Heap Exploitation Primer

kevin47@Balsn

Outline

- GOT Review
- Ptmalloc Overview
- Common Vulnerabilities
- Heap Exploitation Techniques

GOT Review

GOT Workflow Demo

GOT Hijack

https://ppt.cc/fMXpEx

Ptmalloc Overview

- How heap memory is obtained from kernel?
- How efficiently memory is managed?
- Is it managed by kernel or by library or by application itself?
- Can heap memory be exploited?

Ptmalloc Overview

- dlmalloc – General purpose allocator
- ptmalloc2 – glibc
- jemalloc – FreeBSD and Firefox
- tcmalloc – Google
- libumem – Solaris
- …

Ptmalloc Overview

- 大家如果看過或聽過 C++ Primer，就會知道 Primer 都是放假的
- 這次主題既然都叫 Heap Exploitation Primer 了，代表接下來會很難

  大家加油 (๑•̀ω•́)و

- 本投影片如未特別註明都以 64 位元電腦 glibc-2.23 為主，換到 32 位元則大多數的欄位大小都須除二

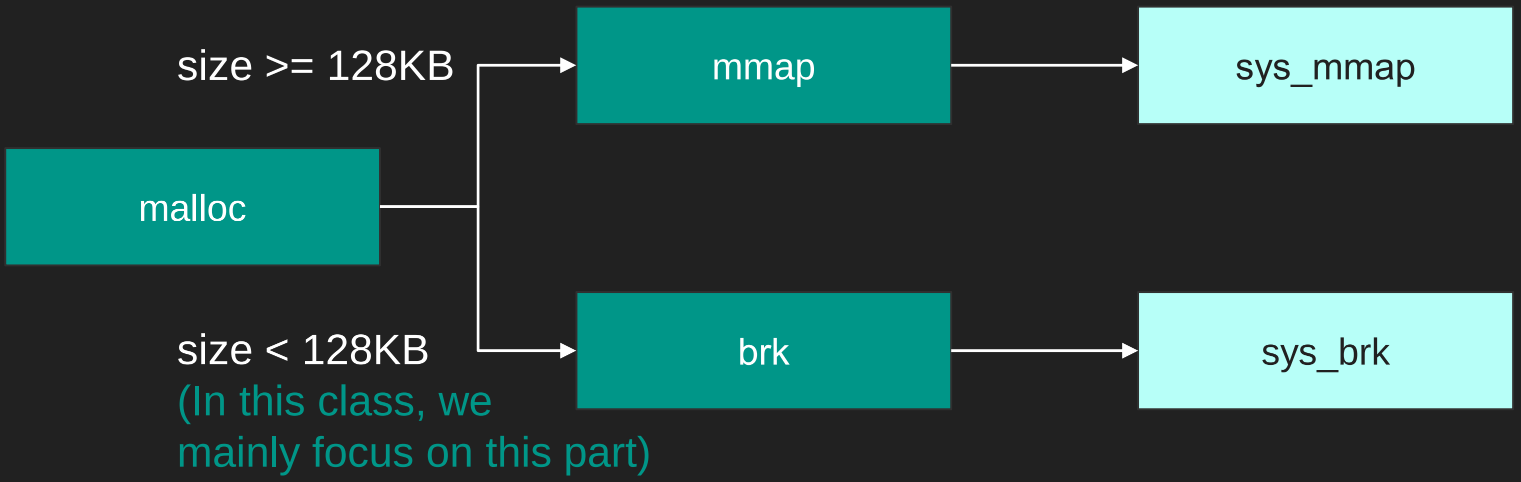
What is malloc

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

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(){
    int len;
    puts("Enter length:");
    scanf("%d", &len);
    // plus 1 for ending null byte
    char *str = (char *)malloc(len+1);
    puts("Enter string:");
    read(0, str, len);
    printf("Hi, %s\n", str);
    free(str);
    // Important! Not clearing pointer could lead to UAF!!!
    str = 0;
}
```

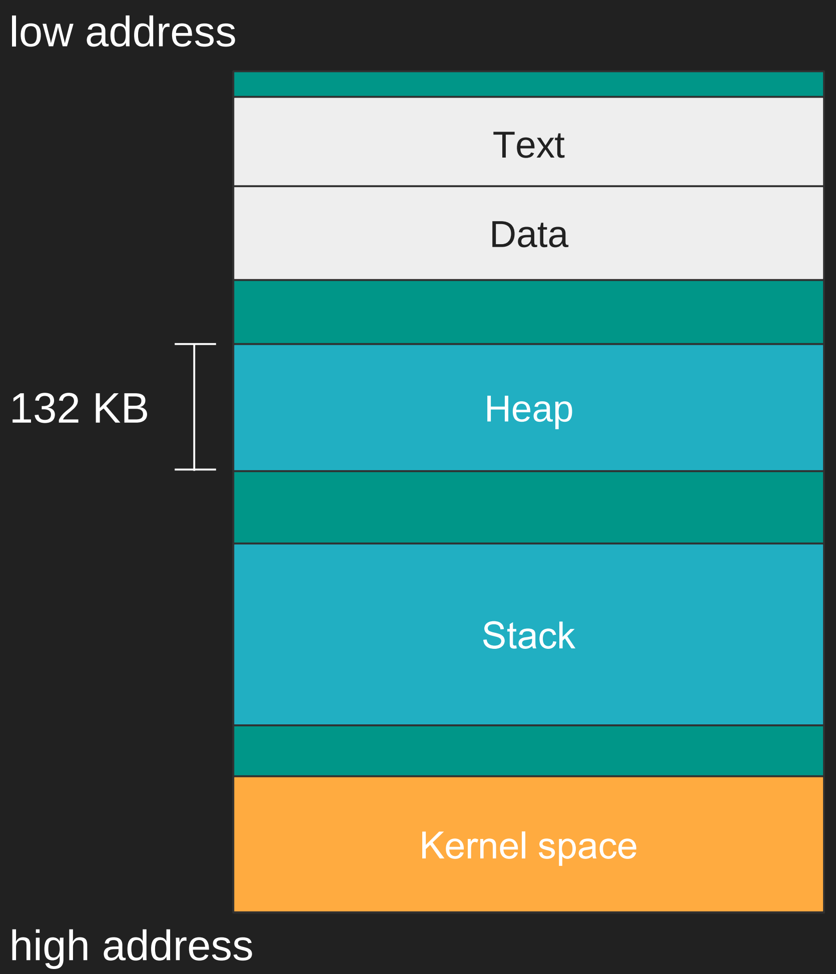
Malloc Workflow

- 第一次執行 malloc



Arenas

- main_arena
  - 無論一開始 malloc 多少空間（ < 128 KB ）kernel 都會給 132 KB 的 heap segment (rw) 這個部分稱為 main arena
- thread_arena
  - 基本上跟 main_arena 一樣，只不過是每個 thread 一個



Main Arena Header

```
gdb-peda$ ptype main_arena
type = struct malloc_state {
    mutex_t mutex;
    int flags;
    mfastbinptr fastbinsY[10];
    mchunkptr top;
    mchunkptr last_remainder;
    mchunkptr bins[254];
    unsigned int binmap[4];
    struct malloc_state *next;
    struct malloc_state *next_free;
    size_t attached_threads;
    size_t system_mem;
    size_t max_system_mem;
}
```

