Heap Exploitation

yuawn

About

- yuawn
- Pwn
- Balsn / DoubleSigma

_yuawn

Outline

- Heap intro
- Common Concept
  - UAF (Use-After-Free), double free
  - heap overflow
  - one gadget, hooks
- Heap Exploitation
  - fastbin attack
  - Tcache
- heap overlap, unsorted bin attack
- unsafe unlink

Environment

- Ubuntu 16.04
  - libc-2.23
- Ubuntu 18.04
  - libc-2.27
- x64

LIBC-2.27

Heap

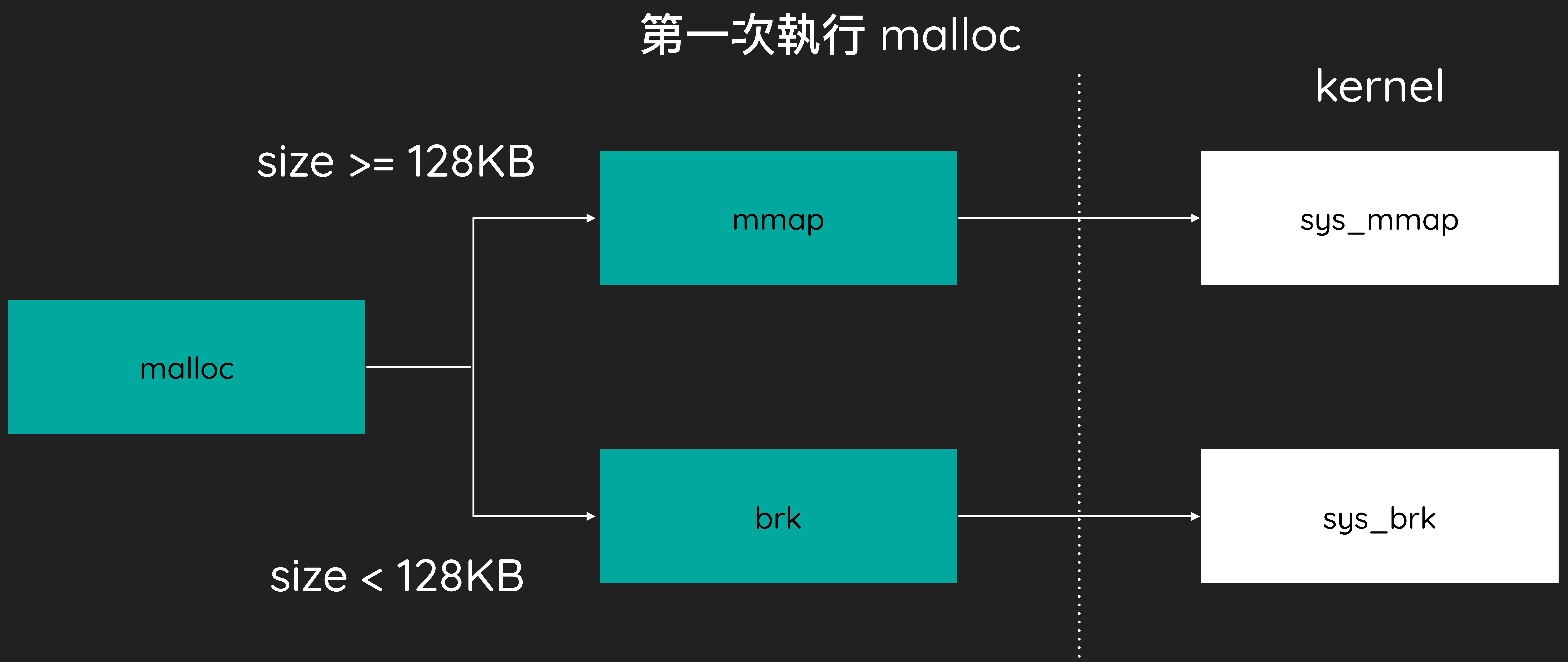
ptmalloc2

- ptmalloc2 - glibc
- tcmalloc - google
- jemalloc
- …

malloc

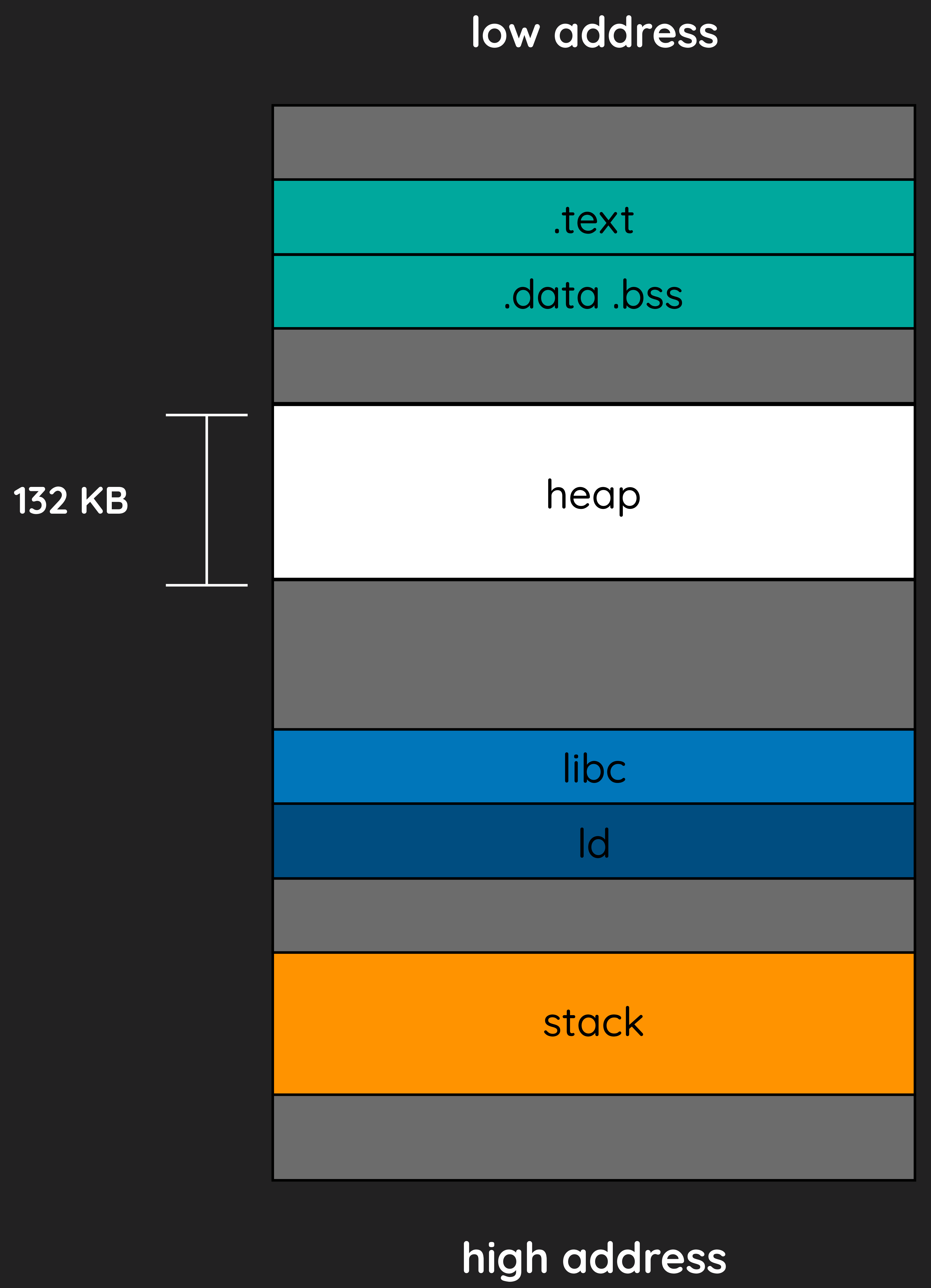
- 要用多少分配多少，提升記憶體分配效率以及避免記憶體空間的浪費。
- void *ptr = malloc(size)

workflow of malloc



main arena

- 一開始malloc < 128 KB，透過brk，kernel會給 132 KB 的heap segment (rw)，稱之 main arena。
- ASLR - heap base



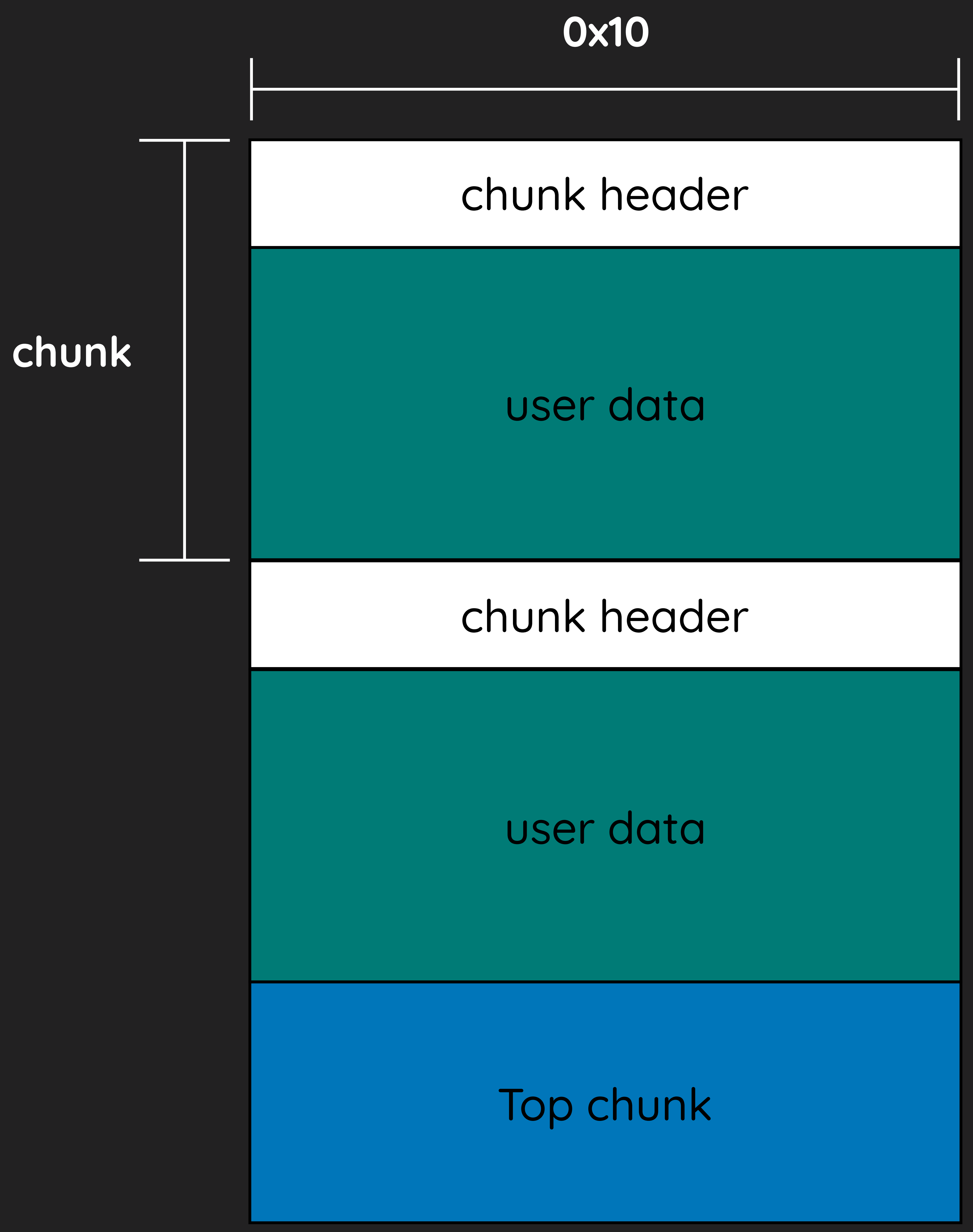
main arena

- struct malloc_state main_arena
- glibc-2.23/malloc/malloc.c

```
struct malloc_state
{
  /* Serialize access.  */
  mutex_t mutex;

  /* Flags (formerly in max_fast).  */
  int flags;

  /* Fastbins */
  mfastbinptr fastbinsY[NFASTBINS];

  /* Base of the topmost chunk -- not otherwise kept in a bin */
  mchunkptr top;

  /* The remainder from the most recent split of a small request */
  mchunkptr last_remainder;

  /* Normal bins packed as described above */
  mchunkptr bins[NBINS * 2 - 2];

  /* Bitmap of bins */
  unsigned int binmap[BINMAPSIZE];

  /* Linked list */
  struct malloc_state *next;

  /* Linked list for free arenas.  Access to this field is serialized
     by free_list_lock in arena.c.  */
  struct malloc_state *next_free;

  /* Number of threads attached to this arena.  0 if the arena is on
     the free list.  Access to this field is serialized by
     free_list_lock in arena.c.  */
  INTERNAL_SIZE_T attached_threads;

  /* Memory allocated from the system in this arena.  */
  INTERNAL_SIZE_T system_mem;
  INTERNAL_SIZE_T max_system_mem;
};
```

chunk

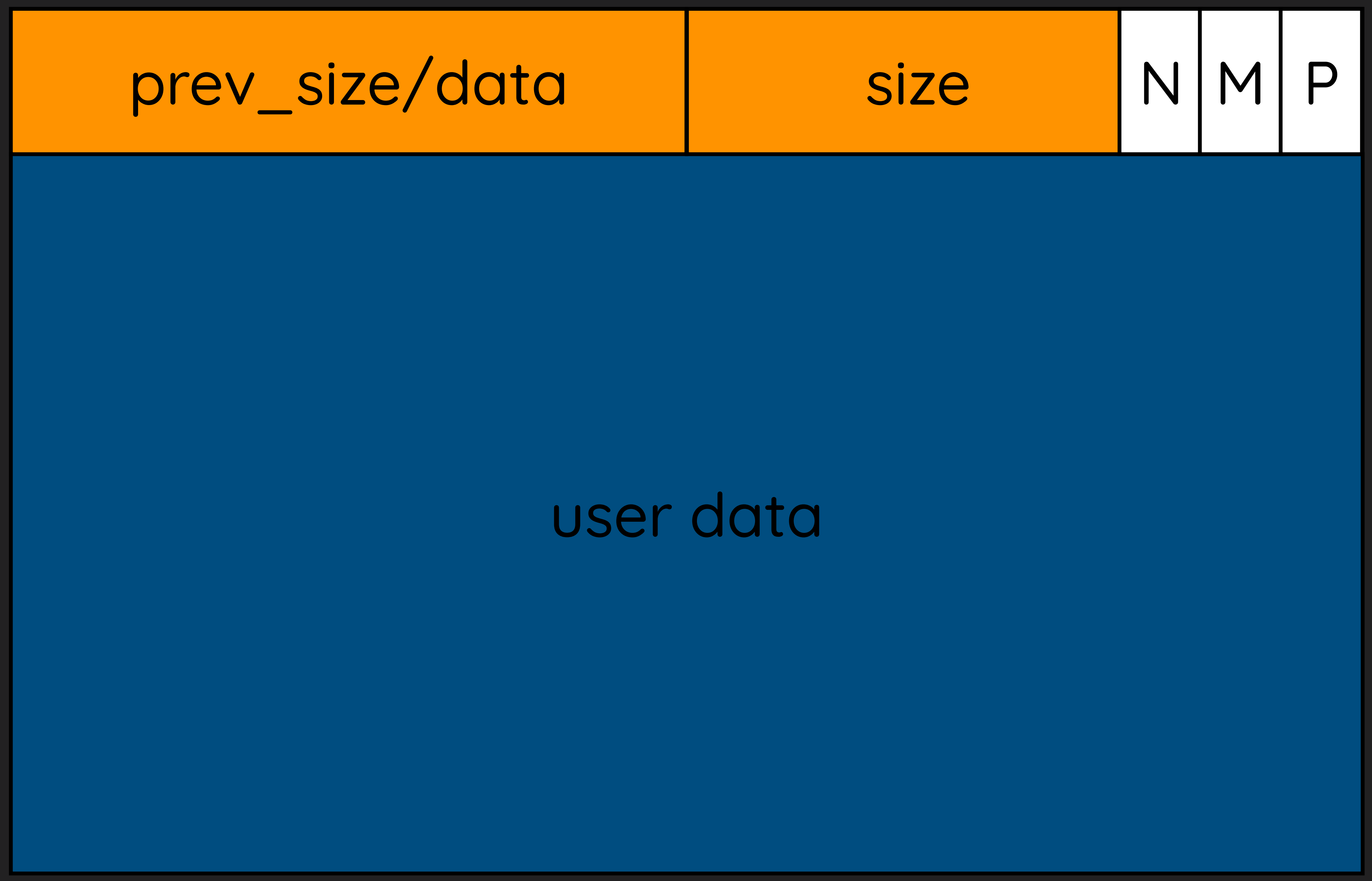
- glibc 實作記憶體管理機制的資料結構 (data structure)
- malloc 拿到的一塊記憶體即為 chunk
  - Allocated chunk
  - Free chunk
  - Top chunk
- size alignment
  - 0x10的倍數，malloc 0x18會拿到 0x20 的大小。
- 整個chunk佔記憶體大小為 header(0x10) + data



chunk header

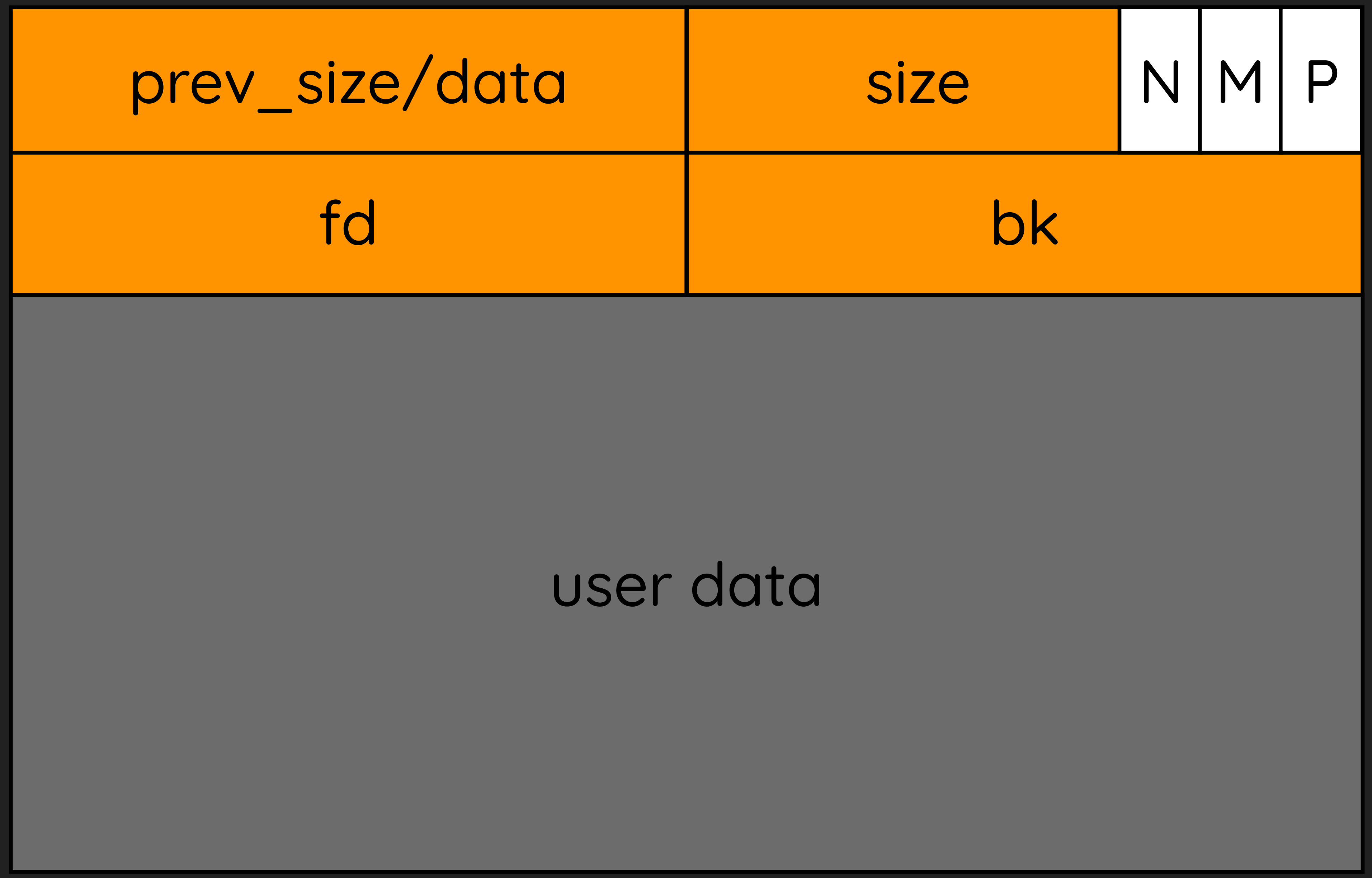
Allocated chunk

- prev_size
  - 連續記憶體上一塊如是 free chunk，則紀錄該size，若是 allocated chunk 則同時為它的 data。
- size
  - chunk size with 3 flags
    - PREV_INUSE(P)：上一個 chunk 是否使用中
    - IS_MMAPED(M)：chunk 是否透過 mmap 出來的
    - NON_MAIN_ARENA(N)：該 chunk 是否不屬於 main arena



