Binary Exploitation aka Pwn
Heap

NTUSTISC

2021/5/31

whoami

- LJP / LJP-TW
- Pwn / Rev
- NTUST / ~~NCTU~~ / NYCU
- 10sec CTF Team

Outline

- What is Heap?
- Tool - Pwngdb
- 基礎知識 – ptmalloc
- 基礎知識 – Chunk
- 基礎知識 – Fastbin
- 基礎知識 – Tcache
- Heap-Based Buffer Overflow
- UAF
- Double Free
- Hooks
- Fastbin Dup
- Tcache Dup
- 基礎知識 – Unsorted Bin
- 基礎知識 – Consolidate
- Unsafe Unlink

What is Heap?

What is Heap

- malloc / new 分配出的記憶體來自於此
- malloc 前

```
Start              End                Offset             Perm Path
0x0000555555554000 0x0000555555555000 0x0000000000000000 r-- /host/mnt/hgfs/tmp/ntustisc/Medium/demo/alloc
0x0000555555555000 0x0000555555556000 0x0000000000001000 r-x /host/mnt/hgfs/tmp/ntustisc/Medium/demo/alloc
0x0000555555556000 0x0000555555557000 0x0000000000002000 r-- /host/mnt/hgfs/tmp/ntustisc/Medium/demo/alloc
0x0000555555557000 0x0000555555558000 0x0000000000002000 r-- /host/mnt/hgfs/tmp/ntustisc/Medium/demo/alloc
0x0000555555558000 0x0000555555559000 0x0000000000003000 rw- /host/mnt/hgfs/tmp/ntustisc/Medium/demo/alloc
0x00007ffff7dc9000 0x00007ffff7dee000 0x0000000000000000 r-- /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x00007ffff7dee000 0x00007ffff7f66000 0x0000000000025000 r-x /usr/lib/x86_64-linux-gnu/libc-2.31.so
```

- malloc 後

```
Start              End                Offset             Perm Path
0x0000555555554000 0x0000555555555000 0x0000000000000000 r-- /host/mnt/hgfs/tmp/ntustisc/Medium/demo/alloc
0x0000555555555000 0x0000555555556000 0x0000000000001000 r-x /host/mnt/hgfs/tmp/ntustisc/Medium/demo/alloc
0x0000555555556000 0x0000555555557000 0x0000000000002000 r-- /host/mnt/hgfs/tmp/ntustisc/Medium/demo/alloc
0x0000555555557000 0x0000555555558000 0x0000000000002000 r-- /host/mnt/hgfs/tmp/ntustisc/Medium/demo/alloc
0x0000555555558000 0x0000555555559000 0x0000000000003000 rw- /host/mnt/hgfs/tmp/ntustisc/Medium/demo/alloc
0x0000555555559000 0x000055555557a000 0x0000000000000000 rw- [heap]
0x00007ffff7dc9000 0x00007ffff7dee000 0x0000000000000000 r-- /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x00007ffff7dee000 0x00007ffff7f66000 0x0000000000025000 r-x /usr/lib/x86_64-linux-gnu/libc-2.31.so
```

What is Heap

- 執行時期動態配置的記憶體區段

- 若過於頻繁呼叫 syscall 則會導致程式經常在 Kernel / User Mode 切換, 導致效能低落

- 所以許多 Library 實作皆為向 Kernel 申請一大塊記憶體, 並自行實作一套機制管理這塊記憶體, 去實作切割、分配、回收、合併等各種操作

What is Heap

- 各種實作
- Glibc：ptmalloc
- Chrome：tcmalloc
- uClibc-ng：dlmalloc
- 這一篇簡報是講 glibc 的 ptmalloc
- Libc 2.31 ptmalloc Source Code

Tool Pwngdb

Pwngdb

- https://github.com/scwuaptx/Pwngdb
- 與 gef 混用, ~/.gdbinit 參考這個連結
- Angelboy 大大寫的好用工具
- 用來觀察 Heap

Pwngdb Demo

Basic Knowledge

ptmalloc

ptmalloc

- 第一次呼叫 malloc 時初始化 main_arena, 並向 Kernel 申請一大塊記憶體, 再從這一大塊記憶體分割出一個 chunk, 讓 malloc 回傳給程式
- main_arena 存在於 libc 裡, 紀錄著各種資訊
  - 各種 bins 鏈表
  - Top chunk 位址
  - ...
- 之後的 malloc/free 都是在分割/回收 chunk, 並利用 main_arena 紀錄的 bins 鏈表管理回收回來的 chunk

ptmalloc

- 一大塊記憶體之後會被切割成大大小小的 chunk

ptmalloc

- 用簡易的例子來幫助想像 (示意圖為簡化過後的版本)

C

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
```

main_arena

Heap

ptmalloc

- 第一次呼叫 malloc, 首先申請一大塊記憶體作為 Heap

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
```

main_arena

Heap

ptmalloc

- 接著切割 0x20 大小的 Chunk 給 ptr1, 剩下的為 Top Chunk

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
```

main_arena

Heap

ptmalloc

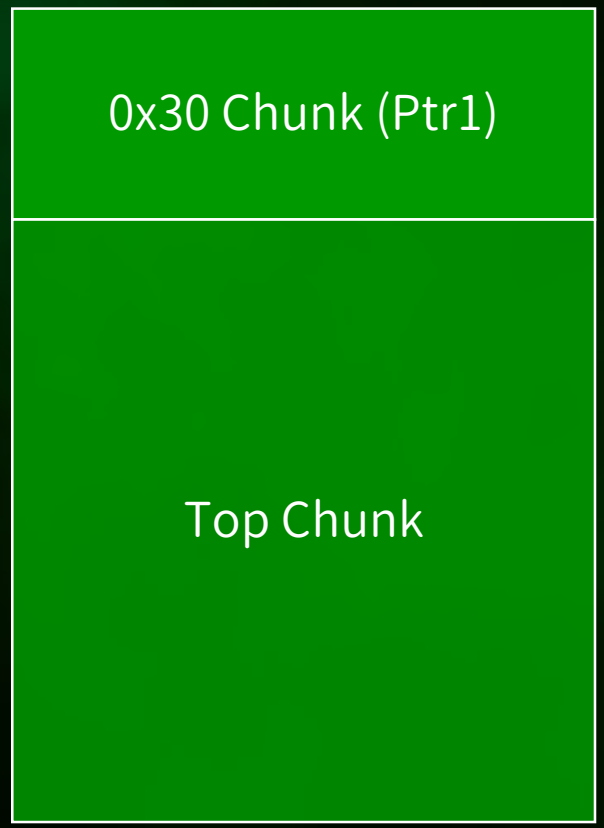
- Chunk 大小怎麼算後續講解

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
```

main_arena



Heap

ptmalloc

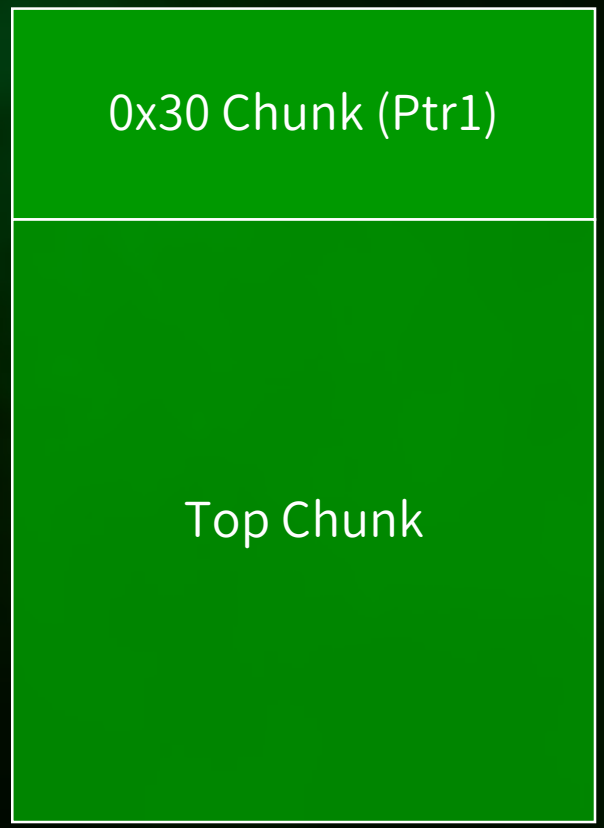
- 從 Top Chunk 切割出 0x20 大小的 Chunk 給 ptr2

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
```

main_arena



Heap

ptmalloc

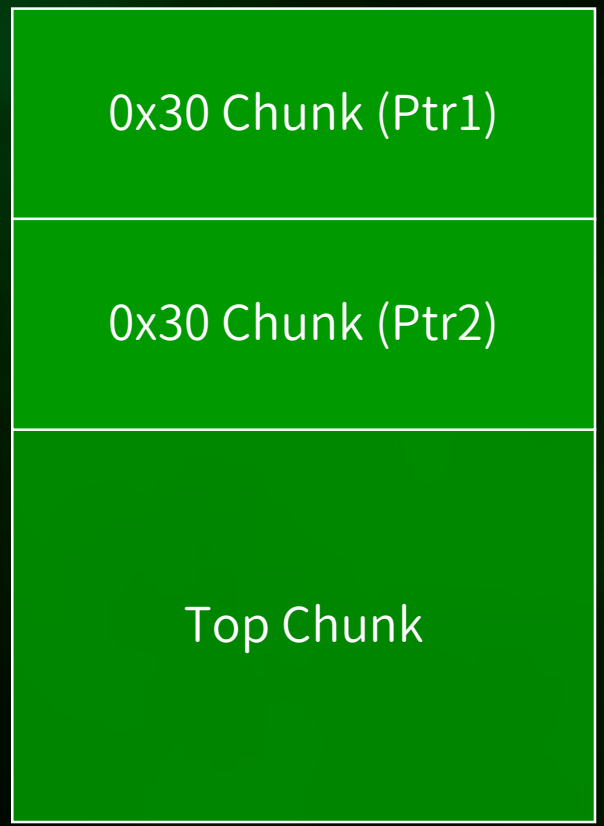
- 從 Top Chunk 切割出 0x20 大小的 Chunk 給 ptr3

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
```

main_arena



Heap

ptmalloc

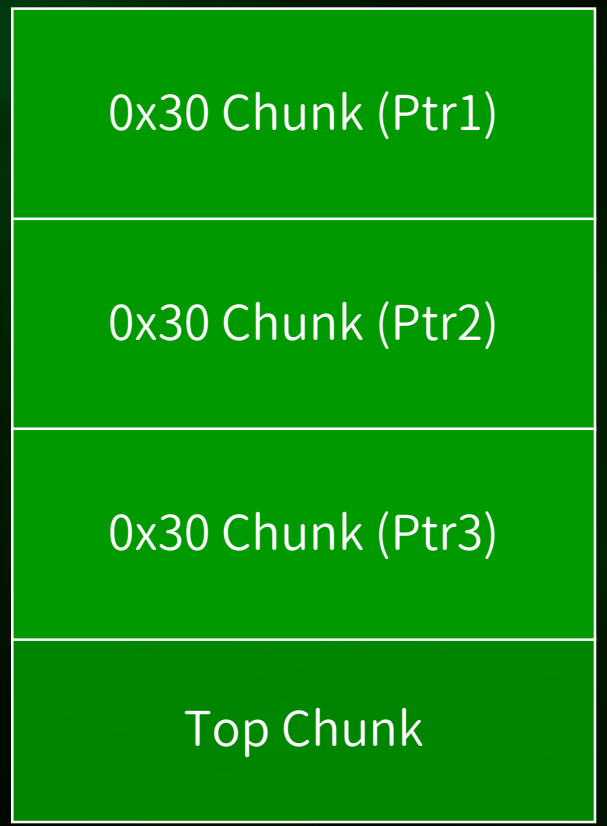
- 往 ptr1 (指向第一個 Chunk 的 Chunk Data) 寫入 0x20 個 A

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
```

main_arena



Heap

ptmalloc

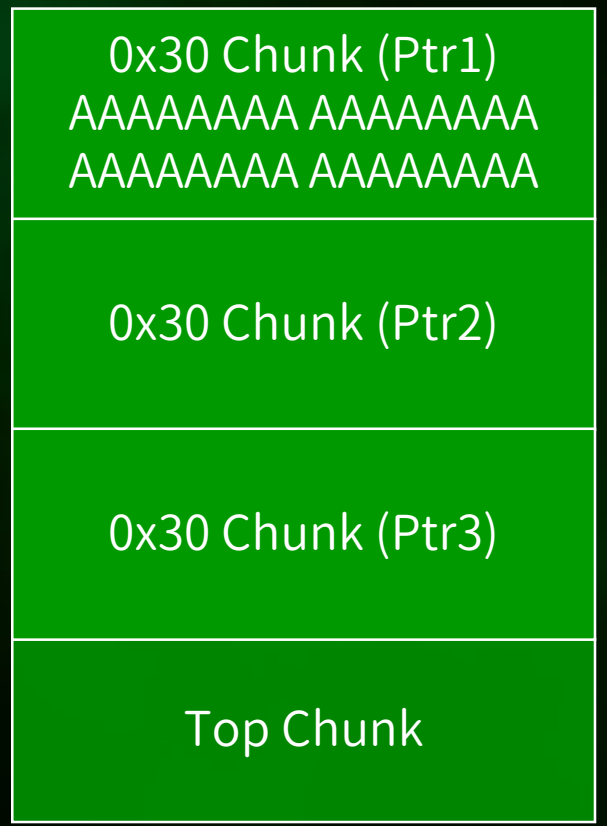
- 釋放 ptr1 指向的 Chunk

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
```

main_arena



Heap

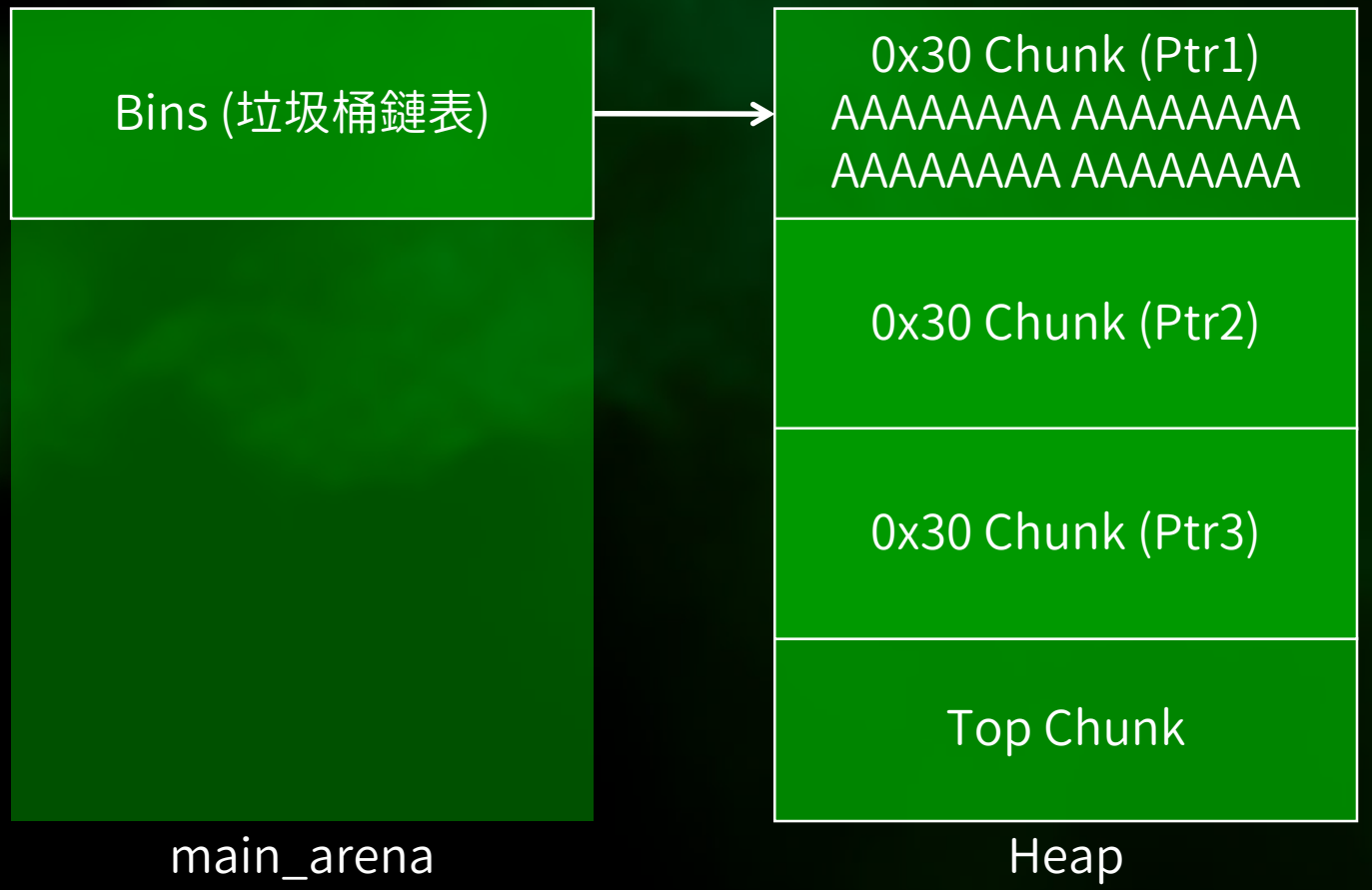
ptmalloc

- 釋放 ptr2 指向的 Chunk

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
```



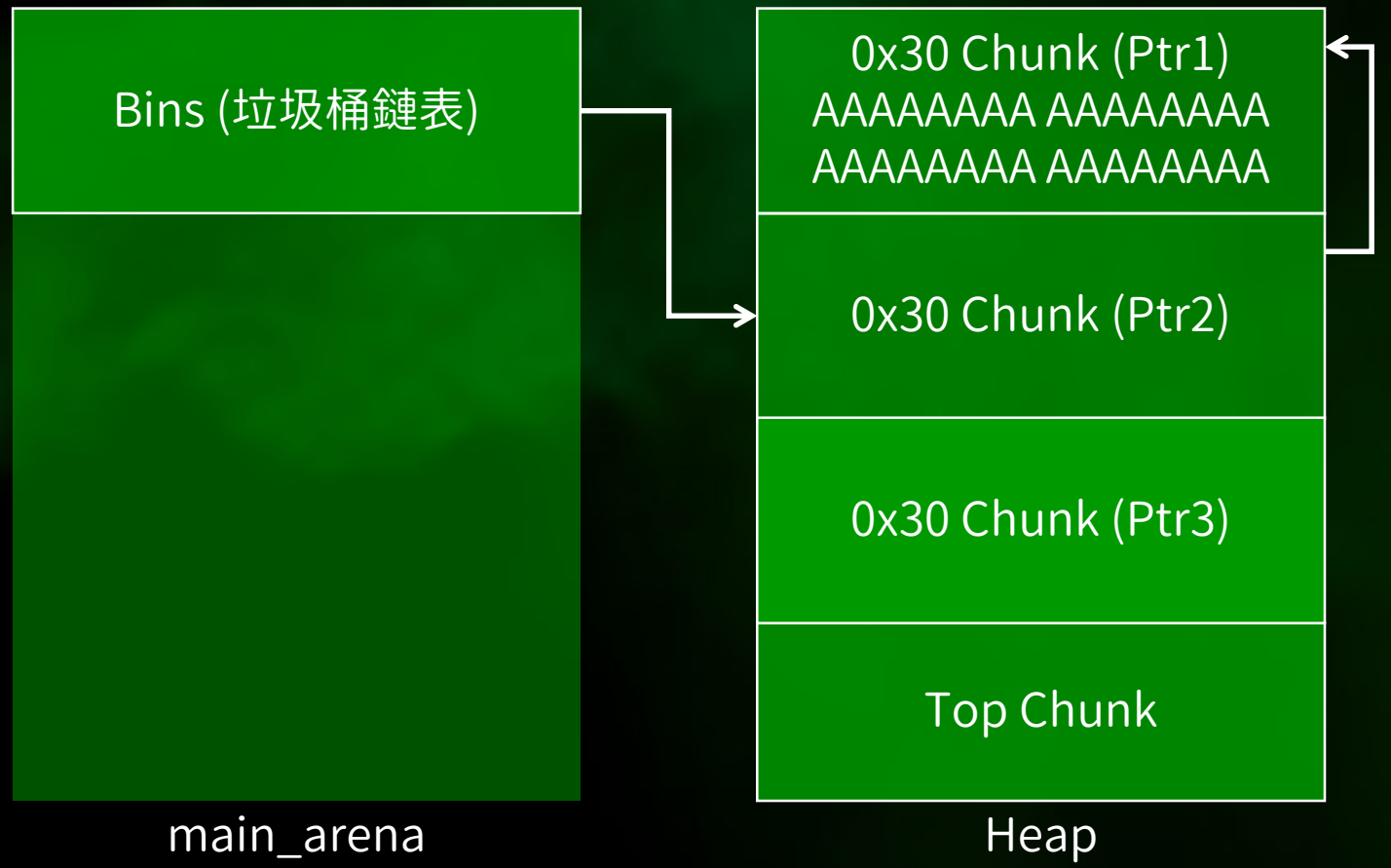
ptmalloc

- 釋放 ptr3 指向的 Chunk

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
```



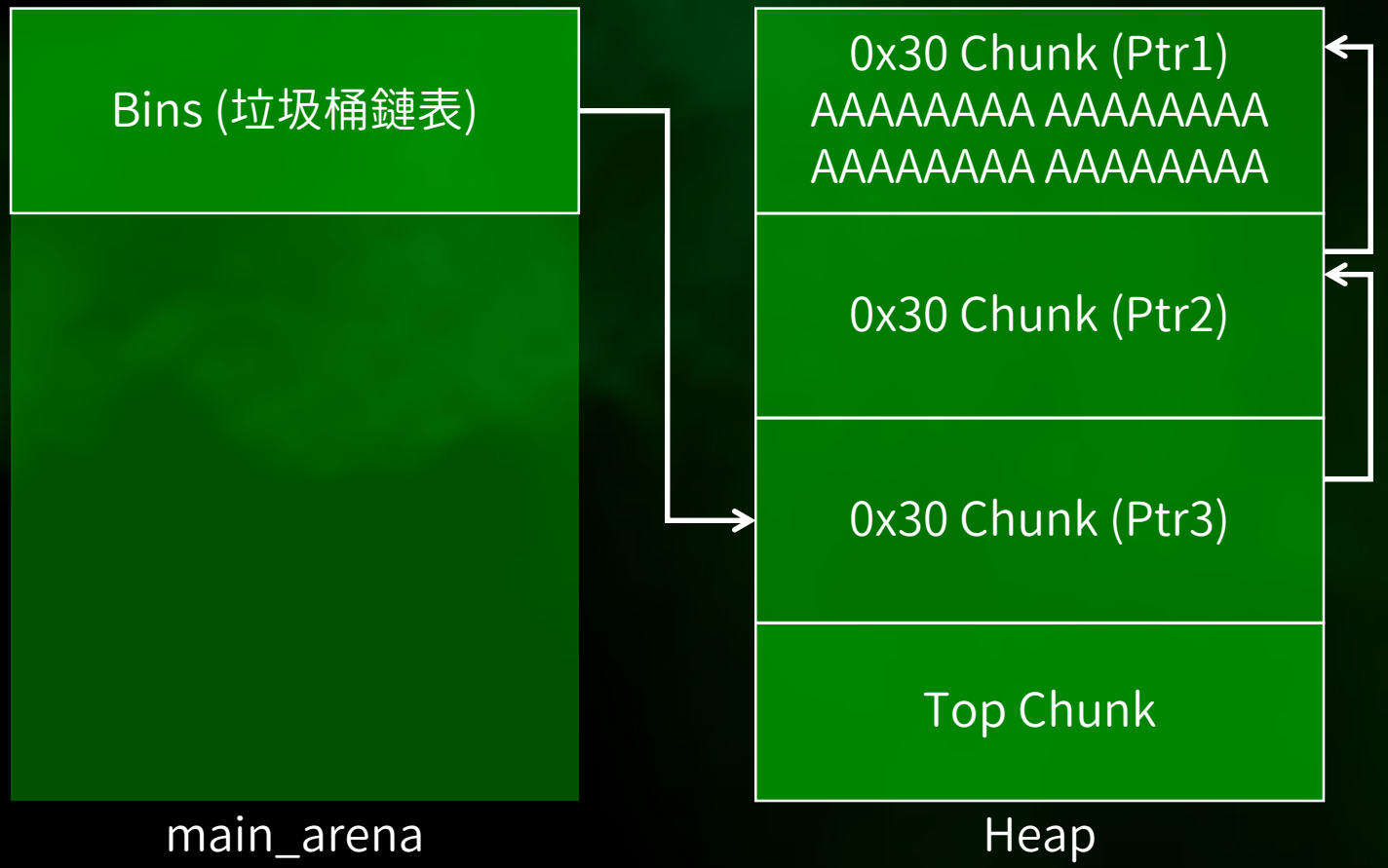
ptmalloc

- 之後再度分配同大小的 Chunk 時, 會從鏈表中拿

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
```



ptmalloc

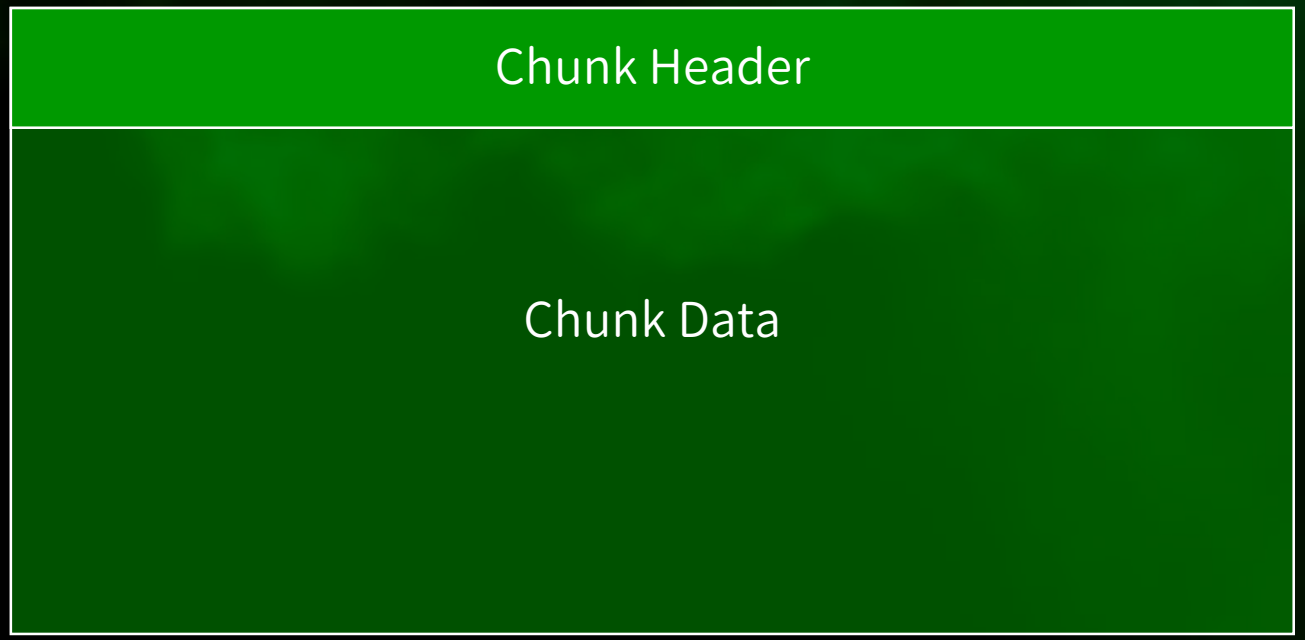
- Chunk 分成
  - Allocated Chunk
  - Free Chunk
  - Top Chunk
- Bins 分成
  - Fast bin
  - Small bin
  - Large bin
  - Unsorted bin
  - Tcache

Basic Knowledge

Chunk

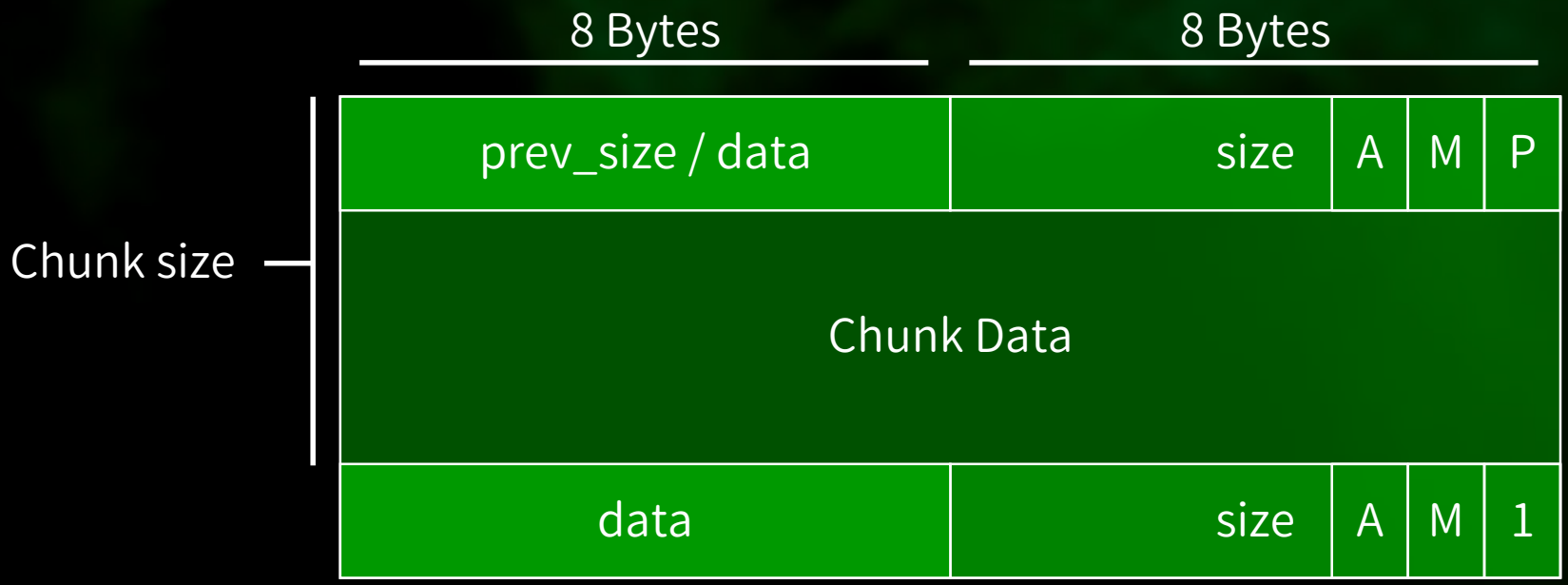
Chunk

- 每種 Chunk 有著大同小異的資料結構
- 大致分為 Chunk Header 與 Chunk Data
- 差異在 Chunk Header



Allocated Chunk

- prev_size/data: 鄰近的上一個 Chunk 的 size 或 data
- size: 此 Chunk 的 size
- A (NON_MAIN_ARENA bit): 是否由其他的 arena 管理, 而非 main_arena
- M (IS_MMAPPED bit): 是否由 mmap 創出來的
- P (PREV_INUSE bit): 鄰近的上一個 Chunk 是否正在使用



Ref: https://elixir.bootlin.com/glibc/glibc-2.31/source/malloc/malloc.c#L1076

Chunk

- 要求了 0x20 大小的空間, 實際上分配出的 Chunk 不只 0x20 大

- 實際計算 Chunk 該多大考慮了以下
  - Chunk Header 大小要算進去
  - 對齊記憶體

Chunk

- 實際計算方式如下方截圖
  - 加上 SIZE_SZ (8) 保留 Chunk 中放 size 的空間
  - 加上 MALLOC_ALIGN_MASK 後 and ~MALLOC_ALIGN_MASK
    - 強制進位, 使齊對齊記憶體
    - 假設 var = req + 8, 若 var 為 0x21 ~ 0x2f, 則進位為 0x30
    - 若 var 為 0x20, 則不進位, 維持 0x20

```
1196  #define request2size(req)                                         \
1197    (((req) + SIZE_SZ + MALLOC_ALIGN_MASK < MINSIZE)  ?             \
1198     MINSIZE :                                                      \
1199     ((req) + SIZE_SZ + MALLOC_ALIGN_MASK) & ~MALLOC_ALIGN_MASK)
```

Ref: https://elixir.bootlin.com/glibc/glibc-2.31/source/malloc/malloc.c#L1196

Chunk

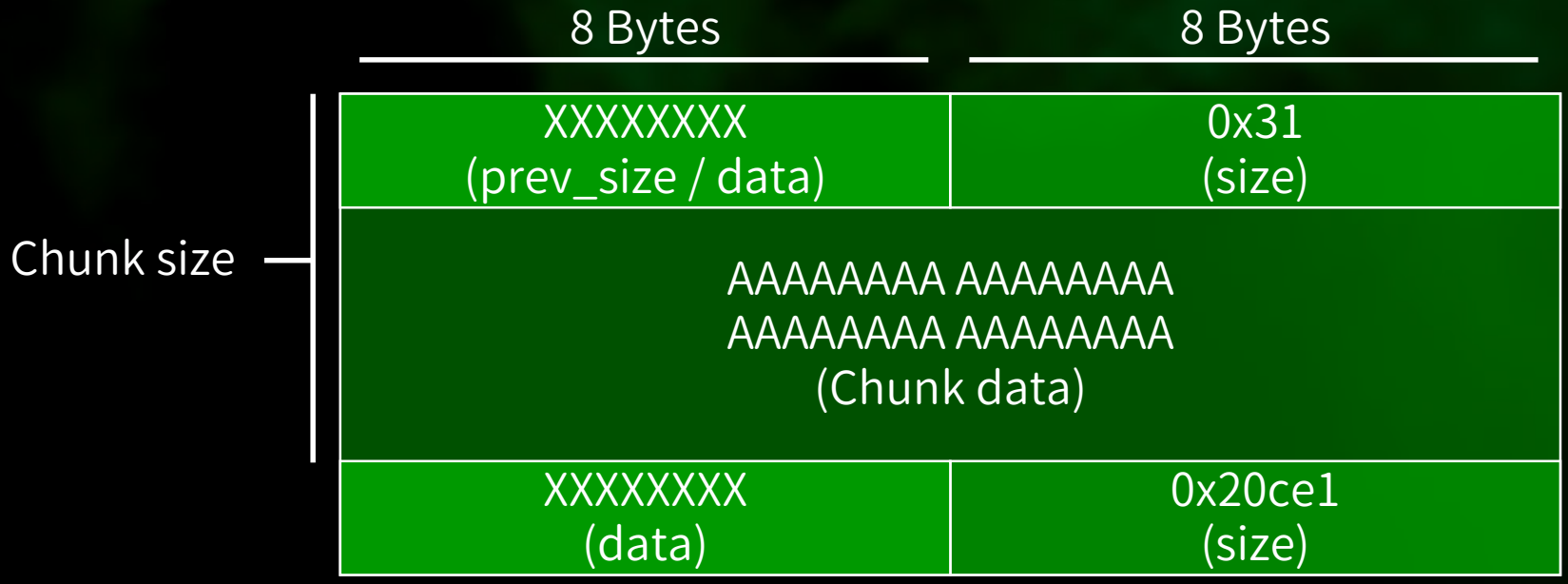
- 要求了 0x20 大小的空間, 實際上分配出的 Chunk 不只 0x20 大
- malloc(0x20) -> Chunk Size: 0x30
- malloc(0x28) -> Chunk Size: 0x30
- malloc(0x29) -> Chunk Size: 0x40
- malloc(0x2f) -> Chunk Size: 0x40
- malloc(0x30) -> Chunk Size: 0x40
- malloc(0x38) -> Chunk Size: 0x40

