Module: Dynamic Allocator Mis

What is the Heap?

Yan Shoshitaishvili
Arizona State University

From pwn.college!

Recap: Types of Memory

Memory comes in different types...

ELF .text: where the code lives
ELF .plt: where library function stubs live
ELF .got: where pointers to imported symbols live
ELF .bss: used for uninitialized global writable data (such as global arrays without initial values)
ELF .data: used for pre-initialized global writable data (such as global arrays with initial values)
ELF .rodata: used for global read-only data (such as string constants)
stack: local variables, temporary storage, call stack metadata

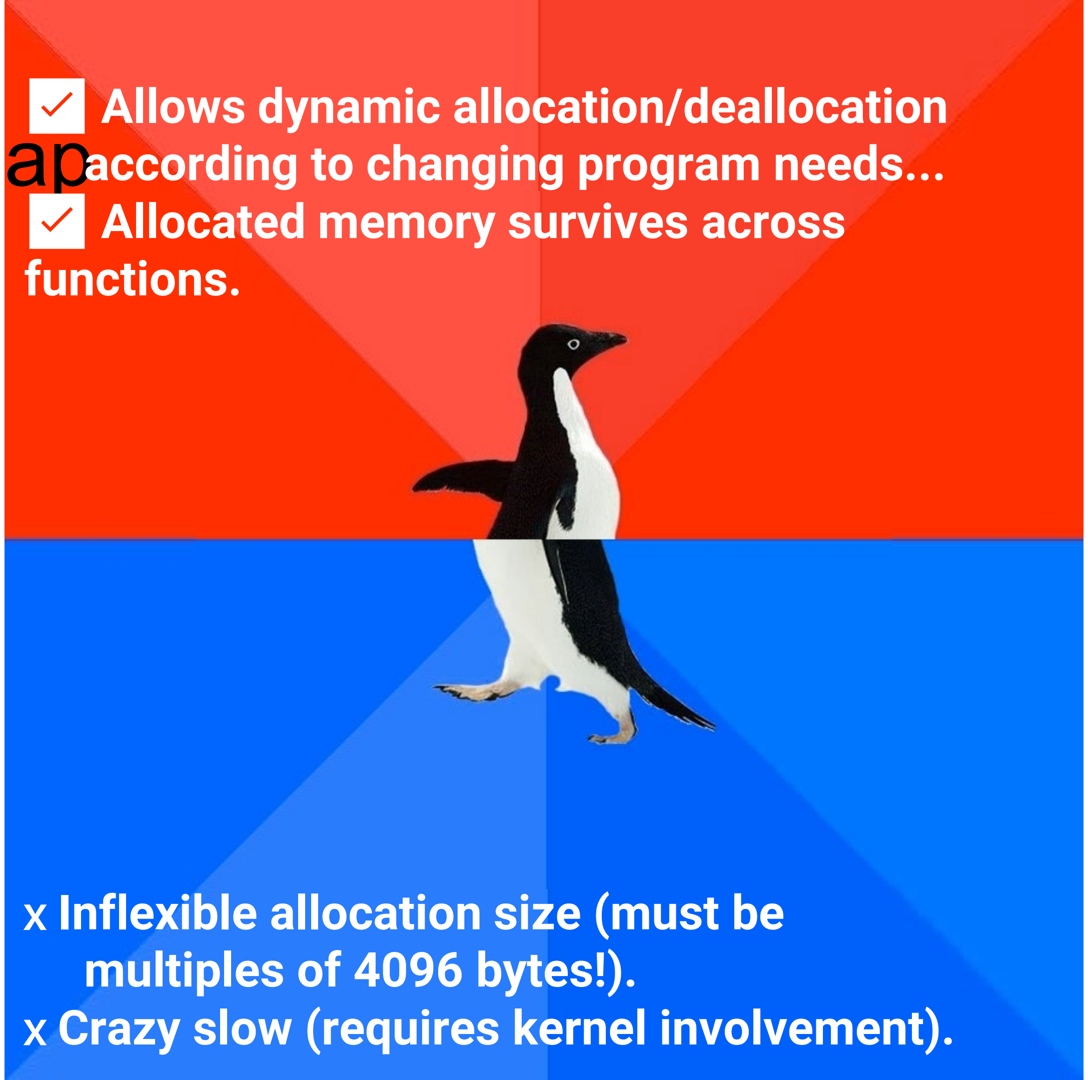
But what if you needed a place to store long-lived *dynamic* memory. Example: a variable-length list of NPCs in a game.

What if you needed *dynamic memory allocation*?

One idea: mmap

What if we **`mmap()`**ed memory as we need it?

```
mmap(0, num_pages*0x1000, ...)
```

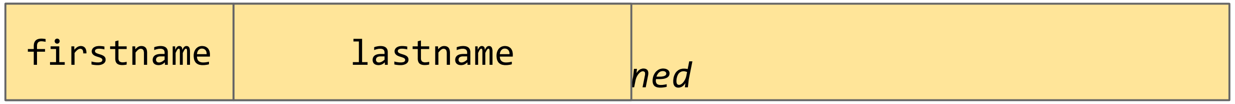

✓ Allows dynamic allocation/deallocation according to changing program needs...
✓ Allocated memory survives across functions.
x Inflexible allocation size (must be multiples of 4096 bytes!).
x Crazy slow (requires kernel involvement).

Smarter solution...

What if we...
... wrote a library...
... that **`mmap()`**ed a bunch of memory...
... and handed out small chunks of it...
... on demand!

The library could be used like:

```
char *firstname = allocate_memory(128);
char *lastname = allocate_memory(256);
scanf("%s %s", firstname, lastname);
printf("Hello %s %s!", firstname, lastname);
free_memory(firstname);
free_memory(lastname);
```



Dynamic Allocators exist!

We're not the first to have this idea:

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

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

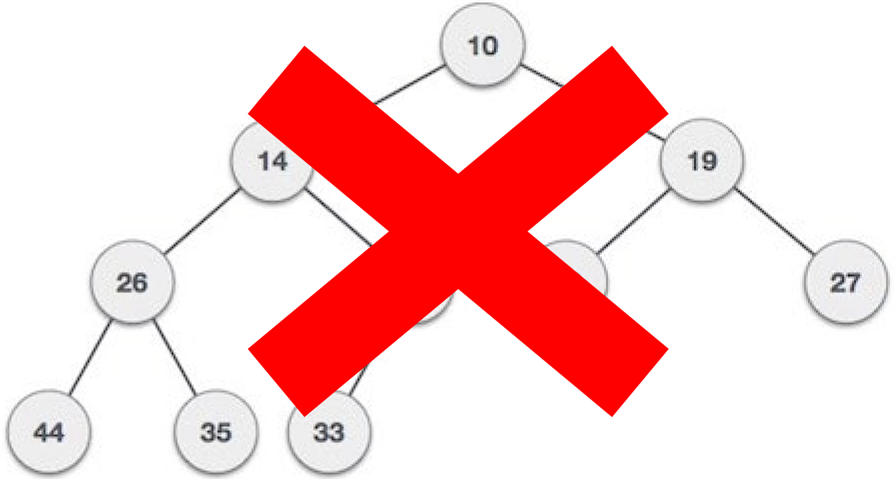
FreeBSD:
jemalloc (also used in Firefox, Android)

Windows:
Segment Heap, NT Heap

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

Terminology: "The Heap"

The memory space managed by a dynamic allocator is colloquially known as **"The Heap"**.



What does the heap do?

The heap, as implemented by ptmalloc/glibc (and analogues), 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

These functions are used, extensively, by practically every single non-trivial piece of software.

How does the heap work?

ptmalloc actually does not use mmap!

The Data Segment:

- historic oddity from segmented memory spaces of yore
- with ASLR, placed randomly into memory *near-ish* the PIE base
- starts out with a size of `0`
- managed by the `brk` and `sbrk` system calls:
  `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`

Under the hood, this is managed just like `mmap()`.

ptmalloc slices off bits of the data segment for small allocations,
and uses `mmap()` for large allocations.

Dangers of the heap

What can go wrong?

The heap is:

1. used by imperfect human programmers
    - humans forget to free memory
    - humans forget all the spots where they store pointers to data
    - humans forget what they've freed
2. a library that strives for *performance*
    - allocation and deallocation needs to be fast, or programs will slow down
    - optimizations often leave security as an afterthought

Bugs caused by #1 become security issues due to #2 if not caught!

Here lies danger...

How to detect issues?

- valgrind can detect heap misuse (if your testcases trigger it)
- glibc itself has some hardening techniques:
  - export MALLOC_CHECK_=1
  - export MALLOC_PERTURB_=1
  - export MALLOC_MMAP_THRESHOLD_=1
- there are various "more secure" allocators being developed (but not really deployed)

Like many other issues, no general techniques exist for detecting dynamic allocation errors...

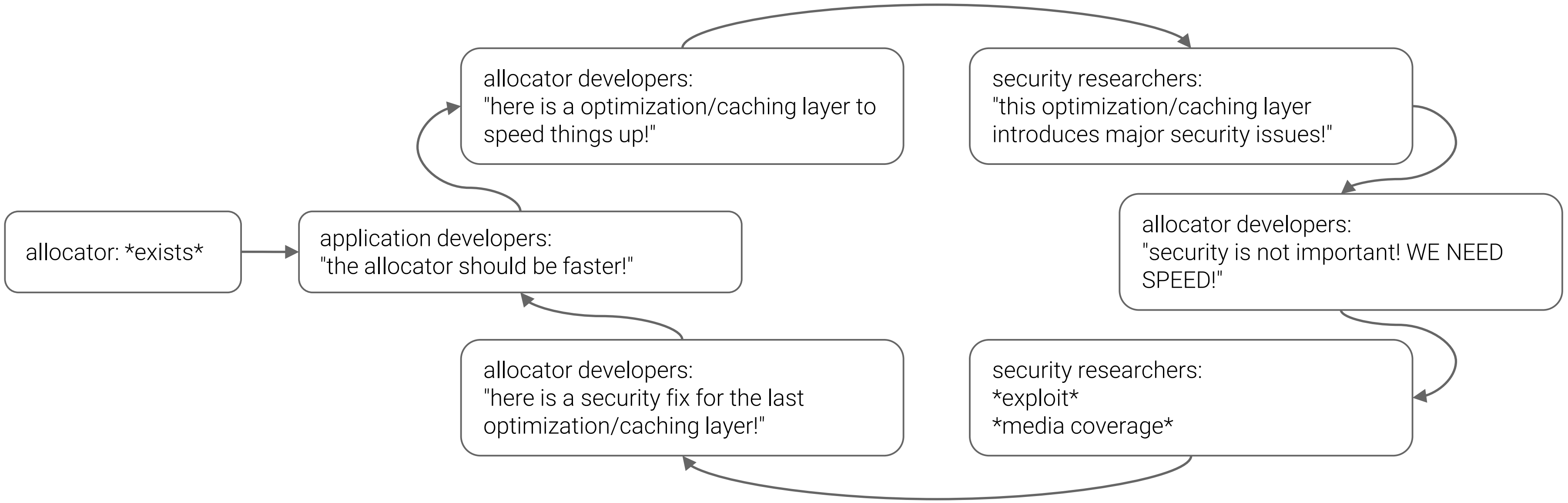
Module: Dynamic Allocator Mis

Here Lies Danger

Yan Shoshitaishvili
Arizona State University

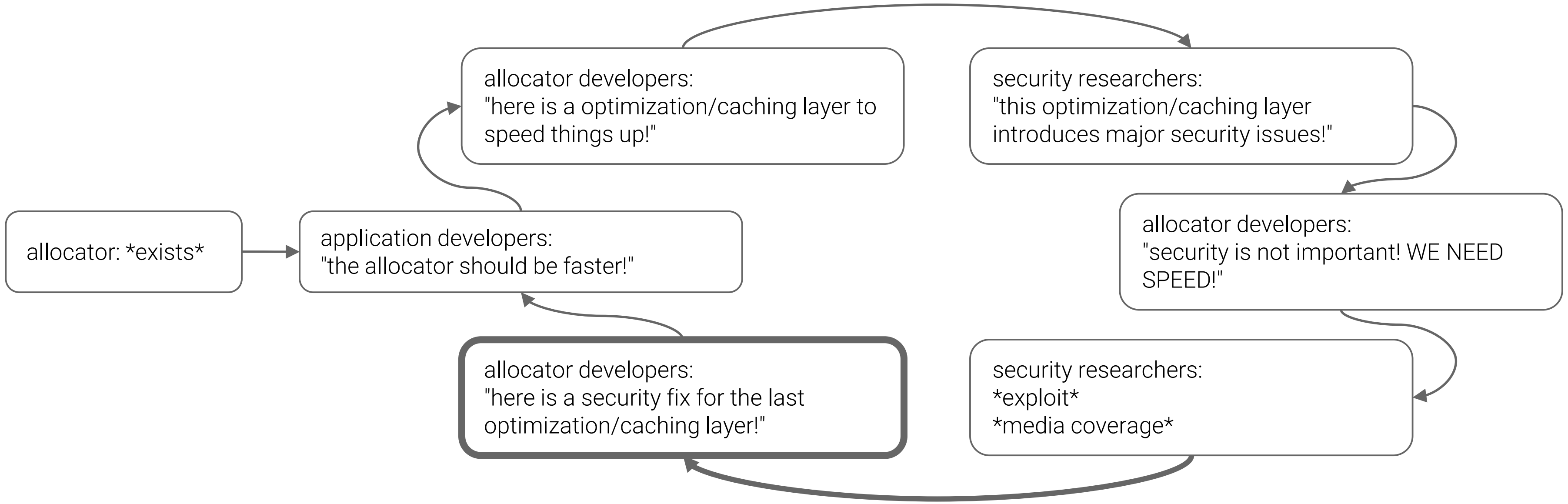
Dangers of the heap security vs performance

