Heap Exploitation

glibc - ptmalloc

angelboy

Outline

- Heap overview
  - Mechanism of glibc malloc
- Vulnerability of Heap
  - Use after free
  - Heap overflow
    - using unlink
    - using malloc maleficarum

Memory allocator

- dlmalloc – General purpose allocator
- ptmalloc2 – glibc
- jemalloc – Firefox
- tcmalloc – chrome
- .....

What is malloc

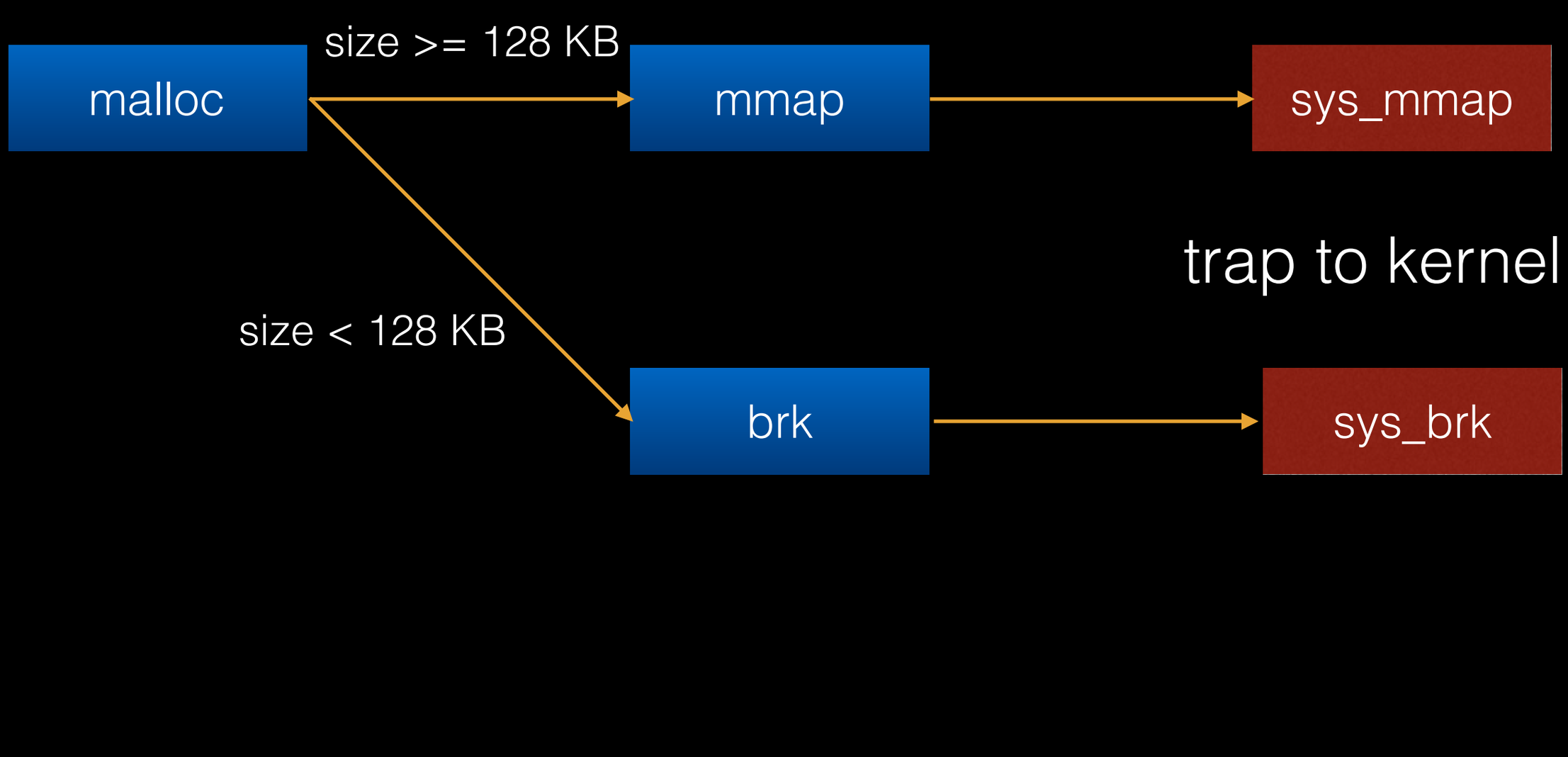
- A dynamic memory allocator
  - 可以更有效率的分配記憶體空間，要用多少就分配多少，不會造成記憶體空間的浪費

```c
#include <stdio.h>

int main(void){
    int size  = 0 ;
    char *p = NULL ;
    puts("Enter your length of name");
    scanf("%d",&size);
    p = (char *)malloc(size+1) ;
    puts("Enter your name");
    read(0,p,size);
    printf("Hello %s\n",p);
    free(p);
    return 0;
}
```

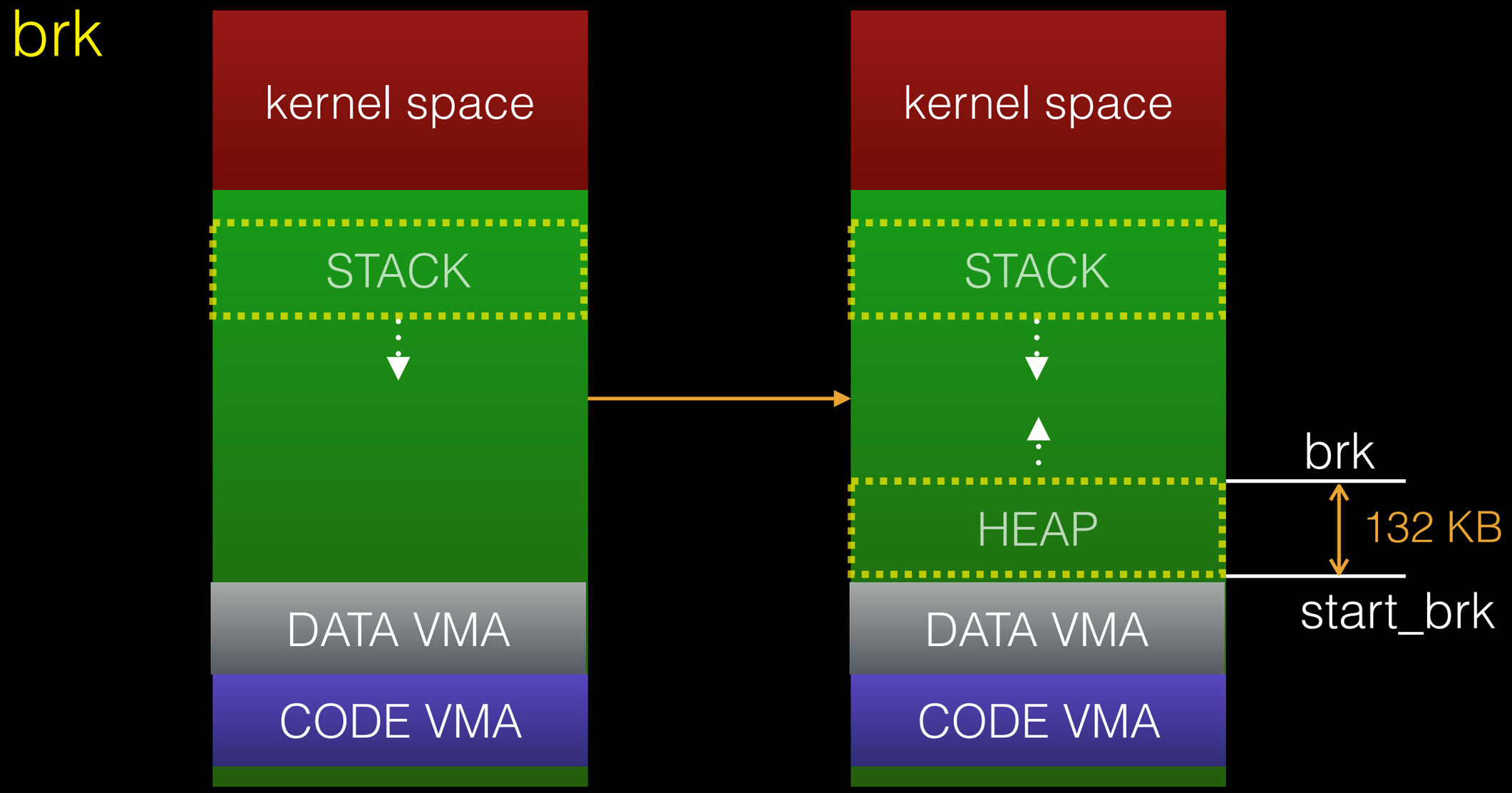
The workflow of malloc

- 第一次執行 malloc



The workflow of malloc

- 無論一開始 malloc 多少空間 < 128 KB 都會 kernel 都會給 132 KB 的 heap segment (rw) 這個部分稱為 main arena



The workflow of malloc

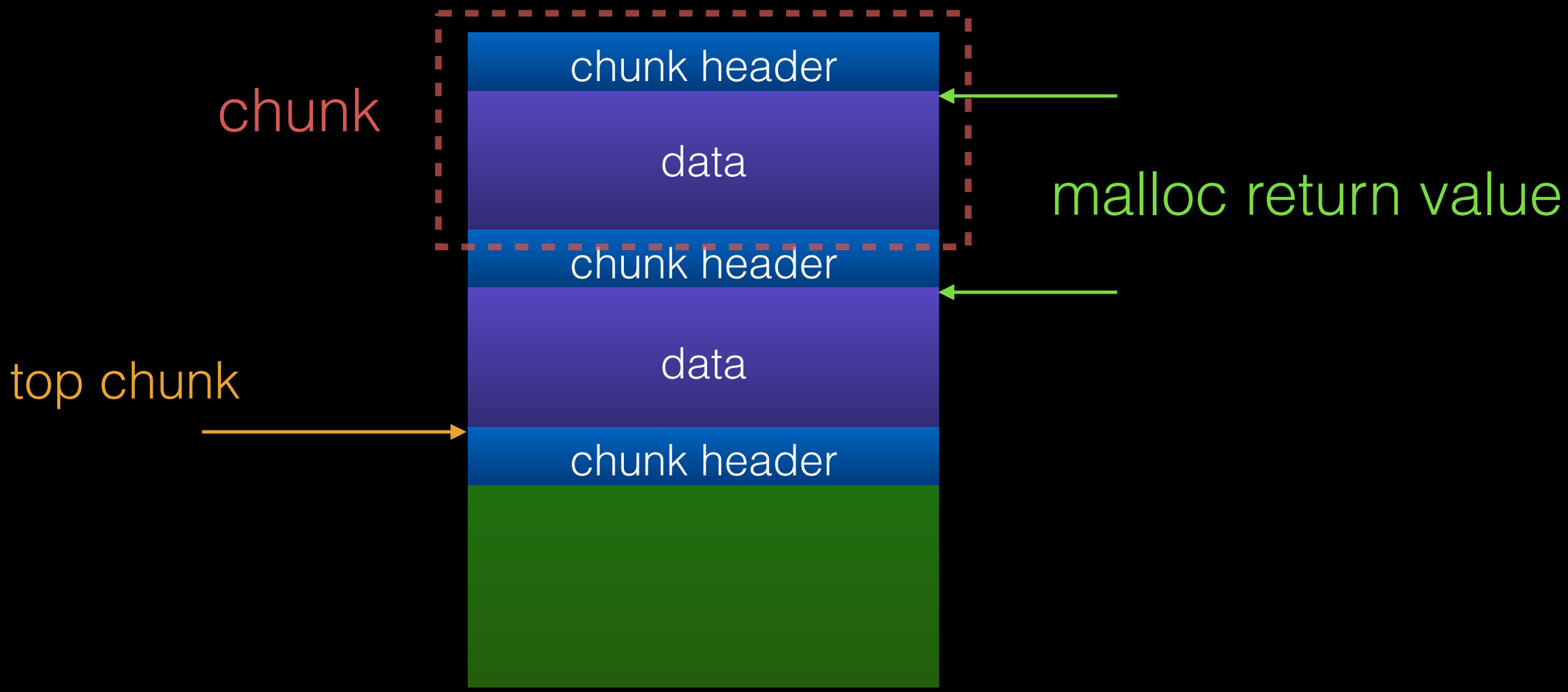
- 第二次執行 malloc 時，只要分配出去的總記憶體空間大小不超過 128 KB ，則不會再執行 system call 跟系統要空間，超過大小才會用 brk 來跟 kernel 要記憶體空間
  - 即使將所有 main arena 所分配出去的記憶體 free 完，也不會立即還給 kernel
- 這時的記憶體空間將由 glibc 來管理
- 本投影片如未特別註明都以 32 位元電腦 glibc-2.19 為主，換到 64 位元則大多數的 size 都需 x2

Mechanism of glibc malloc

- Chunk
  - glibc 在實作記憶體管理時的 data structure
  - 在 malloc 時所分配出去的空間及為一個 chunk
  - chunk header (prev_size + size) + user data
  - 如果該 chunk 被 free 則會將 chunk 加入名為 bin 的 linked list
- 分為
  - Allocated chunk
  - Free chunk
  - Top chunk

Mechanism of glibc malloc

- heap

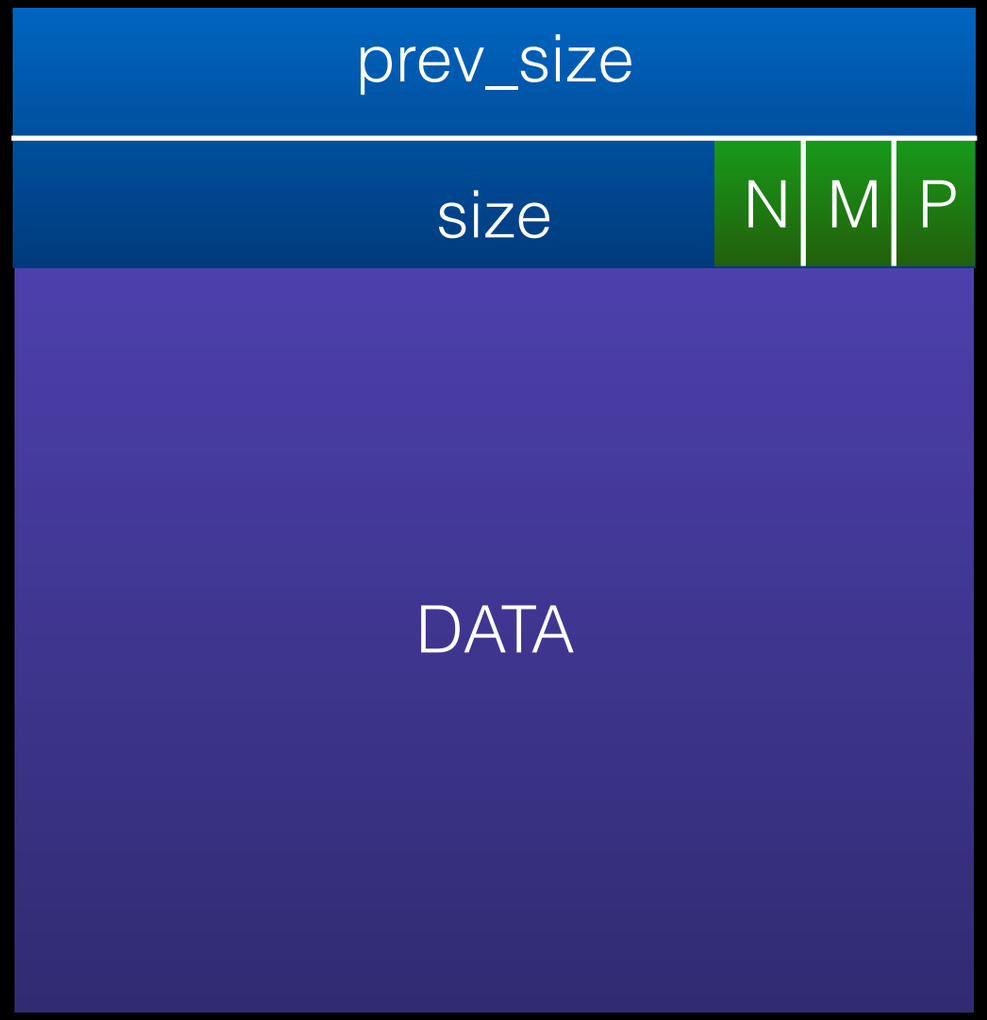


Mechanism of glibc malloc

- Allocated chunk
  - prev_size
    - 如果上一塊的 chunk 是 free 的狀態，則該欄位則會存有上一塊 chunk 的 size (包括 header)
      - 這裏指的上一塊是在連續記憶體中的上一塊
  - size
    - 該 chunk 的大小，其中有三個 flag
      - PREV_INUSE (bit 0) : 上一塊 chunk 是否不是 freed
      - IS_MMAPPED (bit 1) : 該 chunk 是不是由 mmap 所分配的
      - NON_MAIN_ARENA (bit 2)：是否不屬於 main arena

Mechanism of glibc malloc

- Allocated chunk



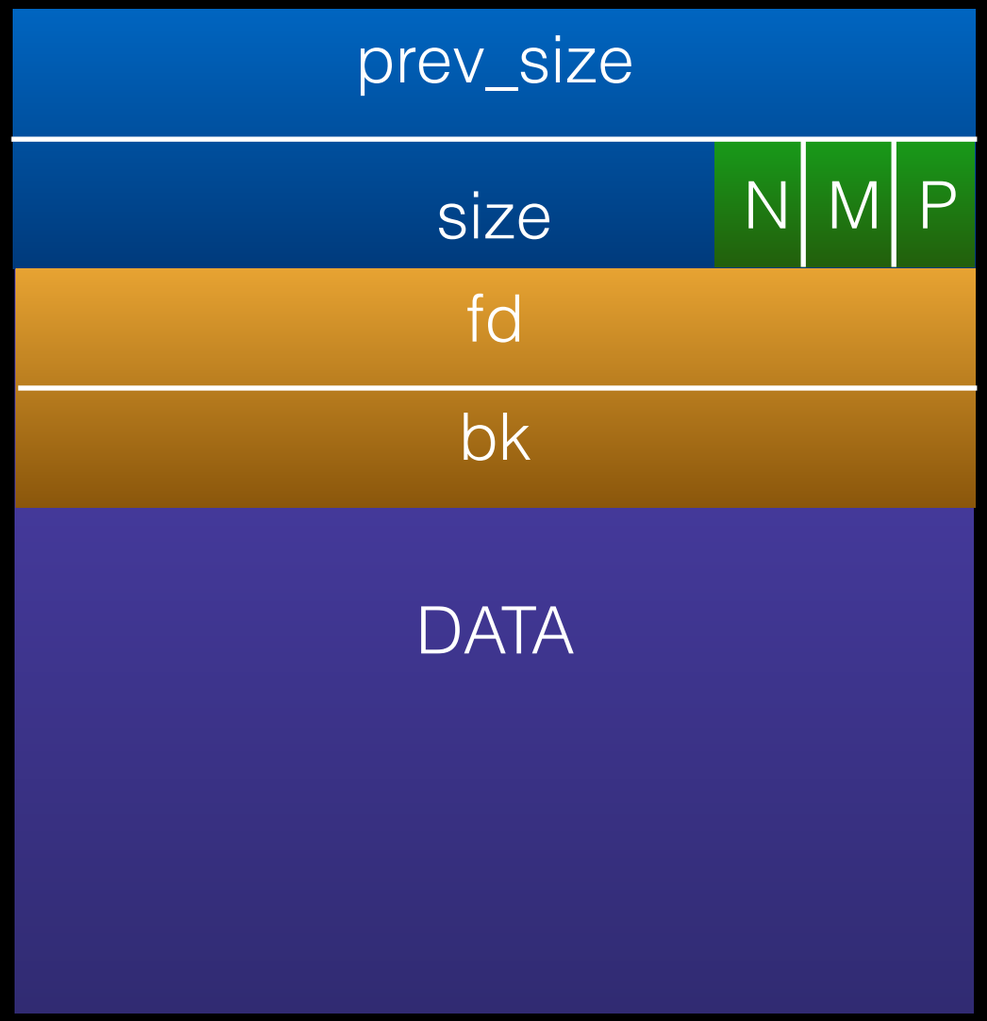
P:PREV INUSER
M:IS_MMAPPED
N:NON_MAIN_ARENA

Mechanism of glibc malloc

- freed chunk
  - prev_size
  - size
  - fd : point to next chunk (包含 bin)
    - 這邊指的是 linked list 中的 next chunk，而非連續記憶體中的 chunk
  - bk : point to last chunk (包含 bin)
    - 這邊指的是 linked list 中的 last chunk，而非連續記憶體中的 chunk
  - fd_nextsize : point to next large chunk（不包含 bin)
  - bk_nextsize : point to last large chunk （不包含 bin)

Mechanism of glibc malloc

- freed chunk



P:PREV INUSER
M:IS_MMAPPED
N:NON_MAIN_ARENA

Mechanism of glibc malloc

- top chunk
- 第一次 malloc 時就會將 heap 切成兩塊 chunk，第一塊 chunk 就是分配出去的 chunk ，剩下的空間視為 top chunk，之後要是分配空間不足時將會由 top chunk 切出去
- prev_size
- size
  - 顯示 top chunk 還剩下多少空間

Mechanism of glibc malloc

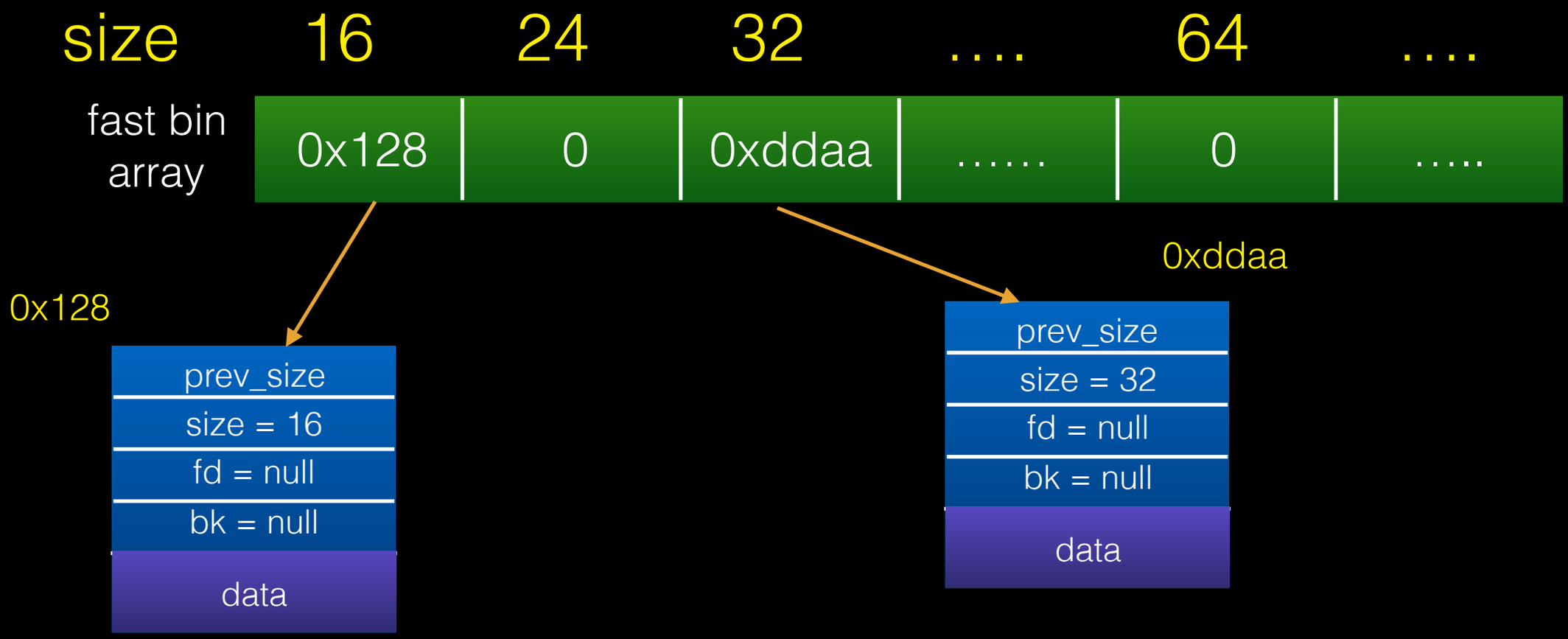
- bin
  - linked list
  - 為了讓 malloc 可以更快找到適合大小的 chunk，因此在 free 掉一個 chunk 時，會把該 chunk 根據大小加入適合的 bin 中
- 根據大小一共會分為
  - fast bin
  - small bin
  - large bin
  - unsorted bin

