System Security - Attack and Defense for Binaries

CS 4390/5390, Spring *2026*

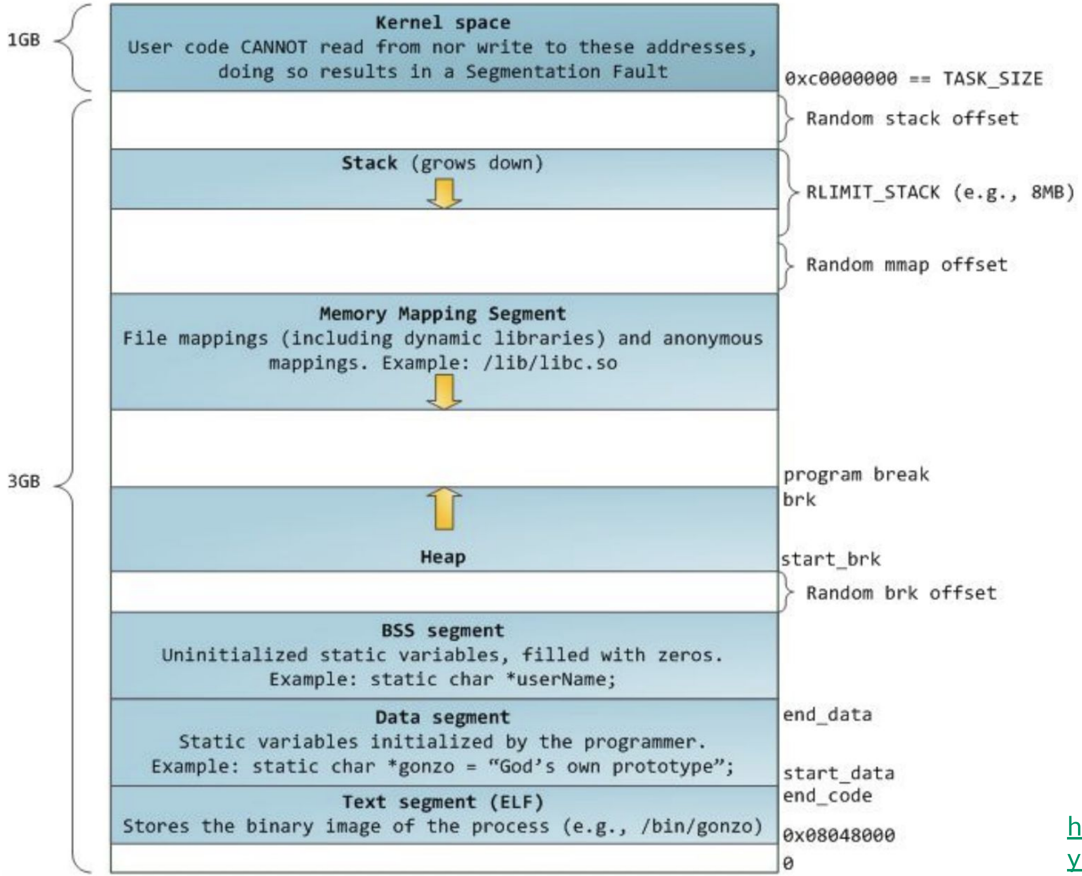
Instructor: MD Armanuzzaman (***Arman***)

Heap exploitation

Heap exploitation

1. What is heap and dynamic memory allocator?
2. Malloc and free interfaces
3. Ptmalloc and tcache
    a. Malloc_chunk
4. Heap exploitation
    a. Overflow
    b. Use-after-free

Memory Map of Linux Process (32 bit system)



https://manybutfinite.com/post/anatomy-of-a-program-in-memory/

The Heap

The heap is *pool* of memory used for **dynamic** allocations at runtime.

Heap memory is different from stack memory in that it is ***persistent between functions.***

- ***malloc()*** grabs memory on the heap; keyword ***new*** in C++
- ***free()*** releases memory on the heap; keyword ***delete*** in C++

Both are standard C library interfaces. Neither of them directly mapps to a system call.

Why not mmap()?

Mmap()

- Mmap() is a system call. So kernel is involved, which means slow.
- Can only allocate multiples of pages (4KB).

Hence, the idea of ***dynamic memory allocator***

Dynamic memory allocators

Doug Lea malloc or dlmalloc: Release to public in 1987. Native version of malloc in some old distributions of Linux (*http://gee.cs.oswego.edu/dl/html/malloc.html*)

ptmalloc: ptmalloc is based on dlmalloc and was extended for use with multiple threads. On Linux systems, ptmalloc has been put to work for years as part of the GNU C library.

tcmalloc: Google's customized implementation of C's malloc() and C++'s operator new (*https://github.com/google/tcmalloc*)

Dynamic memory allocators

jemalloc: jemalloc is a general purpose malloc(3) implementation that emphasizes fragmentation avoidance and scalable concurrency support. Used in FreeBSD, firefox, Android.

Hoard memory allocator: UMass Amherst CS Professor *Emery Berger*- (*creator/maintainer of* *csrankings.org*)

Kmalloc: Linux kernel memory allocator

Kalloc: iOS kernel memory allocator

Segment Heap, NT Heap: Windows implementations.

malloc() and free()

stdlib.h provides with standard library functions to access, modify and manage dynamic memory.

```
void* malloc(size_t size);
```

Allocates size bytes of uninitialized storage. If allocation succeeds, returns a pointer that is suitably aligned for any object type with fundamental alignment.

```
void free(void* ptr);
```

Deallocates the space previously allocated by malloc(), etc.

calloc() and realloc()

```
void *calloc(size_t nitems, size_t size)
```

The difference in malloc and calloc is that malloc does not set the memory to zero whereas calloc sets allocated memory to zero.

```
void *realloc(void *ptr, size_t size)
```

Resize the memory block pointed to by ptr that was previously allocated with a call to malloc or calloc.

How to use malloc() and free()

```
int main()
{
	char * buffer = NULL;

	/* allocate a 0x100 byte buffer */
	buffer = malloc(0x100);

	/* read input and print it */
	fgets(stdin, buffer, 0x100);
	printf(“Hello %s!\n”, buffer);

	/* destroy our dynamically allocated buffer */
	free(buffer);
	return 0;
}
```

Heap vs. Stack

Heap

- Dynamic memory allocations at runtime
- Objects, big buffers, structs, persistence, larger things

Slower, Manual

- Done by the programmer
- malloc/calloc/recalloc/free
- new/delete

Stack

- Fixed memory allocations known at compile time
- Local variables, return addresses, function args

Fast, Automatic; Done by the compiler

- Abstracts away any concept of allocating/de-allocating

Which implementation on our server?