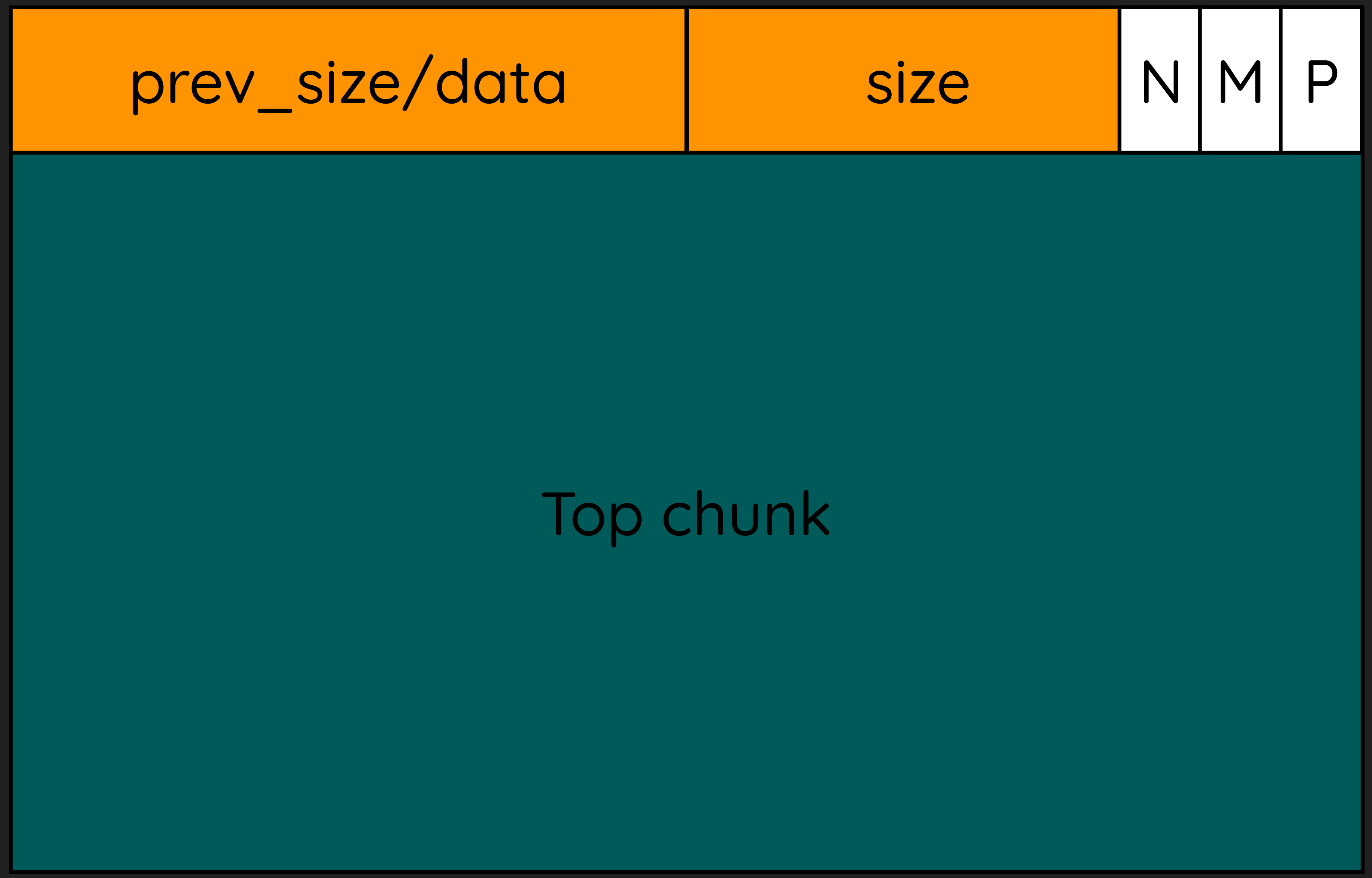
Free chunk

- prev_size
  - 連續記憶體上一塊如是 free chunk，則紀錄該 size，若是allocated chunk 則為它的 data。
- size
  - 連續記憶體下一塊的 P flag 為0
- fd：指向同一 bin 中的前一塊 chunk (linked list)
- bk：指向同一 bin 中的後一塊 chunk (linked list)



Top chunk

- 第一次malloc後，剩下的空間為 top chunk，分配空間時視情況從 top chunk 切割分配。
- free Top chunk 連續記憶體上一塊chunk時，若不是 fastbin 則會與 Top chink merge，top chunk P 恆為1。



bins

bin

- 回收 free chunk 的資料結構
- 主要依據 size 大小，分為：
    - fast bin
    - small bin
    - large bin
    - unsorted bin

Fast bin

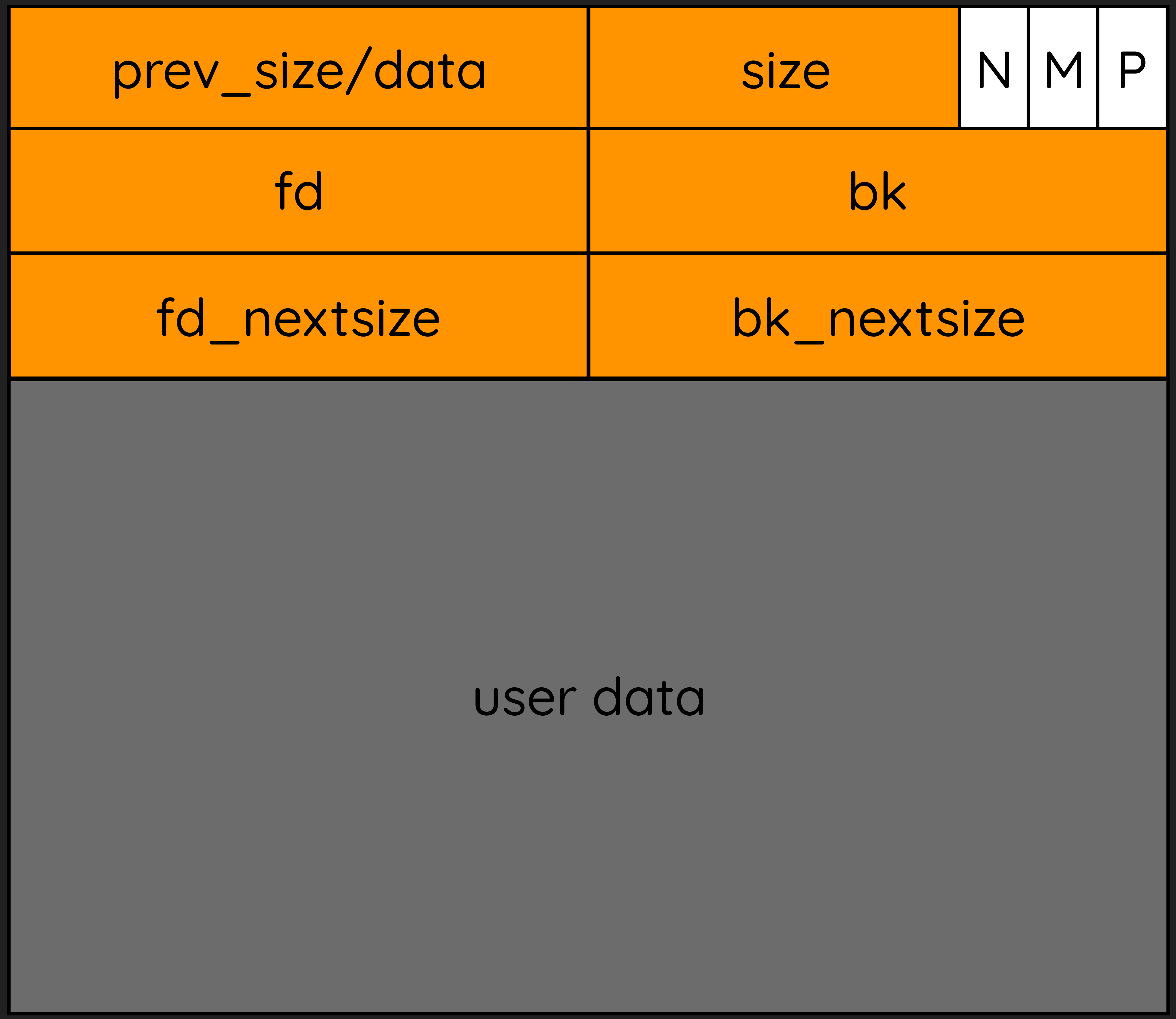
- Size < 0x90 bytes
- bin 中依據 size 劃分為，0x20, 0x30, 0x40 …
    - global_max_fast = 0x80
- Singly linked list，fd 指向前一個，bk 沒用到
- LIFO (Last in, First out)
- free 時不會將下一塊 chunk P flag 設成 0

Small bin

- Circular doubly linked list
- 依據 size 劃分為 62 個bin
  - 0x20, 0x30 ~ 0x3f0
  - 0x20 ~ 0x80 的大小與 fast bin 重疊，會根據機制放到fast bin或small bin
- FIFO (First in, First out)
- free 掉時會將下一塊chunk P 設為0

Large bin

- Circular doubly linked list (依據大小遞減排序)
- size >= 1024 bytes (0x400)
- 63 bins
  - 細節可以參考 source code
- header
  - fd_nextsize
  - bk_nextsize



Unsorted bin

- Circular doubly linked list
- free 的 chunk size 大於 fast bin 時，不會直接放到對應的 bin 裡，會先丟到 unsorted bin 中。
- malloc fast bin size 大小時會先去 fast bin list 裡找，若沒有則會至 unsorted bin 找，如找到一樣大小則回傳，若無但找到大小大於所需大小的 chunk 則切割回傳，剩下的部分會丟回 unsorted bin，若都沒有則從 top chunk 切出來回傳。

Demo

Vulnerability

heap

Heap Overflow

Heap overflow

- 在 heap segment 發生的 overflow
- 和 stack overflow 精神類似，stack overflow 目標是掌控 stack 上可利用資訊，如位於 stack 上的 return address，控制 rip。
- 而heap overflow 則是掌控 heap 中的利用目標，如某個 chunk 分配來是一個 object struct，透過 heap overflow overwrite 來進行偽造，或是控制 chunk header，並結合 glibc malloc free 的記憶體管理機制，做到進一步的利用。

UAF

use after free

UAF

- free(ptr)
- free 完 pointer 後未將 ptr 清空 (ptr = NULL)，稱之 dangling pointer 。
  - 意即有一個 pointer 指著一塊已經被釋放的記憶體 (dangerous)
- 根據不同的存取方式，有各種利用方法，可以進一步去做後續 exploit 的利用。
  - 用來 information leak，存取殘留的 data。
  - Struct Type Confusion
- Double free 就是因為存在 dangling pointer 所以造成 free 一塊已經釋放的 chunk，一樣可以透過一些技巧達到進一步的利用。

UAF - information leak

- free 兩個同 size 的 fastbin，fd 指向 heap，如果存在 UAF，將此 chunk user data 印出來或任何方法得知其值，透過印出 fd 來 leak 出 heap address。
- malloc 一塊非 fastbin size 的 chunk，free 掉他使他被放入 unsorted bin 中，或任何製造出 unsorted bin 的方法，unsorted bin 的 fd 與 bk 會是一個 libc address，直接 UAF 印出 fd 來 leak 出 libc address，或是 malloc 拿回這塊 chunk，印出殘留的 fd 等等。

Demo

Pwn

Heap exploitation

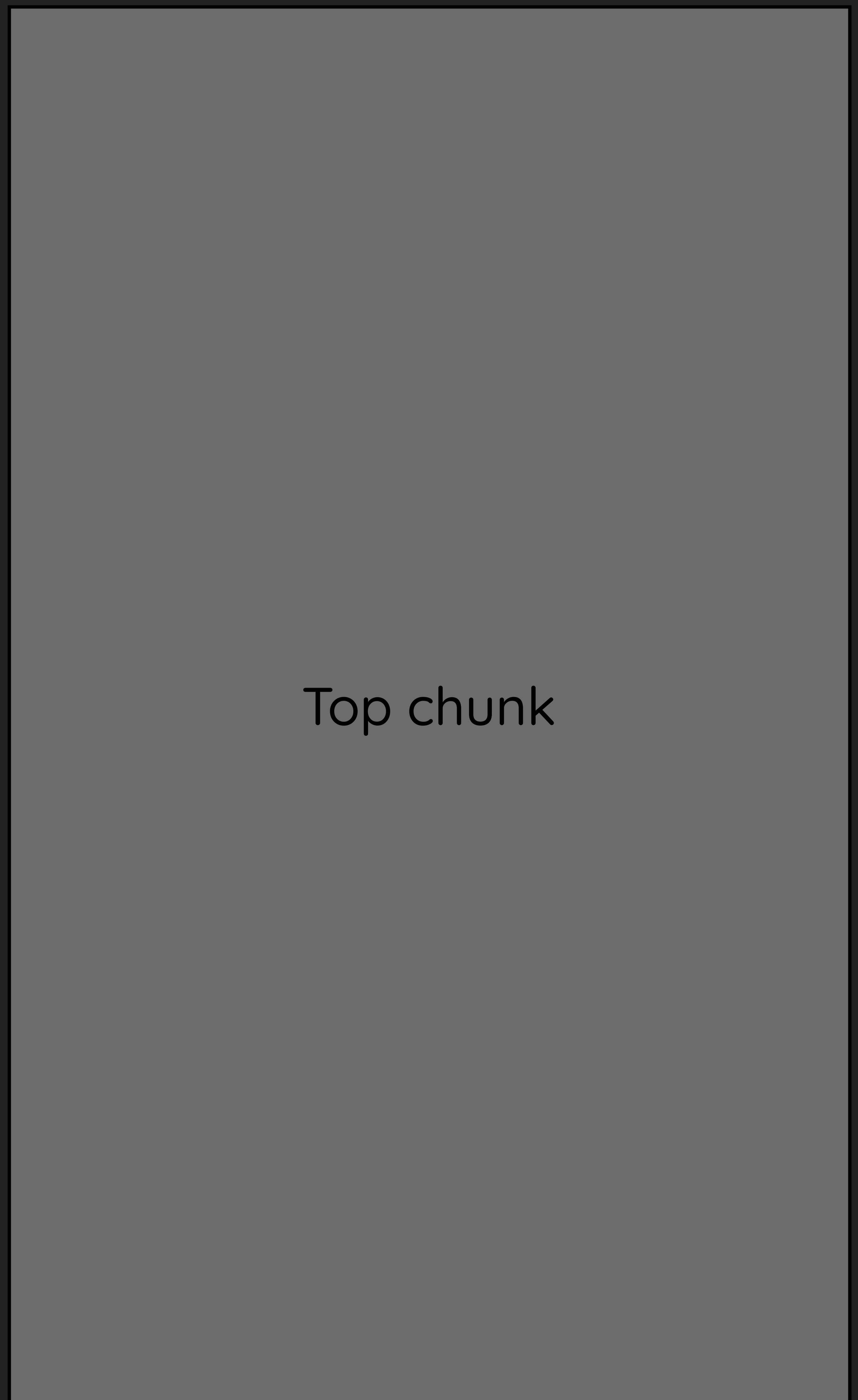
Fastbin attack

Fastbin attack

- fastbin 在檢查 double free 時，只檢查現在 linked list 第一個 chunk 是否等於現在即將要 free 掉的 chunk。
- 若存在 double free ，則可以透過 free(A) free(B) free(A) 的方式繞過此檢查。
- 下次再 malloc 此 fastbin size 時，會拿到 chunk A，而同時 chunk A 依舊存在於 free chunk linked list 中，藉此寫入 data 時修改掉 fd ，接下來連續 malloc 兩次後，第三次則會回傳拿到我們偽造的 address。

Fastbin attack

```
void *A = malloc( 0x68 );

void *B = malloc( 0x68 );

free( A );

free( B );

free( A );

...
```

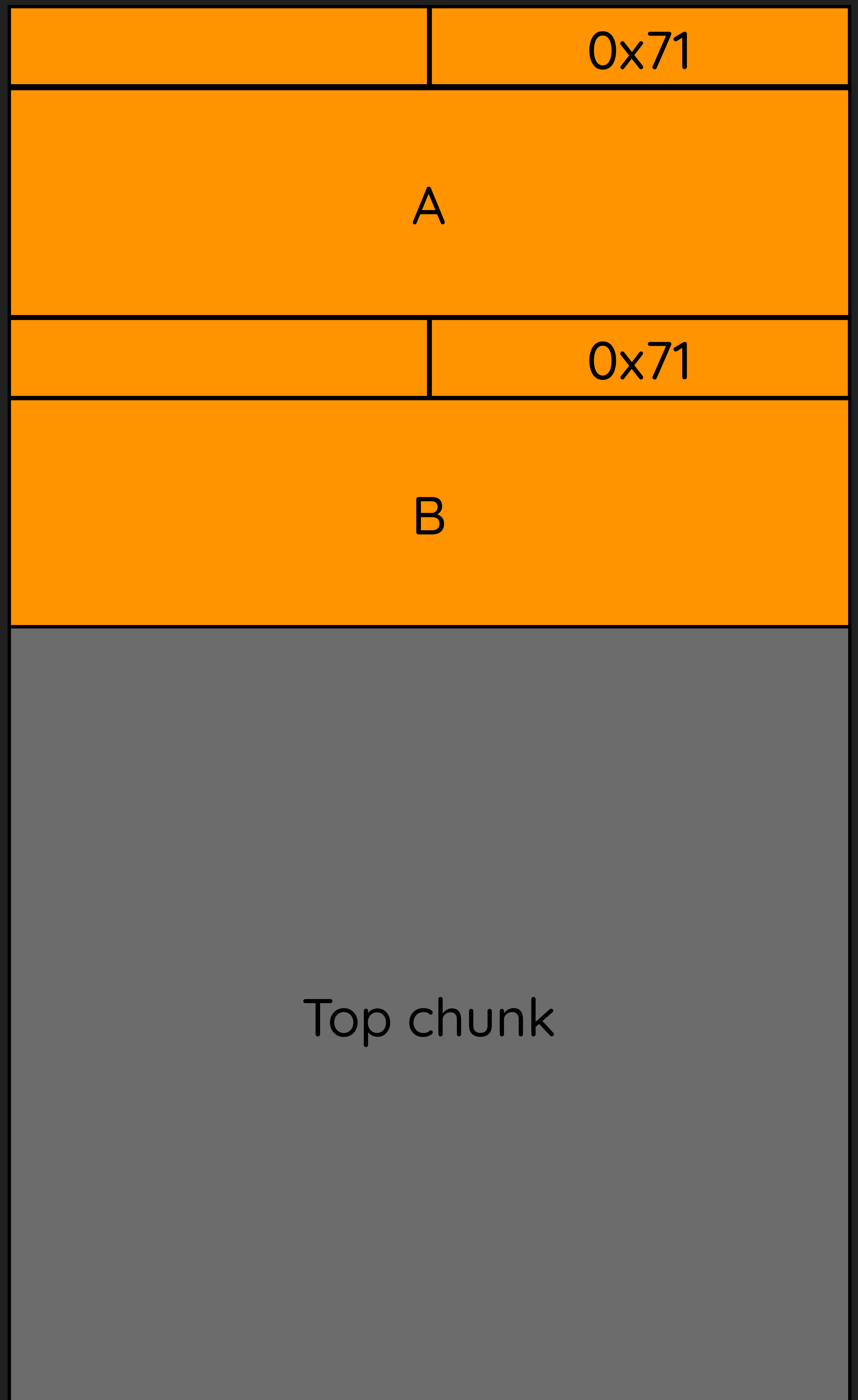


Fastbin attack

```
void *A = malloc( 0x68 );

void *B = malloc( 0x68 );

free( A );

free( B );

free( A );

...
```

