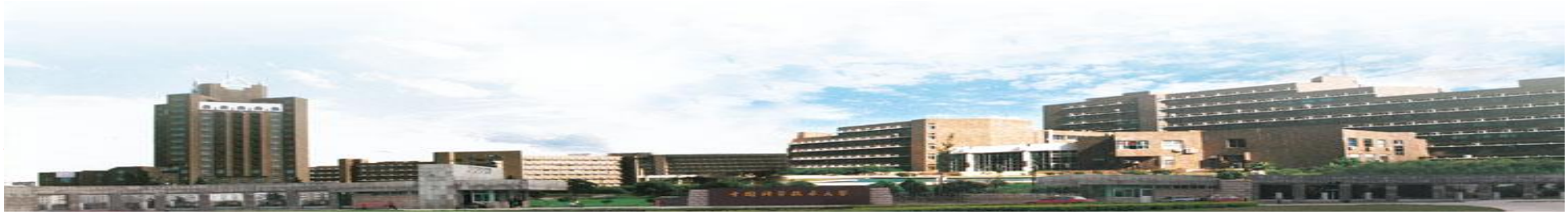
Sponsors of Tomorrow.™

缓冲区溢出
--堆溢出

程绍银

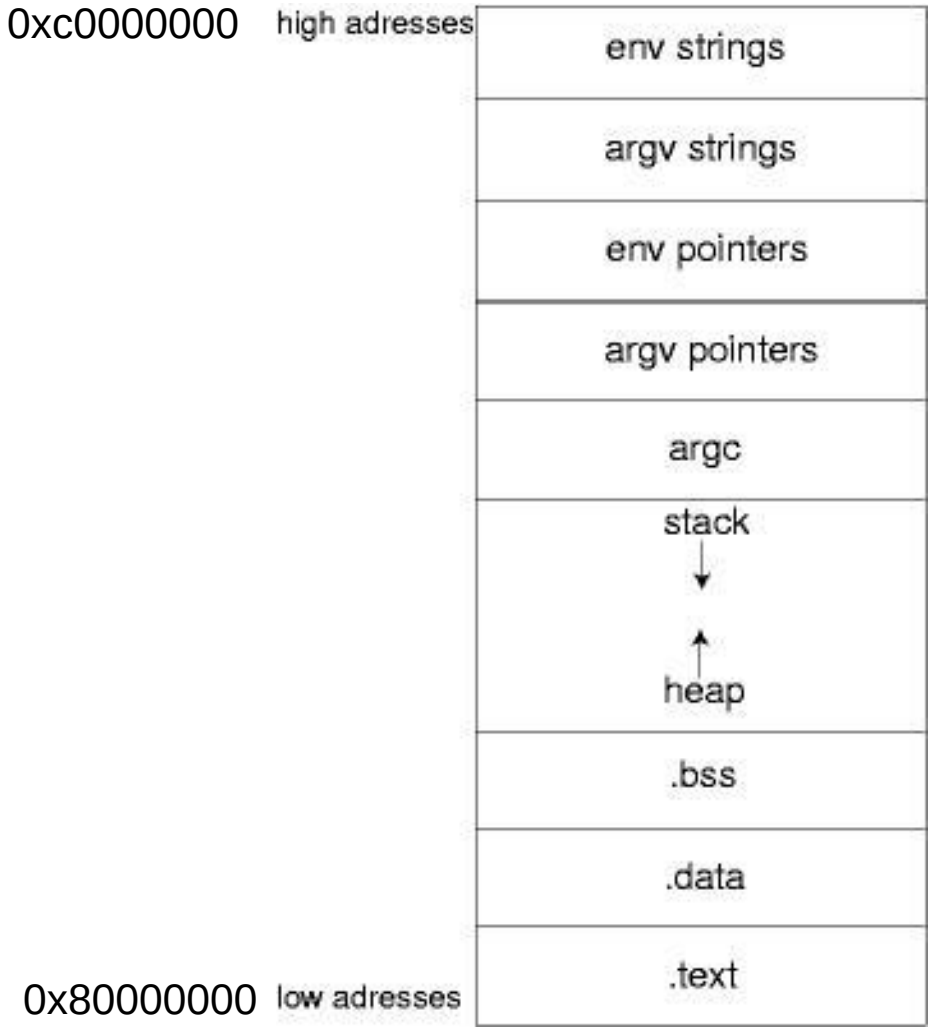
sycheng@ustc.edu.cn



本章内容

- 堆溢出

堆在进程空间中的位置

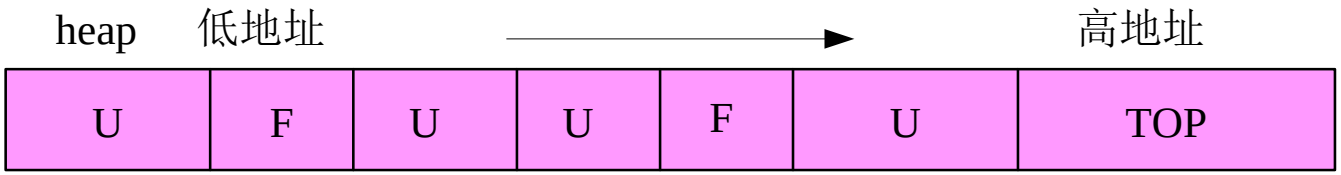

0xc0000000
high adresses
env strings
argv strings
env pointers
argv pointers
argc
stack
heap
.bss
.data
.text
0x80000000
low adresses

