

堆溢出漏洞机理分析





01. **堆内存管理**

02. **堆的数据结构**

03. **堆漏洞机理**

04. **堆漏洞利用**

栈和堆

栈

- 利用栈实现函数/过程
 - 遵循call / ret的调用规则
- OS通过给函数分配栈帧(frame)来保存变量或参数
 - 每个函数尤其独立的栈（不考虑编译优化的规则）
 - 后进先出
- 管理清晰

堆

- 用户动态数据的保存
 - malloc / free
 - new / delete
- 用户负责堆块的创建和释放
- OS/内存管理器负责空闲堆的管理
- 数据繁多、管理复杂

堆内存管理

- 堆内存管理算法
 - 不同的操作系统类型
 - 操作系统版本
 - 平台上差异较大

堆内存管理

- 堆内存管理算法
 - 不同的操作系统类型
 - 操作系统版本
 - 平台上差异较大
- 微软没有公开Win堆管理细节
- 堆结构主要由逆向得出
 - Blackhat 2002, Third Generation Exploitation
 - Blackhat 2004, Windows Heap Overflows
 - Win2000下堆的数据结构及管理算法、溢出方法、利用思路

堆内存管理

- 堆内存管理算法
 - 不同的操作系统类型
 - 操作系统版本
 - 平台上差异较大

堆相关WinAPI函数

- | | |
|-------------------|---|
| • HeapCreate | 创建一个新的堆对象 <ul style="list-style-type: none">• 创建一个只有调用进程才能访问的私有堆• 此外，进程还有默认堆 |
| • HeapDestroy | 销毁一个堆对象 |
| • HeapAlloc | 在堆中申请内存空间 |
| • HeapFree | 释放申请的内存 |
| • HeapWalk | 枚举堆对象的所有内存块 |
| • GetProcessHeap | 取得进程的默认堆对象 |
| • GetProcessHeaps | 取得进程所有的堆对象 |

堆内存管理

- 常见堆内存管理器

- dlmalloc

- 通用堆分配器

- ptmalloc2

- glibc, 衍生自dlmalloc, 提供多线程支持

- jemalloc

- FireFox

- tcmalloc

- google chrome

- libumem

- Solaris

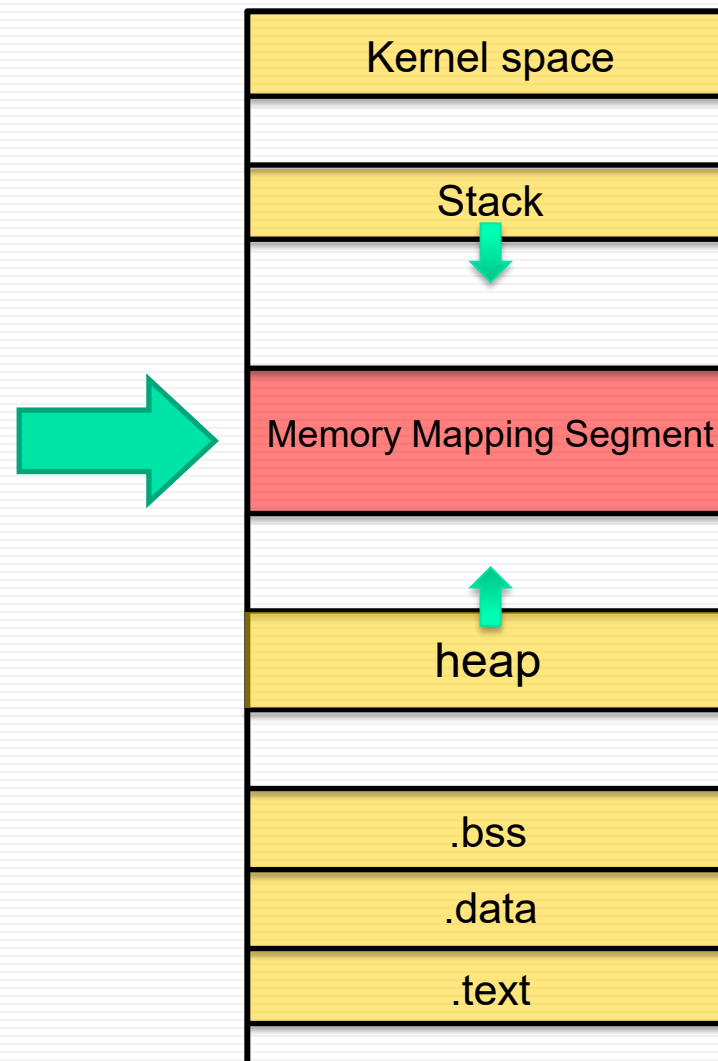
- Windows 10 - segment heap

Linux进程内存管理

- 动态内存分配 ↔ 两个系统调用
 - mmap
 - brk

Linux进程内存管理

- mmap
 - 在文件映射区域申请一块虚拟内存
 - 场景：申请大片内存时
 - 阈值由M_MMAP_THRESHOLD控制



Linux进程内存管理

■ brk

■ 堆区管理

■ brk(new_brk)

- 改变“program break”

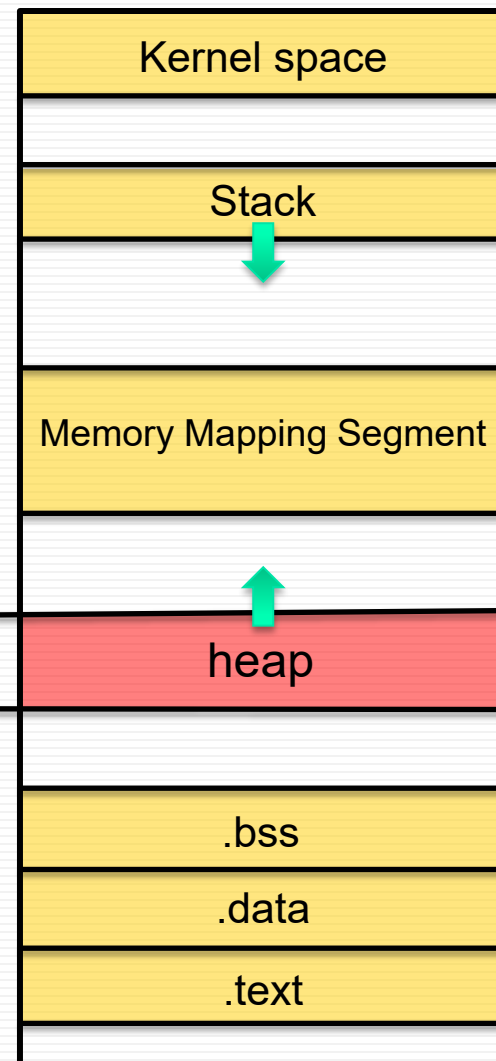
■ sbrk - 库函数

- sbrk(incr)

- 通过将brk增加incr来扩展或收缩堆区

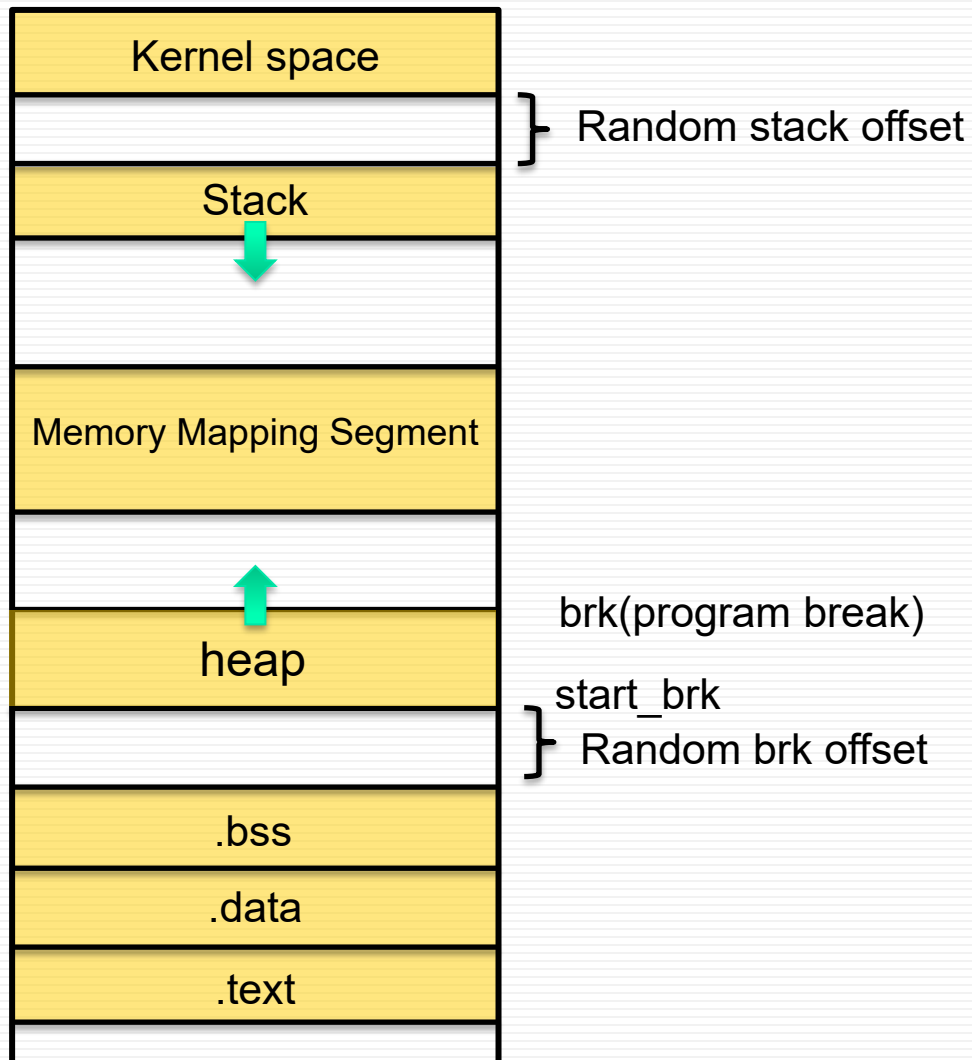
brk(program break)

start_brk



Linux进程内存管理

■ Linux 进程虚拟内存布局





02

堆的数据结构与管理算法

堆的数据结构与管理算法

- 堆区的数据结构与管理算法在不同的操作系统类型、操作系统的不同版本上差异较大
- 堆实现
 - 关注的是堆区内存的组织和管理方式，特别是**空闲内存块**
 - 提高分配和释放效率
 - 降低碎片化，提高空间利用率

