Binary Exploit - II

u1f383 🎃

About

- u1f383 🎃
- Pwner
- Zoolab
- NCtfU / Xx TSJ xX / Goburin’

Outline

- Heap introduction
  - Memory allocation
  - Data structure
- Code tracing
- Vulnerability
  - UAF
  - Heap overflow
  - Double free

Outline

- Exploitation goal
  - Write hook
- Exploitation tech
  - Tcache poisoning
  - Fastbin attack
  - Overlapping chunks
- Appendix

Heap introduction

$ Heap introduction

Basic

- 在 compile time 程式無法確定需要多少空間，因此需要有方法能夠在 runtime 分配以及釋放記憶體，才能有效的利用記憶體
- 服務根據種類以及開發單位，會有不同的記憶體管理機制
  - Glibc - ptmalloc
  - Google - tcmalloc
  - Facebook - jemalloc
  - ...

$ Heap introduction

Basic

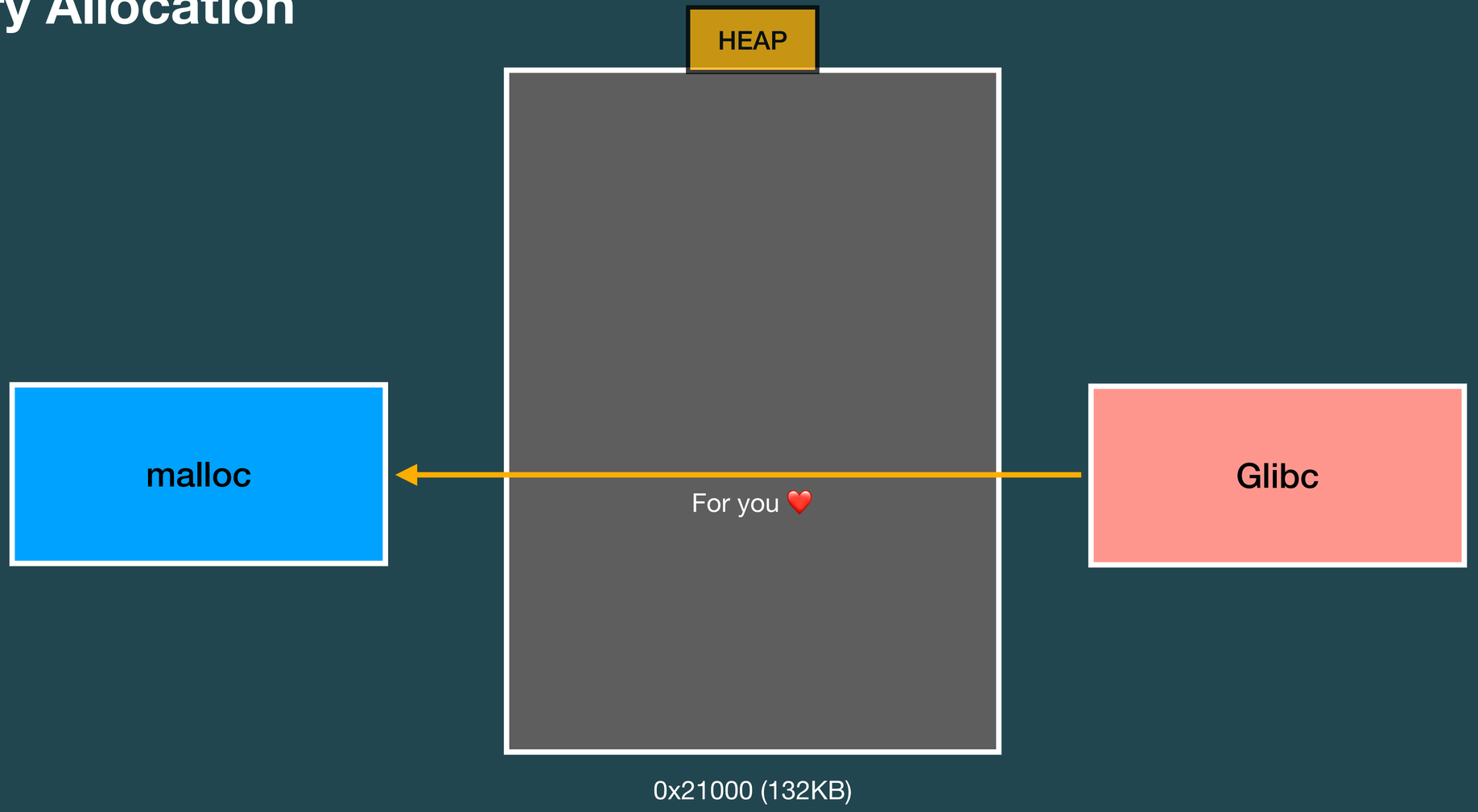
- 在 C 當中，最常用來分配記憶體的 function 為 malloc，而 free 則是用來釋放記憶體
- 其他與 memory allocation 相關的 function
  - realloc - 更新已分配的記憶體大小
  - calloc - 在回傳前清空記憶體的內容

$ Heap introduction

Memory Allocation

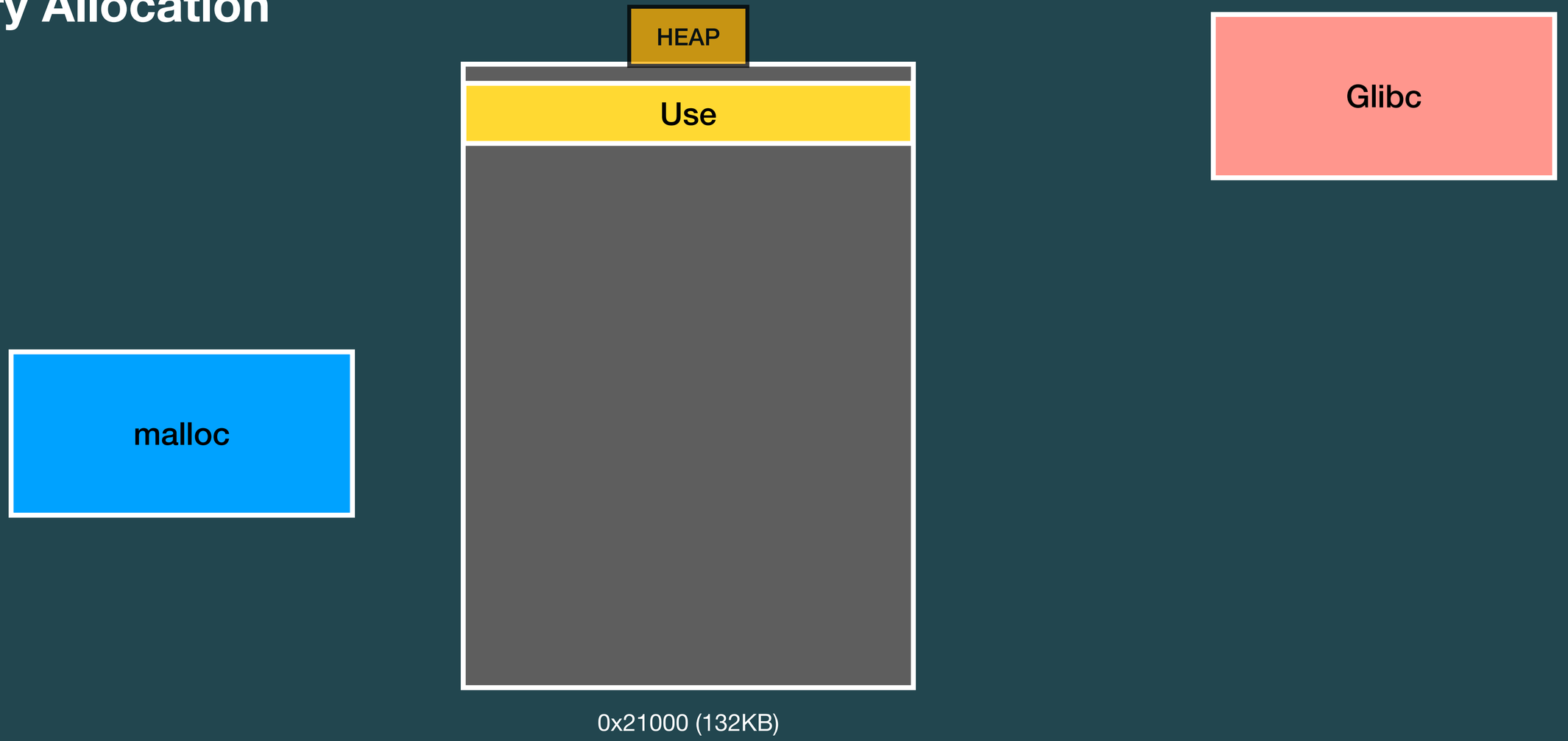
$ Heap introduction

Memory Allocation



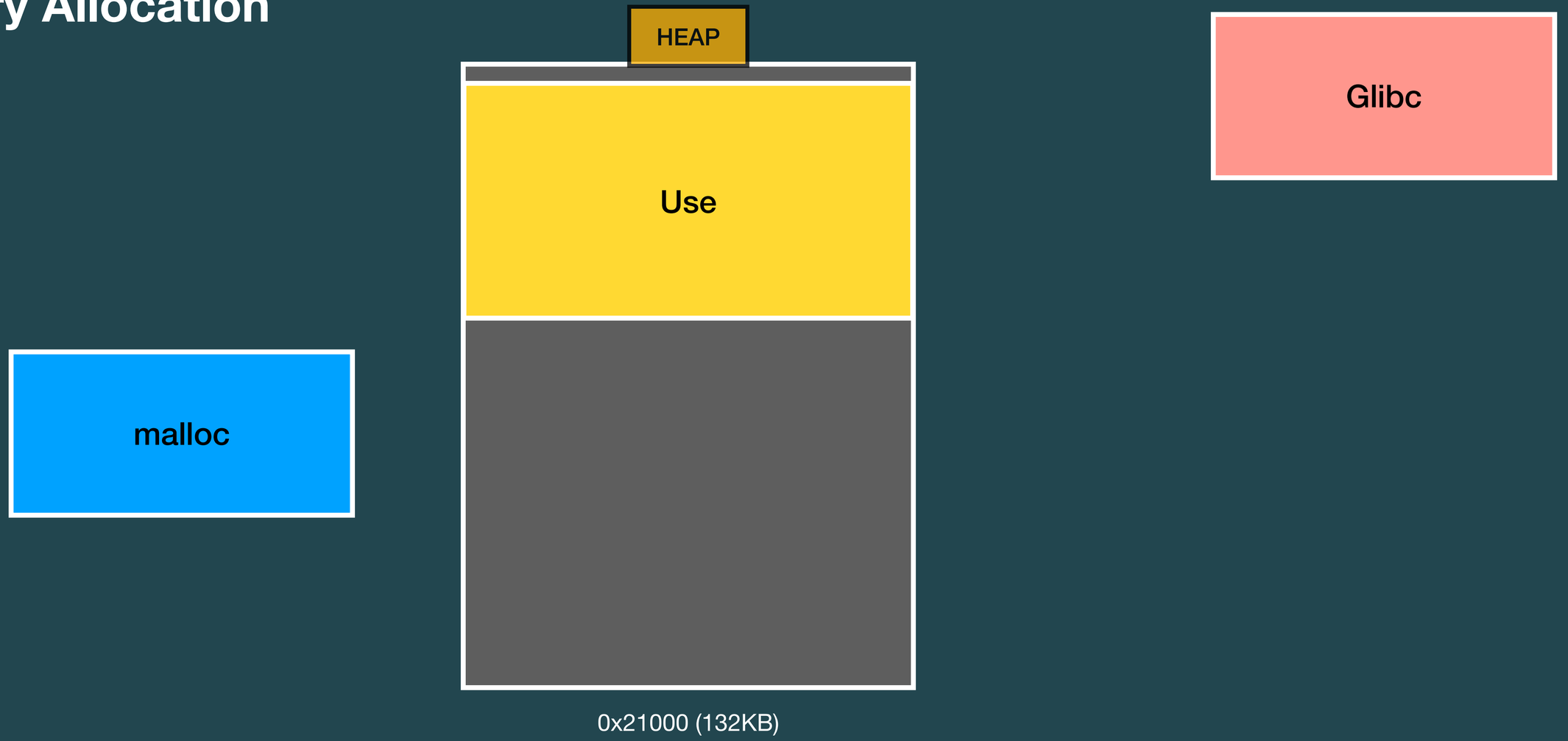
$ Heap introduction

Memory Allocation



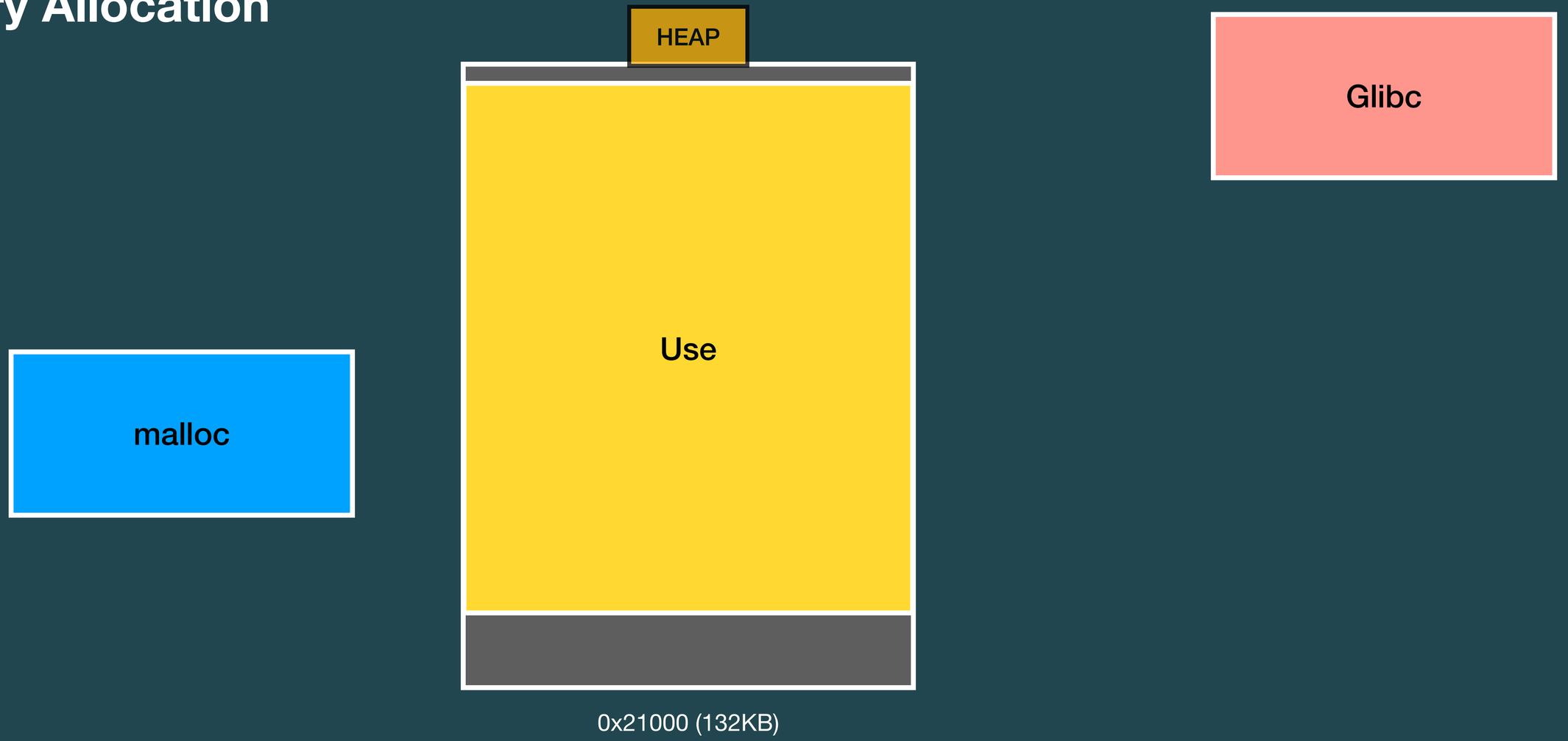
$ Heap introduction

Memory Allocation



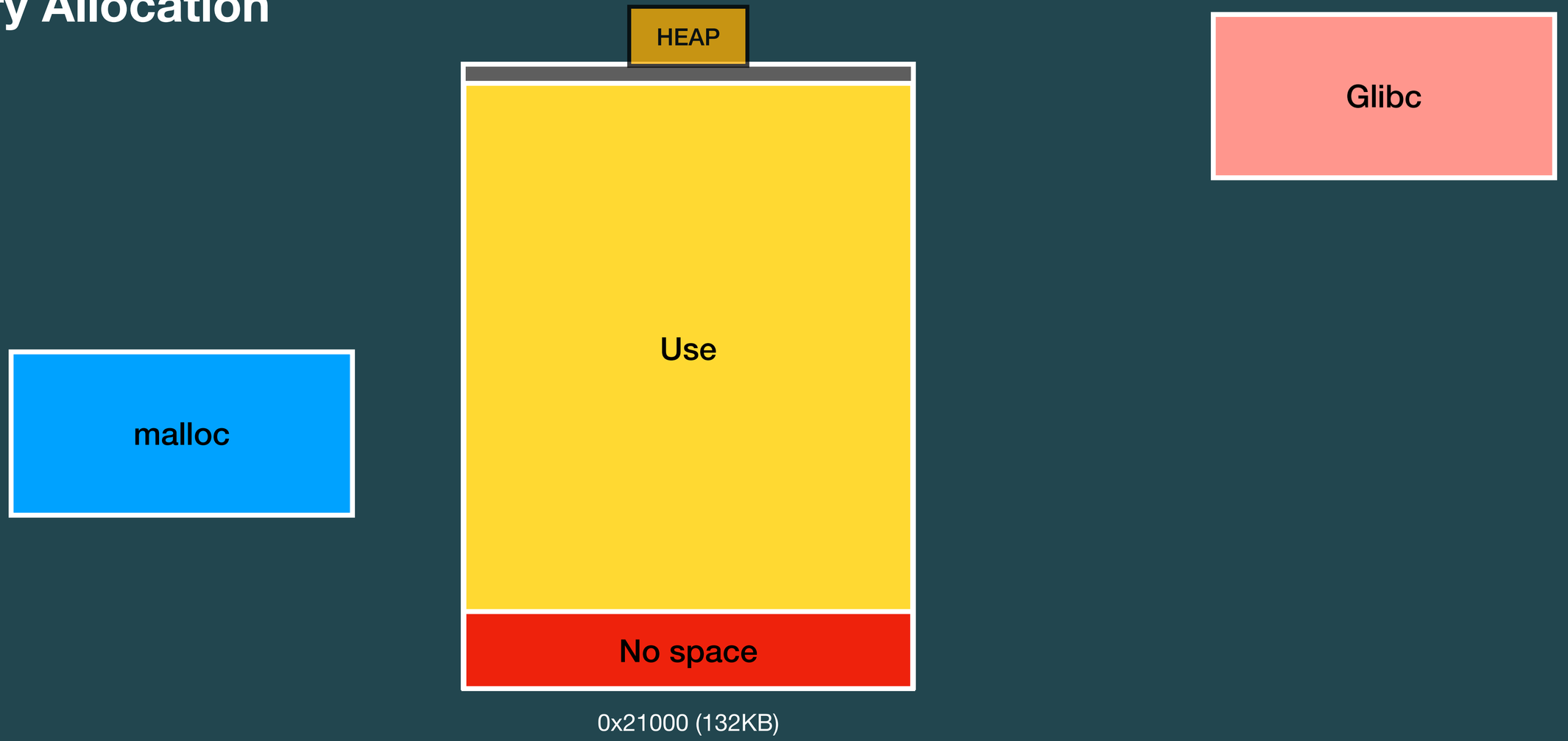
$ Heap introduction

Memory Allocation



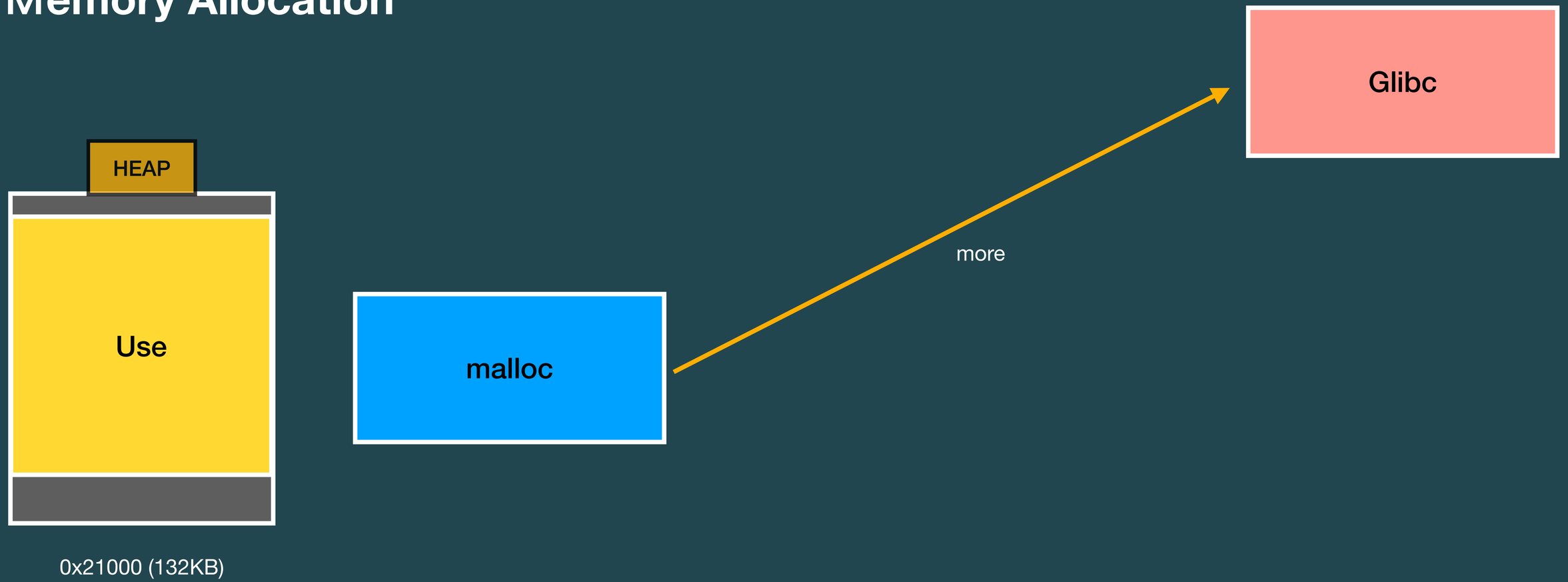
$ Heap introduction

Memory Allocation



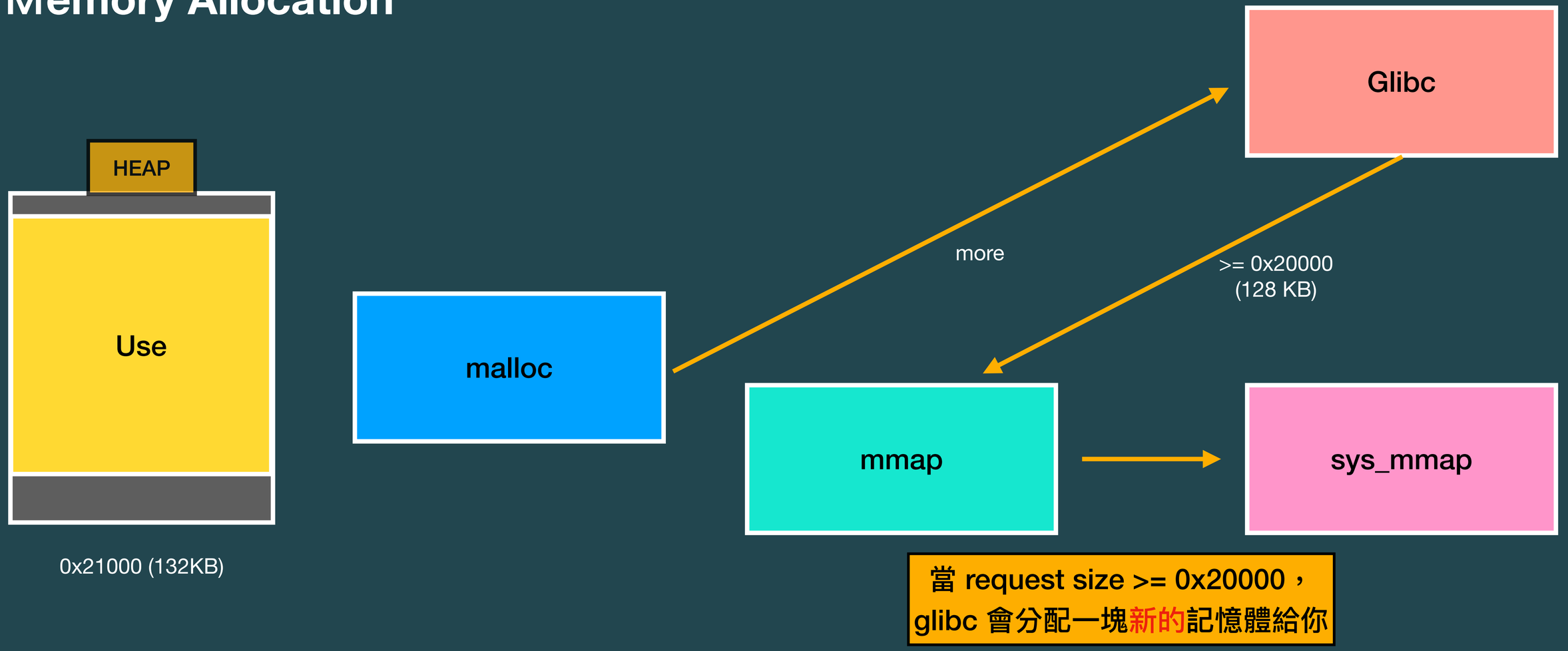
$ Heap introduction

Memory Allocation



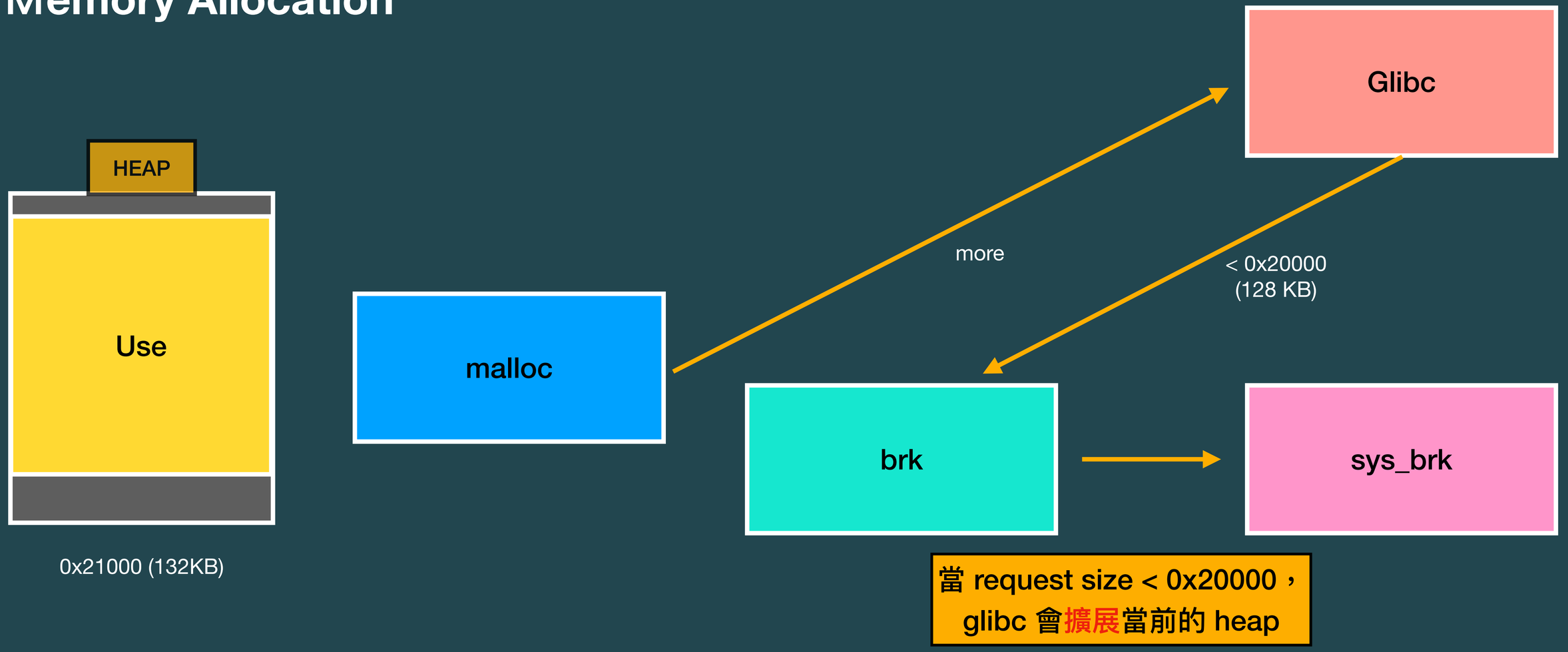
$ Heap introduction

Memory Allocation



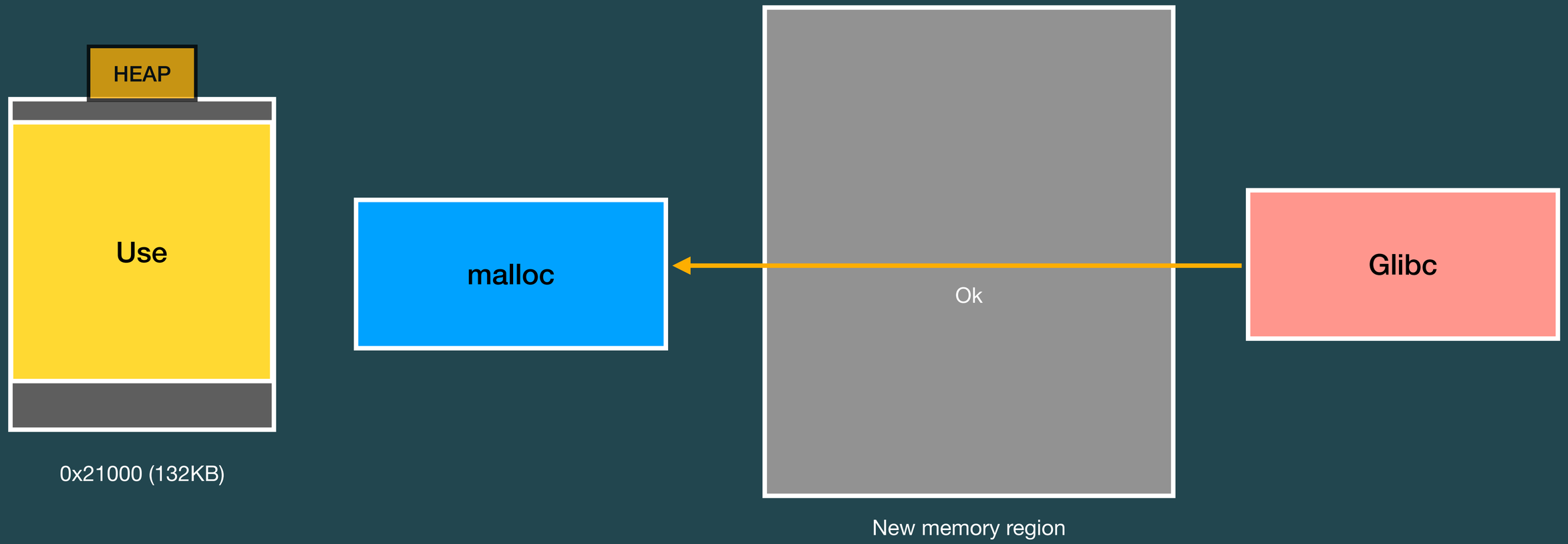
$ Heap introduction

Memory Allocation



$ Heap introduction

Memory Allocation



$ Heap introduction

Memory Allocation

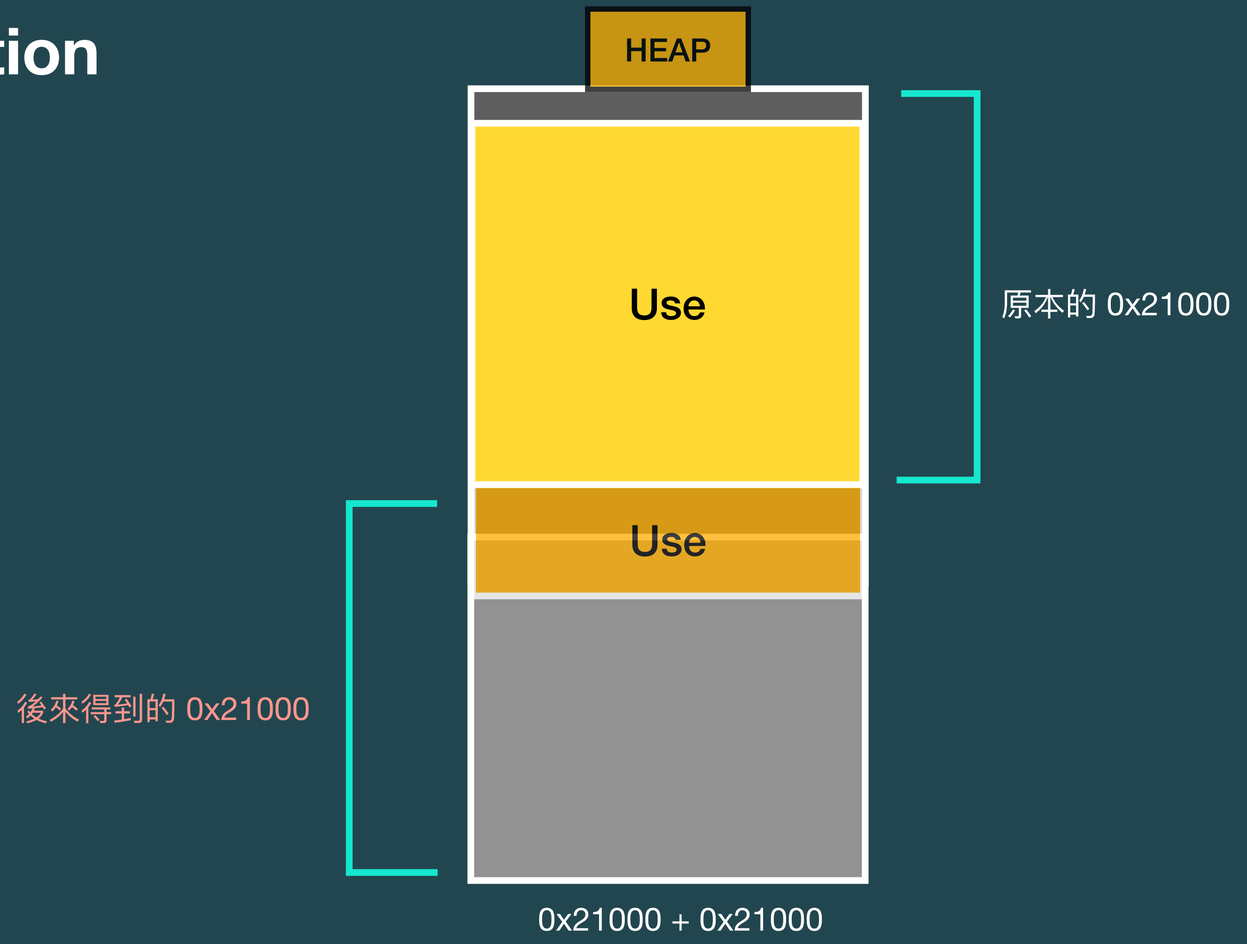
0x21000

對齊 page 後的請求大小

mmap
(size >= 0x20000)

