

2019-2020学年秋季学期

## 漏洞利用与攻防实践

*Exploiting Software Vulnerability-  
Techniques and Practice*

授课教师：霍玮

助    教：邹燕燕

## 漏洞利用与攻防实践

*Exploiting Software Vulnerability-Techniques and Practice*

# [第四讲]堆及堆管理机制概述

授课教师：霍玮

授课时间：2019-9-17

## 提纲

### 一 堆简介

1.1 堆的基本介绍

1.2 堆与栈

### 二 glibc中堆的实现 ( ptmalloc )

2.1 程序员的内存-chunk介绍

2.2 chunk的组织-bin介绍

2.3 bin管理与并发-arena介绍

### 三 windows中堆的实现

3.1 windows堆的历史

3.2 堆的数据结构和管理策略

3.3 堆分配

3.4 堆攻击

## 提纲

### 一 堆简介

1.1 堆的基本介绍

1.2 堆与栈

### 二 glibc中堆的实现 ( ptmalloc )

2.1 程序员的内存-chunk介绍

2.2 chunk的组织-bin介绍

2.3 bin管理与并发-arena介绍

### 三 windows中堆的实现

3.1 windows堆的历史

3.2 堆的数据结构和管理策略

3.3 堆分配

3.4 堆攻击

## Linux内存结构

以32位程序为例，

一个程序可见的逻辑地址空间是4G。

Linux下通常高地址的1个G是内核空间。ELF文件的主要区段在低地址空间。

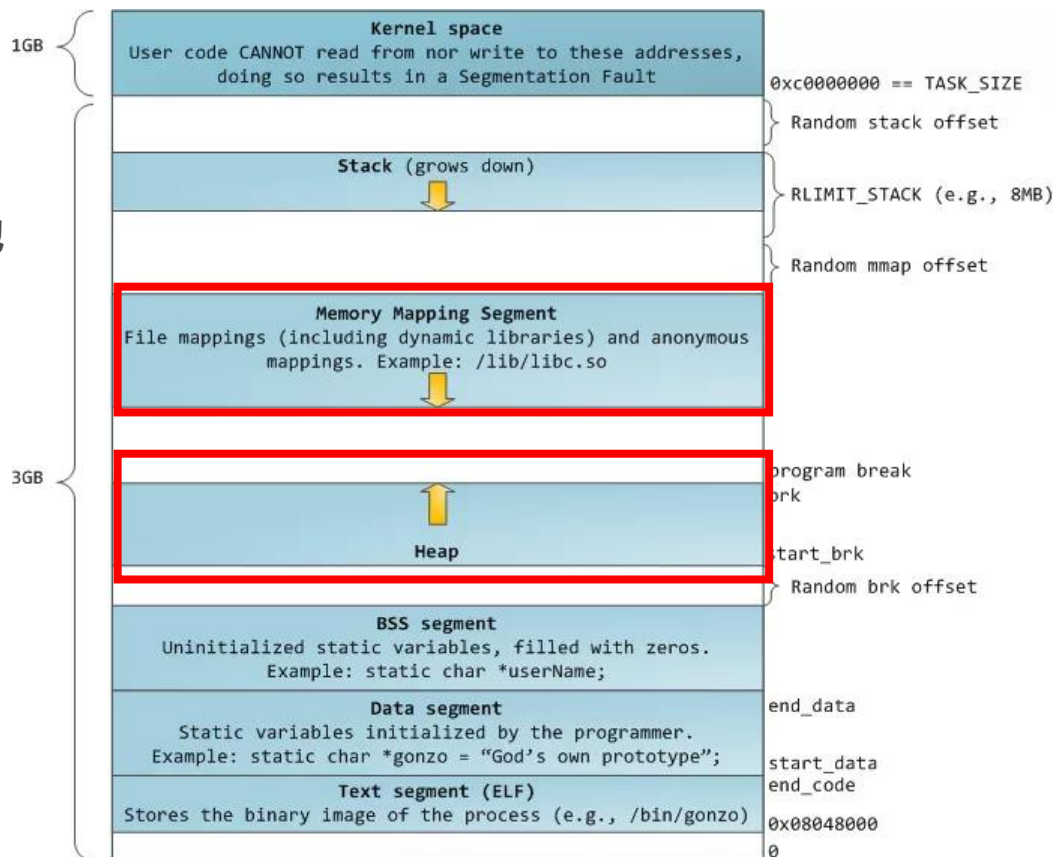
中间夹着堆和栈。

栈从高地址往低地址生长。

堆则是从低地址往高地址生长。

早期这样设计的原因是为了使得堆栈能够使用的空间尽可能大一些。

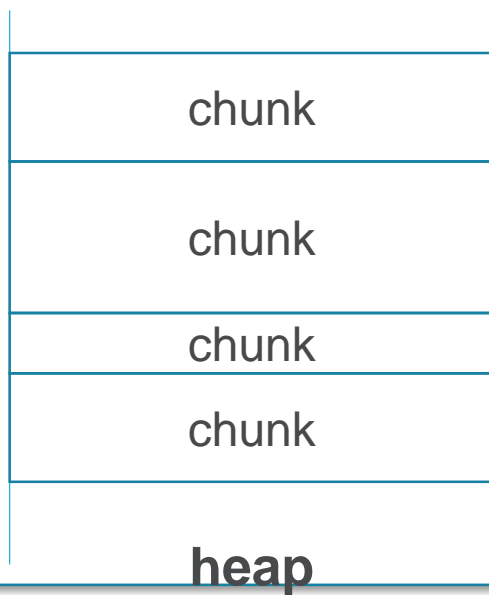
X64程序基本沿袭了这样的内存布局。



# 一堆简介

## 堆

- 堆是可由程序较自由地对内存进行操作的地方。堆上供程序操作的单元叫chunk。
- 众多的chunk是需要进行管理的，在不同的系统对堆的实现略有差异，但基本思想是类似的。



## 堆与栈

	堆内存	栈内存
典型用例	动态增长的链表等数据结构	函数局部数组
申请方式	需要函数申请，通过返回的指针使用。如： <code>p=malloc(8);</code>	在程序直接声明即可，如： <code>char buffer[8];</code>
释放方式	需要把指针传给专用的释放函数	函数返回时由系统自动回收
管理方式	需要程序员处理申请与释放	申请后直接使用，申请与释放由系统自动完成，最后达到栈平衡
所处位置	变化范围很大	0x0012xxxx
增长方向	由内存低地址向高地址排列（不考虑碎片等情况）	由内存高地址向低地址增加