堆的数据结构与管理

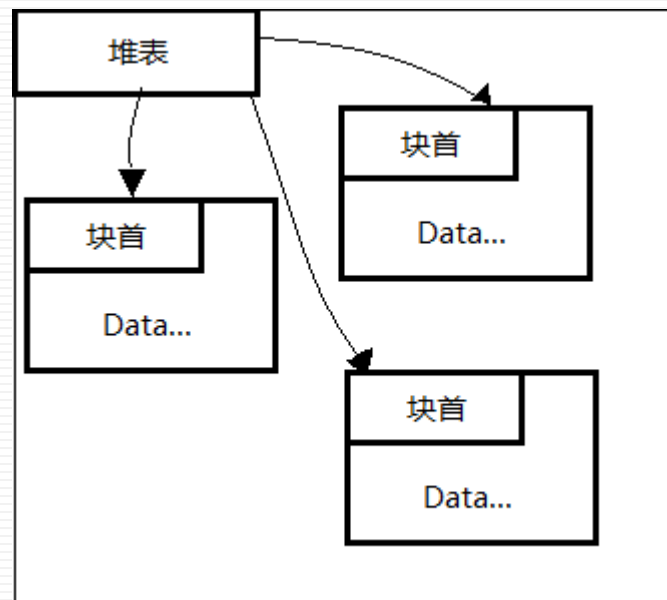
- 堆结构：堆表+堆块

- 堆表

- 位于堆区起始位置,用于索引堆区中所有堆块的重要信息
 - 堆表的数据结构决定了整个堆区的组织方式
 - 分为两类：快表与空表

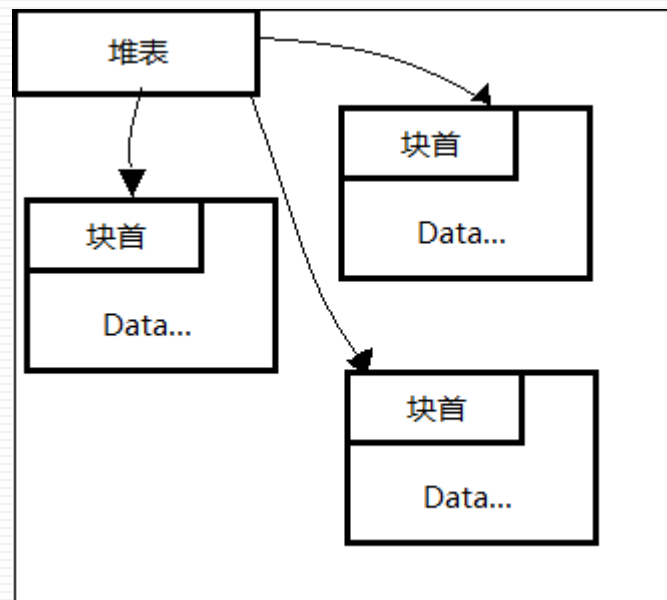
- 堆块

- 块首+块身
 - 块首：大小、状态
 - 块身：用户数据



堆的数据结构与管理

- 堆结构：堆表+堆块
 - 占用态的堆块由被使用者索引
 - Malloc 后要 free
 - 堆表只索引所有的空闲堆块
 - Free 之后空闲块重新归入堆表



堆的数据结构——堆表

- 堆表有很多种，且随OS/内存管理器的不同而不同
- 常见的堆内存管理
 - FreeList(空表)
 - Lookaside (快表)

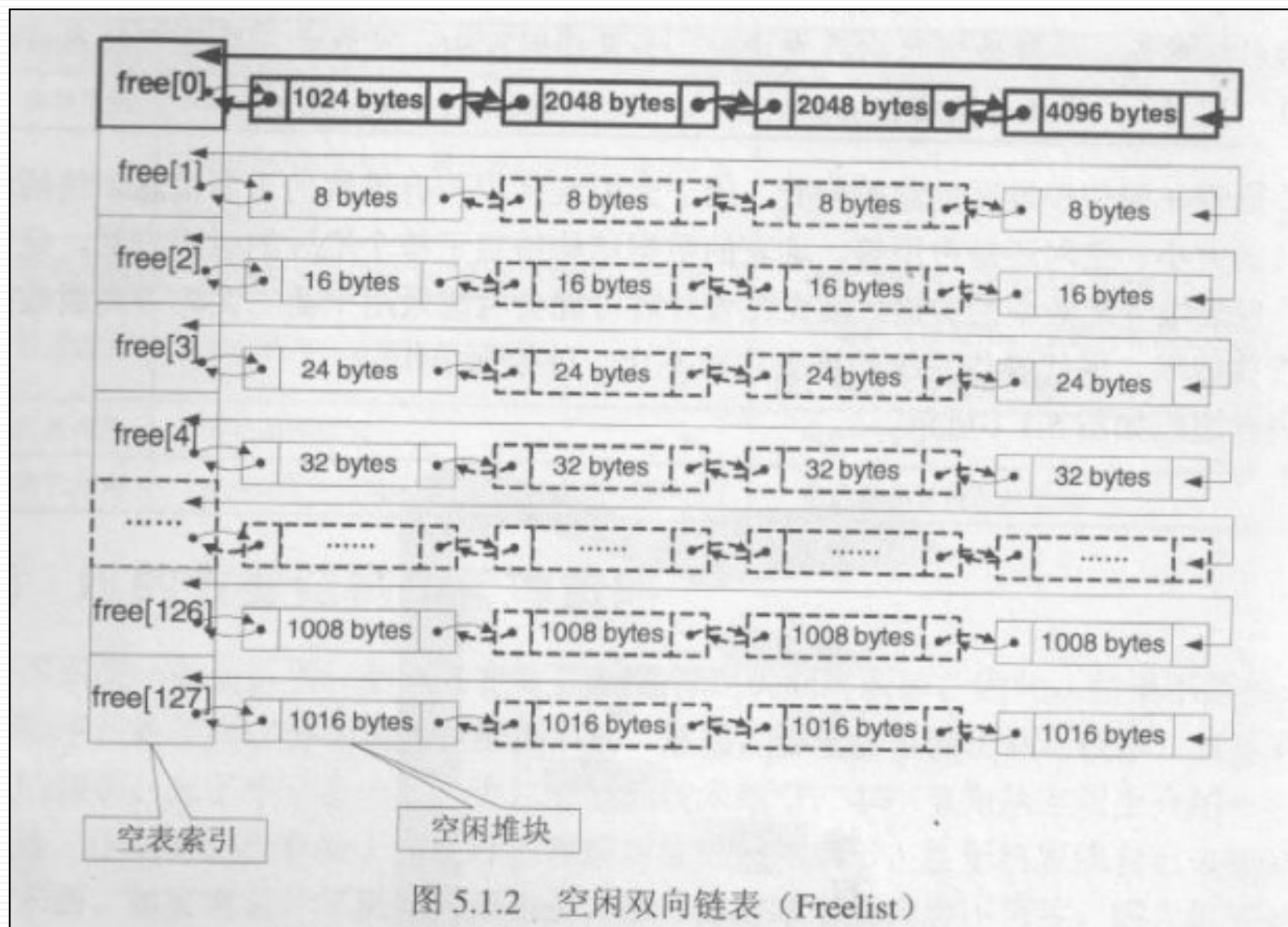
两类重要堆表：1) 空表

- **空表**

- 双向链表
- 有128项，每项标识指定大小的空闲块
- 空闲块大小=索引项 (ID) *8
- 空表索引：128项的指针数组，包含2个指针
- Free[0]标识大于1024的堆块升序索引