ldd --version

GLIBC 2.31, Ptmalloc2

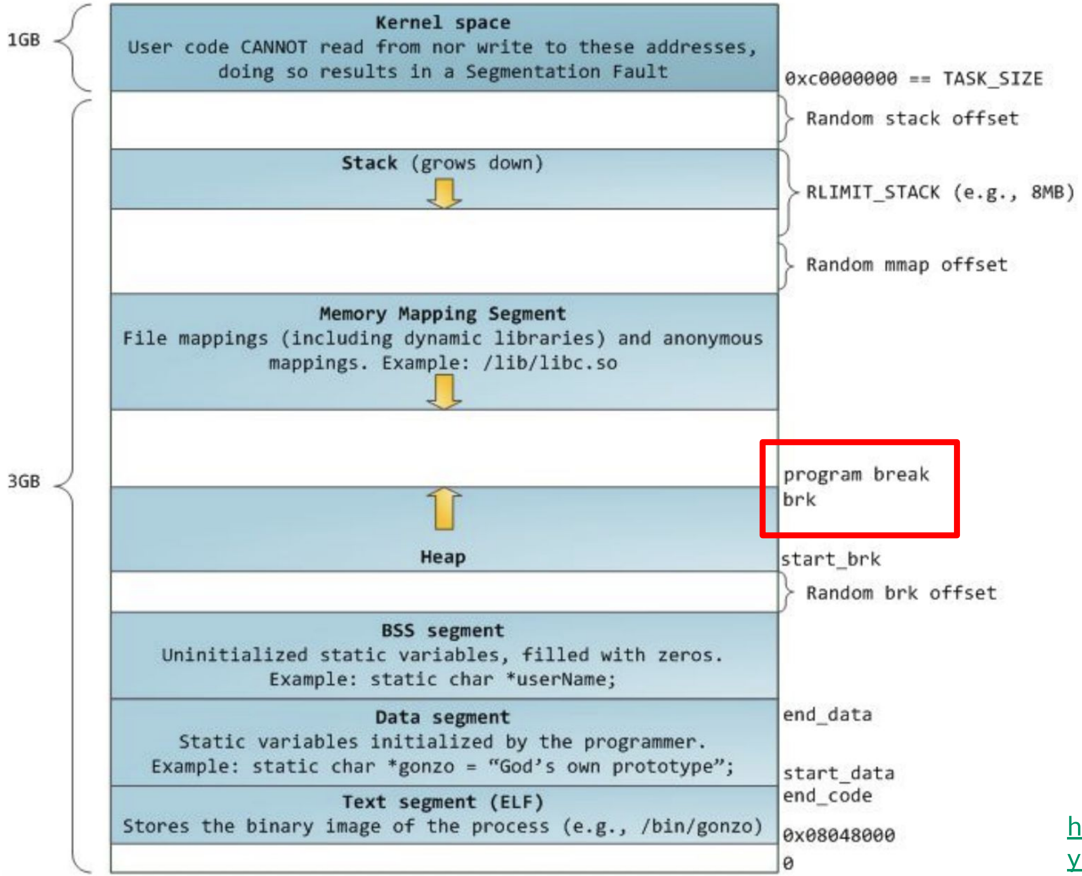
https://elixir.bootlin.com/glibc/glibc-2.31/source/malloc/malloc.c

```
ctf@heapexploitation_heapchunks_32:/$ ldd --version
ldd (Ubuntu GLIBC 2.31-0ubuntu9.2) 2.31
Copyright (C) 2020 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
Written by Roland McGrath and Ulrich Drepper.
```

Disclaimer: Ptmalloc is very complex, and its implementation is constantly changing.

This is an approximation to glibc 2.31

Memory Map of Linux Process (32 bit system)



https://manybutfinite.com/post/anatomy-of-a-program-in-memory/

How does ptmalloc get memory

- Use the mmap() system call for large memory request
- Use brk() and sbrk() system calls
  - sbrk(NULL) returns the current program break
  - sbrk(200) expands program break by 200 bytes
  - brk(addr) expands the program break to address

```
DESCRIPTION
       brk()  and  sbrk()  change  the location of the program break, which defines the end
of the process's data segment (i.e., the program break is the first location after the end o
f the uninitialized
       data segment).  Increasing the program break has the effect of allocating memory to t
he process; decreasing the break deallocates memory.

       brk() sets the end of the data segment to the value specified by addr, when that valu
e is reasonable, the system has enough memory, and the process does not exceed its maximum d
ata size (see setr-
       limit(2)).

       sbrk() increments the program's data space by increment bytes.  Calling sbrk() with a
n increment of 0 can be used to find the current location of the program break.
```

Heap chunk: malloc_chunk (ptmalloc2 in glibc2.31; *no tcache*)

Two states: in-use and freed

For both in-use and freed

```
struct malloc_chunk {

  INTERNAL_SIZE_T      mchunk_prev_size;  /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T      mchunk_size;       /* Size in bytes, including overhead. */

  struct malloc_chunk* fd;         /* double links -- used only if free. */
  struct malloc_chunk* bk;

  /* Only used for large blocks: pointer to next larger size.  */
  struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
  struct malloc_chunk* bk_nextsize;
};
```

Only for freed

The freed chunks are double-linked

INTERNAL_SIZE_T is the same as size_t. 8 bytes in 64 bit;
4 bytes in 32 bits machine.
Pointer is 8/4 bytes on a 64/32 bit machine, respectively.

Alignment is defined as 2 * (sizeof(size_t))

https://elixir.bootlin.com/glibc/glibc-2.31/source/malloc/malloc.c

Heap chunk: malloc_chunk (ptmalloc2 in glibc2.31; *no tcache*)

```
An allocated chunk looks like this:


    chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of previous chunk, if unallocated (P clear)  |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of chunk, in bytes                     |A|M|P|
buf   mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             User data starts here...                          .
            .                                                               .
            .             (malloc_usable_size() bytes)                      .
            .                                                               |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             (size of chunk, but used for application data)    |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of next chunk, in bytes                |A|0|1|
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Where "chunk" is the front of the chunk for the purpose of most of
the malloc code, but "mem" is the pointer that is returned to the
user.  "Nextchunk" is the beginning of the next contiguous chunk.

Chunks always begin on even word boundaries, so the mem portion
(which is returned to the user) is also on an even word boundary, and
thus at least double-word aligned.
```

*glibc 2.3 allows for many heaps arranged into several **arenas**—one arena for each thread*

Chunk Size: Size of entire chunk including overhead

Flags: Because of byte alignment, the lower 3 bits of the chunk size field would always be zero. Instead they are used for flag bits.

0x01 **P**REV_INUSE – set when previous chunk is in use

0x02 IS_**M**MAPPED – set if chunk was obtained with mmap()

0x04 NON_MAIN_**A**RENA – set if chunk belongs to a thread arena

Heap chunk: malloc_chunk (ptmalloc2 in glibc2.31; *no tcache*)

```
    Free chunks are stored in circular doubly-linked lists, and look like this:

    chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of previous chunk, if unallocated (P clear)  |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    `head:' |             Size of chunk, in bytes                     |A|0|P|
      mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Forward pointer to next chunk in list             |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Back pointer to previous chunk in list            |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Unused space (may be 0 bytes long)                .
            .                                                               .
            .                                                               |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    `foot:' |             Size of chunk, in bytes                           |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of next chunk, in bytes                |A|0|0|
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Bins (*no tcache*)

A bin is a list (doubly or singly linked list) of free (non-allocated) chunks.

Bins are differentiated based on the size of chunks they contain:

- Fast bin. Introduced long before tcache (part of original ptmalloc design). Used for very small chunks (e.g., ≤ 64 bytes). Each fast bin is a single-linked list (no coalescing on free). Shared between threads (i.e., global per arena). Chunks are added here when tcache is full or not enabled.
- Small bin. Manage small freed chunks not handled by fast bins or tcache. Double-linked circular lists.
- Large bin. Manage freed chunks larger than 512 bytes.
- Unsorted bin. Temporary holding place for freed chunks before being placed into small or large bins.
- Top chunk. unallocated space at the top of the heap. No existing bin has a suitable chunk, and heap grows via sbrk.

The “modern” Tcache Design

"Thread Cache" in ptmalloc, to speed up repeated (small) allocations in a single thread. Size range is configurable.

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

```
/* There is one of these for each thread, which contains the
   per-thread cache (hence "tcache_perthread_struct").  Keeping
   overall size low is mildly important.  Note that COUNTS and ENTRIES
   are redundant (we could have just counted the linked list each
   time), this is for performance reasons.  */
typedef struct tcache_perthread_struct
{
  uint16_t counts[TCACHE_MAX_BINS];
  tcache_entry *entries[TCACHE_MAX_BINS];
} tcache_perthread_struct;
```

https://elixir.bootlin.com/glibc/glibc-2.31.9000/source/malloc/malloc.c#L2906

ptmalloc2 in glibc2.31; tcache design

1. Every bin is a singly-linked list of chunks of that specific size.
2. Each thread has its own **tcache_perthread_struct**, which contains an array of these bins.

Heap chunk: malloc_chunk (ptmalloc2 in glibc2.31; tcache)

Two states: in-use and freed

fastbin/smallbin

Not used in tcache. Can be used by the previous chunk

Both in-use and freed

```
struct malloc_chunk {

  INTERNAL_SIZE_T      mchunk_prev_size;  /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T      mchunk_size;       /* Size in bytes, including overhead. */

  struct malloc_chunk* fd;         /* double links -- used only if free. */
  struct malloc_chunk* bk;

  /* Only used for large blocks: pointer to next larger size.  */
  struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
  struct malloc_chunk* bk_nextsize;
};
```

https://elixir.bootlin.com/glibc/glibc-2.31.9000/source/malloc/malloc.c#L2890

Heap chunk: malloc_chunk (ptmalloc2 in glibc2.31; tcache)

Two states: in-use and freed

fastbin/smallbin