The lifecycle of allocator security.


allocator: *exists*
application developers: "the allocator should be faster!"
allocator developers: "here is a optimization/caching layer to speed things up!"
security researchers: "this optimization/caching layer introduces major security issues!"
allocator developers: "security is not important! WE NEED SPEED!"
security researchers: *exploit* *media coverage*
allocator developers: "here is a security fix for the last optimization/caching layer!"

Dangers of the heap security vs performance

The lifecycle of allocator security.



allocator: *exists*

application developers:
"the allocator should be faster!"

allocator developers:
"here is a optimization/caching layer to speed things up!"

security researchers:
"this optimization/caching layer introduces major security issues!"

allocator developers:
"security is not important! WE NEED SPEED!"

security researchers:
exploit
media coverage

allocator developers:
"here is a security fix for the last optimization/caching layer!"

Dangers of the heap

More than one way to misuse the heap!

- Forgetting to free memory.
    - leads to resource exhaustion
- Forgetting that we *have* freed memory.
    - using free memory
    - freeing free memory
- Corrupting metadata used by the allocator to keep track of heap state.
    - conceptually similar to corruption internal function state on the stack

Dangers of the heap: Memory Leaks

Problem: Allocated memory must be explicitly freed.

```
int foo()
{
        char *blah = malloc(1024);
        // use blah in safe ways
        return 1;
}
```

What happens with the memory pointed to by blah?

Why is this a security issue?

Dangers of the heap: Use After Free

Pointers to an allocation remain valid after **`free()`**ing the allocation, and might be used afterwards! Why is this bad?

```
int main() {
    char *user_input = malloc(8);

    printf("Name? ");
    scanf("%7s", user_input);
    printf("Hello %s!\n", user_input);
    free(user_input);

    long *authenticated = malloc(8);
    *authenticated = 0;

    printf("Password? ");
    scanf("%7s", user_input);

    if (getuid() == 0 || strcmp(user_input, "hunter2") == 0) *authenticated = 1;
    if (*authenticated) sendfile(0, open("/flag", 0), 0, 128);
}
```

Dangers of the heap: Memory Disclosure

Simple case: UAF, but with a `printf()` instead of a `scanf()`.

Complex case: Some heap implementations (including dlmalloc and ptmalloc) reuse `free()`d chunks to store metadata.

```
int main() {
    char *password = malloc(8);
    char *name = malloc(8);

    printf("Password? ");
    scanf("%7s", password);
    assert(strcmp(password, "hunter2") == 0);
    free(password);

    printf("Name? ");
    scanf("%7s", name);
    printf("Hello %s!\n", name);
    free(name);

    printf("Goodbye, %s!\n", name);
}
```

Dangers of the heap: Metadata Corruption

Allocator metadata can be *written*, not just read, to cause crazy effects.

One of the earliest widespread heap exploits, developed by Solar Designer in 2000:
https://www.openwall.com/articles/JPEG-COM-Marker-Vulnerability

Soon formalized in hacker literature in 2001:

- Vudo malloc tricks, by MaXX:
  http://phrack.org/issues/57/8.html
- Once upon a free(), by anonymous:
  http://phrack.org/issues/57/9.html

Now a whole genre in the hacking scene!

Hackers are Weird: The Rise of the Houses

Phantasmal Phantasmagoria developed a "lore" around heap metadata corruption: https://seclists.org/bugtraq/2005/Oct/118

Described and named a number of metadata corruption techniques:

- The House of Prime
- The House of Mind
- The House of Force
- The House of Lore
- The House of Spirit
- The House of Chaos

Things got out of hand quick. Later work:

- House of Underground
- House of Orange
- House of Einherjar
- House of Rabbit
- House of Botcake

The Danger of Overlapping Allocations

Typically, heap metadata corruption is used to confuse the allocator into allocating *overlapping memory*. As we saw with UAF, this can be extremely dangerous.

If two pointers are pointing to the same memory
... and one of the pointers is treated by the program in a security-critical manner
... and the other one can be written to or read by an attacker
... it's game over!

"Security-critical manner?"

- `authentication` variables
- function pointers (control flow hijack)
- program metadata such as length (inducing memory errors)
- sensitive data (such as the flag)

Module: Dynamic Allocator Mis

Metadata and Chunks

Yan Shoshitaishvili
Arizona State University

Heap Metadata and its Corruption

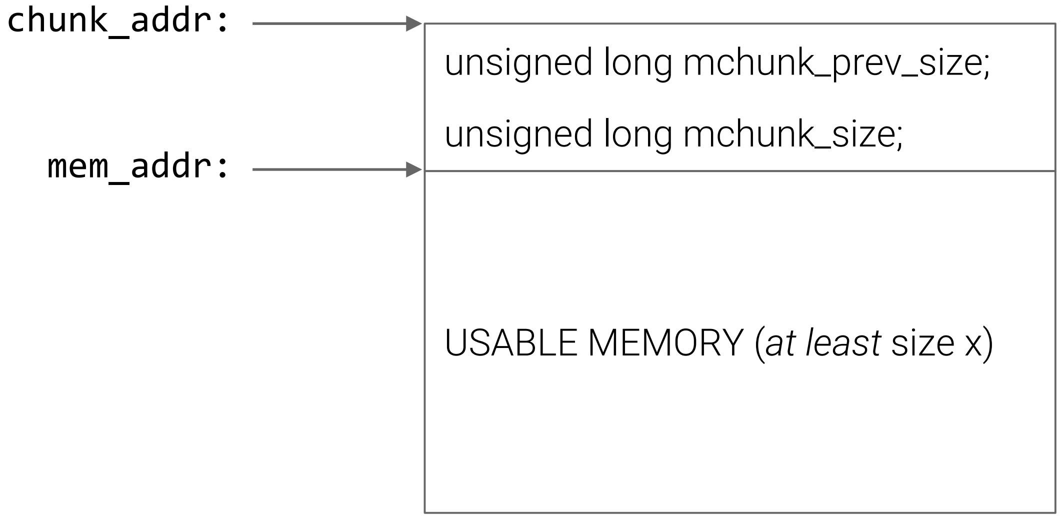
As we saw with tcache, the ptmalloc uses a bunch of metadata to track its operation. It keeps them in:

1. global metadata (i.e., the tcache structure)
2. per-chunk metadata

What's a chunk?

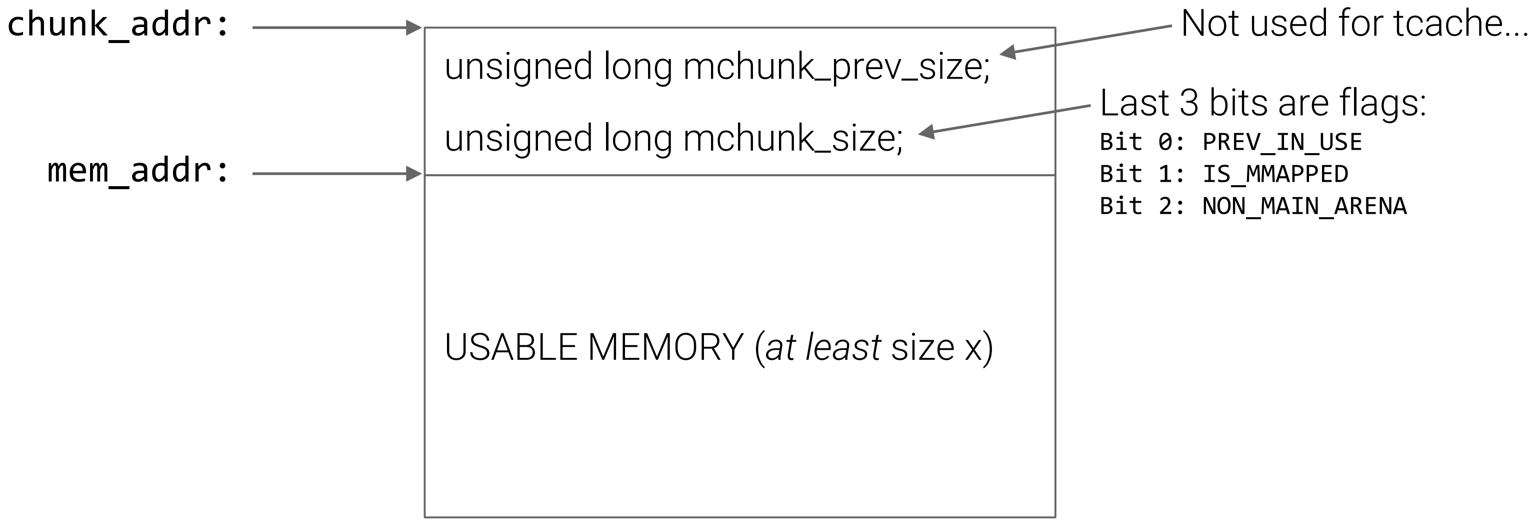
Metadata: Allocated Chunks

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


chunk_addr:
mem_addr:
unsigned long mchunk_prev_size;
unsigned long mchunk_size;
USABLE MEMORY (*at least* size x)

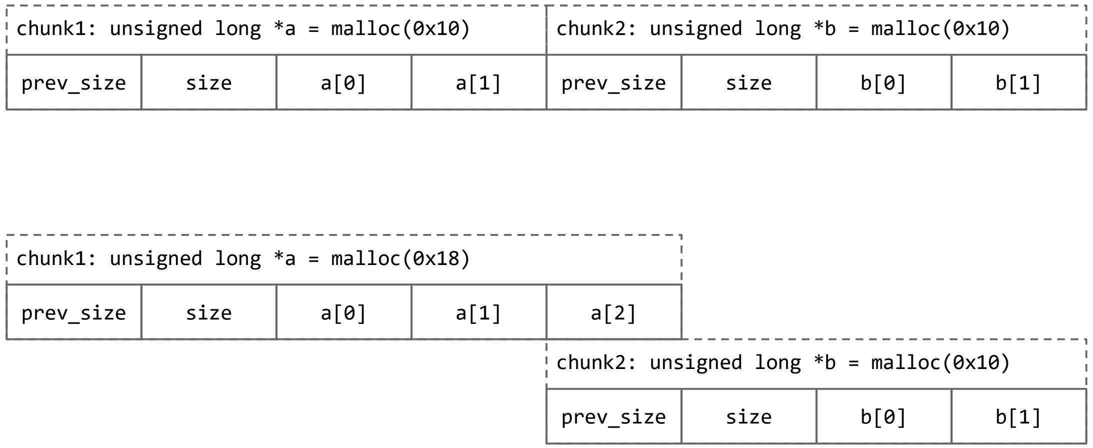
Metadata: Size?

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



Metadata: Overlapping metadata!