Linux堆管理算法

- Linux系统通过glibc程序库提供堆内存管理功能，存在两种堆管理算法
  - glibc2.2.4及以下版本是使用Doug Lea的实现方法
  - glibc2.2.5及以上版本采用了Wolfram Gloger的ptmalloc/ptmalloc2代码。ptmalloc2代码是从Doug Lea的代码移植过来的，增加了对多线程的支持，并引进了fastbin机制
- 从Doug lea的实现方法说起

Linux堆管理结构

- Linux整个堆区被划分成若干个连续的块(chunk)，类似下图分布



- 标记：

  U — 正在被使用的块

  F — 空闲的块

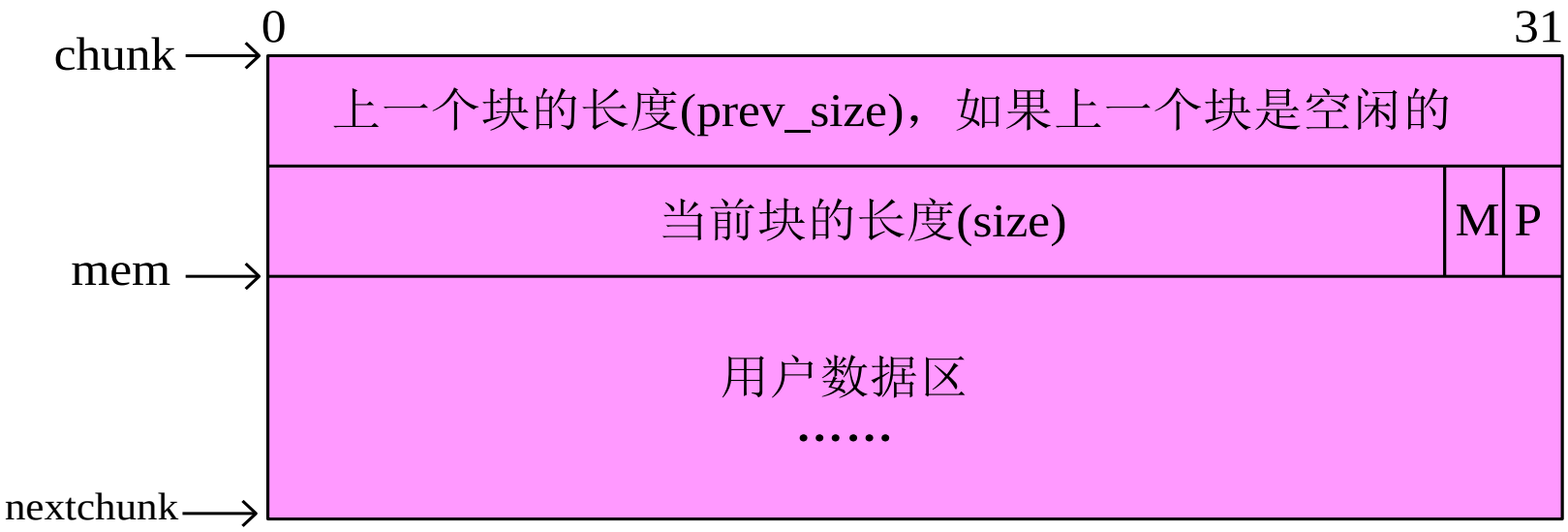
  TOP — 位于高地址最边缘的那个块

堆块(chunk)的结构

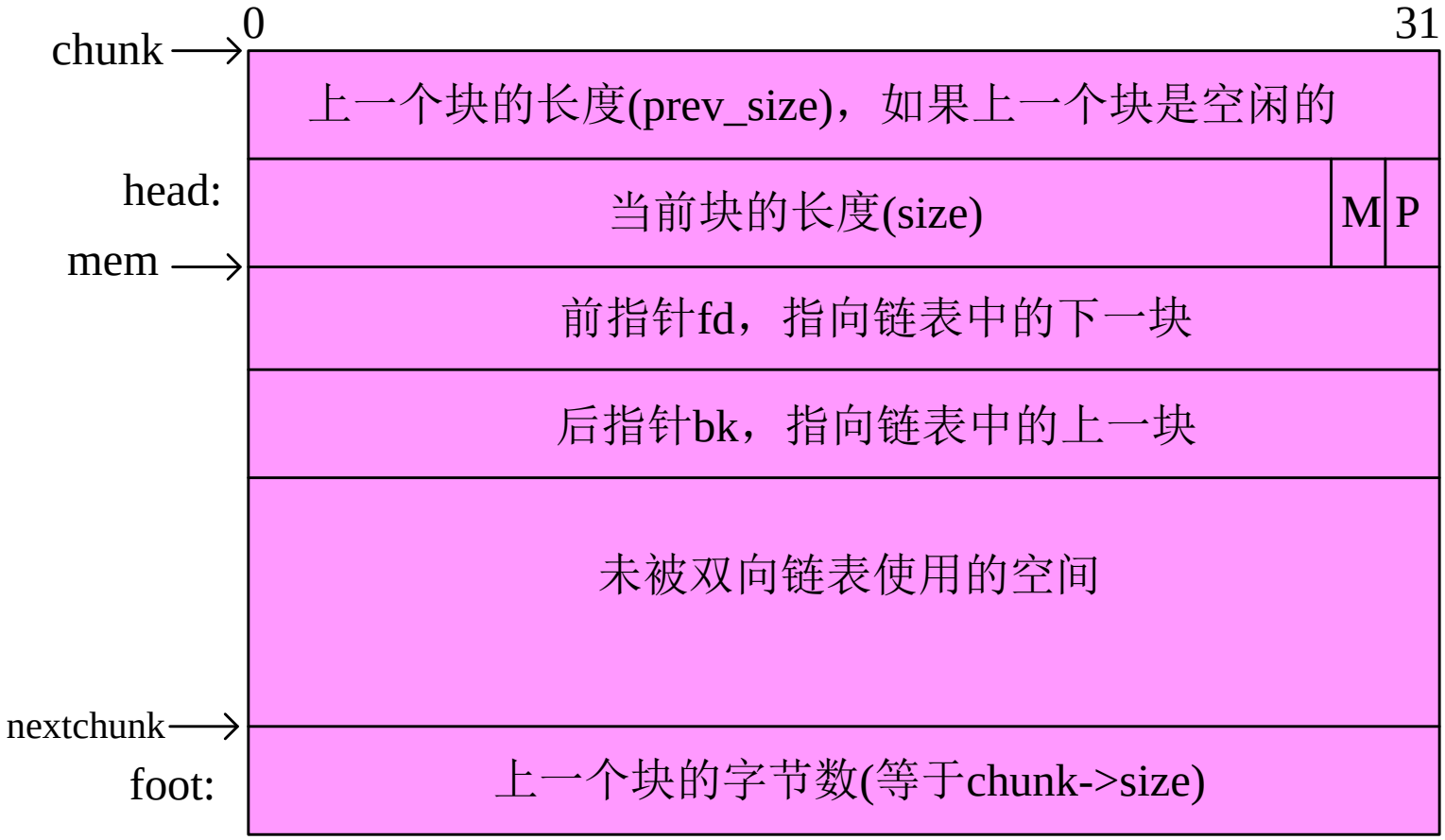
- Chunk的头部有一个用于管理当前块的管理结构，这个管理结构的长度对于正在使用的chunk是8字节，而对于空闲的chunk则为16字节
- 管理结构的定义如下：

```
struct malloc_chunk
{
  int  prev_size; // 如果上一块是空闲，此值为上一块的长度
  int size;      // 当前块的长度包括管理结构本身
  struct malloc_chunk* fd;  //双向链表的前指针
