

Linux binary Exploitation

Heap exploitation - Glibc
angelboy@chroot.org

Outline

- Glibc memory allocator Overview
- Use After Free
- Heap Overflow
- Appendix - Detection in Glibc

Outline

- Glibc memory allocator Overview
- Use After Free
- Heap Overflow
- Appendix - Detection in Glibc

Memory allocator

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

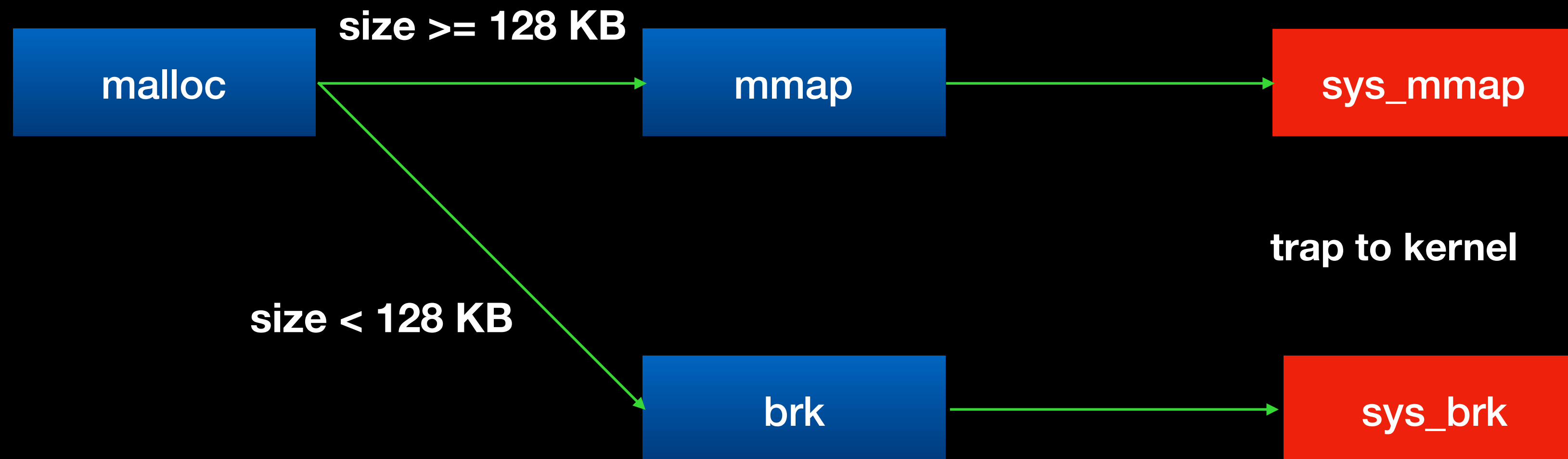
What is malloc

- 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

```
1 #include <stdio.h>
2
3 int main(void){
4     int size = 0 ;
5     char *p = NULL ;
6     puts("Enter your length of name");
7     scanf("%d",&size);
8     p = (char *)malloc(size+1) ;
9     puts("Enter your name");
10    read(0,p,size);
11    printf("Hello %s\n",p);
12    free(p);
13    return 0;
14 }
```

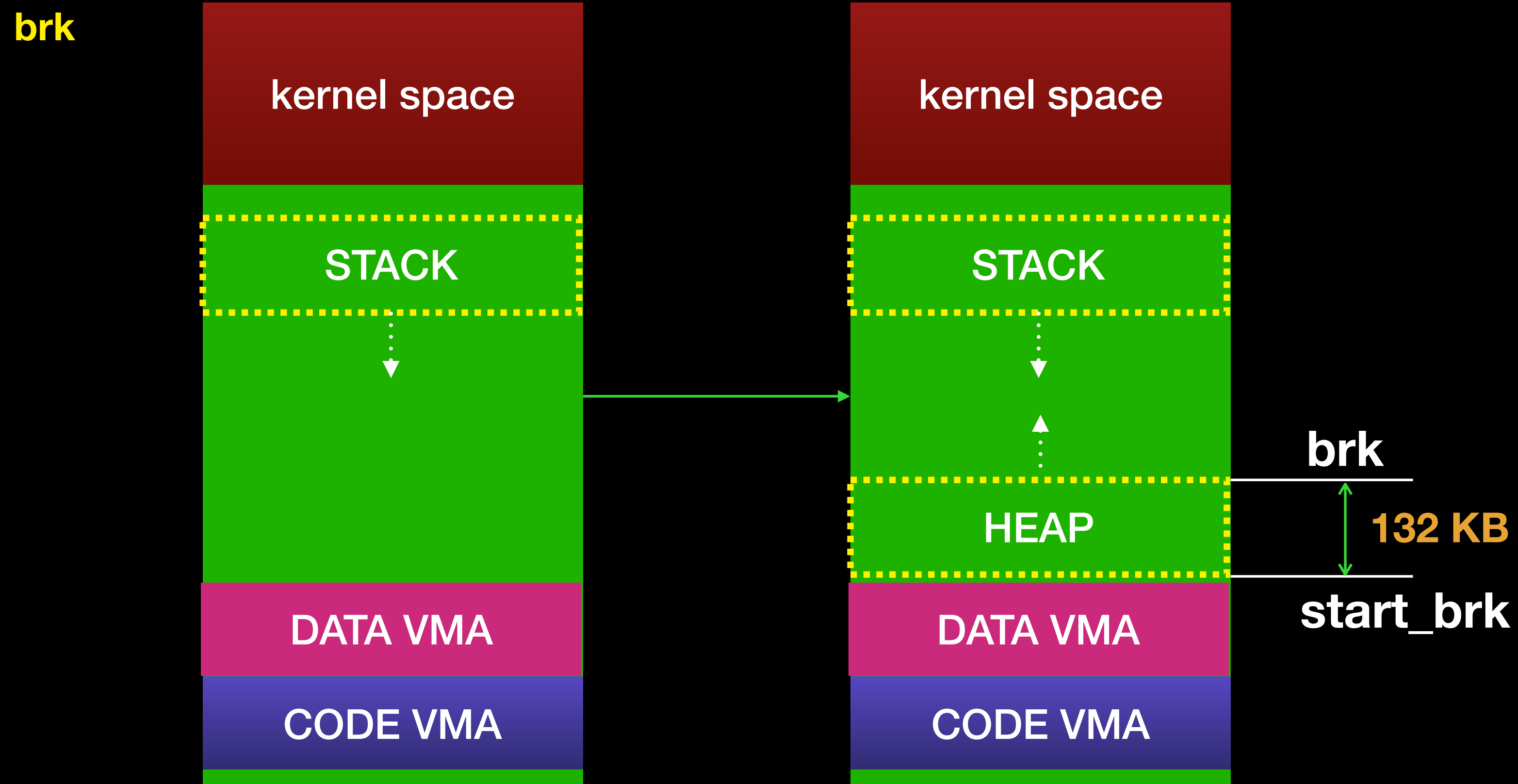
The workflow of malloc

- 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

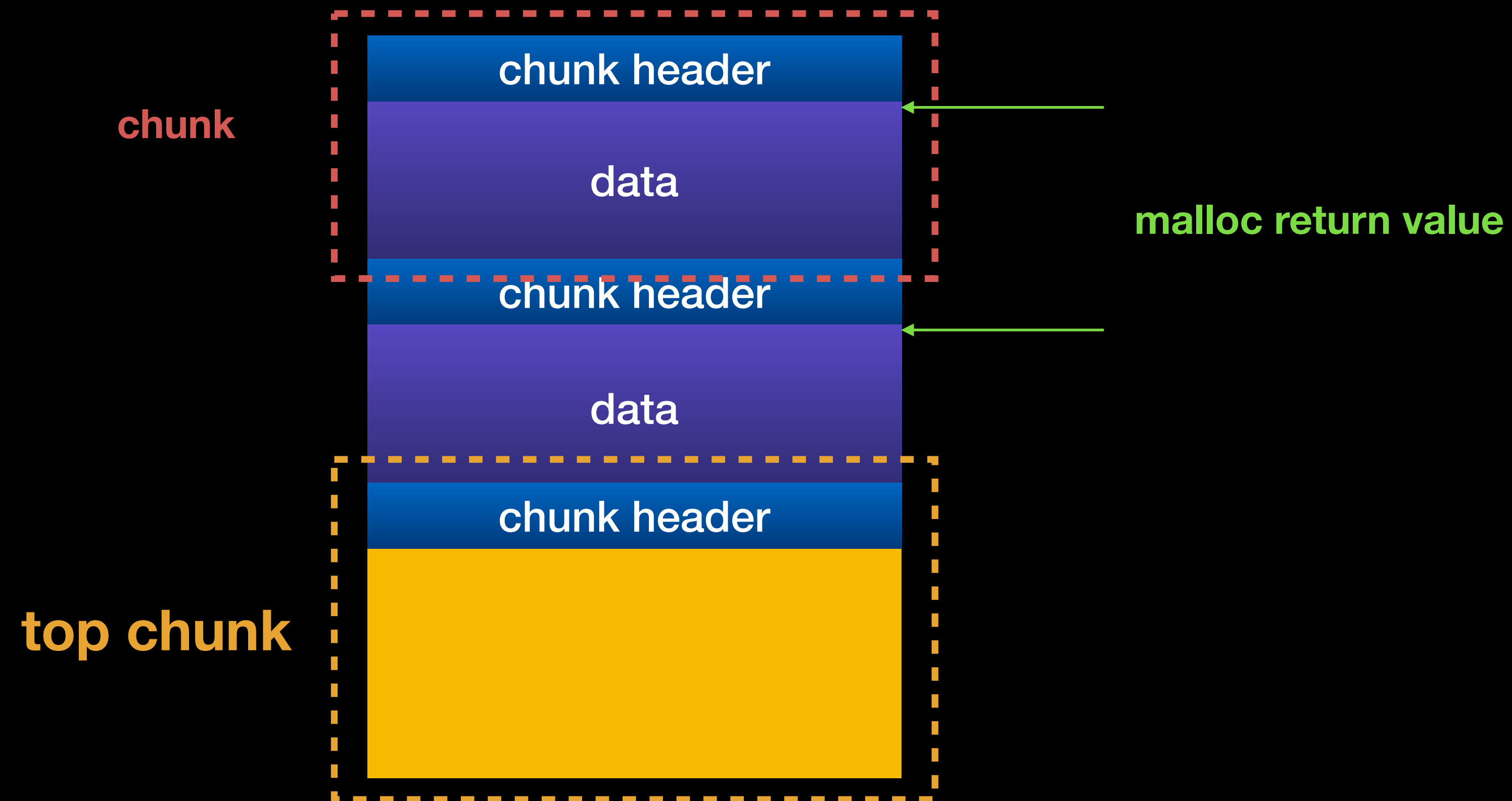
- 第二次執行 malloc 時，只要分配出去的總記憶體空間大小不超過 128 KB，則不會再執行 system call 跟系統要空間，超過大小才會用 brk 來跟 kernel 要記憶體空間
 - 即使將所有 main arena 所分配出去的記憶體 free 完，也不會立即還給 kernel
- 這時的記憶體空間將由 glibc 來管理
- 本投影片如未特別註明都以 64 位元電腦 glibc-2.23 為主，換到 32 位元則大多數的欄位大小都須除二

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即為一個 chunk (最小為 $\text{SIZE_T} \times 4$)
 - $\text{SIZE_T} = \text{unsigned long int}$
 - 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

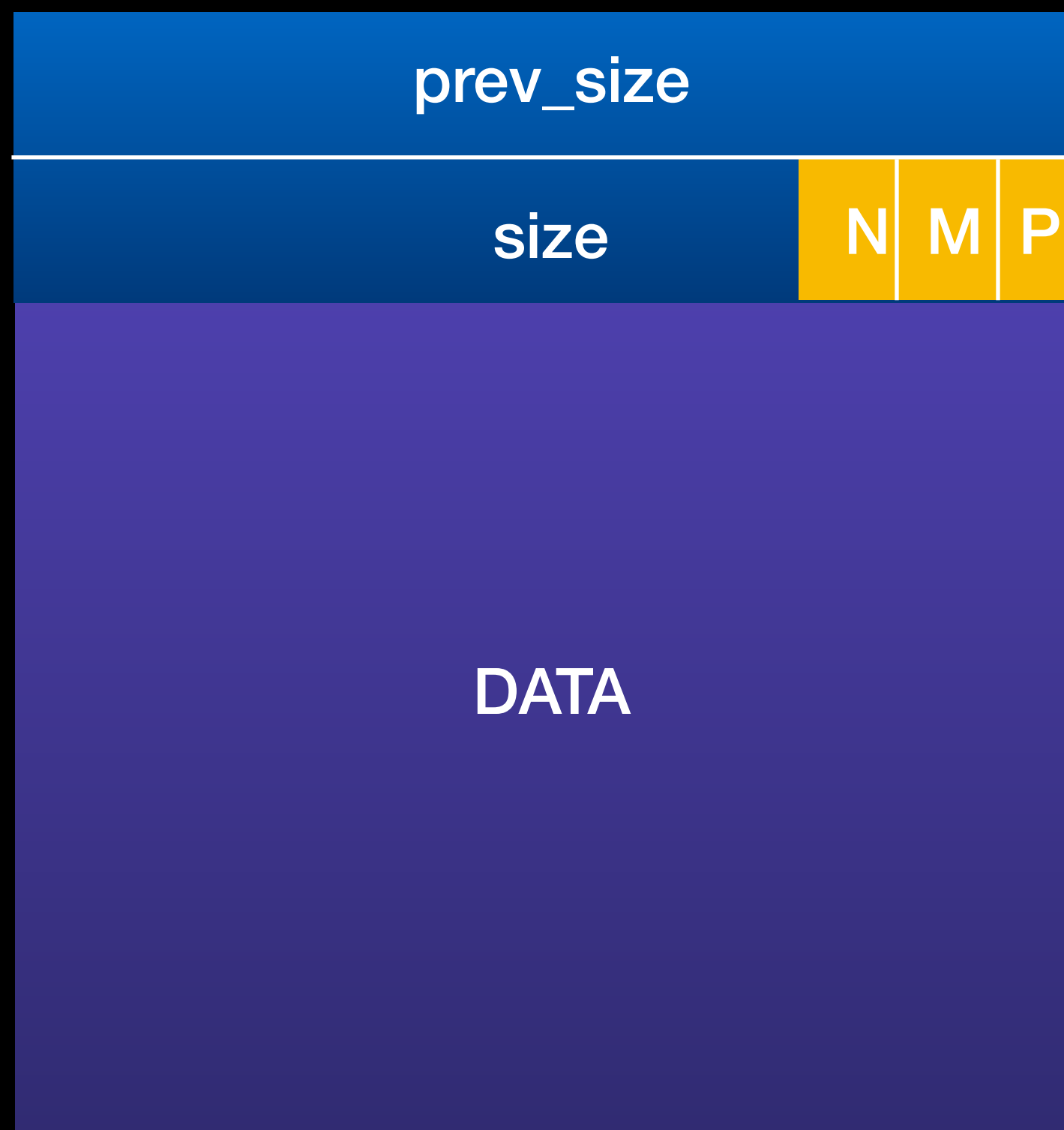


Mechanism of glibc malloc

- 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



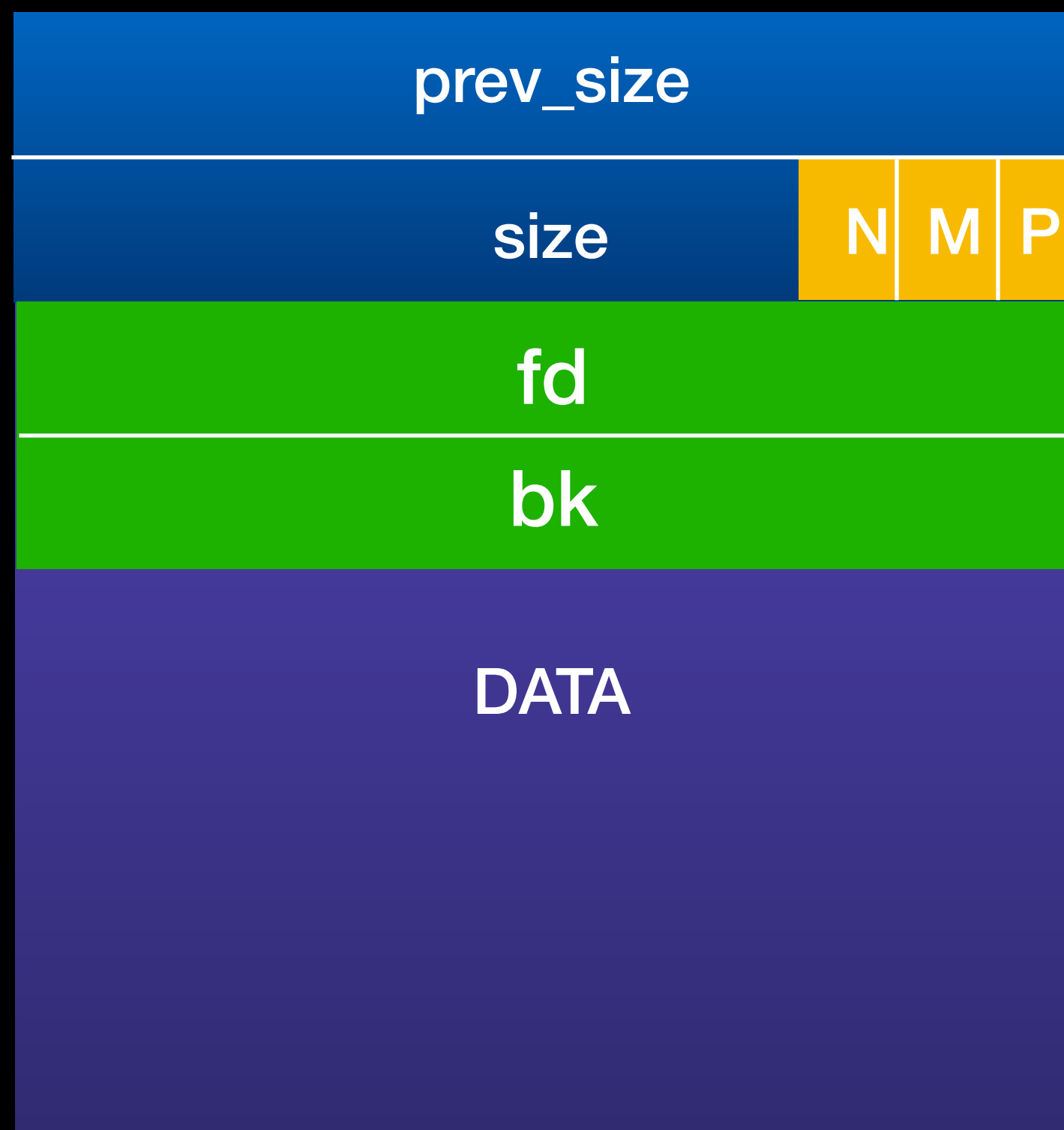
P:PREV_INUSED
M:IS_MMAPPED
N:NON_MAIN_ARENA

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_INUSED
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

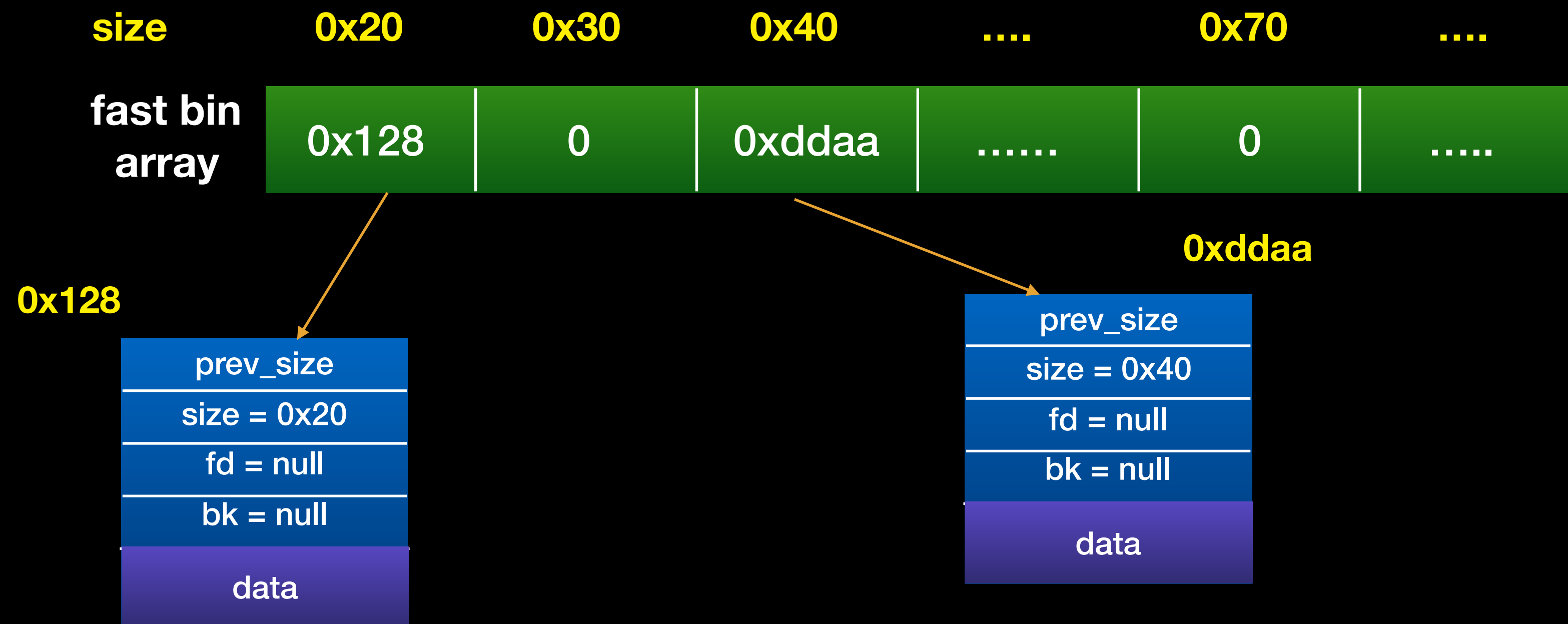
- bin
 - linked list
 - 為了讓 malloc 可以更快找到適合大小的 chunk，因此在 free 掉一個 chunk 時，會把該 chunk 根據大小加入適合的 bin 中
- 根據大小一共會分為
 - fast bin
 - small bin
 - large bin
 - unsorted bin

Mechanism of glibc malloc

- fast bin
 - a singly linked list
 - chunk size < 144 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 0x20,0x30,0x40...
- 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

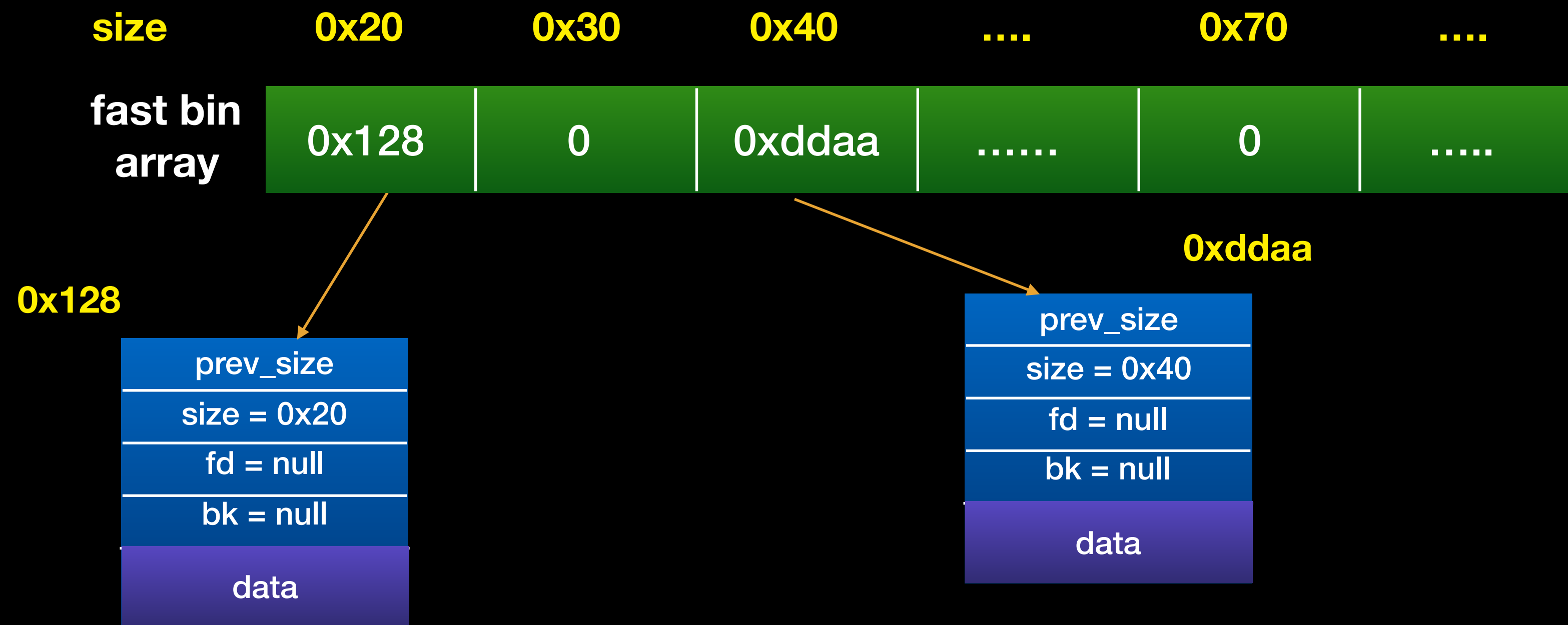
Mechanism of glibc malloc

- fast bin



Mechanism of glibc malloc

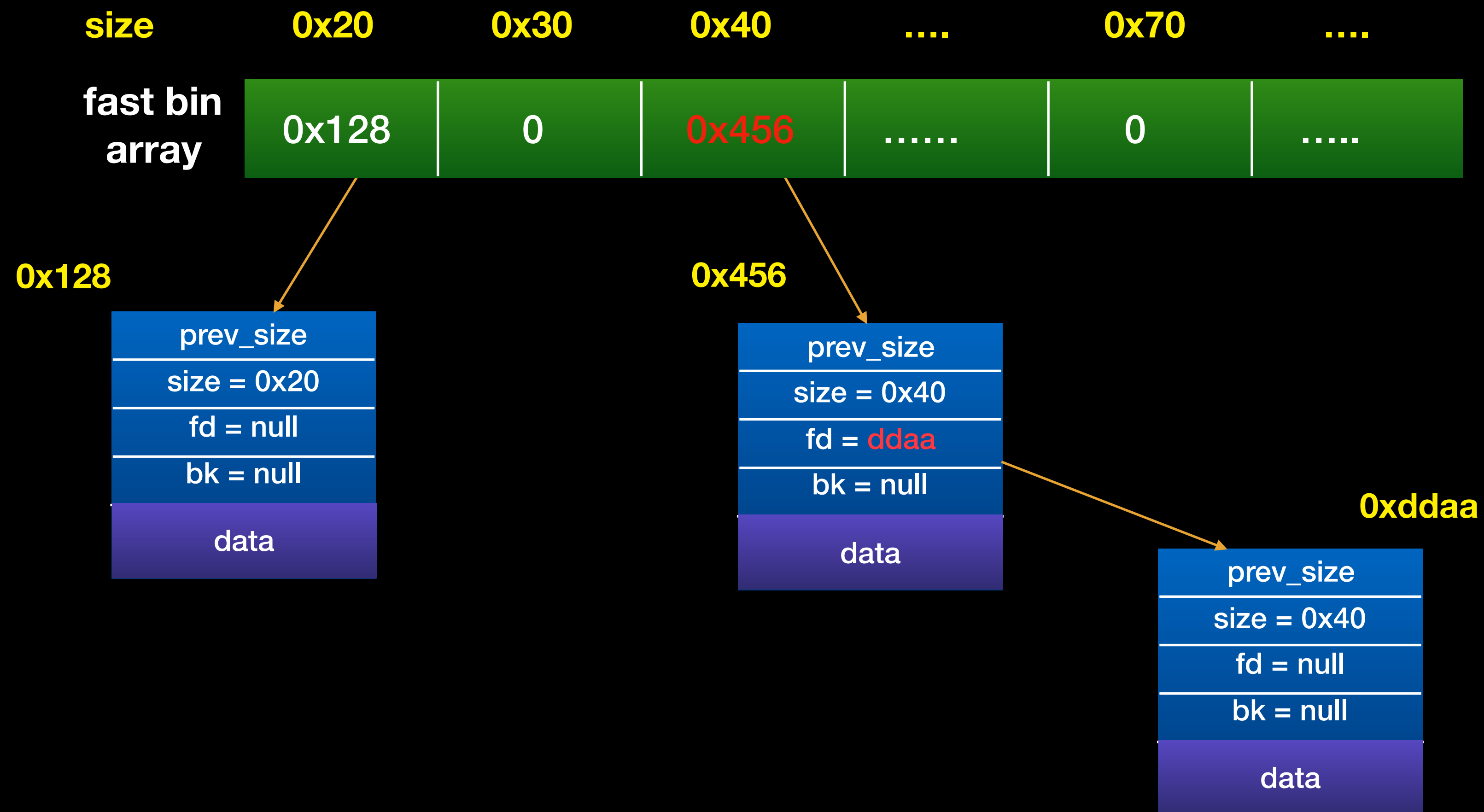
- fast bin



free(0x466)