```
1196  #define request2size(req)                                         \
1197    (((req) + SIZE_SZ + MALLOC_ALIGN_MASK < MINSIZE)  ?             \
1198     MINSIZE :                                                      \
1199     ((req) + SIZE_SZ + MALLOC_ALIGN_MASK) & ~MALLOC_ALIGN_MASK)
```

Ref: https://elixir.bootlin.com/glibc/glibc-2.31/source/malloc/malloc.c#L1196

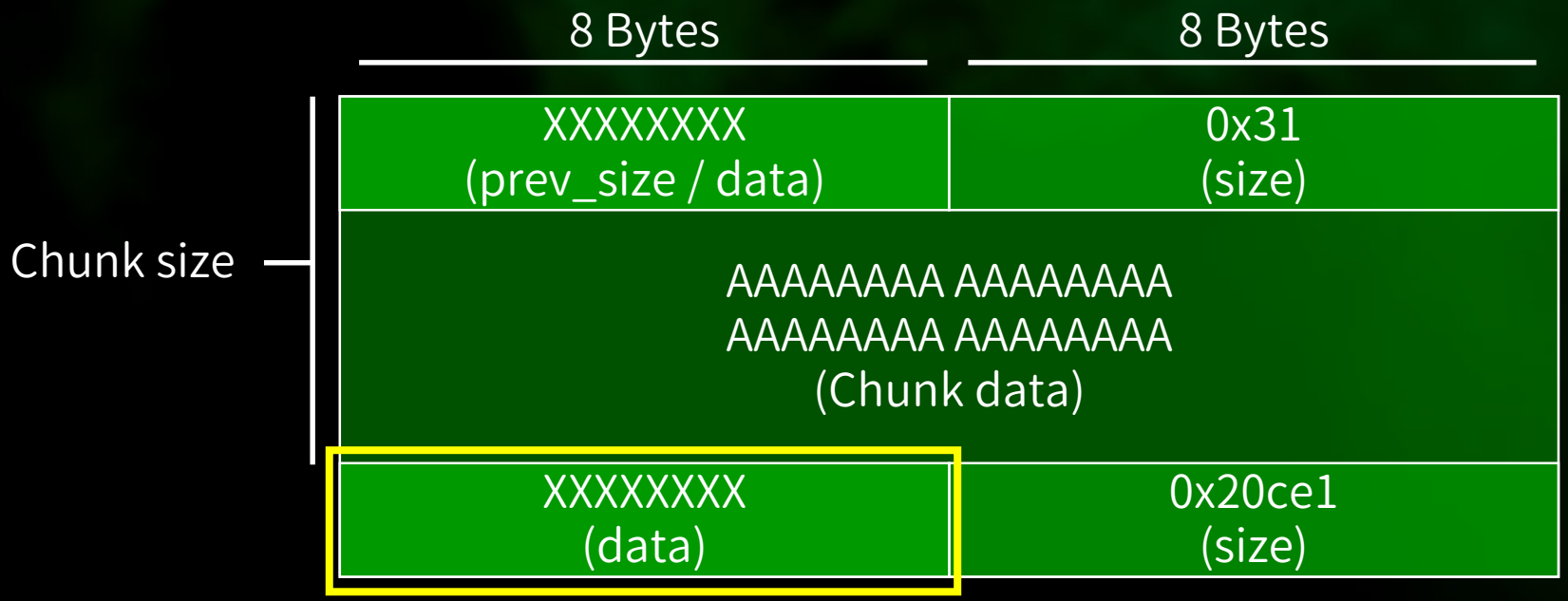
Allocated Chunk

- 來個栗子
- malloc(0x20) 後寫入 0x20 個 A
- 計算 Chunk Size: (0x20 + 0x8 + 0xf) & ~0xf = 0x30
  - A bit 為 0, 表示在 main_arena
  - M bit 為 0, 表示非 mmap 分配
  - P bit 為 1, 表示鄰近的上一塊 Chunk 正在使用中



Allocated Chunk

- Chunk 正在使用中, 圈選處目前作為 Data

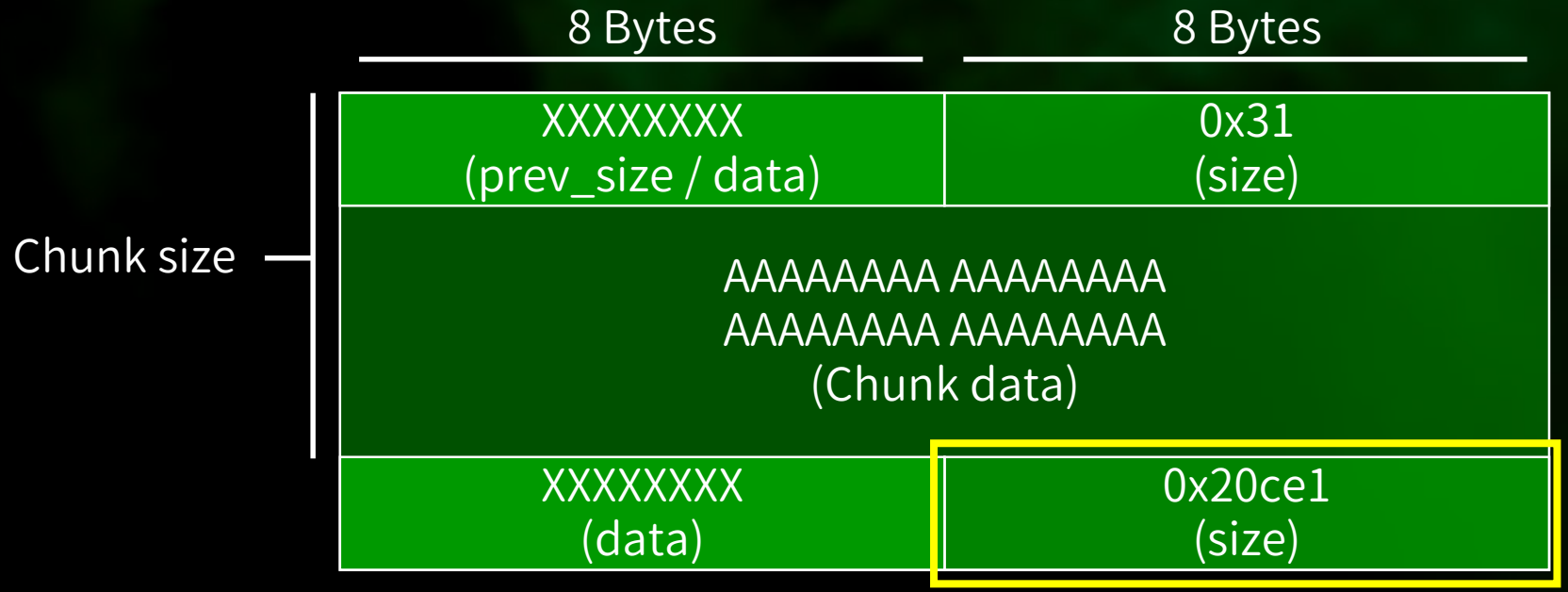


Allocated Chunk

- 下一個 Chunk 的 P bit 為 1, 表示其鄰近的上一塊 Chunk 目前正在使用中

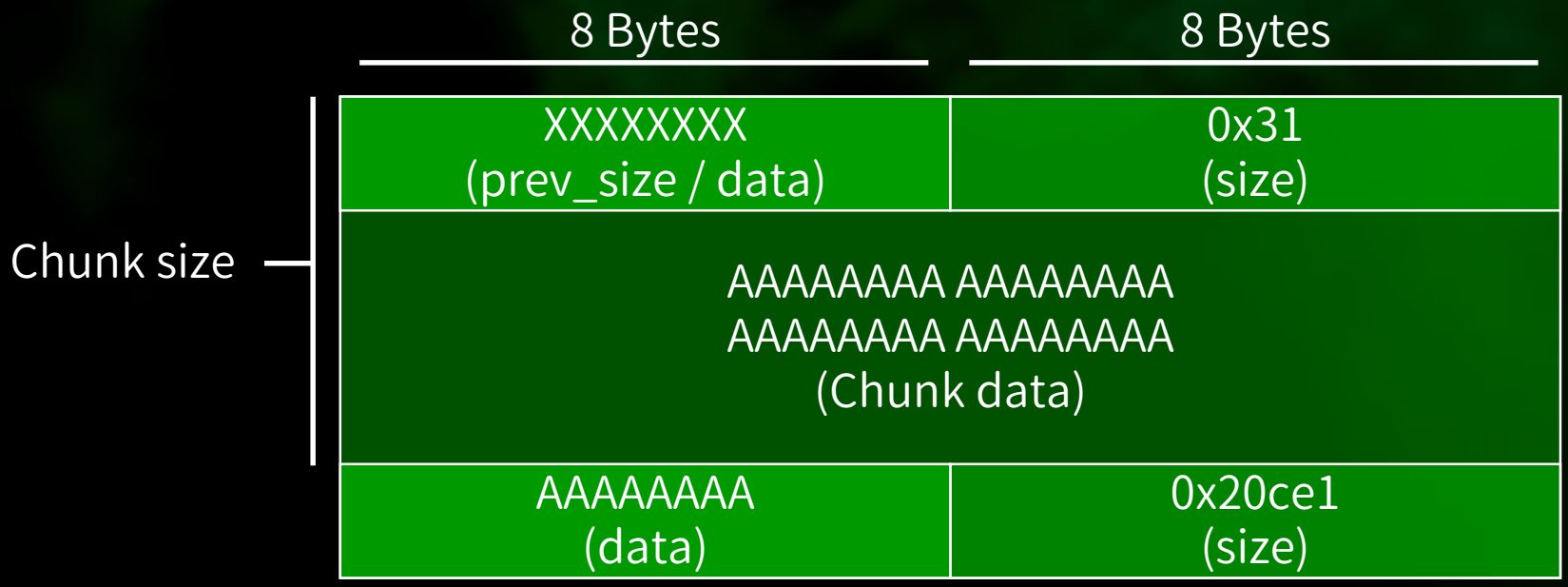
0xe1 -> 0b11100001

A M P



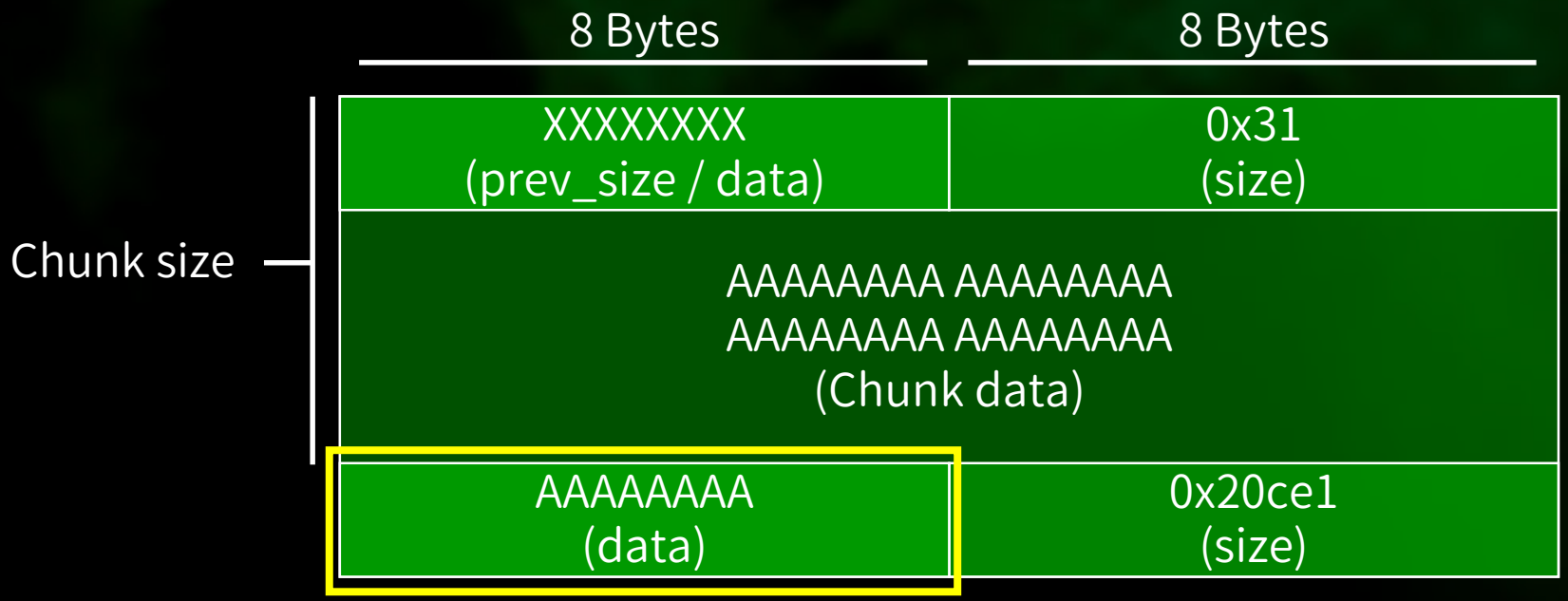
Allocated Chunk

- 來個不同的栗子
- malloc(0x28) 後寫入 0x28 個 A
- 計算 Chunk Size: (0x28 + 0x8 + 0xf) & ~0xf = 0x30



Allocated Chunk

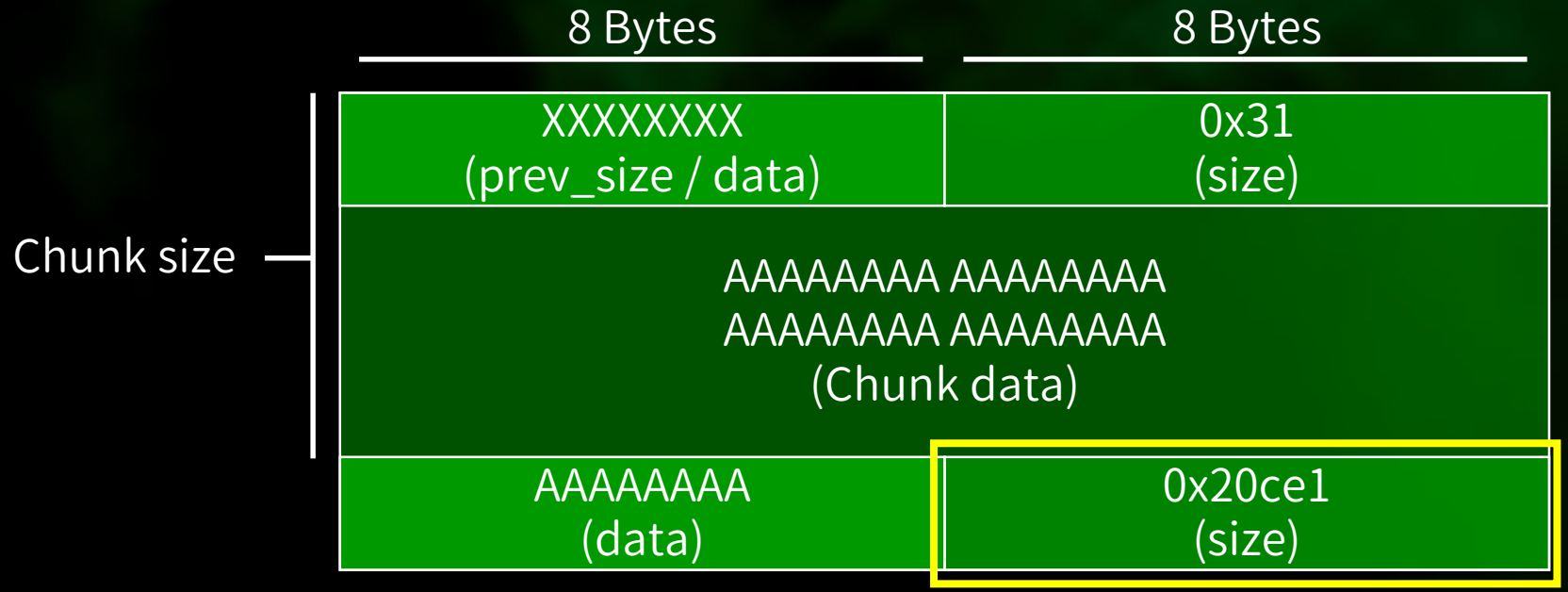
- Chunk 正在使用中, 圈選處目前作為 Data



Allocated Chunk

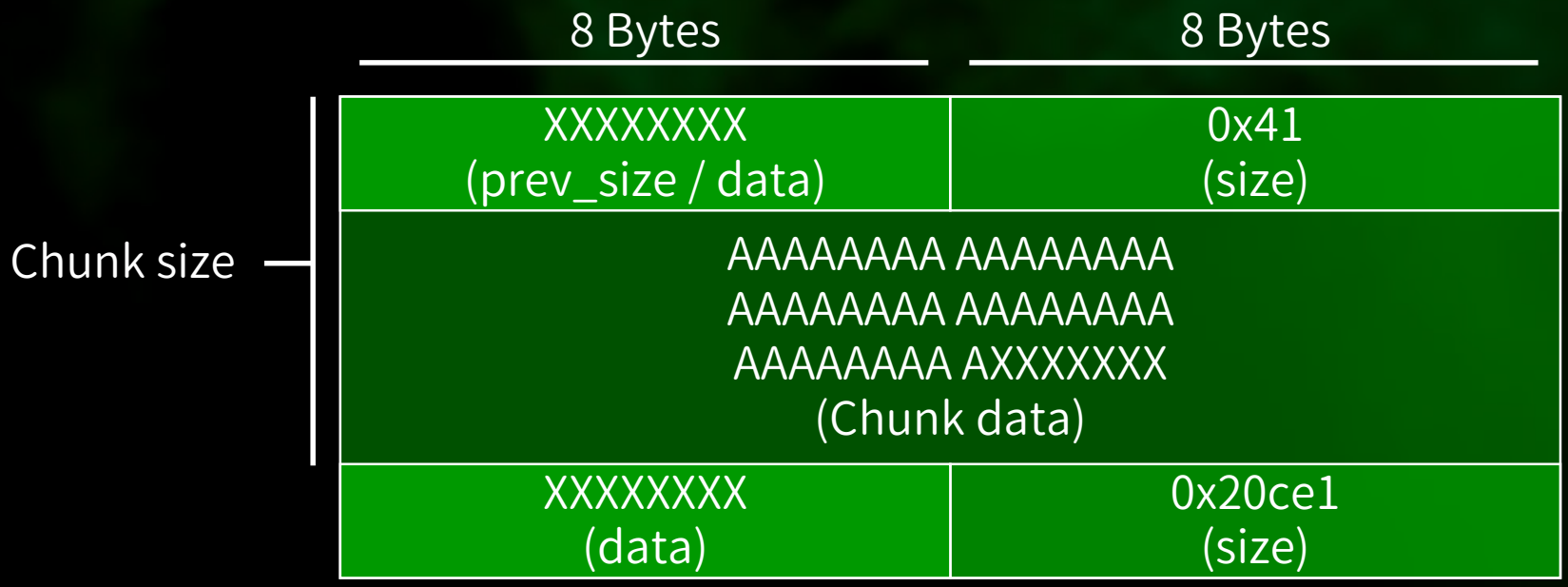
- 下一個 Chunk 的 P bit 為 1, 表示其鄰近的上一塊 Chunk 目前正在使用中

0xe1 -> 0b11100001
A M P



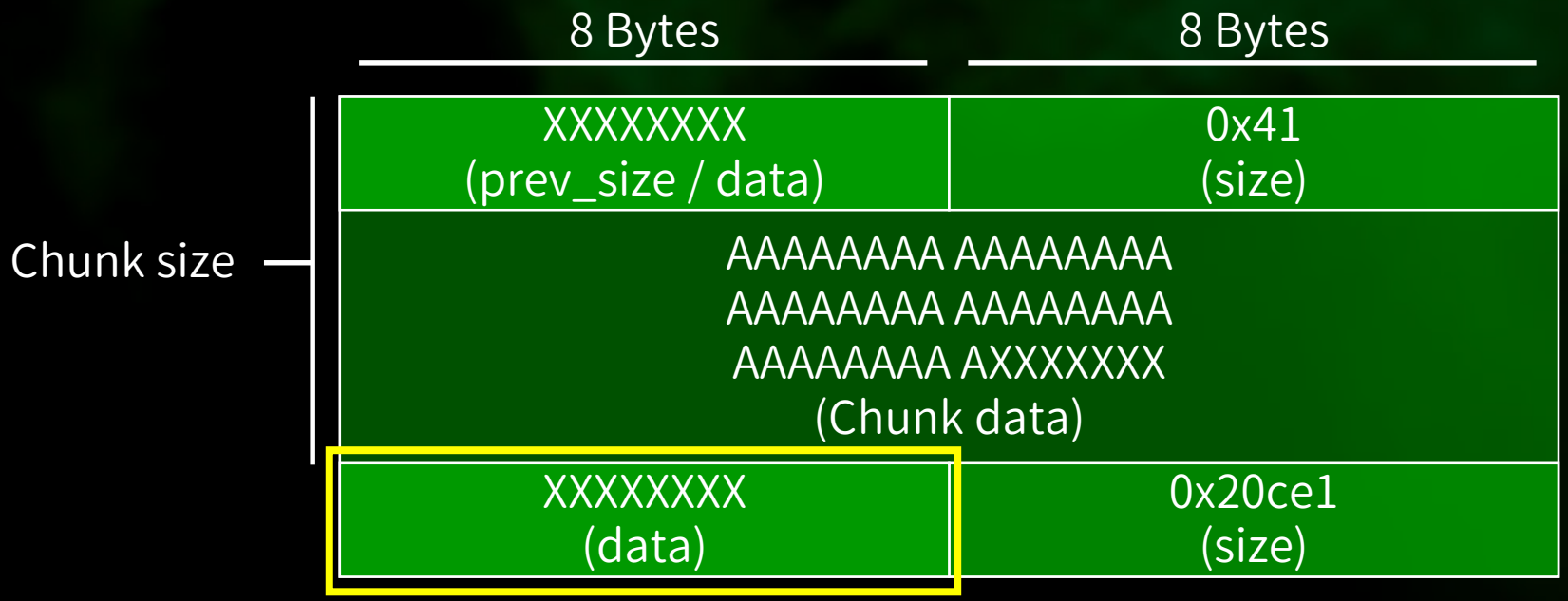
Allocated Chunk

- 再來個不同的栗子
- malloc(0x29) 後寫入 0x29 個 A
- 計算 Chunk Size: (0x29 + 0x8 + 0xf) & ~0xf = 0x40



Allocated Chunk

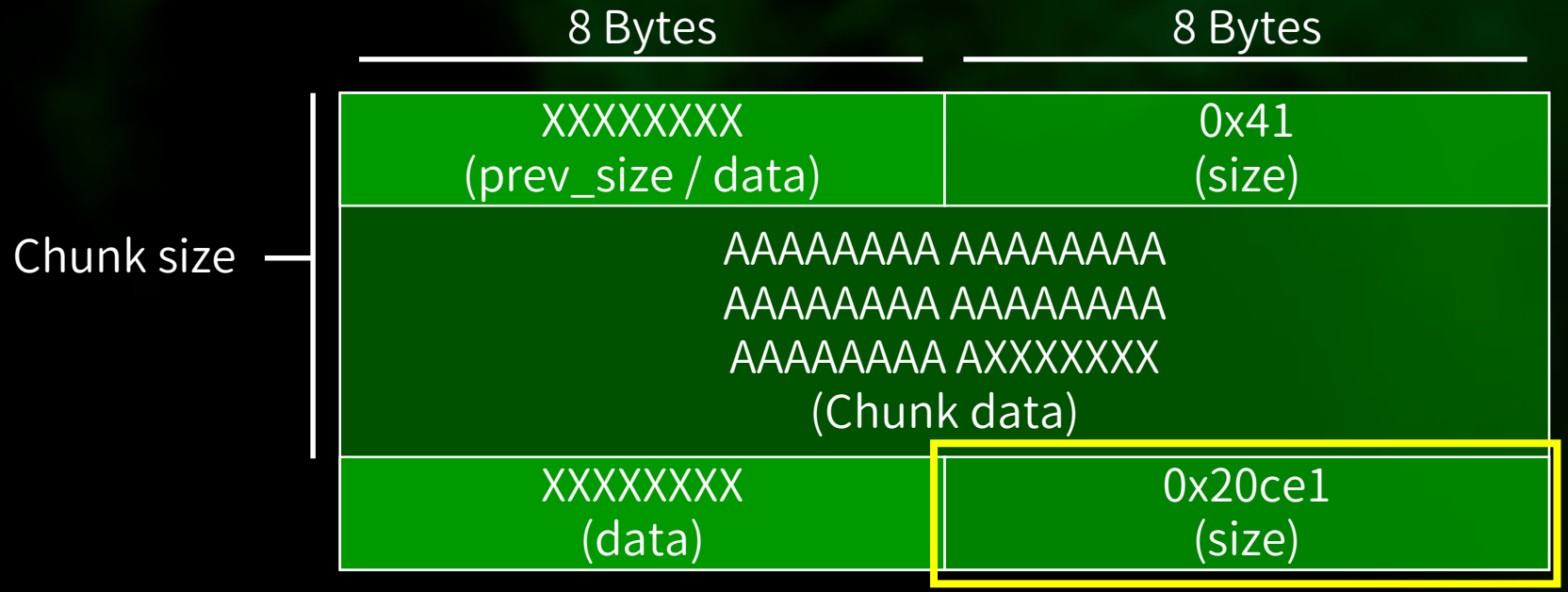
- Chunk 正在使用中, 圈選處目前作為 Data



Allocated Chunk

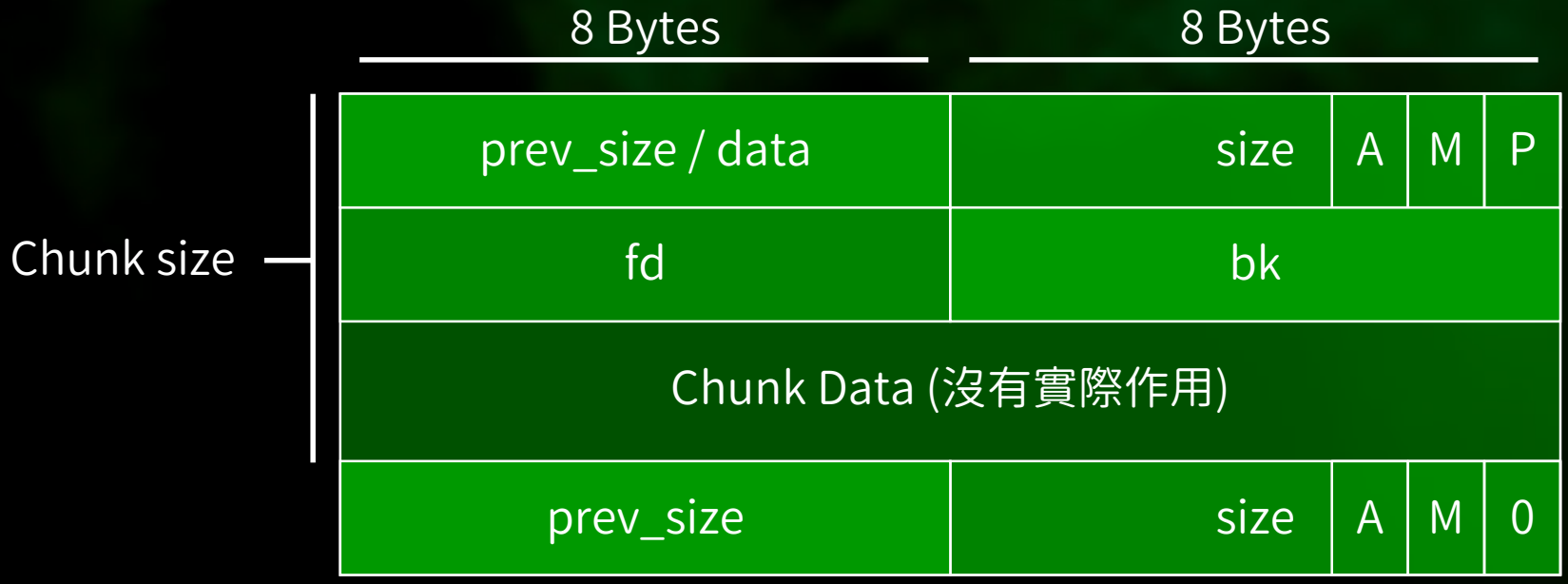
- 下一個 Chunk 的 P bit 為 1, 表示其鄰近的上一塊 Chunk 目前正在使用中

0xe1 -> 0b11100001
A M P



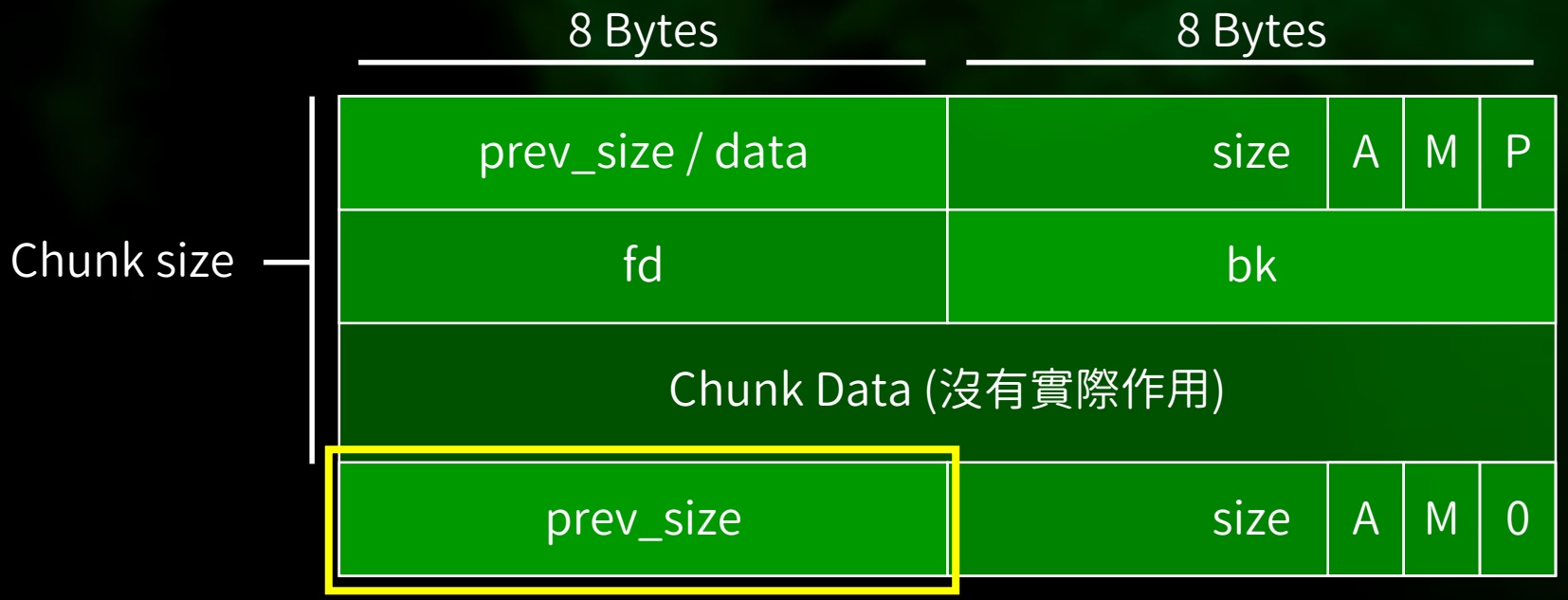
Free Chunk

- Free 掉後的 Chunk 會根據 Size 而進到不同的 Bins 中
- fd: Forward Pointer, 指向下一塊 Free 的 Chunk
- bk: Backward Pointer, 指向上一塊 Free 的 Chunk
- 以 fd, bk 將各個 Free Chunk 串聯起來



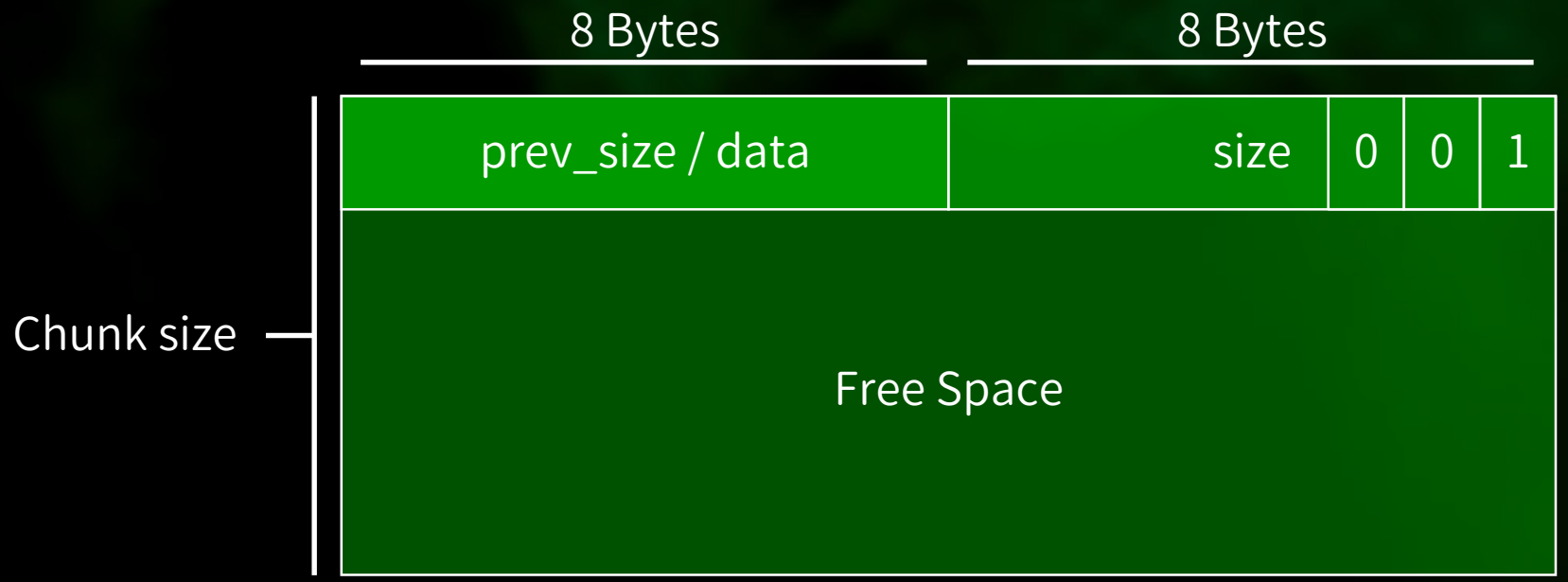
Free Chunk

- 若是 Free Chunk, 則圈選處作為 prev_size
- 下一塊 Chunk 通過 P 為 0 得知上一塊 Chunk 是 Free Chunk
- 下一塊 Chunk 通過 prev_size 得知上一塊 Chunk 大小



Top Chunk

- 在 Heap 頂端的 Chunk, 代表著剩餘的空間



Basic Knowledge

Fastbin

Fastbin

- Free 掉 Chunk Size 小於等於 global_max_fast 的 Chunk, 會回收至 Fastbin
- global_max_fast 預設為 0x80
- Fastbin 共有 7 個, 分別為 [0x20, 0x30, 0x40, ⋯, 0x80]
- 為 singly linked list
- e.g.
  - Free 掉 Chunk Size 為 0x20 的 Chunk, 會進到代表 0x20 Fastbin 的鏈表
- Free 這類 Chunk 時, 不會清除下一塊 Chunk 的 P bit

Fastbin

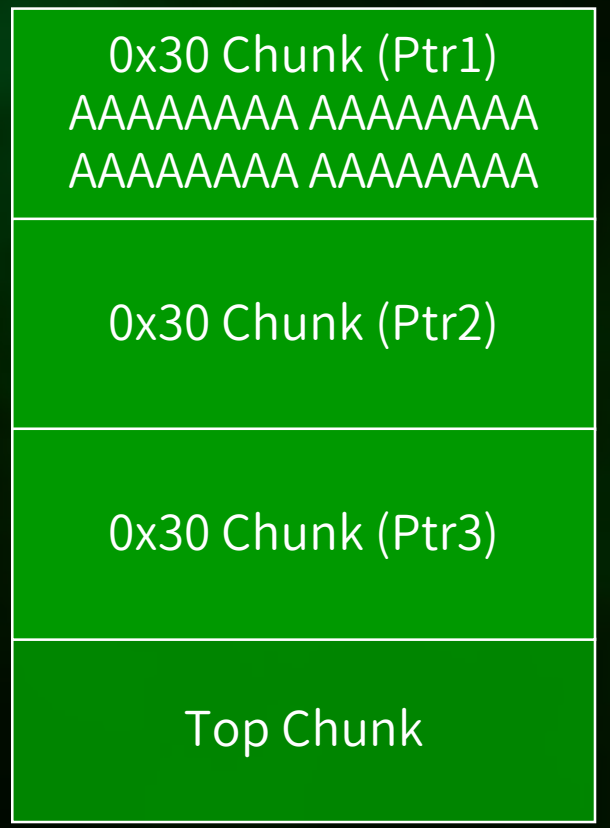
- 釋放 ptr1 指向的 Chunk, 首先先把圖改詳細一點

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
```

main_arena.fastbinsY



Heap

Fastbin

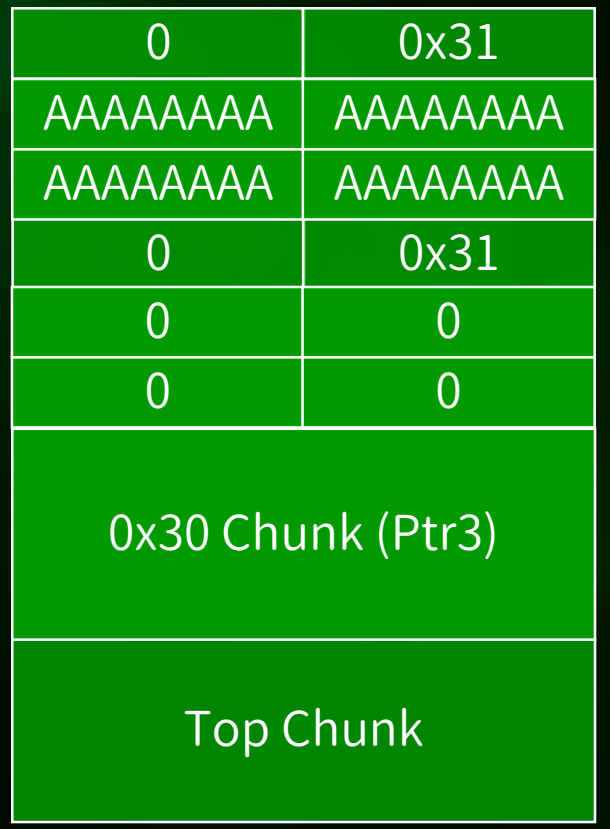
- 釋放 ptr1 指向的 Chunk

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
```

main_arena.fastbinsY



Heap

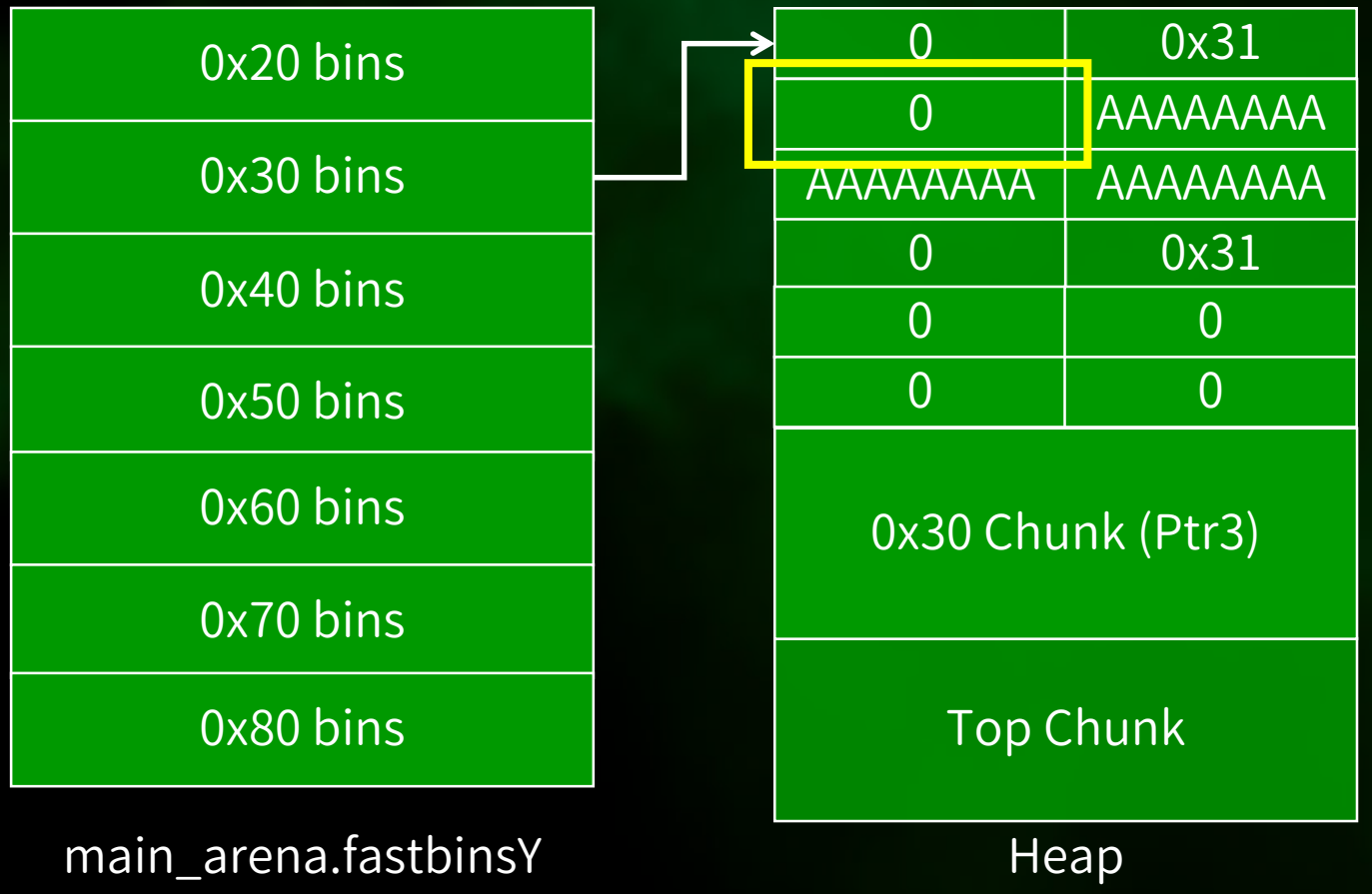
Fastbin

- 入鏈, 向 fd 寫入 list head 後, 將 list head 指向該 Chunk

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
```