```
gdb-peda$ p/x main_arena
$3 = {
  mutex = 0x0,
  flags = 0x1,
  fastbinsY = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0},
  top = 0x555555758060,
  last_remainder = 0x0,
  bins = {0x7ffff7dd1b78, 0x7ffff7dd1b78, 0x7ffff7dd1b88, 0x7ffff7dd1b88, 0x7ffff7dd1b98, 0x7ffff7dd1b98, 0x7ffff7dd1ba8,
    0x7ffff7dd1ba8, 0x7ffff7dd1bb8, 0x7ffff7dd1bb8, 0x7ffff7dd1bc8, 0x7ffff7dd1bc8, 0x7ffff7dd1bd8, 0x7ffff7dd1bd8,
    0x7ffff7dd1be8, 0x7ffff7dd1be8, 0x7ffff7dd1bf8, 0x7ffff7dd1bf8, 0x7ffff7dd1c08, 0x7ffff7dd1c08, 0x7ffff7dd1c18,
    0x7ffff7dd1c18, 0x7ffff7dd1c28, 0x7ffff7dd1c28, 0x7ffff7dd1c38, 0x7ffff7dd1c38, 0x7ffff7dd1c48, 0x7ffff7dd1c48,
    0x7ffff7dd1c58, 0x7ffff7dd1c58, 0x7ffff7dd1c68, 0x7ffff7dd1c68, 0x7ffff7dd1c78, 0x7ffff7dd1c78, 0x7ffff7dd1c88,
    0x7ffff7dd1c88, 0x7ffff7dd1c98, 0x7ffff7dd1c98, 0x7ffff7dd1ca8, 0x7ffff7dd1ca8, 0x7ffff7dd1cb8, 0x7ffff7dd1cb8,
    0x7ffff7dd1cc8, 0x7ffff7dd1cc8, 0x7ffff7dd1cd8, 0x7ffff7dd1cd8, 0x7ffff7dd1ce8, 0x7ffff7dd1ce8, 0x7ffff7dd1cf8,
    0x7ffff7dd1cf8, 0x7ffff7dd1d08, 0x7ffff7dd1d08, 0x7ffff7dd1d18, 0x7ffff7dd1d18, 0x7ffff7dd1d28, 0x7ffff7dd1d28,
    0x7ffff7dd1d38, 0x7ffff7dd1d38, 0x7ffff7dd1d48, 0x7ffff7dd1d48, 0x7ffff7dd1d58, 0x7ffff7dd1d58, 0x7ffff7dd1d68,
    0x7ffff7dd1d68, 0x7ffff7dd1d78, 0x7ffff7dd1d78, 0x7ffff7dd1d88, 0x7ffff7dd1d88, 0x7ffff7dd1d98, 0x7ffff7dd1d98,
    0x7ffff7dd1da8, 0x7ffff7dd1da8, 0x7ffff7dd1db8, 0x7ffff7dd1db8, 0x7ffff7dd1dc8, 0x7ffff7dd1dc8, 0x7ffff7dd1dd8,
    0x7ffff7dd1dd8, 0x7ffff7dd1de8, 0x7ffff7dd1de8, 0x7ffff7dd1df8, 0x7ffff7dd1df8, 0x7ffff7dd1e08, 0x7ffff7dd1e08,
    0x7ffff7dd1e18, 0x7ffff7dd1e18, 0x7ffff7dd1e28, 0x7ffff7dd1e28, 0x7ffff7dd1e38, 0x7ffff7dd1e38, 0x7ffff7dd1e48,
    0x7ffff7dd1e48, 0x7ffff7dd1e58, 0x7ffff7dd1e58, 0x7ffff7dd1e68, 0x7ffff7dd1e68, 0x7ffff7dd1e78, 0x7ffff7dd1e78,
    0x7ffff7dd1e88, 0x7ffff7dd1e88, 0x7ffff7dd1e98, 0x7ffff7dd1e98, 0x7ffff7dd1ea8, 0x7ffff7dd1ea8, 0x7ffff7dd1eb8,
    0x7ffff7dd1eb8, 0x7ffff7dd1ec8, 0x7ffff7dd1ec8, 0x7ffff7dd1ed8, 0x7ffff7dd1ed8, 0x7ffff7dd1ee8, 0x7ffff7dd1ee8,
    0x7ffff7dd1ef8, 0x7ffff7dd1ef8, 0x7ffff7dd1f08, 0x7ffff7dd1f08, 0x7ffff7dd1f18, 0x7ffff7dd1f18, 0x7ffff7dd1f28,
    0x7ffff7dd1f28, 0x7ffff7dd1f38, 0x7ffff7dd1f38, 0x7ffff7dd1f48, 0x7ffff7dd1f48, 0x7ffff7dd1f58, 0x7ffff7dd1f58,
    0x7ffff7dd1f68, 0x7ffff7dd1f68, 0x7ffff7dd1f78, 0x7ffff7dd1f78, 0x7ffff7dd1f88, 0x7ffff7dd1f88, 0x7ffff7dd1f98,