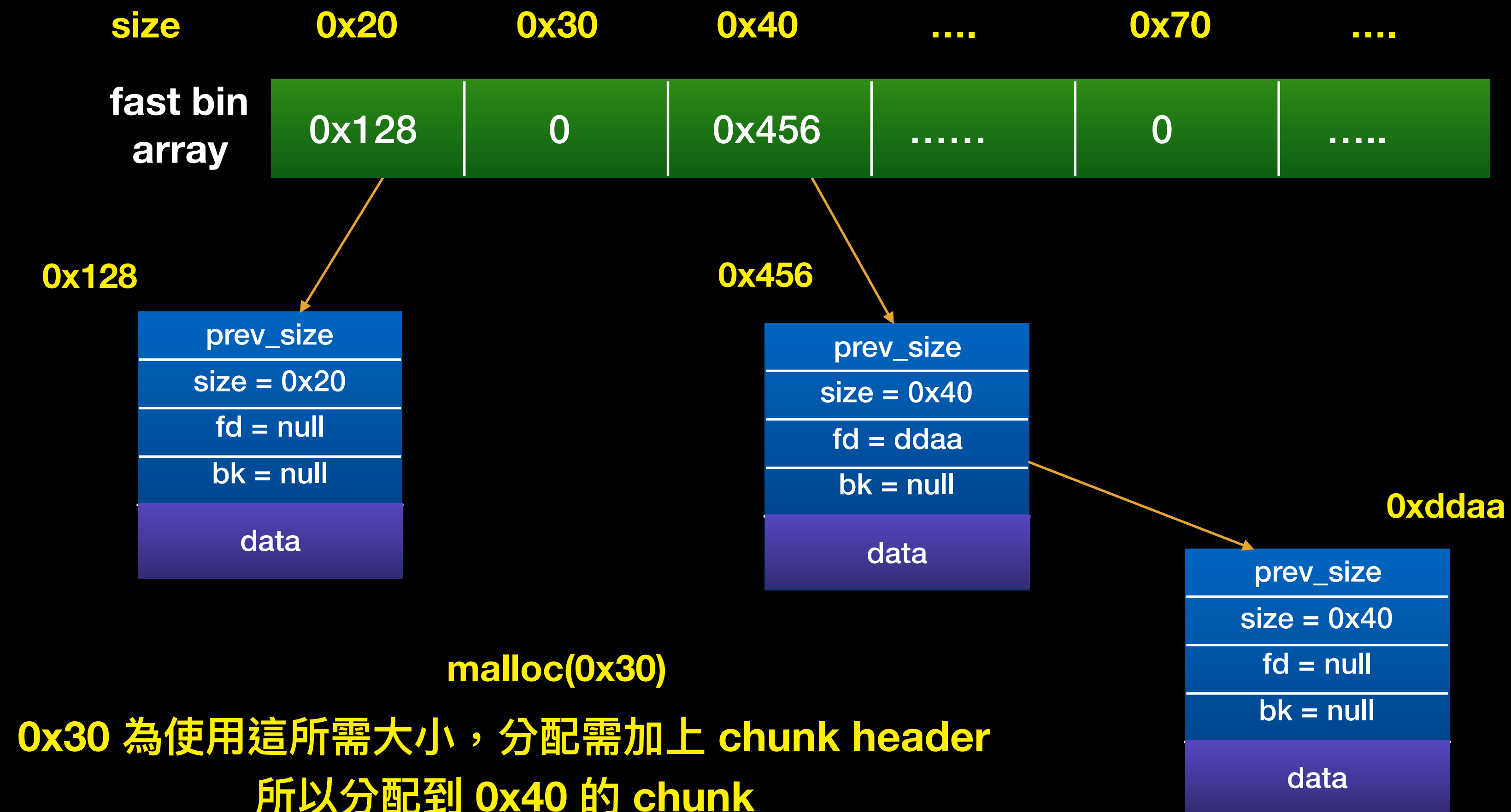
Mechanism of glibc malloc

- fast bin



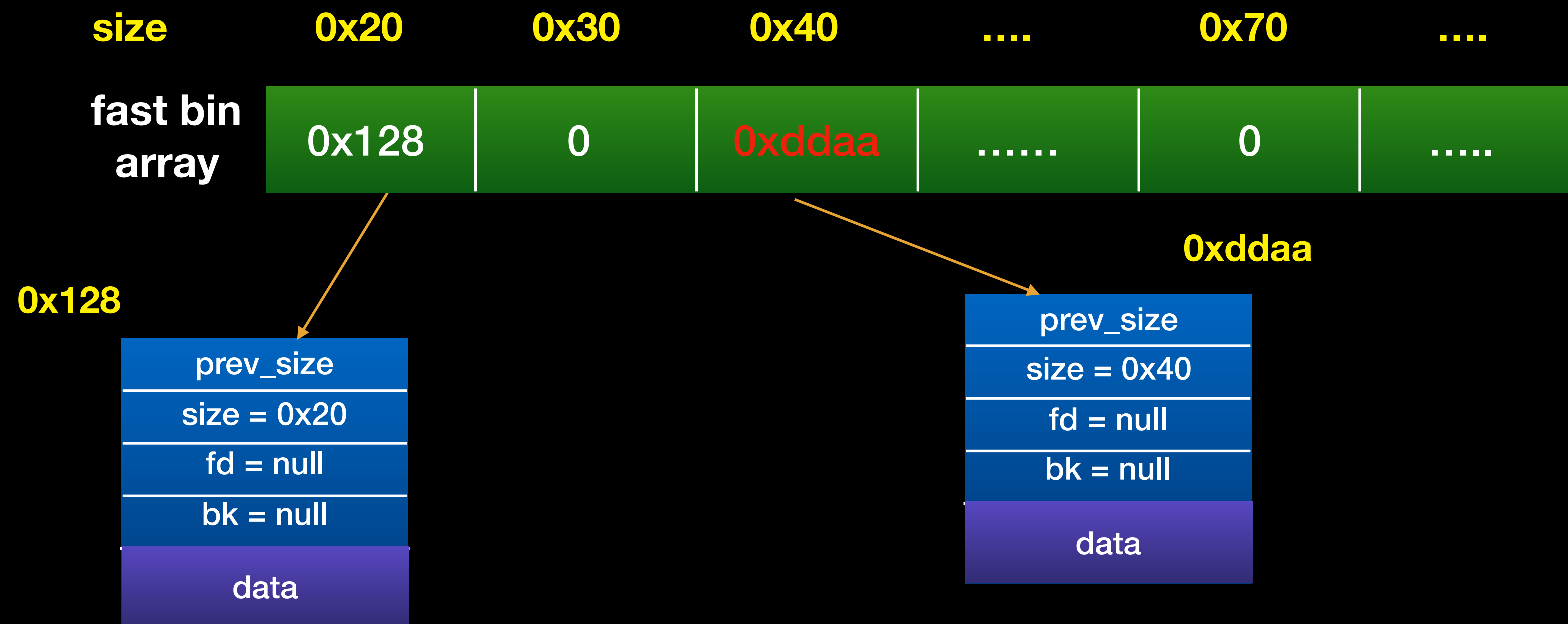
Mechanism of glibc malloc

- fast bin



Mechanism of glibc malloc

- fast bin



Mechanism of glibc malloc

- unsorted bin
- circular doubly linked list
- 當 free 的 chunk 大小大於等於 144 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
- 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，此時會順便把 unsorted bin 的 chunk 放到對應的 bin 中，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 < 1024 byte
 - FIFO
 - 根據大小在分成 62 個大小不同的 bin
 - 0x20,0x30...0x60,0x70,0x90,.....1008

Mechanism of glibc malloc

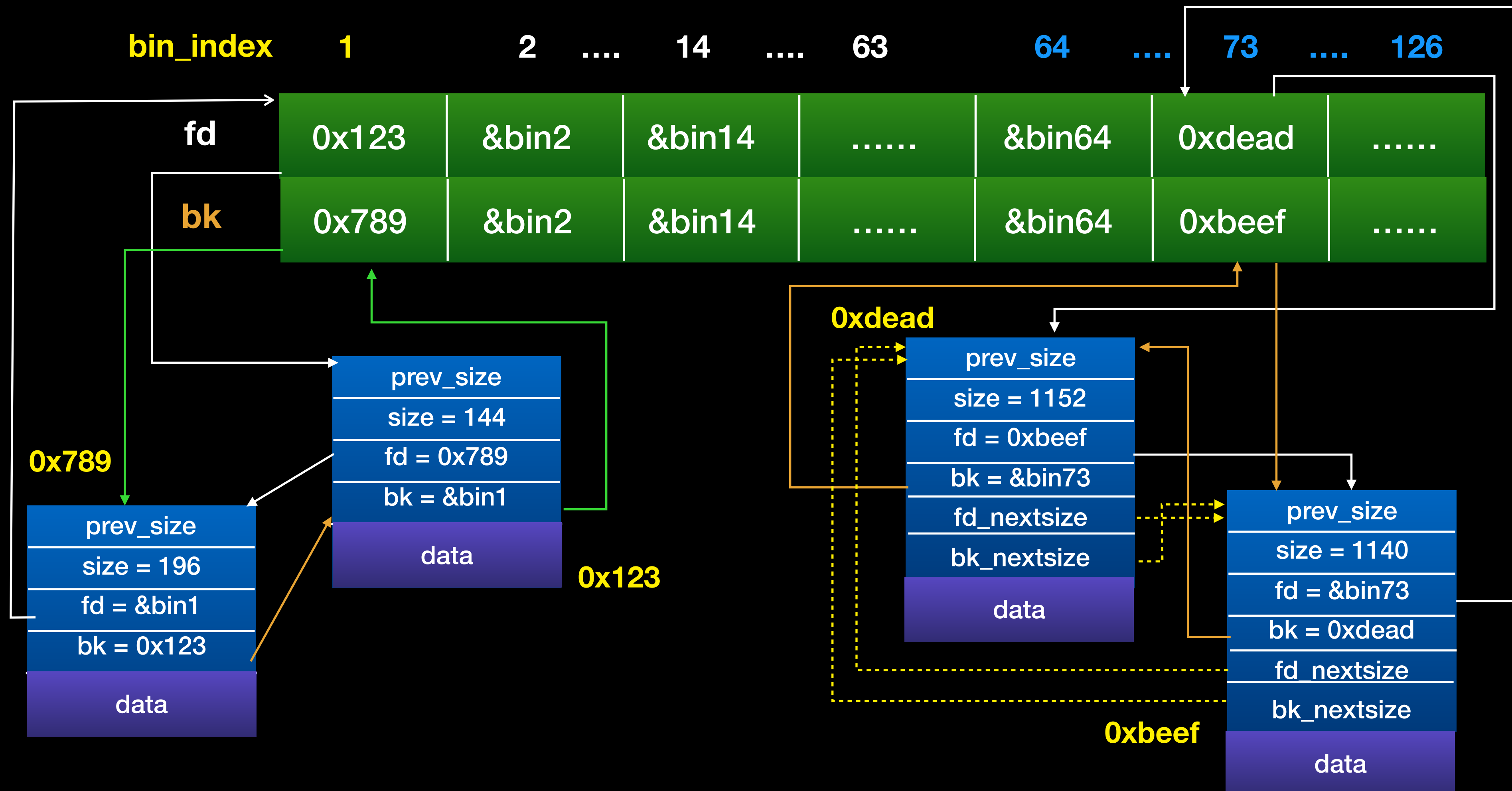
- large bin
 - circular doubly linked list (sorted list)
 - chunk size ≥ 1024
 - freed chunk 多兩個欄位 `fd_nextsize` 、`bk_nextsize` 指向前一塊跟後一塊 large chunk
 - 根據大小在分成 63 個 bin 但大小不再是一固定大小增加
 - 前 32 個 bin 為 $0x400 + 64 * i$
 - 32 - 48 bin 為 $0x1380 + 512 * j$
 -依此類推
 - 不再是每個 bin 中的 chunk 大小都固定，每個 bin 中存著該範圍內不同大小的 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
- FIFO

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 remainder 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
 - 存有所有的 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

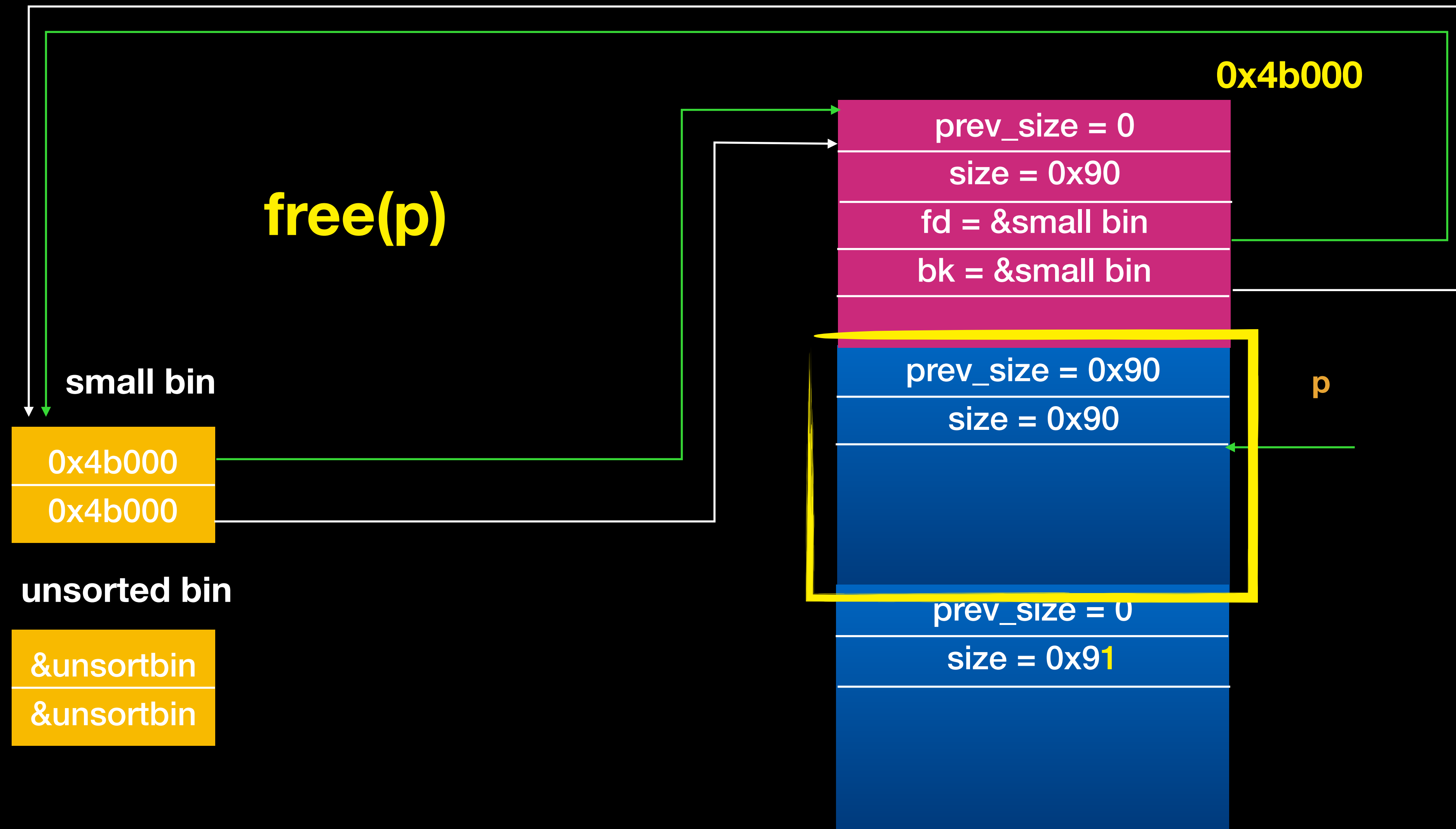
Mechanism of glibc malloc

- Merge freed chunk
 - 執行 free 後 unlink 的條件，再 chunk 為非 mmaped 出來的 chunk 時
 - 如果連續記憶體中下一塊是 top chunk，且上一塊是 free chunk
 - 最後合併到 top chunk
 - 如果連續記憶體中下一塊不是 top chunk
 - 上一塊是 free chunk
 - 下一塊是 free chunk

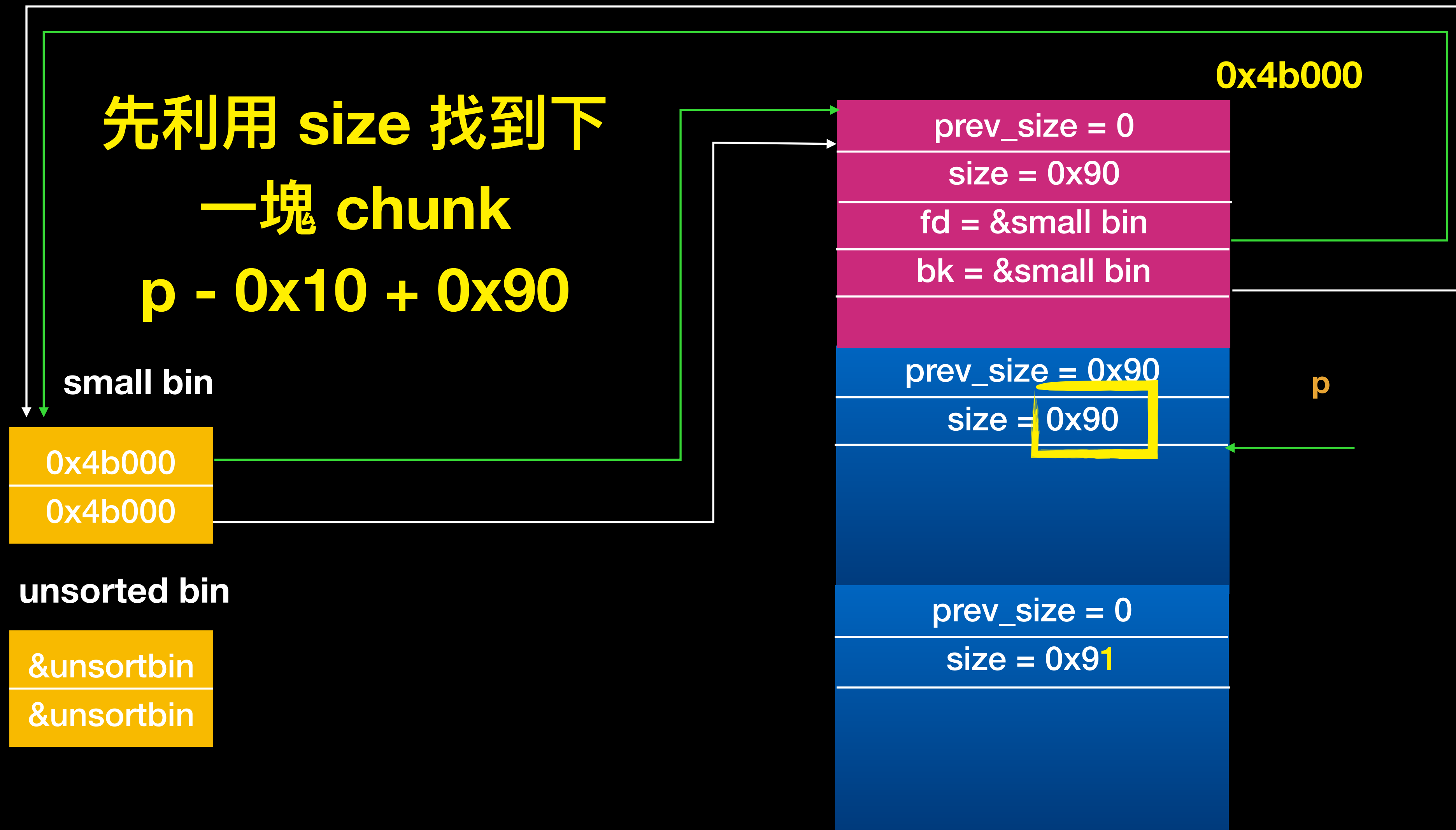
Mechanism of glibc malloc

- Merge freed chunk
 - 合併流程
 - 如果上一塊是 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 ，並對下一塊做 unlink ，最後加入 unsortbin
 - inuse : 加入 unsortbin