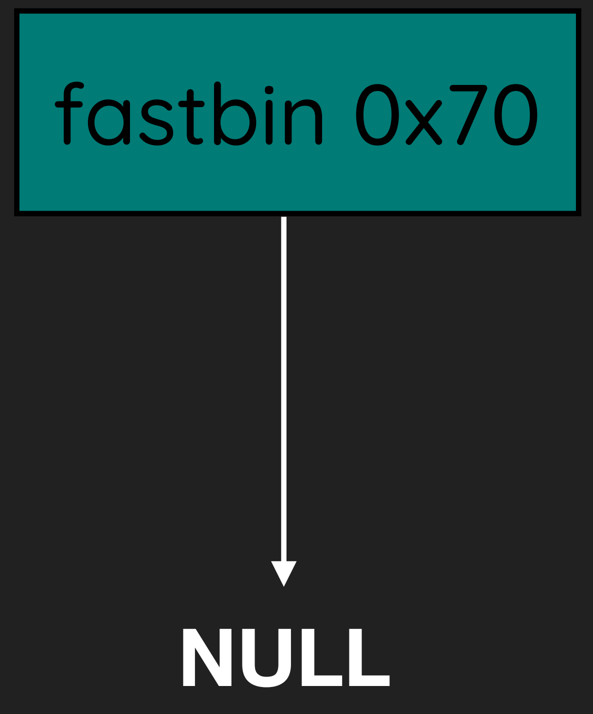
Fastbin attack

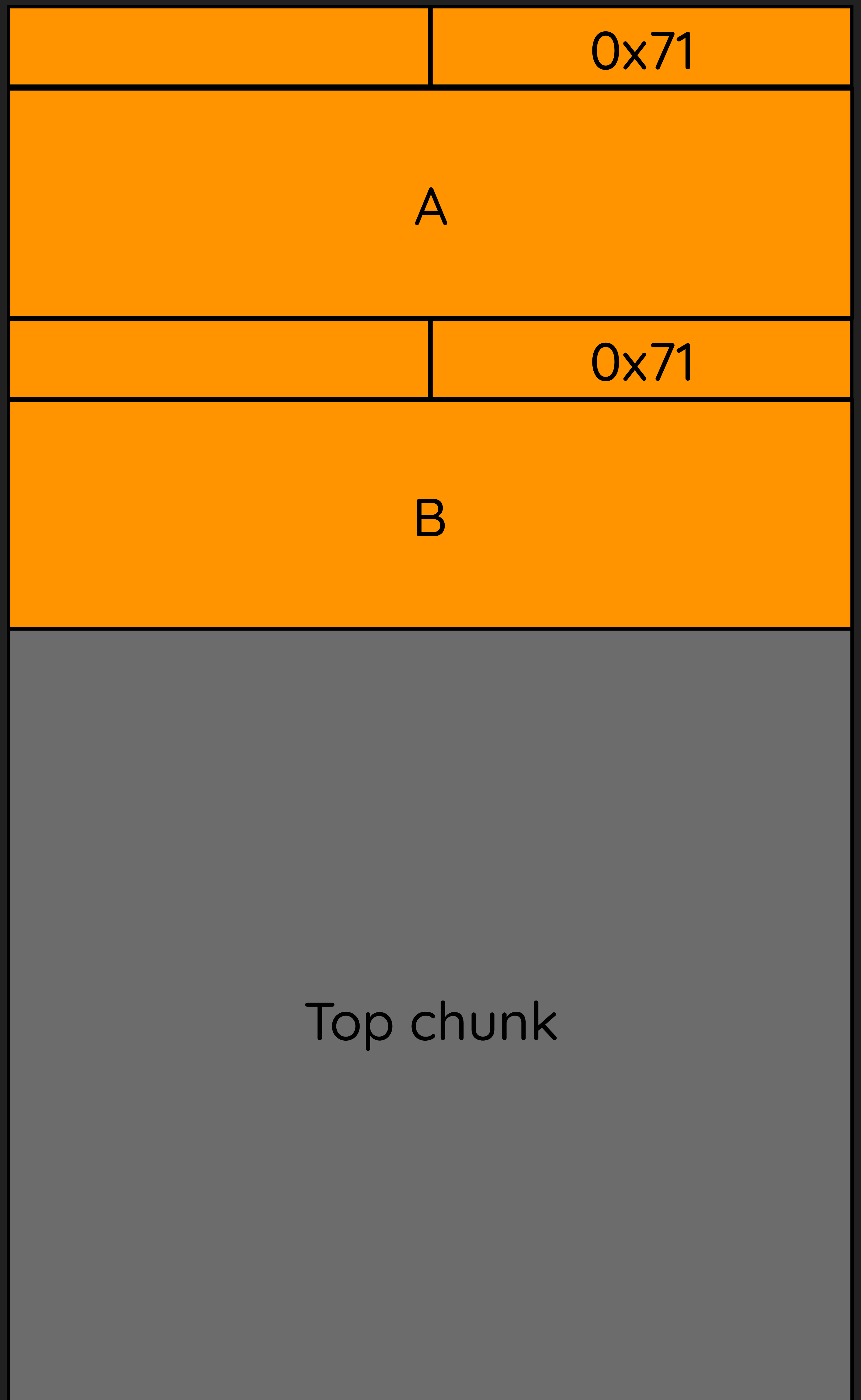
```
void *A = malloc( 0x68 );

void *B = malloc( 0x68 );

free( A );

free( B );

free( A );

...
```



Fastbin attack

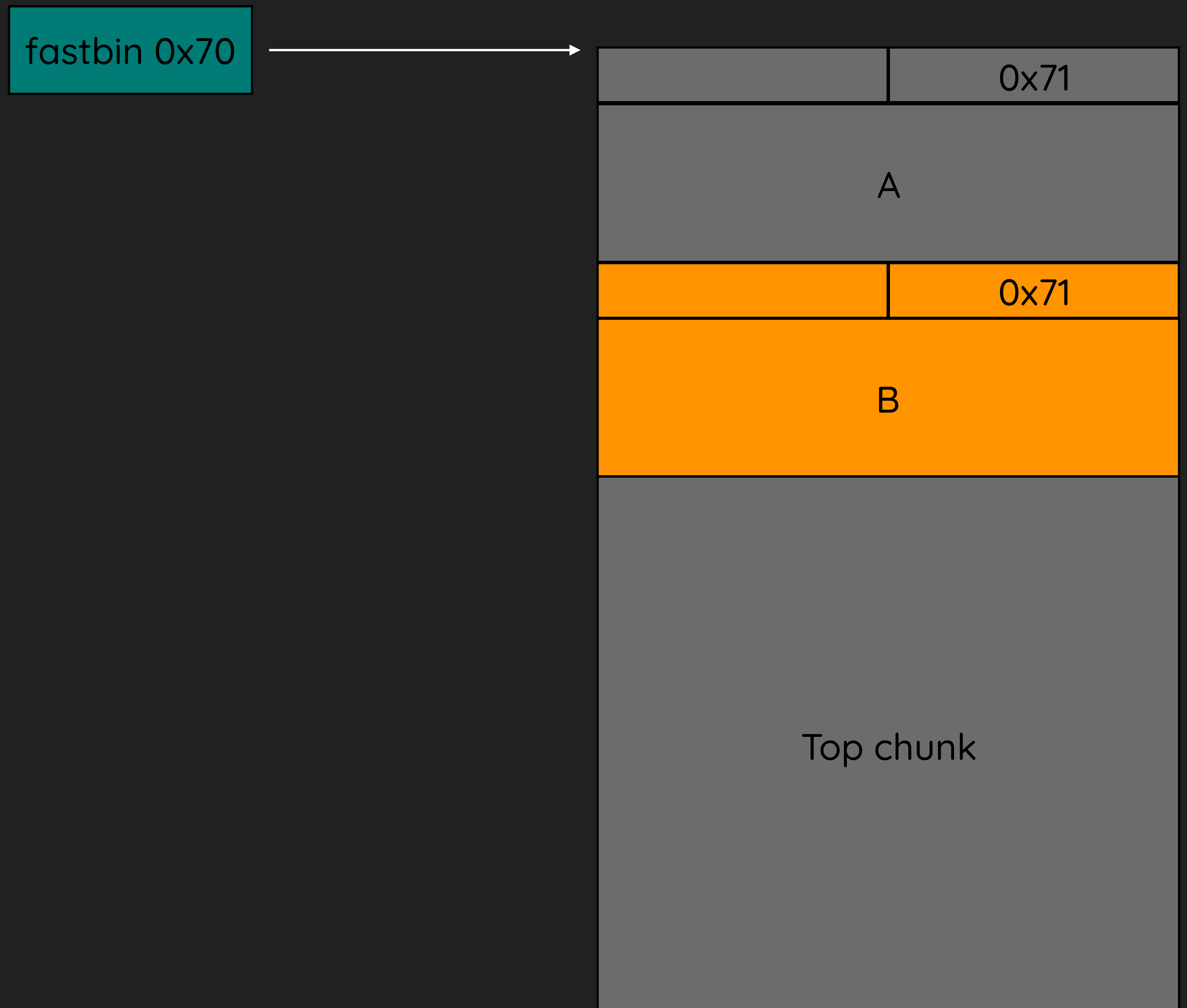
```
void *A = malloc( 0x68 );

void *B = malloc( 0x68 );

free( A );

free( B );

free( A );

...
```





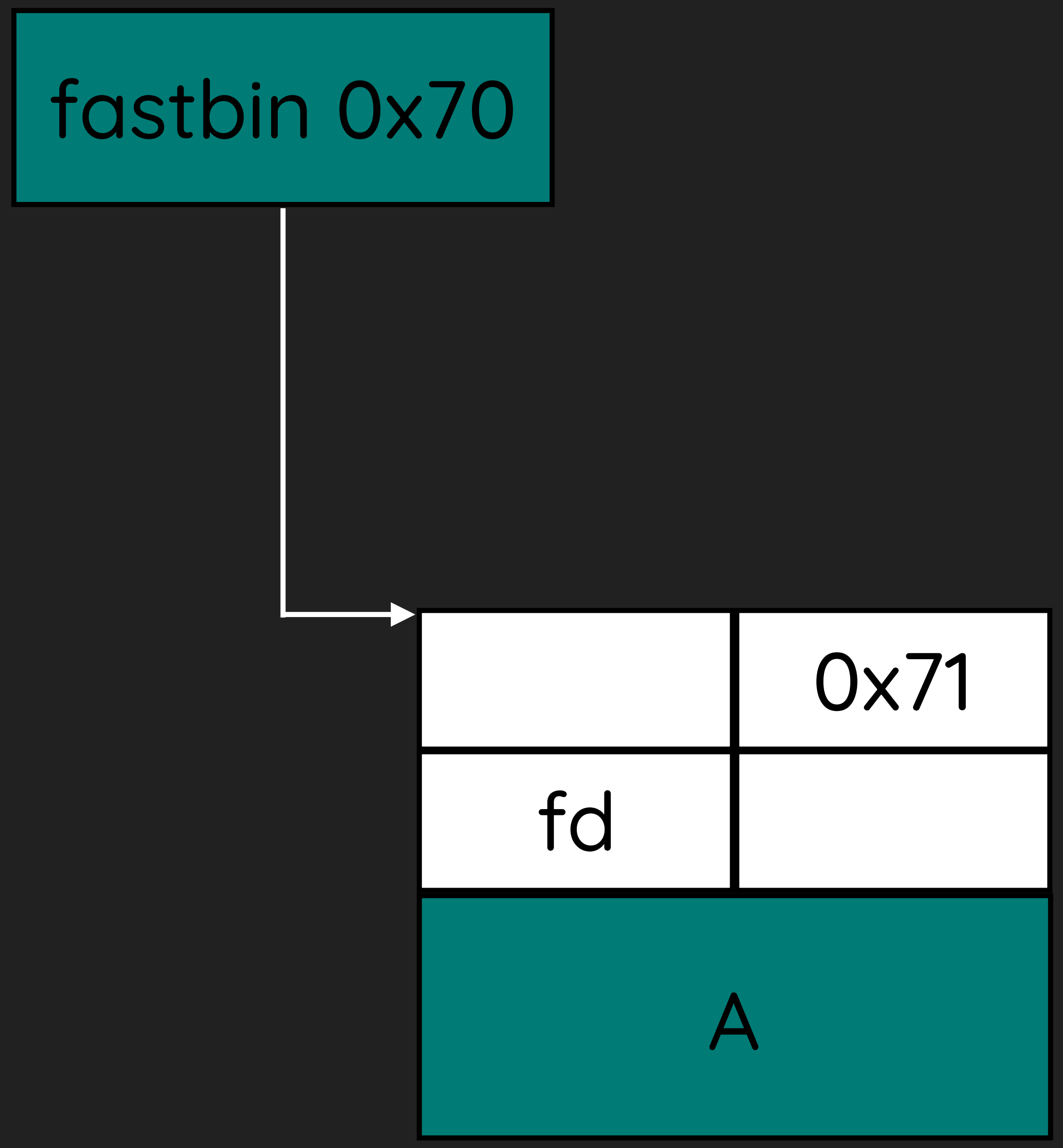
Fastbin attack

```
void *A = malloc( 0x68 );

void *B = malloc( 0x68 );

free( A );

free( B );

free( A );

...
```



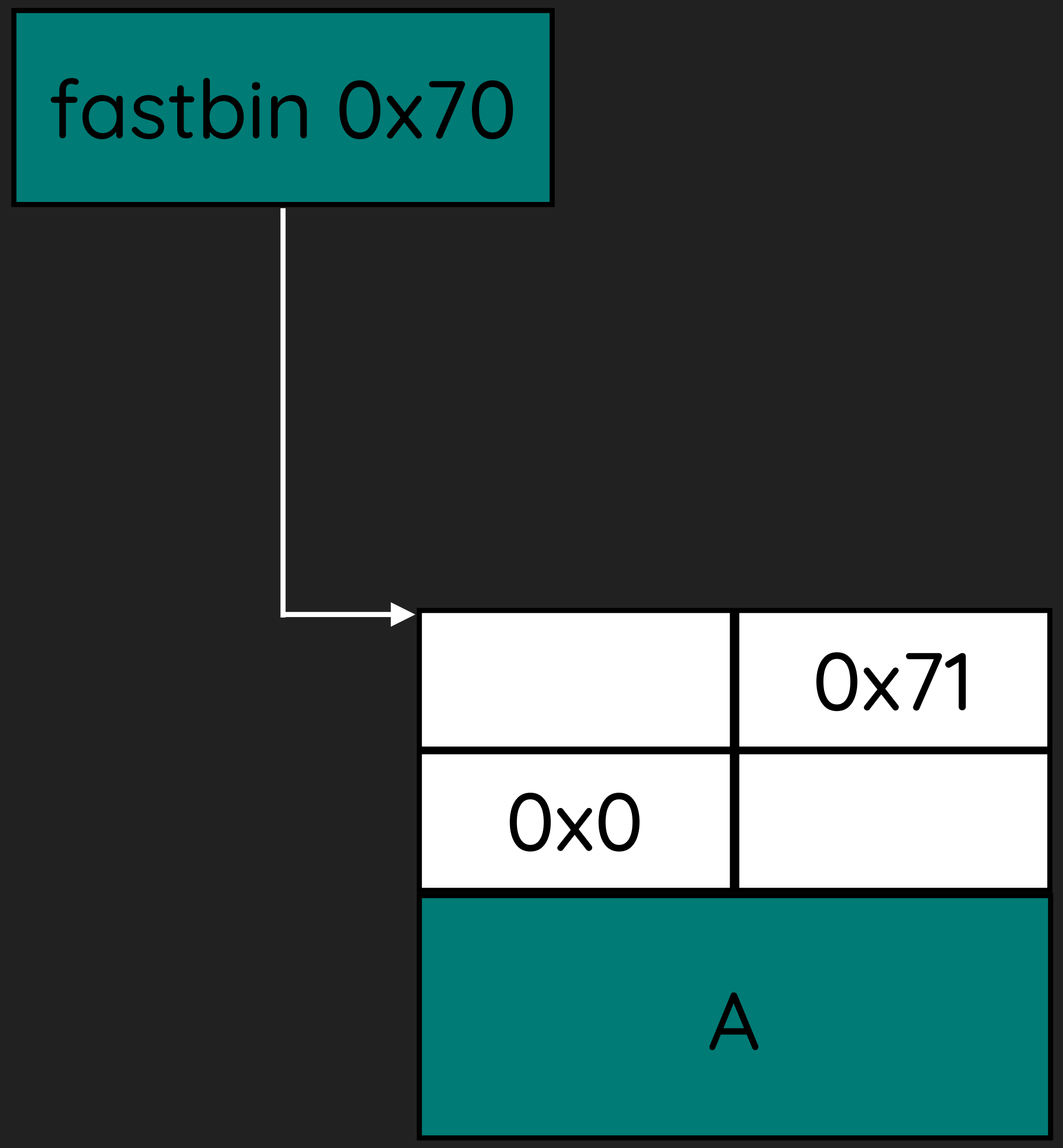
Fastbin attack

```
void *A = malloc( 0x68 );

void *B = malloc( 0x68 );

free( A );

free( B );

free( A );

...
```



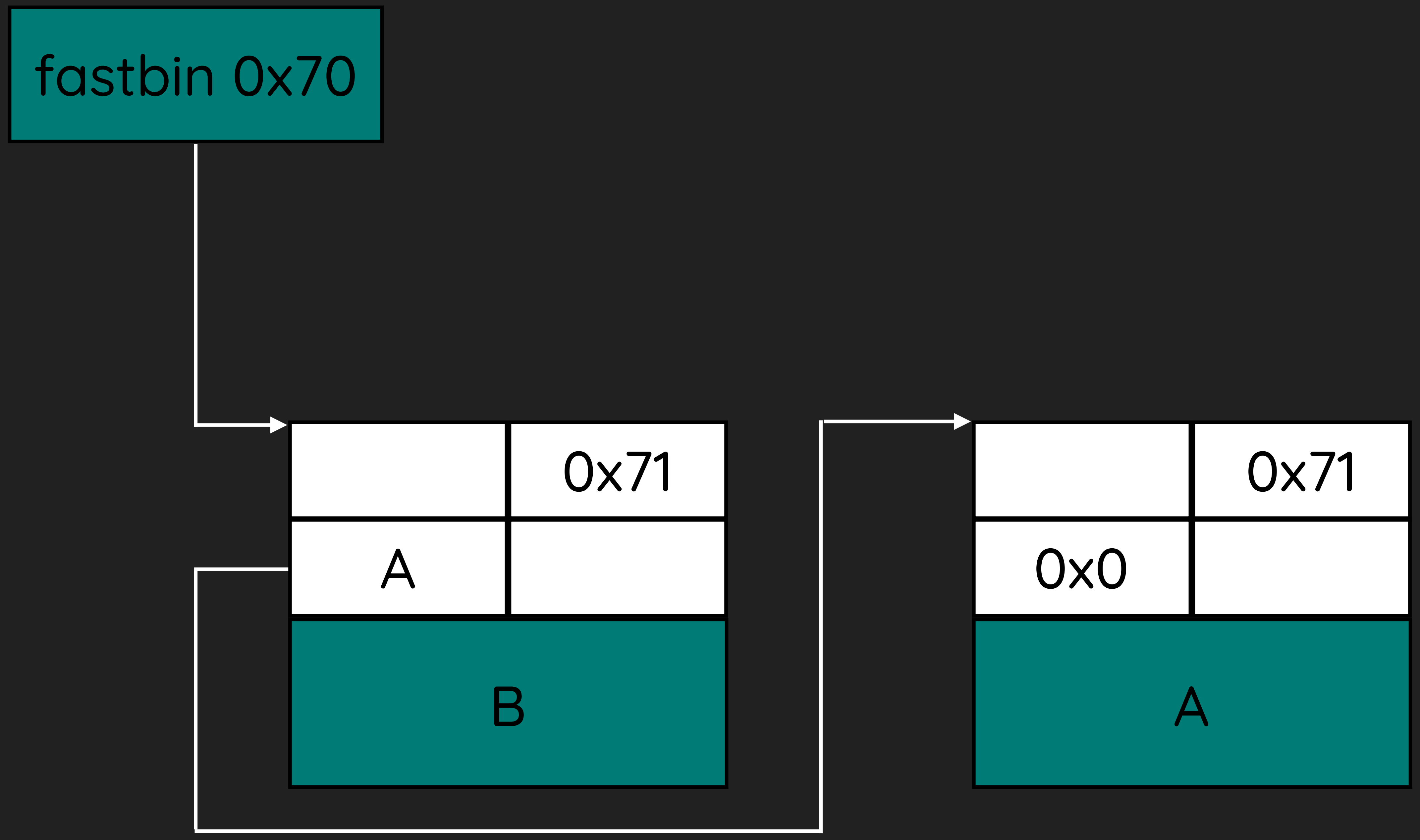
Fastbin attack

```
void *A = malloc( 0x68 );

void *B = malloc( 0x68 );

free( A );

free( B );

free( A );

...
```



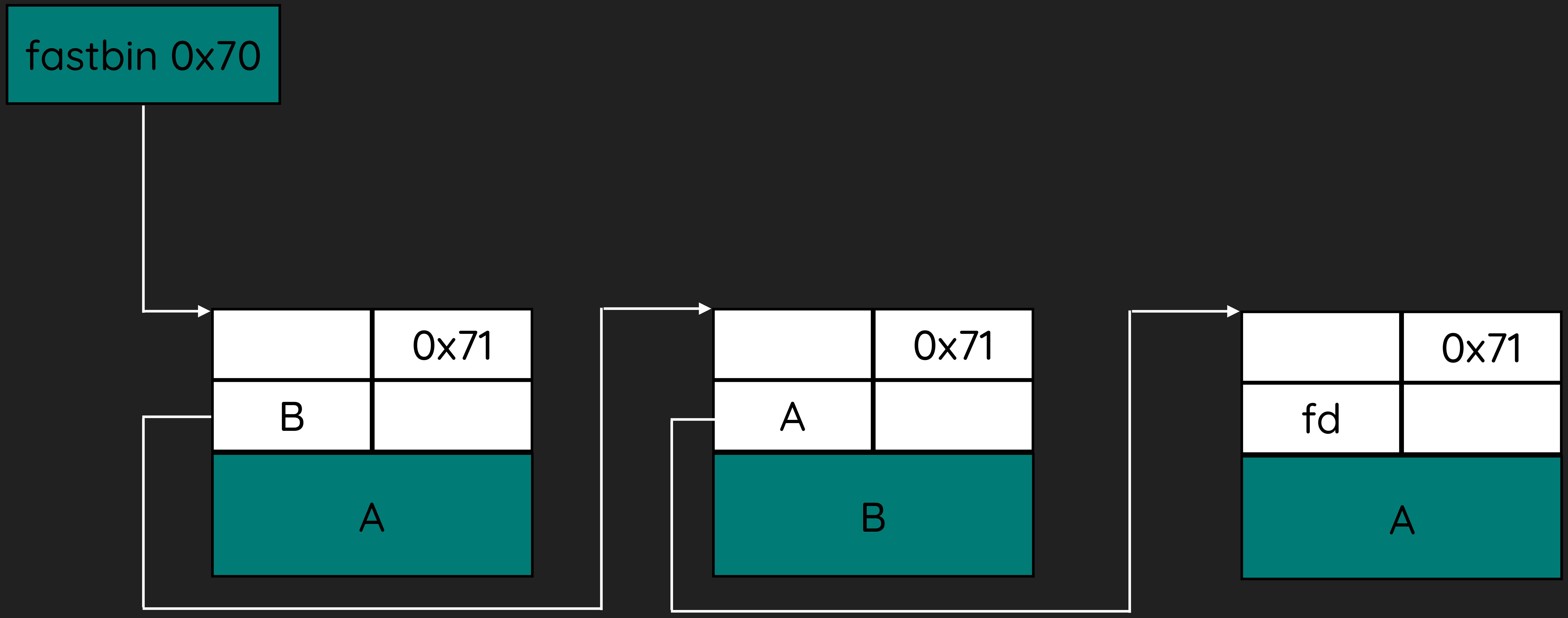
Fastbin attack

```
void *A = malloc( 0x68 );

void *B = malloc( 0x68 );

free( A );

free( B );

free( A );

...
```