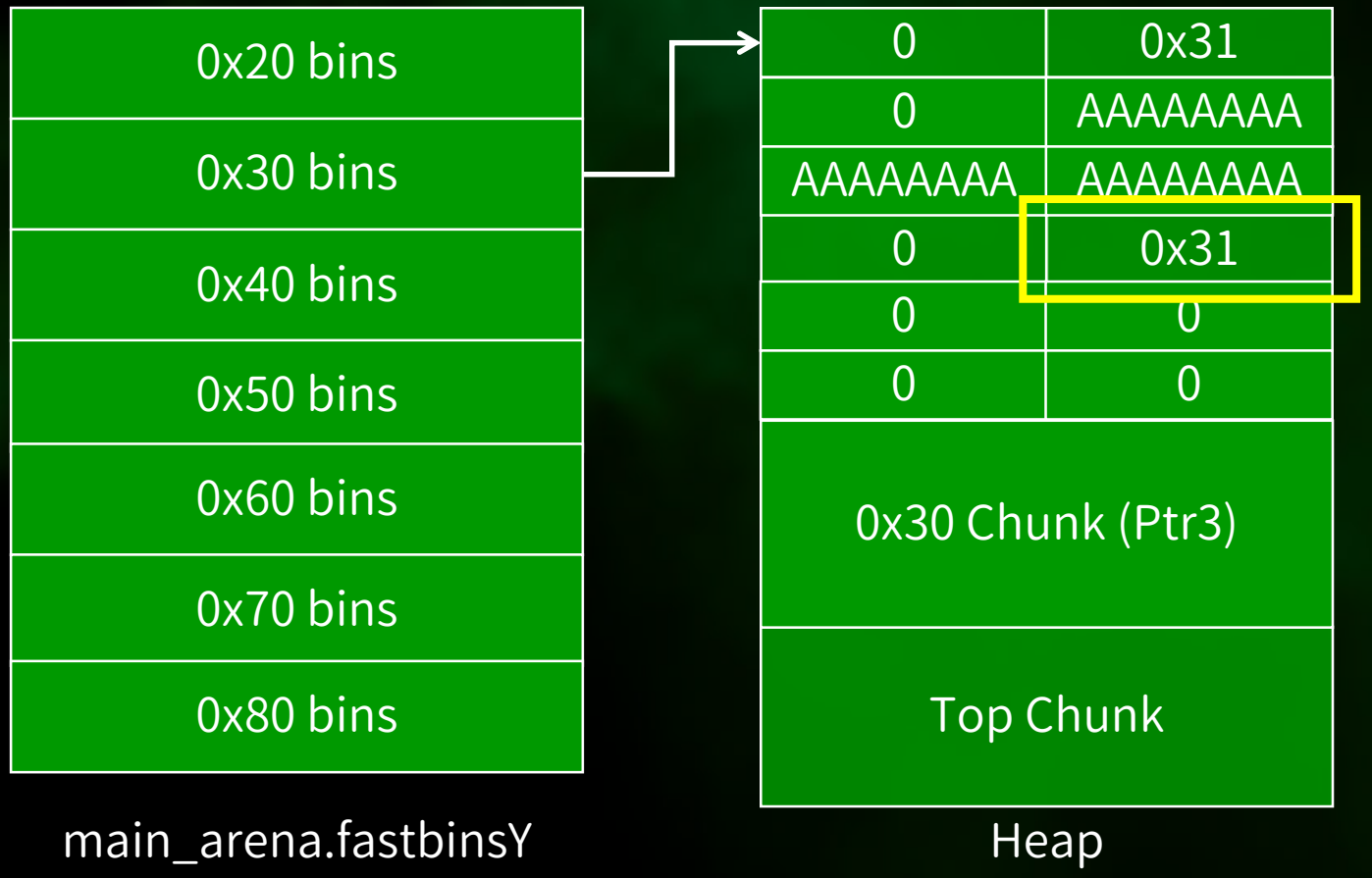
Fastbin

- 可以注意到下一塊 Chunk 的 P bit 並沒有被清除

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
```



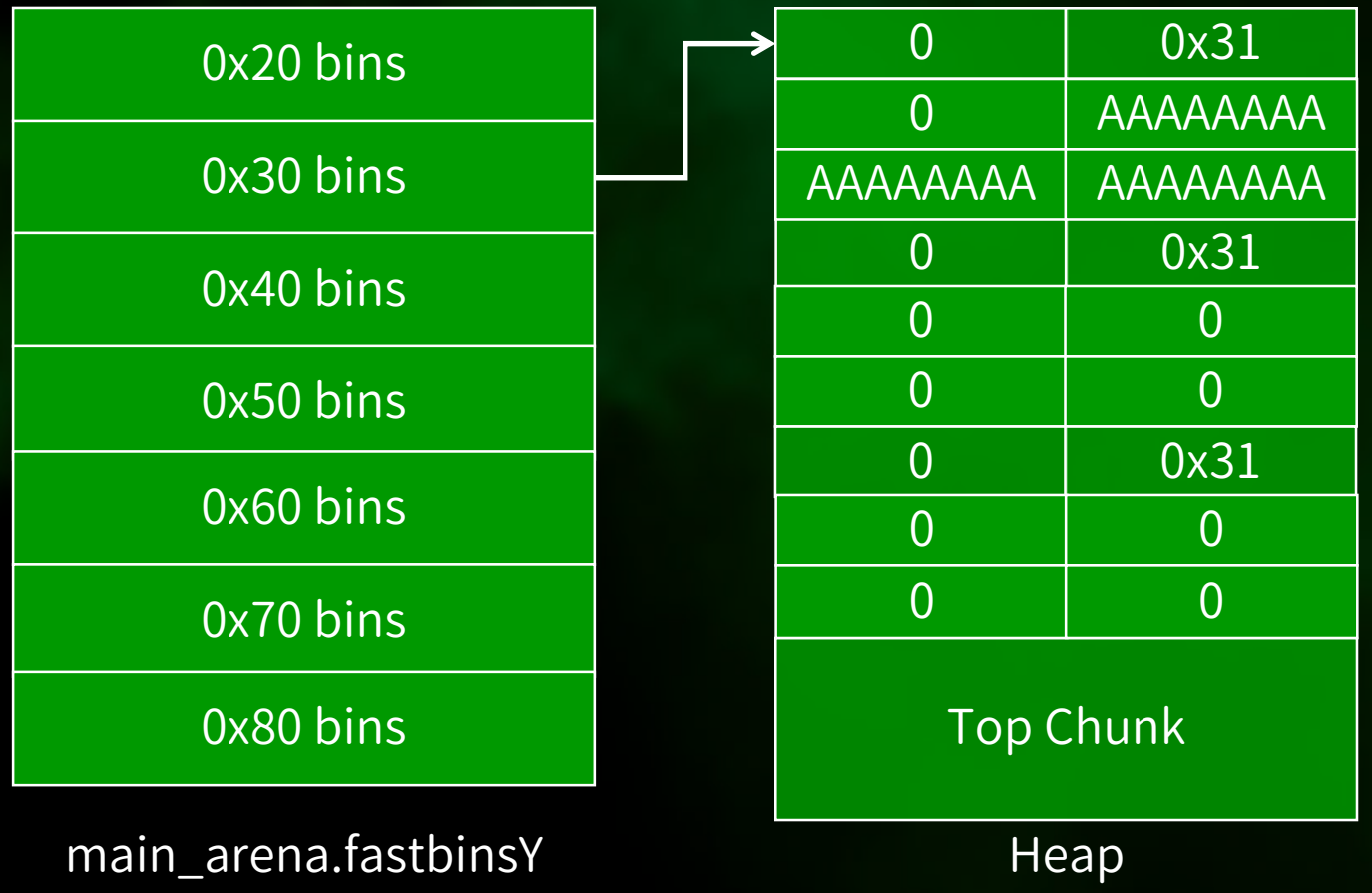
Fastbin

- 繼續 free ptr2 指向的 Chunk

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
```



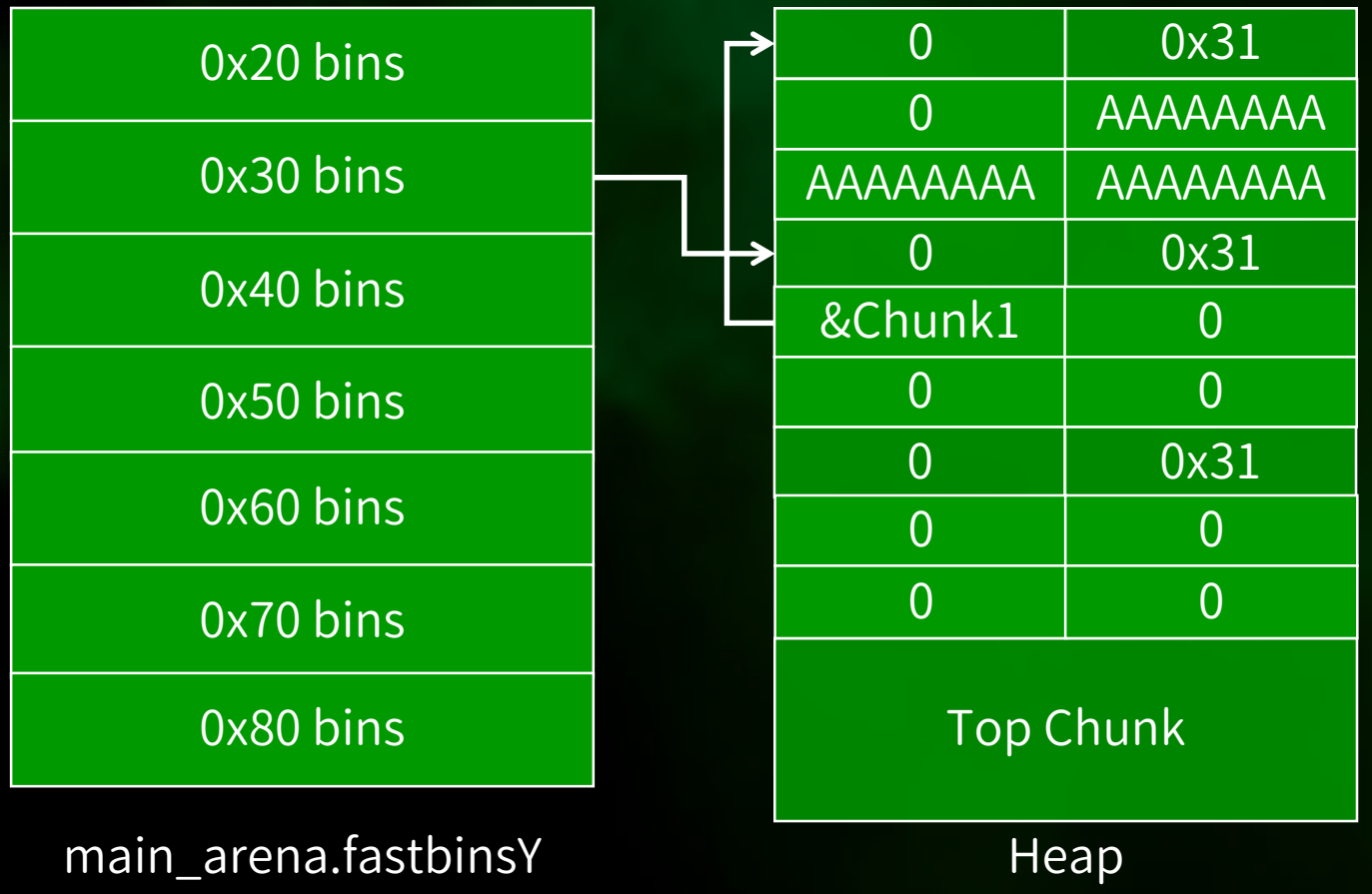
Fastbin

- 入鏈, 向 fd 寫入 list head 後, 將 list head 指向該 Chunk

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
```



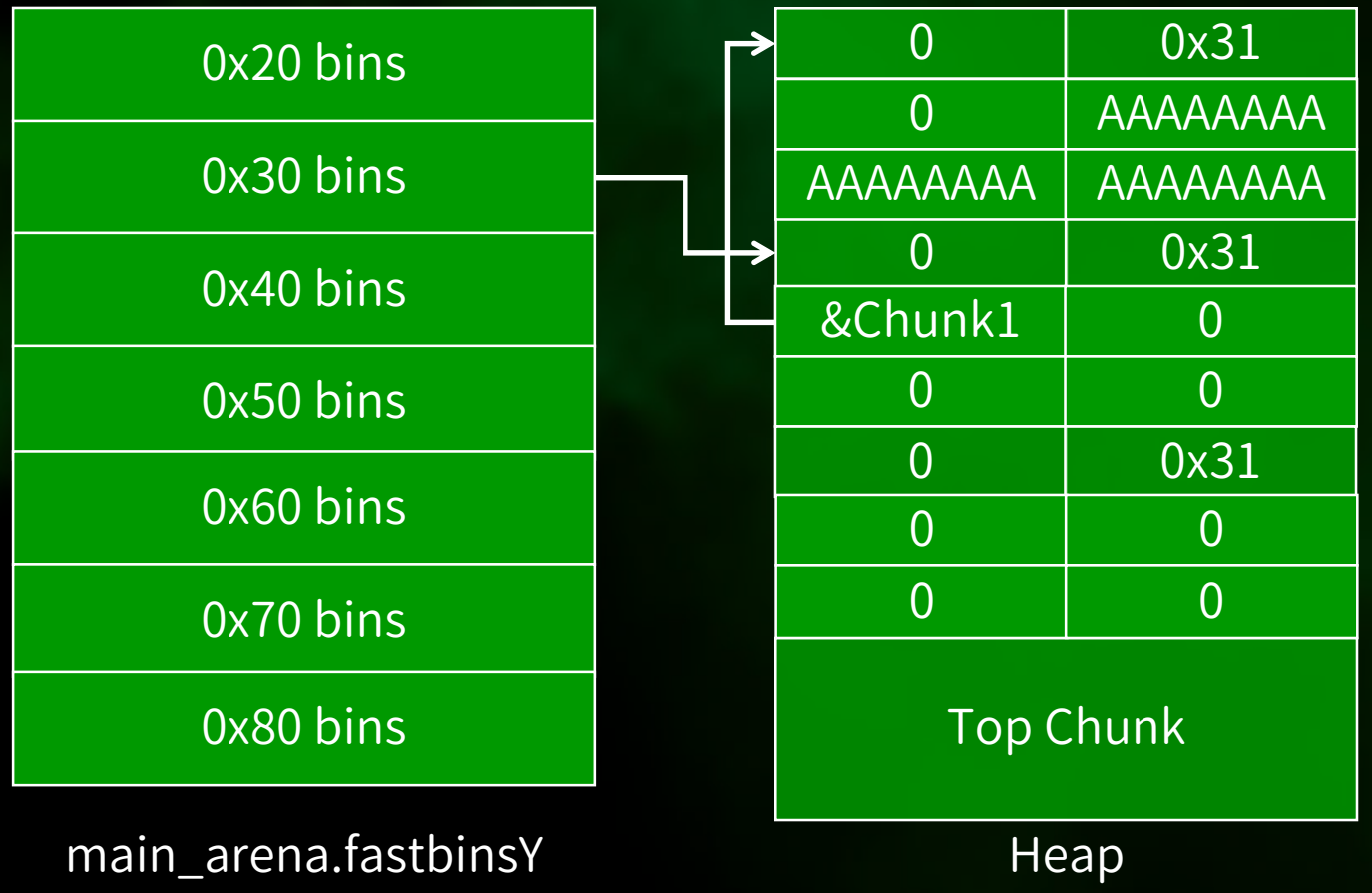
Fastbin

- 繼續 free ptr3 指向的 Chunk

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
```



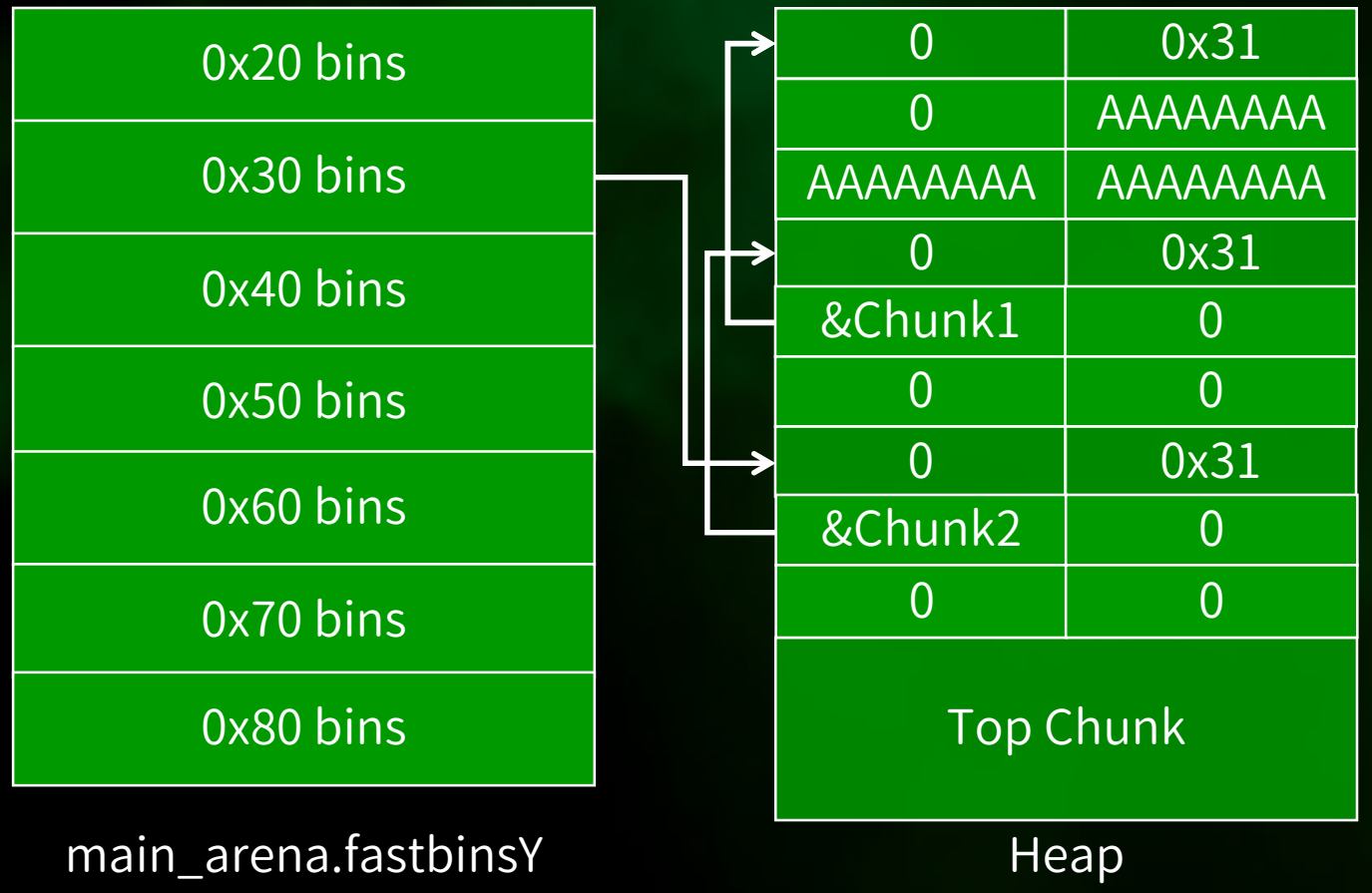
Fastbin

- 入鏈, 向 fd 寫入 list head 後, 將 list head 指向該 Chunk

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
```



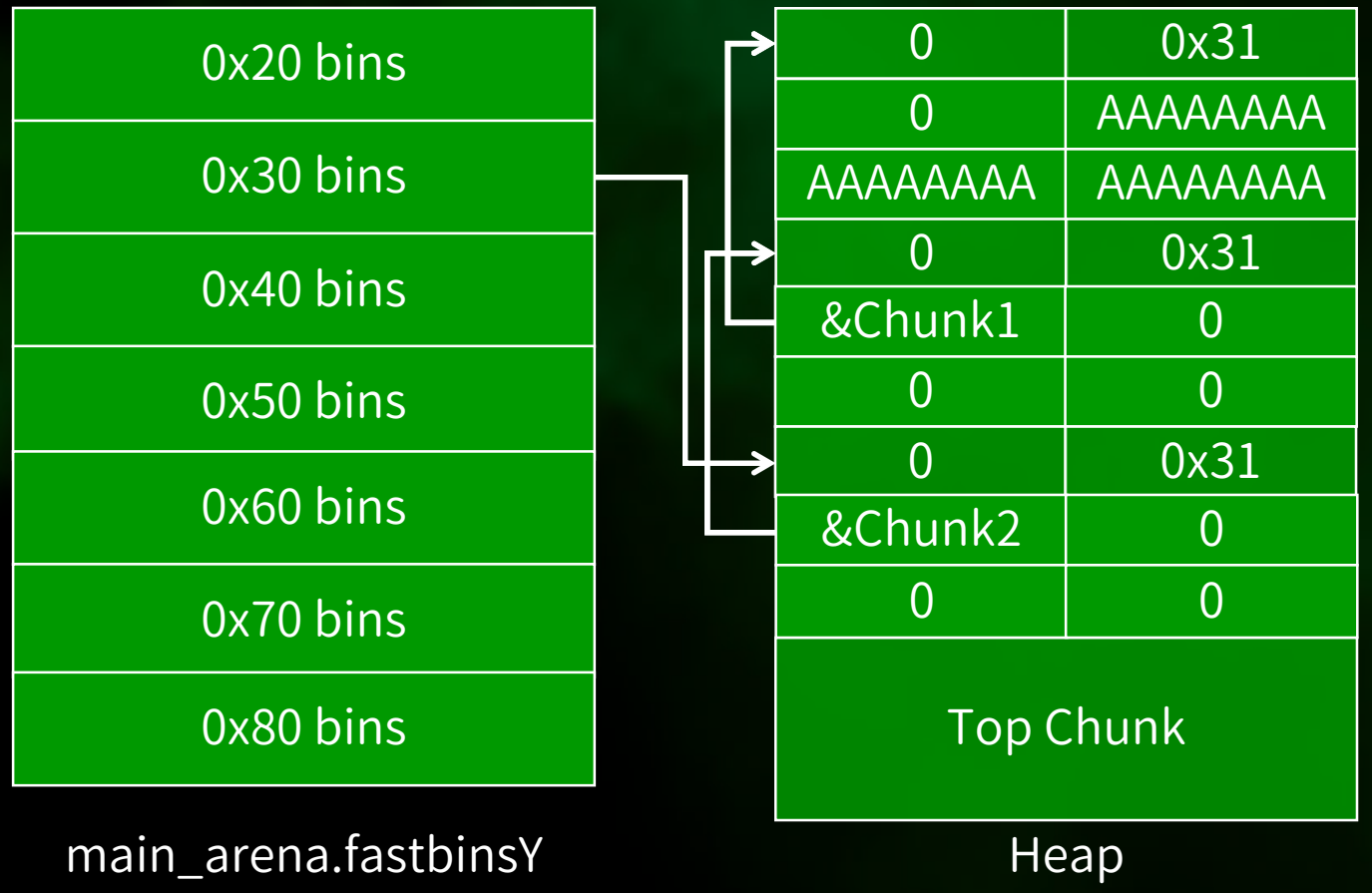
Fastbin

- 若後續 malloc 的 Chunk Size 為 0x30

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
char *ptr4 = malloc(0x20);
```



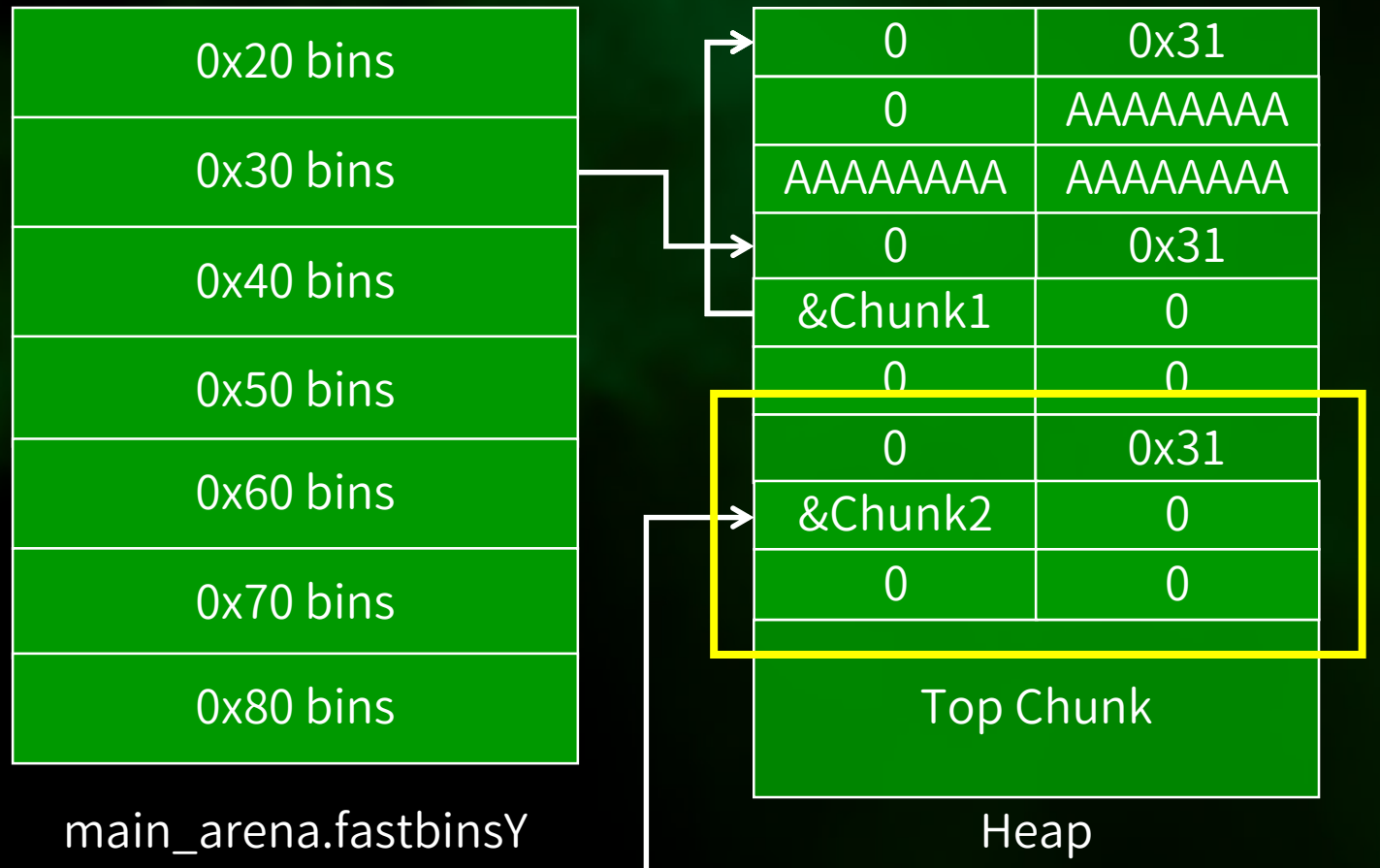
Fastbin

- 則從 0x30 Fastbin 拿出一個 Chunk

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
char *ptr4 = malloc(0x20);
```



Fastbin

- 小總結
- LIFO
- fd 指向下一塊 Free Chunk 的 Chunk Header
- 不會改鄰近的下一塊 Chunk 的 P bit
- 在 free 時, 如果下一塊是 Top Chunk, 並不會被合併進去

Fastbin in Libc 2.23

Source Code Reading

Fastbin in Libc 2.23

Demo

Basic Knowledge

Tcache

Tcache

- 從 libc 2.26 開始使用
- 為了再加速程式效率而誕生
- Tcache 有許多, 分別為 [0x20, 0x30, 0x40, ⋯, 0x410]
- 為 singly linked list
- 每個 Tcache 最多收 7 個 Chunks
- Free 這類 Chunk 時, 不會清除下一塊 Chunk 的 P bit
- 用結構 tcache_perthread_struct 管理 Tcache
  - 指向此結構的指標存在於 TLS 中

Tcache

- 第一次呼叫 malloc, 首先申請一大塊記憶體作為 Heap

C

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
char *ptr4 = malloc(0x20);
```

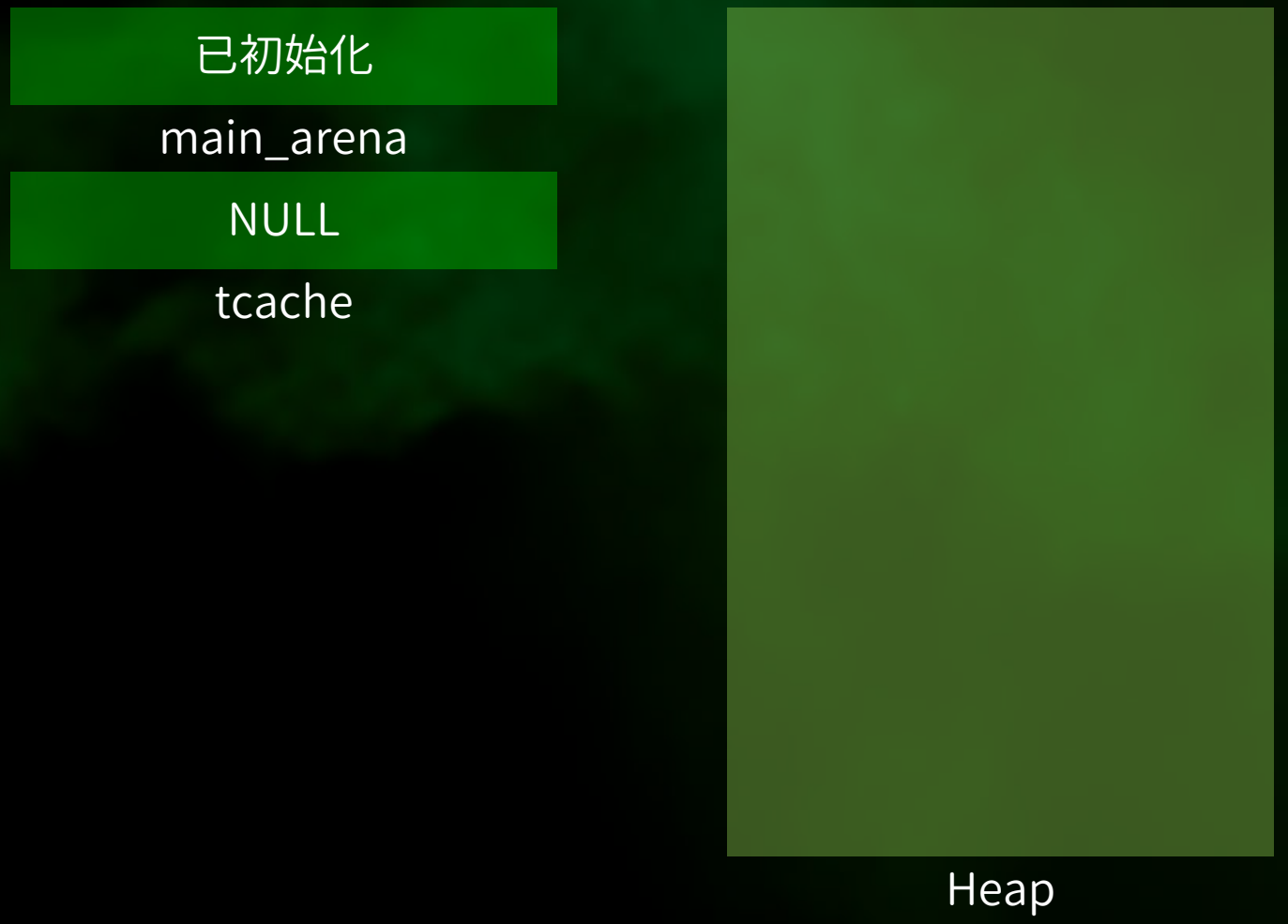
Tcache

- 接著初始化 tcache_perthread_struct

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
char *ptr4 = malloc(0x20);
```



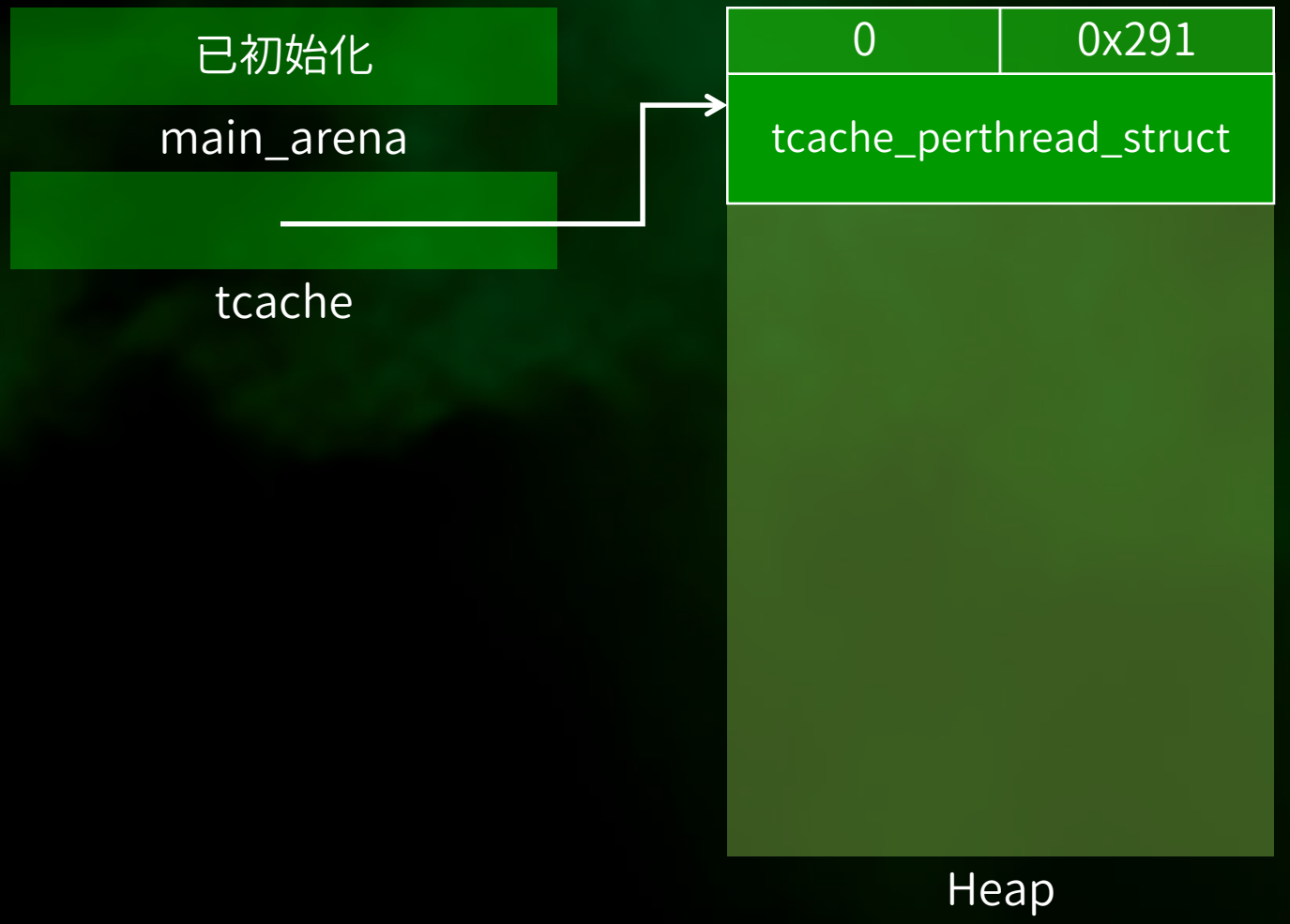
Tcache

- 接著初始化 tcache_perthread_struct, 展開看一下內部

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
char *ptr4 = malloc(0x20);
```



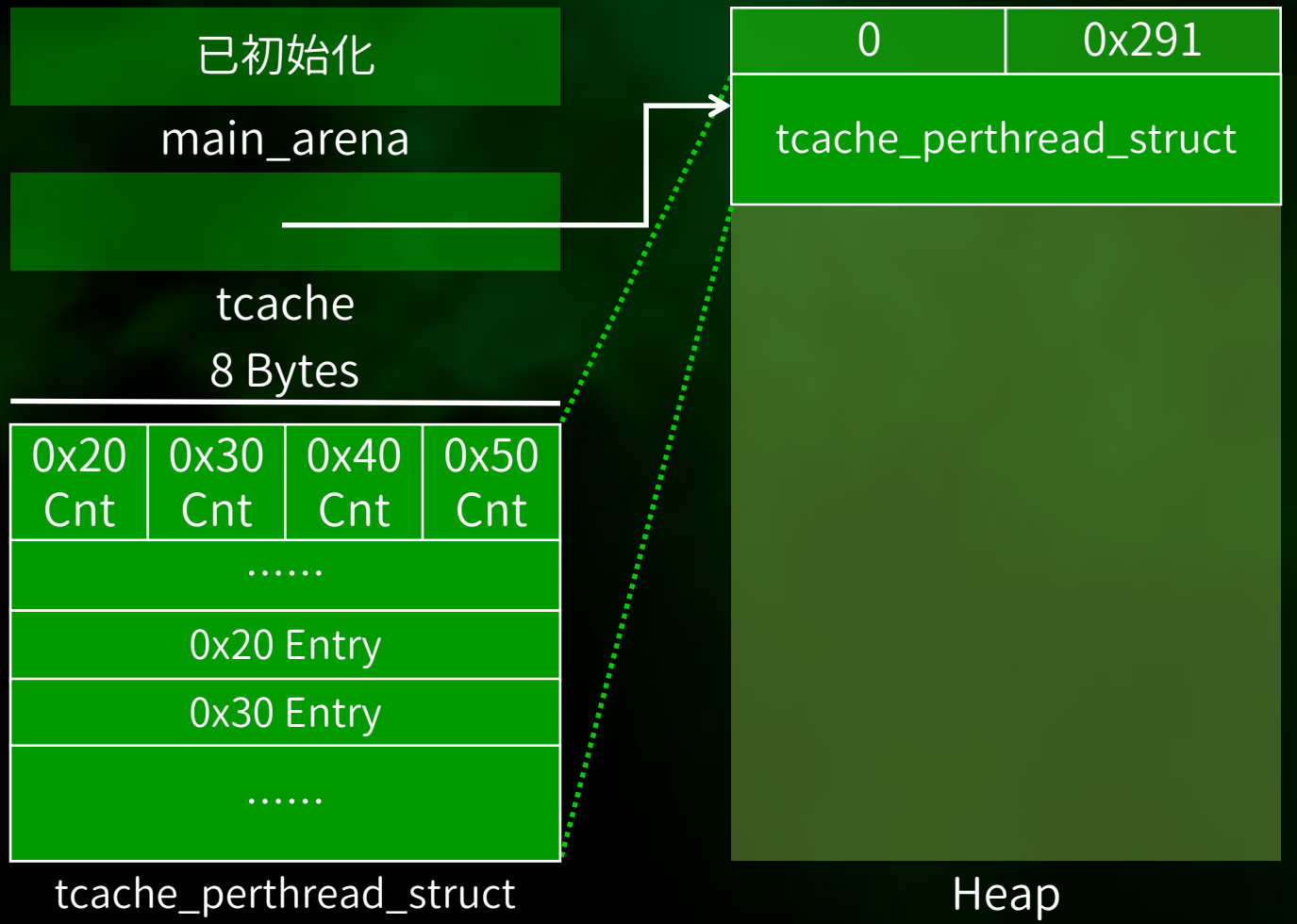
Tcache

- tcache_perthread_struct 分成 Counts 和 Entries

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
char *ptr4 = malloc(0x20);
```



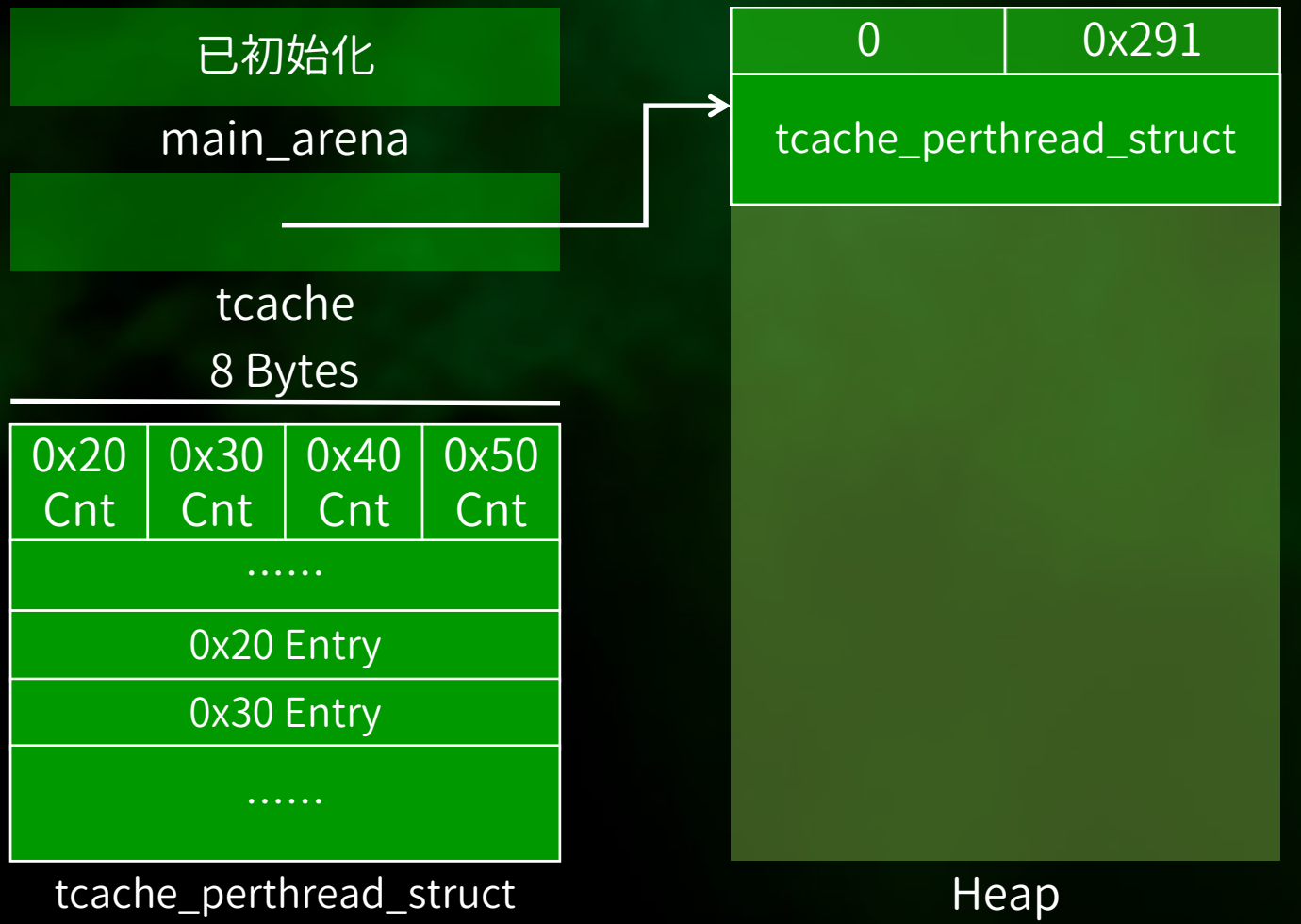
Tcache

- 初始化都做完後才分配 Chunk

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
char *ptr4 = malloc(0x20);
```