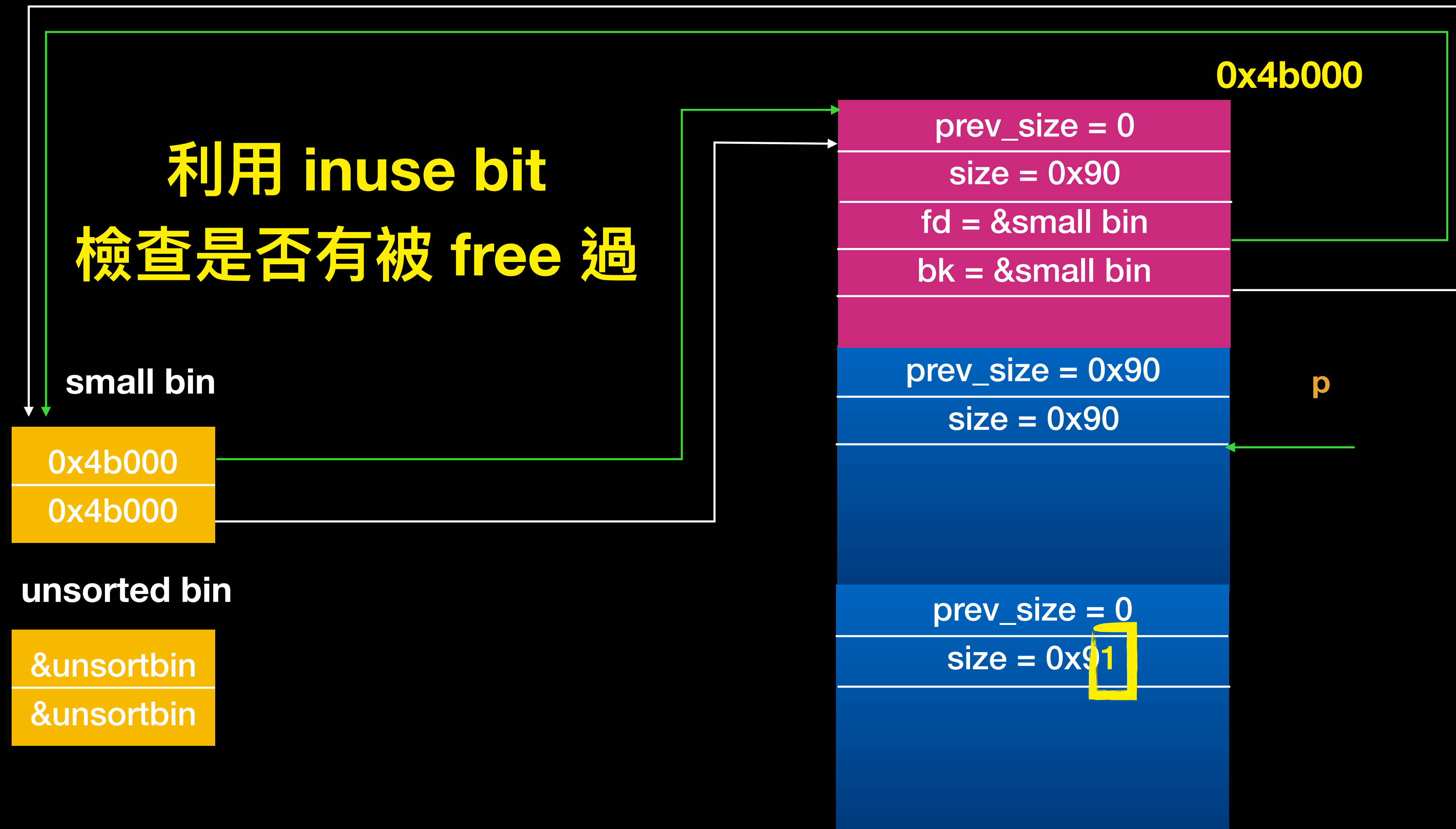
Mechanism of glibc malloc



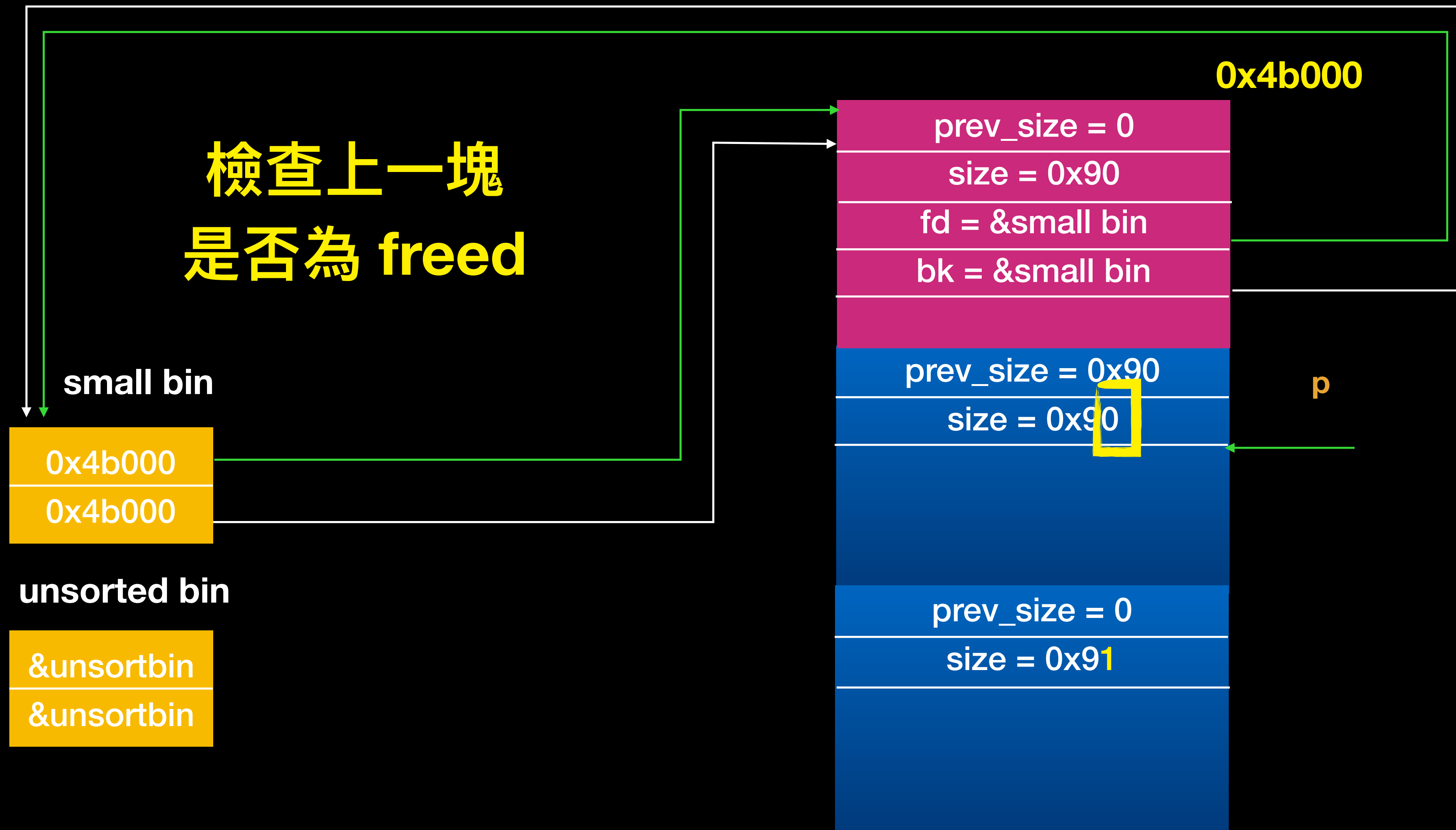
Mechanism of glibc malloc



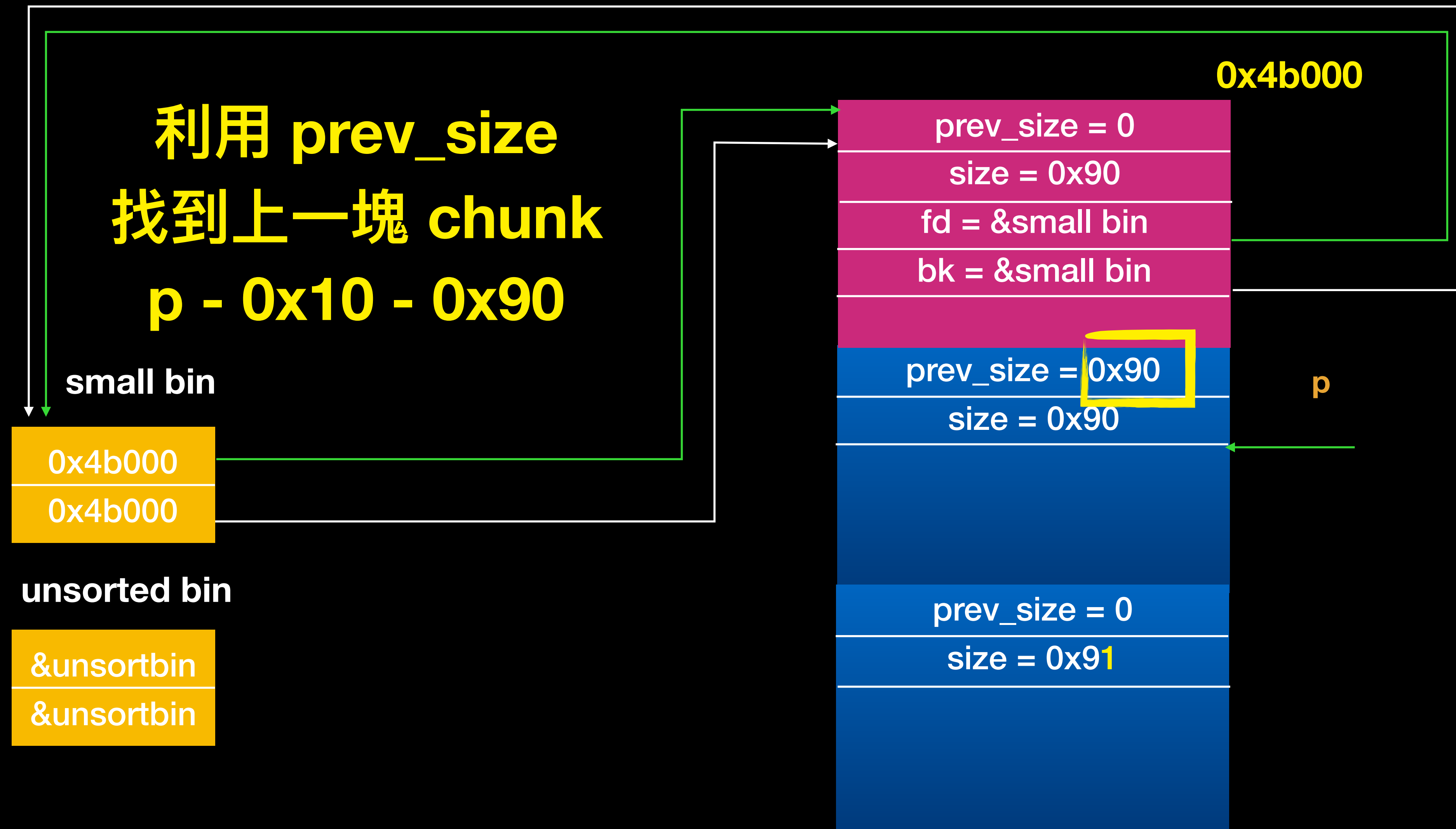
Mechanism of glibc malloc



Mechanism of glibc malloc



Mechanism of glibc malloc



Mechanism of glibc malloc

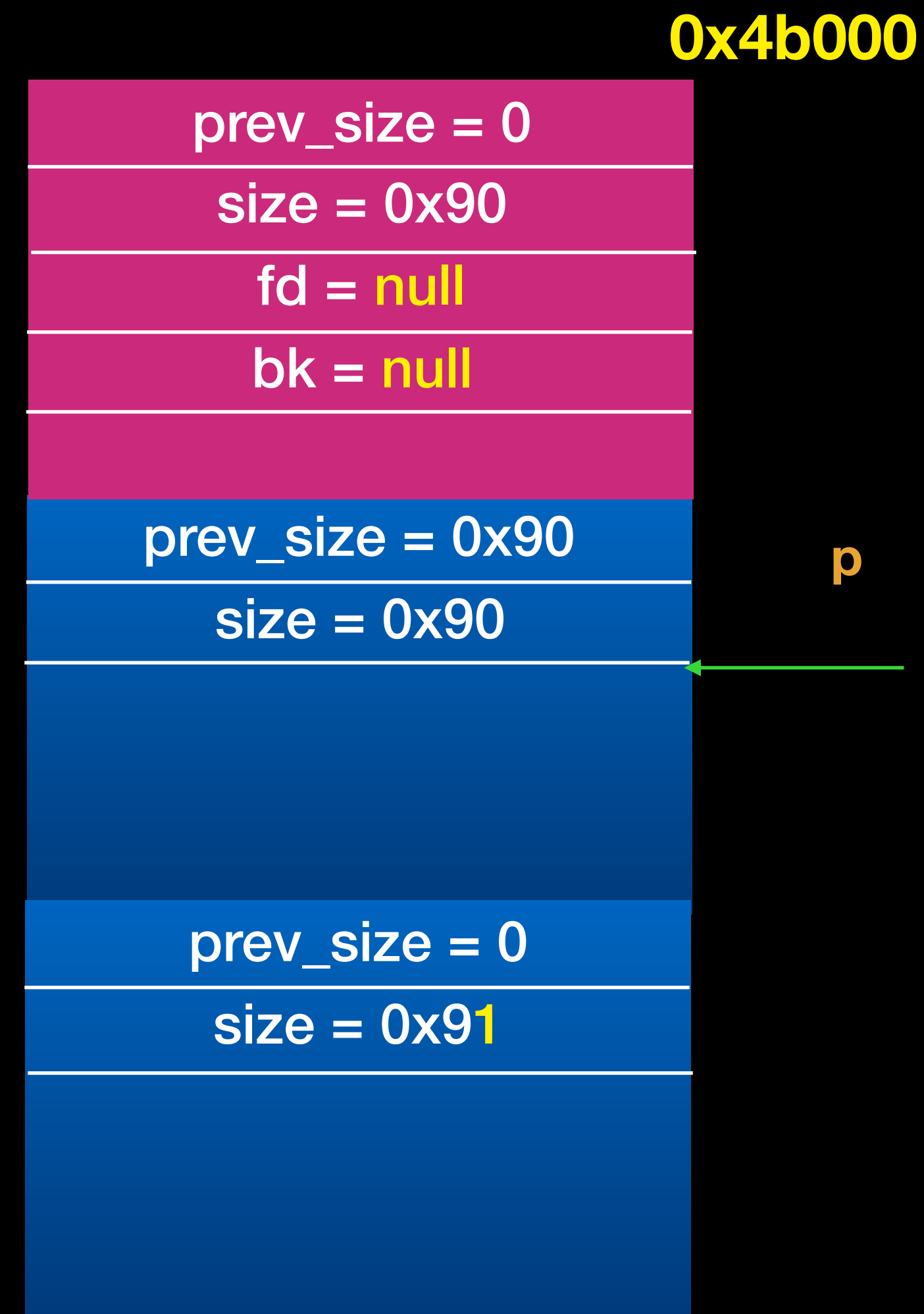
進行 unlink
將上一塊 chunk
從 bin 中移除

small bin

&smallbin
&smallbinm

unsorted bin

&unsortbin
&unsortbin



Mechanism of glibc malloc

merge

small bin

&smallbin

&smallbinm

unsorted bin

&unsortbin

&unsortbin

0x4b000

prev_size = 0

size = 0x120

fd = null

bk = null

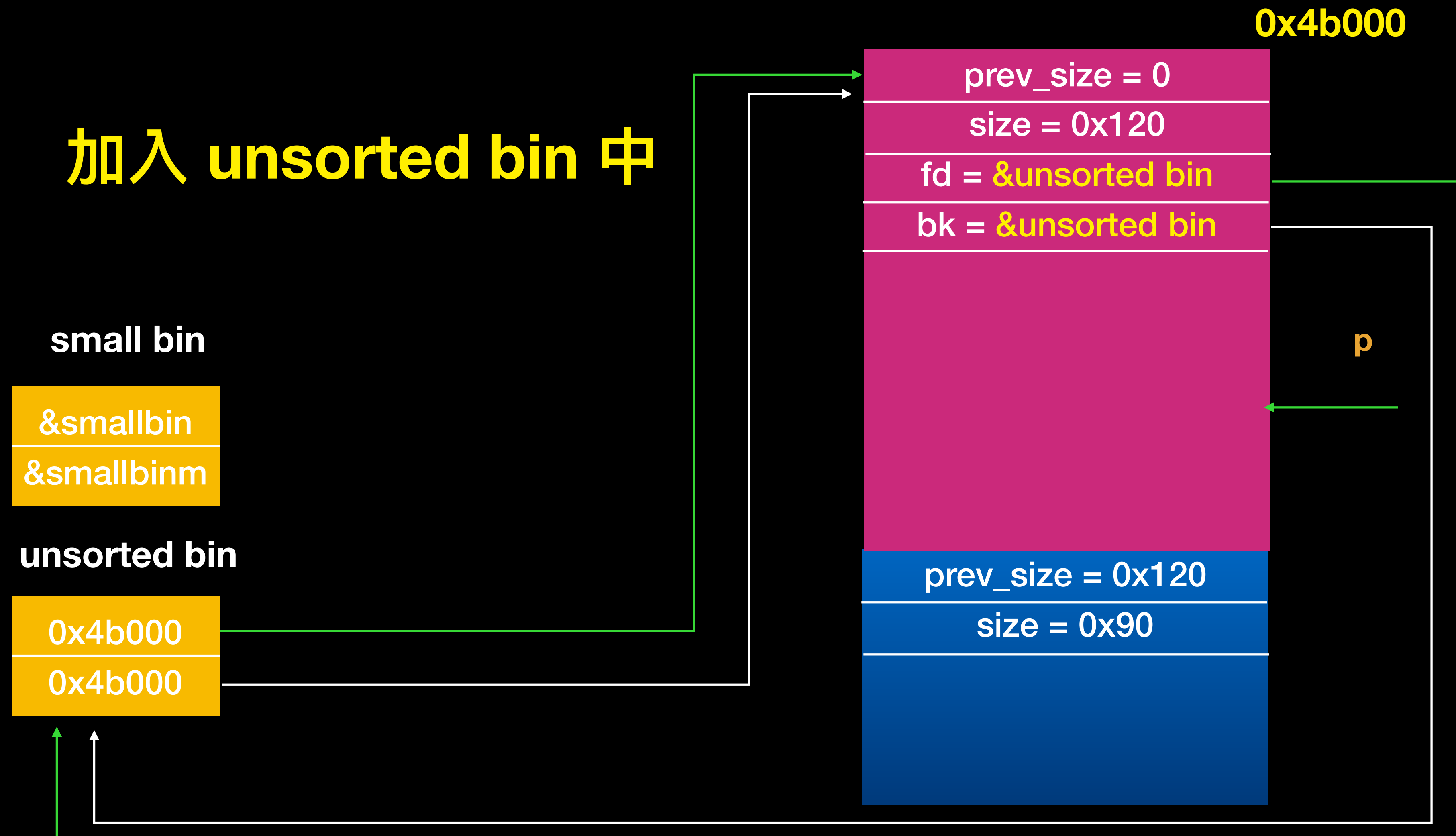
p

prev_size = 0x120

size = 0x90

Mechanism of glibc malloc

加入 **unsorted bin** 中



Practice

- Pwngdb
 - <https://github.com/scwuaptx/Pwngdb>
- 觀察不同 malloc 大小時 chunk 被放入的位置
- 觀察 merge 的現象

Outline

- Glibc memory allocator Overview
- Use After Free
- Heap Overflow
- Appendix - Detection in Glibc

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

```
1 #include <stdio.h>
2
3 int main(){
4     char *p = (char *)malloc(100);
5     free(p);
6     read(0, p, 100);
7 }
```

Use after free

- Assume there exists a **dangling ptr**
 - p is a pointer of movement
 - **free(p)**

```
struct stu
{
    long stu_id ;
    name[24];
}

struct movement
{
    void (*playctf)();
    void (*playball)();
    void (*playgame)();
    char note[8];
}
```

Use after free

- Assume there exists 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
{
    long stu_id ;
    name[24];
}

struct movement
{
    void (*playctf)();
    void (*playball)();
    void (*playgame)();
    char note[8];
}
```

Use after free

- Assume there exists 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 = 0x616161616161**

```
struct stu
{
    long stu_id ;
    name[24];
}

struct movement
{
    void (*playctf)();
    void (*playball)();
    void (*playgame)();
    char note[8];
}
```

Use after free

- Assume there exists 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 = 0x616161616161

- **p.playctf()**

```
struct stu
{
    long stu_id ;
    name[24];
}
```

```
struct movement
{
    void (*playctf)();
    void (*playball)();
    void (*playgame)();
    char note[8];
}
```


Use after free

- Assume there exists 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 = 0x616161616161
 - p.playctf()  **rip = 0x616161616161**

```
struct stu
{
    long stu_id ;
    name[24];
}

struct movement
{
    void (*playctf)();
    void (*playball)();
    void (*playgame)();
    char note[8];
}
```


Practice

- `hacknote`

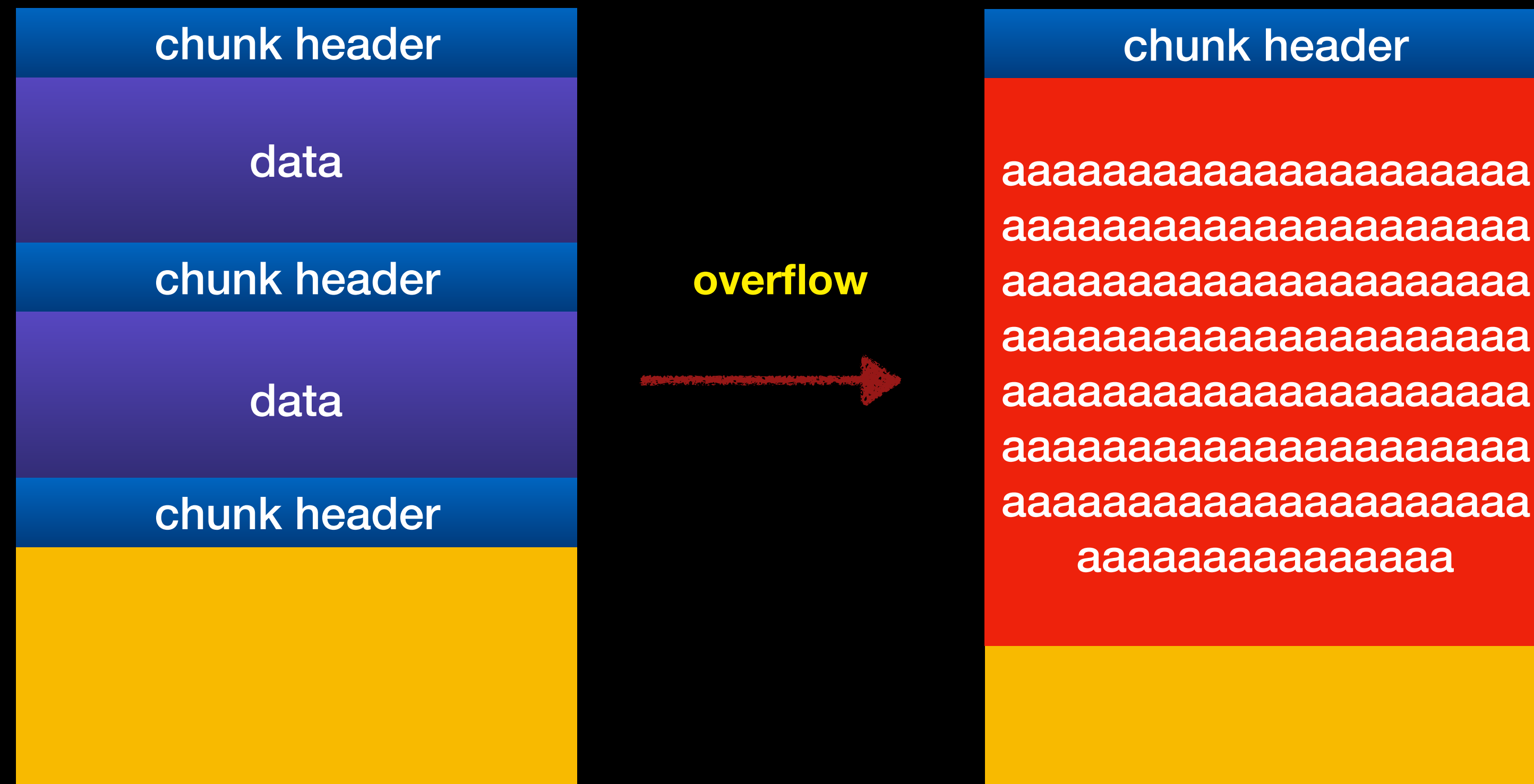
Outline

- Glibc memory allocator Overview
- Use After Free
- Heap Overflow
- Appendix - Detection in Glibc

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 malloc 時或 free 的造成的行為來間接達成任意位置寫入，進而控制 eip

Heap overflow



Heap overflow

- Unlink
- Malloc maleficarum
- Overwrite Fastbin

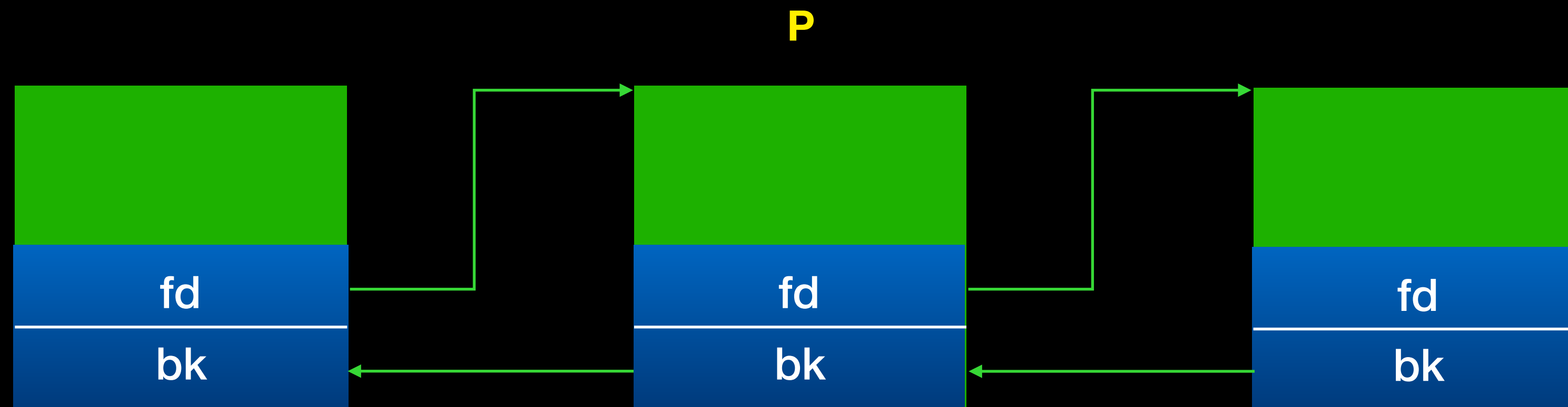
Heap overflow

- 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

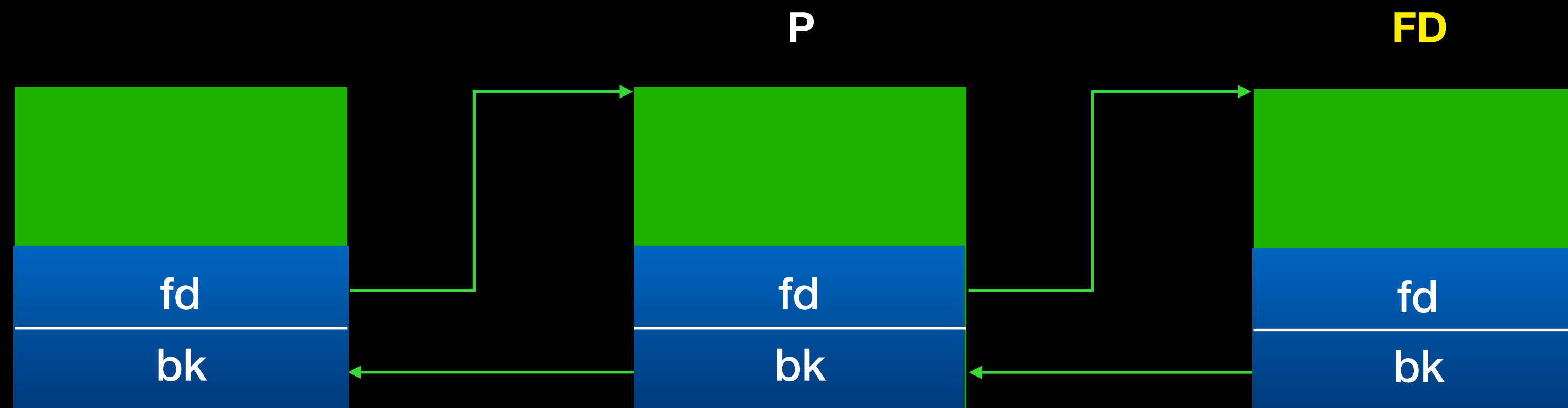


unlink(P,BK,FD)