  struct malloc_chunk* bk;  // 双向链表的后指针
}
```

U块的结构


chunk
0
31
上一个块的长度(prev_size)，如果上一个块是空闲的
当前块的长度(size)
M
P
mem
用户数据区
……
nextchunk

F块的结构


chunk
0
31
上一个块的长度(prev_size)，如果上一个块是空闲的
head:
当前块的长度(size)
M
P
mem
前指针fd，指向链表中的下一块
后指针bk，指向链表中的上一块
未被双向链表使用的空间
nextchunk
foot:
上一个块的字节数(等于chunk->size)

P标志位

- “P”标志是“当前块字节数”(chunk->size)中的最低一位，表示是否上一块正在被使用
- 如果P位置为1，则表示上一块正在被使用，这时chunk->prev_size通常为零
- 如果P位为零，则表示上一块是空闲块,这时chunk->prev_size字段其实是上一个块用户数据区的一部分
- 任何一个块的空闲与否是由下一个块的chunk->size的P标志来确定的

M标志位

- “M”位是表示此内存块是不是由mmap()分配的
- 如果置1，则是由mmap()分配的，那么在释放时会以另外的方式处理，与现在的讨论无关
- P和M标志位的定义如下：
  - #define PREV_INUSE 0x1
  - #define IS_MMAPPED 0x2