Mechanism of glibc malloc

- fast bin
  - a singly linked list
  - chunk size < 64 byte
    - 不取消 inuse flag
    - 依據 bin 中所存的 chunk 大小，在分為 10 個 fast bin 分別為 size 16,24,32…
    - LIFO
      - 當下次 malloc 大小與這次 free 大小相同時，會從相同的 bin 取出，也就是會取到相同位置的 chunk

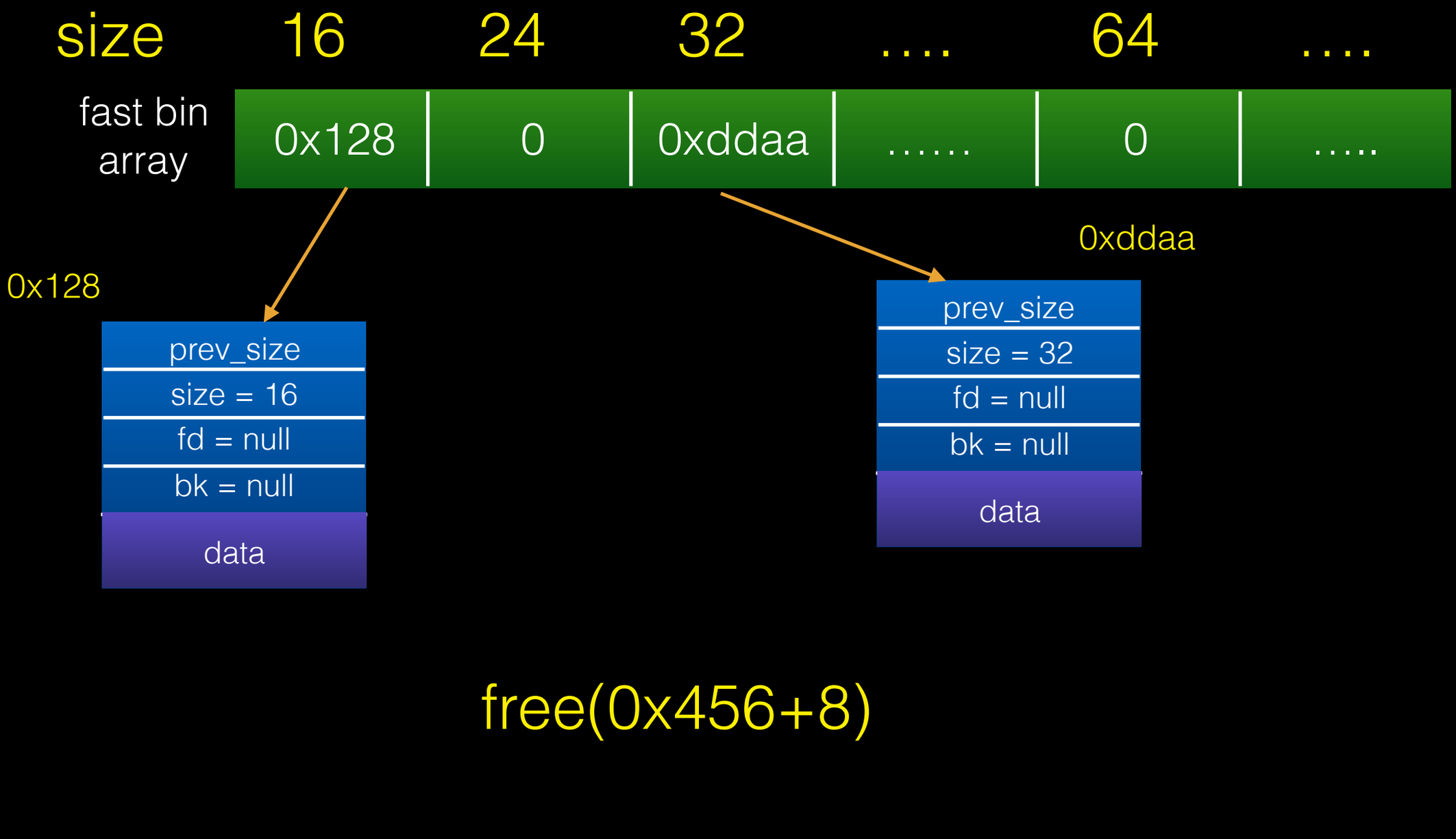
Mechanism of glibc malloc

- fast bin



Mechanism of glibc malloc

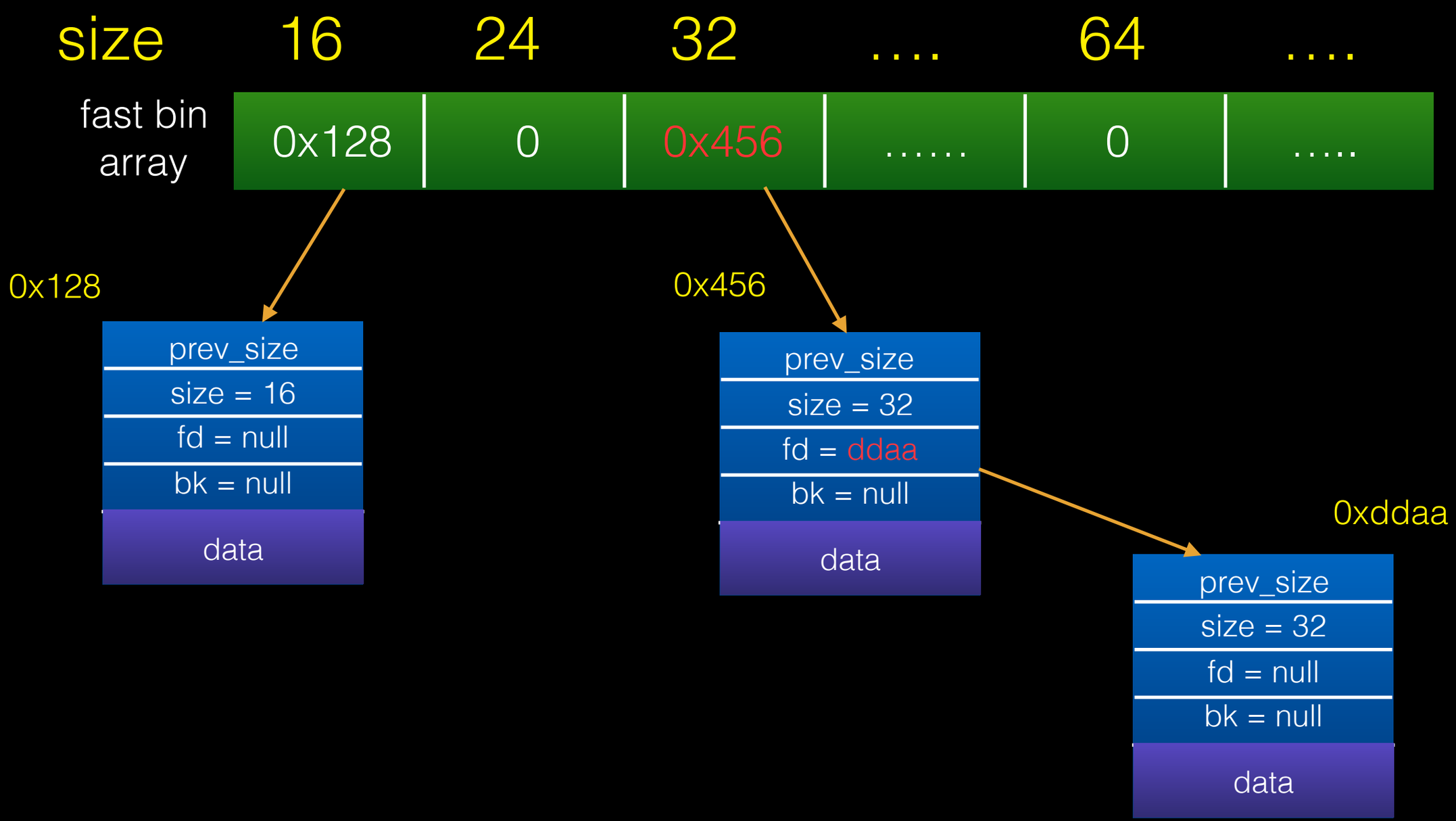
- fast bin



free(0x456+8)

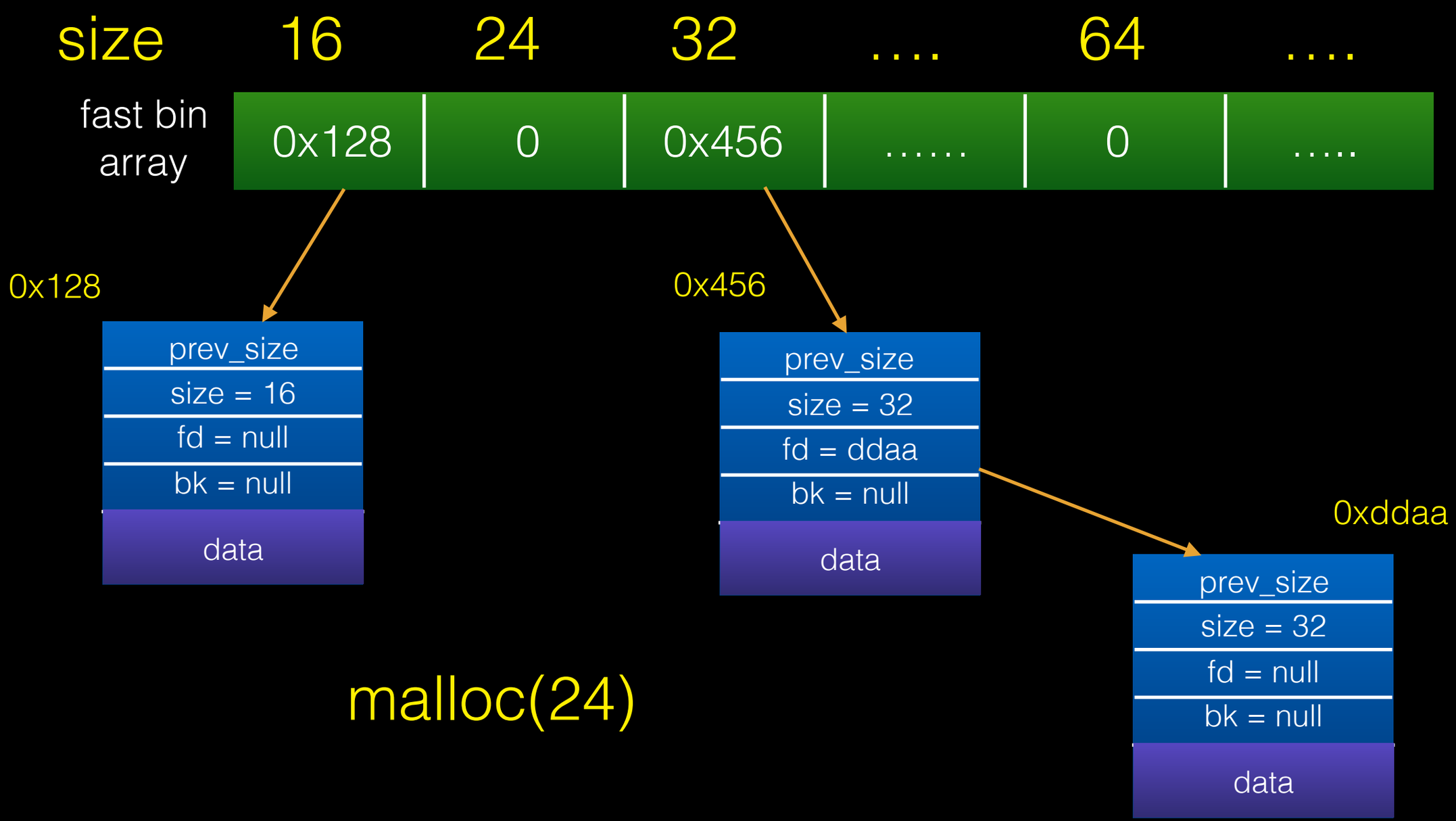
Mechanism of glibc malloc

- fast bin



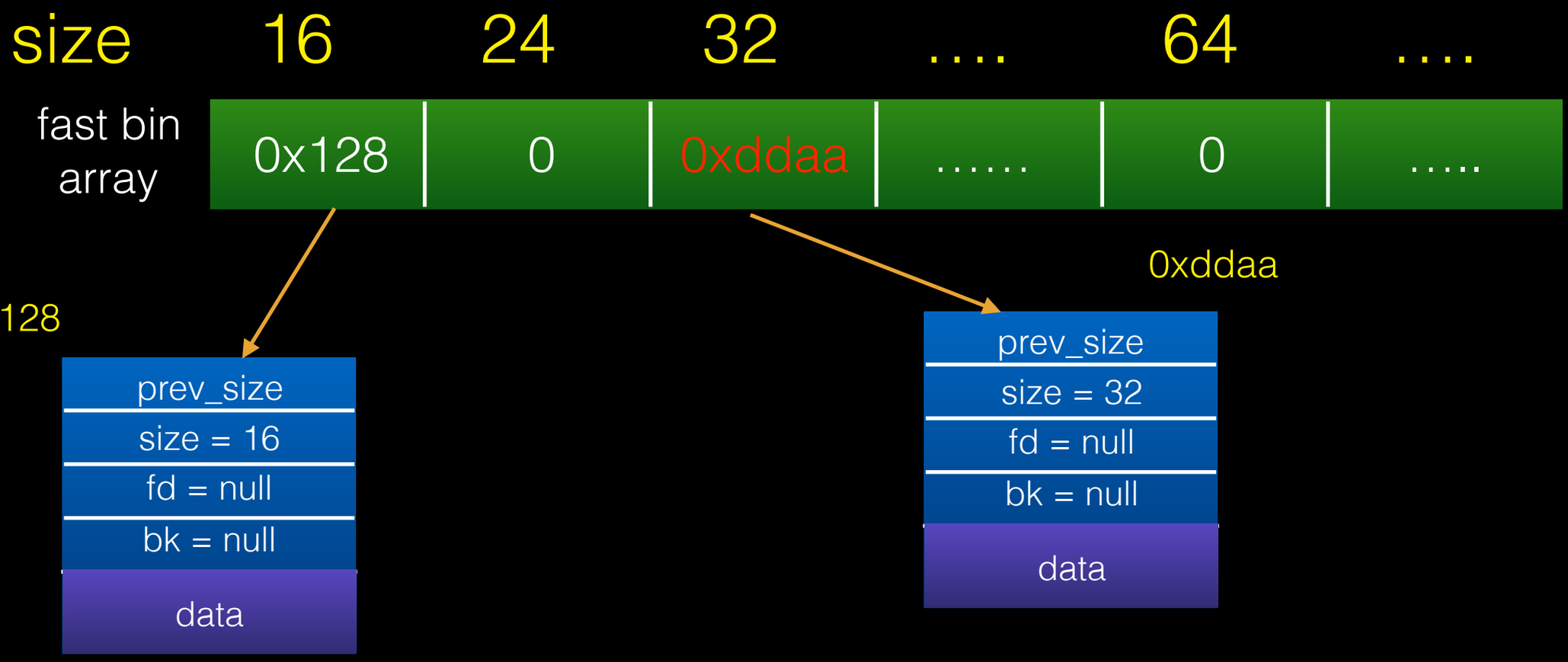
Mechanism of glibc malloc

- fast bin



Mechanism of glibc malloc

- fast bin



Mechanism of glibc malloc

- unsorted bin
  - circular doubly linked list
  - 當 free 的 chunk 大小大於 64 byte 時，為了效率，glibc 並不會馬上將 chunk 放到相對應的 bin 中，就會先放到 unsorted bin，在一段時間後，再慢慢將 unsorted bin 中的 chunk 加入到相對應的 bin 中
  - 而下次 malloc 時將會先找找看 unsorted bin 中是否有適合的 chunk ，找不到才會去對應得 bin 中尋找，但 small bin 除外，為了效率，反而先從 small bin 找

Mechanism of glibc malloc

- small bin
  - circular doubly linked list
    - chunk size < 512 byte
    - FIFO
    - 根據大小在分成 62 個大小不同的 bin
      - 16,24…64,72,80,88……508

Mechanism of glibc malloc

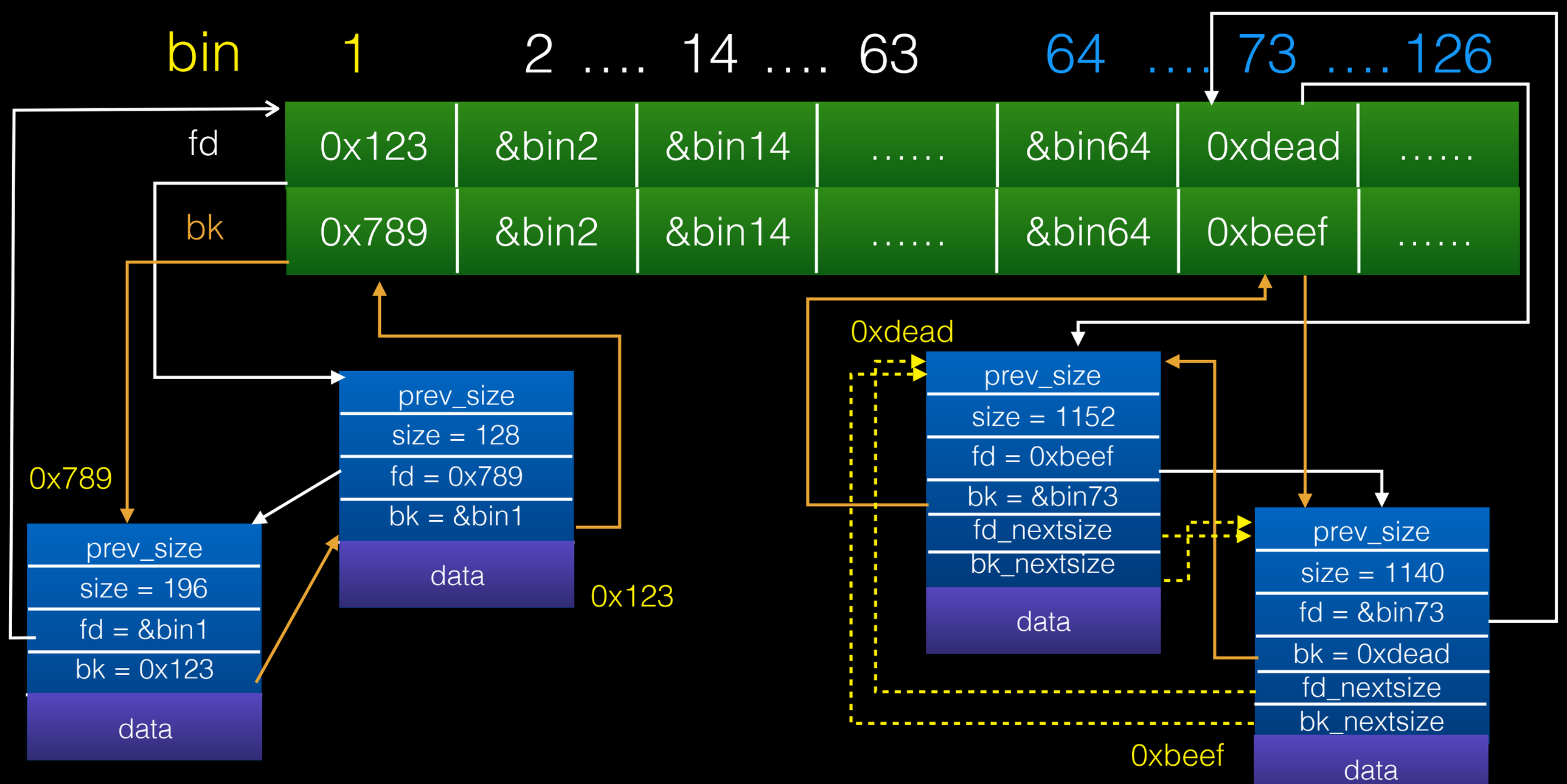
- large bin
  - circular doubly linked list (sorted list)
  - chunk size >= 512
  - freed chunk 多兩個欄位 fd_nextsize 、bk_nextsize 指向前一塊跟後一塊 large chunk
  - 根據大小在分成 63 個 bin 但大小不再是一固定大小增加
    - 前 32 個 bin 為 512+64*i
    - 32 - 48 bin 為 2496 + 512*j
    - ....依此類推
  - 不再是每個 bin 中的 chunk 大小都固定，每個 bin 中存著該範圍內不同大小的 bin 並在存的過程中進行 sort 用來加快 search 的速度，大的 chunk 會放在前面，小的 chunk 會放在後面

Mechanism of glibc malloc

- last remainder chunk
  - 在 malloc 一塊 chunk 時，如果有找到比較大的 chunk 可以給 user 會做 split 將 chunk 切成兩部分，多的那一部分會成為一塊 chunk 放到 last remander 中，unsortbin 也會存這一塊
  - 當下次 malloc 時，如果 last remainder chunk 夠大，則會繼續從 last remainder 切出來分配給 user

Mechanism of glibc malloc

- unsorted bin, small bin , large bin (chunk array)



Mechanism of glibc malloc

- main arena header
  - malloc_state
    - 存有 bins、fast bin 、 top chunk 等資訊
    - 位於 libc 的 bss 段中

Mechanism of glibc malloc

- Merge freed chunk
  - 為了避免 heap 中存在太多支離破碎的 chunk ，在 free 的時候會檢查周圍 chunk 是否為 free 並進行合併
- 合併後會進行 unlink 去除 bin 中重複的 chunk

Mechanism of glibc malloc

- Merge freed chunk
  - 執行 free 後 unlink 的條件，再 chunk 為非 mmaped 時
    - 如果下一塊是 top chunk，且上一塊是 free chunk
      - 最後合併到 top chunk
- 如果下一塊不是 top chunk
  - 上一塊是 free chunk
  - 下一塊是 free chunk

