Heap from hero to zero 🚀

@Serotav

@Marco

Today's road map 🤠

-What is the heap? How does it work?
-Anatomy of a chunk
-Once upon malloc / free
-Tcache & tcache poisoning
-Fast bin & double free
-Safe linking
-Malloc & free hook
-Unsorted bin

-Lab

What is the heap?

Memory comes in different types...

ELF .bss: uninitialized global writable data
ELF .data: pre-initialized global writable data
ELF .rodata: global read-only data
stack: local variables, temporary storage, call stack metadata

What if you needed **dynamic memory** *allocation*?

What is the heap?

Introducing **Dynamic Allocators!**

General Purpose:
Doug Lea releases dlmalloc into public domain in 1987.

Glibc:
😏 ptmalloc (**P**osix **T**hread aware fork of dlmalloc)

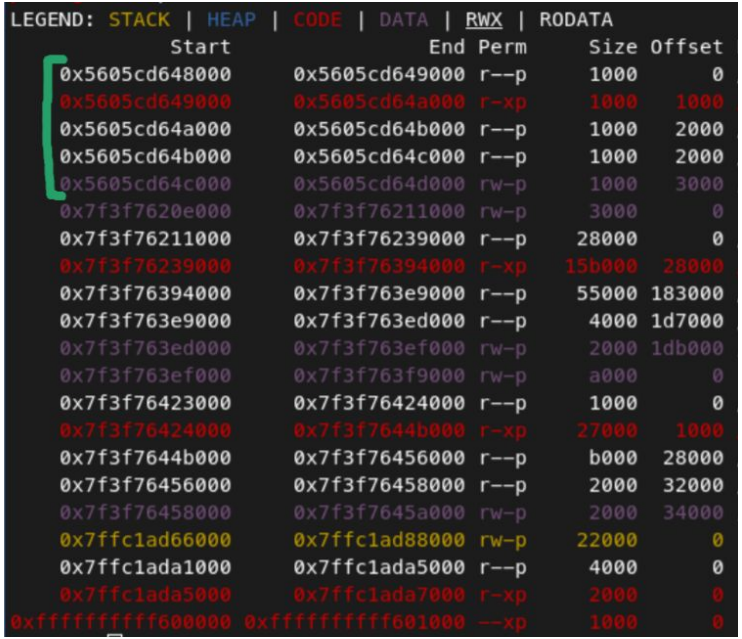
FreeBSD:
jemalloc (also used in Firefox, Android)

Windows:
Segment Heap, NT Heap

Kernel allocators:
SLUB allocator (Linux kernel memory allocator)
kalloc (iOS kernel memory allocator)

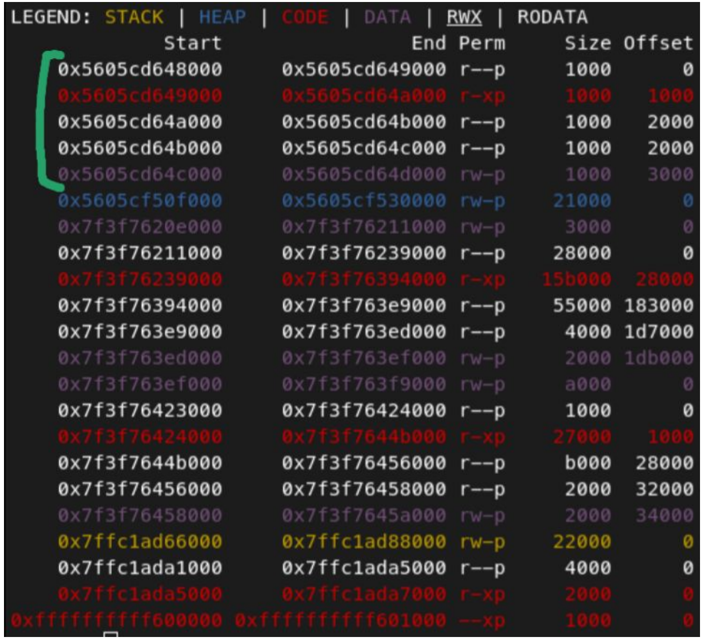
What is the heap?

- The heap is one or more memory pages used to store data (rw-)
- Also referred to as the Data Segment, initially has a size of 0
- With ASLR is placed somewhere near the binary



LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA

Start	End	Perm	Size	Offset
0x5605cd648000	0x5605cd649000	r--p	1000	0
0x5605cd649000	0x5605cd64a000	r-xp	1000	1000
0x5605cd64a000	0x5605cd64b000	r--p	1000	2000
0x5605cd64b000	0x5605cd64c000	r--p	1000	2000
0x5605cd64c000	0x5605cd64d000	rw-p	1000	3000
0x7f3f7620e000	0x7f3f76211000	rw-p	3000	0
0x7f3f76211000	0x7f3f76239000	r--p	28000	0
0x7f3f76239000	0x7f3f76394000	r-xp	15b000	28000
0x7f3f76394000	0x7f3f763e9000	r--p	55000	183000
0x7f3f763e9000	0x7f3f763ed000	r--p	4000	1d7000
0x7f3f763ed000	0x7f3f763ef000	rw-p	2000	1db000
0x7f3f763ef000	0x7f3f763f9000	rw-p	a000	0
0x7f3f76423000	0x7f3f76424000	r--p	1000	0
0x7f3f76424000	0x7f3f7644b000	r-xp	27000	1000
0x7f3f7644b000	0x7f3f76456000	r--p	b000	28000
0x7f3f76456000	0x7f3f76458000	r--p	2000	32000
0x7f3f76458000	0x7f3f7645a000	rw-p	2000	34000
0x7ffc1ad66000	0x7ffc1ad88000	rw-p	22000	0
0x7ffc1ada1000	0x7ffc1ada5000	r--p	4000	0
0x7ffc1ada5000	0x7ffc1ada7000	r-xp	2000	0
0xffffffffff600000	0xffffffffff601000	--xp	1000	0



LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA

Start	End	Perm	Size	Offset
0x5605cd648000	0x5605cd649000	r--p	1000	0
0x5605cd649000	0x5605cd64a000	r-xp	1000	1000
0x5605cd64a000	0x5605cd64b000	r--p	1000	2000
0x5605cd64b000	0x5605cd64c000	r--p	1000	2000
0x5605cd64c000	0x5605cd64d000	rw-p	1000	3000
0x5605cf50f000	0x5605cf530000	rw-p	21000	0
0x7f3f7620e000	0x7f3f76211000	rw-p	3000	0
0x7f3f76211000	0x7f3f76239000	r--p	28000	0
0x7f3f76239000	0x7f3f76394000	r-xp	15b000	28000
0x7f3f76394000	0x7f3f763e9000	r--p	55000	183000
0x7f3f763e9000	0x7f3f763ed000	r--p	4000	1d7000
0x7f3f763ed000	0x7f3f763ef000	rw-p	2000	1db000
0x7f3f763ef000	0x7f3f763f9000	rw-p	a000	0
0x7f3f76423000	0x7f3f76424000	r--p	1000	0
0x7f3f76424000	0x7f3f7644b000	r-xp	27000	1000
0x7f3f7644b000	0x7f3f76456000	r--p	b000	28000
0x7f3f76456000	0x7f3f76458000	r--p	2000	32000
0x7f3f76458000	0x7f3f7645a000	rw-p	2000	34000
0x7ffc1ad66000	0x7ffc1ad88000	rw-p	22000	0
0x7ffc1ada1000	0x7ffc1ada5000	r--p	4000	0
0x7ffc1ada5000	0x7ffc1ada7000	r-xp	2000	0
0xffffffffff600000	0xffffffffff601000	--xp	1000	0

How does it work?

Managed by the `brk` and `sbrk`:
- `sbrk(NULL)` returns the end of the data segment
- `sbrk(delta)` expands the end of the data segment by **delta** bytes
- `brk(addr)` expands the end of the data segment to **addr**
- `mmap/munmap` only for huge allocations

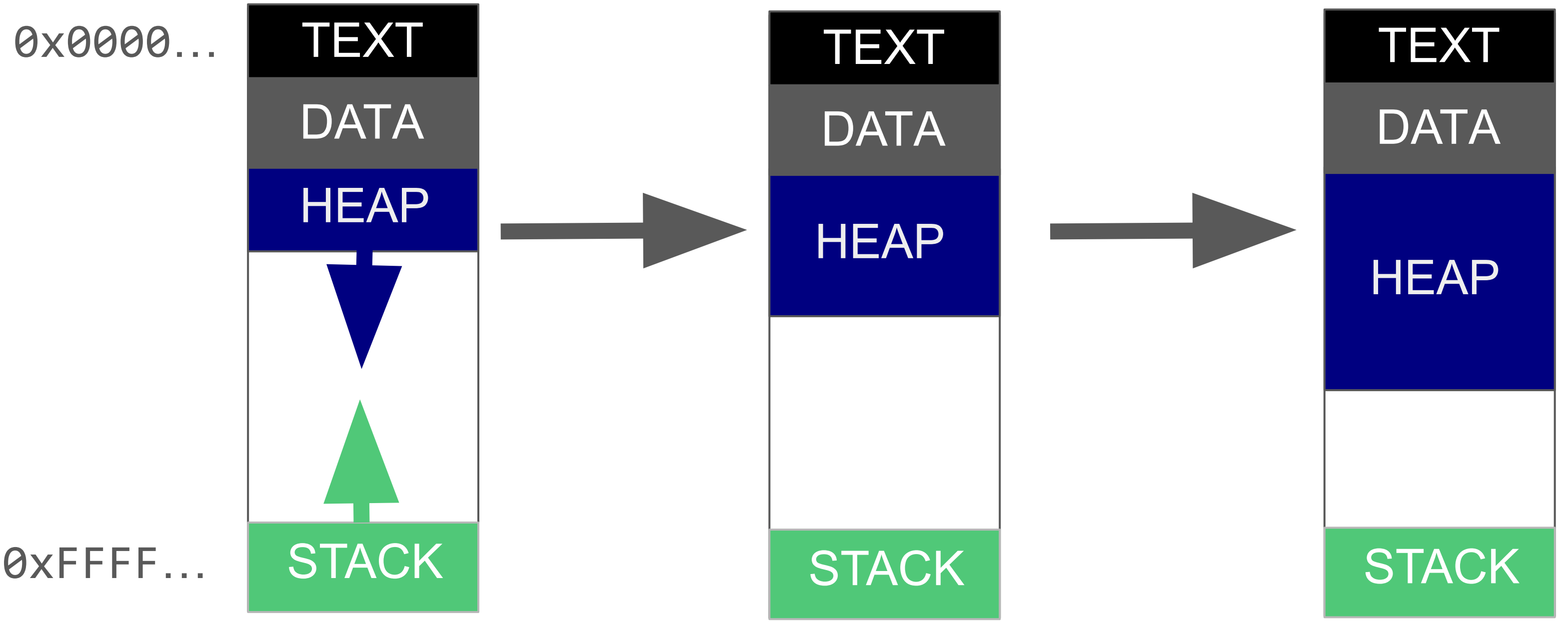
```c
void setup(void) {
    setbuf(stdin, NULL);
    setbuf(stdout, NULL);
    setbuf(stderr, NULL);
}

int main() {
    setup();
    printf("Before heap creation\n");
    malloc(0x10);
    printf("After heap creation\n");

}
```

```
write(1, "Before heap creation", 20Before heap creation)    = 20
write(1, "\n", 1
)                             = 1
getrandom("\xb4\x8b\x83\xcc\x03\xd3\xa7\x7b", 8, GRND_NONBLOCK) = 8
brk(NULL)                                   = 0x5643340ae000
brk(0x5643340cf000)                         = 0x5643340cf000
write(1, "After heap creation", 19After heap creation)     = 19
write(1, "\n", 1
)                             = 1
```

0x21000

How does it work?

