

IERG 4851 – 2026 Fall



Heap Vulnerabilities and Attacks

Yajin ZHOU

<https://yajin.org>

Block A — Observe the heap



malloc() and free()

- malloc(size) returns a pointer to at least size usable bytes
- free(p) returns that memory to the allocator
- p is invalid afterwards — do not read, write, or free it again
- the allocator keeps its own bookkeeping in front of every allocation



malloc() carves chunks out of the top

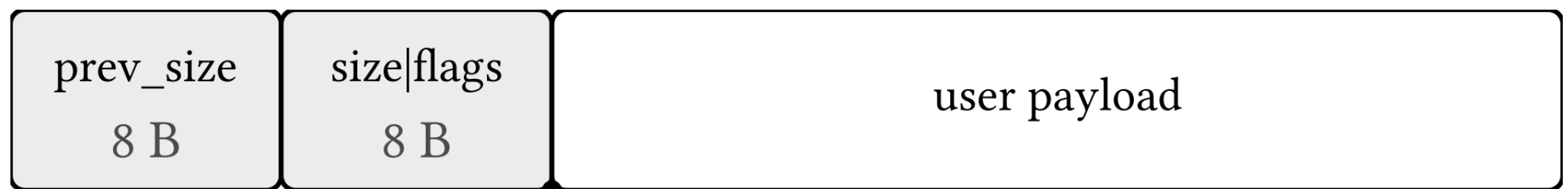
free() puts a chunk back on a free list

freed chunks are not returned to the kernel — they wait here, on a free list



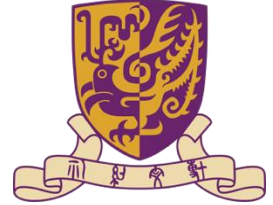
The chunk header

- every allocation is a chunk: prev_size, then **size | flags**, then your data
- malloc() returns the address of the first payload byte, not the header
- size is rounded up to a multiple of 16, and its low 3 bits are flags
- malloc(16) → a 0x20-byte chunk → size field 0x21