$ Heap introduction

Memory Allocation

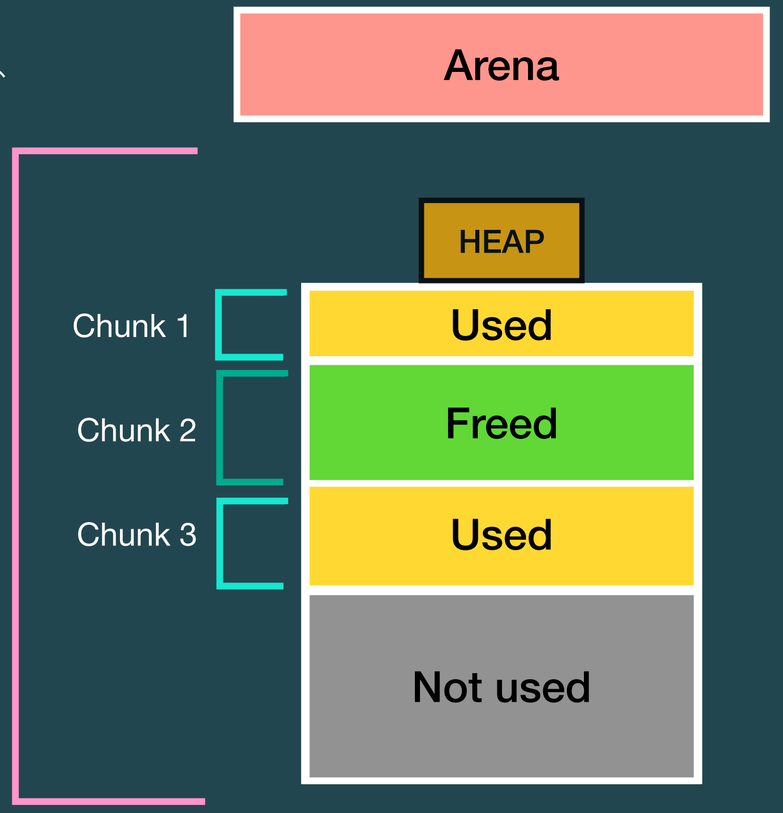


$ Heap introduction

Data Structure

- 整個記憶體的分配機制由三個部分構成：
  - 使用者透過 malloc 取得的小塊記憶體 - chunk
  - 用來分發這些小塊記憶體的大塊記憶體 - heap
  - 用來儲存大塊記憶體的相關資訊的資料結構 - arena

哪些還沒用、
釋放掉的 chunk 位址、
...



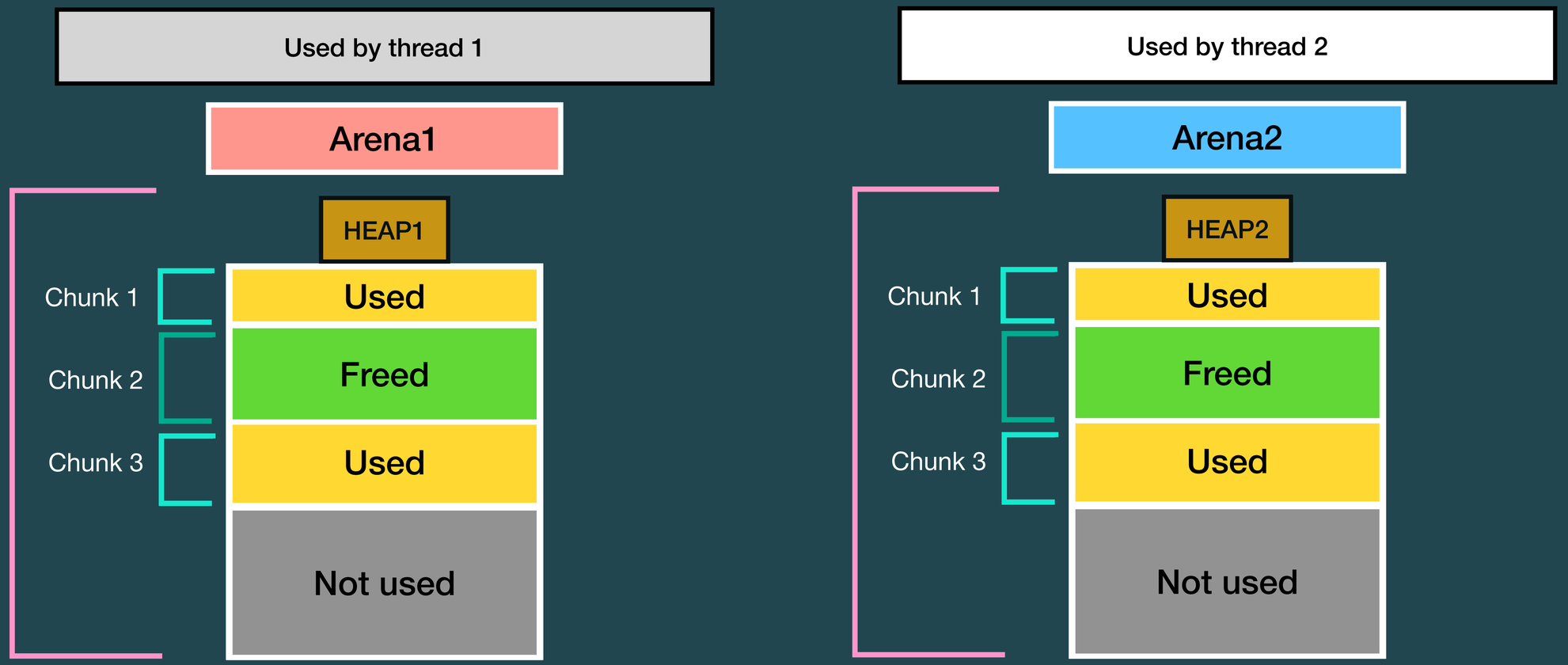
$ Heap introduction

Data Structure

- Chunk - **struct malloc_chunk**
- Heap - **struct _heap_info**
  - 每個 thread 會有一個 heap
- Arena - **struct malloc_state**
  - 正常情況下，每個 thread 會有一個 arena
  - 但是在 thread 的數量太多的情況下，多個 thread 會共同使用一個 arena
  - main thread 使用的 arena 稱作 main_arena

$ Heap introduction

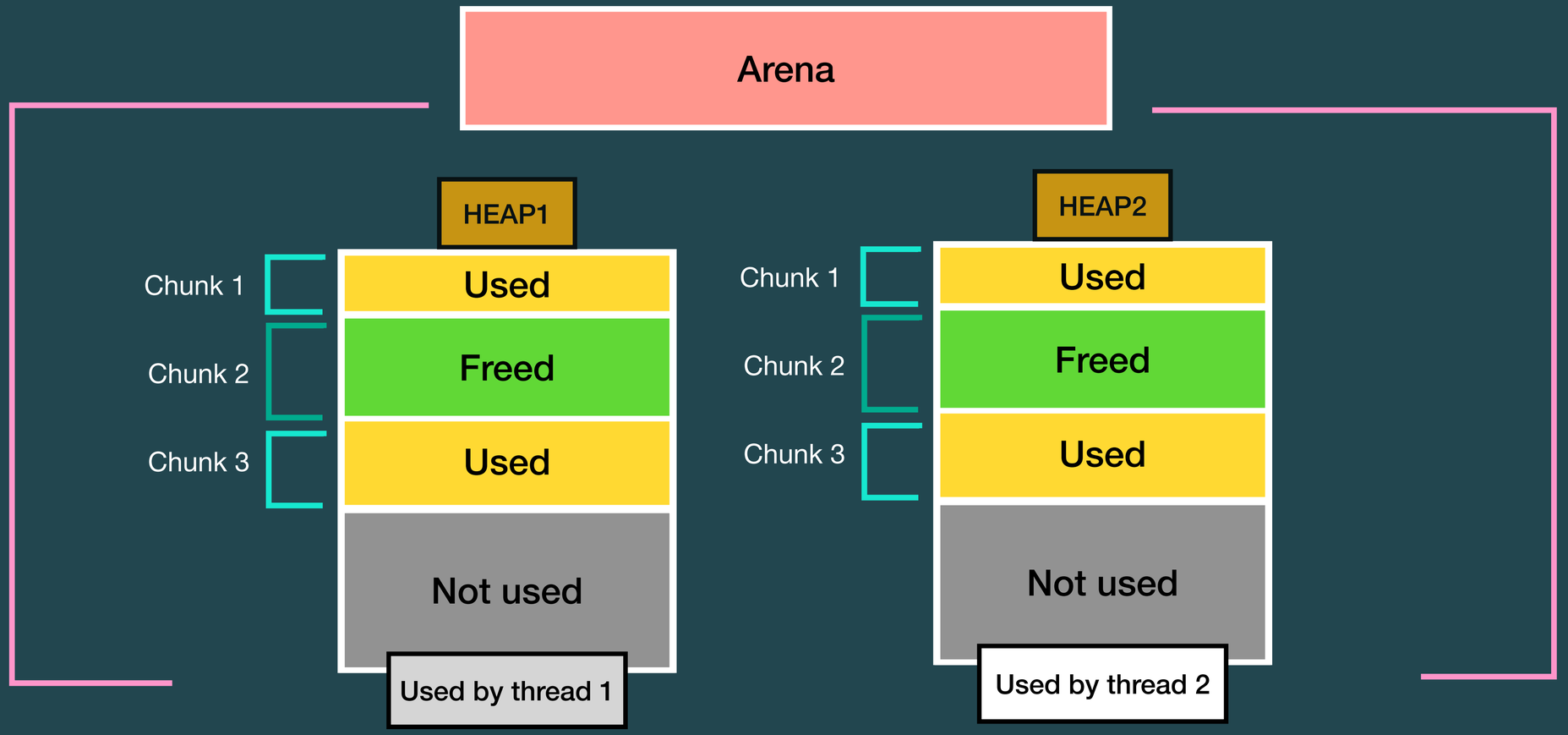
Data Structure



正常情況

$ Heap introduction

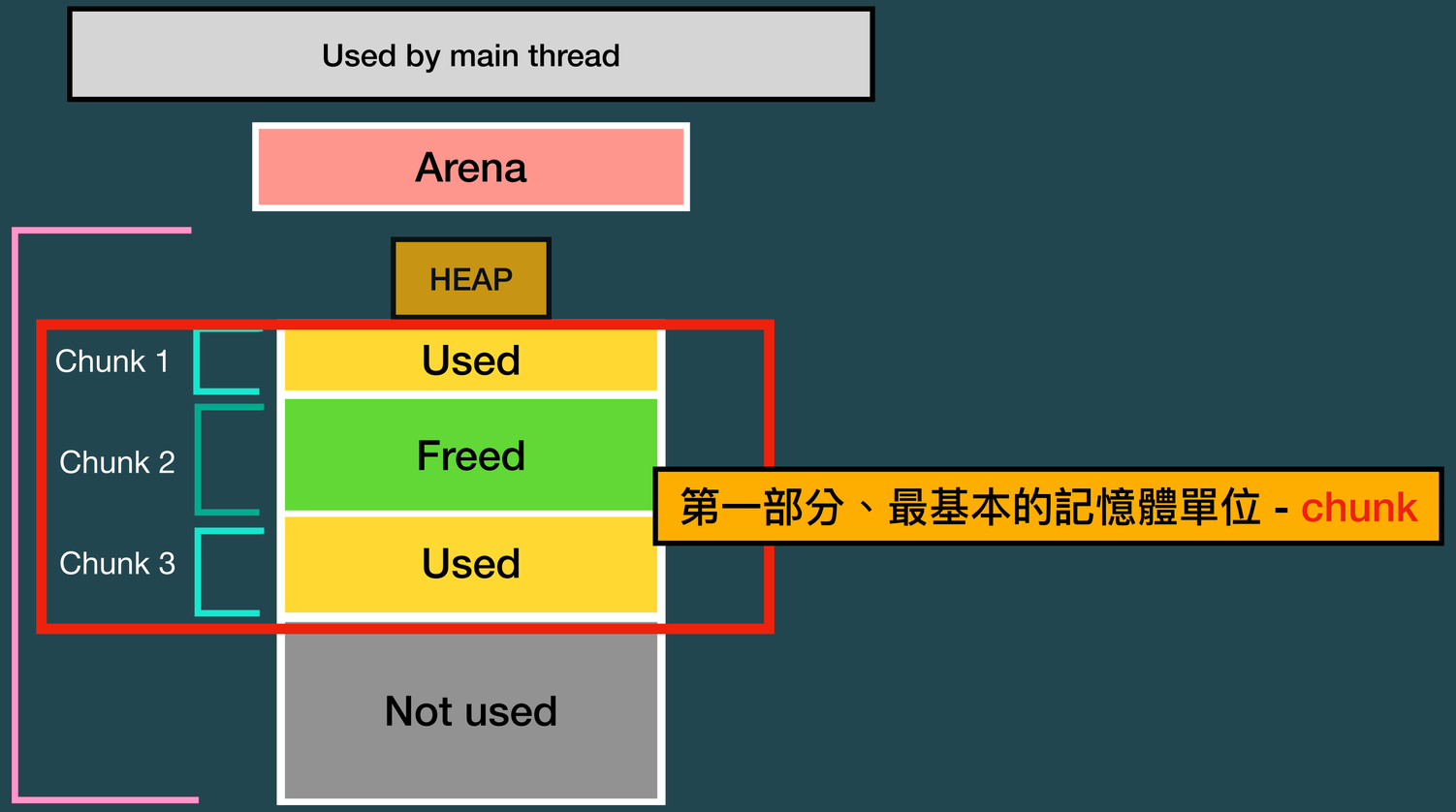
Data Structure



Thread 太多

$ Heap introduction

Data Structure



$ Heap introduction

Data Structure - Chunk

- **Chunk** 為動態記憶體分配的基本單位，最小為 0x20 bytes
- 會對齊 0x10 bytes，也就是大小會是 0x20, 0x30, 0x40 ...
- 正在使用中的 chunk 稱作 **allocated** chunk
- 已被釋放的 chunk 稱作 **freed** chunk
  - 在釋放時，chunk 會根據情況放在不一樣的回收區，稱作 **bin**
  - 後續再請求記憶體時，就會從這些 **bin** 取出 chunk 給使用者用
  - Chunk 的結構與在 bin 當中的 freed chunk 的連接方式有很大的關係

$ Heap introduction

Data Structure - Chunk

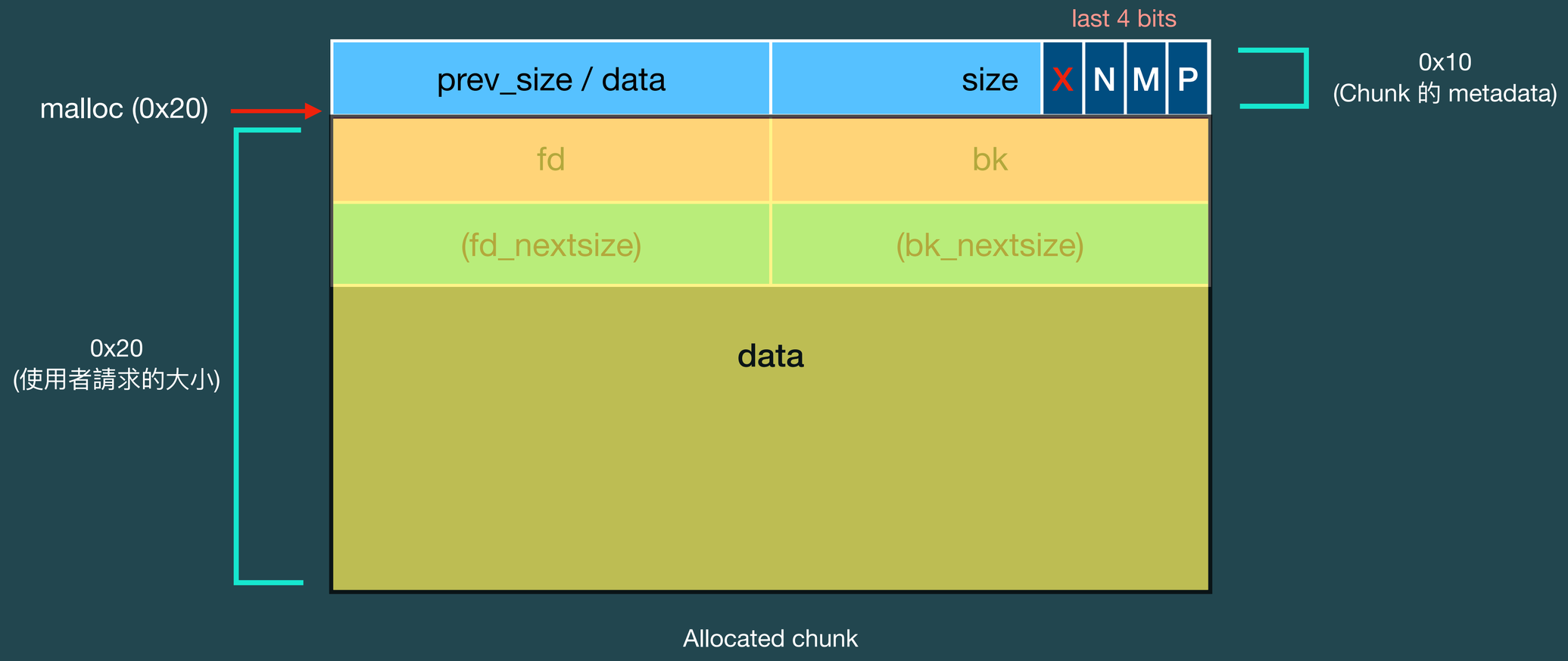
u1f383@u1f383:/ ⌥⌘1

```
$
struct malloc_chunk {

  INTERNAL_SIZE_T      mchunk_prev_size;  /* 如果前一個為 freed chunk，儲存其大小 */
  INTERNAL_SIZE_T      mchunk_size;       /* 儲存當前 chunk 的大小 */

  /* ———————————— 以下只有 freed chunk 才會使用 ———————————— */
  struct malloc_chunk* fd;         /* forward */
  struct malloc_chunk* bk;         /* back */

  /* —— 以下只有 large bin chunk 才會使用 —— */
  struct malloc_chunk* fd_nextsize;
  struct malloc_chunk* bk_nextsize;
};
```

https://elixir.bootlin.com/glibc/glibc-2.31/source/malloc/malloc.c#L1048

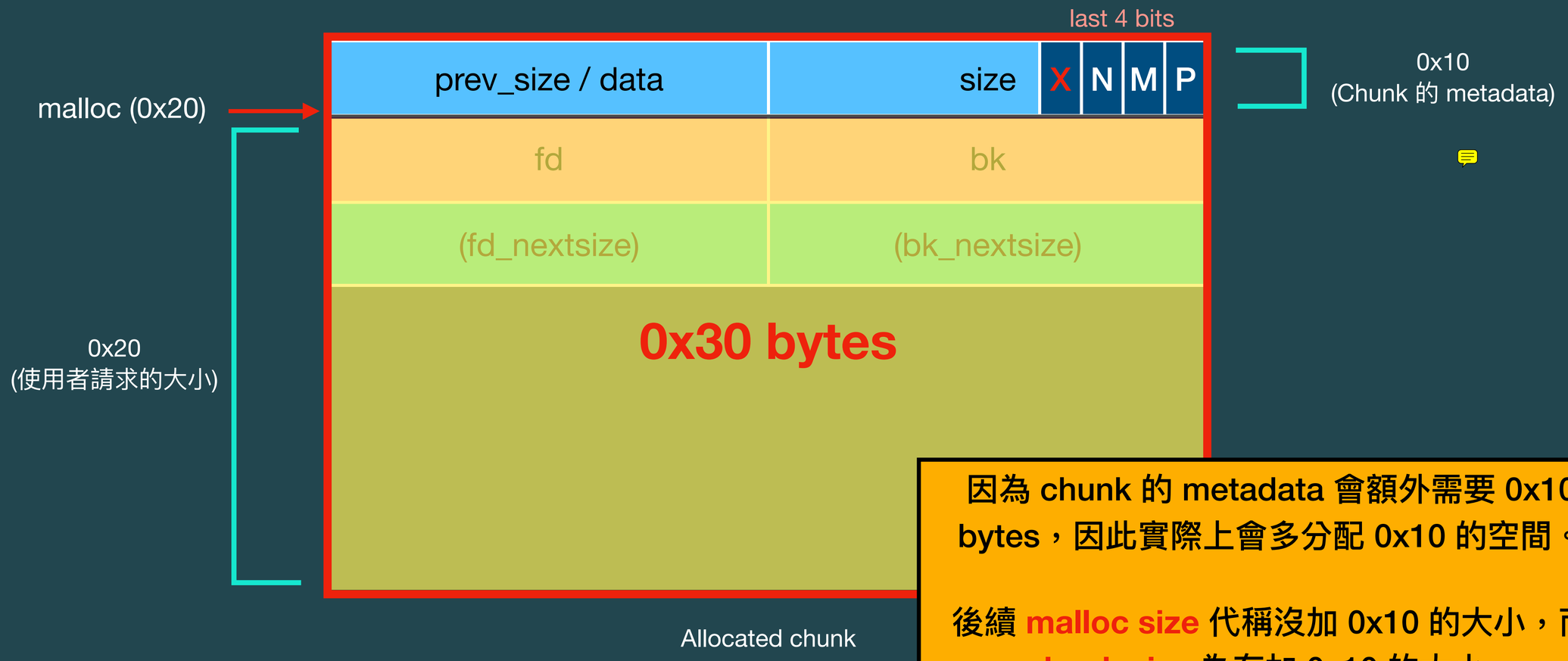
$ Heap introduction

Data Structure - Chunk



$ Heap introduction

Data Structure - Chunk

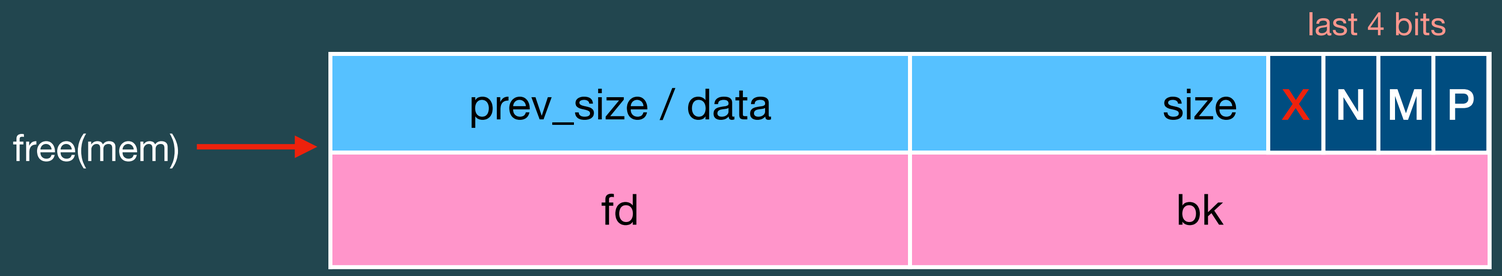


因為 chunk 的 metadata 會額外需要 0x10 bytes，因此實際上會多分配 0x10 的空間。

後續 **malloc size** 代稱沒加 0x10 的大小，而 **chunk size** 為有加 0x10 的大小。

$ Heap introduction

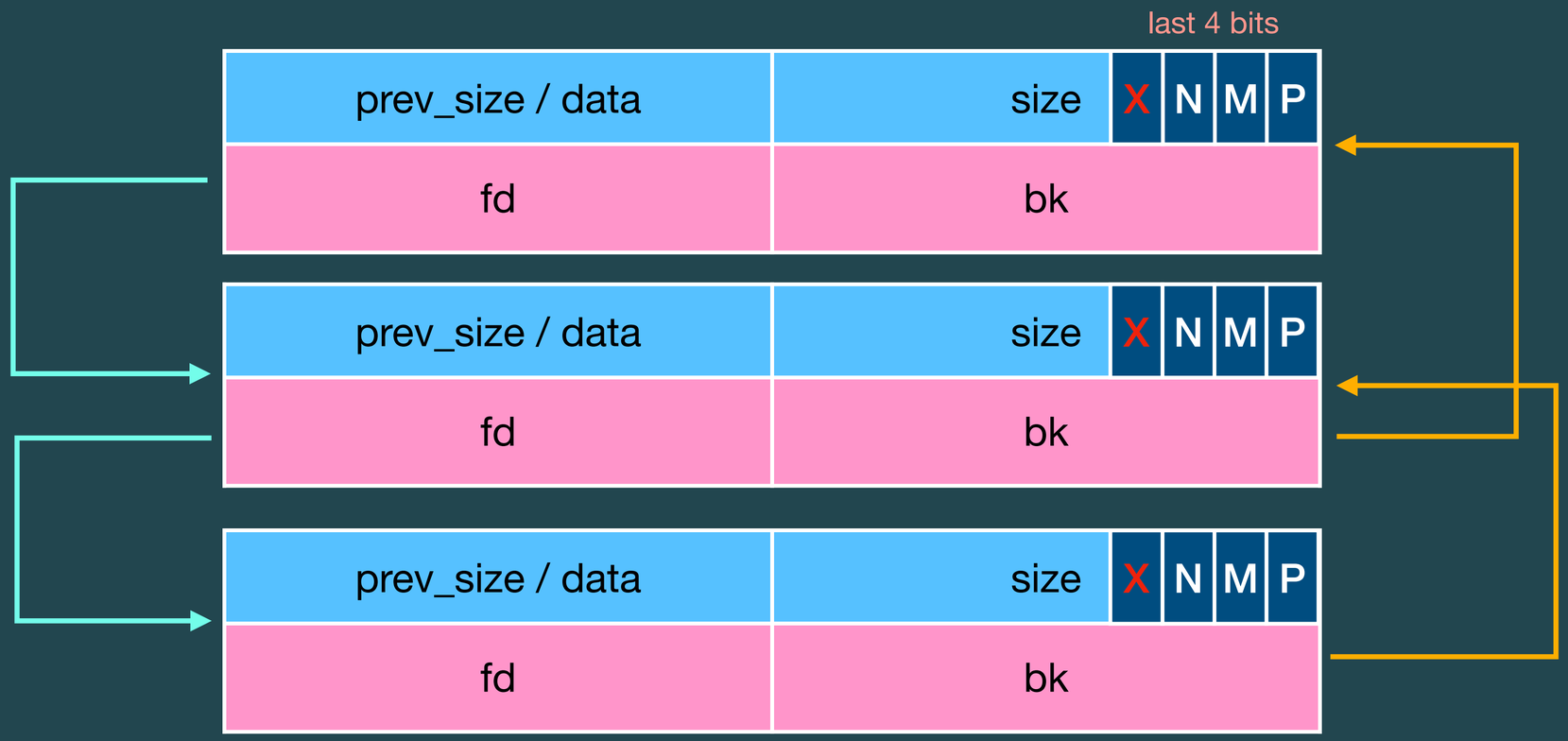
Data Structure - Chunk



freed chunk

$ Heap introduction

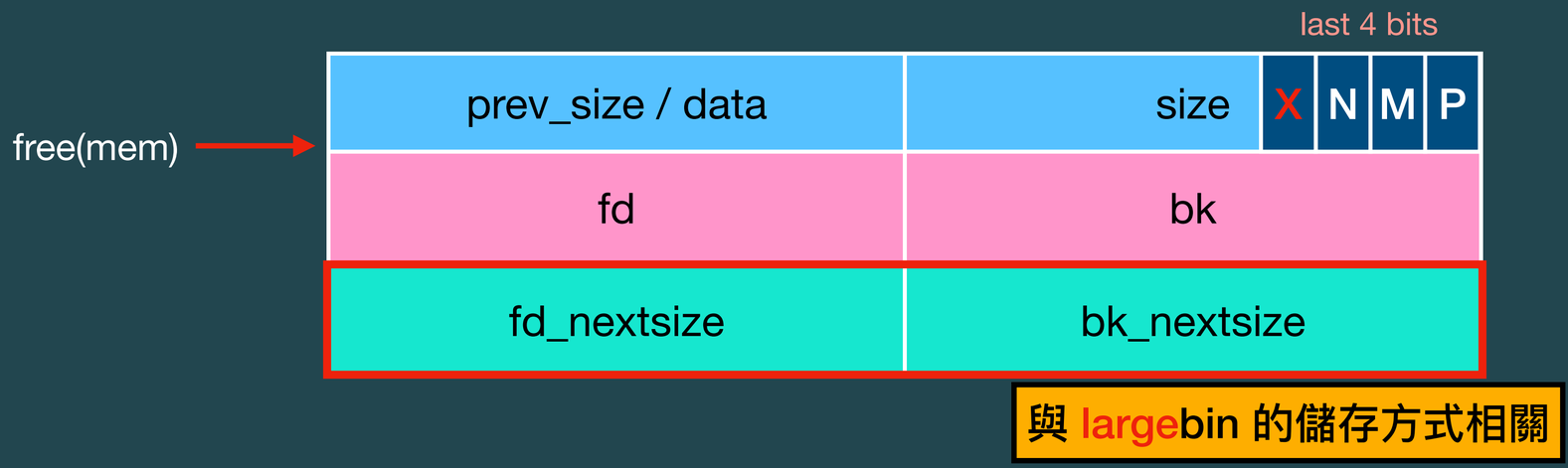
Data Structure - Chunk



freed chunk

$ Heap introduction

Data Structure - Chunk

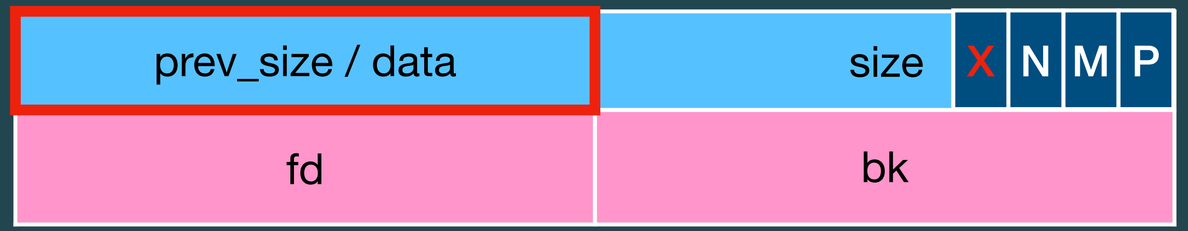


large bin chunk

$ Heap introduction

Data Structure - Chunk

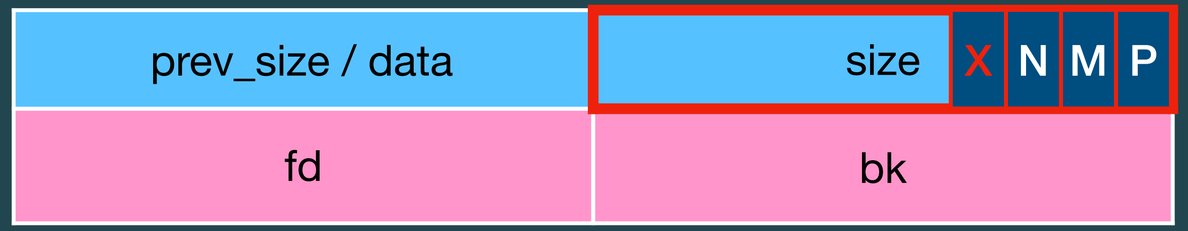
- **prev_size / data -** 看前一塊 chunk 是否正在使用
  - 如果前一塊為 **freed chunk -** 存前一塊 chunk 的 chunk size (prev_size)
  - 如果前一塊為 **allocated chunk -** 用來給前一塊 chunk 存資料 (data)
- 因為 chunk 可以多使用下個 chunk 的 data 欄位，因此在請求記憶體時能夠多分配 8 bytes
  - E.g. malloc size 為 0x28 時會拿到 chunk size 為 0x30 的 chunk —> 0x28 (request) - 0x8 (data) + 0x10 (header) —對齊0x10—> 0x30
  - 0x29 則會拿到 chunk size 0x40 的 chunk 0x31—對齊0x10—> 0x40