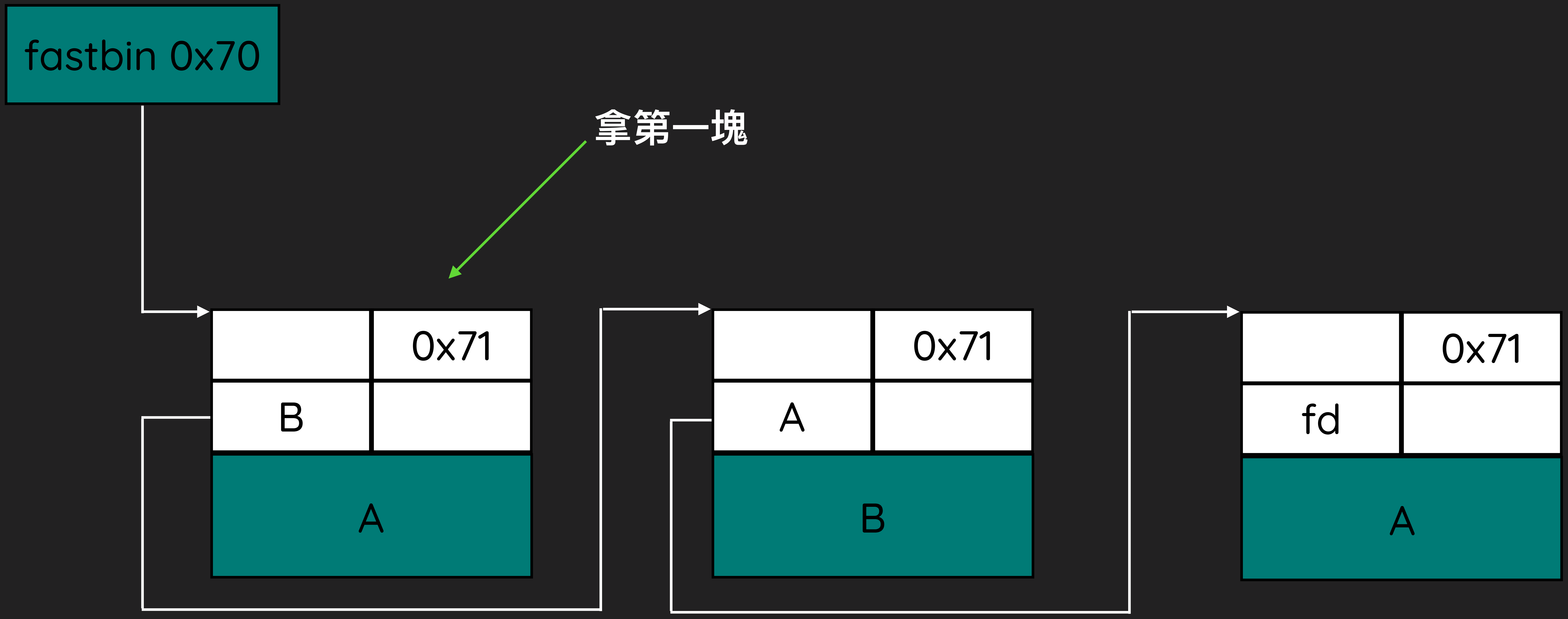
Fastbin attack

```
void *A = malloc( 0x68 );

void *B = malloc( 0x68 );

free( A );

free( B );

free( A );

...
```



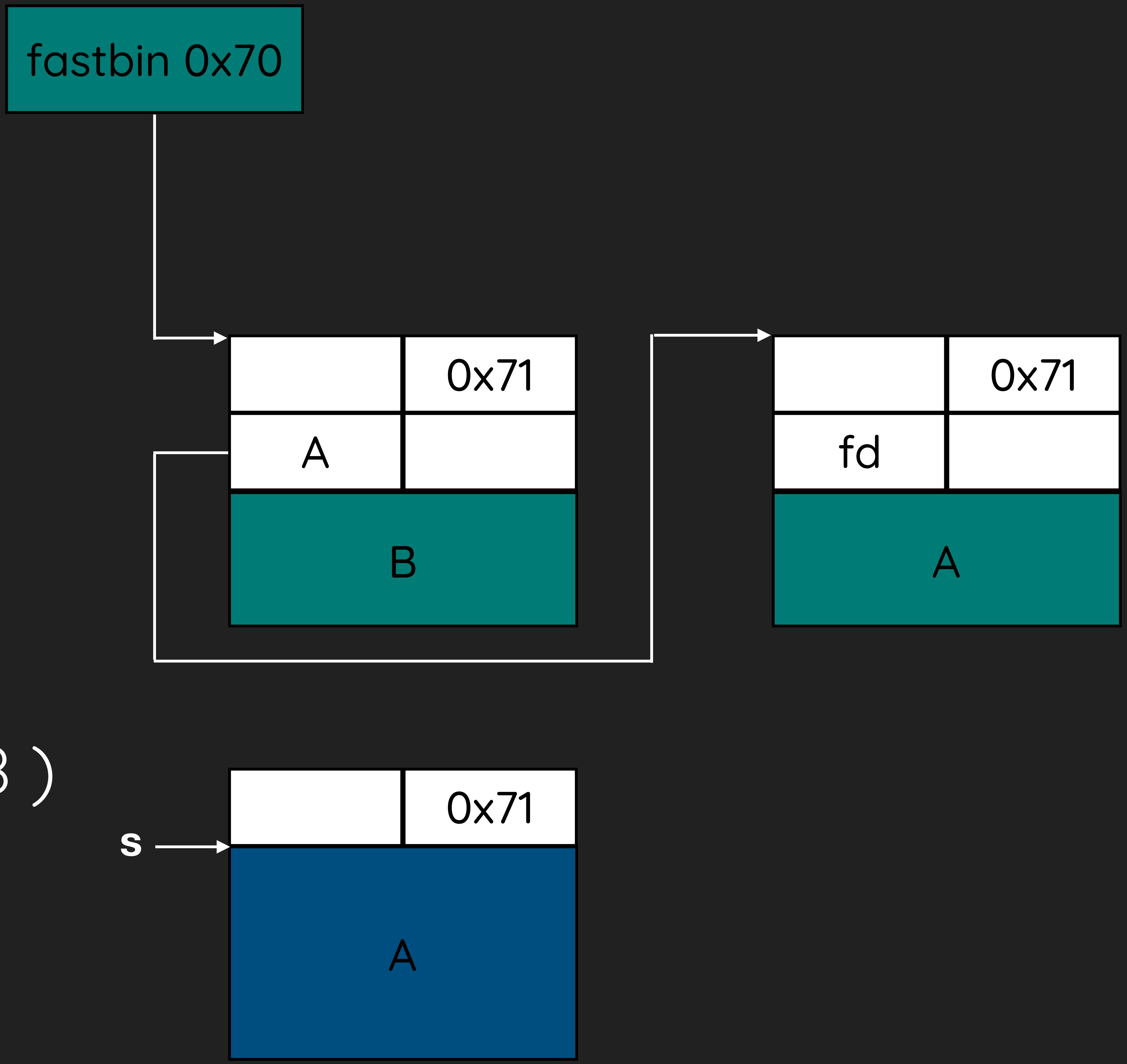
Fastbin attack

```
char *s = malloc( 0x68 );

read( 0 , s , 0x68 );

malloc( 0x68 )

malloc( 0x68 )

void *fake = malloc( 0x68 )
```



Fastbin attack

```
char *s = malloc( 0x68 );

read( 0 , s , 0x68 );

malloc( 0x68 )

malloc( 0x68 )

void *fake = malloc( 0x68 )
```



Fastbin attack

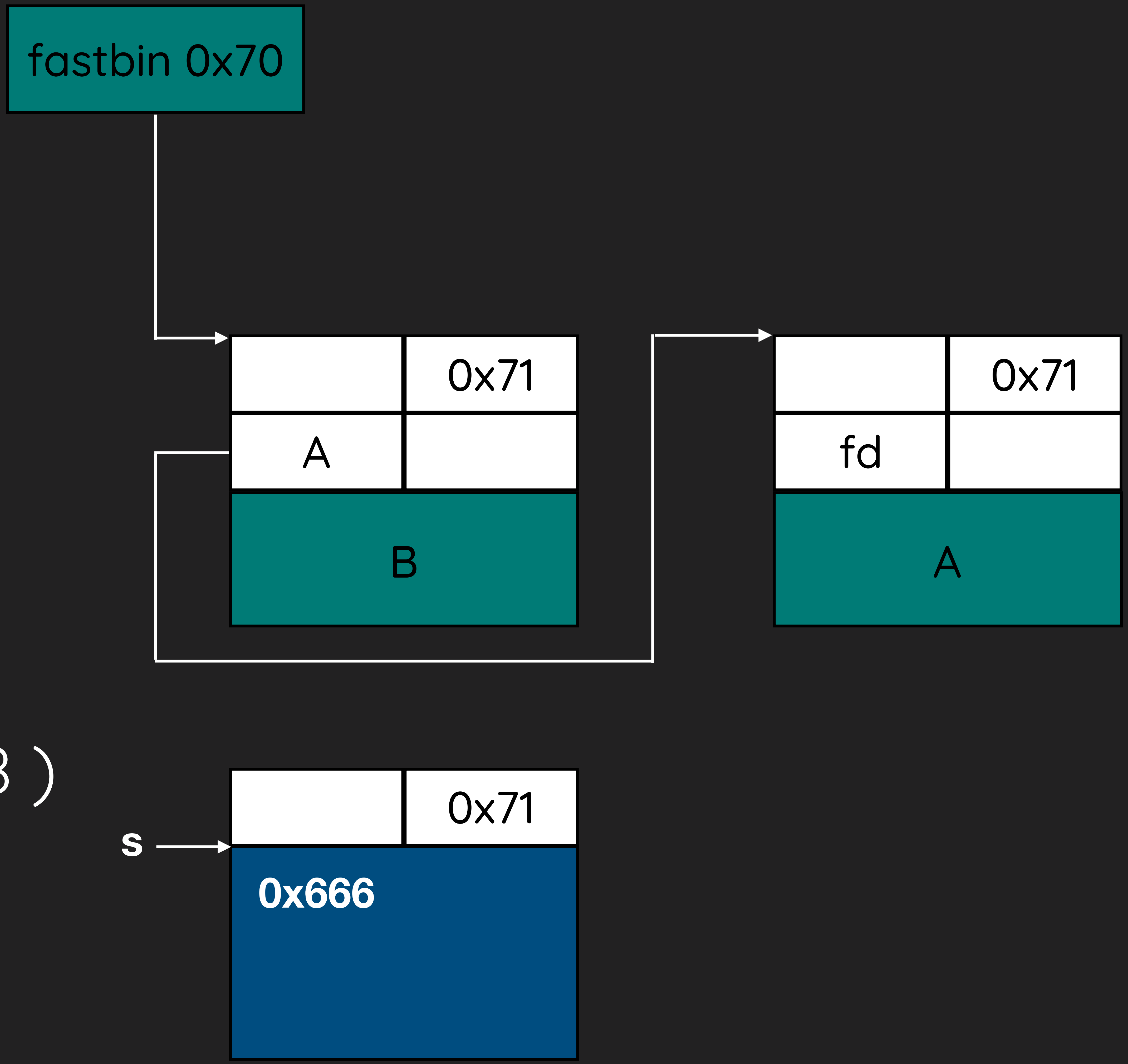
char *s = malloc(0x68);

read(0 , s , 0x68);

malloc(0x68)

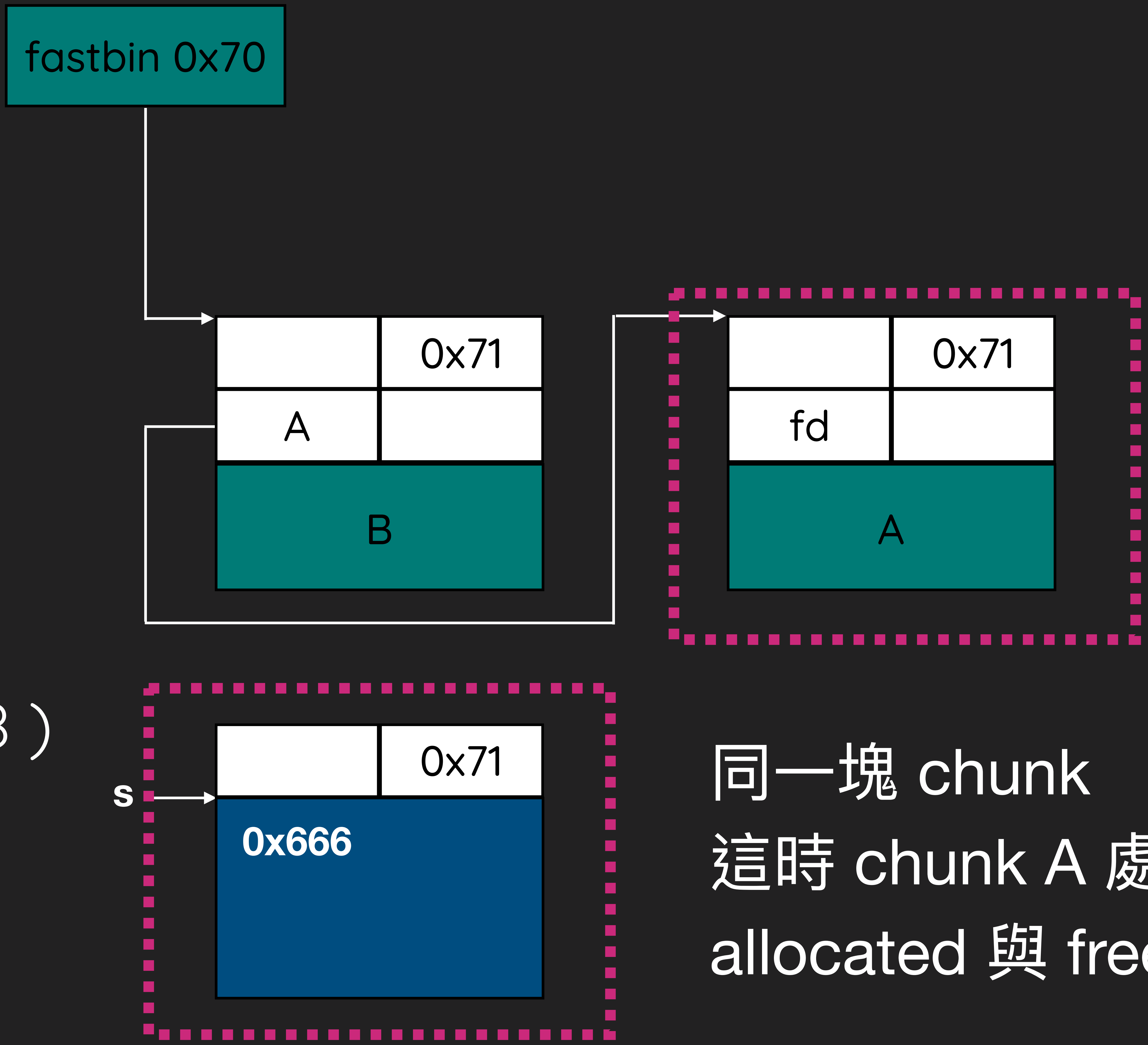
malloc(0x68)

void *fake = malloc(0x68)



Fastbin attack

```
char *s = malloc( 0x68 );

read( 0 , s , 0x68 );

malloc( 0x68 )

malloc( 0x68 )

void *fake = malloc( 0x68 )
```



同一塊 chunk
這時 chunk A 處於
allocated 與 freed 的疊加態 ಠ_ಠ

Fastbin attack

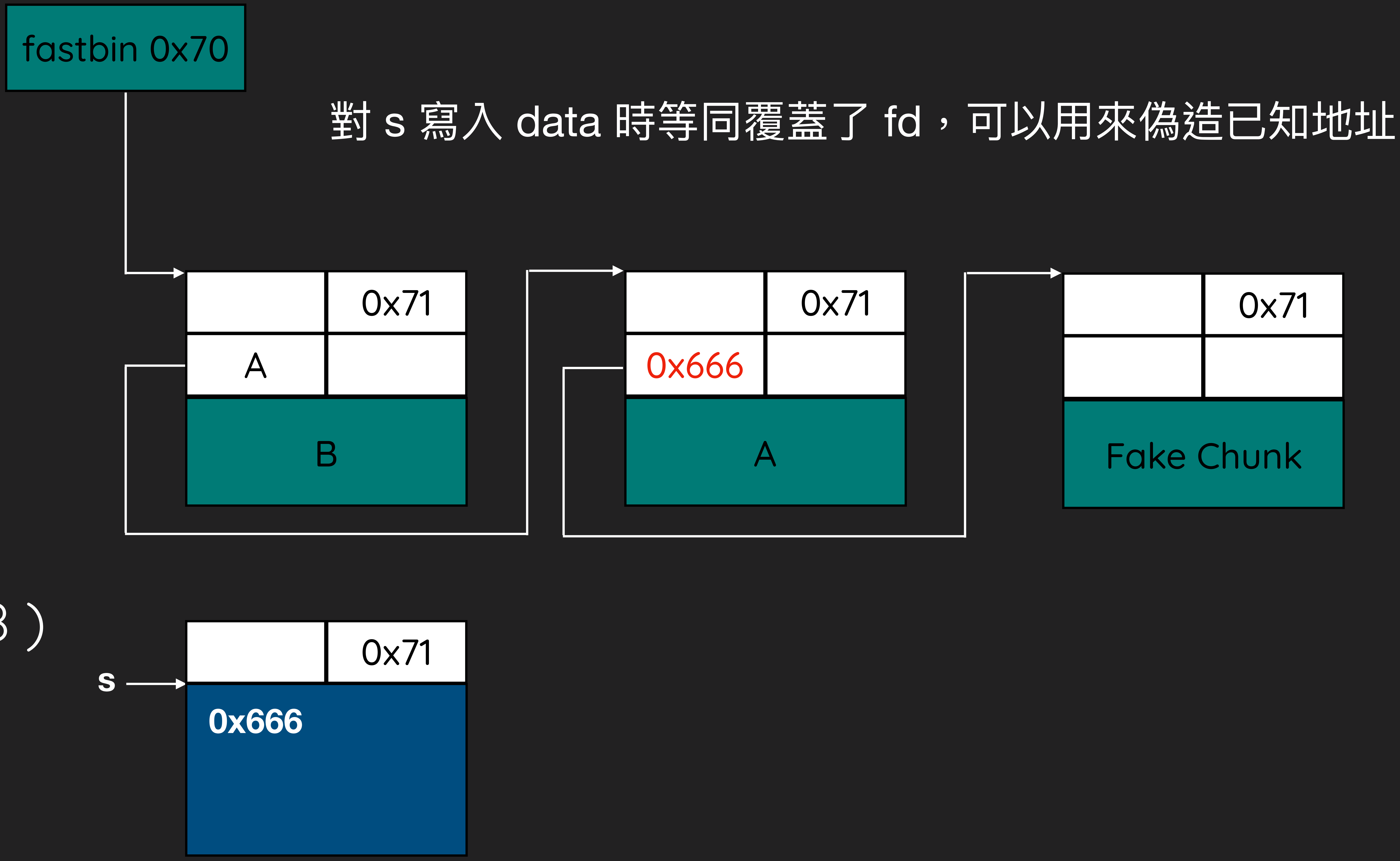
char *s = malloc(0x68);

read(0 , s , 0x68);

malloc(0x68)