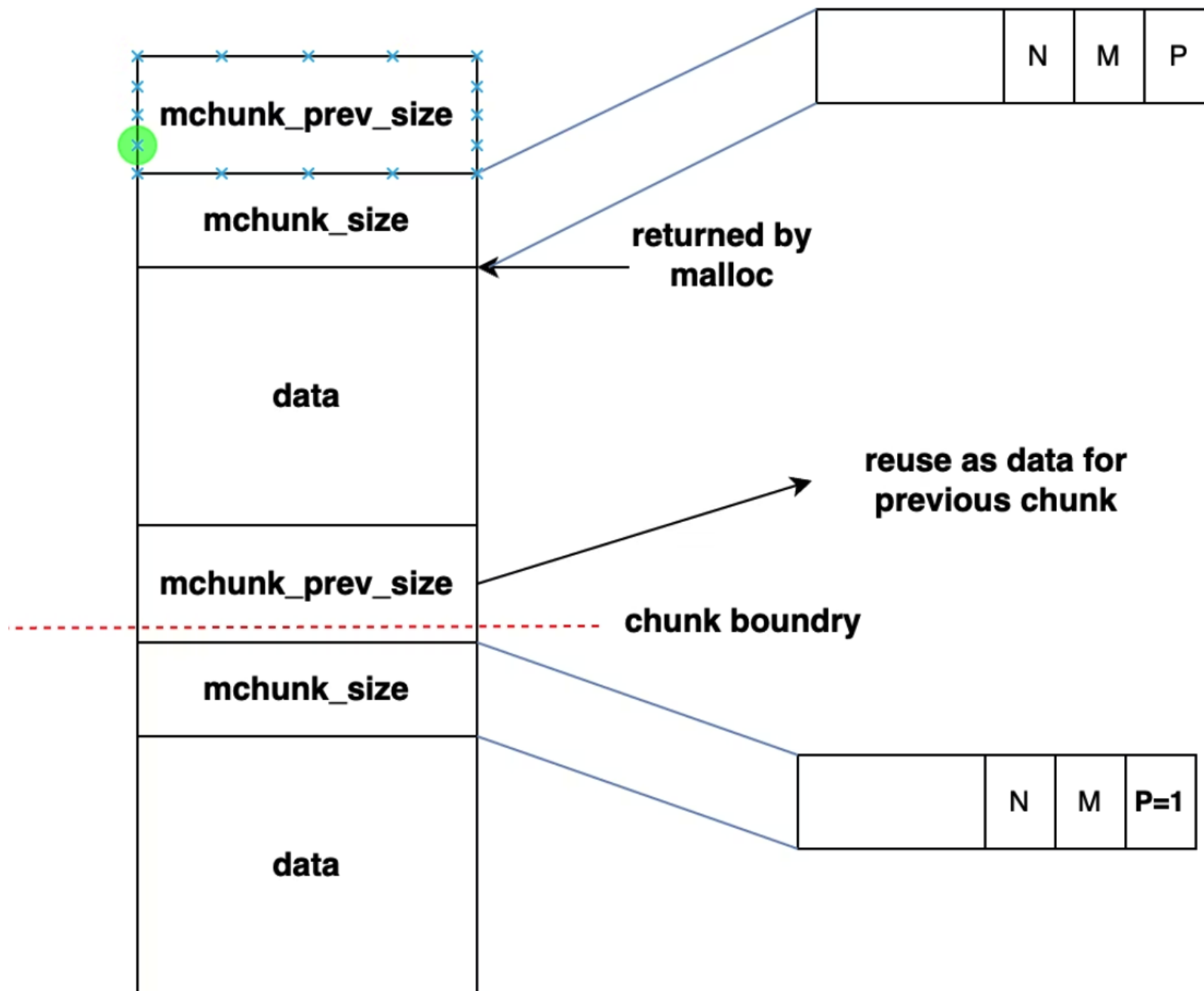


malloc() returns this address (first payload byte)

size is a multiple of 16, minimum 0x20 · the low 3 bits are flags (bit 0 = PREV_INUSE)



Chunk layout





mchunk_size

- mchunk_size: the size of the current chunk, including metadata
- Reuse the last 3 bits as flags
 - P (PREV_INUSE):
 - 0 when the previous chunk (not the previous chunk in the linked list, **but the one directly before it in memory**) is free (and hence the size of the previous chunk is stored in the first field). The very first chunk allocated has this bit set.
 - If it is 1, then we cannot determine the size of the previous chunk.



mchunk_prev_size

- mchunk_prev_size
 - If the previous chunk is free, this indicates the size of the previous free chunk.
 - If the previous chunk is **not** free, it can be used by the previous chunk as data.



Task A1 · the chunk header

- Run it: `cd ~/ierg4851/week6/in-class-lab/A1-chunk-header && ./chunk`
- Predict first: **how many bytes sit in front of the pointer, and what do they say?**



Task A1b · the next chunk's prev_size

- Run it: `cd ~/ierg4851/week6/in-class-lab/A1b-prev-size && ./prevsize`
- How many bytes can you write at a before you touch the next chunk's header?



Free lists

- `free()` usually does not return memory to the kernel
- it puts the chunk on **a free list for that size**, and writes an fd pointer into it



Why bins exist

- The heap manager needs to keep track of freed chunks so that malloc can reuse them during allocation requests
 - A naive implementation: the heap manager could do this by simply storing all freed chunks together on some enormous linked list → Slow
 - To improve performance, the heap manager instead maintains **a series of lists called “bins”**, which are designed to maximize speed of allocations and frees.



The bins

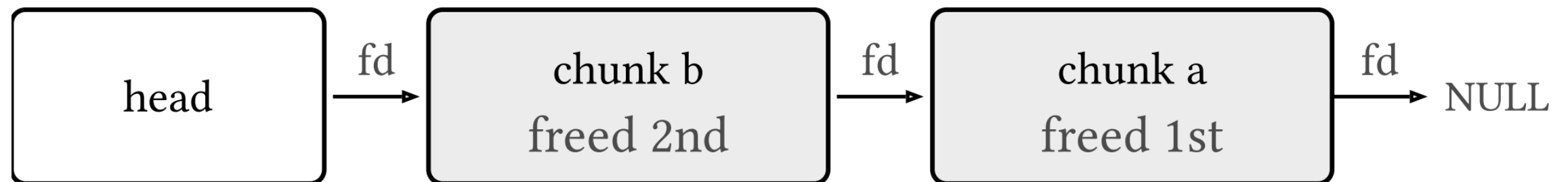
- There are 5 types of bins: **62 small bins**, **63 large bins**, **1 unsorted bin**, **10 fast bins** and **64 tcache bins** per thread (if tcache is enabled)
 - Small bins — 64-bit: 32 to 1008 bytes, step 16 (32-bit: 16 to 504 bytes, step 8)
- **Tcache bin: LIFO**
 - Single link list
 - Each has 7 chunks of specified size