```
struct malloc_chunk {

  INTERNAL_SIZE_T      mchunk_prev_size;  /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T      mchunk_size;       /* Size in bytes, including overhead. */

  tcache_entry entry;
};
```

Not used in tcache. Can be used by the previous chunk

Both in-use and freed

https://elixir.bootlin.com/glibc/glibc-2.31.9000/source/malloc/malloc.c#L2890

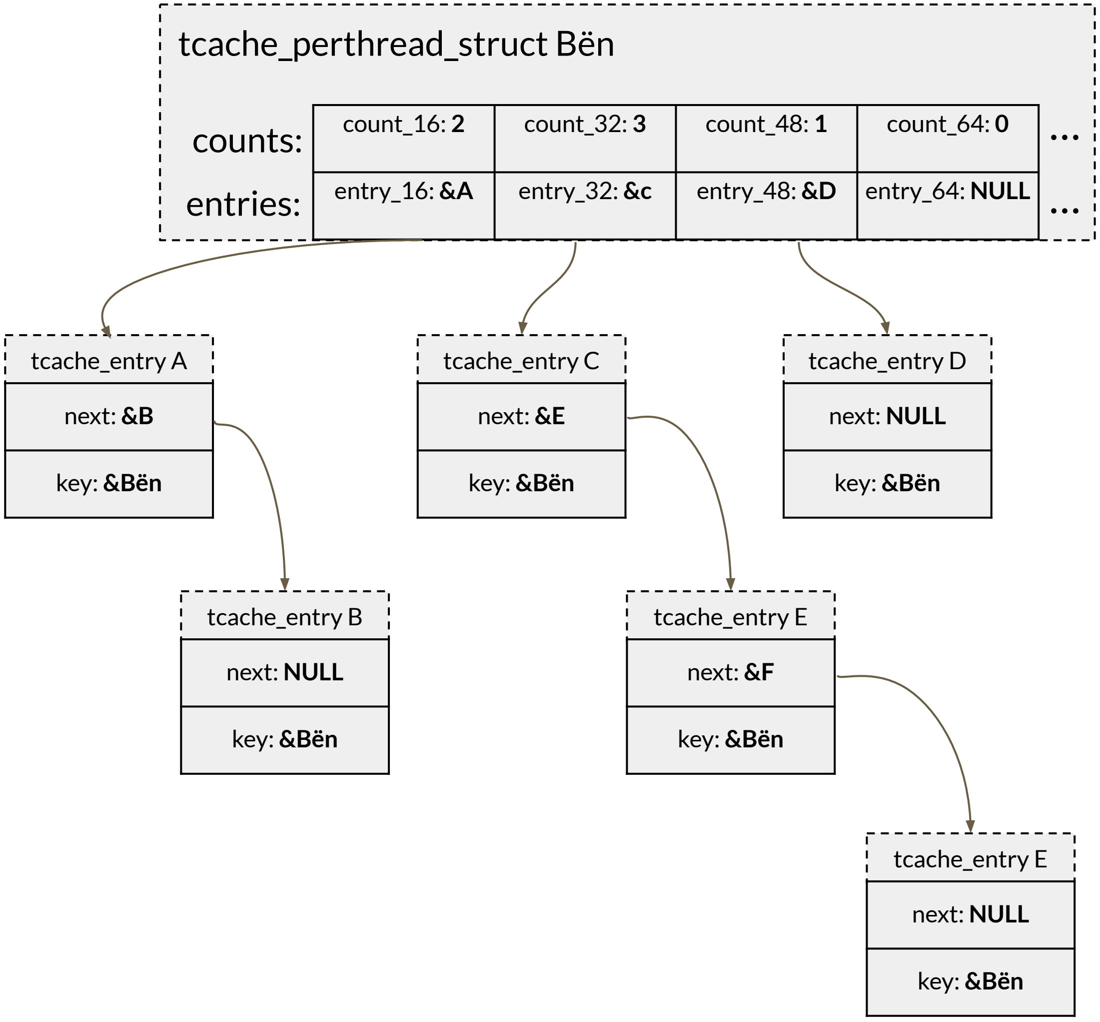
ptmalloc2 in glibc2.31: tcache design

fastbin/smallbin

```
#if USE_TCACHE

/* We overlay this structure on the user-data portion of a chunk when
   the chunk is stored in the per-thread cache.  */
typedef struct tcache_entry
{
  struct tcache_entry *next;
  /* This field exists to detect double frees.  */
  struct tcache_perthread_struct *key;
} tcache_entry;
```

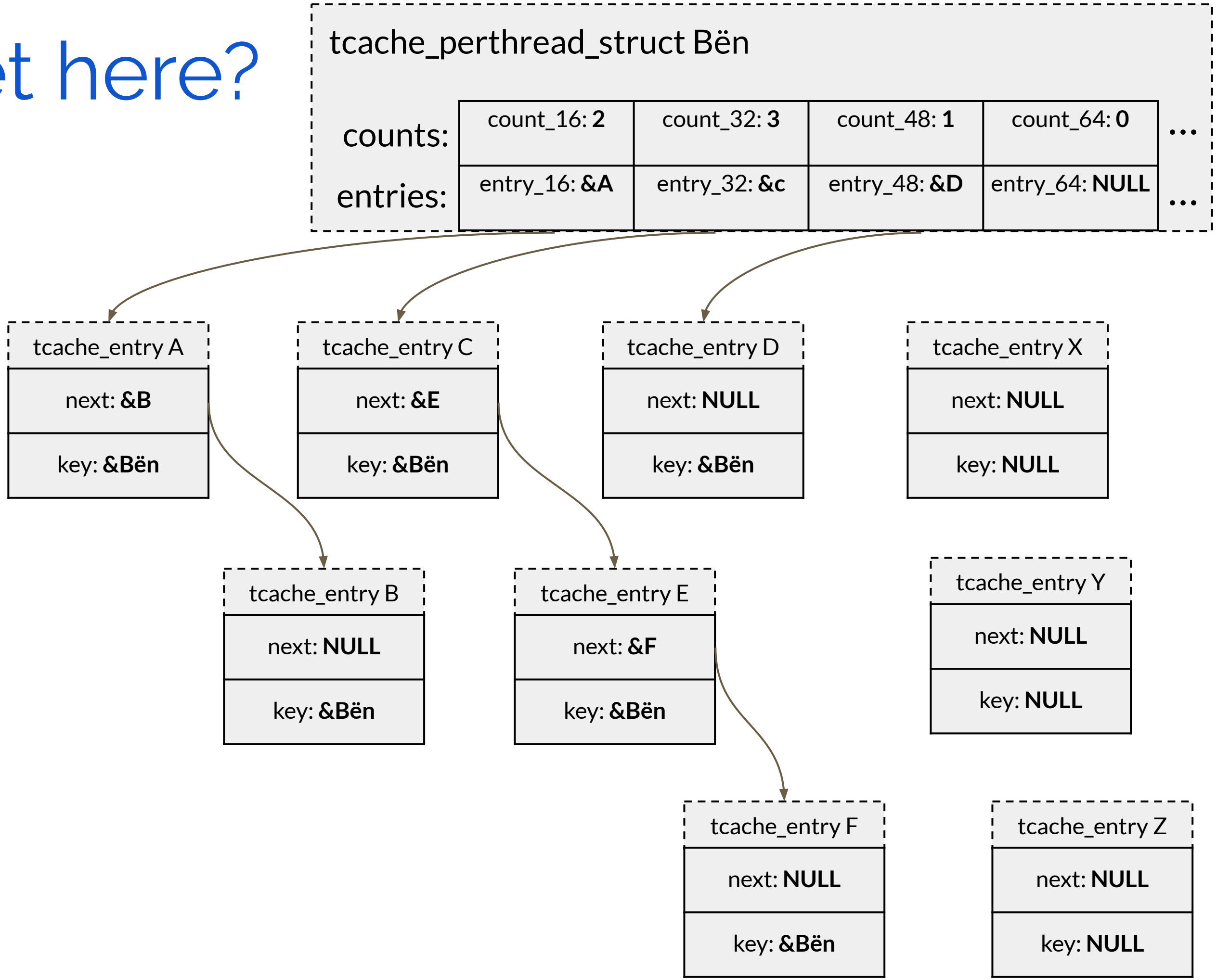
Singly-linked list