Mechanism of glibc malloc

- Merge freed chunk
    - 合併流程
        - 如果上一塊事 freed
            - 合併上一塊 chunk ，並對上一塊做 unlink
    - 如果下一塊是
        - top : 合併到 top
        - 一般 chunk :
            - freed : 合併下一塊 chunk ，並最下一塊做 unlink ，最後加入 unsortbin
            - inuse : 加入 unsortbin

Mechanism of glibc malloc

free(p)

small bin

0x4b000

0x4b000

unsorted bin

&unsortbin

&unsortbin

0x4b000

prev_size = 0

size = 0x80

fd = &small bin

bk = &small bin

prev_size = 0x80

size = 0x80

p

prev_size = 0

size = 0x81

Mechanism of glibc malloc

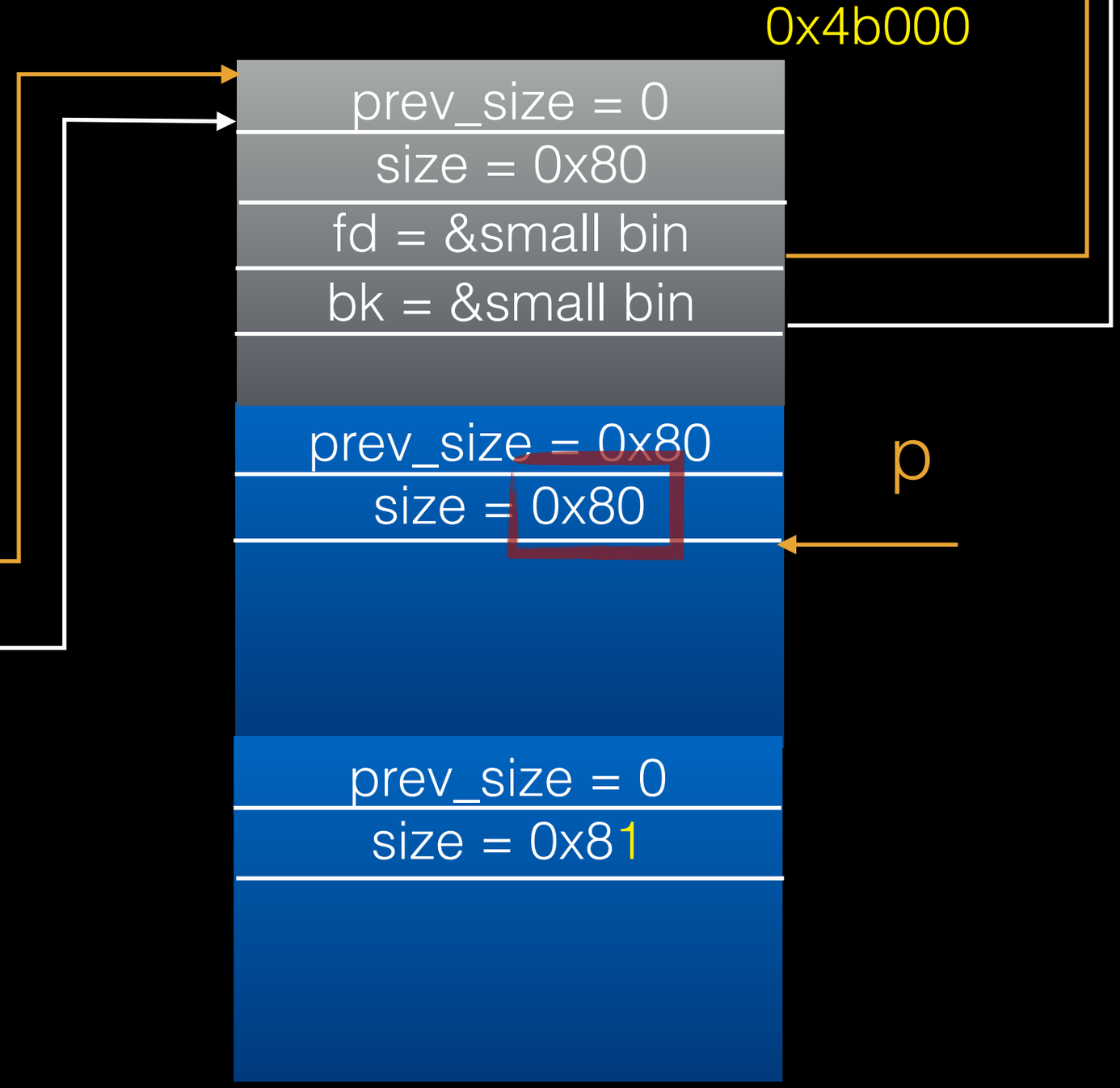
先利用 size 找到下
一塊 chunk
p - 8 + 0x80

small bin

0x4b000
0x4b000

unsorted bin

&unsortbin
&unsortbin

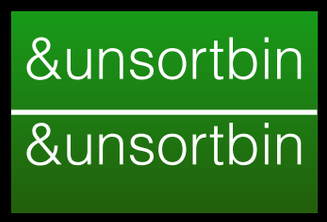


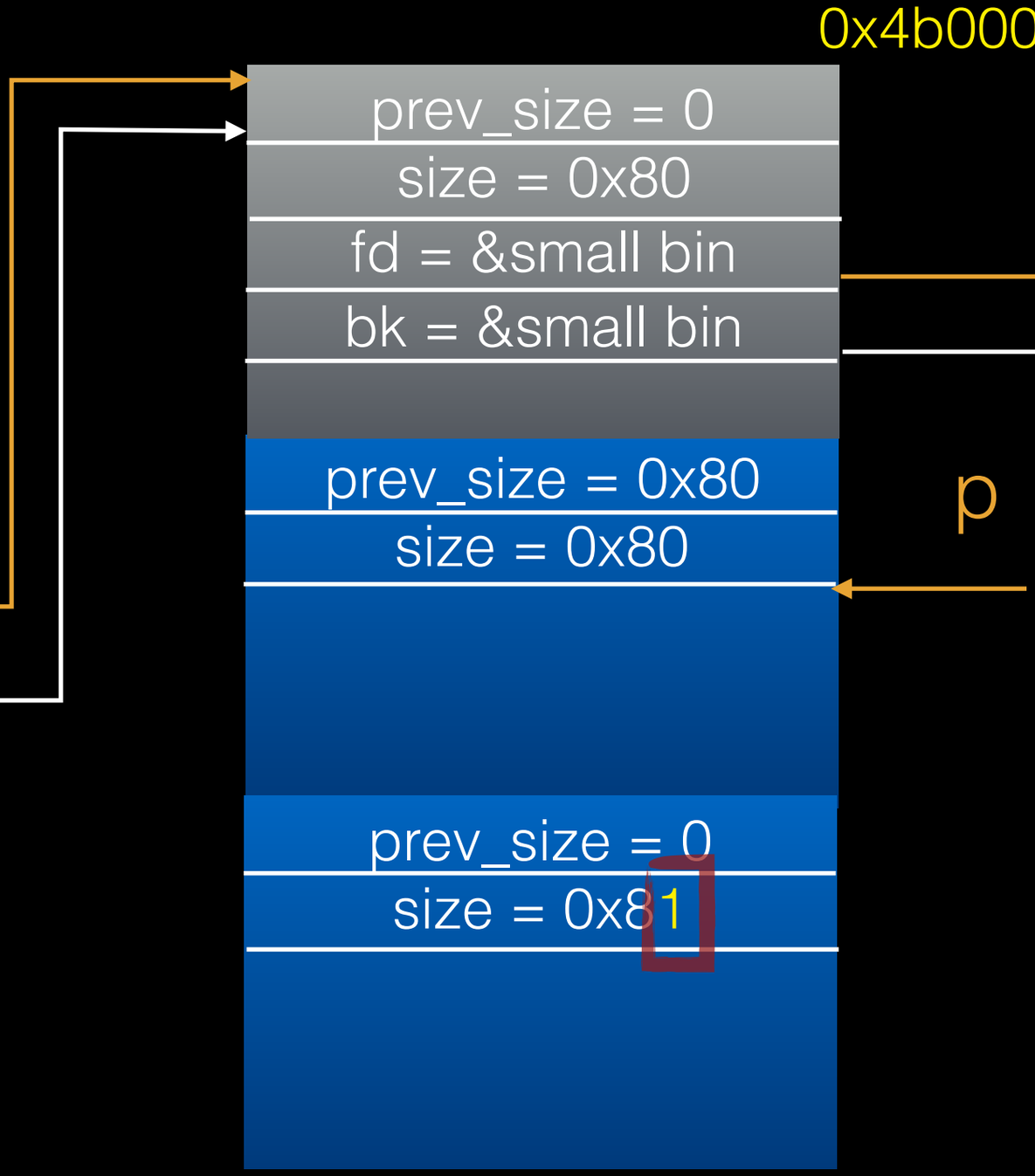
Mechanism of glibc malloc

利用 inuse bit
檢查是否有被 free 過

small bin

unsorted bin





Mechanism of glibc malloc

檢查上一塊
是否為 freed

small bin

0x4b000

0x4b000

unsorted bin

&unsortbin

&unsortbin

0x4b000

prev_size = 0

size = 0x80

fd = &small bin

bk = &small bin

prev_size = 0x80

size = 0x80

p

prev_size = 0

size = 0x81

Mechanism of glibc malloc

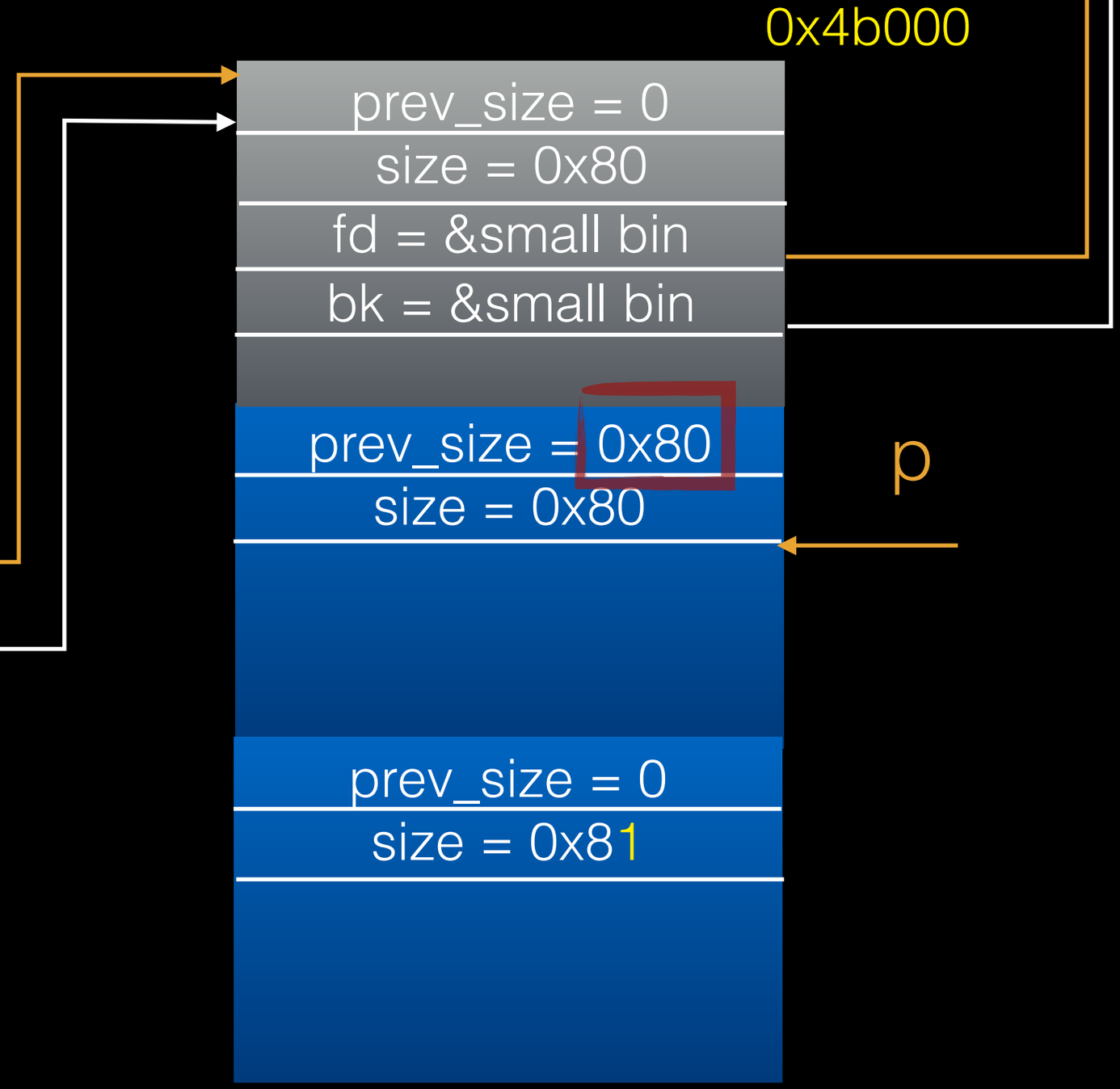
利用 prev_size
找到上一塊 chunk
p - 8 - 0x80

small bin

0x4b000
0x4b000

unsorted bin

&unsortbin
&unsortbin



Mechanism of glibc malloc

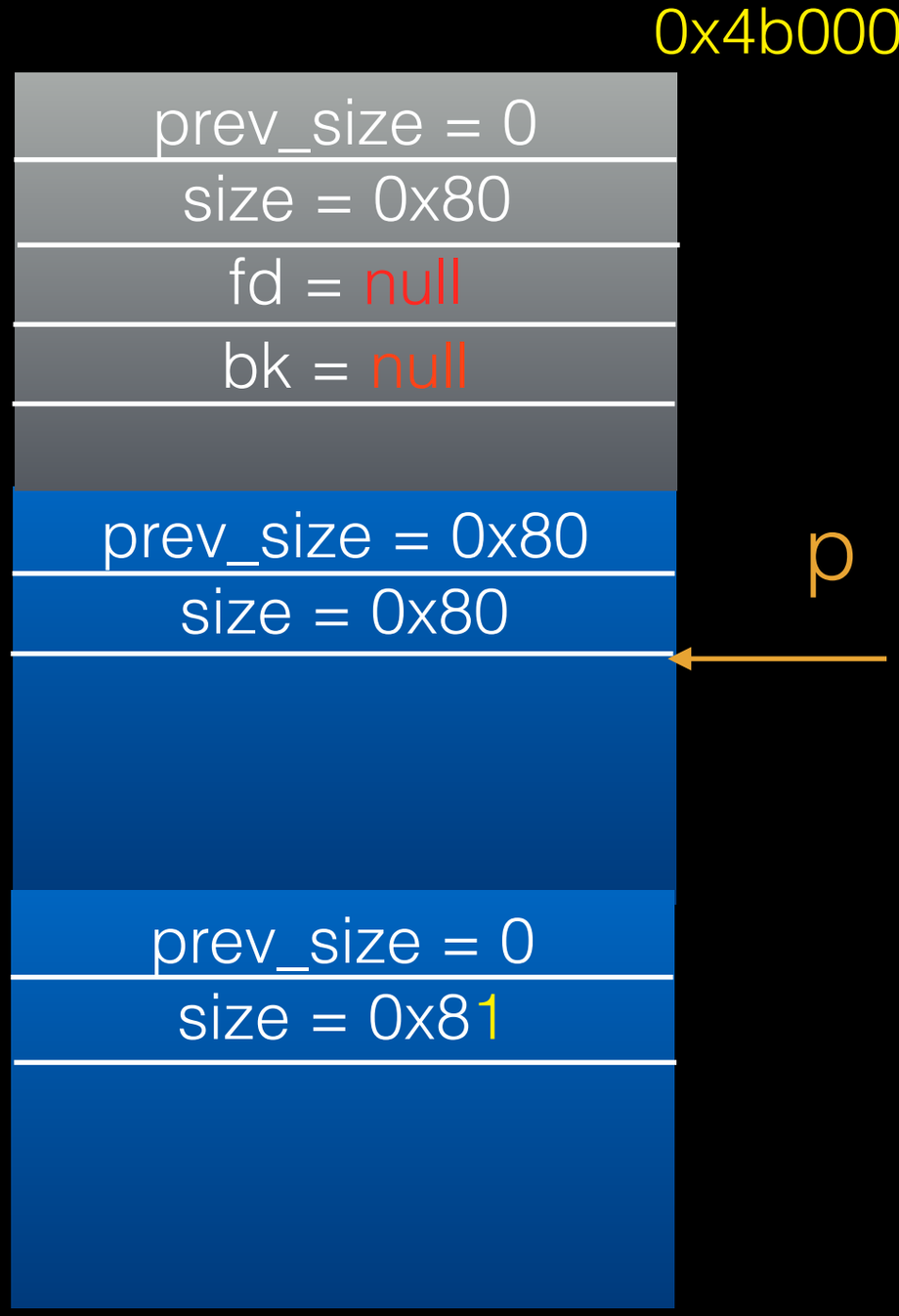
進行 unlink
將上一塊 chunk
從 bin 中移除

small bin

&smallbin
&smallbin

unsorted bin

&unsortbin
&unsortbin



Mechanism of glibc malloc

merge

small bin

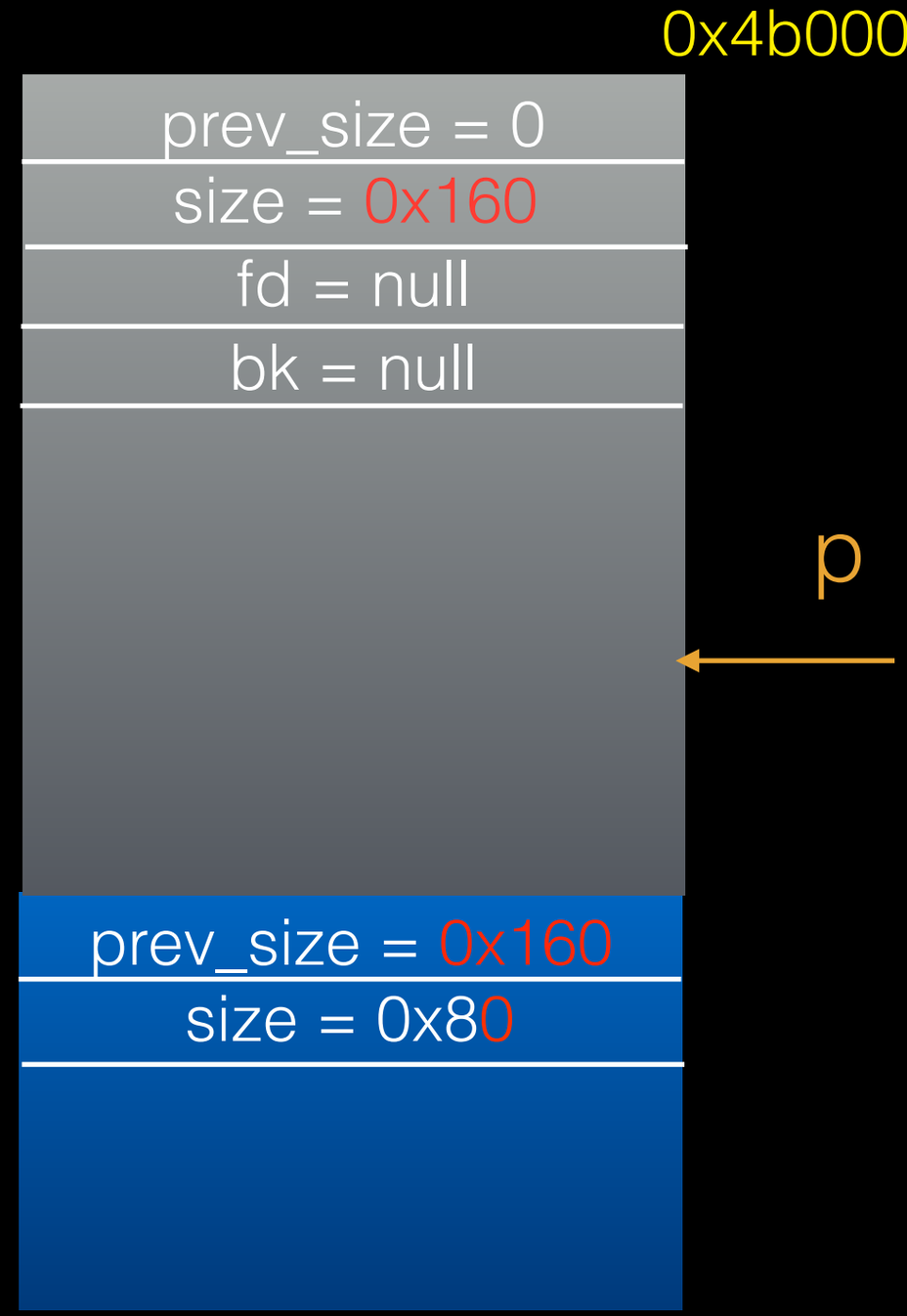
&smallbin

&smallbin

unsorted bin

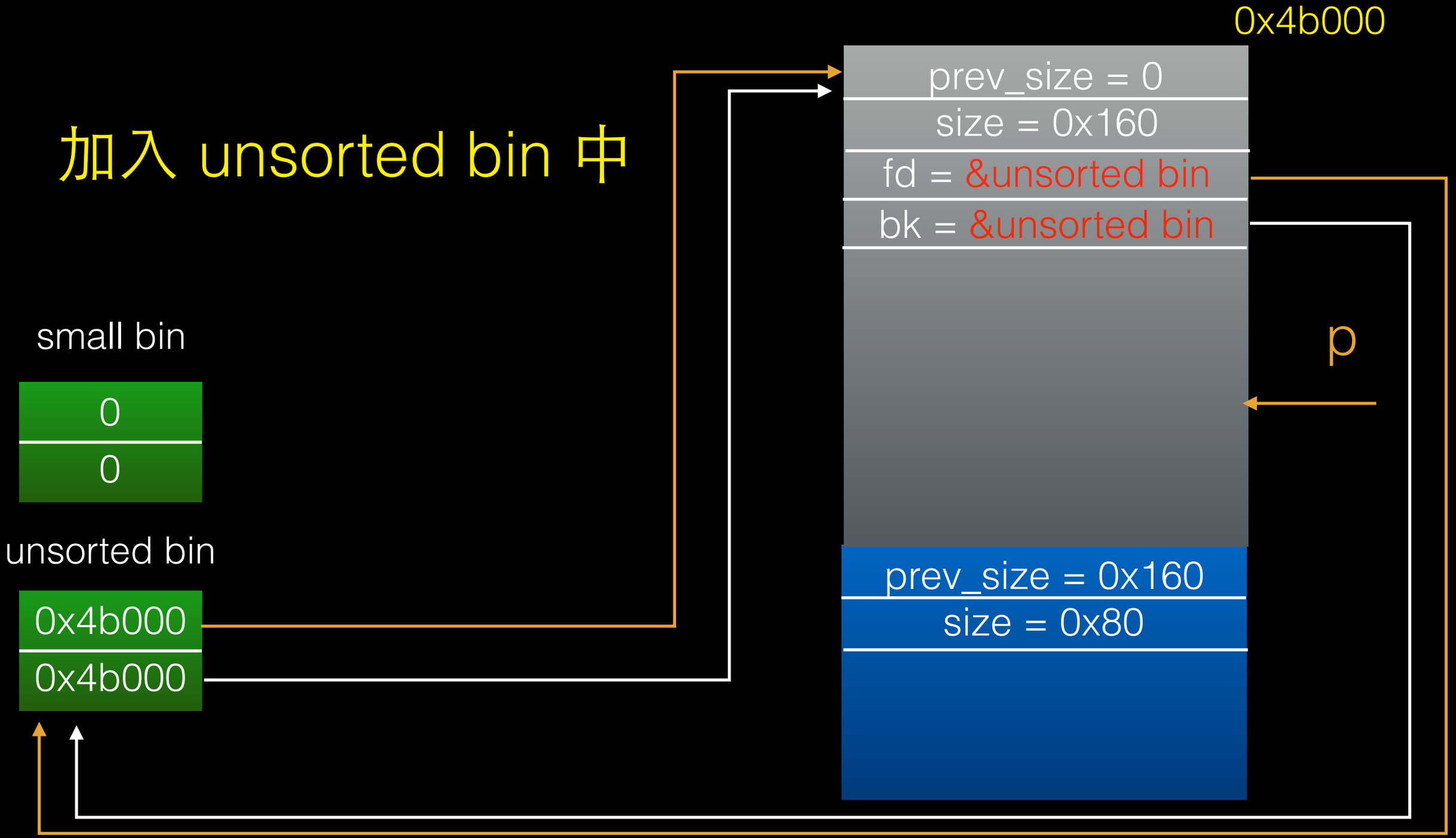
&unsortbin

&unsortbin



Mechanism of glibc malloc

加入 unsorted bin 中



Use after free

- 當 free 完之後，並未將 pointer 設成 null 而繼續使用該 pointer 該 pointer 稱為 dangling pointer
- 根據 use 方式不同而有不同的行為，可能造成任意位置讀取或是任意位置寫入，進一步造成控制程式流程

