堆溢出

原理：

漏洞的主要原因来源于以下几个问题：

1. 为了节约内存，被使用之后的chunk和未使用的chunk的内存布局不相同，但是都用了相同的大小，于是free chunk具有更多的数据。

2. glibc的堆空间控制是用链表处理的，其中除了fastbin（bin可以认为是链表的头结点指针，用来标志不同的链表），都使用了双向链表的结构，即使用fd和bk指针指向前者和后者，这恰巧是free chunk才有的额外数据。

3. 在分配或是合并的时候需要删除链表中的一个结点，学过数据结构应该很清楚其操作，大概是P->fd->bk = P->bk; P->bk->fd = P->fd;,而在做这个操作之前会有一个简单的检查，即查看P->fd->bk == P && P->bk->fd= == P,但是这个检查有个致命的弱点，就是因为他查找fd和bk都是通过相对位置去查找的，那么虽然P->fd和P->bk都不合法，但是P->fd->bk和P->bk->fd合法就可以通过这个检测，而在删除结点的时候就会造成不同的效果了。

漏洞程序：

void Malloc();

void Free();

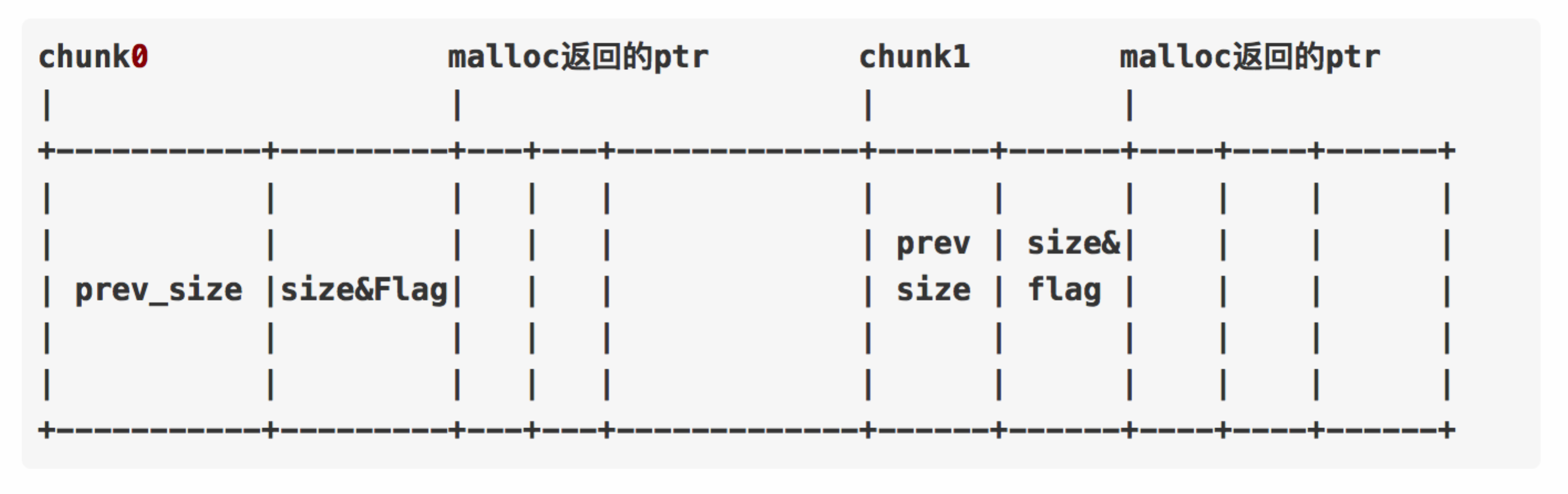
void Edit();

void Print();

漏洞程序主要含有上面四个函数，作用分别是创建堆块，删除堆块，修改堆块内容，打印堆块内容。

攻击程序：

1. 我们可以首先分配两个长度合适的堆块。



这时候这两块的fd和bk区域其实都是空的，因为他们都是正在使用的

1. 对第一块进行编辑，编辑的过程中设置好第零块的bk和fd指针并溢出第一块，改好第一块的chunk头的控制信息(如下图所示)



我们为了欺骗glibc，让它以为堆块零malloc返回的指针(我们后文中简记为p)出就是chunk0指针，所以我们伪造了prev\_size和size的部分，然后溢出堆块1，改掉第1个堆块的prev\_size,数值应该是上图所示 new\_size 的大小；另外第1块的size部分还要把prev\_inuse的flag给去掉。如此就做好了unlink触发之前的准备工作。

3.删掉chunk1,触发unlink(p)，将p给改写。

在删除堆块1时，glib会检查一下自己的size部分的prev\_inuse FLAG，发现到到比较早的一个chunk是空闲的(实际是我们伪造的)，glibc希望将即将出现的两个空闲块合并。glibc会先将chunk0从它的Binlist中解引用，所以触发unlink(p)。

4.对p再次写入，修改p为free@got地址

5.现在p已经是free@got了，我们只要使用一次List功能便可以知道free函数的真实地址，进而算出libc的基址来过掉ASLR。

6.根据已经算出的libc基址再次算出system函数的真实地址。向p再次写入便可以将free@got改为system函数。 (如果没有libc，可以考虑简历多个chunk，在改p为free@got时候将其他的堆块的指针也改为libc里面的函数，这样在list时，我们可以得到两个libc函数的真实地址，根据其偏移，便可以找出服务器上的libc，若保护再够复杂无法改got，我们还可以构造ropchain，同样利用这样的方式，把ropchain丢进全局数组中)

7.因为free已经变成了system，只要再建立一个内容为 /bin/sh 的块，再删掉，就可以得到shell，由此全部利用完成。