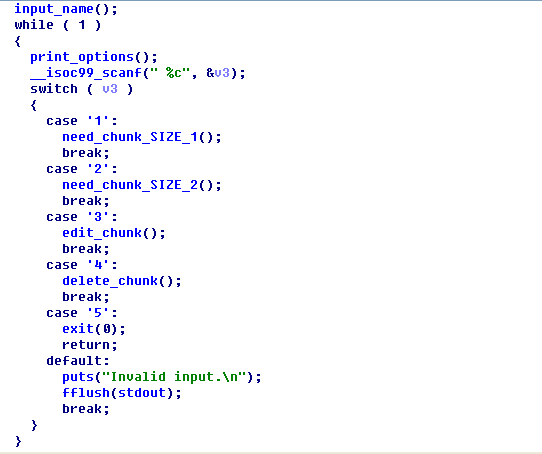
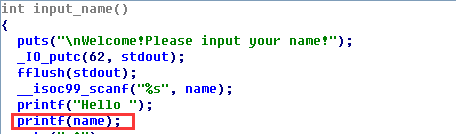
**一. 题目描述：**



**二. 漏洞分析：**

1. 在input\_name()函数中，存在格式化字符串漏洞，可以用于泄露so库基址，因此得到system函数的地址



2.   **知识点一：堆分配**

          堆分配的大小会自动填充到最接近的2的n次幂，如要求分配124字节，系统会分配128字节，其中包括前8字节固定的堆结构，后面的120字节才是用户真正合法的数据写入区，如果用户写入的数据超过了120字节，则视为堆溢出

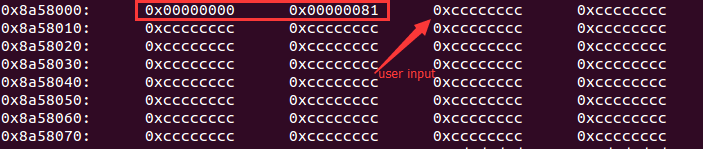
**知识点二：32bit下堆结构**

 使用中的堆块：

|  |  |
| --- | --- |
| 4字节  pre\_size | 前一块大小（前一块若是空闲，则pre\_size=前一块大小；前一块若是被分配状态，则pre\_size=0） |
| 4字节  size & pre\_flag | 本块大小，因为本块大小肯定是8的整数倍，所以后三位一定会空闲出来作为三个flag（最低位:指示前一个chunk是否正在使用;倒数第二位:指示这个chunk是否是通过mmap方式产生的;倒数第三位:这个chunk是否属于一个线程的arena） |
| （堆长度-8）字节 | 用户合法使用 |

空闲列表中的堆块：

|  |  |
| --- | --- |
| 4字节  pre\_size | pre\_size = 上一个空闲块大小 |
| 4字节  size & pre\_flag | size = 本块大小，pre\_flag =  0 |
| 4字节 | 前向指针fd |
| 4字节 | 后向指针bk |
| （堆长度-16）字节 |  |



      如图所示，用户malloc(124)，实际上系统分配了128个字节，前4字节0x00000000表示上一块不是空闲块，第二个4字节 0x00000081 表示本堆块大小为 0x80=128，最后一位为1，表示上一块不是空闲块。用户的输入是从 第三个4字节 开始存放起走的。

      need\_chunk\_SIZE\_1()，need\_chunk\_SIZE\_2()函数  分别是申请124和248个字节的堆空间，实际系统将分配128和256字节。

      edit\_chunk()函数分别可以对堆块进行编辑，分别可以往堆块中写入124和248字节的数据。

C:\Users\Carter\AppData\Local\YNote\data\m_gj1993@163.com\0faba661fa04470a8c882f5fc67b3c94\clipboard.png

C:\Users\Carter\AppData\Local\YNote\data\m_gj1993@163.com\18567c917da84af89f812233b7e22491\clipboard.png

漏洞出现在：堆块一系统实际分配128字节，实际上合法的可供用户写入的只有128-8=120字节，edit\_chunk函数允许用户往堆块1中写入124字节，刚好可以覆盖堆块二结构的前4个字节，同时scanf函数自动会在末尾加上 \x00，正好修改堆块二 第二个4字节中 的 pre\_flag从1变为0，导致出现堆块1是空闲块的假象。当释放堆块二时，系统检查到堆块二的前一块（即堆块一）是空闲的，因此会将其当做是在空闲列表中的块，将其解链，执行unlink操作。如果在堆块一中事先精心布置好数据，利用unlink就达到任意内存写。在此将free函数的got表改为system函数的地址

**三. 漏洞利用：**

1. 首先执行 need\_chunk\_SIZE\_1()，need\_chunk\_SIZE\_2()，need\_chunk\_SIZE\_1(），分配三块内存