Use after free

- Assume exist a dangling ptr
  - p is a pointer of movement
  - free(p)

```
struct stu
{
    int stu_id ;
    name[20]
}
```

```
struct movement
{
    void (*playctf)();
    void (*playball)();
    void (*playgame)();
    char note[12]
}
```

Use after free

- Assume exist a dangling ptr
  - p is a pointer of movement
  - free(p) // p is dangling ptr
  - q = (*struct stu)malloc(sizeof(stu))
  - 此時因為 fast bin 的關係使 p == q

```
struct stu
{
    int stu_id ;
    name[20]
}
```

```
struct movement
{
    void (*playctf)();
    void (*playball)();
    void (*playgame)();
    char note[12]
}
```

Use after free

- Assume exist a dangling ptr
  - p is a pointer of movement
  - free(p) // p is dangling ptr
  - q = (*struct stu)malloc(sizeof(stu))
  - 此時因為 fast bin 的關係使 p == q
  - set id = 0x61616161

```
struct stu
{
    int stu_id ;
    name[20]
}
```

```
struct movement
{
    void (*playctf)();
    void (*playball)();
    void (*playgame)();
    char note[12]
}
```

Use after free

- Assume exist a dangling ptr
  - p is a pointer of movement
  - free(p) // p is dangling ptr
  - q = (*struct stu)malloc(sizeof(stu))
  - 此時因為 fast bin 的關係使 p == q
  - set id = 0x61616161
  - p.playctf()

```
struct stu
{
    int stu_id ;
    name[20]
}
```

```
struct movement
{
    void (*playctf)();
    void (*playball)();
    void (*playgame)();
    char note[12]
}
```

Use after free

- Assume exist a dangling ptr
  - p is a pointer of movement
  - free(p) // p is dangling ptr
  - q = (*struct stu)malloc(sizeof(stu))
  - 此時因為 fast bin 的關係使 p == q
  - set id = 0x61616161
  - p.playctf() ⟶ eip = 0x61616161

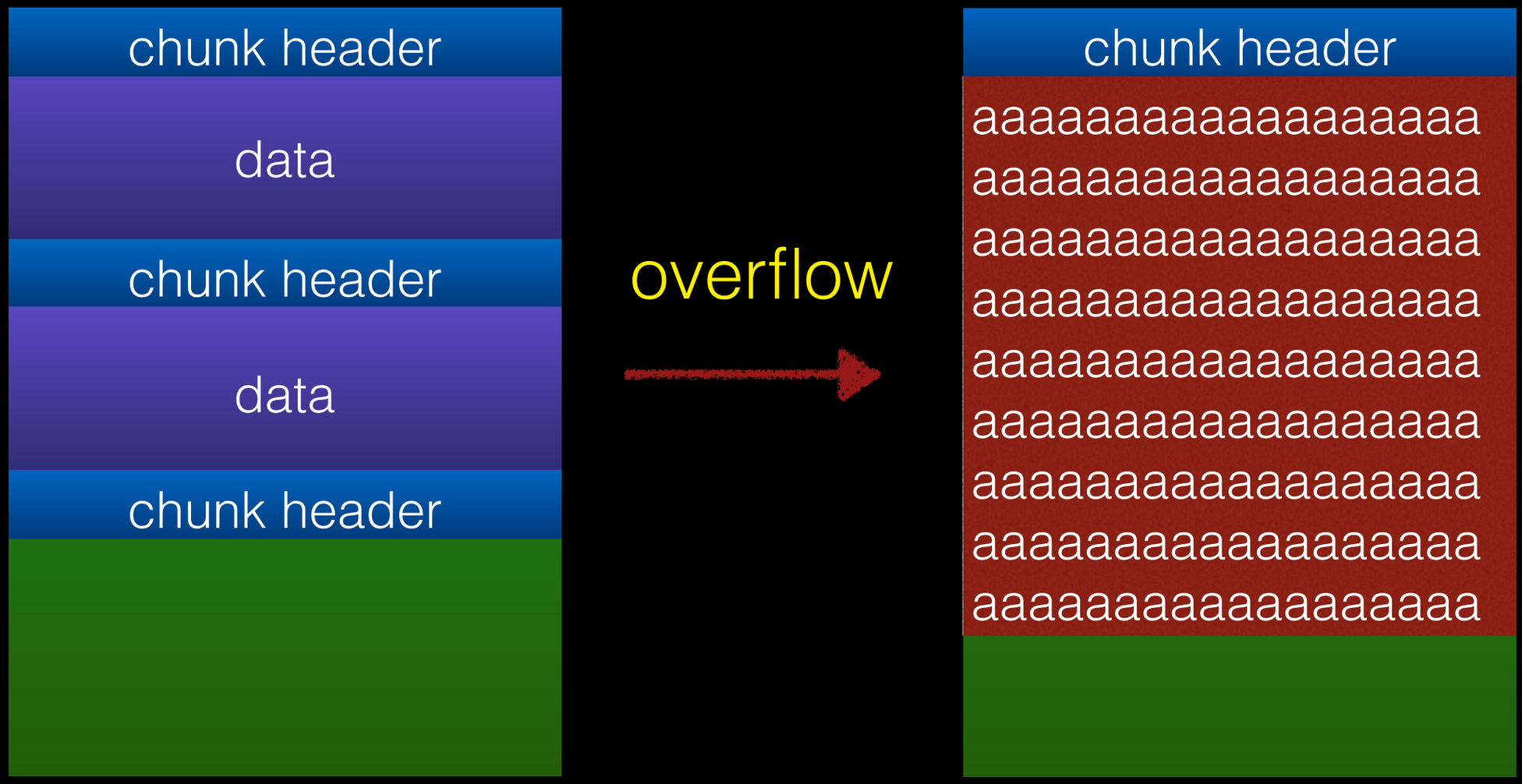
```
struct stu
{
    int stu_id ;
    name[20]
}
```

```
struct movement
{
    void (*playctf)();
    void (*playball)();
    void (*playgame)();
    char note[12]
}
```

Heap overflow

- 在 heap 段中發生的 buffer overflow
- 通常無法直接控制 eip 但可以利用蓋下一個 chunk header ，並利用 unlink 時的行為來間接達成任意位置寫入，進而控制 eip

Heap overflow



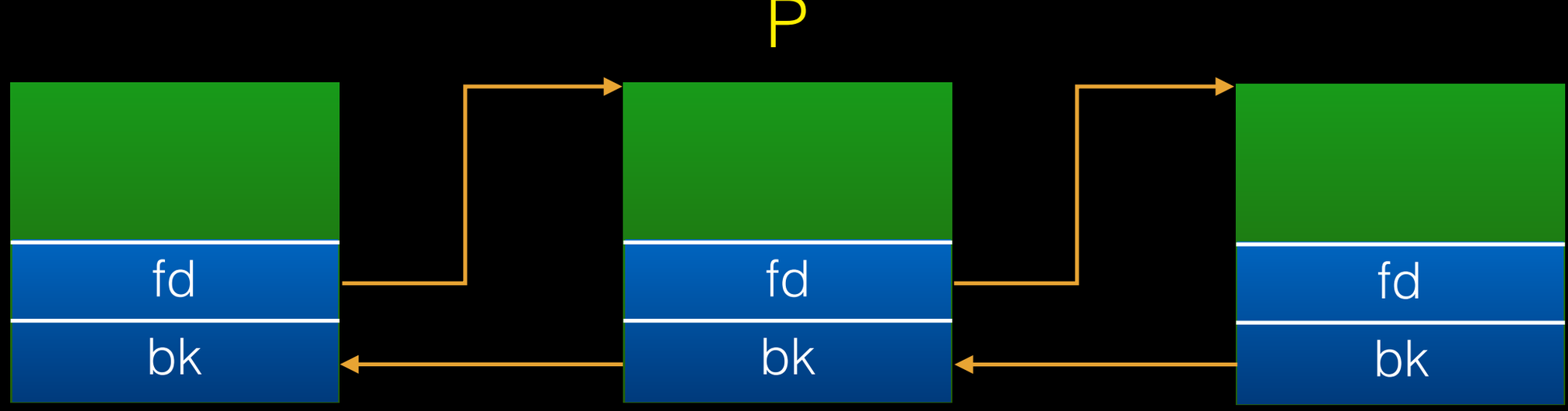
Heap overflow

- using unlink
  - 透過 overflow 蓋掉 freed chunk 中的， fd 及 bk ，再利用 unlink 中 FD->bk = BK 及 BK->fd = FD 來更改任意記憶體位置

```
Unlink(P, BK, FD) {
    FD = P->fd ;
    BK = P->bk ;
    FD->bk = BK ;
    BK->fd = FD ;
}
```

Heap overflow

- 複習一下 doubly linked list 中， delete 一個 node 時的過程

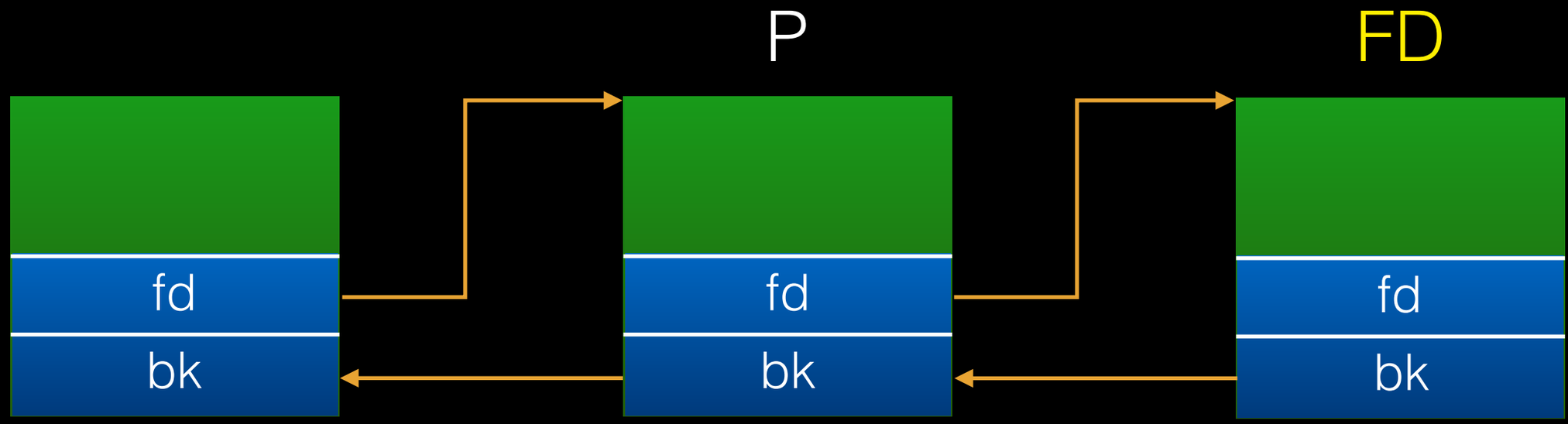


unlink(P,BK,FD)

```
Unlink(P, BK, FD) {
    FD = P->fd ;
    BK = P->bk ;
    FD->bk = BK ;
    BK->fd = FD ;
}
```

Heap overflow

- 複習一下 doubly linked list 中， delete 一個 node 時的過程

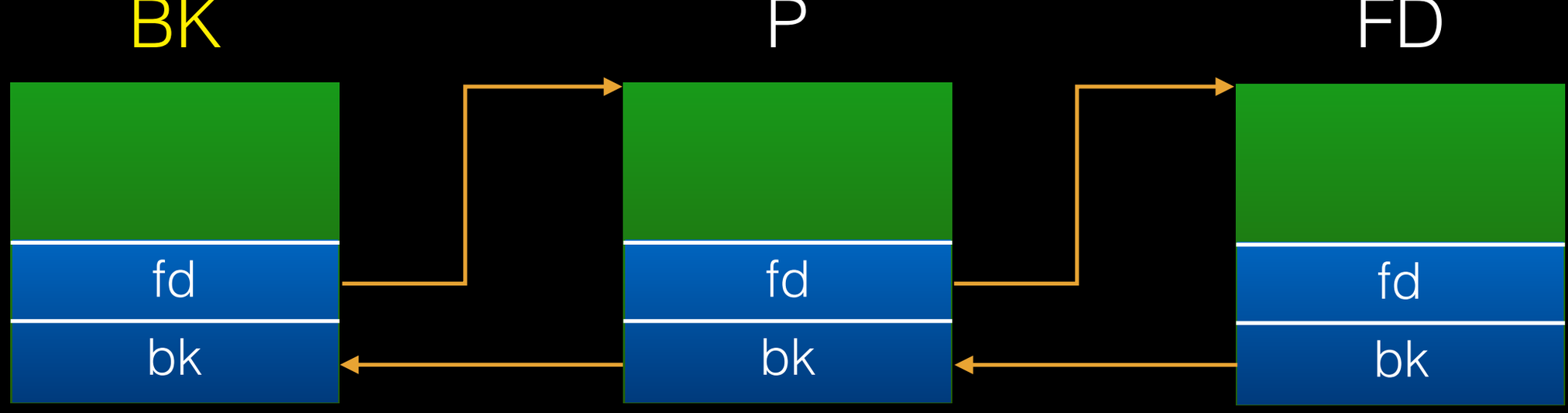


FD = P->fd

```
Unlink(P, BK, FD) {
    FD = P->fd ;
    BK = P->bk ;
    FD->bk = BK ;
    BK->fd = FD ;
}
```

Heap overflow

- 複習一下 doubly linked list 中， delete 一個 node 時的過程

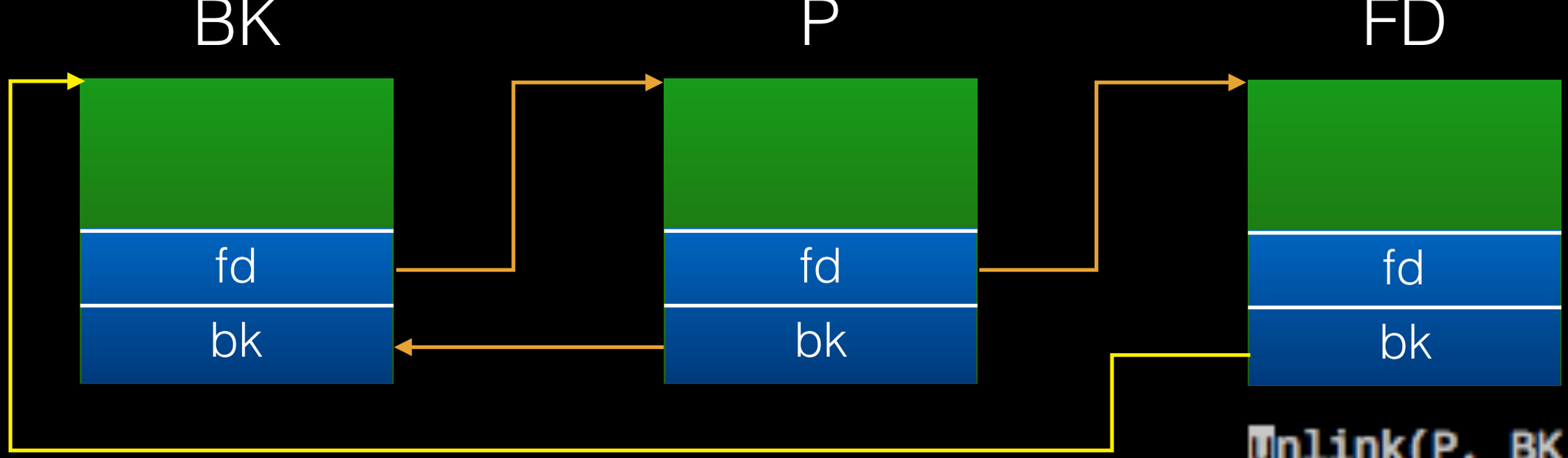


BK = P->bk

```
Unlink(P, BK, FD) {
    FD = P->fd ;
    BK = P->bk ;
    FD->bk = BK ;
    BK->fd = FD ;
}
```

Heap overflow

- 複習一下 doubly linked list 中， delete 一個 node 時的過程

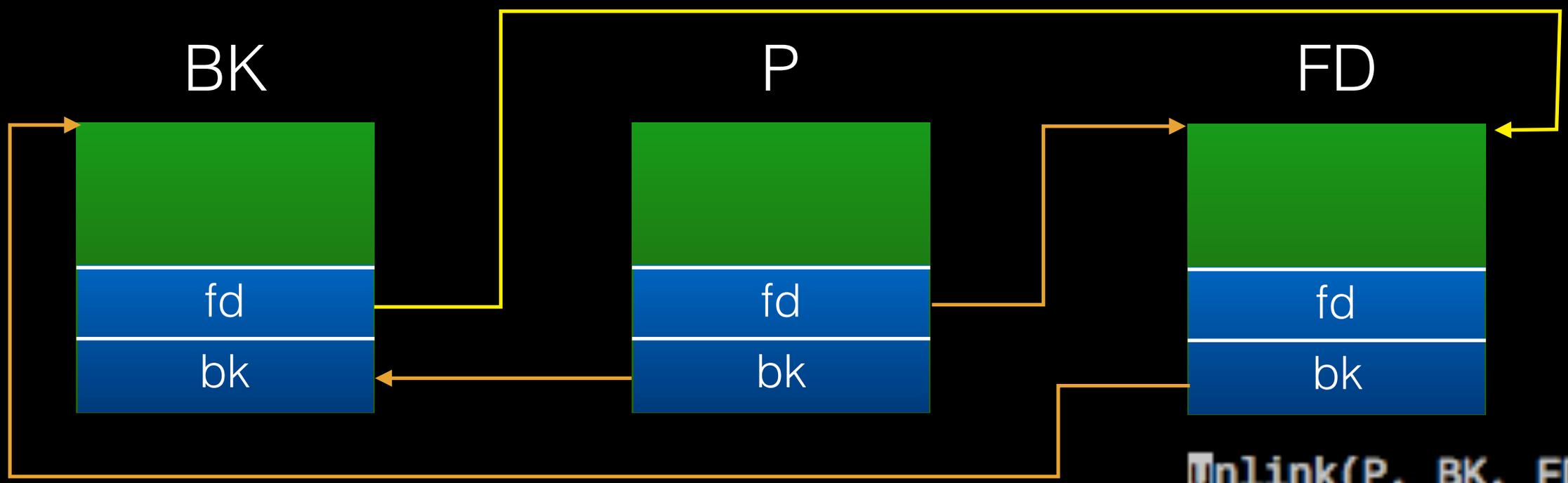



```
Unlink(P, BK, FD) {
    FD = P->fd ;
    BK = P->bk ;
    FD->bk = BK ;
    BK->fd = FD ;
}
```

FD->bk = BK

Heap overflow

- 複習一下 doubly linked list 中， delete 一個 node 時的過程

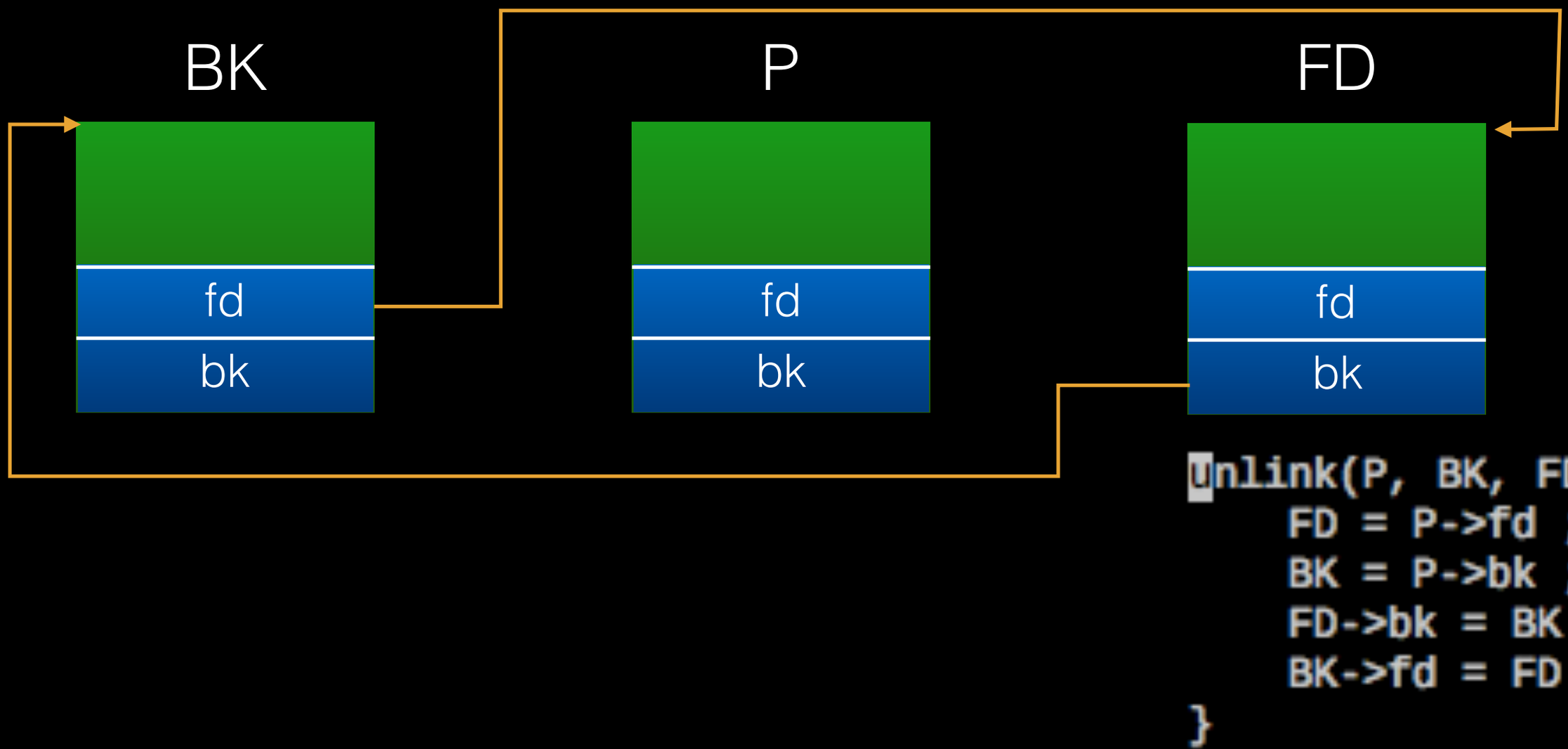



```
Unlink(P, BK, FD) {
    FD = P->fd ;
    BK = P->bk ;
    FD->bk = BK ;
    BK->fd = FD ;
}
```