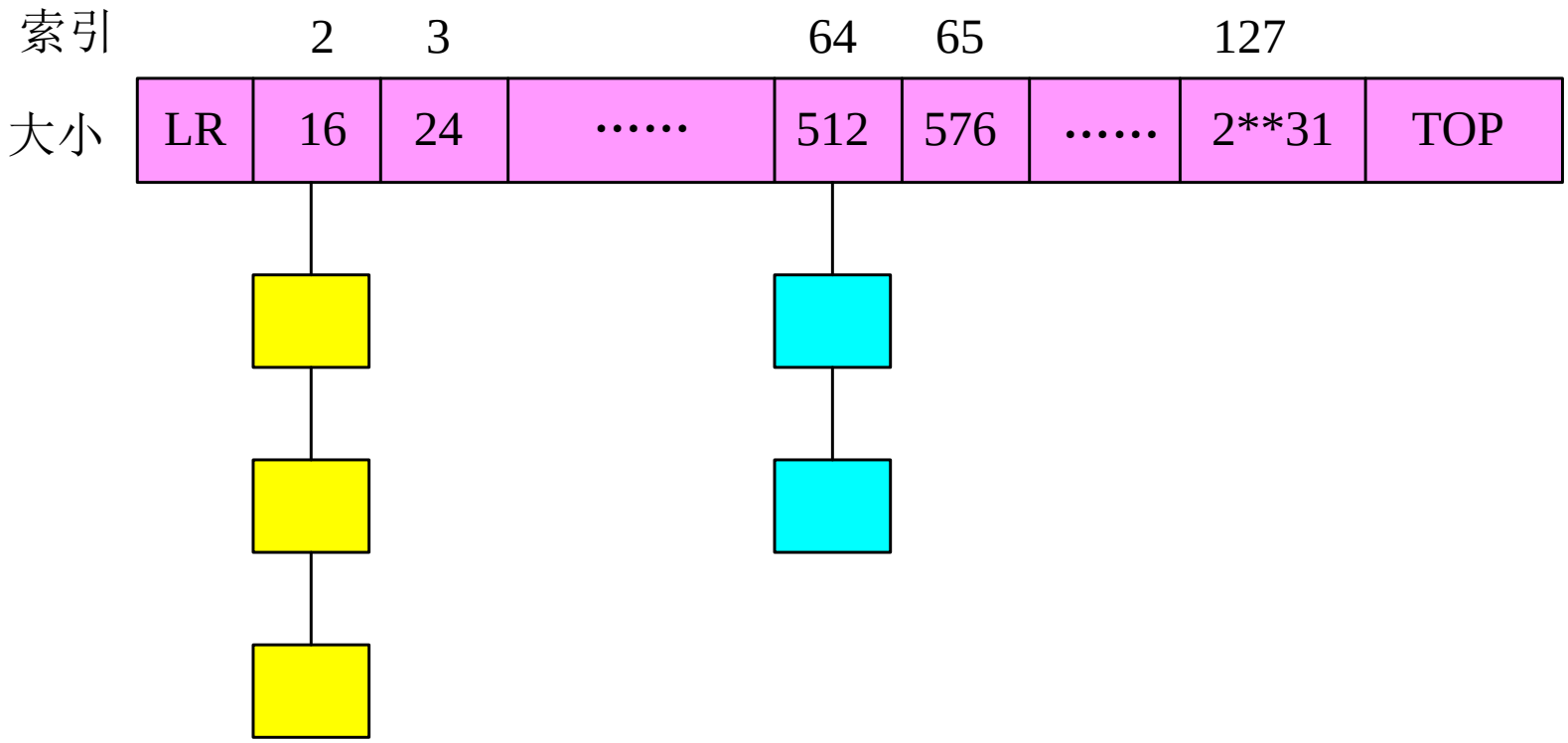
双向链表Bin

- 在Doug Lea实现的Malloc中，除了TOP块外的所有空闲块以长度大小被分组，它们的信息被存放在称为Bin的双向循环链表中，共有128个表项，它们分别存放特定长度的空闲内存块信息
- 系统在分配内存时首先会到Bin链表中寻找是否存在合适大小的内存块，如果找不到才会从TOP块中分割出一块来并把指针返回给用户进程