```
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

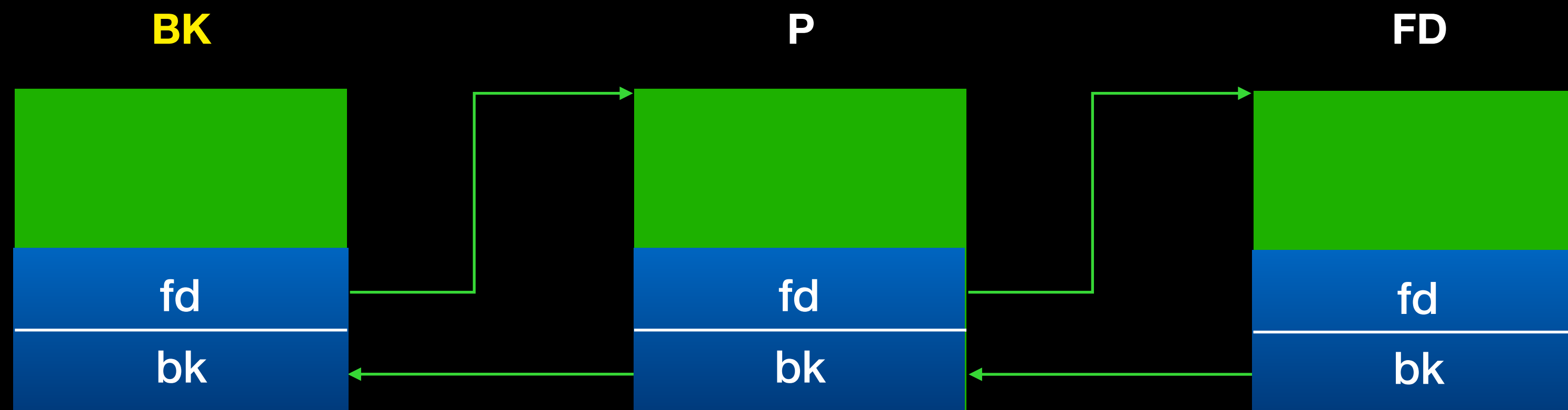


FD = P->fd

```
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

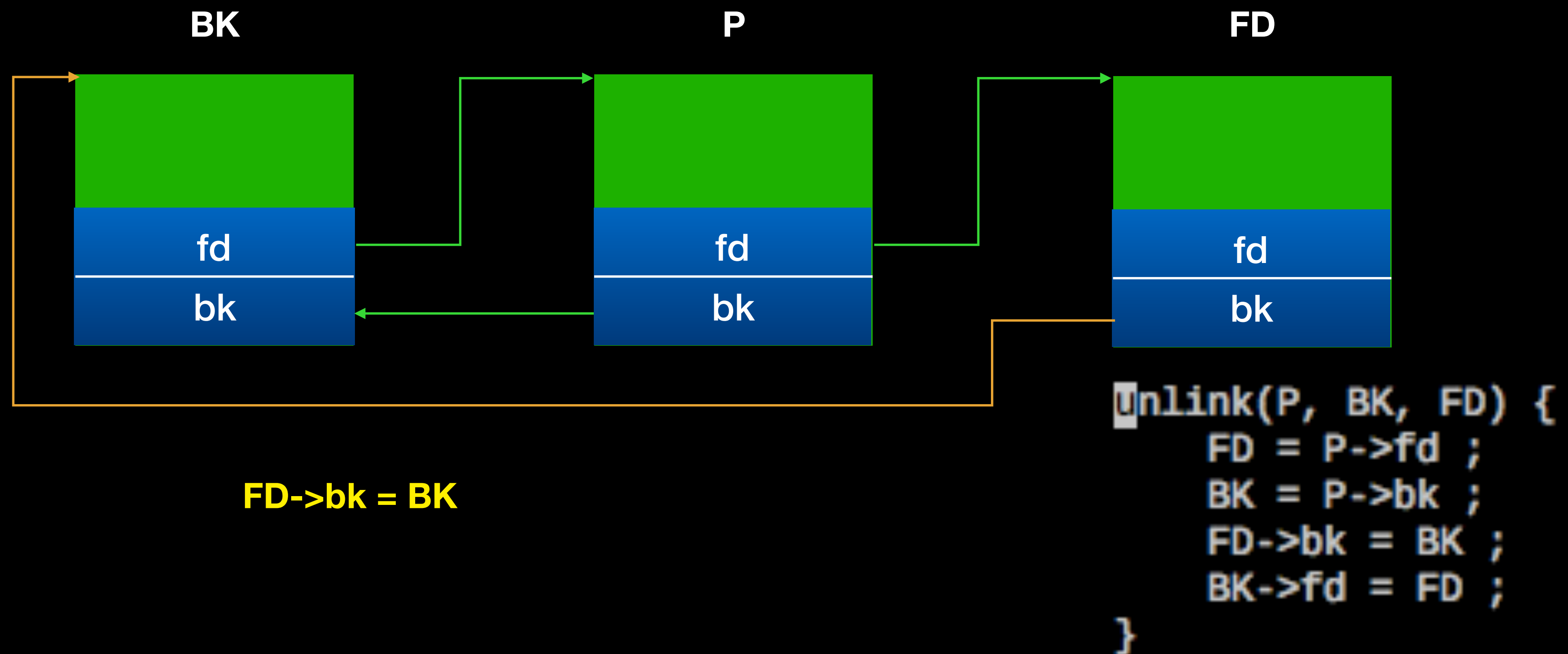


BK = P->bk

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

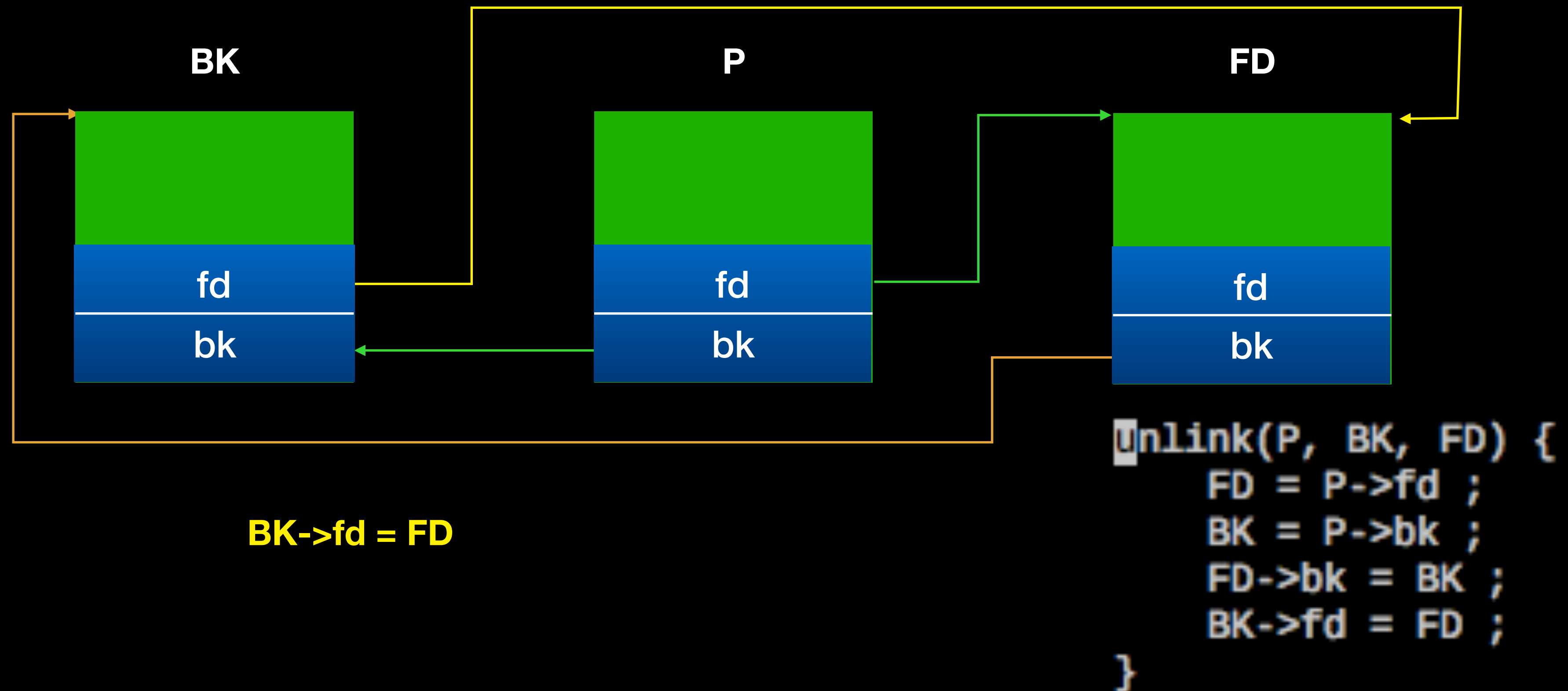
Heap overflow

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



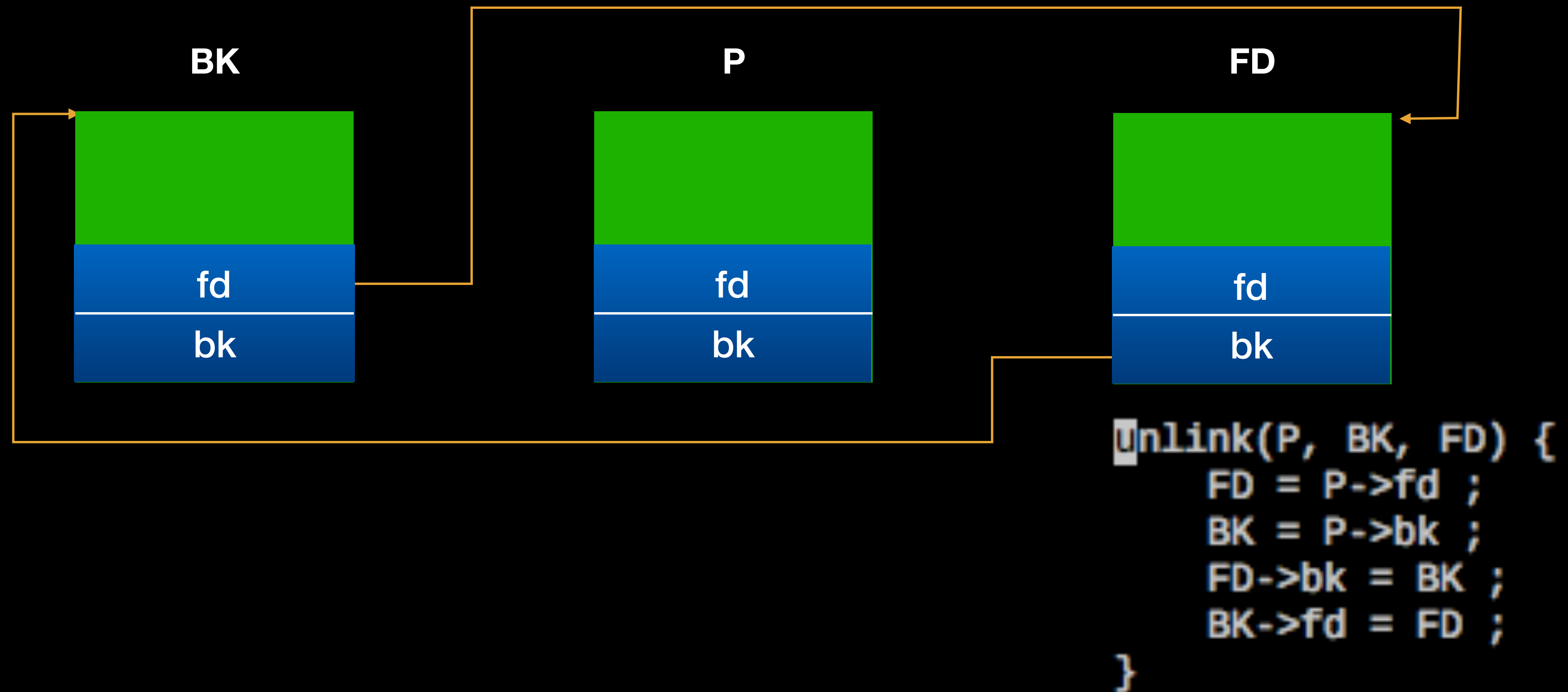
Heap overflow

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



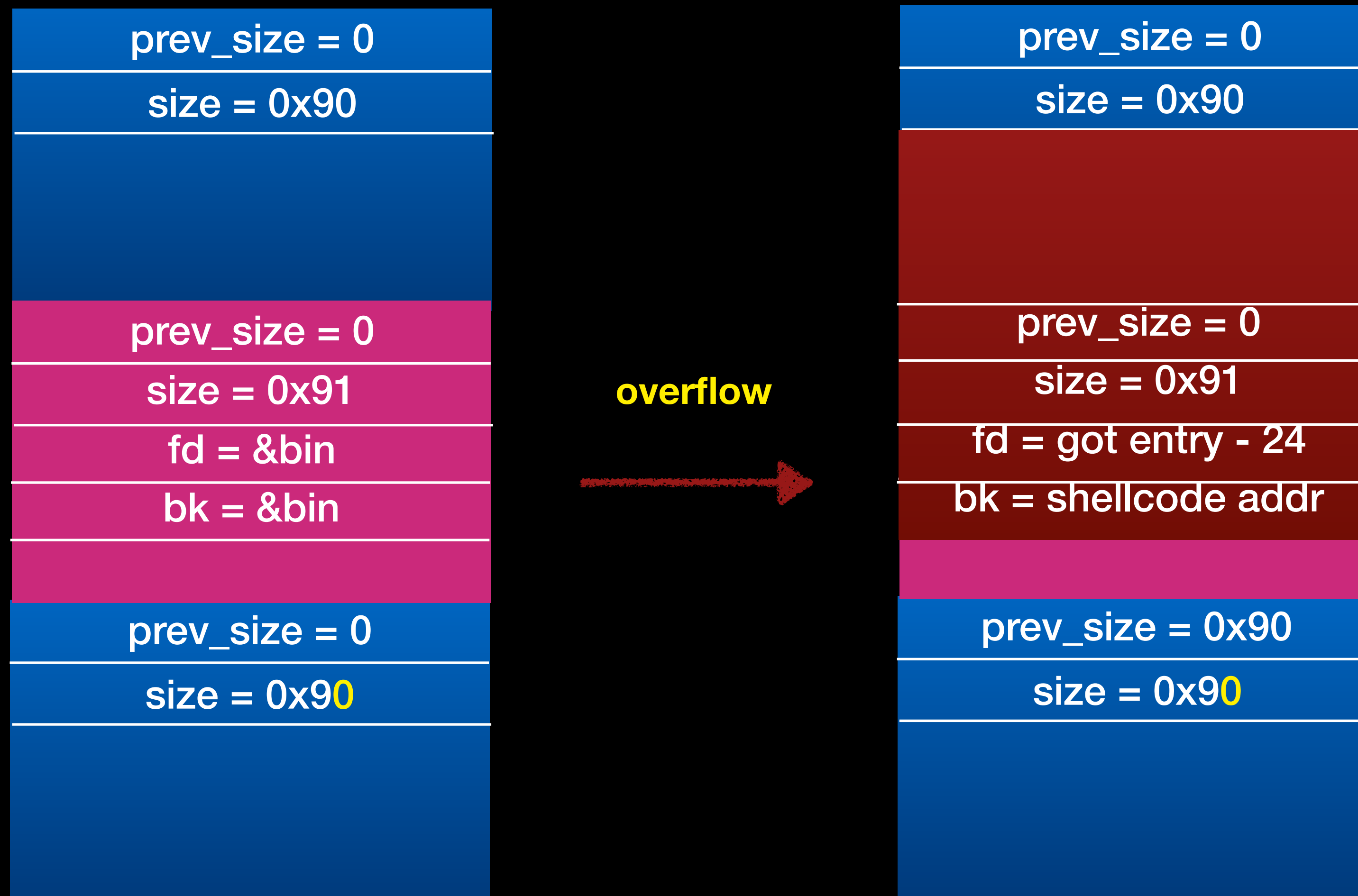
Heap overflow

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



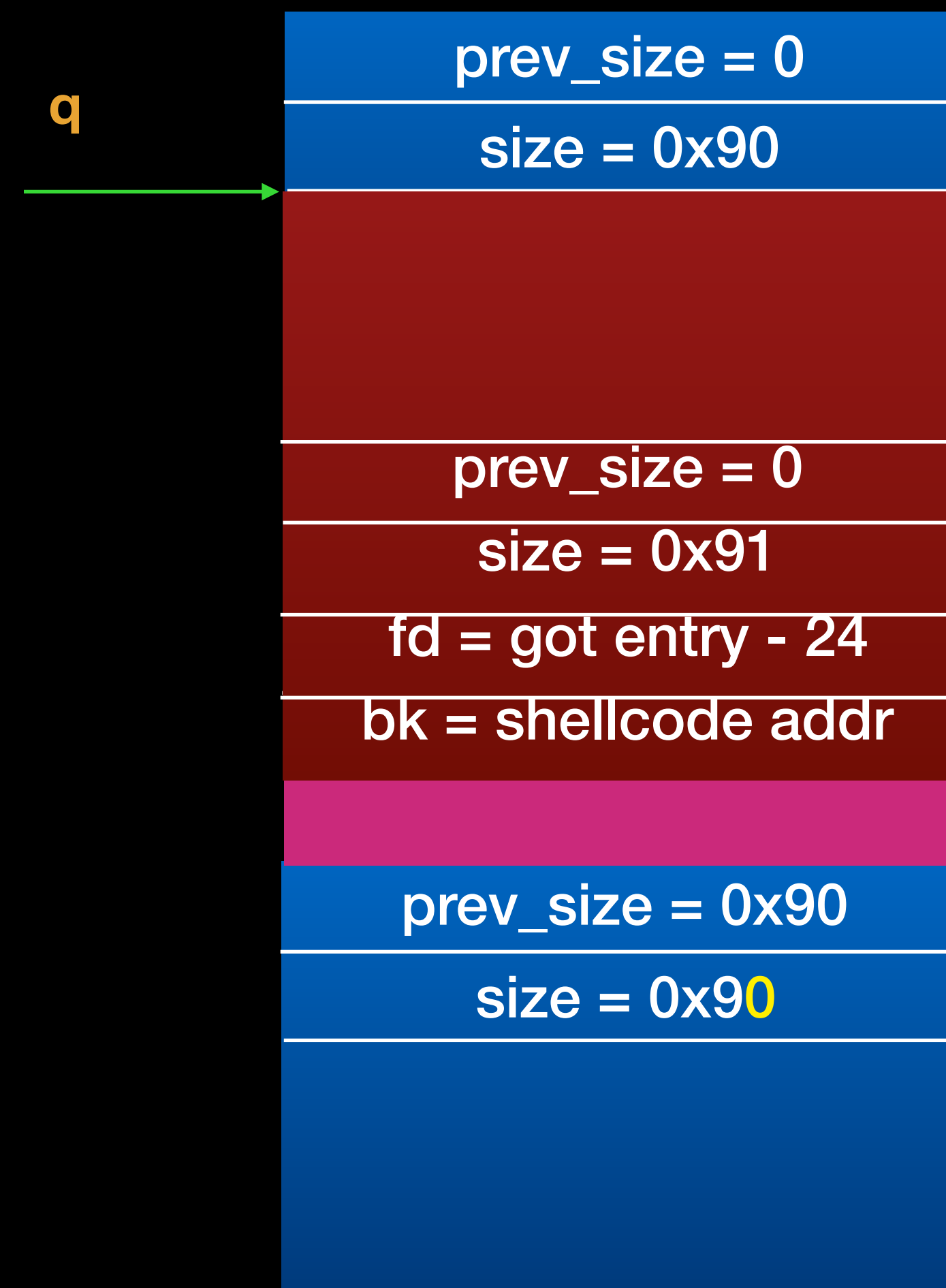
Heap overflow

- using unlink



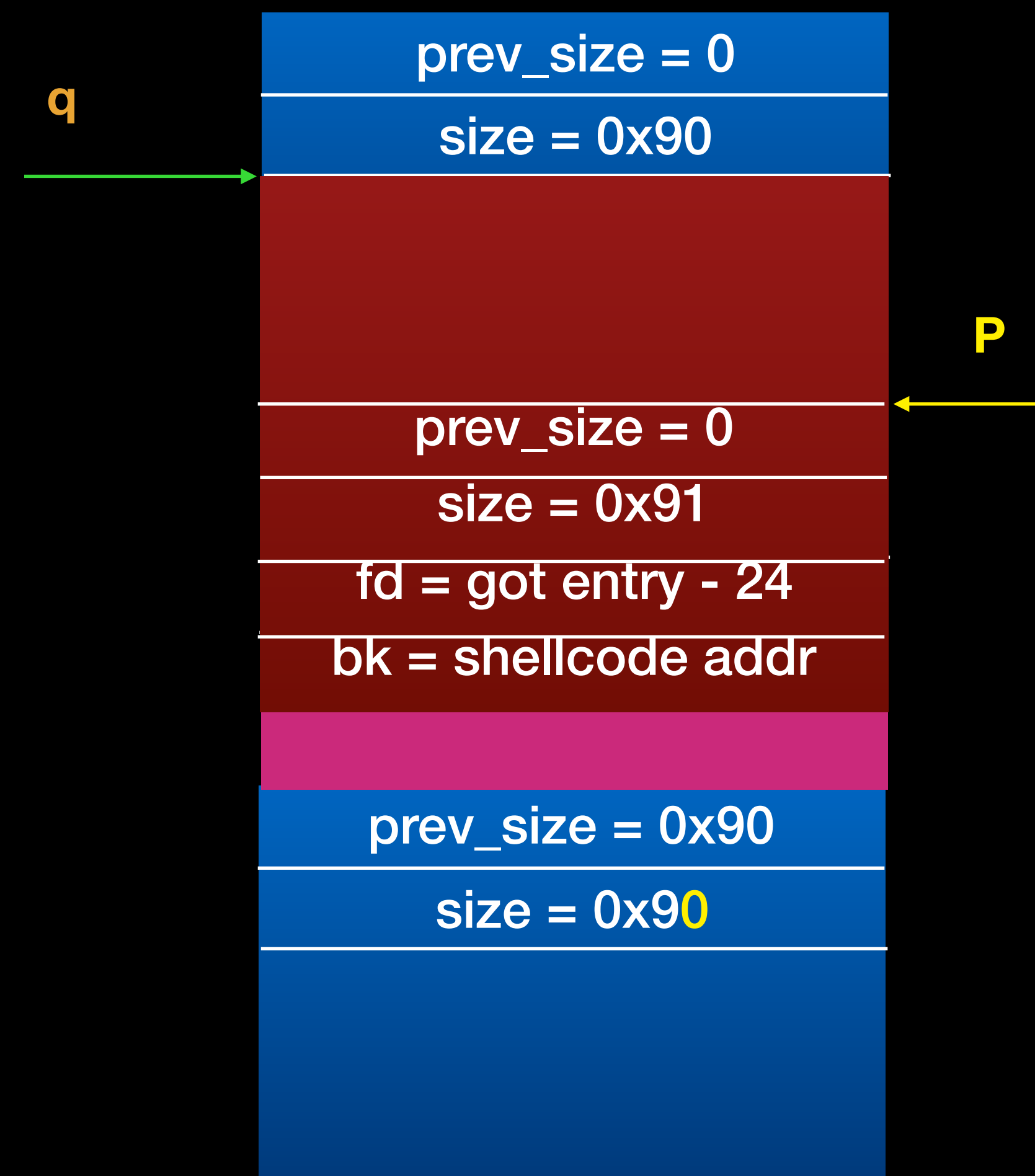
Heap overflow

- using unlink
- free(q)



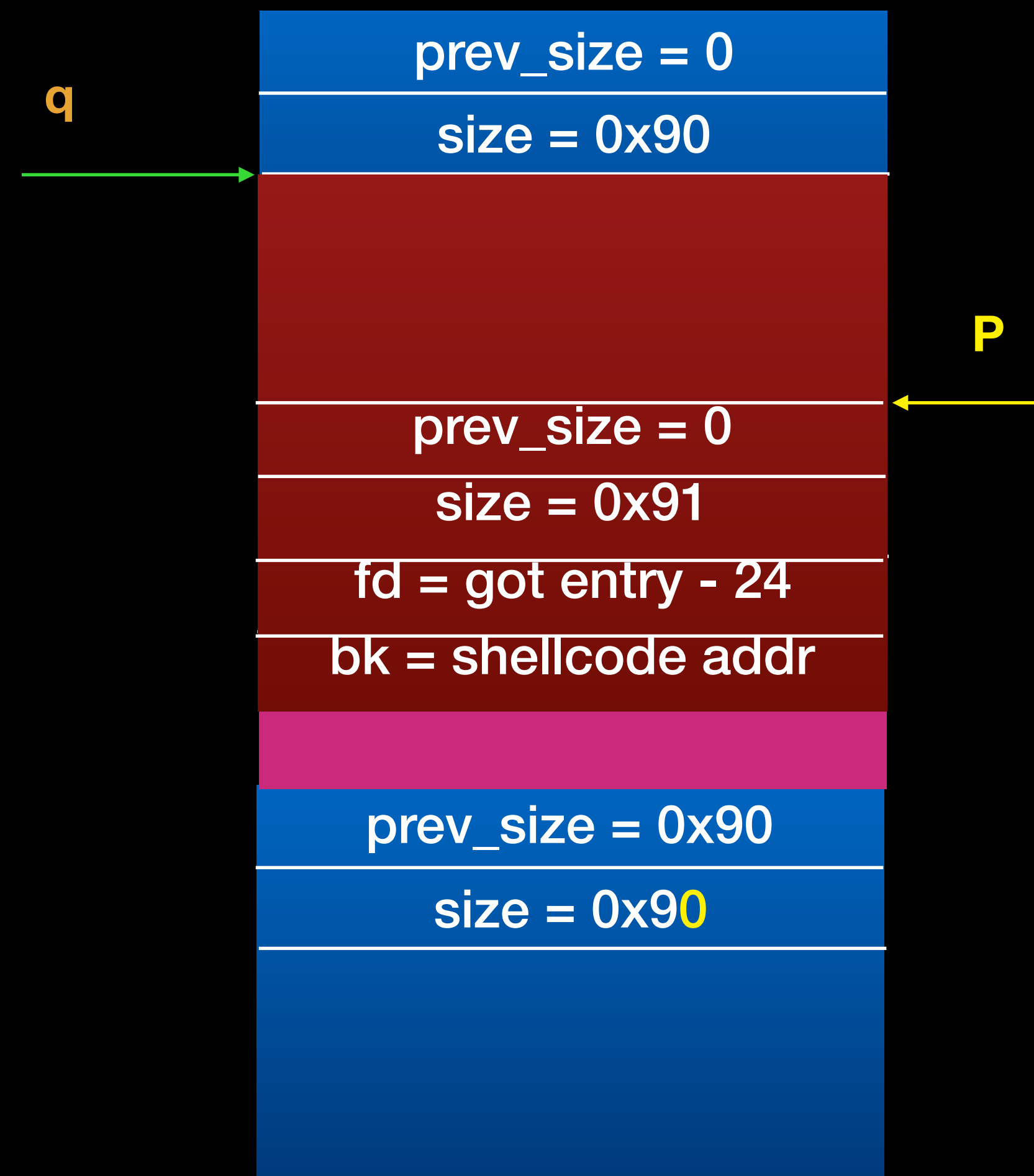
Heap overflow

- using unlink
- free(q)
- $FD = P \rightarrow fd = \text{got entry} - 24$
- $BK = P \rightarrow bk = \text{sc addr}$



Heap overflow

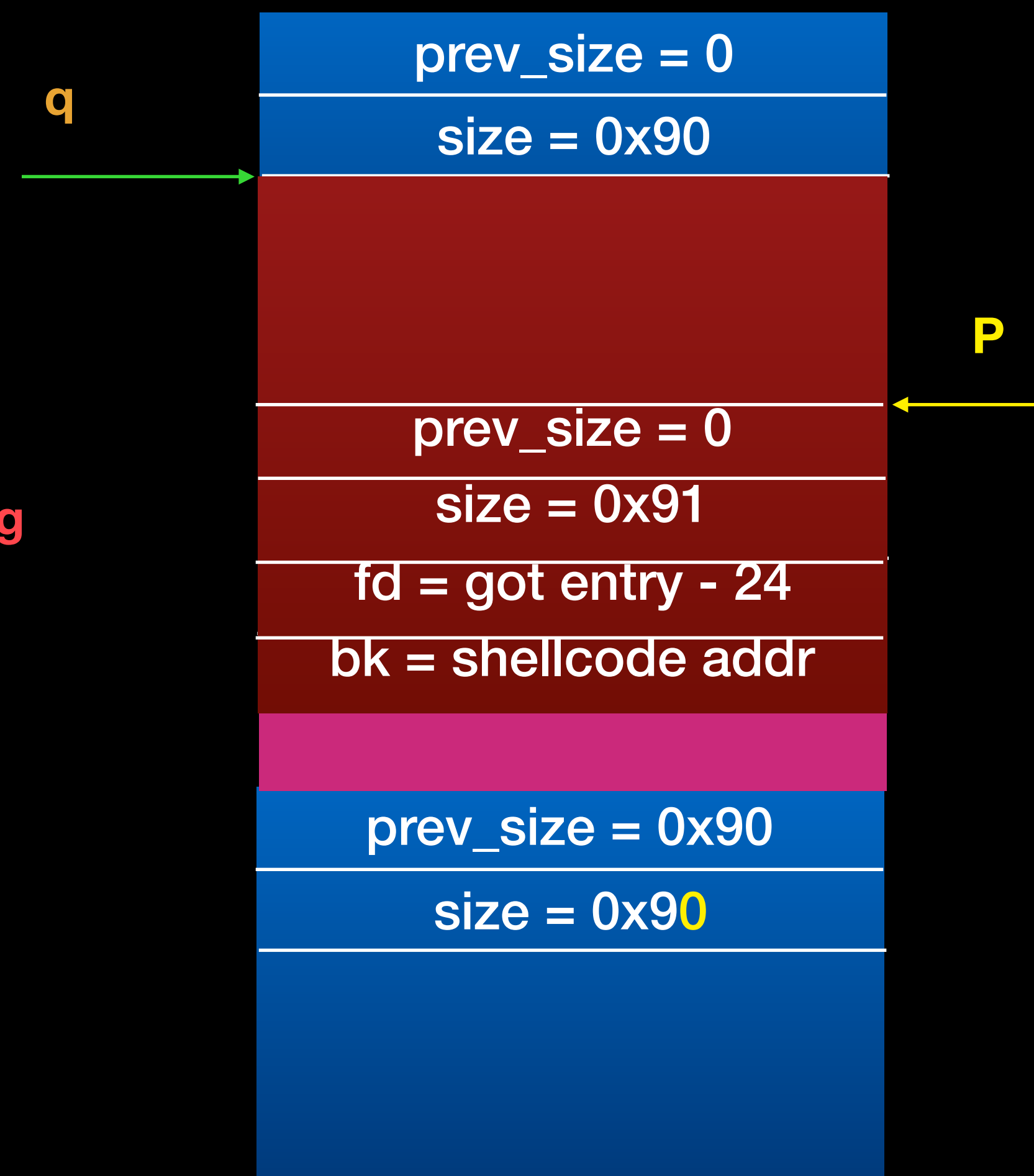
- using unlink
 - free(q)
 - $FD = P \rightarrow fd = \text{got entry} - 24$
 - $BK = P \rightarrow bk = \text{sc addr}$
 - $FD \rightarrow bk = BK$
 - $\text{got entry} - 24 + 24 = \text{sc addr}$
 - $BK \rightarrow fd = FD$
 - $\text{sc addr} + 16 = \text{got entry} - 24$



Heap overflow

- using unlink
 - free(q)
 - $FD = P \rightarrow fd = \text{got entry} - 24$
 - $BK = P \rightarrow bk = \text{sc addr}$
 - $FD \rightarrow bk = BK$
 - $\text{got entry} - 24 + 24 = \text{sc addr}$
 - $BK \rightarrow fd = FD$
 - $\text{sc addr} + 16 = \text{got entry} - 24$

GOT hijacking

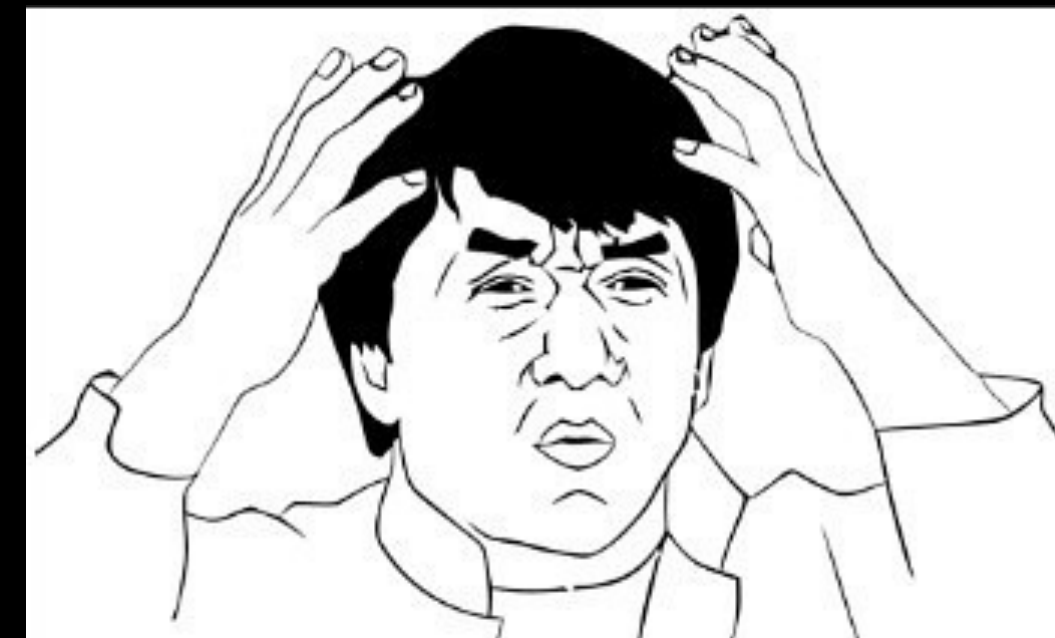


Heap overflow

- using unlink
 - 不過 shellcode 的第16 到第 24 個 byte 會因為 $sc\ addr + 16 = got\ entry - 24$ 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
 - Corrupted size vs. prev_size
 -



Heap overflow

- detection in **unlink**
- **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
 - $P \rightarrow bk \rightarrow fd == P$
 - $P \rightarrow fd \rightarrow bk == P$

Heap overflow