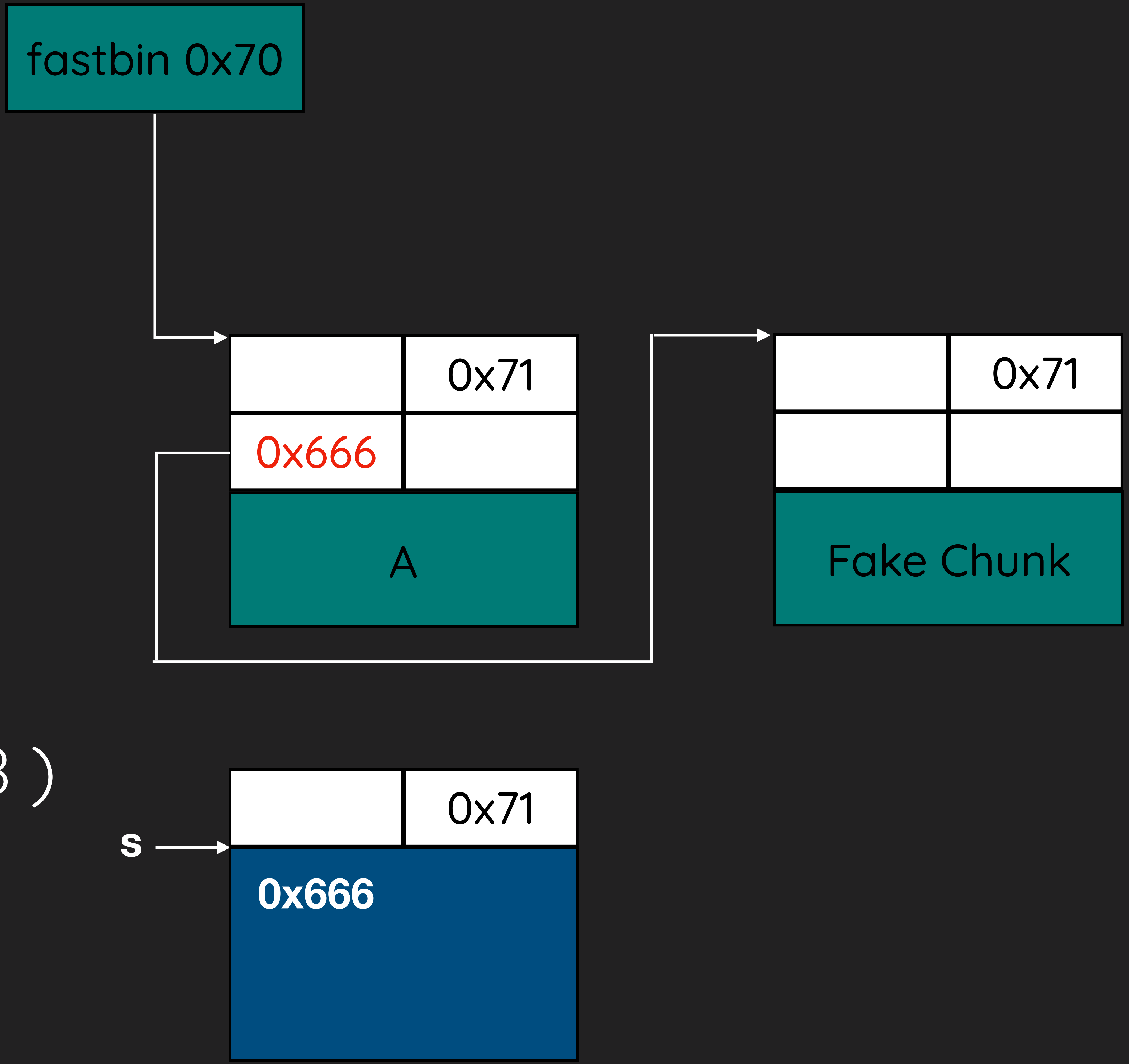
malloc(0x68)

void *fake = malloc(0x68)



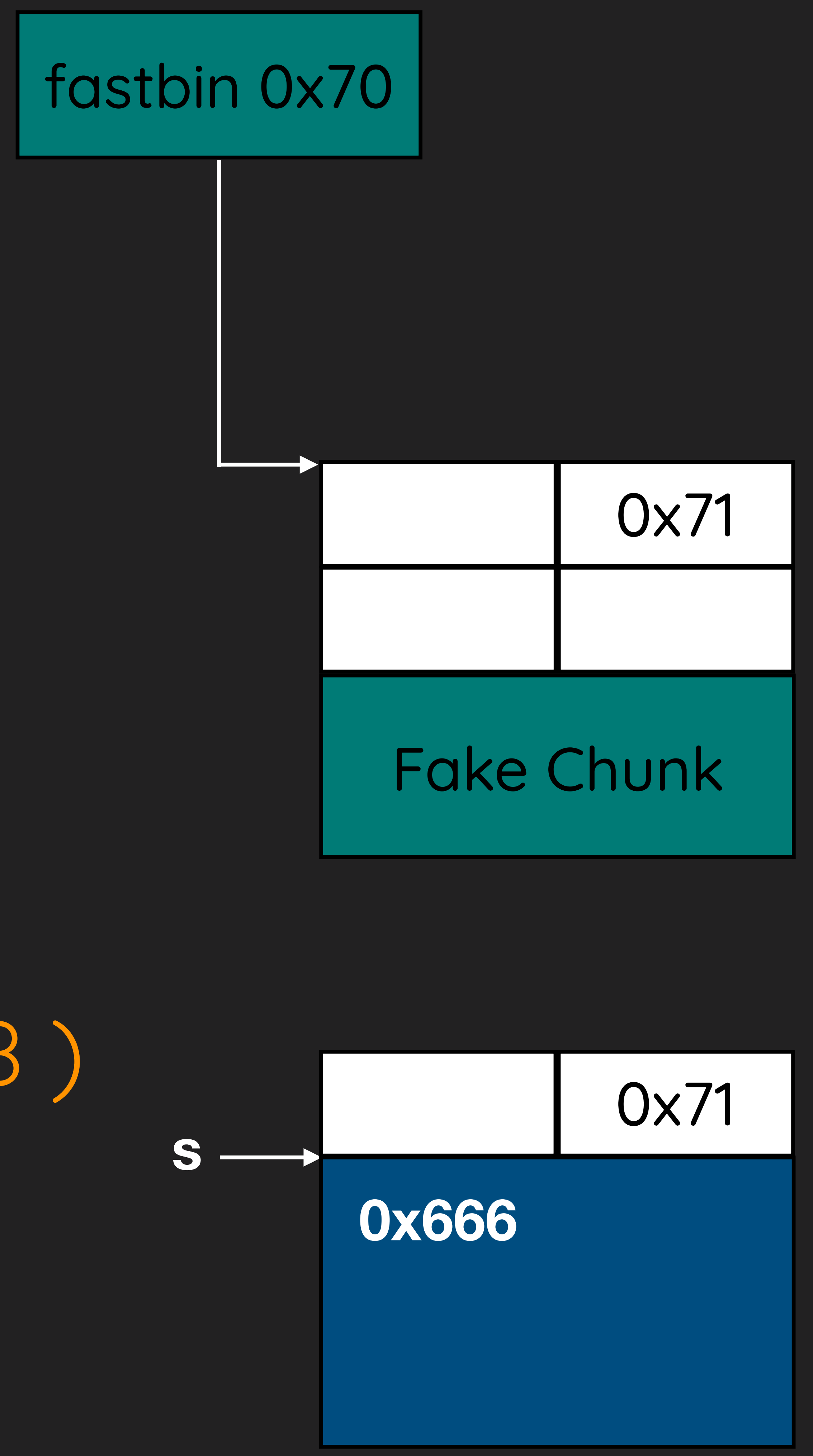
Fastbin attack

```
char *s = malloc( 0x68 );

read( 0 , s , 0x68 );

malloc( 0x68 )

malloc( 0x68 )

void *fake = malloc( 0x68 )
```



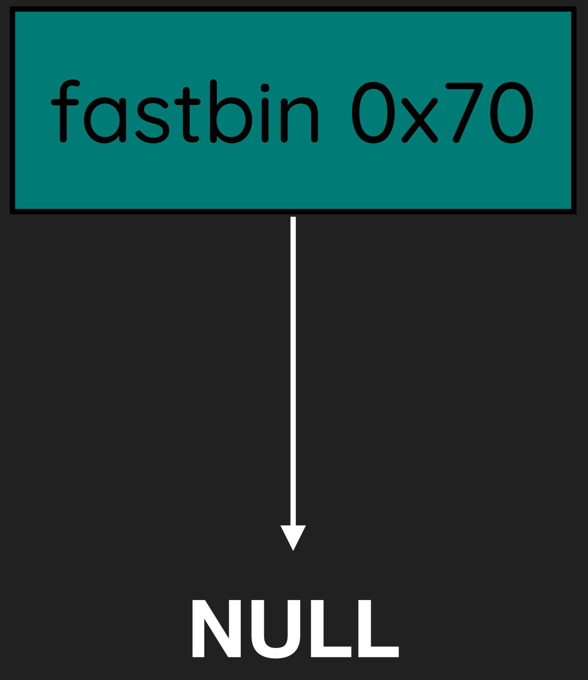
Fastbin attack

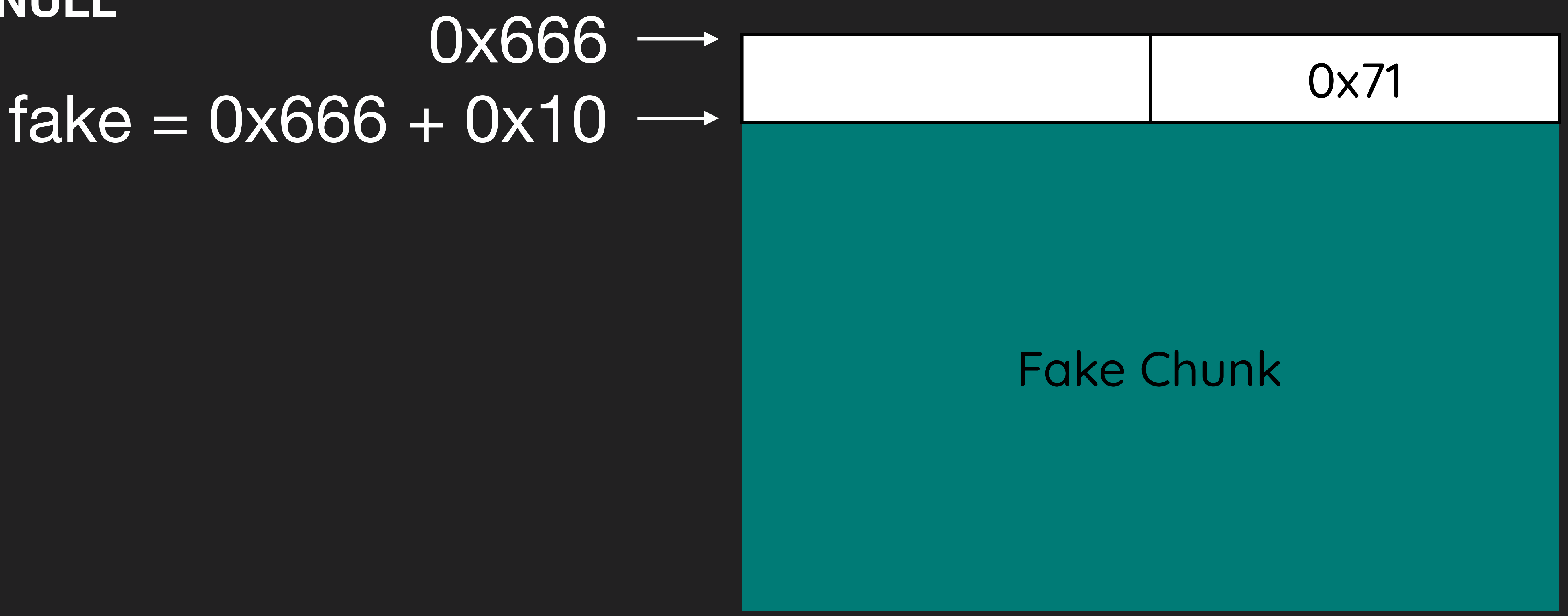
```
char *s = malloc( 0x68 );

read( 0 , s , 0x68 );

malloc( 0x68 )

malloc( 0x68 )

void *fake = malloc( 0x68 )
```



Fastbin attack

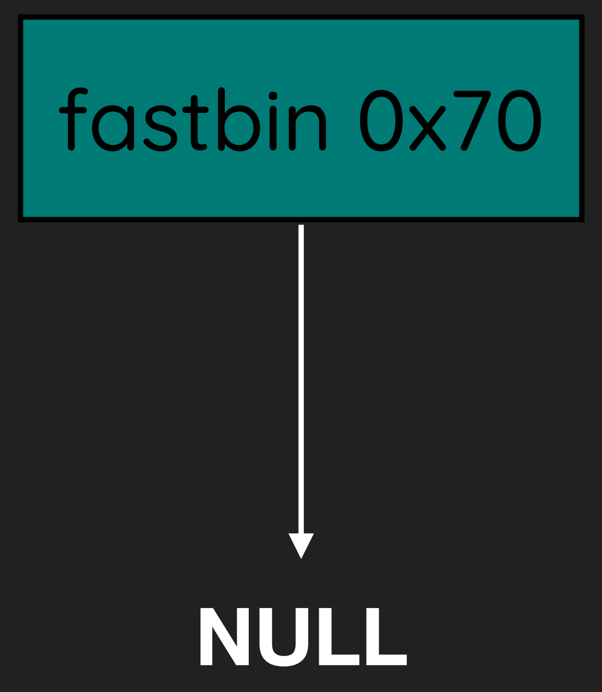
```
char *s = malloc( 0x68 );

read( 0 , s , 0x68 );

malloc( 0x68 )

malloc( 0x68 )

void *fake = malloc( 0x68 )
```



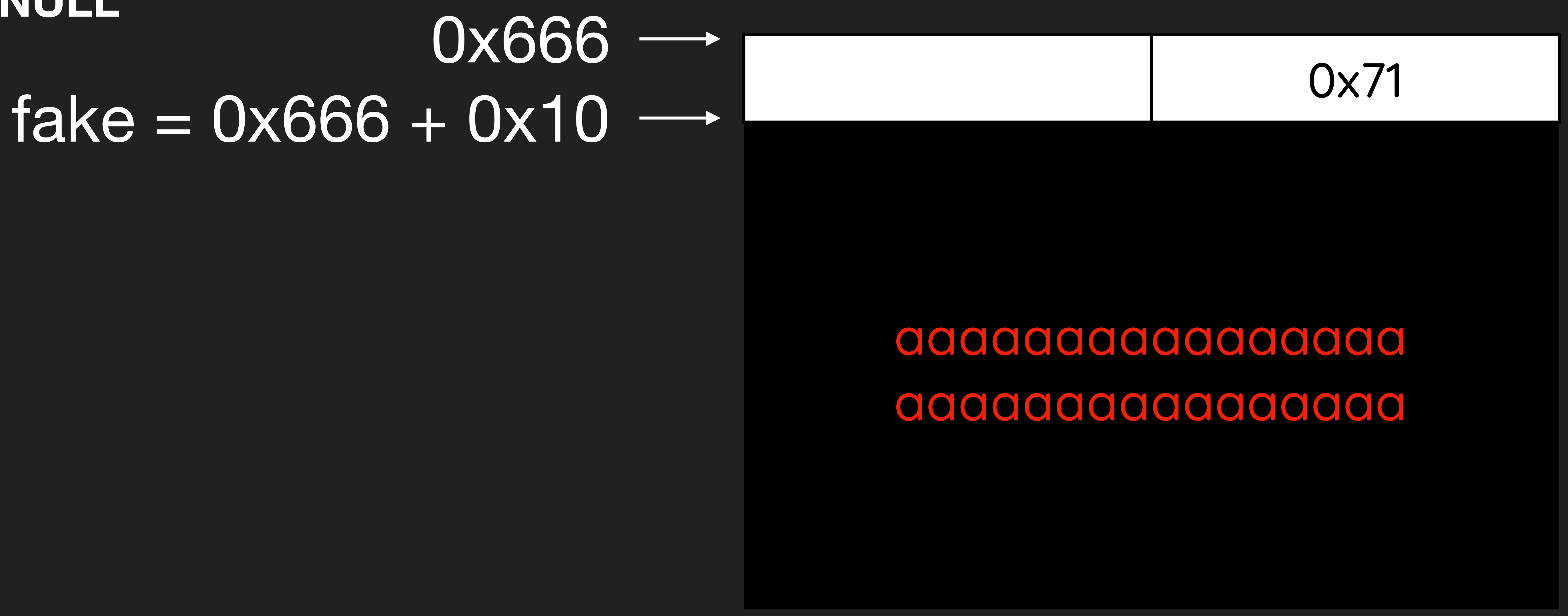


Fastbin attack

```
char *s = malloc( 0x68 );

read( 0 , s , 0x68 );

malloc( 0x68 )

malloc( 0x68 )

void *fake = malloc( 0x68 )
```



Write everywhere!



Fastbin attack

char *s = malloc(0x68);

read(0 , s , 0x68);

malloc(0x68)

malloc(0x68 [redacted]

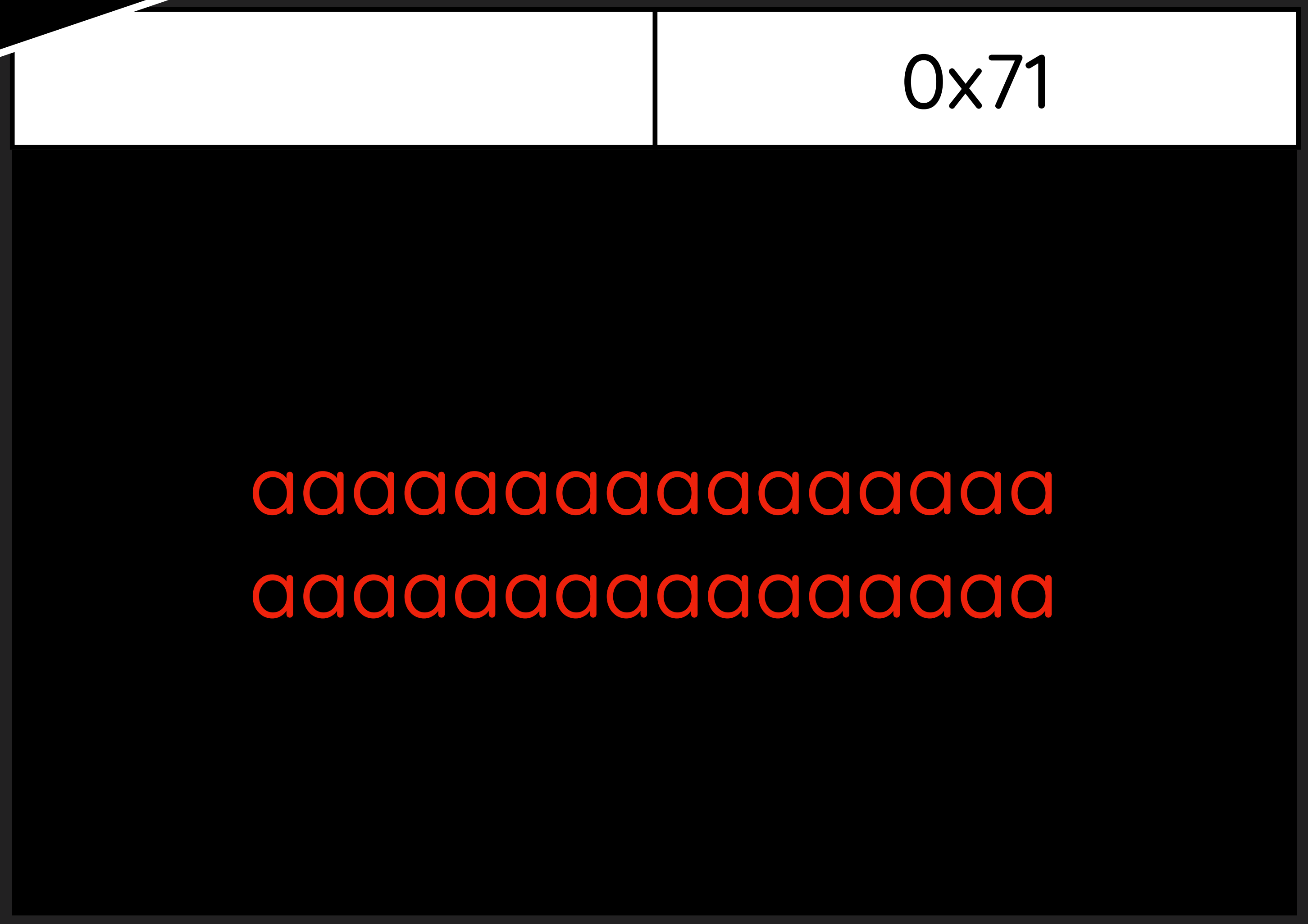
[redacted]malloc(0x68)

fastbin 0x70

NULL

fak[redacted]x10 →

Write every[redacted]



PWNED ☠️

Demo

Fastbin attack - constraints

- malloc 的時候會檢查 chunk size 正不正確。
  - 在 目標地址 + 0x8 的地方偽造size
  - 尋找附近存在正確 size 之目標位置
- 增加可行性的利用技巧
  - address alignment weakness
  - libc address hardcode
  - 檢查 size 時是抓取 4 bytes int。

Demo

Hooks

Hooks

- glibc 中存在許多 function hooks，在攻擊時如果能達到 arbitrary write 或任意寫，hooks 會是一個很好的寫入目標，來做到 control flow。
    - __malloc_hook
    - __free_hook
    - __realloc_hook 等等
- 在執行該 function 時，發現該 function hook 有值，則當作 function pointer 跳上去執行。
- 結合 fastbin attack 拿到位於 hooks 附近位置的 fake chunk，來 overwrite hooks 的值。

Hooks

```c
void *
__libc_malloc (size_t bytes)
{
  mstate ar_ptr;
  void *victim;

  void *(*hook) (size_t, const void *)
    = atomic_forced_read (__malloc_hook);
  if (__builtin_expect (hook != NULL, 0))
    return (*hook)(bytes, RETURN_ADDRESS (0));

  ...

}
```

Hooks

```
void *
__libc_malloc (size_t bytes)
{
  mstate ar_ptr;
  void *victim;

  void *(*hook) (size_t, const void *)
    = atomic_forced_read (__malloc_hook);
  if (__builtin_expect (hook != NULL, 0))
    return (*hook)(bytes, RETURN_ADDRESS (0));

  ...

}
```