    0x7ffff7dd1f98, 0x7ffff7dd1fa8, 0x7ffff7dd1fa8, 0x7ffff7dd1fb8, 0x7ffff7dd1fb8, 0x7ffff7dd1fc8, 0x7ffff7dd1fc8,
    0x7ffff7dd1fd8, 0x7ffff7dd1fd8, 0x7ffff7dd1fe8, 0x7ffff7dd1fe8, 0x7ffff7dd1ff8, 0x7ffff7dd1ff8, 0x7ffff7dd2008,
    0x7ffff7dd2008, 0x7ffff7dd2018, 0x7ffff7dd2018, 0x7ffff7dd2028, 0x7ffff7dd2028, 0x7ffff7dd2038, 0x7ffff7dd2038,
    0x7ffff7dd2048, 0x7ffff7dd2048, 0x7ffff7dd2058, 0x7ffff7dd2058, 0x7ffff7dd2068, 0x7ffff7dd2068, 0x7ffff7dd2078,
    0x7ffff7dd2078, 0x7ffff7dd2088, 0x7ffff7dd2088, 0x7ffff7dd2098, 0x7ffff7dd2098, 0x7ffff7dd20a8, 0x7ffff7dd20a8,
    0x7ffff7dd20b8, 0x7ffff7dd20b8, 0x7ffff7dd20c8, 0x7ffff7dd20c8, 0x7ffff7dd20d8, 0x7ffff7dd20d8, 0x7ffff7dd20e8,
    0x7ffff7dd20e8, 0x7ffff7dd20f8, 0x7ffff7dd20f8, 0x7ffff7dd2108, 0x7ffff7dd2108, 0x7ffff7dd2118, 0x7ffff7dd2118,
    0x7ffff7dd2128, 0x7ffff7dd2128, 0x7ffff7dd2138, 0x7ffff7dd2138, 0x7ffff7dd2148, 0x7ffff7dd2148, 0x7ffff7dd2158,
    0x7ffff7dd2158, 0x7ffff7dd2168, 0x7ffff7dd2168, 0x7ffff7dd2178, 0x7ffff7dd2178, 0x7ffff7dd2188, 0x7ffff7dd2188,
    0x7ffff7dd2198, 0x7ffff7dd2198, 0x7ffff7dd21a8, 0x7ffff7dd21a8...},
  binmap = {0x0, 0x0, 0x0, 0x0},
  next = 0x7ffff7dd1b20,
  next_free = 0x0,
  attached_threads = 0x1,
  system_mem = 0x21000,
  max_system_mem = 0x21000
}
```

Chunks

- glibc 在實作記憶體管理時的 data structure
- 在 malloc 時所分配出去的空間即為一個 chunk (最小為 SIZE_T*4)
  - SIZE_T = unsigned long int
- malloc(n) 實際上分配的大小是：(n+8) 向上取整到 16 的倍數
- 如果該 chunk 被 free 則會將 chunk 加入名為 bin 的 linked list

Chunks

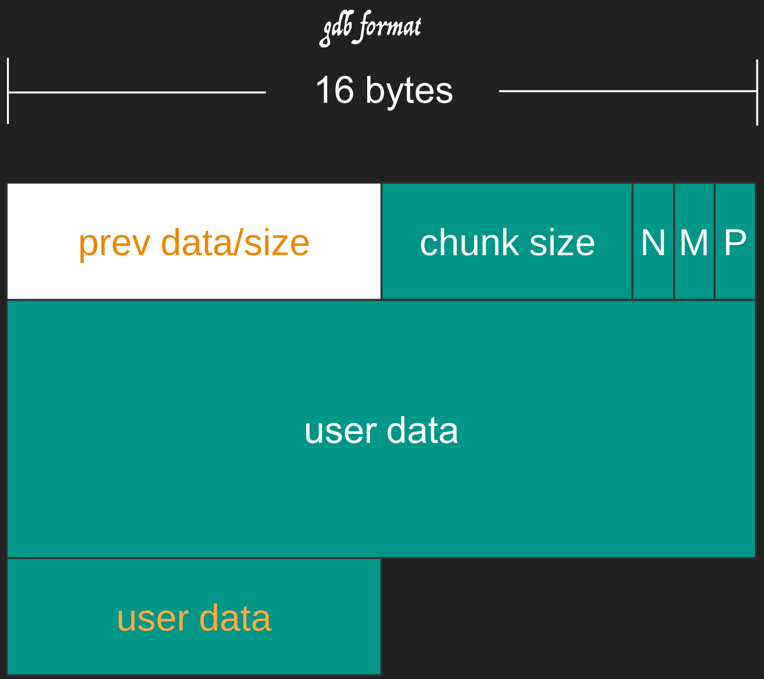
malloc(n) 對應到實際的 chunk size 為 (n+8) 向上取到 16 的倍數

分為：

- Allocated chunk
- Free chunk
- Top chunk

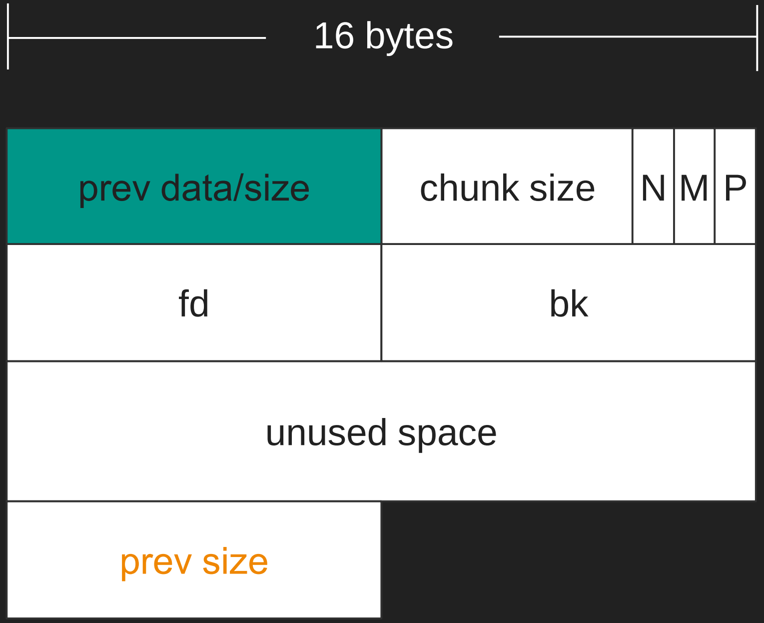
Allocated chunk

- Previous size/data：
  - 若前一個 chunk 使用中，則此欄位為前一個 chunk 的 data
  - 若前一個 chunk 為 free，則此欄位為前一個 chunk 的大小
- Chunk size 的 3 個 LSB 為 flag
- PREV_INUSE(P)：前一個 chunk 使用中則為 1
- IS_MMAPPED(M)：chunk 為 mmap 出來的則為1
- NON_MAIN_ARENA(N)：chunk 屬於 thread arena 而非 main arena 時為 1



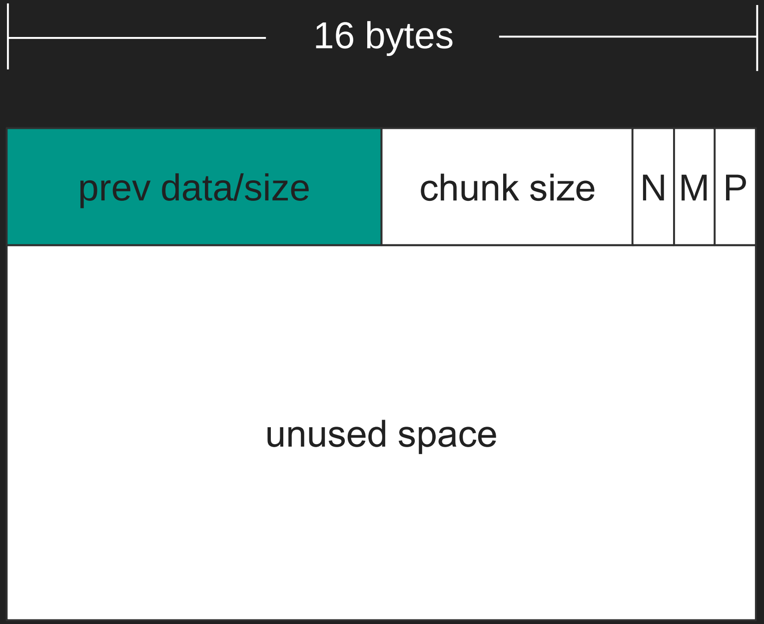
Free chunk

- Previous size/data 和 chunk size 跟 allocated chunk 一樣
- 下一個 chunk 的 P 為 0
- 多了 fd 和 bk 兩個欄位
  - Fd 指向同一 bin 當中的下一個 chunk
  - Bk 指向同一 bin 當中的前一個 chunk



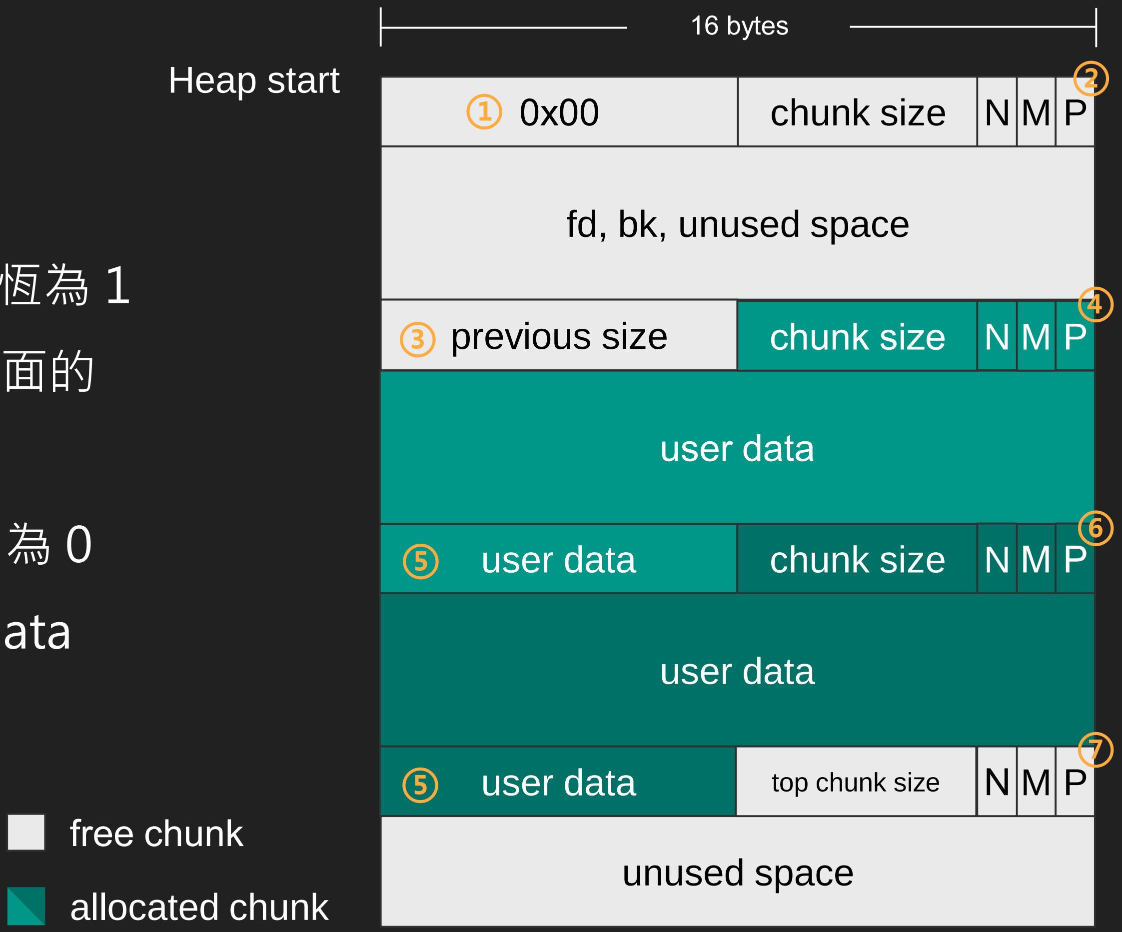
Top chunk

- 第一次 malloc 時就會將 heap 切成兩塊 chunk
- 第一塊 chunk 就是分配出 去的 chunk
- 剩下的空間視為 top chunk，之後要是分配空間不足時將會由 top chunk 切出去