$ Heap introduction

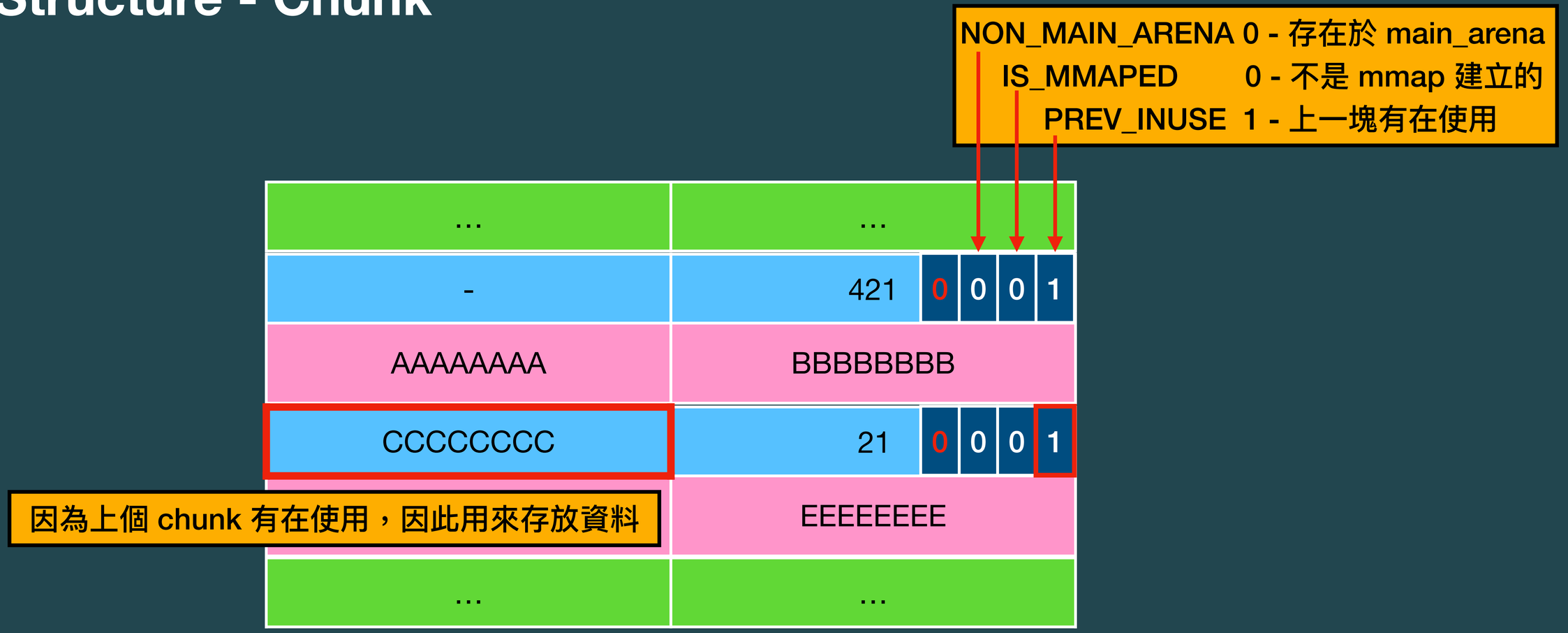
Data Structure - Chunk

- **size -** 存 chunk 的 chunk size，並且 size 會對齊 0x10 bytes (0b10000)
- 由於對齊，末四個 bits 會沒用到 (0b10000)，因此做為 chunk 的 metadata
  - **X -** 沒用
  - **N (NON_MAIN_ARENA) -** chunk 是否存在於 main_arena
  - **M (IS_MMAPPED) -** chunk 是否透過 mmap 建立的
  - **P (PREV_INUSE) -** 前一塊是否正在使用



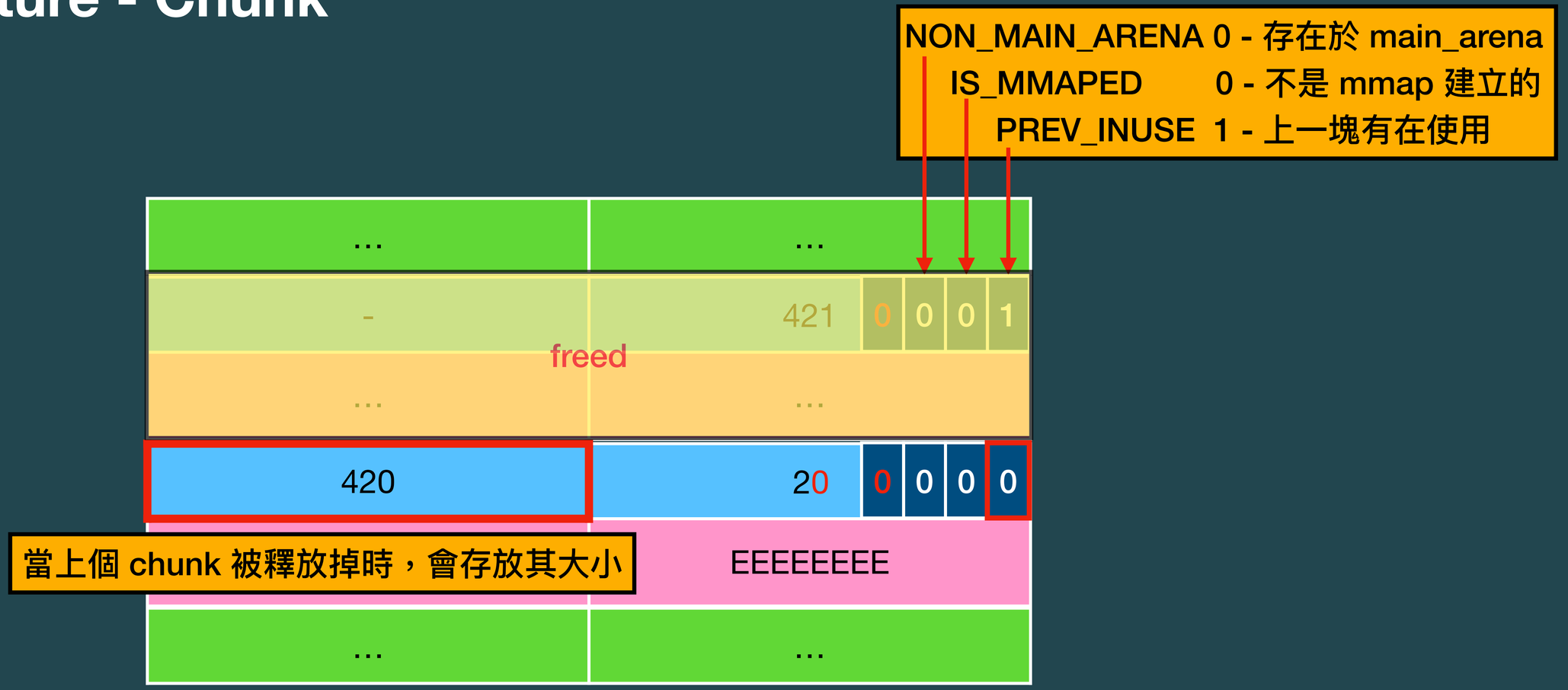
$ Heap introduction

Data Structure - Chunk



$ Heap introduction

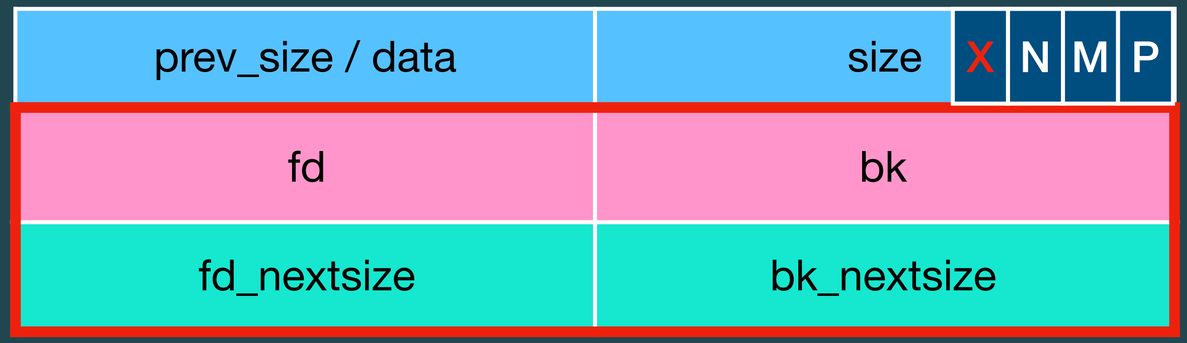
Data Structure - Chunk



$ Heap introduction

Data Structure - Chunk

- **fd -** 指向同個 bin 中的下一塊 freed chunk (forward)
- **bk -** 指向同個 bin 中的上一塊 freed chunk (back)
- **fd_nextsize -** 在 large bin 的 freed chunk 指向下一個大小的 freed chunk
- **bk_nextsize -** 在 large bin 的 freed chunk 指向上一個大小的 freed chunk



$ Heap introduction

Data Structure - Chunk

- bin 用來紀錄 freed chunk，而根據 chunk 的大小與 glibc 的優化機制，一共分成 5 種不同的 bin

名稱	Chunk size	使用方式	優先度	補充
Tcache bin	0x20 ~ 0x410	FILO (stack)	最高 (1)	被 free 後，並不會 unset 下個 chunk 的 PREV_INUSE bit
Fastbin	0x20 ~ 0x80	FILO (stack)	2	被 free 後，並不會 unset 下個 chunk 的 PREV_INUSE bit
Small bin	0x20 ~ 0x3f0	FIFO (queue)	3	-
Large bin	>= 0x400	FIFO (queue)	最低 (5)	-
Unsorted bin	>= 0x90	-	4	當對應大小的 tcache 滿時才會進來

$ Heap introduction

Data Structure - Chunk

- **Tcache (bin) -** 剛被釋放的 chunk 的快取空間
- **Fastbin -** 儲存大小較小的 chunk
- **Small bin -** 儲存大小中等的 chunk
- **Large bin -** 儲存大小較大的 chunk
- **Unsorted bin -** 剛被釋放的 chunk 的暫存區
- 在 tcache / fastbin / smallbin 當中，每隔 0x10 大小會有一個 subbin
  - E.g. fastbin 的 chunk size 為 0x20~0x80，因此會有 0x20、0x30、…、0x80 subbin

$ Heap introduction

Data Structure - Tcache

- **Tcache** 存放 chunk size 為 0x20 ~ 0x410 的 chunk，使用方式為 FILO
- 每個 subbin 最多只能存放 **7 個** chunk，彼此用單向的 linked list 相連
- 優先度最高，在對應大小的 tcache 滿後 (存放 7 個) 才會改用其他 bin

$ Heap introduction

Data Structure - Tcache

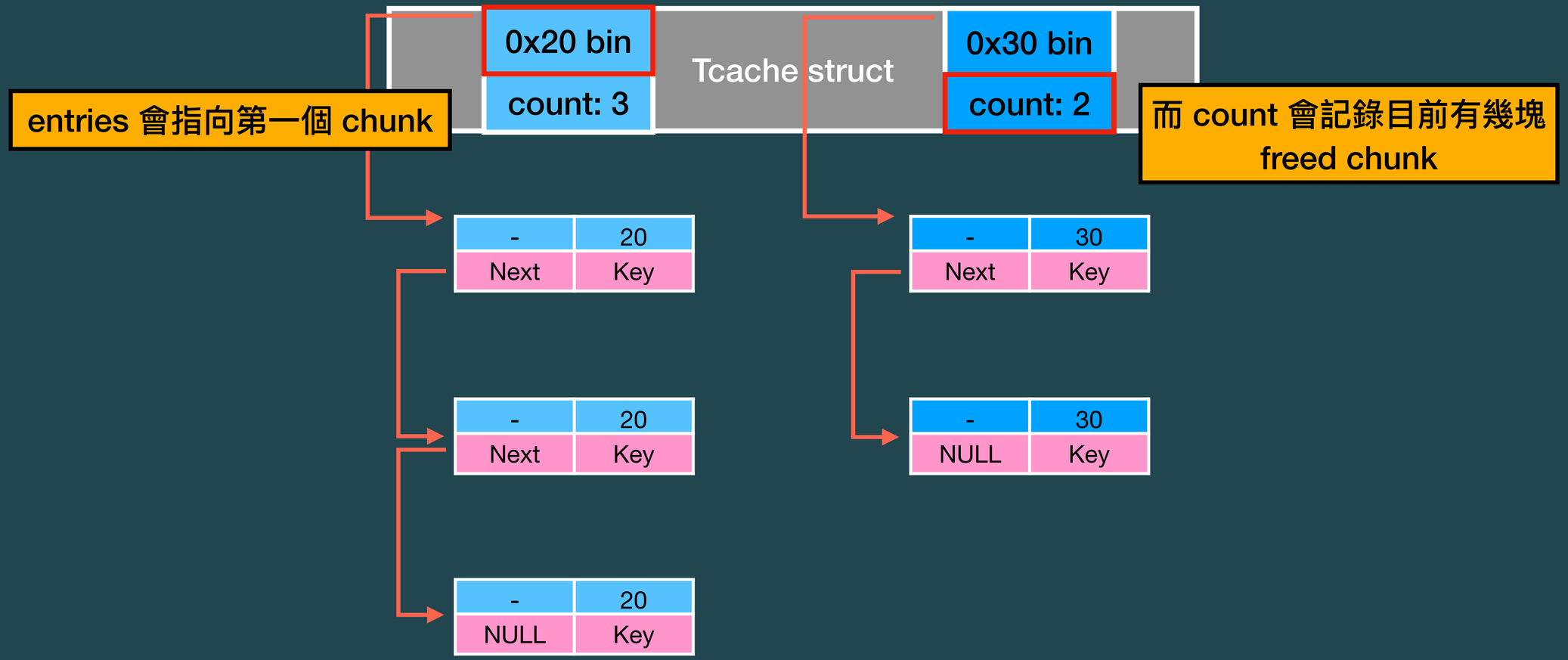
- struct tcache_entry 結構特別用來描述在 tcache 當中的 chunk
- struct tcache_perthread_struct 結構用來記錄 subbin 的第一塊 tcache chunk 位址，以及目前 subbin 共有多少個 tcache chunk
  - heap 初始化時就會被分配，大小為 0x290

```
u1f383@u1f383:/
$
typedef struct tcache_entry
{
  struct tcache_entry *next;
  struct tcache_perthread_struct *key;
} tcache_entry;
```

```
u1f383@u1f383:/
$
typedef struct tcache_perthread_struct
{
  /* TCACHE_MAX_BINS 大小為 64 */
  uint16_t counts[TCACHE_MAX_BINS];
  tcache_entry *entries[TCACHE_MAX_BINS];
} tcache_perthread_struct;
```

$ Heap introduction

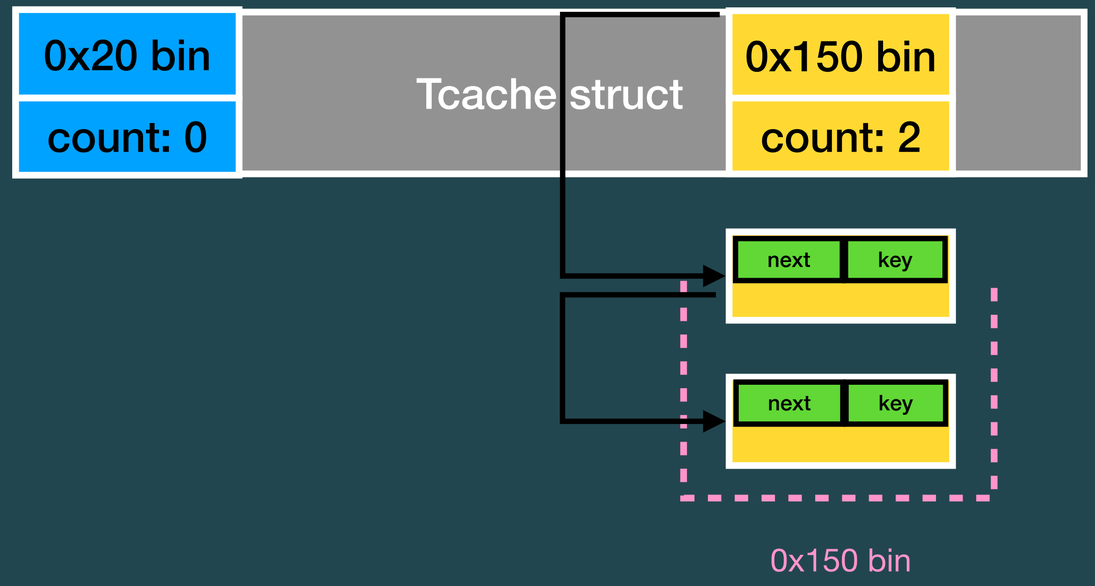
Data Structure - Tcache



$ Heap introduction

Data Structure - Tcache

```
a = malloc(0x10);
b = malloc(0x10);
c = malloc(0x10);
d = malloc(0x10);

free(a);
malloc(0x10);
free(b);
free(c);
free(d);
```

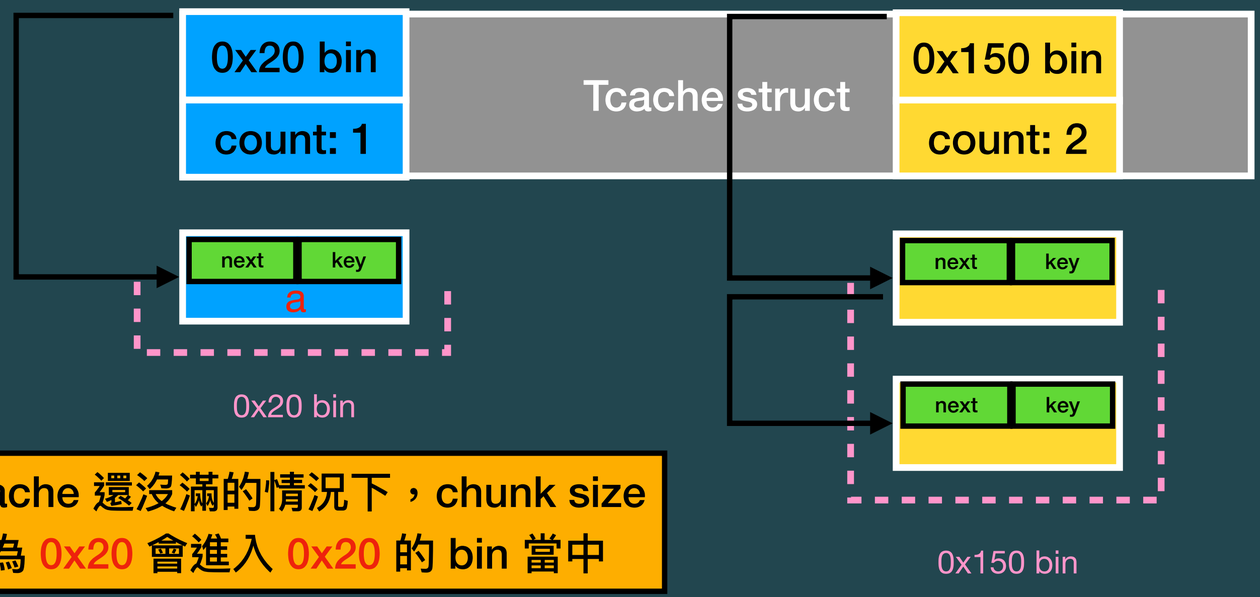


Tcache instance

$ Heap introduction

Data Structure - Tcache

```
a = malloc(0x10);
b = malloc(0x10);
c = malloc(0x10);
d = malloc(0x10);

free(a);
malloc(0x10);
free(b);
free(c);
free(d);
```



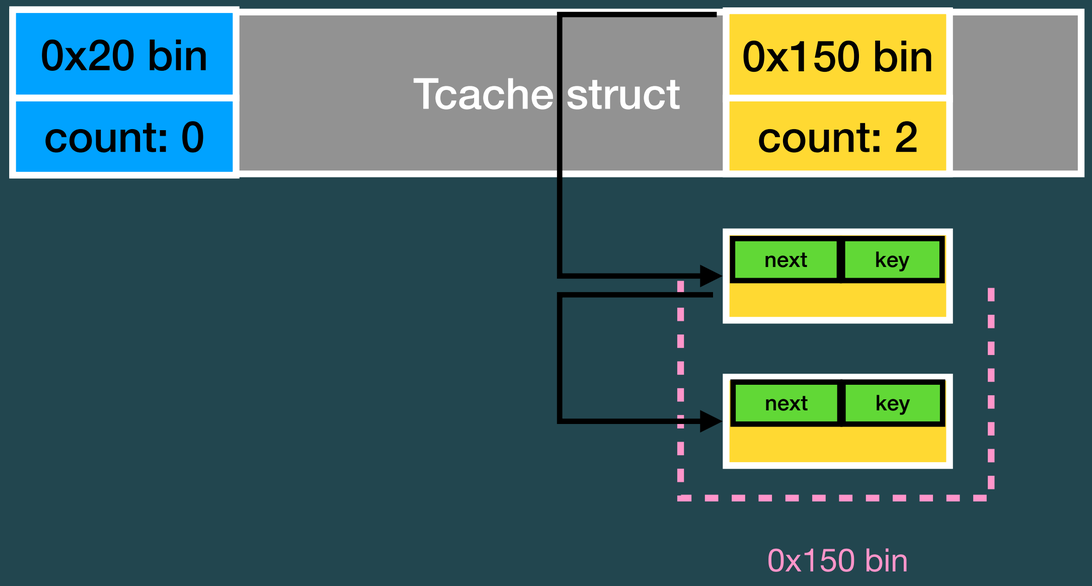
tcache 還沒滿的情況下，chunk size 為 0x20 會進入 0x20 的 bin 當中

Tcache instance

$ Heap introduction

Data Structure - Tcache

```
a = malloc(0x10);
b = malloc(0x10);
c = malloc(0x10);
d = malloc(0x10);

free(a);
malloc(0x10);
free(b);
free(c);
free(d);
```

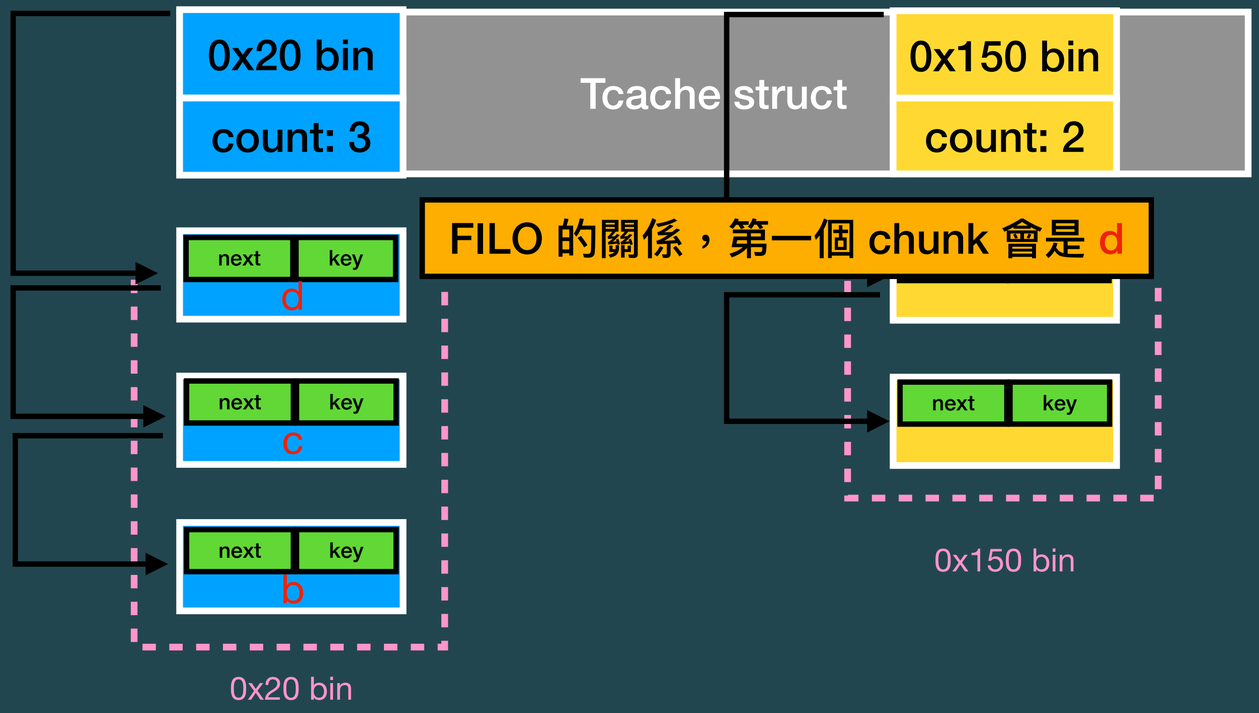


Tcache instance

$ Heap introduction

Data Structure - Tcache

```
a = malloc(0x10);
b = malloc(0x10);
c = malloc(0x10);
d = malloc(0x10);

free(a);
malloc(0x10);
free(b);
free(c);
free(d);
```