Hooks

- __malloc_hook = 0xc0ffee
- trigger malloc()
- rip = 0xc0ffee

```
void *
__libc_malloc (size_t bytes)
{
  mstate ar_ptr;
  void *victim;

  void *(*hook) (size_t, const void *)
    = atomic_forced_read (__malloc_hook);
  if (__builtin_expect (hook != NULL, 0))
    return (*hook)(bytes, RETURN_ADDRESS (0));

  ...

}
```

One gadget

magic gadget

One gadget

- 跳過去即執行 execve("/bin/sh" , argv[] , envp[])，跳上去即開 shell！
  - magic gadget
- 有一些前提 (constraints) 需要滿足。
- 常搭配 hooks 來使用。
- https://github.com/david942j/one_gadget

One gadget

- __malloc_hook = one_gadget
- trigger malloc()

```
void *
__libc_malloc (size_t bytes)
{
  mstate ar_ptr;
  void *victim;

  void *(*hook) (size_t, const void *)
    = atomic_forced_read (__malloc_hook);
  if (__builtin_expect (hook != NULL, 0))
    return (*hook)(bytes, RETURN_ADDRESS (0));

  ...

}
```

One gadget

- rip = one_gadget
- Shell!

```
void *
__libc_malloc (size_t bytes)
{
  mstate ar_ptr;
  void *victim;

  void *(*hook) (size_t, const void *)
    = atomic_forced_read (__malloc_hook);
  if (__builtin_expect (hook != NULL, 0))
    return (*hook)(bytes, RETURN_ADDRESS (0));

  ...

}
```

One gadget

- rip = one_gadget
- Shell!

```
void *
__libc_malloc (size [redacted]
{
  mstate [redacted]
[redacted] *(*hook) (size_t, const void *)
    = atomic_forced_read (__malloc_hook);
  if (__builtin_expect (hook != NULL, 0))
    return (*hook)(bytes, RETURN_ADDRESS (0));

  ...

}
```

PWNED ☠️

Demo

Tcache

per-thread cache

Tcache

- glibc >= 2.26
    - Ubuntu 17.10 之後
- 新的機制，提升 performance

Tcache

```c
typedef struct tcache_perthread_struct
{
  char counts[TCACHE_MAX_BINS];
  tcache_entry *entries[TCACHE_MAX_BINS];
} tcache_perthread_struct;
```

- 第一次 malloc 時，會先分配一塊記憶體，存放 tcache_perthread_struct，一個 thread 一個 tcache_perthread_struct。
- 根據 size 分為不同大小的 tcache
- smallbin 大小範圍的 chunk 都會使用 tcache

Tcache

```c
typedef struct tcache_entry
{
  struct tcache_entry *next;
} tcache_entry;
```

- 以 fastbin 來說，free 的時候不會直接放到 fastbin 裡，而是放到對應的 size 的 Tcache 中，當滿 7 個時，再 free 才會再放至 fastbin 中。
- fastbin 的 fd 是指向整個 chunk 的頭，也就是 header，而 tcache fd 則是指向 user data。

Tcache

- malloc 時優先從 tcache 取出，tcahce 為空才會從原本的 bin 中開始找。
- 若 tcache 為空，而原本的 bin 中有剛好大小的 chunk 時，會從 bin 中填補至 tcache 中直到填滿，再從 tcache 中取出，tcache 中的順序會與 bin 中相反。
    - 對於 fastbin 來說，會先將 bin 中第一塊取出，才將後面做填補。

Tcache

- 提升效能，而降低了安全性
  - 沒有檢查 double free
  - malloc 時沒有檢查 size 是否合法
- 不需要偽造 chunk，偽造 size，就能拿到任意記憶體位置。
- 有許多進階玩法跟技巧。
  - 如透過漏洞掌控整個 tcache_perthread_struct

Tcache

- 提升效能，而降低了安全性
  - 沒有檢查 double free 😮
  - malloc 時沒有檢查 size 是否合法
- 不需要偽造 chunk，偽造 size，就能拿到任意記憶體位置。
- 有許多進階玩法跟技巧。
  - 如透過漏洞掌控整個 tcache_perthread_struct

Demo

Some Heap Technique

Heap Overlap

Heap overlap

- 可能一開始漏洞無法直接性做到太多事情。
- 透過各式偽造 chunk size 的方式，以及玩弄 malloc free 的流程，使得不同的 chunk 發生重疊 (heap overlap) 的情形。
- 假設 chunk A 的 data 與 chunk B 的不可寫部分重疊
    - chunk B 可能是一個 struct，有如 char* data pointer，亦或是 function pointer，則可以透過 chunk A 來偽造，overwrite data pinter 達到任意讀寫，overwrite function pinter 則可以 hijack control flow。
    - 可以更進一步偽造 chunk header，偽造 heap chunk，玩弄記憶體管理機制。

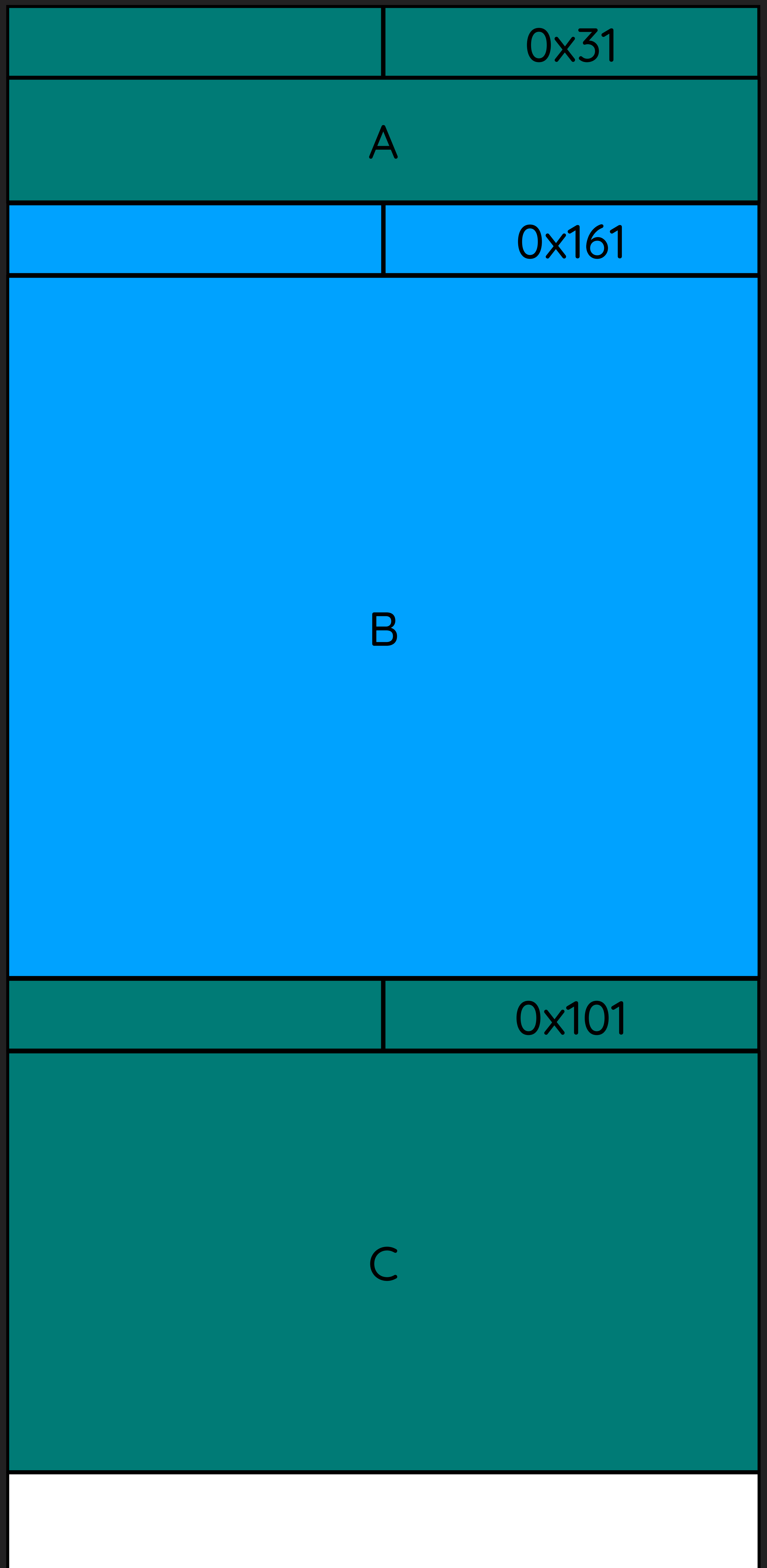
Heap overlap

- 舉個例子，只有一個 byte 的 heap overflow，且該 byte 的值不可控只能是 NULL byte 0x0。
- off-by-one null byte overflow
    - "The poisoned NUL byte" - Google Project Zero
        - https://googleprojectzero.blogspot.com/2014/08/the-poisoned-nul-byte-2014-edition.html
- 某種程度上算很容易發生的漏洞，如宣告 size 剛剛好的字元陣列，一些 libc function 操作會自動 append null byte 或自行邊界操作不當等等。

「星星之火，可以燎原。」

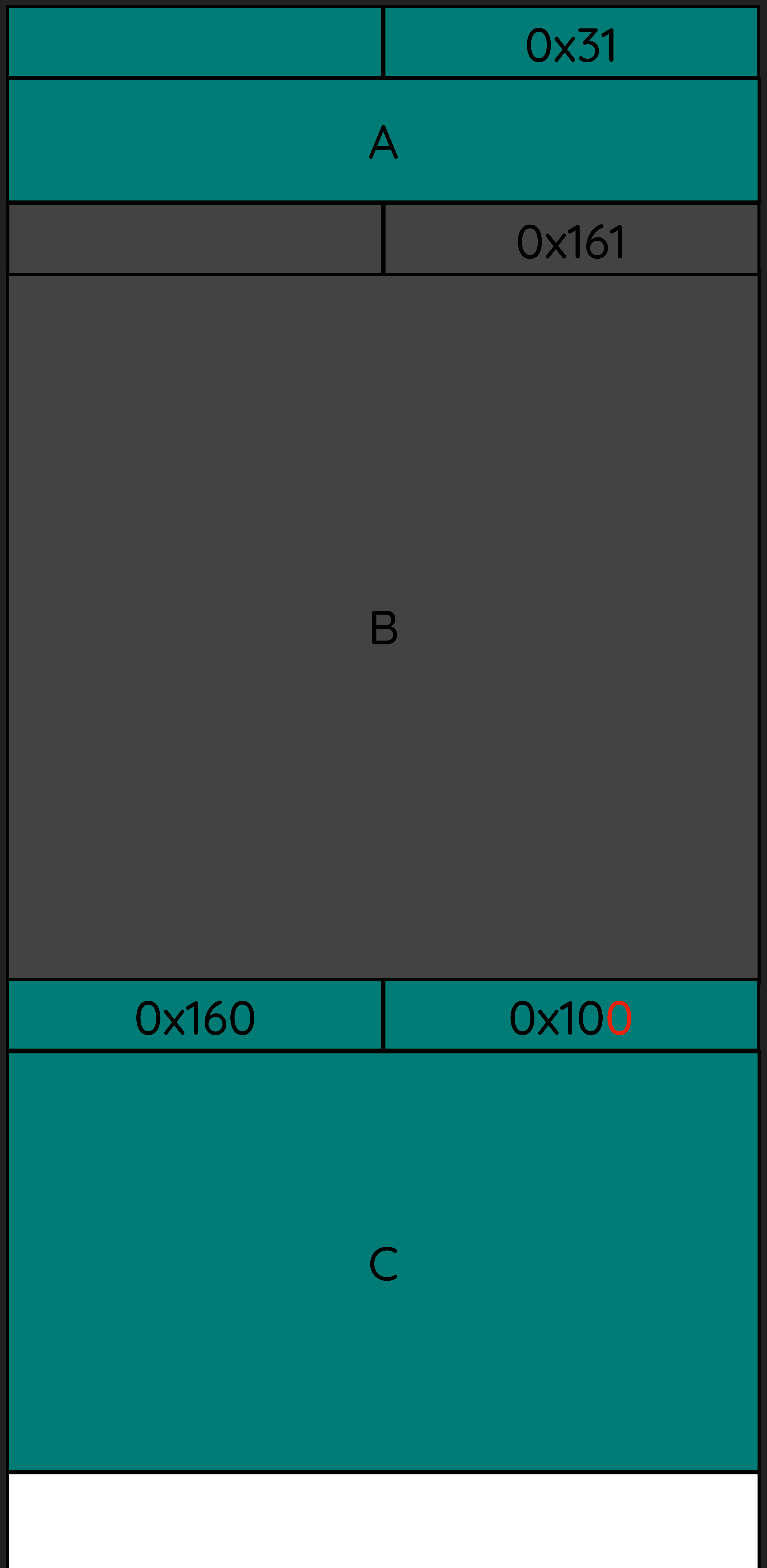
Heap overlap

- free(B)
- free(A)
- malloc(0x28)
- malloc(0x88)
- malloc(0x48)
- free(B)
- free(C)
- malloc(0x258)



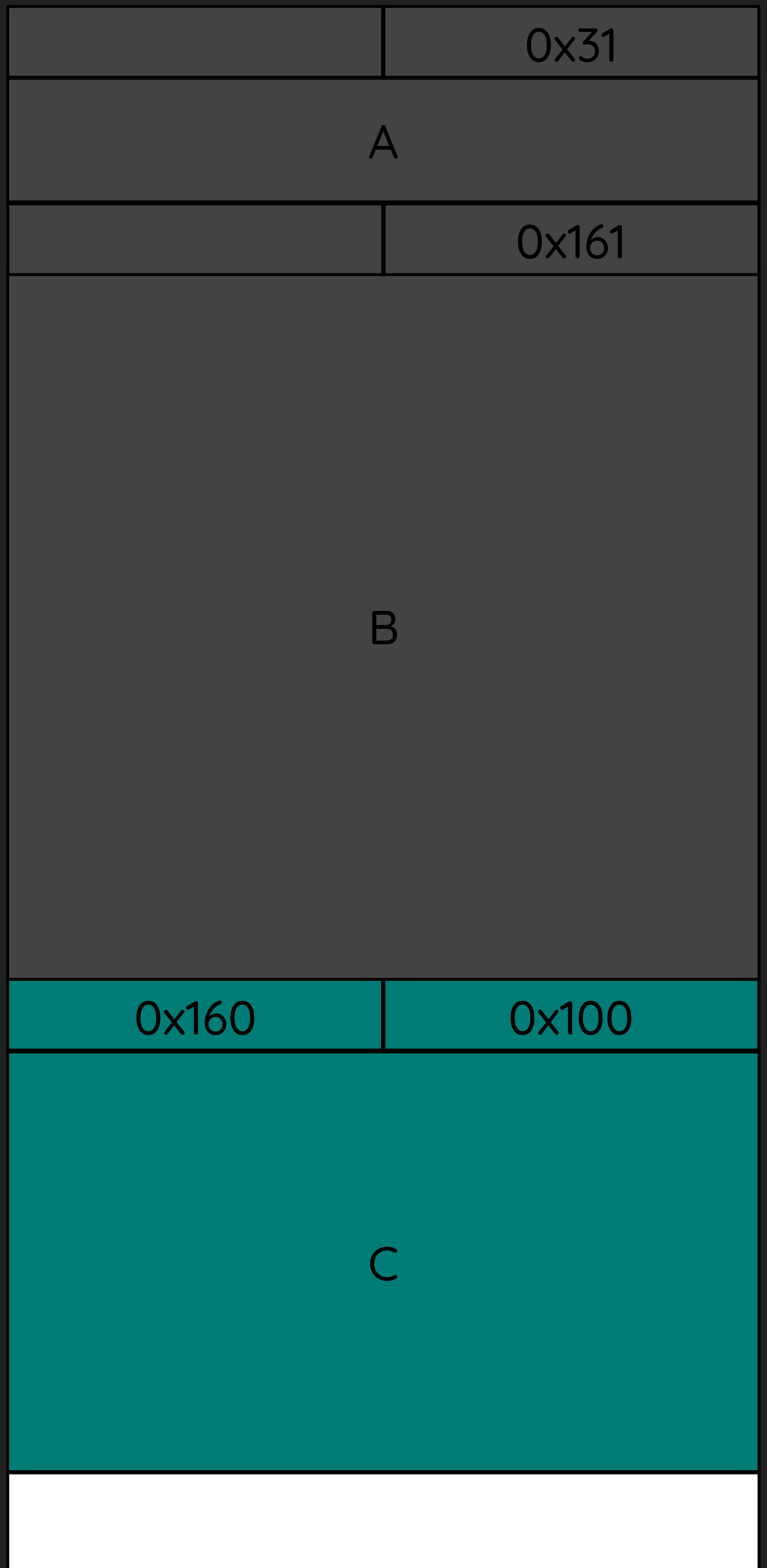
Heap overlap

- free(B)
- free(A)
- malloc(0x28)
- malloc(0x88)
- malloc(0x48)
- free(B)
- free(C)
- malloc(0x258)



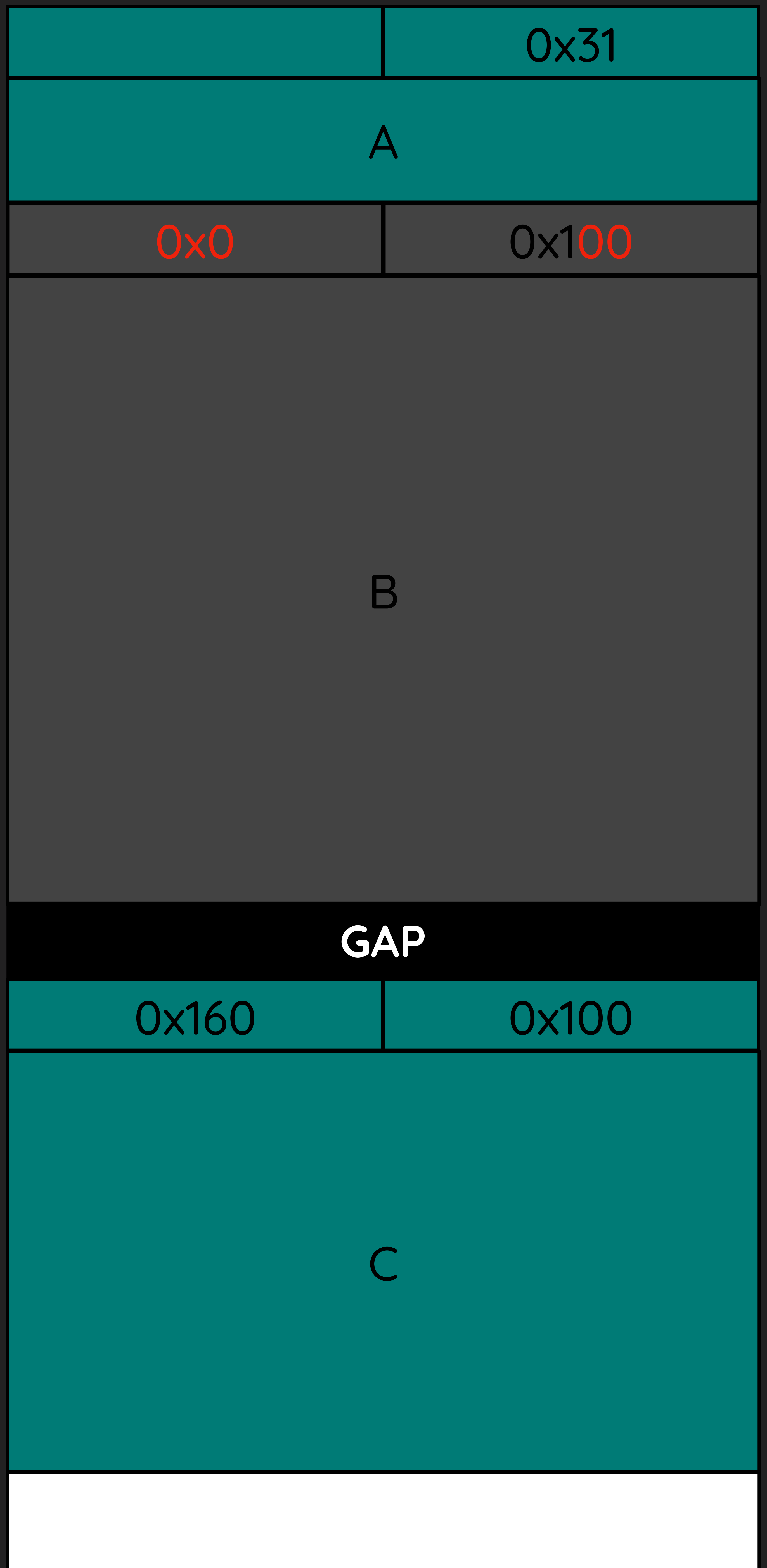
Heap overlap

- free(B)
- free(A)
- malloc(0x28)
- malloc(0x88)
- malloc(0x48)
- free(B)
- free(C)
- malloc(0x258)