If the current heap can't handle the memory request, the program uses **`brk`** to map more pages into its memory, effectively making the heap bigger



How does it work?

The heap, as implemented by ptmalloc/glibc provides:

- `malloc()` - allocate some memory
- `free()` - free a prior allocated chunk

And some auxiliary functions:

- `realloc()` - change the size of an allocation
- `calloc()` - allocate and zero-out memory

How does it work?

Principles of Heap Design:

1) **Speed**: The heap must allocate and deallocate memory quickly to ensure high performance.

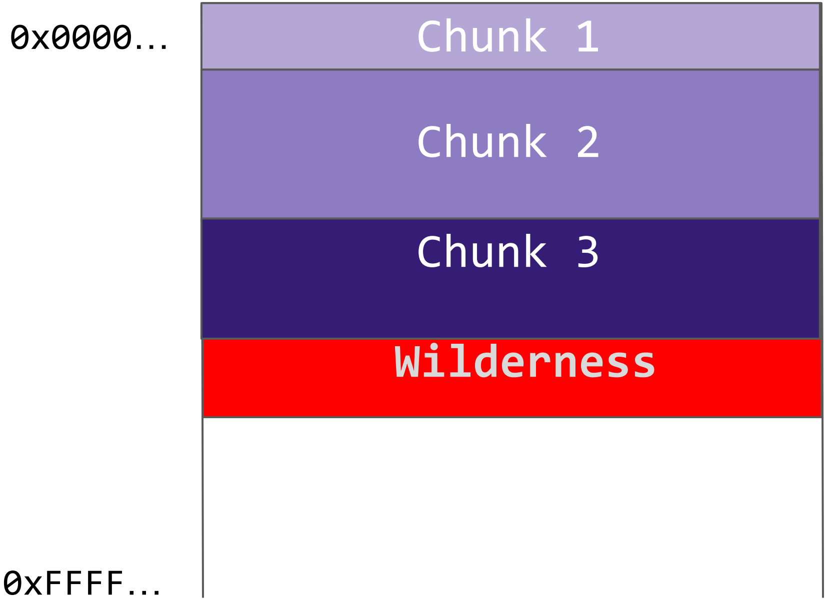
2) **Memory Fragmentation**: The heap must minimize fragmentation to use memory efficiently and avoid waste.

How does it work?

Once the heap is created, all available memory is contained in a special chunk called the **wilderness** or **top chunk**.

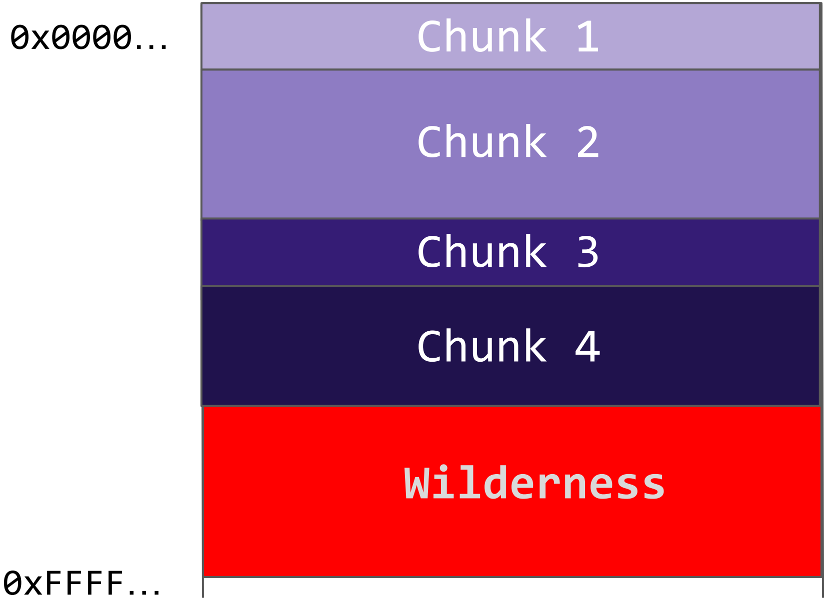
How does it work?

Each *new* chunk will be carved out of this, making the top chunk shrink.



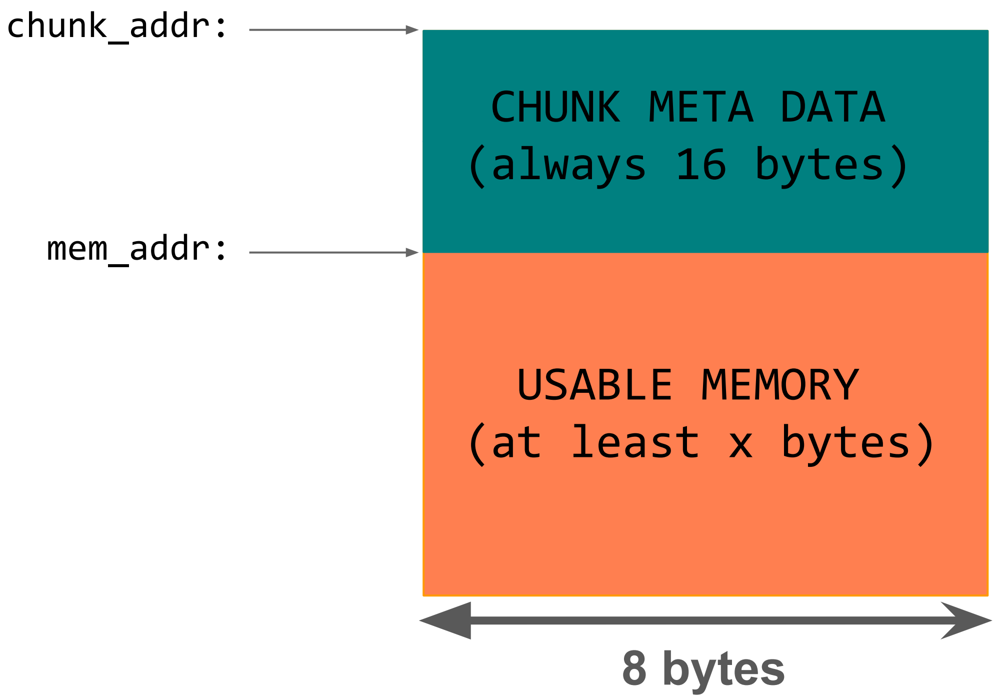
How does it work?

Once there's no more space on the heap, **`brk`** is called to allocate more pages to the heap and the top chunk is expanded



Anatomy of a chunk

`malloc`(x) returns **`mem_addr`**, but in actuality, ptmalloc tracks **`chunk_addr`**:

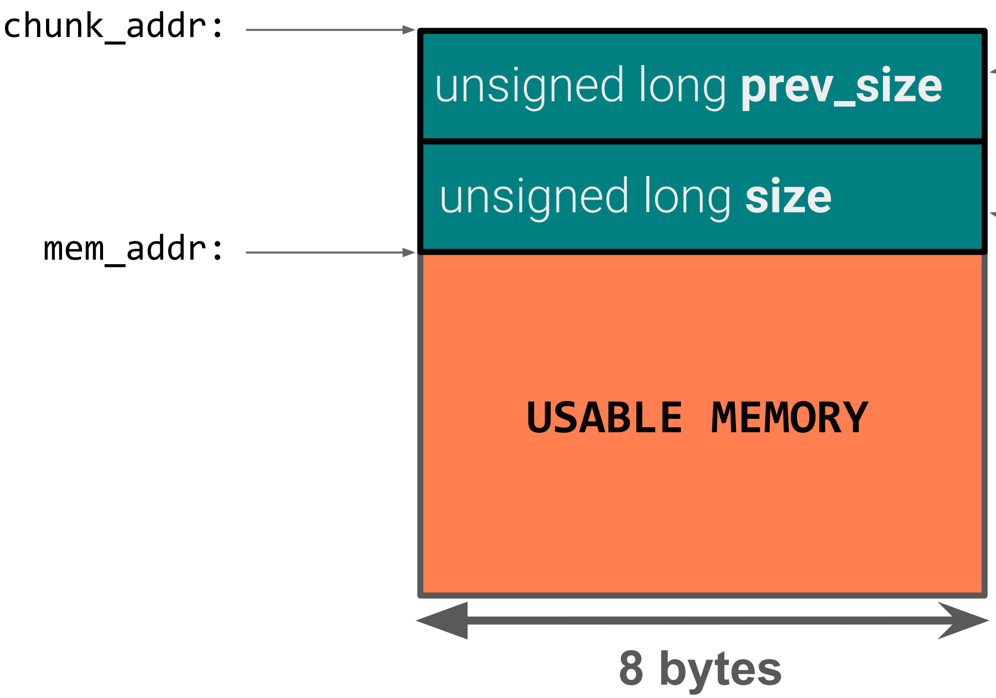


The metadata section is 16 bytes. If the previous chunk is in use, the first 8 bytes of this can be utilized by it.

`malloc(n)` guarantees *at least* ***n*** usable space, but chunks sizes are multiples of 0x10.

Anatomy of a chunk

`malloc`(x) returns **`mem_addr`**, but in actuality, ptmalloc tracks **`chunk_addr`**:



If the previous chunk is in use,this can be utilized by it.

