中相应的项为 `func@got`, 链接器在初始化时将 `func@got` 中的值填充为“prepare resolver”指令处的地址。

第一次调用 `func` 函数时, 首先会跳转到 PLT 执行 `jmp *(func@got)`, 由于该函数没被调用过, `func@got` 中的值不是 `func` 函数的地址, 而是 PLT 中的“prepare resolver”指令的地址, 所以会跳转到“prepare resolver”执行, 接着会调用 `_dl_runtime_resolve` 解析 `func` 函数的地址, 并将该函数真正的地址填充到 `func@got`, 最后跳转到 `func` 函数继续执行代码。



当再次调用 `func` 函数时, 由于 `func@got` 中已填充正确的函数地址, 此时执行 PLT 中的 `jmp *(func@got)` 即可成功跳转到 `func` 函数中执行。



Q5: 尝试Hook其它函数以达到赢得游戏的目的

答：还可以直接 Hook `rand` 函数，让它始终输出 0（Rock），这样我们就可以通过一直输入 P 来获得胜利。将其编译为动态链接库，并使用 `LD PRELOAD` 进行加载。结果如下。

```
Terminal
~ /S/Lab5/hook LD_PRELOAD="./evil_libc_rand.so" ./roshambo
[R]ock-[P]aper-[S]cissors!
P
Your Paper vs. npc's Rock
GM:u win

[R]ock-[P]aper-[S]cissors!
P
Your Paper vs. npc's Rock
GM:u win

[R]ock-[P]aper-[S]cissors!
P
Your Paper vs. npc's Rock
GM:u win

[R]ock-[P]aper-[S]cissors!
P
Your Paper vs. npc's Rock
GM:u win

[R]ock-[P]aper-[S]cissors!
P
Your Paper vs. npc's Rock
GM:u win

[R]ock-[P]aper-[S]cissors!
```

## 2 Race condition

运行结果如下，可以看到读取到了 flag。

[illegible]

1. 分析rc程序源代码，解释其中的漏洞原因。

答：由于该程序同时运行了两个线程，并且这两个线程都会对全局变量 `index` 做 +1 的操作，但程序中只对 `index` 变量是否等于 30 进行了检验。因此有可能在两次检验之间，`index` 的值从小于 30 增加到了大于 30，从而通过了检验，读取到了 flag。

2. 如果将源代码中的两处usleep函数去掉，结果会怎样，为什么？

答：运行结果如下，可以看到程序陷入了死循环，并且没有读取到 flag。

[illegible]

原因如下:

- 如果去掉了 `usleep` 函数，由于循环体内只有一个 `print` 函数，耗时很短，因此在一个循环的时间内，两个线程同时增加 `index` 的值的概率很小，所以基本上不可能通过检验。
- 如果没有去掉 `usleep` 函数，那么一个线程在 sleep 时，另一个线程很有可能增加了 `index` 的值，从而通过检验。

3. 请你为上述程序打上补丁，避免此漏洞的发生

答：可以在操作前后加上 Mutex，使得一个进程在操作的同时，另一个进程会被阻塞，避免了一个循环内多次增加的情况出现。修改后的部分代码如下。

```

1  .....
2
3  pthread_mutex_t mutex;
4
5  void *t2(void *arg)
6  {
7      puts("break in!");
8      for(;;)
9      {
10         {
11             if (index == 30)
12                 index = 0;
13             if (pthread_mutex_lock(&mutex) != 0) {
14                 fprintf(stdout, "lock error\n");
15             }
16             usleep(rand()%2);
17             printf("%d%c", *(int*)arg, content[index++]);
18             usleep(rand()%2);
19             pthread_mutex_unlock(&mutex);
20         }
21         if(index > 60)
22         {
23             puts("get it");
24             exit(0);
25         }

```

```
26     }
27     puts("t2 exit");
28 }
29 int main()
30 {
31     if (pthread_mutex_init(&mutex, NULL) != 0) {
32         return 1;
33     }
34     .....
```

4. 解释race condition漏洞产生的根本原因以及比较通用的解决办法(选做)

答：根本原因：多个进程/线程同时访问和操作相同的数据。

通用的解决方法：增加检查次数、将操作原子化、加互斥锁等等。

---

1. <https://www.cnblogs.com/net66/p/5609026.html> 

2. <http://0x4c43.cn/2018/0429/linux-lazy-binding-mechanism/> 