## 提纲

### 一 堆简介

1.1 堆的基本介绍

1.2 堆与栈

### 二 glibc中堆的实现 ( ptmalloc )

2.1 程序员的内存-chunk介绍

2.2 chunk的组织-bin介绍

2.3 bin管理与并发-arena介绍

### 三 windows中堆的实现

3.1 windows堆的历史

3.2 堆的数据结构和管理策略

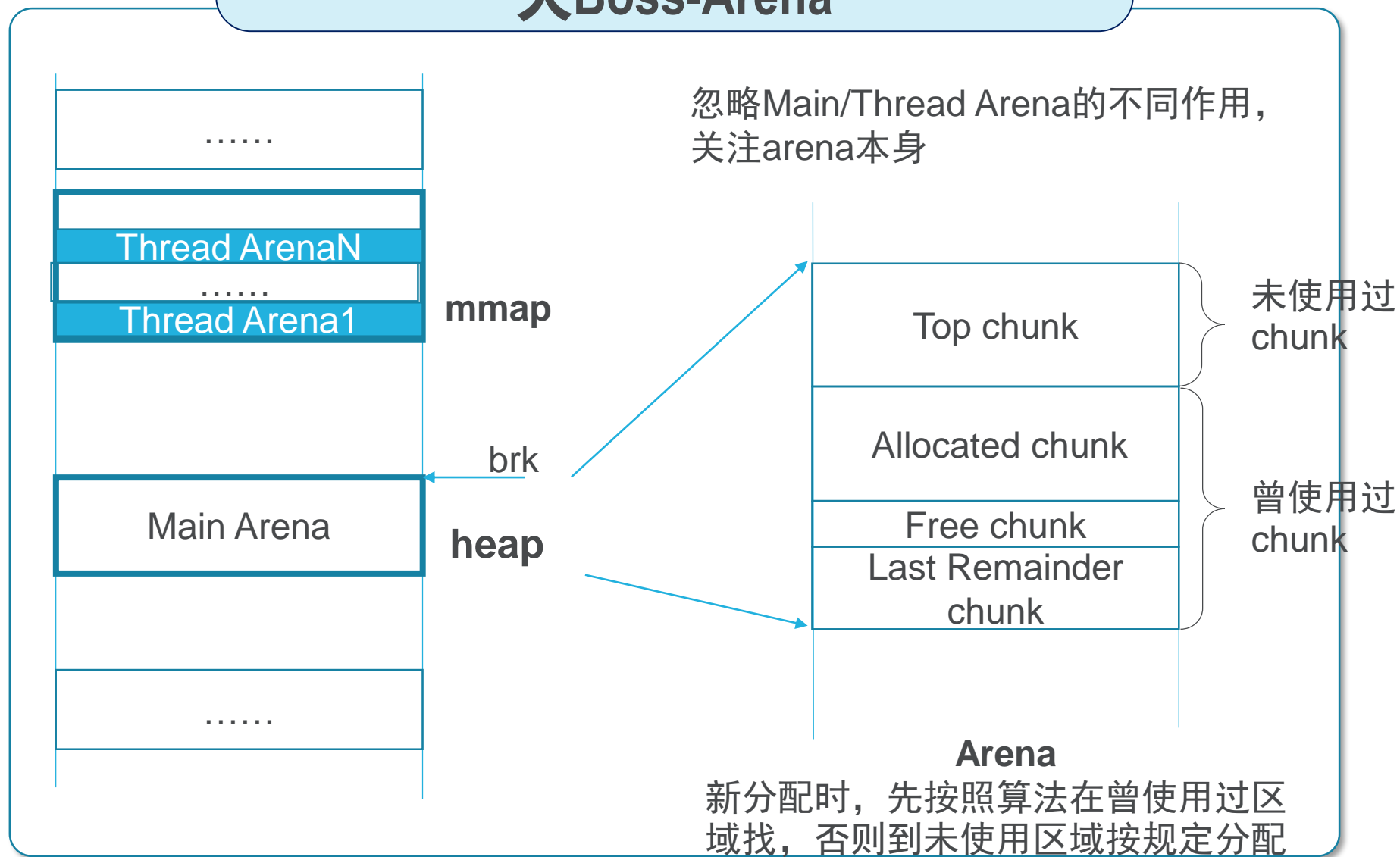
3.3 堆分配

3.4 堆攻击



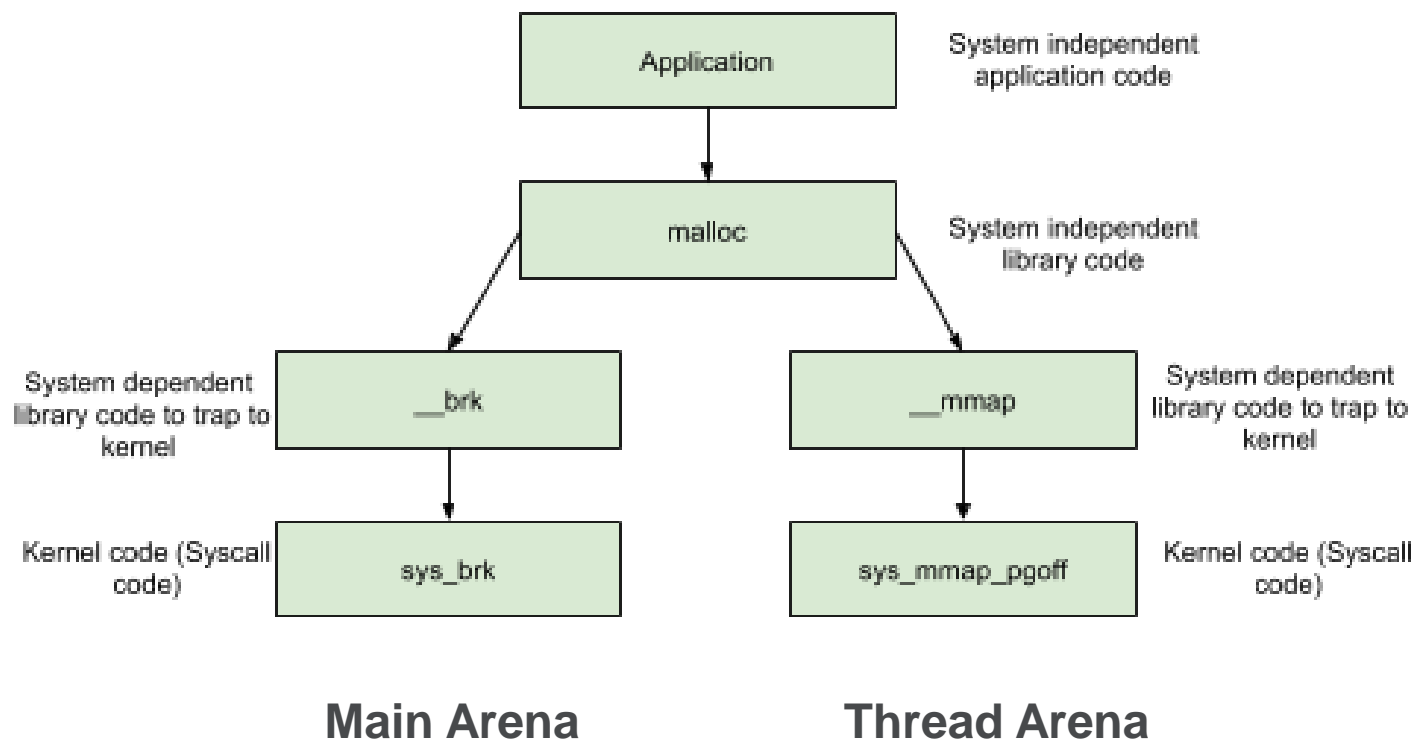
## 二 glibc中堆的介绍

### 大Boss-Arena



## 二 glibc中堆的介绍

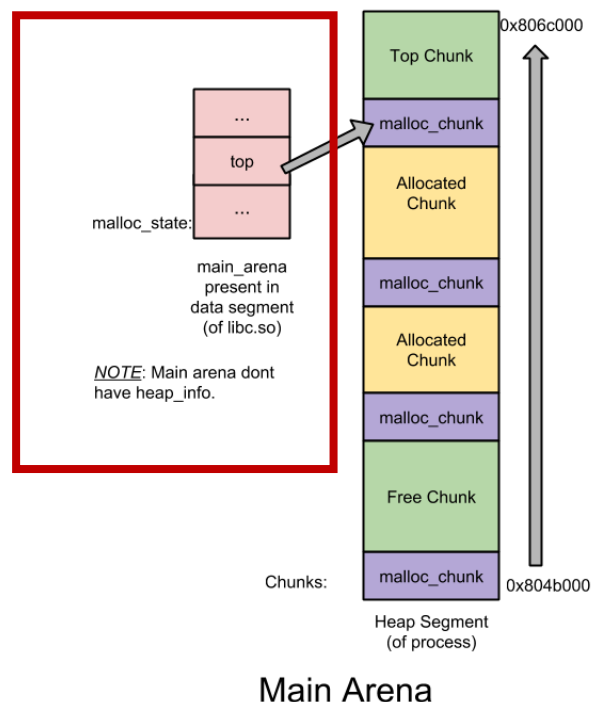
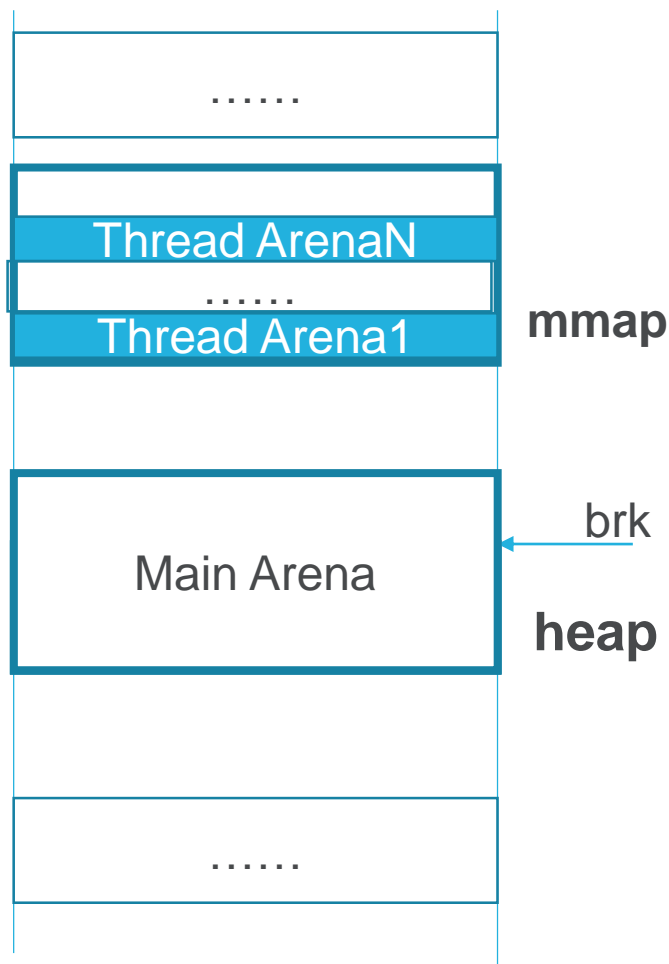
### Arena分配