- P 恆為 1
- Top 上一個 chunk 被 free 時，大小大於等於 0x90 時則直接被 top chunk merge



All combined

- ① 沒使用的空間，恆為 0
- ② 因為前面沒 chunk，因此恆為 1
- ③ chunk 是 free，因此為前面的 chunk_size (沒有 N, M, P)
- ④ 前一個 chunk 是 free，P 為 0
- ⑤ chunk 使用中，為 user data
- ⑥ chunk 使用中，P 為 1
- ⑦ top chunk P 恆為 1



Bins

Bin 是儲存 free chunk 的資料結構，根據 free chunk 大小分為：

- Fast bin
- Small bin
- Large bin
- Unsorted bin

Fast bin

因為*size*太小，為了效率所以不改*flag*, 只加上*fd*

- Singly linked list (只有 fd 沒有 bk)
- Chunk size < 144 bytes (0x90)
- Free 掉時，不會將下一個 chunk 的 P 設成 0
- 依據 bin 中所存的 chunk 大小，再分為 10 個 fast bin 分別為 size 0x20, 0x30, ..., 0xb0，預設只有用到 0x80，再大就是 small bin
- LIFO *stack* 結構，*last in first out* *can see heap information in peda-gdb with heapinfo*

Small bin

small bin 如果有相鄰的free chunk (also small bin), they will merge into one chunk.

- Circular doubly linked list (有 fd, bk)
- Chunk size < 1024 bytes (0x400)
- Free 掉時，會將下一個 chunk 的 P 設成 0，用來檢查有沒有 double free
- 依據大小分為 62 個 fast bin 分別為 size 0x20, 0x30 , ..., 0x3f0
  - 0x20 ~ 0x80 的部份大小會跟 fast bin 重疊，這時候 chunk 會視情況在 fast bin 或 small bin
- FIFO *queue structure*

重疊：看到一個0x20~0x80大小的chunk，他有可能是small bin or fast bin

同理，small bin裡面可能含有0x20~0x80大小的chunk

Large bin

- Circular doubly linked list (sorted list)
- Chunk size >= 1024 bytes (0x400)
- Free 掉時，會將下一個 chunk 的 P 設成 0，用來檢查有沒有 double free
- 除了 fd, bk 之外，多了兩個欄位 fd_nextsize, bk_nextsize，指向同一個 bin 裡前/下一個大小的 chunk

Large bin

- 不再是同一個 bins 裡面大小固定
- 63 個 bins
  - 32 個 bins 大小一次增加 64
  - 16 個 bins 大小一次增加 512
  - 8 個 bins 大小一次增加 4096
  - 4 個 bins 大小一次增加 32768
  - 2 個 bins 大小一次增加 262144
  - 1 個 bin 裝剩於所有大小

Unsorted bin

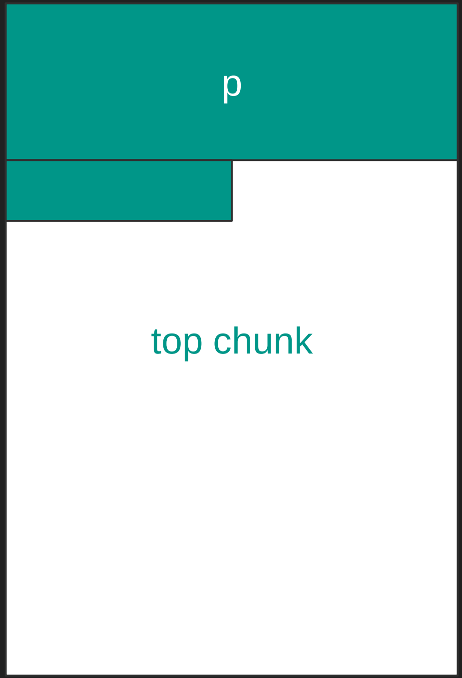
free 完以後，所有大於*fast bin*的*bin*都會被丟到*unsorted bin*暫放
等到下次*malloc*時才會順便把*bin*依照大小放到各自的*bin*裏

- Circular doubly linked list (有 fd, bk)
- 當要 free 的 chunk 大小超過 fast bin 時，為了效率，因此不會直接放到對應的 bin 裡，而是先丟到 unsorted bin
- 下次 malloc 時，會先去 unsorted bin 找大小一模一樣的 chunk，同時把大小不一樣的丟到對應的 bins
- Unsorted bin 裡沒找到大小一樣的 chunk 才會去對應的 bin 當中尋找大於等於所需大小的的 chunk，切割完剩下的部份會再被丟到 unsorted bin

Why small bin has size of 0x20 ~ 0x80

void *p = malloc(0x50);

free(p);



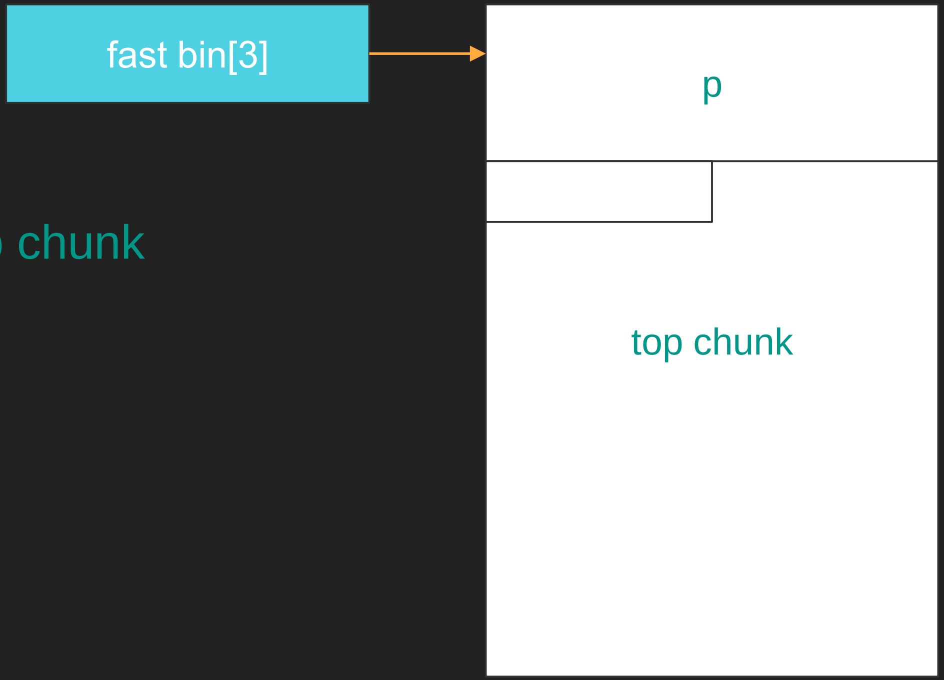
Why small bin has size of 0x20 ~ 0x80

void *p = malloc(0x50);

free(p);

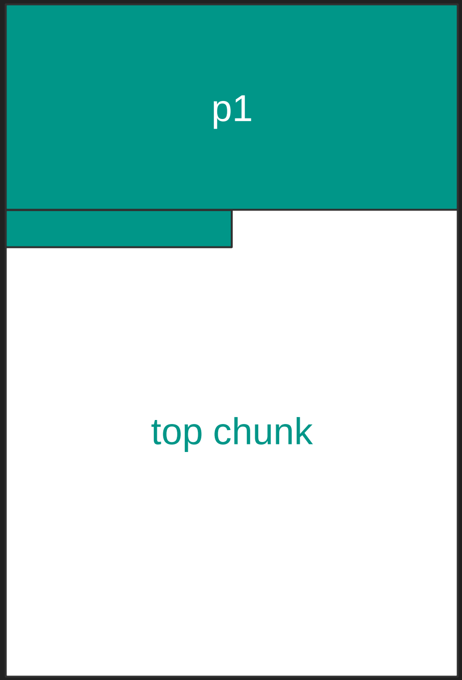
// p 大小在 fastbin 因此不會被 top chunk

merge



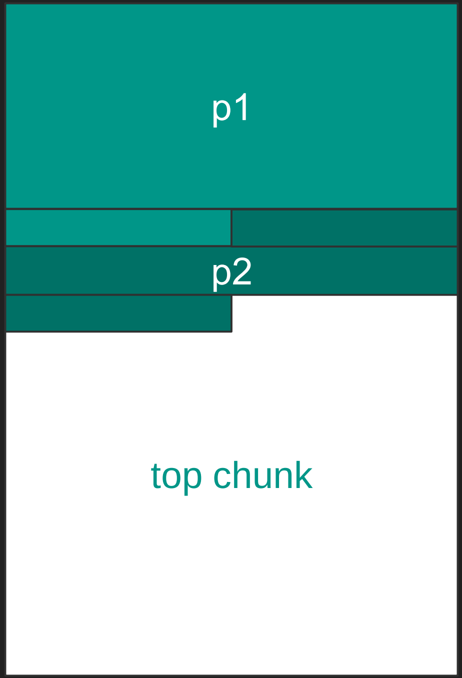
Why small bin has size of 0x20 ~ 0x80