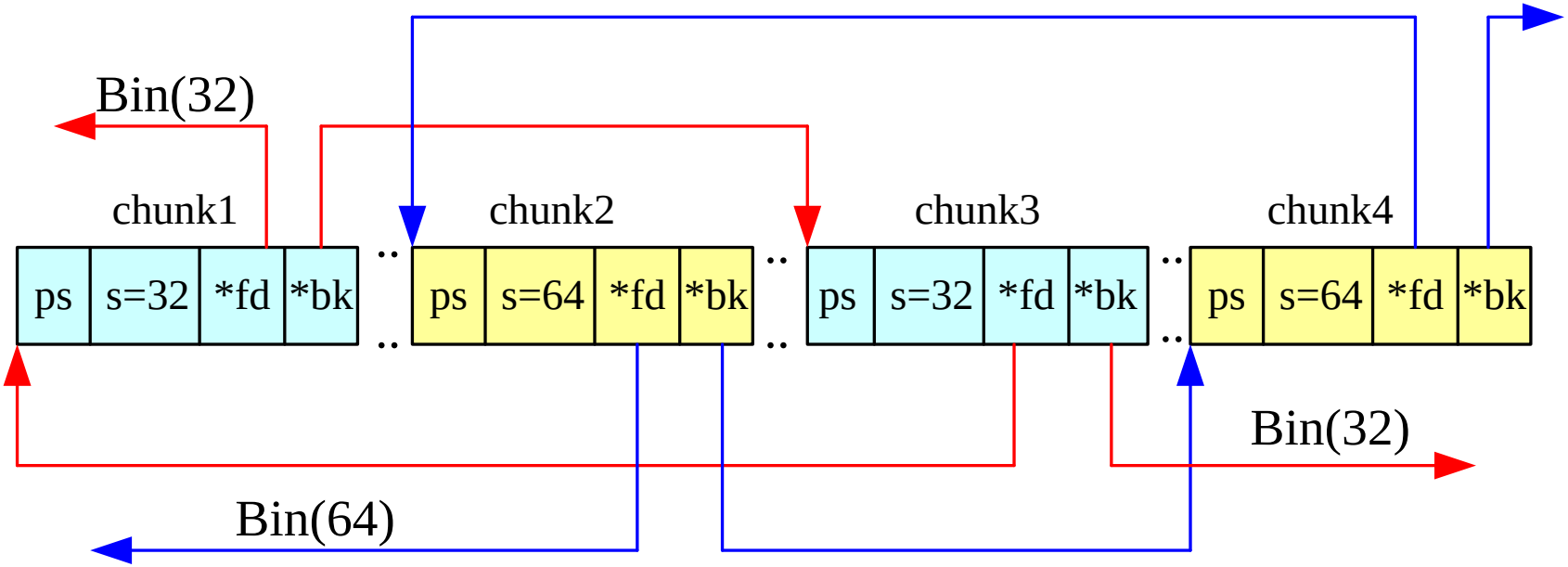
Bin链表索引

- Bin链表的索引指针存放在一个数组中，Malloc根据块的大小来获取Bin链表指针在数组中的索引
- 小于512字节的空闲内存块被认为是小块，由于内存块的大小一定大于等于16字节而且是8的倍数，这些小块的信息被存放在62个Bin链表中
  - 第1个Bin链表存放16字节大小内存块的信息；第2个链表存放24字节；第3个链表存放32字节的；
  - 以此类推，第62个Bin链表存放504字节大小的空闲块信息
  - 所有小块单个Bin链表所存放的块的大小都是一样的
- 所有大于504字节的空闲块被认为是大块，它们的信息存放在后66个Bin链表中，每个Bin链表存放了一定长度范围的空闲块

Bin链表示意图



Bin链表的链接情况

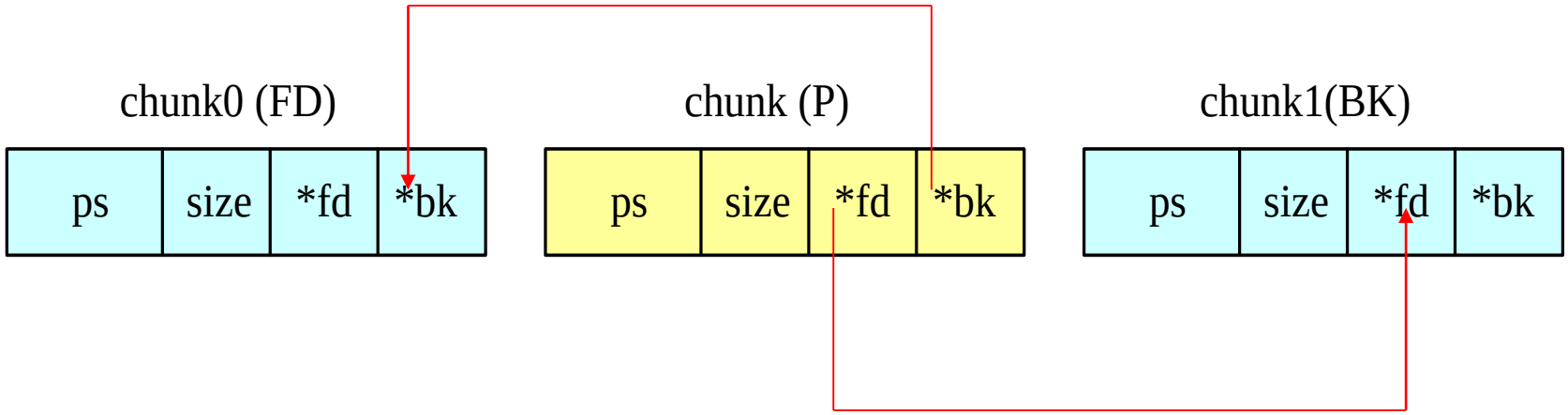

Bin(32)
chunk1
chunk2
chunk3
chunk4
ps
s=32
*fd
*bk
ps
s=64
*fd
*bk
ps
s=32
*fd
*bk
ps
s=64
*fd
*bk
Bin(32)
Bin(64)