BK->fd = FD

Heap overflow

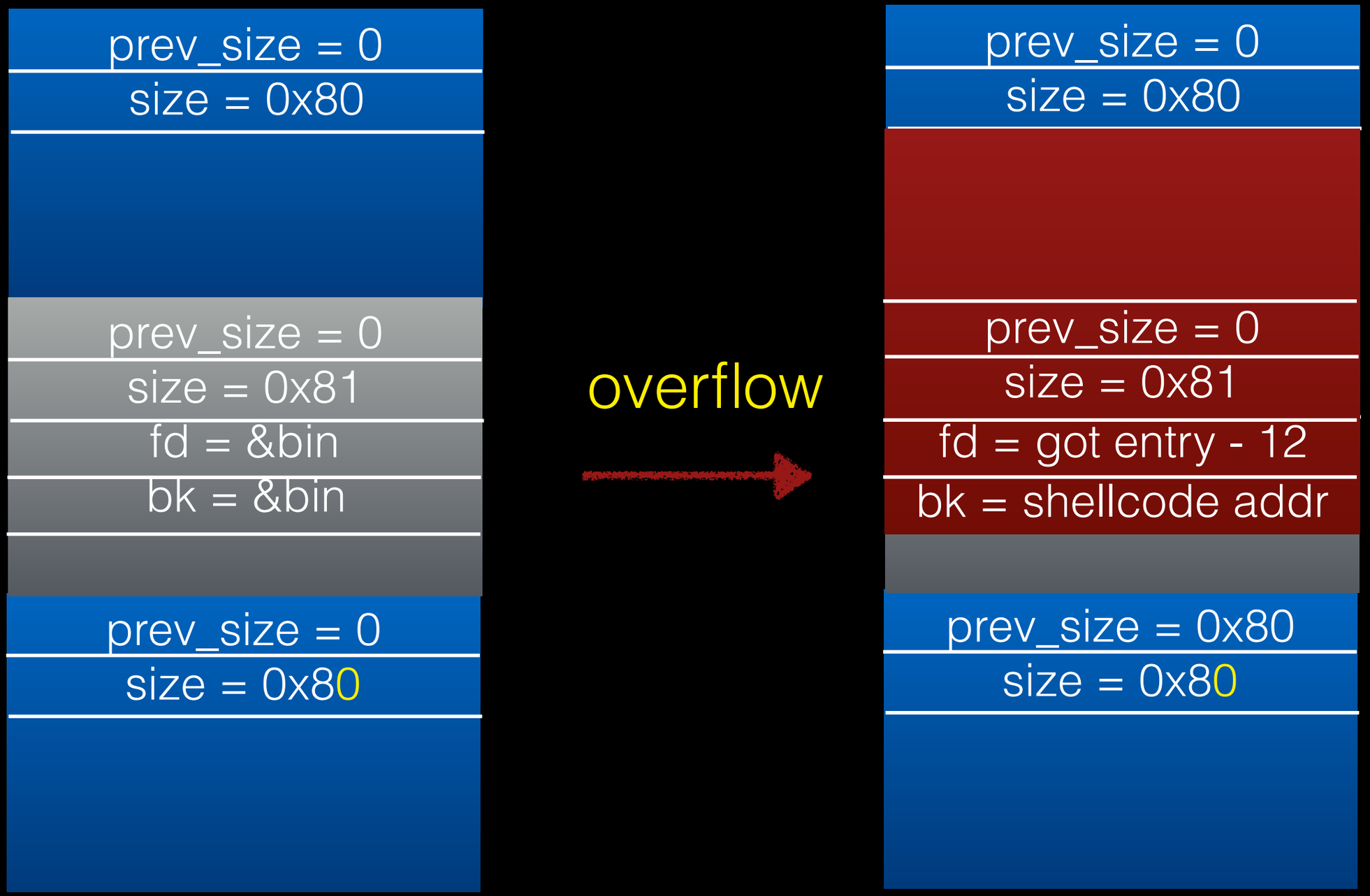
- 複習一下 doubly linked list 中， delete 一個 node 時的過程




```
Unlink(P, BK, FD) {
    FD = P->fd ;
    BK = P->bk ;
    FD->bk = BK ;
    BK->fd = FD ;
}
```

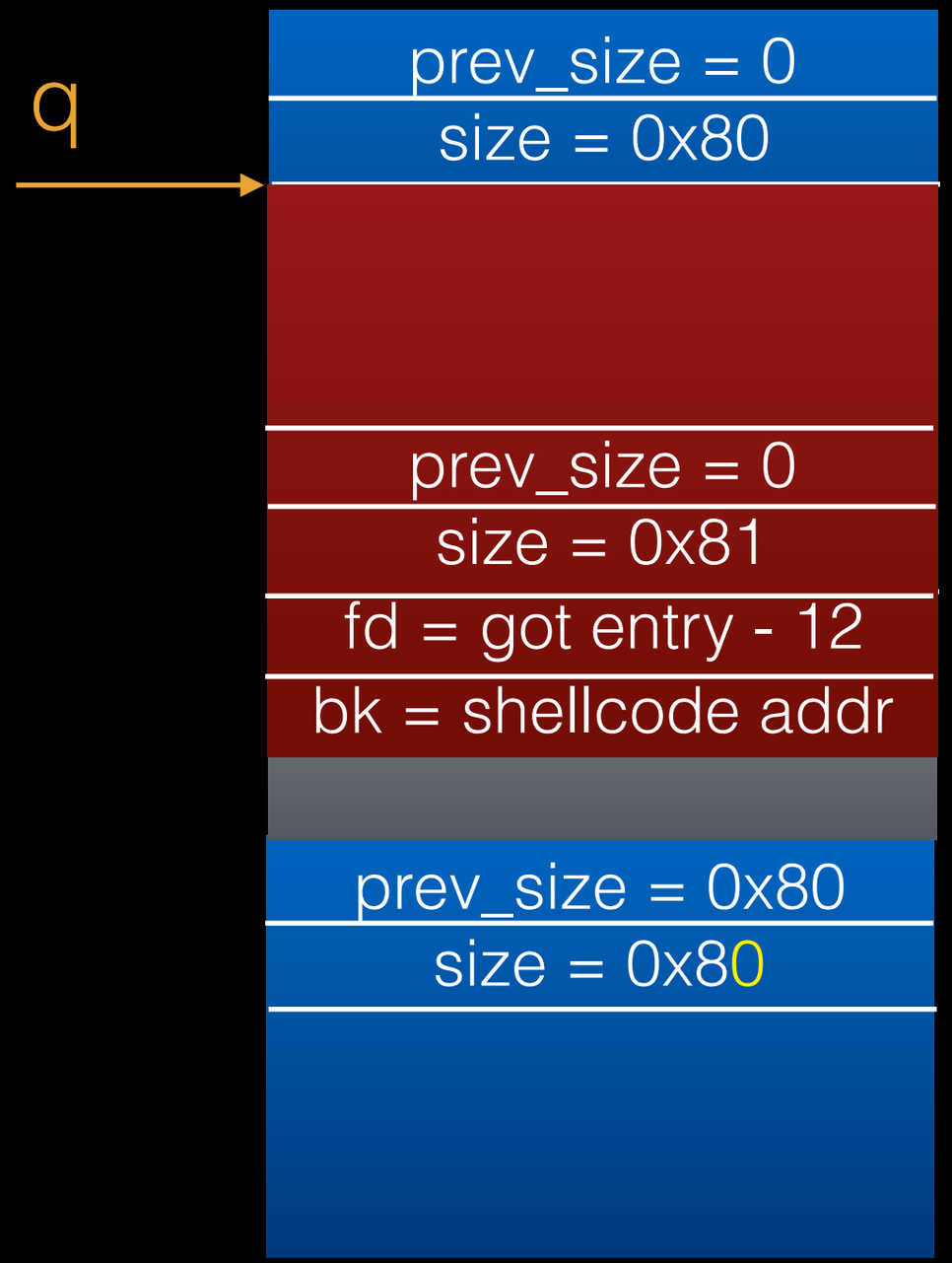
Heap overflow

- using unlink



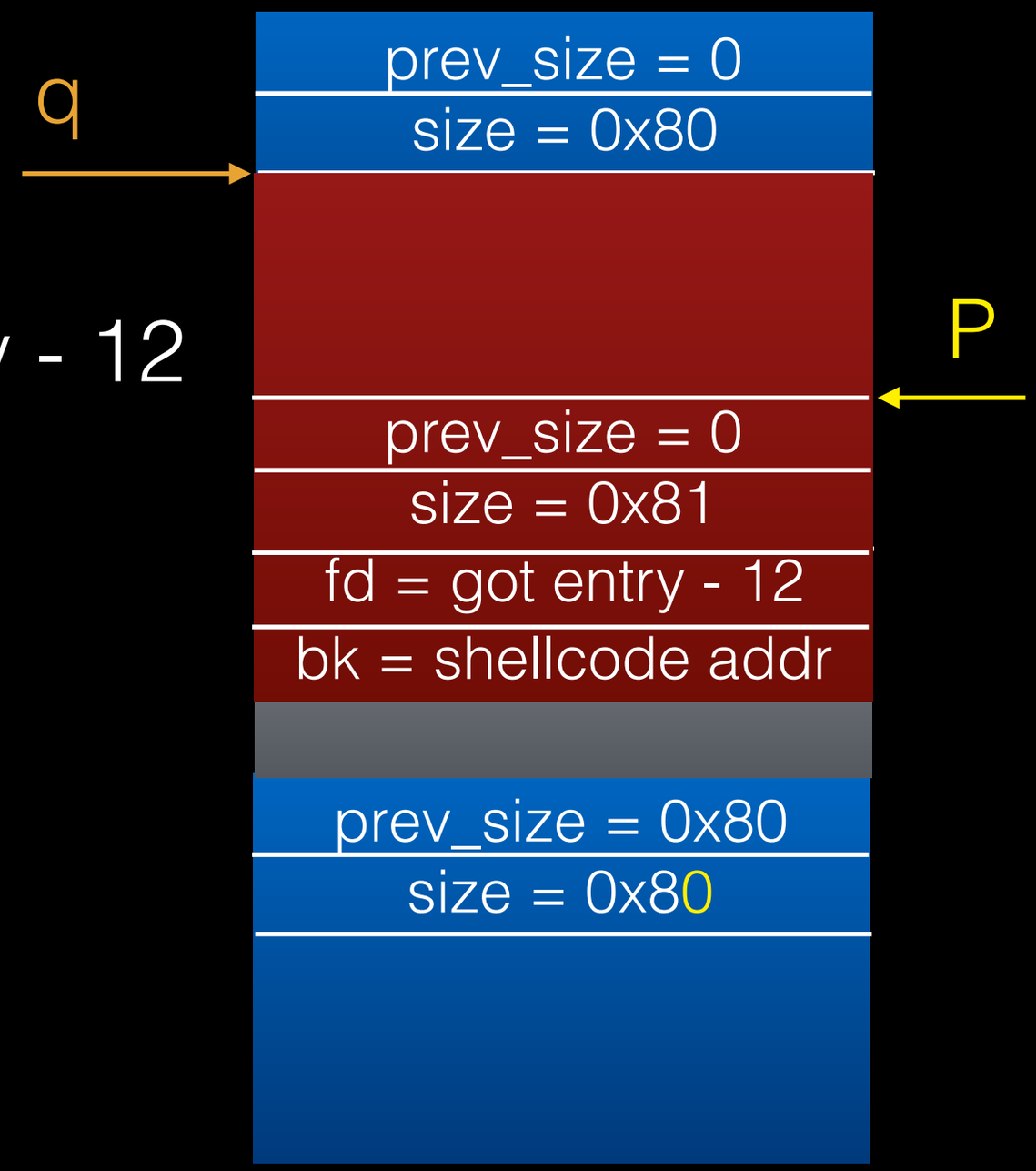
Heap overflow

- using unlink
  - free(q)



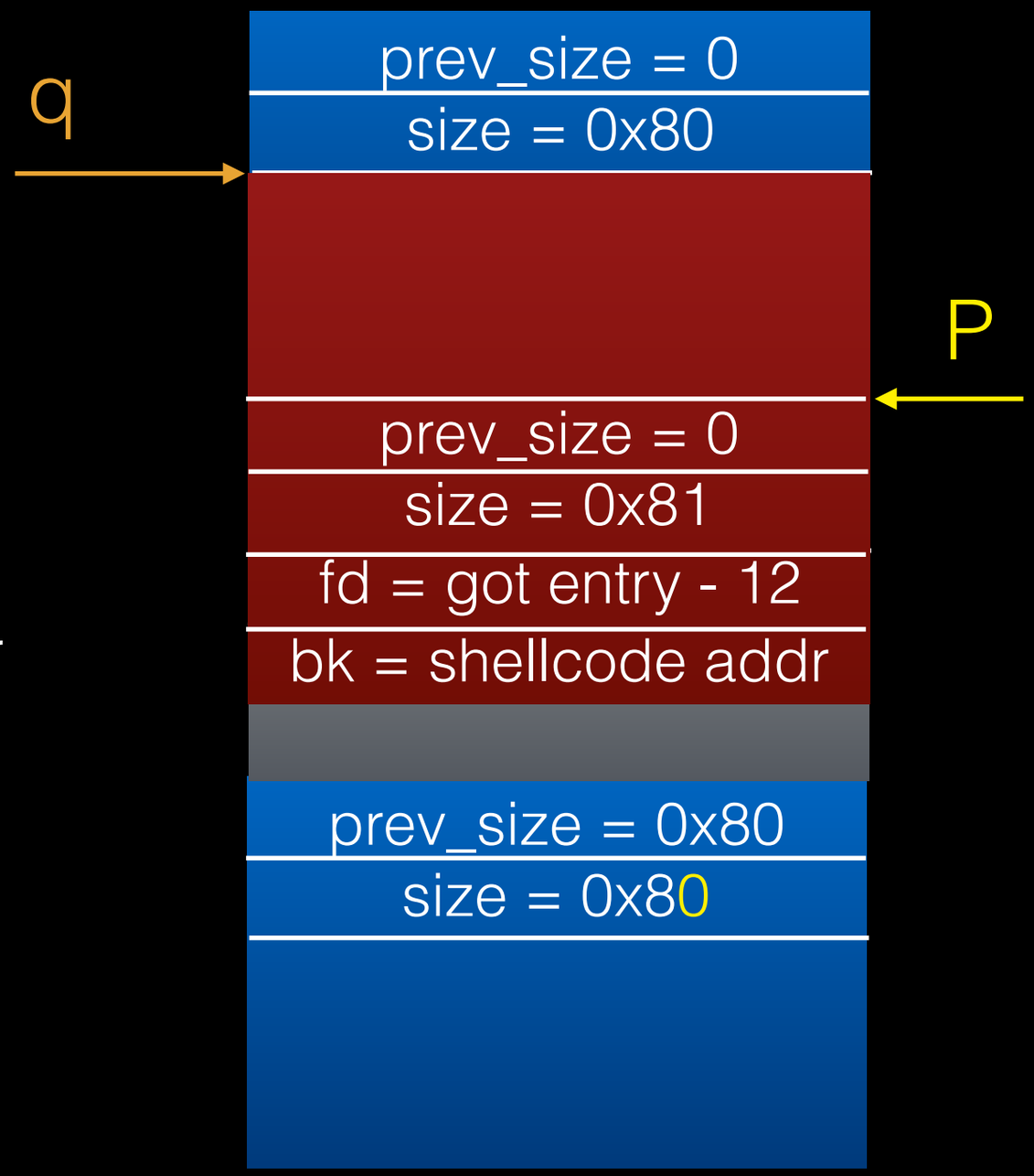
Heap overflow

- using unlink
  - free(q)
  - FD = P->fd = got entry - 12
  - BK = P->bk = sc addr



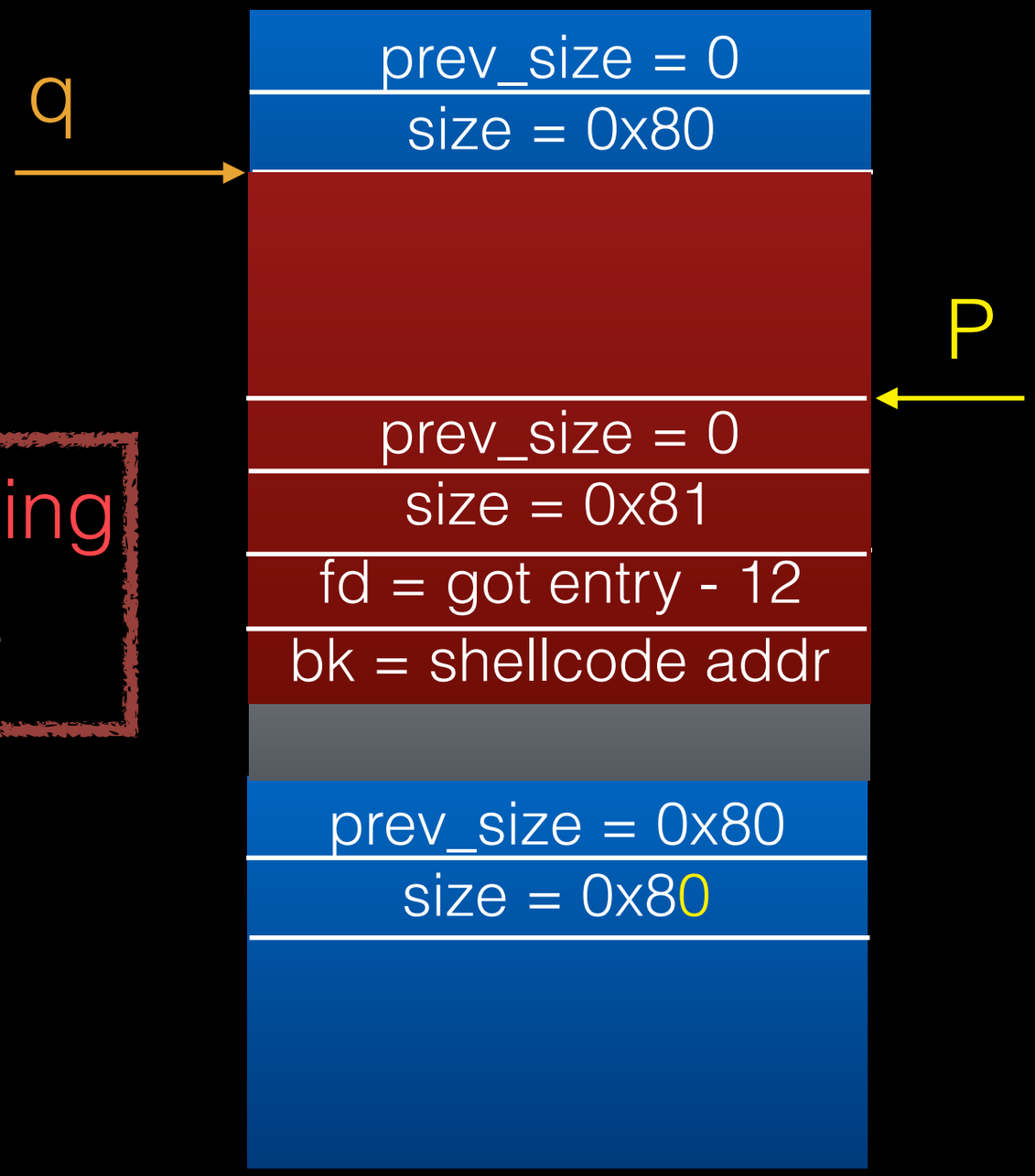
Heap overflow

- using unlink
  - free(q)
  - FD = P->fd = got entry - 12
  - BK = P->bk = sc addr
  - FD->bk = BK
    - got entry - 12 + 12 = sc addr
  - BK->fd = FD
    - sc addr + 8 = got entry - 12



Heap overflow

- using unlink
  - free(q)
  - FD = P->fd = got entry - 12
  - BK = P->bk = sc addr
  - FD->bk = BK GOT hijacking
    - got entry - 12 + 12 = sc addr
  - BK->fd = FD
    - sc addr + 8 = got entry - 12

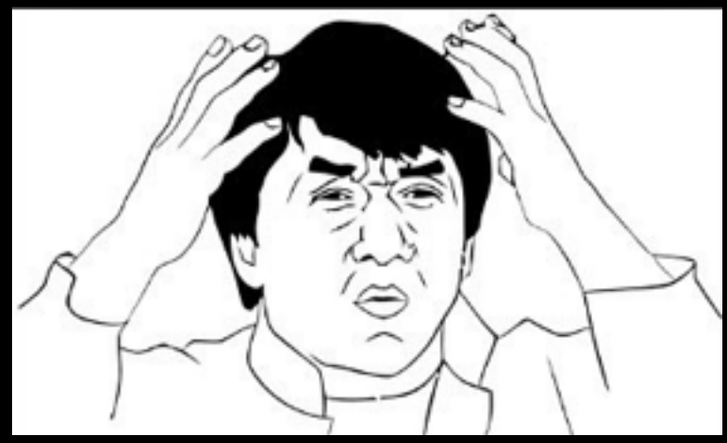


Heap overflow

- using unlink
  - 不過 shellcode 的第四到八個 byte 會因為 sc addr + 8 = got entry - 12 而被破壞
  - 需修正為 shellcode 改為 jmp 的方式跳到後面去執行
  - 當下次 call 到 got entry 時便會跳到 shellcode 去執行了

Heap overflow

- using unlink (modern)
  - 但現實是殘酷的，現代 glibc 中有各種針對 chunk 的檢查及其他保護機制 (DEP)，使得該方法無法使用
    - double free detect
    - Invalid next size
    - corrupted double linked list
    - ……



Heap overflow

- detection in malloc
  - size 是 fastbin 的情況
    - memory corruption (fast)
      - 從 fastbin 取出的第一塊 chunk 的 (unsigned long) size 不屬於該 fastbin 中的
      - 主要檢查方式是根據 malloc 的 bytes 大小取得 index 後，到對應的 fastbin 找，取出第一塊後後檢查該 chunk 的 (unsigned long) size 是否屬於該 fastbin
        - 但實際比較的時候是先以fastbin 中第一塊 size 取得fastbin 的 index ，再去比 index 跟剛剛算的 index 是否相同，不過這取 index 的方式是用 unsigned int (4 byte)