## 二 glibc中堆的介绍

### Arena管理与增长-Main Arena

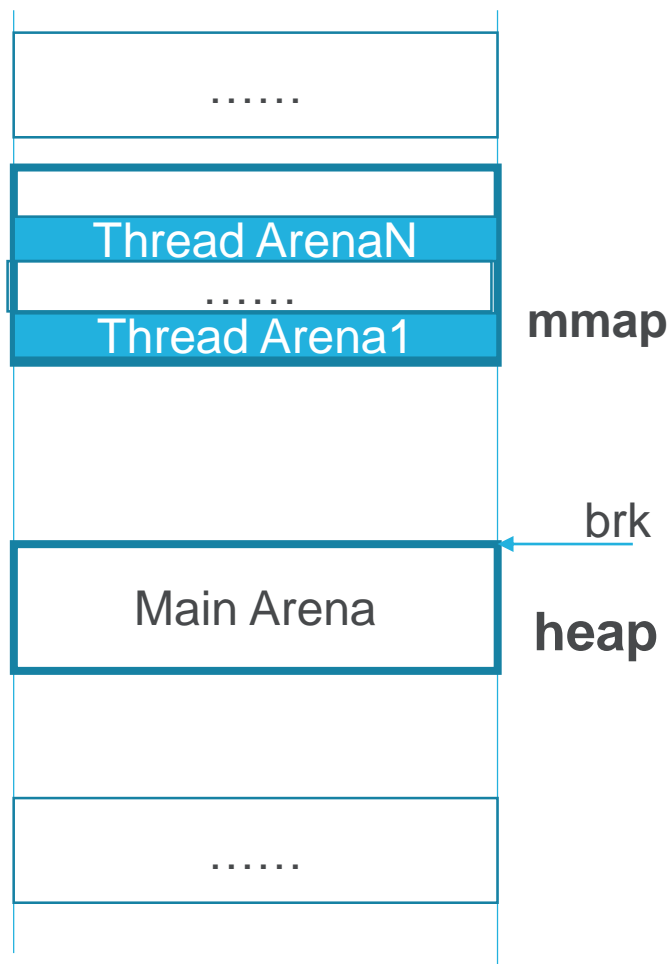
在设计上，main arena的malloc\_state是位于libc.so映射数据段中的一个全局变量。



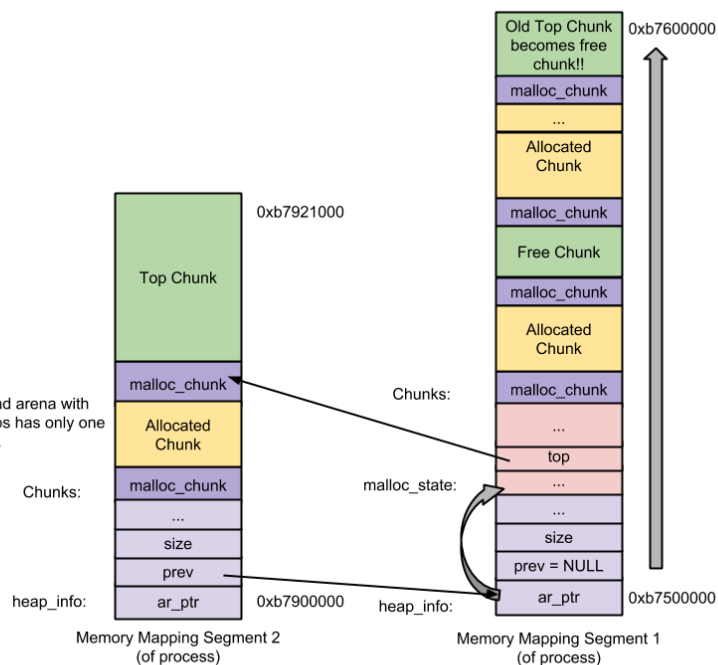
## 二 glibc中堆的介绍

### Arena管理与增长-Thread Arena

在设计上，thread arena的增长方式是使用mmap再分配一个segment。使用malloc\_state管理arena，使用heap\_info管理segment



*NOTE:* Thread arena with multiple heaps has only one malloc\_state.

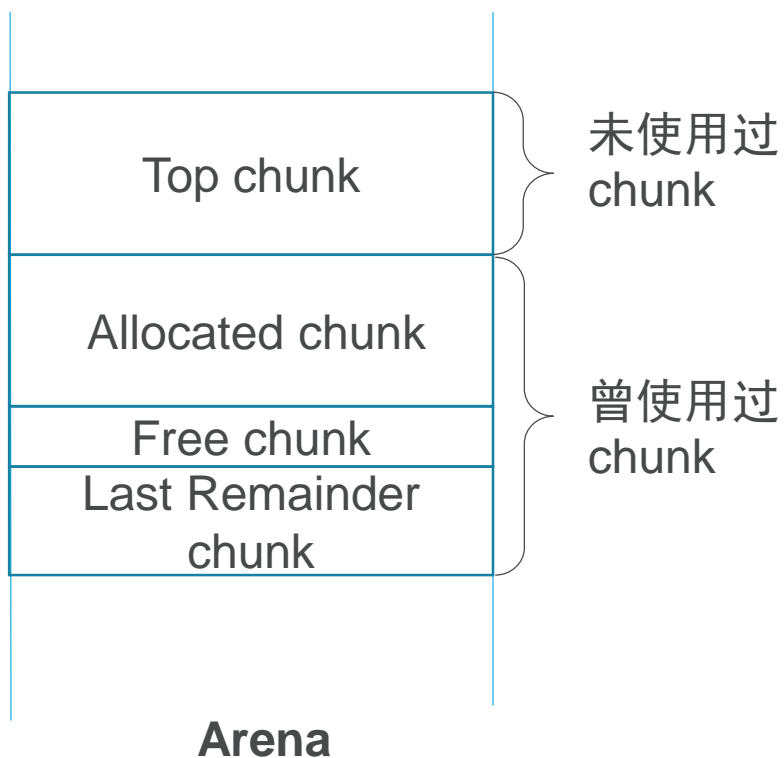


Thread Arena (with multiple heaps)

## 二 glibc中堆的介绍

### 员工-chunk

每次内存分配(malloc), 返回给用户的是一个"chunk"(不准确的说)



### Chunk共分为4类:

1. 已经分配的叫allocated chunk
2. 已经释放(意味着曾经分配过)的叫free chunk
3. 等待第一次分配的叫top chunk
4. 已经使用过的chunk中, 因为分配而分割剩下的, 叫last remainder chunk

### 内存分配基本流程:

1. 首先在已经释放的chunk中, 按算法找大小合适的
2. 如果已经释放的chunk中都比所需要的小, 在top chunk中分配
3. 如果top chunk比需要的小, 通过增长arena进而增长top chunk, 直到满足需要

## 二 glibc中堆的介绍

### Chunk的结构

首先，我们来看看在我们请求堆分配的时候，系统给我们的到底是怎么样一个数据。

不妨做一个小实验：下面这个32位单线程程序演示了一下通过计算返回地址指针的偏移量，看看每次malloc得到的地址大小究竟是多少。（为什么可以假设这些地址是连续的后面再详细解释）

由下可见，最小的分配大小是16字节，然后以8字节的大小递增。为什么这样呢？

说明申请下来的内存极可能不是简单一块内存，而是一个数据结构。这个数据结构中的数据部分是程序员申请用于操作的。这个结构体称为chunk。

```
void main(){
    char* a = malloc(0);
    char* b = malloc(4);
    char* c = malloc(8);
    char* d = malloc(12);
    char* e = malloc(16);
    char* f = malloc(20);
    char* g = malloc(24);
    printf("gap(a,b): %d \n", b-a);
    printf("gap(b,c): %d \n", c-b);
    printf("gap(c,d): %d \n", d-c);
    printf("gap(d,e): %d \n", e-d);
    printf("gap(e,f): %d \n", f-e);
    printf("gap(f,g): %d \n", g-f);
}
```