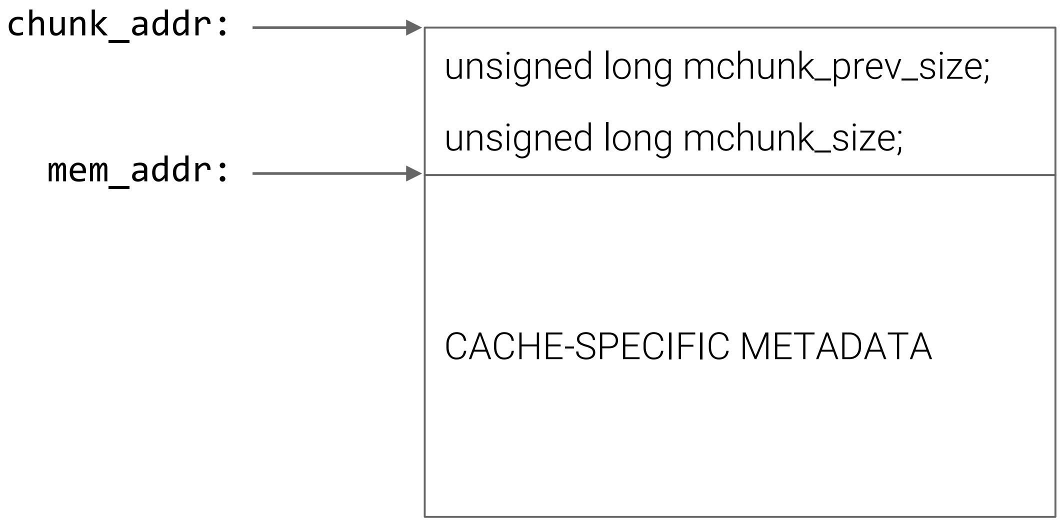
To save memory, the `prev_size` field of a chunk whose PREV_INUSE flag is set (i.e., the previous chunk is not free) **is used by the previous chunk**!



chunk1: unsigned long *a = malloc(0x10)
chunk2: unsigned long *b = malloc(0x10)
prev_size
size
a[0]
a[1]
prev_size
size
b[0]
b[1]
chunk1: unsigned long *a = malloc(0x18)
prev_size
size
a[0]
a[1]
a[2]
chunk2: unsigned long *b = malloc(0x10)
prev_size
size
b[0]
b[1]

Metadata: Freed Chunks

As we saw with tcache, a **`free()`**d chunk has additional metadata about the location of other chunks:



Metadata: Different Caches

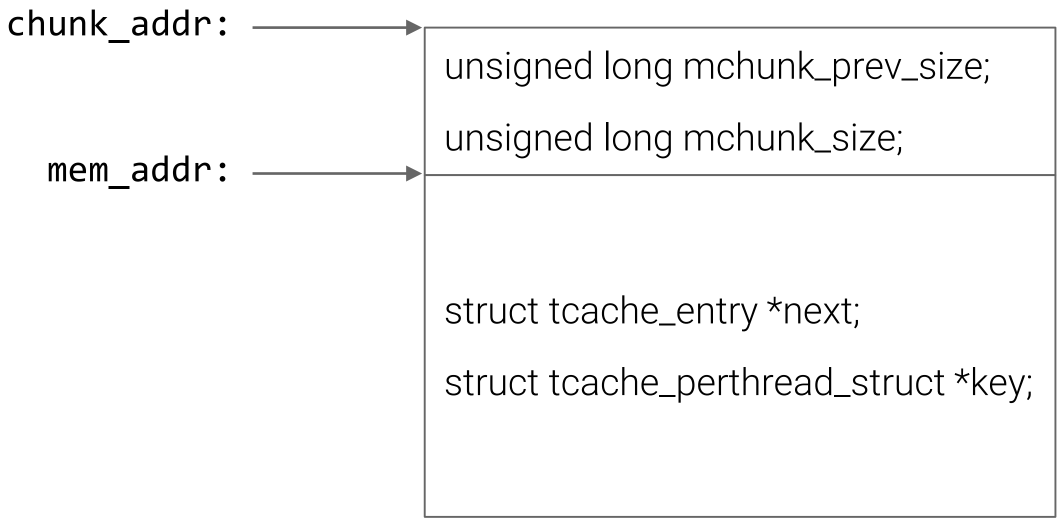
This information is constantly changing (see: tcache) and PTMALLOC IS VERY COMPLEX. This is an approximation.

Currently, the ptmalloc caching design is (in order of use):

1. 64 singly-linked tcache bins for allocations of size 16 to 1032 (functionally "covers" fastbins and smallbins)
2. 10 singly-linked "fast" bins for allocations of size up to 160 bytes
3. 1 doubly-linked "unsorted" bin to quickly stash **`free()d`** chunks that don't fit into tcache or fastbins
4. 64 doubly-linked "small" bins for allocations up to 512 bytes
5. doubly-linked "large" bins (anything over 512 bytes) that contain different-sized chunks

Metadata: tcache Chunks

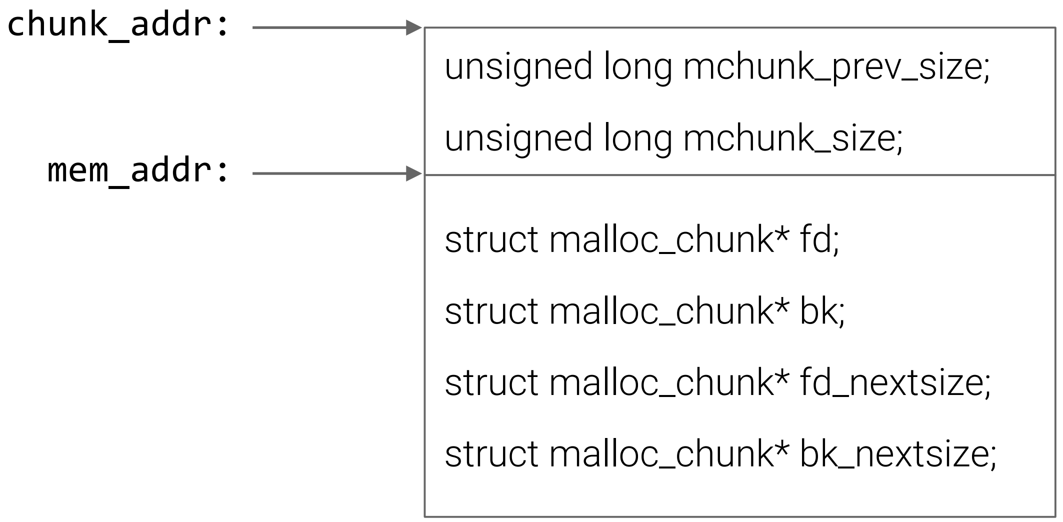
Free tcache-cached chunks have a pointer to the allocated space of the next chunk and a pointer to the per-thread struct.


chunk_addr:
mem_addr:
unsigned long mchunk_prev_size;
unsigned long mchunk_size;
struct tcache_entry *next;
struct tcache_perthread_struct *key;

Metadata: largebin Chunks

When `free()d`, large are:

1. Consolidated with adjacent free chunks.
2. Put into an "unsorted" bin regardless of size.
3. Properly put into a doubly-linked list later, during the next allocation that they fail to "satisfy".


chunk_addr:
mem_addr:
unsigned long mchunk_prev_size;
unsigned long mchunk_size;
struct malloc_chunk* fd;
struct malloc_chunk* bk;
struct malloc_chunk* fd_nextsize;
struct malloc_chunk* bk_nextsize;

Metadata: The Wilderness

The heap is a finite-sized allocation that needs to be *manually* expanded.

During allocation, **`malloc()`** will (simplified view):

1. Look for a free chunk that will satisfy the allocation, and return it.
2. Otherwise, check the "available" space left at the end of the heap. If there is enough there, return that and reduce the "available" space.
3. If there isn't enough "available" space, **`malloc()`** will **`mmap()`** certain large allocations.
4. Otherwise, malloc will grow the heap with **`brk()`** and go to #2.

How does **`malloc()`** store the available space? In the "Wilderness", a fake chunk at the end of the heap that stores the available space.

Other Allocators?

Allocators are different! This metadata discussion, and tcache, is very ptmalloc-specific.

Example: jemalloc has no inline metadata!

Module: Dynamic Allocator Mis

Heap Metadata Corruption

Yan Shoshitaishvili
Arizona State University

Recap: Metadata Corruption Goals

What might we want to achieve with heap metadata corruption?

Modify arbitrary memory.

Achieve an **overlapping allocation** (with other heap structures, stack, etc).

Use either of those capabilities for further control.

```
struct malloc_chunk {
  unsigned long mchunk_prev_size;
  unsigned long mchunk_size;
  struct malloc_chunk* fd;
  struct malloc_chunk* bk;
  struct malloc_chunk* fd_nextsize;
  struct malloc_chunk* bk_nextsize;
};

typedef struct malloc_chunk* mchunkptr;
```

Historical: The Unlink Attack

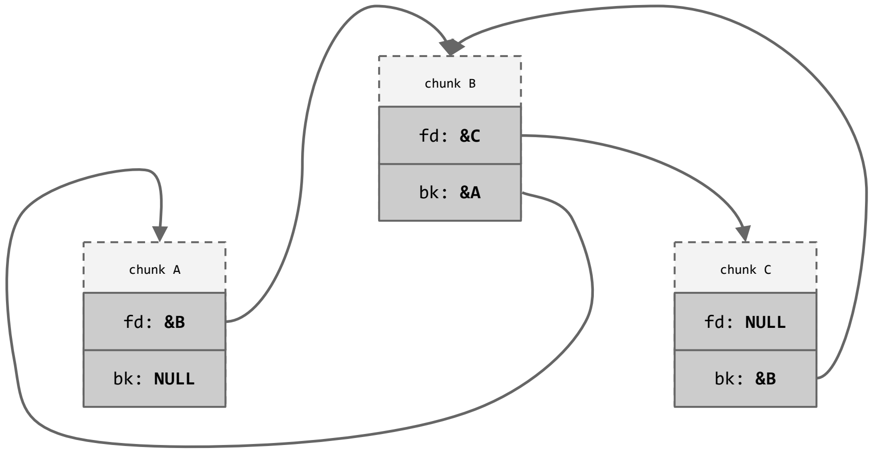
Recall the struct (to the right).

When a chunk (bigger than the tcache size) is allocated, it is removed from the doubly-linked list. This looks like:

```
chunk->fd->bk = chunk->bk;
chunk->bk->fd = chunk->fd;
```

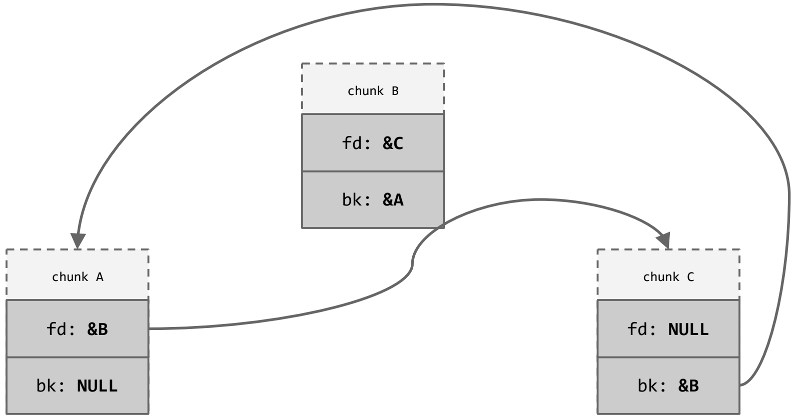
Historical: The Unlink Attack

```
struct malloc_chunk {
  unsigned long mchunk_prev_size;
  unsigned long mchunk_size;
  struct malloc_chunk* fd;
  struct malloc_chunk* bk;
  struct malloc_chunk* fd_nextsize;
  struct malloc_chunk* bk_nextsize;
};

typedef struct malloc_chunk* mchunkptr;
```



```
struct malloc_chunk {
  unsigned long mchunk_prev_size;
  unsigned long mchunk_size;
  struct malloc_chunk* fd;
  struct malloc_chunk* bk;
  struct malloc_chunk* fd_nextsize;
  struct malloc_chunk* bk_nextsize;
};

typedef struct malloc_chunk* mchunkptr;
```

Historical: The Unlink Attack



```
struct malloc_chunk {
  unsigned long mchunk_prev_size;
  unsigned long mchunk_size;
  struct malloc_chunk* fd;
  struct malloc_chunk* bk;
  struct malloc_chunk* fd_nextsize;
  struct malloc_chunk* bk_nextsize;
};

typedef struct malloc_chunk* mchunkptr;
```

Historical: The Unlink Attack

Recall the struct (to the right).