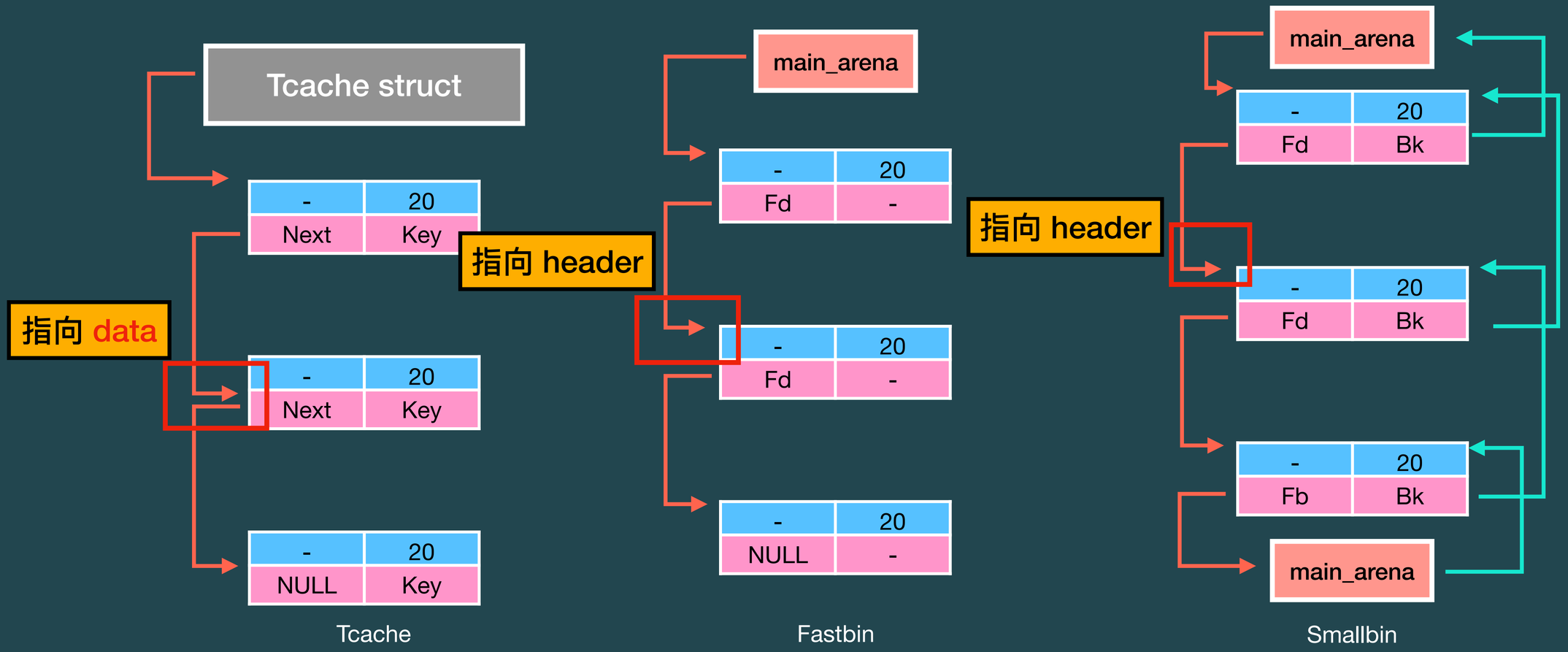


Tcache instance

$ Heap introduction

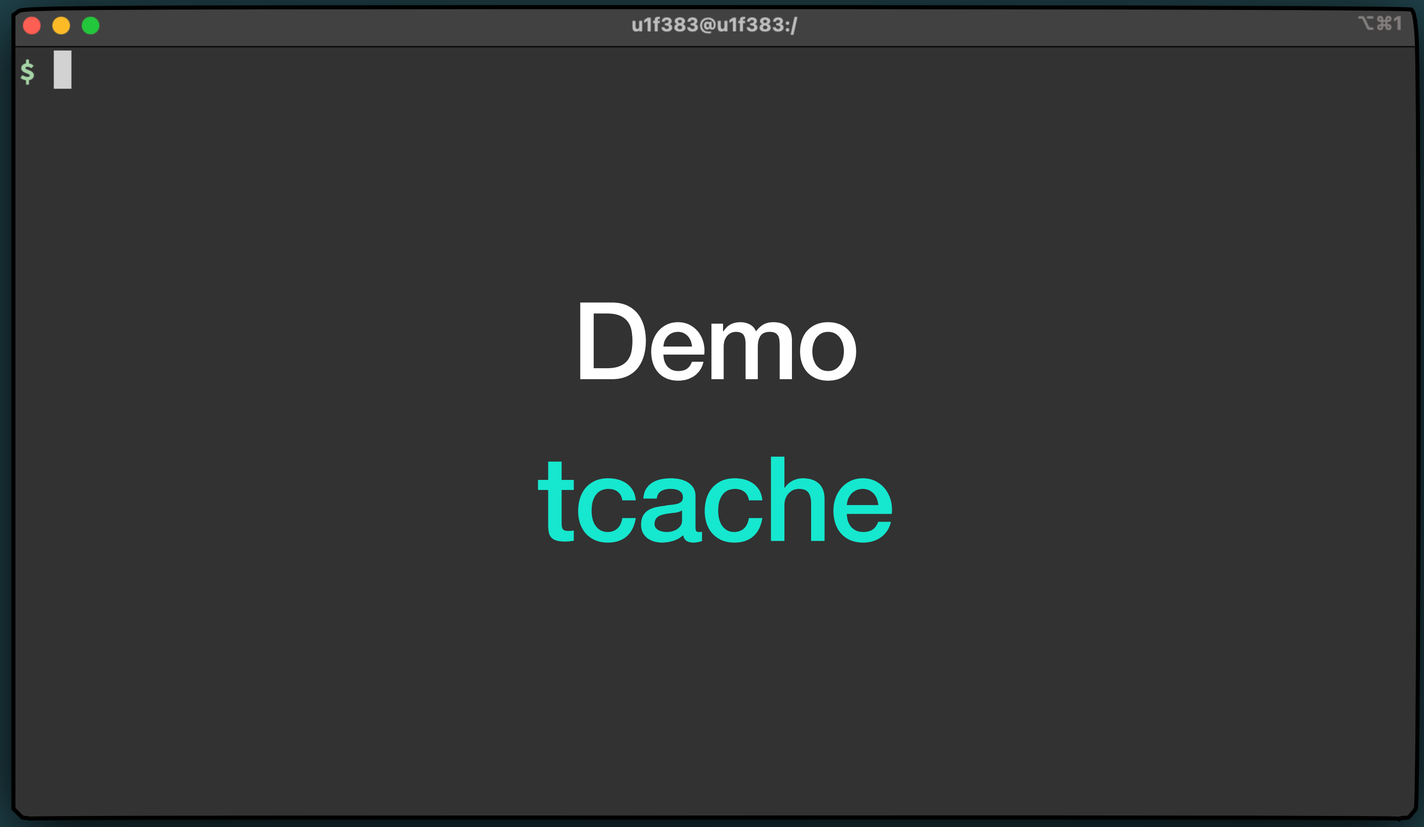
Data Structure - Tcache

- Tcache 的 fd 不像其他種類的 chunk 指向 header，而是指向 data



$ Heap introduction

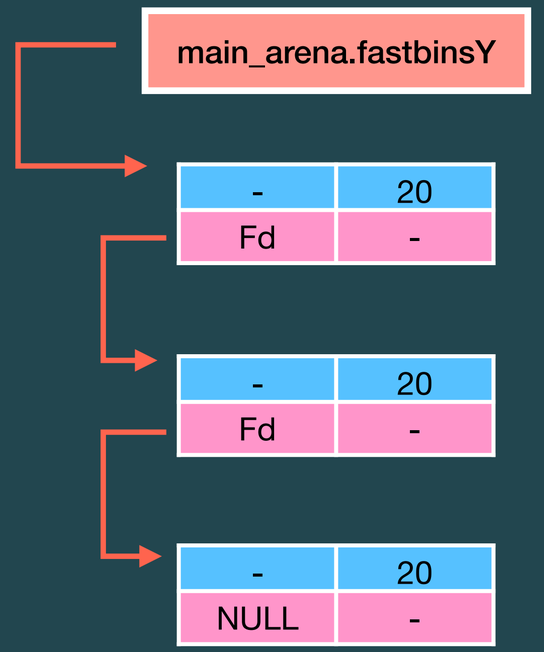
Data Structure - Tcache



$ Heap introduction

Data Structure - Fastbin

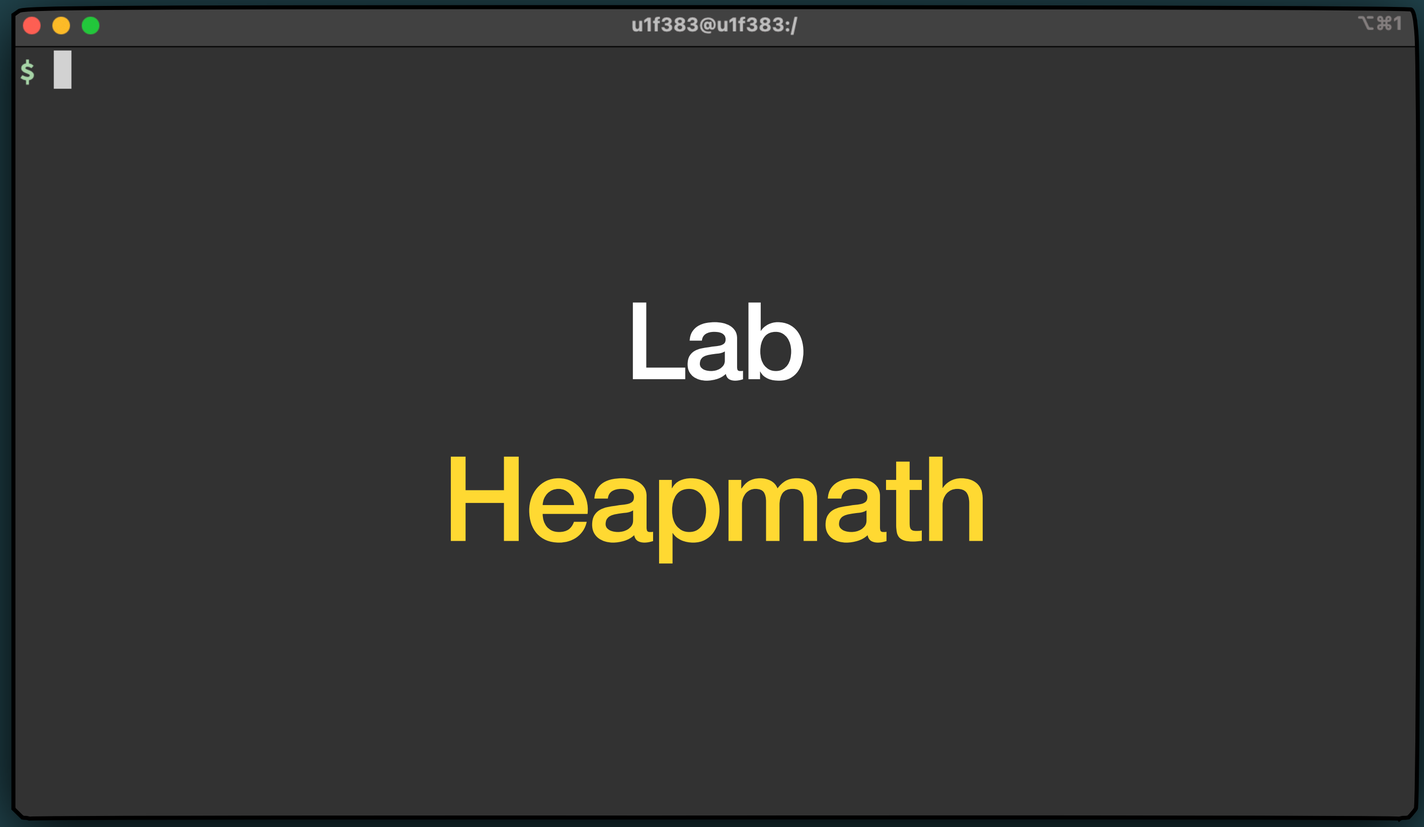
- **Fastbin** 存放 chunk size 為 0x20 ~ 0x80 的 chunk，並且數量沒有限制
- 使用方式與 tcache 相同，為 FILO
- 當 tcache 滿時，如果 free 大小為 0x20 ~ 0x80 的 chunk，則會直接進入 fastbin



$ Heap introduction

Data Structure - Fastbin

$ Heap introduction



$ Heap introduction

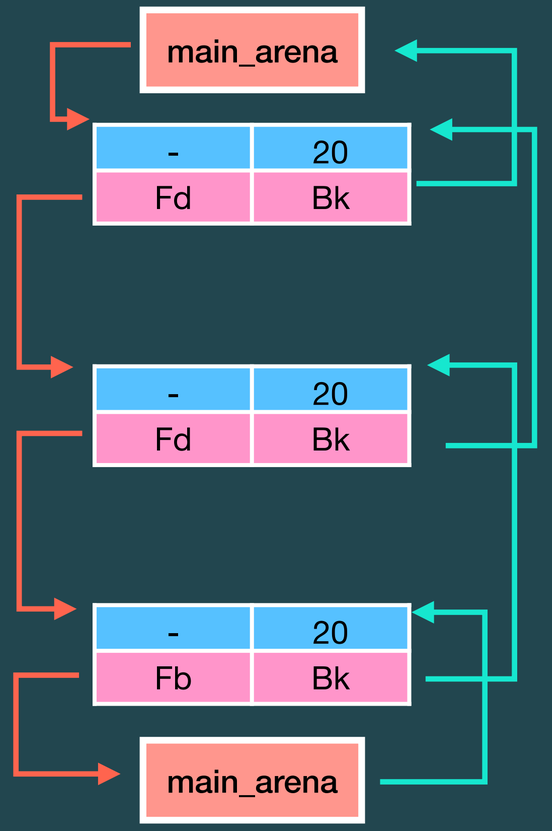
Lab - Heapmath

- 了解 tcache、fastbin bin 的連接方式是非常基本的
- 此 lab 會需要回答：
  - Freed chunk 在 tcache 內的連接順序
  - 一組 chunk 的距離運算
  - Tcache fd 的計算
  - Fastbin fd 的計算

$ Heap introduction

Data Structure - Small bin

- **Smallbin** 存放 chunk size 為 0x20 ~ 0x3f0 的 chunk
- 使用方式為 FIFO
- 其中 0x20 ~ 0x80 的大小與 fastbin 重疊
  - 直接 free 的 chunk 會進 fastbin
  - 從 unsorted bin 放回去時會進 smallbin



Smallbin

$ Heap introduction

Data Structure - Small bin

$ Heap introduction

Data Structure - Large bin

- **Largebin** 存放 chunk size >= 0x400 的 chunk，使用方式為 FIFO
- 每個 subbin 都存放多種不同 size 的 chunk
  - 32 bins — +0x40
    - 0x400, 0x410, 0x420, 0x430 ➔ 0-bin
    - 0x440, 0x450, 0x460, 0x470 ➔ 1-bin
    - ...

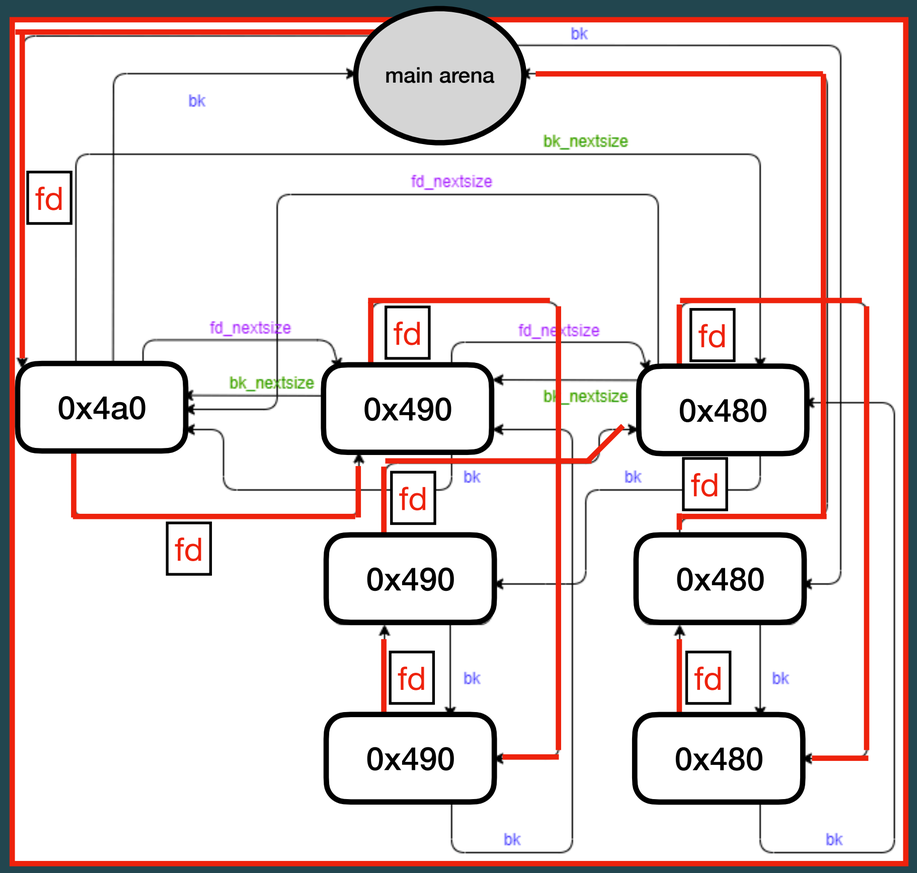
$ Heap introduction

Data Structure - Large bin

- 每個 subbin 都存放多種不同 size 的 chunk (cont.)
  - 16 bins — +0x200
  - 8 bins — +0x1000
  - 4 bins — +0x8000
  - 2 bins — +0x40000
  - 1 bins — the rest
- 使用 fd_nextsize 與 bk_nextsize 將不同 size 的 chunk 串起來

$ Heap introduction

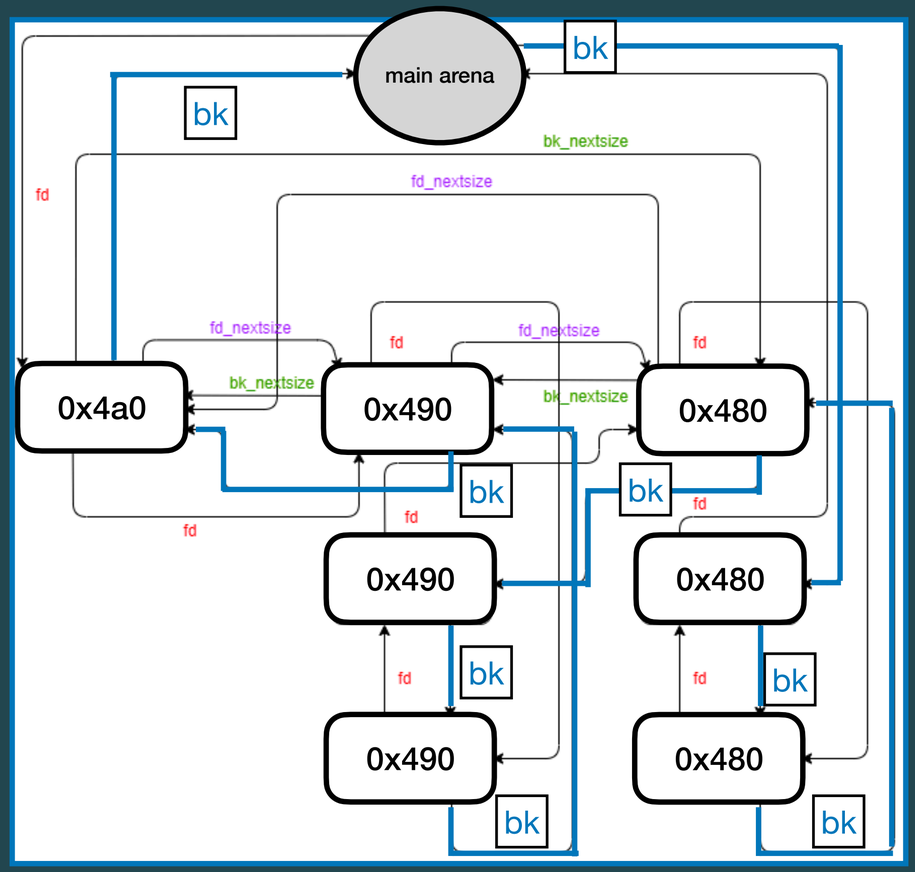
Data Structure - Large bin



2-bin

$ Heap introduction

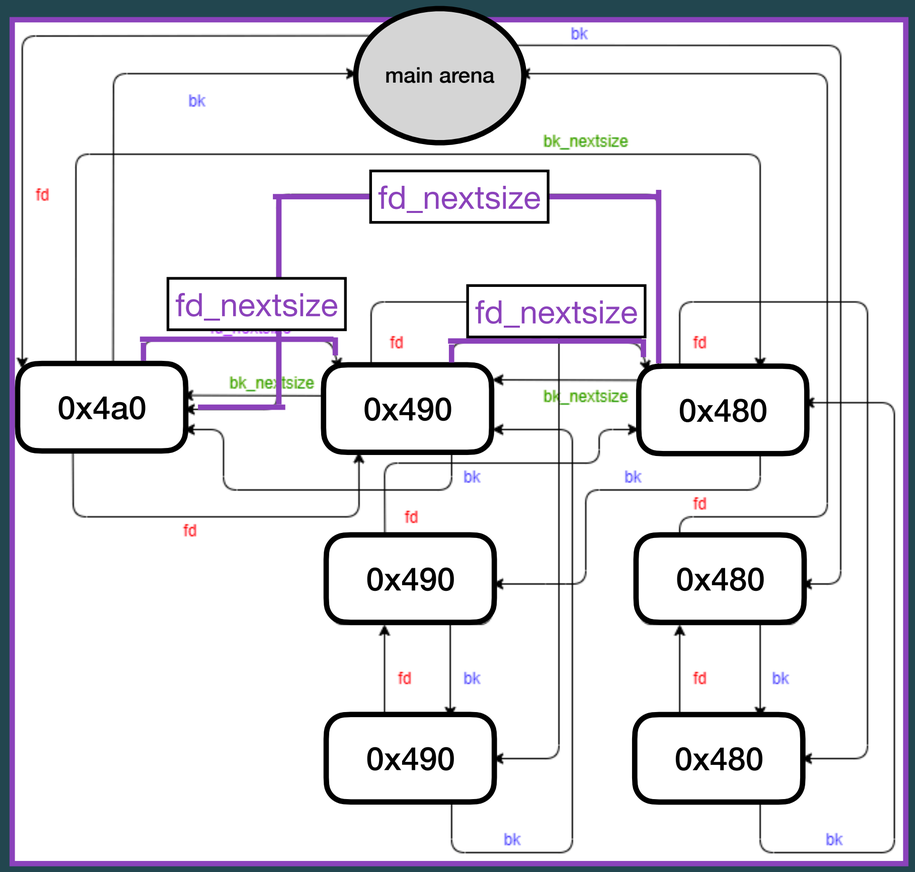
Data Structure - Large bin



2-bin

$ Heap introduction

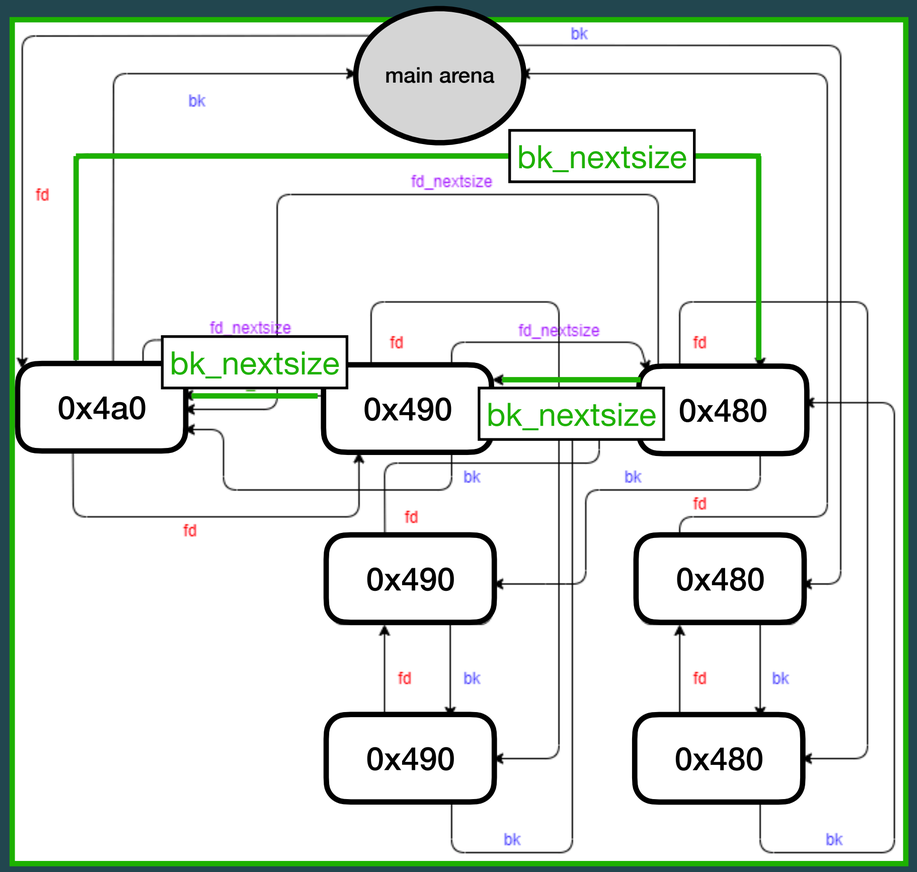
Data Structure - Large bin



2-bin

$ Heap introduction

Data Structure - Large bin



2-bin

$ Heap introduction

Data Structure - Large bin

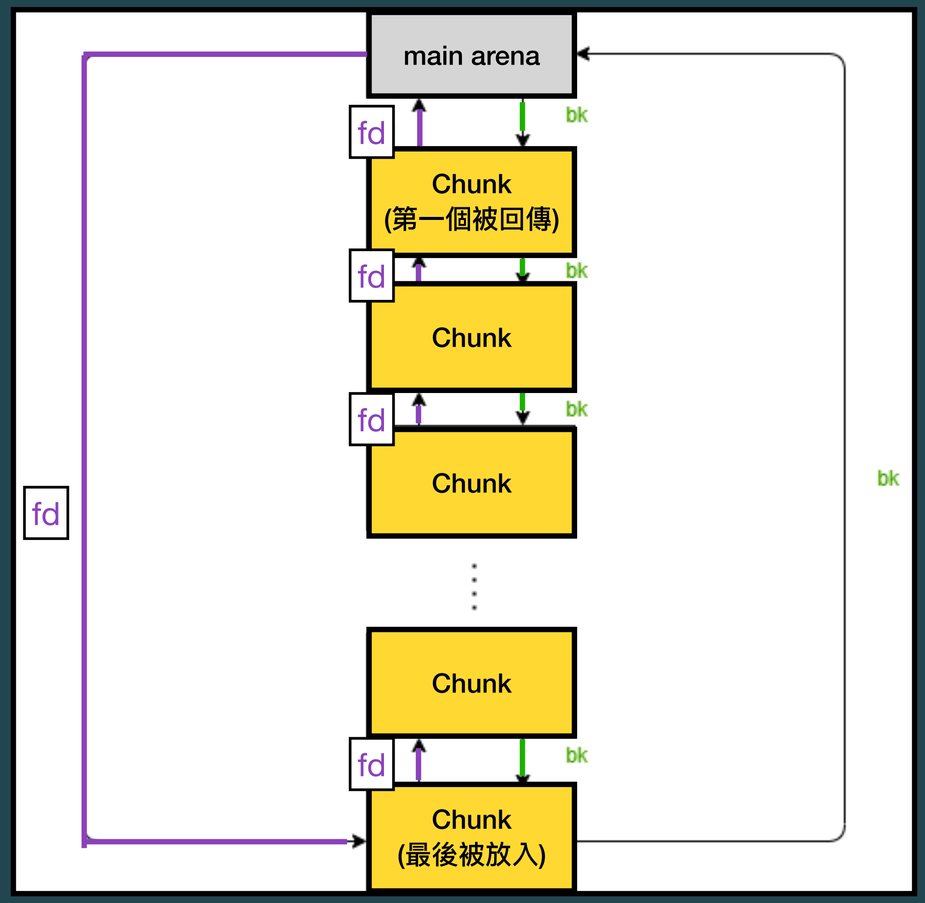
$ Heap introduction

Data Structure - Unsorted bin

- **Unsorted bin -** 對應大小的 tcache 已經滿的情況下，並且 chunk size 非 fastbin chunk 的大小，則此 chunk 被 free 後會暫存在 unsorted bin
  - 當收到請求後，如果對應大小的 tcache、fastbin、smallbin 為空，則會遍歷整個 unsorted bin
  - 若 unsorted bin 中有 chunk 的大小 >= (請求的大小 + 0x20)，會直接從該塊 chunk 切下來回傳
    > 0x20 為最小的 chunk size
  - 若沒有，則會將這些暫存的 chunk 放至對應的 bin (small bin, large bin, ...)