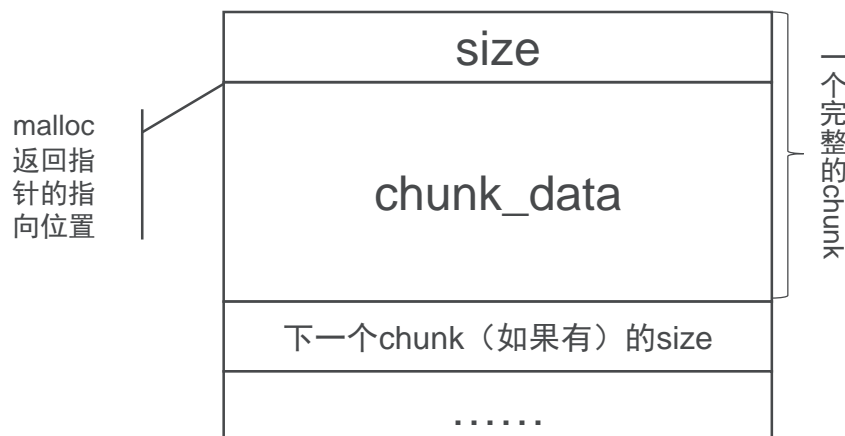
## 二 glibc中堆的介绍

### chunk的结构

chunk是一个系统能够进行管理的数据结构，**对于地址空间连续的chunk，使用隐式链表进行管理**。所以最小的空间开销也需要一个指针来实现单链表的结果。在glibc中，这个指针是一个指示当前区块大小的整型数，叫size。这个size字段是计算包含自身以及数据部分整个chunk的大小，这样，系统就可以沿着这个size字段往下寻找下一个size，从而知道哪些空间被使用了。

glibc的设计很巧妙，对于chunk的大小，glibc规定必须是8字节的倍数（我们也能从上图中看出来），所以size中的低3个比特就空余出来可以用作标志位。

结构如下所示，是一个正在使用的chunk（allocated chunk）的结构。



## 二 glibc中堆的介绍

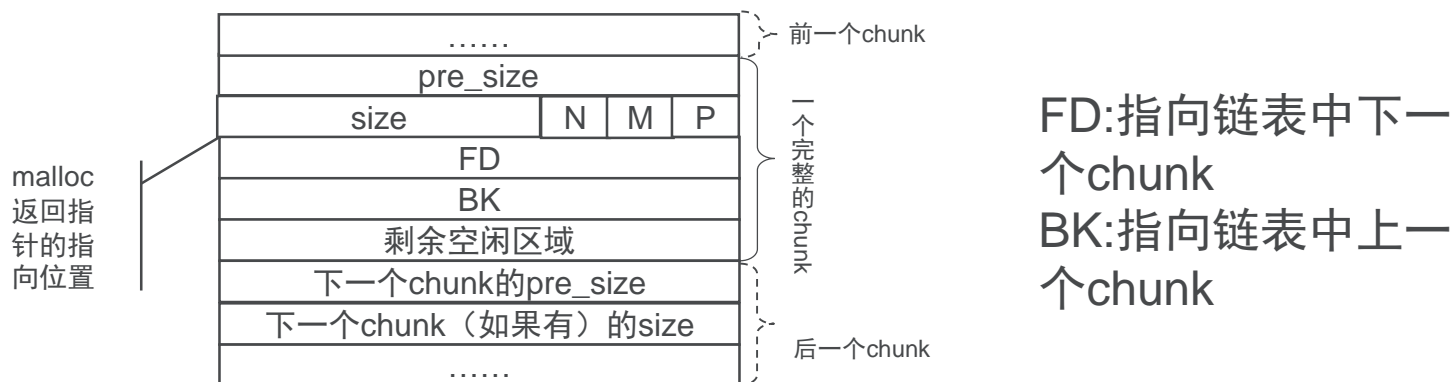
### free chunk的结构

对于已经释放的chunk叫做free chunk。考虑到碎片化的问题，系统需要有对连续free chunk进行整合的能力，这就需要当前chunk有一个双链表结构能够找到前后相邻的chunk的位置。

glibc中size字段的最低位是pre\_inuse位，表明前一个chunk是否在使用。在一个chunk为free的状态下，修改最后4个字节(紧挨下一个chunk)为chunk的大小，这个字段是给下一个chunk用的，称为pre\_size。这样，通过检测pre\_inuse标志位，再在当前chunk负4字节偏移的地方找到pre\_size，就实现了反向的查找。

同时由于free chunk本身的分散性，为了支持不连续chunk的记录与管理，free chunk需要显示支持双链表（例如后面会提到的fast bin），在free chunk中会添加两个叫做FD和BK的指针。

如下，就是free chunk的结构示意图：



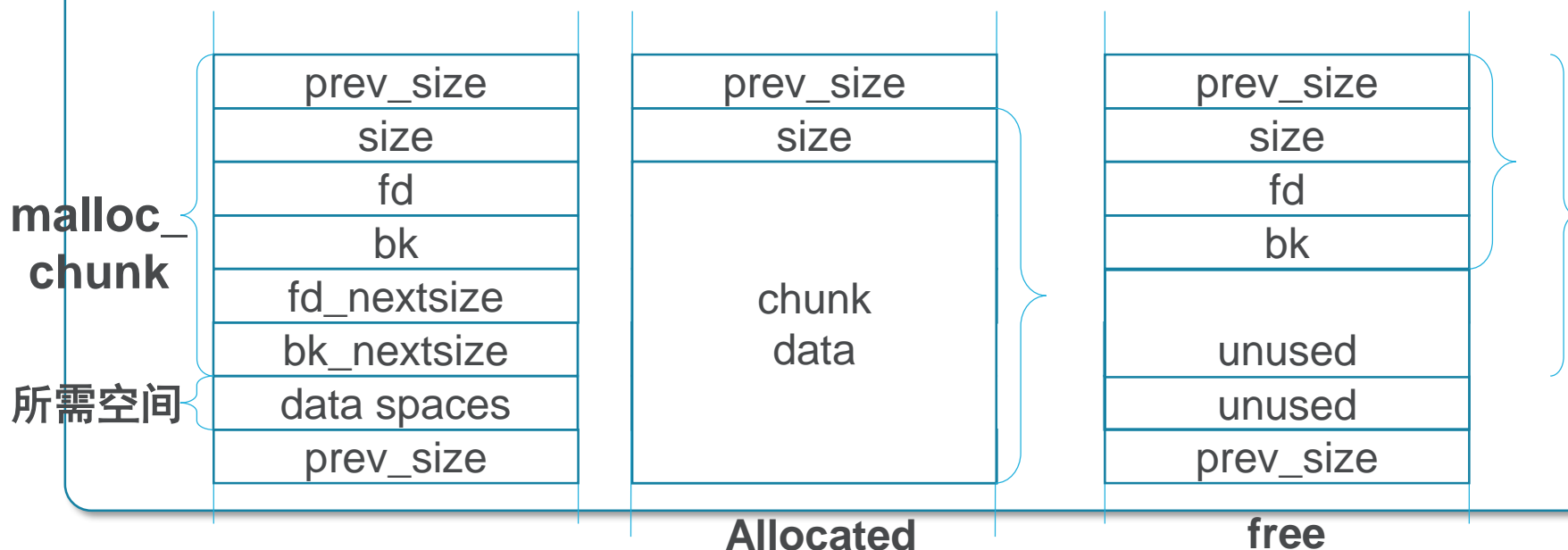


## 二 glibc中堆的介绍

### Chunk复用

为了最大化利用内存空间，chunk巧妙的通过空间复用提高内存利用效率

```
struct malloc_chunk { /* #define INTERNAL_SIZE_T size_t */
    INTERNAL_SIZE_T prev_size; /* Size of previous chunk (if free). */
    INTERNAL_SIZE_T size; /* Size in bytes, including overhead. */
    struct malloc_chunk* fd; /* double links -- used only if free. */
    struct malloc_chunk* bk;
    /* Only used for large blocks: pointer to next larger size. */
    struct malloc_chunk* fd_nextsize; // double links -- used only if free.
    struct malloc_chunk* bk_nextsize; };
```



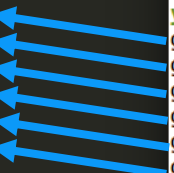
## 二 glibc中堆的介绍

### chunk的结构

所以，再回来看这个结果，由于size字段长度固定，两个指针相减的结果可以视作上一个chunk的大小。

所以最小chunk就是一个size占4字节，一个预留给FD和BK指针大小的8字节空间，再加上预留给pre\_size的4字节，一共16字节。在malloc（16）时，需要的空间大小变成了20字节 = 16字节数据+4字节的size。所以申请到的chunk大小变成了24字节。

```
void main(){
    char* a = malloc(0);
    char* b = malloc(4);
    char* c = malloc(8);
    char* d = malloc(12);
    char* e = malloc(16);
    char* f = malloc(20);
    char* g = malloc(24);
    printf("gap(a,b): %d \n", b-a);
    printf("gap(b,c): %d \n", c-b);
    printf("gap(c,d): %d \n", d-c);
    printf("gap(d,e): %d \n", e-d);
    printf("gap(e,f): %d \n", f-e);
    printf("gap(f,g): %d \n", g-f);
}
```



```
gap(f,g): 10
youngc@ubuntu:~/ycproject/college/heap$ gcc chur
youngc@ubuntu:~/ycproject/college/heap$ ./a.out
gap(a,b): 16
gap(b,c): 16
gap(c,d): 16
gap(d,e): 16
gap(e,f): 24
gap(f,g): 24
youngc@ubuntu:~/ycproject/college/heap$
youngc@ubuntu:~/ycproject/college/heap$
youngc@ubuntu:~/ycproject/college/heap$
youngc@ubuntu:~/ycproject/college/heap$
youngc@ubuntu:~/ycproject/college/heap$
youngc@ubuntu:~/ycproject/college/heap$
youngc@ubuntu:~/ycproject/college/heap$
youngc@ubuntu:~/ycproject/college/heap$
youngc@ubuntu:~/ycproject/college/heap$
```