链表单元的删除

- Malloc的实现使用两个宏来完成对于Bin链表的插入和删除操作
- 用于删除单元的unlink宏定义如下：

```
#define unlink( P, BK, FD ) {          \
    BK = P->bk;                        \
    FD = P->fd;                        \
    FD->bk = BK;                       \
    BK->fd = FD;                       \
}
```

删除操作图示



箭头方向为数据移动方向，宏实际所进行的操作就是：

chunk1->fd <== chunk->fd

chunk0->bk <== chunk->bk

也就是：

chunk->bk + 8 <== chunk->fd

chunk->fd + 12 <== chunk->bk

目标

- unlink宏有两个写内存的操作。 fd被写到(chunk->bk + 8)中，而bk被写到(chunk->fd + 12)中
- 如果能控制fd和bk这两个指针的值，就可以将任意4个字节内容写到任意一个内存地址中去！这正是所期望的
- 目标就是设法控制fd和bk中的内容，并按照期望触发unlink宏操作，改变程序执行流程
- 首先必须清楚unlink宏的调用位置

unlink宏的调用位置

- malloc()和free()的实现里面都使用到了unlink宏。触发malloc()里的unlink操作比较困难，所以先从free()说起

```
void free(Void_t* mem){
    ...
    if(chunk_is_mmapped(p)) // 如果IS_MMAPPED位被设置
    {
      munmap_chunk(p);
      return;
    }
    ...
    p = mem2chunk(mem);  // 将用户地址转换成内部地址: p = mem - 8
    ...
    chunk_free(ar_ptr, p);
}
```

chunk_free的执行流程

如果下一块是top节点，则与之合并

如果上一块是空闲的，则与之合并

```
p = chunk_at_offset(p, -prevsz);
unlink(p, bck, fwd);  /* 从链表中删除上一个结点 */
```

...

top=(ar_ptr) = p;//合并到top块

如果下一块不是top节点

如果上一块是空闲的，则与之合并

```
p = chunk_at_offset(p, -prevsz);
unlink(p, bck, fwd);  /* 从链表中删除上一个结点 */
```

如果下一个块是空闲的，则与之合并

```
unlink(next, bck, fwd); /* 从链表中删除下一个结点 */
```

如果前后两块都不是空闲的，则做一些设置工作，然后将当前块插入到空闲链表中

利用需满足的条件

- 有3个地方调用了unlink.如果想要执行它们，需要满足下列条件：
  1. 当前块的IS_MMAPPED位必须被清零，否则不会执行chunk_free()
  2. 上一个块是个空闲块 (当前块size的PREV_INUSE位清零)或者
  3. 下一个块是个空闲块(下下一个块（p->next->next）size的PREV_INUSE位清零)

一个实例程序

```
#include <stdlib.h>
int main (int argc, char *argv[])
{
        char *buf, *buf1;
        buf = malloc (16); /* 分配两块16字节内存 */
        buf1 = malloc (16);
        if (argc > 1)
            memcpy (buf, argv[1], strlen (argv[1])); /* 这里会发生溢出 */
        printf ("%#p [ buf  ] (%.2d) : %s \n", buf, strlen(buf), buf);
        printf ("%#p [ buf1 ] (%.2d) : %s \n", buf1, strlen(buf1), buf1);
        printf ("From buf to buf1 : %d\n\n", buf1 - buf);
        printf ("Before free buf\n");
        free (buf); /* 释放buf */
        printf ("Before free buf1\n");
        free (buf1); /* 释放buf1 */
        return 0;
} /* End of main */
```