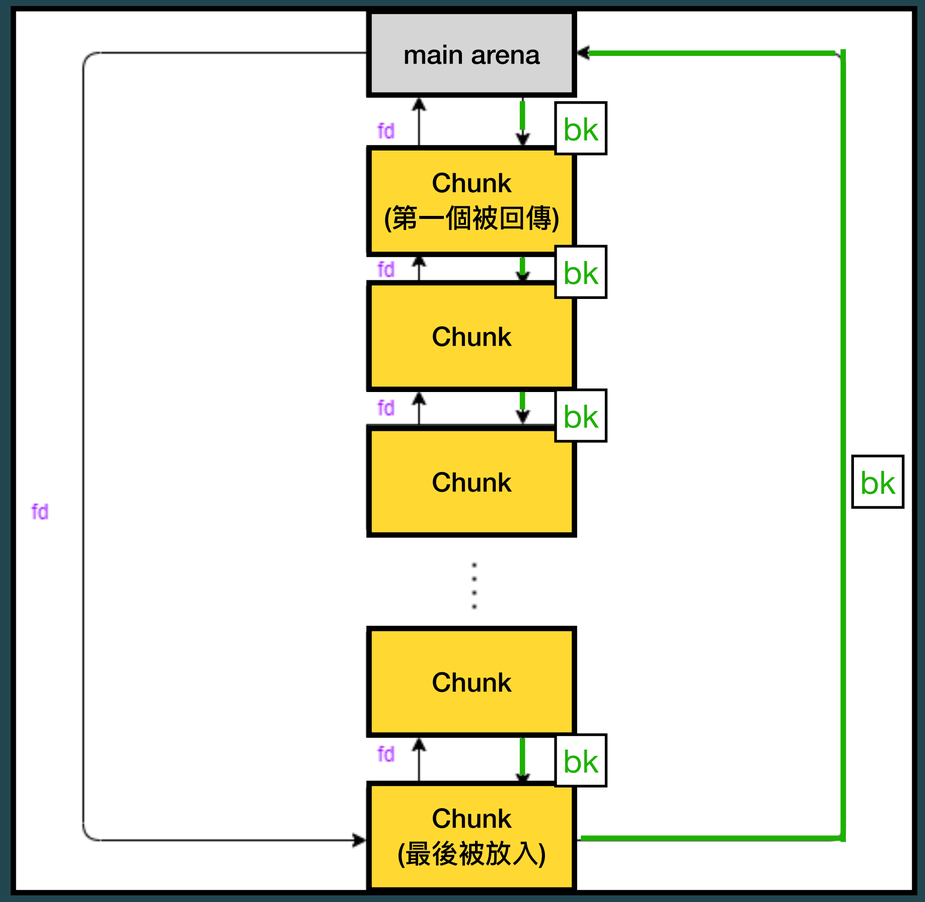
$ Heap introduction

Data Structure - Unsorted bin



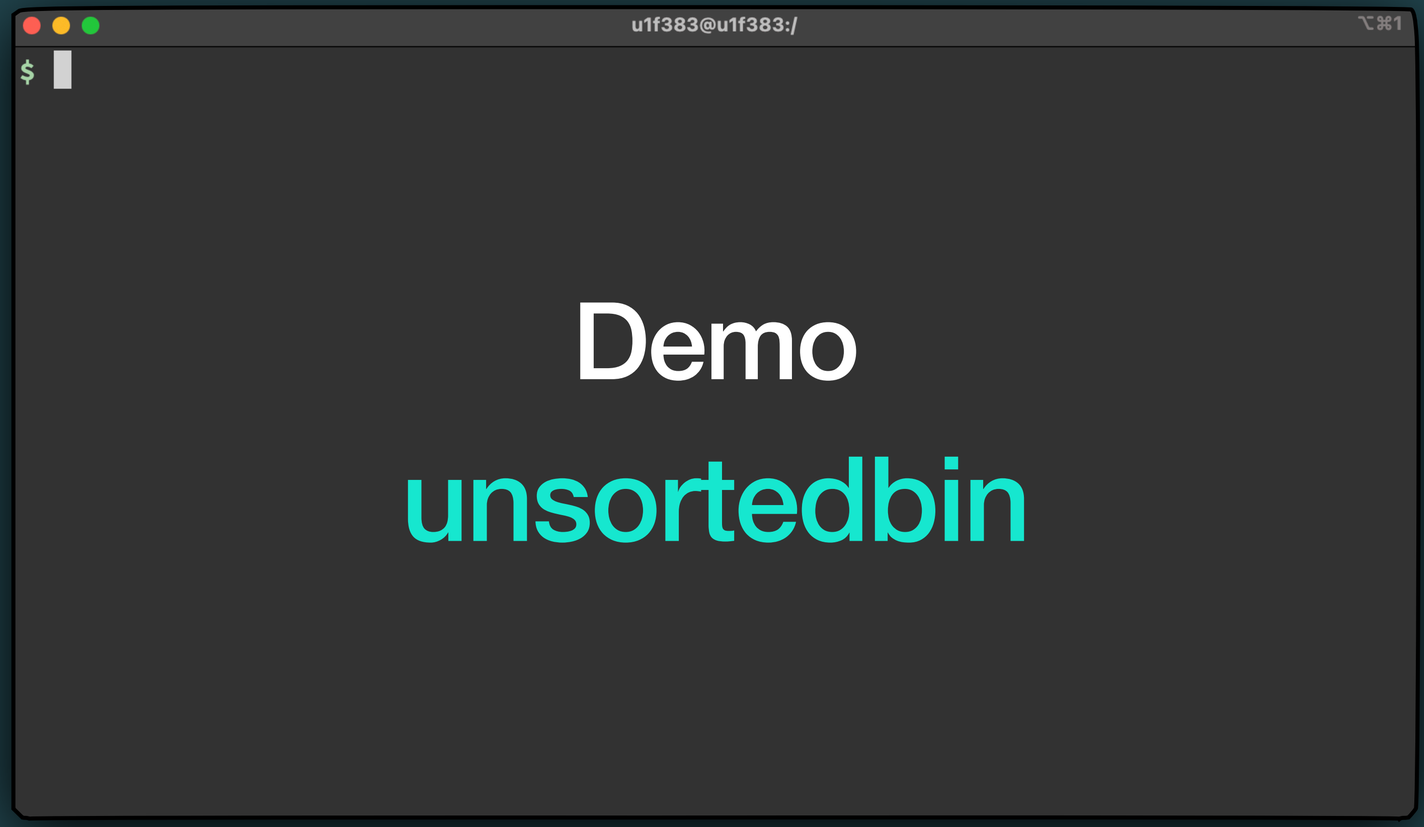
$ Heap introduction

Data Structure - Unsorted bin



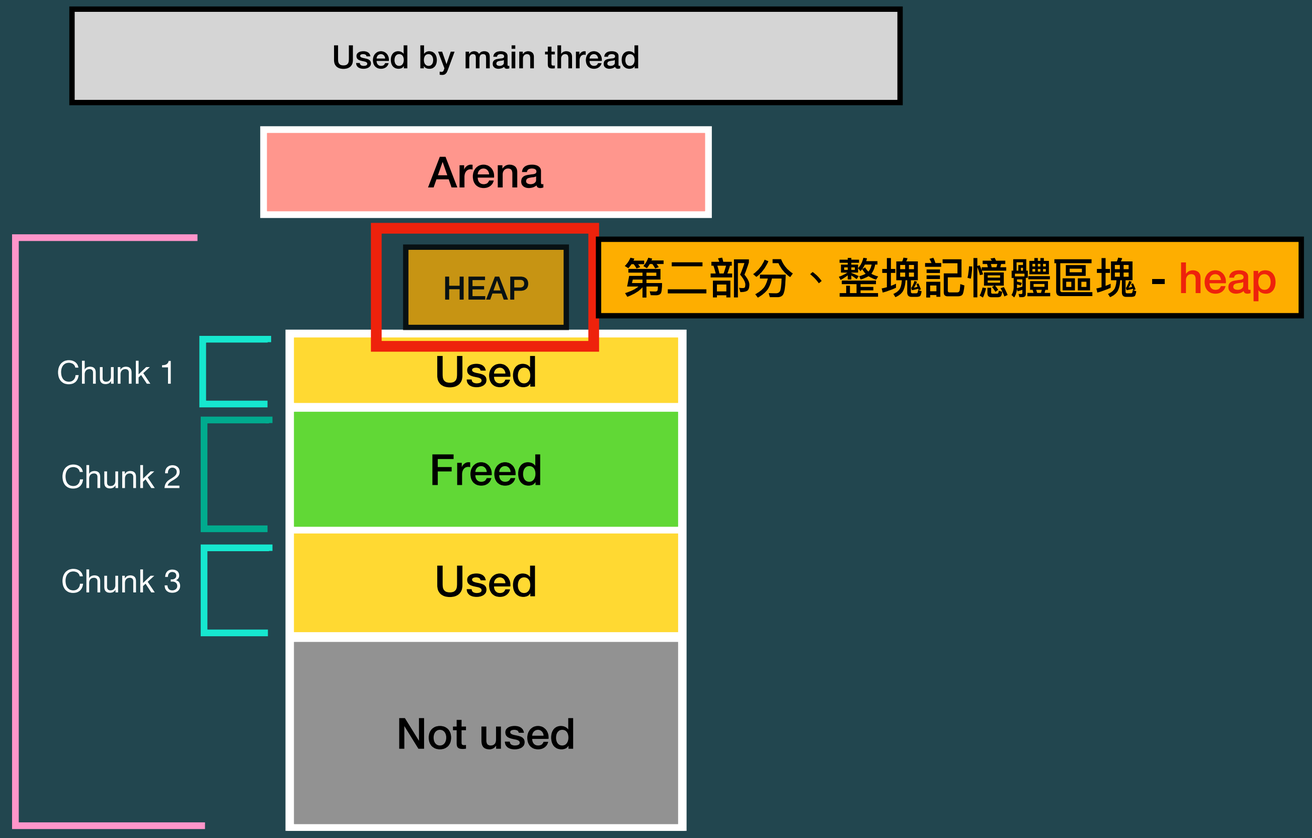
$ Heap introduction

Data Structure - Unsorted bin



$ Heap introduction

Data Structure - Heap



$ Heap introduction

Data Structure - Heap

u1f383@u1f383:/ ⌥⌘1

```
$
```

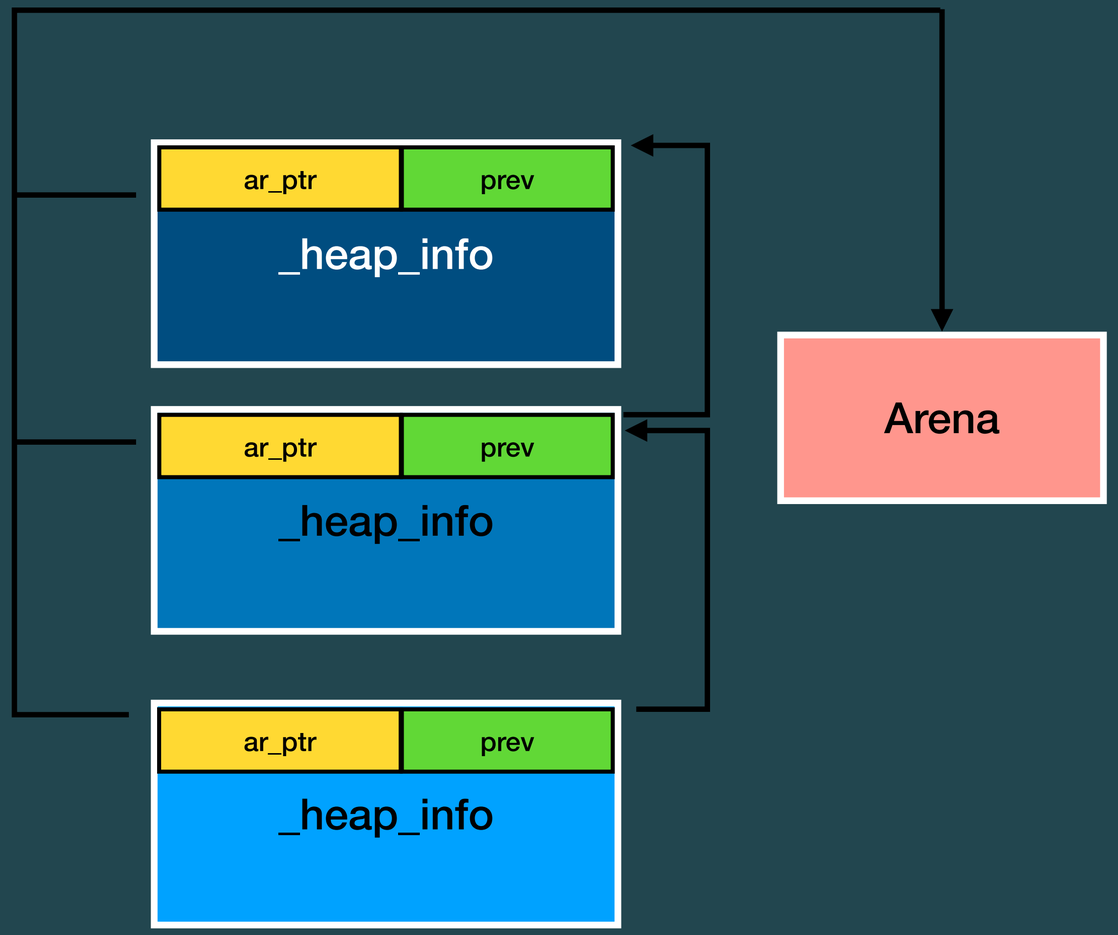
```c
typedef struct _heap_info
{
  mstate ar_ptr;             /* 指向當前 heap 使用的 arena */
  struct _heap_info *prev; /* 指向上一塊 heap */
  size_t size;               /* 當前 heap 的大小 */
  size_t mprotect_size;    /* prot 為 RROT_READ|PROT_WRITE 的空間大小 */

  /* 用來做 alignment */
  char pad[-6 * SIZE_SZ & MALLOC_ALIGN_MASK];
} heap_info;
```

https://elixir.bootlin.com/glibc/glibc-2.31/source/malloc/arena.c#L53

$ Heap introduction

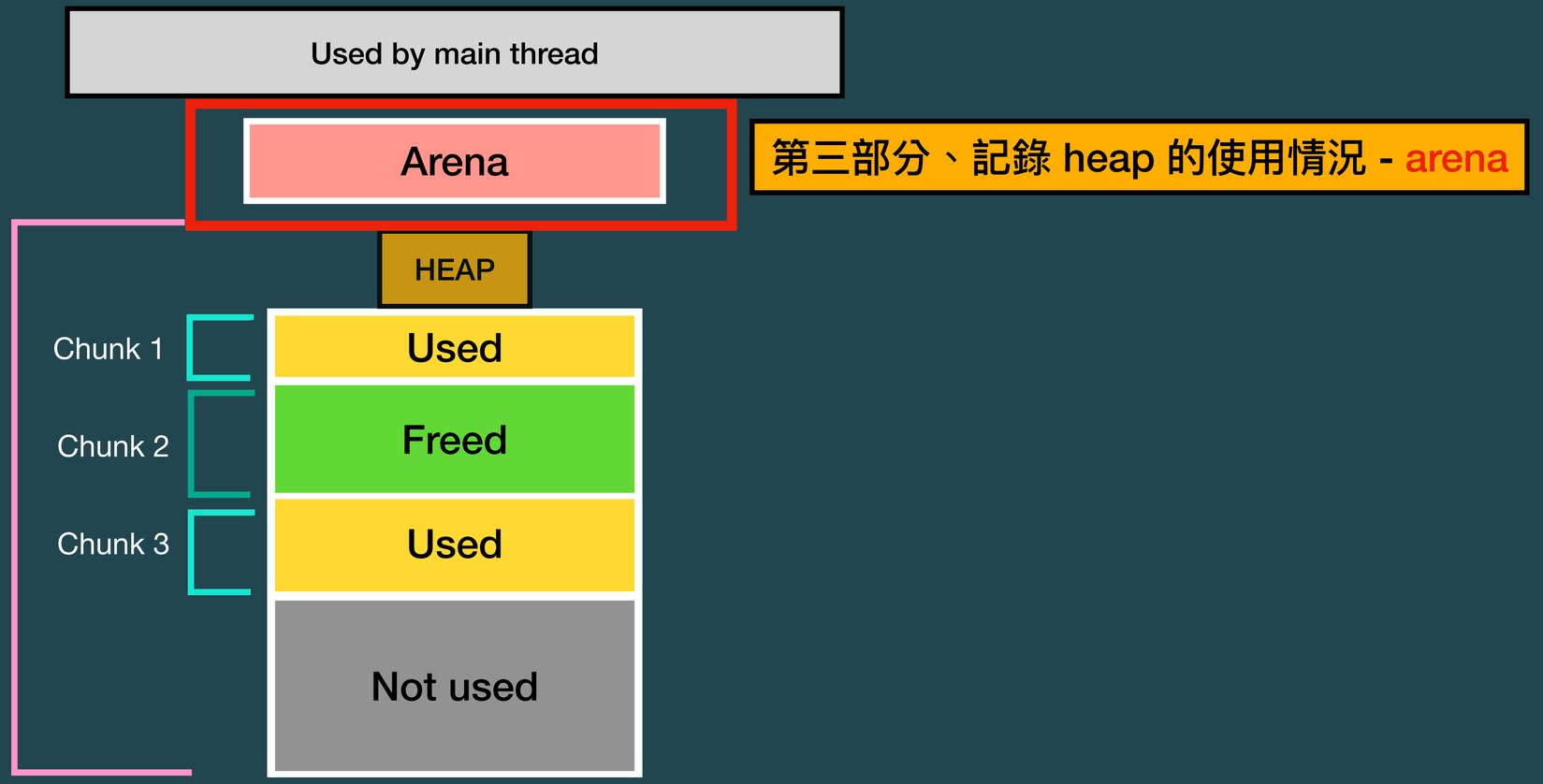
Data Structure - Heap

- Size 與 mprotect_size 預設為 0x21000
- Main thread 本身是沒有 _heap_info，
  因此 heap exploit 並不常用到 _heap_info



$ Heap introduction

Data Structure - Arena



$ Heap introduction

Data Structure - Arena

u1f383@u1f383:/ ⌥⌘1

```
$
```

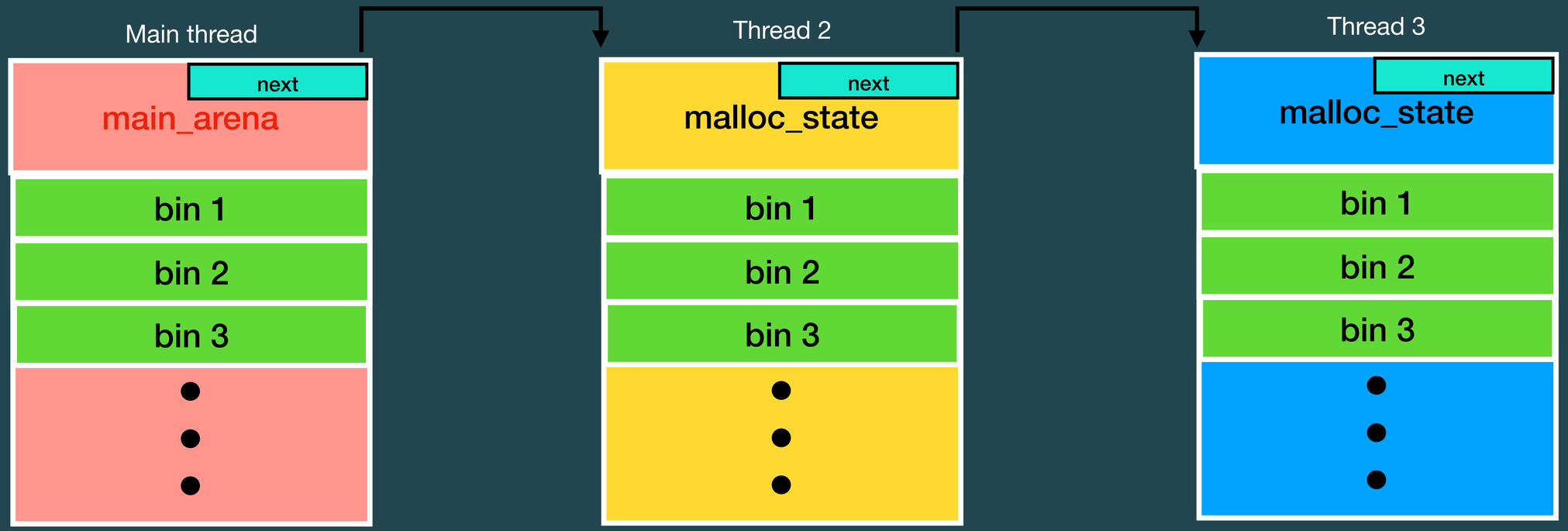
```c
struct malloc_state /* 也稱作 arena */
{
  /* fastbins 陣列，NFASTBINS 為 10 */
  mfastbinptr fastbinsY[NFASTBINS];
  /* 指向 top chunk 的位址 */
  mchunkptr top;
  /* 指向最近被切的 chunk 所剩下的 chunk */
  mchunkptr last_remainder;
  /* small bin, large bin 還有 unsorted bin 都在這 */
  mchunkptr bins[NBINS * 2 - 2];


  /* 以下 member 相較來說不是這麼重要 */
  struct malloc_state *next;
  struct malloc_state *next_free;
  INTERNAL_SIZE_T attached_threads;
  INTERNAL_SIZE_T system_mem;
  INTERNAL_SIZE_T max_system_mem;
};
```


https://elixir.bootlin.com/glibc/glibc-2.31/source/malloc/malloc.c#L1655

$ Heap introduction

Data Structure - Arena

- Main thread 的 malloc_state 也稱作 **main_arena**
- 由於後續只考慮一個 thread 的情況，基本上只會使用到 **main_arena**



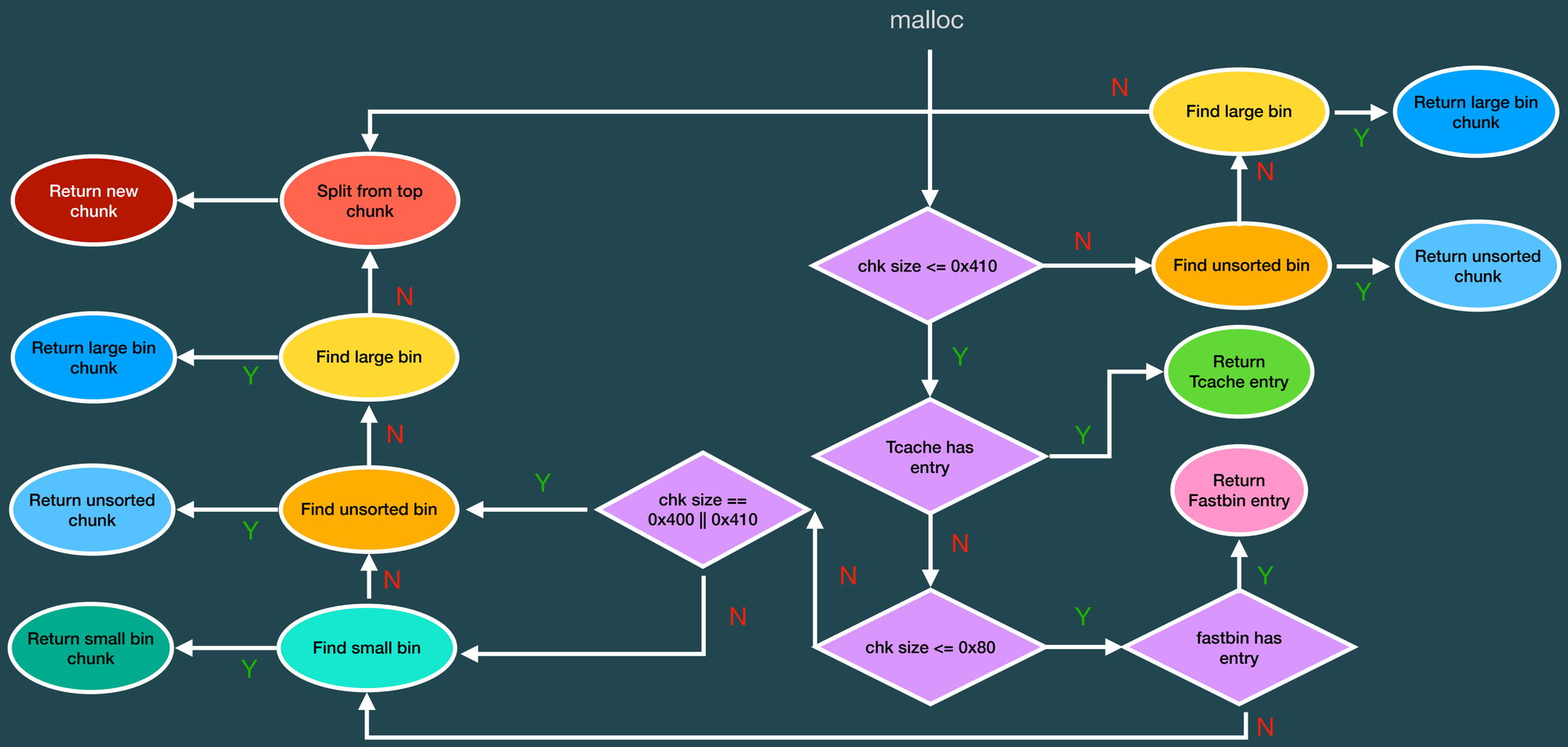
$ Heap introduction

Data Structure - Arena

Code tracing

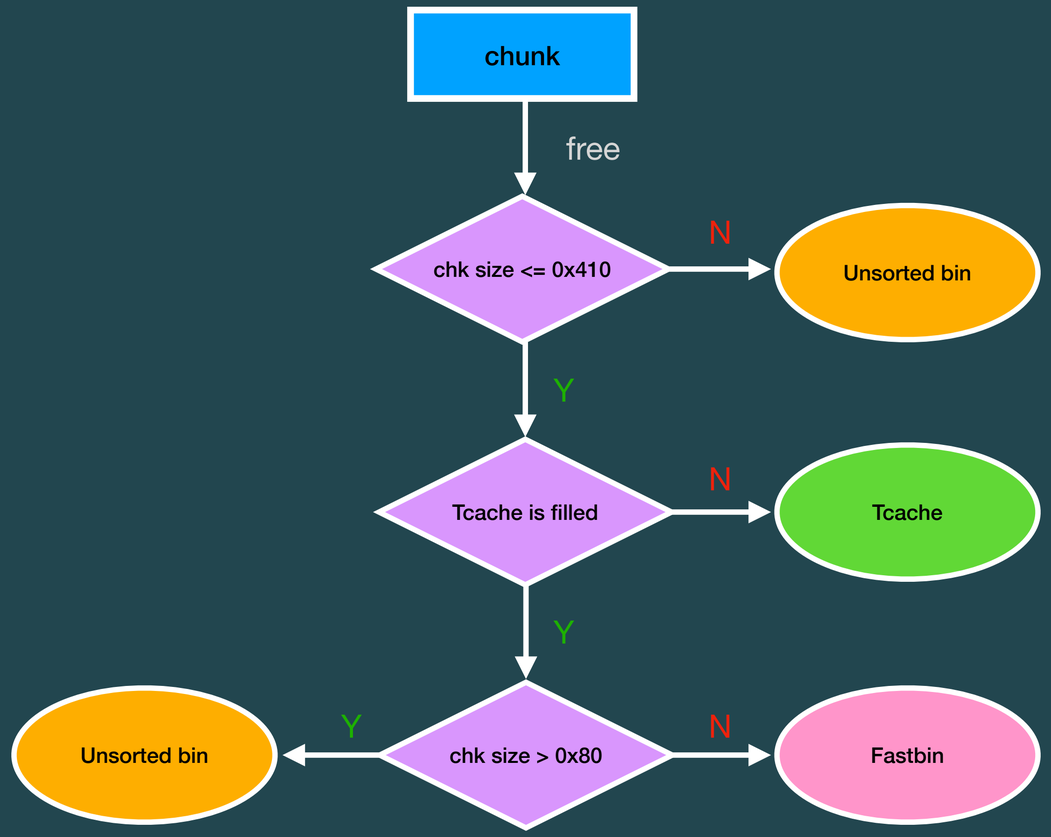
$ Code tracing

Malloc



$ Code tracing

Free



Vulnerability

$ Vulnerability

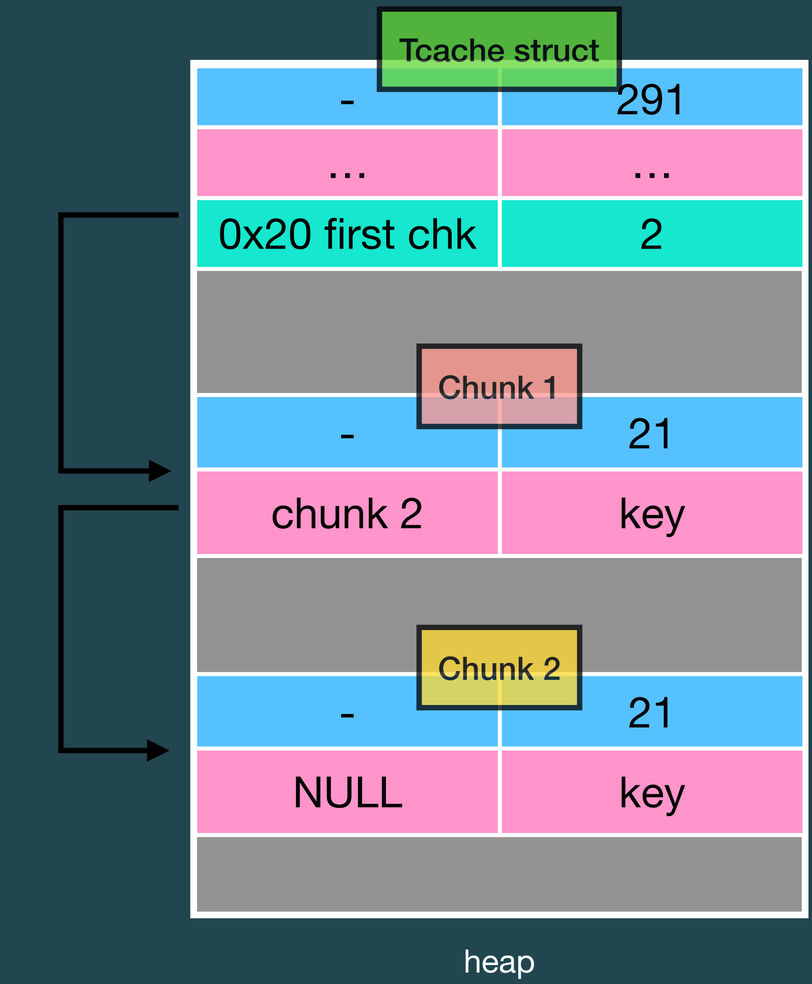
UAF

- Use-After-Free
- Chunk 在被 free 後並沒有把 pointer 設為 NULL，導致使用到已經被釋放的記憶體
- 這種指向已釋放資源的 pointer 又稱 dangling pointer

$ Vulnerability

UAF - Example

```
unsigned long *a = malloc(0x10);
*a = 0xdeadbeef;
free(a);
*a = 0xdeadbeef;
malloc(0x10);
malloc(0x10);
```

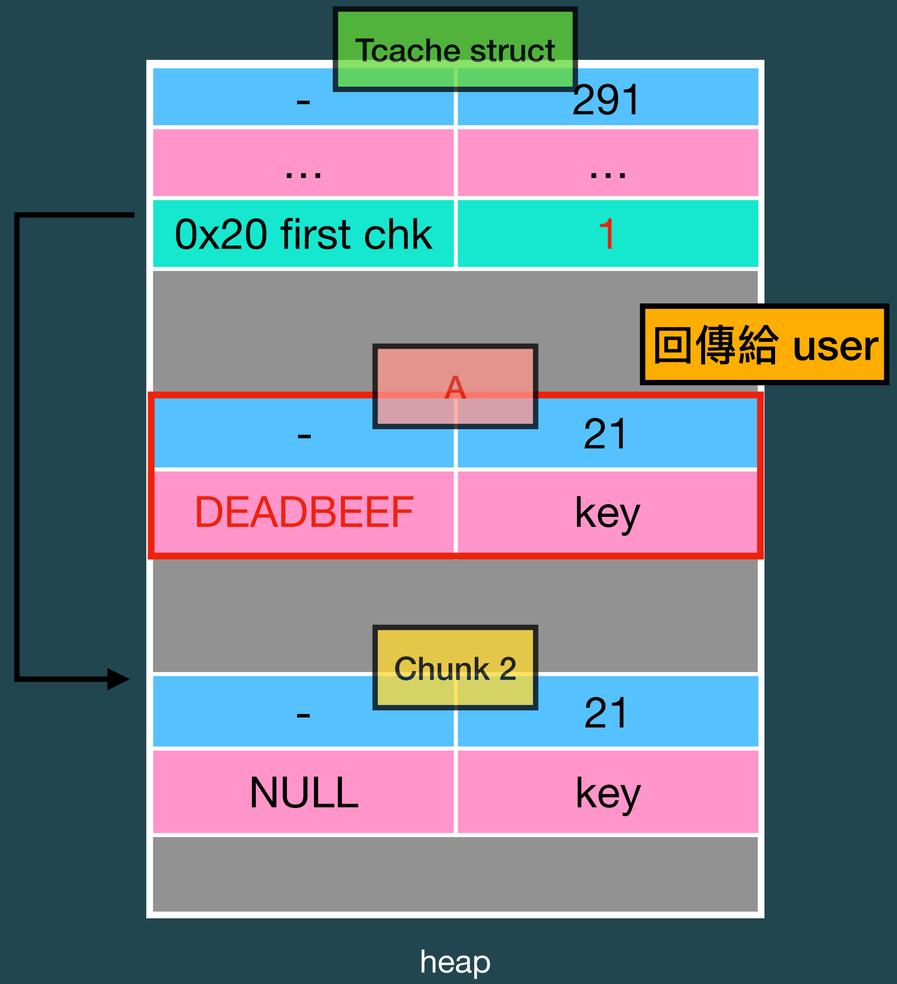


heap

$ Vulnerability

UAF - Example

```
unsigned long *a = malloc(0x10);
*a = 0xdeadbeef;
free(a);
*a = 0xdeadbeef;
malloc(0x10);
malloc(0x10);
```

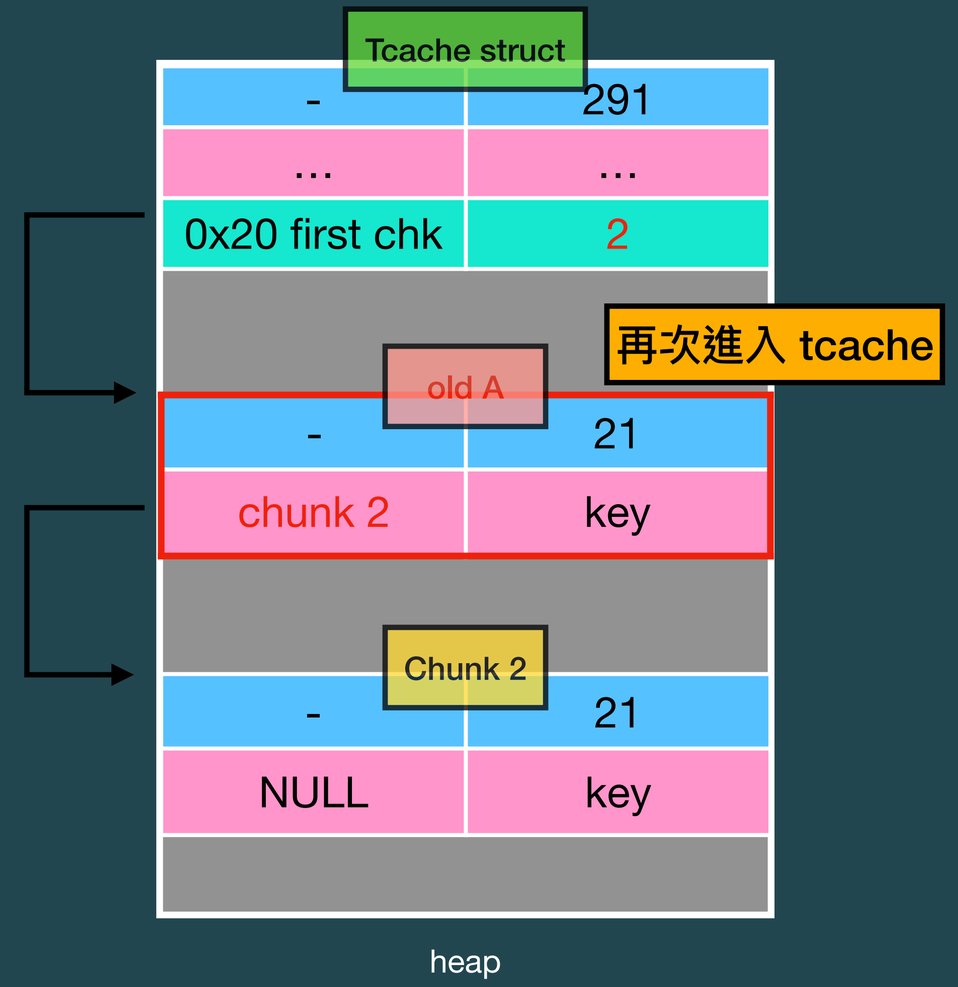


heap

$ Vulnerability

UAF - Example

```
unsigned long *a = malloc(0x10);
*a = 0xdeadbeef;
free(a);
*a = 0xdeadbeef;
malloc(0x10);
malloc(0x10);
```

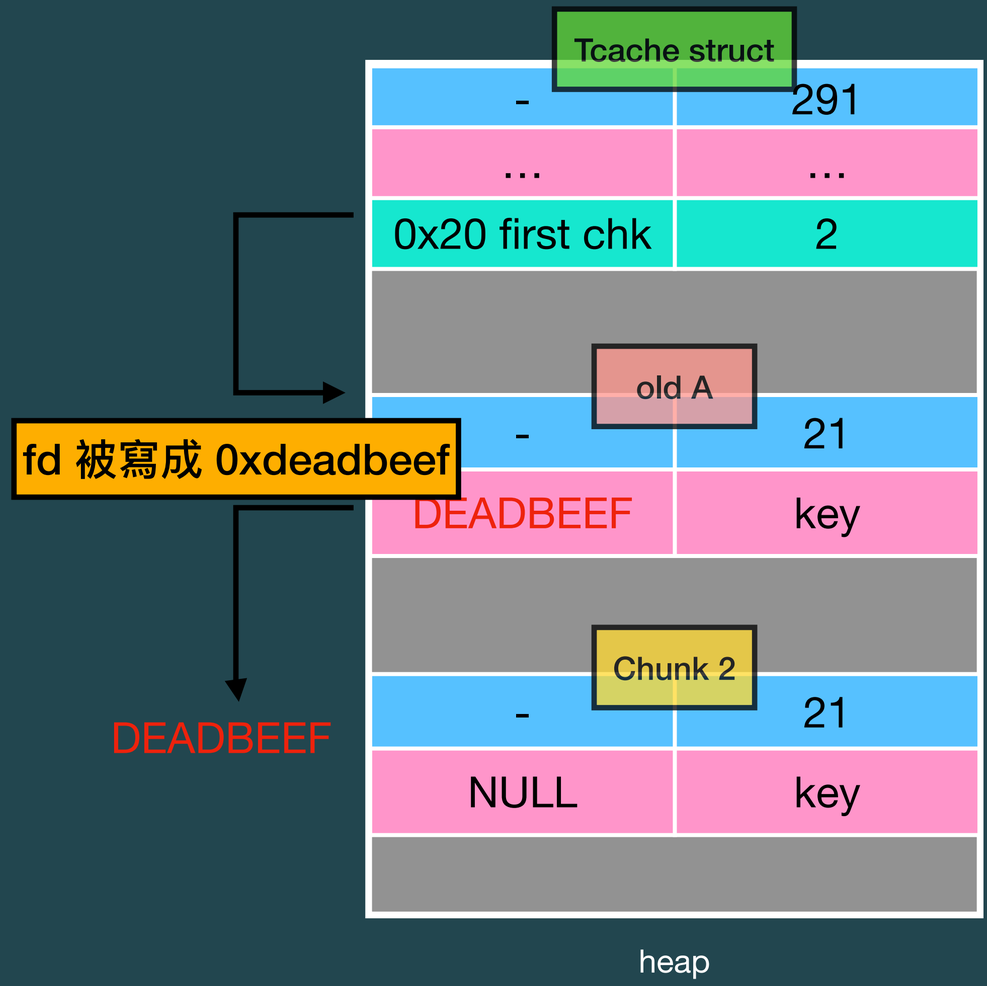


$ Vulnerability

UAF - Example

```
unsigned long *a = malloc(0x10);
*a = 0xdeadbeef;
free(a);
*a = 0xdeadbeef;
malloc(0x10);
malloc(0x10);
```