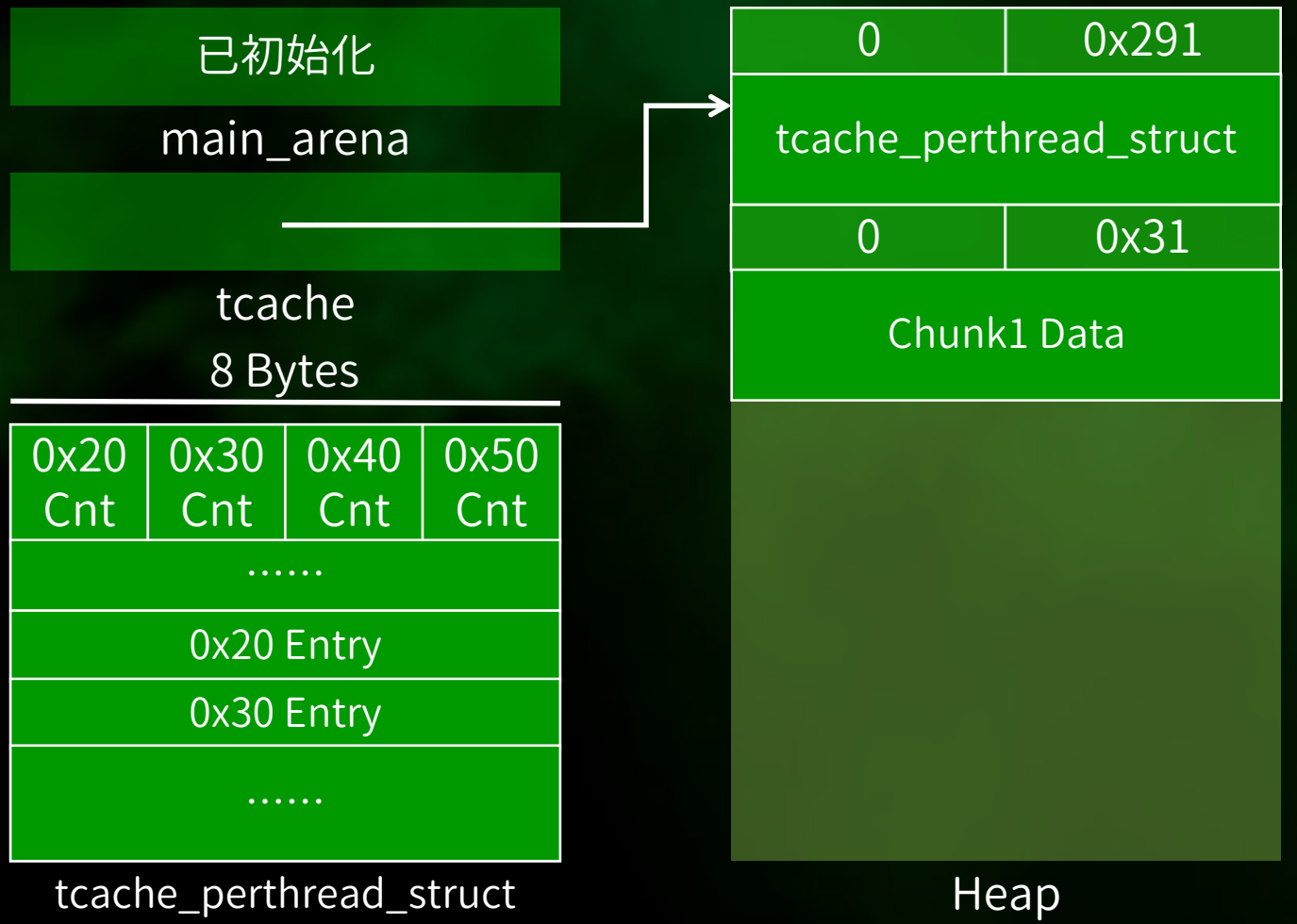
Tcache

- 分配 Chunk2

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
char *ptr4 = malloc(0x20);
```



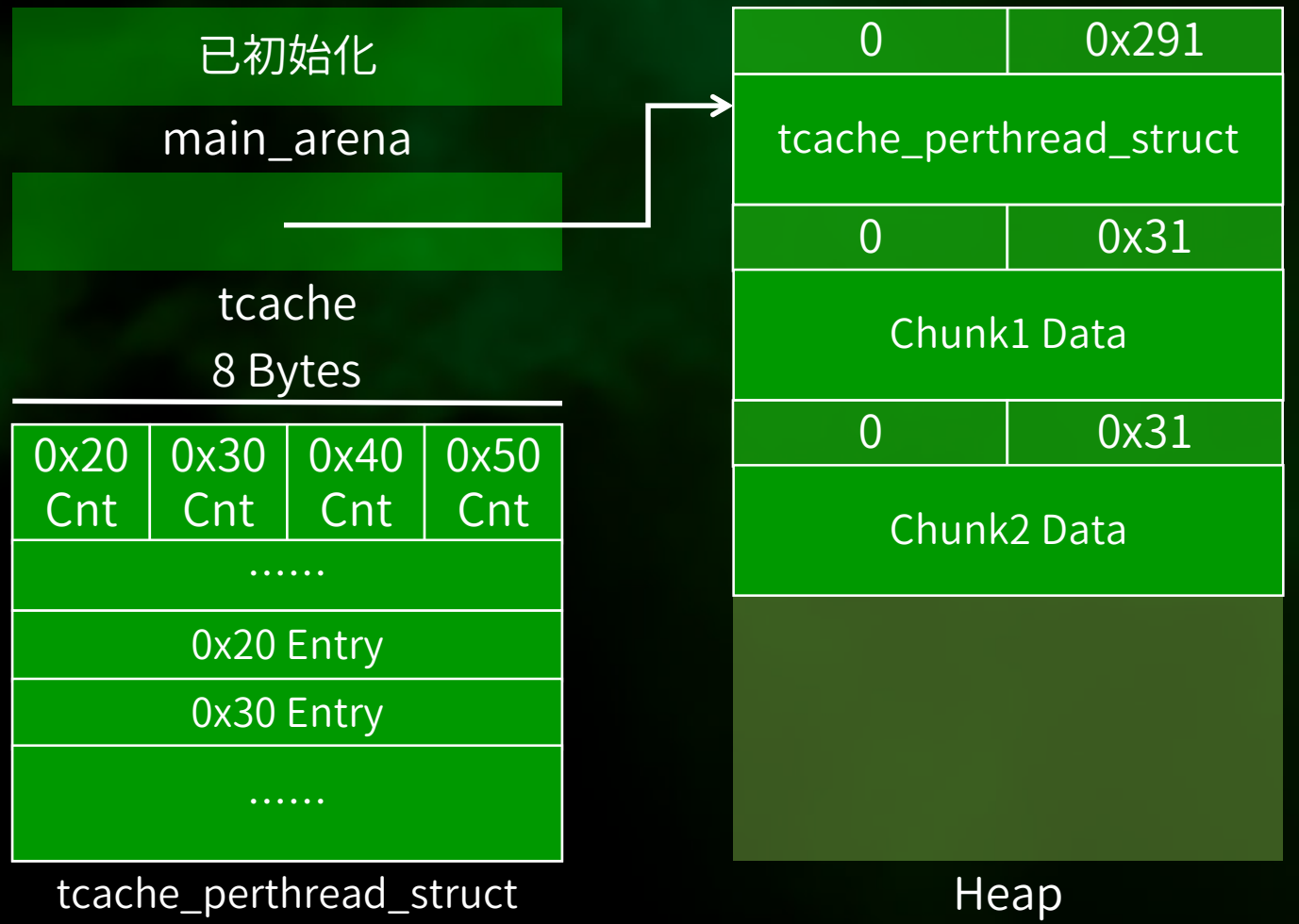
Tcache

- 分配 Chunk3

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
char *ptr4 = malloc(0x20);
```



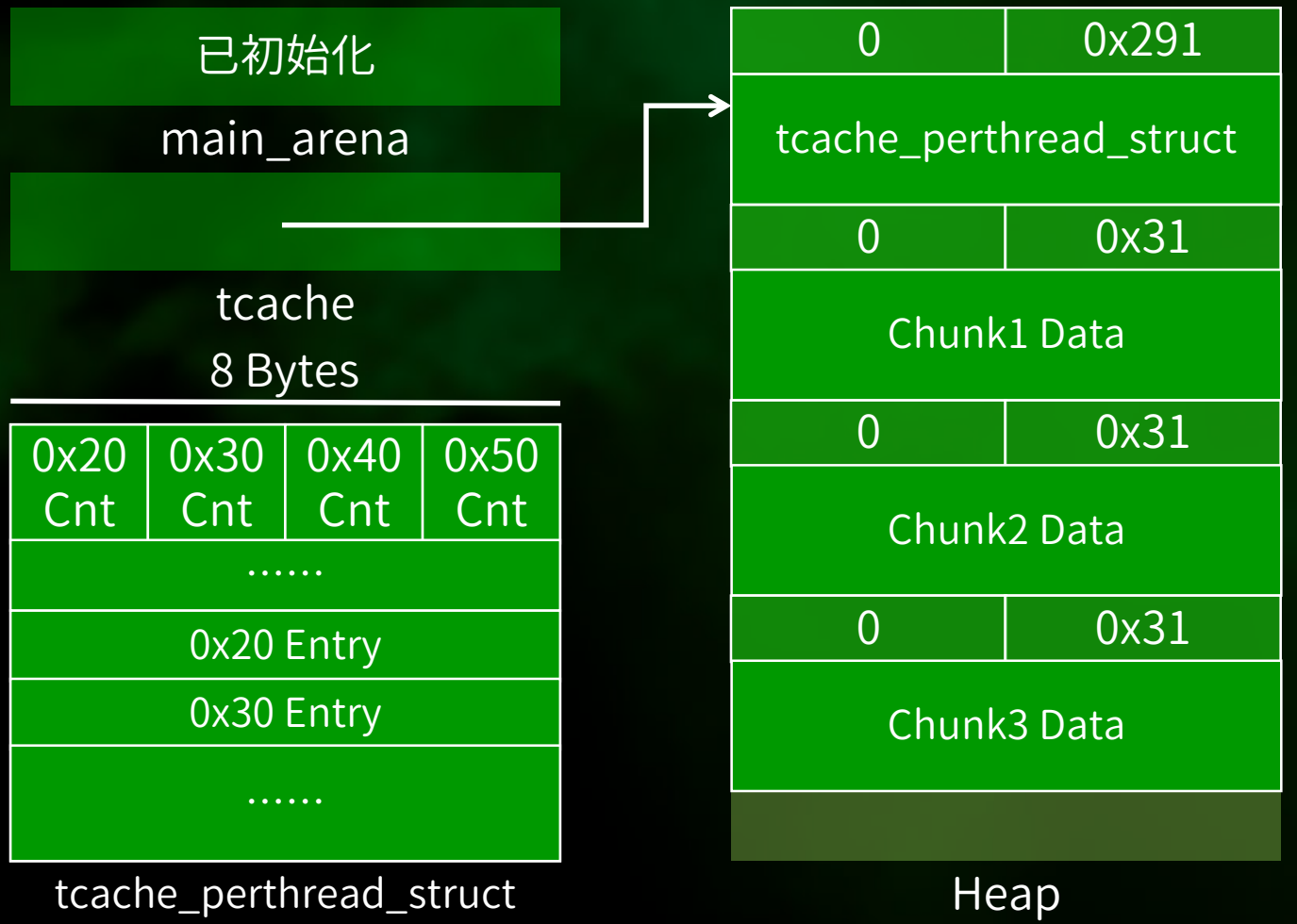
Tcache

- 寫入 0x20 個 A 至 Chunk1

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
char *ptr4 = malloc(0x20);
```



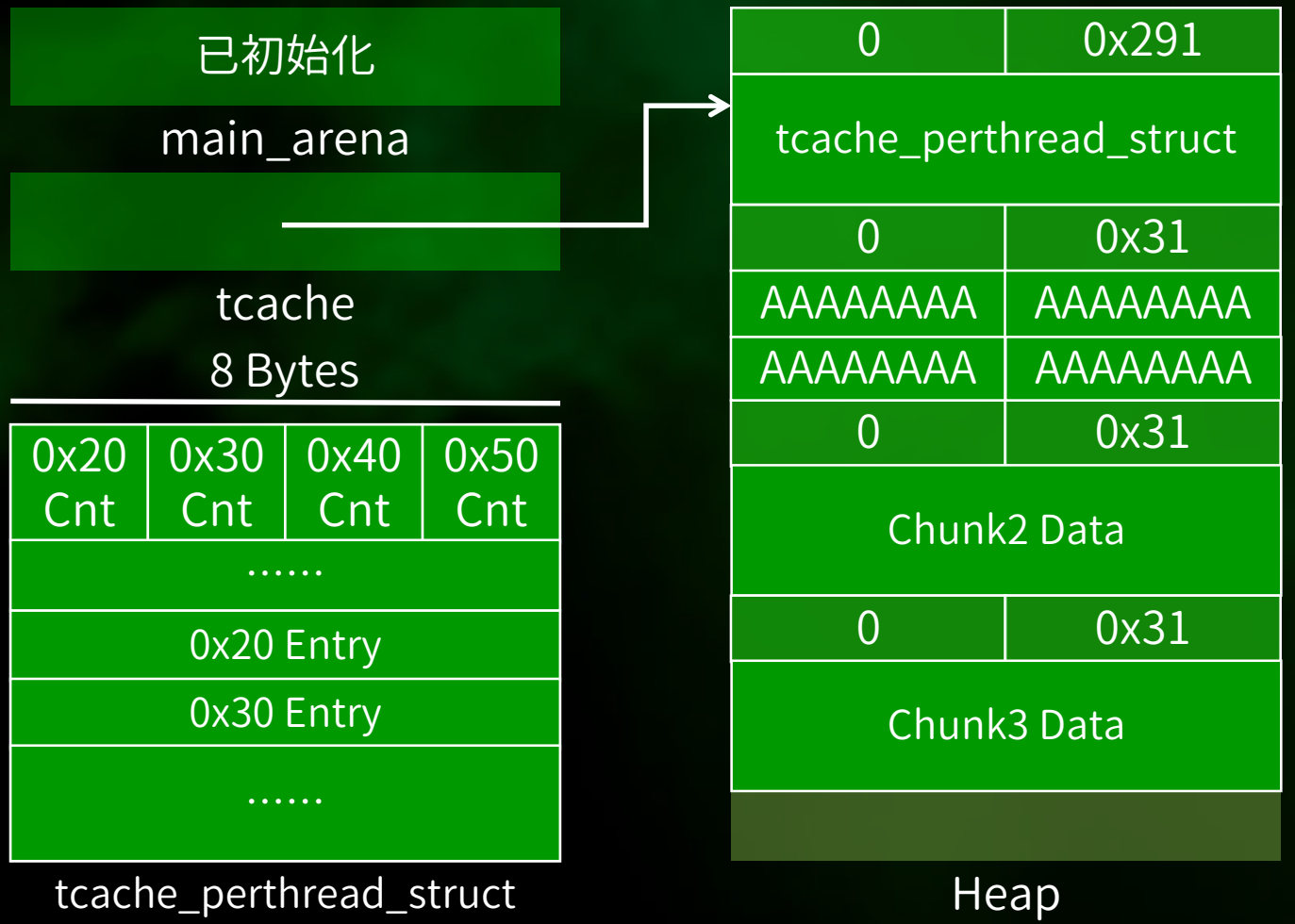
Tcache

- Free Chunk1

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
char *ptr4 = malloc(0x20);
```



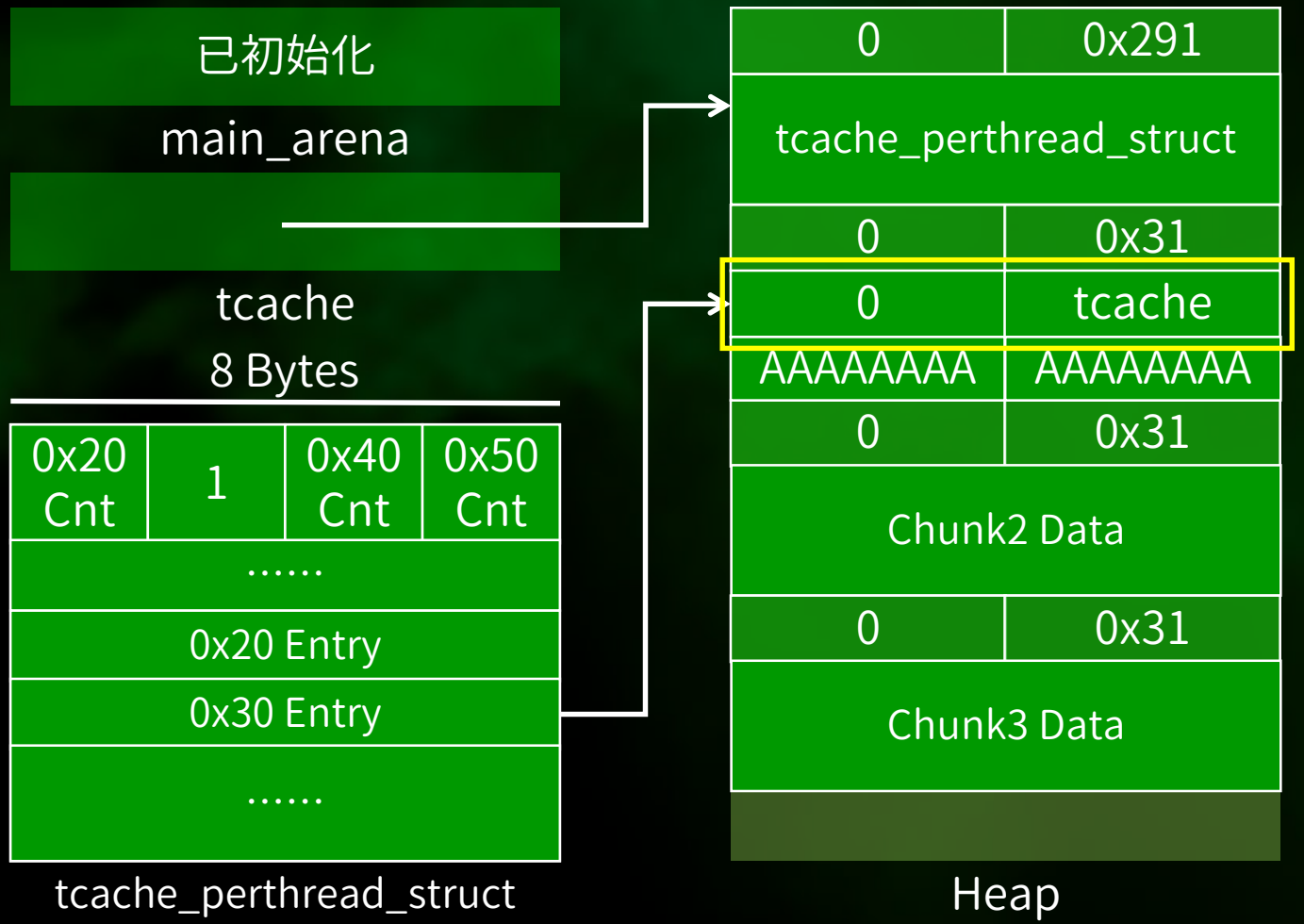
Tcache

- 填入 fd, 此時 bk 不代表 bk, 而是代表 Key, 用作於安全檢查

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
char *ptr4 = malloc(0x20);
```



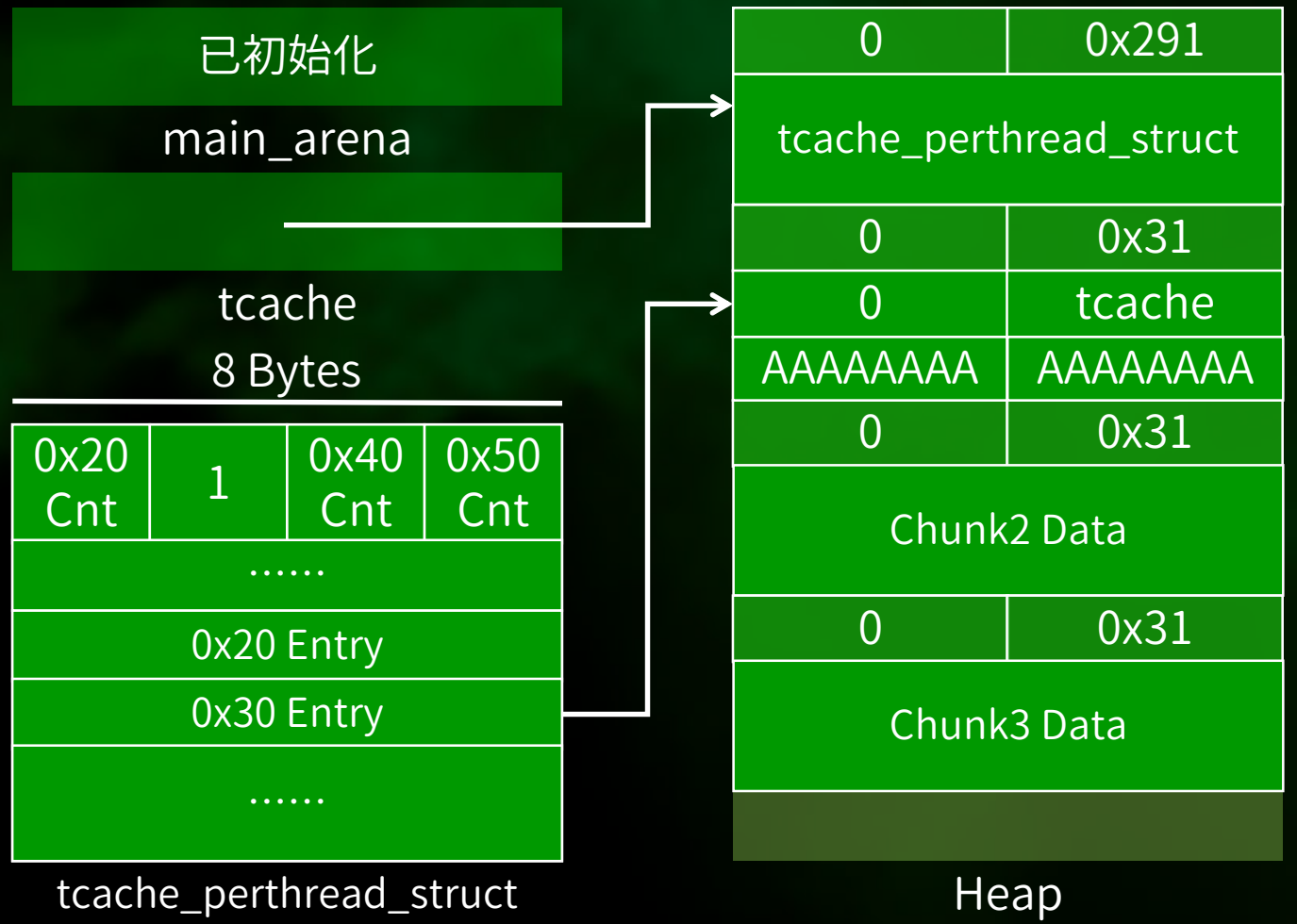
Tcache

- Free Chunk2

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
char *ptr4 = malloc(0x20);
```



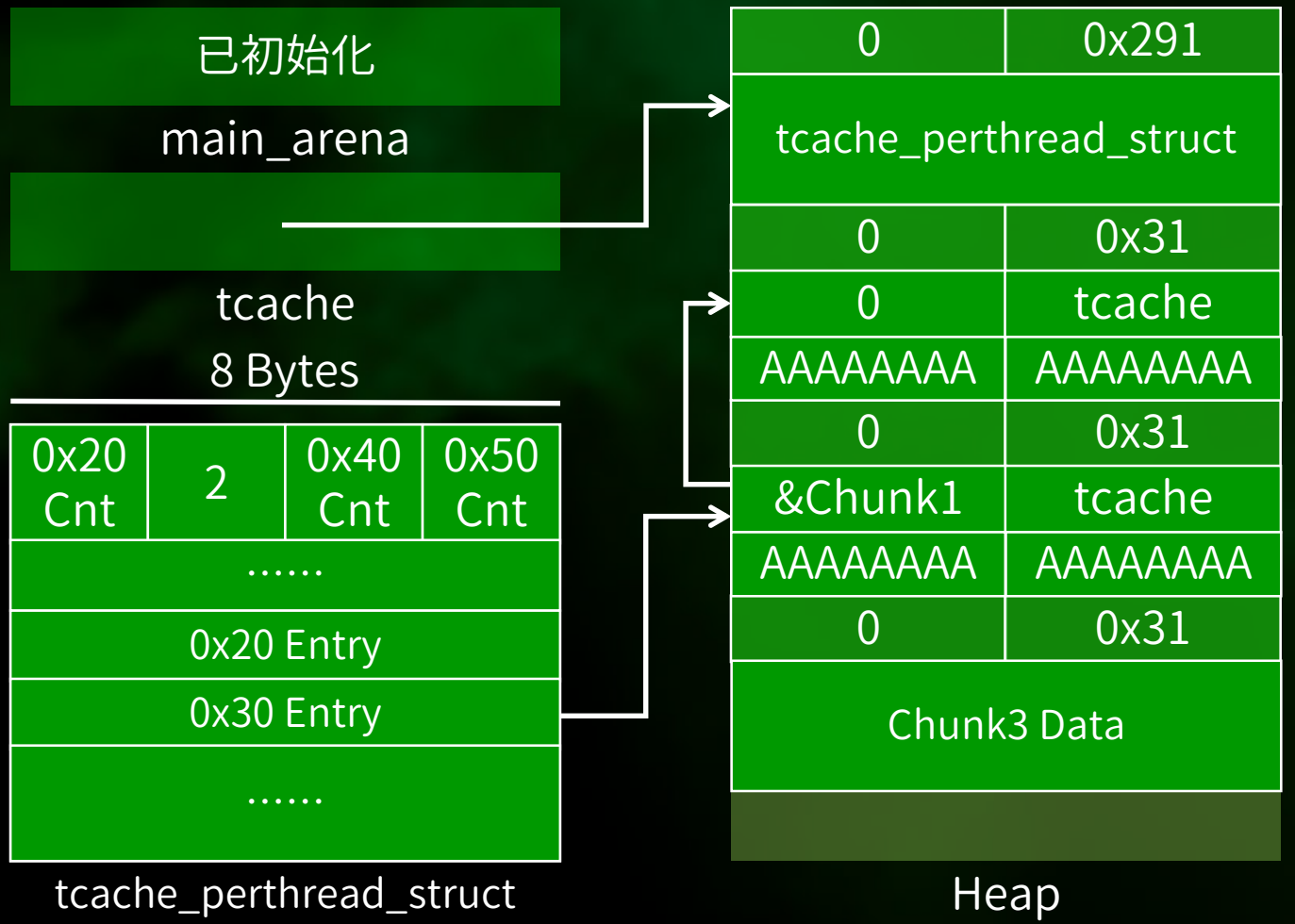
Tcache

- Free Chunk3

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
char *ptr4 = malloc(0x20);
```



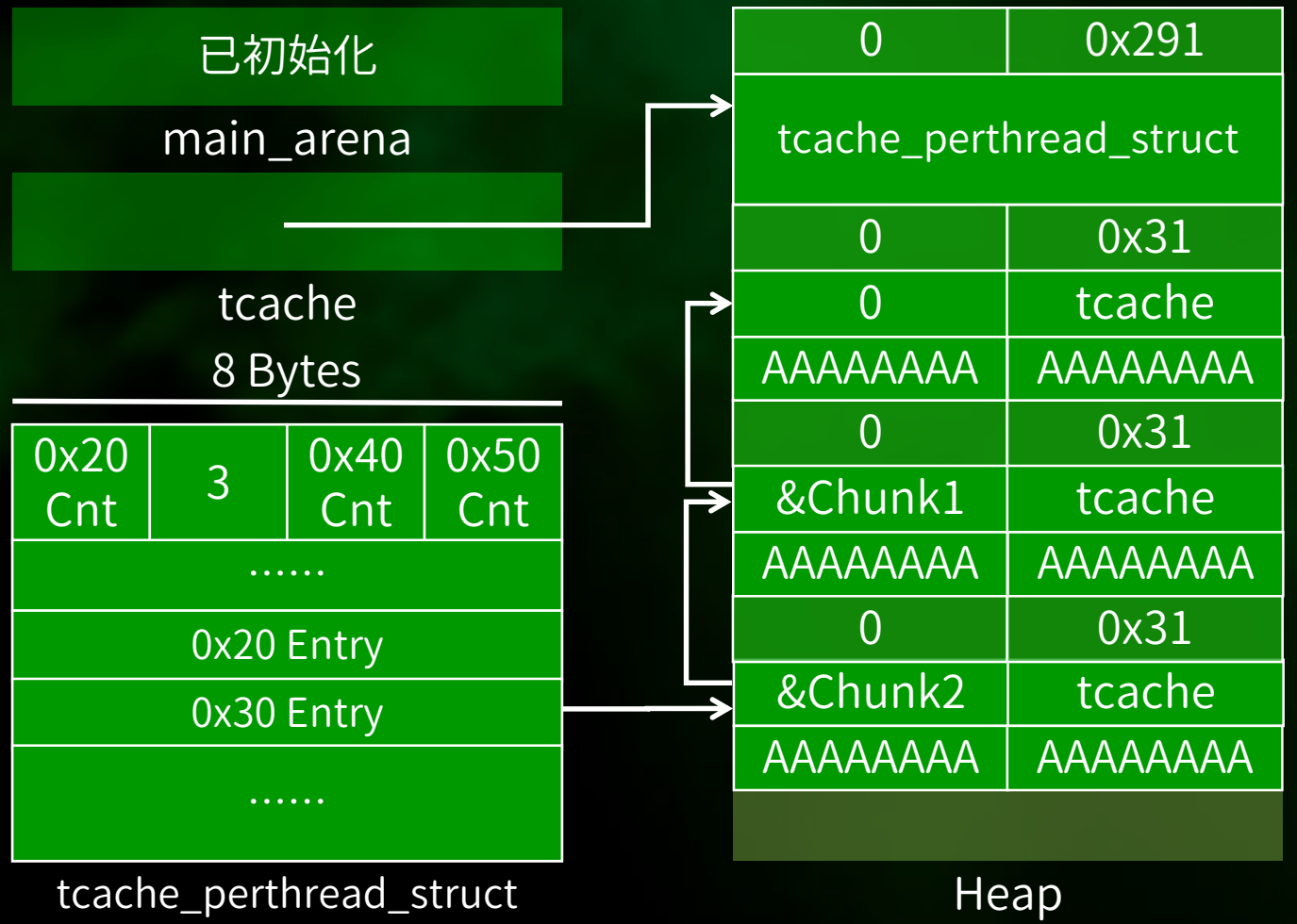
Tcache

- 再度分配 Chunk size 0x30

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
char *ptr4 = malloc(0x20);
```



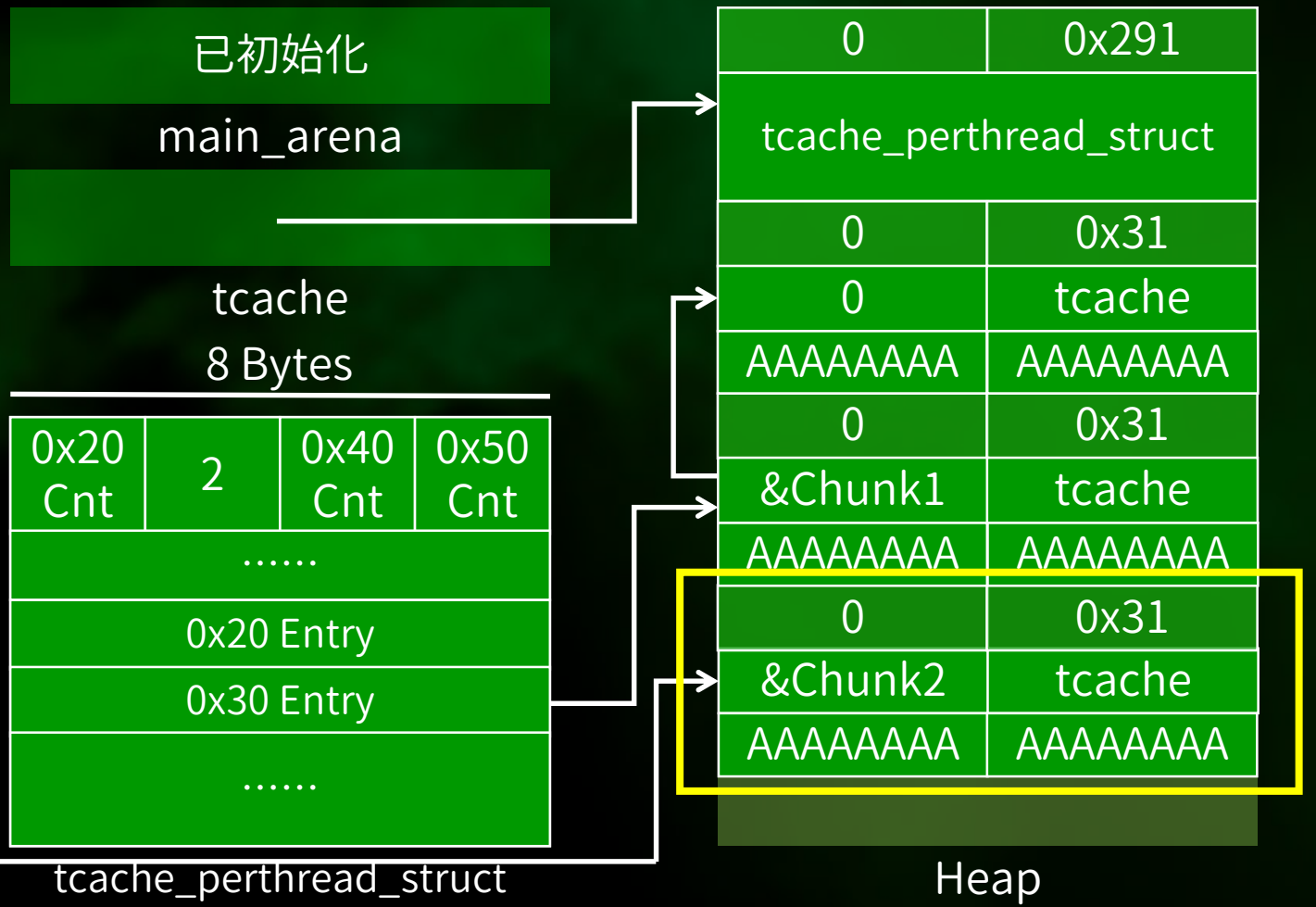
Tcache

- 從 Tcache 拿出 Size 剛好的 Chunk

```c
char *ptr1 = malloc(0x20);
char *ptr2 = malloc(0x20);
char *ptr3 = malloc(0x20);

memset(ptr1, 'A', 0x20);

free(ptr1);
free(ptr2);
free(ptr3);
char *ptr4 = malloc(0x20);
```



Tcache

- 小總結
- 跟 Fastbin 很像
  - LIFO
  - fd 指向下一塊 Free Chunk 的 Chunk Data
  - 不會改鄰近的下一塊 Chunk 的 P bit
  - 在 free 時, 如果下一塊是 Top Chunk, 並不會被合併進去
- Size range 比 Fastbin 大很多
- 每個 Tcache 最多只能裝 7 個 Chunk
- Fastbin 的 fd 是指到 Chunk Header; Tcache 的 fd 是指到 Chunk Data

Tcache in Libc 2.31

Source Code Reading

Tcache in Libc 2.31

Demo

Heap-Based Buffer Overflow

Heap-Based Buffer Overflow

- 在分配於 Heap 的變數上越界寫入
- 導致下一塊 Chunk 被改掉

Heap-Based Buffer Overflow

0	0x41
0	0x41
User\0\0\0\0	\0\0\0\0\0\0\0\2
0	0
0	0

Heap-Based Buffer Overflow

0	0x41
AAAAAAAA	AAAAAAAA
0	0x41
User\0\0\0\0	\0\0\0\0\0\0\0\2
0	0
0	0

Heap-Based Buffer Overflow

0	0x41
AAAAAAAA	AAAAAAAA
AAAAAAAA	AAAAAAAA
AAAAAAAA	AAAAAAAAA
AAAAAAAA	AAAAAAAA
Admin\0\0\0	\0\0\0\0\0\0\0\2
AAAAAAAA	AAAAAAAA
AAAAAAAA	AAAAAAAA

Heap-Based Buffer Overflow

Demo

UAF

UAF

- UAF 全名 Use-After-Free
- 把一個指標當作參數傳給 free, 會釋放指標指向的 Chunk
- 此時指標變成 dangling pointer
- 後續程式碼又從此指標寫入/讀取資料, 就是 UAF