tcache is LIFO

- Tcache: the last chunk freed is the first one malloc() hands back (LIFO)
 - Why LIFO: fast!

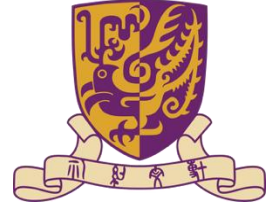
the free list for this chunk size (tcache / fastbin)



free(a) then free(b) → the list is b → a → NULL

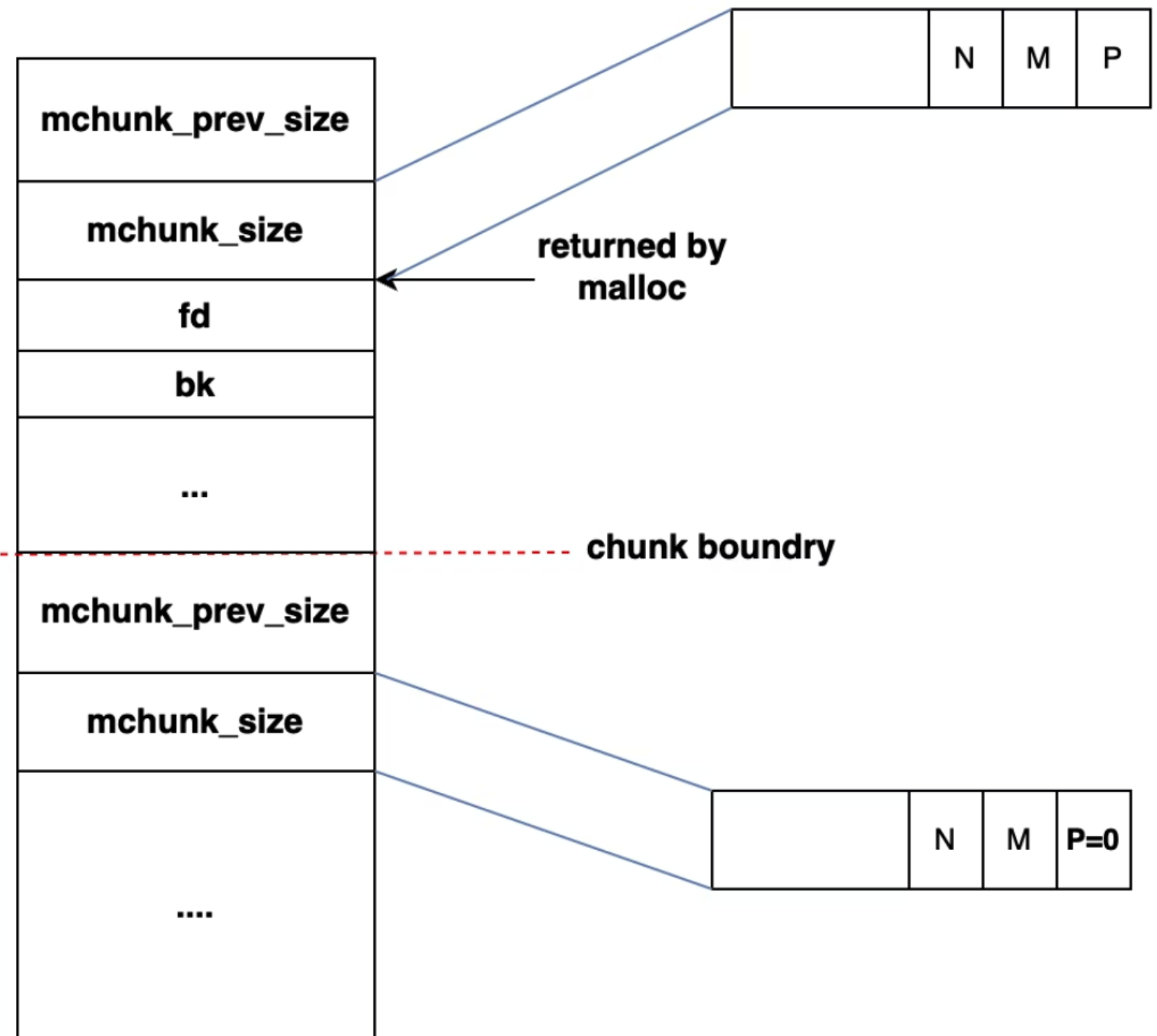
the LAST freed chunk is the FIRST one malloc() reuses (LIFO)