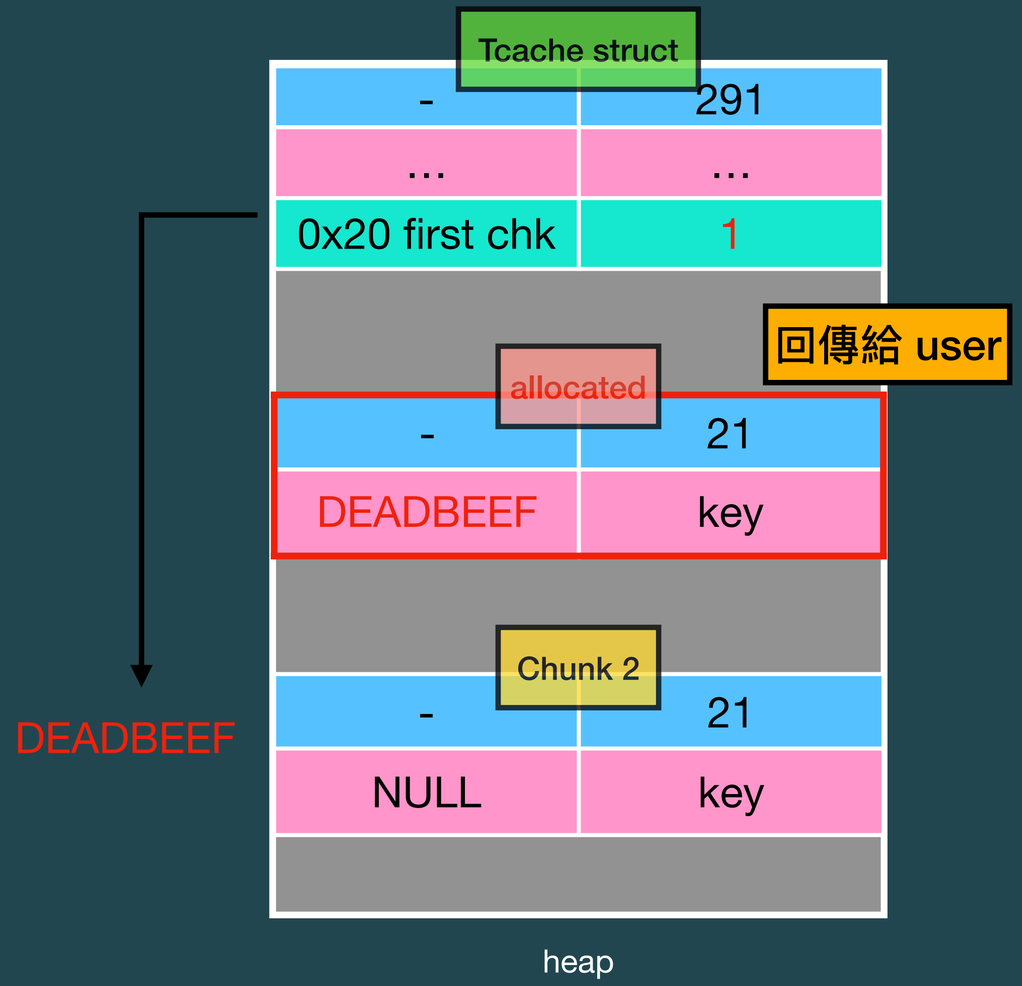
a 被釋放後仍被使用



$ Vulnerability

UAF - Example

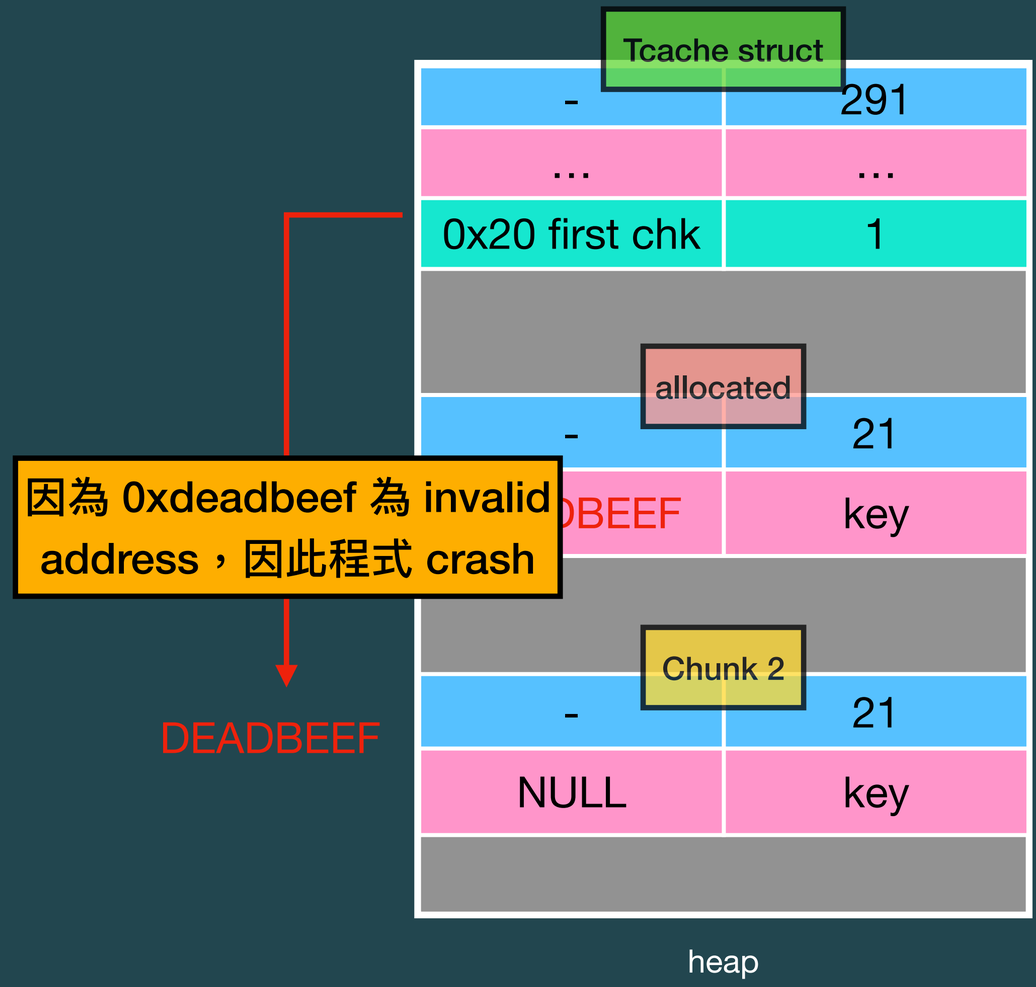
```
unsigned long *a = malloc(0x10);
*a = 0xdeadbeef;
free(a);
*a = 0xdeadbeef;
malloc(0x10);
malloc(0x10);
```



$ Vulnerability

UAF - Example

```
unsigned long *a = malloc(0x10);
*a = 0xdeadbeef;
free(a);
*a = 0xdeadbeef;
malloc(0x10);
malloc(0x10);
```



$ Vulnerability

UAF

$ Vulnerability

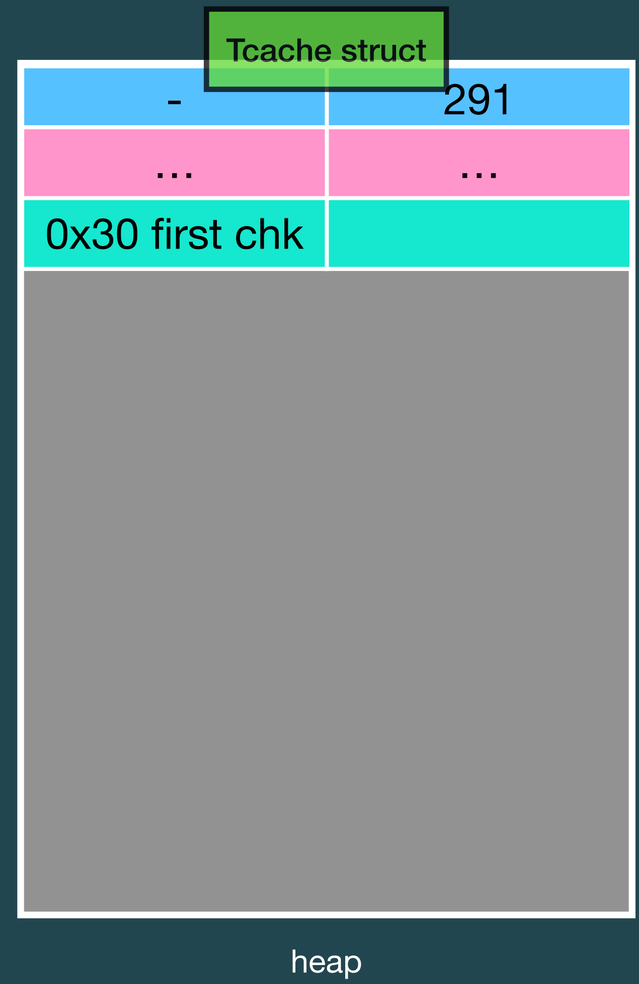
Heap overflow

- 在讀取資料時並沒有檢查好長度，導致可以越界寫到其他 chunk
  - 如果越界寫到的是 allocated chunk，可以寫 chunk 內的
    - 敏感資料，例如 function pointer
    - Size，就可以釋放任意 size 的 chunk
  - 如果越界寫到的是 freed chunk，可以寫 chunk 的
    - Fd、bk，改變 linked list 的連接
    - PREV_INUSE bit，讓 glibc 誤以為上一塊 chunk 已經被釋放

$ Vulnerability

Heap overflow - Example

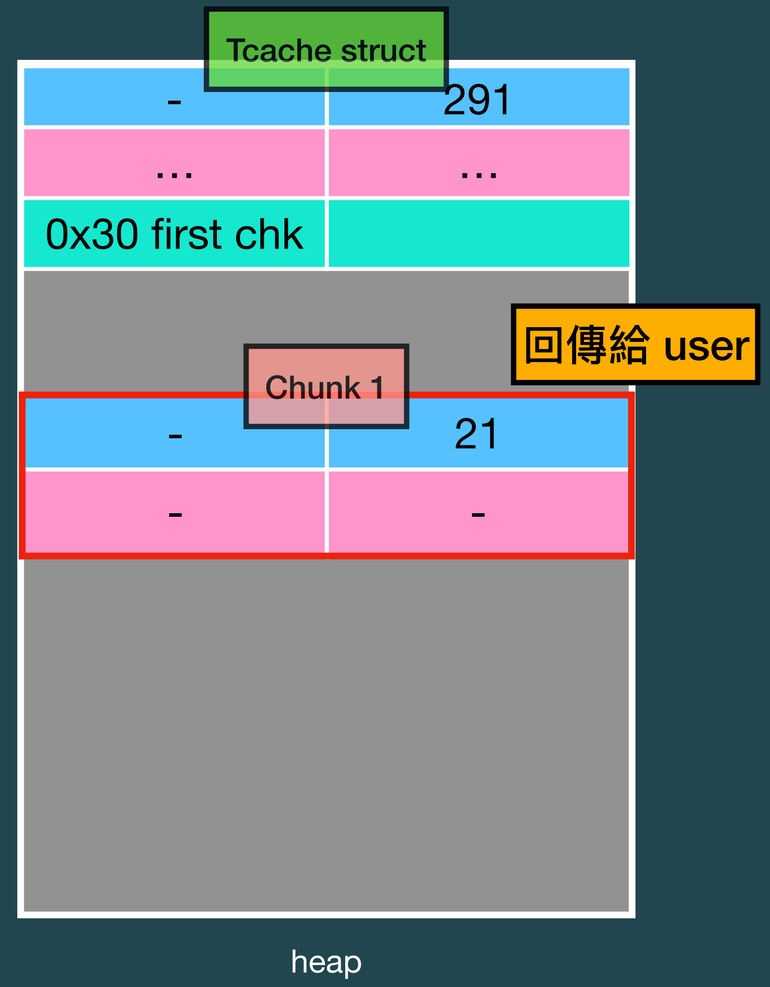
```
unsigned long *chk1 = malloc(0x10);
void *chk2 = malloc(0x10);
chk1[3] = 0x31;
free(chk2);
```



$ Vulnerability

Heap overflow - Example

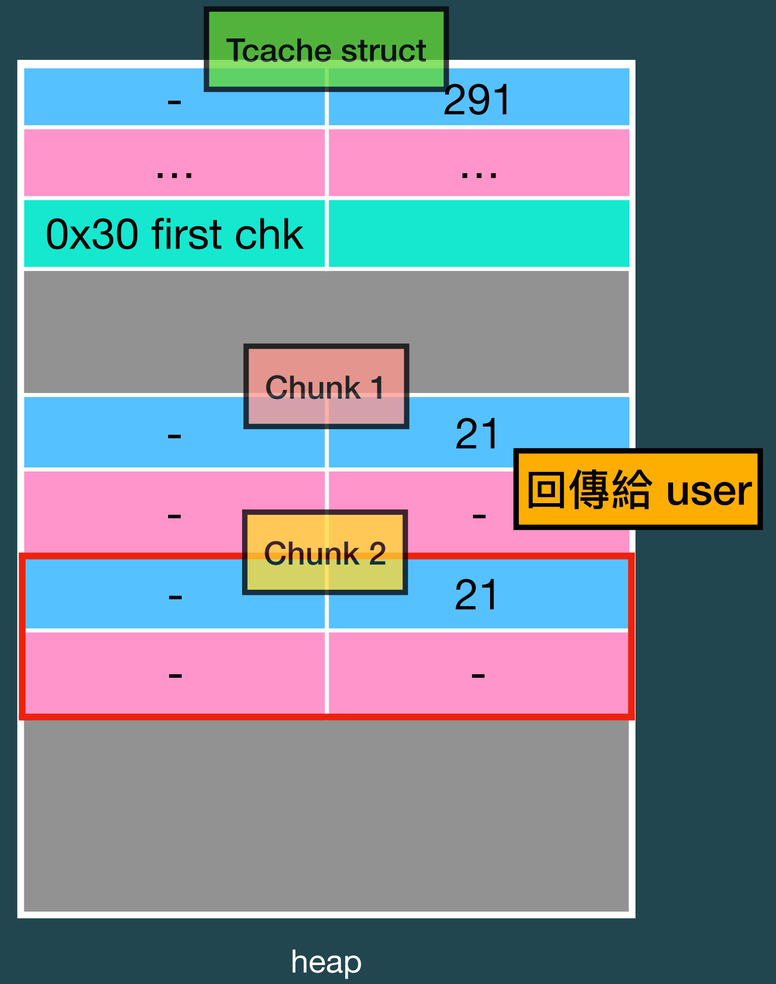
```
unsigned long *chk1 = malloc(0x10);
void *chk2 = malloc(0x10);
chk1[3] = 0x31;
free(chk2);
```



$ Vulnerability

Heap overflow - Example

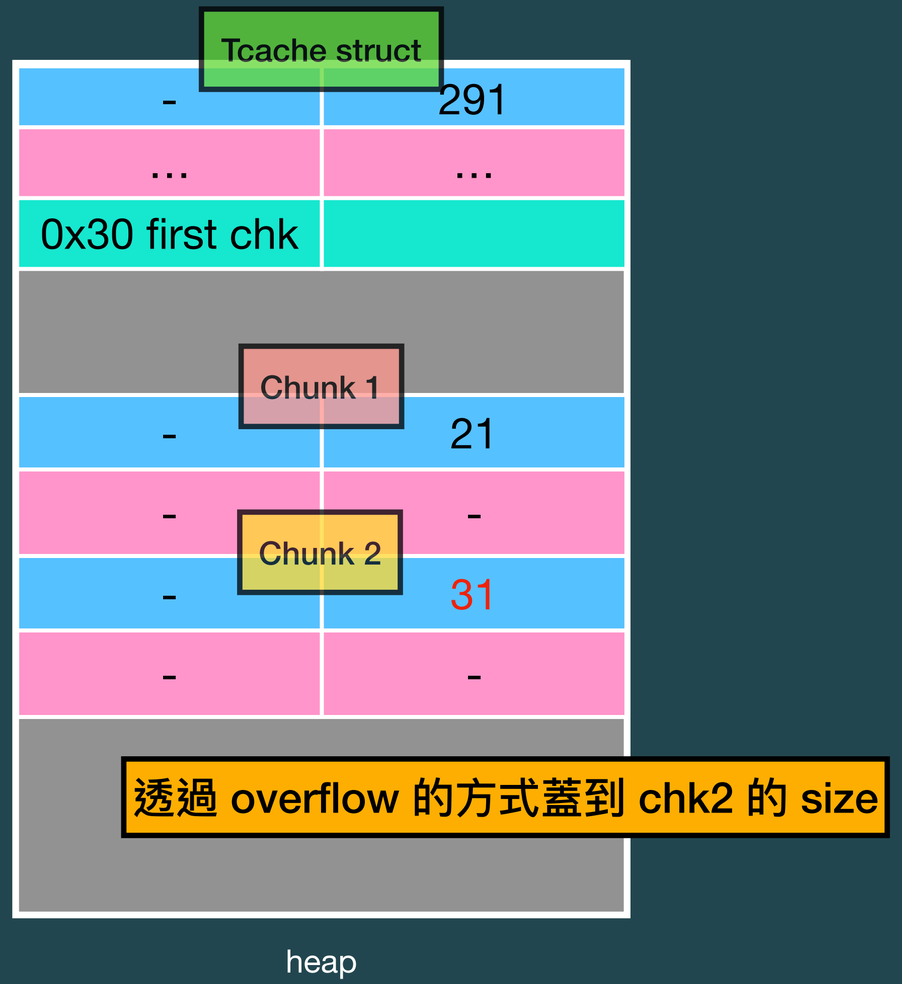
```
unsigned long *chk1 = malloc(0x10);
void *chk2 = malloc(0x10);
chk1[3] = 0x31;
free(chk2);
```



$ Vulnerability

Heap overflow - Example

```
unsigned long *chk1 = malloc(0x10);
void *chk2 = malloc(0x10);
chk1[3] = 0x31;
free(chk2);
```

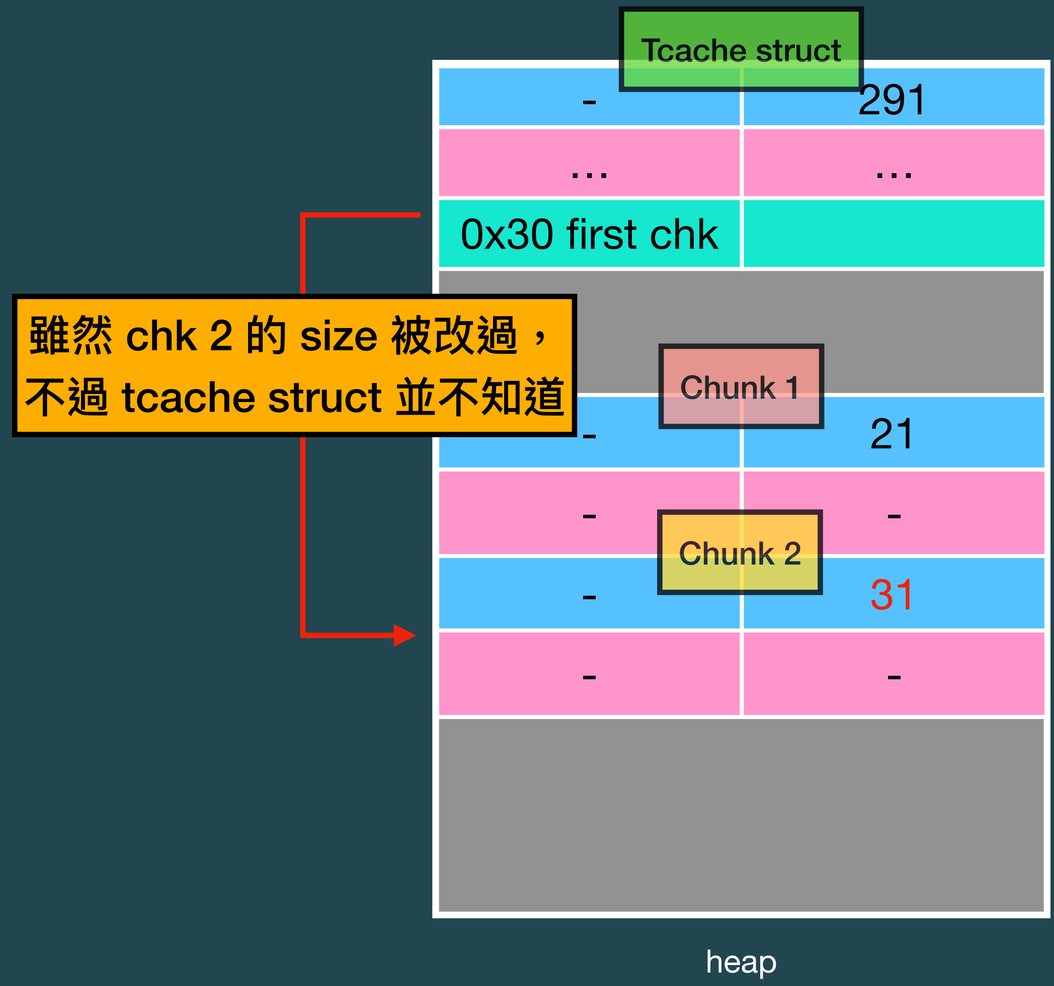


heap

$ Vulnerability

Heap overflow - Example

```
unsigned long *chk1 = malloc(0x10);
void *chk2 = malloc(0x10);
chk1[3] = 0x31;
free(chk2);
```



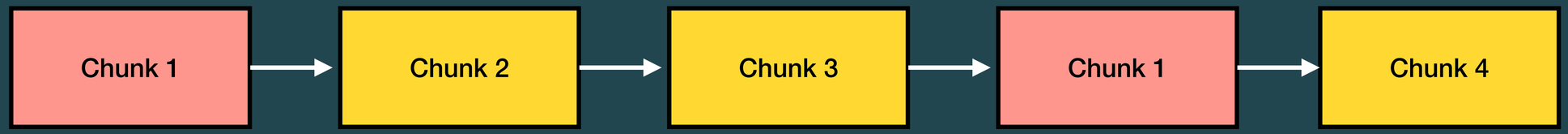
$ Vulnerability

Heap overflow - Example

$ Vulnerability

Double Free

- 當 chunk 被 free 兩遍，導致 bin 當中記錄了兩個相同的 chunk，就被稱作 double free
- 當透過 malloc 取出 chunk 時，因為 chunk 仍被 bin 所記錄，因此對 chunk 做修改就等於直接改動了 freed chunk
- 由於 double free 常發生，因此 glibc 當中有許多檢查機制來偵測這些異常的行為，如果要利用的話，就需要繞過那些檢查機制，讓 glibc 認為目前的行為是合法的

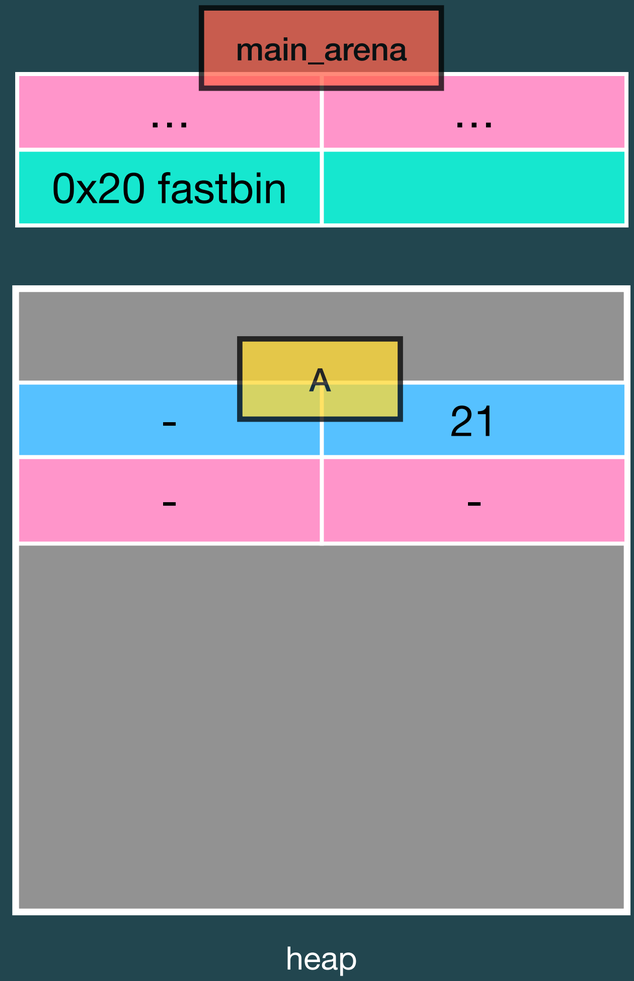


$ Vulnerability

Double Free - Example

假設 tcache 已經被填滿

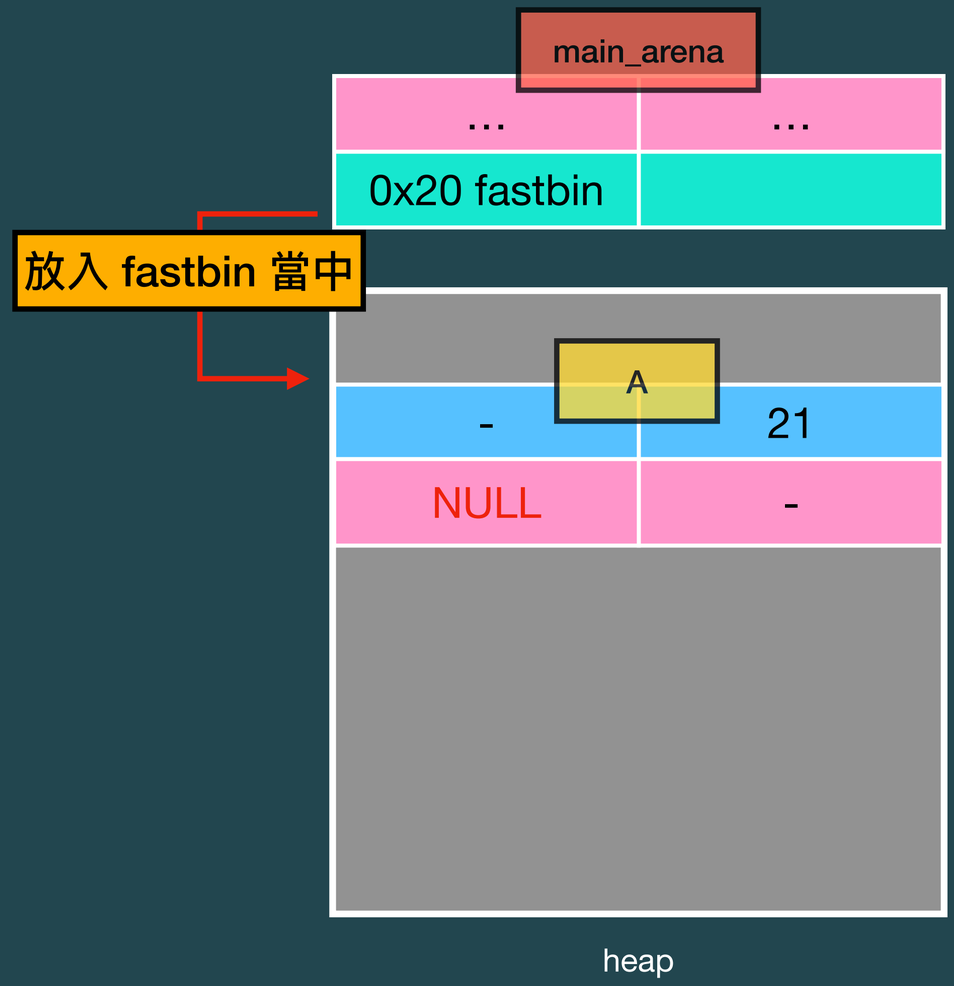
```
free(A);
free(A);
```



$ Vulnerability

Double Free - Example

```
free(A);
free(A);
```

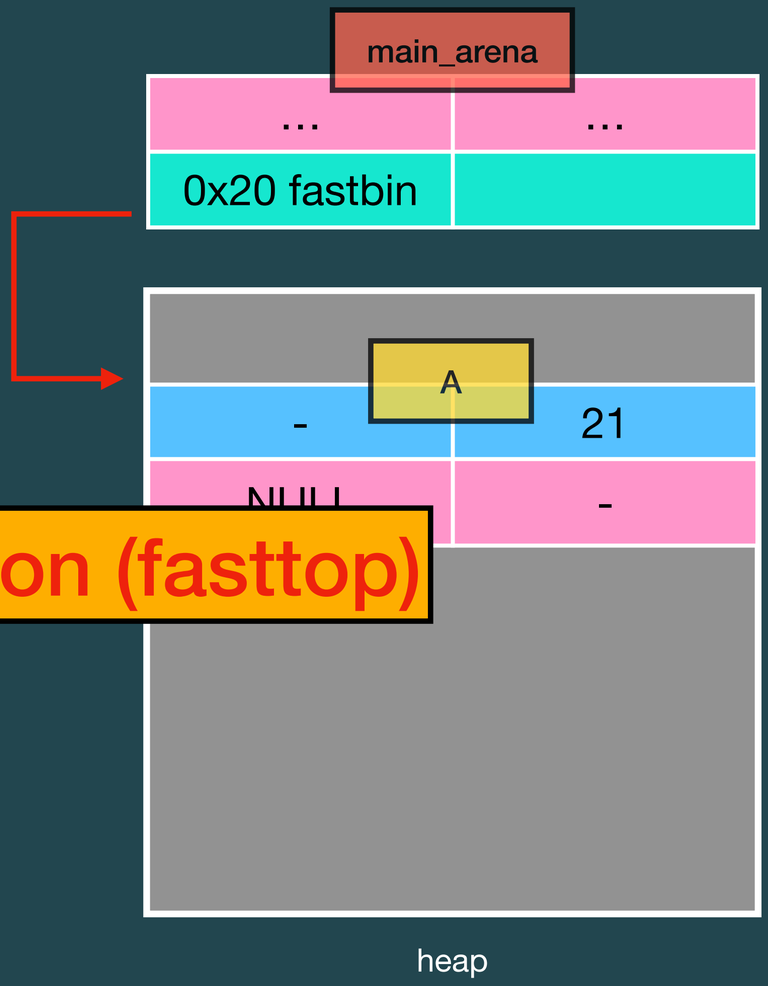


$ Vulnerability

Double Free - Example

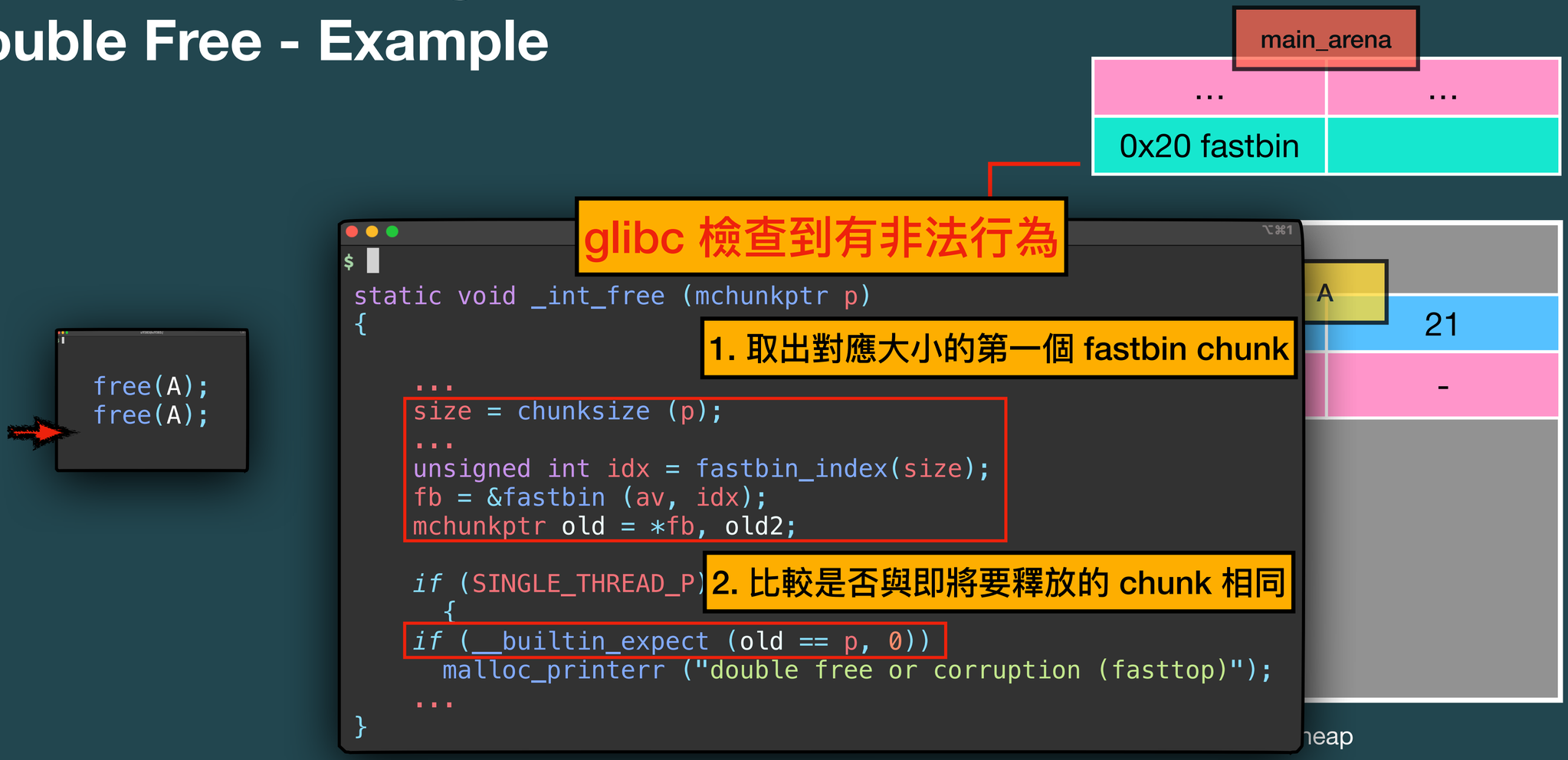
```
free(A);
free(A);
```

double free or corruption (fasttop)