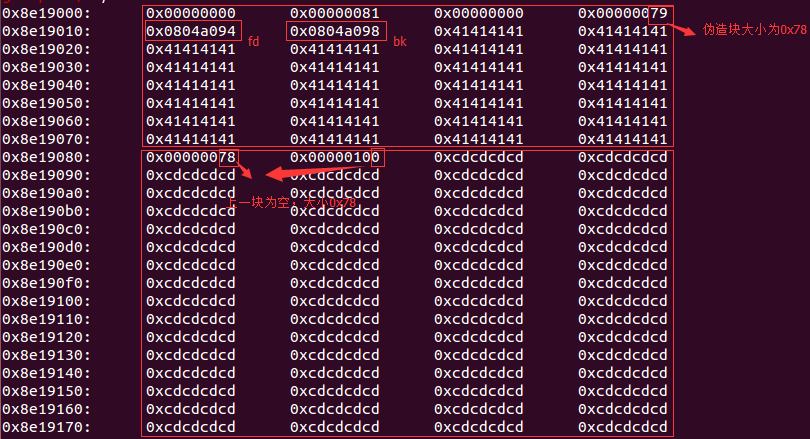
C:\Users\Carter\AppData\Local\YNote\data\m_gj1993@163.com\f218573de96b48d18a6cbe099b4623d3\clipboard.png

每次malloc（）函数分配结束后，返回堆块起始位置到 $EAX。注意下一条指令，这条指令的意思是将堆块的起始地址记录到堆管理结构中。通过计算知道堆块管理结构为：0x804a0a0

当分配完三块内存时，查看堆块管理结构的情况如下：挨个记录了三个堆块的起始位置

C:\Users\Carter\AppData\Local\YNote\data\m_gj1993@163.com\99910d23c5384467abdf0da801052256\clipboard.png

2. 往第三个堆块上写入 /bin/sh ，为最后调用system做好准备

C:\Users\Carter\AppData\Local\YNote\data\m_gj1993@163.com\e31c73ae527f454692383dbb88f751e7\clipboard.png

3. 往第一个块中写入精心构造的数据

4. 释放第二块，根据此时内存数据，如上图，判断第二块前一块为空闲块，因此将第一块执行unlink操作（目的是改变堆管理结构0x804a0a0的值）

    1. BK = p ->bk = 0x804a098 = 0x804a0a0 - 0x8

    2. FD = p ->fd =  0x804a094 = 0x804a0a0 - 0xc

    3. 判断 BK->fd=(0x804a0a0-0x8)+0x8=0x804a0a0  和    FD->bk=(0x804a0a0-0xc)+0xc=0x804a0a0  是否相等，在此通过验证

    4. \*（FD->bk） =BK 即   \*0x804a0a0 **=**0x804a0a0 - 0x8

    5. \*（BK ->fd） =FD    即   \*0x804a0a0 =   0x804a0a0 - 0xc

最终结果就是将堆管理结构首地址内容改写为 0x804a0a0 - 0xc = 0x804a094

C:\Users\Carter\AppData\Local\YNote\data\m_gj1993@163.com\17bf6dfda22a44ffbf3dafc38a7dcb4c\clipboard.png

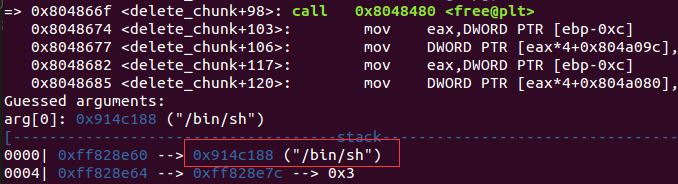
5. 往堆块1中写入数据，填充12个'A'之后，将 0x804a0a0地址内容写为：我们即将要改变内容的地址 [free@got](mailto:free@got) = 0x804a014，下次再向堆块一中写内容时，就等价于向 [free@got](mailto:free@got) 表地址写内容，达到改写got表的目的

C:\Users\Carter\AppData\Local\YNote\data\m_gj1993@163.com\657b606960ee4889a6f39972e4c528ef\clipboard.png

6. 将system函数的地址写入第一块堆块，改写[free@got](mailto:free@got) = system函数地址

C:\Users\Carter\AppData\Local\YNote\data\m_gj1993@163.com\1f5a61b01e78491d9c55b7f6f901bada\clipboard.png

7. 释放第三块，调用free函数，参数/bin/sh早已在第三块上布置好，得到shell



最终exp：

参考链接：

<https://sploitfun.wordpress.com/2015/06/09/off-by-one-vulnerability-heap-based/>

<http://blog.chinaunix.net/uid-20565779-id-1633969.html>

<http://drops.wooyun.org/tips/7326#!>