```
void *p1 = malloc(0x98);

void *p2 = malloc(0x10);

free(p1);

void *p3 = malloc(0x48);

void *p4 = malloc(0x200);
```



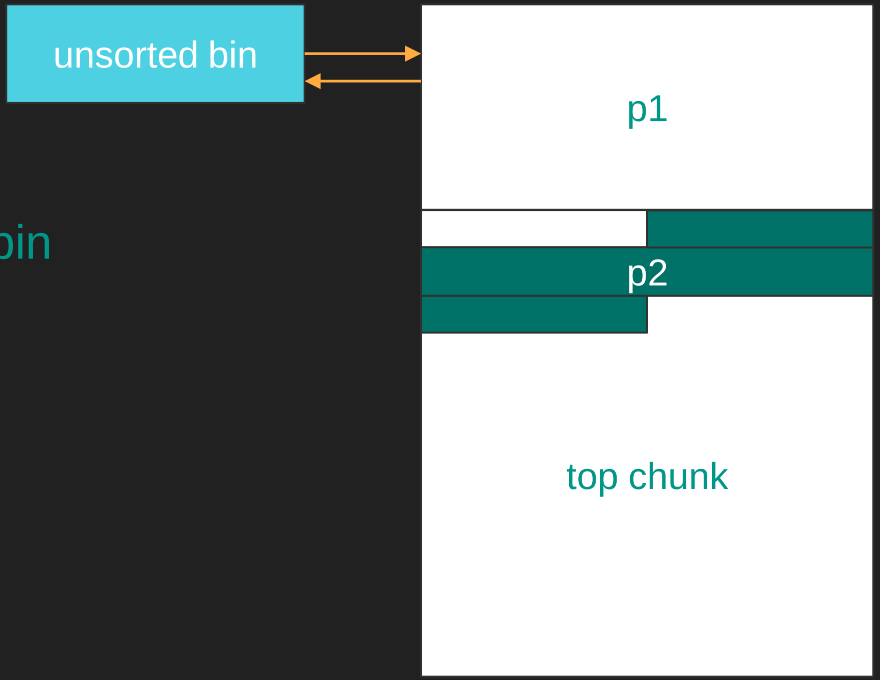
Why small bin has size of 0x20 ~ 0x80

```
void *p1 = malloc(0x98);

void *p2 = malloc(0x10);

// 沒有這行 free(p1) 會被 top chunk 吃掉

free(p1);

void *p3 = malloc(0x48);

void *p4 = malloc(0x200);
```



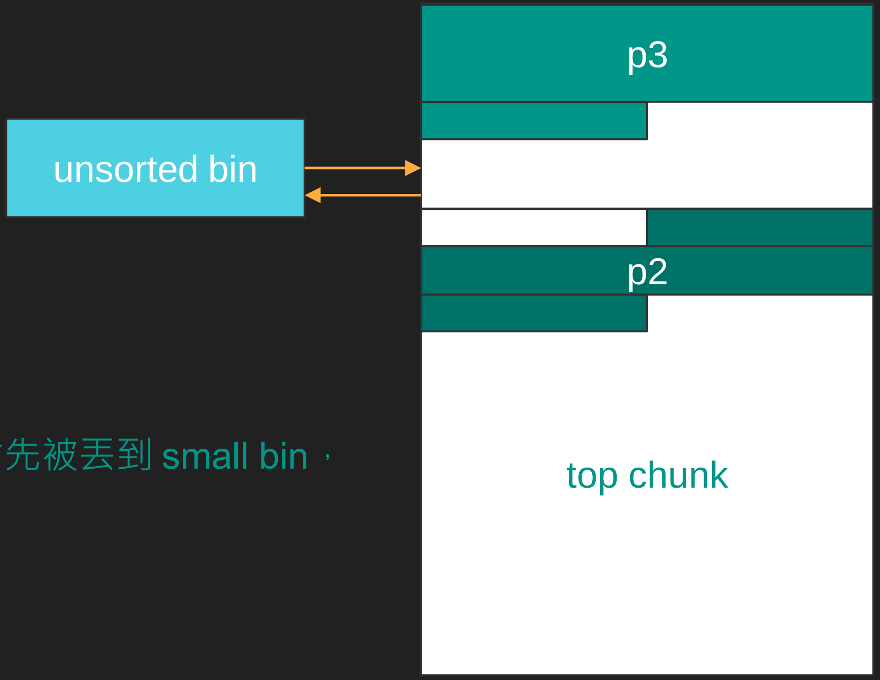
Why small bin has size of 0x20 ~ 0x80

```
void *p1 = malloc(0x98);

void *p2 = malloc(0x10);

free(p1); // size>=0x90, to unsorted bin

void *p3 = malloc(0x48);

void *p4 = malloc(0x200);
```



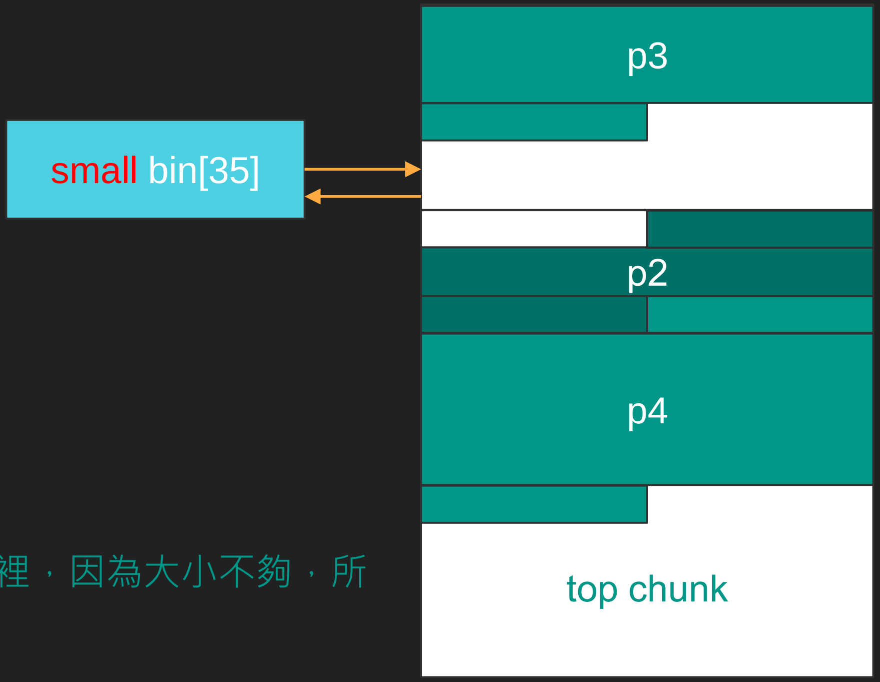
Why small bin has size of 0x20 ~ 0x80

```
void *p1 = malloc(0x98);

void *p2 = malloc(0x10);

free(p1);

void *p3 = malloc(0x48);

// unsorted bin 因為大小不是剛好0x50，所以會先被丟到 small bin，

small bin 切完剩下的才回到 unsorted bin

void *p4 = malloc(0x200);
```



Why small bin has size of 0x20 ~ 0x80

```
void *p1 = malloc(0x98);

void *p2 = malloc(0x10);

free(p1);

void *p3 = malloc(0x48);

void *p4 = malloc(0x200);