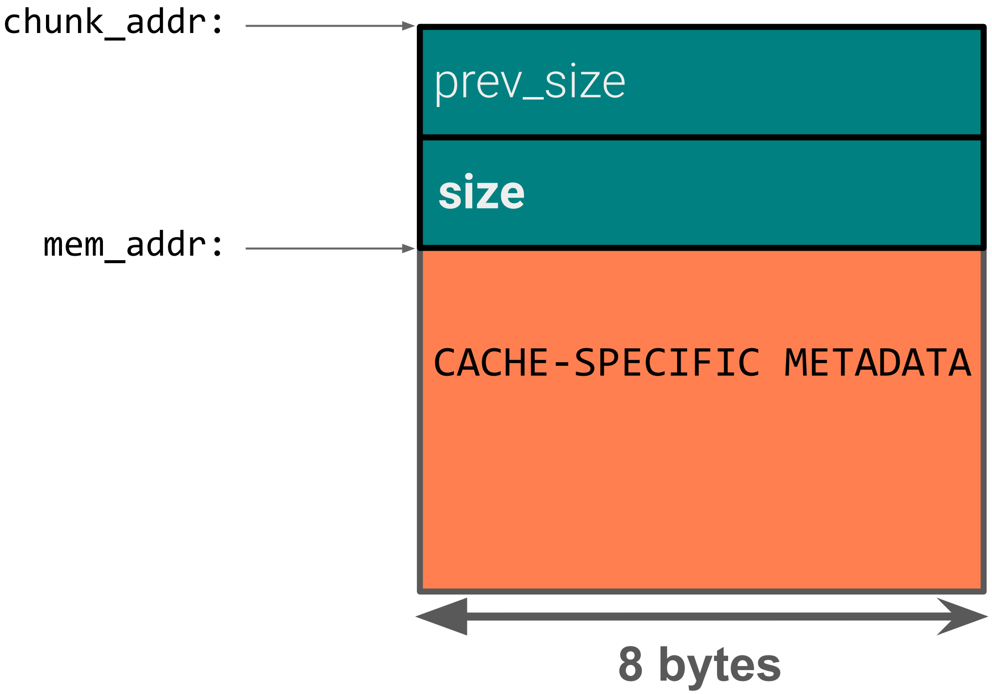
Lowest 3 bits are flags:

`Bit 0: PREV_IN_USE`
`Bit 1: IS_MMAPPED*`
`Bit 2: NON_MAIN_ARENA`

`* Allocations of 128kb or more are done using mmap.`
`Can be changed with:`
mallopt(M_MMAP_THRESHOLD, val)

Anatomy of a chunk

A `free()`d chunk has additional metadata about the location of other chunks

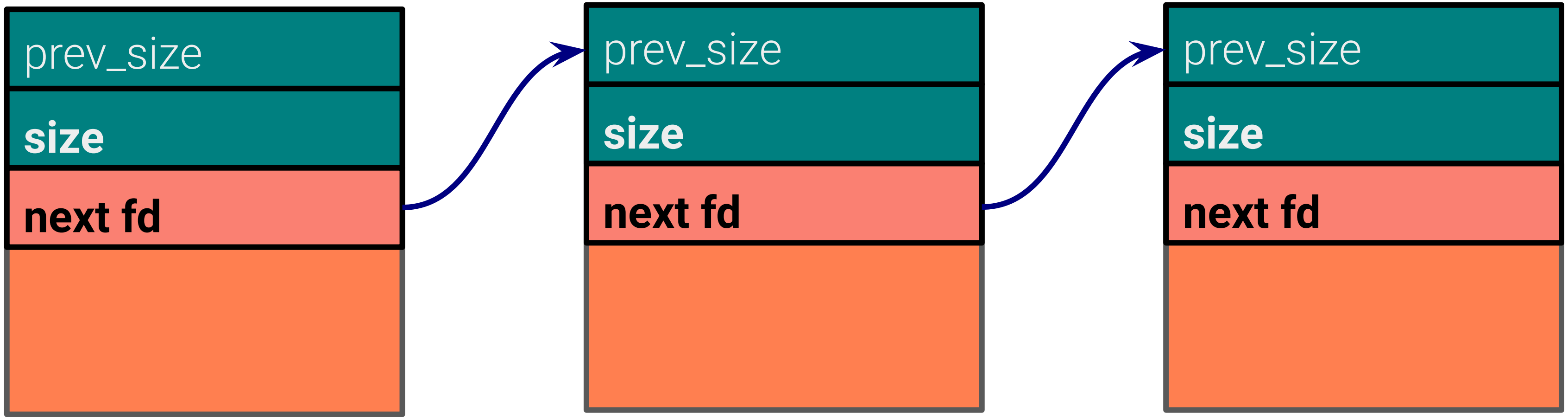


The allocator uses multiple cache layers, each with specific metadata.

These caches are organized as different kinds of linked lists.

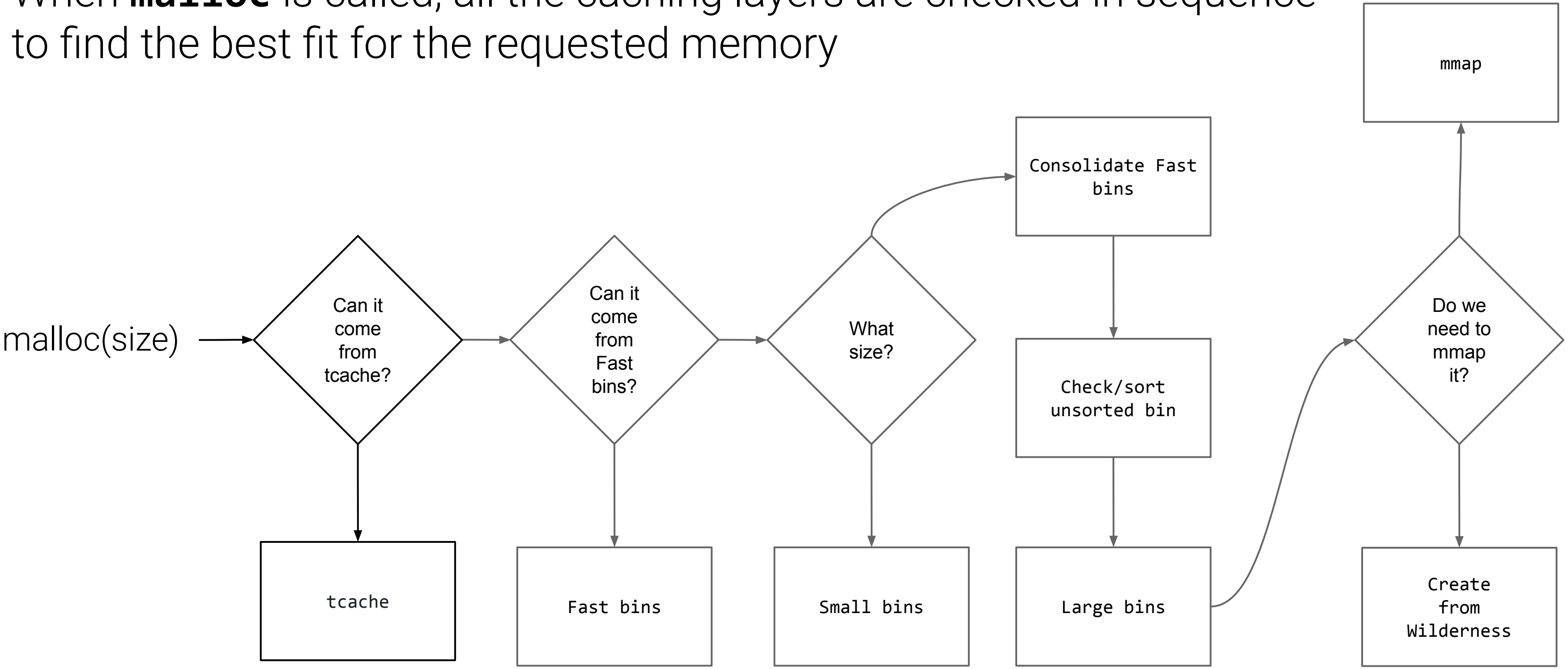
Anatomy of a chunk

In all linked list each chunk points to the next. For now, we'll focus only on this.



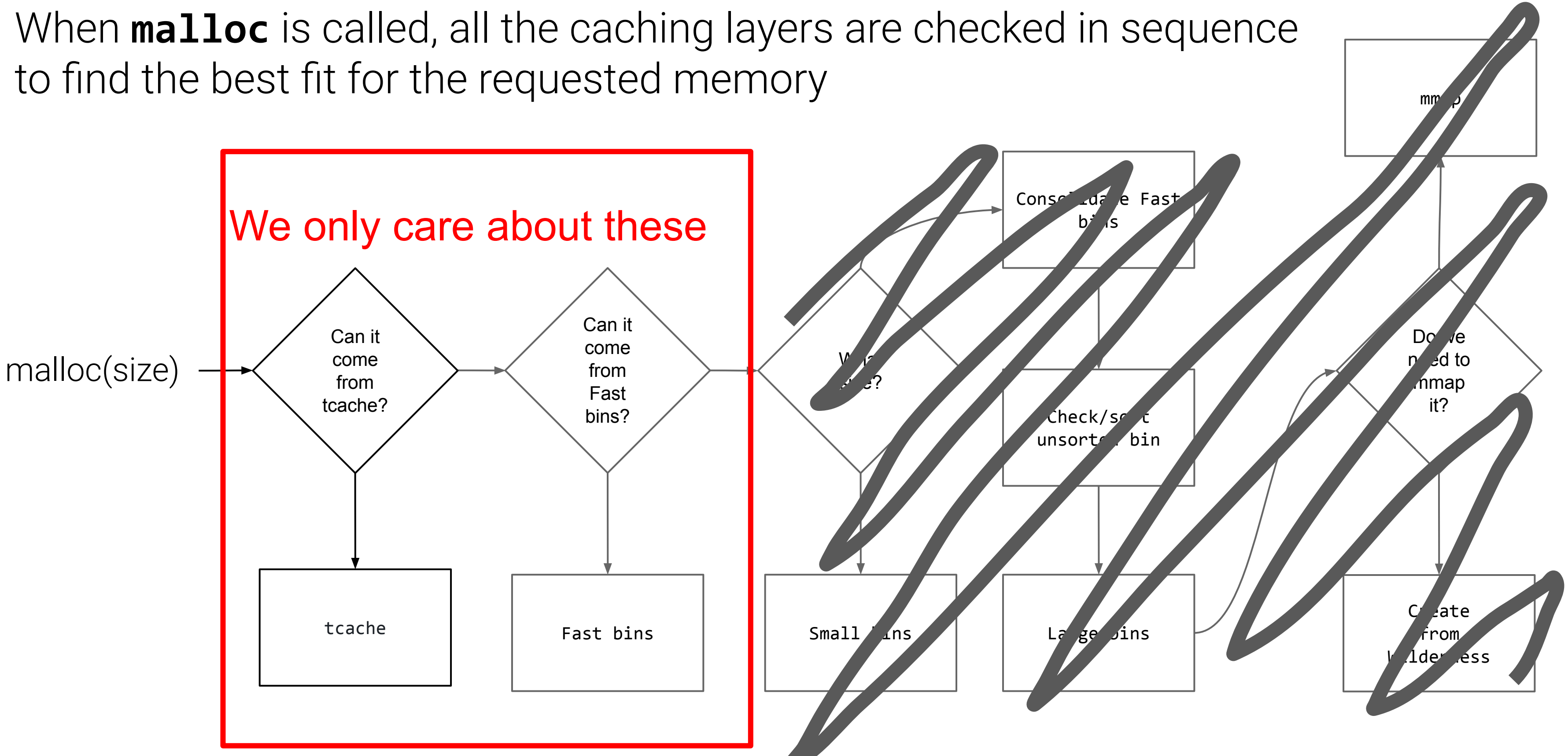
Once upon malloc

When **`malloc`** is called, all the caching layers are checked in sequence to find the best fit for the requested memory