两类重要堆表：1) 空表

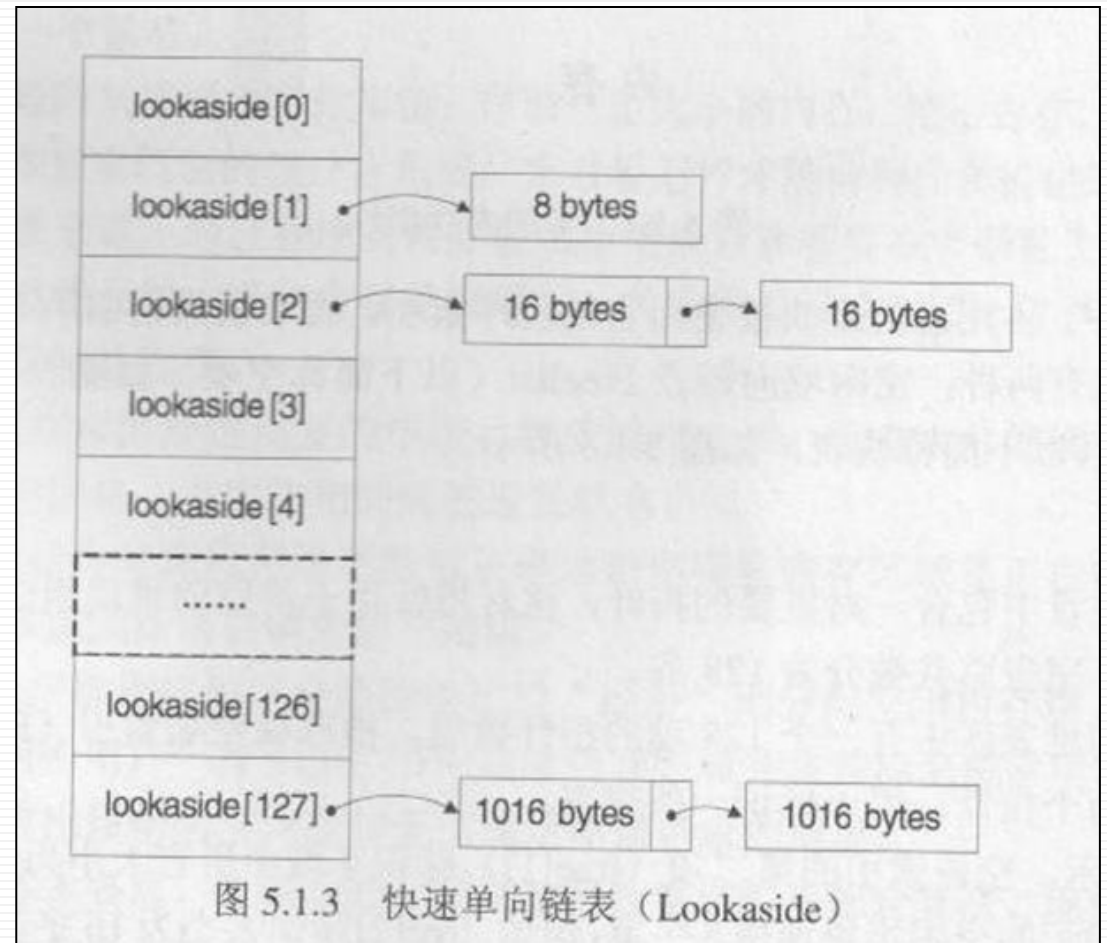
- 空闲双向链表



两类重要堆表：2) 快表

- 快表

- 加速分配
- 128项
- 采用单向链表
- 链中的堆一般不发生合并
- 每项最多4个节点



堆的数据结构——堆块

- 块首
 - 头部几个字节,用来标识自身信息 (如大小,空闲还是占有等)
- 块身
 - 数据存储区域, 紧跟块首
- 堆块的管理
 - 分配
 - 释放: 修改状态, 链于空表中
 - 合并: 相邻的堆块合并, 链于空表中

堆数据结构与管理实例——ptmalloc2

- 常见堆内存管理器

- dmalloc

- 通用堆分配器

- **ptmalloc2**

- glibc, 衍生自dmalloc, 提供多线程支持

- jemalloc

- FireFox

- tcmalloc

- google chrome

- libumem

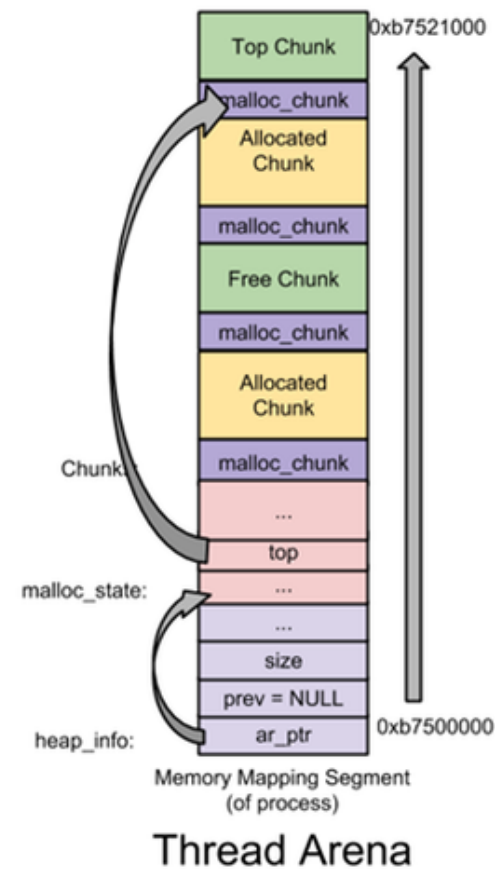
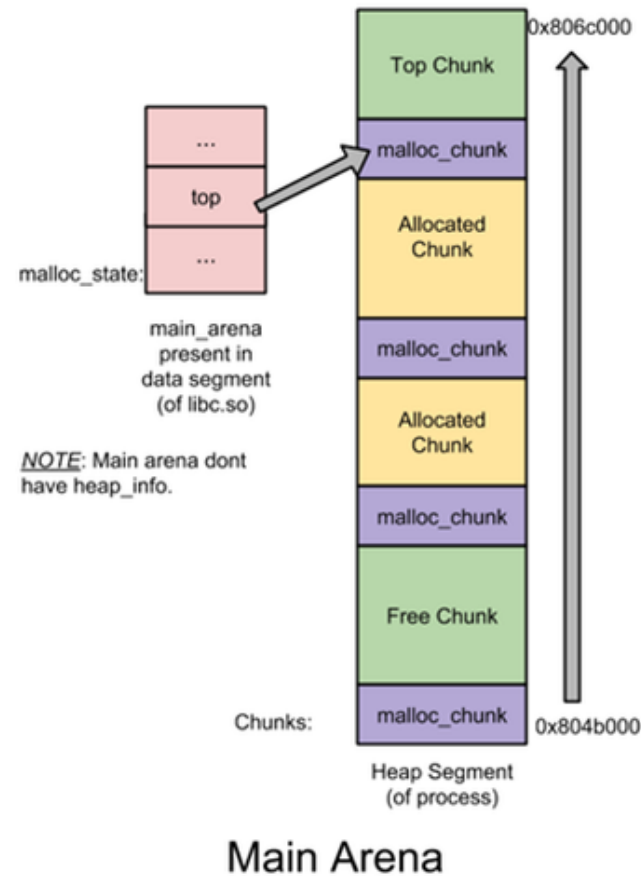
- Solaris

- Windows 10 - segment heap

堆的数据结构与管理

■ Glibc堆管理实现

- arena
- malloc_state
- chunks
- bins



堆的数据结构与管理

■ arena

- 指的是堆内存区域本身，而非一个结构
- 主线程创建的堆叫main_arena
- 不同线程维护不同的堆，per_thread_arena
- Arena数量受CPU限制
 - 2*核心数(32位系统)
 - 8*核心数(64位系统)
- main_arena通过sbrk创建
- 其他线程的arena通过mmap创建

堆的数据结构与管理

■ malloc_state

- 储存了arena的状态，包括用于管理空闲块的bins链表等

```
struct malloc_state
{
    __libc_lock_define(, mutex); /* Serialize access. */
    int flags; /* Flags (formerly in max_fast). */
    int have_fastchunks; /* Set if the fastbin chunks contain recently inserted free blocks. */
    mfastbinptr fastbinsY[NFASTBINS]; /* Fastbins */
    mchunkptr top; /* Base of the topmost chunk -- not otherwise kept in a bin */
    mchunkptr last_remainder; /* The remainder from the most recent split of a small request */
    mchunkptr bins[NBINS * 2 - 2]; /* Normal bins packed as described above */
    unsigned int binmap[BINMAPSIZE]; /* Bitmap of bins */
    struct malloc_state *next; /* Linked list */
    struct malloc_state *next_free; /* Linked list for free arenas. */
    INTERNAL_SIZE_T attached_threads; /* Number of threads attached to this arena. */
    INTERNAL_SIZE_T system_mem; /* Memory allocated from the system in this arena. */
    INTERNAL_SIZE_T max_system_mem;
};

static struct malloc_state main_arena; /*global variable in libc.so*/
```


堆的数据结构与管理

- malloc_state

- fastbinsY : 存放了所有fastbin链的数组;
 - fastbin链: 单向链表
- Top : 存放了当前top chunk位置
- last_remainder : 指向最后一次分割后的残留部分
- Bins: 记录管理和组织空闲chunk的链表的数组
 - 空闲chunk链: 双向链表
- binmap
 - 记录每个bin是否为空

堆的数据结构与管理

- **chunks**

- 堆内存块遵循的结构，其定义位于malloc.c

```
/*
 * This struct declaration is misleading (but accurate and necessary).
 * It declares a "view" into memory allowing access to necessary
 * fields at known offsets from a given base. See explanation below.
 */
struct malloc_chunk {

    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;
};
```

堆的数据结构与管理

- **chunks**

- glibc malloc将整个堆内存空间分成了连续的、大小不一的 chunk
- 对于堆内存管理而言， chunk就是最小操作单位
- Chunk总共分为4类
 - 1)allocated chunk;
 - 2)free chunk;
 - 3)top chunk;
 - 4)Last remainder chunk
- 所有类型的chunk都是内存中一块连续的区域

堆的数据结构与管理

- chunks
 - prev_size
 - 如果前一个chunk(指内存空间上的, 而非链表上的)为空闲的话, 用来表示其大小; 否则可作用户数据使用
 - size
 - 表示当前chunk的大小, 后三位为**标志位(P M A)**
 - fd与bk
 - 与其他chunk相连, 形成双向链表
 - fd_nextsize与bk_nextsize
 - 主要是用于大的内存块