## 二 glibc中堆的介绍

### Top chunk 和 last remainder chunk

除了allocated chunk 和 freed chunk两个提供功能的chunk类型外，还有两个用于系统分配效率提升的chunk：top chunk和remainder chunk

#### Top chunk:

它处于一个arena的最顶部。该chunk并不属于任何bin，而是在系统当前的所有free chunk都无法满足用户请求的内存大小的时候，将此chunk用户使用，具体是：

如果top chunk的大小比用户请求的大小要大的话，就将该top chunk分作两部分：1) 低地址部分作用户请求的chunk；2) 剩余的高地址部分成为新的top chunk。否则，就需要扩展heap或分配新的heap了——在main arena中通过sbrk扩展heap，而在thread arena中通过mmap分配新的heap。

*在前面的实验中，我们没有任何的freed chunk，所以所有新的malloc行为都从top chunk上最低地址处切割下来，所以在地址空间上就是连续的。*

#### Last remainder chunk:

它和unsorted bin 关系密切，简言之就是在一些分配过程中遗留下来的“边角料”，或者是很可能再次需要的空间。设计的目的在于提高内存分配的局部性。

## 二 glibc中堆的介绍

### 为什么要有bin

类似磁盘管理，如果在顺序分配内存地址后，没有有效的方式进行回收利用，就会造成碎片化和利用率低下。

glibc使用一组双向链表来存储已经free的chunk。再下一次分配时，如果已经free的chunk可以满足程序需求，那将会优先使用这些chunk。

bin成组出现，每组由共计136条双向链表构成，每组中都有如下的链表：

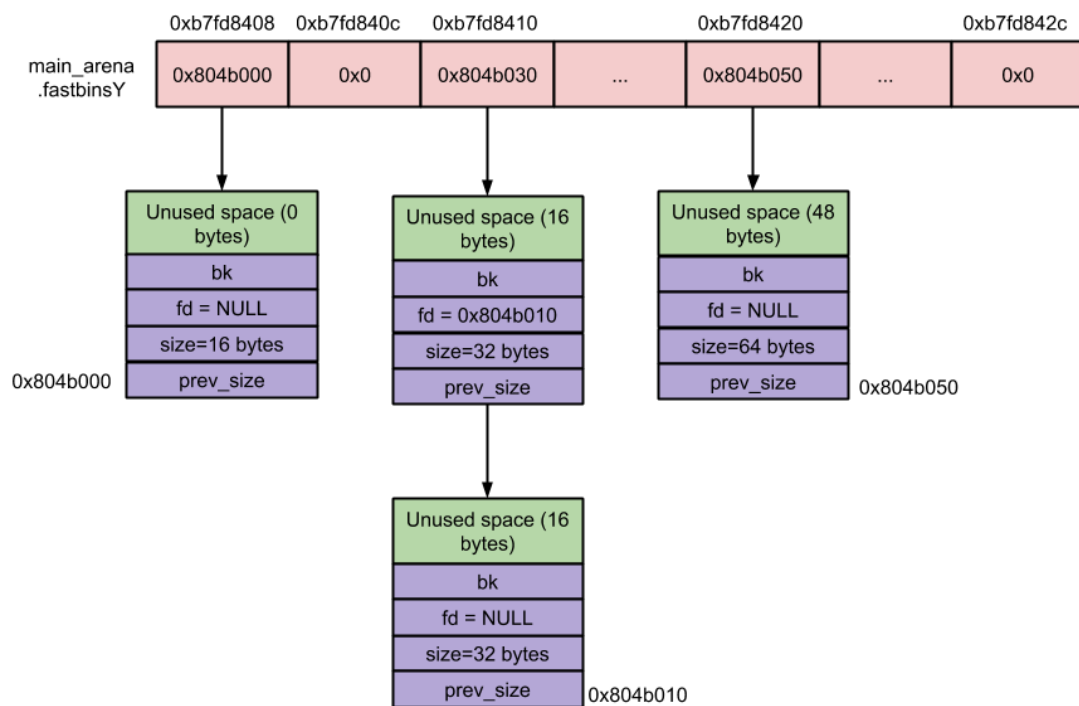
①fastbinY 指向10条fastbin的指针数组。

②bins

- 1 条unsorted bin 链
- 第2号到63号是small bin 链
- 第64到126号是large bin 链

# 二 glibc中堆的介绍

## fastbin——快速的小内存分配



- ① fastbinY中一共有10个fastbin的链表，每个链表的chunk大小都一致，分配的chunk的范围从0到80字节。
- ② 每个fast bin都是一个单链表(只使用fd指针)。在fast bin中无论是添加还是移除fast chunk，都是对“链表尾”进行操作，即LIFO(后入先出)算法。添加操作(free内存)就是将新的fast chunk加入链表尾，删除操作(malloc内存)就是将链表尾部的fast chunk删除。
- ③ fastbinsY数组中每个fastbin元素均指向了该链表的尾结点，而尾结点通过其fd指针指向前一个结点。
- ④ 不会对free chunk进行合并操作。设计fastbin的初衷就是进行快速的小内存分配和释放，因此系统将属于fast bin的chunk的P(pre\_inuse)总是设置为1，这样即使当fast bin中有某个chunk同一个free chunk相邻的时候，系统也不会进行自动合并操作。
- ⑤ 在初始化的时候fast bin支持的最大内存大小以及所有fast bin链表都是空的，所以当最开始使用malloc申请内存的时候，即使申请的内存大小属于fast chunk的内存大小(即0到80字节)，它也不会交由fast bin来处理，而是向下传递交由small bin来处理，如果small bin也为空的话就交给unsorted bin处理。

## 二 glibc中堆的介绍

### 其他bins和差异

#### Unsorted bin

当释放较小或较大的chunk的时候，如果系统没有将它们添加到对应的bins中，系统就将这些chunk添加到unsorted bin(双向循环链表)中。

利用unsorted bin，可以加快内存的分配和释放操作，因为整个操作都不再需要花费额外的时间去查找合适的bin了。

#### Small bin

小于512字节的chunk称之为small chunk，small bin就是用于管理small chunk的。就内存的分配和释放速度而言，small bin比larger bin快，但比fast bin慢。

同时Small bin (双向循环链表)采用FIFO(先入先出)算法

同一个small bin中所有chunk大小是一样的，且第一个small bin中chunk大小为16字节，后续每个依次增加8字节，直至512字节。

当释放small chunk的时候，先检查该chunk相邻的chunk是否为free，如果是的话就进行合并操作：将这些chunks合并成新的chunk，然后将它们从small bin中移除，最后将新的chunk添加到unsorted bin中。

#### Large bin

large bin (双向循环链表)用于管理大于512字节的large chunk

鉴于同一个large bin中每个chunk的大小不一定相同，就将同一个large bin中的所有chunk按照chunk size进行从大到小的排列：最大的chunk放在链表的front end，最小的chunk放在near end。

malloc时首先确定用户请求的大小属于哪一个large bin。从队尾开始找到第一个size相等或接近的chunk。如果该chunk大于用户请求的size的话，就将该chunk拆分为两个chunk：前者返回给用户，且size等同于用户请求的size；剩余的部分做为一个新的chunk添加到unsorted bin中。

## 二 glibc中堆的介绍

### 分配流程

1. 获取分配区的锁，为了防止多个线程同时访问同一个分配区。
2. 将用户请求的大小转换位实际需要分配的chunk空间的大小。
3. 判断所需分配的chunk大小是否小于max\_fast，如果大于则跳转第五步。
4. 首先在fast bins中查找一个chunk给用户，如果存在则分配结束。
5. 判断所需大小是否处在small bins中，如果chunk大小处在small bins中，则根据所需chunk大小，找到具体所在的某个small bin，从bin的尾部摘取一个满足大小的chunk返回给用户。否则到第六步。
6. 到了这一步说明分配的是一个大块内存或者small bins中没有合适的chunk。那么首先会遍历fast bins中的chunk与相邻chunk合并链接到unsorted bin中，然后遍历unsorted bin中的chunk，如果只有一个上次分配已经使用的chunk，并且这个chunk满足small bins的要求且这个chunk符合用户所要求的大小，那么直接对这个chunk进行分割返回给用户，否则将它放入small bins或者large bins中。
7. 排除了fast bins和unsorted bin中的chunk，到了这一步从large bins中找到一个合适的chunk，从中划分一块所需大小的chunk，并将剩余部分链接回bins中，若分配成功则结束，否则下一步。
8. 如果bins中都没有合适的chunk，那么则对top chunk进行分割来分配给用户，成功则结束，否则下一步。
9. 到了这一步证明top chunk不能满足需求，那么对于主分配区就调用sbrk()来增加top chunk的大小；对于非主分配区则会调用mmap来分配一个新的sub-heap增加top chunk的大小或者直接mmap()直接映射分配。