```
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;
```



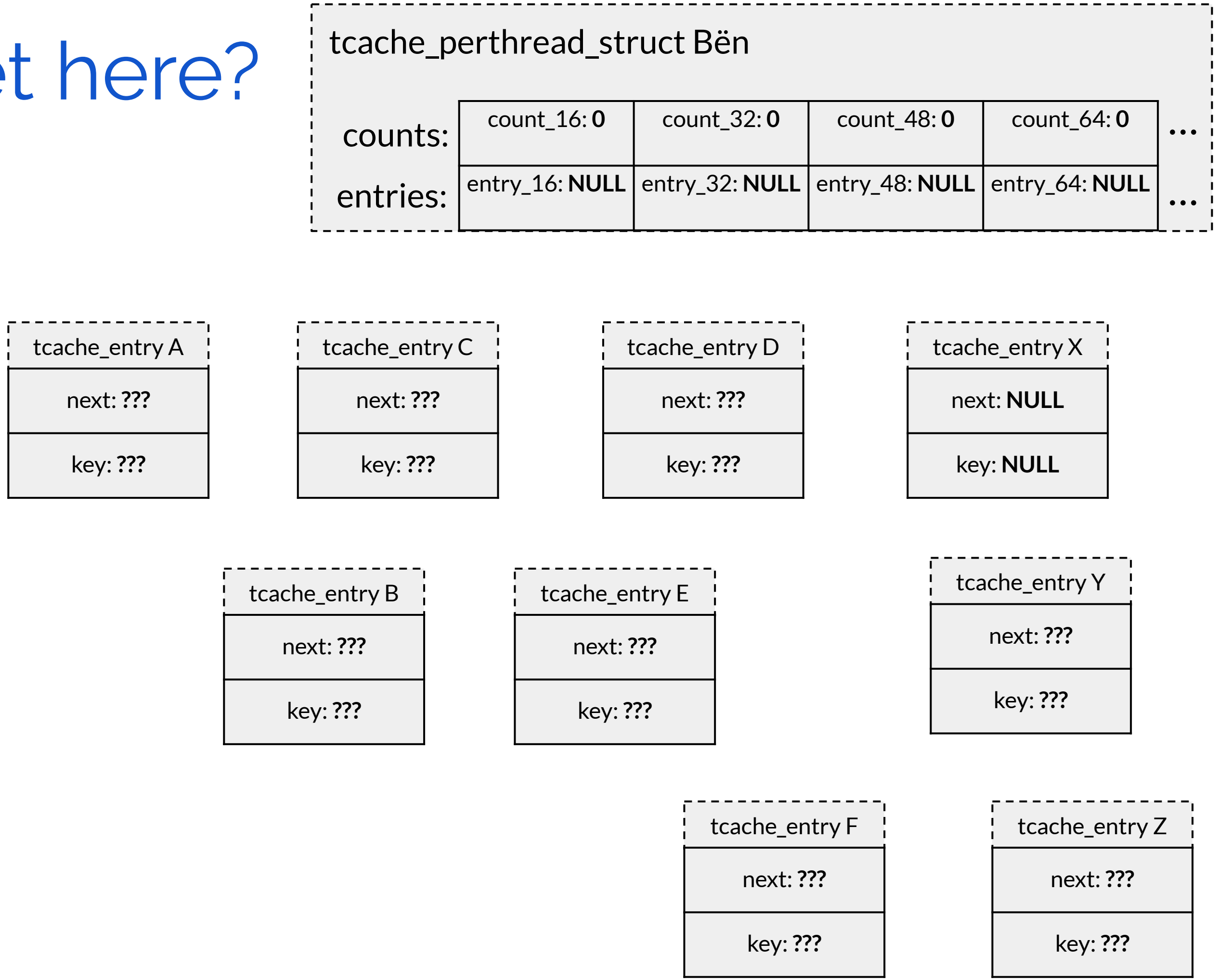
How did we get here?

```
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);
```



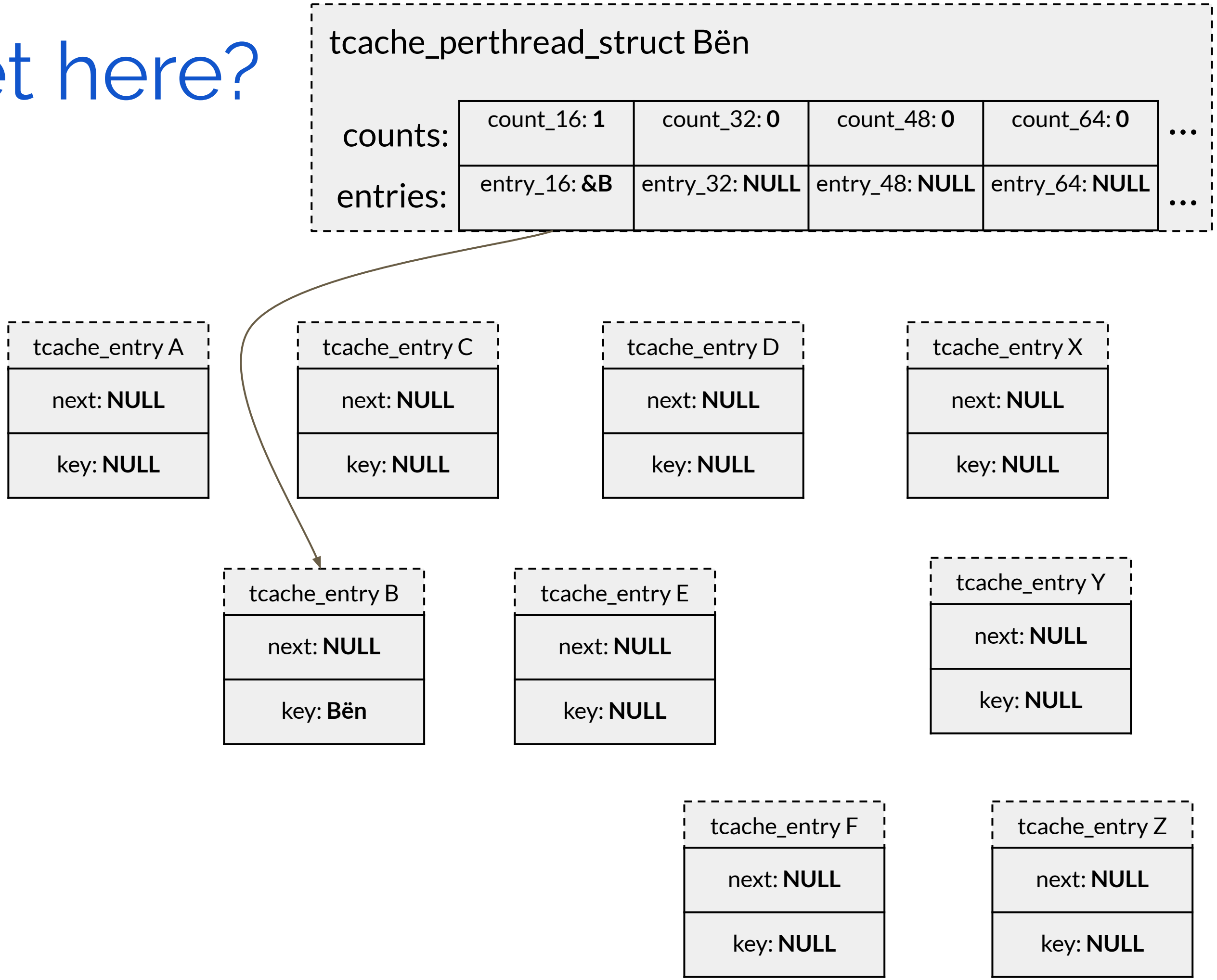
How did we get here?