UAF

- Free 掉 Chunk1

0	0x41
0	0x41

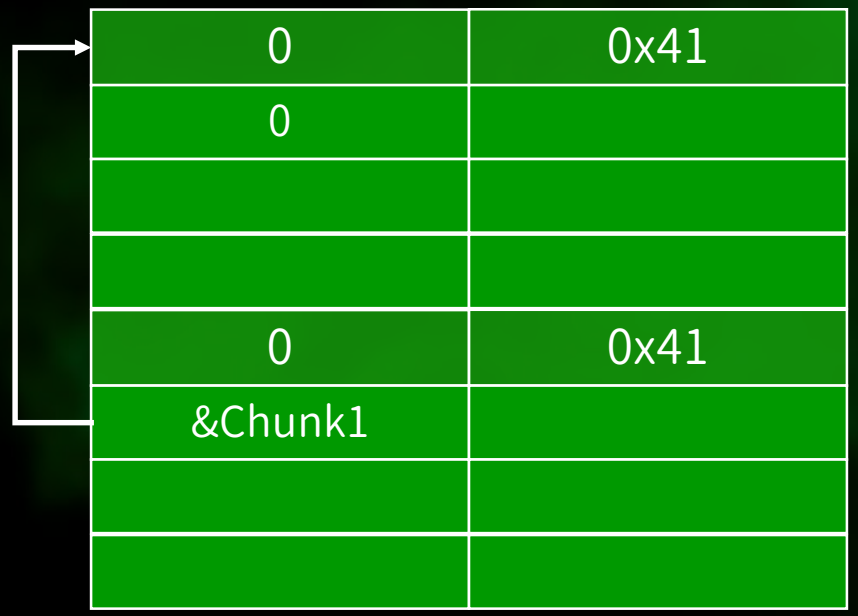
UAF

- Free 掉 Chunk2

0	0x41
0	
0	0x41

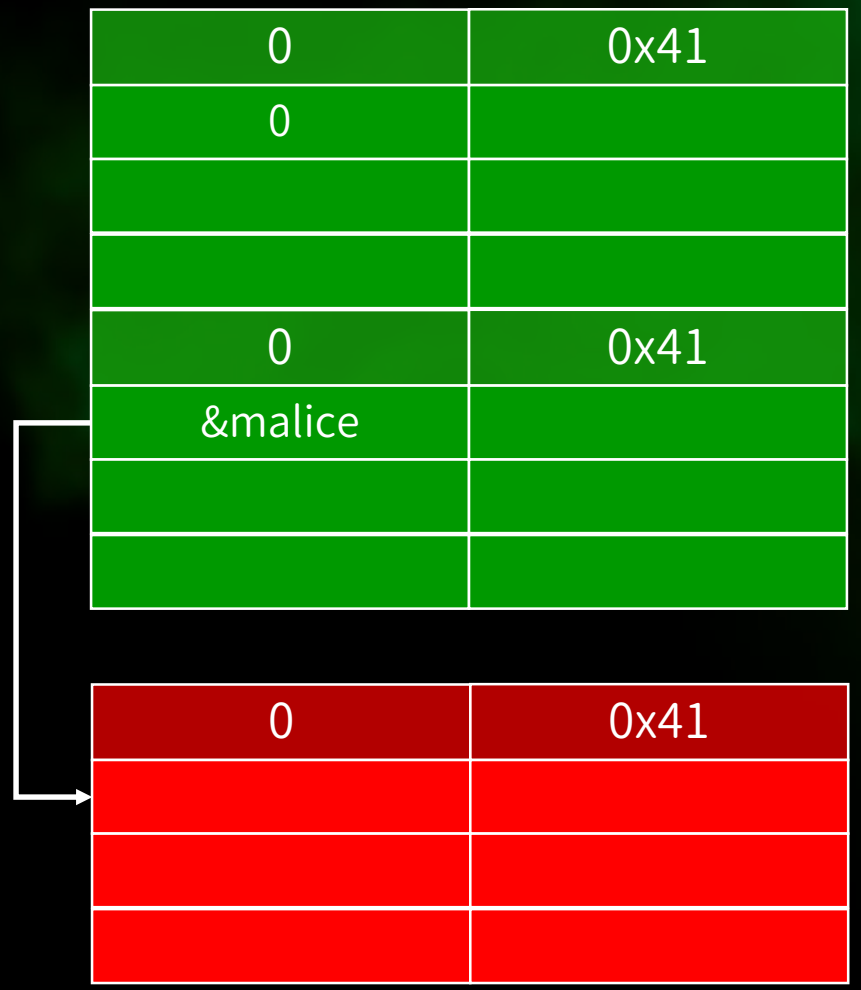
UAF

- UAF 讀取 Chunk2 8 Bytes, 就能取得 Heap Address



UAF

- UAF 寫入 Chunk2 8 Bytes, 就能控制 Tcache/Fastbin 鏈表



UAF Demo

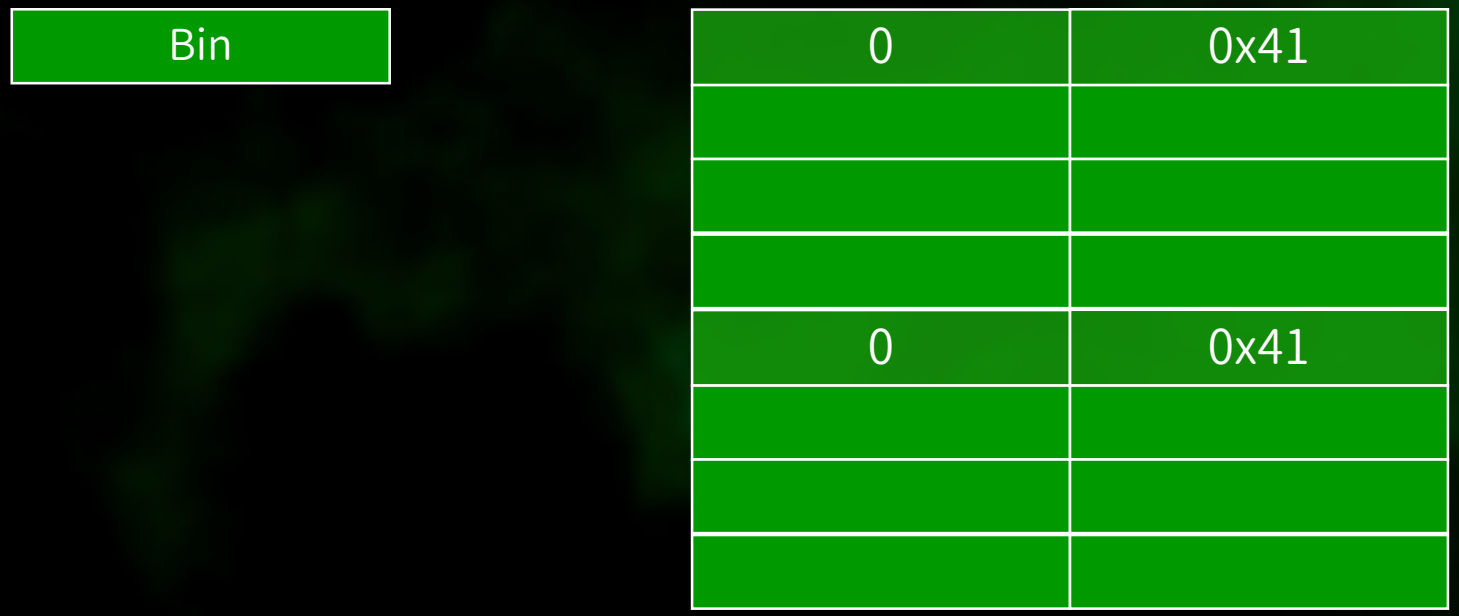
Double Free

Double Free

- Free 同一塊 Chunk 兩次
- 若成功, 則鏈表上會出現此 Chunk 兩次
- 再一次 malloc, 得到此 Chunk, 此時這個 Chunk 在鏈表中還是存在
- 薛丁格的 Chunk: 是 Alllocated Chunk, 也是 Free Chunk

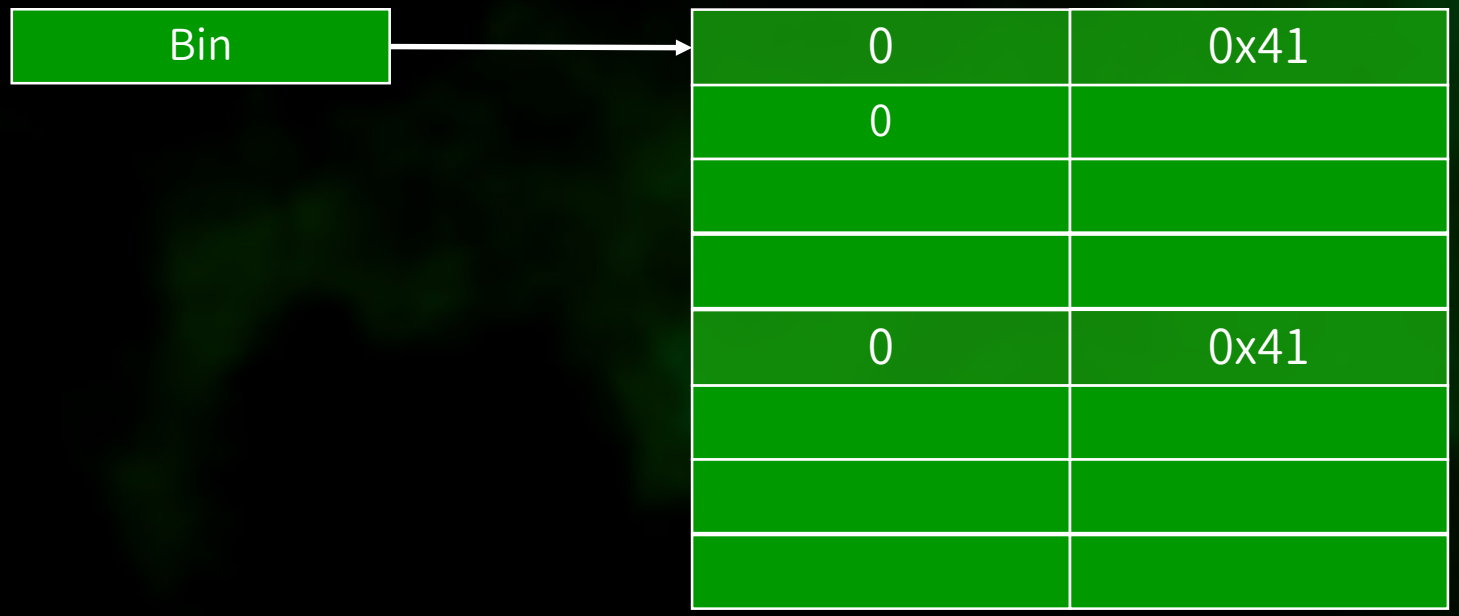
Double Free

- Free 掉 Chunk1



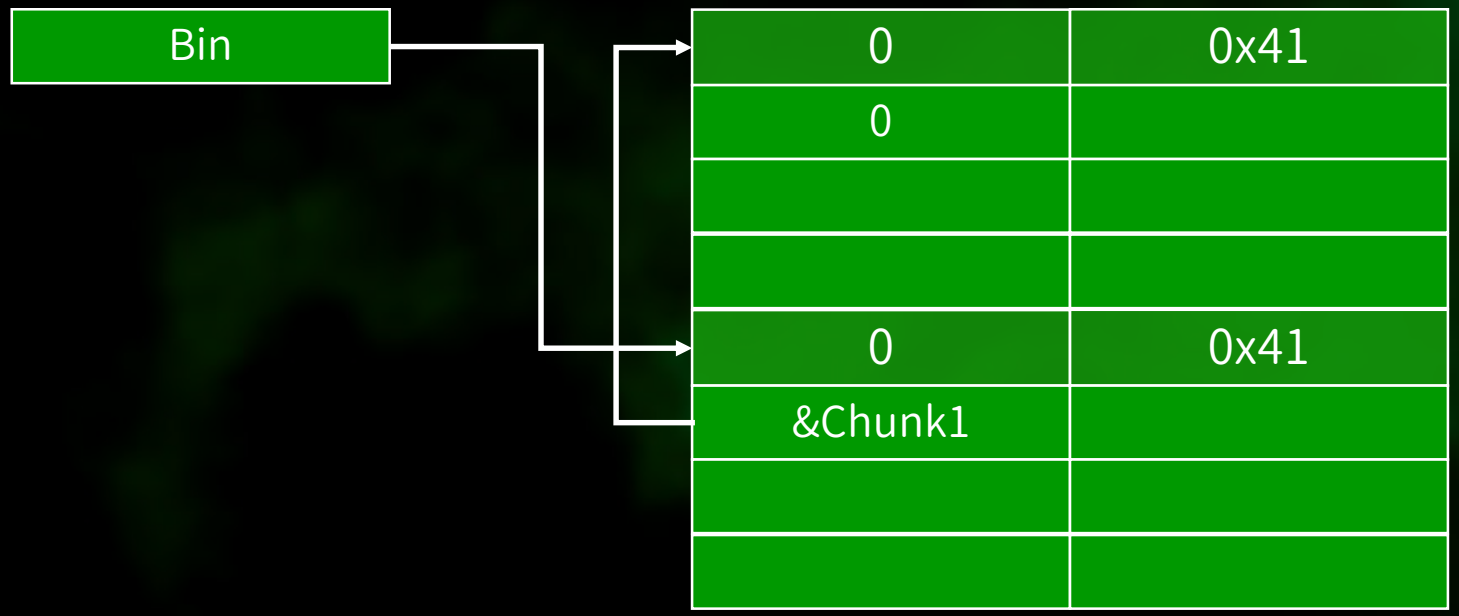
Double Free

- Free 掉 Chunk2



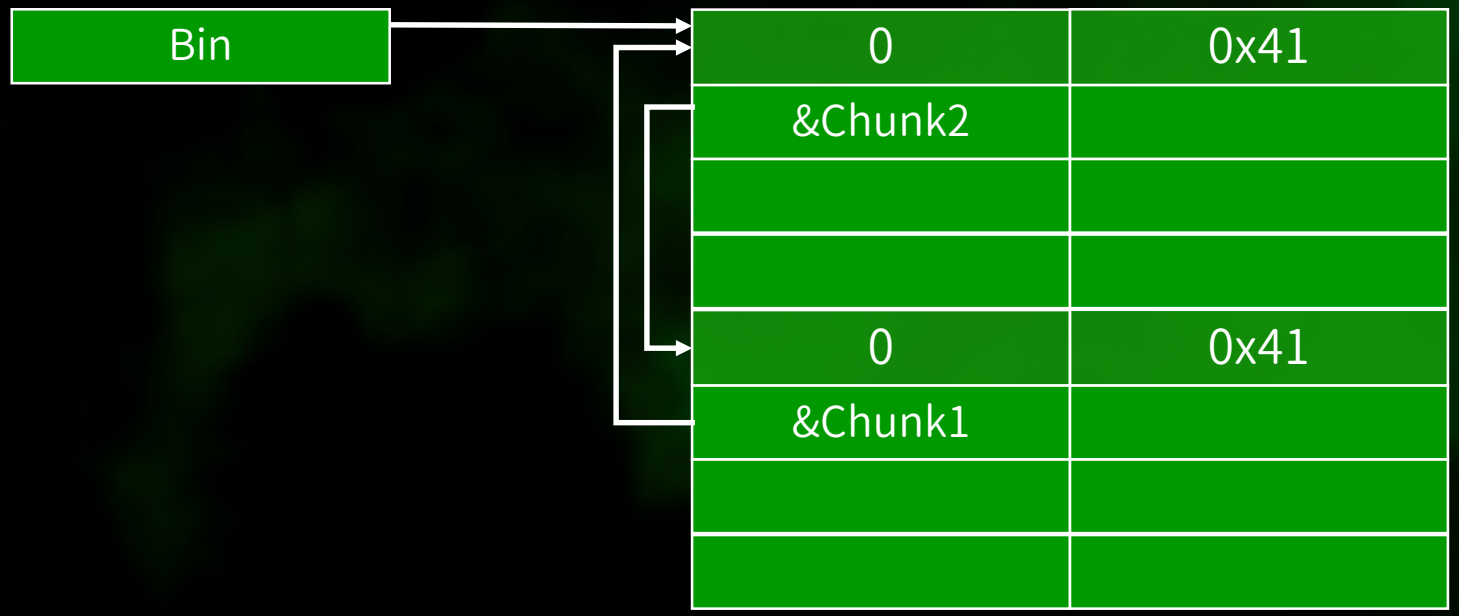
Double Free

- 再 Free 掉一次 Chunk1



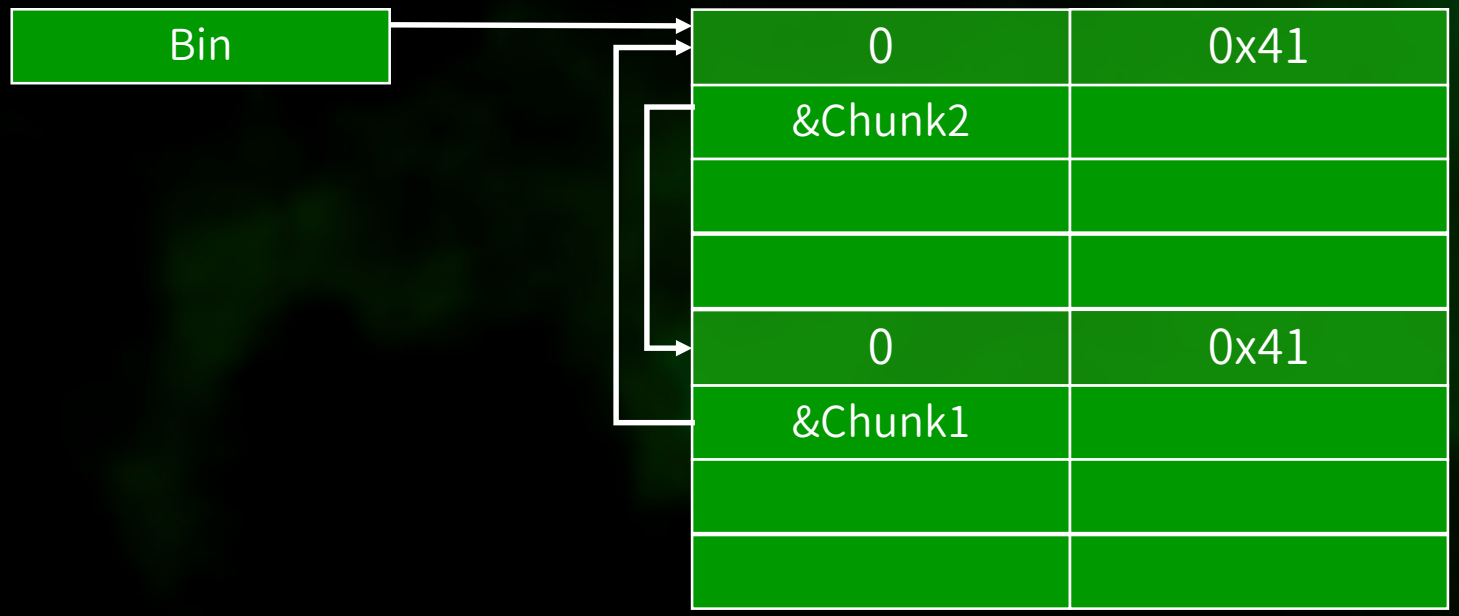
Double Free

- 觀察一下鏈表



Double Free

- 觀察一下鏈表



Double Free

Demo

Hooks

Hooks

- 在 malloc/free/calloc/realloc 進到主要分配演算法之前, 若有設定 hook function, 則會先執行 hook function
- 在寫 exploit 時, 是個很好的利用對象
- 朝 hook function 中寫入, 就能控制執行流程
- 寫入 One Gadget 就能得到 shell

Hooks Source Code Reading

Hooks Demo

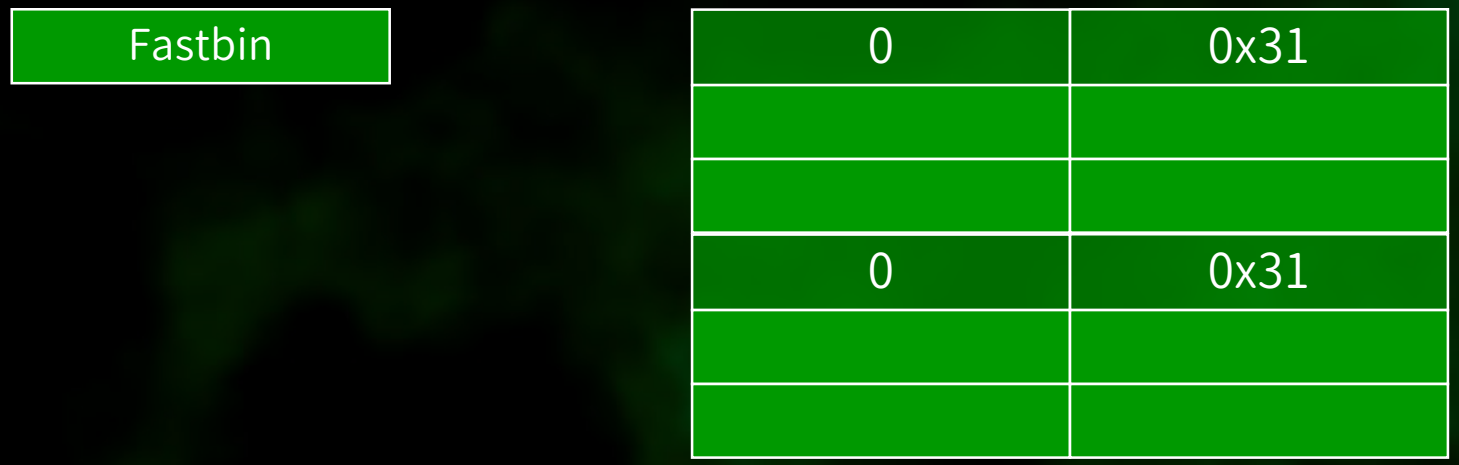
Fastbin dup

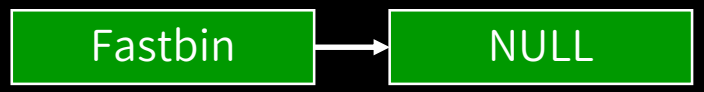
Fastbin Dup

- 用 Double Free 使 Chunk 在 Fastbin 中兩次
- Malloc 得到 Chunk 後, 將 fd 改寫成任意位址
- 再次 malloc, 取得位在剛剛改寫位址的 Chunk
- 效力形同任意寫入