// malloc 把經過的 unsorted bin 丟到 small bin 裡，因為大小不夠，所以從 top chunk 切
```



Brief sum up

- Arenas
- Chunks
  - Allocated chunk
  - Free chunk
  - Top chunk
- Bins
  - Fast bin
  - Small bin
  - Large bin
  - Unsorted bin

Common Vulnerabilities

- Use After Free
- Heap Overflow

Use After Free (UAF)

- 當 free 完之後，並未將 pointer 設成 null 而繼續 使用該 pointer 該 pointer 稱為 dangling pointer
- 根據 use 方式不同而有不同的行為，可能造成任 意位置讀取或是任意位置寫入，進一步造成控制 程式流程
- free(p);
- p = NULL; // 沒有這行的話就有可能有 UAF

Heap Overflow

- 在 heap 段中發生的 buffer overflow
- 通常無法直接控制 rip 但可以利用蓋下一個 chunk 的 size, fd 或 bk， 搭配 malloc 內部的機制，達到(幾乎)任意位置寫入，進而控製 rip

heap無法直接控制rip, unlike stack

Heap Exploitation Techinques

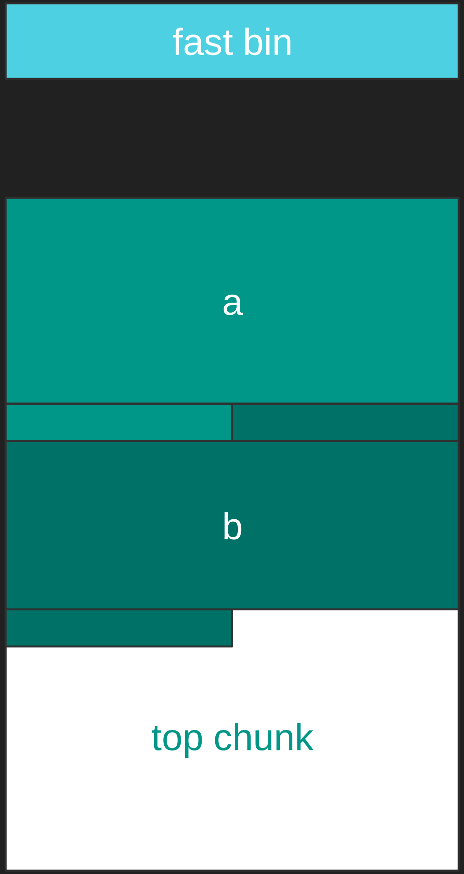
- Fastbin dup attack
- Unsafe unlink
- House of Orange, House of Rabbit, House of Storm, House of Roman, House of Force, House of Lore, ……
- https://github.com/shellphish/how2heap

Fastbin dup attack

- Fast bin 被 free 時，因為不會將下一個 chunk 的 P 設成 0，所以沒辦法直接檢查是否有 double free
- 檢查 double free 的方法是去對應的 fast bin 中，看它的第一個 chunk 是不是自己
- 因此我們可以這樣繞過：

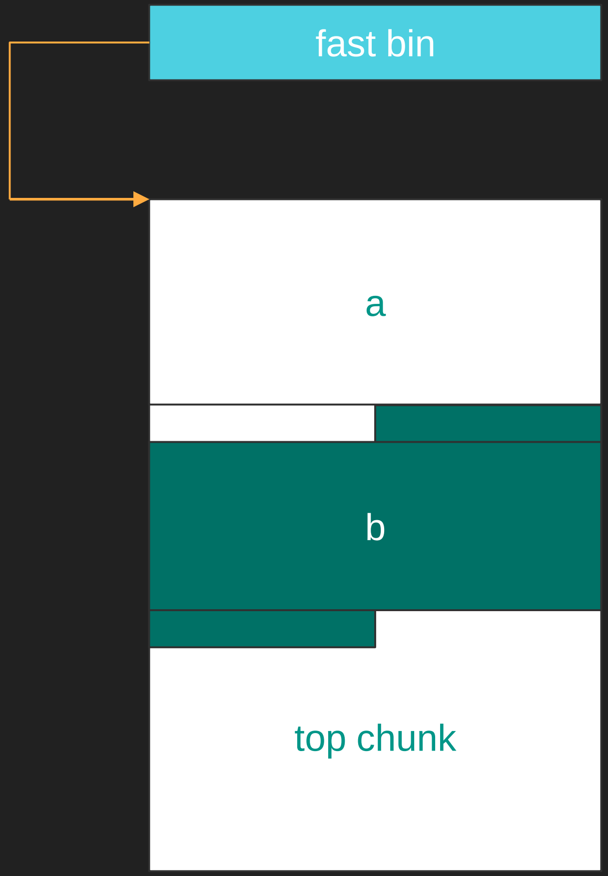
Fastbin dup attack

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

char *b = malloc(0x68);

free(a);

free(b);

free(a);
```



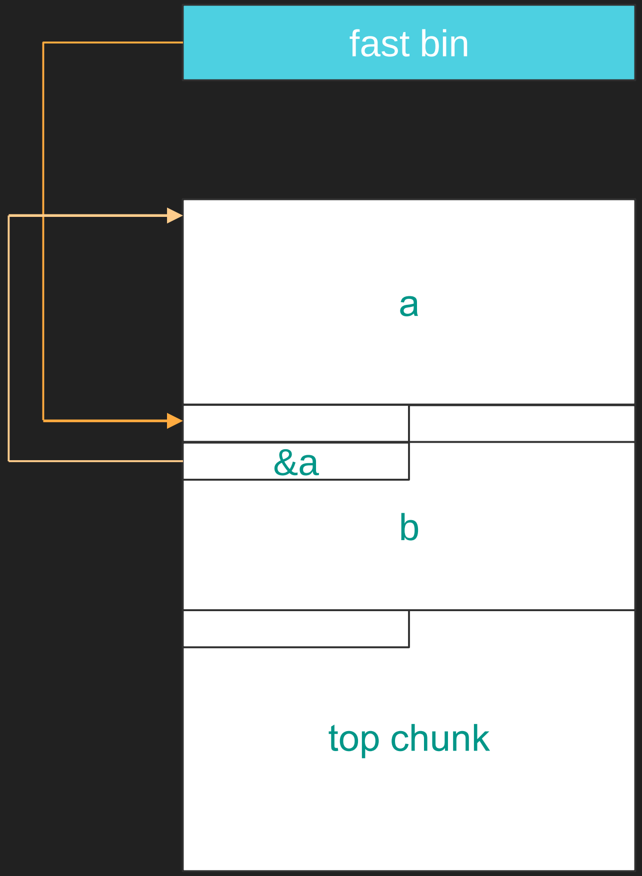
Fastbin dup attack

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

char *b = malloc(0x68);

free(a);

free(b);

free(a);
```



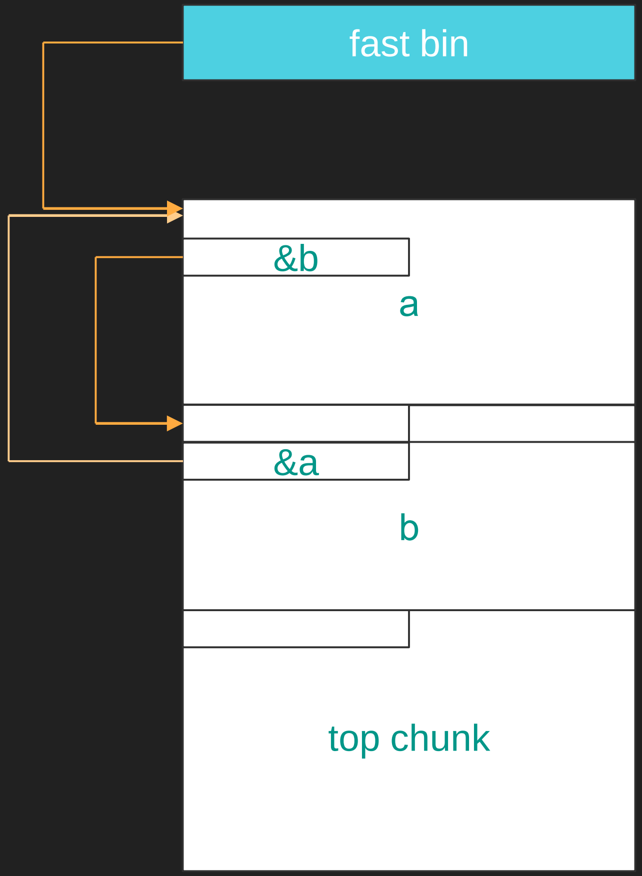
Fastbin dup attack

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

char *b = malloc(0x68);

free(a);

free(b);

// fastbin 不會被 top merge

free(a);
```