$ Vulnerability

Double Free - Example

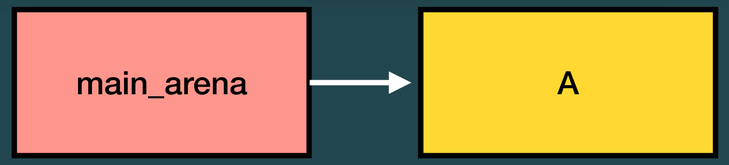


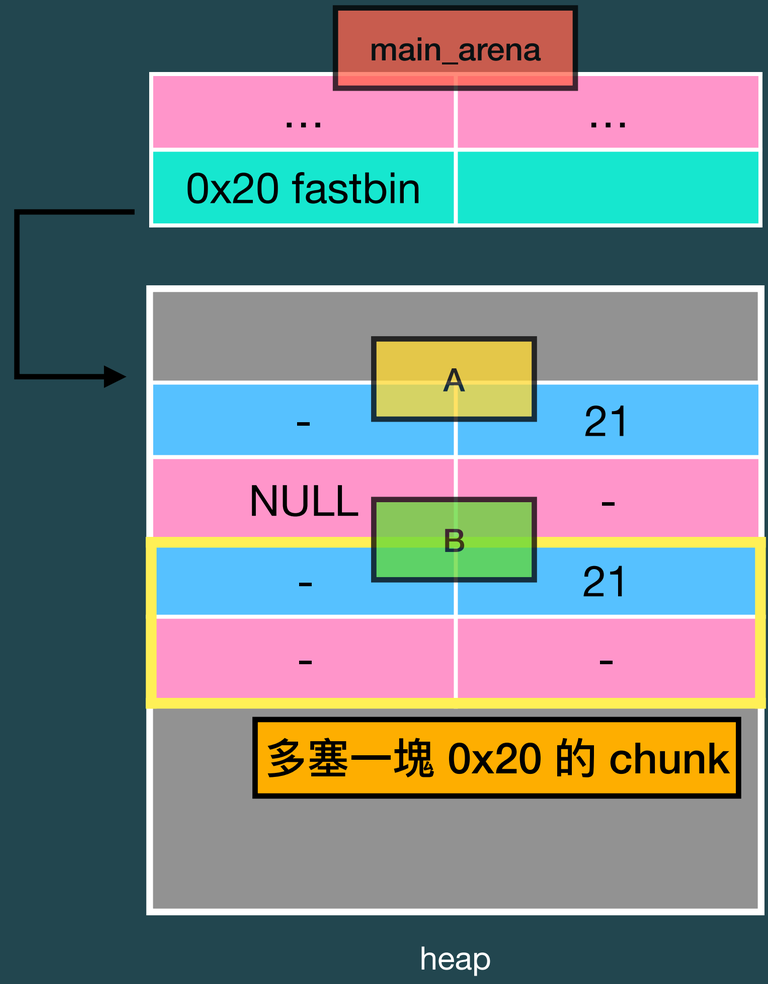
https://elixir.bootlin.com/glibc/glibc-2.31/source/malloc/malloc.c#L4263

$ Vulnerability

Double Free - Example

```
free(A);
free(B);
free(A);
// clean tcache
A = malloc(0x10);
*A = 0xdeadbeef;
malloc(0x10);
malloc(0x10);
malloc(0x10); // 0xdeadbeef
```



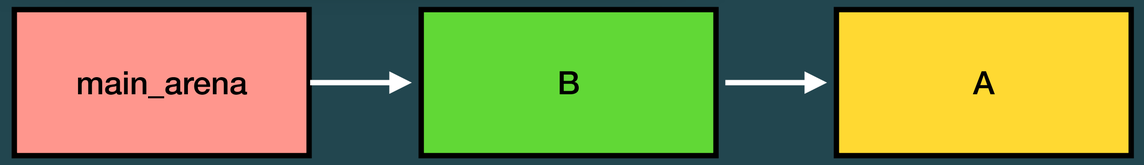


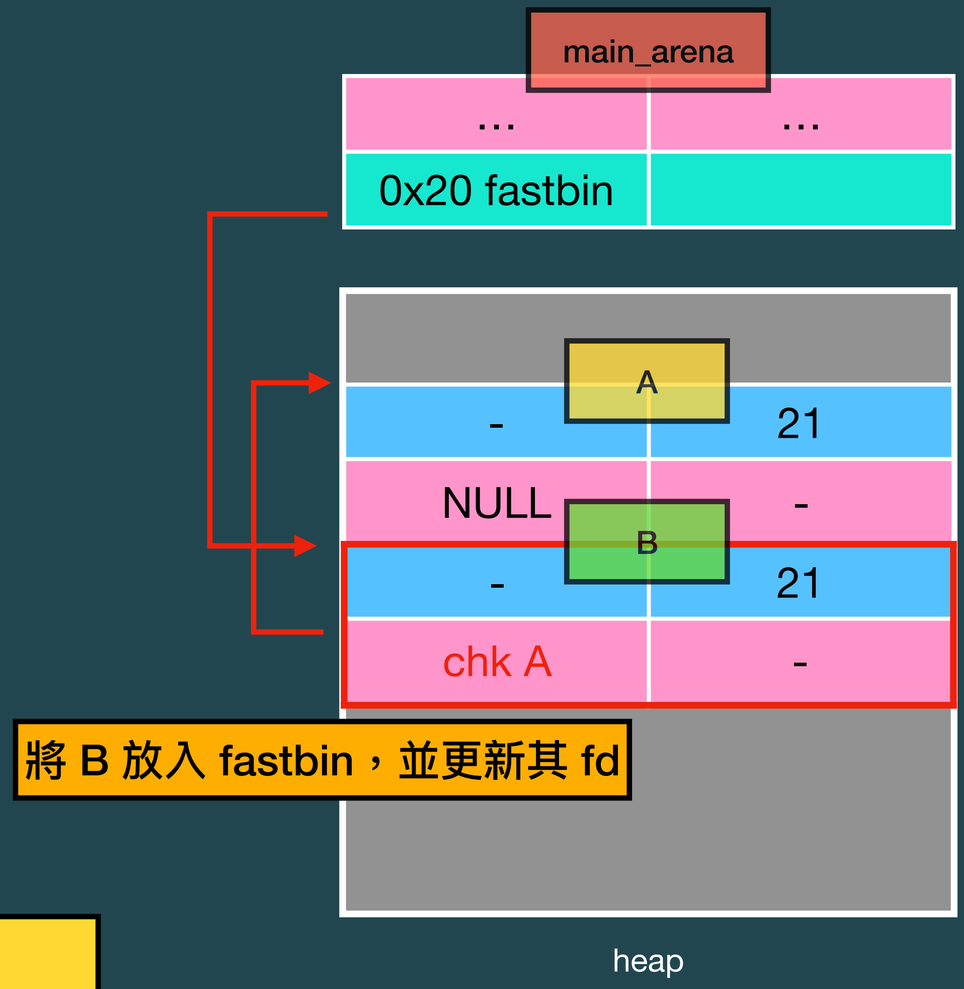
heap

$ Vulnerability

Double Free - Example

```
free(A);
free(B);
free(A);
// clean tcache
A = malloc(0x10);
*A = 0xdeadbeef;
malloc(0x10);
malloc(0x10);
malloc(0x10); // 0xdeadbeef
```





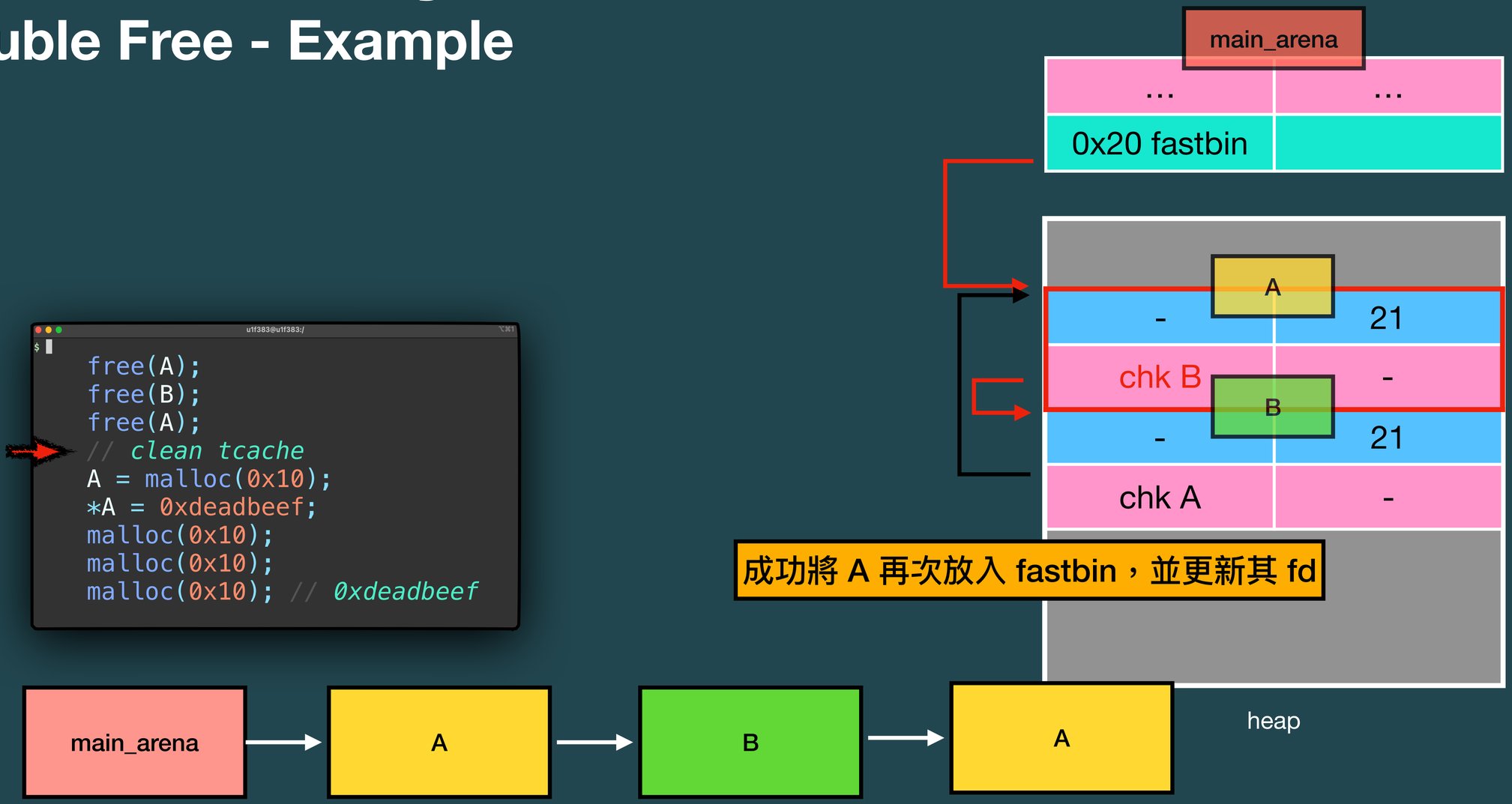
$ Vulnerability

Double Free - Example

$ Vulnerability

Double Free - Example

```
free(A);
free(B);
free(A);
// clean tcache
A = malloc(0x10);
*A = 0xdeadbeef;
malloc(0x10);
malloc(0x10);
malloc(0x10); // 0xdeadbeef
```



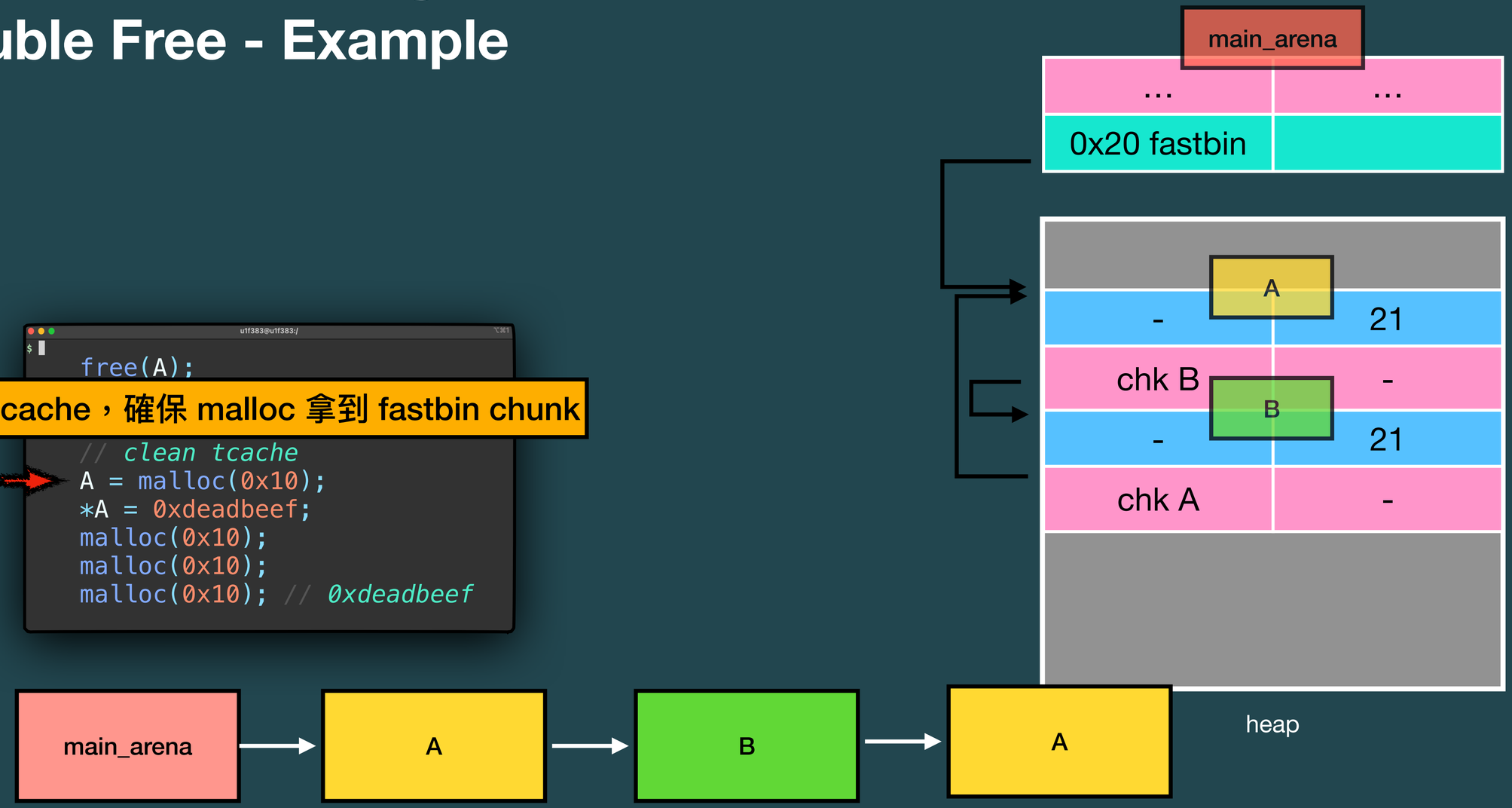
$ Vulnerability

Double Free - Example

```
free(A);
```

清空 tcache，確保 malloc 拿到 fastbin chunk

```
// clean tcache
A = malloc(0x10);
*A = 0xdeadbeef;
malloc(0x10);
malloc(0x10);
malloc(0x10); // 0xdeadbeef
```

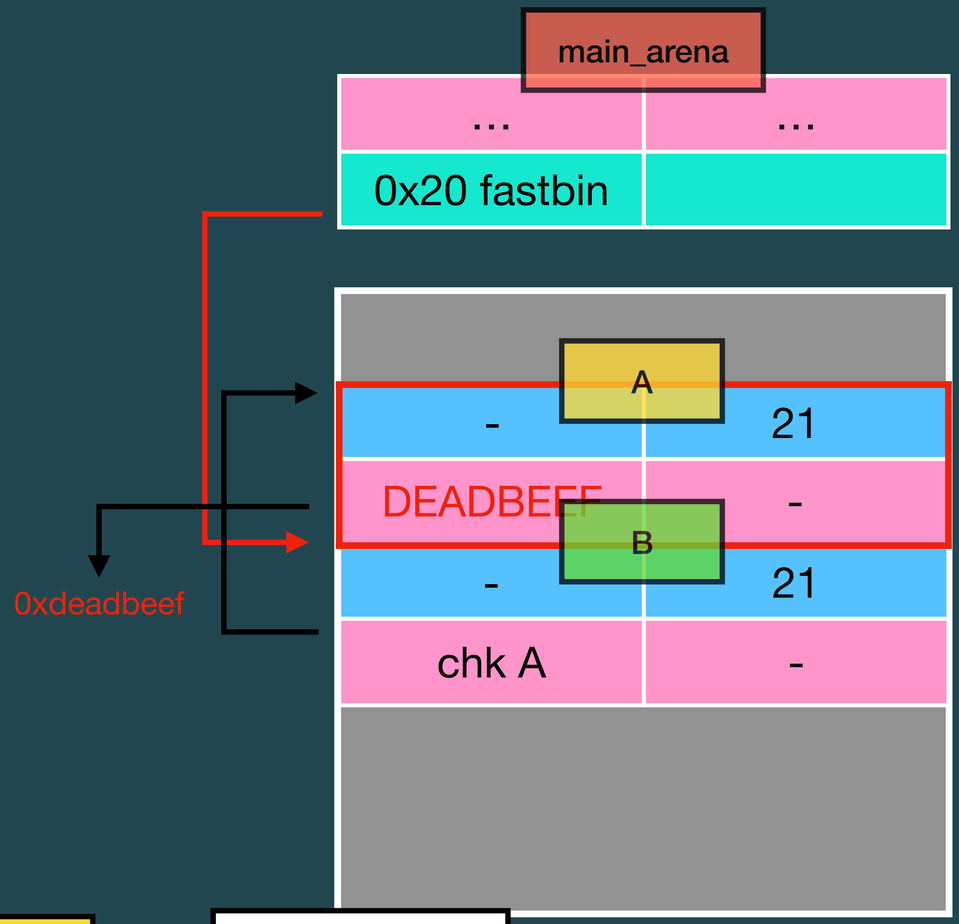


$ Vulnerability

Double Free - Example

```
free(A);
free(B);
A = malloc(0x10);
*A = 0xdeadbeef;
malloc(0x10);
malloc(0x10);
malloc(0x10); // 0xdeadbeef
```

取出 chk A，並且將 fd 改成 0xdeadbeef

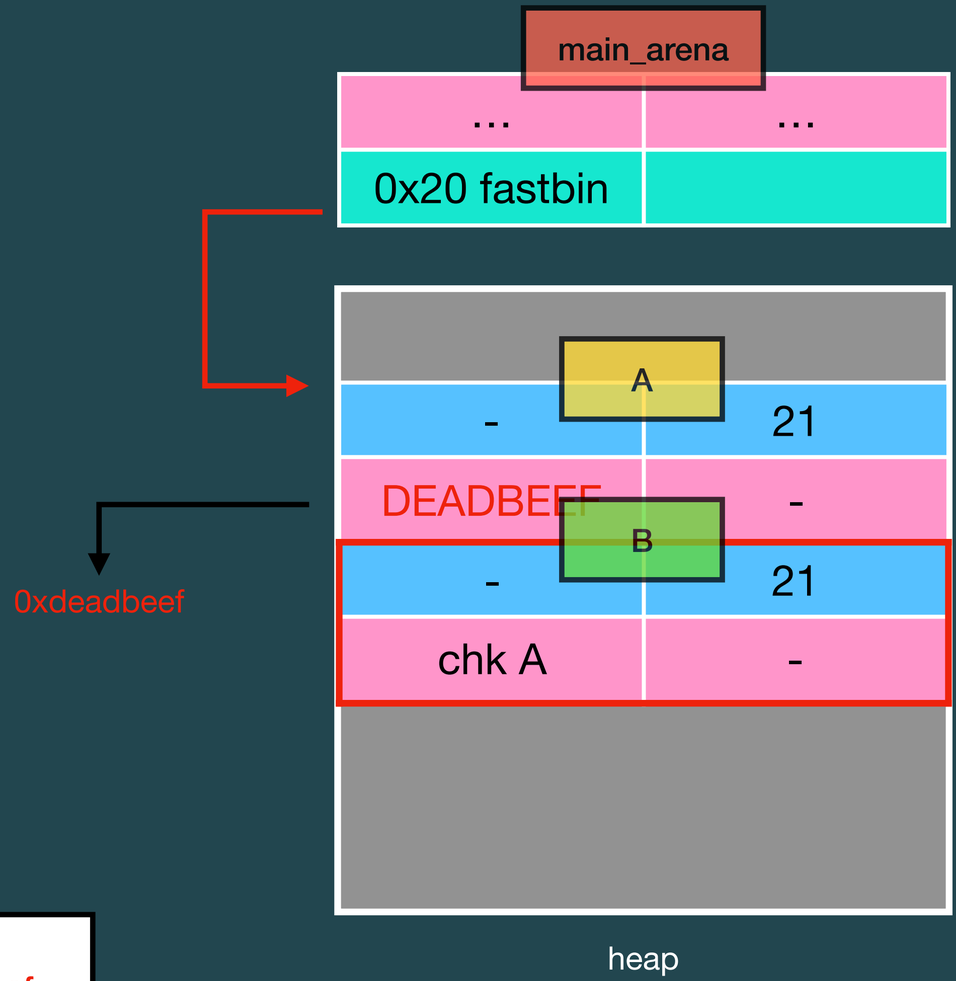


$ Vulnerability

Double Free - Example

```
free(A);
free(B);
free(A);
// clean tcache
A = malloc(0x10);
*A = 0xdeadbe
malloc(0x10);
malloc(0x10);
malloc(0x10); // 0xdeadbeef
```

取出 chk B

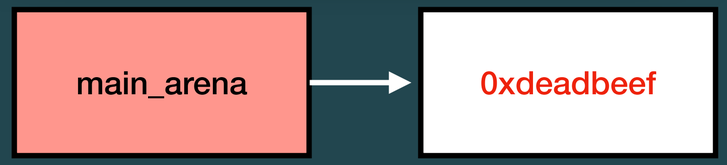


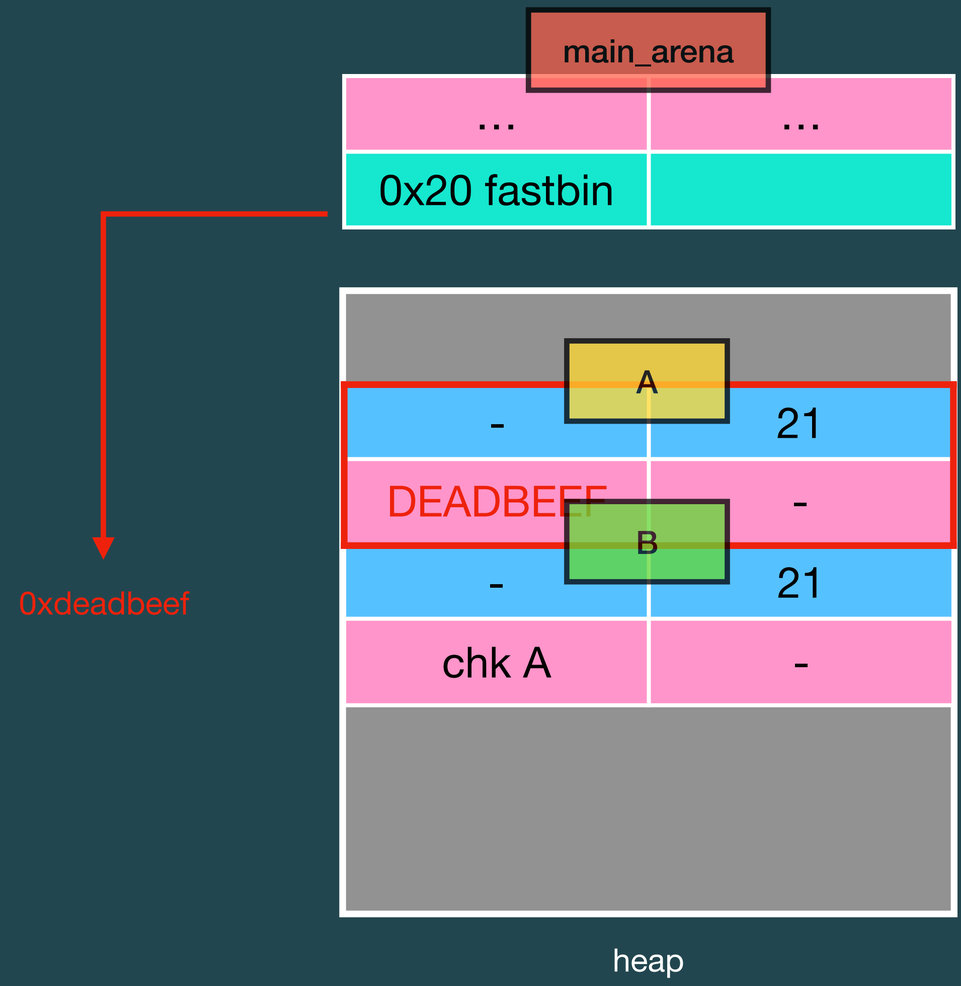
$ Vulnerability

Double Free - Example

```
free(A);
free(B);
free(A);
// clean tcache
A = malloc(0x10);
*A = 0xdeadbeef;
malloc(0x10); // 0xdeadbeef
```

再次取出 chk A，此時 0x20 fastbin 指向 0xdeadbeef



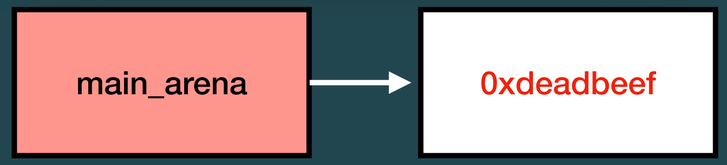


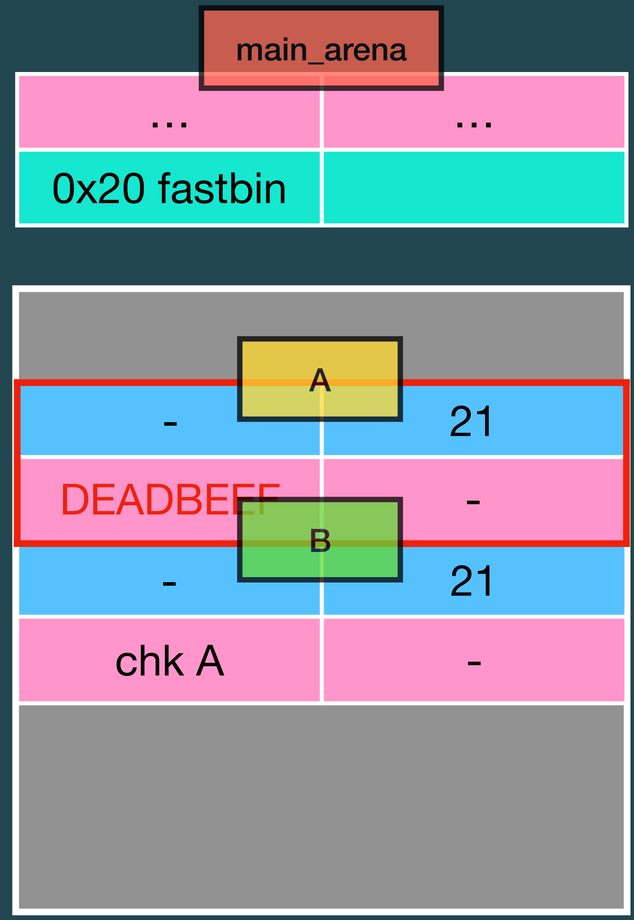
$ Vulnerability

Double Free - Example

```
free(A);
free(B);
free(A);
// clean tcache
A = malloc(0x10);
*A = 0xdeadbeef;
malloc(0x10);
malloc(0x10);
malloc(0x10); // 0xdeadbeef
```

程式 crash，因為 0xdeadbeef 為 invalid address





$ Vulnerability

$ Vulnerability

Lab - Market

- 程式設計中需要好好的初始化，並且在釋放記憶體時清除相關敏感資料
- 若初始化的流程是使用者可以控制的，則很有可能透過殘留在記憶體內的資料做利用

Exploitation goal

$ Exploitation goal

hook

- 跟記憶體分配相關的 function 都可以加上 hook 來自定義行為
- 常見的有 __free_hook、__malloc_hook 以及 __realloc_hook
- 透過定義這些 hook ，可以讓程式在呼叫 hooked function 時，跳轉至我們在 hook 中所寫入的位址

$ Exploitation goal

__free_hook

- __free_hook 不為 NULL 時，內容可以作為 function pointer 取代 free 的行為
- 例如寫入 system 的位址後，若 chunk 內容為 “/bin/sh”，則在呼叫 free 時會變成執行 system(“/bin/sh”)

u1f383@u1f383:/ ⌥⌘1

```
$
void
__libc_free (void *mem)
{
  mstate ar_ptr;
  mchunkptr p;                          /* chunk corresponding to mem */

  void (*hook) (void *, const void *)
    = atomic_forced_read (__free_hook);
    if (__builtin_expect (hook != NULL, 0))
    {
      (*hook)(mem, RETURN_ADDRESS (0));
      return;
    }
  ...
}
```

$ Exploitation goal

__malloc_hook, __realloc_hook

- __malloc_hook 與 __realloc_hook 不為 NULL 時，內容可以作為 function pointer 取代 malloc / realloc 的行為
- 寫入 one gadget，在呼叫 malloc / realloc 時被 trigger

u1f383@u1f383:/

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

  _Static_assert (PTRDIFF_MAX <= SIZE_MAX / 2,
                  "PTRDIFF_MAX is not more than half of SIZE_MAX");

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

__malloc_hook

u1f383@u1f383:/

```
$
void *
__libc_realloc (void *oldmem, size_t bytes)
{
  mstate ar_ptr;
  INTERNAL_SIZE_T nb;         /* padded request size */

  void *newp;             /* chunk to return */

  void *(*hook) (void *, size_t, const void *) =
    atomic_forced_read (__realloc_hook);
  if (__builtin_expect (hook != NULL, 0))
    return (*hook)(oldmem, bytes, RETURN_ADDRESS (0));
  ...
}
```