Fastbin dup

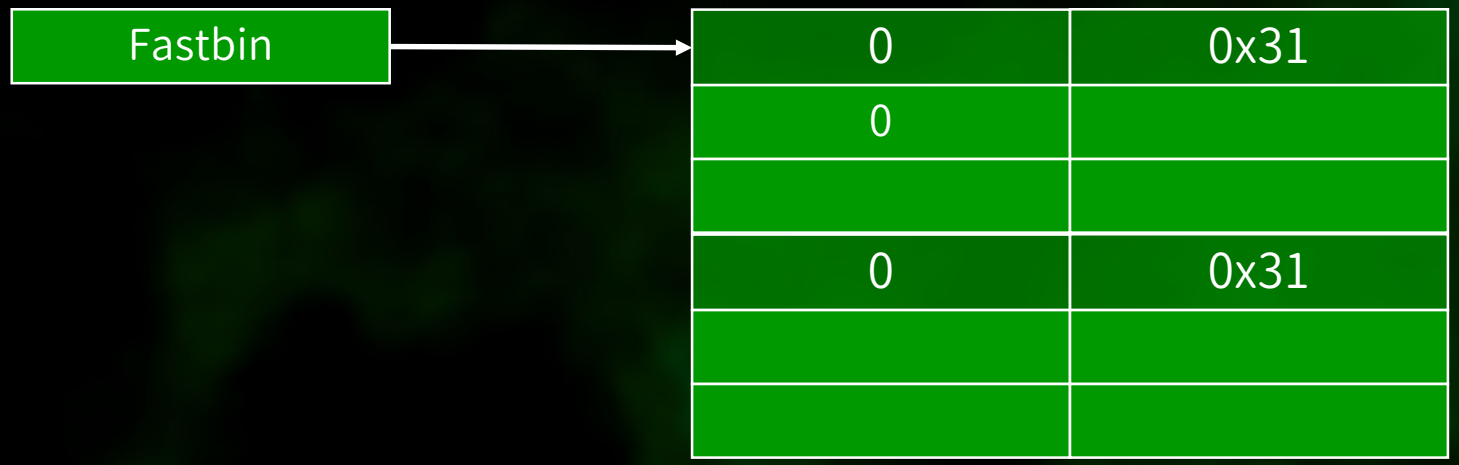
- Free 掉 Chunk1





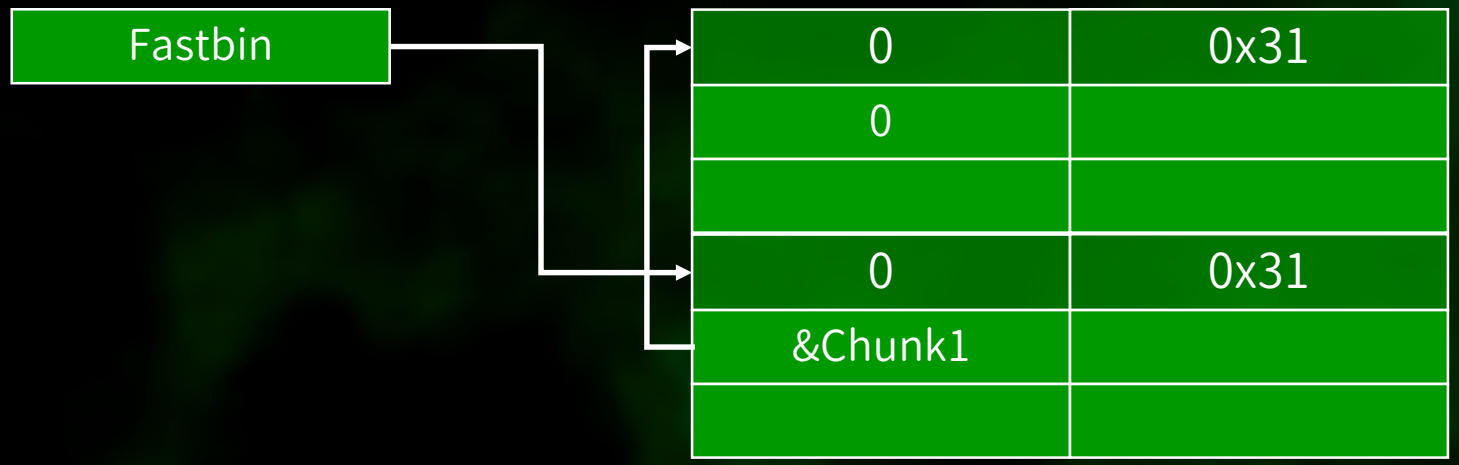
Fastbin dup

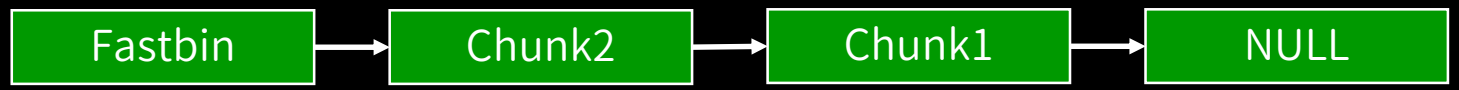
- Free 掉 Chunk2



Fastbin dup

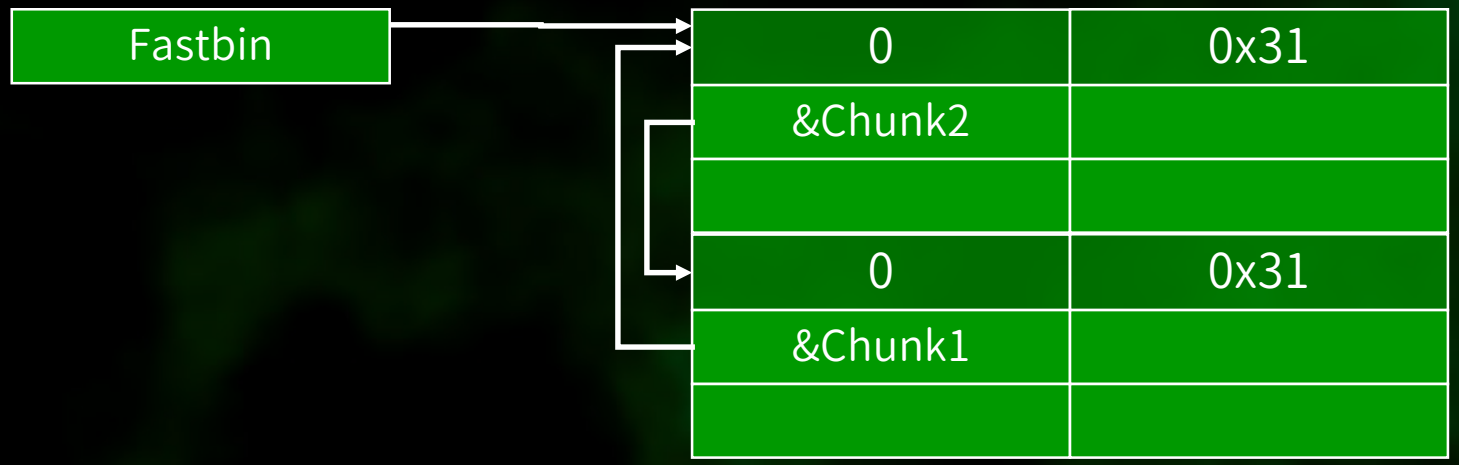
- 再次 free 掉 Chunk1





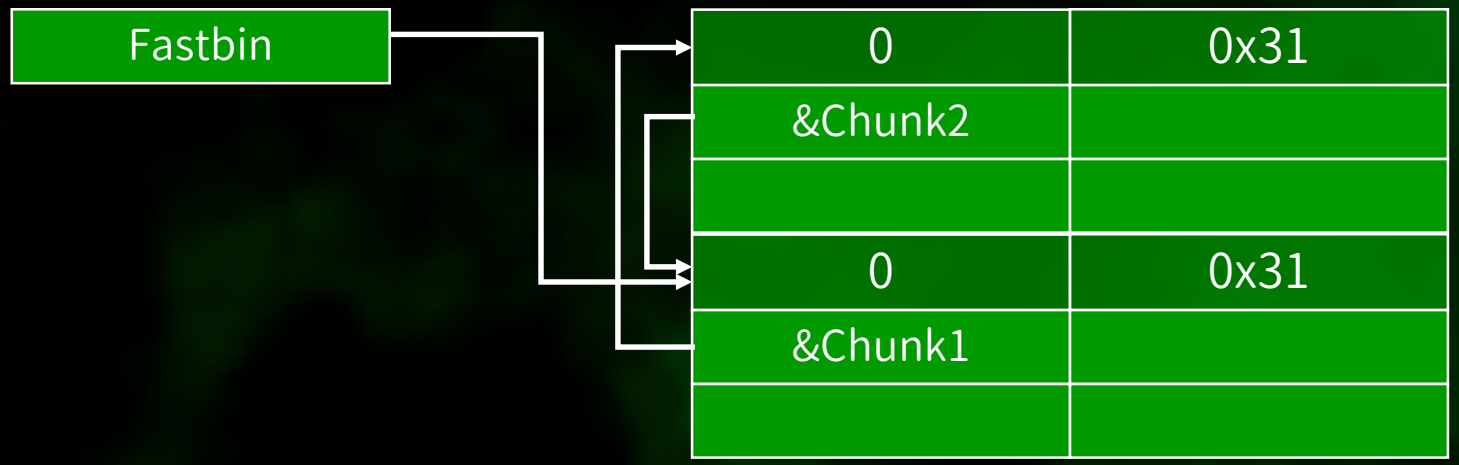
Fastbin dup

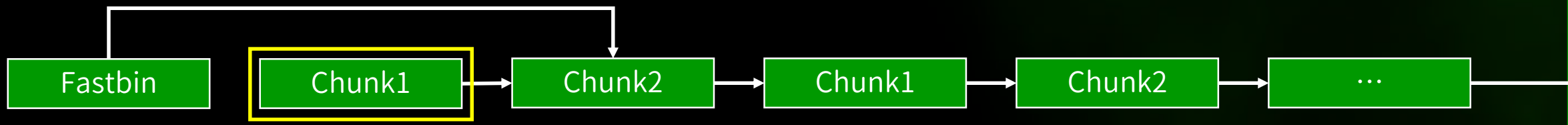
- 觀察一下鏈表



Fastbin dup

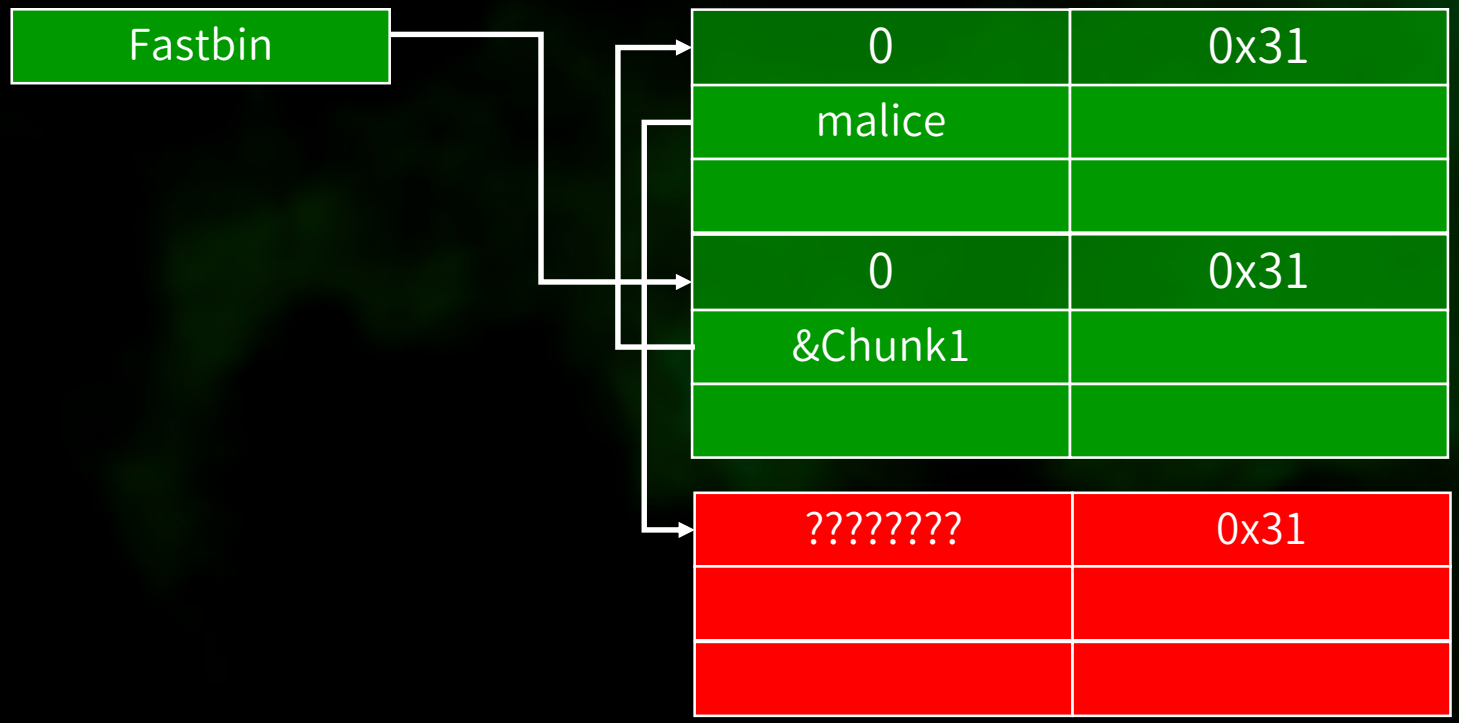
- Malloc 過後取得 Chunk1, Chunk2 遮補

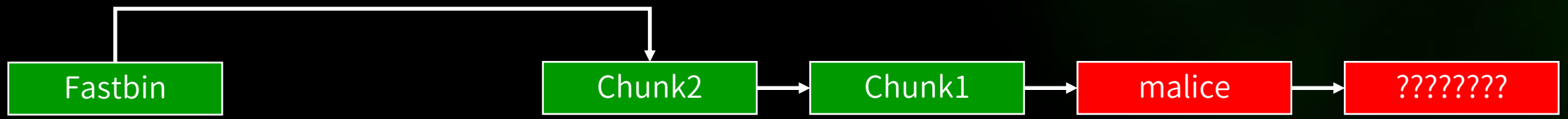




Fastbin dup

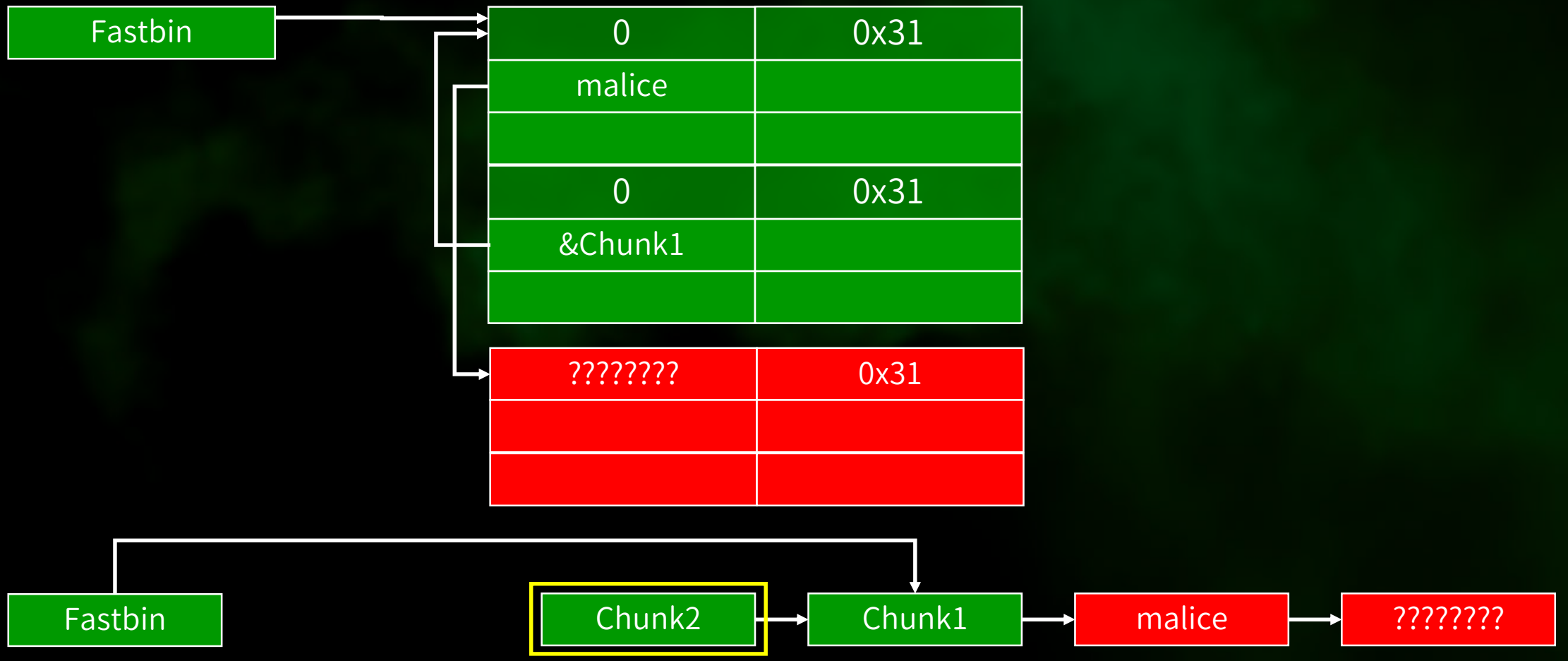
- 取得 Chunk1 後, 向其寫入 malice





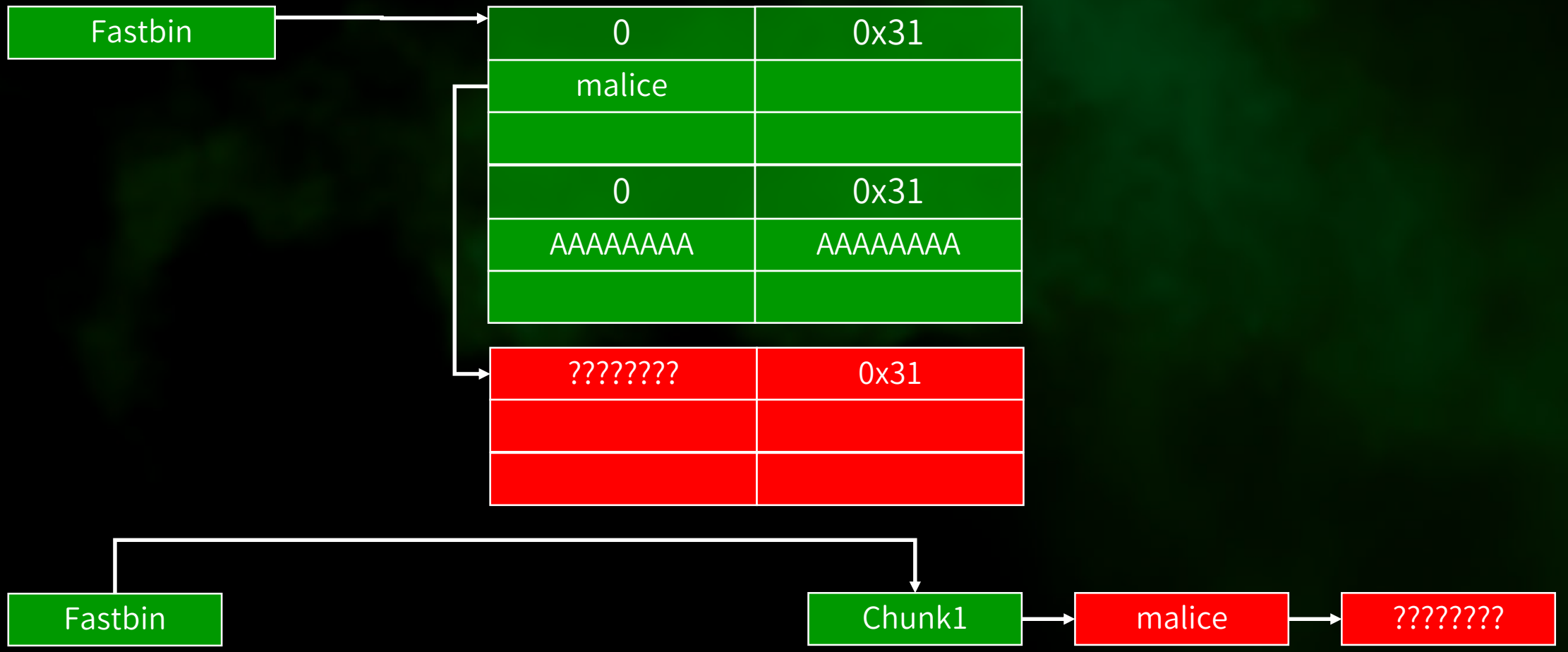
Fastbin dup

- 再次 malloc 過後取得 Chunk2, Chunk1 遮補



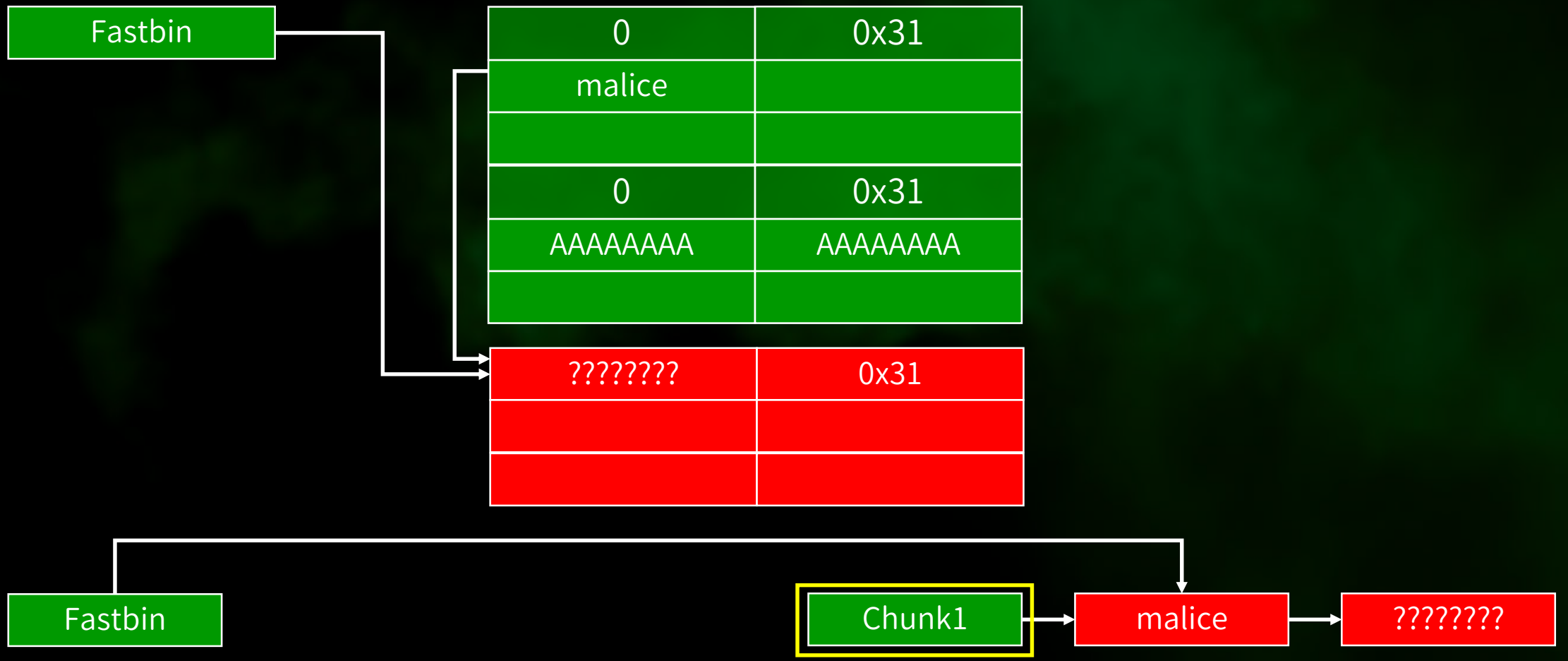
Fastbin dup

- 取得 Chunk2 後, 向其寫入什麼無所謂



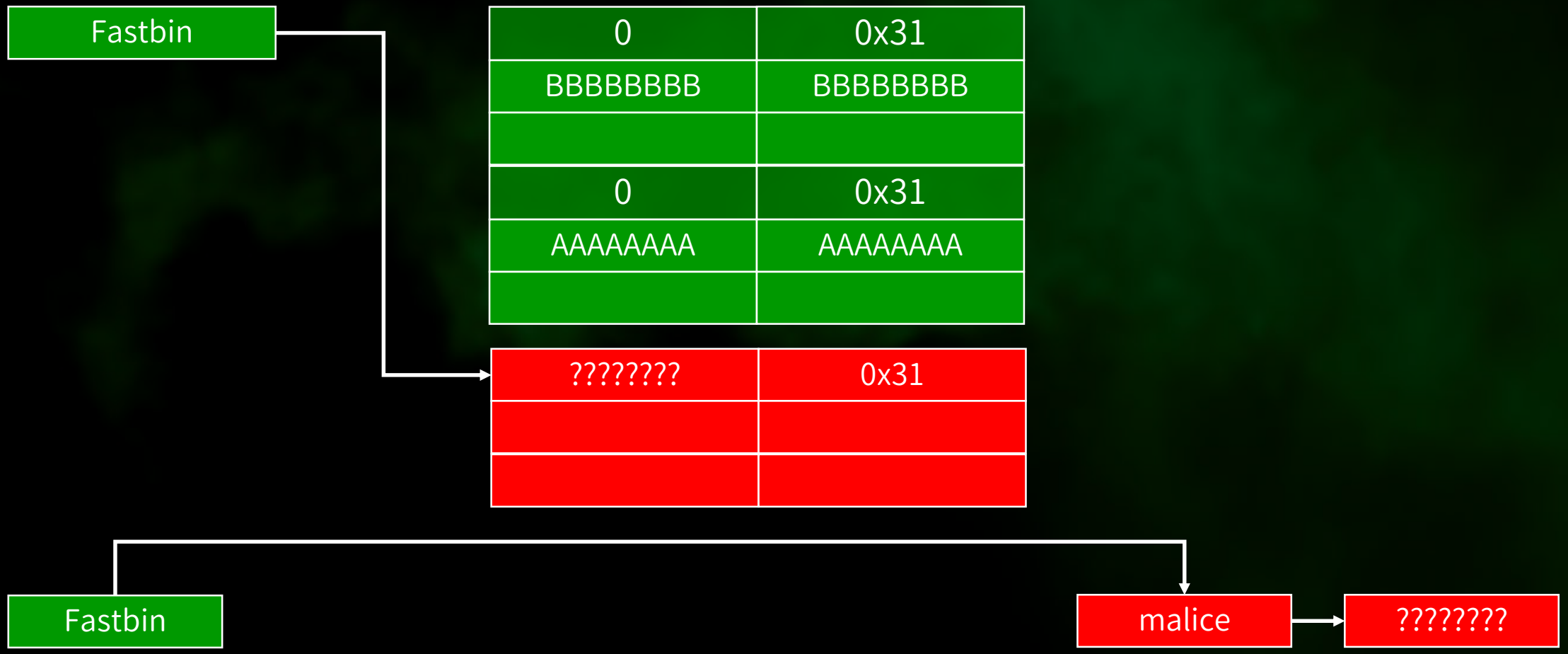
Fastbin dup

- 再次 malloc 過後取得 Chunk1, malice 遮補



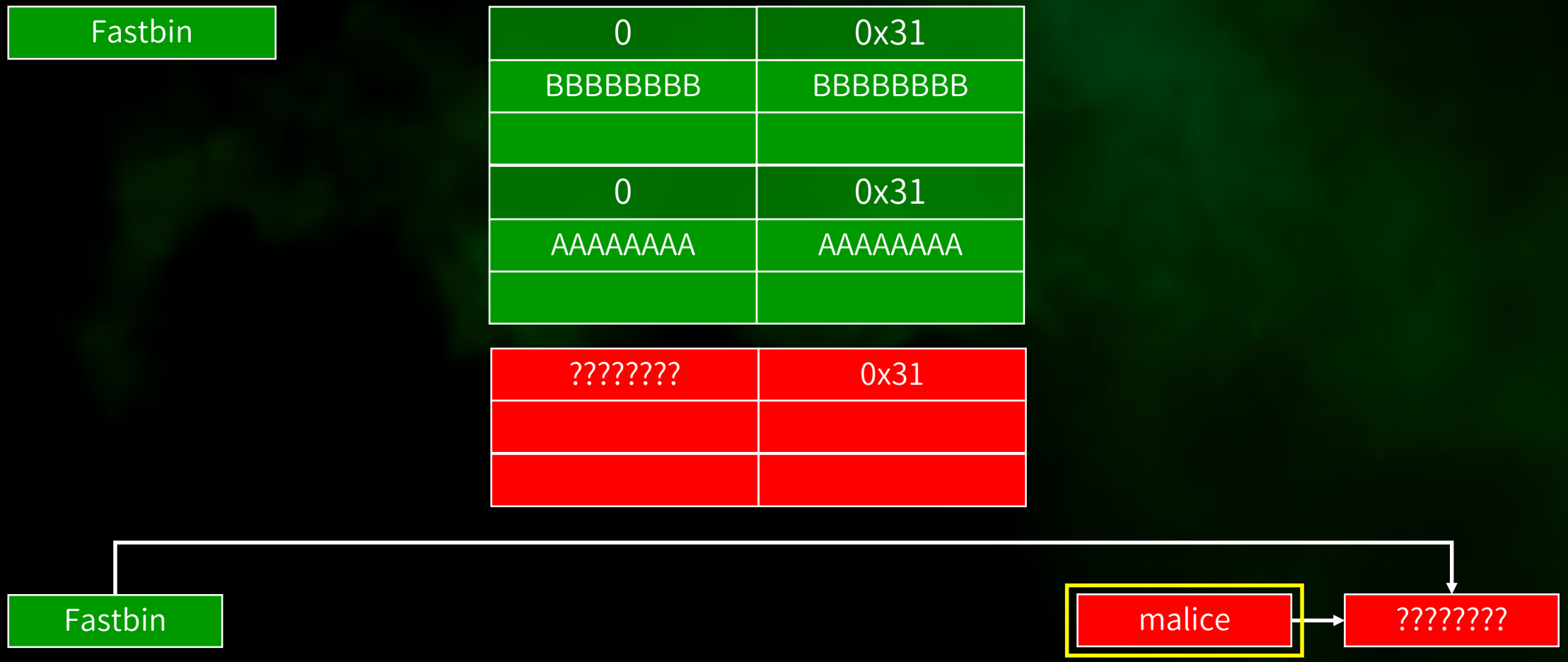
Fastbin dup

- 這次取得 Chunk1 後, 向其寫入什麼無所謂



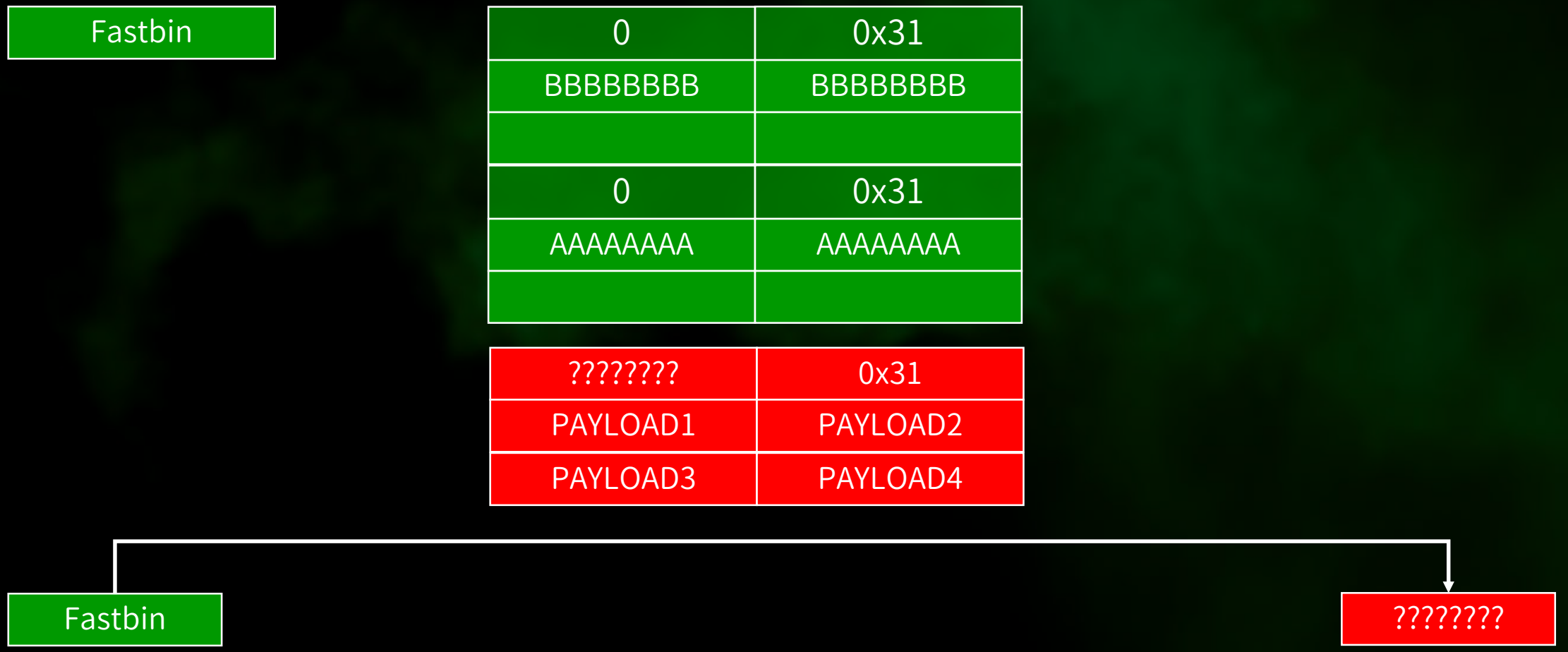
Fastbin dup

- 再次 malloc 過後取得 malice, ???????? 遮補



Fastbin dup

- 這次取得位於 malice 的 Chunk 後, 向其寫入想寫的值



Fastbin Dup

- 可以將目標地址設定為 Return Address 或 Hook Function 這類可控執行流程的位址
- 注意 fastbin fd 是指向 Chunk Header
- Malicious Chunk 的 Size 須符合 Fastbin 所屬的 Size
- 在 libc 2.26 後, Fast Chunk 會先進 Tcache

Fastbin Dup

Source Code Reading

Fastbin Dup

- Libc 2.23
- _int_free
- 有幾道安全檢查
- 檢查鄰近的下一塊 Chunk size 是否合理
- 0x10 < Size < av->system_mem

```
if (__builtin_expect (chunk_at_offset (p, size)->size <= 2 * SIZE_SZ, 0)
    || __builtin_expect (chunksize (chunk_at_offset (p, size))
                         >= av->system_mem, 0))
  {
    /* We might not have a lock at this point and concurrent modifications
       of system_mem might have let to a false positive.  Redo the test
       after getting the lock.  */
    if (have_lock
        || ({ assert (locked == 0);
              mutex_lock(&av->mutex);
              locked = 1;
              chunk_at_offset (p, size)->size <= 2 * SIZE_SZ
                || chunksize (chunk_at_offset (p, size)) >= av->system_mem;
            }))
      {
        errstr = "free(): invalid next size (fast)";
        goto errout;
      }
    if (! have_lock)
      {
        (void)mutex_unlock(&av->mutex);
        locked = 0;
      }
  }
```

Fastbin Dup

- Libc 2.23
- _int_free
- 有幾道安全檢查
- 鏈上的第一個 Free Chunk 若和目前要 free 的 Chunk 一樣, 則為 Double Free
- 繞過方式: 再 free 同塊 Chunk 之前, 先 free 其他 Chunk, 這就是示意圖中 Chunk2 存在的意義

```
free_perturb (chunk2mem(p), size - 2 * SIZE_SZ);

set_fastchunks(av);
unsigned int idx = fastbin_index(size);
fb = &fastbin (av, idx);

/* Atomically link P to its fastbin: P->FD = *FB; *FB = P;  */
mchunkptr old = *fb, old2;
unsigned int old_idx = ~0u;
do
  {
    /* Check that the top of the bin is not the record we are going to add
       (i.e., double free).  */
    if (__builtin_expect (old == p, 0))
      {
        errstr = "double free or corruption (fasttop)";
        goto errout;
      }
    /* Check that size of fastbin chunk at the top is the same as
       size of the chunk that we are adding.  We can dereference OLD
       only if we have the lock, otherwise it might have already been
       deallocated.  See use of OLD_IDX below for the actual check.  */
    if (have_lock && old != NULL)
      old_idx = fastbin_index(chunksize(old));
    p->fd = old2 = old;
  }
while ((old = catomic_compare_and_exchange_val_rel (fb, p, old2)) != old2);

if (have_lock && old != NULL && __builtin_expect (old_idx != idx, 0))
  {
    errstr = "invalid fastbin entry (free)";
    goto errout;
  }
```

Fastbin Dup

- Libc 2.23
- _int_malloc
- 檢查拿出的 Chunk Size 是否和 Fastbin 所屬的 Size 相同

```
if ((unsigned long) (nb) <= (unsigned long) (get_max_fast ()))
  {
    idx = fastbin_index (nb);
    mfastbinptr *fb = &fastbin (av, idx);
    mchunkptr pp = *fb;
    do
      {
        victim = pp;
        if (victim == NULL)
          break;
      }
    while ((pp = catomic_compare_and_exchange_val_acq (fb, victim->fd, victim))
           != victim);
    if (victim != 0)
      {
        if (__builtin_expect (fastbin_index (chunksize (victim)) != idx, 0))
          {
            errstr = "malloc(): memory corruption (fast)";
          errout:
            malloc_printerr (check_action, errstr, chunk2mem (victim), av);
            return NULL;
          }
        check_remalloced_chunk (av, victim, nb);
        void *p = chunk2mem (victim);
        alloc_perturb (p, bytes);
        return p;
      }
  }
```

Fastbin Dup

- Libc 2.27
- _int_free
- Fastbin 的部分基本相同
- Tcache 裝滿 7 個 Fast Chunk 後, 才會往 Fastbin 放

```
if (__builtin_expect (chunksize_nomask (chunk_at_offset (p, size))
                      <= 2 * SIZE_SZ, 0)
    || __builtin_expect (chunksize (chunk_at_offset (p, size))
                         >= av->system_mem, 0))
```

```
if (SINGLE_THREAD_P)
  {
    /* Check that the top of the bin is not the record we are going to
       add (i.e., double free).  */
    if (__builtin_expect (old == p, 0))
      malloc_printerr ("double free or corruption (fasttop)");
    p->fd = old;
    *fb = p;
  }
```

Fastbin Dup

- Libc 2.27
- _int_malloc
- 一樣會檢查拿出的 Chunk 是否屬於此 Fastbin
- 須注意的是, 若 Tcache 沒放滿, 則會把 Fastbin 中的 Chunk 放入 Tcache

```
          if (__glibc_likely (victim != NULL))
            {
              size_t victim_idx = fastbin_index (chunksize (victim));
              if (__builtin_expect (victim_idx != idx, 0))
                malloc_printerr ("malloc(): memory corruption (fast)");
              check_remalloced_chunk (av, victim, nb);
#if USE_TCACHE
              /* While we're here, if we see other chunks of the same size,
                 stash them in the tcache.  */
              size_t tc_idx = csize2tidx (nb);
              if (tcache && tc_idx < mp_.tcache_bins)
                {
                  mchunkptr tc_victim;

                  /* While bin not empty and tcache not full, copy chunks.  */
                  while (tcache->counts[tc_idx] < mp_.tcache_count
                         && (tc_victim = *fb) != NULL)
                    {
                      if (SINGLE_THREAD_P)
                        *fb = tc_victim->fd;
                      else
                        {
                          REMOVE_FB (fb, pp, tc_victim);
                          if (__glibc_unlikely (tc_victim == NULL))
                            break;
                        }
                      tcache_put (tc_victim, tc_idx);
                    }
                }
#endif
              void *p = chunk2mem (victim);
              alloc_perturb (p, bytes);
              return p;
```

Fastbin Dup

- Libc 2.31 則和 Libc 2.27 差不多
- 統整以上, Fastbin Dup 在以上版本皆能打
- 在支援 Tcache 的 Libc 版本需考慮 Tcache 要放滿
- Q: 在 Tcache 放滿的情況下,
  malloc 不會拿 fastbin 而是拿 Tcache, 那 Fastbin Dup 怎打?
- A: 利用 calloc 不會拿 Tcache 的特性繞過

Fastbin dup

Demo

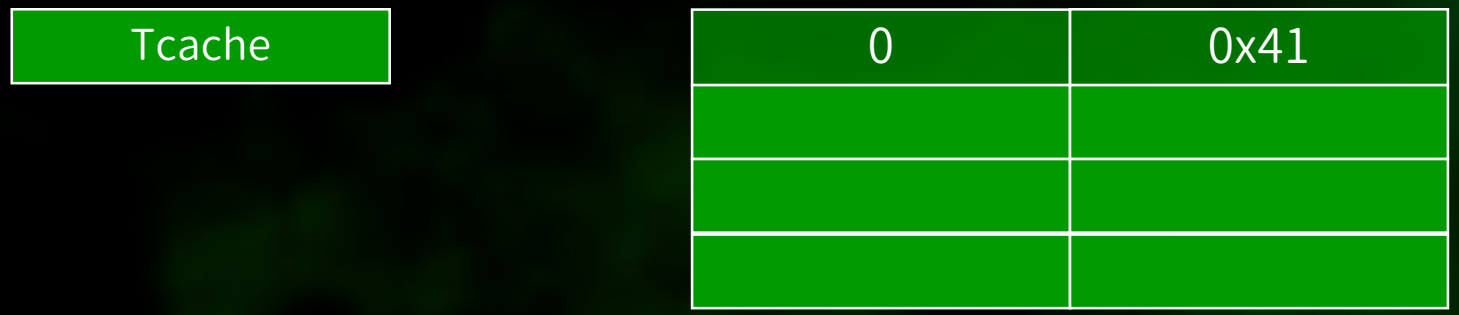
Tcache Dup

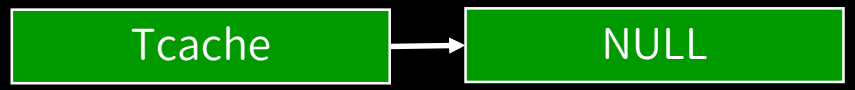
Tcache Dup

- 和 Fastbin Dup 很像
  - 用 Double Free 使 Chunk 在 Tcache 中兩次
  - Malloc 得到 Chunk 後, 將 fd 改寫成任意位址
  - 再次 malloc, 取得位在剛剛改寫位址的 Chunk
  - 效力形同任意寫入
- 在 libc 2.26 ~ libc 2.28 中可使用, 之後的版本有加安全檢查
- libc 2.27 中, 並無檢查鏈上第一個 Chunk 是否就是要被 free 的 Chunk, 因此可以直接 Free 自己
- 利用上更簡單

Tcache dup

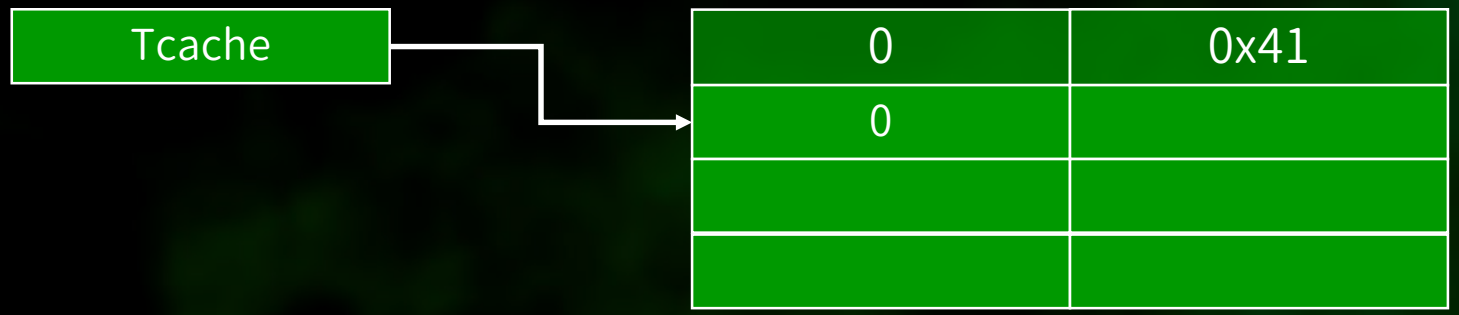
- Free 掉 Chunk1





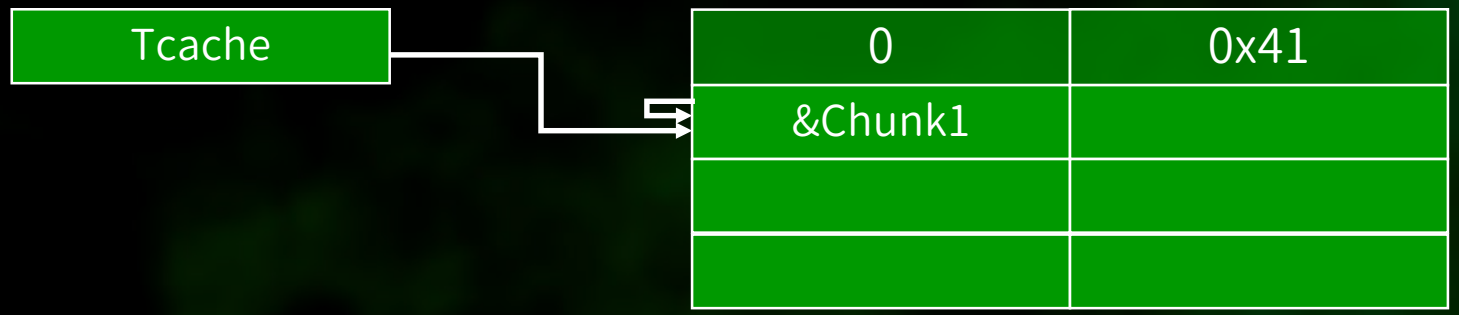
Tcache dup

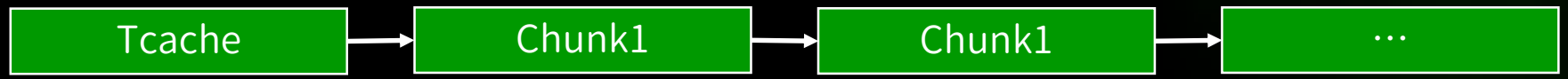
- 直接再 free 掉一次 Chunk1



Tcache dup

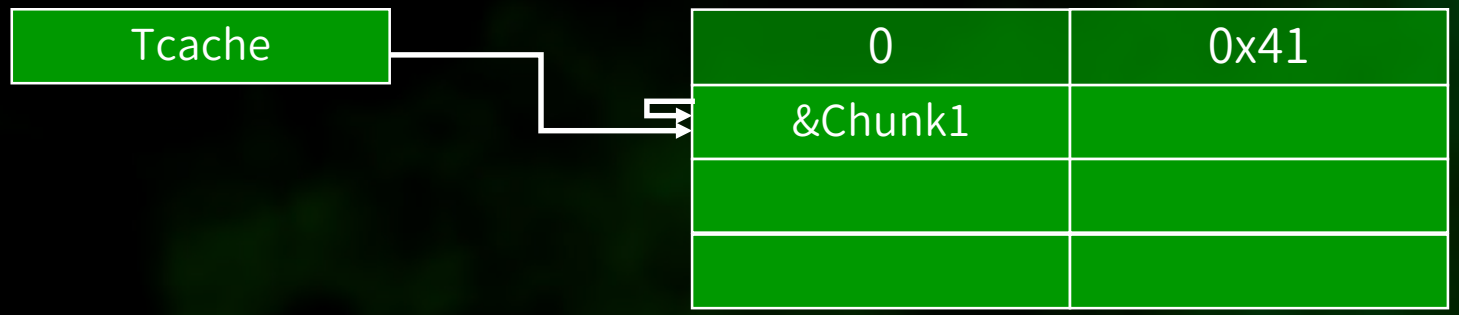
- 觀察一下鏈表

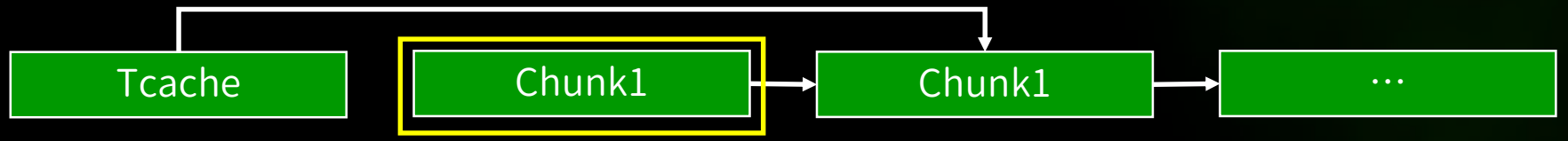




Tcache dup

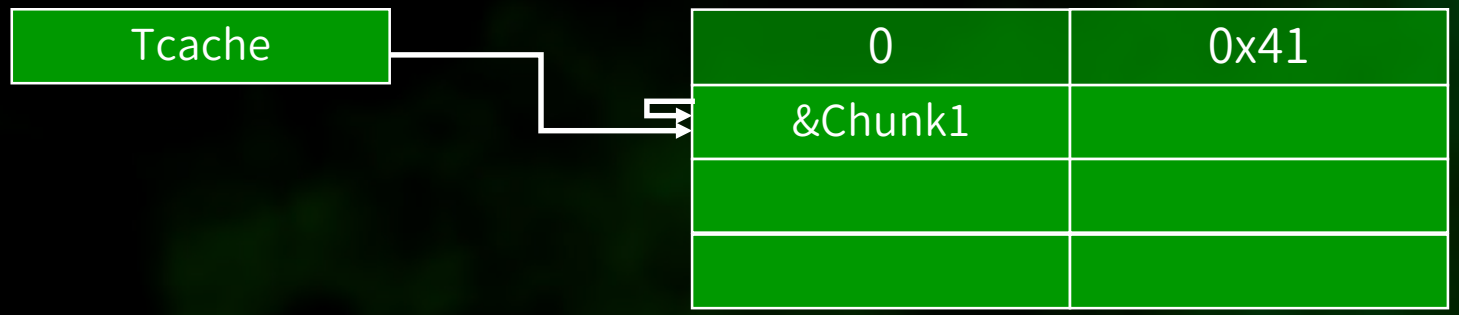
- Malloc 取得 Chunk1, 從鏈上取走 Chunk1, Chunk1 遞補