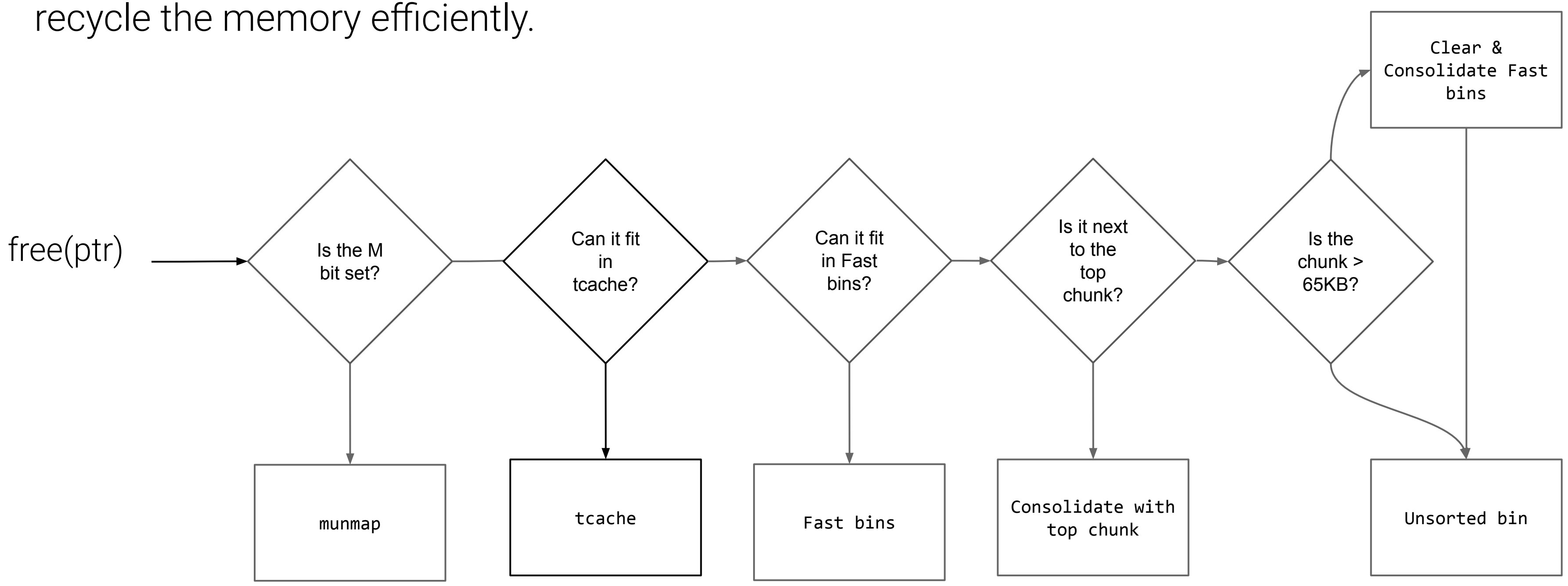
Once upon malloc

When **`malloc`** is called, all the caching layers are checked in sequence to find the best fit for the requested memory



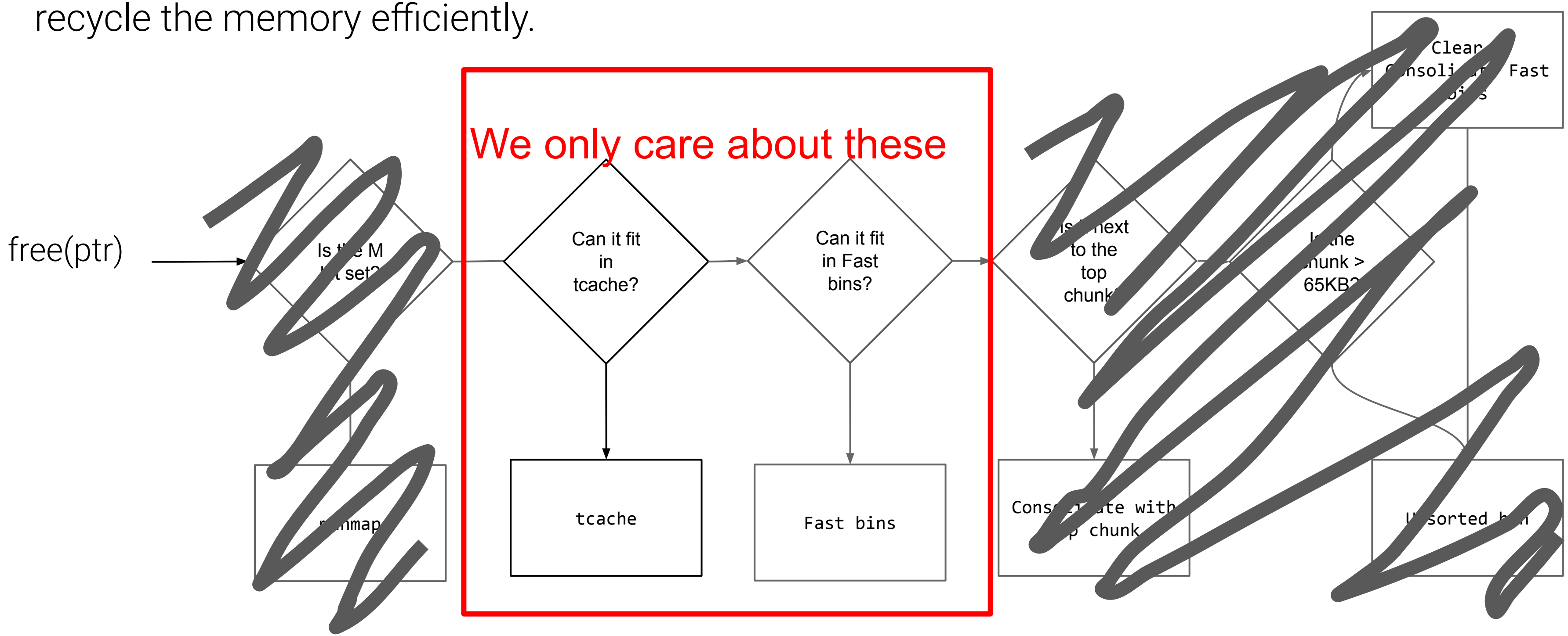
Once upon free

When **`free`** is called the allocators follows a series of checks and steps to manage and recycle the memory efficiently.



Once upon free

When **`free`** is called the allocators follows a series of checks and steps to manage and recycle the memory efficiently.



Tcache

The **Thread Local Cache** is used for allocation up to 1032 bytes.
Implemented as a **singly-linked** list, with each thread having a list header for different-sized allocations, each list can hold at max 7 elements. **This structure is allocated on the heap**.

```
typedef struct tcache_perthread_struct
{
  char counts[TCACHE_MAX_BINS];
  tcache_entry *entries[TCACHE_MAX_BINS]; // just non-mangled void*’s to chunks!
} tcache_perthread_struct;
```

tcache_perthread_struct

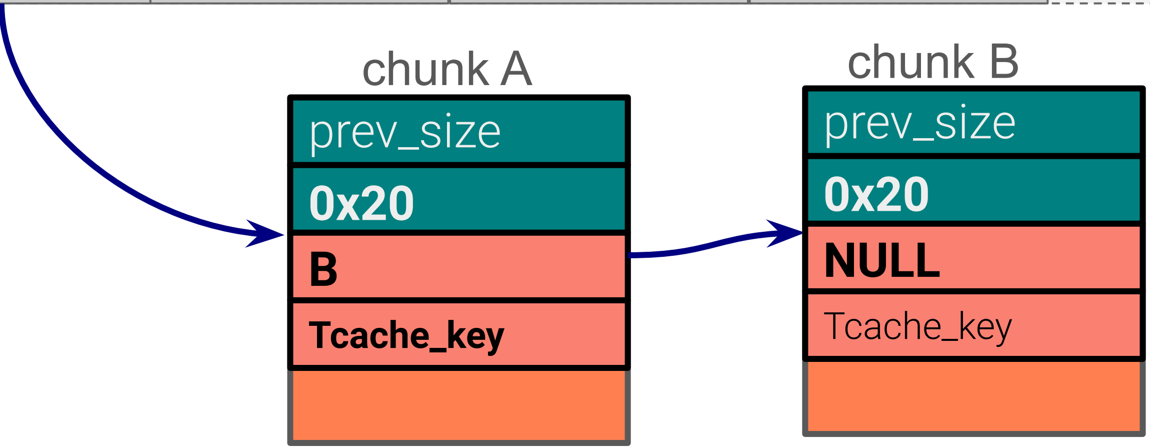
	list 0x20	list 0x30	list 0x40	list 0x50
counts:	3	3	1	0
entries:	A	C	D	NULL

Tcache

tcache_perthread_struct

	list 0x20	list 0x30	list 0x40	list 0x50
counts:	2	3	1	0
entries:	A	C	D	NULL

Pointers in the tcache point to the user usable memory, not the beginning of the chunk. Chunks are added and removed from the head of the list