When a chunk (bigger than the tcache size) is allocated, it is removed from the doubly-linked list. This looks like:

```
    chunk->fd->bk = chunk->bk;
    chunk->bk->fd = chunk->fd;
```

If you control chunk->fd and chunk->bk, you can overwrite an arbitrary location in memory with an arbitrary (but valid) pointer.

Extra checks (`chunk->fd->bk == chunk && chunk->bk->fd == chunk`) have heavily weakened this attack (though you can still pass this check if you inject a chunk!).

```
struct malloc_chunk {
  unsigned long prev_size;
  unsigned long size;
  struct malloc_chunk* fd;
  struct malloc_chunk* bk;
  struct malloc_chunk* fd_nextsize;
  struct malloc_chunk* bk_nextsize;
};
```

Historical: Poison Null Byte

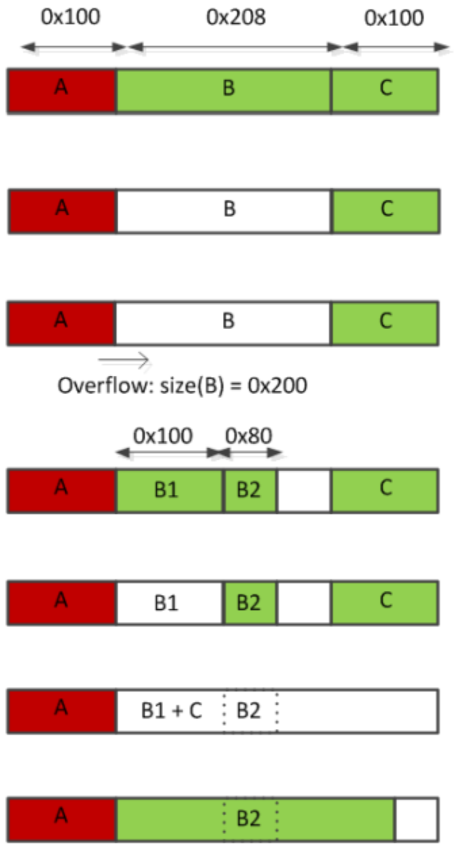
What if all we have is a single 0-byte write? This is common, with off-by-one string operations!

```
    buf = malloc(0x1008);
    int read_length = read(0, 0x1008, 128);
    buf[read_length] = 0;
```

What's the problem here?

```
struct malloc_chunk {
  unsigned long prev_size;
  unsigned long size;
  struct malloc_chunk* fd;
  struct malloc_chunk* bk;
  struct malloc_chunk* fd_nextsize;
  struct malloc_chunk* bk_nextsize;
};
```

Historical: Poison Null Byte



Initial state

B is free

Overflow into B
- Size truncated to 0x200 from 0x208
- Further allocations in that space do not properly update C's "prev_size" field

Two allocations within the old B chunk
The first is not a fastbin

The beginning of the old B chunk is free

C is freed and merged with the old B, where a valid non-fastbin free chunk resides

1+ allocations larger than B1's initial size
B2 is overlapped

Figure source: https://heap-exploitation.dhavalkapil.com/attacks/shrinking_free_chunks.html

Historical: House of Force

My personal favorite house, but now patched...

The wilderness is just hanging out on the heap!

What if we overwrite it with a humongous number?

We could control where stuff will be allocated, and get an overlapping allocation!!!

House of Spirit

Straight-forward: exploit the fact that there are very few checks done at `free()` time!

1. Forge something that looks like a chunk.
2. `free()` it.
3. The next `malloc()` will return that chunk to you!

With a pointer overwrite, can be used to later `malloc()` a stack pointer.

Can be done with or without tcache.

Miscellaneous Heapery

How do you trigger a `malloc()` in an uncooperative program?

`printf()`, `scanf()`, and friends will use `malloc()` to allocate space during operation!

With the right setup, can lead straight to an overwrite.

If you want to disable this in your own programs:

```
setbuf(stdout, NULL);
    setbuf(stdin, NULL);
```

Major Heap Headache: Heap Massaging

Heap exploitation requires precise heap layout.

Programs are constantly messing with the heap.

Some automation is being worked on (https://www.usenix.org/conference/usenixsecurity18/presentation/heelan)!

Mastering this is **hard**.

Further reading

Educational heap exploitation resources by Shellphish:
https://github.com/shellphish/how2heap

A guidebook on the heap:
https://heap-exploitation.dhavalkapil.com

Automated heap security analysis engine:
https://github.com/angr/heaphopper

Automatic Techniques to Systematically Discover New Heap Exploitation Primitives

Insu Yun, Dhaval Kapil, and Taesoo Kim

Georgia Institute of Technology

Module: Dynamic Allocator Mis

tcache

Yan Shoshitaishvili
Arizona State University

tcache

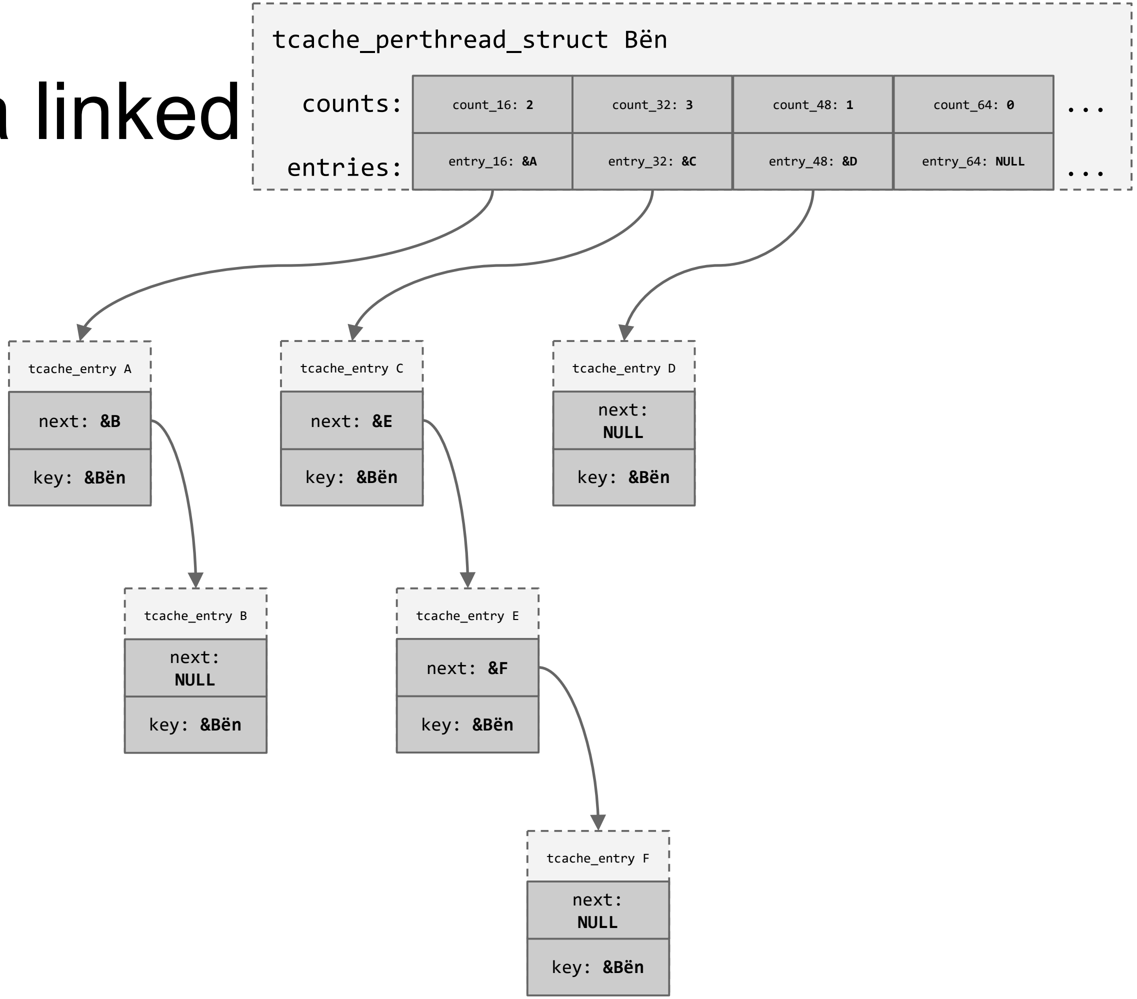
"Thread Local Caching" in ptmalloc, to speed up repeated (small) allocations in a single thread.

Implemented as a **singly-linked** list, with each thread having a list header for different-sized allocations.

```
typedef struct tcache_perthread_struct
{
  char counts[TCACHE_MAX_BINS];
  tcache_entry *entries[TCACHE_MAX_BINS];
} tcache_perthread_struct;
```

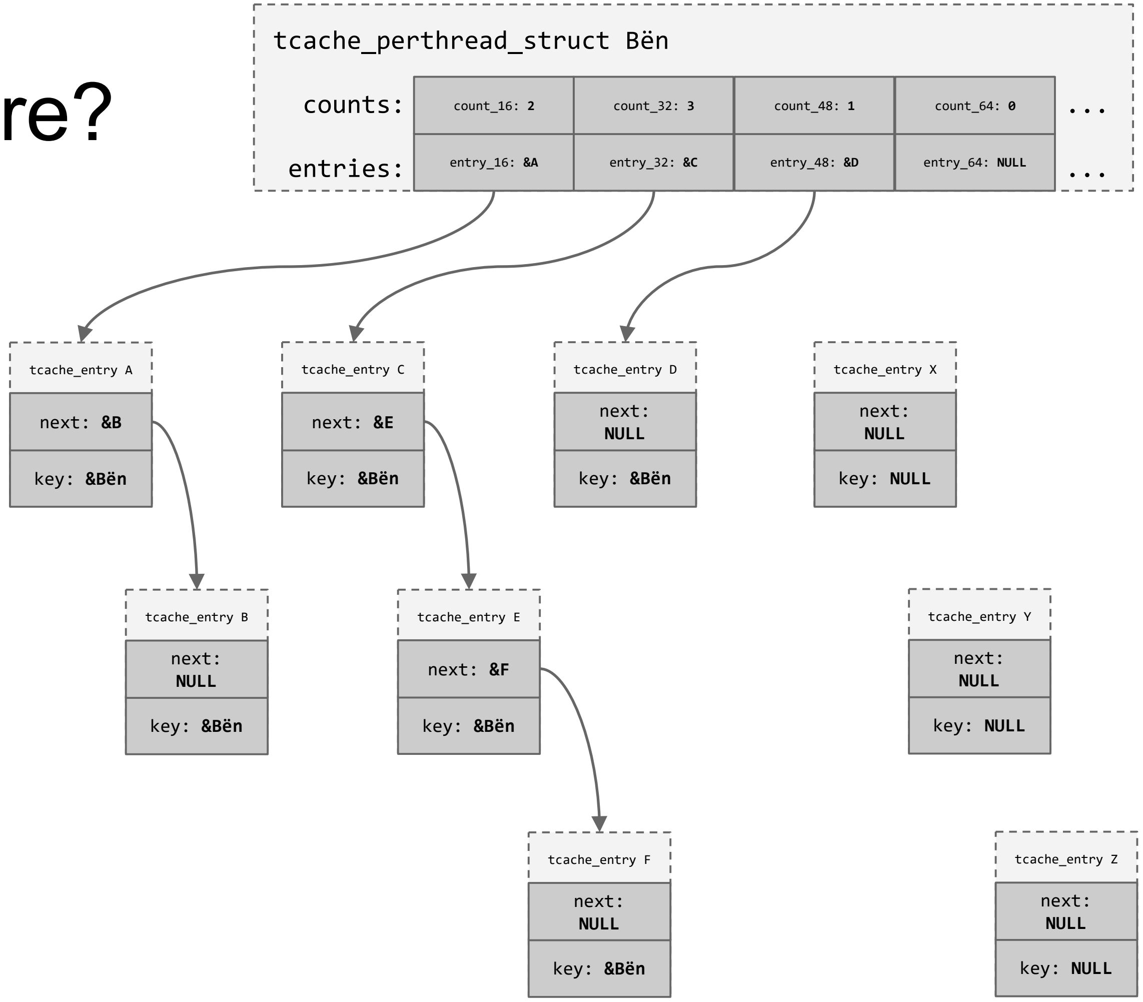
Interlude: what is a linked

```
typedef struct tcache_perthread_struct
{
  char counts[TCACHE_MAX_BINS];
  tcache_entry *entries[TCACHE_MAX_BINS];
} tcache_perthread_struct;

typedef struct tcache_entry
{
  struct tcache_entry *next;
  struct tcache_perthread_struct *key;
} tcache_entry;
```