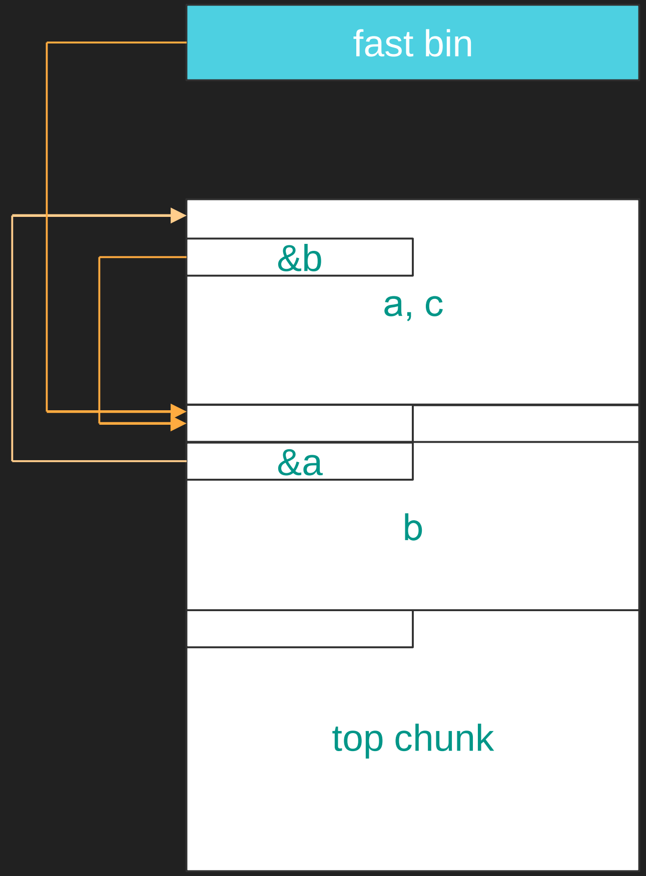
Fastbin dup attack

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

char *b = malloc(0x68);

free(a);

free(b);

free(a);
```



Fastbin dup attack

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

read(0, c, 8);

char *d = malloc(0x68);

char *e = malloc(0x68);

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



Fastbin dup attack

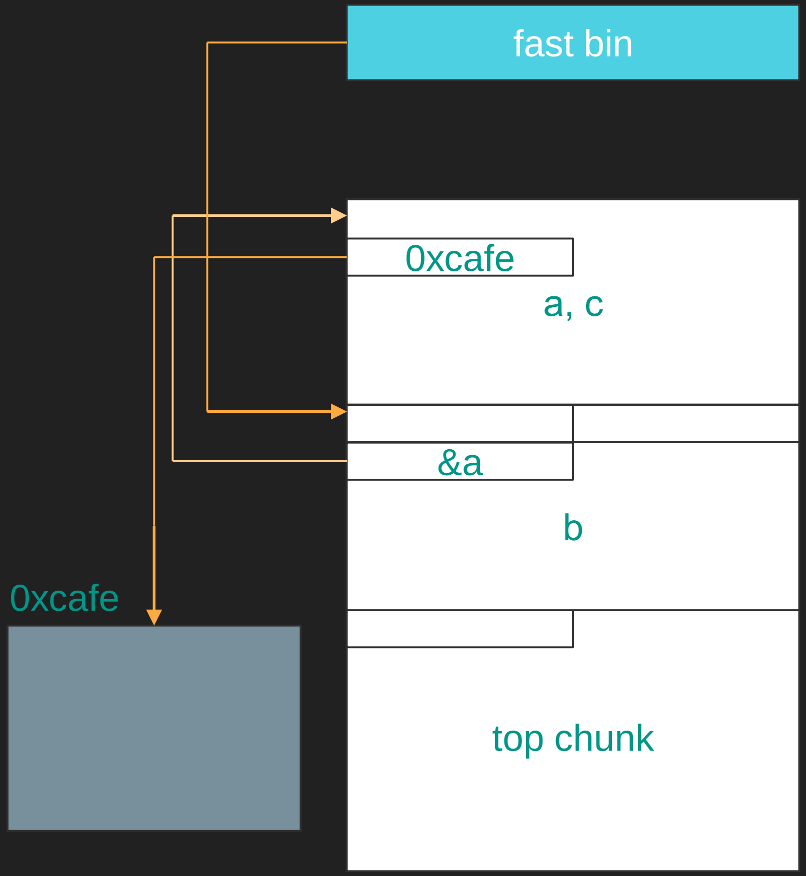
char *c = malloc(0x68);

read(0, c, 8); // 0xcafe

char *d = malloc(0x68);

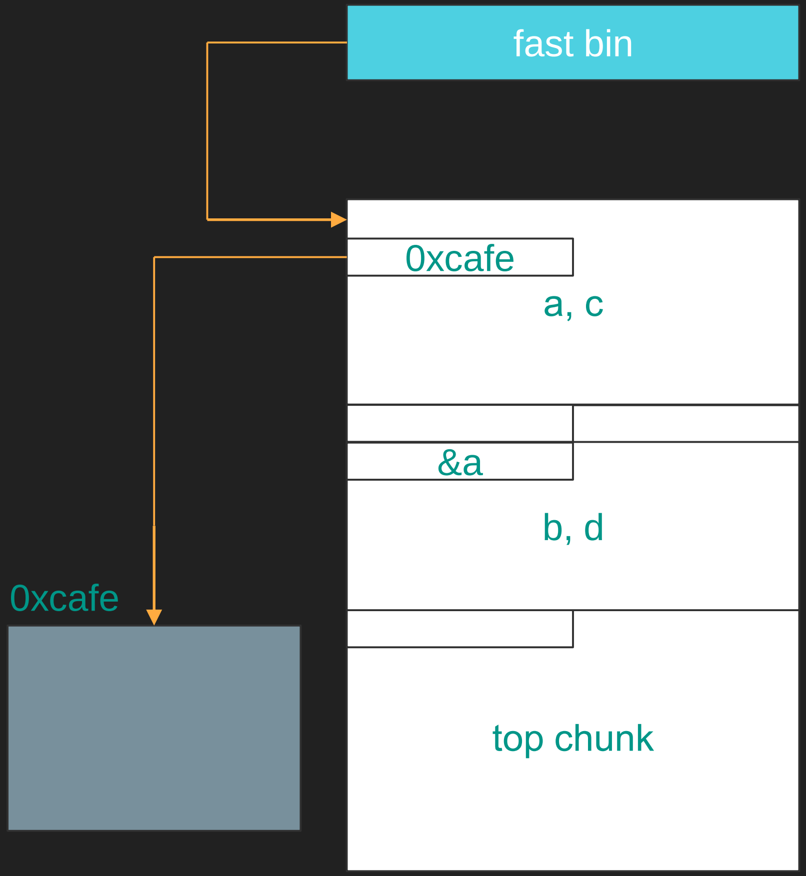
char *e = malloc(0x68);

char *f = malloc(0x68);



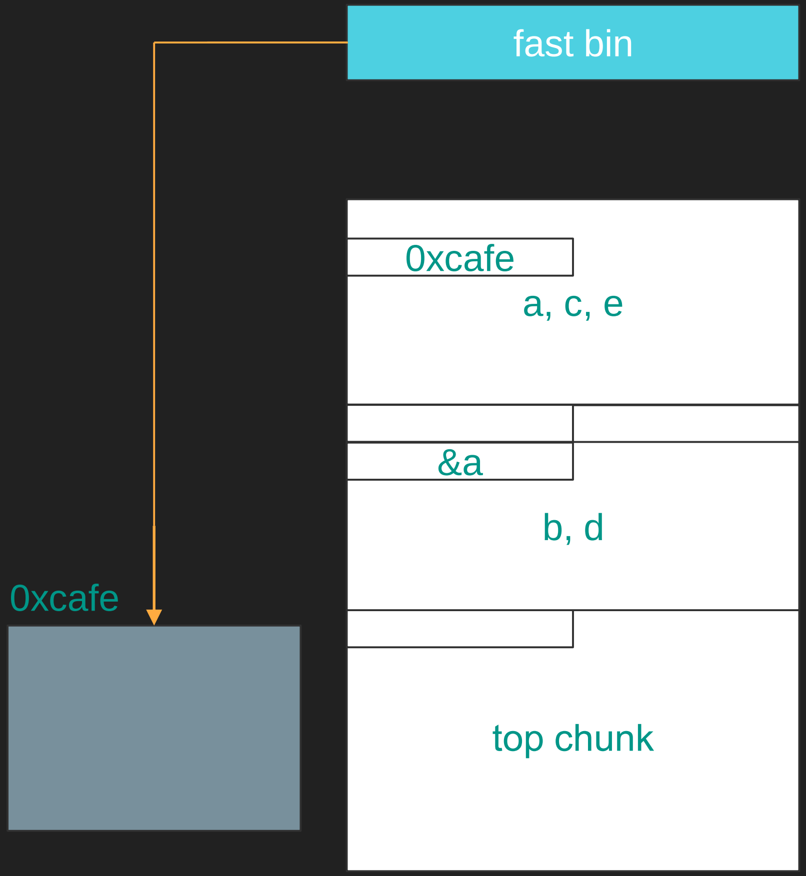
Fastbin dup attack

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

read(0, c, 8);

char *d = malloc(0x68);

char *e = malloc(0x68);

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



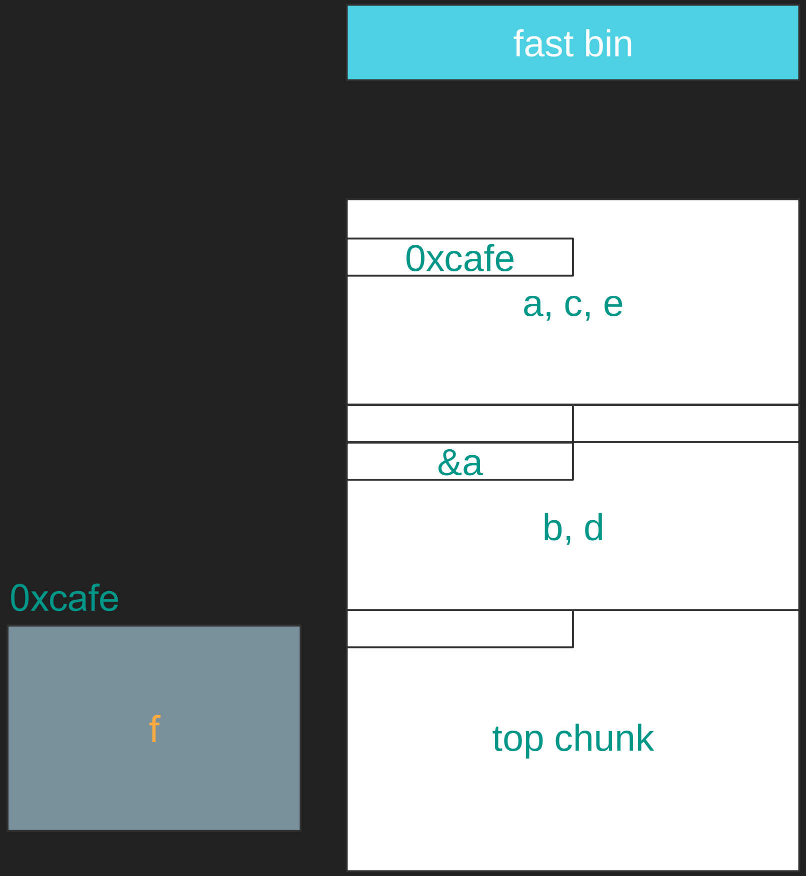
Fastbin dup attack

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

read(0, c, 8);

char *d = malloc(0x68);

char *e = malloc(0x68);

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



Fastbin dup attack

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

read(0, c, 8);

char *d = malloc(0x68);

char *e = malloc(0x68);

char *f = malloc(0x68);

// 寫 f 等同任意位置寫入
```