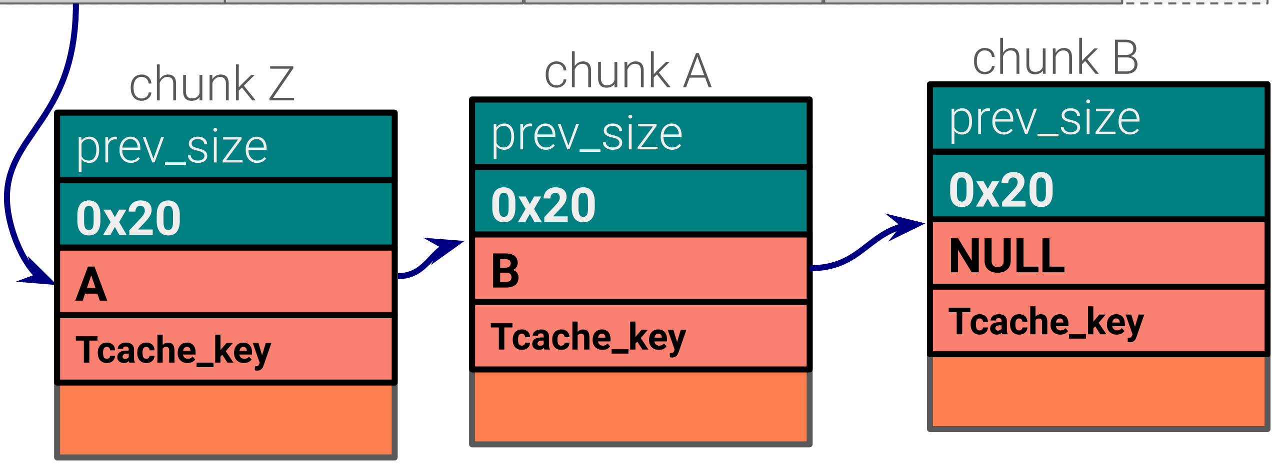


Tcache

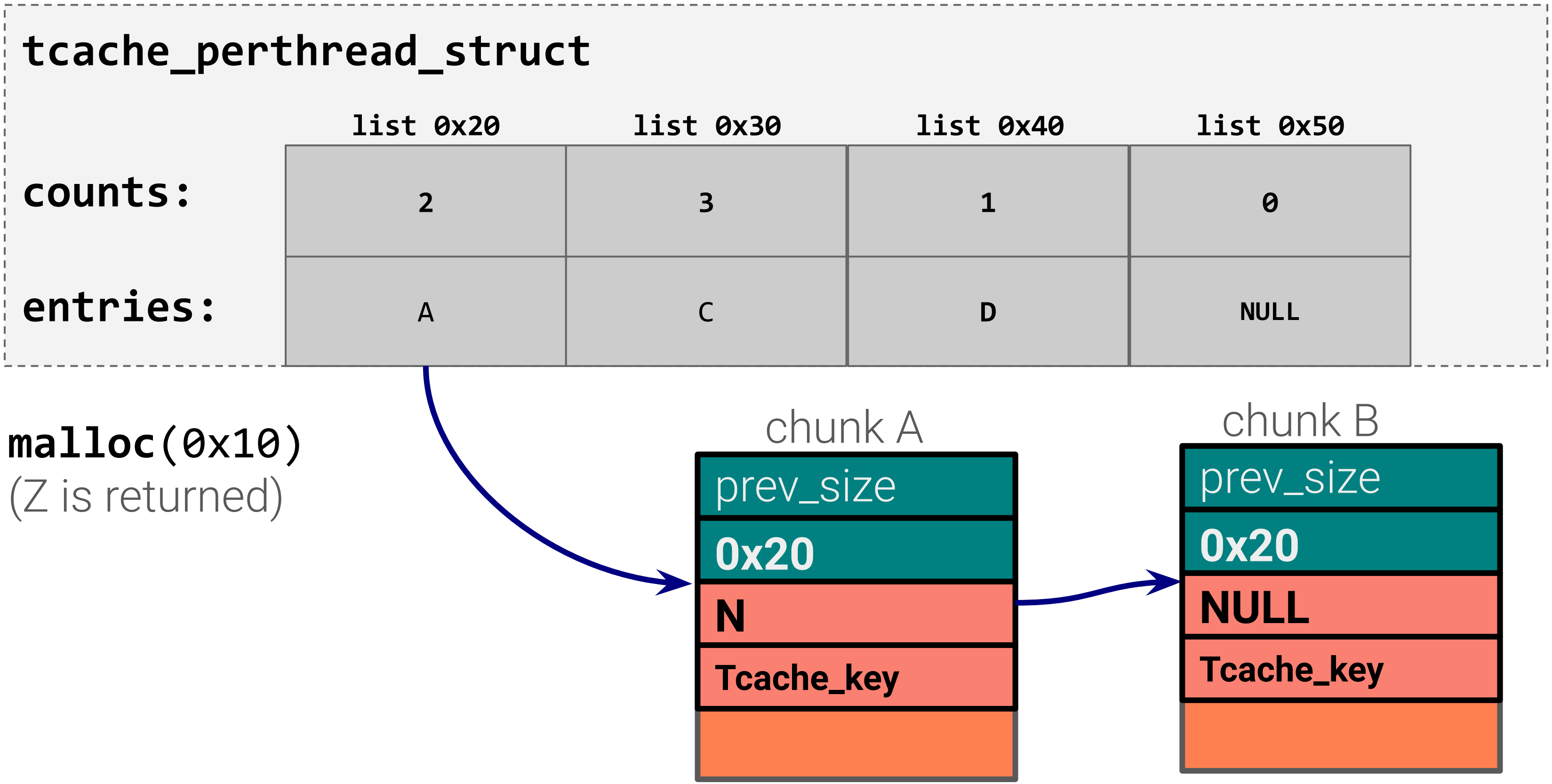
tcache_perthread_struct

	list 0x20	list 0x30	list 0x40	list 0x50
counts:	2	3	1	0
entries:	Z	C	D	NULL

free(Z)



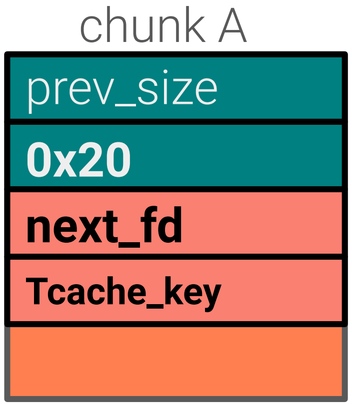
Tcache


tcache_perthread_struct
list 0x20
list 0x30
list 0x40
list 0x50
counts:
2
3
1
0
entries:
A
C
D
NULL
malloc(0x10)
(Z is returned)
chunk A
prev_size
0x20
N
Tcache_key
chunk B
prev_size
0x20
NULL
Tcache_key

Tcache

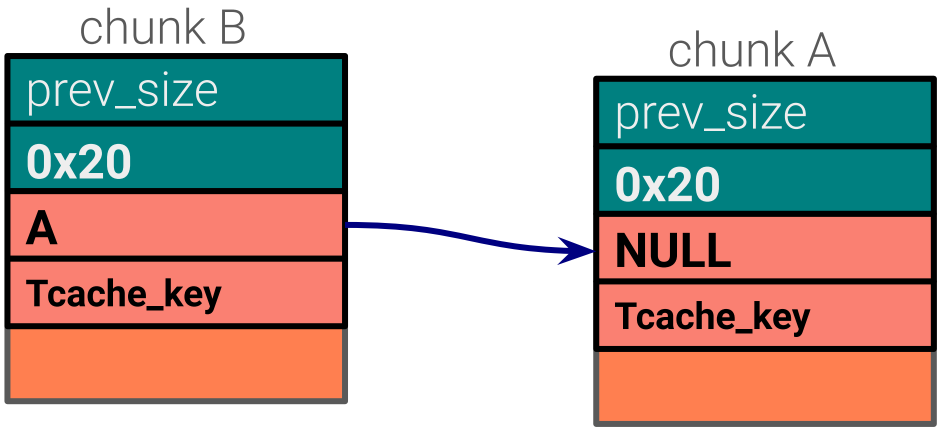
Before 2.29:
When a chunk is freed it's compared to the ones in the tcache list it'll be put in, preventing double free completely.

Now:
Each chunk in the list has a **Tcache_key**, unique for each Tcache (pointer to the **`tcache_perthread_struct`** until **2.34**, then random).
When a chunk is freed, if it contains the key (a qword at offset 8), the normal double free check is performed. But if it doesn't, the key is added and the chunk is put into the tcache.
When a chunk is allocated the key is removed.
This check can be bypassed by overwriting the key before freeing the chunk again. This is strictly a security **downgrade**.



Tcache attacks

Let's say that we have a use after free vuln, what can we do?
If we free 2 chunks and read the data from the last freed one, we can leak the heap.

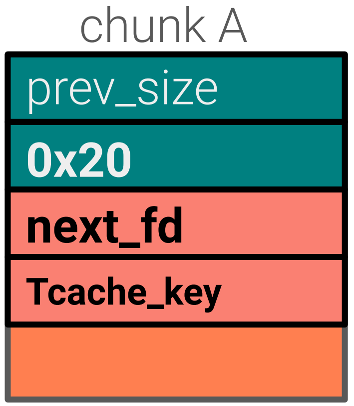


*ignoring safe linking for now

Tcache poisoning

Before returning a chunk, malloc always checks that the chunk is **correctly aligned** (it must be a multiple of 0x10).

Apart from this check and the one described in the previous slide, chunks from the tcache don't undergo any other validation.
This means if you manage to overwrite a fd pointer (with use after free), you can trick malloc into returning any pointer you want.