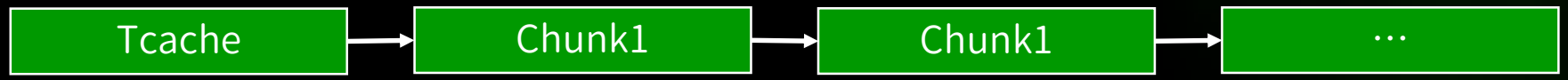




Tcache dup

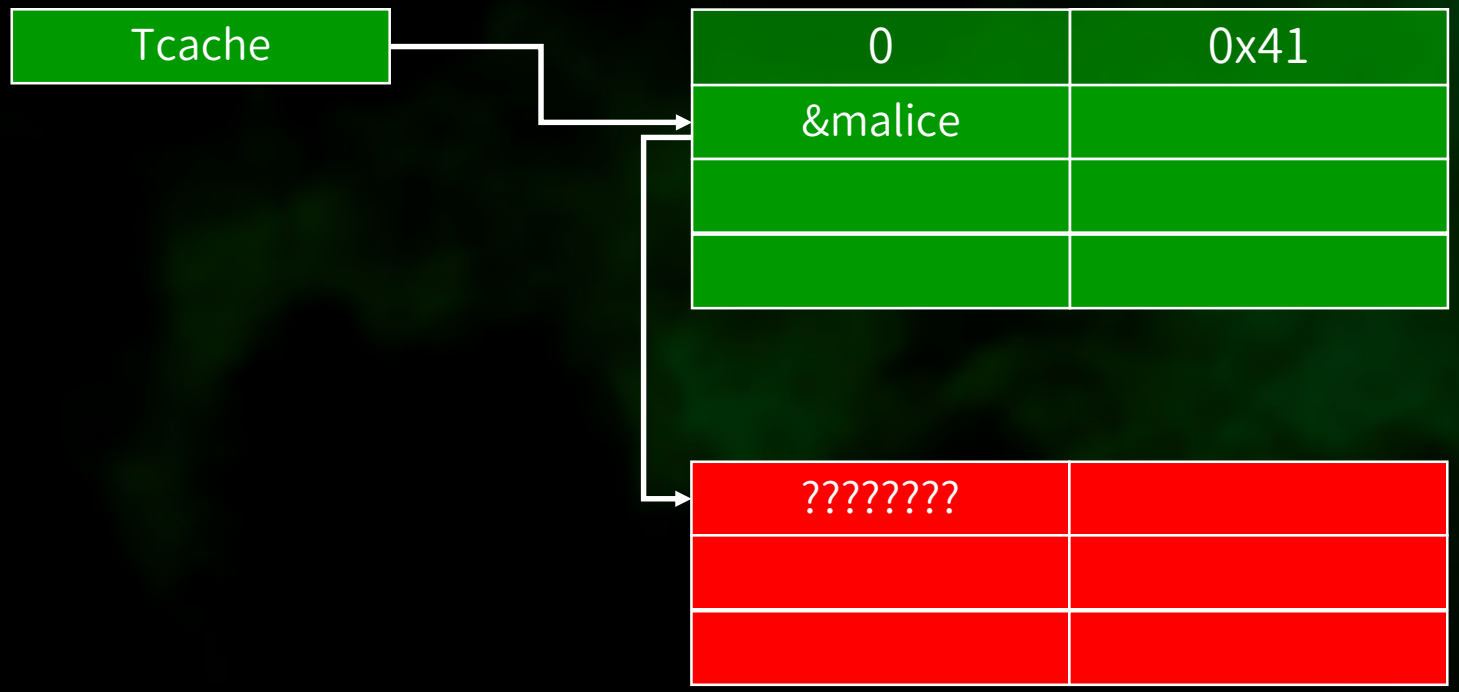
- 朝 malloc 回傳的 ptr 寫入 malice

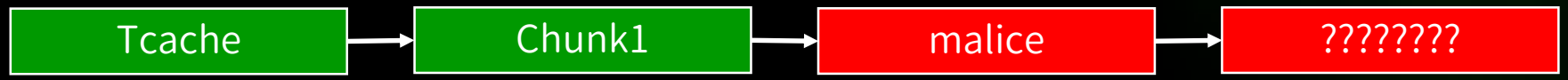




Tcache dup

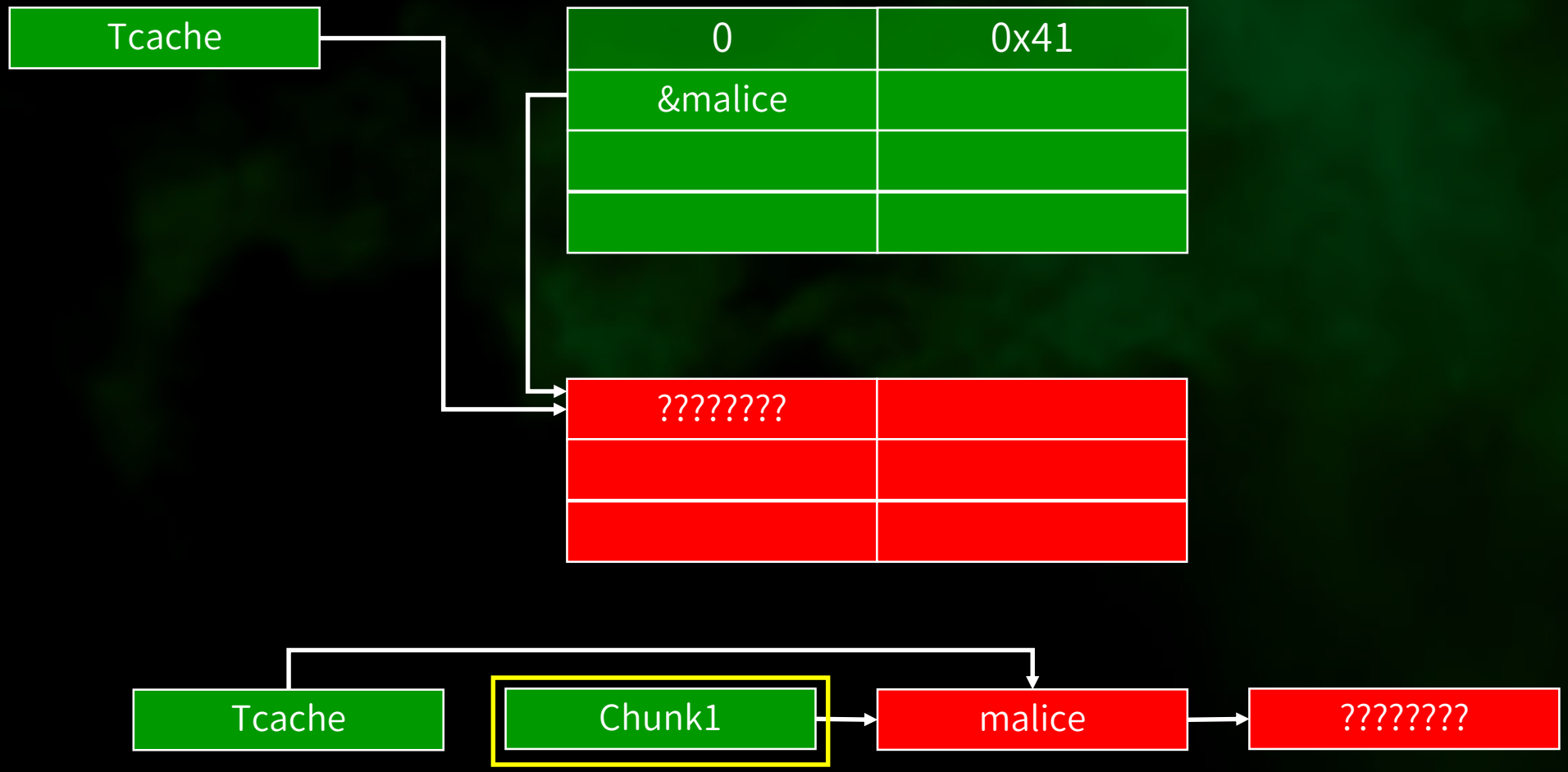
- 觀察一下鏈表





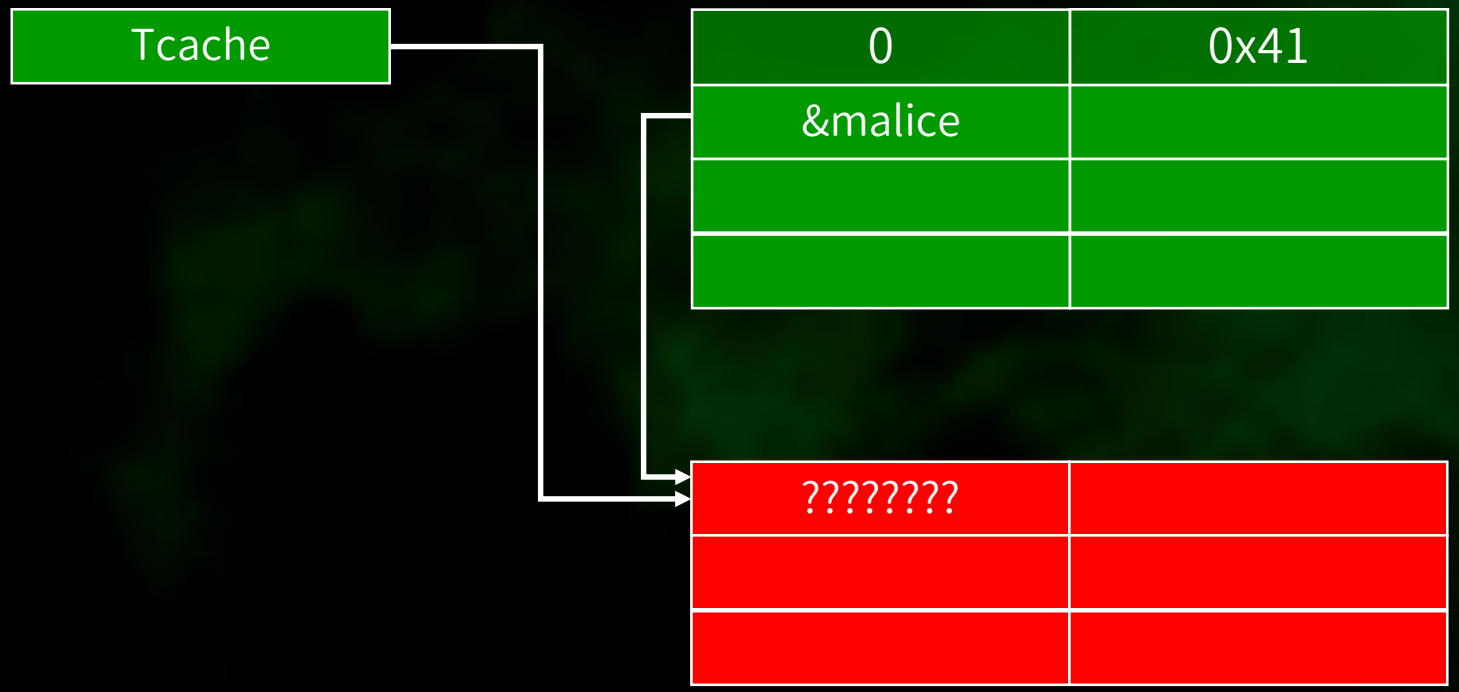
Tcache dup

- 再次 malloc,從鏈上取走 Chunk1, malice 遞補



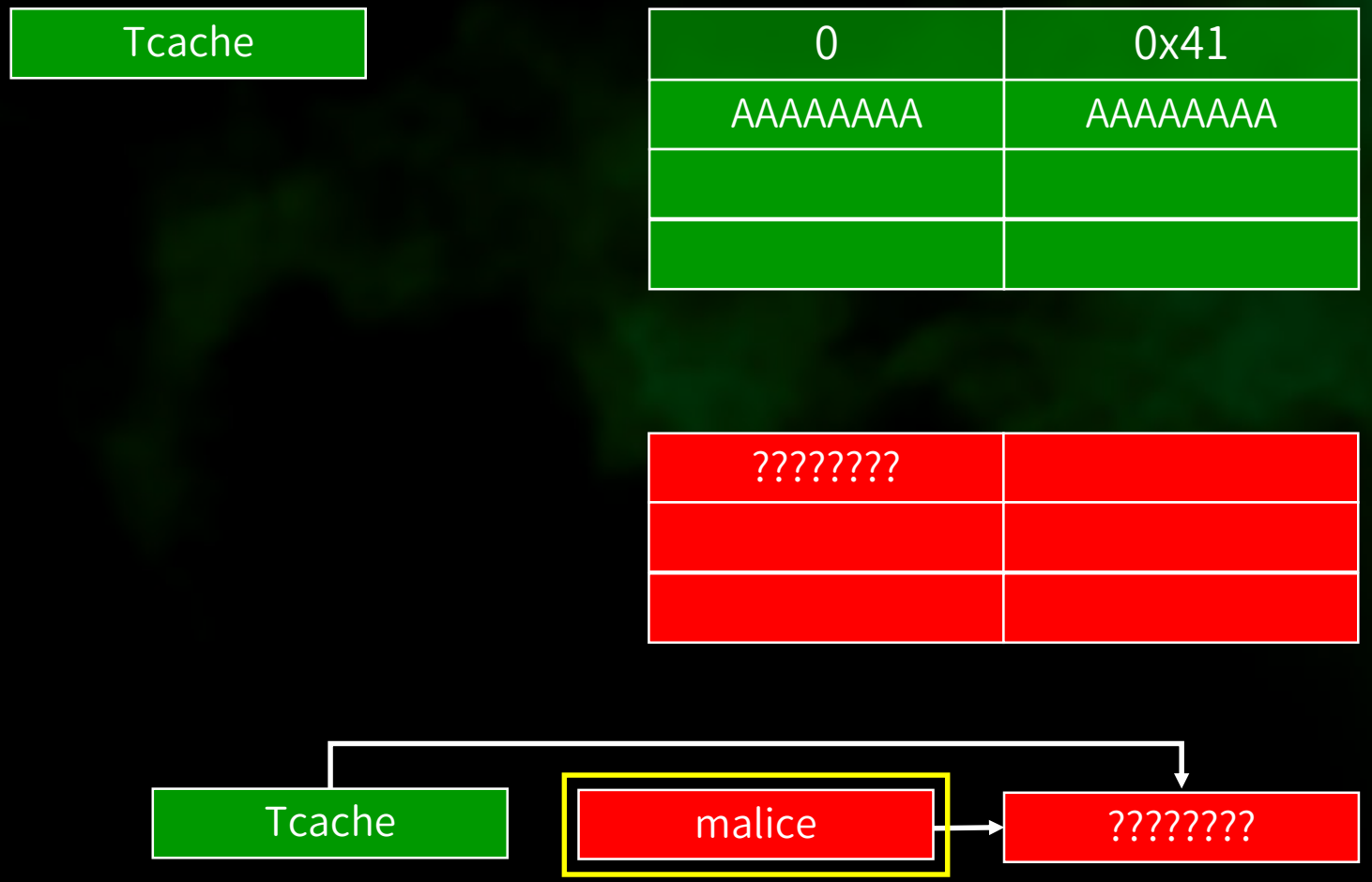
Tcache dup

- 此次朝 Chunk1 寫入什麼無所謂



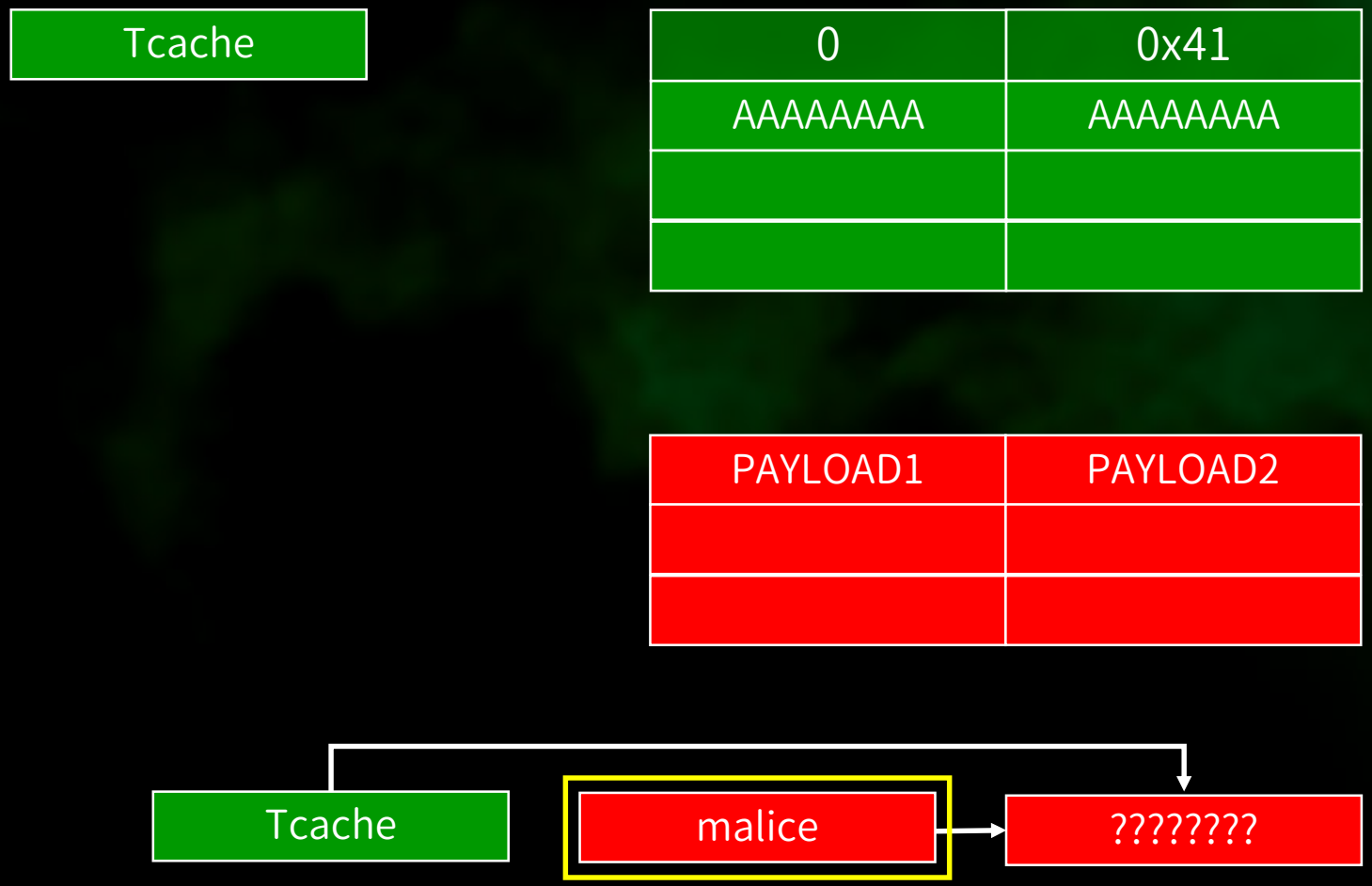
Tcache dup

- 重點是下次 malloc 回傳的 ptr 就是指向 malice



Tcache dup

- 這次寫入資料就是往 malice 寫入



Tcache Dup

- 在 libc 2.29 後加上了安全檢查
- 相對於 Fastbin fd 是指向 Chunk Header, Tcache fd 是直接指向 Chunk Data, 好用

Tcache Dup

Source Code Reading

Tcache Dup

- Libc 2.27

```
tcache_put (mchunkptr chunk, size_t tc_idx)
{
  tcache_entry *e = (tcache_entry *) chunk2mem (chunk);
  assert (tc_idx < TCACHE_MAX_BINS);
  e->next = tcache->entries[tc_idx];
  tcache->entries[tc_idx] = e;
  ++(tcache->counts[tc_idx]);
}
```

```
static __always_inline void *
tcache_get (size_t tc_idx)
{
  tcache_entry *e = tcache->entries[tc_idx];
  assert (tc_idx < TCACHE_MAX_BINS);
  assert (tcache->entries[tc_idx] > 0);
  tcache->entries[tc_idx] = e->next;
  --(tcache->counts[tc_idx]);
  return (void *) e;
}
```

```
#if USE_TCACHE
  {
    size_t tc_idx = csize2tidx (size);

    if (tcache
        && tc_idx < mp_.tcache_bins
        && tcache->counts[tc_idx] < mp_.tcache_count)
      {
        tcache_put (p, tc_idx);
        return;
      }
  }
#endif
```

_int_free

```
if (tc_idx < mp_.tcache_bins
    /*&& tc_idx < TCACHE_MAX_BINS*/ /* to appease gcc */
    && tcache
    && tcache->entries[tc_idx] != NULL)
  {
    return tcache_get (tc_idx);
  }
```

_libc_malloc

Tcache Dup

- Libc 2.31, 可以看到多放了 Key

```
static __always_inline void
tcache_put (mchunkptr chunk, size_t tc_idx)
{
  tcache_entry *e = (tcache_entry *) chunk2mem (chunk);

  /* Mark this chunk as "in the tcache" so the test in _int_free will
     detect a double free.  */
  e->key = tcache;

  e->next = tcache->entries[tc_idx];
  tcache->entries[tc_idx] = e;
  ++(tcache->counts[tc_idx]);
}
```

```
static __always_inline void *
tcache_get (size_t tc_idx)
{
  tcache_entry *e = tcache->entries[tc_idx];
  tcache->entries[tc_idx] = e->next;
  --(tcache->counts[tc_idx]);
  e->key = NULL;
  return (void *) e;
}
```

Tcache Dup

- Libc 2.31
- _int_free 多了檢查
- 被 free 過的 Chunk, key 會寫入 tcache
- 若要 free 的 Chunk key 為 tcache, 則懷疑他已被 free 過一次, 遍尋 tcache list 檢查是否已被 free
- 若能改寫 key 就能繞過

```
size_t tc_idx = csize2tidx (size);
if (tcache != NULL && tc_idx < mp_.tcache_bins)
  {
    /* Check to see if it's already in the tcache.  */
    tcache_entry *e = (tcache_entry *) chunk2mem (p);

    /* This test succeeds on double free.  However, we don't 100%
       trust it (it also matches random payload data at a 1 in
       2^<size_t> chance), so verify it's not an unlikely
       coincidence before aborting.  */
    if (__glibc_unlikely (e->key == tcache))
      {
        tcache_entry *tmp;
        LIBC_PROBE (memory_tcache_double_free, 2, e, tc_idx);
        for (tmp = tcache->entries[tc_idx];
             tmp;
             tmp = tmp->next)
          if (tmp == e)
            malloc_printerr ("free(): double free detected in tcache 2");
        /* If we get here, it was a coincidence.  We've wasted a
           few cycles, but don't abort.  */
      }

    if (tcache->counts[tc_idx] < mp_.tcache_count)
      {
        tcache_put (p, tc_idx);
        return;
      }
  }
```

Tcache Dup

- Libc 2.31
- _int_malloc 基本沒變
- 檢查 counts 的方式從 != NULL 改為 > 0

```
if (tc_idx < mp_.tcache_bins
    && tcache
    && tcache->counts[tc_idx] > 0)
  {
    return tcache_get (tc_idx);
  }
```

Tcache Dup

Demo

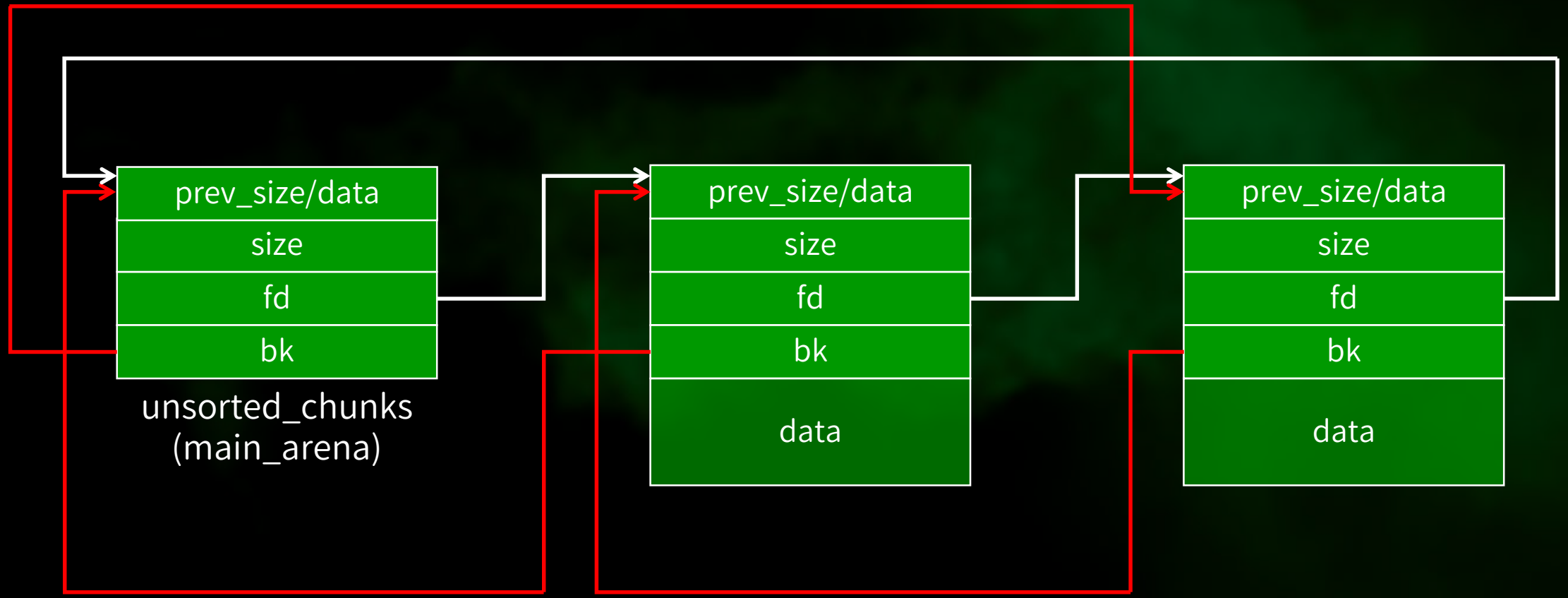
Basic Knowledge

Unsorted Bin

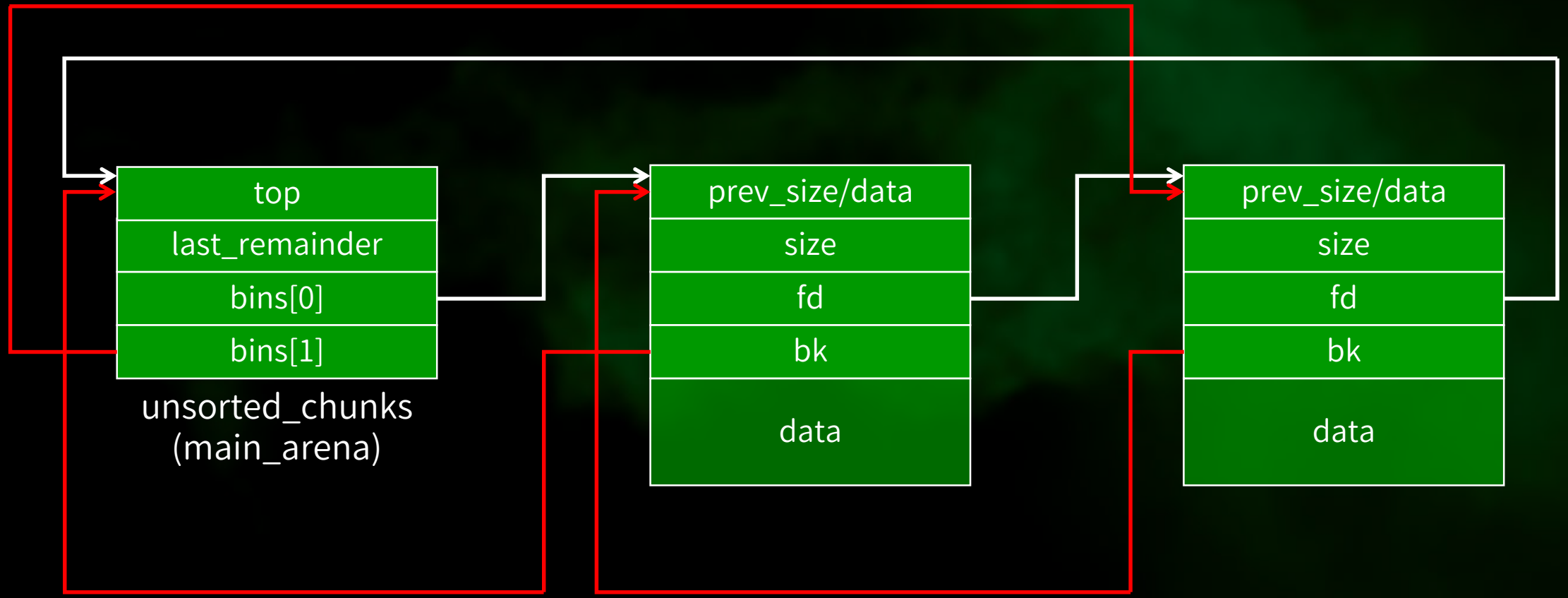
Unsorted Bin

- 被 free 的 Chunk 沒被放到 Tcache 和 Fastbin 時, 則
  - 若鄰近的上一塊 Chunk 為 Free, 則合併到它裡
  - 若鄰近的下一塊 Chunk 是 Top Chunk, 則合併到 Top Chunk 裡
  - 若下一塊不是, 那再看它是不是 Free, 是則合併進此 Chunk 裡, 並加到 Unsorted Bin 鏈表中
- Unsorted Bin 目的是給被回收的 Chunk 至少一次的機會再被分配出去
- Unsorted Bin 為 Circular Doubly Linked List

Unsorted Bin



Unsorted Bin



Unsorted Bin in Libc 2.31

Source Code Reading

Unsorted Bin in Libc 2.31

Demo

Basic Knowledge

Consolidate

Consolidate

- 大致有三種合併方式
- Backward consolidate
- Forward consolidate
- malloc_consolidate
- 有兩塊 Free Chunk 要合併成一塊, 調整其中一塊的大小, 並將另一塊從鏈表中移除

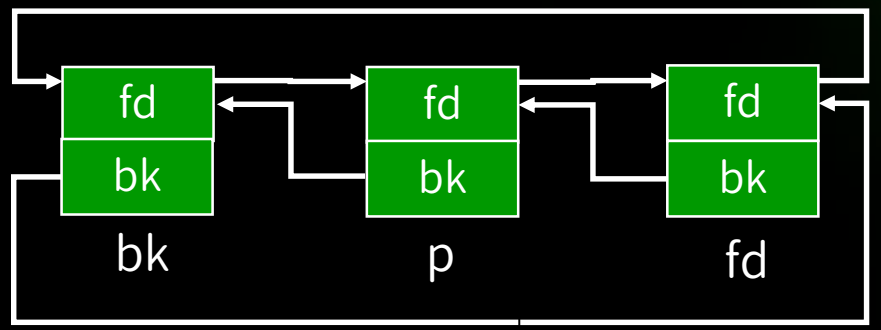
```
/* consolidate backward */
if (!prev_inuse(p)) {
  prevsize = prev_size (p);
  size += prevsize;
  p = chunk_at_offset(p, -((long) prevsize));
  if (__glibc_unlikely (chunksize(p) != prevsize))
    malloc_printerr ("corrupted size vs. prev_size while consolidating");
  unlink_chunk (av, p);
}
```

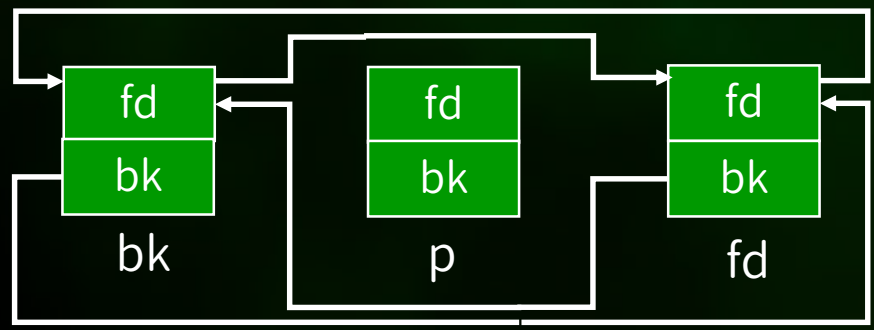
```
/* consolidate forward */
if (!nextinuse) {
  unlink_chunk (av, nextchunk);
  size += nextsize;
} else
```

```
if ((unsigned long)(size) >= FASTBIN_CONSOLIDATION_THRESHOLD) {
  if (atomic_load_relaxed (&av->have_fastchunks))
    malloc_consolidate(av);
```

Consolidate

- unlink_chunk
- 檢查 p 的 size 跟下一塊 Chunk 紀錄的 prev_size 是否一樣
- fd = p->fd
- bk = p->bk
- 檢查 fd->bk 為 p, 且 bk->fd 為 p
- fd->bk = bk
- bk->fd = fd





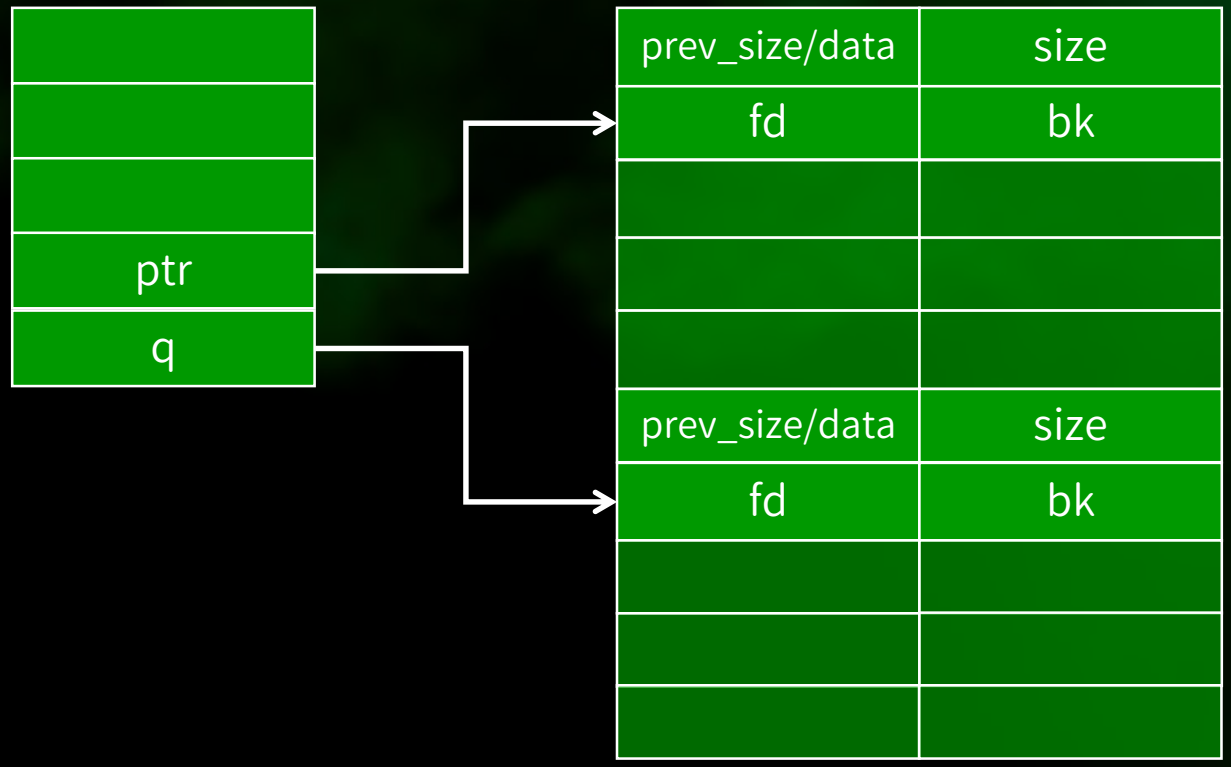
Consolidate in Libc 2.31

Source Code Reading

Unsafe Unlink

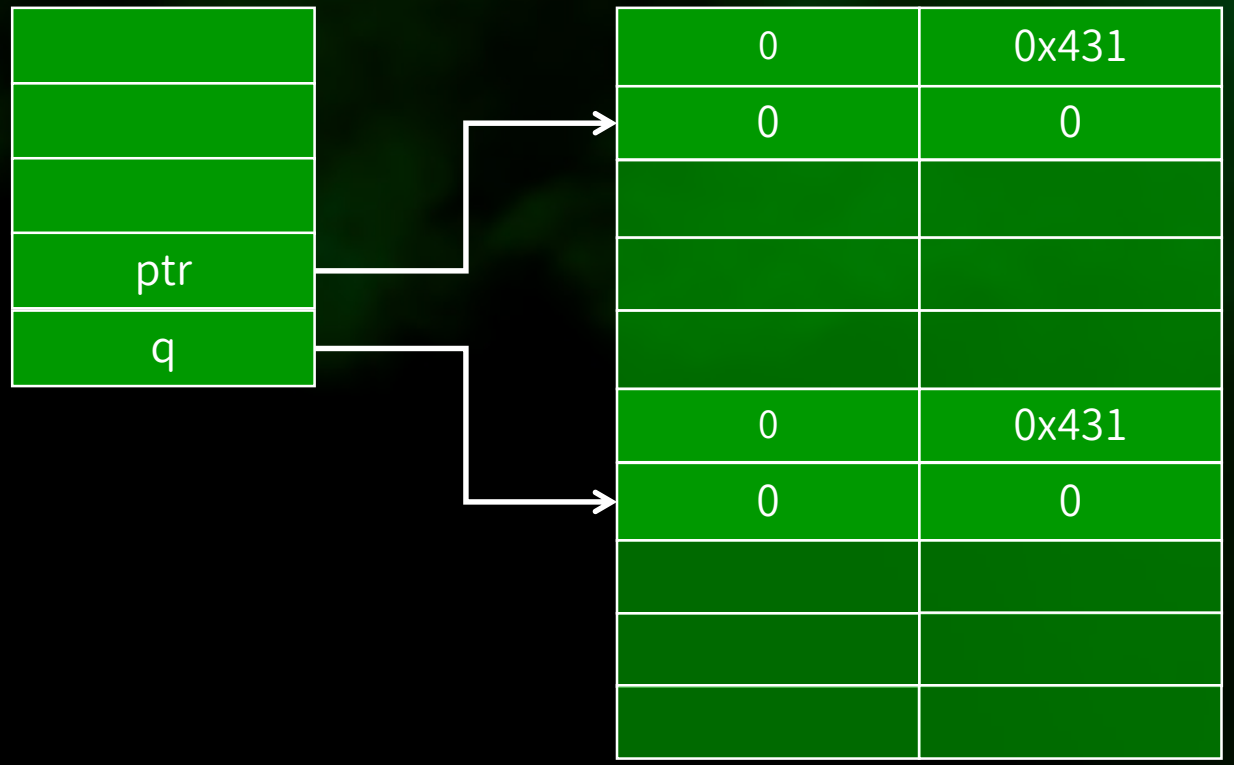
Unsafe Unlink

- 假設一下 Chunk 內容



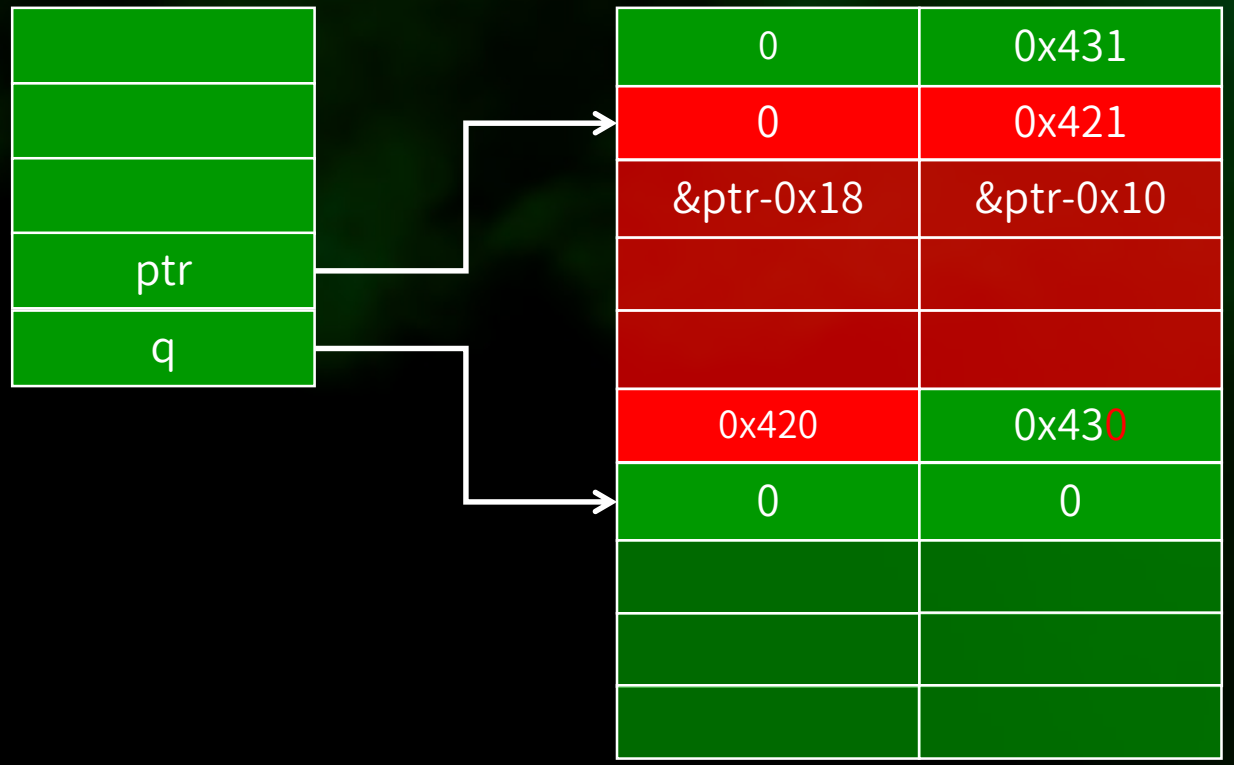
Unsafe Unlink

- 並假設 ptr 有越界寫 1 Byte 的漏洞



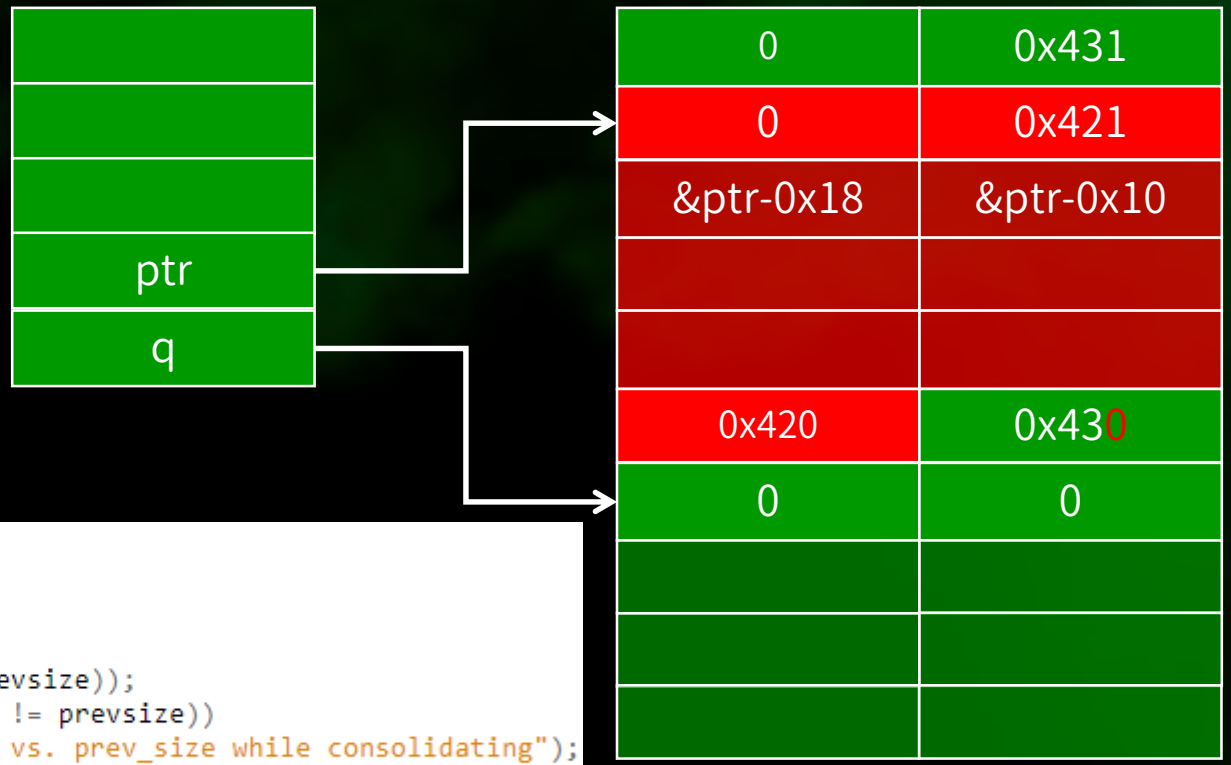
Unsafe Unlink

- 越界寫成以下後, free(q)