ref: 1:17:00

Fastbin dup attack

寫入的value要符合fast bin的data structure
因為fast bin會對size做檢查

- 要注意的是，malloc fast bin 時，唯一會做的檢查是看 size 是不是合法的，也就是說 0xcafe 那邊要有一個 0x70，但這樣限制很多，因為不是所有地方都有我們想要的數字

0xcafe + 0x1的位置要填size，0x20~0x80，而0x70是最好利用的

- 我們可以利用 stack、libc address 開頭是 0x7f 這點來增加許多可以 malloc 的點

libc/stack address偏移5bytes 剛好就是0x7f會在LSB

Lab 6-1

Unsafe Unlink

- Free chunk merge
  - 大於 fast bin 在 free 掉時，會跟前後的 free chunk 合併

```
/* consolidate backward */
if (!prev_inuse(p)) {
  prevsize = p->prev_size;
  size += prevsize;
  p = chunk_at_offset(p, -((long) prevsize));
  unlink(p, bck, fwd);
}
```

unlink() macro

- unlink() 是用來把 chunk 移出 bin 用的，從 double linked-list 移除

```
#define unlink(P, BK, FD) {                                         \
    FD = P->fd;                                                     \
    BK = P->bk;                                                     \
    FD->bk = BK;                                                    \
    BK->fd = FD;                                                    \
}
```

Unlink Exploitation

- 如果有 overflow 可以覆蓋到某個 chunk p 的 fd, bk，free q 時傳入 unlink(p)
- 利用 FD->bk = BK 和 BK->fd = FD，可以同時寫入兩個目標，例如:
  - FD = p->fd = free@GOT - 0x18
  - BK = p->bk = &shellcode

```
#define unlink(P, BK, FD) {
    FD = P->fd;
    BK = P->bk;
    FD->bk = BK;
    BK->fd = FD;
}
```

Sadly Truth….QQ

- 實際的 unlink 會檢查 double linked-list 是不是合法，指過去要能指回來，即 p->fd->bk == p

```
#define unlink(P, BK, FD) {                                          \
  FD = P->fd;                                                        \
  BK = P->bk;                                                        \
  if (FD->bk != P || BK->fd != P)                                    \
    malloc_printerr (check_action, "corrupted d...", P);             \
  else {                                                             \
    FD->bk = BK;                                                     \
    BK->fd = FD;                                                     \
  }                                                                  \
}
```

Unsafe Unlink

- 實際上 unlink 的檢查還是有辦法繞過，而且可以得到一種很實用的利用方式，需要有:
  - 一個指向 heap 內的指標
  - 放該指標的位址已知 (例如: 該指標是全域變數)
  - 可以對該指標寫入多次

Unsafe Unlink

- FD = p->fd = &p - 0x18
- BK = p->bk = &p - 0x10

```
#define unlink(P, BK, FD) {                                              \
  FD = P->fd;                                                            \
  BK = P->bk;                                                            \
  if (FD->bk != P || BK->fd != P)                                        \
    malloc_printerr (check_action, "corrupted d...", P);                 \
  else {                                                                 \
    FD->bk = BK;                                                         \
    BK->fd = FD;                                                         \
  }                                                                      \
}
```

- Result: p = &p – 0x18

Miscellaneous

- _malloc_hook, __free_hook, __realloc_hook
- one_gadget

Lab 6-2

References

- https://www.slideshare.net/AngelBoy1/heap-exploitation-51891400
- https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/
- https://blog.csdn.net/Plus_RE/article/details/80211488
- https://sploitfun.wordpress.com/2015/02/26/heap-overflow-using-unlink/
- https://github.com/shellphish/how2heap

Thanks for Listening

~~(Final CTF 可能比 HW 還難)~~

~~(現在停修還來的及)~~

~~(塊陶RRRR~~~~~~)~~