堆的数据结构与管理

■ 空闲chunks

[illegible]

堆的数据结构与管理

- 已分配chunks

[illegible]

堆的数据结构与管理

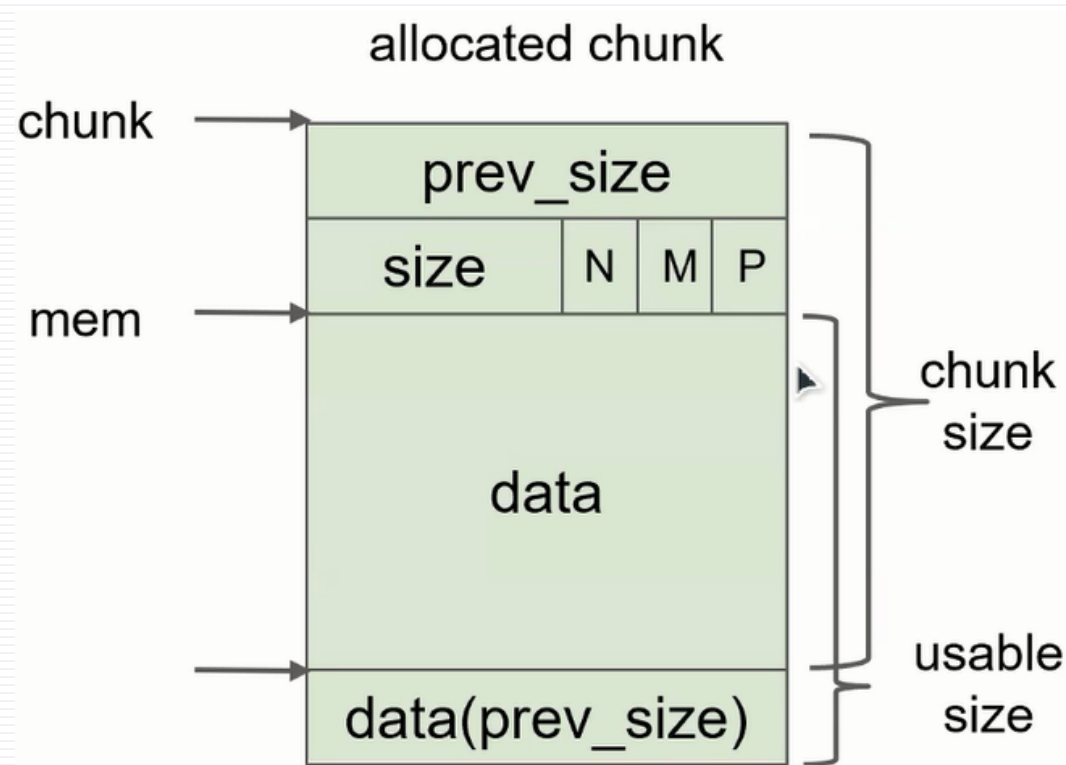
■ 字段复用

```
-+-+-+-+-+
Size of previous chunk, if unallocated (P clear) |
-+-+-+-+-+
Size of chunk, in bytes                          |A|M|P|
-+-+-+-+-+
User data starts here ...                        .
                                                .
(malloc_usable_size() bytes)                  .
                                                |
-+-+-+-+-+
(size of chunk, but used for application data) |
-+-+-+-+-+
Size of next chunk, in bytes                    |A|0|1|
-+-+-+-+-+
```

```
-+-+-+-+-+
Size of previous chunk, if unallocated (P clear) |
-+-+-+-+-+
Size of chunk, in bytes                          |A|0|P|
-+-+-+-+-+
Forward pointer to next chunk in list            |
-+-+-+-+-+
Back pointer to previous chunk in list           |
-+-+-+-+-+
Unused space (may be 0 bytes long)              .
                                                .
                                                |
-+-+-+-+-+
Size of chunk, in bytes                          |
-+-+-+-+-+
Size of next chunk, in bytes                    |A|0|0|
-+-+-+-+-+
```

堆的数据结构与管理

- malloc参数与分配的chunk大小
 - 返回的chunk只需要保证可用大小大于等于用户申请
 - chunk对齐
 - 8字节(x86 32位平台)
 - 16字节(x64平台)



堆的数据结构与管理

- Bins：用来管理和组织空闲内存块的链表结构
 - 根据大小和管理方式，分为
 - fast bin：用于管理小的chunk
 - small bin：用于管理中等大小的chunk
 - large bin：用于管理较大的chunk
 - unsorted bin：用于存放未被整理的chunk(垃圾回收)
 - fast bin以fastbinY存放在malloc_state中
 - 其他bins以bins数组存放在malloc_state中

```
#define NBINS          128 /*bin的总数 实际是127  unsorted bin(1)+smallbin(62)+largebin(63)+1 */
struct malloc_state{
    ...
    mfastbinptr fastbinsY[NFASTBINS];/*Fastbins */
    ...
    /* Normal bins packed as described above */
    mchunkptr bins[NBINS * 2 - 2];/* Normal bins packed as described above,unsorted bin small bin large bin */
    ...
}
```

堆的数据结构与管理

■ fastbin

- 大小：16~64字节(32位系统)；32~128(64位系统)
- fastbins为了速度而产生(后面还有了tcache)，在设计上和其他的bins是不同的
 - fastbin中的chunk以单链表的形式构成chunk链
 - 对每个fastbin的链，其中的chunk大小相同
 - fastbin采用后进先出(后释放的会被先申请)
 - 由于是单链表，fastbin的bk指针并无作用，只会用到fd指针
 - fastbin中的chunk的prev_inuse一直处于set状态，保证他们不会与相邻的空闲chunk合并
- malloc_state中fastbinsY数组存放的就是每个fastbin链的头指针

堆的数据结构与管理

■ fastbin内存结构示例

```
pwndbg> x/50gx 0x10c3000
0x10c3000: 0x0000000000000000 0x0000000000000021
0x10c3010: 0x4141414141414141 0x4141414141414141
0x10c3020: 0x0000000000000000 0x0000000000000021
0x10c3030: 0x4242424242424242 0x4242424242424242
0x10c3040: 0x0000000000000000 0x0000000000000021
0x10c3050: 0x4343434343434343 0x4343434343434343
0x10c3060: 0x0000000000000000 0x00000000000020fa1
0x10c3070: 0x0000000000000000 0x0000000000000000
0x10c3080: 0x0000000000000000 0x0000000000000000
0x10c3090: 0x0000000000000000 0x0000000000000000
0x10c30a0: 0x0000000000000000 0x0000000000000000
0x10c30b0: 0x0000000000000000 0x0000000000000000
0x10c30c0: 0x0000000000000000 0x0000000000000000
0x10c30d0: 0x0000000000000000 0x0000000000000000
0x10c30e0: 0x0000000000000000 0x0000000000000000
0x10c30f0: 0x0000000000000000 0x0000000000000000
0x10c3100: 0x0000000000000000 0x0000000000000000
0x10c3110: 0x0000000000000000 0x0000000000000000
0x10c3120: 0x0000000000000000 0x0000000000000000
0x10c3130: 0x0000000000000000 0x0000000000000000
0x10c3140: 0x0000000000000000 0x0000000000000000
0x10c3150: 0x0000000000000000 0x0000000000000000
0x10c3160: 0x0000000000000000 0x0000000000000000
0x10c3170: 0x0000000000000000 0x0000000000000000
0x10c3180: 0x0000000000000000 0x0000000000000000
pwndbg> 
```

```
#include <stdio.h>
int main(){
```

```
    char *p1 = malloc(0x10);
    memset(p1,0x41,0x10);
    char *p2 = malloc(0x10);
    memset(p2,0x42,0x10);
    char *p3 = malloc(0x10);
    memset(p3,0x43,0x10);
    printf("malloc done\n");
    free(p1);
    free(p2);
    printf("free done\n");
    return 0;
```

```
}
```

堆的数据结构与管理

■ fastbin内存结构示例

```
0x80: 0x0
pwndbg> x/32gx 0x10c3000
0x10c3000: 0x0000000000000000 0x0000000000000021 chunk1
0x10c3010: 0x0000000000000000 0x4141414141414141
0x10c3020: 0x0000000000000000 0x0000000000000021 chunk2
0x10c3030: 0x00000000010c3000 0x4242424242424242
0x10c3040: 0x0000000000000000 0x0000000000000021
0x10c3050: 0x4343434343434343 0x4343434343434343 chunk3
0x10c3060: 0x0000000000000000 0x0000000000000411
0x10c3070: 0x6420636f6c6c616d 0x000000000a656e6f
0x10c3080: 0x0000000000000000 0x0000000000000000
0x10c3090: 0x0000000000000000 0x0000000000000000
0x10c30a0: 0x0000000000000000 0x0000000000000000
0x10c30b0: 0x0000000000000000 0x0000000000000000
0x10c30c0: 0x0000000000000000 0x0000000000000000
0x10c30d0: 0x0000000000000000 0x0000000000000000
0x10c30e0: 0x0000000000000000 0x0000000000000000
0x10c30f0: 0x0000000000000000 0x0000000000000000
pwndbg> fastbin
fastbins
0x20: 0x10c3020 ← 'BBBBBBBB'
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
pwndbg>
```

```
#include <stdio.h>
int main(){
```

```
    char *p1 = malloc(0x10);
    memset(p1,0x41,0x10);
    char *p2 = malloc(0x10);
    memset(p2,0x42,0x10);
    char *p3 = malloc(0x10);
    memset(p3,0x43,0x10);
    printf("malloc done\n");
    free(p1);
    free(p2);
    printf("free done\n");
    return 0;
```

```
}
```