Heap overflow

- detection in malloc
  - size 是 smallbin 的情況
    - smallbin double linked list corrupted
      - 從相對應的 smallbin 中拿最後一個時，要符合 smallbin 是 double linked list
      - victim == smallbin 最後一塊 chunk
      - bck = victim->bk
      - bck->fd == victim

Heap overflow

- detection in malloc
  - unsortbin 中有 chunk
    - memory corruption
      - 取 unsortbin 的最後一塊 chunk 作為 victim
      - victim->size 要符合規定
        - size 必須大於 2 x SIZE_SZ (0x8)
        - size 必須小於 system_mem
          - system mem : heap 的大小通常為 132k

Heap overflow

- detection in malloc
  - size 是 largebin 的情況
- corrupted unsorted chunks
  - 在找到適合的 chunk 切給 user 後，剩下的空間會放到 last remainder，然後加到 unsortbin 中
  - 這時會取 unsortbin 的第一個的 fd 是否等於 unsortbin 的位置

Heap overflow

- detection in free
  - invalid pointer
    - 檢查 alignment
      - chunk address 是否為 8 的倍數
    - 檢查 chunk address 是否小於 - size

Heap overflow

- detection in free
  - invalid size
    - 檢查 chunk size 是否合法
      - size 必須為 8 的倍數 (不含最低 3 bit)
        - 也就是是否有符合 alignment
      - size 必須大於 MINSIZE (16 byte)

Heap overflow

- detection in free
  - size 是 fastbin 的情況
    - invalid next size (fast)
      - 檢查下一塊 chunk size 是否合法
        - size 必須大於 MINSIZE (0x10 byte)
        - size 必須小於 system_mem
          - system mem : heap 的大小通常為 132k

Heap overflow

- detection in free
  - size 是 fastbin 的情況
    - double free or corruption (fasttop)
      - 檢查 fastbin 中的第一塊 chunk 跟正要 free 的 chunk 是否不同
      - 要是相同就會 abort

Heap overflow

- detection in free
  - size 是 smallbin & largebin 的情況 (非 mmap)
    - double free or corruption (top)
      - 檢查正要 free 的 chunk 跟 top chunk 的位置是否不同
      - 要是相同就會 abort

Heap overflow

- detection in free
  - size 是 smallbin & largebin 的情況 (非 mmap)
    - double free or corruption (out)
      - 計算出來下一塊 chunk 的位置是否超出 heap 的邊界
      - 超出 heap 邊界就會 abort

Heap overflow

- detection in free
  - size 是 smallbin & largebin 的情況 (非 mmap)
    - double free or corruption (!prev)
      - 根據下一塊 chunk 的 inuse bit 來檢查正要 free 的 chunk 是否已被 free 過

Heap overflow

- detection in free
  - size 是 smallbin & largebin 的情況 (非 mmap)
    - invalid next size (normal)
      - 檢查下一塊 chunk size 是否合法
        - size 必須大於 2 x SIZE_SZ (0x8)
        - size 必須小於 system_mem
          - system mem : heap 的大小通常為 132k

Heap overflow

- detection in free
  - size 是 smallbin & largebin 的情況 (非 mmap)
    - corrupted unsorted chunks
      - 在 unlink 後要放到 unsortbin 時，會先從 unsortbin 取第一塊 chunk 出來，然後檢查該 chunk 的 bk 是否等於 unsortbin

Heap overflow

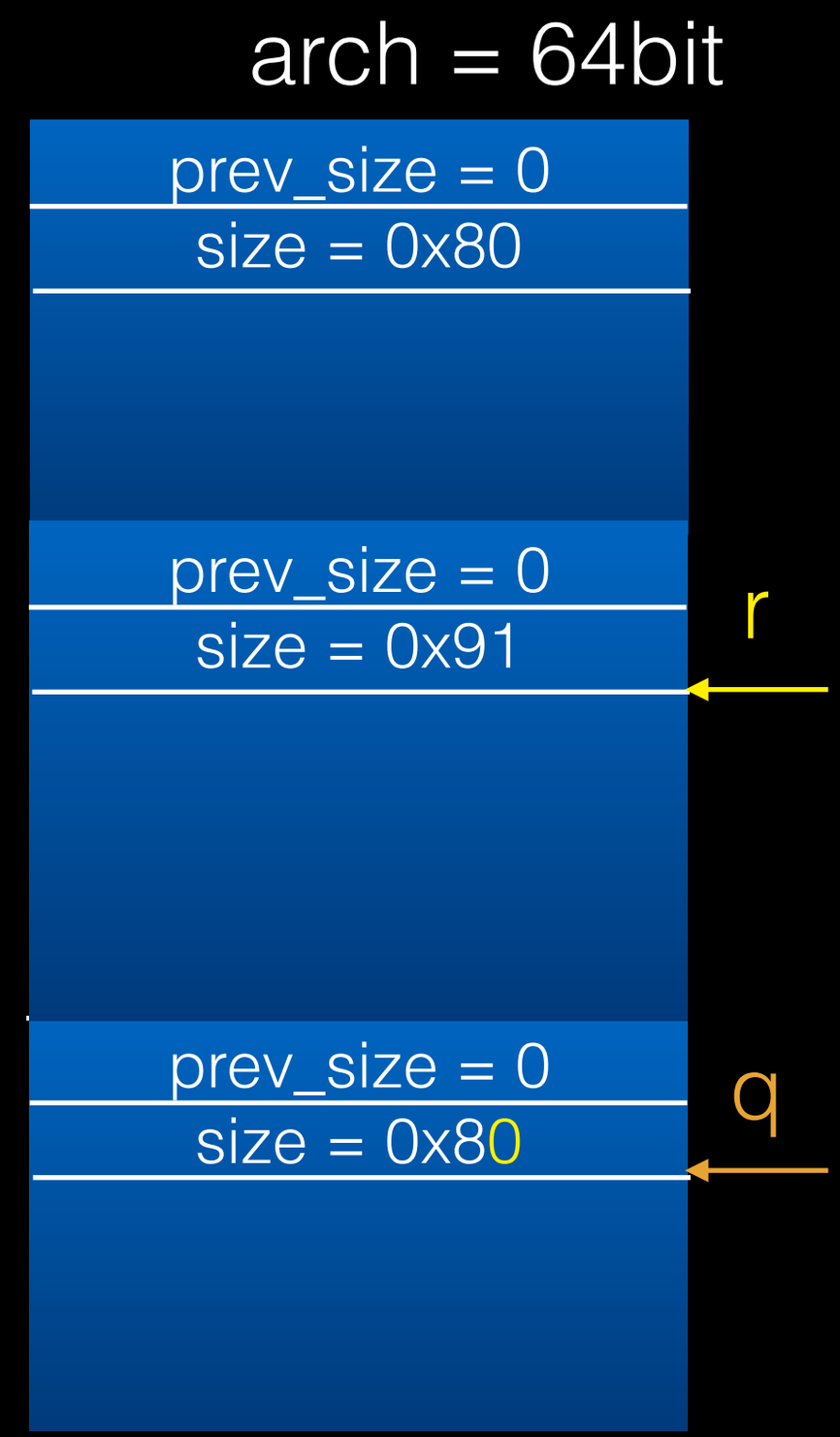
- detection in free
  - 在做 unlink 時
  - corrupted double linked list
    - 檢查 circular doubly linked list 的完整性，指出去在指回來必須指回自己，否則就會顯示 corrupted double-linked list 並中斷程式
      - P->bk->fd == P
      - P->fd->bk == P

Heap overflow

- using unlink (modern)
  - bypass the detection
    - 必須偽造 chunk 結構
    - 必須找到指向偽造 chunk 的 pointer 及該 pointer 的 address
      - 因此能直接改的地方有限，通常要間接去讀取或寫入任意位置

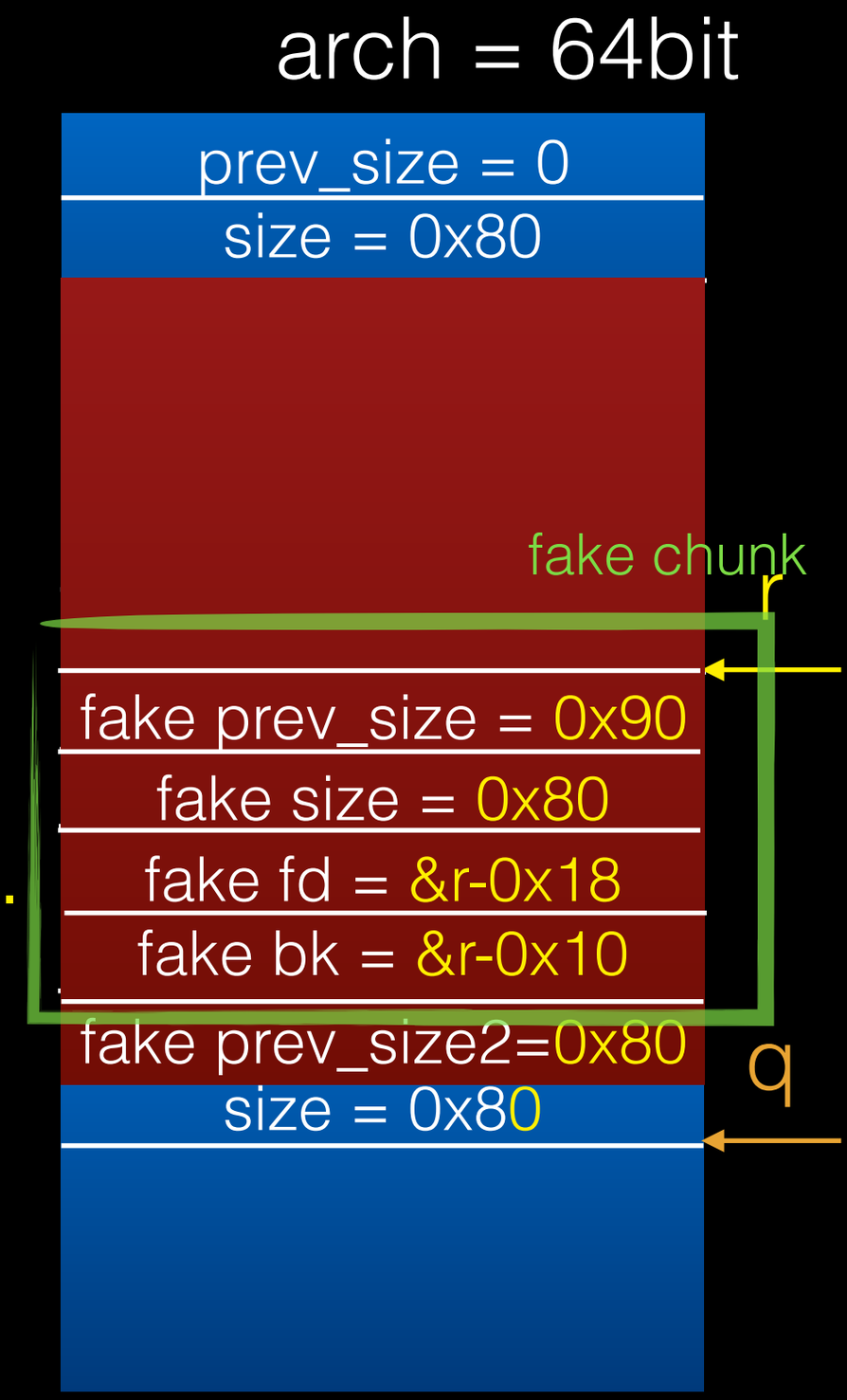
Heap overflow

- using unlink (modern)
  - bypass the detection
  - there’re a pointer r point to data of the second chunk.



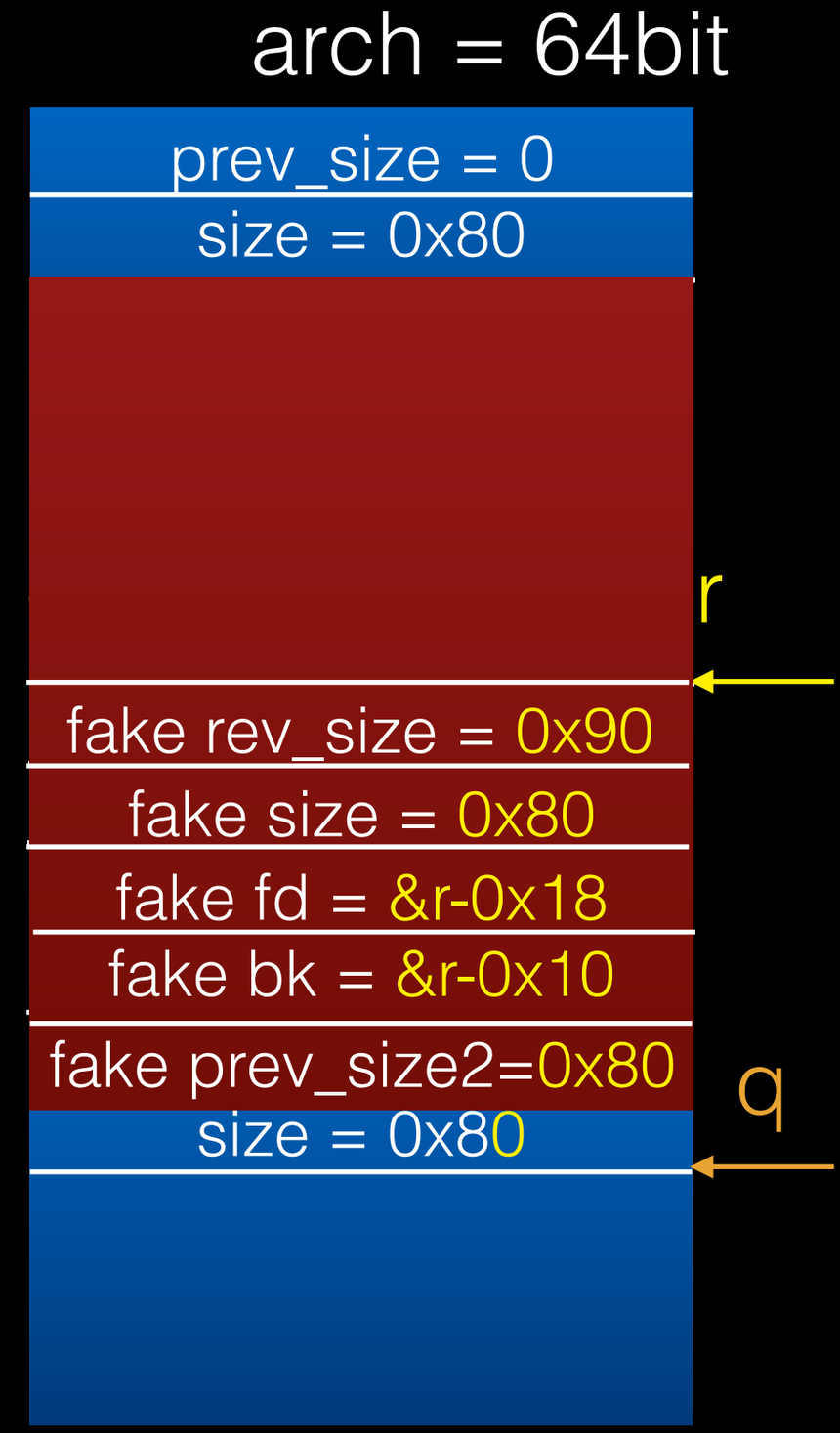
Heap overflow

- using unlink (modern)
  - bypass the detection
    - there’re a pointer r point to data of the second chunk.
  - overflow and forge chunks.



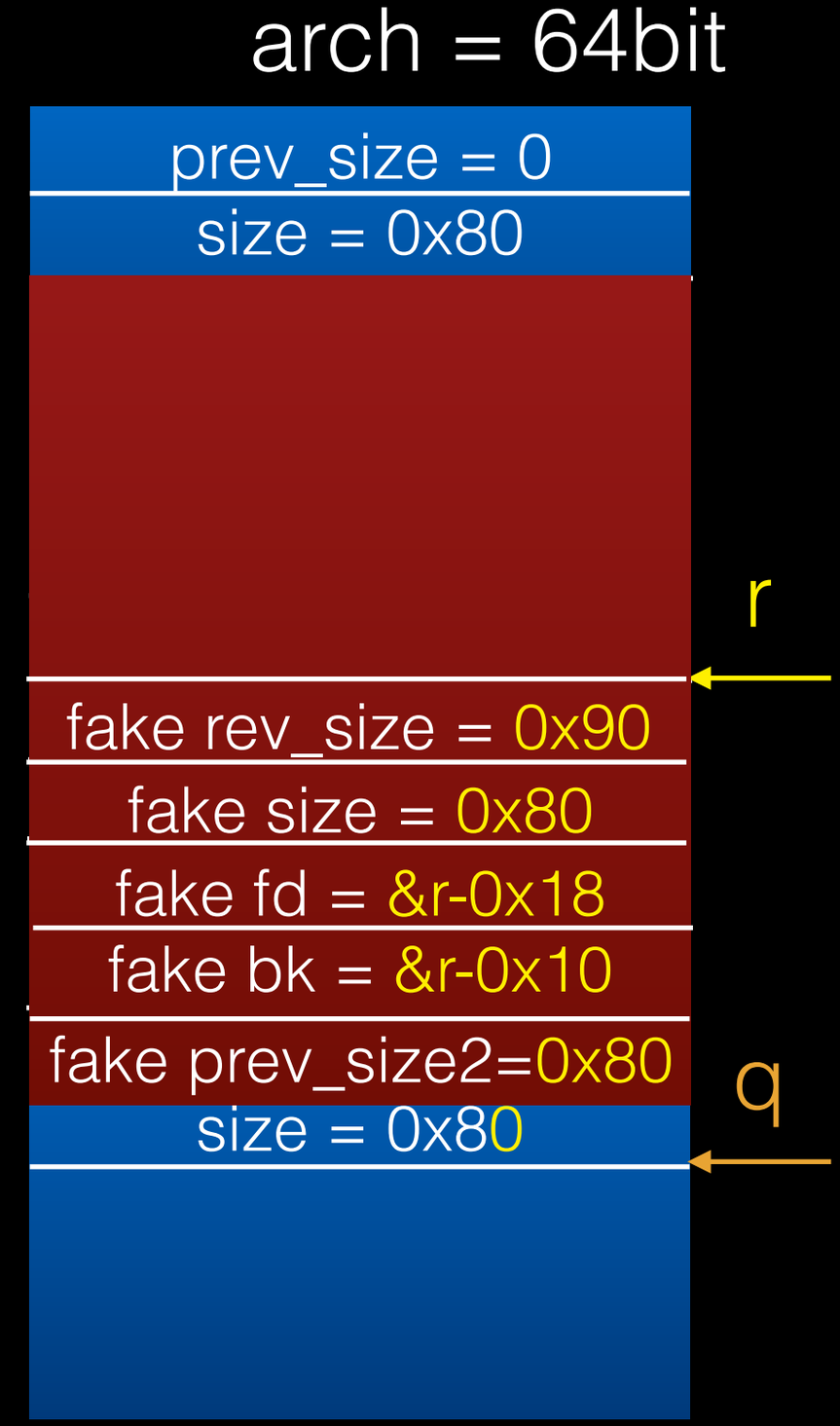
Heap overflow

- using unlink (modern)
  - bypass the detection
    - there’re a pointer r point to data of the second chunk.
  - overflow and forge chunks.
  - you can see the pointer r is point to the fake chunk



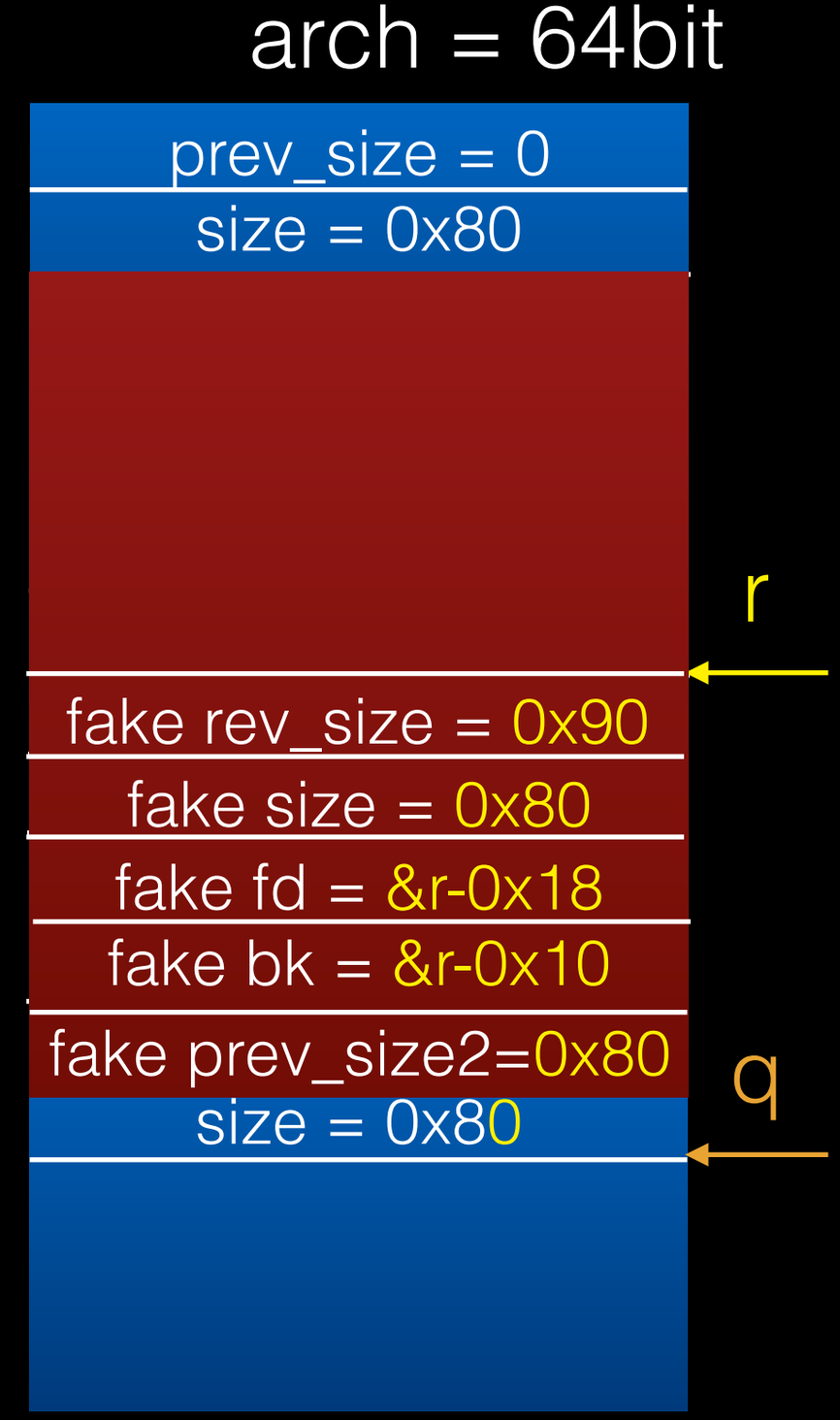
Heap overflow

- using unlink (modern)
  - bypass the detection
    - free(q)