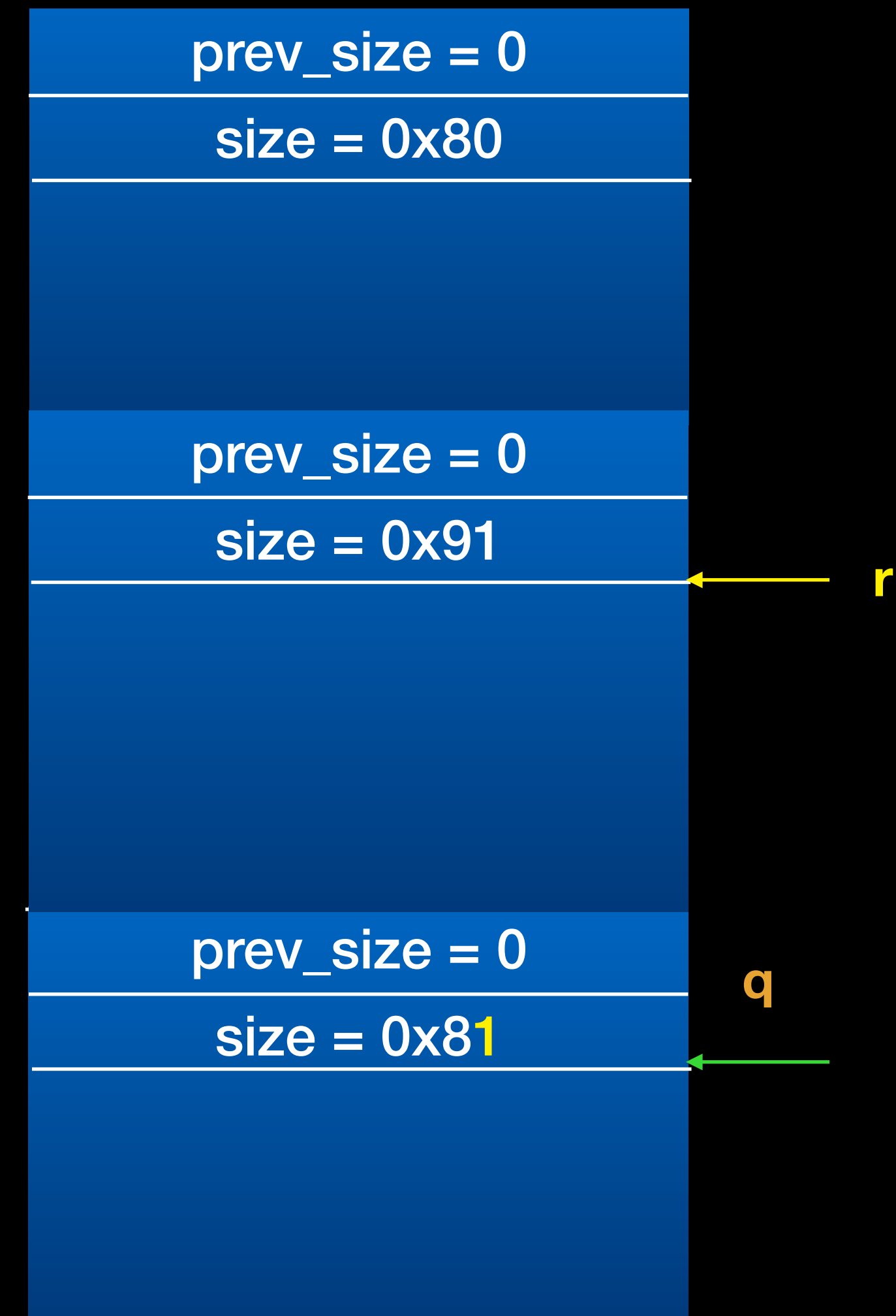
- detection in `unlink`
- Corrupted size vs. prev_size
 - 預防 size 或 prev_size 被竄改
 - `chunksize(P) == next_chunk(P)->prev_size`
 - glibc 2.26 後新增的檢查，ubuntu 16.04 glibc 有 update 所以也會有這樣檢查

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
 - chunk size 及 next_chunk->prev_size 要一起偽造

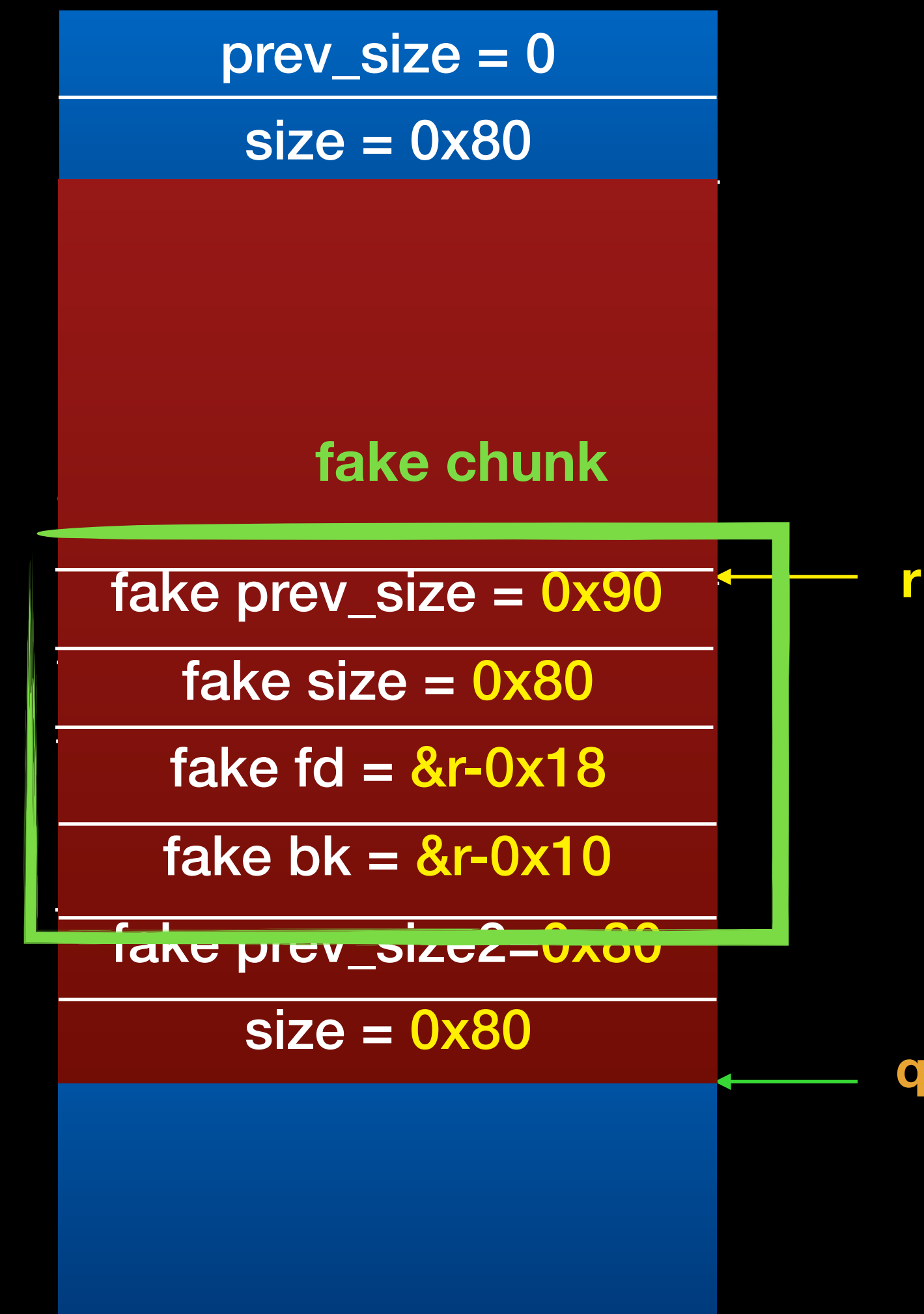
Heap overflow

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



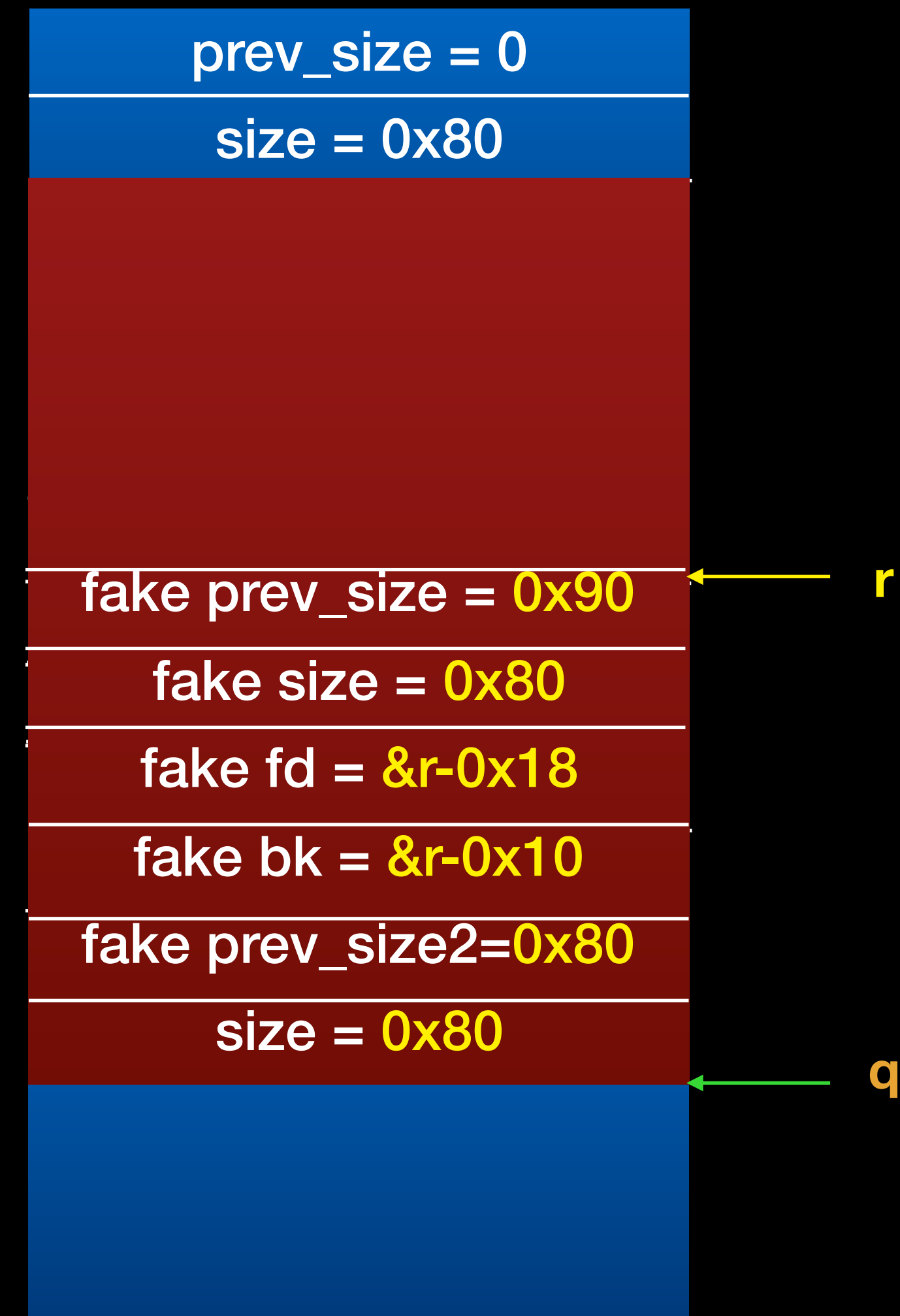
Heap overflow

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



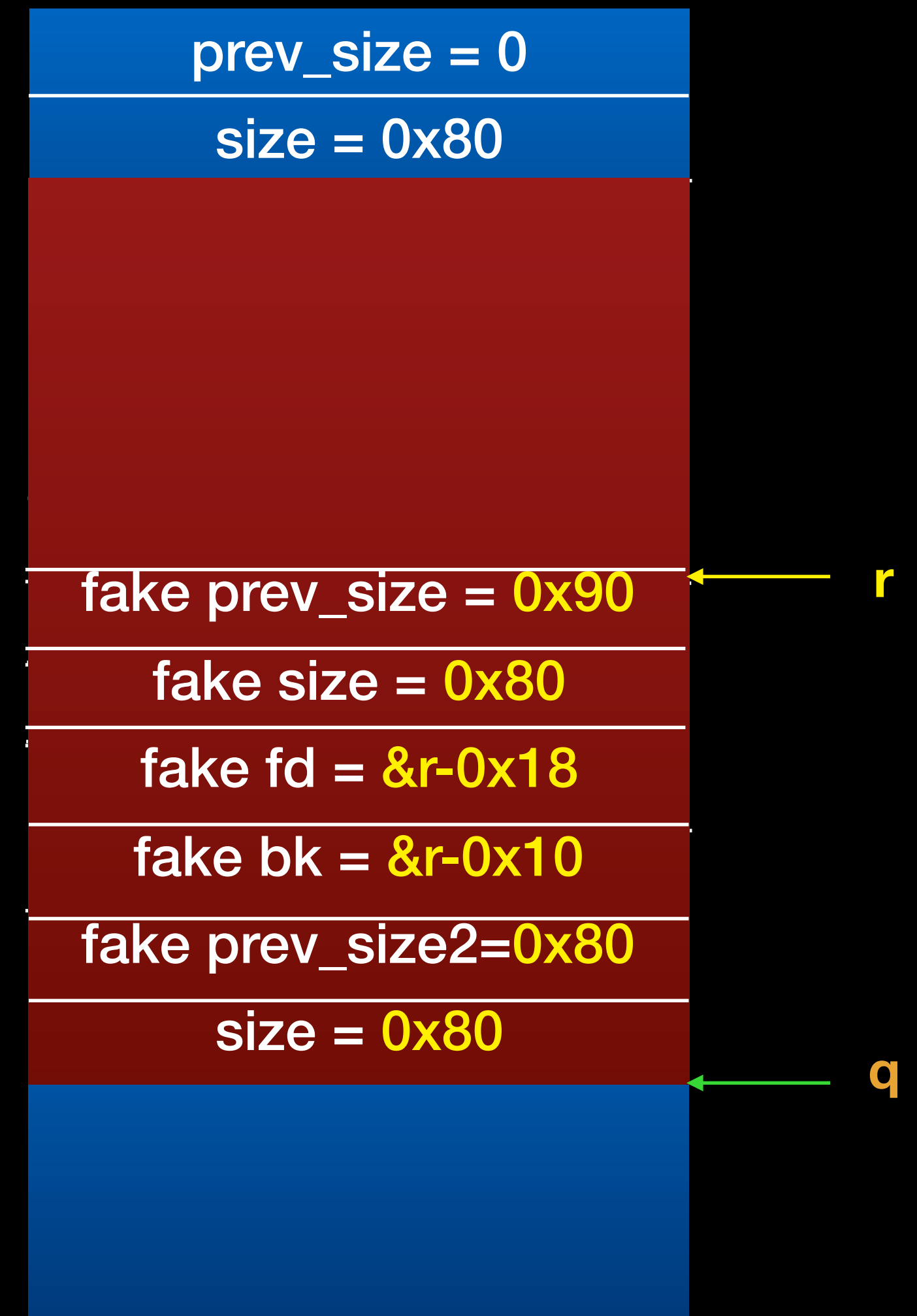
Heap overflow

- 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



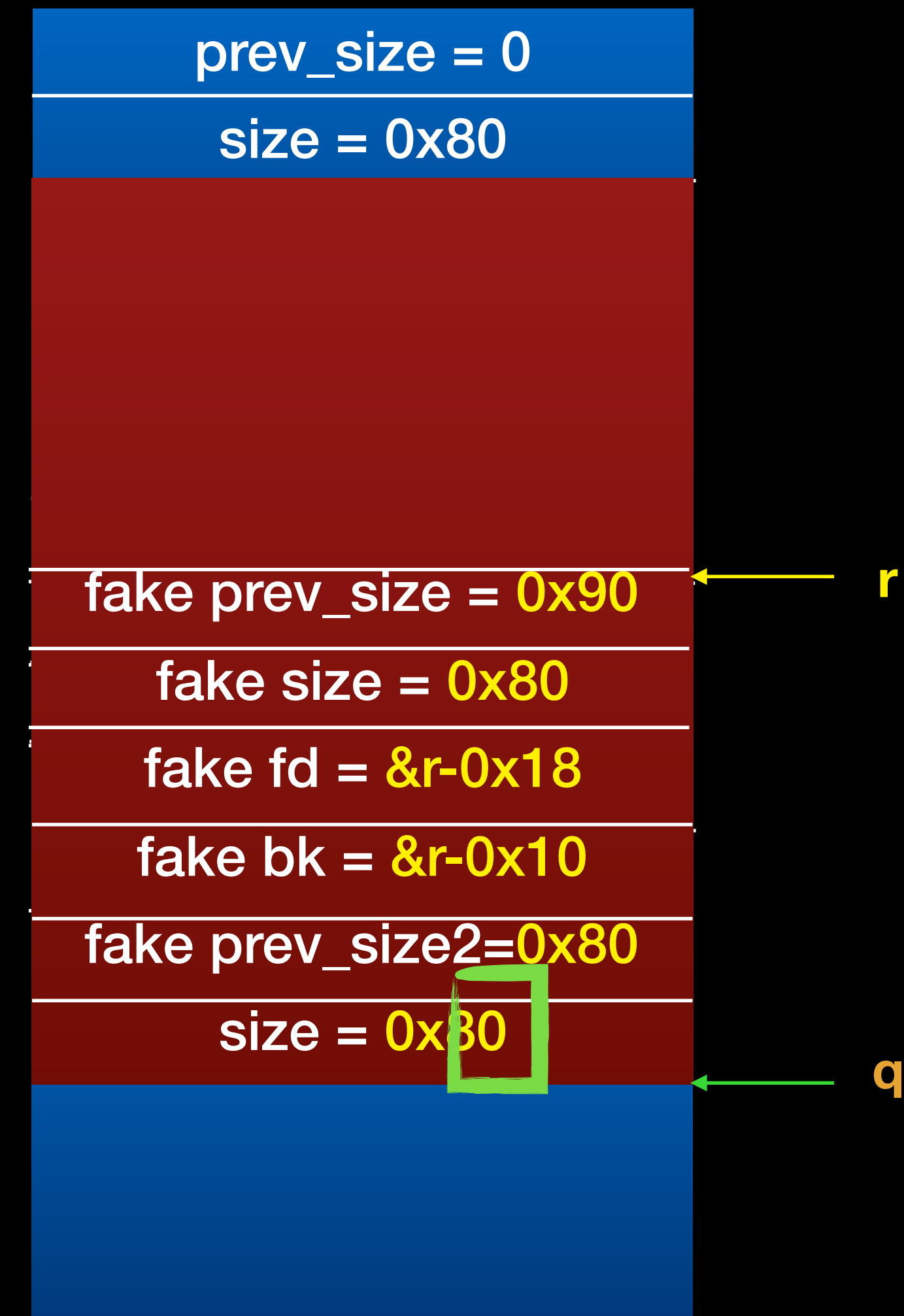
Heap overflow

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



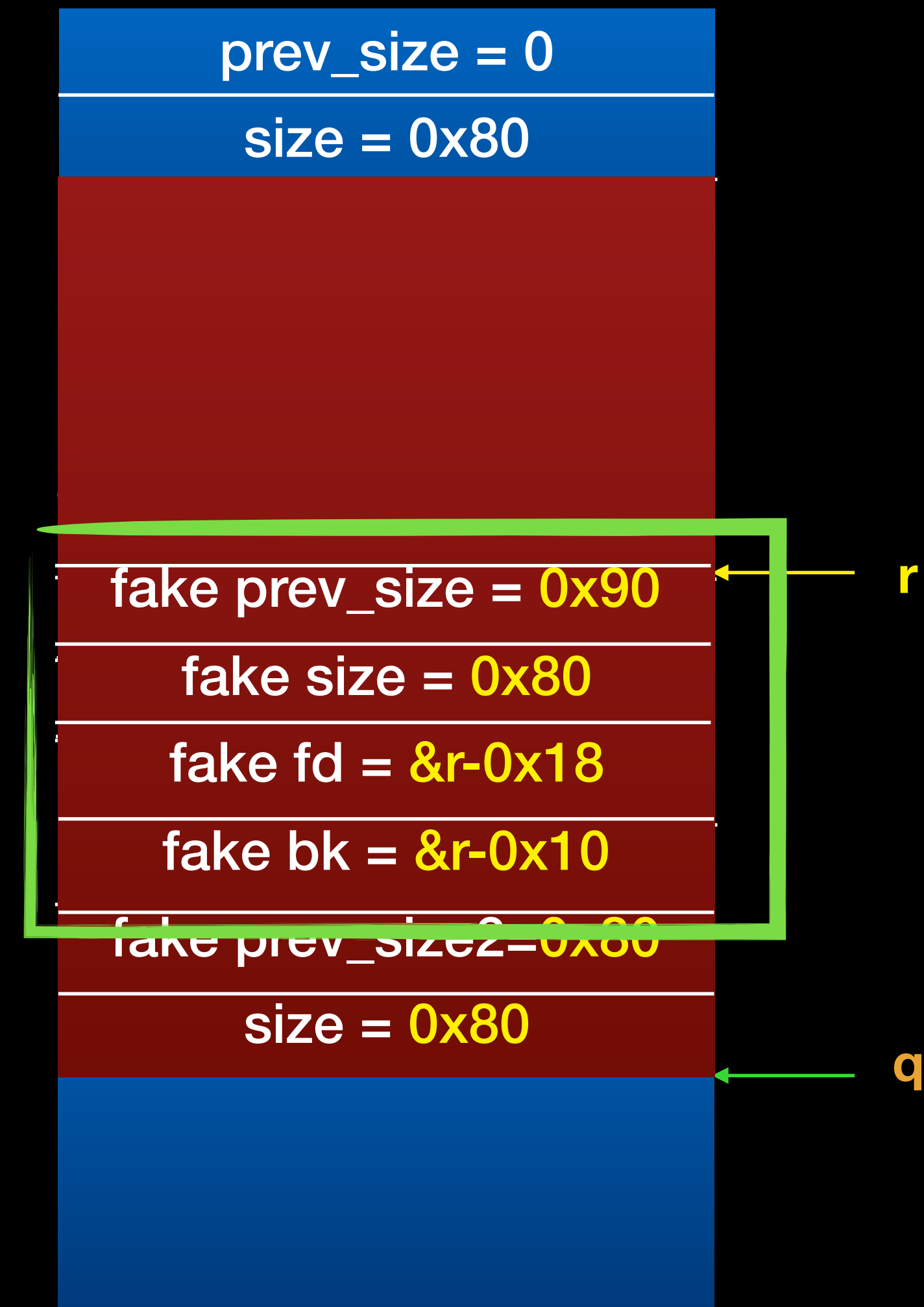
Heap overflow

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



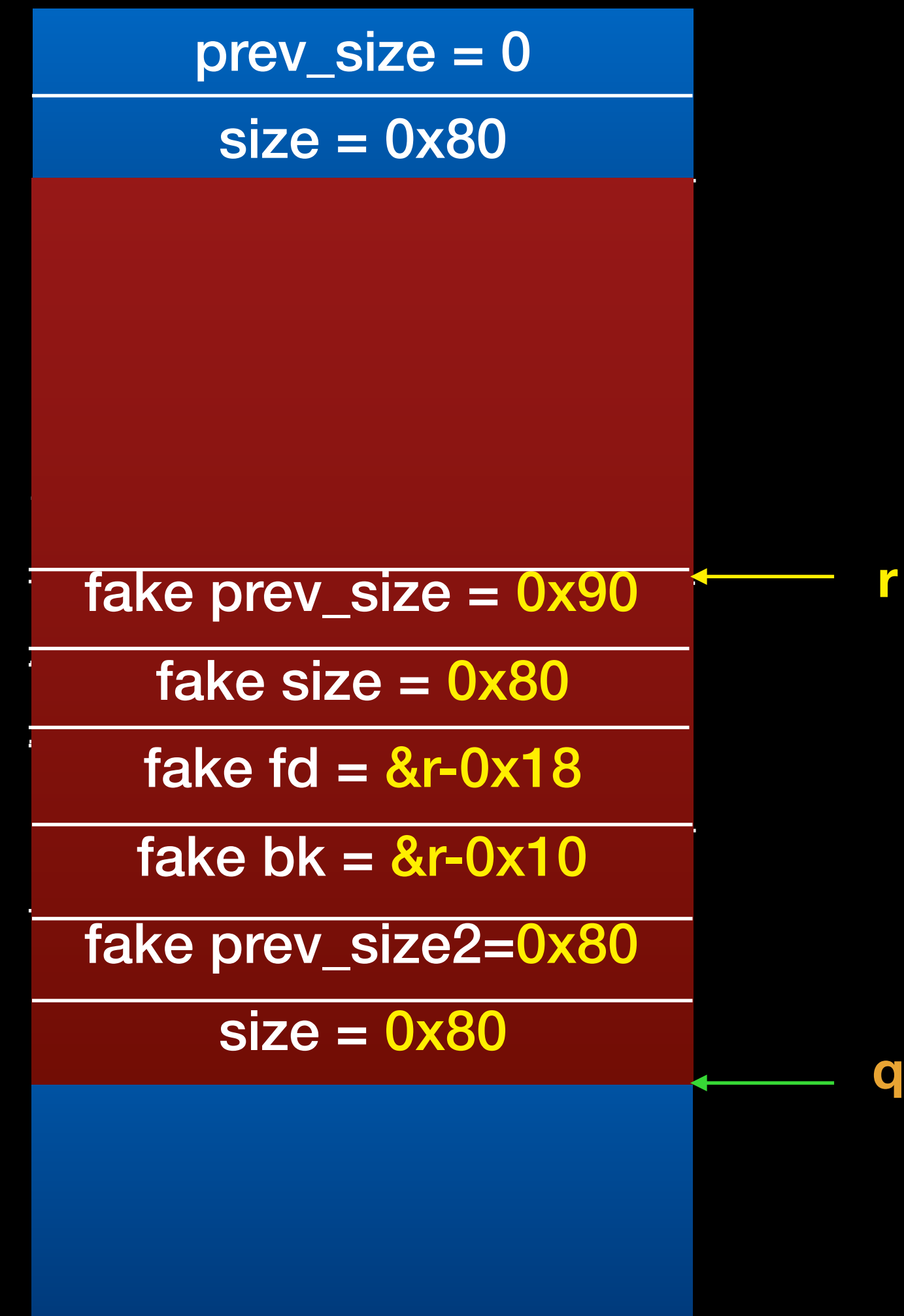
Heap overflow

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



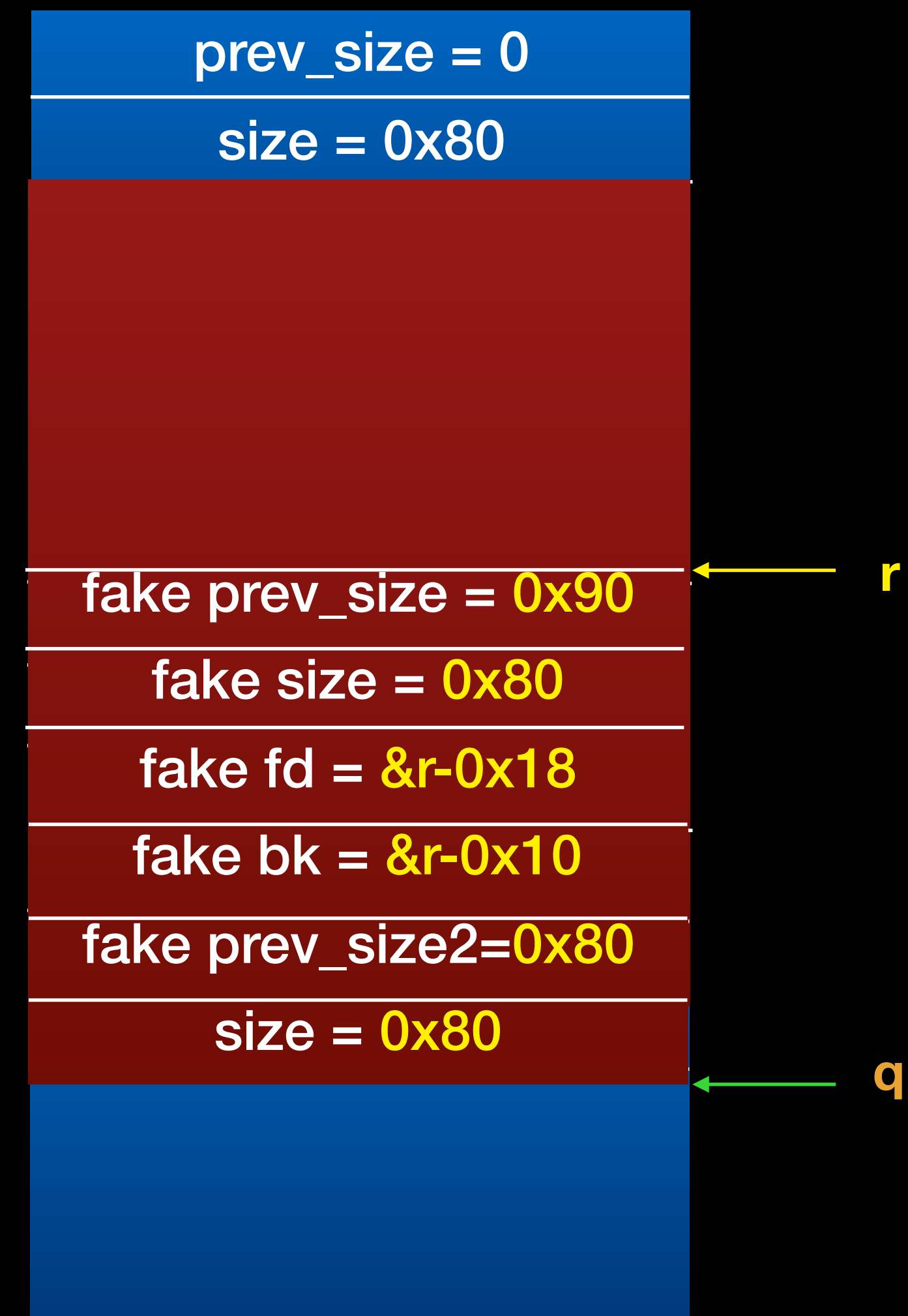
Heap overflow

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



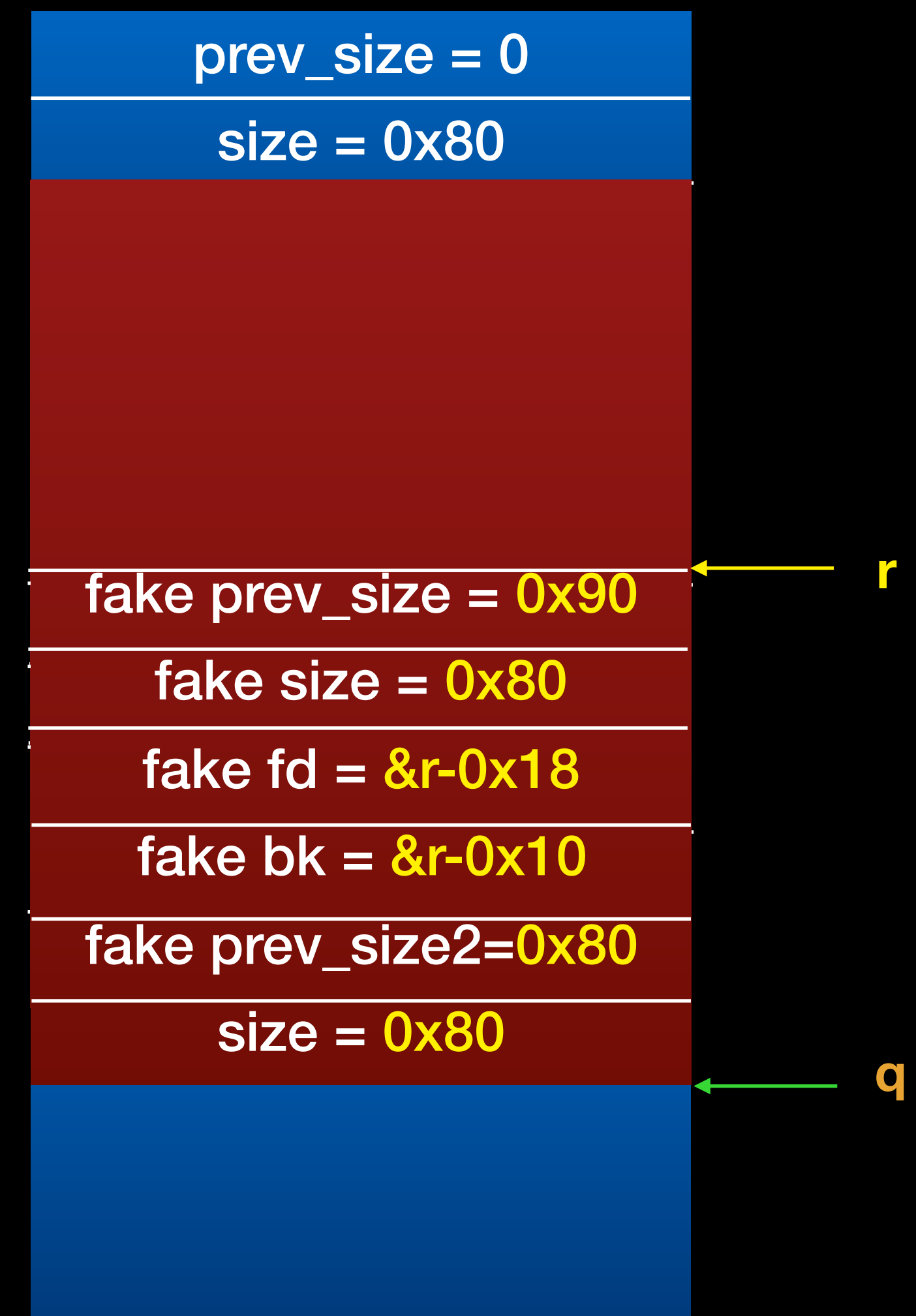
Heap overflow

- using unlink (modern)
 - bypass the detection
 - unlink(r,FD,BK)
 - check
 - $\text{prev_size2} == \text{fake_size} == 0x80$
 - $r \rightarrow \text{fd} \rightarrow \text{bk} == r = *(&r - 0x18 + 0x18) = r$
 - $r \rightarrow \text{bk} \rightarrow \text{fd} == r = *(&r - 0x10 + 0x10) = r$



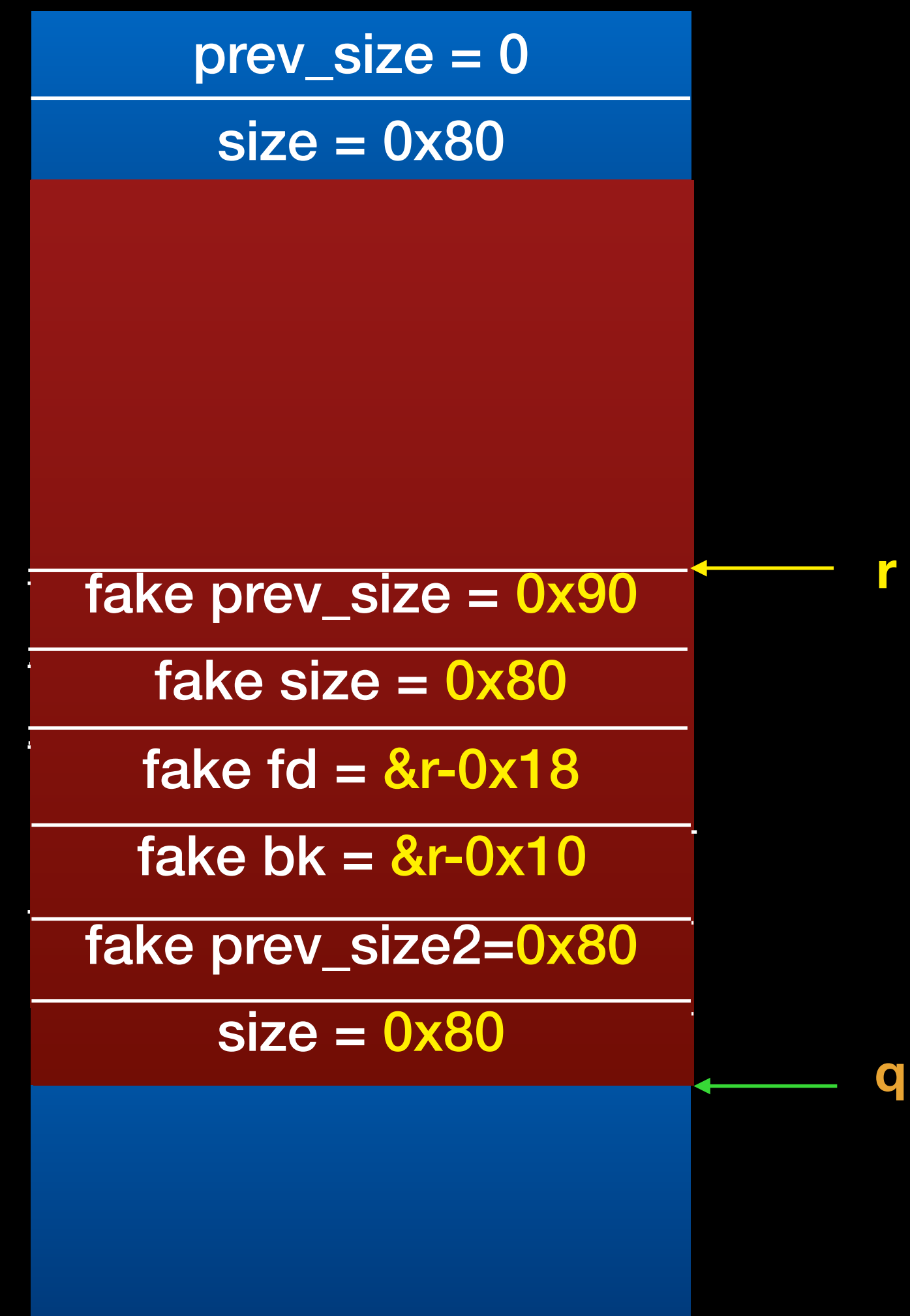
Heap overflow

- using unlink (modern)
 - bypass the detection
 - $FD \rightarrow bk = BK$
 - $\ast(&r - 0x18 + 0x18) = \&r - 0x10$
 - $BK \rightarrow fd = FD$
 - $\ast(&r - 0x10 + 0x10) = \&r - 0x18$



Heap overflow

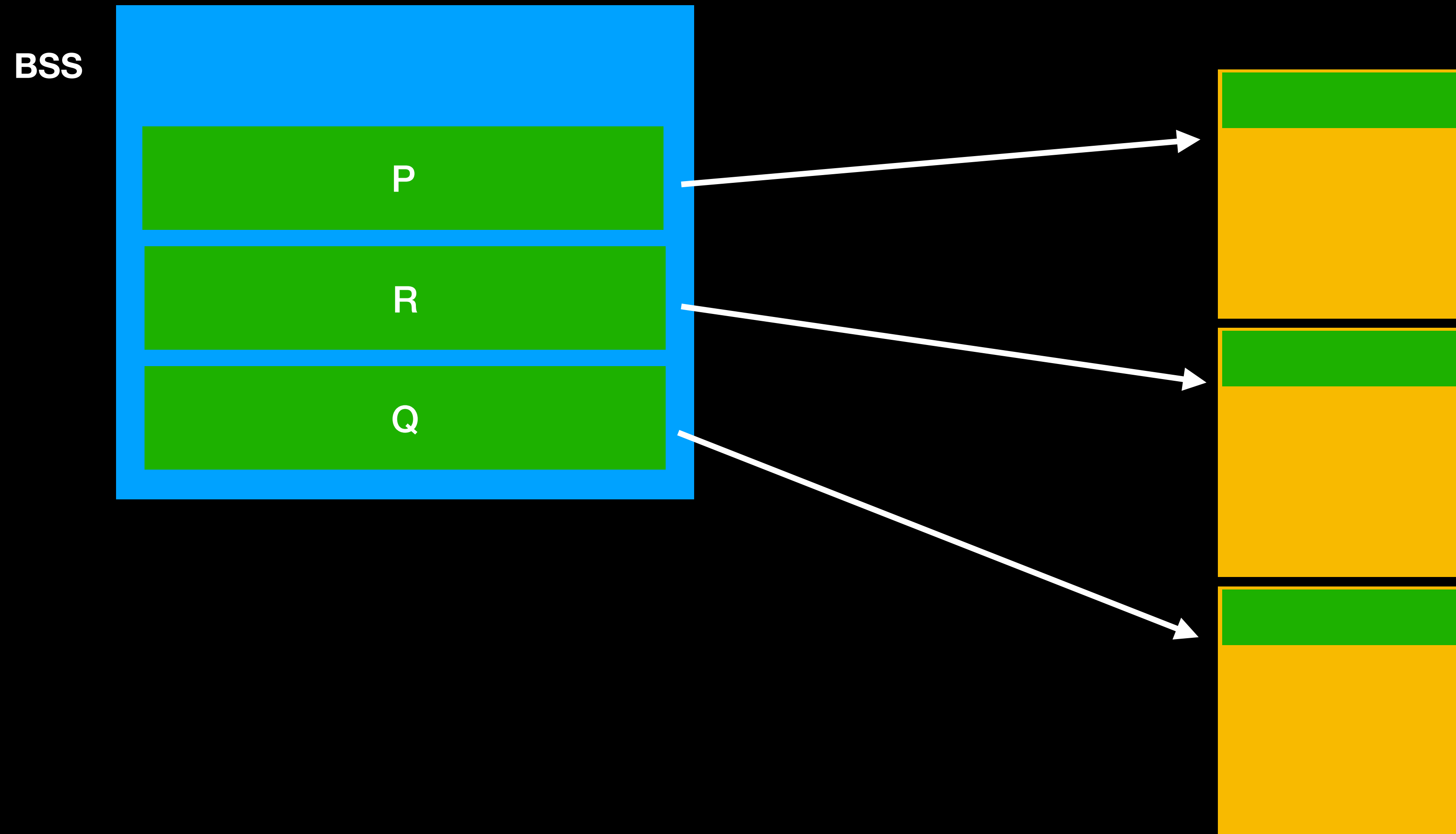
- using unlink (modern)
 - bypass the detection
 - $FD \rightarrow bk = BK$
 - $\ast(\&r - 0x18 + 0x18) = \&r - 0x10$
 - $BK \rightarrow fd = FD$
 - $\ast(\&r - 0x10 + 0x10) = \&r - 0x18$
 - We change the value of $\&r$ successful then we may change other thing in mem.