堆的数据结构与管理

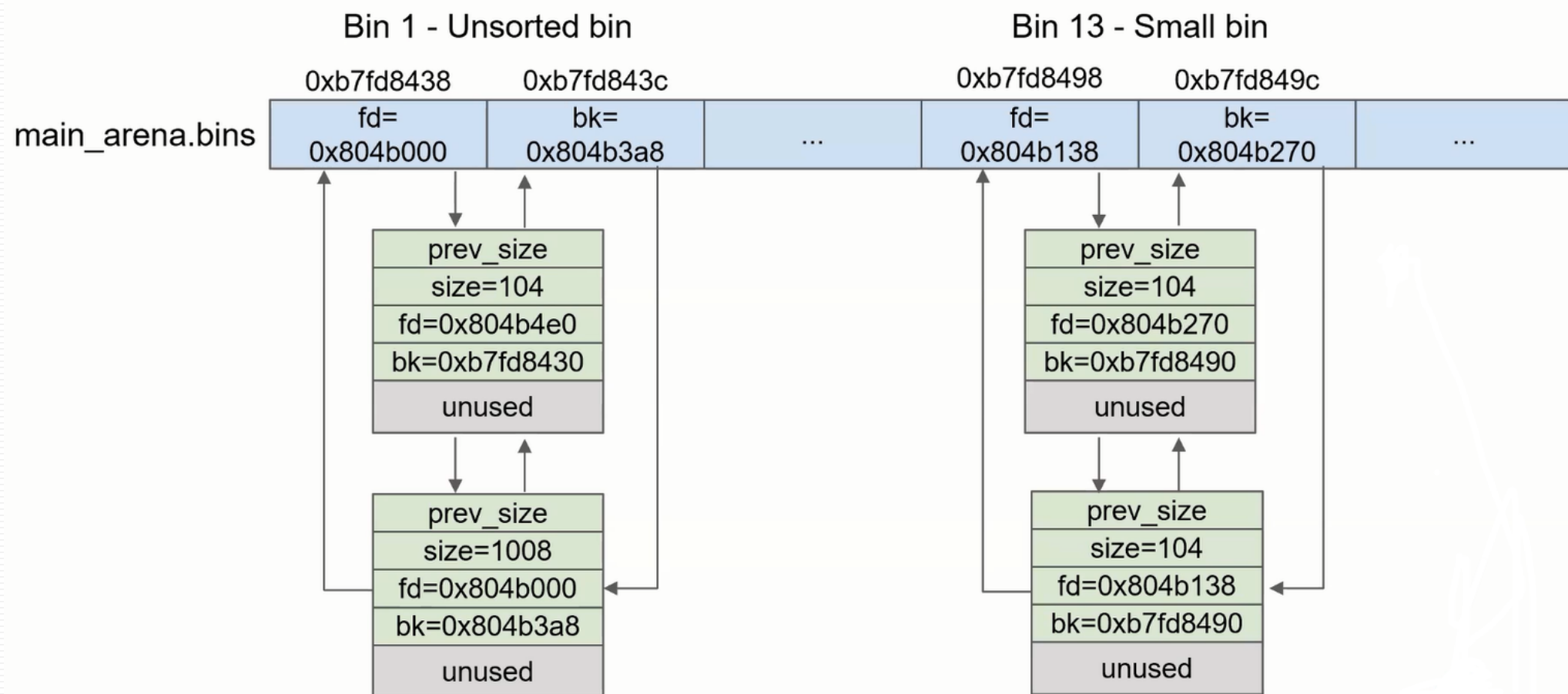
■ 其他bins

- small bin/large bin/unsorted bin
- 与fastbin的实现不同，每个bin都是一个双向链表
- chunk链表的头指针和尾指针存放在bins数组中
- 如下图，申请了254个指针存放这些bins，每一组指针(例如bin[2]bin[3]存放一个双链表的头和尾指针)

```
#define NBINS          128 /*bin的总数 实际是127  unsorted bin(1)+smallbin(62)+largebin(63)+1 */
struct malloc_state{
    ...
    /* Normal bins packed as described above */
    mchunkptr bins[NBINS * 2 - 2];/* Normal bins packed as described above,unsorted bin small bin large bin */
    ...
}
```

堆的数据结构与管理

■ bins结构示意图





03 堆漏洞



堆漏洞

- 与堆利用相关的堆漏洞
 - 溢出
 - 溢出多个字节
 - off-by-one
 - 时序类漏洞
 - Use After Free
 - Double free

堆漏洞

- 堆溢出
 - 内存移动/复制导致的溢出
 - 溢出后的影响

```
-----+
Size of previous chunk, if unallocated (P clear) |
-----+
Size of chunk, in bytes                          |A|M|P|
-----+
User data starts here ...                        .
                                                .
(malloc_usable_size() bytes)                  .
                                                |
-----+
(size of chunk, but used for application data)  |
-----+
Size of next chunk, in bytes                     |A|0|1|
-----+
```

```
-----+
Size of previous chunk, if unallocated (P clear) |
-----+
Size of chunk, in bytes                          |A|0|P|
-----+
Forward pointer to next chunk in list            |
-----+
Back pointer to previous chunk in list          |
-----+
Unused space (may be 0 bytes long)              .
                                                .
(size of chunk, but used for application data)  |
-----+
Size of chunk, in bytes                          |
-----+
Size of next chunk, in bytes                     |A|0|0|
-----+
```

堆漏洞

- heap based off-by-one

- off-by-one指的就是溢出一个字节的漏洞
 - 循环边界次数错误
 - 字符串拷贝溢出结束符\x00，也称为NULL off-by-one
 - strlen计算字符串长度，后strcpy复制字符串
 - 前者不计算\x00，后者会拷贝到\x00，造成1个字节的溢出
- off-by-one容易导致下一个chunk其size的最后一个字节被覆盖掉，而由前面可知，size最后一个字节中存在prev_inuse标识位标识了上一个chunk的使用状态

```
-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-----+
Size of previous chunk, if unallocated (P clear) |
-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-----+
Size of chunk, in bytes                           |A|M|P|
-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-----+
User data starts here ...                          .
                                                    .
(malloc_usable_size() bytes)                      .
                                                    |
-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-----+
(size of chunk, but used for application data)    |
-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-----+
Size of next chunk, in bytes                       |A|0|1|
-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-----+
```

堆漏洞

■ Use After Free

- 一个内存块被释放后又再次被使用
- 内存free后主要有两种情况
 - free后并将指针置为NULL，这就是正确的free操作
 - free后未将指针置为NULL，也就是dangling pointer
 - 在下一次使用前并没有再修改这个内存块，这种一般也能正常运行
 - 在下一次使用前修改了这个内存块，当再次使用时就会出现问题的
- UAF只是提供了去写入一个被free的chunk的内容的能力，需要配合堆利用手法
 - 比如利用UAF去修改chunk的fd和bk进行unlink

堆漏洞

- Double Free

- 一个内存块被释放后又再次被释放
- Double free 的影响
 - Free操作会影响链表结构
 - Free 后会把当前堆块加入到相应的链表
 - Fastbin 单向链表
 - 其它bins 双向链表
 - Fastbin double free 利用

04 堆漏洞利用

从栈利用到堆利用

- 栈溢出利用：重写关键数据结构

- 返回地址
- 函数指针 (SEH/GOT)
- EBP
- 任意地址写

- 堆溢出利用：重写关键数据结构

- 各种各样的指针
- 任意地址写？

3.堆溢出

- 堆溢出就是寻找内存写的机会

- `ptr = malloc () ; *prt = write`
- `free();`
- `free(); new(); write; use_after_free();`
- `free(ptr1); free(ptr2); free(ptr1); double_free`

3.堆溢出

- “DWORD SHOOT” 把任意数据写入任意地址
 - 栈中函数返回地址
 - 栈帧中的SEH
 - 函数调用（指针）地址
- 精髓
 - 用精心构造的数据去溢出下一个堆块的块首
 - 改写块首中的前向指针（fd）与后向指针（bk）

堆管理的双链表操作

- 双链表中的节点删除如何操作

```
Remove (ListNode *node)
```

```
{
```

```
    node->bk > fd = node->fd; (1)
```

```
    node->fd > bk = node->bk;
```

```
}
```

- 控制 (1)

- node->bk 任意内存地址

- node->fd 任意数据

- 或者 控制 (2)