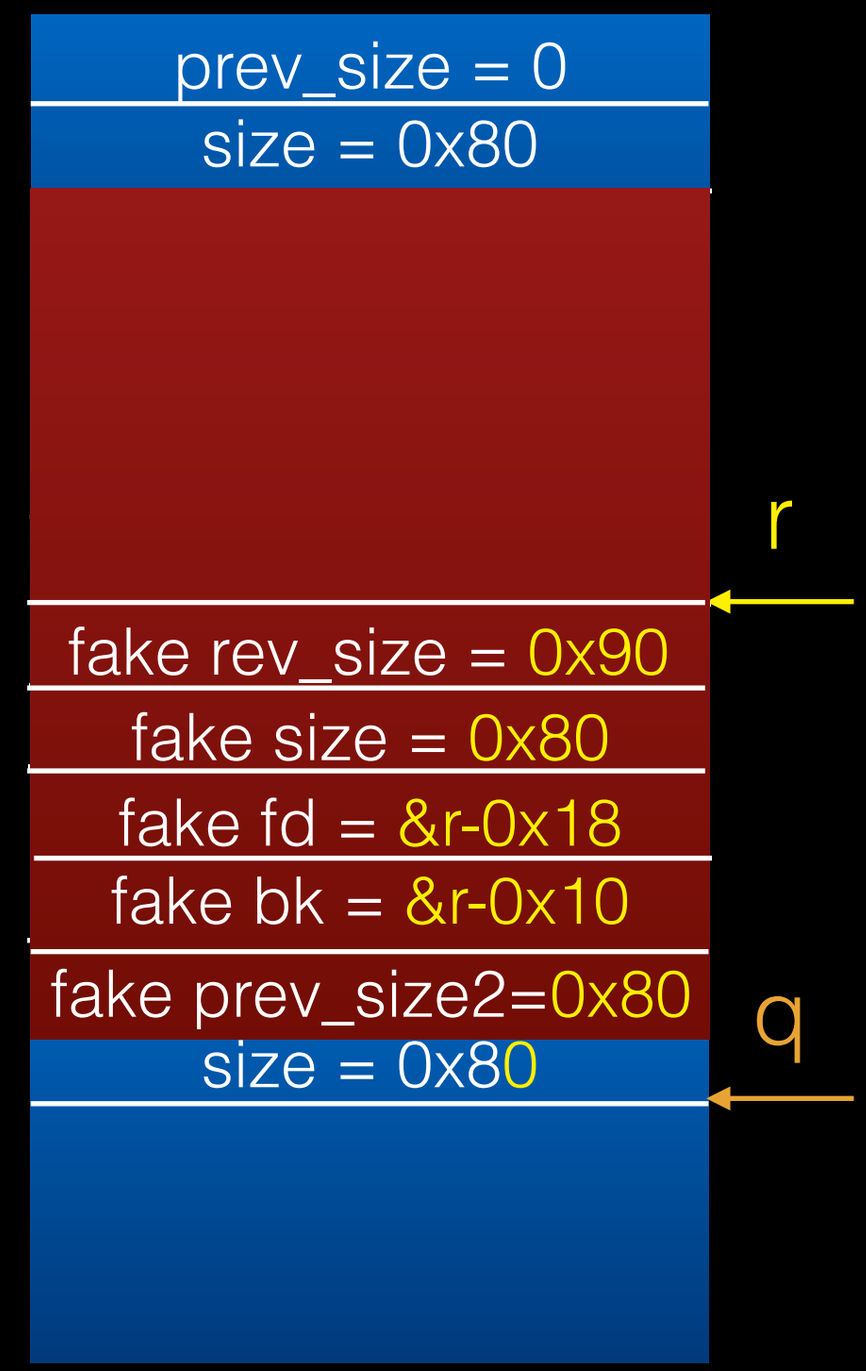
Heap overflow

- using unlink (modern)
  - bypass the detection
    - free(q)
  - check q & r is freed



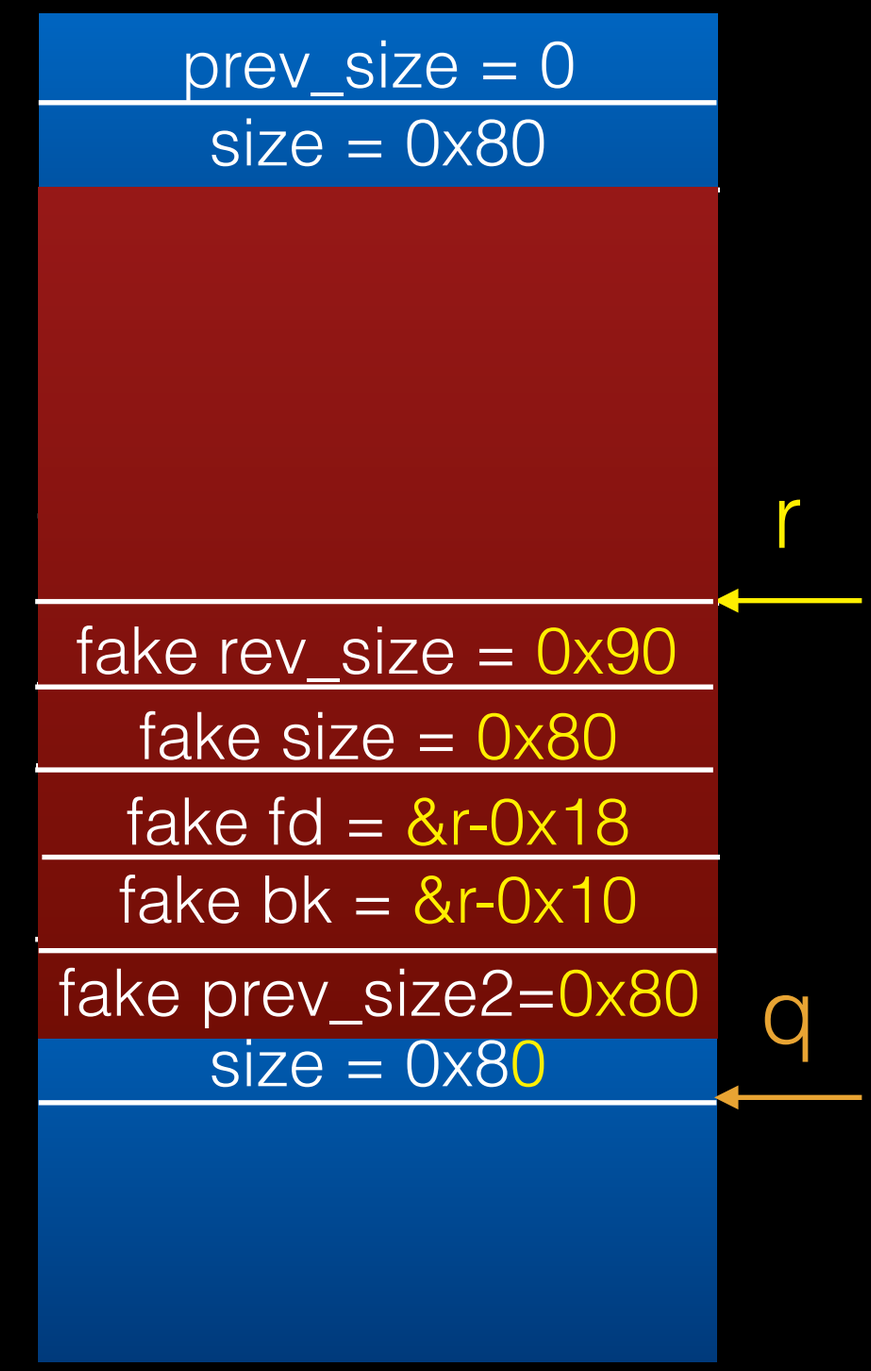
Heap overflow

- using unlink (modern)
  - bypass the detection
    - free(q)
  - check q & r is freed
  - find the last chunk of q
    - q - 0x10 - prev_size2 = r



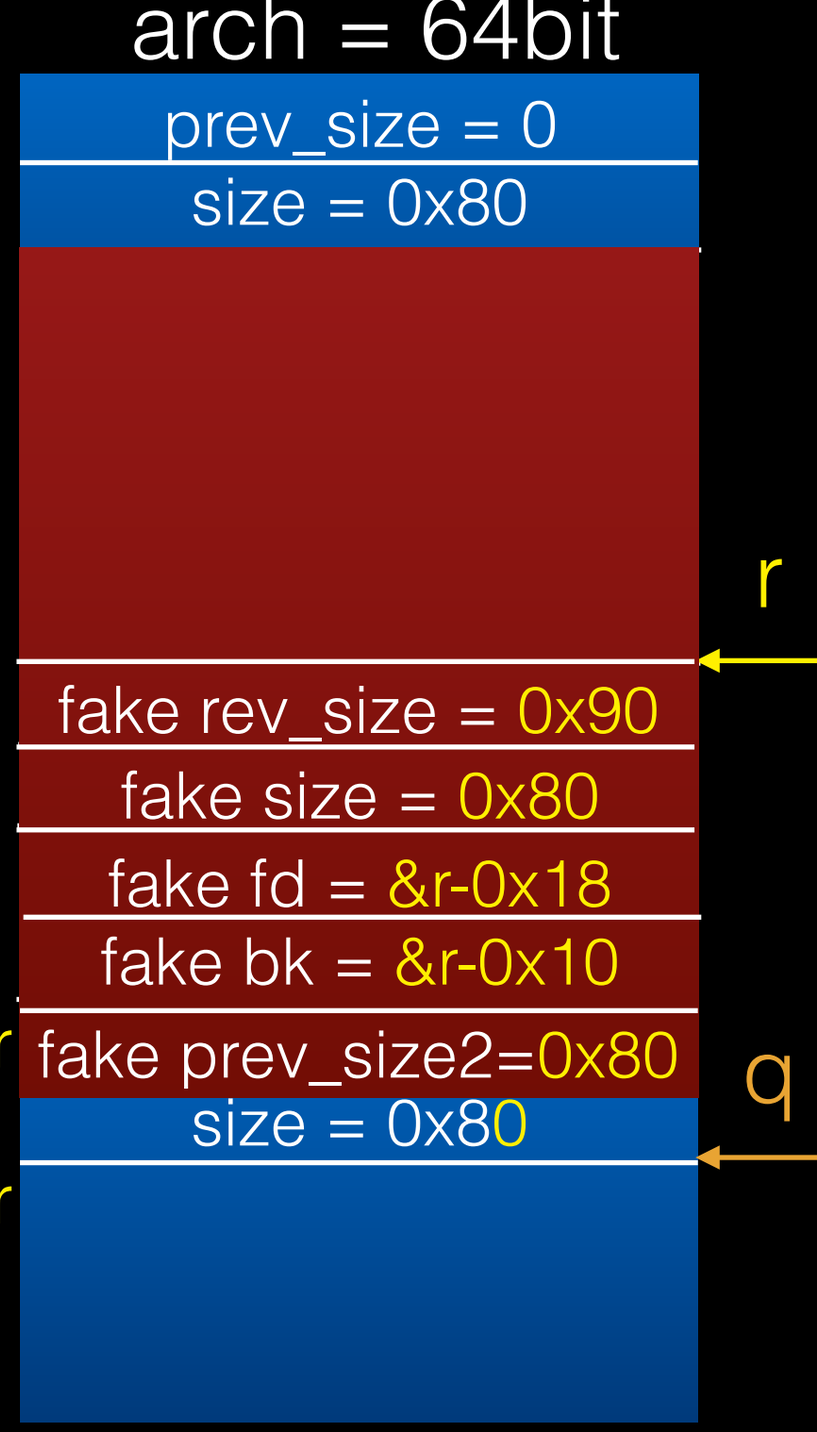
Heap overflow

- using unlink (modern)
  - bypass the detection
    - unlink(r,FD,BK)
      - FD = r->fd = &r - 0x18
      - BK = r->bk = &r - 0x10



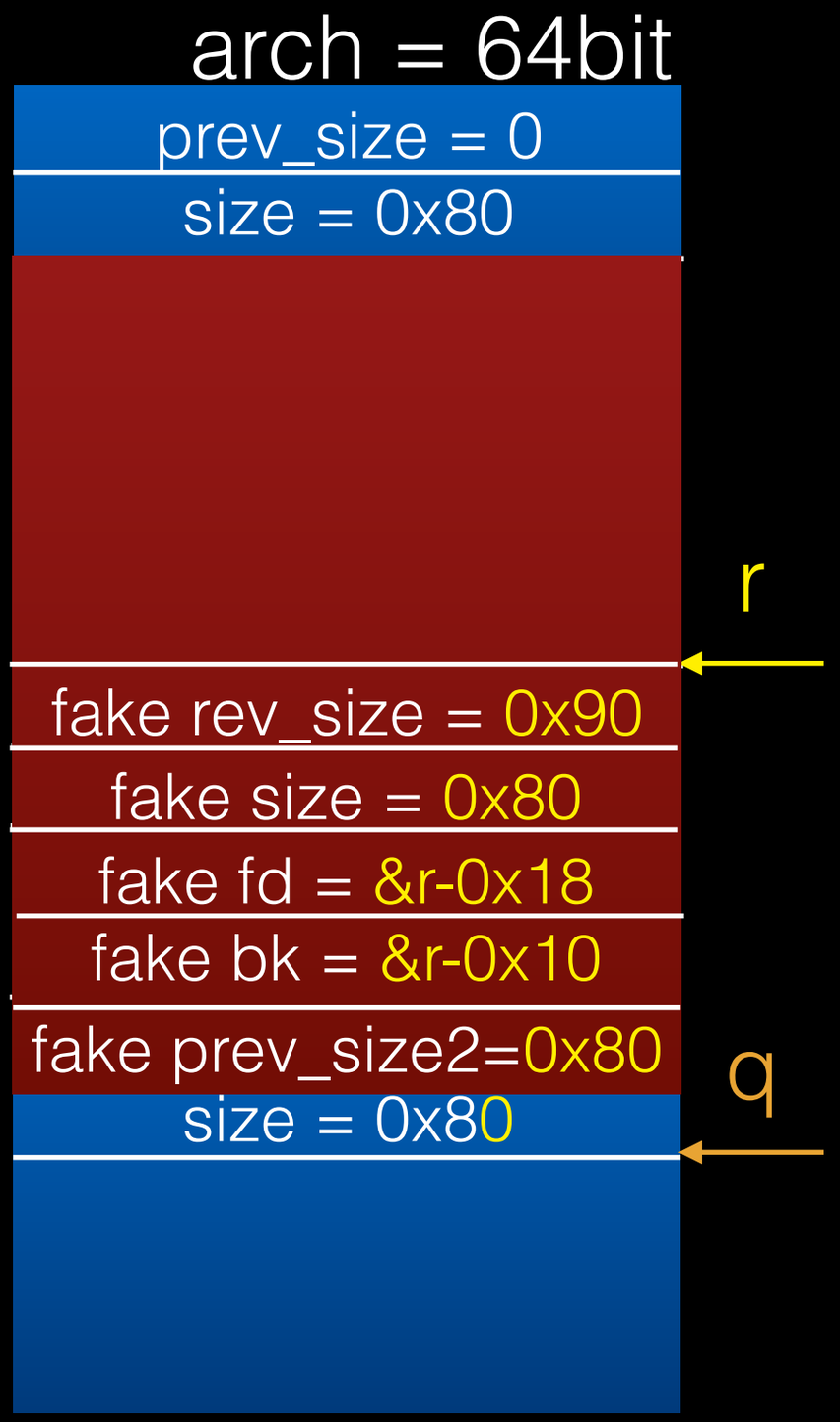
Heap overflow

- using unlink (modern)
  - bypass the detection
    - unlink(r,FD,BK)
      - FD = r->fd = &r - 12
      - BK = r->bk = &r - 8
      - check
        - r->fd->bk == r = *(&r-0x18+0x18) = r
        - r->bk->fd == r = *(&r-0x10+0x10) = r



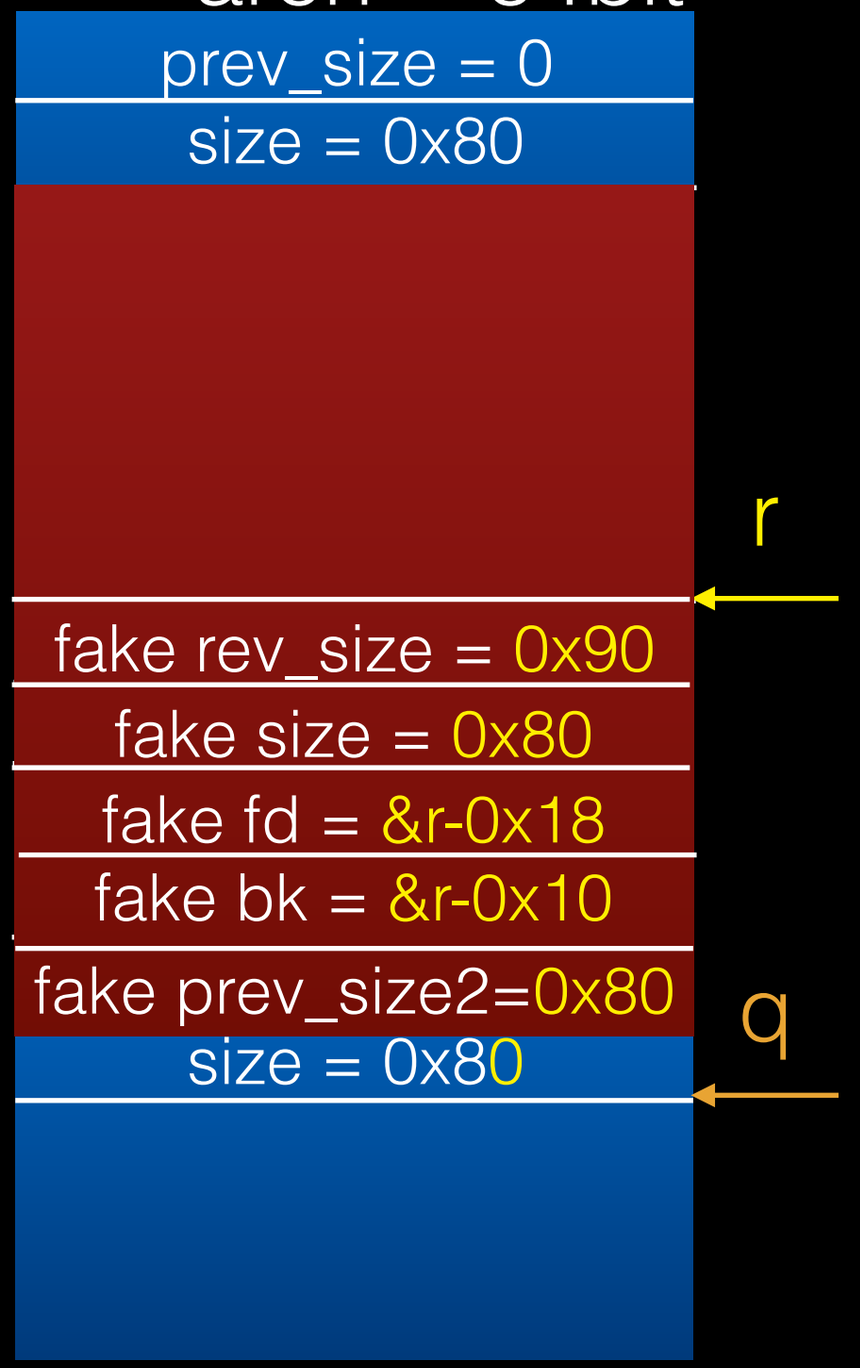
Heap overflow

- using unlink (modern)
  - bypass the detection
    - FD->bk = BK
      - *(&r-0x18+0x18) = &r-0x10
    - BK->fd = FD
      - *(&r-0x10+0x10) = &r-0x18



Heap overflow

- using unlink (modern)
  - bypass the detection
    - FD->bk = BK
      - *(&r-0x18+0x18) = &r-0x10
    - BK->fd = FD
      - *(&r - 0x10 + 0x10) = &r-0x18
- We change the value of &r successful then we may change other thing in mem.