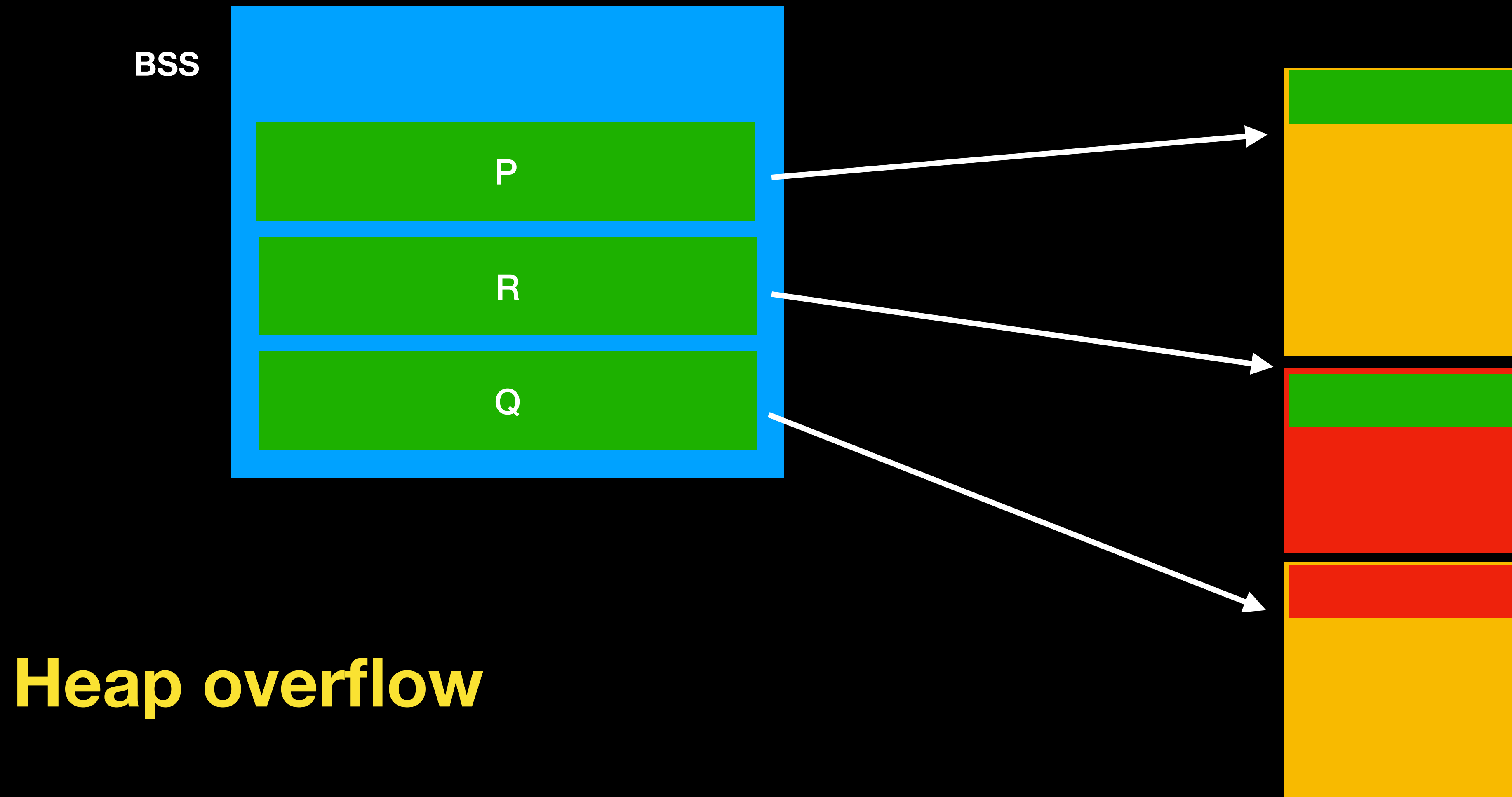
Heap overflow

- 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

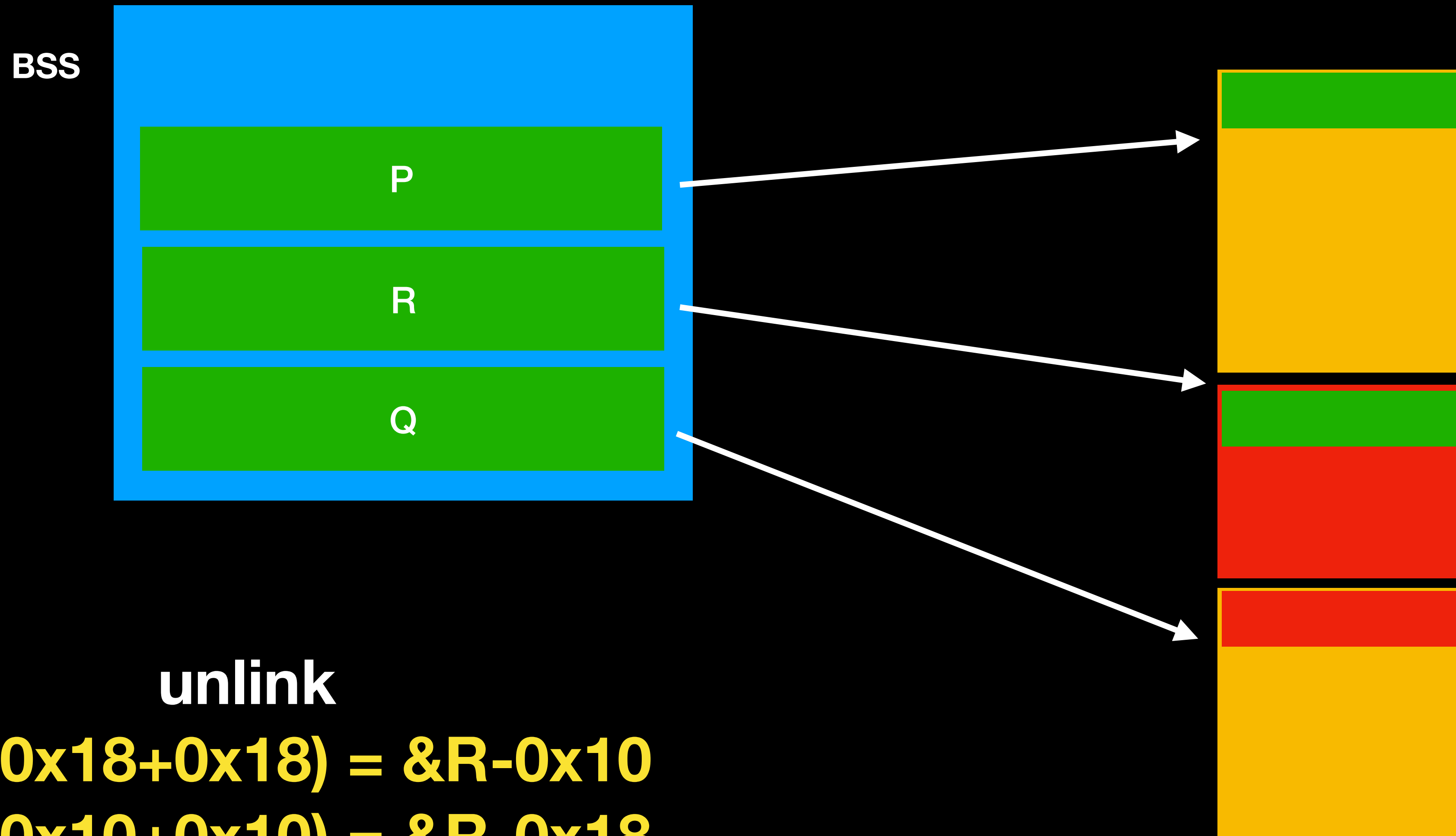
Heap overflow



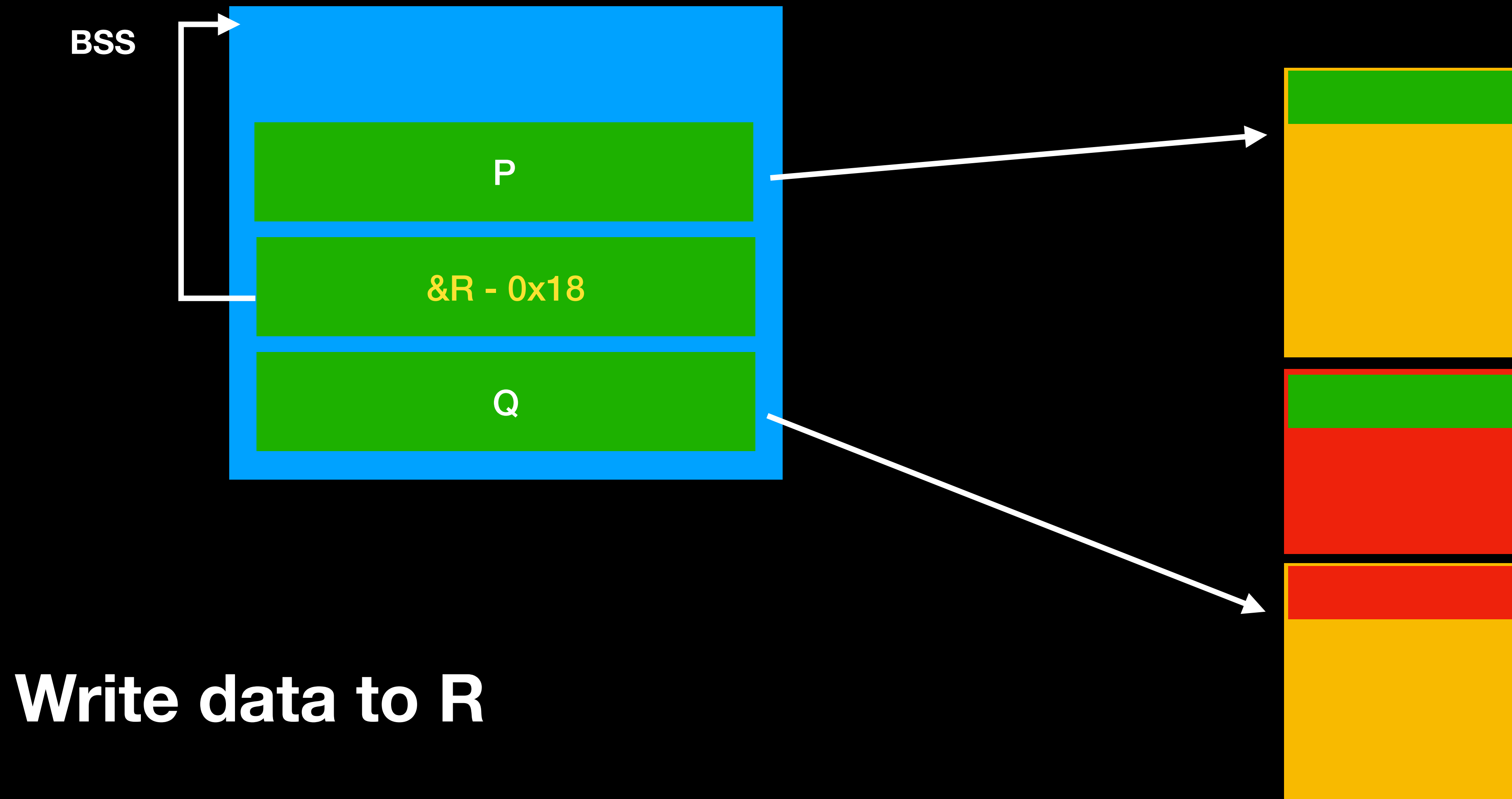
Heap overflow



Heap overflow

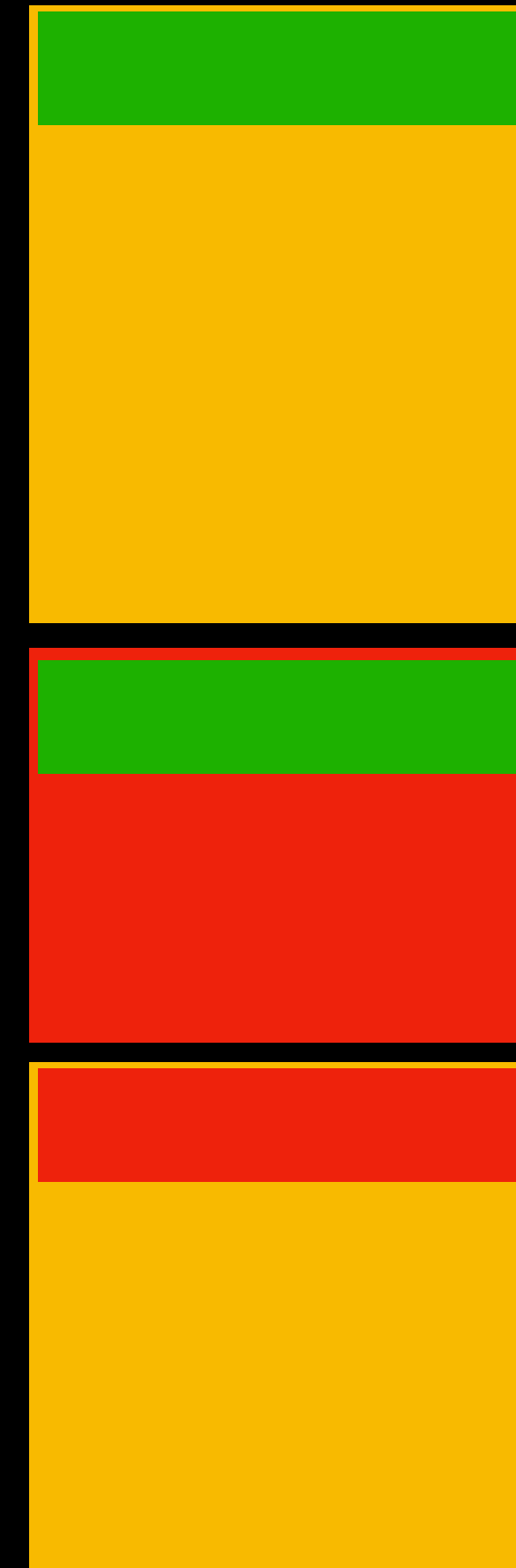


Heap overflow



Heap overflow

BSS



Heap overflow

BSS



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

The House of Spirit

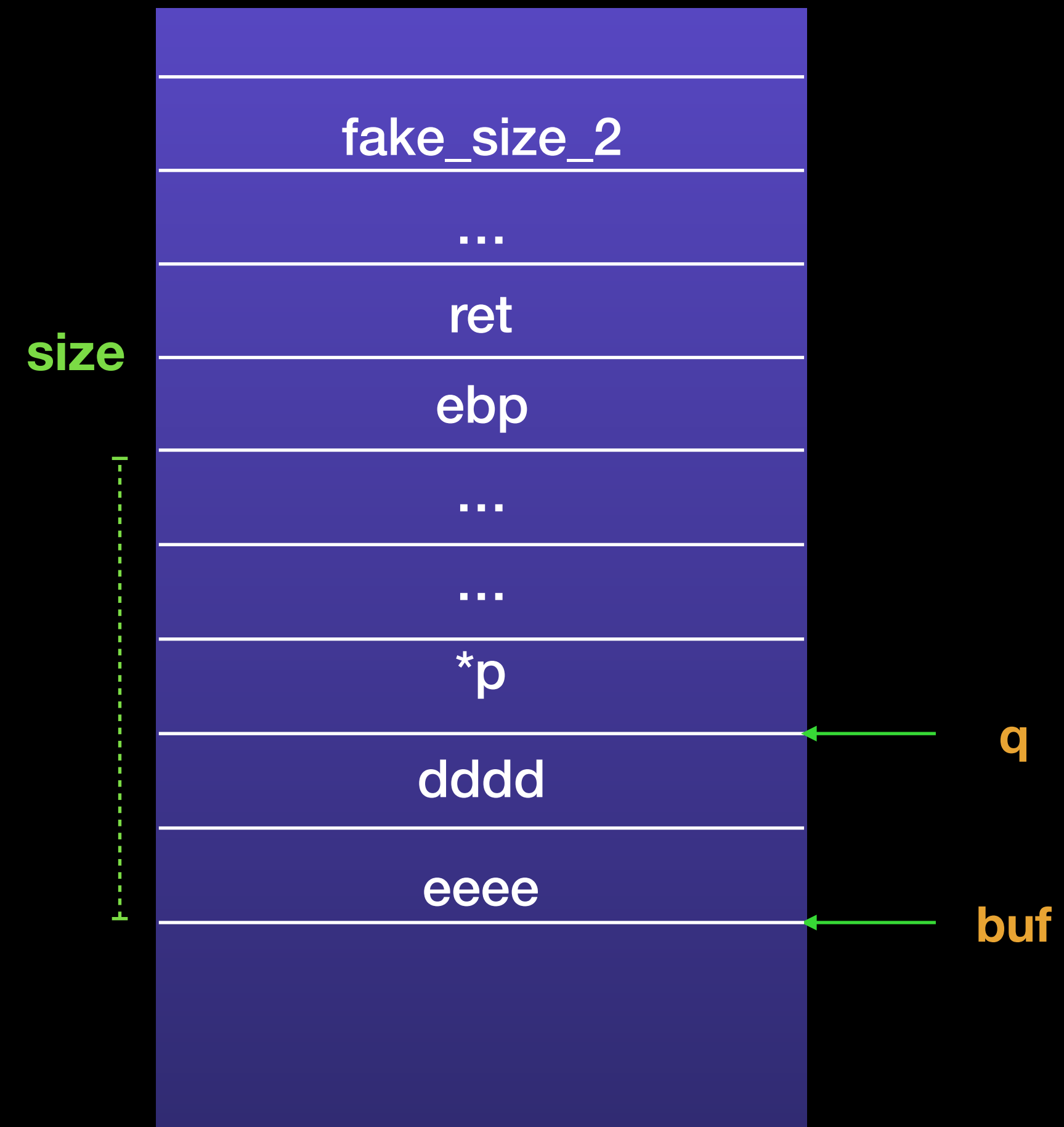
- stack overflow
 - 當 stack overflow 不夠蓋到 ret 時
 - 利用 stack overflow 蓋過要 free 的 ptr 並偽造 chunk
 - 須針對 prev_size 及 size 做處理，通過檢查
- using fastbin
 - 當下次 malloc 時 就會取得偽造的 chunk

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 是否合法為 0x10 的倍數，size 的取決是很重要的
 - 32 bit 為 0x8 倍數

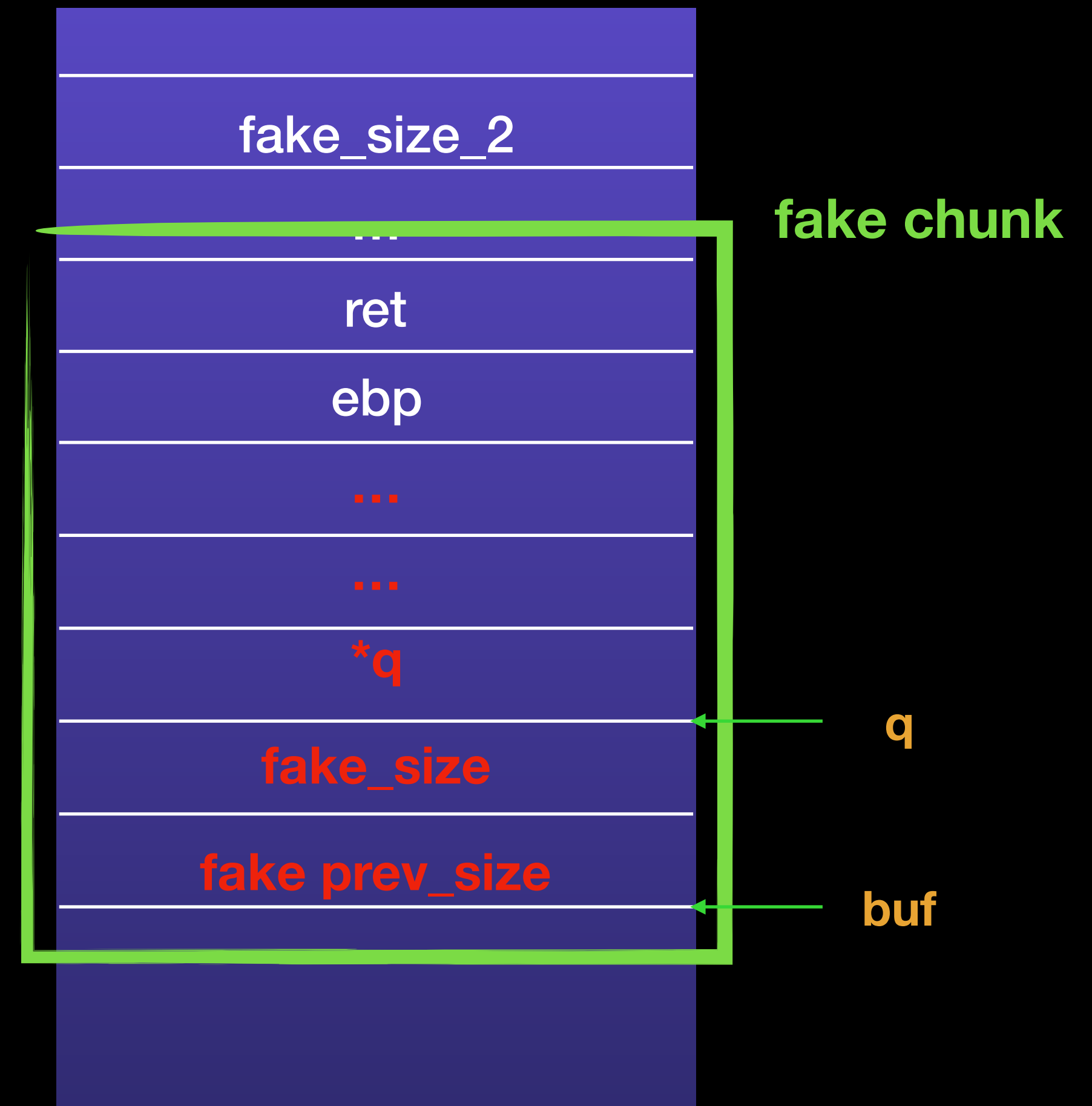
The House of Spirit

- Assume exist free(p)
- read(0,buf,size)
- read 不夠長到可以蓋 ret



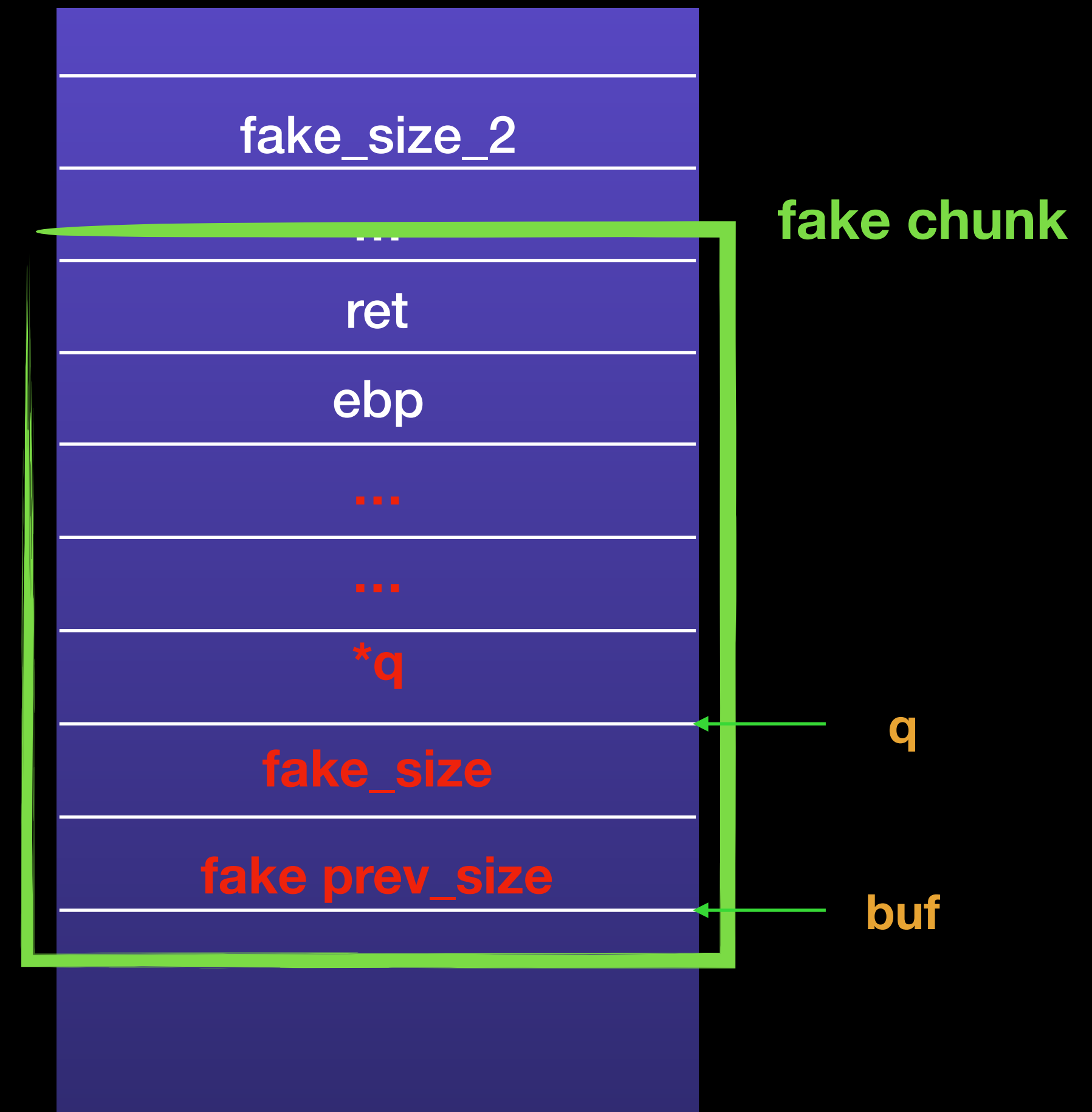
The House of Spirit

- overflow *p



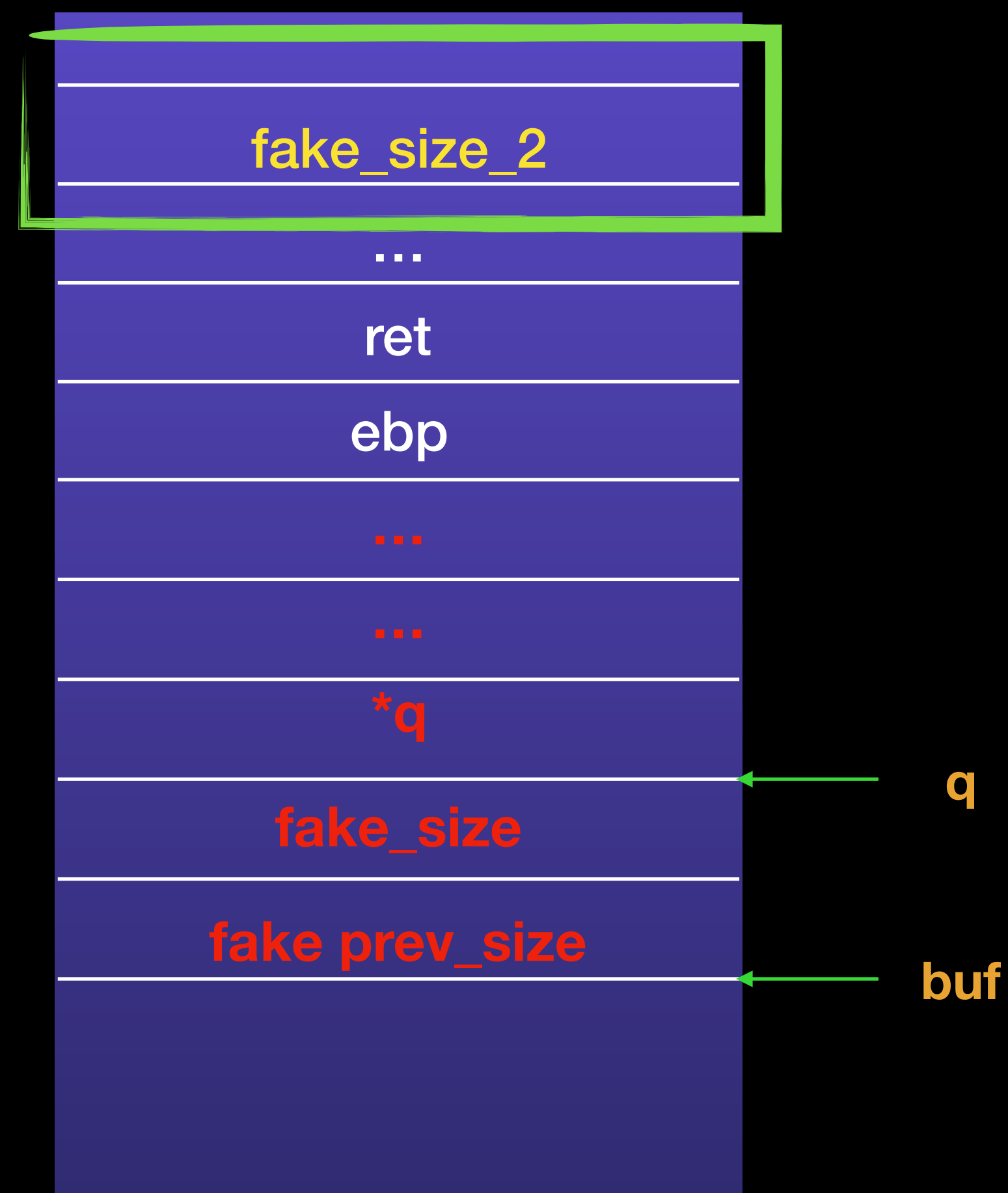
The House of Spirit

- overflow *p
- free(p) -> free(q)



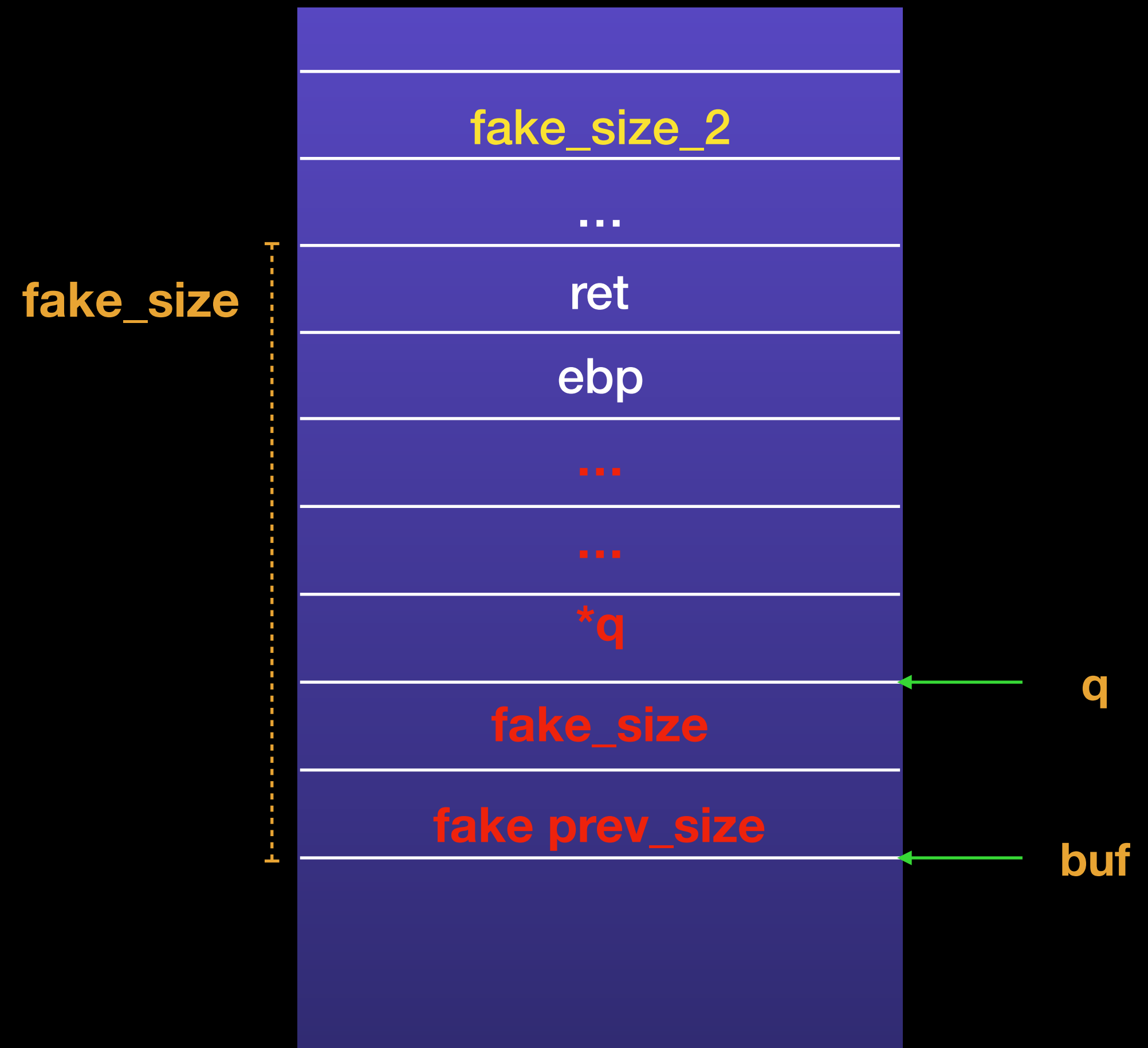
The House of Spirit

- overflow *p
- free(q)
- 下一塊 size 要合法



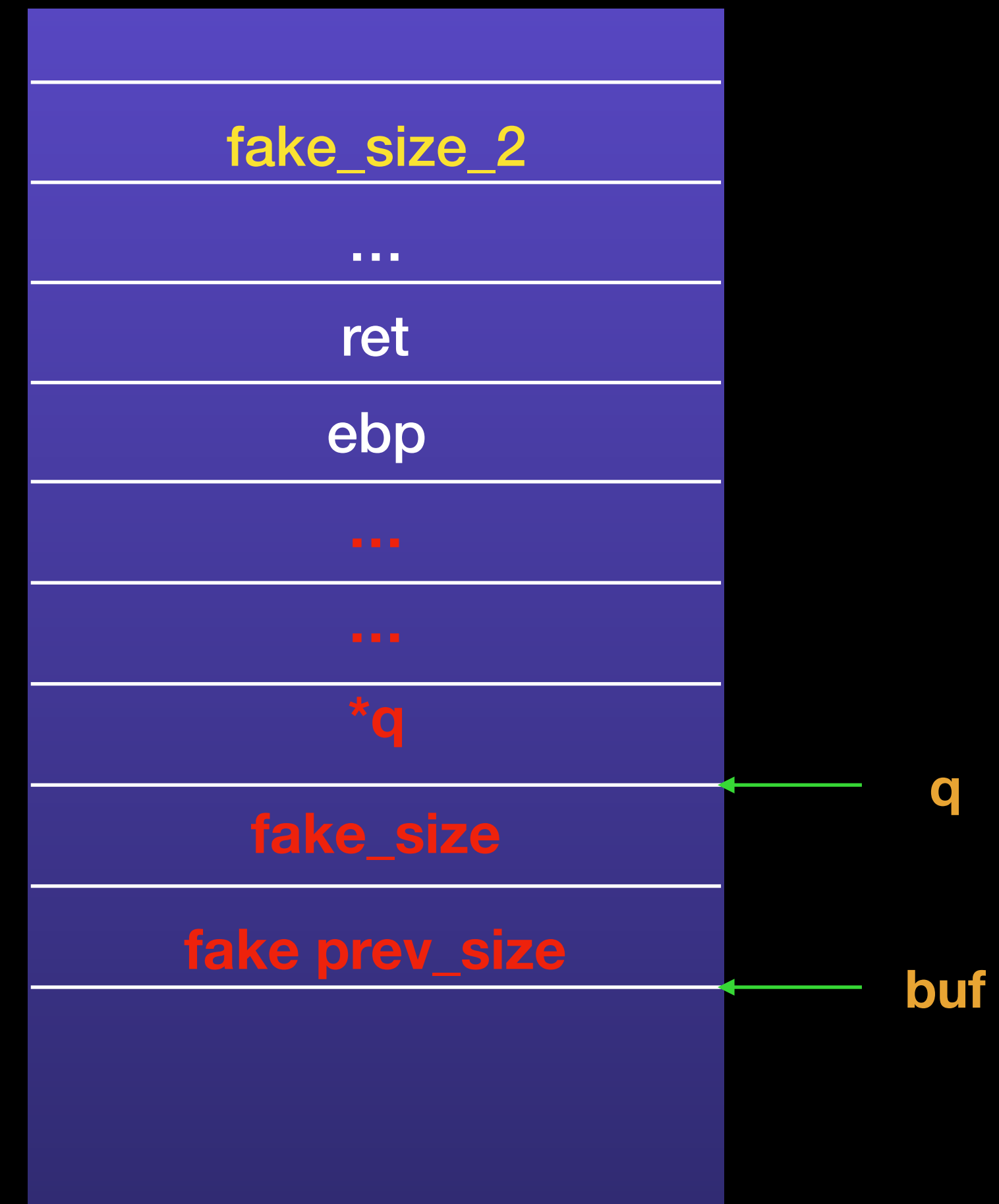
The House of Spirit

- overflow *p
- free(q)
- malloc(fake_size-16)
 - it will return q



The House of Spirit

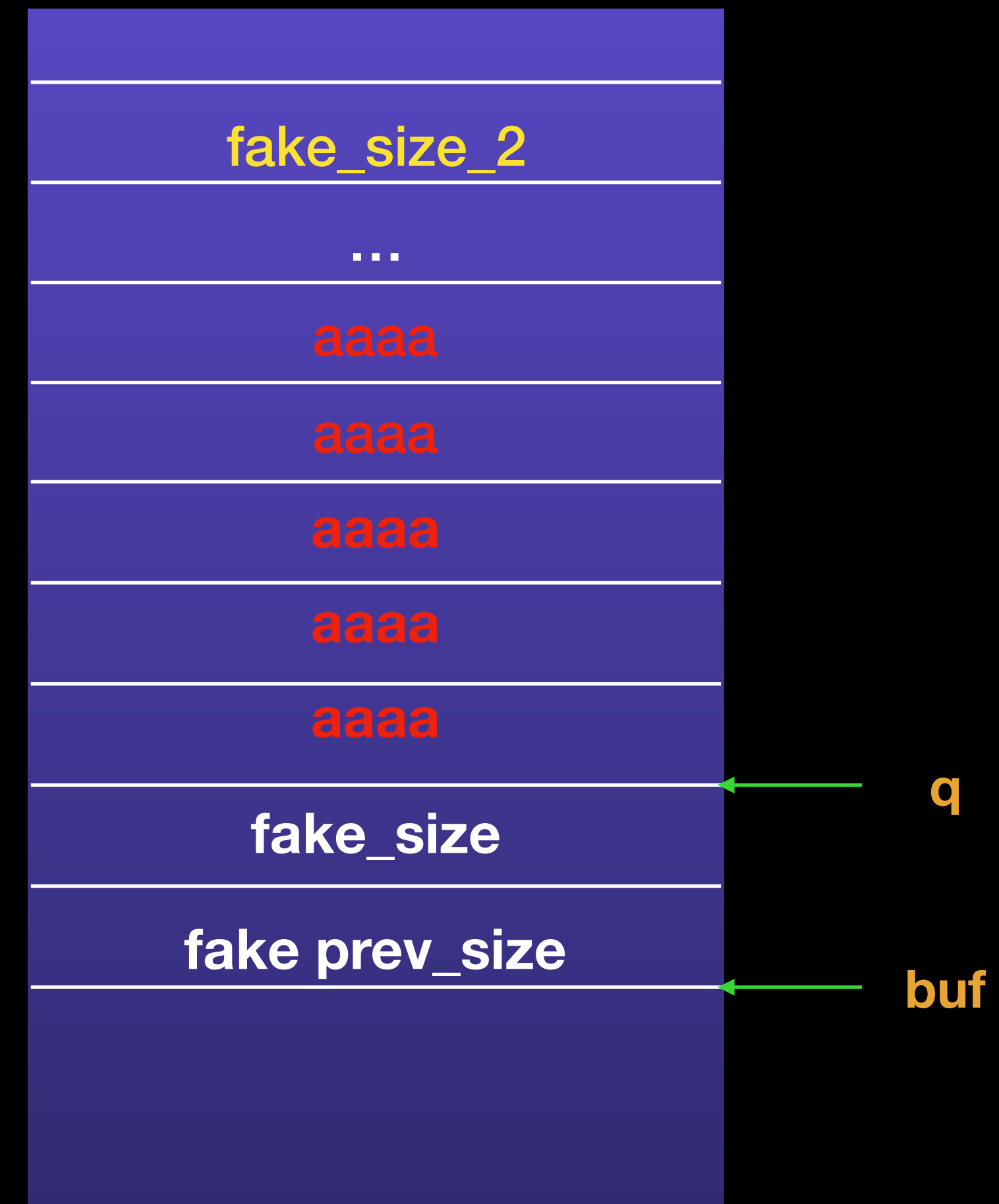
- overflow *p
- free(q)
- malloc(fake_size-16)
 - it will return q
- read(0,q,fake_size-16)



The House of Spirit

- overflow *p
- free(q)
- malloc(fake_size-16)
 - it will return q
- read(0,q,fake_size-16)

→ **overflow**

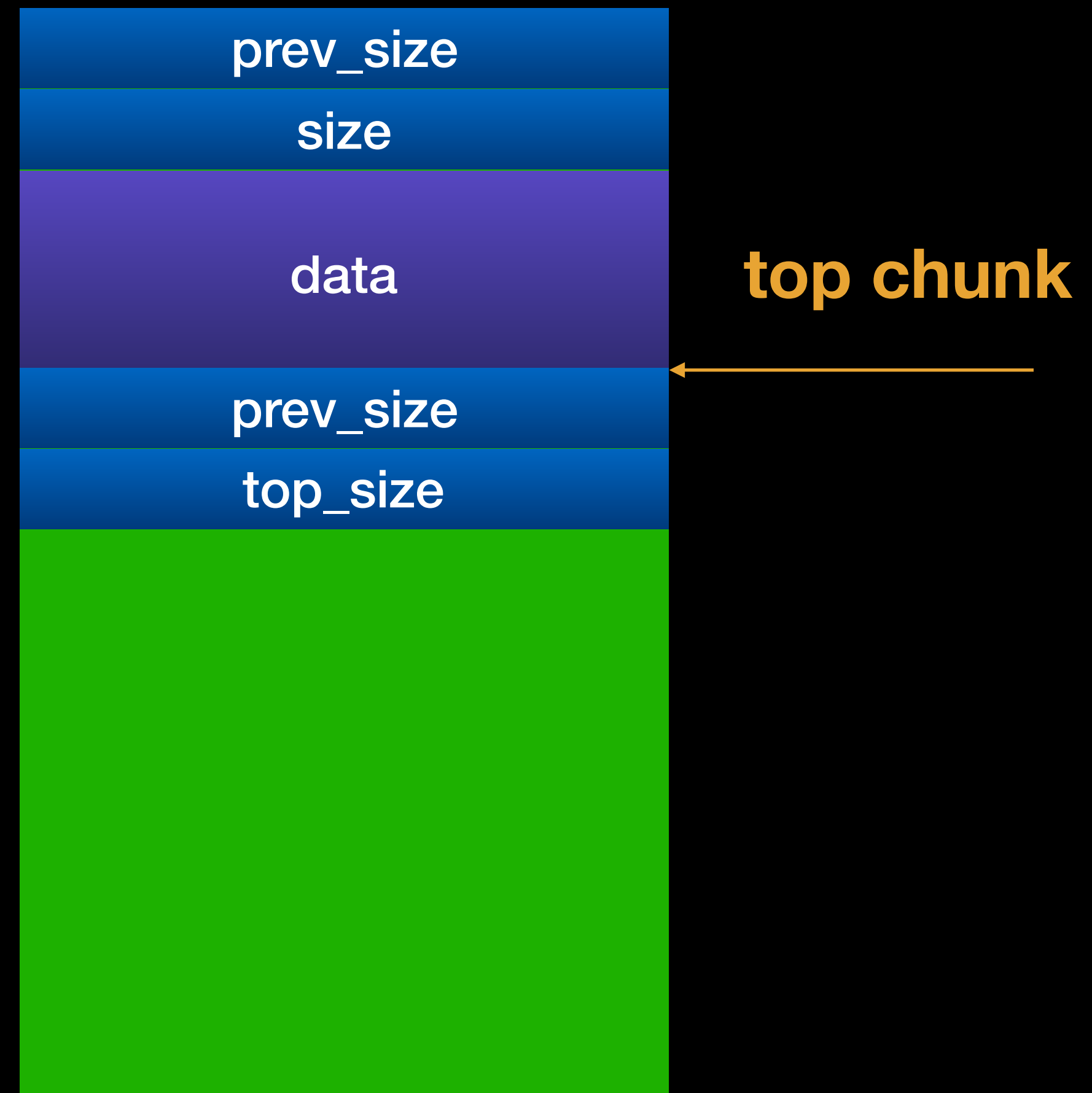


The House of Force

- malloc 從 top 分配空間時，top 位置會以當前位置+分配 chunk 大小，作為新的 top 位置
- nb = malloc 時 user 所需大小
- $\text{new top} = \text{old top} + \text{nb} + \text{sizeof}(\text{chunk header})$

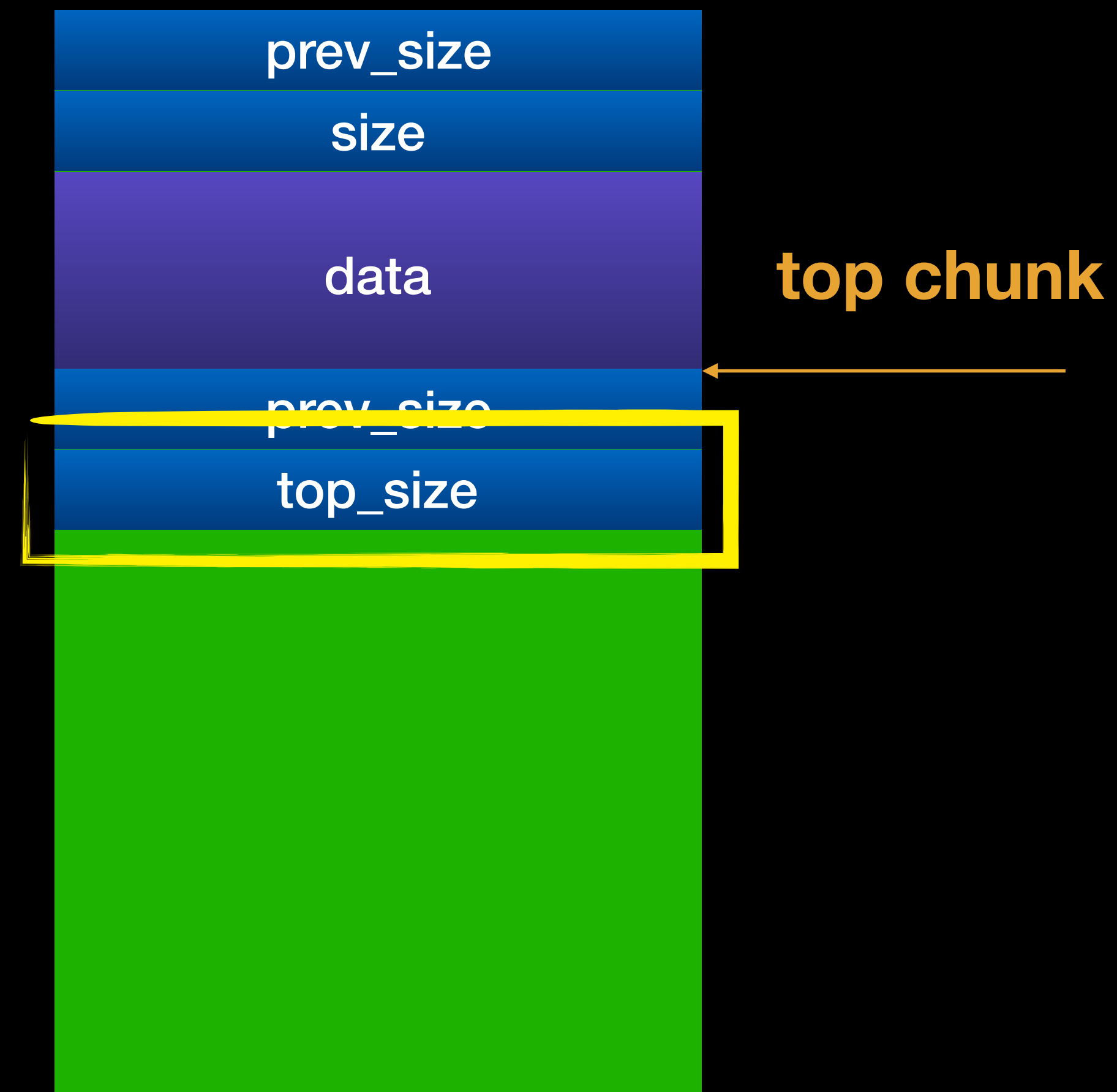
The House of Force

- malloc(nb)



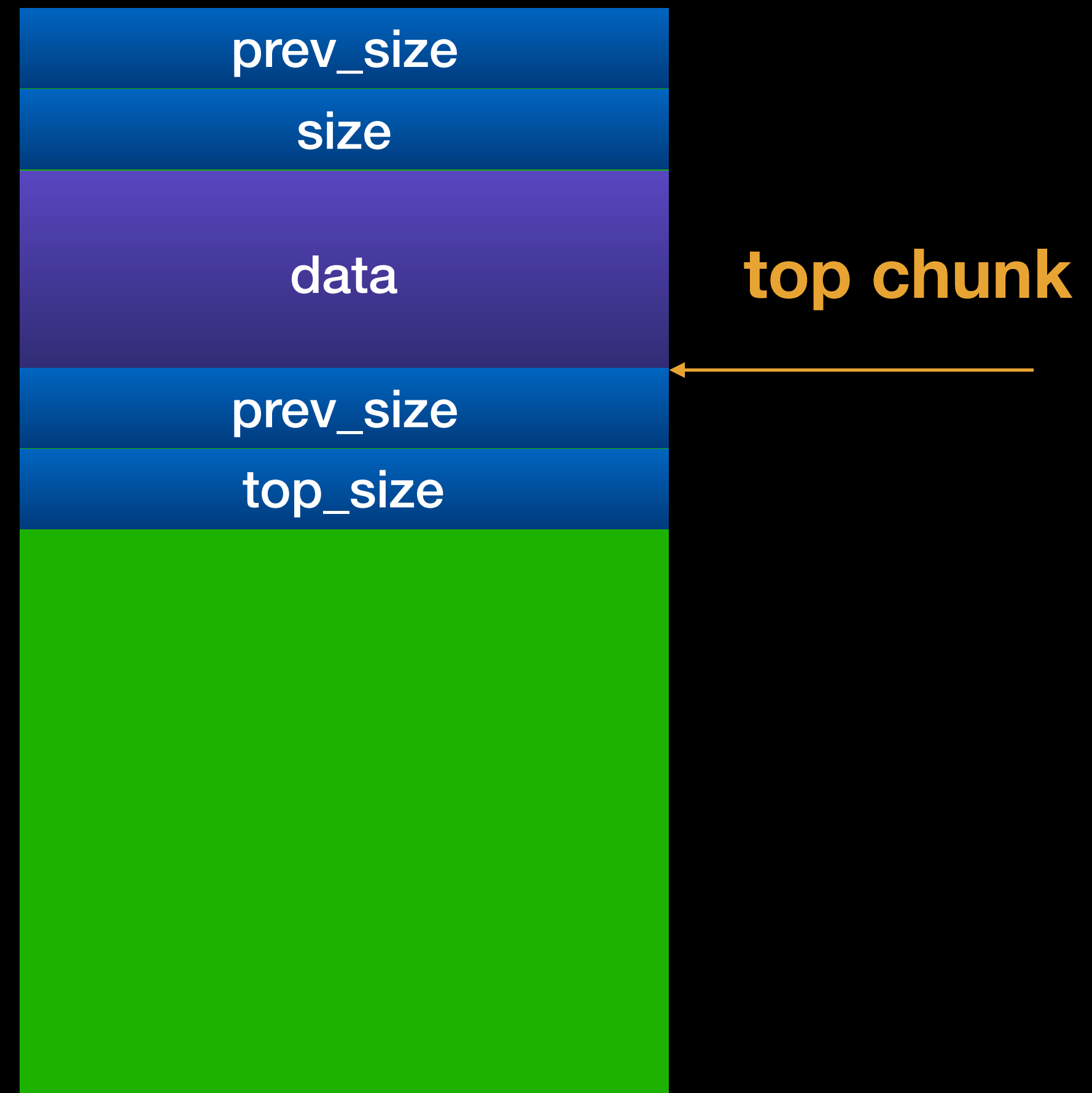
The House of Force

- malloc(nb)
 - 檢查 top_size 是否夠分配給使用者
 - $\text{top_size} - \text{nb} > 0$?
 - true : 從 top 分配