Heap overlap

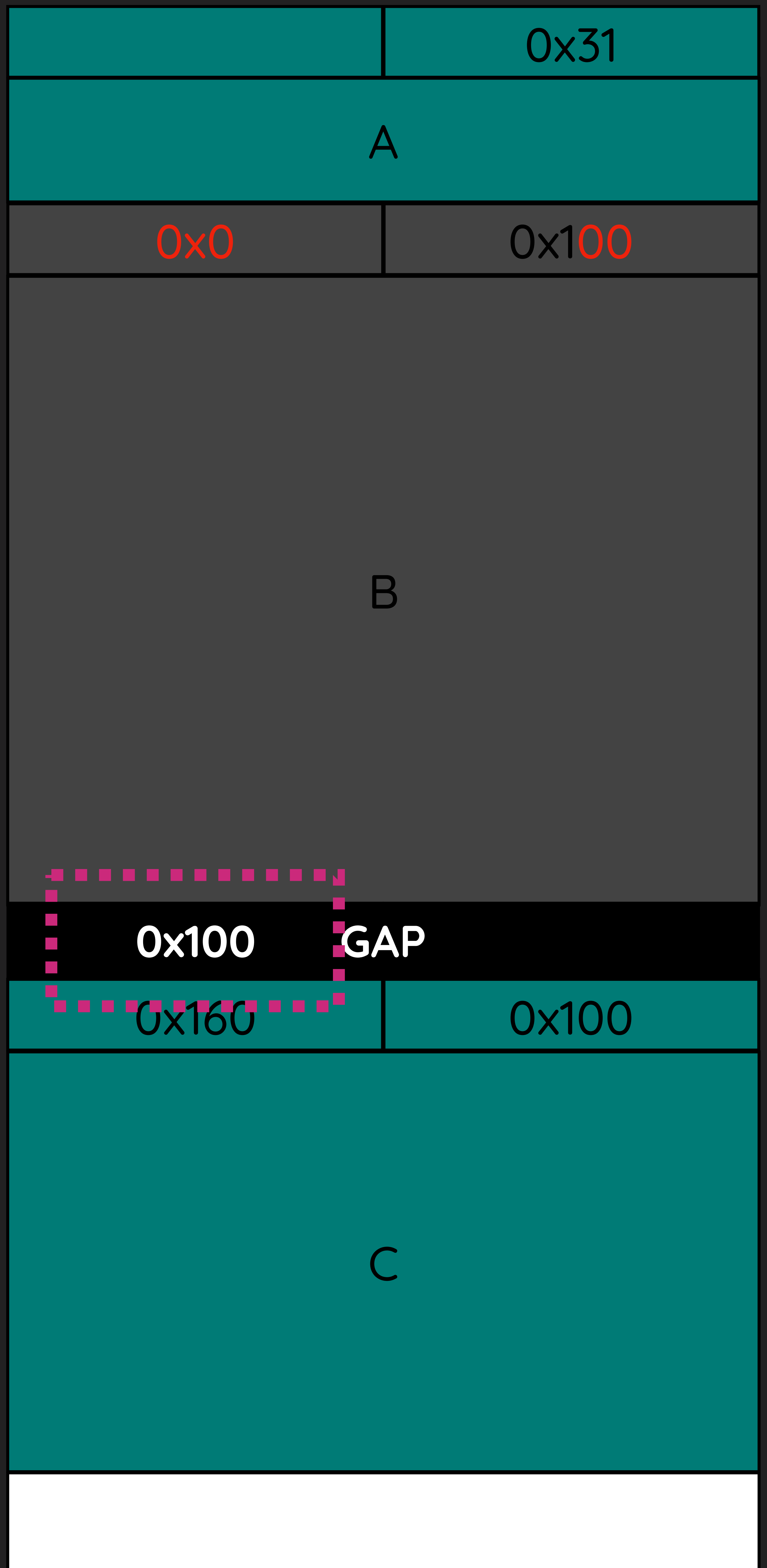
- free(B)
- free(A)
- malloc(0x28)
- malloc(0x88)
- malloc(0x48)
- free(B)
- free(C)
- malloc(0x258)



Heap overlap

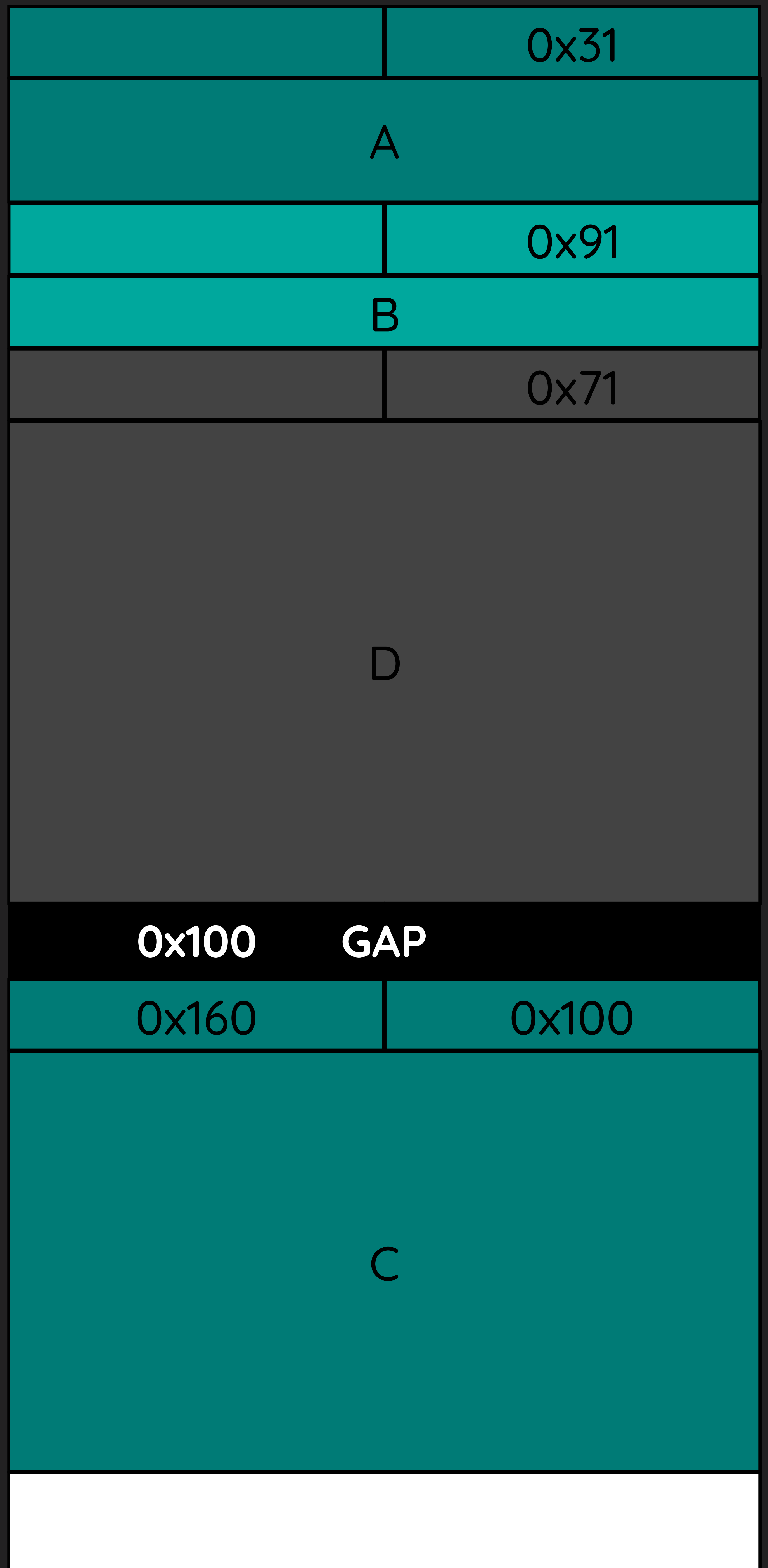
- free(B)
- free(A)
- malloc(0x28)
- malloc(0x88)
- malloc(0x48)
- free(B)
- free(C)
- malloc(0x258)

先前 allocate B chunk 時，
需要先預留 fake prev_size，
bypass 檢查，不然後面 malloc(0x88) 時
會吃 corrupted size vs. prev_size



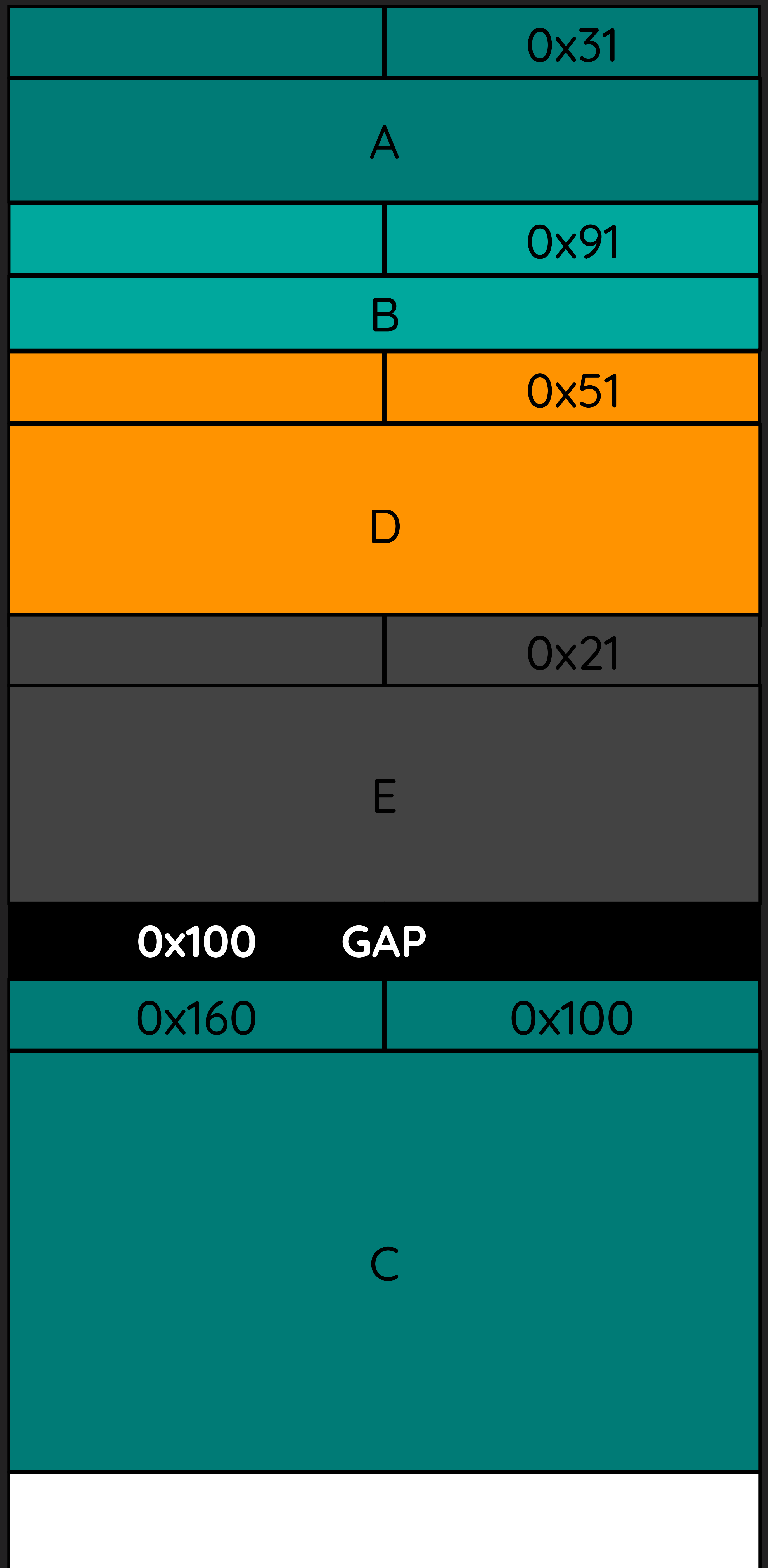
Heap overlap

- free(B)
- free(A)
- malloc(0x28)
- malloc(0x88)
- malloc(0x48)
- free(B)
- free(C)
- malloc(0x258)



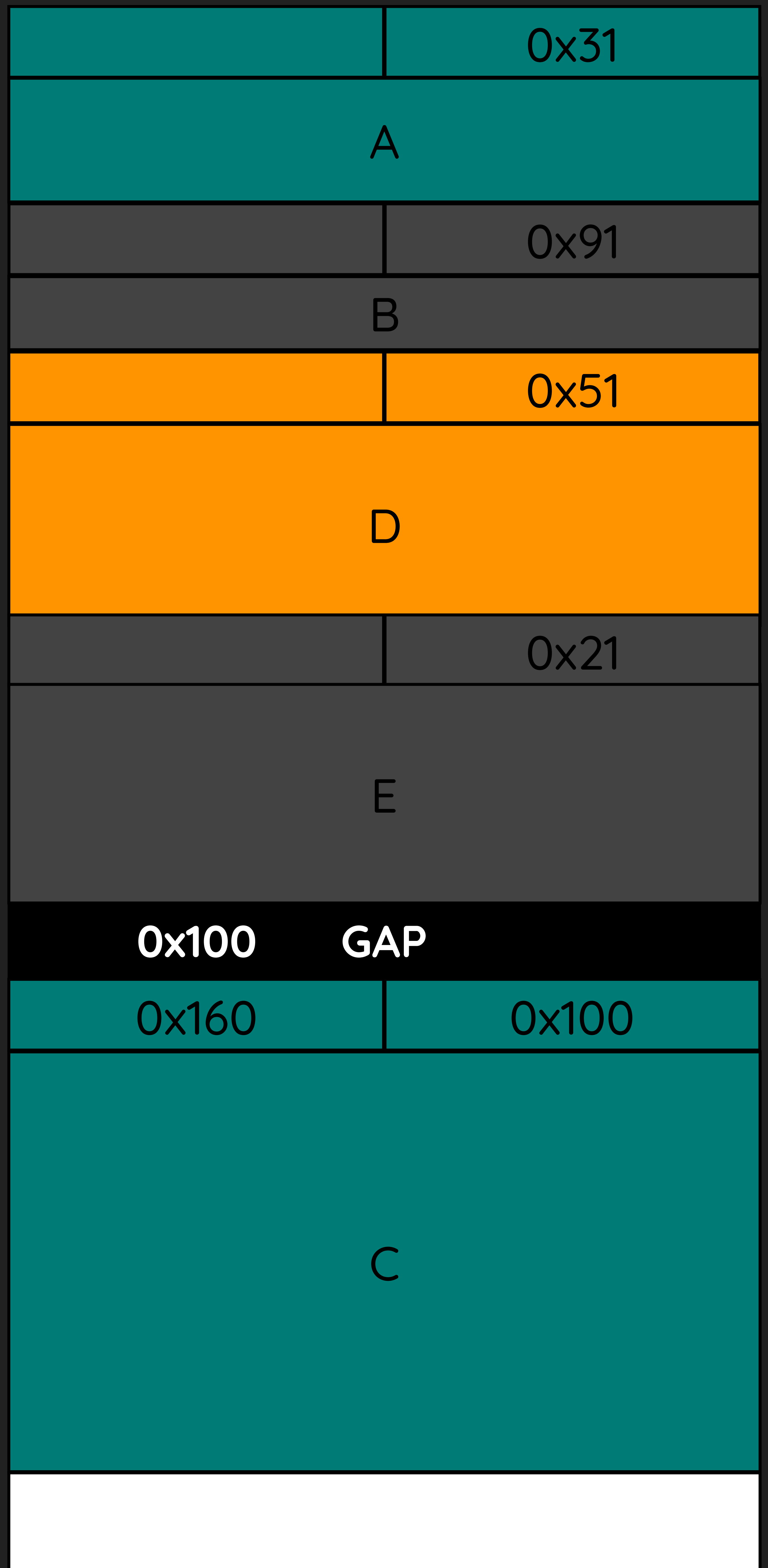
Heap overlap

- free(B)
- free(A)
- malloc(0x28)
- malloc(0x88)
- malloc(0x48)
- free(B)
- free(C)
- malloc(0x258)



Heap overlap

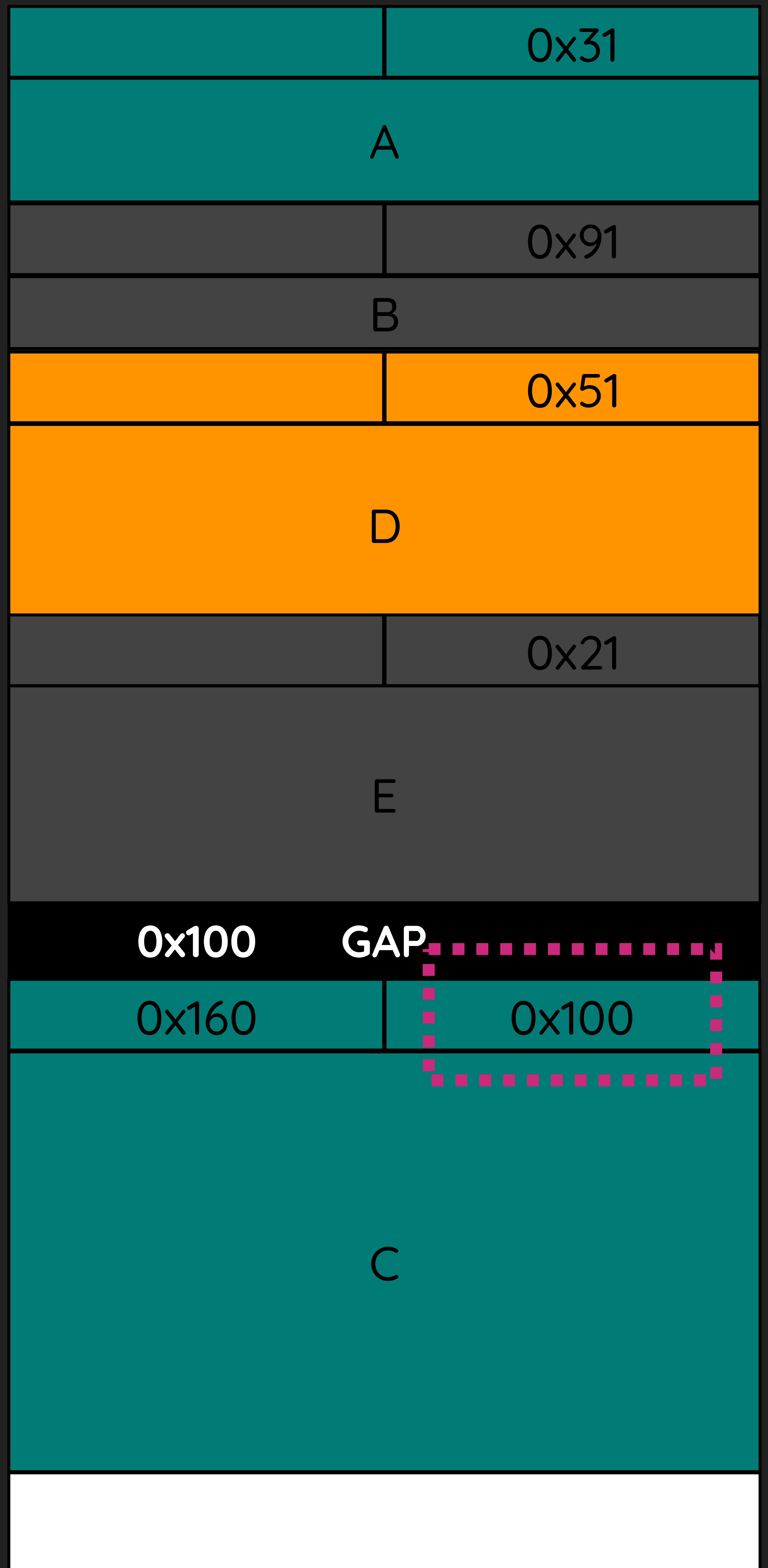
- free(B)
- free(A)
- malloc(0x28)
- malloc(0x88)
- malloc(0x48)
- free(B)
- free(C)
- malloc(0x258)