```
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);
```



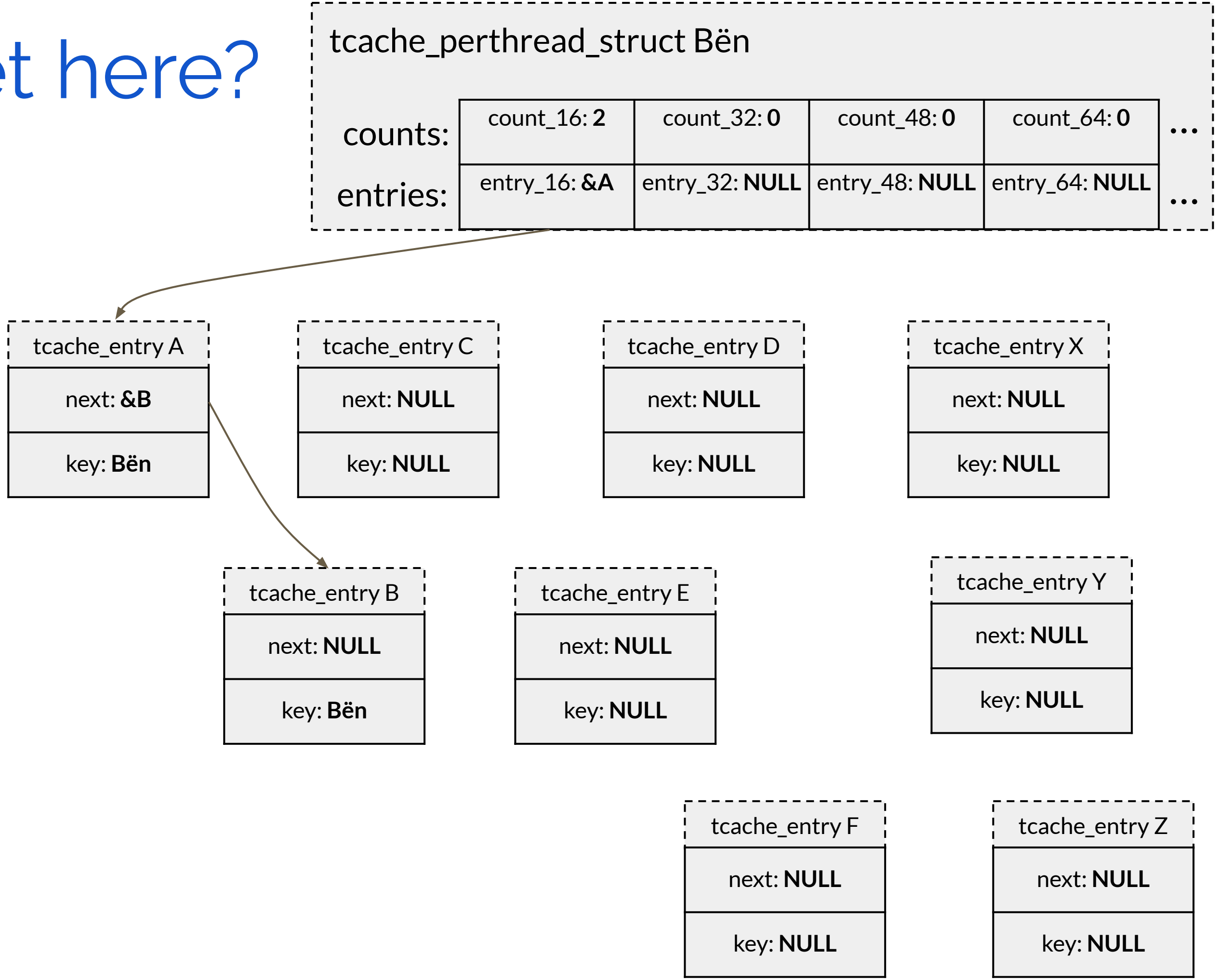
How did we get here?

```
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);
```



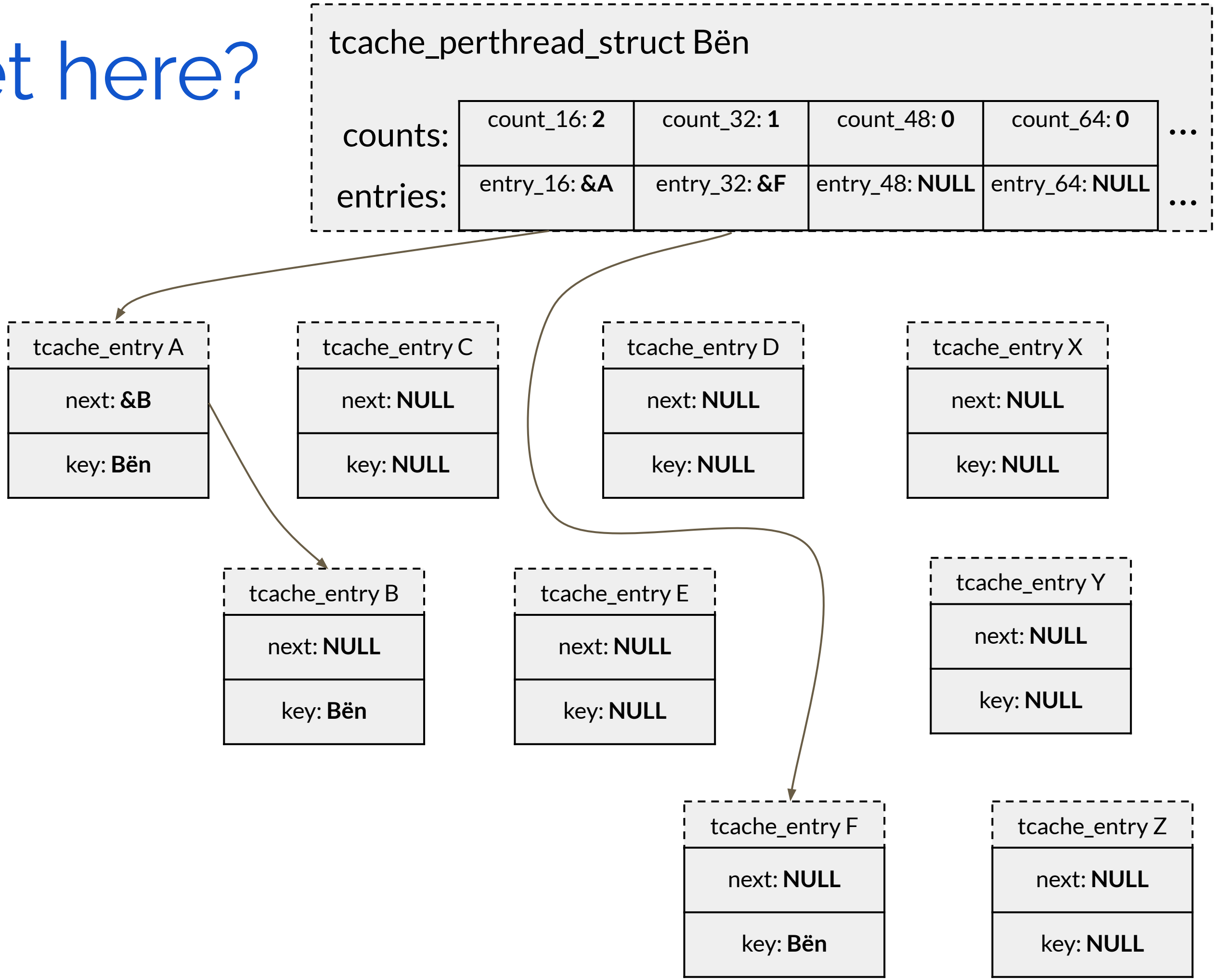
How did we get here?