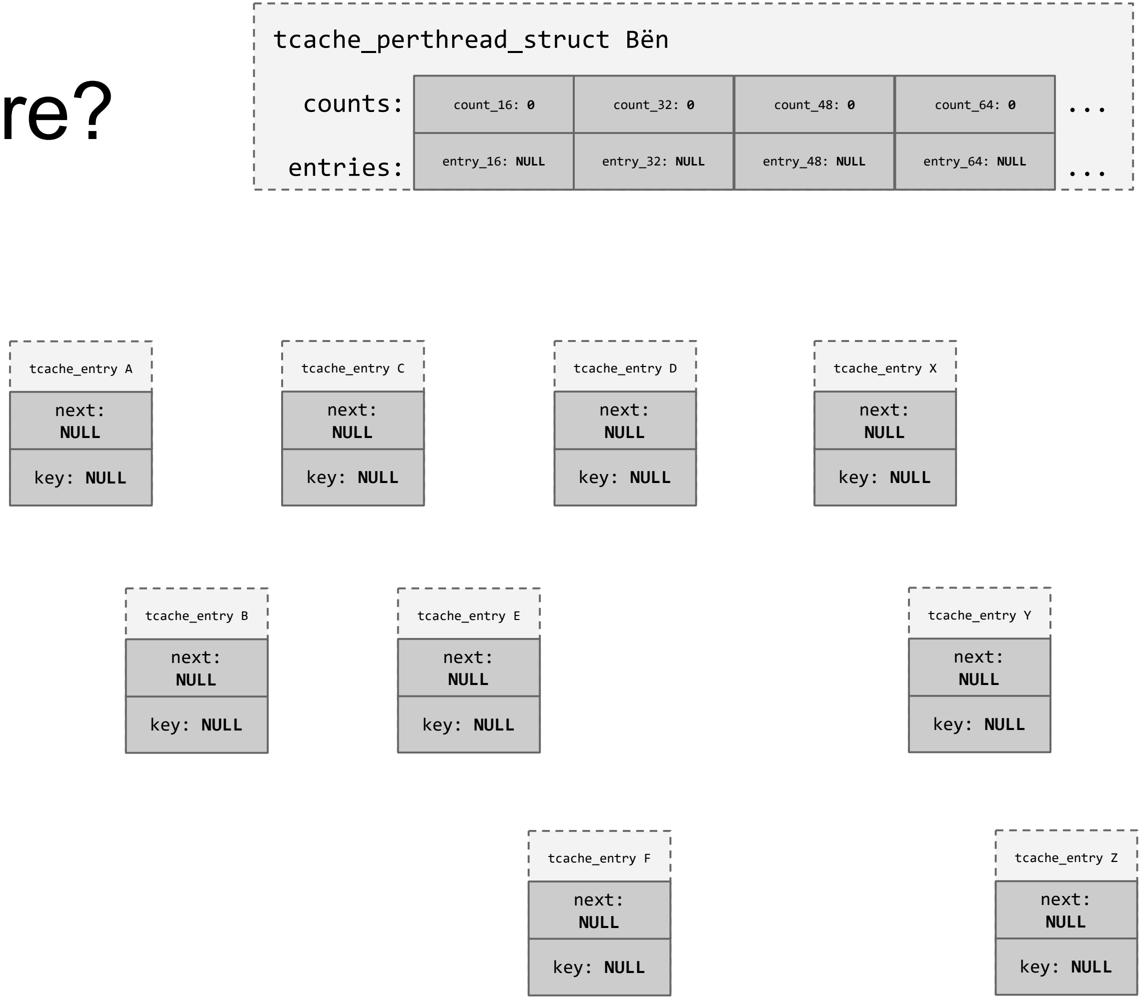
How did we get here?

```
a = malloc(16);
b = malloc(16);
c = malloc(32);
d = malloc(48);
e = malloc(32);
f = malloc(32);

// allocations that are not freed
// don't show up in the tcache!
x = malloc(64);
y = malloc(64);
z = malloc(64);

// later freed allocations show up
// earlier in the tcache list order
free(b);
free(a);
free(f);
free(e);
free(c);
free(d);
```



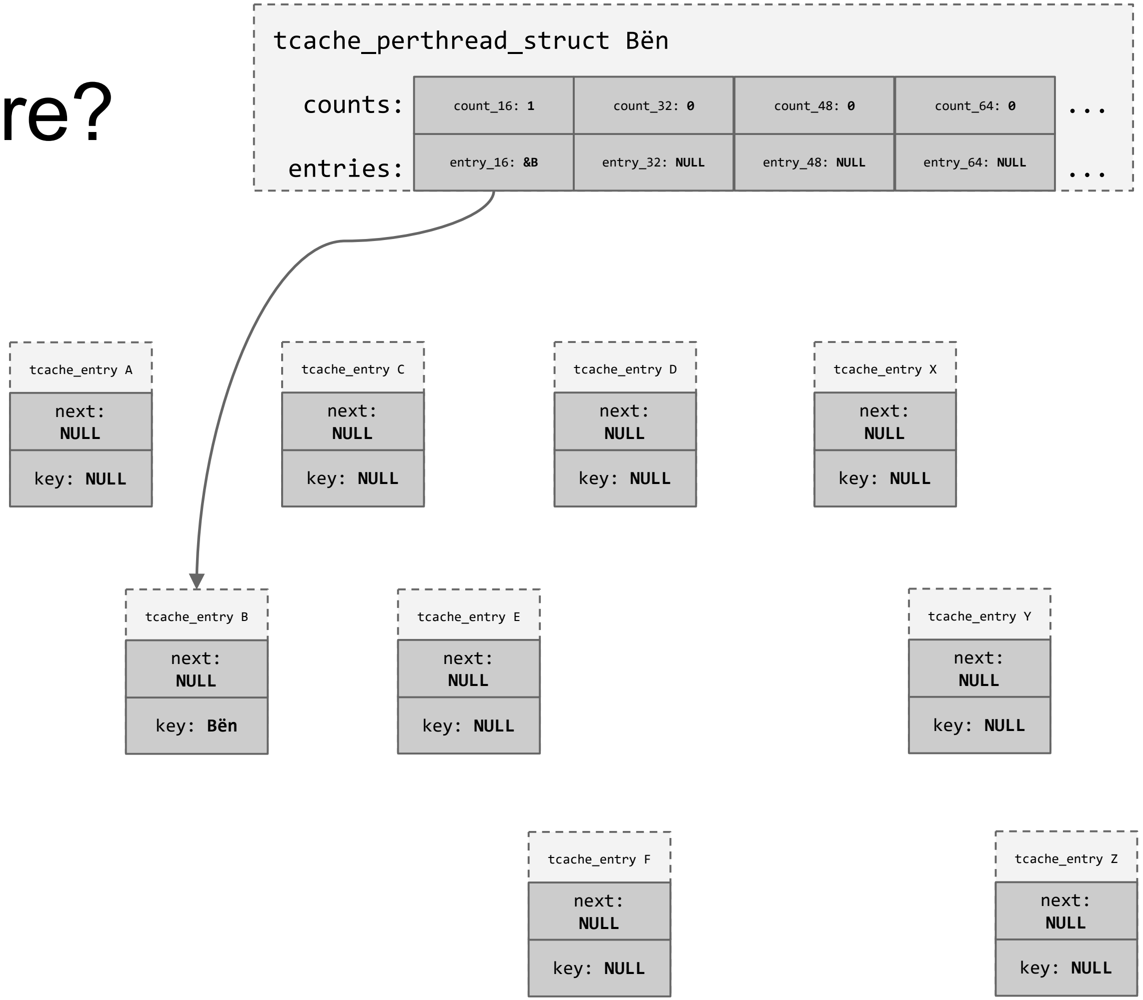
How did we get here?

```
a = malloc(16);
b = malloc(16);
c = malloc(32);
d = malloc(48);
e = malloc(32);
f = malloc(32);

// allocations that are not freed
// don't show up in the tcache!
x = malloc(64);
y = malloc(64);
z = malloc(64);

// later freed allocations show up
// earlier in the tcache list order
free(b);
free(a);
free(f);
free(e);
free(c);
free(d);
```



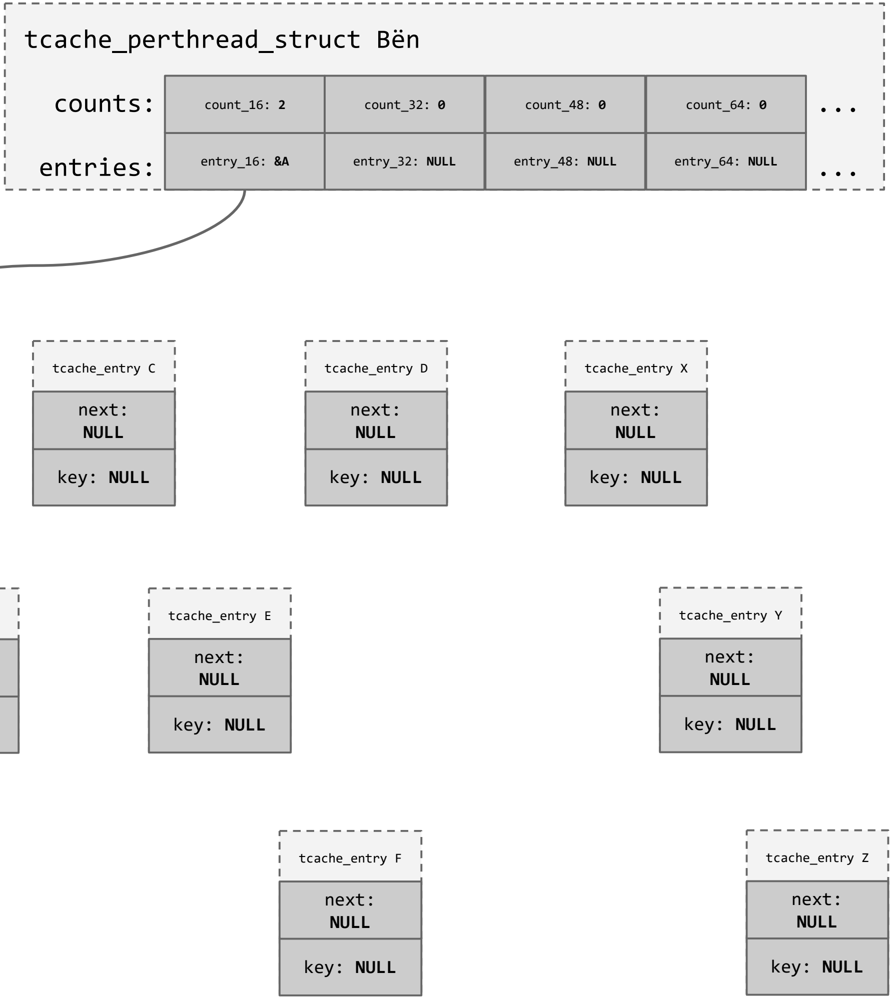
How did we get here?

```
a = malloc(16);
b = malloc(16);
c = malloc(32);
d = malloc(48);
e = malloc(32);
f = malloc(32);

// allocations that are not freed
// don't show up in the tcache!
x = malloc(64);
y = malloc(64);
z = malloc(64);

// later freed allocations show up
// earlier in the tcache list order
free(b);
free(a);
free(f);
free(e);
free(c);
free(d);
```