a free chunk's first 8 payload bytes become its fd pointer



fd and bk

- fd: Forward pointer to the next free chunk in list
- bk: Back pointer to the previous free chunk in list





Task A2 · which chunk comes back?

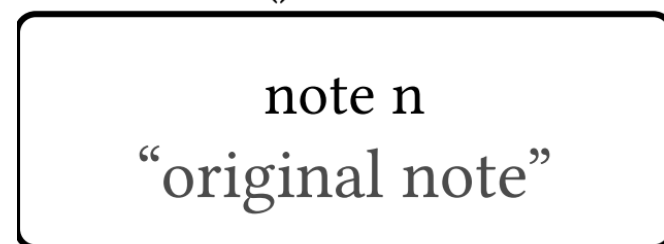
- Run it: `cd ~/ierg4851/week6/in-class-lab/A2-tcache-lifo && ./lifo`
- two chunks are freed — which one does the next `malloc()` return?



Use-after-free (UAF)

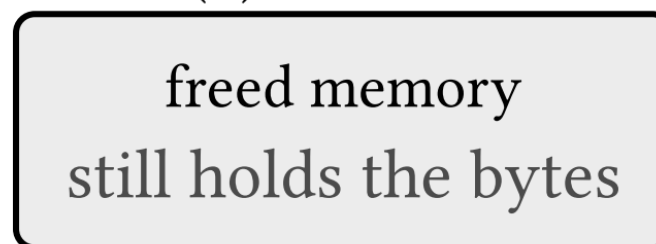
- a pointer that outlives free() is a use-after-free
- the memory is still there; only the allocator has moved on
- allocate the same size again and malloc() may hand you the very same chunk
- now two pointers name one chunk — the stale one and the new one

1 · malloc()



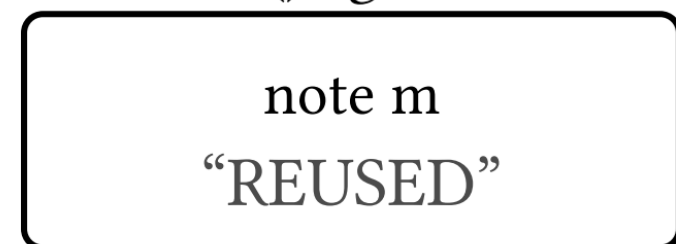
sizeof(struct note)

2 · free(n)



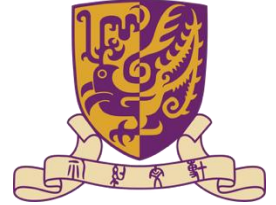
n is not cleared

3 · malloc() again



same size

the stale pointer n follows the memory — reading it reads m's data



Use-after-free — the code

```
struct note { char text[32]; };

static struct note *n;                /* the program keeps this pointer */

void open_note(const char *msg)
{
    n = malloc(sizeof(struct note));  /* 1. allocate */
    strcpy(n->text, msg);             /* fill it */
}

void close_note(void)
{
    free(n);                          /* 2. free ... */
    /* n is NOT set to NULL <- the bug: n is now a dangling pointer */
}

void reopen(const char *msg)           /* 3. the allocator hands the same */
{                                     /* chunk to the next same-size */
    struct note *m = malloc(sizeof(struct note)); /* malloc() */
    strcpy(m->text, msg);
    puts(n->text);                     /* reads m's data through n -- n and m are one chunk */
}
```

Fix: `n = NULL;` after `free(n)`, and never use a pointer you have freed.