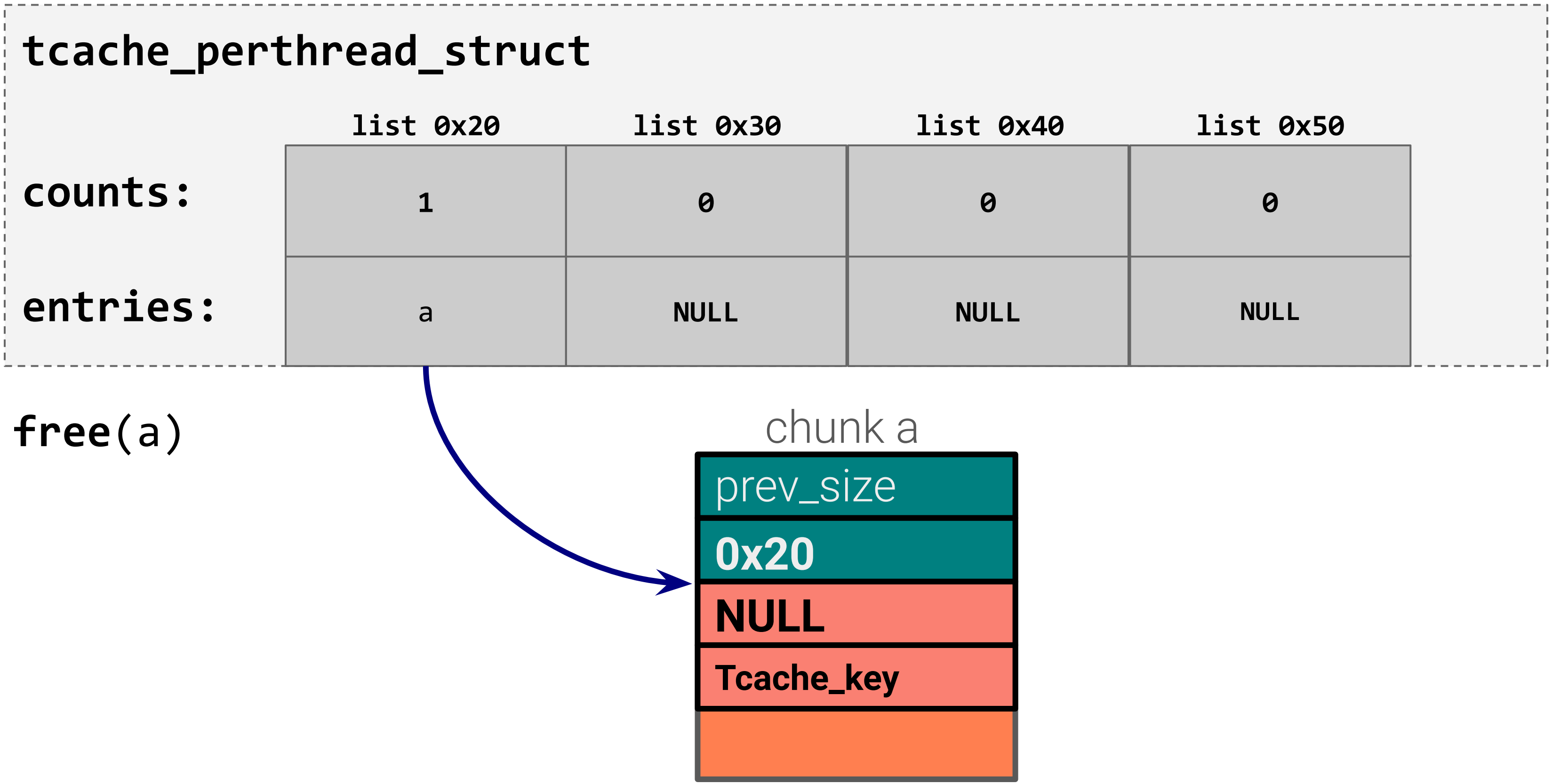


Tcache poisoning

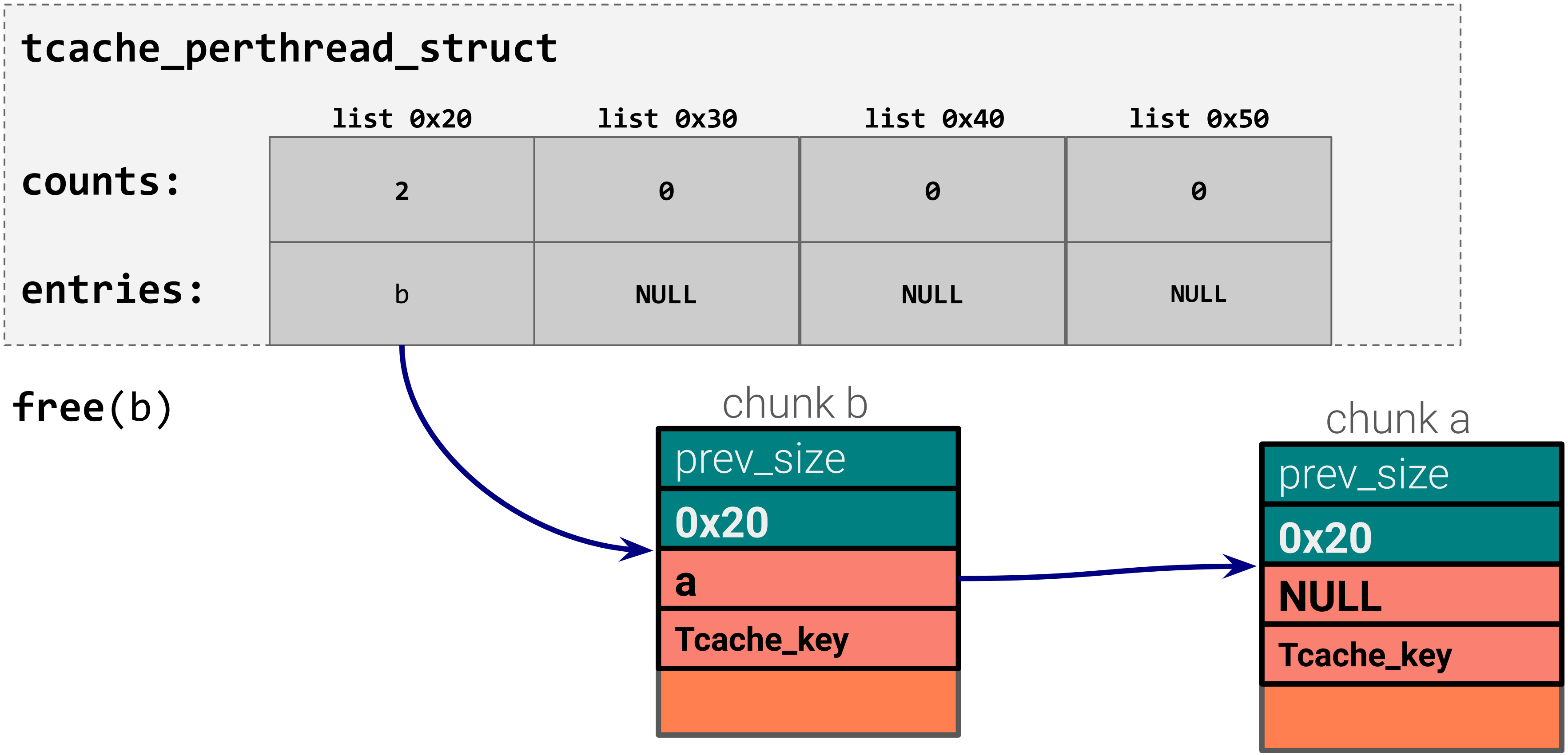
```c
int main() {
    long t = 0x42069;

    long *a = malloc(0x10);
    long *b = malloc(0x10);
    printf("a:  %p \nb:  %p\n&t: %p\nt:  %p\n\n\n", a, b, &t, t);

    free(a);
    free(b);

    *b = &t;

    long *c = malloc(0x10);
    long *d = malloc(0x10);
    printf("c:  %p \nd:  %p\n&t: %p\nt:  %p\n*d: %p\n", c, d, &t,t,*d);

}
```

```
a:   0x19472a0
b:   0x19472c0
&t:  0x7ffd38c13540
t:   0x42069



c:   0x19472c0
d:   0x7ffd38c13540
&t:  0x7ffd38c13540
t:   0x42069
*d:  0x42069
```

Tcache poisoning



Tcache poisoning

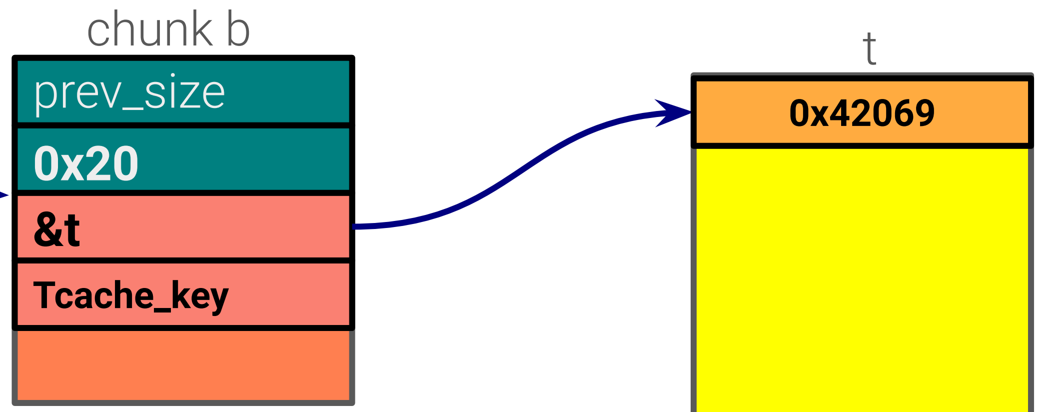


Tcache poisoning

tcache_perthread_struct

	list 0x20	list 0x30	list 0x40	list 0x50
counts:	2	0	0	0
entries:	b	NULL	NULL	NULL

*b = &t;

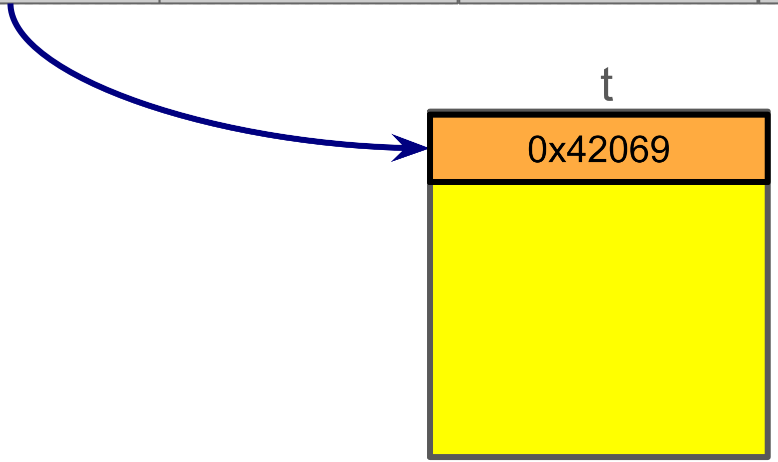


Tcache poisoning

tcache_perthread_struct

	list 0x20	list 0x30	list 0x40	list 0x50
counts:	1	0	0	0
entries:	&t	NULL	NULL	NULL

```
c = malloc(0x20)
```

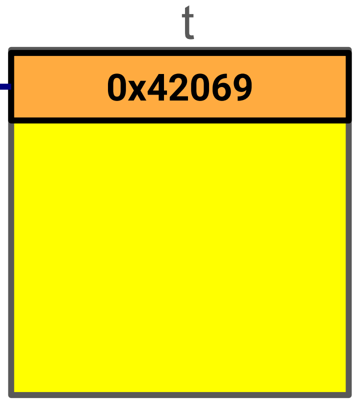


Tcache poisoning

tcache_perthread_struct

	list 0x20	list 0x30	list 0x40	list 0x50
counts:	0	0	0	0
entries:	NULL	NULL	NULL	NULL

d = `malloc`(0x20)



Tcache

Since the **tcache_perthread_struct** is allocated on the heap, if we can leak the heap address and poison a chunk, we can obtain a pointer to the **tcache** itself and write to it directly.
This is particularly useful if the version of libc in use employs safe linking, as pointers in the tcache head are not mangled.

Note: Don't forget to update the count accordingly.

tcache_perthread_struct

	list 0x20	list 0x30	list 0x40	list 0x50
counts:	3	0	0	0
entries:	B	NULL	NULL	NULL

Fast bins

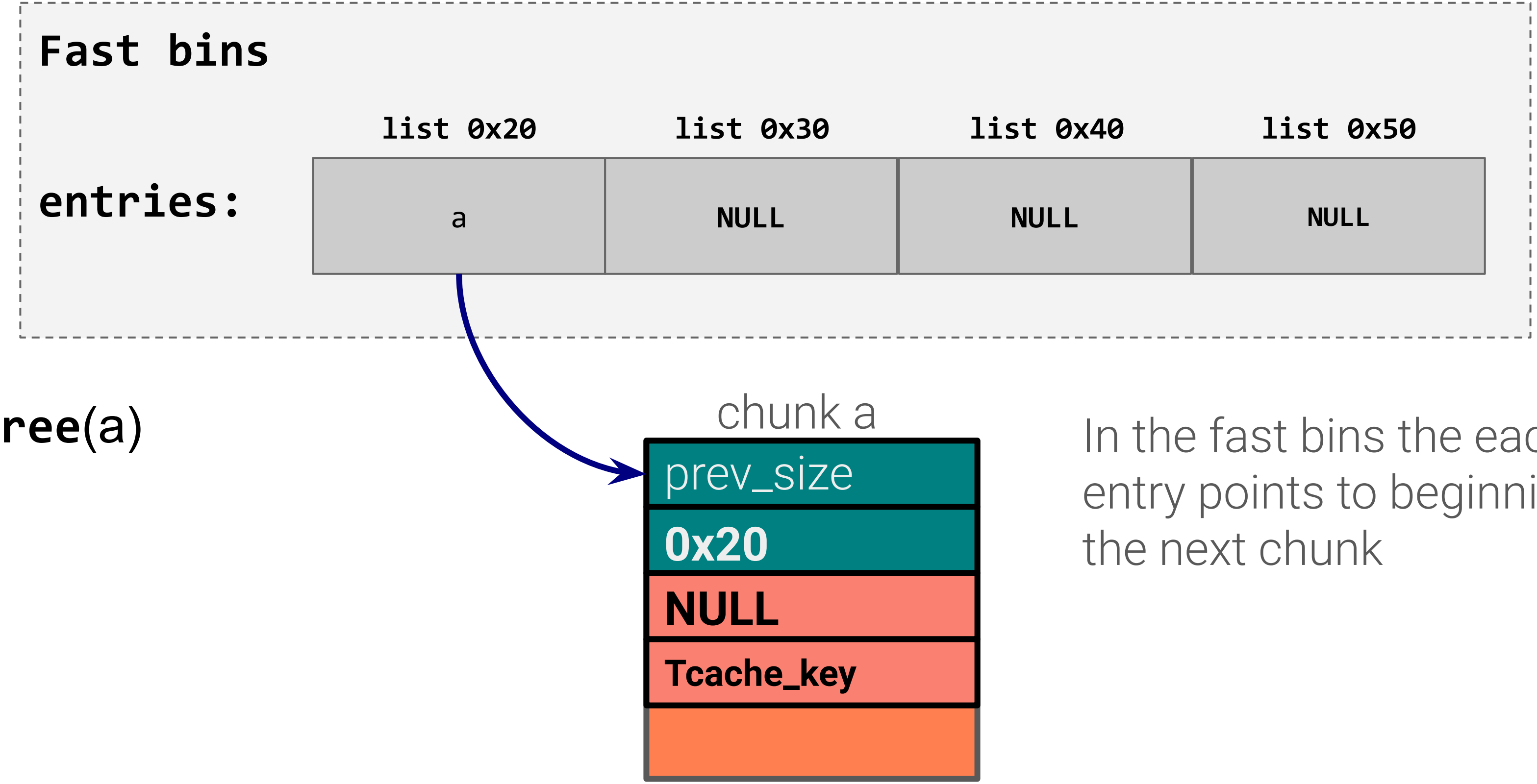
Singly linked lists similar to tcache, but each bin list grows to unlimited length, chunks are of sizes up to 0x80 bytes.
In order to avoid double free only the **chunk in the head** is checked.