- node->bk 任意数据

- node->fd 任意内存地址

3.堆溢出

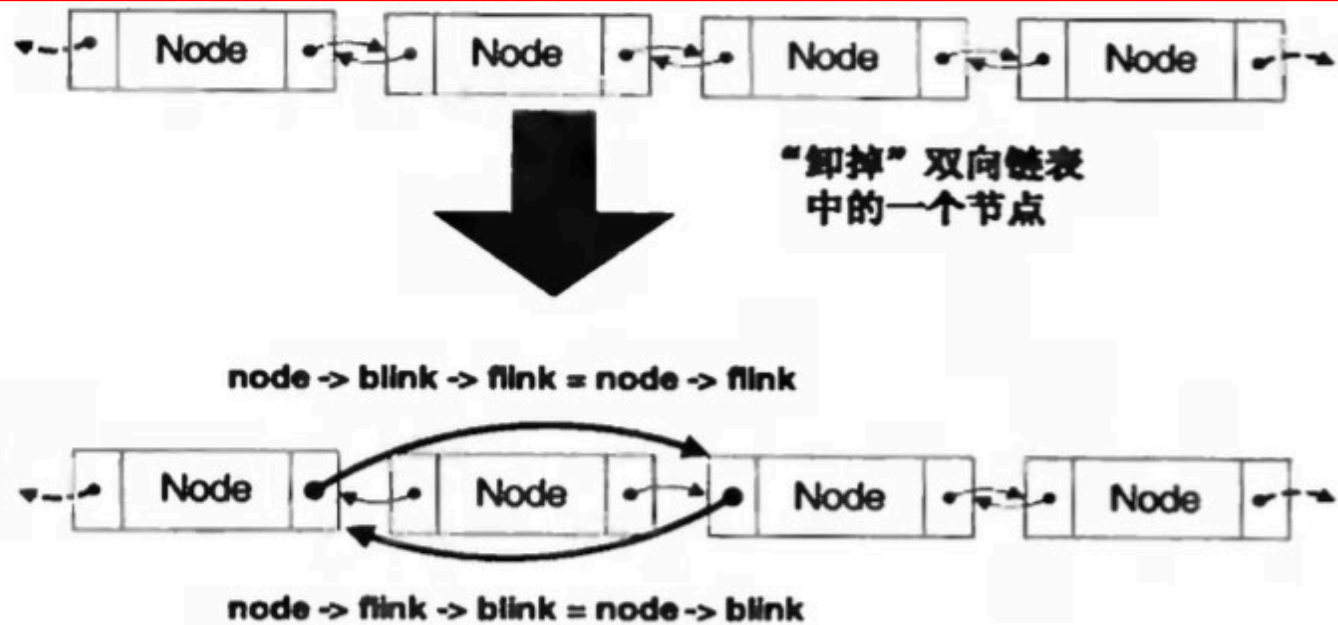
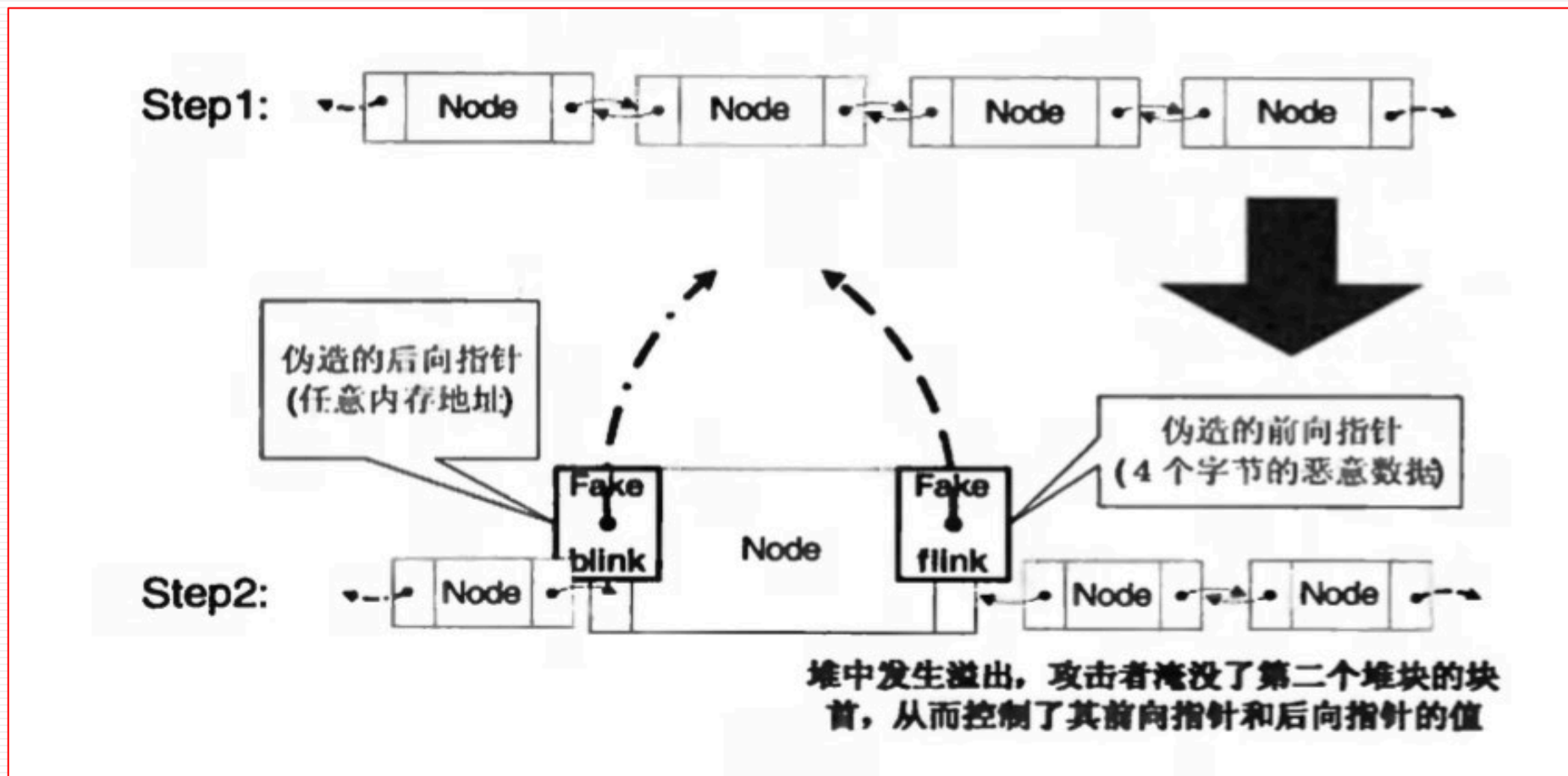


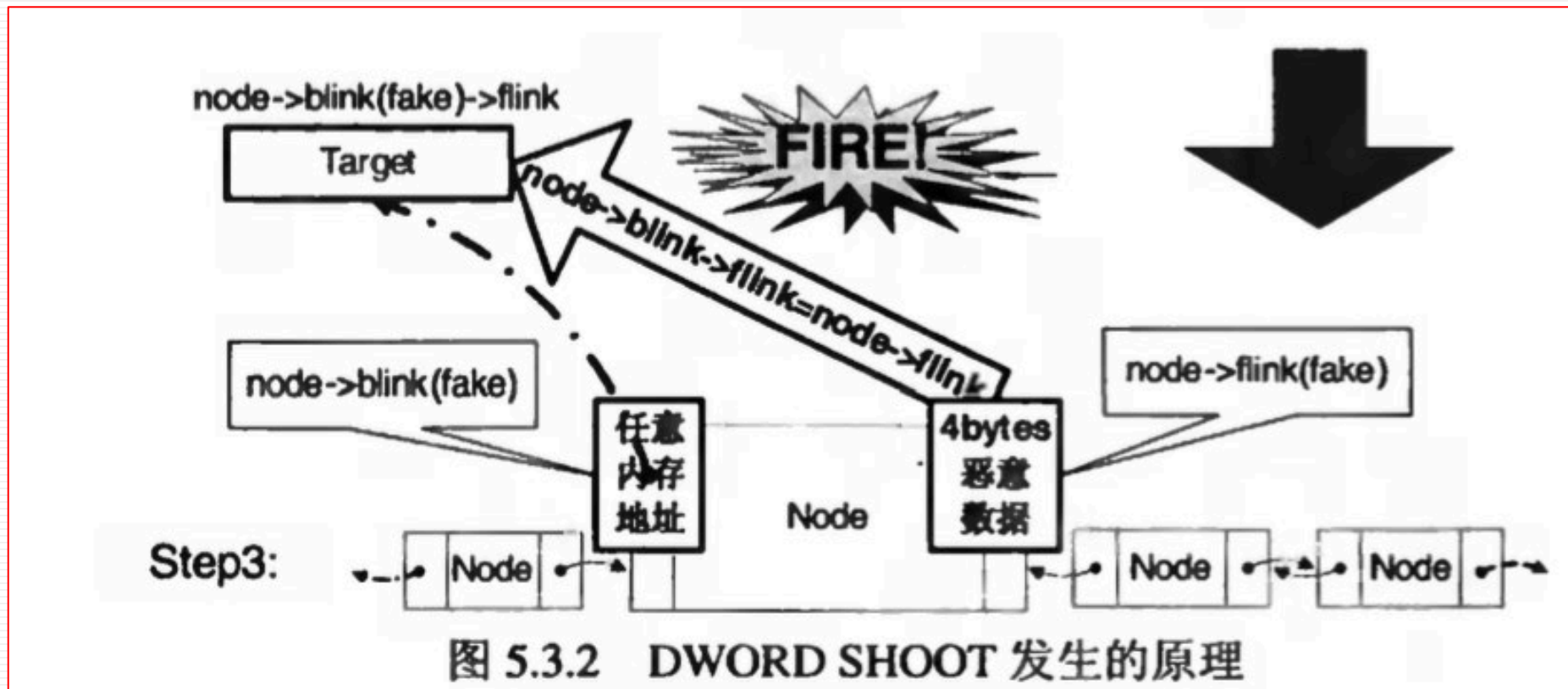
图 5.3.1 空闲双向链表的拆卸

3.堆溢出



参考文献：0day安全-王清

3.堆溢出



堆漏洞利用

■ unlink

- 如果利用chunk1的溢出覆盖chunk2的fd和bk字段，而free掉chunk1的时候由于chunk2空闲，所以会将chunk2进行unlink，然后与chunk1合并，但是这时候chunk2的fd和bk已经被修改了
- 假设fd被修改为target_addr，bk被修改为target_value，根据unlink操作

```
FD = p->fd; //→ target_addr
BK = p->bk; //→ target_value
FD->bk = BK; //→以target_addr作为chunk的bk偏移处内存
           //被修改为target_value(任意地址读写)
BK->fd = FD; //→ 以target_value作为chunk的fd偏移处内存被修改为target_addr
           //当然还需要保证target_value这个地方可写
```