 - false : 使用 mmap or 擴大 heap 空間



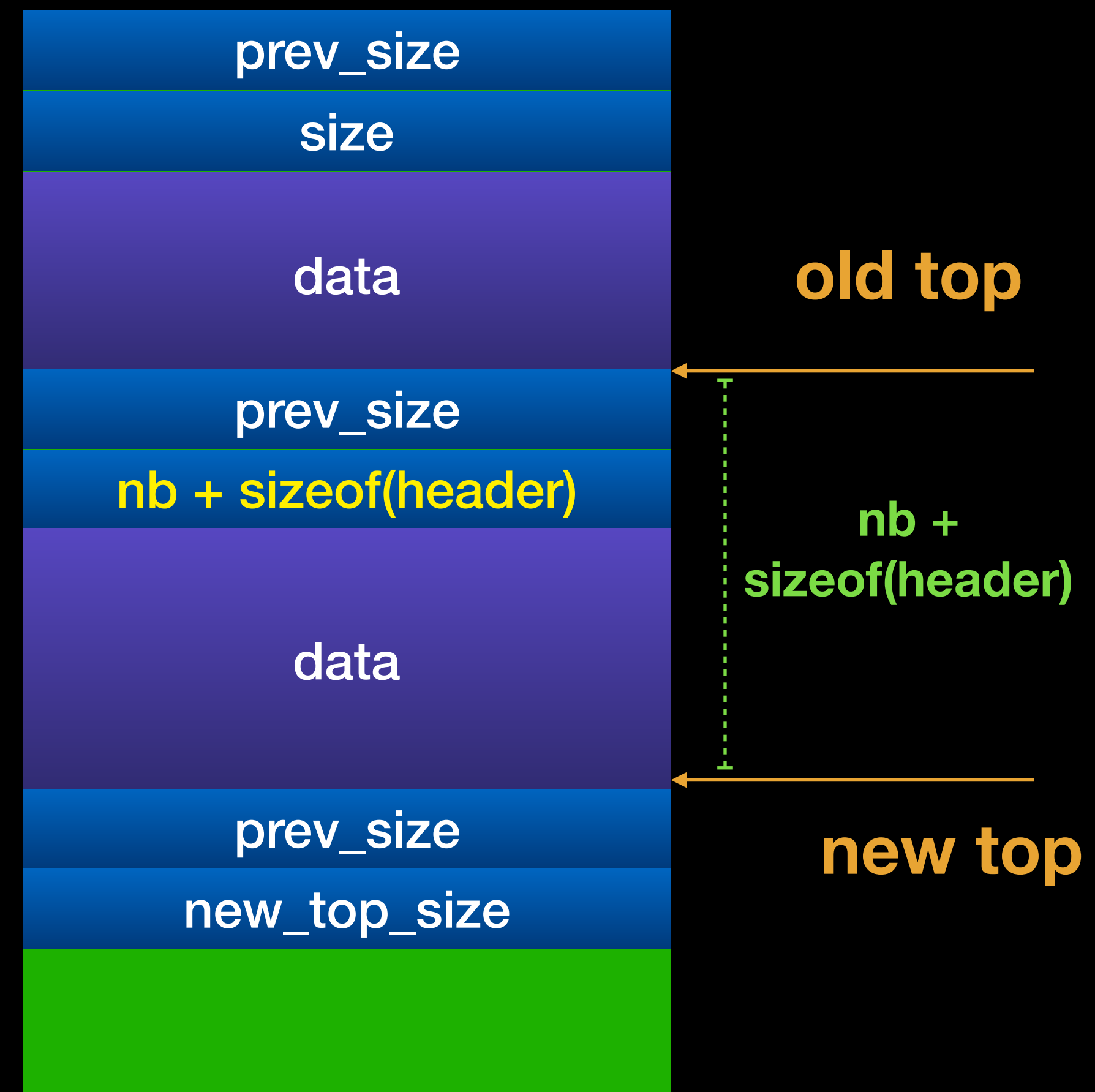
The House of Force

- `malloc(nb)`
- **`new top = old top + nb + sizeof(chunk header)`**



The House of Force

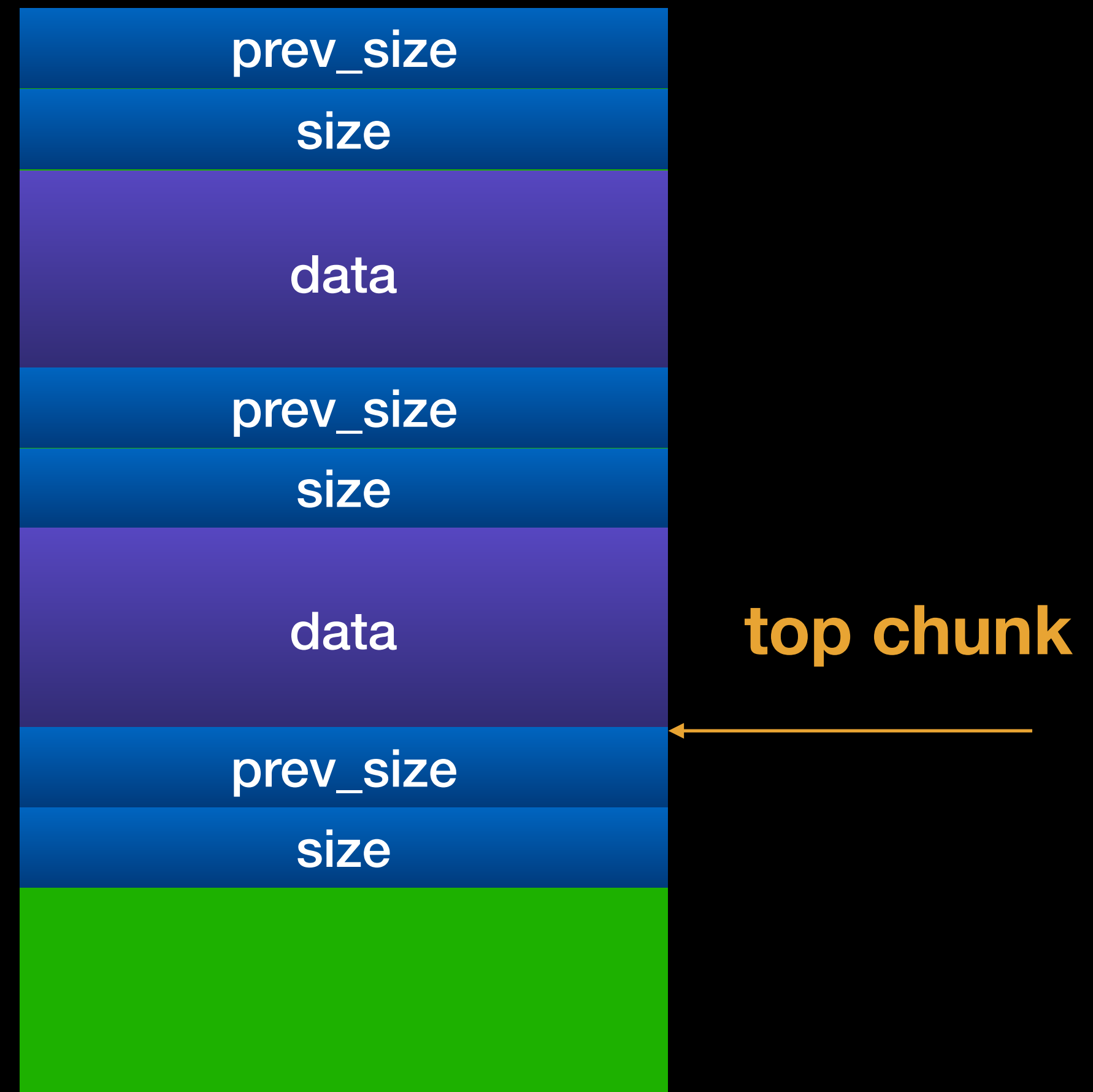
- `malloc(nb)`
- `new top = old top + nb + sizeof(chunk header)`
- `new_top_size = top_size - (nb + sizeof(chunk header))`



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
 - $\text{new top chunk} = \text{old top} + \text{nb} + \text{sizeof}(\text{header})$
 - 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 $\text{fake size} - \text{nb} > 0$ 就會讓 glibc 以為 top 還有空間可以給，因 nb 是負的，所以 top 會往前，造成 overlap
- 這樣下次 malloc 的位置將會是 new top chunk 的位置

The House of Force



The House of Force



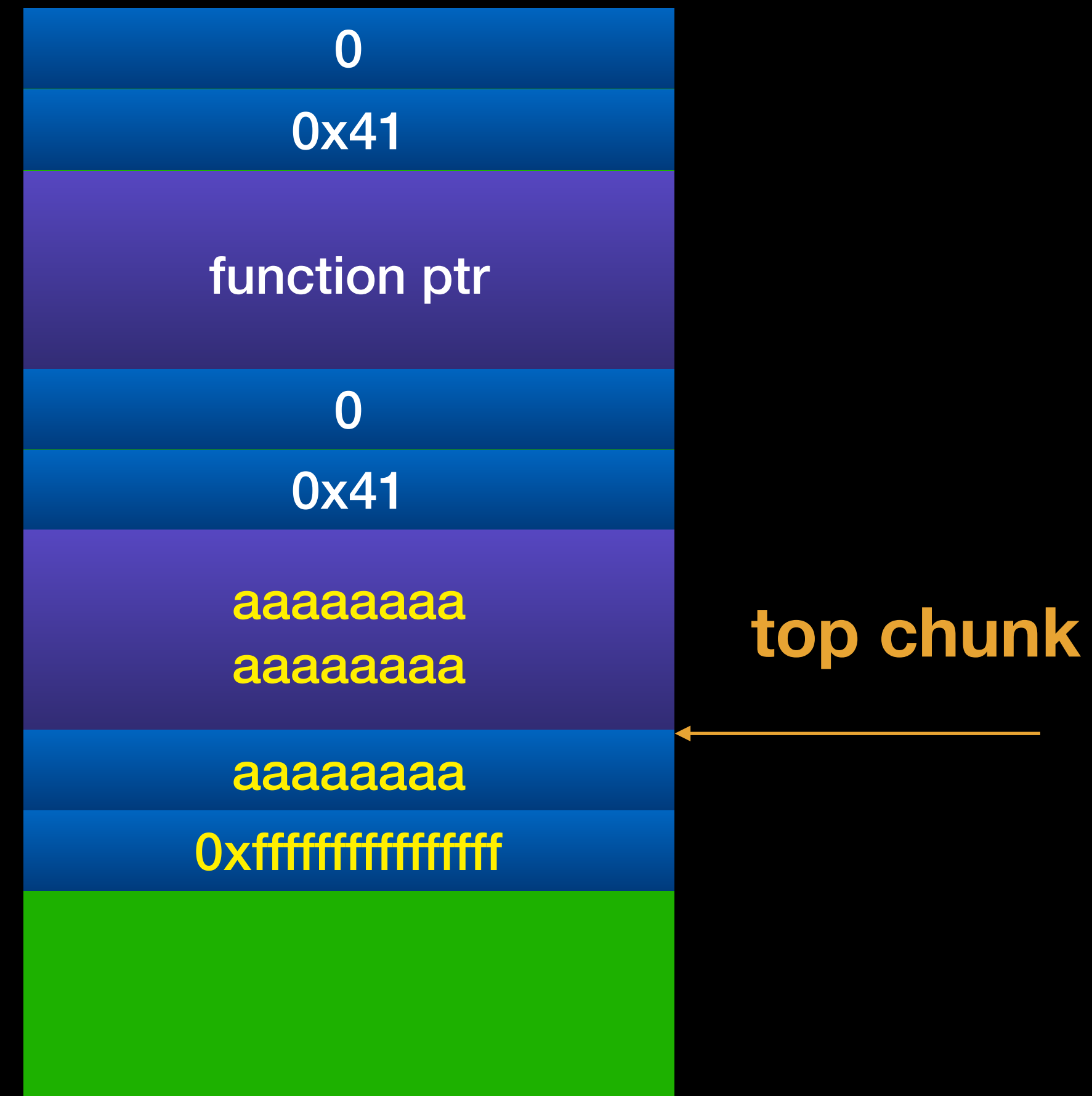
The House of Force

- Overflow
 - top size -> a large value



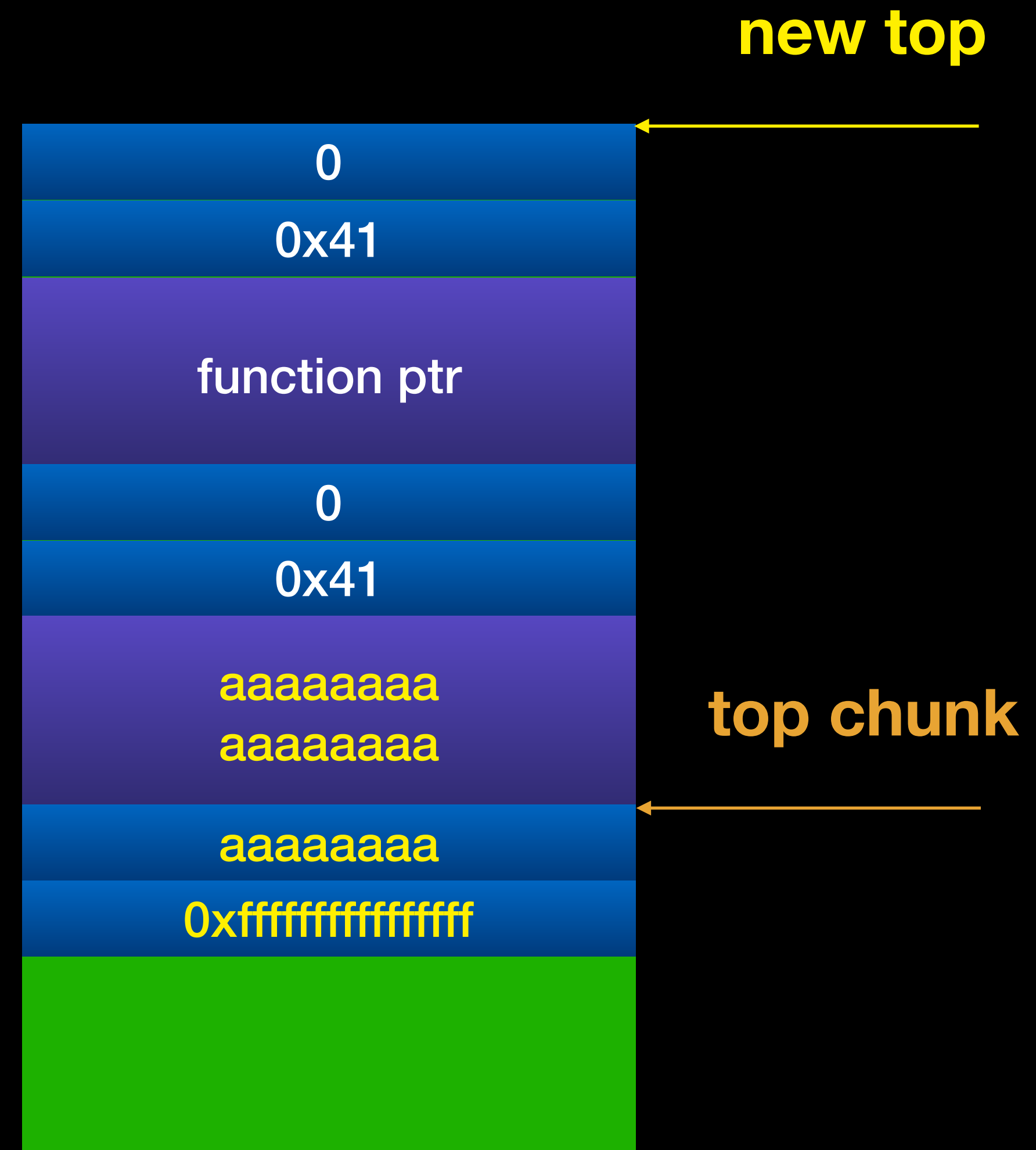
The House of Force

- Overflow
 - top size -> a large value
- malloc(nb)
 - $\text{new top chunk} = \text{old top} + \text{nb} + 16$
 - $\text{nb} = \text{new top} - \text{old top} - 16$



The House of Force

- Overflow
 - top size -> a large value
- malloc(nb)
 - new top chunk = old top + nb + 16
 - $nb = -(0x40+0x40) - 0x10 = -0x90 = -144$



The House of Force

- Overflow
 - top size -> a large value
- malloc(-144)
 - check
 - $0xffffffff - (-144) > 0$



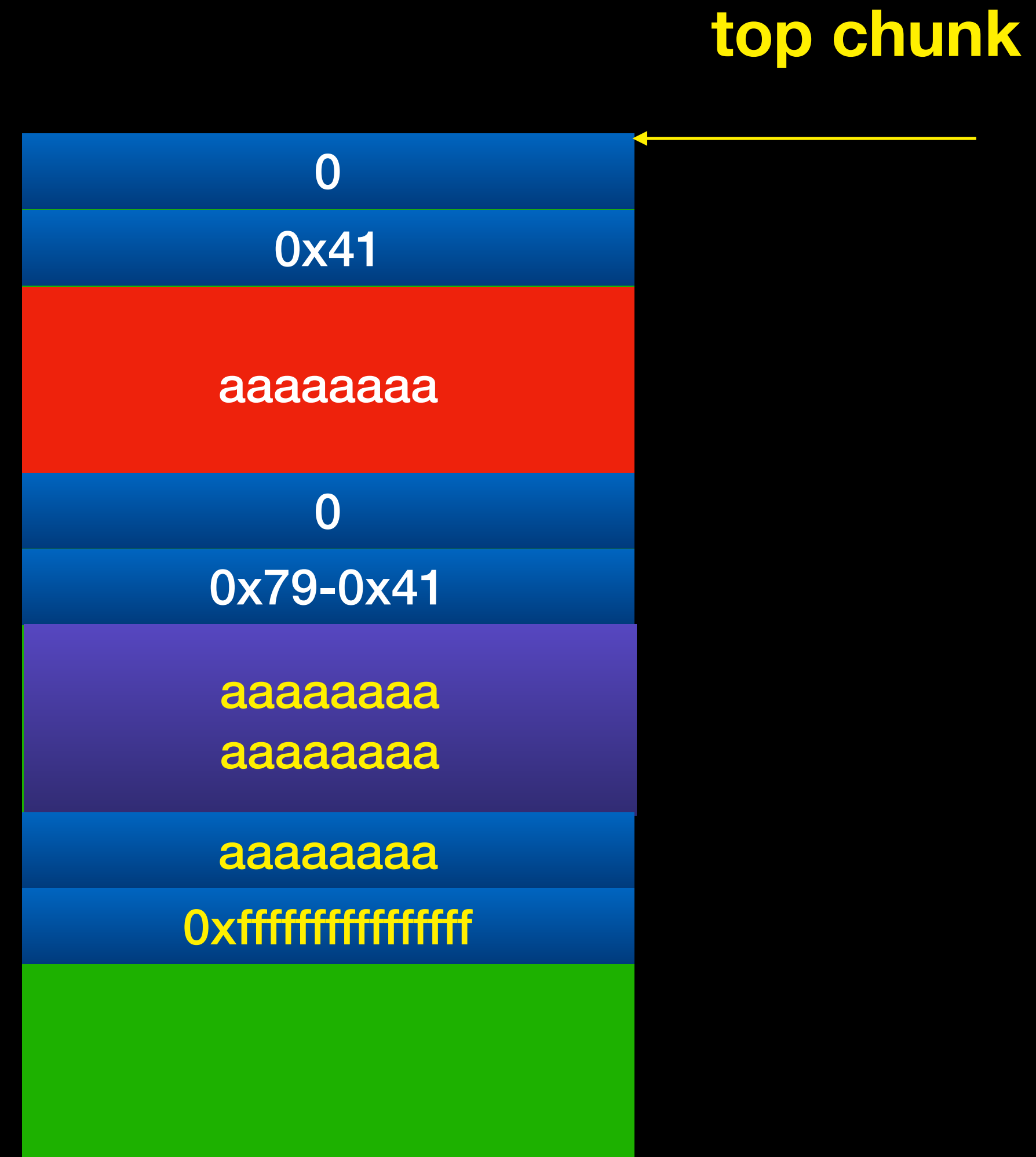
The House of Force

- Overflow
 - top size -> a large value
- malloc(-144)
- malloc(0x30)



The House of Force

- Overflow
 - top size -> a large value
- malloc(-144)
- malloc(0x30)
- call function ptr



The House of Force

- Overflow

- top size ->

- malloc(-144)

- malloc(0x30)

- call function

top chunk

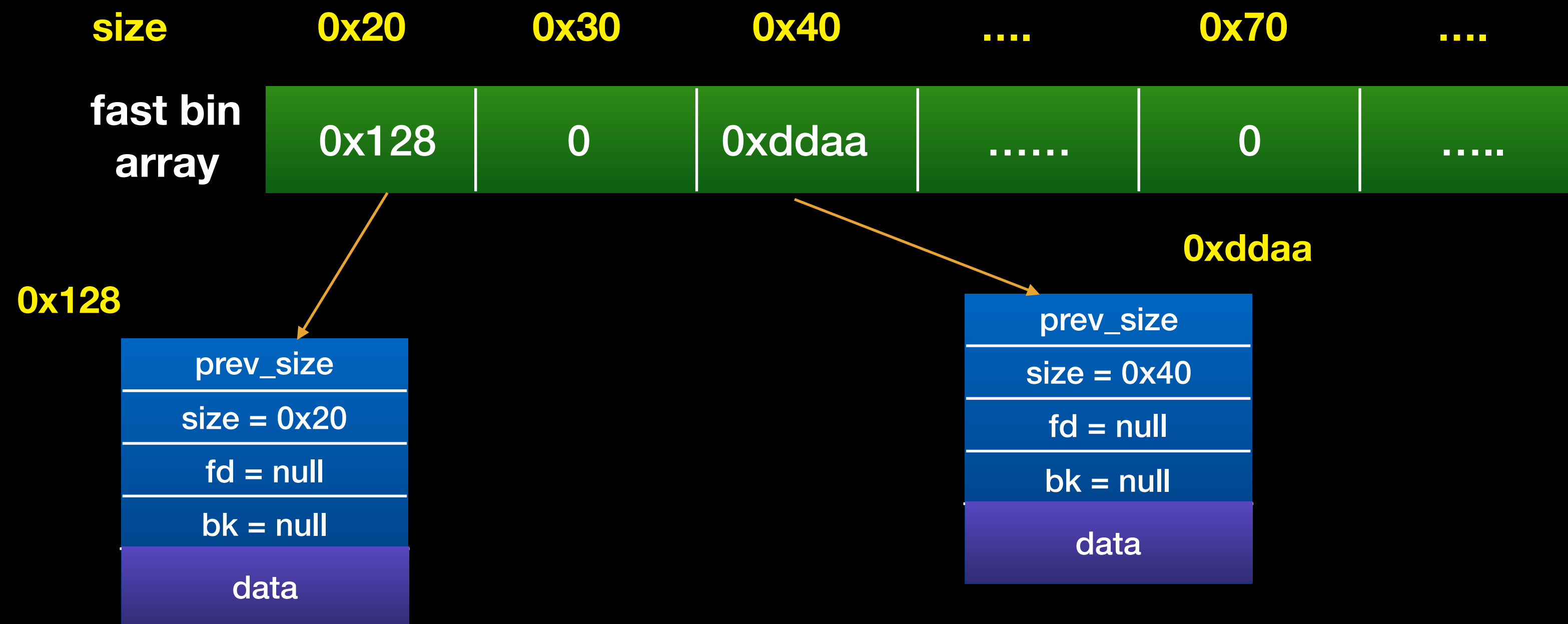


Control RIP

Overwrite Fastbin

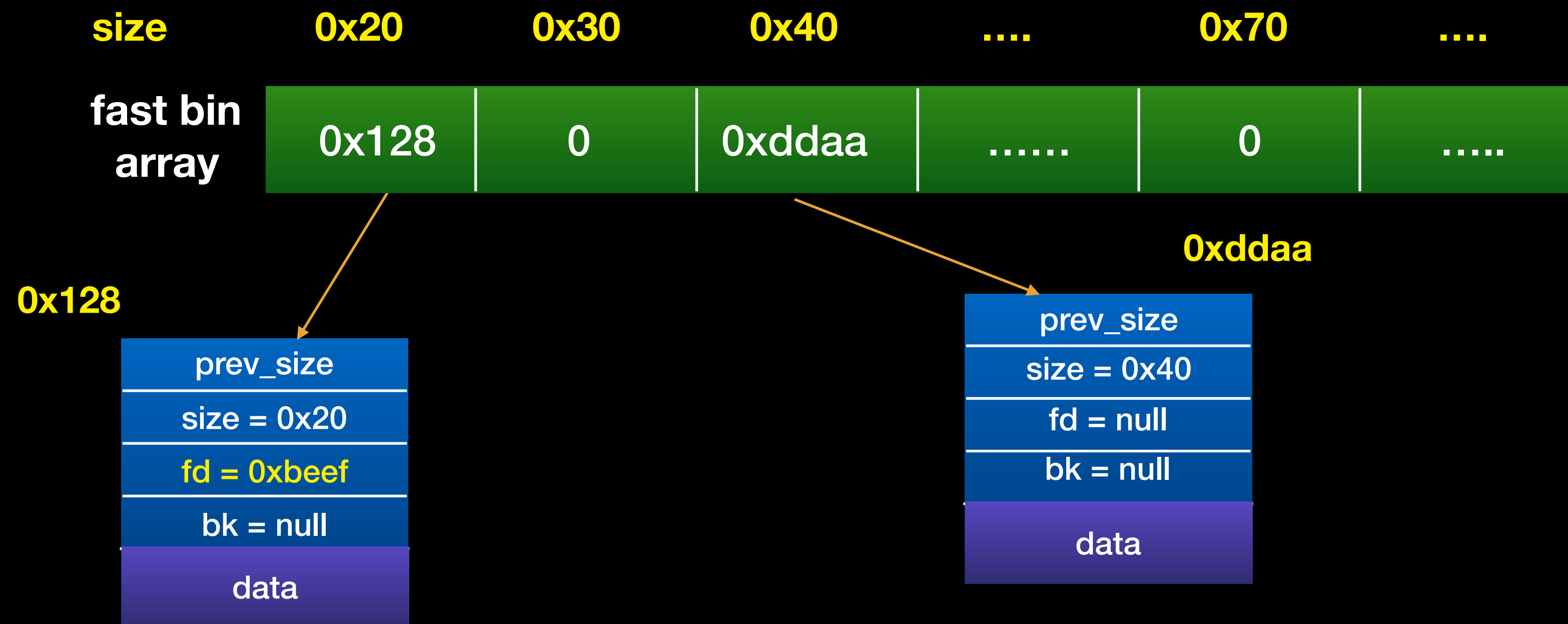
- 類似 House of Spirit
- 如果可以改到 fastbin 的 free chunk 可以將 fd 改成 fake chunk 的位置，只要符合 size 是屬於該 fastbin 就好，因為在下一次 malloc 只會檢查這項
- 下下次 malloc(size-0x10) 時，就會取得該 fake chunk 的位置
- fake chunk 可以是任意記憶體位置

Overwrite Fastbin



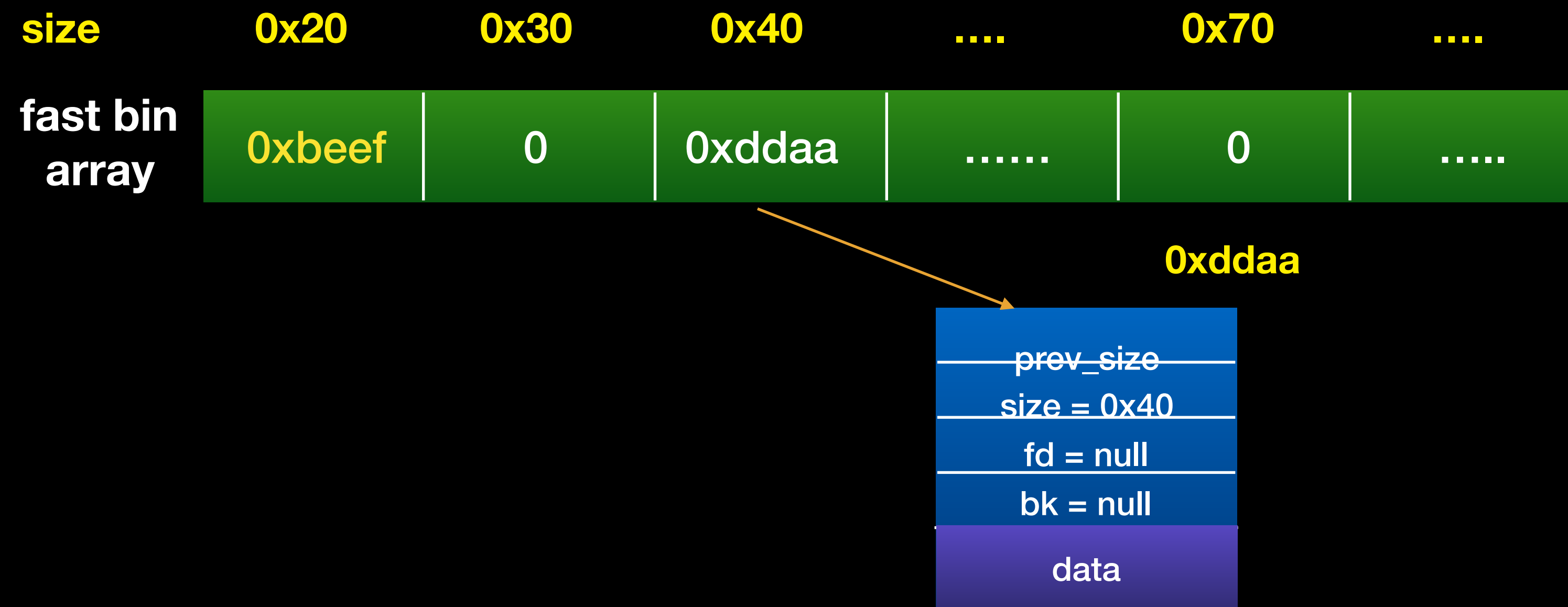