Heap overflow

- using unlink (modern)
  - bypass the detection
    - 通常 r 會是個 data pointer
    - 可以利用他再去改變其他存在 &r 附近的 pointer 然後再利用這些 pointer 去造成任意位置讀取及寫入，如果存在 function pointer 更可直接控制 eip

Heap overflow

- Using malloc maleficarum
  - The House of Mind
  - The House of Prime
  - The House of Spirit
  - The House of Force
  - The House of Lore

Heap overflow

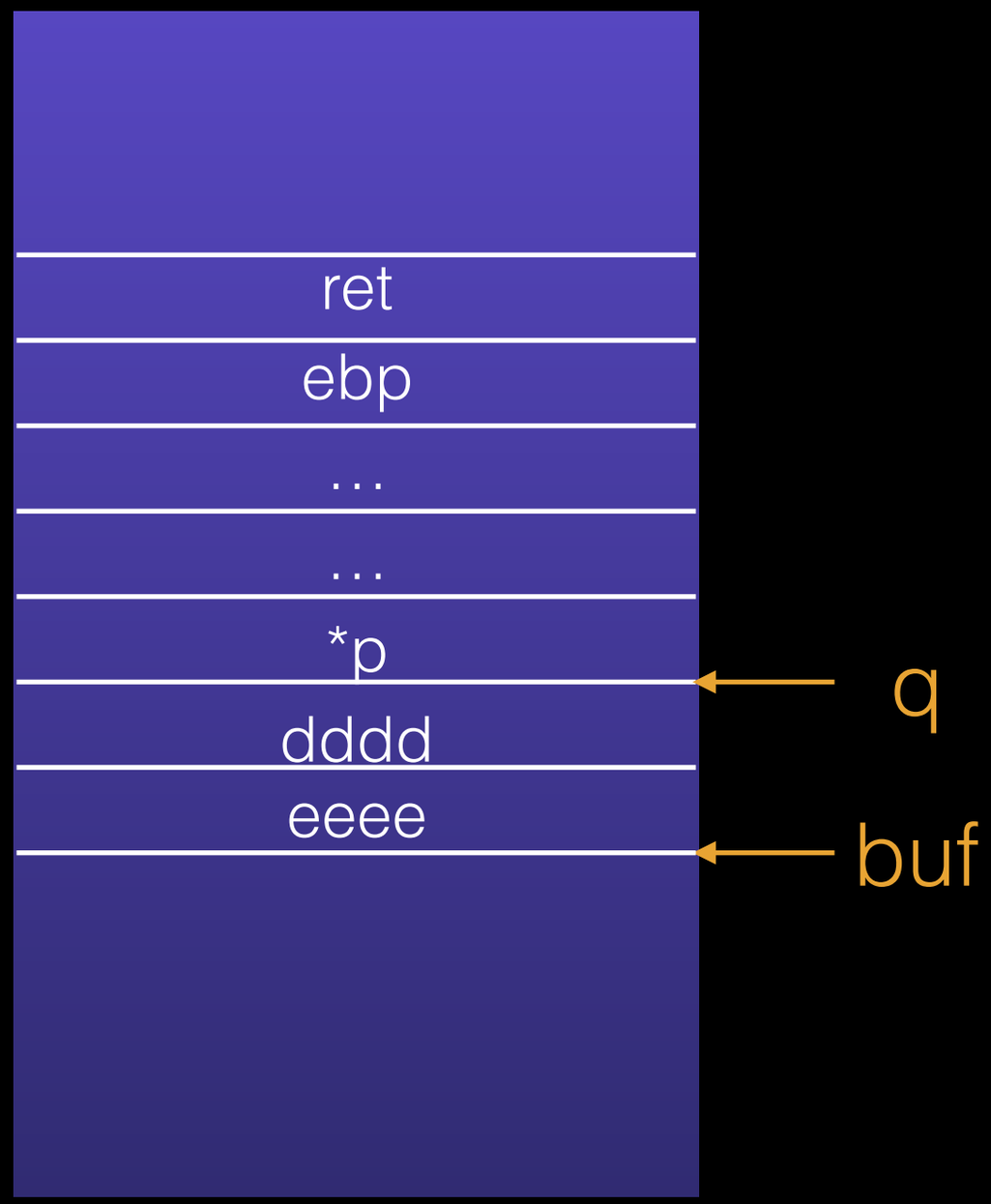
- Using malloc maleficarum
  - The House of Spirit
    - stack overflow
      - 當 stack overflow 不夠蓋到 ret 時
      - 利用 stack overflow 蓋過要 free 的 ptr 並偽造 chunk
      - 須針對 prev_size 及 size 做處理，通過檢查
  - using fastbin
    - 當下次 malloc 時 就會取得偽造的 chunk

Heap overflow

- Using malloc maleficarum
  - The House of Spirit
    - 可以做 information leak
    - 也可加大 stack overflow 的距離
    - 要先算算看在 stack 中取下一塊 chunk 的 size 是否為 8 的倍數，size 的取決是很重要的
      - 64 bit 為 0x10 倍數

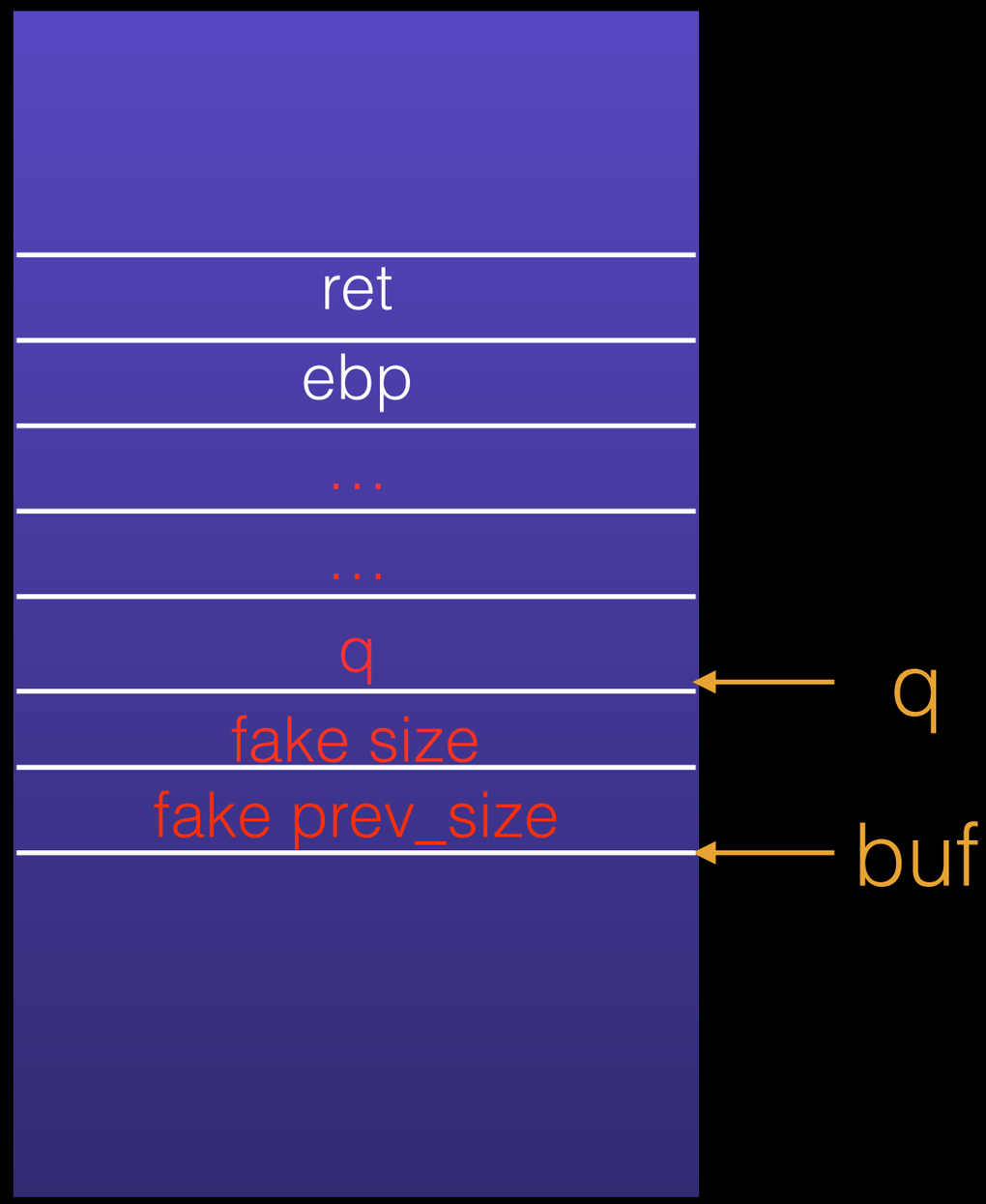
Heap overflow

- Using malloc maleficarum
  - The House of Spirit
    - Assume exist free(p)
    - read(0,buf,size)
    - read 不夠長到可以蓋 ret



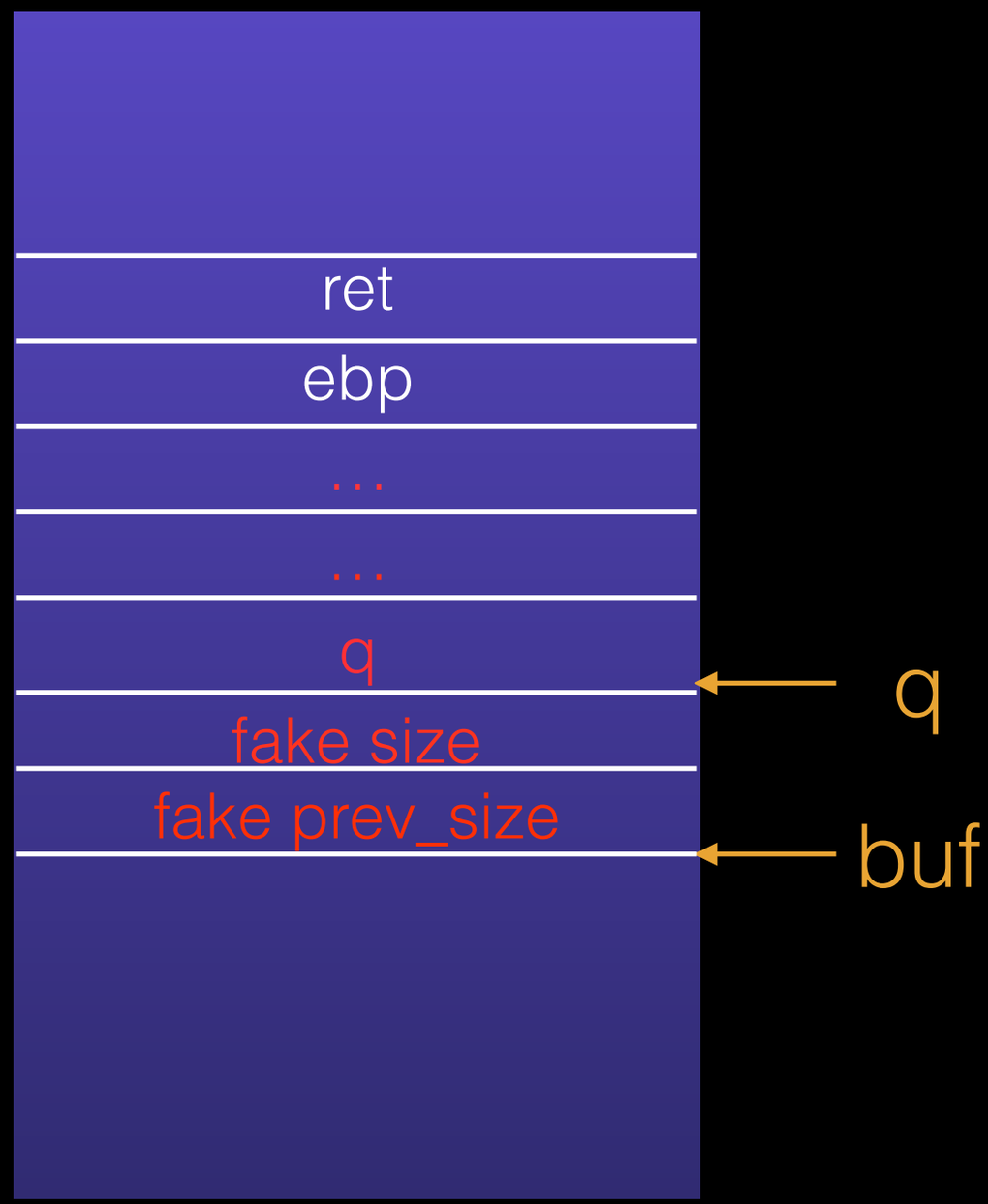
Heap overflow

- Using malloc maleficarum
  - The House of Spirit
    - overflow *p



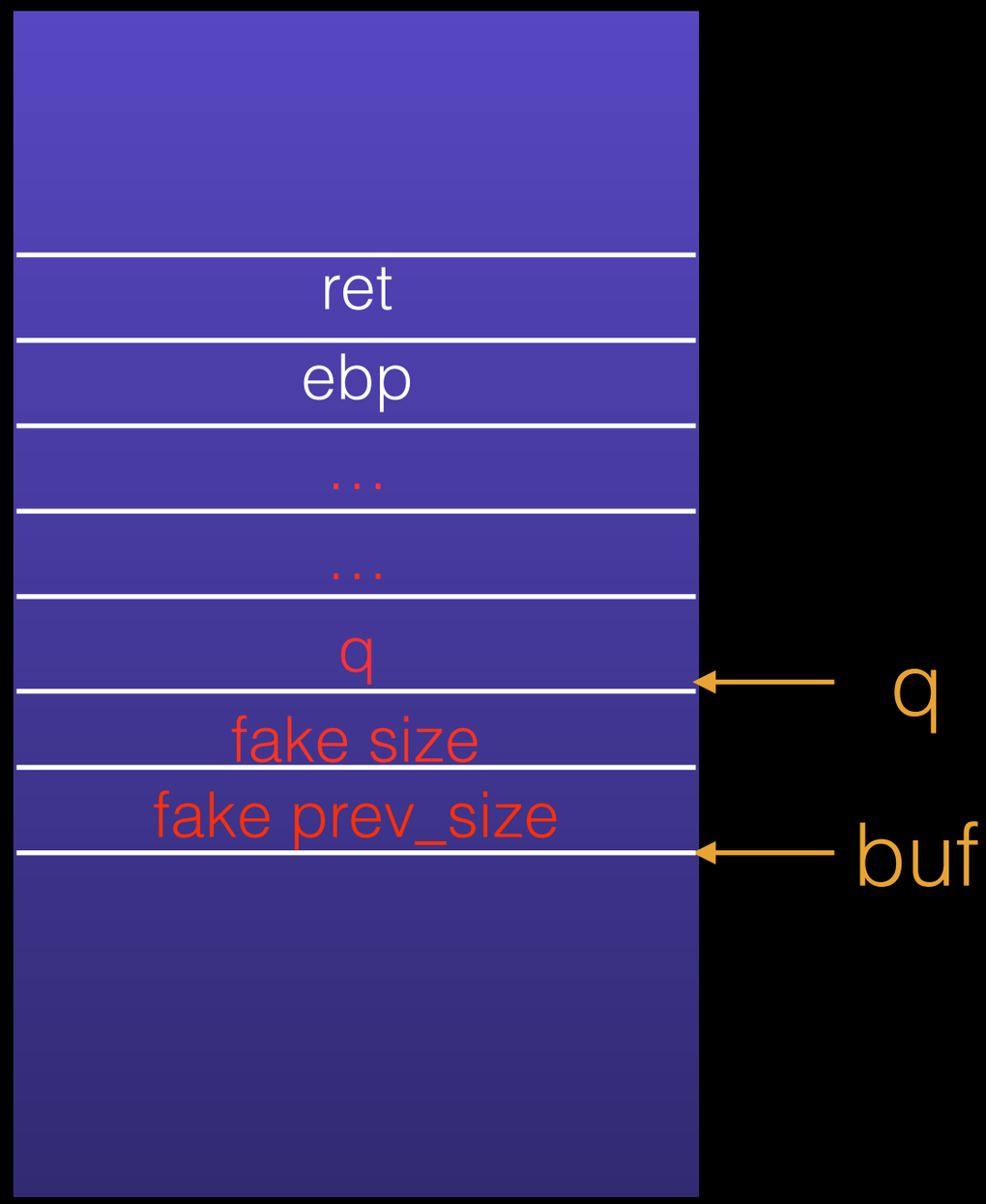
Heap overflow

- Using malloc maleficarum
  - The House of Spirit
    - overflow *p
    - free(q)



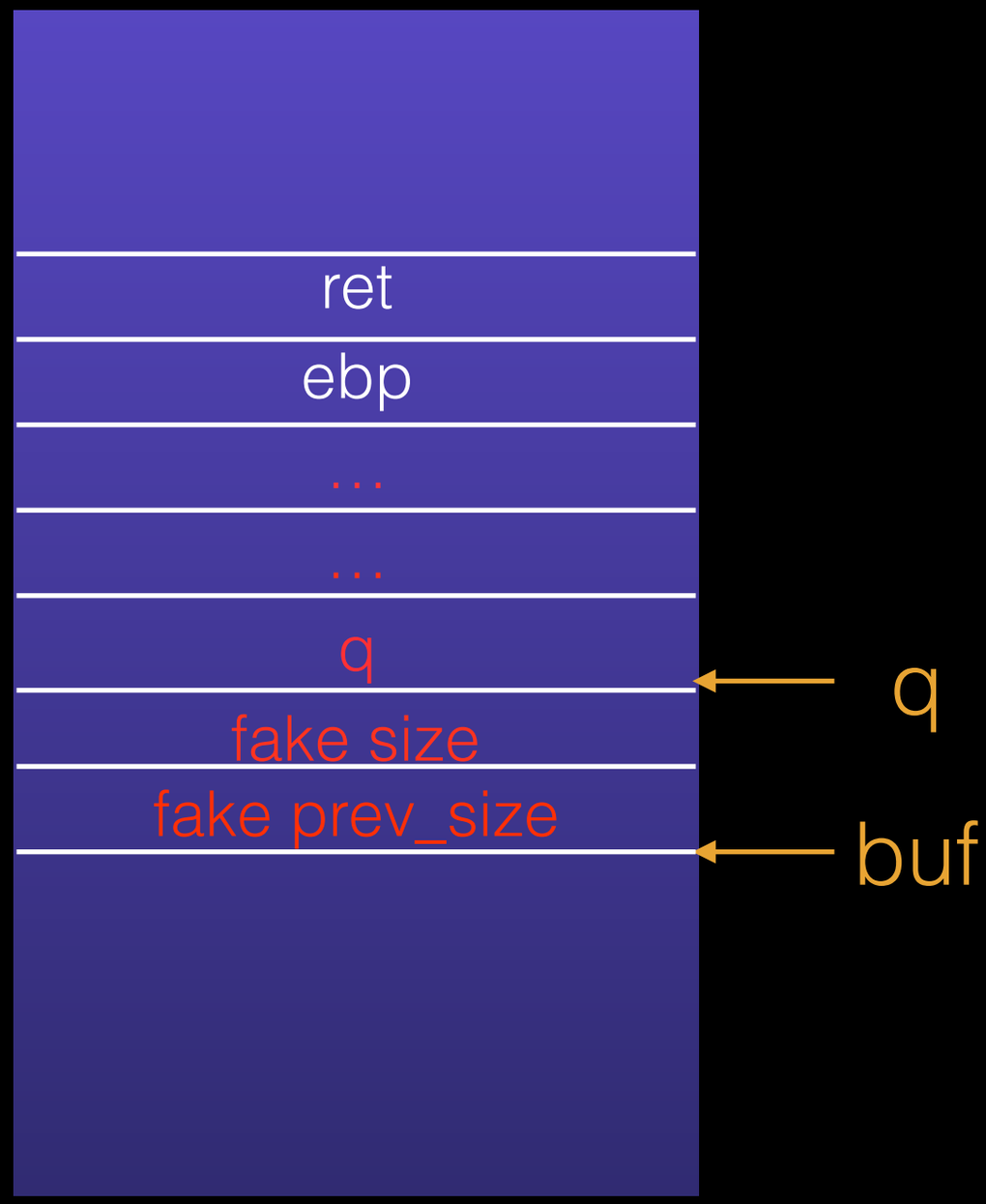
Heap overflow

- Using malloc maleficarum
  - The House of Spirit
    - overflow *p
    - free(q)
    - malloc(size-8)
      - it will return q



Heap overflow

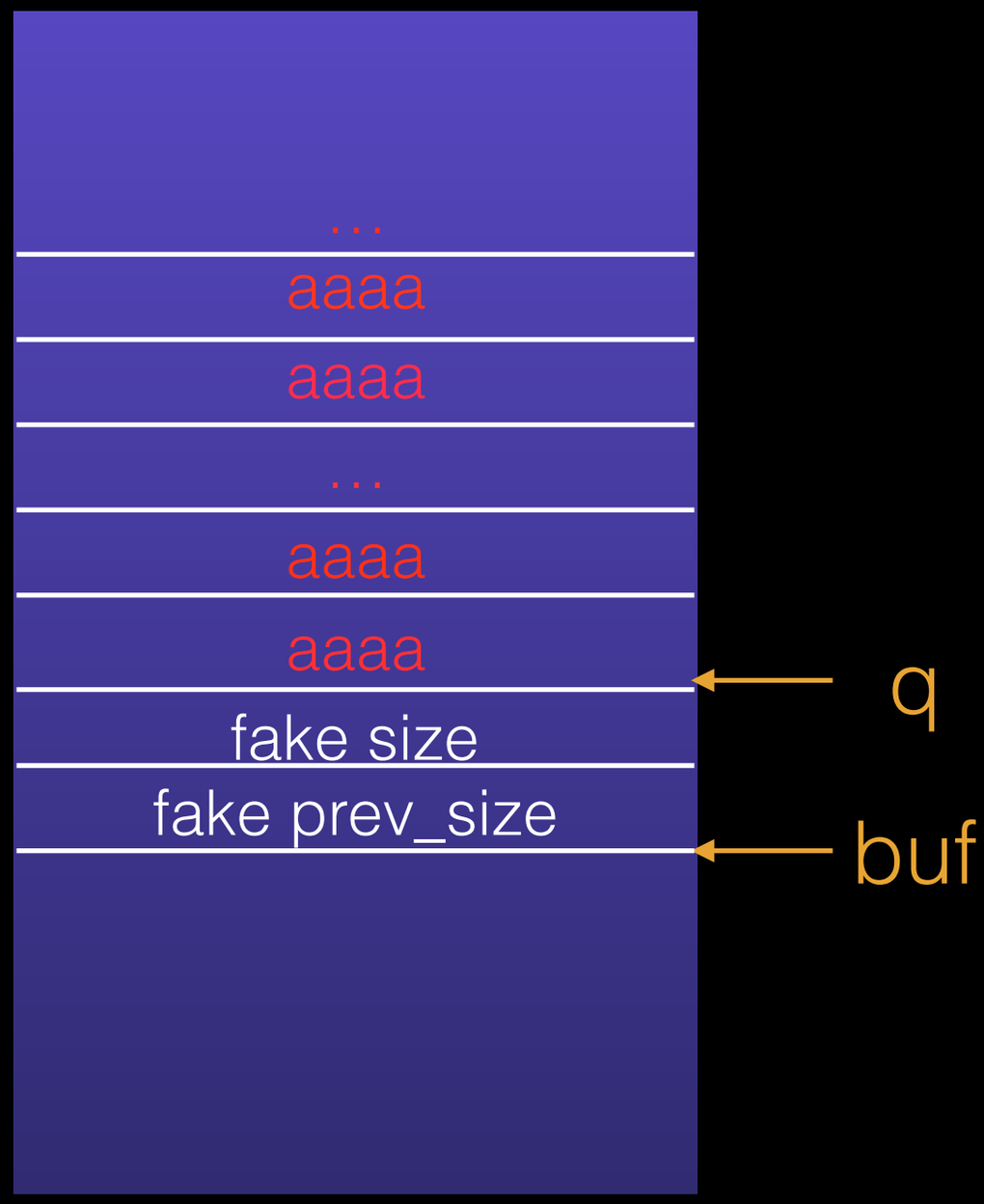
- Using malloc maleficarum
  - The House of Spirit
    - overflow *p
    - free(q)
    - malloc(size-8)
      - it will return q
- read(0,q,size)



Heap overflow

- Using malloc maleficarum
  - The House of Spirit
    - overflow *p
    - free(q)
    - malloc(size-8)
      - it will return q
- read(0,q,size)

overflow



Heap overflow

- Using malloc maleficarum
  - The House of Force
    - heap overflow 蓋過 top chunk 的 size，變成一個很大的值
      - 下次 malloc 時，malloc 一個很大的數字(nb)，然後 arena header 中的 top chunk 的位置會改變
        - new top chunk = old top + nb + 8
        - nb 可以是負的，因為傳進去會被自動轉成很大的數字，只要讓 fake size - nb > 0 就會讓 glibc 以為 top 還有空間可以給
      - 這樣下次 malloc 的位置將會是 new top chunk 的位置

Heap overflow

- Using fastbin
  - 類似 House of Spirit
  - 如果可以改到 fastbin 的 free chunk 可以將 fd 改成 fake chunk 的位置，只要符合 size 是屬於該 fastbin 就好，因為在下一次 malloc 只會檢查這項
  - 下下次 malloc(size-0x10) 時，就會取得該 fake chunk 的位置
  - fake chunk 可以是任意記憶體位置

Heap overflow

- Some useful global variable
  - __free_hook、__malloc_hook、__realloc_hook
    - free,malloc,realloc 前會先檢查是否上面三個全域變數是否有值，若有則當 func ptr 執行
  - __malloc_initialize_hook
    - ptmalloc_init 時（第一次執行 malloc 會用），會檢查該變數是否有值，若有則當 func ptr 執行
  - __after_morecore_hook
    - 在跟用 brk 跟系統要 heap 空間時，會檢查該變數是否有值，若有則當 func ptr 執行

Practice

- bamboobox1
  - http://train.cs.nctu.edu.tw/problems/12
    - House of force
- bamboobox2
  - http://train.cs.nctu.edu.tw/problems/13
    - Using unlink

Reference

- understanding-glibc-malloc
- Modern Binary Exploitation - heap exploitation
- MallocMaleficarum
- glibc cross reference
- SP heap exploitation by dada