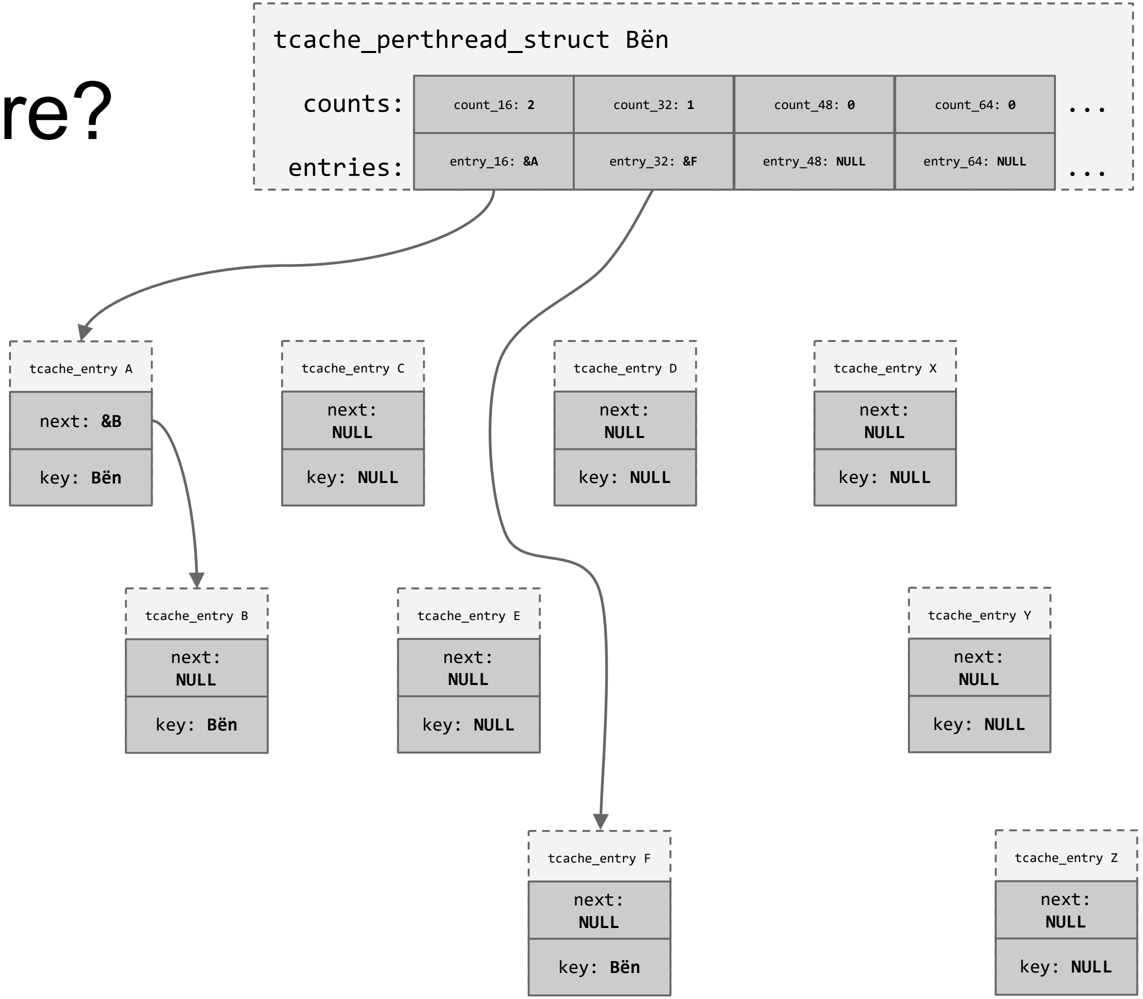
How did we get here?

```
a = malloc(16);
b = malloc(16);
c = malloc(32);
d = malloc(48);
e = malloc(32);
f = malloc(32);

// allocations that are not freed
// don't show up in the tcache!
x = malloc(64);
y = malloc(64);
z = malloc(64);

// later freed allocations show up
// earlier in the tcache list order
free(b);
free(a);
free(f);
free(e);
free(c);
free(d);
```



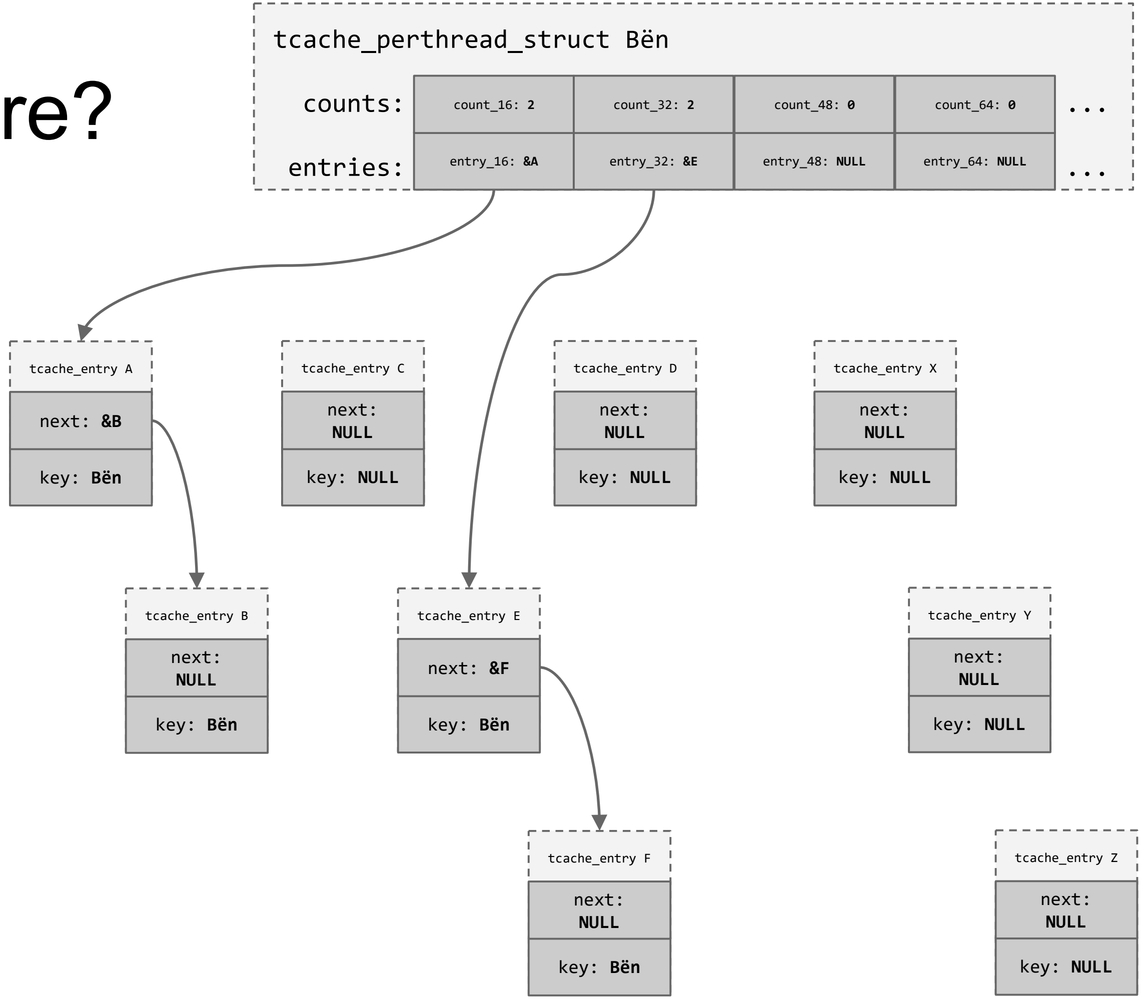
How did we get here?

```
a = malloc(16);
b = malloc(16);
c = malloc(32);
d = malloc(48);
e = malloc(32);
f = malloc(32);

// allocations that are not freed
// don't show up in the tcache!
x = malloc(64);
y = malloc(64);
z = malloc(64);

// later freed allocations show up
// earlier in the tcache list order
free(b);
free(a);
free(f);
free(e);
free(c);
free(d);
```



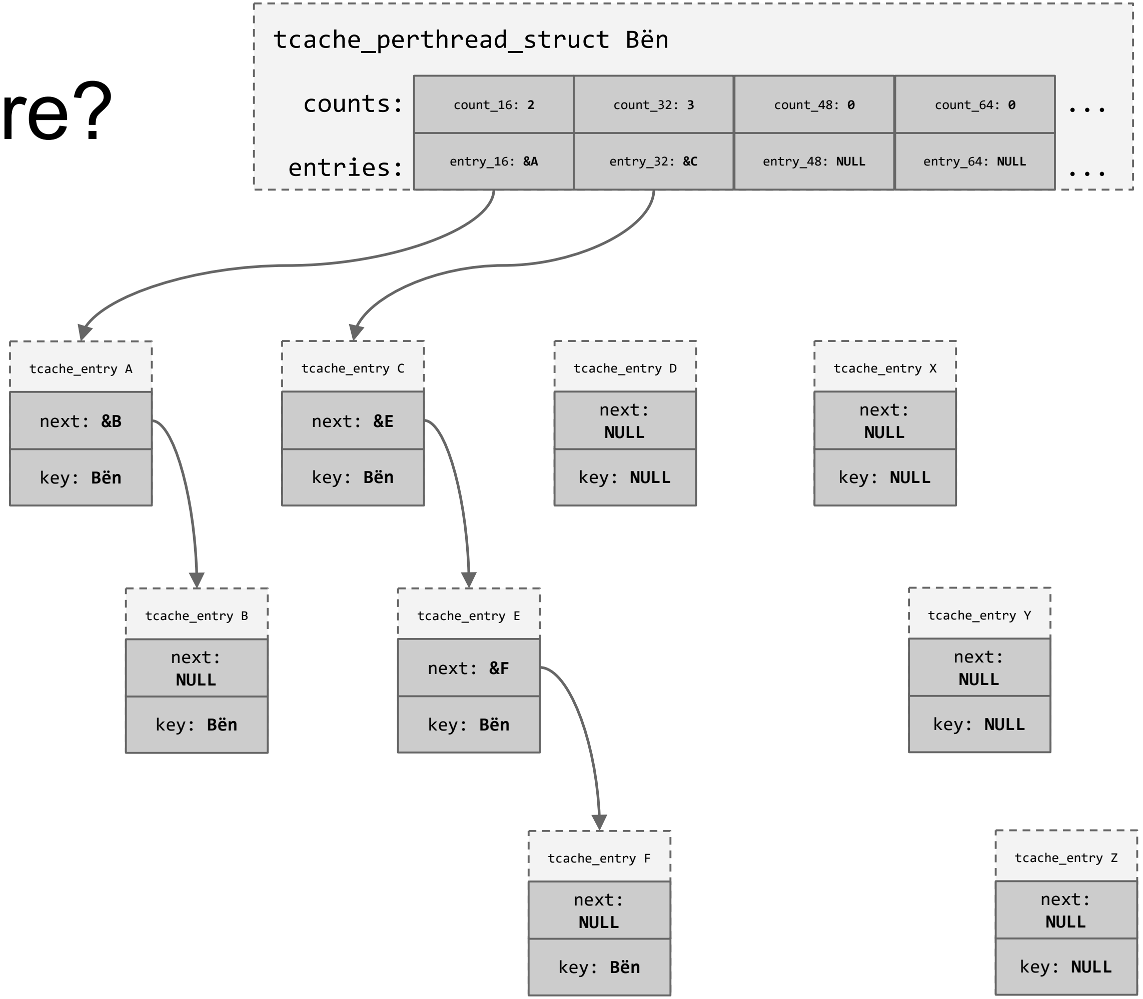
How did we get here?

```
a = malloc(16);
b = malloc(16);
c = malloc(32);
d = malloc(48);
e = malloc(32);
f = malloc(32);

// allocations that are not freed
// don't show up in the tcache!
x = malloc(64);
y = malloc(64);
z = malloc(64);

// later freed allocations show up
// earlier in the tcache list order
free(b);
free(a);
free(f);
free(e);
free(c);
free(d);
```