For this reason a double free attacks is possible.

Fast bins

	list 0x20	list 0x30	list 0x40	list 0x50
entries:	a	NULL	NULL	NULL

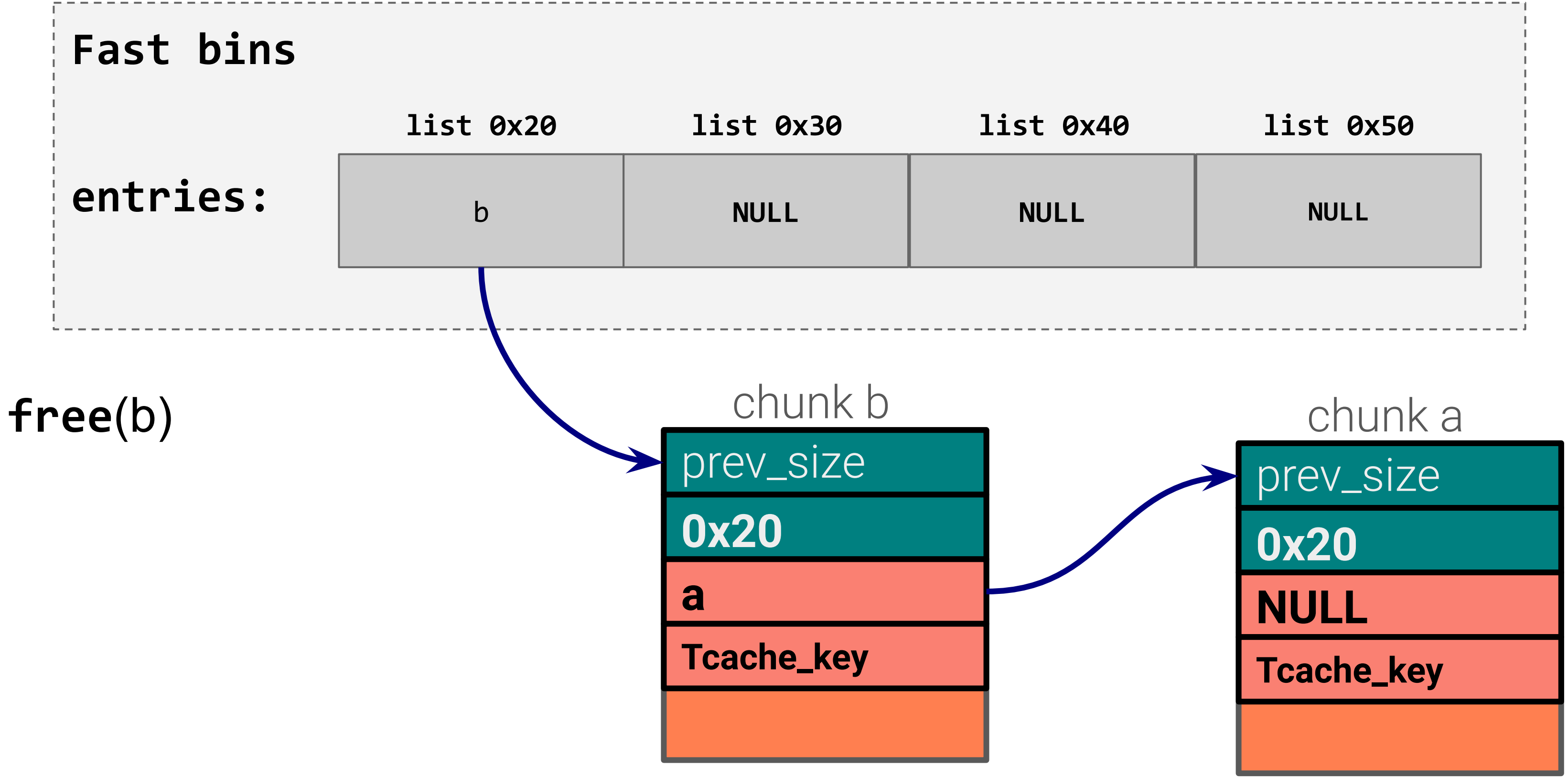
Fast bins double free



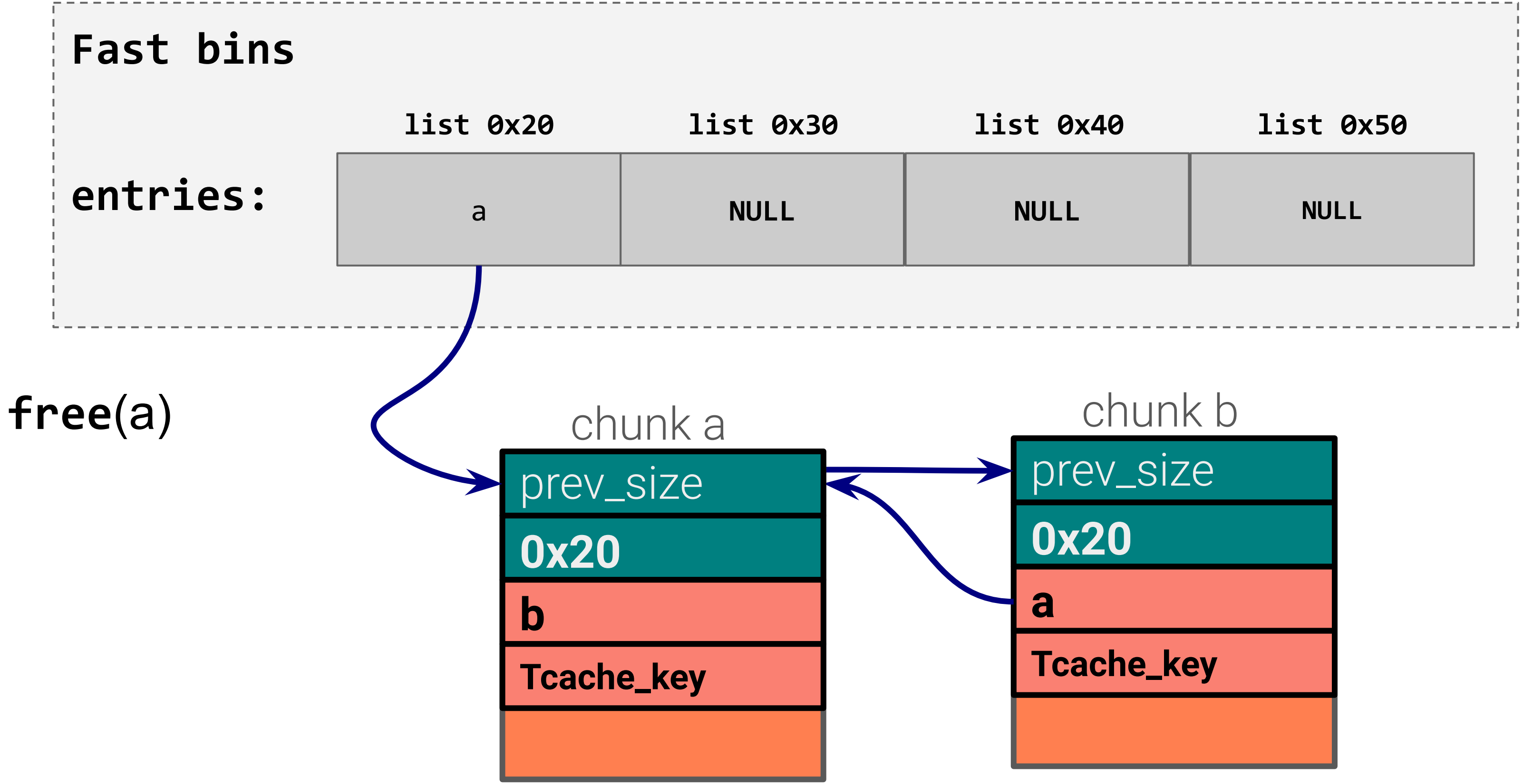
free(a)

In the fast bins the each entry points to beginning of the next chunk

Fast bins double free



Fast bins double free



Safe linking

Since libc 2.32 tcache and fast bin next pointers are mangled

```
#define PROTECT_PTR(pos, ptr)
  ((__typeof (ptr)) ((((size_t) pos) >> 12) ^ ((size_t) ptr)))

#define REVEAL_PTR(ptr)  PROTECT_PTR (&ptr, ptr)
```

```
uint64_t *a = malloc(16), b = malloc(16);
free(a);
free(b);
// tcache mangled ptr (b points to a)
assert(*b == ((uint64_t)a>>12)^(uint64_t)b);
```

Safe-linking - Inside of malloc() - REVEAL_PTR

```
static __always_inline void *
tcache_get (size_t tc_idx)
{
  tcache_entry *e = tcache->entries[tc_idx];
  if (__glibc_unlikely (!aligned_OK (e)))
    malloc_printerr ("malloc(): unaligned tcache chunk detected");
  tcache->entries[tc_idx] = REVEAL_PTR (e->next);
  --(tcache->counts[tc_idx]);
  e->key = 0;
  return (void *) e;
 }
```

Safe-linking - Inside of free() - PROTECT_PTR

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

  e->key = tcache_key;