Unsafe Unlink

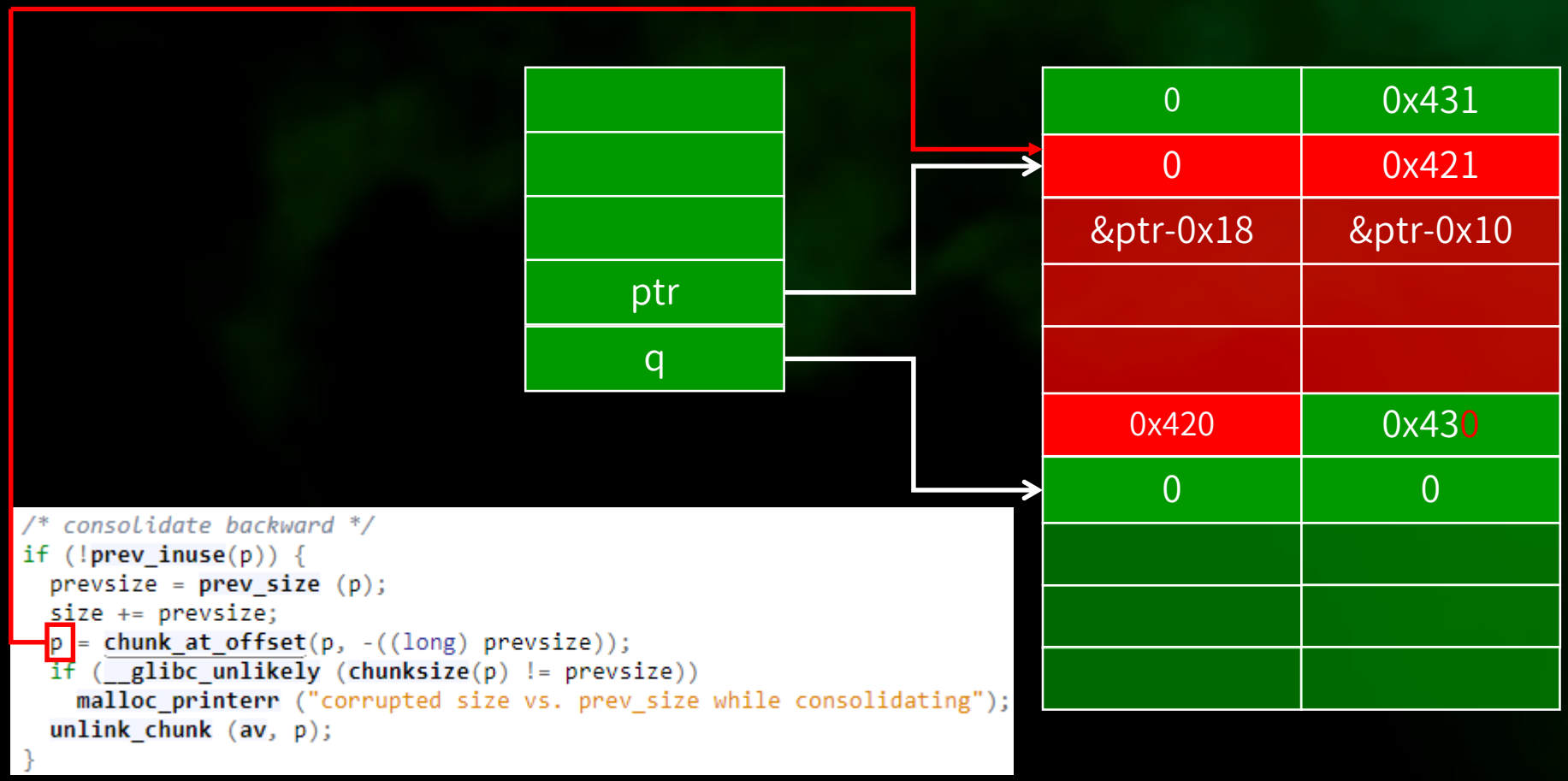
- free(q), q 的 size 不進入 fastbin & Tcache, 並由於 q 的 Prev_inuse 為 0, 進行往前合併




```c
/* consolidate backward */
if (!prev_inuse(p)) {
  prevsize = prev_size (p);
  size += prevsize;
  p = chunk_at_offset(p, -((long) prevsize));
  if (__glibc_unlikely (chunksize(p) != prevsize))
    malloc_printerr ("corrupted size vs. prev_size while consolidating");
  unlink_chunk (av, p);
}
```

Unsafe Unlink

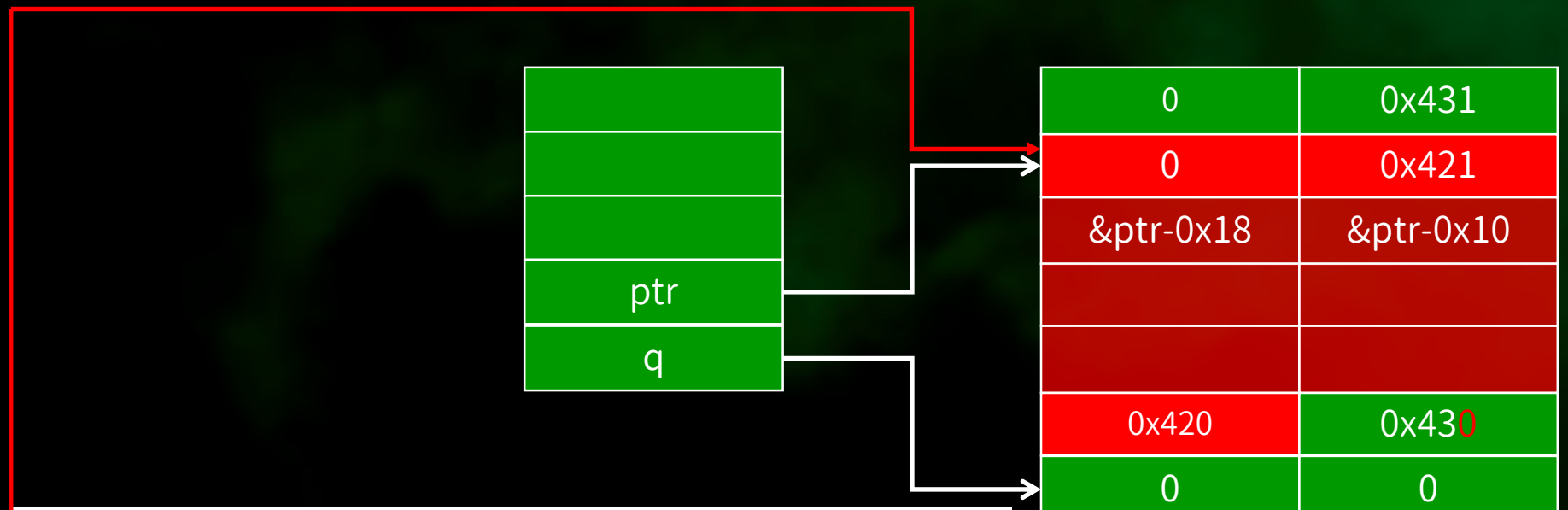
- 由於我們設置 prev_size 為 0x420, 所以 p 的指向如圖所示




```
/* consolidate backward */
if (!prev_inuse(p)) {
  prevsize = prev_size (p);
  size += prevsize;
  p = chunk_at_offset(p, -((long) prevsize));
  if (__glibc_unlikely (chunksize(p) != prevsize))
    malloc_printerr ("corrupted size vs. prev_size while consolidating");
  unlink_chunk (av, p);
}
```

Unsafe Unlink

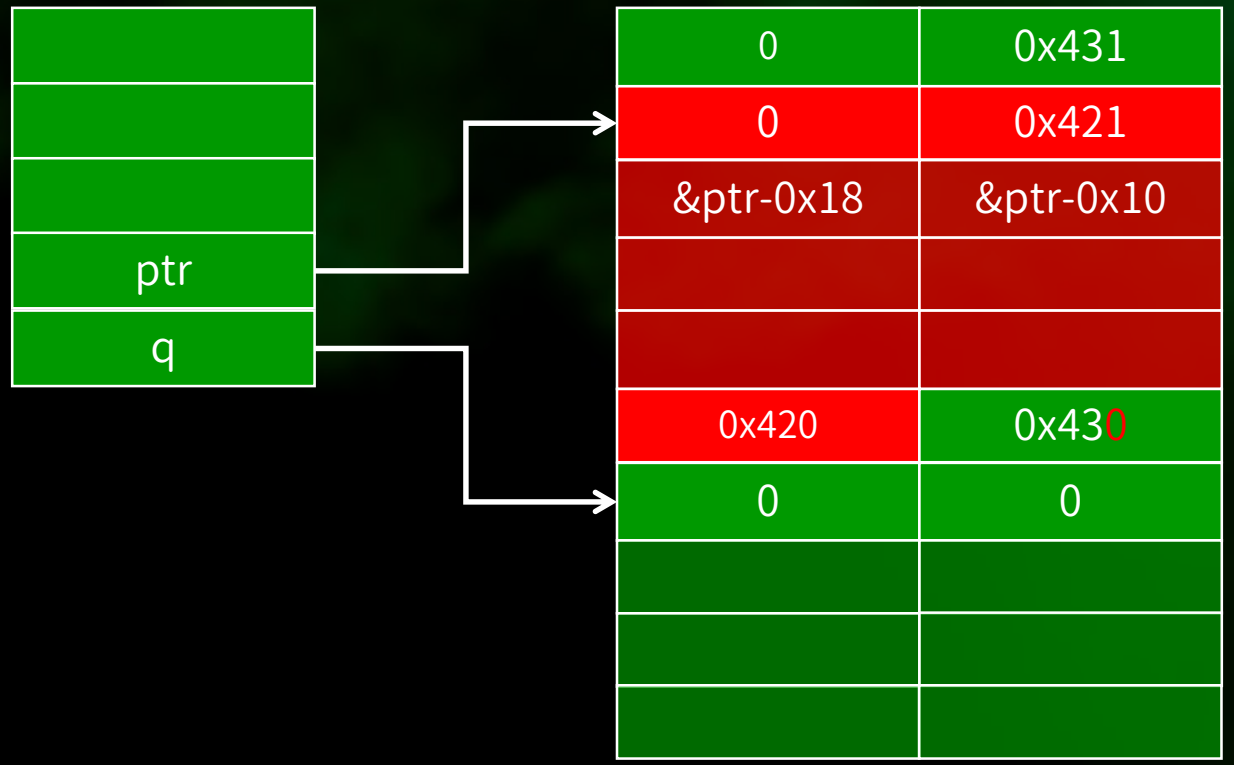
- unlink_chunk(av, p)




```
/* consolidate backward */
if (!prev_inuse(p)) {
  prevsize = prev_size (p);
  size += prevsize;
  p = chunk_at_offset(p, -((long) prevsize));
  if (__glibc_unlikely (chunksize(p) != prevsize))
    malloc_printerr ("corrupted size vs. prev_size while consolidating");
  unlink_chunk (av, p);
}
```

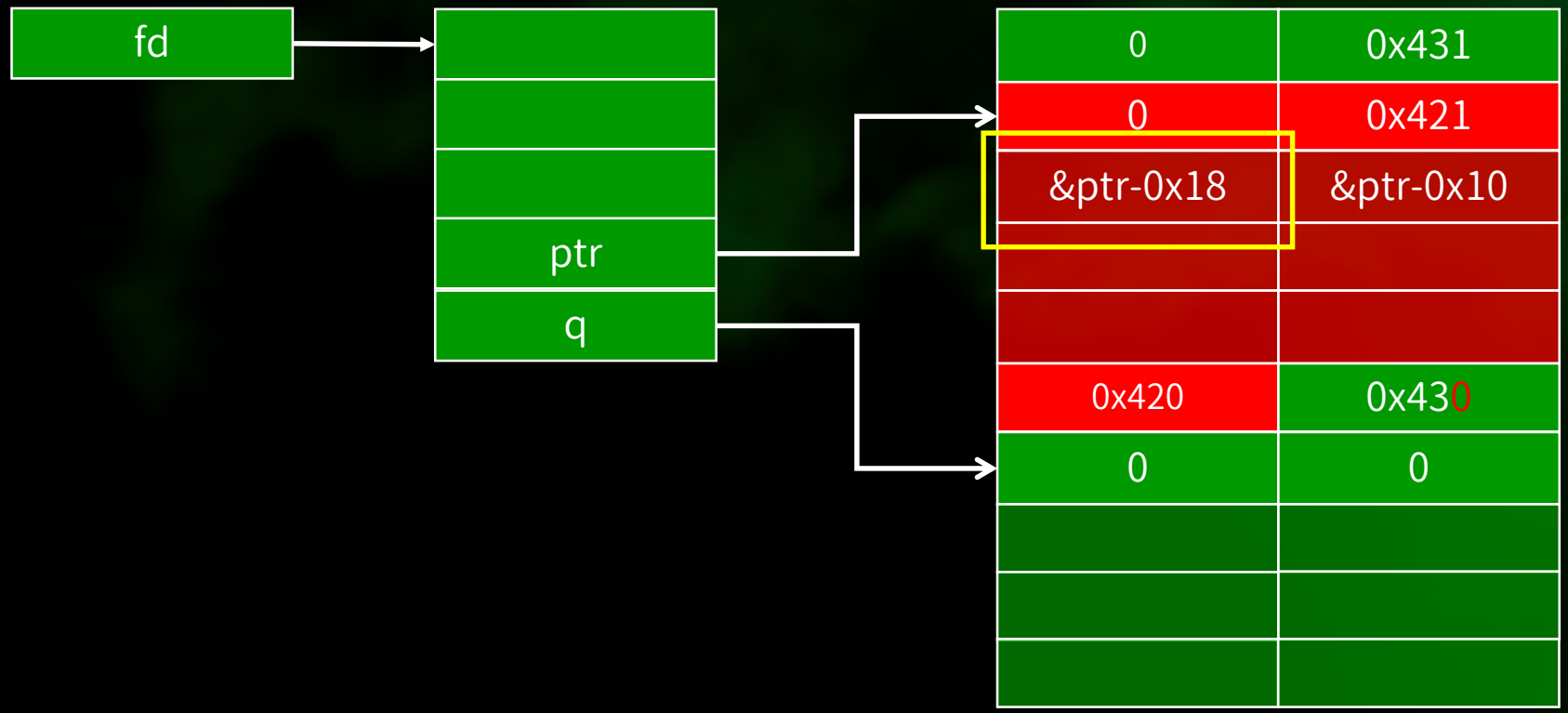
Unsafe Unlink

- 檢查 p 的 size 跟下一塊 Chunk 紀錄的 prev_size 是否一樣
- OK



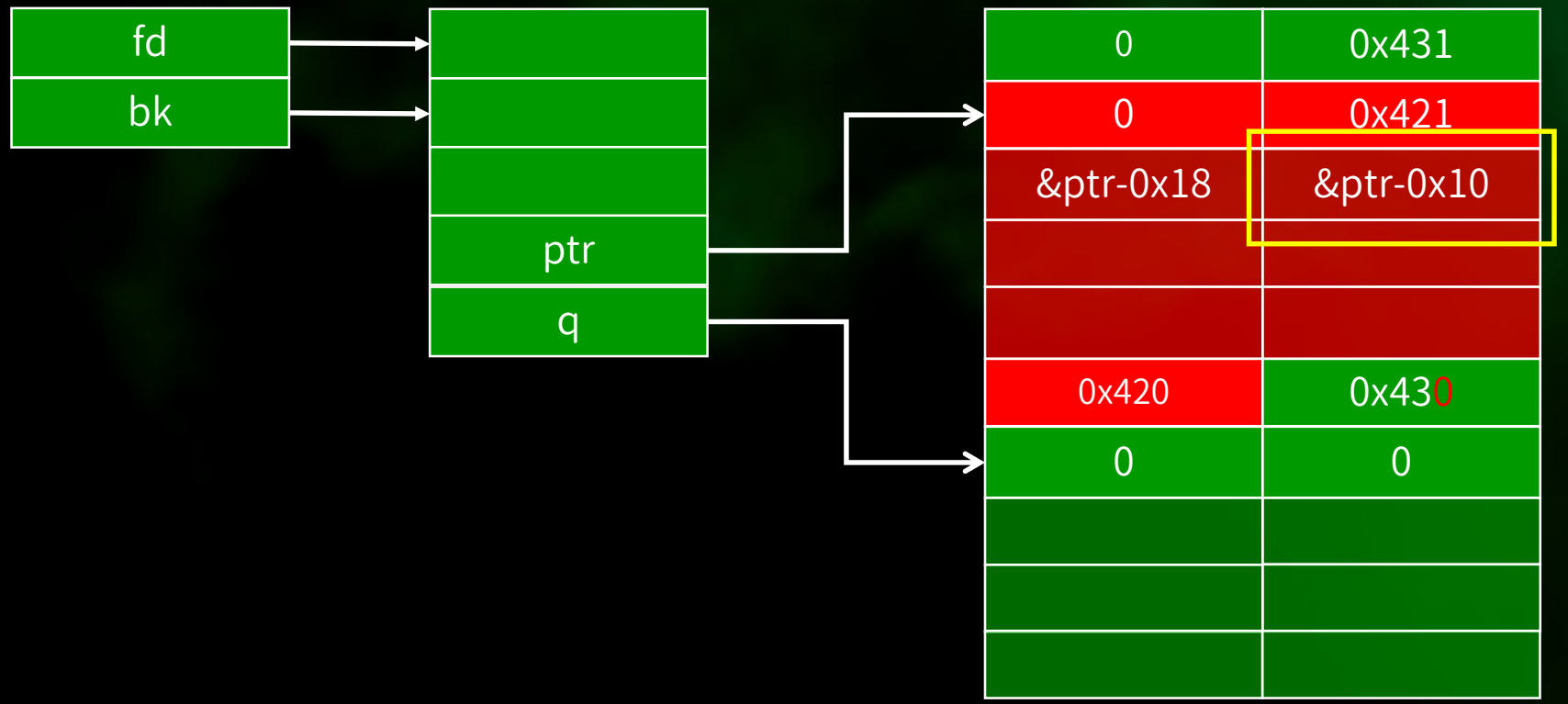
Unsafe Unlink

- fd = p->fd



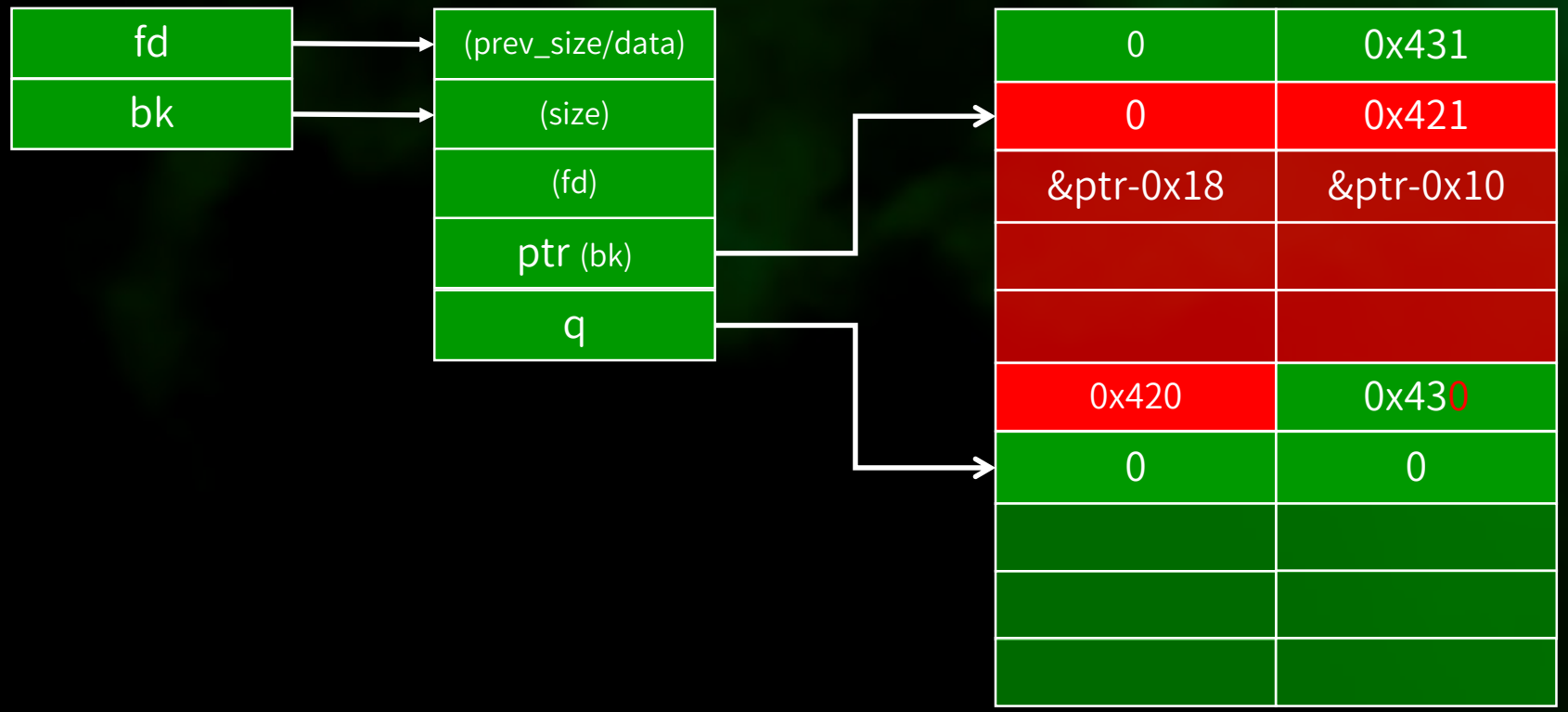
Unsafe Unlink

- bk = p->bk



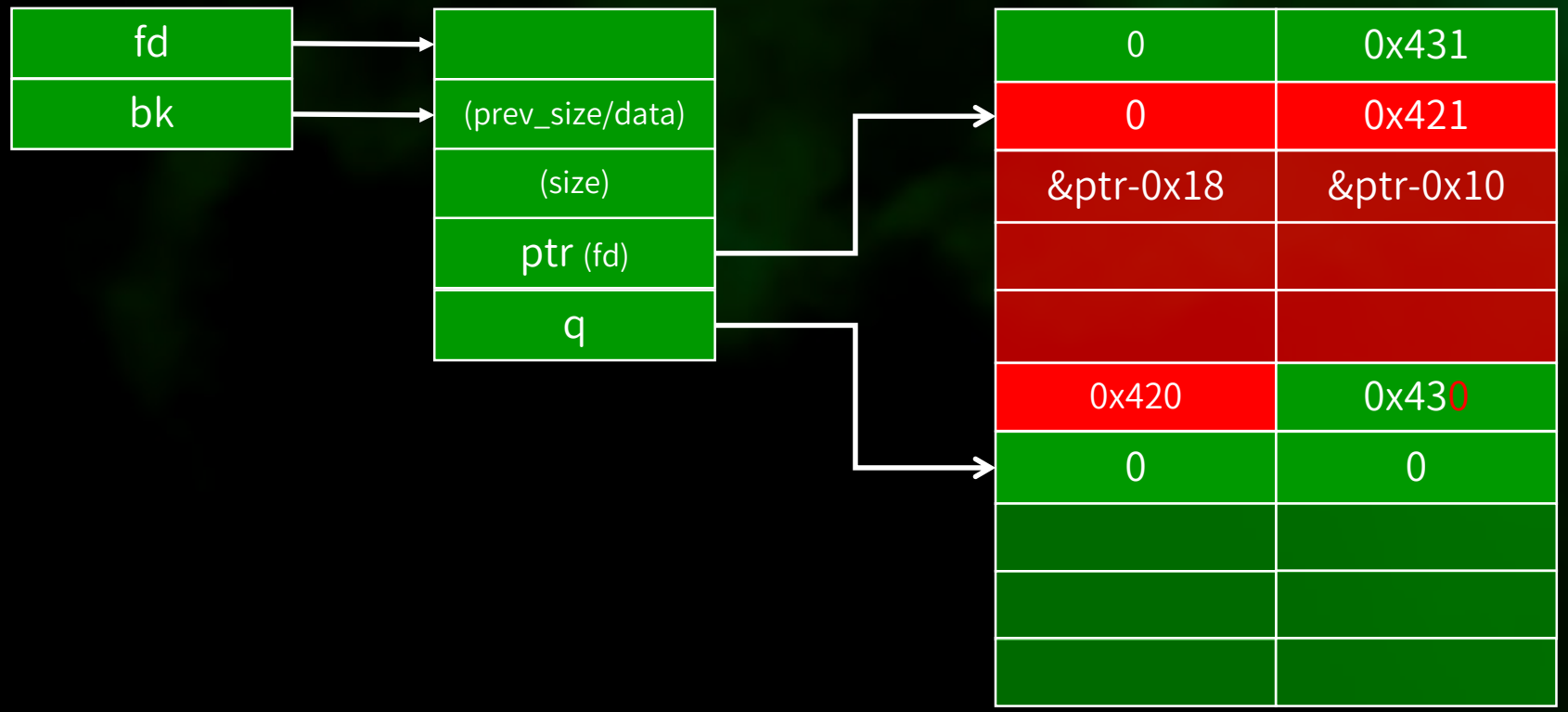
Unsafe Unlink

- 檢查 fd->bk 為 p
- OK



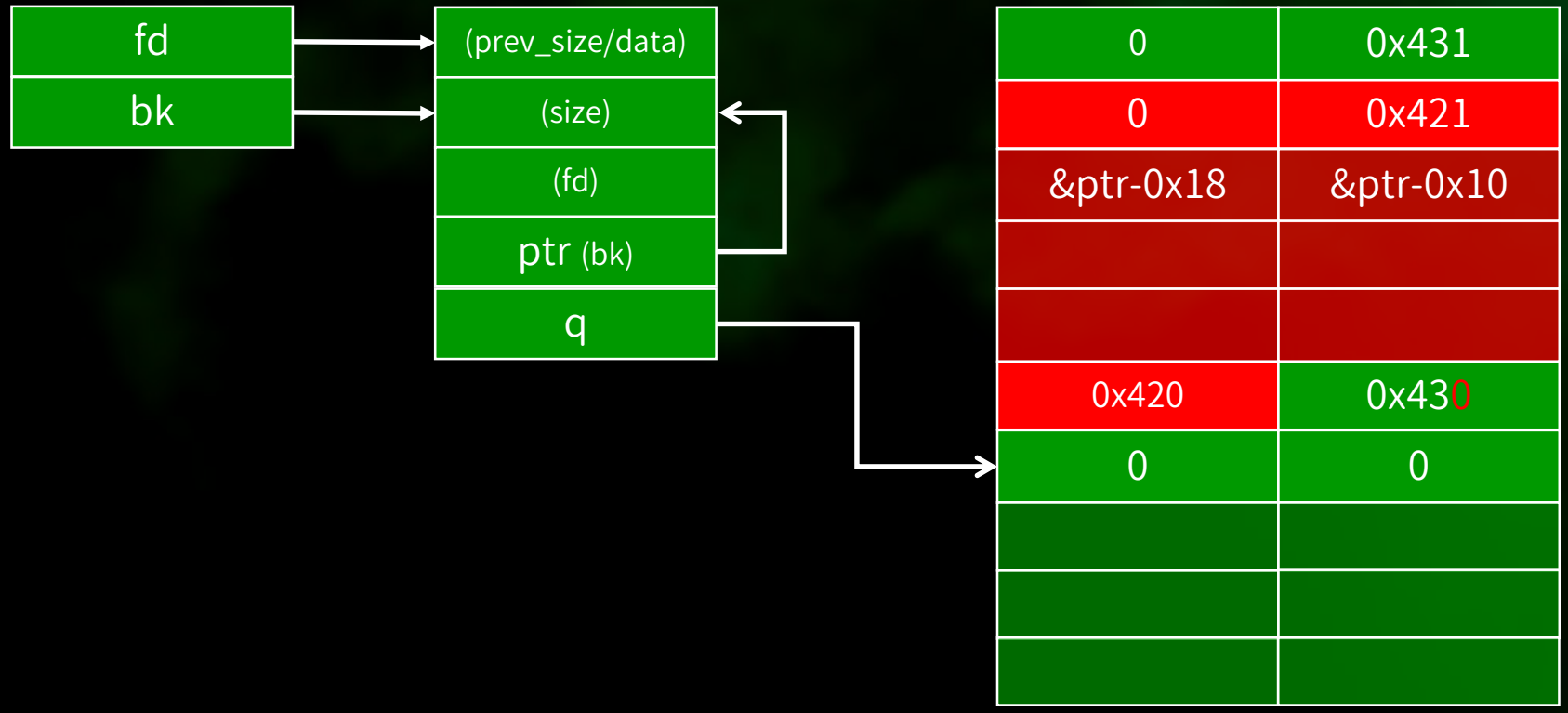
Unsafe Unlink

- 檢查 bk->fd 為 p
- OK



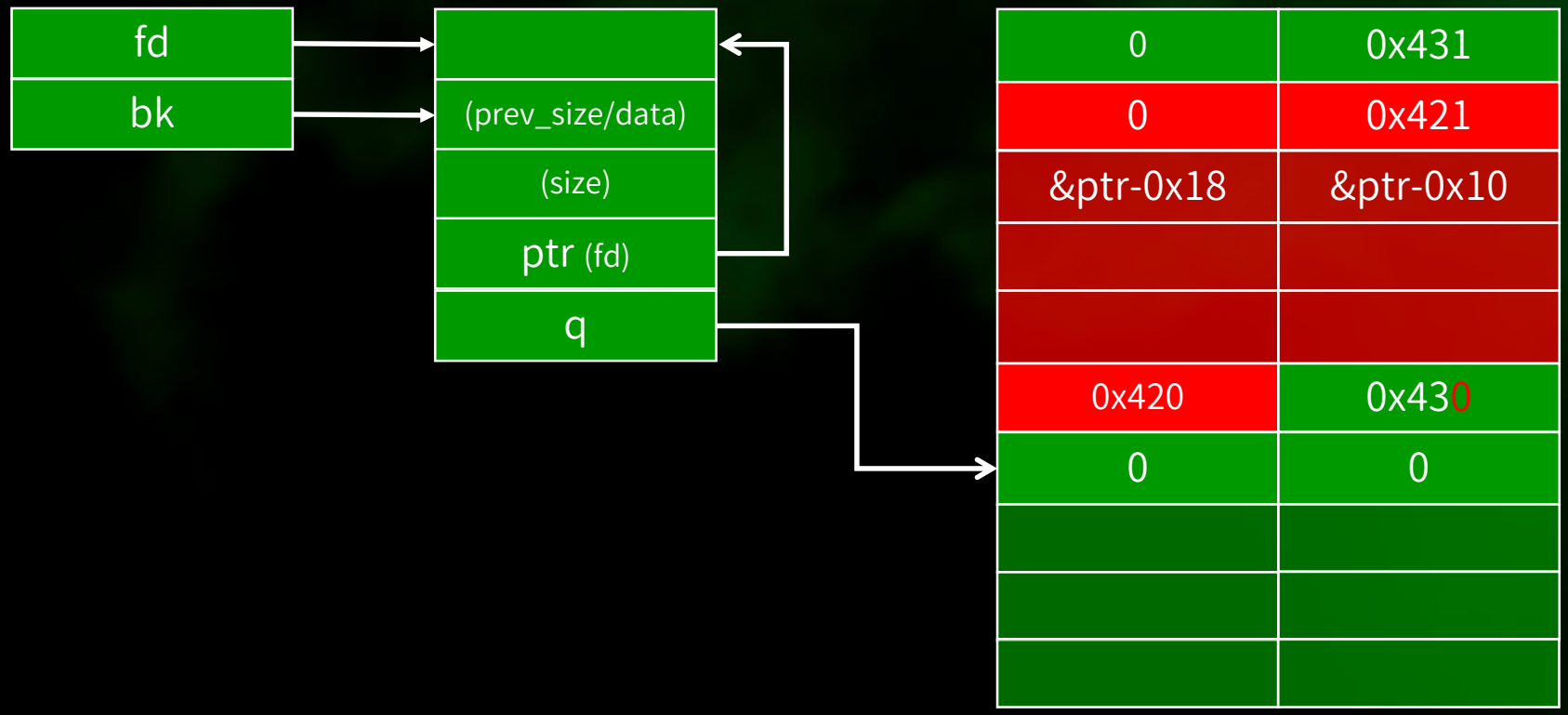
Unsafe Unlink

- fd->bk = bk



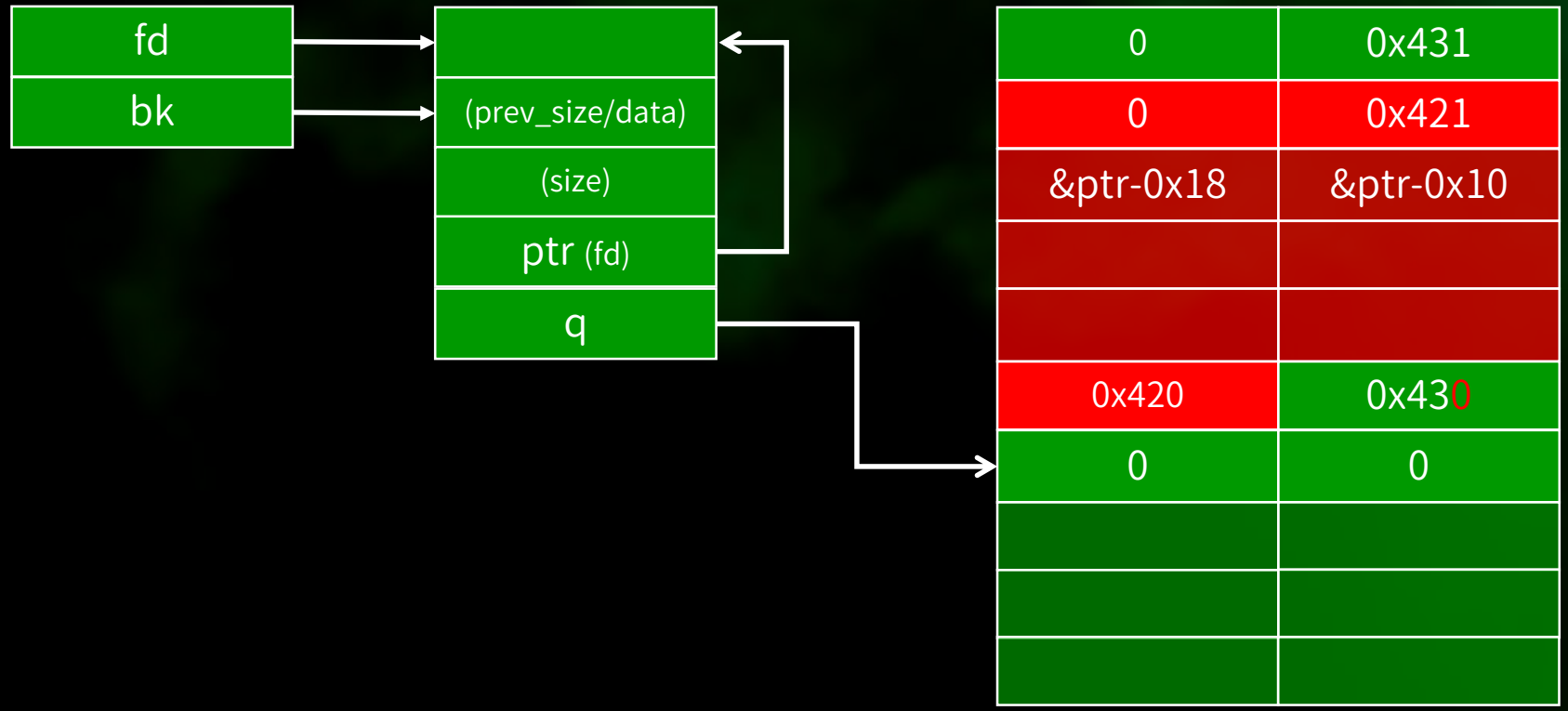
Unsafe Unlink

- bk->fd = fd



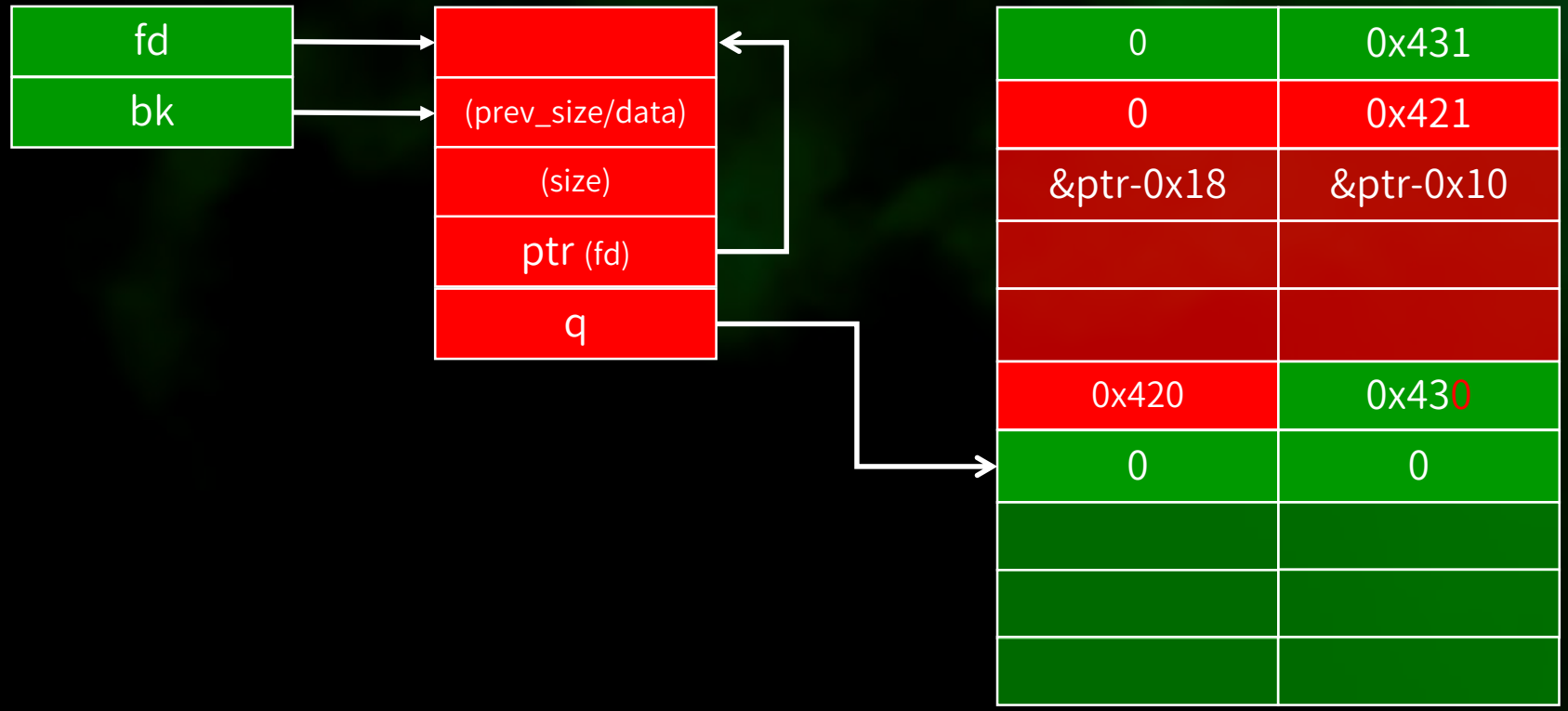
Unsafe Unlink

- ptr 作為資料 pointer, 通常能再次寫入
- 正常使用時, ptr 應只指向 heap, 然而攻擊後會指向 &ptr – 0x18



Unsafe Unlink

- 再次往 ptr 寫入就能再蓋 ptr, 就能製造任意位置寫入



Q & A

Thanks

疫情期間少出門勤洗手