## 二 glibc中堆的介绍

### 释放流程

1. free()函数也会先获取分配区的锁。
2. 判空，如果是空直接return。
3. 判断所需释放的chunk是不是mmaped chunk，若是是，直接调用munmap()释放mmaped chunk，解除内存映射。否则下一步。
4. 判断chunk的大小和所处的位置，如果chunk\_size<=max\_fast，并且chunk不在堆顶，也就不与top chunk相邻，则转到下一步，否则转到第六步。
5. 将chunk放到fast bins中，将chunk放入到fast bins中，并不修改当前的chunk的使用标志P，之后释放结束从free()中返回。
6. 判断前一个chunk是否在使用当中，如果前一个块也是空闲块那么进行合并。
7. 判断当前释放的下一个块是不是top chunk，如果是则转到第九步，否则下一步。
8. 判断下一个chunk是否处在使用中，如果下一个chunk也是空闲的则合并后放入unsorted bin中。
9. 如果执行到这一步，说明释放了一个和top chunk相邻的chunk。
10. 判断合并后的chunk大小是否大于FASTBIN\_CONSOLIDATION\_THRESHOLD(默认64KB)，如果是则触发进行fast bins的合并操作。
11. 判断top chunk是否大于mmap的收缩阈值（默认128K），如果是的话，对于主分配区，则会试图归还top chunk中的一部分给操作系统。



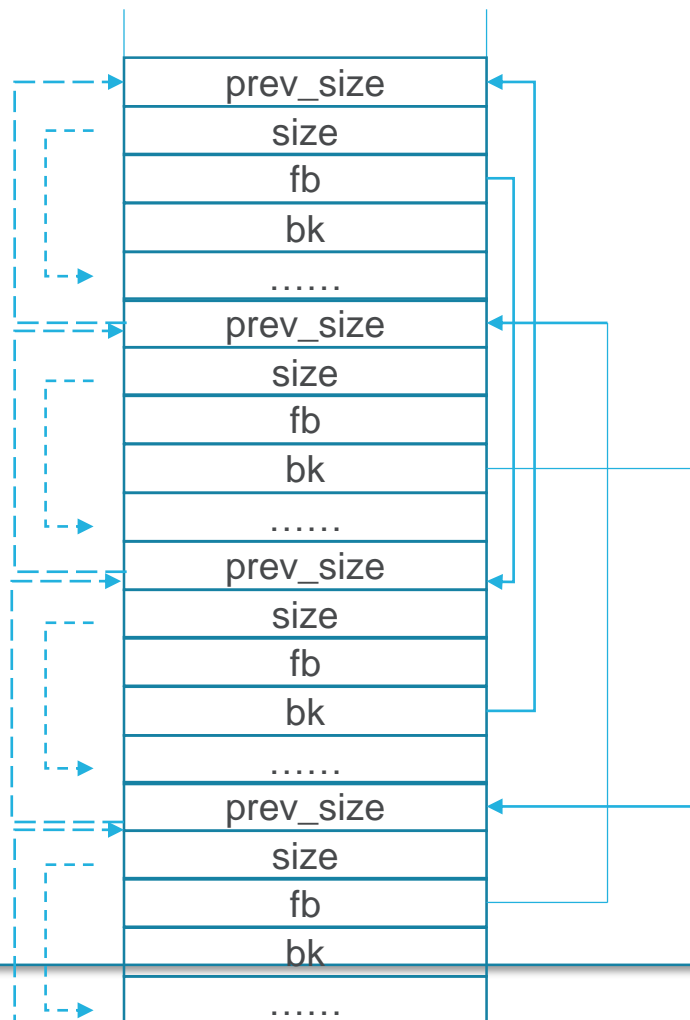
## 二 glibc中堆的介绍

### 重要函数-unlink

隐式链表

为什么需要unlink?

显式链表



**Chunk合并时，在维护隐式链表的同时，需要维护显示链表**

## 二 glibc中堆的介绍

### unlink函数

- 当某个chunk从双链表中脱节点时，会调用unlink，原先的函数类似左图，基本的链表操作，因此可以被利用做写地址。
- 后来glibc将unlink更新成了右图，即会检测该节点的fd、bk两个指针是否真的构成双链表。因此后面再想利用的时候就需要精心伪造chunk。

```
FD = *P + 8;  
BK = *P + 12;  
FD + 12 = BK;  
BK + 8 = FD;
```



```
/* Take a chunk off a bin list */  
void unlink(malloc_chunk *P, malloc_chunk *BK,  
            malloc_chunk *FD)  
{  
    FD = P->fd;  
    BK = P->bk;  
    if (__builtin_expect (FD->bk != P || BK->fd  
!= P, 0))  
  
        malloc_printerr(check_action, "corrupted  
double-linked list", P);  
    else {  
        FD->bk = BK;  
        BK->fd = FD;  
    }  
}
```

## 二 glibc中堆的介绍

### arena

arena是与线程相关的，如下小实验展示了arena的创建和因thread而异的过程。

当主线程首次调用malloc的时候，glibc malloc会直接为它分配一个main arena，而不需要任何附加条件

```
fffe859fc000-7ffe859fe000 r--p 00000000 process ID is:10497
fffe859fe000-7ffe859fa000000 r-xp 00000000 Before malloc in main thread
fffffffff6000000-fffffffff601000 r-xp
youngc@ubuntu:~/ycproject/college/heap$ cat /proc/10497/maps
08048000-08049000 r-xp 00000000 08:01 1711300 /home/youngc/ycproject/college/heap/a.out
08049000-0804a000 r--p 00000000 08:01 1711300 /home/youngc/ycproject/college/heap/a.out
0804a000-0804b000 rw-p 00001000 08:01 1711300 /home/youngc/ycproject/college/heap/a.out
09210000-09231000 rw-p 00000000 00:00 0 [heap]
7523000-f7524000 rw-p 00000000 00:00 0
7524000-f76d4000 r-xp 00000000 08:01 138409 /lib/i386-linux-gnu/libc-2.23.so
76d4000-f76d6000 r--p 001af000 08:01 138409 /lib/i386-linux-gnu/libc-2.23.so
76d6000-f76d7000 rw-p 001b1000 08:01 138409 /lib/i386-linux-gnu/libc-2.23.so
76d7000-f76da000 rw-p 00000000 00:00 0
76da000-f76f3000 r-xp 00000000 08:01 138243 /lib/i386-linux-gnu/libpthread-2.23.so
76f3000-f76f4000 r--p 00018000 08:01 138243 /lib/i386-linux-gnu/libpthread-2.23.so
76f4000-f76f5000 rw-p 00019000 08:01 138243 /lib/i386-linux-gnu/libpthread-2.23.so
76f5000-f76f7000 rw-p 00000000 00:00 0
7711000-f7713000 rw-p 00000000 00:00 0
7713000-f7715000 r--p 00000000 00:00 0 [vvar]
7715000-f7717000 r-xp 00000000 00:00 0 [vdso]
7717000-f7739000 r-xp 00000000 08:01 137033 /lib/i386-linux-gnu/ld-2.23.so
7739000-f773a000 rw-p 00000000 00:00 0
773a000-f773b000 r--p 00022000 08:01 137033 /lib/i386-linux-gnu/ld-2.23.so
773b000-f773c000 rw-p 00023000 08:01 137033 /lib/i386-linux-gnu/ld-2.23.so
fd52000-ffd73000 rw-p 00000000 00:00 0 [stack]
```

有栈无堆

## 二 glibc中堆的介绍

### arena

在进行了一次malloc之后，紧接着数据段出现了堆段（heap segment），大小是0x21000，132KB。这132KB的堆空间叫做arena，此时因为是主线程分配的，所以叫做main arena。由于132KB显然比malloc的大小大很多，所以主线程后续再申请堆空间的话，就会先从这132KB的剩余部分中申请，直到用完或不够用的时候，再通过增加program break location的方式来增加main arena的大小。同理，当main arena中有过多空闲内存的时候，也会通过减小program break location的方式来缩小main arena的大小。

```
f7717000-f7739000 r-xp 00000000 08:01 process ID is:10497
f7739000-f773a000 rw-p 00000000 00:00 Before malloc in main thread
f773a000-f773b000 r--p 00022000 08:01
f773b000-f773c000 rw-p 00023000 08:01 After malloc and before free in main thread
ffd52000-ffd73000 rw-p 00000000 00:00
youngc@ubuntu:~/ycproject/college/heap$ cat /proc/10497/maps
38048000-08049000 r-xp 00000000 08:01 1711300 /home/youngc/ycproject/college/heap/a.out
38049000-0804a000 r--p 00000000 08:01 1711300 /home/youngc/ycproject/college/heap/a.out
3804a000-0804b000 rw-p 00001000 08:01 1711300 /home/youngc/ycproject/college/heap/a.out
39210000-09231000 rw-p 00000000 00:00 0 [heap]
f7523000-f7524000 rw-p 00000000 00:00 0
f7524000-f76d4000 r-xp 00000000 08:01 138409 /lib/i386-linux-gnu/libc-2.23.so
f76d4000-f76d6000 r--p 001af000 08:01 138409 /lib/i386-linux-gnu/libc-2.23.so
f76d6000-f76d7000 rw-p 001b1000 08:01 138409 /lib/i386-linux-gnu/libc-2.23.so
f76d7000-f76da000 rw-p 00000000 00:00 0
f76da000-f76f3000 r-xp 00000000 08:01 138243 /lib/i386-linux-gnu/libpthread-2.23.so
f76f3000-f76f4000 r--p 00018000 08:01 138243 /lib/i386-linux-gnu/libpthread-2.23.so
f76f4000-f76f5000 rw-p 00019000 08:01 138243 /lib/i386-linux-gnu/libpthread-2.23.so
f76f5000-f76f7000 rw-p 00000000 00:00 0
f7711000-f7713000 rw-p 00000000 00:00 0
f7713000-f7715000 r--p 00000000 00:00 0 [vvar]
f7715000-f7717000 r-xp 00000000 00:00 0 [vdso]
f7717000-f7739000 r-xp 00000000 08:01 137033 /lib/i386-linux-gnu/ld-2.23.so
f7739000-f773a000 rw-p 00000000 00:00 0
f773a000-f773b000 r--p 00022000 08:01 137033 /lib/i386-linux-gnu/ld-2.23.so
f773b000-f773c000 rw-p 00023000 08:01 137033 /lib/i386-linux-gnu/ld-2.23.so
ffd52000-ffd73000 rw-p 00000000 00:00 0 [stack]
```