堆漏洞利用

- 新的unlink

- 在修复后的unlink版本中加入了验证

- 判断了下一个chunk的bk指针是否指向当前的chunk

- `fd_chunk->bk == chunk`

- 判断了上一个chunk的fd指针是否指向当前chunk

- `bk_chunk->fd == chunk`

- 假设是64位系统,就是

- `*(fd_chunk+0x18) == chunk`

- `*(bk_chunk + 0x10) == chunk`

- 效果：使指向chunk 的指针 ptr 变为 `ptr - 0x18`

堆漏洞利用

- 堆利用方法
 - fastbin attack
 - Chunk Extend and Overlapping
 - Unlink
 - Unsorted Bin Attack等

堆漏洞利用

- Fastbin attack
 - 包含所有基于fastbin机制的漏洞利用方式。
 - 一般通过堆溢出或者UAF等控制chunk内容的漏洞实现对fastbin类型的chunk的修改以进行攻击。
 - 主要有下面几种
 - Fastbin Double Free
 - House Of Spirit
 - Arbitrary Alloc

堆漏洞利用

- Fastbin Double Free

- 利用了fastbinfree时的漏洞

- fast chunk在free时仅验证了当前chunk是否与对应chunk链头部相同，而并未对该chunk链上的每一个chunk作验证

```
if (__builtin_expect (old == p, 0))  
    malloc_printerr ("double free or corruption (fasttop)");  
p->fd = old2 = old;
```

堆漏洞利用

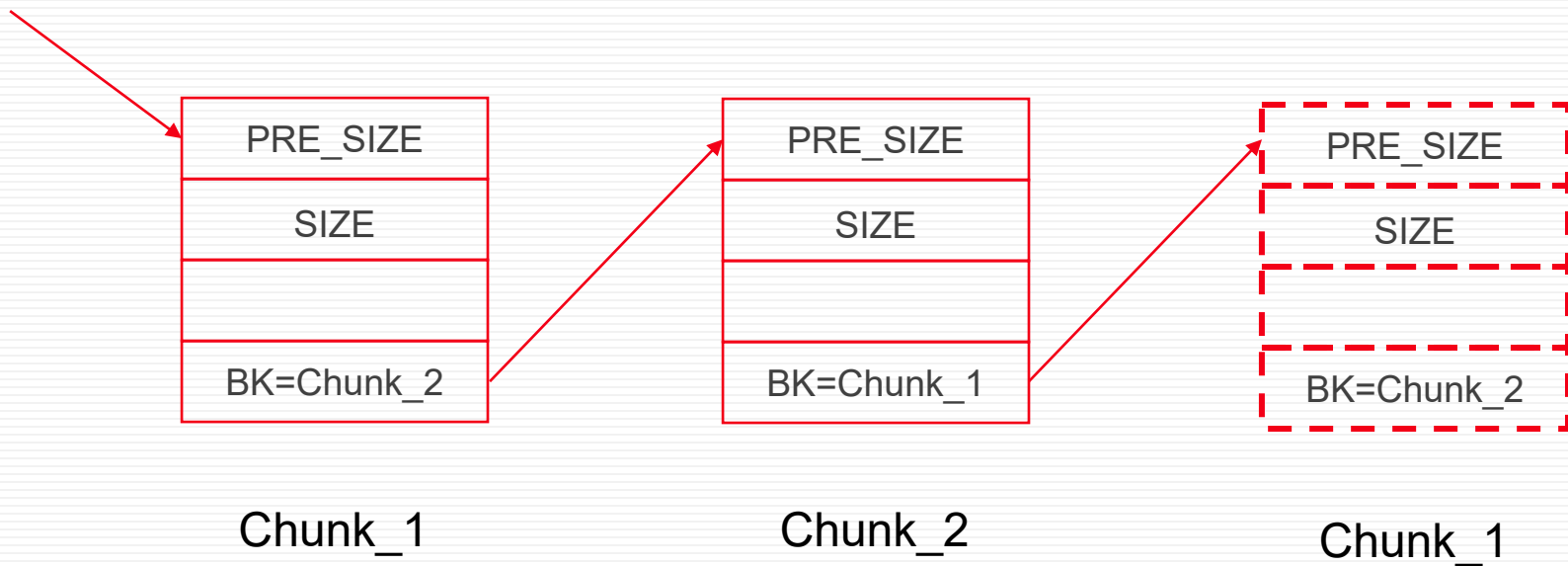
- Fastbin Double Free

- 效果

- 通过 fastbin double free 可以使多个指针控制同一个堆块
 - 用于篡改一些堆块中的关键数据。如果修改 fd 指针, 则能够实现任意地址分配堆块的效果 (当然对应位置的size需要满足一定条件), 相当于任意地址写任意值