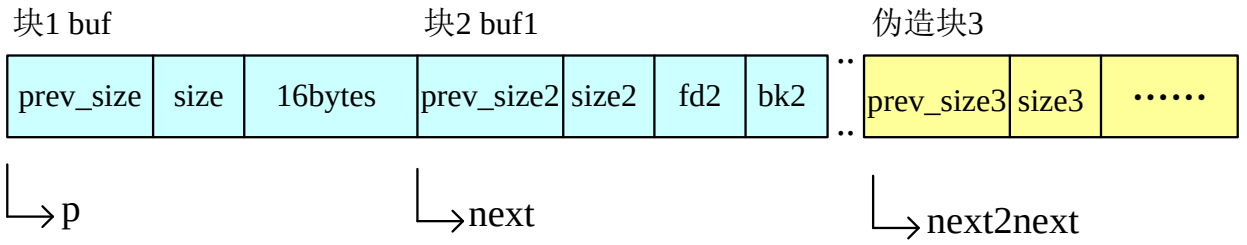
heap_overflow.c

弱点程序分析

- 弱点程序发生溢出时，可以覆盖下一个块的内部结构，但是并不能修改当前块的内部结构，因此条件2是满足不了的。只能寄希望于条件3
- 所谓下下一个块的地址其实是由下一个块的数据来推算出来的，既然可以完全控制下一个块的数据，就可以让下下一个块的size的PREV_INUSE位为零。这样程序就会认为下一个块是个空闲块了

利用思路

- 假设当前块为块1,下一个块为块2，下下一个块为块3，如下图所示：



next = p + (size & ~PREV_INUSE)

next2next = next + (size2 & ~(PREV_INUSE|IS_MMAPPED))

- 只要能够通过修改size2，使得next2next指向一个可以控制的地址；之后在这个地址伪造一个块3，使得此块的size3的PREV_INUSE位置零即可；然后，在fd2处填入要覆盖的地址，例如函数返回地址，.dtors.GOT等等
- Solar Designer建议可以使用__free_hook()的地址，这样再下一次调用free()时就会执行我们的代码。在bk2处可以填入shellcode的地址