新增了堆

## 二 glibc中堆的介绍

### arena

从输出结果可以看出thread1的heap segment已经分配完毕了，同时从这个区域的起始地址并不同程序的数据段相邻可以看出，它不是通过brk分配的，而是通过mmap分配。这里只有可读写的132KB空间才是thread1的堆空间，即thread1 arena。

这种一个线程有独自的arena的特性是ptmalloc为了支持多线程的设计。不过在不同的系统架构中，arena的数目上限不同。

```
youngc@ubuntu:~/ycproject/college/heap$ cat /proc
```

```
08048000-08049000 r-xp 00000000 08:01 1711300
08049000-0804a000 r--p 00000000 08:01 1711300
0804a000-0804b000 rw-p 00001000 08:01 1711300
09210000-09231000 rw-p 00000000 00:00 0
f6c00000-f6c21000 rw-p 00000000 00:00 0
f6c21000-f6d00000 ---p 00000000 00:00 0
f6d22000-f6d23000 ---p 00000000 00:00 0
f6d23000-f7524000 rw-p 00000000 00:00 0
f7524000-f76d4000 r-xp 00000000 08:01 138409
f76d4000-f76d6000 r--p 001af000 08:01 138409
f76d6000-f76d7000 rw-p 001b1000 08:01 138409
f76d7000-f76da000 rw-p 00000000 00:00 0
```

After malloc and before free in thread 1

/home/youngc/ycproject/college/heap/a.out  
[heap]

main arena 创建后一直存在

线程的堆

### BRK和MMAP?

brk是将段的最高地址指针往高地址推，分配的空间是连续的；

mmap是在进程的虚拟地址空间中（堆和栈中间，称为文件映射区）找一块空闲的能够满足大小的虚拟内存进行分配。

## 二 glibc中堆的介绍

### Arena header与 heap header

Arena Header又称作`malloc_state`，每个thread只含有一个Arena Header。Arena Header包含bins的信息、互斥锁信息、top chunk以及最后一个remainder chunk等。

Main thread不含有多个heaps所以也就不含有`heap_info`结构体。当需要更多堆空间的时候，就通过扩展`sbrk`的heap segment来获取更多的空间，直到它延展触碰到内存mapping区域为止。

heap header 则是堆的描述符，存在于所有thread heap上。由于线程堆段的位置不确定性以及数量的不确定性（在当前heap不够用的时候，`malloc`会通过系统调用`mmap`申请新的堆空间，新的堆空间会被添加到当前thread arena中，便于管理），heap header存在的意义是指示线程如何进行堆的管理，第一个字段是指向thread arena的指针，第二个是一个heap header的单向链表指针用于将多个heap连接起来供当前线程使用。

## 二 glibc中堆的介绍

### Arena header与 heap header

多线程中arena的共享：

在arena达到数量上限后，新线程请求malloc时，glibc malloc循环遍历所有可用的arena，在遍历的过程中，它会尝试lock该arena。如果成功lock(该arena当前对应的线程并未使用堆内存则表示可lock)，比如将main arena成功lock住，那么就将main arena返回给用户，即表示该arena被当前线程共享使用。具体表现就是在heap header中指向arena的指针与某个线程相同。

而如果不能找到可用的arena，那么就将线程3的malloc操作阻塞，直到有可用的arena为止。

## 提纲

### 一 堆简介

1.1 堆的基本介绍

1.2 堆与栈

### 二 glibc中堆的实现 ( ptmalloc )

2.1 程序员的内存-chunk介绍

2.2 chunk的组织-bin介绍

2.3 bin管理与并发-arena介绍

### 三 windows中堆的实现

3.1 windows堆的历史

3.2 堆的数据结构和管理策略

3.3 堆分配

3.4 堆攻击



## Windows堆的历史

由于微软没有完全公开操作系统堆管理细节。Windows堆的了解主要基于黑客和安全专家和逆向工程师的个人研究成果。

### Windows堆的演变过程：

- (1) Windows 2000 ~ Windows XP SP1：只考虑分配任务和性能，不考虑安全，比较容易被攻击者利用。
- (2) Windows XP2 ~ Windows 2003：加入了安全因素，比如修改块首格式加入安全cookie，双向链表节点在删除时会做指针验证，堆溢出困难但是高级攻击可能利用成功。
- (3) Windows Vista ~ Windows 7 :堆的效率和安全性都提高。
- (4) 目前堆管理机制兼顾了内存有效利用、分配决策速度、健壮性、安全性等因素。

逆向专家对相对简单的Windows 2000~Windows XP SP1研究较多，下面以此为例讨论。

## 堆的数据结构

操作系统将复杂的堆封装为API，程序员使用堆只涉及到申请一定大小的内存、使用内存、释放内存。

- 杂乱：反复申请释放后连续的内存块呈现大小不等的空闲块
- 标志：哪些是占用的哪些是空闲的
- 恰当：合理分配空闲块

**堆块：** 以堆块为单位标识，不是按字节标识，分为块首和块身。块首是用来标识堆块自身信息的几个字节，例如：本块的大小、本块占用或空闲标志；块身跟在块首后面是分配给用户的数据区。（注意：堆管理系统返回的指针指向块身的起始地址）

**堆表：** 位于堆区的起始位置，索引堆块的重要信息，包括：堆块的位置、堆块的大小、空闲占用状态。只索引空闲块，会用到平衡二叉树数据结构优化查找效率。

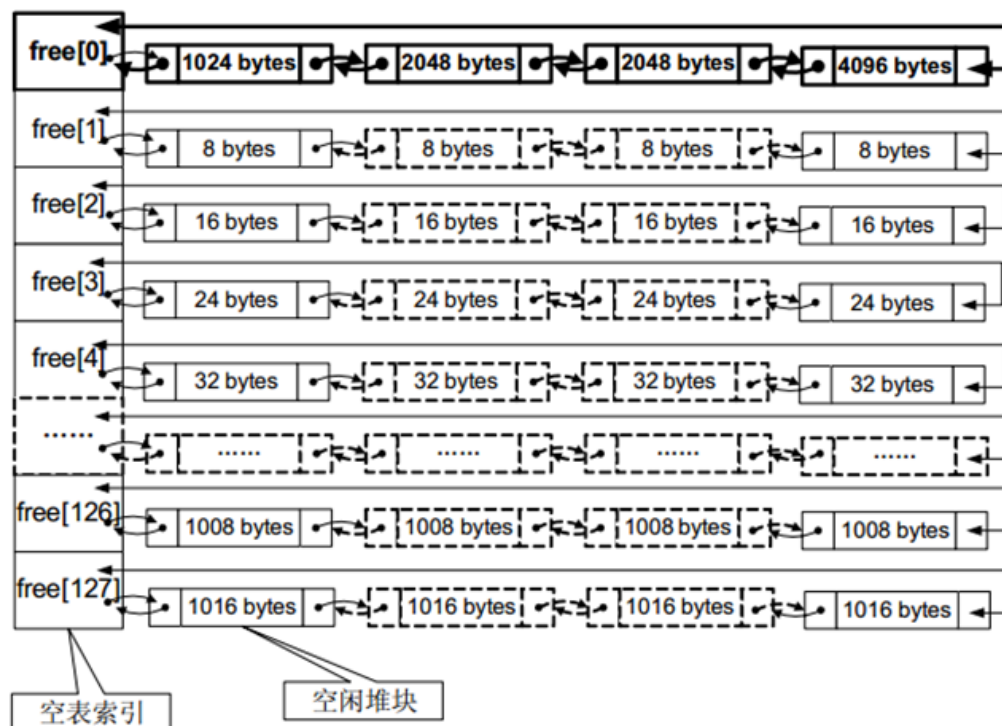
堆表：空表和快表。

自身长度	前一个块的长度	段索引	Flags	未使用的字节	标签索引 (调试用)
下一个块		前一个块			

# 三 windows中堆的简介

## (1) 空表

空闲堆块块首有一对将空闲堆块组织成双向链表的指针，按堆块大小空表分为128条。堆表区中有一个128项称为空表索引的指针数组。数组每项包括两个指针，用于标识一条空表。