```
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);
```



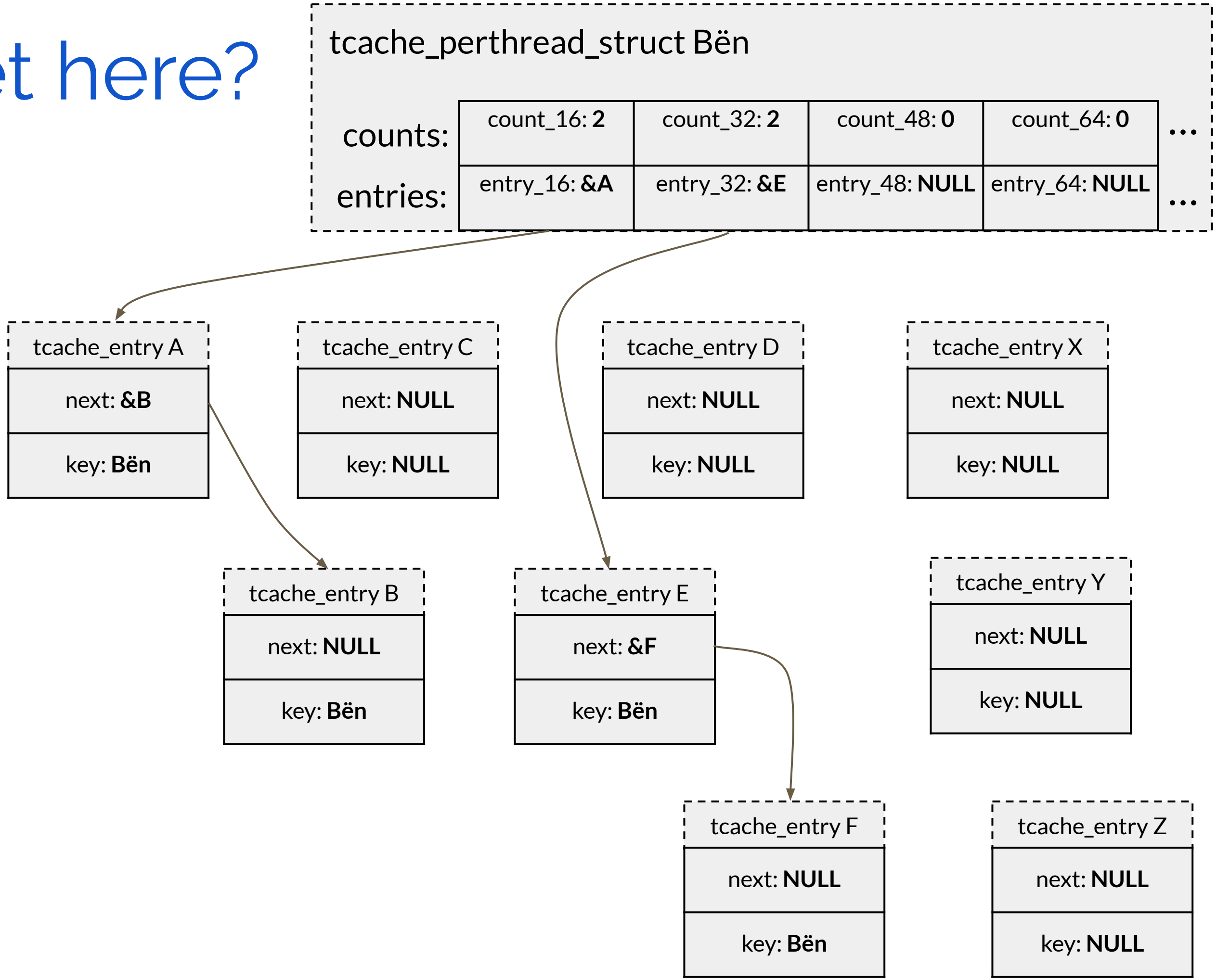
How did we get here?

```
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);
```



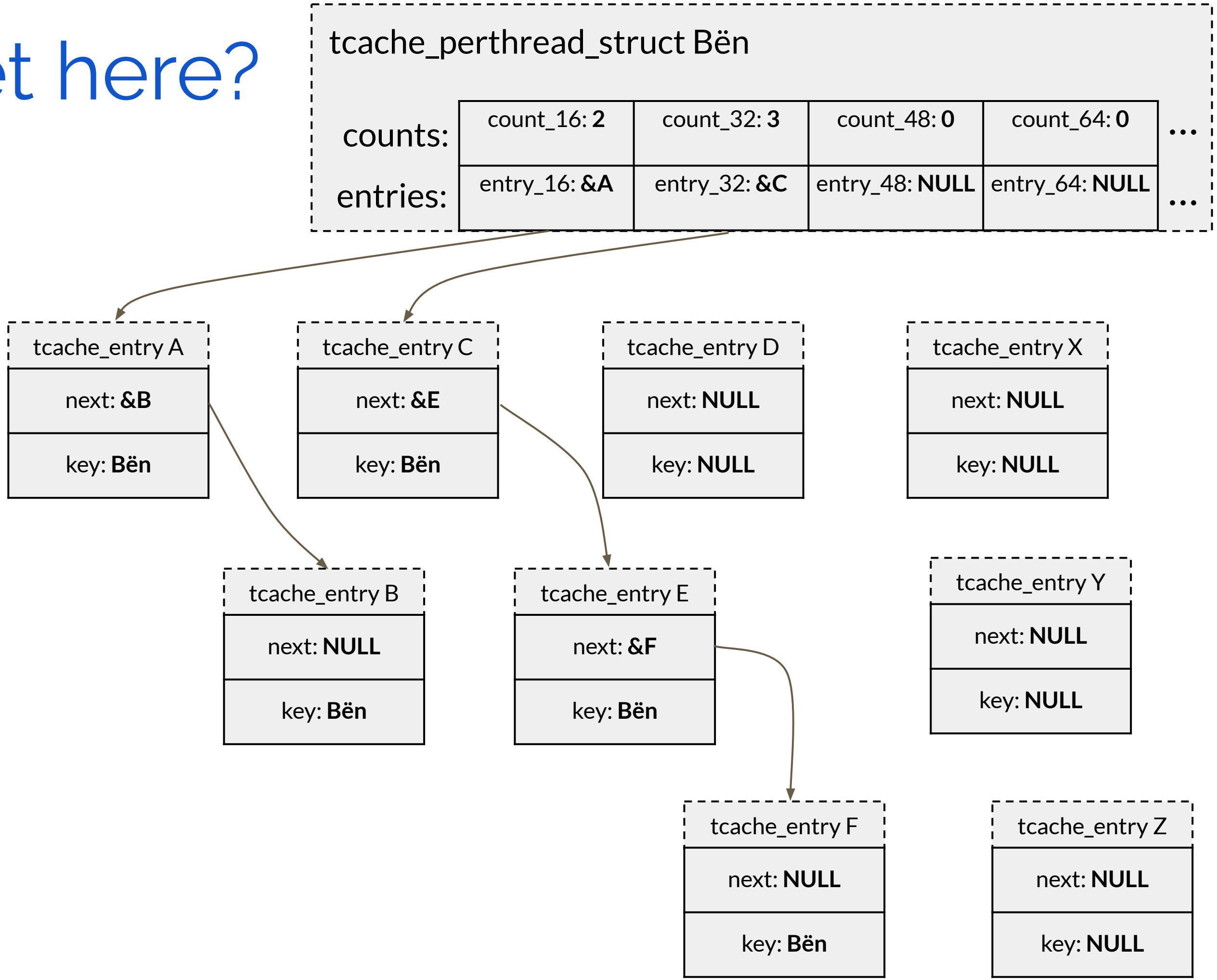
How did we get here?

```
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);
```



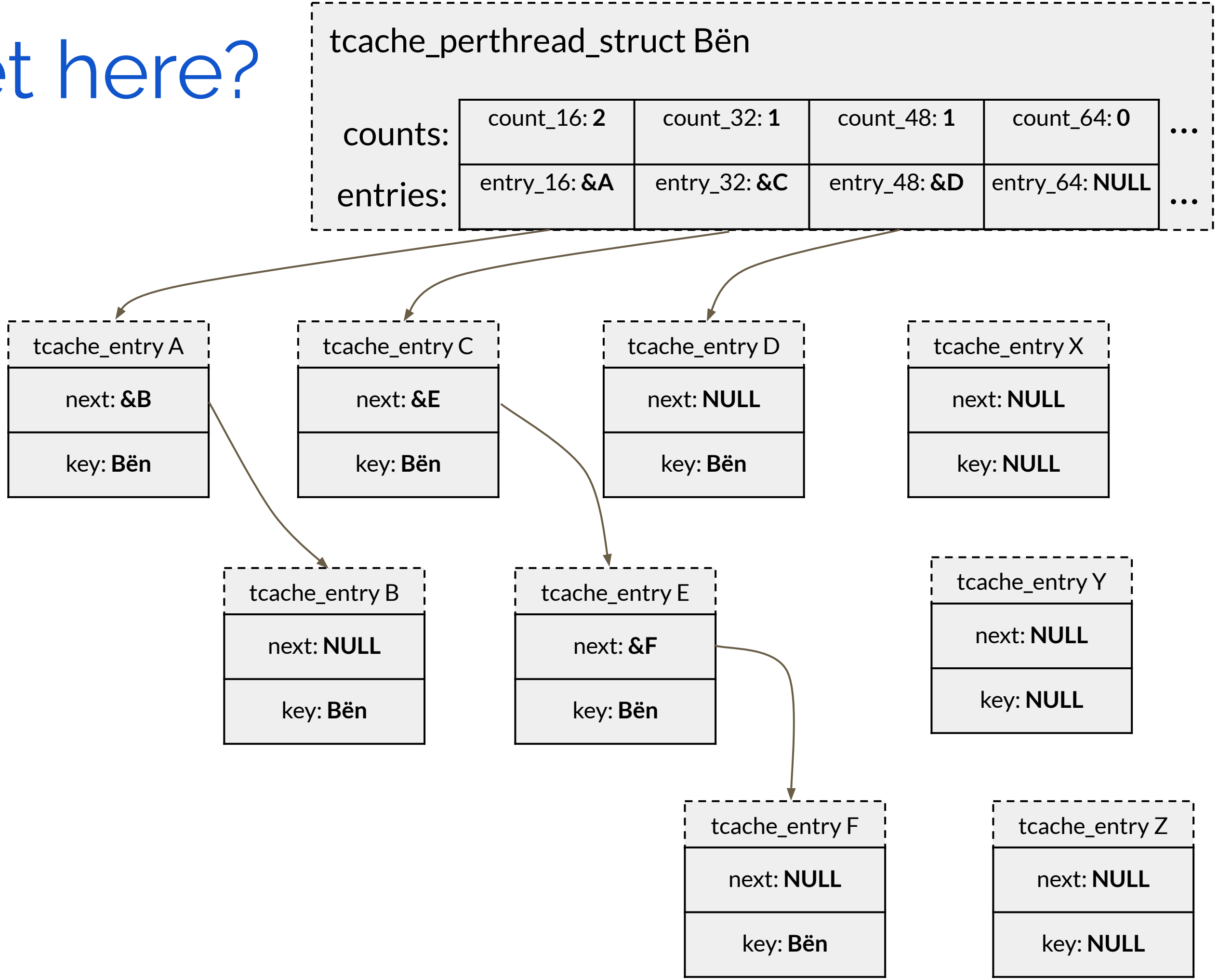
How did we get here?