Heap overlap

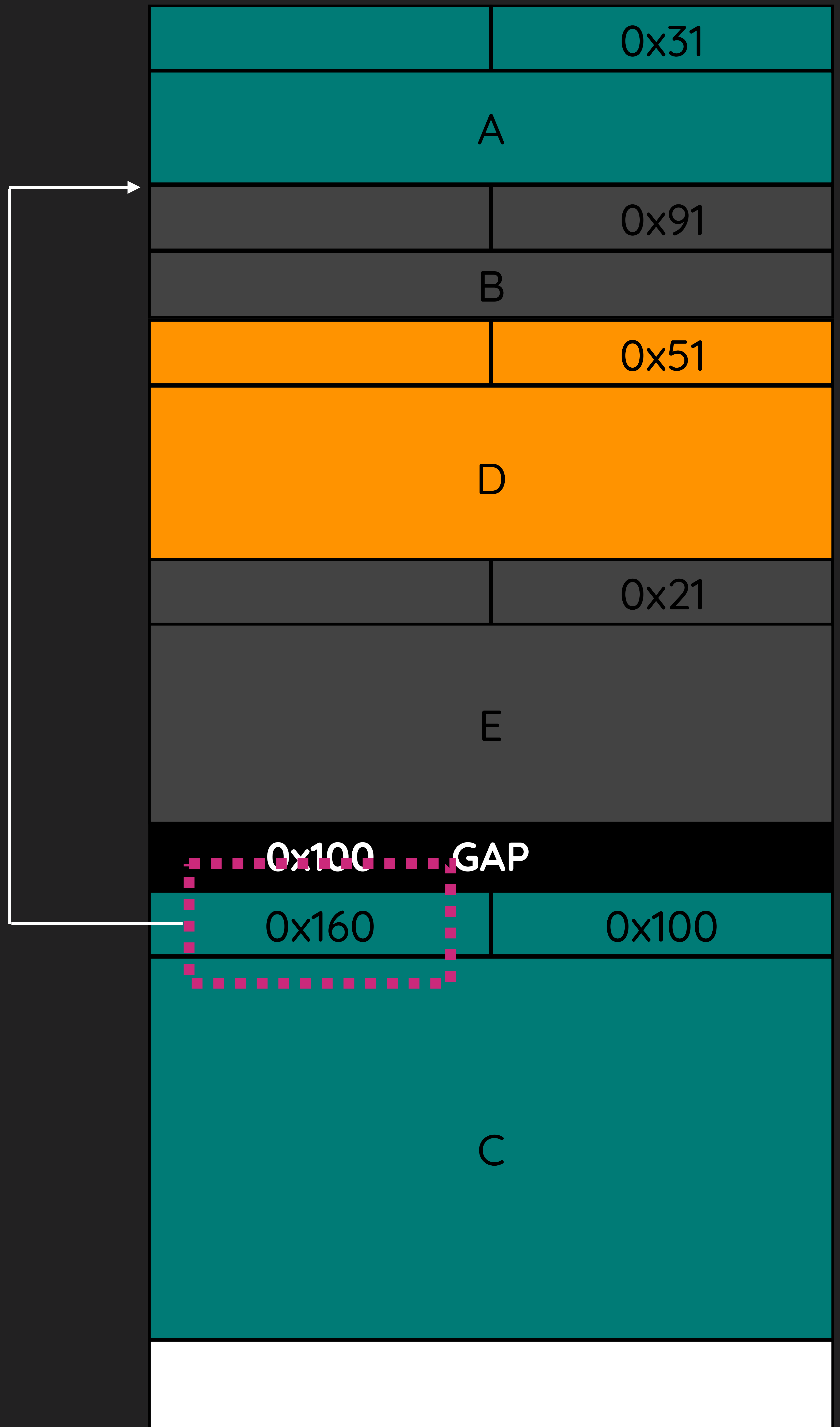
- free(B)
- free(A)
- malloc(0x28)
- malloc(0x88)
- malloc(0x48)
- free(B)
- free(C)
- malloc(0x258)

chunk C 是 small bin
檢查上一塊是否 inuse (P)
上一塊是是 free chunk
根據 prev_size 合併



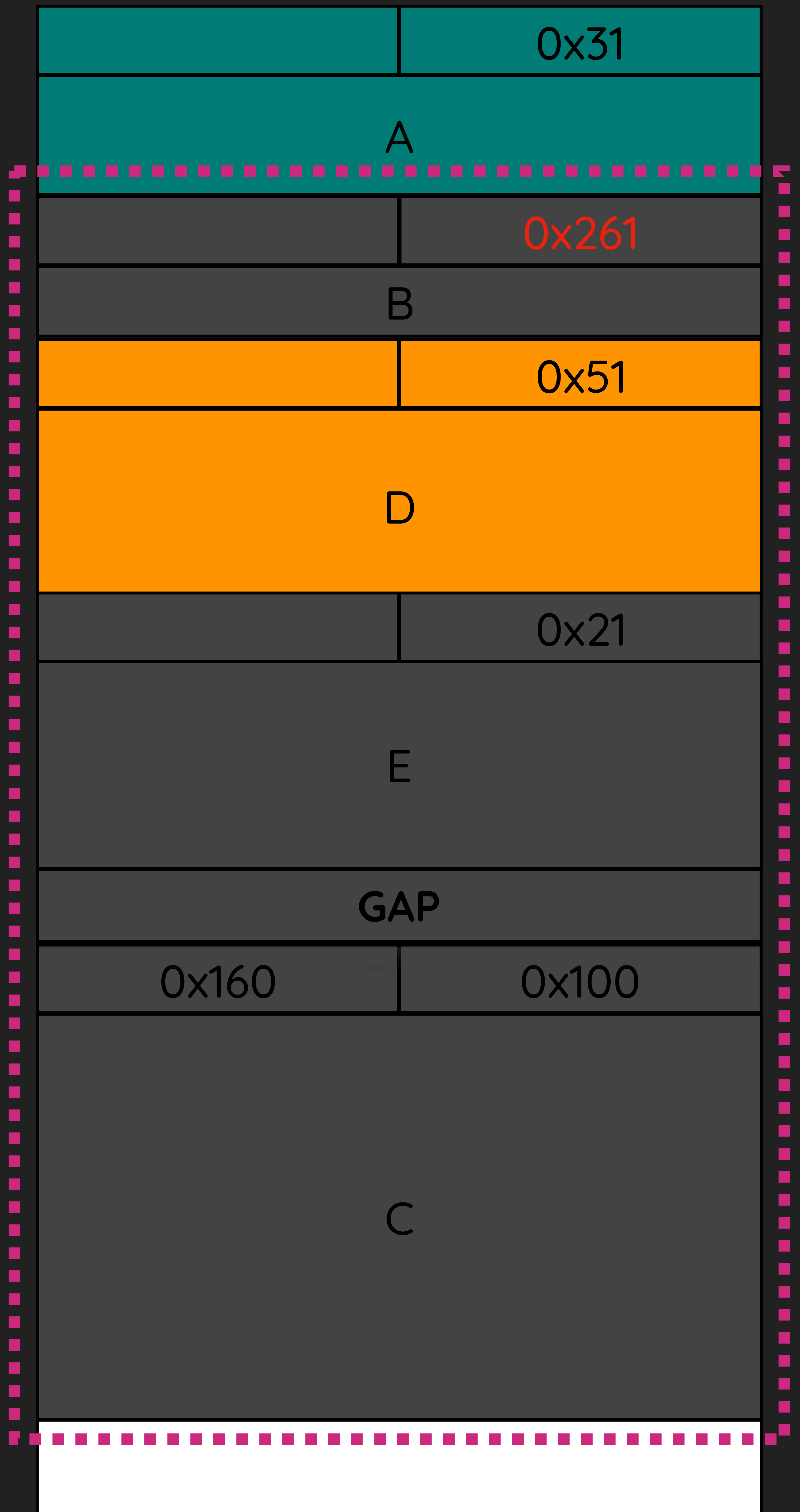
Heap overlap

- free(B)
- free(A)
- malloc(0x28)
- malloc(0x88)
- malloc(0x48)
- free(B)
- free(C)
- malloc(0x258)



Heap overlap

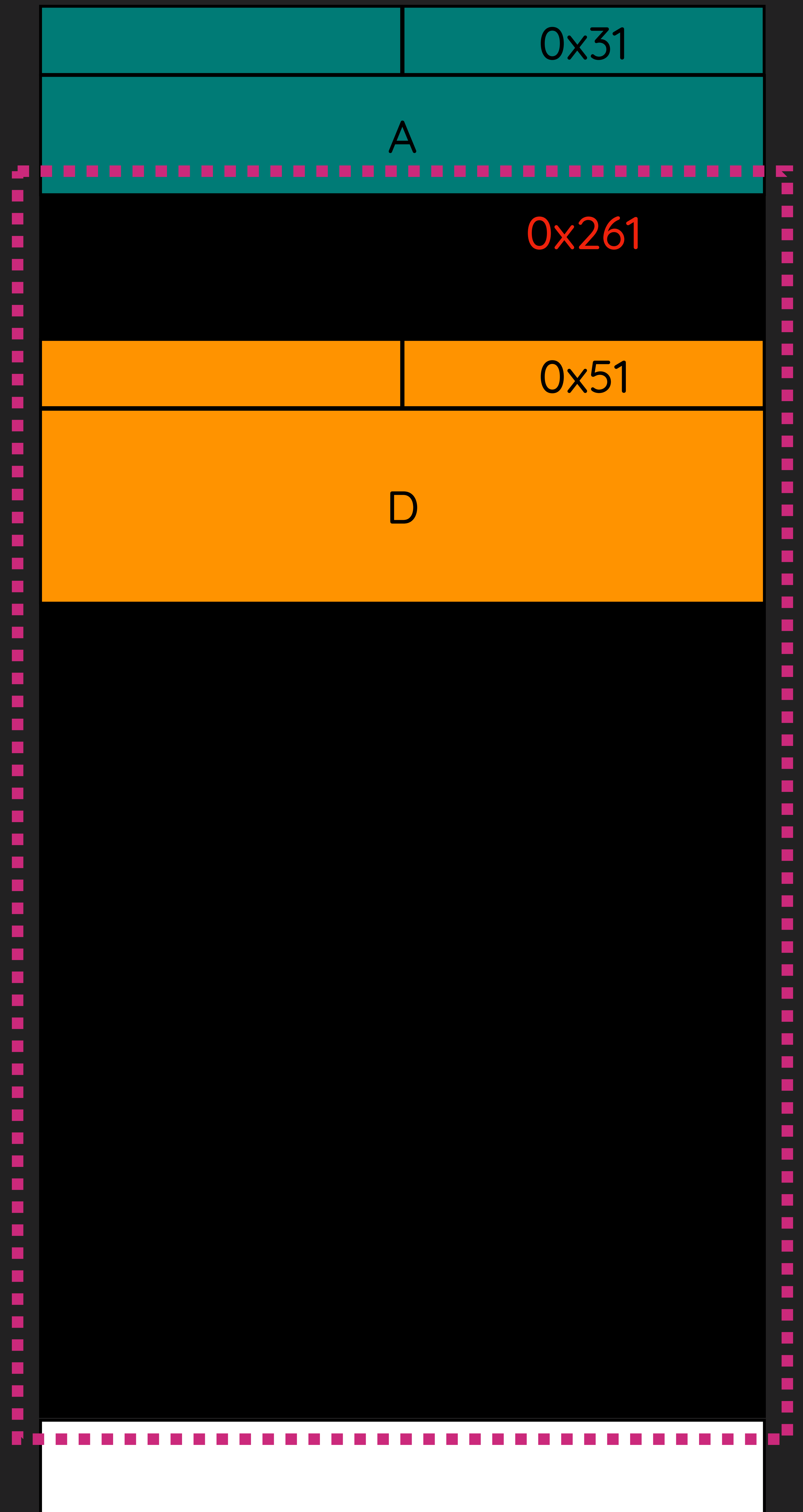
- free(B)
- free(A)
- malloc(0x28)
- malloc(0x88)
- malloc(0x48)
- free(B)
- free(C)
- malloc(0x258)



Heap overlap

- free(B)
- free(A)
- malloc(0x28)
- malloc(0x88)
- malloc(0x48)
- free(B)
- free(C)
- malloc(0x258)

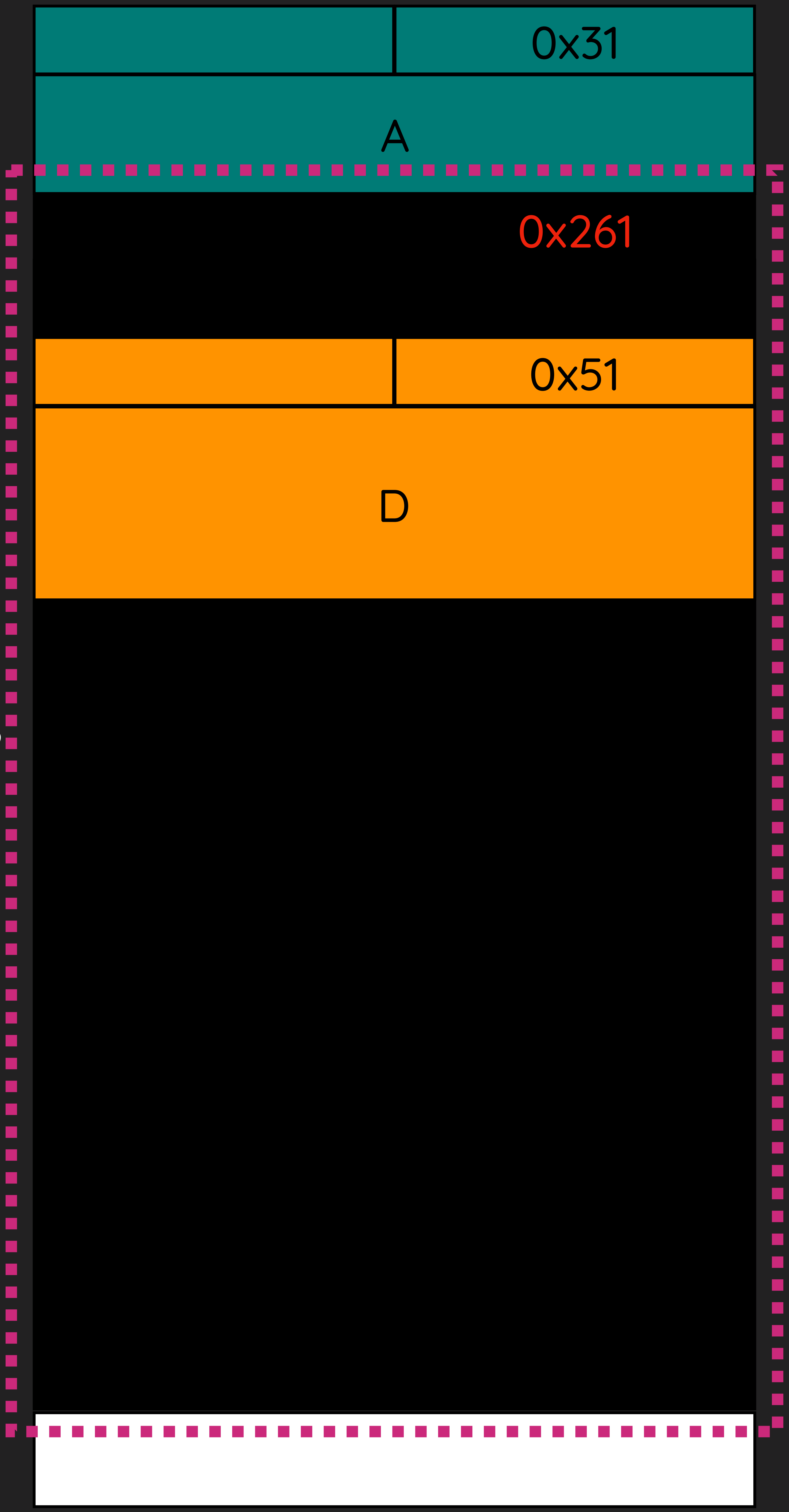
Overlap!



Heap overlap

- 此時 chunk D 中若存在 data pointer, function pointer 等，因都位於 chunk B 的 data 可以透過 chunk B 來 overwrite，達到任意讀寫等。

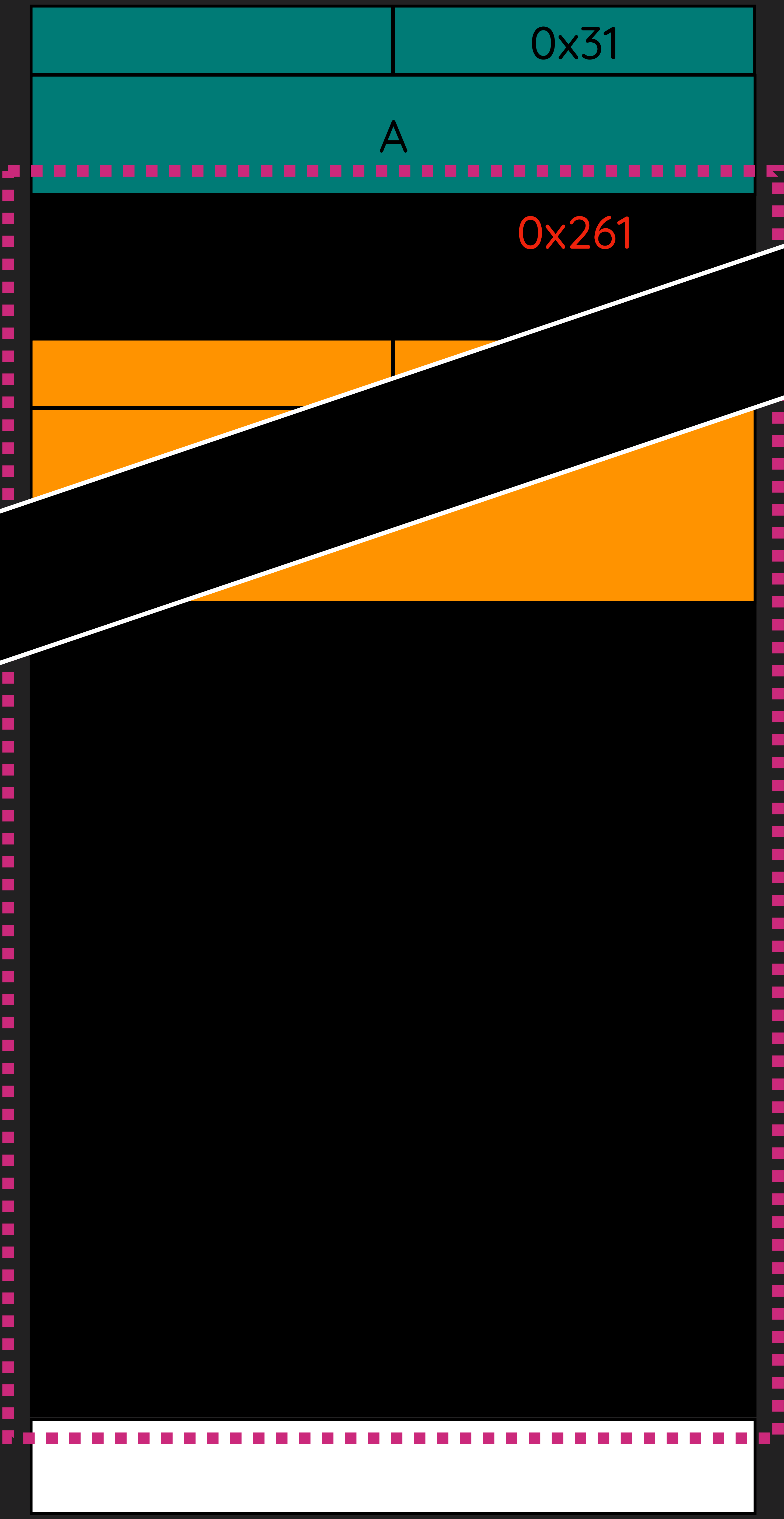
- 或是進一步偽造 heap，如把 D free 掉 overwrite fd，玩 fastbin attack等等



Heap overlap

- 此時 chunk D 中若存在 data pointer, functio[redacted]因都位於 chunk B 的 data 可以透過[redacted]verwrite，達到任意讀寫等。

- 或是進[redacted]，如把 D free 掉 overwrite fd，[redacted]ttack等等



Unsafe Unlink

unsafe unlink

- doubly linked list - unlink
- unlink(p)
  - p->fd = bk
  - p->bk = fd

```c
/* Take a chunk off a bin list */
#define unlink(AV, P, BK, FD) {
    FD = P->fd;
    BK = P->bk;
    if (__builtin_expect (FD->bk != P || BK->fd != P, 0)) \
      malloc_printerr (check_action, "corrupted double-linked list", P, AV);
    else {
        FD->bk = BK;
        BK->fd = FD;
```

- 古典做法，再有漏洞的前提下，偽造 p 的 fd, bk，透過 unlink，可以對 memory 做寫入，如：
  - FD = p->fd = free@GOT - 0x18
  - BK = p->bk = one_gadget
    - FD->bk = (free@GOT - 0x18 + 0x18) = *free@GOT = p->bk = one_gadget
    - *free@GOT = one_gadget

unsafe unlink

```c
/* Take a chunk off a bin list */
#define unlink(AV, P, BK, FD) {
    FD = P->fd;
    BK = P->bk;
    if (__builtin_expect (FD->bk != P || BK->fd != P, 0)) \
      malloc_printerr (check_action, "corrupted double-linked list", P, AV);
    else {
        FD->bk = BK;
        BK->fd = FD;
```

- 後來增加了檢查，驗證 doubly linked list 的完整性。
- 指過去要能也指回來
    - P 的下一個的上一個要是 P
    - P 的上一個的下一個要是 P

unsafe unlink

- 後來的利用方式與其說是繞過，比較像是妥協。
- 利用條件
  - 已知 address 底下存放著 p 指標，已知 &p 的意思
  - p 為 data pinter 可以多次寫入，或其他同效果方法

unsafe unlink

- FD = p->fd = &p - 0x18
- BK = p->bk = &p - 0x10
- 如此滿足這項檢查
  - FD->bk = *((&p - 0x18) + 0x18) == p
  - BK->fd = *((&p - 0x10) + 0x10) == p
- unlink 後會使的 FD->bk = BK, BK->fd = FD
  - BK->fd = *((&p - 0x10) + 0x10) = FD = &p - 0x18
  - p = &p - 0x18

unsafe unlink

- p = &p - 0x18
- 之後再對 data pointer p 做寫入，便可以覆蓋掉 p，overwrite 為任意地址。
- 再次寫入即會對目標地址寫入，達到 write everywhere 的效果。

Unsorted bin attack

Unsorted bin attack

- 在有漏洞的前提下
- 將 unsorted bin 的 bk 填成 address - 0x10，再 malloc 相同大小的 size，address 的地方會被填上 libc 的 address，指向 main_arena 中。
- 無法直接性做到太多事情，可以用來間接進一步利用。
    - 如將一個地方填上一個很大的數子。
    - information leak
    - 進階 heap exploit 技巧搭配

Something Good

- how2heap - https://github.com/shellphish/how2heap
- Angelboy - https://www.slideshare.net/AngelBoy1
- pwnable.tw

Summary

- 這些手法跟技巧都是以經典形式呈現。
- heap exploitation 的精神主要就是在理解 glibc 機制，以及底層運作原理的情況下，操控機制。
- 玩法與做法都是變形無數。

“Heap 是一門藝術。”

–Angelboy

HW 0x08

“See you in final CTF! 😉”

–yuawn

Thanks!

_yuawn

yuawn