空闲堆块大小=索引项 (ID) X 8 (字节)  
free[0] 链入所有大于1024字节小于512K字节的堆块。

## (2) 快表

单链表组织，不会发生堆块合并（块首设置占用态），快表128条，每条快表最多四个节点。

内存块按照大小分为三类：

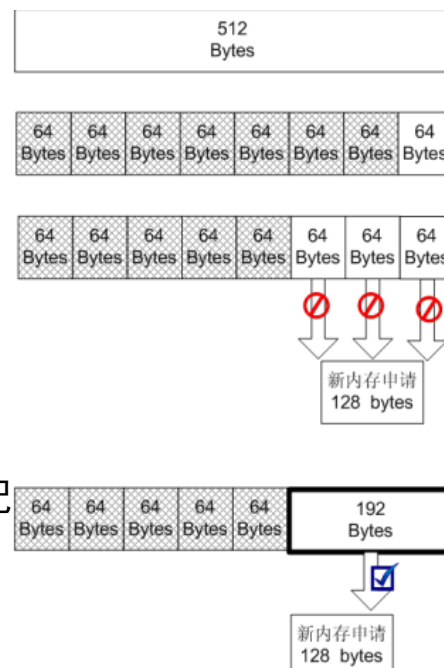
- 小块：SIZE<1KB
- 大块：1KB<=SIZE<512KB
- 巨块：SIZE>=512KB

堆中操作分为：堆块分配、堆块释放和堆块合并。前两个程序提交申请和执行，合并由堆管理系统自动完成。

堆块分配：快表分配寻找大小匹配空闲块；普通空表分配，先找最优否则次优；free[0]分配是先反向比较最大再正向找最小满足的。多余部分重新标注块首放入空表。

堆块释放：状态改为空闲，链入堆表末尾分配时从末尾拿，快表最多4项。

堆块合并：碎片合并



1: 内存中有1个空闲块，大小为512字节

2: 连续7次请求64字节的内存

3: 在使用中，2个堆块被提前释放，这时程序再次申请一个128字节的堆块，虽然堆区内目前有 $3 \times 64 = 192$ 字节的空闲内存，但由于他们是不连续的“内存碎片”，故无法满足申请要求

4: 如果能把小堆块合并成大堆块，则能够更有效的利用内存，从而完成分配

## 堆管理策略

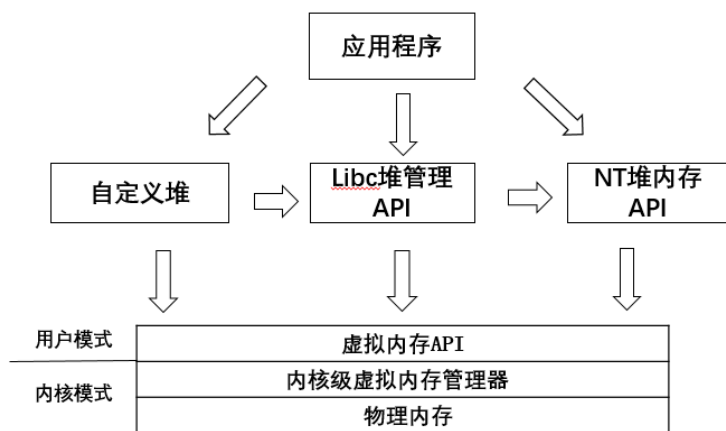
	分配	释放
小块	首先快表分配空闲块，不成功则普通表分配，不成功则堆缓存分配空闲块，不成功则零号空表分配，还是不行进行内存碎片整理后分配，都不行时返回null。	优先链入快表，快表满了将其链入空表。
大块	堆缓存分配，否则零号表分配	优先堆缓存，否则链入零号表
巨块	罕见，虚分配方法	直接释放

### Windows堆管理要点：

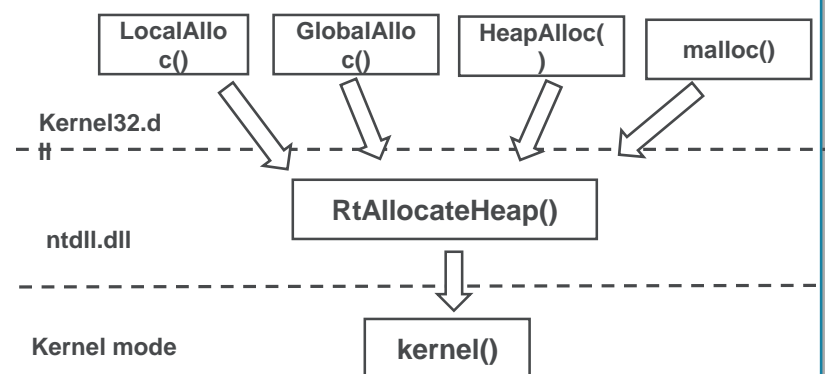
- (1) 快表没有合并操作；
- (2) 快表精确匹配时才会用；
- (3) 快表是单链表，插入删除少用很多指令；
- (4) 分配和释放时优先使用快表，失败时用空表；
- (5) 快表只有4项很容易满，因此空表也被频繁使用。

# 三 windows中堆的简介

## 堆分配



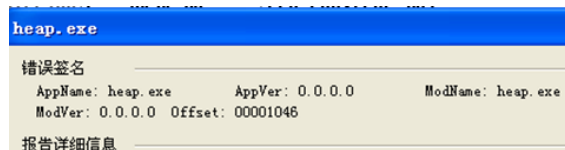
Windows堆分配体系架构



Windows堆分配API调用关系

用户态最底层的堆分配函数：RtAllocateHeap(),位于ntdll.dll动态链接库

## 堆分配实例



```
Heap address: 00360000
h1: 00360688
h2: 00360698
h3: 003606A8
h4: 003606B8
h5: 003606C8
h6: 003606E8
```

堆句柄	请求字节数	实际分配（堆单位）	实际分配（字节）
h1	3	2	16
h2	5	2	16
h3	6	2	16
h4	8	2	16
h5	19	4	32
h6	24	4	32

### • 调试堆和常态堆有区别：

- (1) 调试堆只用空表；
- (2) 所有程序尾部加上16字节防程序溢出（不是堆），8字节0xAB,8字节0x00；
- (3)块首标志位不同。

```
int main(int argc, char **argv)
{
    HLOCAL h1, h2, h3, h4, h5, h6;
    HANDLE hp;
    hp = HeapCreate(0, 0x1000, 0x10000);
    printf("Heap address: %p\n", hp);

    // 为了避免程序监测出调试器而使用调试堆管理策略
    __asm int 3

    h1 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 3);
    printf("h1: %p\n", h1);
    h2 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 5);
    printf("h2: %p\n", h2);
    h3 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 6);
    printf("h3: %p\n", h3);
    h4 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 8);
    printf("h4: %p\n", h4);
    h5 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 19);
    printf("h5: %p\n", h5);
    h6 = HeapAlloc(hp, HEAP_ZERO_MEMORY, 24);
    printf("h6: %p\n", h6);
    HeapFree(hp, 0, h1);
    HeapFree(hp, 0, h3);
    HeapFree(hp, 0, h5);
    HeapFree(hp, 0, h4);
    return 0;
}
```

## 堆攻击

### 典型堆和栈的协同攻击--Heap Spray

Heap Spray没有官方定义，简单而言是在shellcode的前面加上大量的slide code（滑板指令），组成一个注入代码段。然后向系统申请大量内存，并且反复用注入代码段来填充。这样就使得进程的地址空间被大量的注入代码所占据。然后结合其他的漏洞攻击技术控制程序流（例如栈），使得程序执行到堆上，最终将导致shellcode的执行。

### 常见的堆溢出和利用技术有：

- Unlink
- Use afterfree
- Double free

常规情况下堆布局

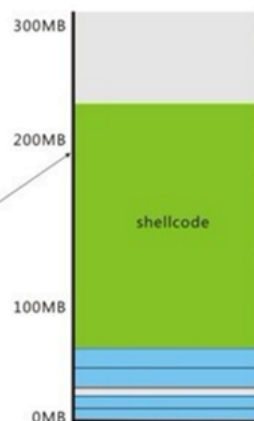
■：已使用内存  
■：空闲内存



堆喷 (heap spraying)之后

■：已使用内存  
■：空闲内存  
■：shellcode

地址0x0C0C0C0C已经包含shellcode的可能性非常高  
 $200 \times 1024 \times 1024 = 0x0C800000 > 0x0C0C0C0C$





## 漏洞利用与攻防实践

Q&A