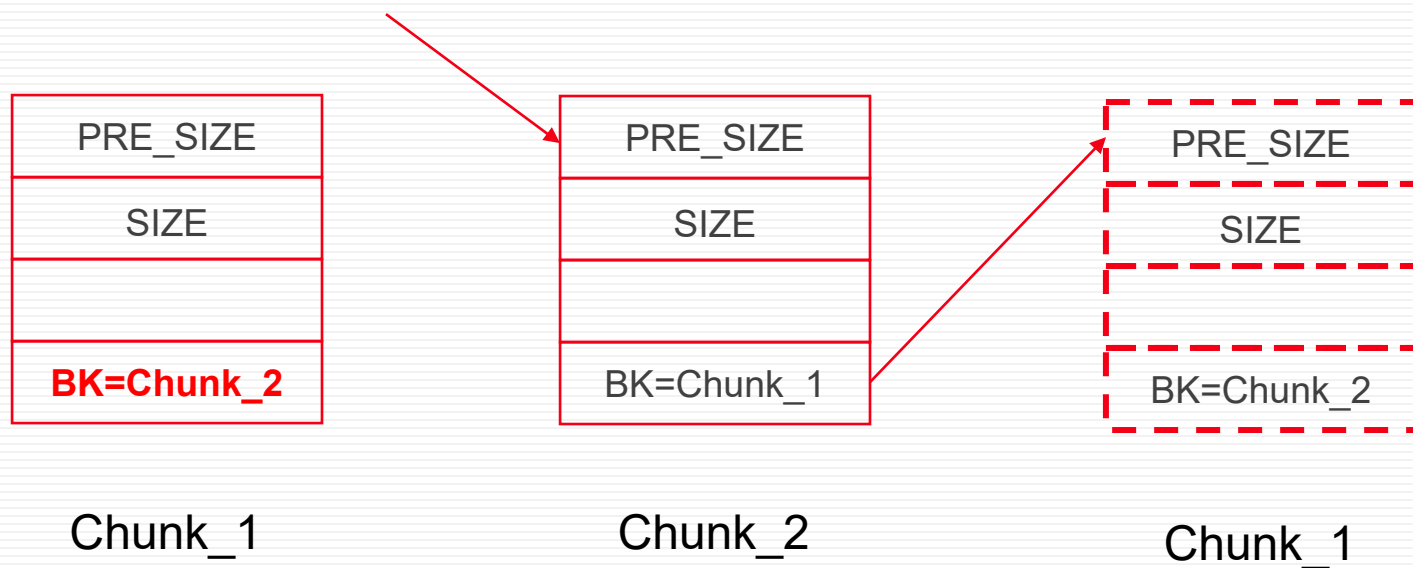
Fastbin double free利用示例

- 利用过程
- `Free(chunk_1); Free(chunk_2); Free(chunk_1);`



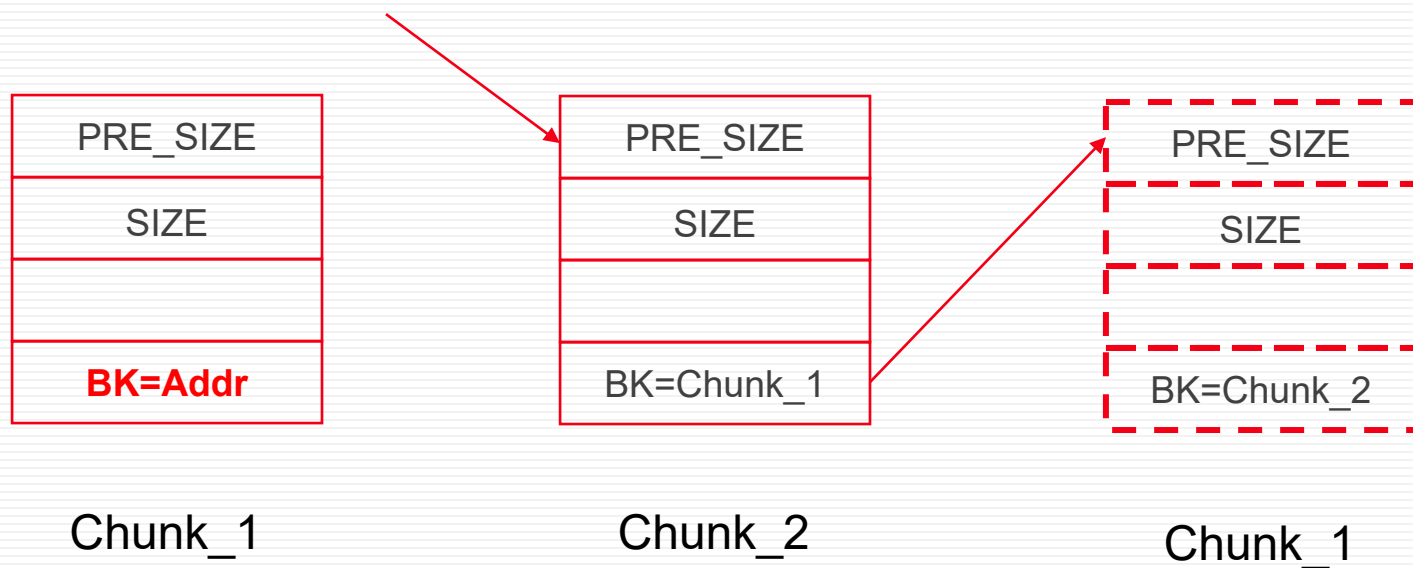
Fastbin double free利用示例

- 利用过程
- `Free(chunk_1); Free(chunk_2); Free(chunk_1);`
- `Malloc();`



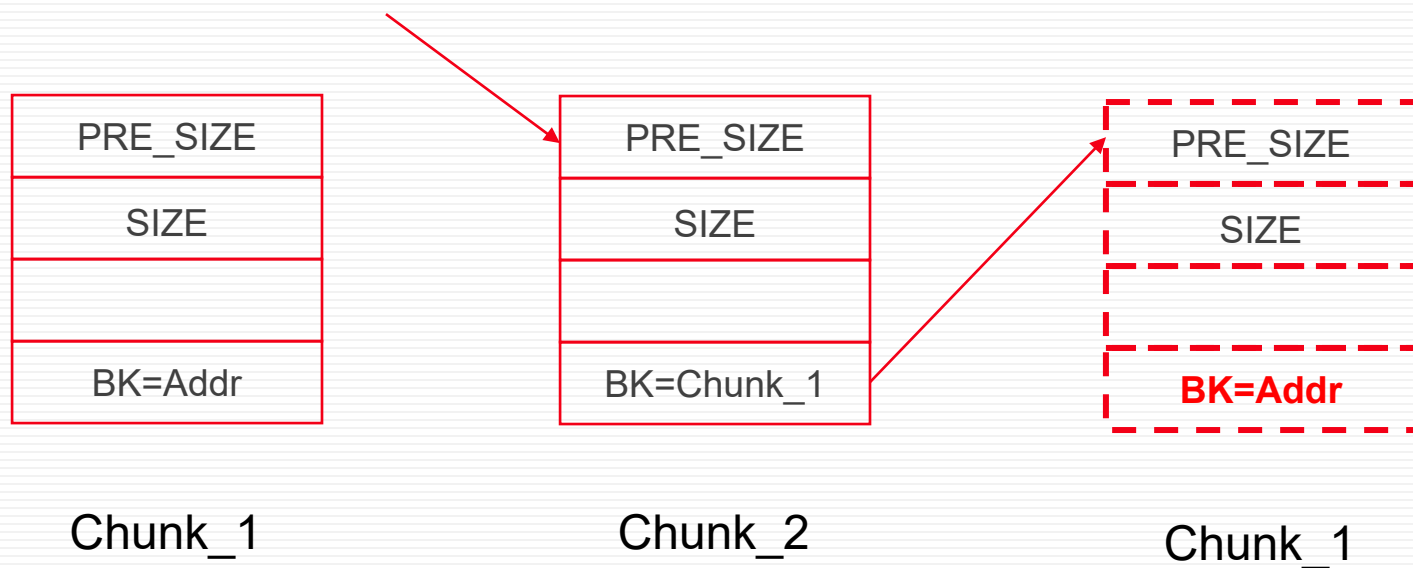
Fastbin double free利用示例

- 利用过程
- `Free(chunk_1); Free(chunk_2); Free(chunk_1);`
- `Malloc(); write_to_chunk_1;`



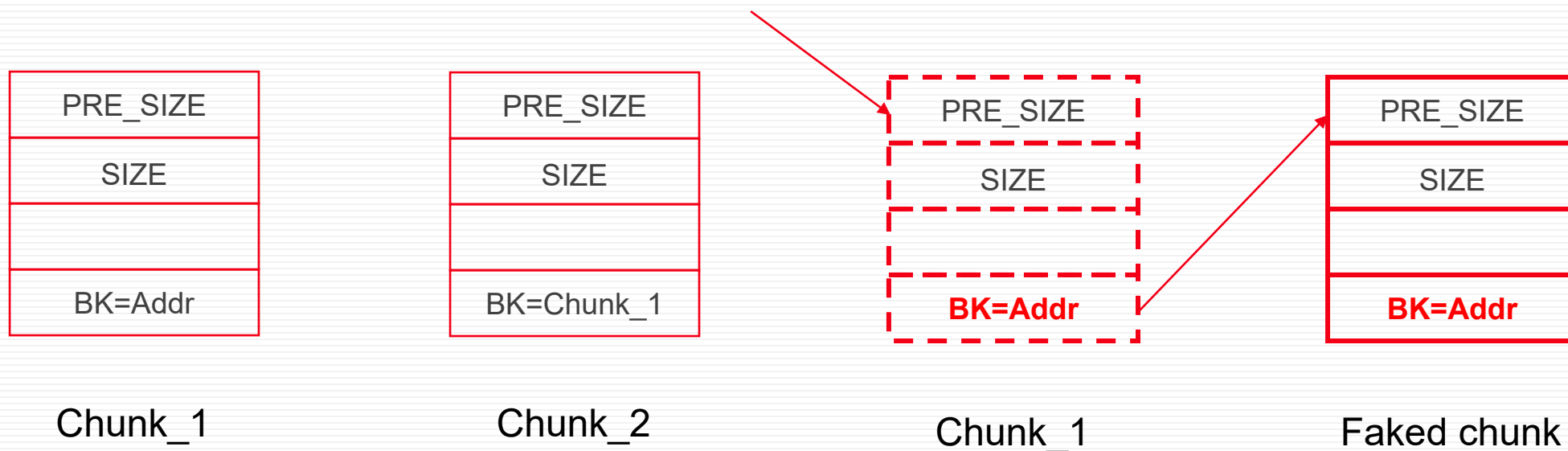
Fastbin double free利用示例

- 利用过程
- `Free(chunk_1); Free(chunk_2); Free(chunk_1);`
- `Malloc(); write_to_chunk_1;`



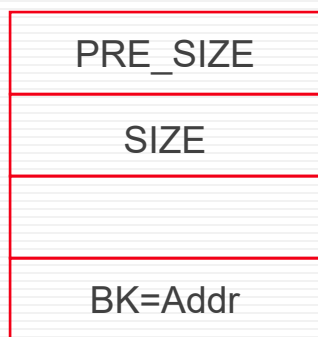
Fastbin double free利用示例

- 利用过程
- `free(chunk_1); free(chunk_2); free(chunk_1);`
- `malloc(); write_to_chunk_1; malloc();`

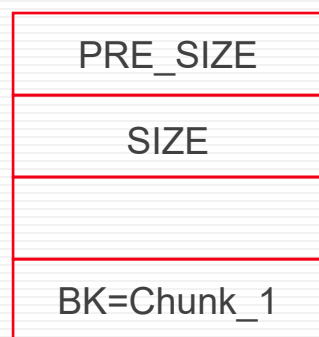


Fastbin double free利用示例

- 利用过程
- `free(chunk_1); free(chunk_2); free(chunk_1);`
- `malloc(); write_to_chunk_1; malloc(); malloc();`



Chunk_1



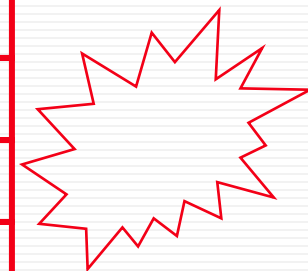
Chunk_2



Chunk_1

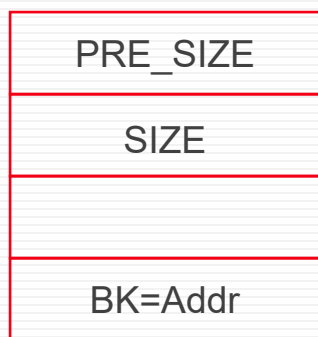


Faked chunk

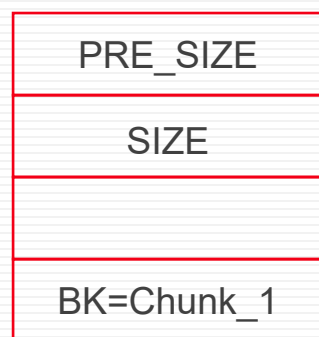


Fastbin double free利用示例

- 利用过程
- `free(chunk_1); free(chunk_2); free(chunk_1);`
- `malloc(); write_to_chunk_1; malloc(); malloc();`
- `p=malloc(); *p = 任意地址写`



Chunk_1



Chunk_2



Chunk_1



Faked chunk



堆漏洞利用

- 其他利用方式

- house of系列
- unsorted bin attack
- large bin attack
- tcache attack

- 总结

- 阅读源码，发现堆管理的某个部分缺少了对应的检查(或者说为了速度省略了某些检查)，便可以尝试对这部分缺陷代码进行利用。

学习资源

- How2heap
 - <https://github.com/shellphish/how2heap.git>
- ctf-wiki
 - <https://ctf-wiki.org/>
- black hat heap exploitation
- glibc source code
- Linux Heap Internals (memeda@0ops)