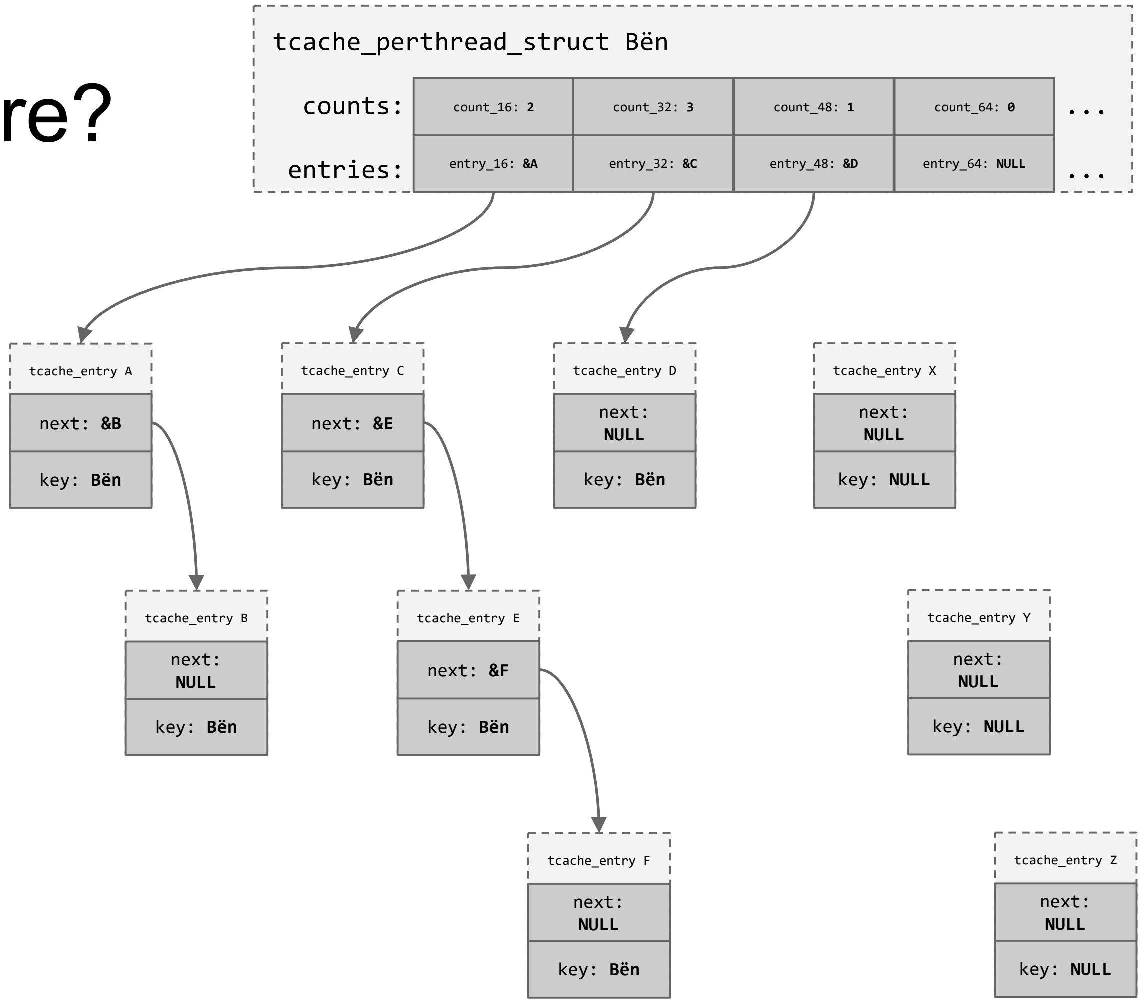
How did we get here?

```
a = malloc(16);
b = malloc(16);
c = malloc(32);
d = malloc(48);
e = malloc(32);
f = malloc(32);

// allocations that are not freed
// don't show up in the tcache!
x = malloc(64);
y = malloc(64);
z = malloc(64);

// later freed allocations show up
// earlier in the tcache list order
free(b);
free(a);
free(f);
free(e);
free(c);
free(d);
```



How did we get here?

```
a = malloc(16);
b = malloc(16);
c = malloc(32);
d = malloc(48);
e = malloc(32);
f = malloc(32);

// allocations that are not freed
// don't show up in the tcache!
x = malloc(64);
y = malloc(64);
z = malloc(64);

// later freed allocations show up
// earlier in the tcache list order
free(b);
free(a);
free(f);
free(e);
free(c);
free(d);
```



tcache freeing

Each `tcache_entry` is actually the exact allocation that was freed! On `free()`, the following happens:

Select the right "bin" based on the size:

```
idx = (freed_allocation_size - 1) / 16;
```

Check to make sure the entry hasn't already been freed (double-free):

```
((unsigned long*)freed_allocation)[1] == &our_tcache_perthread_struct;
```

Push the freed allocation to the front of the list!

```
((unsigned long*)freed_allocation)[0] = our_tcache_perthread_struct.entries[idx];
our_tcache_perthread_struct.entries[idx] = freed_allocation;
our_tcache_perthread_struct.count[idx]++;
```

Record the tcache_perthread_struct associated with the freed allocation (for checking against double-frees)

```
((unsigned long*)freed_allocation)[1] = &our_tcache_perthread_struct
```

tcache allocation

On allocation, the following happens:

Select the bin number based on the requested size:
```
idx = (requested_size - 1) / 16;
```

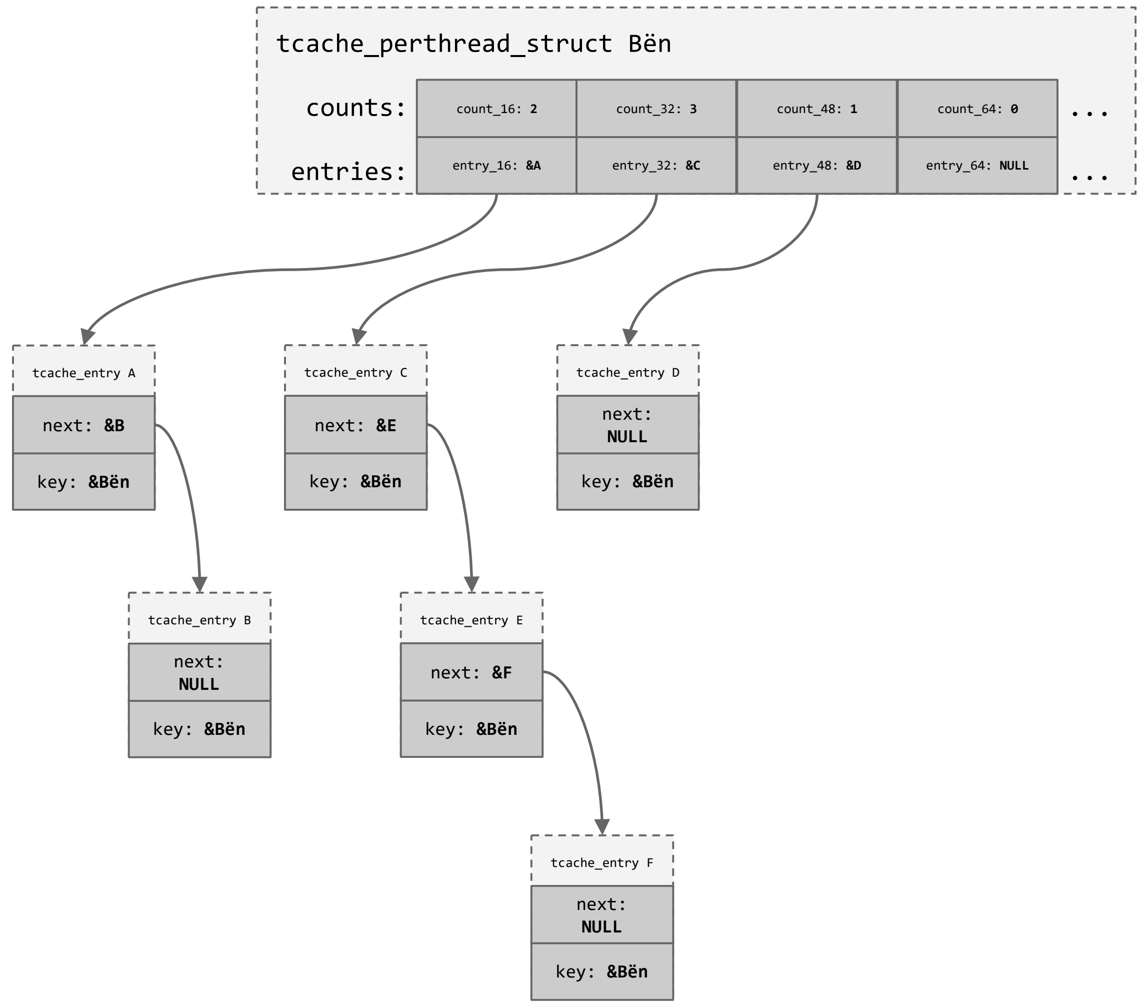
Check the appropriate cache for available entries:
```
if our_tcache_perthread_struct.count[idx] > 0;
```

Reuse the allocation in the front of the list if available:
```
unsigned long *to_return = our_tcache_perthread_struct.entries[idx];
tcache_perthread_struct.entries[idx] = to_return[0];
tcache_perthread_struct.count[idx]--;
return to_return;
```

Things that are **not** done:

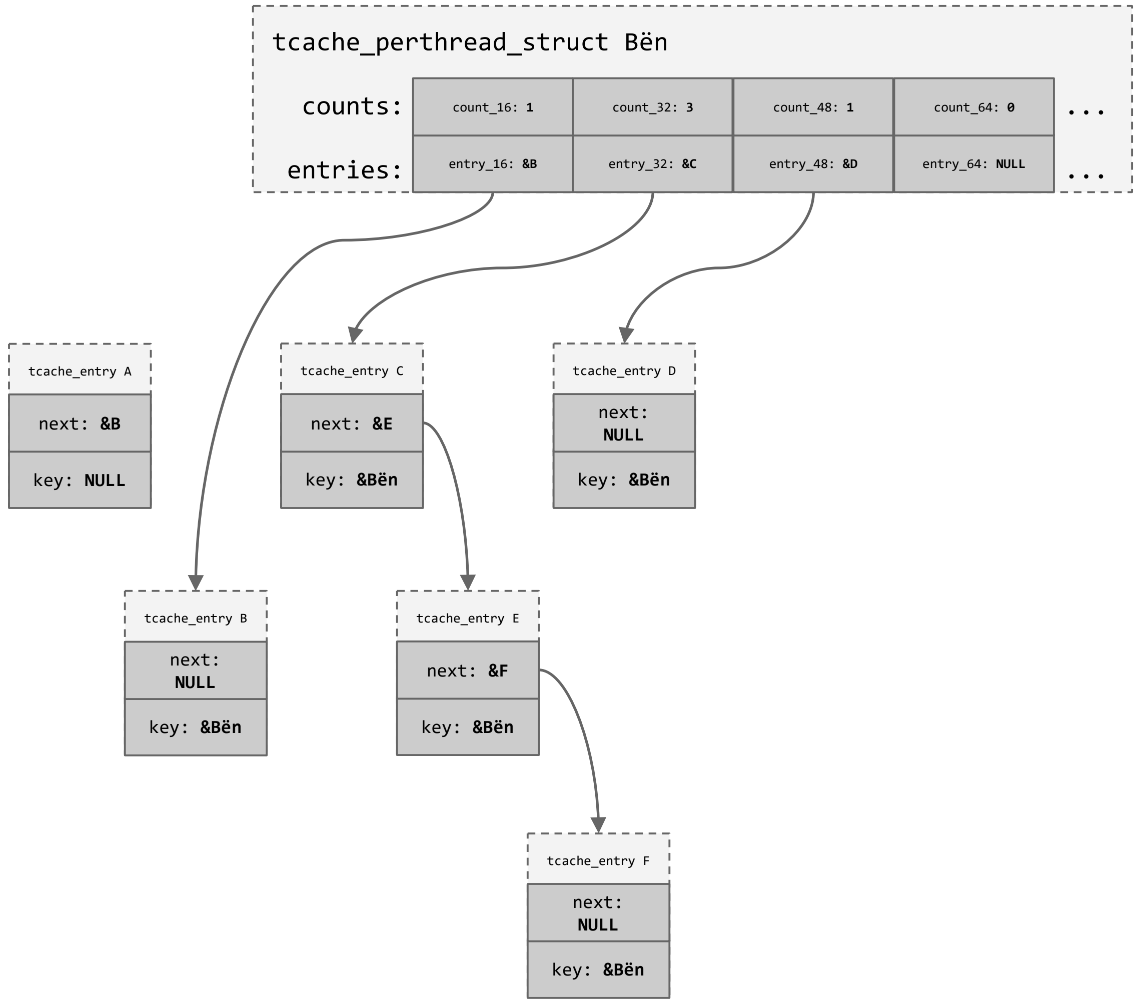
- clearing all sensitive pointers (only `key` is cleared for some reason).
- checking if the `next` (`return[0]`) address makes sense

Onward!