__realloc_hook

Exploitation tech

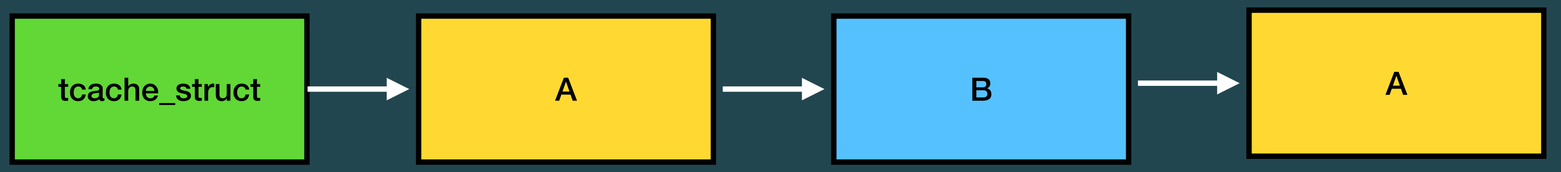
$ Exploitation tech

- 由於攻擊手法會根據功能跟情況有所不同，在此只介紹 how2heap repo 所記錄 glibc-2.31 的攻擊方法中，常見的三種利用方式
  - Tcache poisoning
  - Fastbin attack (fastbin dup)
  - Overlapping chunks

$ Exploitation tech

Tcache poisoning

- **Explanation** - 使用 double free 讓 tcache 當中存在兩個相同的 chunk，並利用修改 fd 的方式，將對應位址視為 chunk 分配給 user
  - Tcache 拿 chunk 時並不會檢查 chunk size 是否合法，因此常會拿 __free_hook 寫 system
- **Protection1** - 當釋放 chunk 時，如果 chunk + 8 (key) 位置的值與當前 heap 的 &tcache_struct 相等，則會遍歷所有 entry，檢查是否有相同的 chunk，確保沒有 double free 的發生



$ Exploitation tech

Tcache poisoning

```
$ static void _int_free (..., mchunkptr p,
  {
      ...
      tcache_entry *e = (tcache_entry *) chunk2mem (p);
      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");
      ...
  }
```

1. 將要釋放的 chunk 視為 tcache entry，並檢查 key 欄位是否為 tcache struct 的位址

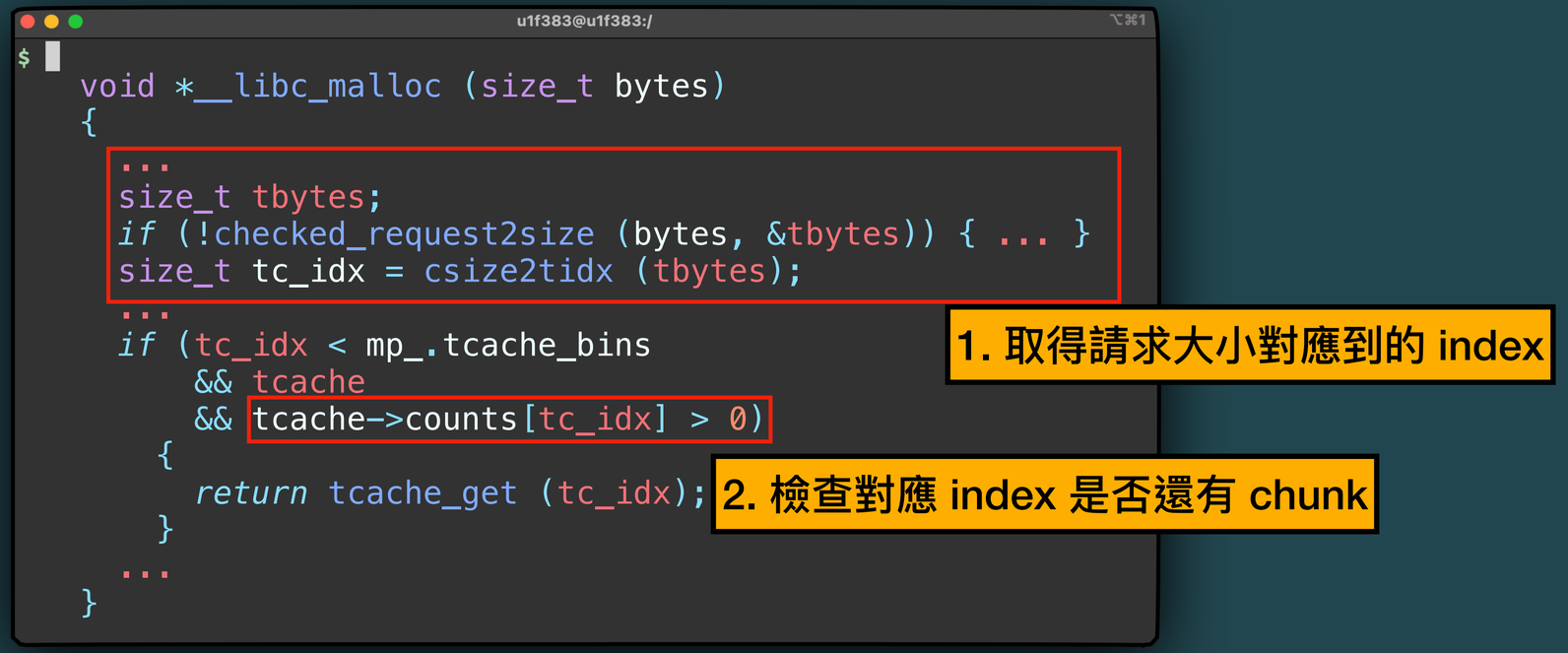
2. 如果是的話，則遍歷所有 entry，檢查此 chunk 使否已經存在

https://elixir.bootlin.com/glibc/glibc-2.31/source/malloc/malloc.c#L4187

$ Exploitation tech

Tcache poisoning

- **Protection2** - 當取出 chunk 時，會檢查對應大小的 counter 是否大於 0，如果是的話才會取出 tcache_struct 當中指向的第一塊 chunk



https://elixir.bootlin.com/glibc/glibc-2.31/source/malloc/malloc.c#L3049

$ Exploitation tech

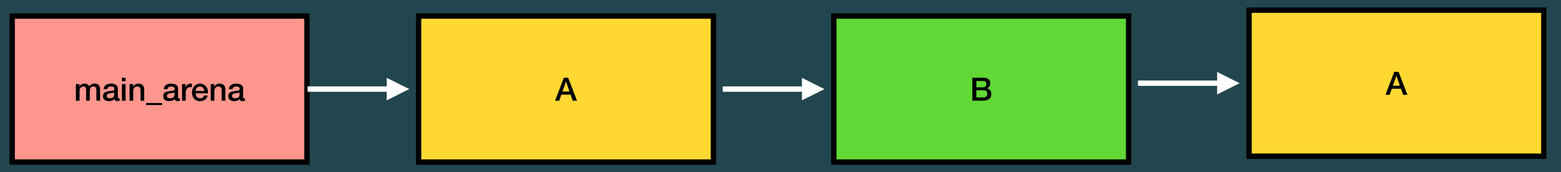
Tcache poisoning

- **Bypass Protection1** - 透過 UAF 或是 heap overflow，修改 chunk 的 key 欄位
- **Bypass Protection2**
  - 拿到 tcache_struct 的 chunk 後修改 counts 欄位成非 0 的值
  - 多次 free 相同的 chunk

$ Exploitation tech

Fastbin Attack

- **Explanation** - 使用 double free 讓 tcache 當中存在兩個相同的 chunk，並利用修改 fd 的方式，將對應位址視為 chunk 分配給 user
- **Protection1** - 當釋放 chunk 時，檢查 fastbin 的第一個 chunk 與此 chunk 是否相同，確保沒有 double free 的發生 (link)
- **Protection2** - 在取得 chunk 時，會檢查請求大小是否與被回傳的 chunk 的大小相同



$ Exploitation tech

Fastbin Attack

u1f383@u1f383:/

```
$ static void *_int_malloc (..., size_t bytes)
  {
    ...
    if (!checked_request2size (bytes, &nb)) { ... }
    if ((unsigned long) (nb) <= (unsigned long) (get_max_fast ()))
      {
        idx = fastbin_index (nb);
        mfastbinptr *fb = &fastbin (av, idx);
        mchunkptr pp;
        victim = *fb;

        if (victim != NULL)
      {
        *fb = victim->fd;
        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)");
    ...
  }
```

1. 取得 chunk size，並檢查是否位於 fastbin 的範圍內

2. 取得第一個 fastbin chunk，存在變數 victim

3. 檢查 victim 對應到的 fastbin index 是否與當前請求的 size 相同，
舉例來說，請求 0x30 但是卻拿到 0x40 大小的 chunk

$ Exploitation tech

Fastbin Attack

- **Bypass Protection1** - 能利用 free(a); free(b); free(a); 的方式繞過檢查
- **Bypass Protection2**
  - 因為 library address 都為 **0x7f** 開頭，而 0x70 為合法的 fastbin size，因此可以通過 fastbin 的保護機制
  - 而 __malloc_hook 上方會有 library address，因此可以先取得上方的 chunk，在寫 one gadget 到 __malloc_hook 來做 exploit

$ Exploitation tech

Overlapping chunks

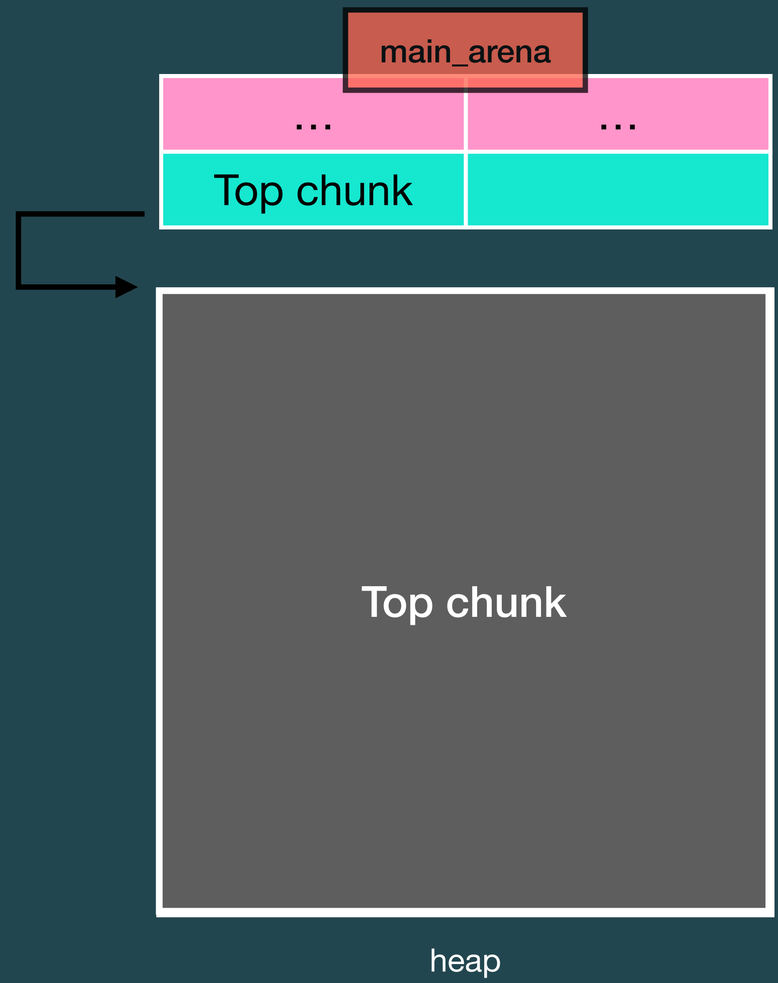
- **Explanation** - 透過修改 chunk size，讓 chunk 在被釋放時 trigger consolidation，使得正在使用的 chunk 與已經釋放的 chunk 有部分重疊，也就代表
  - 使用中的 chunk 可以更改 freed chunk 中的 fd、bk
  - freed chunk 在被分配時，會分配到與使用中的 chunk 相同的區塊，可以修改敏感資料
- **Consolidation** -當釋放記憶體時，若檢查到相鄰的 chunk 沒有被使用，會將其合併成一塊更大的 freed chunk
  - 為 glibc 用來減少 heap fragmentation 的機制
  - Chunk 在被合併前，會先透過 unlink 的機制來離開原本的 bin

$ Exploitation tech

Overlapping chunks

分配大小不會進 tcache 的 chunk A

```
$
A = malloc(0x410);
B = malloc(0x10);

*(A-1) = 0x421 + 0x20; // 0x20 == B 的 chunk size
free(A); // consolidate to top chunk

A = malloc(0x430);
total = (0x430 / 8);
A[total - 2] = 0xdeadbeef;
B[0] == 0xdeadbeef;
```



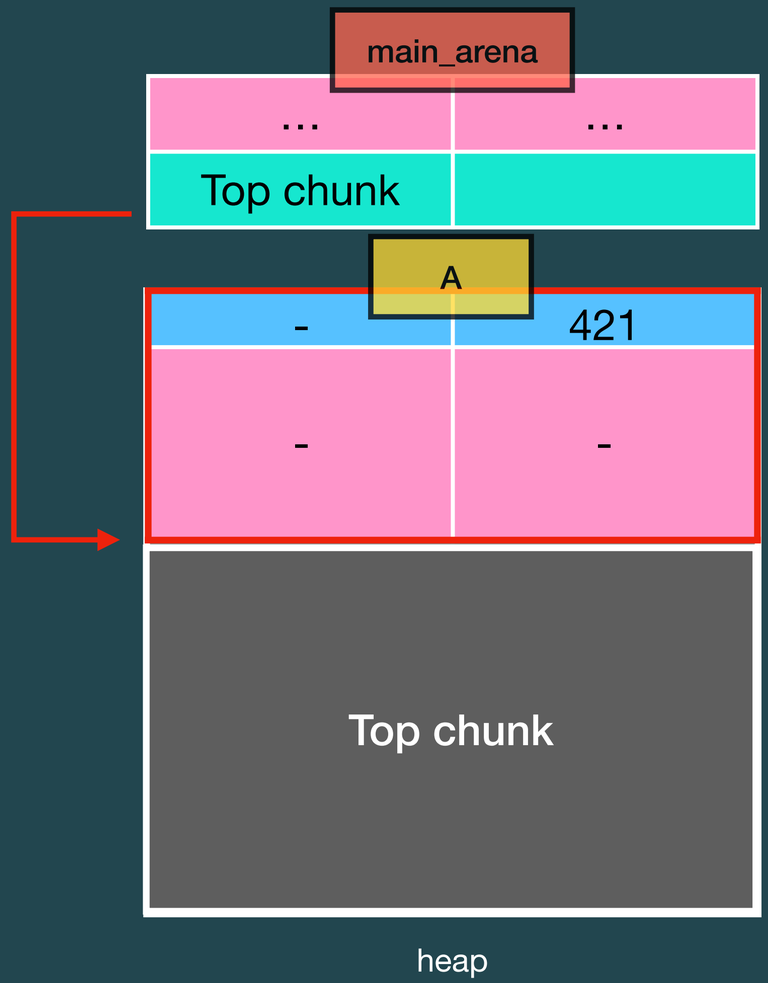
$ Exploitation tech

Overlapping chunks

u1f383@u1f383:/

```
$
A = malloc(0x
B = malloc(0x10);

*(A-1) = 0x421 + 0x20; // 0x20 == B 的 chunk size
free(A); // consolidate to top chunk

A = malloc(0x430);
total = (0x430 / 8);
A[total - 2] = 0xdeadbeef;
B[0] == 0xdeadbeef;
```

分配要被 overlap 的 chunk B

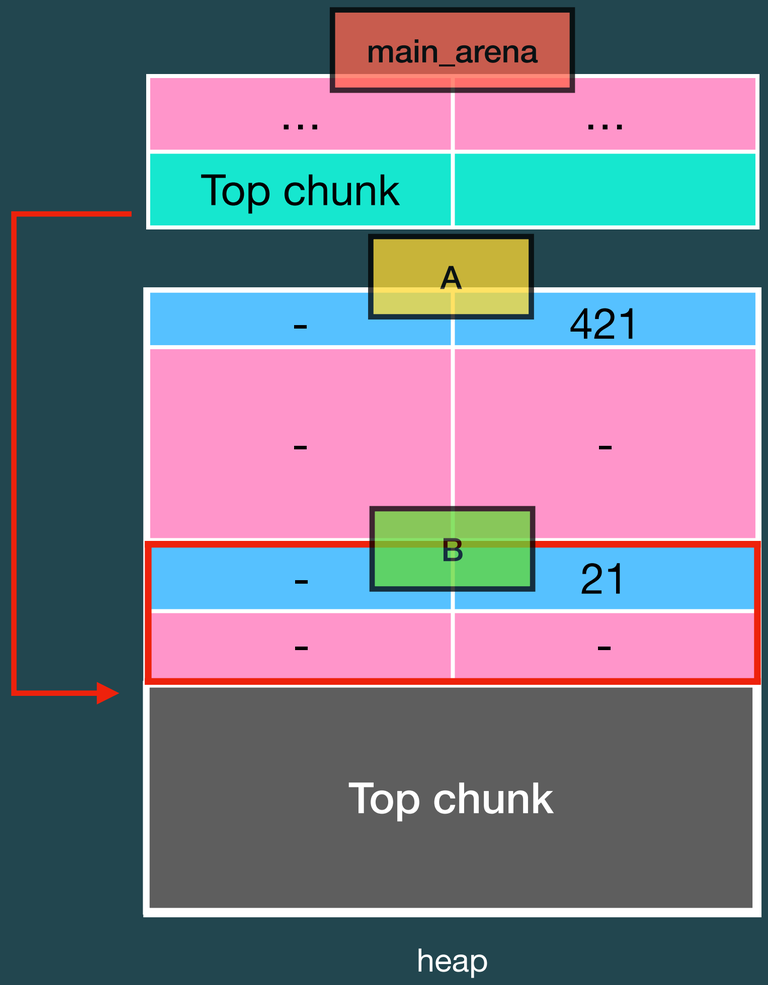


$ Exploitation tech

Overlapping chunks

u1f383@u1f383:/

```
$
A = malloc(0x410);
B = 修改 chunk size，剛好涵蓋整個 chunk B

*(A-1) = 0x421 + 0x20; // 0x20 == B 的 chunk size
free(A); // consolidate to top chunk

A = malloc(0x430);
total = (0x430 / 8);
A[total - 2] = 0xdeadbeef;
B[0] == 0xdeadbeef;
```



$ Exploitation tech

Overlapping chunks

u1f383@u1f383:/ ⌥⌘1

```
$ 
static void _int_free (..., mchunkptr p, ...)
{
  ...
  size = chunksize (p);
  ...
  else if (!chunk_is_mmapped(p)) {
    ...
    /* consolidate backward */
    if (!prev_inuse(p)) {
      // ... some operation
      unlink_chunk (av, p);
    }
    ...
    if (nextchunk != av->top) {
      nextinuse = inuse_bit_at_offset(nextchunk, nextsize);
      /* consolidate forward */
      if (!nextinuse) {
        unlink_chunk (av, nextchunk);
        size += nextsize;
      }
      ...
    } else { /* consolidate with top chunk */
      size += nextsize;
      set_head(p, size | PREV_INUSE);
      av->top = p;
      check_chunk(av, p);
    }
    ...
}
```

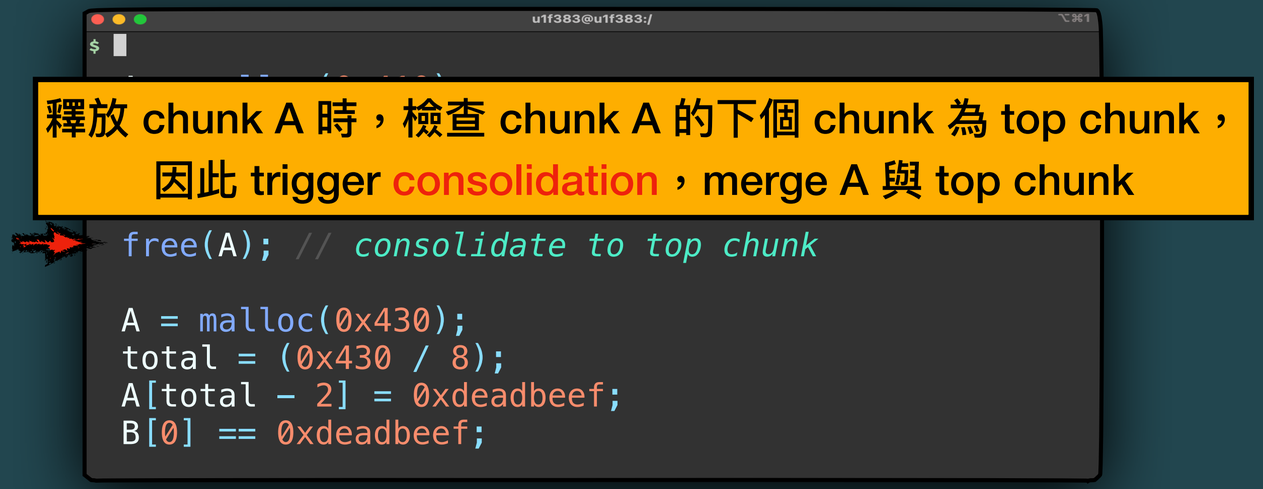
1. 當前一塊 chunk 沒有在使用，則 trigger consolidate，並且用 unlink_chunk 來離開 bin

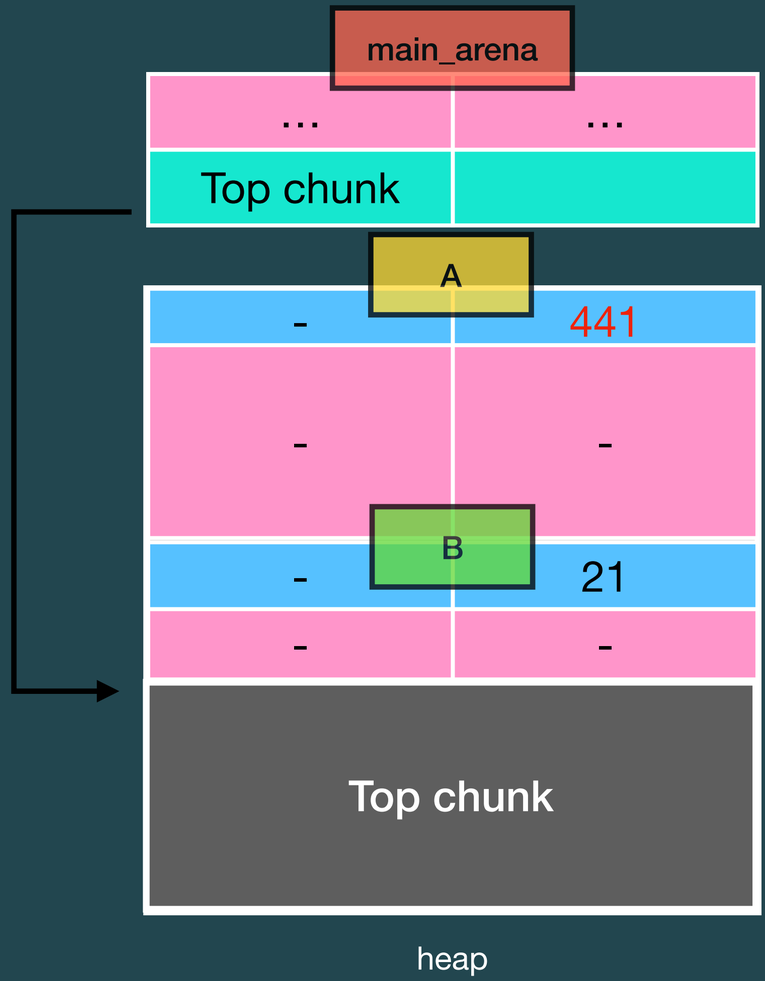
2. 下一塊 chunk 沒在用，並且不是 top chunk 的話，一樣 trigger consolidate

3. 如果下一塊是 top chunk，則與 top chunk 做 consolidate

$ Exploitation tech

Overlapping chunks



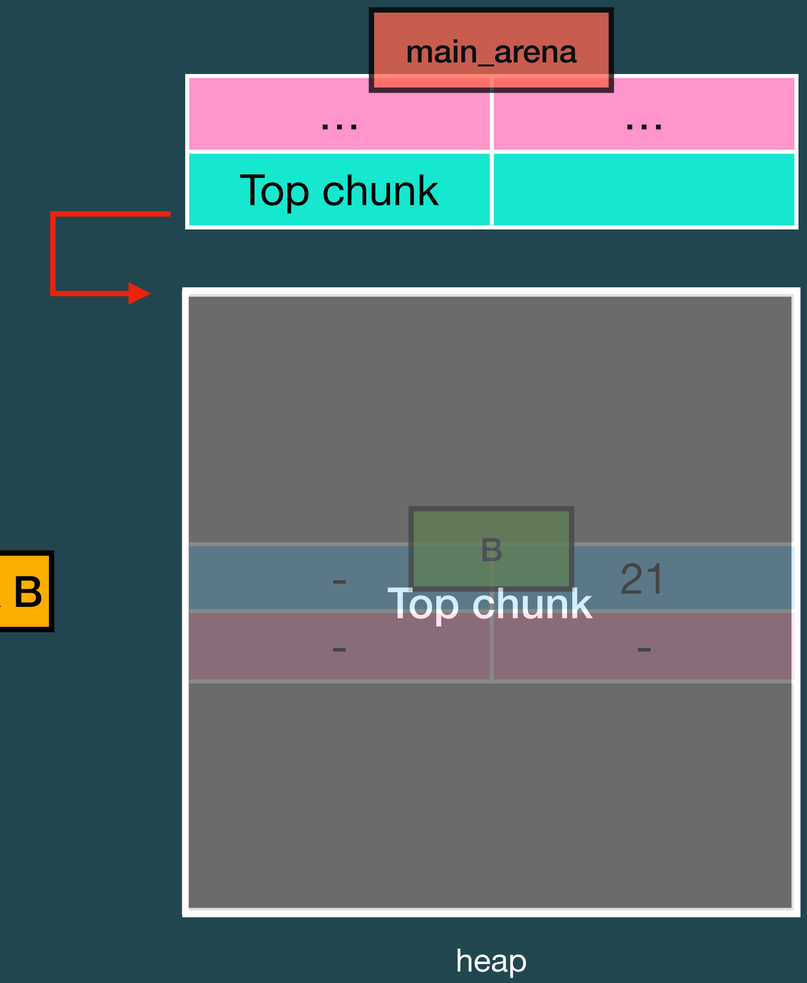


$ Exploitation tech

Overlapping chunks

```
u1f383@u1f383:/
$
A = malloc(0x410);
B = malloc(0x10);

*(A-1) = 0x421 + 0x20; // 0x20 == B 的 chunk size

A = malloc(0x430);
total = (0x430 / 8);
A[total - 2] = 0xdeadbeef;
B[0] == 0xdeadbeef;
```

此時 B 已經包含在 top chunk，分配 0x440 大小的 chunk 會包含 chunk B

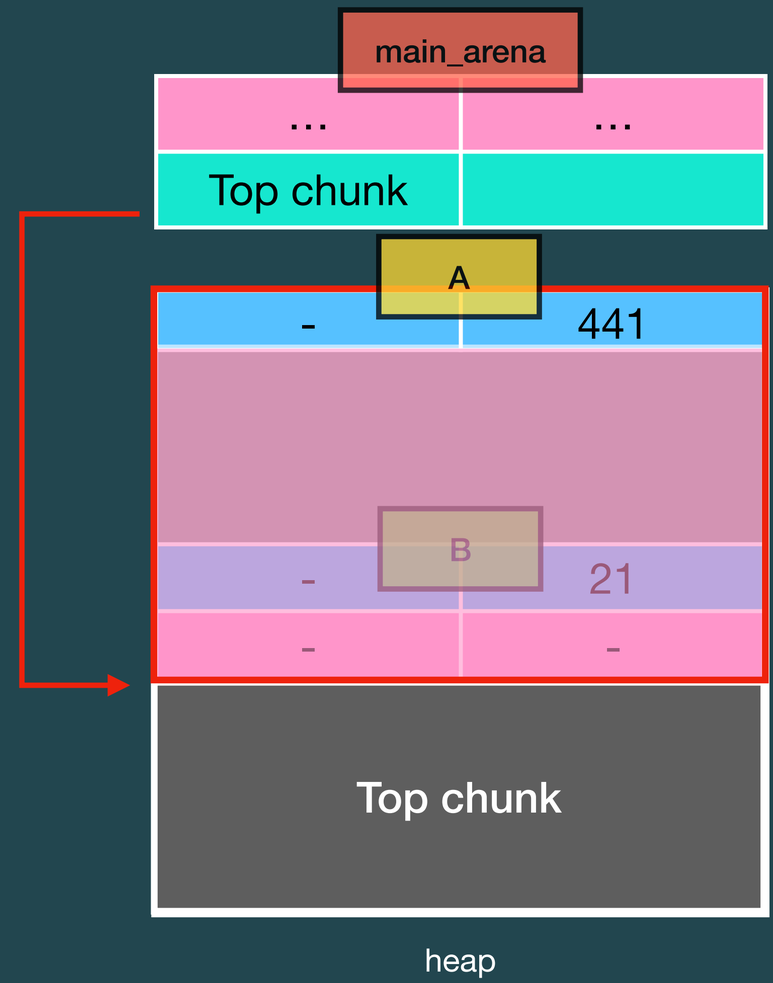


$ Exploitation tech

Overlapping chunks

```
u1f383@u1f383:/
$
A = malloc(0x410);
B = malloc(0x10);

*(A-1) = 0x421 + 0x20; // 0x20 == B 的 chunk size
free(A); // consolidate to top chunk
A = malloc(0x430);
total = (0x430 / 8);
A[total - 2] = 0xdeadbeef;
B[0] == 0xdeadbeef;
```

當修改 chunk A 時，chunk B 也會受到影響

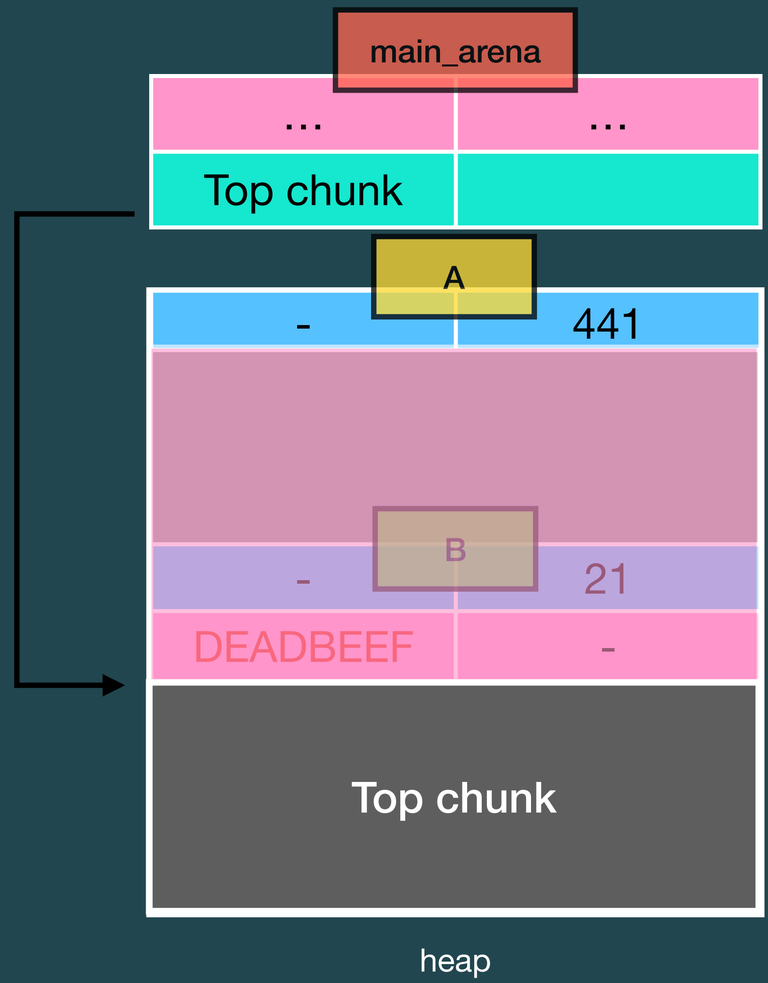


$ Exploitation tech

Overlapping chunks

```
u1f383@u1f383:/
$
A = malloc(0x410);
B = malloc(0x10);

*(A-1) = 0x421 + 0x20; // 0x20 == B 的 chunk size
free(A); // consolidate to top chunk

A = mal
total =
A[total
B[0] == 0xdeadbeef;
```

此範例中單純蓋寫 chunk B 的資料，
而 size 或是 fd 也是很好的利用點




u1f383@u1f383:/
⌥⌘1
$
$ Thanks for listening 🎃

$ Appendix

Resources

- how2heap - heap exploitation tech collection