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

Safe linking

```
#define PROTECT_PTR(pos, ptr)
  ((__typeof (ptr)) ((((size_t) pos) >> 12) ^ ((size_t) ptr)))

#define REVEAL_PTR(ptr)  PROTECT_PTR (&ptr, ptr)
```

Since both pos and ptr are on the heap, we can reveal a pointer using the mangled pointer alone:

```
def reveal_alone(ptr,offset=0):
    mid = ptr ^ ((ptr>>12)+offset)
    return mid ^ (mid>>24)
```

Malloc and free hook

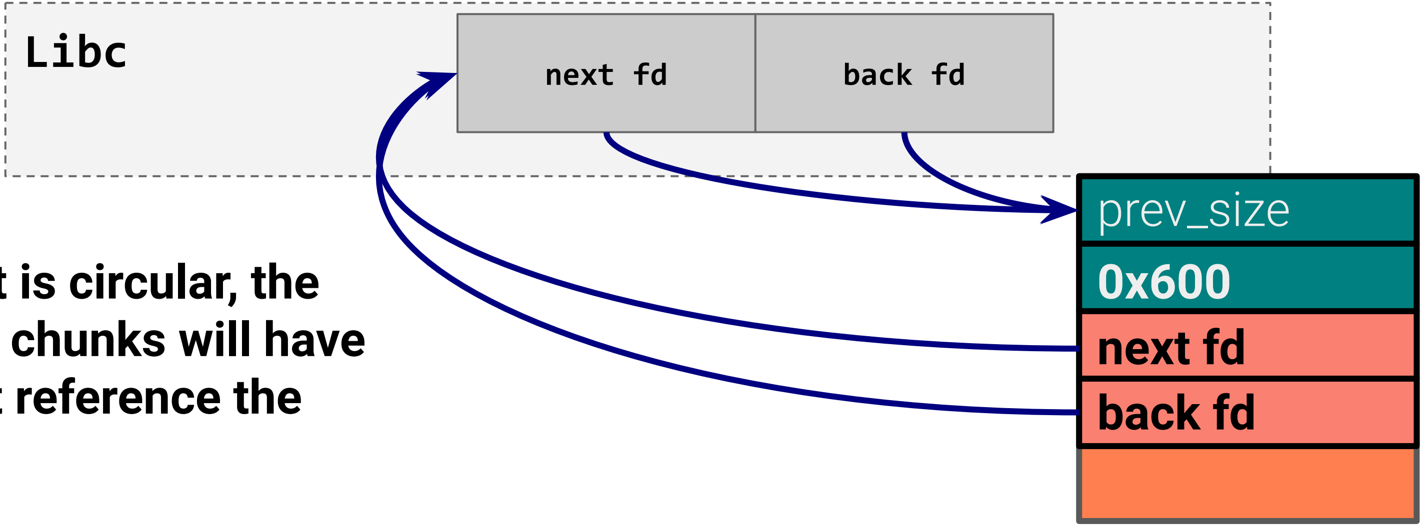
Before 2.34:
Malloc and free hook were a mechanism provided by glibc that allows you to intercept calls to the malloc and free function. By setting a custom hook, you can execute **your own code** whenever malloc or free is called, which can be useful for debugging.

```
void* my_malloc(size_t size) {

    if (size > 0x500) {
        printf("Allocating large chunk: %zu bytes\n", size);
    }

    __malloc_hook = NULL;
    void *result = malloc(size);
    __malloc_hook = my_malloc;

    return result;
}
```

If you have arbitrary write access, you can exploit those hooks to execute a **one-gadget** or call **system**: by setting the free hook to execute system() and passing a chunk containing '/bin/sh', you can trigger a shell execution.

Unsorted bin

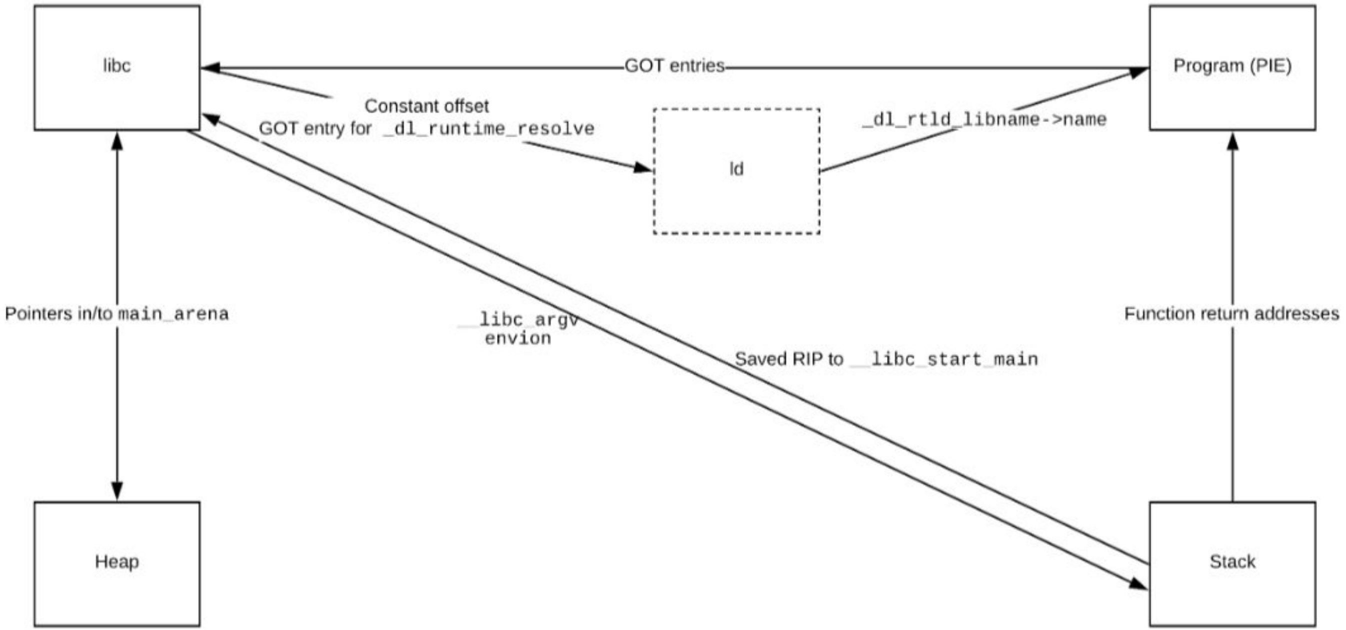
The unsorted bin is a **circular doubly linked list** that holds large and small chunks. When a chunk larger than what fits in the fast bins is freed, it ends up in the unsorted bin.
On malloc after the tcache and the fast bins, the unsorted bin is checked. If a chunk does not satisfy malloc, it is placed in the appropriate small/large bin.



Since the list is circular, the first and last chunks will have pointers that reference the libc itself!

Pivoting around memory

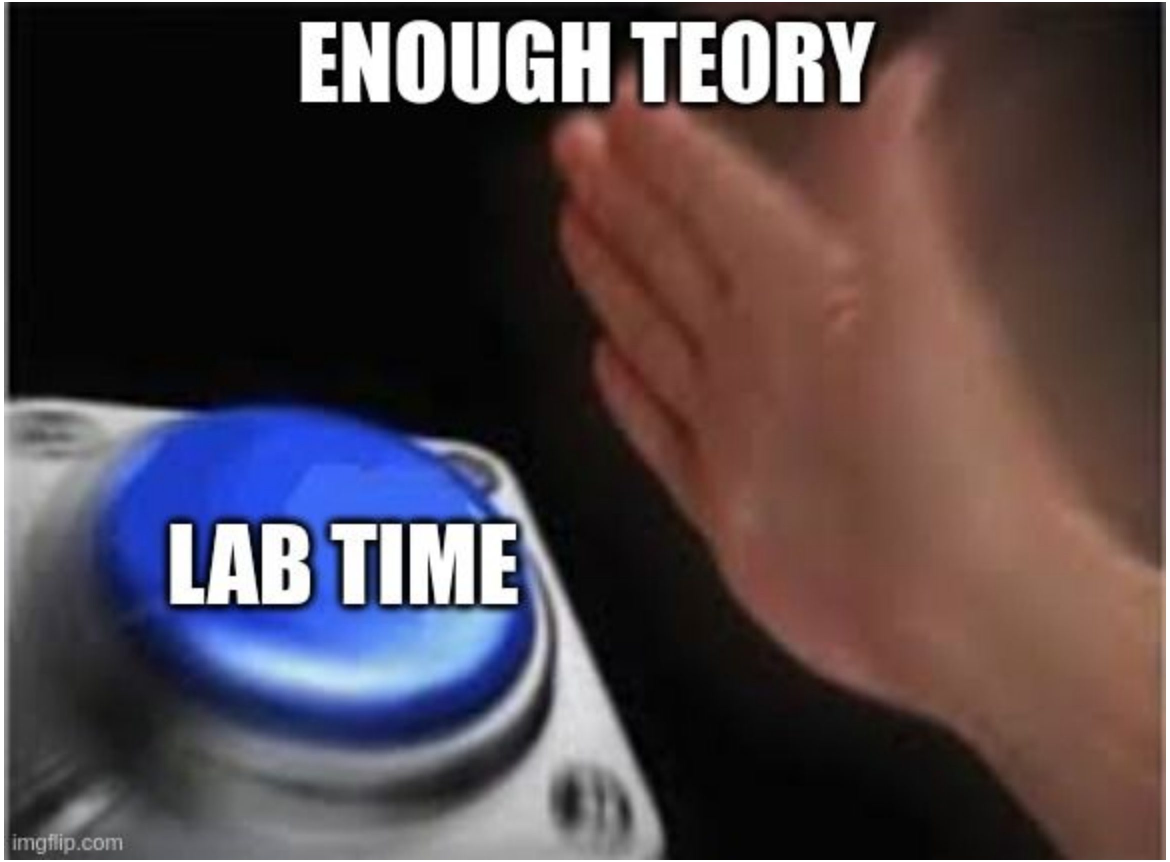
Once you leak the libc from the unsorted bin you can move around and leak other memory regions.



https://blog.osiris.cyber.nyu.edu/2019/04/06/pivoting-around-memory/

Further learning

https://azeria-labs.com/heap-exploitation-part-1-understanding-the-glibc-heap-implementation/
https://azeria-labs.com/heap-exploitation-part-2-glibc-heap-free-bins/
https://0x434b.dev/overview-of-glibc-heap-exploitation-techniques/
https://elixir.bootlin.com/glibc/latest/source/malloc/malloc.c
https://github.com/shellphish/how2heap


ENOUGH TEORY
LAB TIME
imgflip.com

Useful pwndbg heap commands

`bins` : state of all bins (tcache, fastbins, small/large & unsorted)
`heap` : state of the heap, all chunks & their metadata
`vis`🤢: hexdump of the heap colored based on chunks + some metadata (bin pointers & top chunk)
`try_free` : see if `free(addr)` would succeed, with a breakdown of the security checks and failures; adapts to the linked libc!
`malloc_chunk` : see the metadata of the chunk at `addr`
`find_fake_fast` : helps you fake fastbin chunks overlapping the memory at `addr` (sizes & alignment constraints)