Task A3 · the stale pointer read

- Run it: `cd ~/ierg4851/week6/in-class-lab/A3-uaf-read && ./uaf`
- Predict first: after `free(n)`, a new `malloc()` of the same size — what does `n` read now?

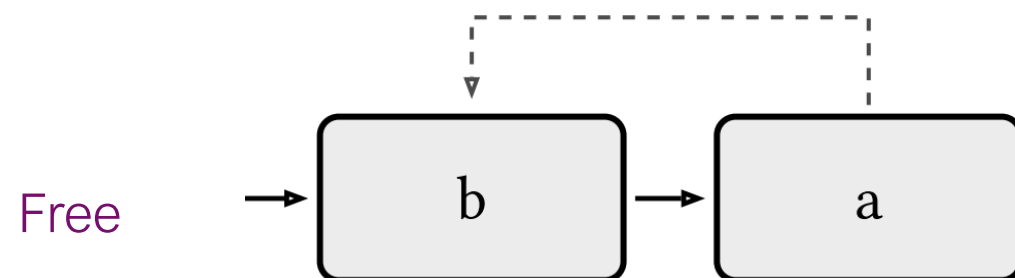
Block B — Turn reuse into control



Double free

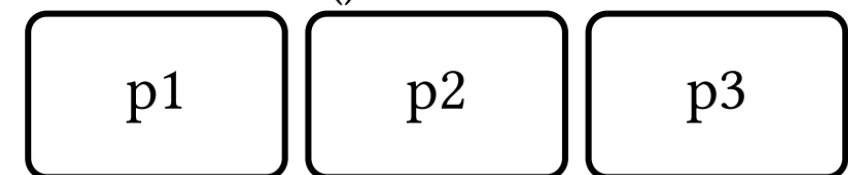
- Remember tcache bin: the free list is a chain of addresses (chunks)
- `free(p)` twice pushes `p` twice; each `malloc()` pops one entry off the chain
- so two `malloc()` calls hand back the same chunk while both pointers are still live
- nothing in the chunk says “I am free” — its header still reads in use
- 2.27 cannot tell; 2.29+ writes a key into the freed buffer and checks it

`free(a); free(b); free(a);`



the list becomes $a \rightarrow b \rightarrow a \rightarrow b \rightarrow \dots$ (a cycle, no NULL)

then `malloc()` three times



a

b

a

`p1` and `p3` are two live pointers into one chunk



Double free — the code

```
char *a = malloc(40);
char *b = malloc(40);

free(a);
free(b);
free(a);          /* the same chunk, twice */

char *p1 = malloc(40);    /* = a */
char *p2 = malloc(40);    /* = b */
char *p3 = malloc(40);    /* = a <- same chunk as p1 */

strcpy(p3, "written through p3");
puts(p1);              /* reads what p3 wrote: one chunk, two owners */
```

On glibc 2.29+ this aborts with “double free detected in tcache 2” — which is exactly why this lab runs on glibc 2.27.



Task B1 · the same chunk, twice

- Run it: `cd ~/ierg4851/week6/in-class-lab/B1-double-free && ./twice`
- Predict first: two 0x30 chunks, freed three times — how many entries does the list hold?



Writing through a freed pointer

- If the program still trusts the pointer, it trusts your bytes
- allocate the same size and the freed object comes back to you
- your input lands exactly where the program's fields live
- now the program's own check reads the value you chose
- the fix is one line: clear the pointer when you free it



Task B2 · a write after free

- Run it: `cd ~/ierg4851/week6/in-class-lab/B2-uaf-write && ./dangle`
- Predict first: after `logout()`, where will the message's `malloc(40)` land?



Task B3 · escalate an account to root

- Run it: `cd ~/ierg4851/week6/in-class-lab/B3-double-free-exploit && ./escalate`
- Predict first: root is one specific uid, not “any value” — how do you get that exact value in?



After class — reading

- Arenas, and the small / large / unsorted / fast / tcache bins
- Coalescing, and what the unlink macro checks
- Off-by-null → unlink: the classic “house” family of techniques

Wrap



- malloc gives you bookkeeping + payload; free puts the chunk on a list
- a stale pointer becomes a leak (read) or control (write)
- modern glibc makes all of this harder — know which libc you are on
- before you leave, one line: the thing that is still fuzzy