tcache_perthread_struct Bën
counts:
count_16: 2
count_32: 3
count_48: 1
count_64: 0
...
entries:
entry_16: &A
entry_32: &C
entry_48: &D
entry_64: NULL
...
tcache_entry A
next: &B
key: &Bën
tcache_entry C
next: &E
key: &Bën
tcache_entry D
next: NULL
key: &Bën
tcache_entry B
next: NULL
key: &Bën
tcache_entry E
next: &F
key: &Bën
tcache_entry F
next: NULL
key: &Bën

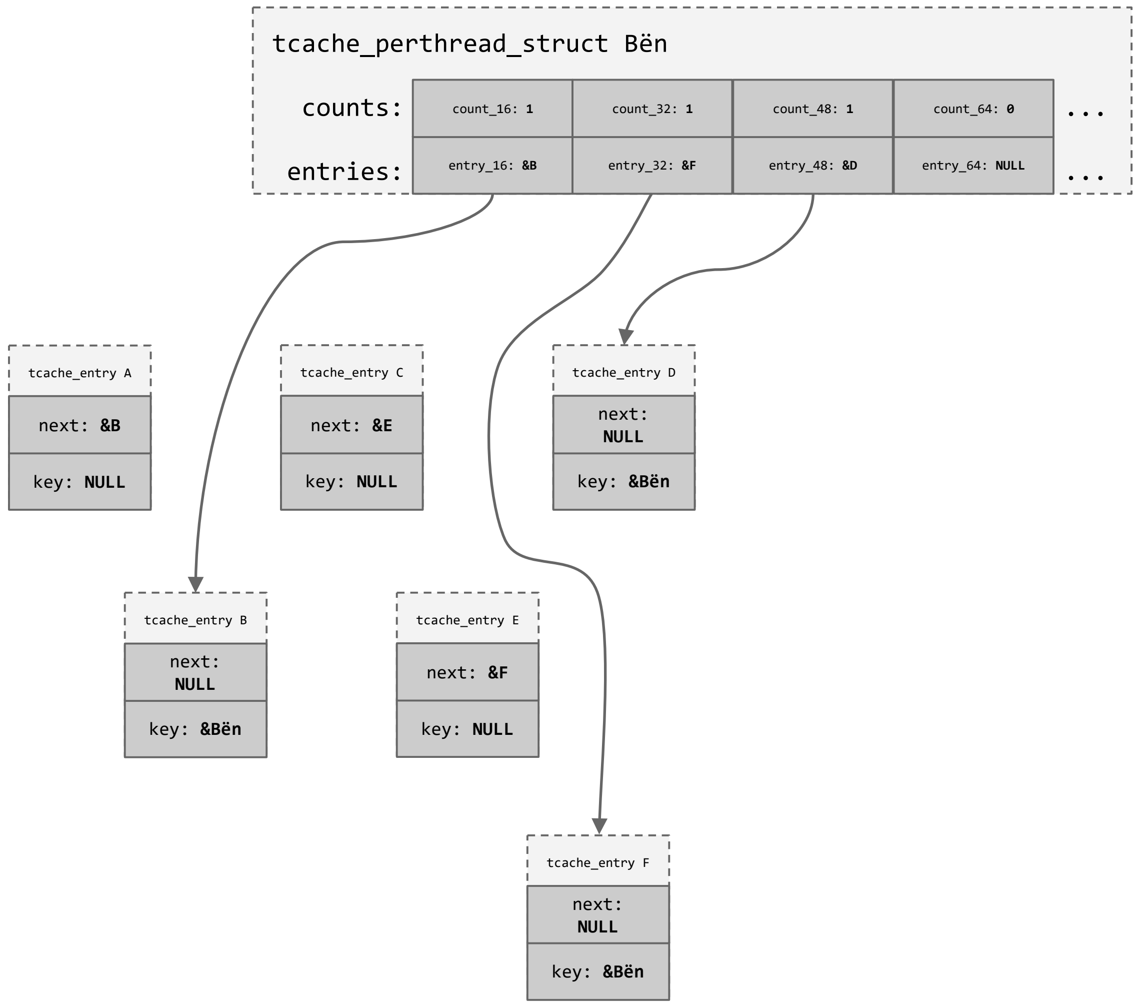
Onward!

```
malloc(16) == a
```



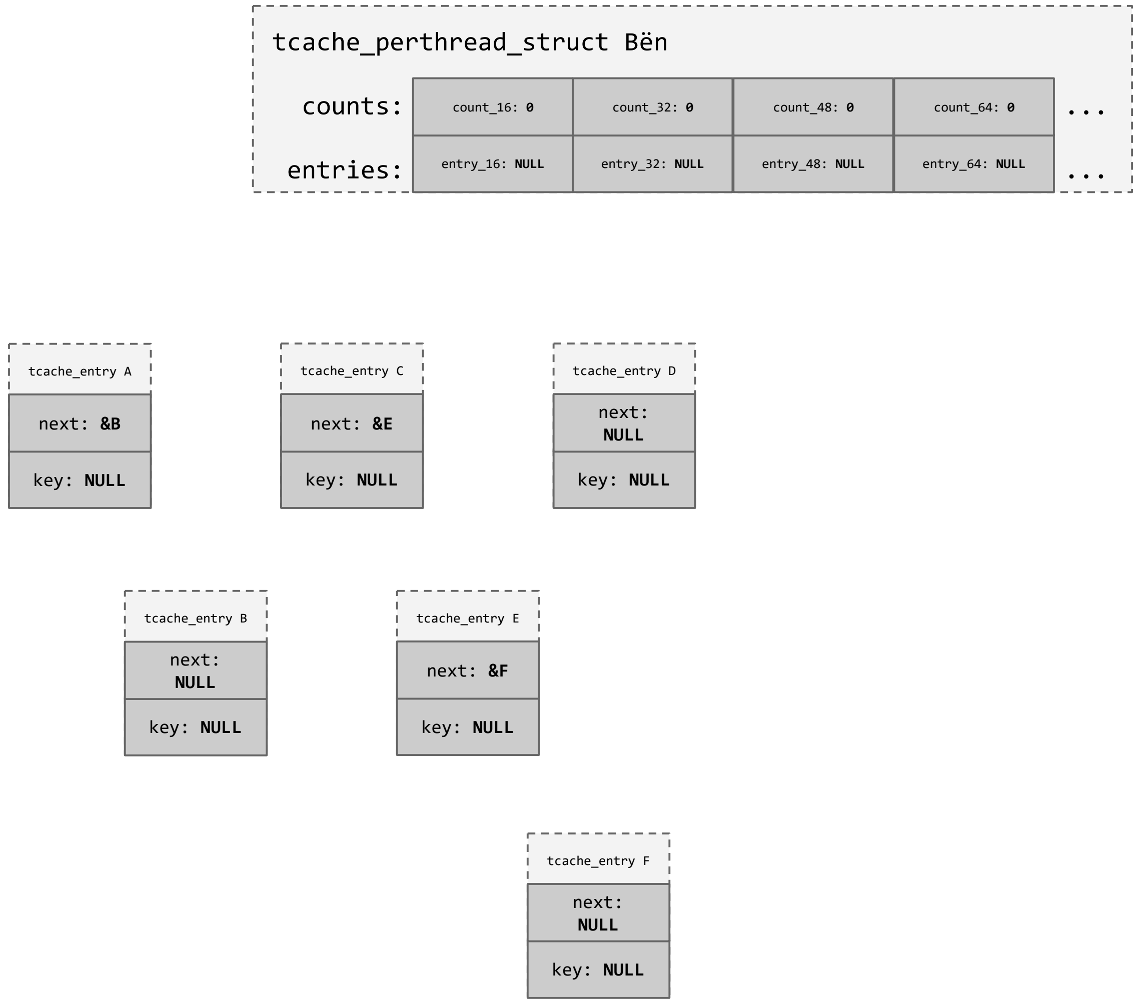
Onward!

```
malloc(16) == a
malloc(32) == c
malloc(32) == e
```



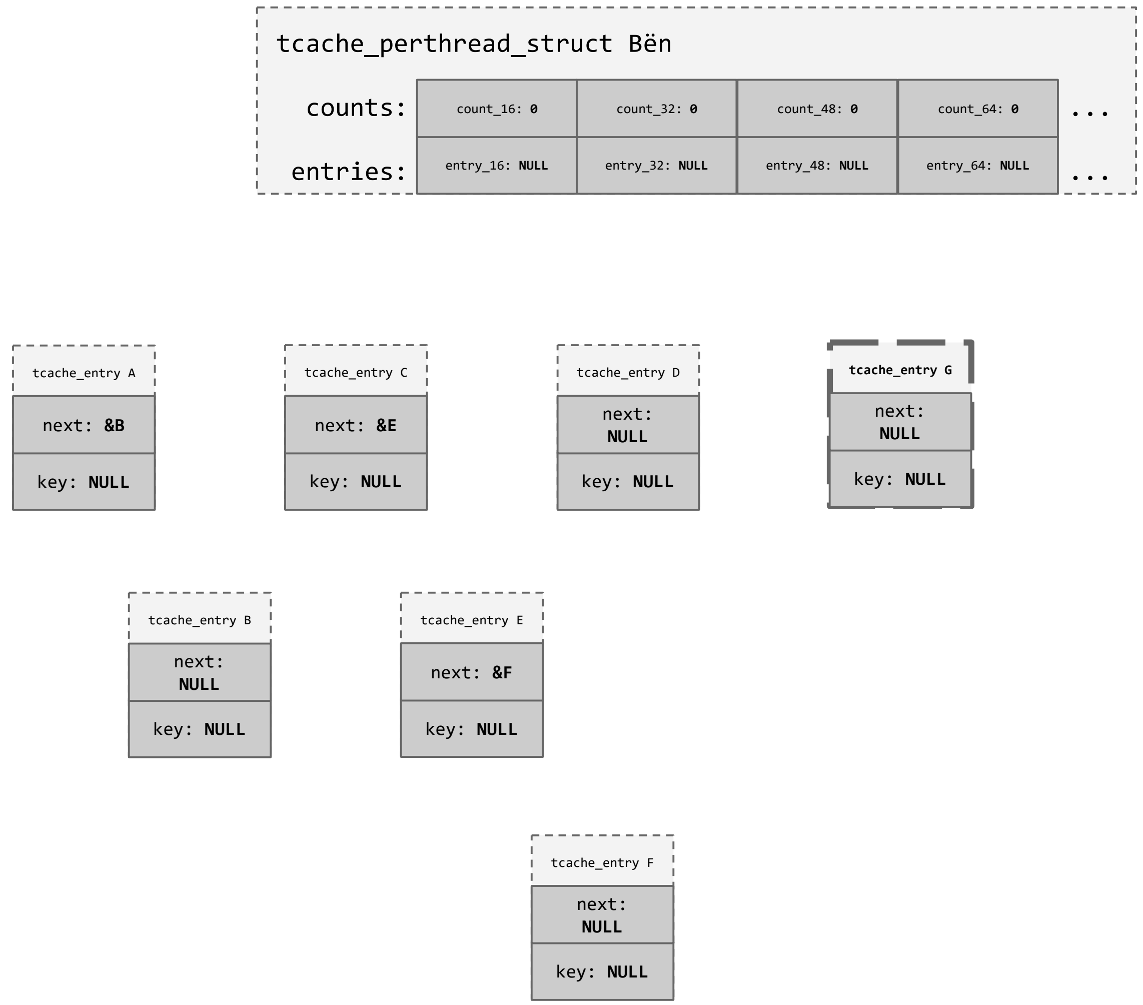
Onward!

```
malloc(16) == a
malloc(32) == c
malloc(32) == e
malloc(48) == d
malloc(16) == b
malloc(32) == f
```



Onward!

```
malloc(16) == a
malloc(32) == c
malloc(32) == e
malloc(48) == d
malloc(16) == b
malloc(32) == f
malloc(64) == g
```



Dangers of the heap-double free

Problem: Pointers to an allocation remain valid after `free()`ing the allocation, and are sometimes `free()`d again!

Solution: New versions of glibc/ptmalloc introduced the `key` check.

What if we overwrite `key` after free()ing our allocation...

Dangers of the heap - tcache poisoning

What happens if we corrupt `tcache_entry->next`?

tcache - summary

tcache is:
... a **caching** layer for "small" allocations (<1032 bytes on amd64)
... makes a **singly-linked-list** using the first word of the free chunk
... **very few** security checks

It gets even more insane...