Overwrite fd

Overwrite Fastbin



malloc(0x10) = 0x128+0x10

Overwrite Fastbin



malloc(0x10) = Get the 0xbeef chunk

Overwrite Fastbin

size

fast b
arra

可對 0xbeef 任意寫值

malloc(0x10) = Get the 0xbeef chunk

Practice

- bamboobox1
 - House of force
- bamboobox2
 - Using unlink to Get shell

Homework

- hacknote2
- profile manager

Outline

- Glibc memory allocator Overview
- Use After Free
- Heap Overflow
- Appendix - Detection in Glibc

Detection in Glibc

- 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)

Detection in Glibc

- 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`

Detection in Glibc

- 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 (0x10)`
 - `size` 必須小於 `system_mem`
 - `system mem` : `heap` 的大小通常為 `132k`

Detection in Glibc

- 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 unsortedbin 中
 - 這時會取 **unsortedbin** 的第一個的 fd 是否等於 **unsortedbin** 的位置

Detection in Glibc

- detection in free
 - invalid pointer
 - 檢查 alignment
 - chunk address 是否為 0x10 的倍數
 - 檢查 chunk address 是否小於 - size

Detection in Glibc

- detection in **free**
 - **invalid size**
 - 檢查 chunk size 是否合法
 - size 必須為 0x10 的倍數 (不含最低 3 bit)
 - 也就是是否有符合 alignment
 - size 必須大於 MINSIZE (0x20 byte)

Detection in Glibc

- 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

Detection in Glibc

- 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

Detection in Glibc

- 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

Detection in Glibc

- 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

Detection in Glibc

- 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

Detection in Glibc

- 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 $2 \times \text{SIZE_SZ}$ (0x10)
 - size 必須小於 system_mem
 - system mem : heap 的大小通常為 132k

Detection in Glibc

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

Reference

- [understanding-glibc-malloc](#)
- [Modern Binary Exploitation - heap exploitation](#)
- [MallocMaleficarum](#)
- [glibc cross reference](#)
- [SP heap exploitation by dada](#)

Q & A