```
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;
```

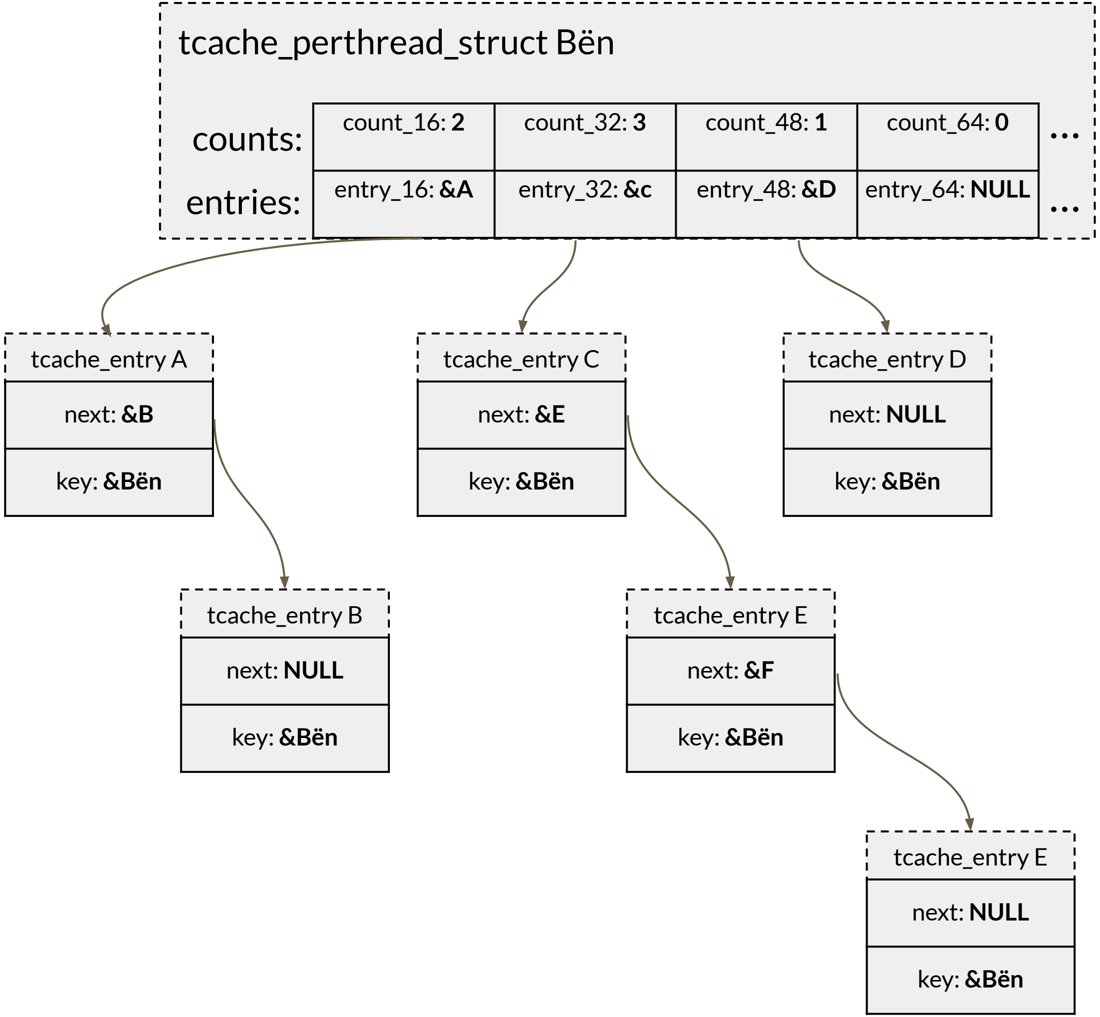
Reuse the allocation in the front of the list if available:

```
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:

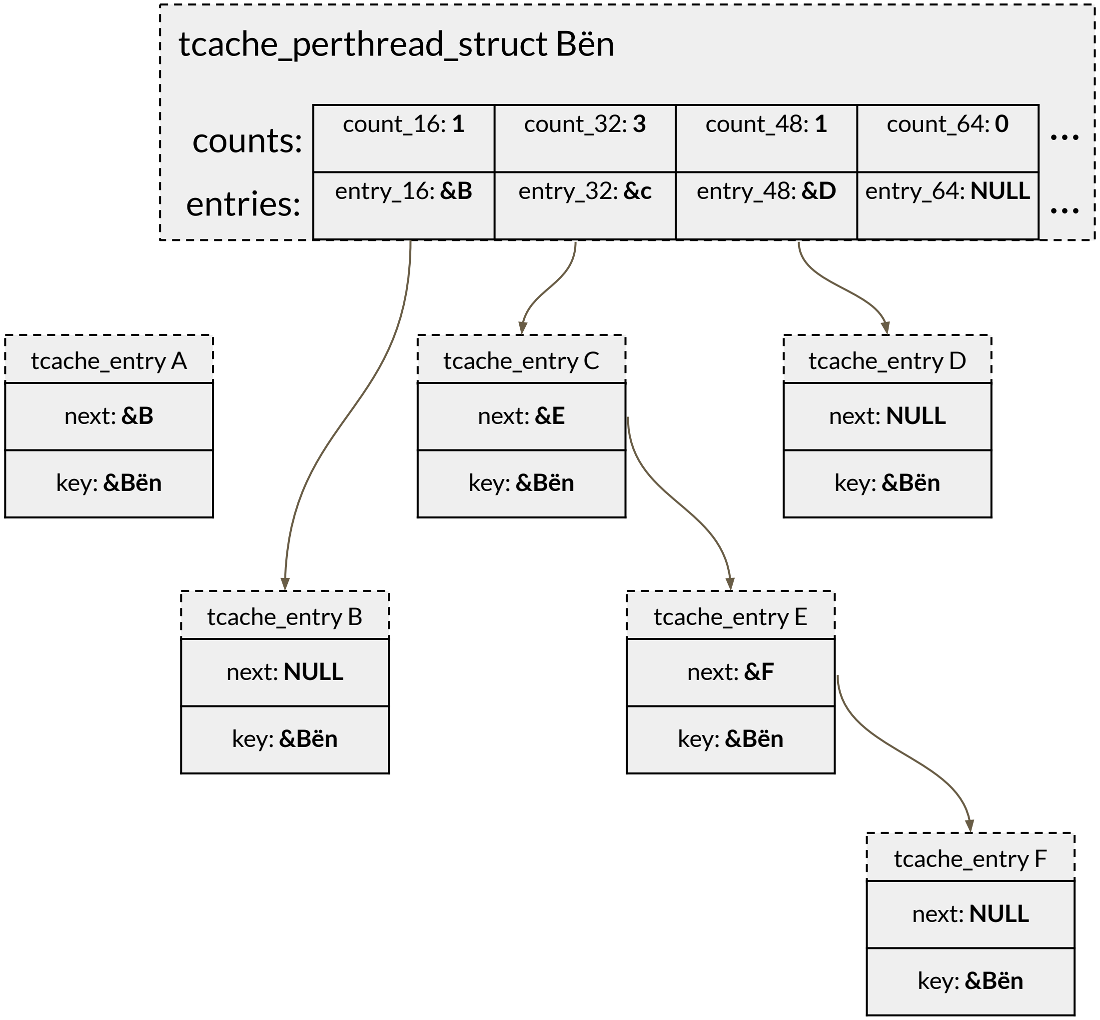
- clearing all sensitive pointers (only key is cleared for some reason).
- checking if the next (return[0]) address makes sense