如何构造块？

- 实际构造的时候块2的结构如下：

  prev_size2 = 0x11223344 /* 可以使用任意值 */

  size2 = (next2next - next) /* 这个数值必须是4的倍数 */

  fd2 = __free_hook - 12 /* 将shellcode地址刚好覆盖到__free_hook地址处 */

  bk2= shellcode

- 伪造的块3则要求很低，只需要让size3的最后一位为0即可：

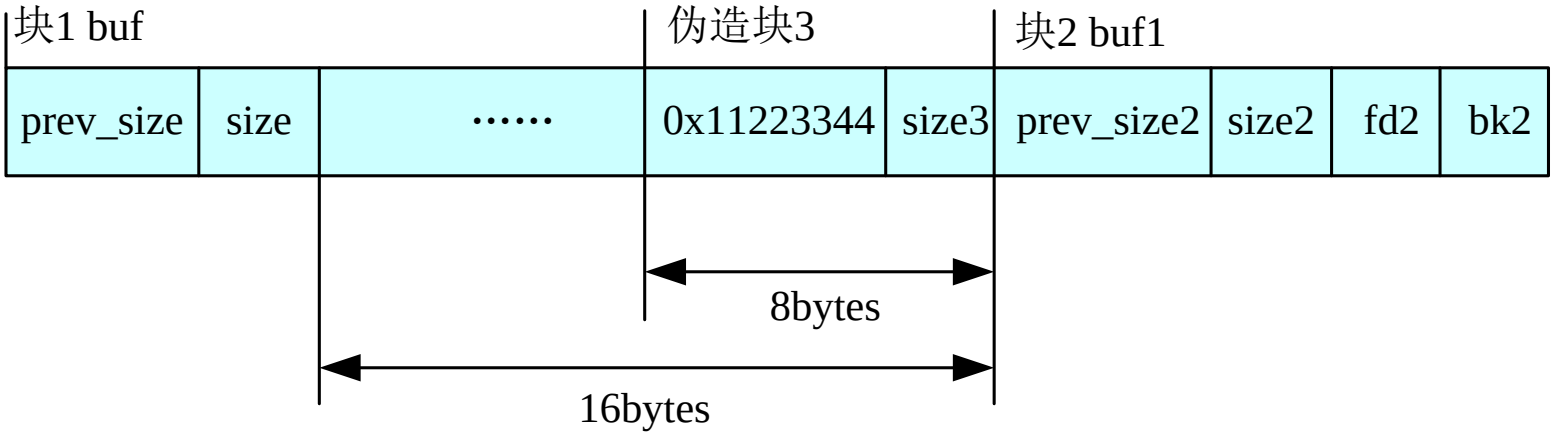
  prev_size3 = 0x11223344 /* 可以使用任意值 */

  size3 = 0xffffffff & ~PREV_INUSE /* 这里的0xffffffff可以用任意非零值替换 */

伪造块的位置问题

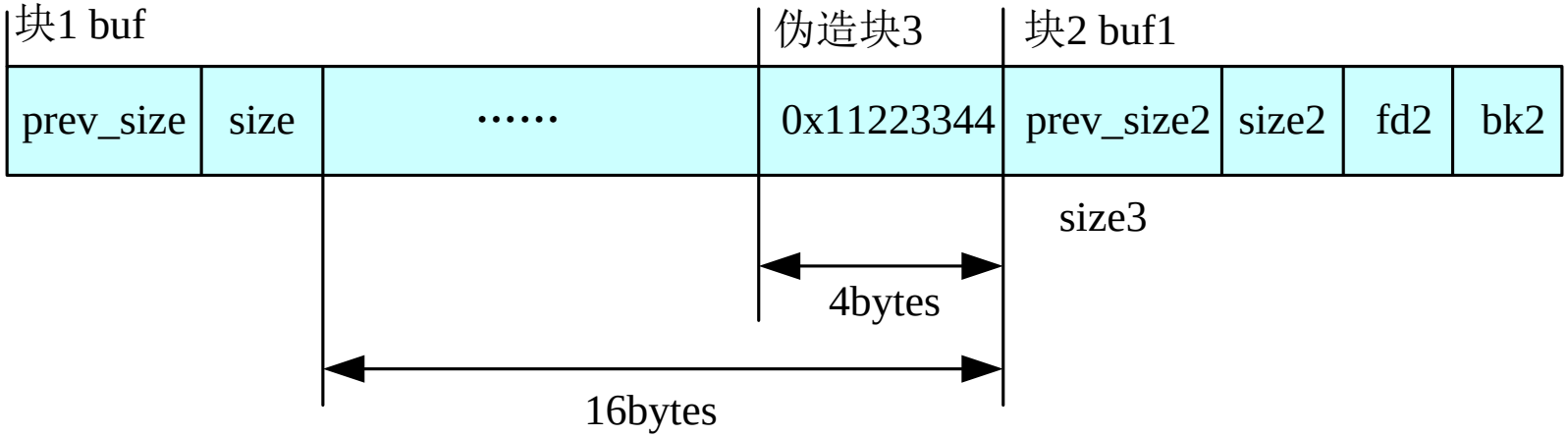
- 伪造的块3可以放在任意可能的位置，例如块2的前面或者后面
- 如果要放在块2的后面，由于size2是4个字节，因此如果距离比较小的话，那么size2是肯定要包含零字节的，这会中断数据拷贝，因此距离必须足够远，以至于四个字节均不为零，堆栈段是一个不错的选择，通过设置环境变量等方法也可以准确的得到块3的地址
- 如果将块3放到块2的前面，那么size2就是个负值，通常是0xffffffxx等等。这肯定满足size2不为零的要求，另外，这个距离也可以很精确的指定。因此决定采用这种方法

第一种构造方法



- 在上面的图上，将块3的8字节的内部结构放在了块1的用户数据区中，而块3的用户数据区实际上是从块2开始的
- 但是既然根本不关心块3的prev_size以及数据段，而块2的prev_size也不关心，因此还可以有更简化的版本：将块3往右移动4个字节，即让siez3与prev_size2重合

更精简的构造办法



这样next2next - next = -4 = 0xfffffffc .则块2就可以重新构造一下：

prev_size2 = 0x11223344 & ~PREV_INUSE /* 用原来的size3代替 */

size2 = 0xfffffffc /* 长度为-4 */

fd2 = __free_hook - 12 /*将shellcode地址刚好覆盖到__free_hook地址处*/

bk2 = shellcode

堆溢出－小结

- 要达到利用free()函数调用来攻击的目的，需要满足以下条件：
  1、通过某些漏洞（例如堆溢出）来覆盖将要被free()的chunk
  2、在被覆盖chunk的位置上构造fake_chunk
  3、fake_chunk要确保在free()函数调用过程中运行unlink宏
  4、unlink宏所操作的内存将修改程序的流程

堆溢出－小结

- 一般有两种方法：
  1. 如果想利用上一块的unlink进行攻击，需要保证:
     I. chunk->size的IS_MMAPPED位为0
     II. chunk->size的PREV_INUSE位为0
     III. chunk + chunk->prev_size指向一个我们控制的伪造块结构；
     IV. 在一个确定的位置构造一个伪块
  2. 如果想利用下一个块的unlink进行攻击，需要保证:
     I. chunk->size的IS_MMAPPED位为0
     II. chunk->size的PREV_INUSE位为1
     III. chunk + nextsz 指向一个我们控制的伪造块结构。
     (nextsz = chunk->size & ~(PREV_INUSE|IS_MMAPPED))
     IV. 在一个确定的位置构造一个伪块

堆溢出－其他

- 前一部分说的都是利用free()的unlink()宏来改变程序流程的方法，其实在malloc()中也有unlink()宏，所以如果程序写的有问题的话，照样可以被用来利用改变程序流程，例如著名的double-free漏洞利用
- 新版堆管理算法ptmalloc2引入的fastbin机制客观上增加了溢出攻击的难度。必须设法绕过fastbin机制

堆溢出－参考资料

- 《网络渗透技术》
- Heap/bss溢出总结
- Ptmalloc2堆溢出利用初探
- LIBC环境下double-free堆操作漏洞利用原理及相关漏洞分析