Onward!


tcache_perthread_struct Bën
counts:
entries:
count_16: **2**
count_32: **3**
count_48: **1**
count_64: **0**
...
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 E
next: **NULL**
key: **&Bën**

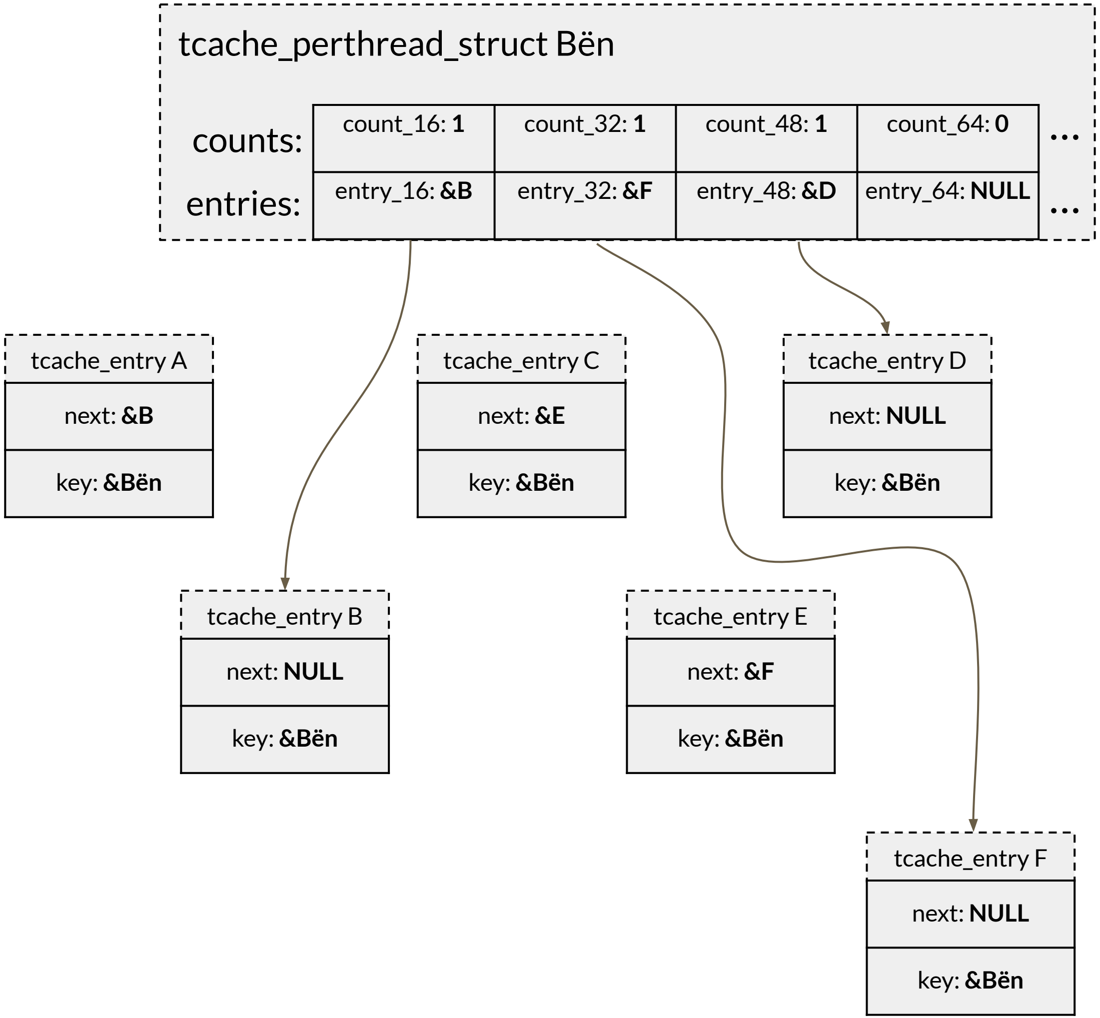
Onward!

malloc(16) == a



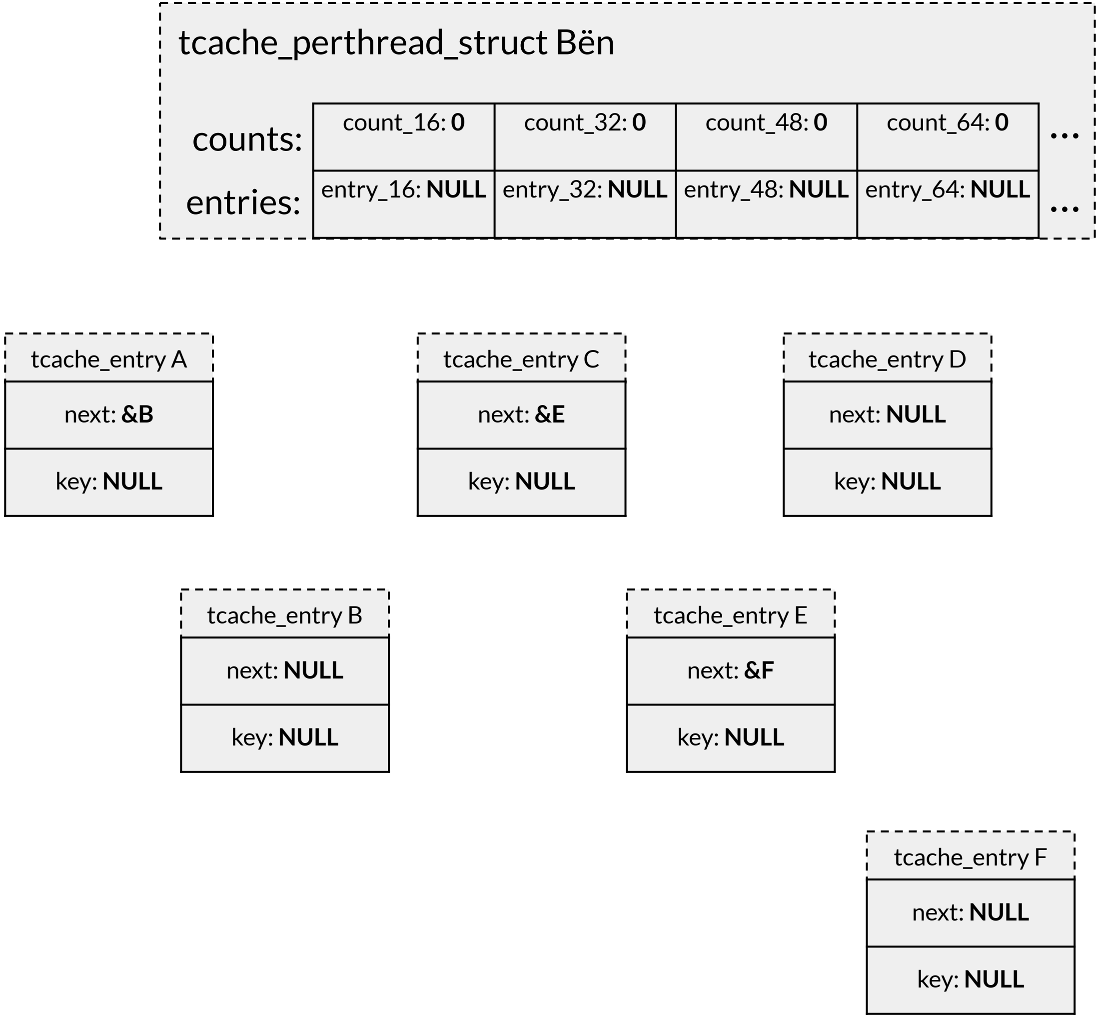
Onward!

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



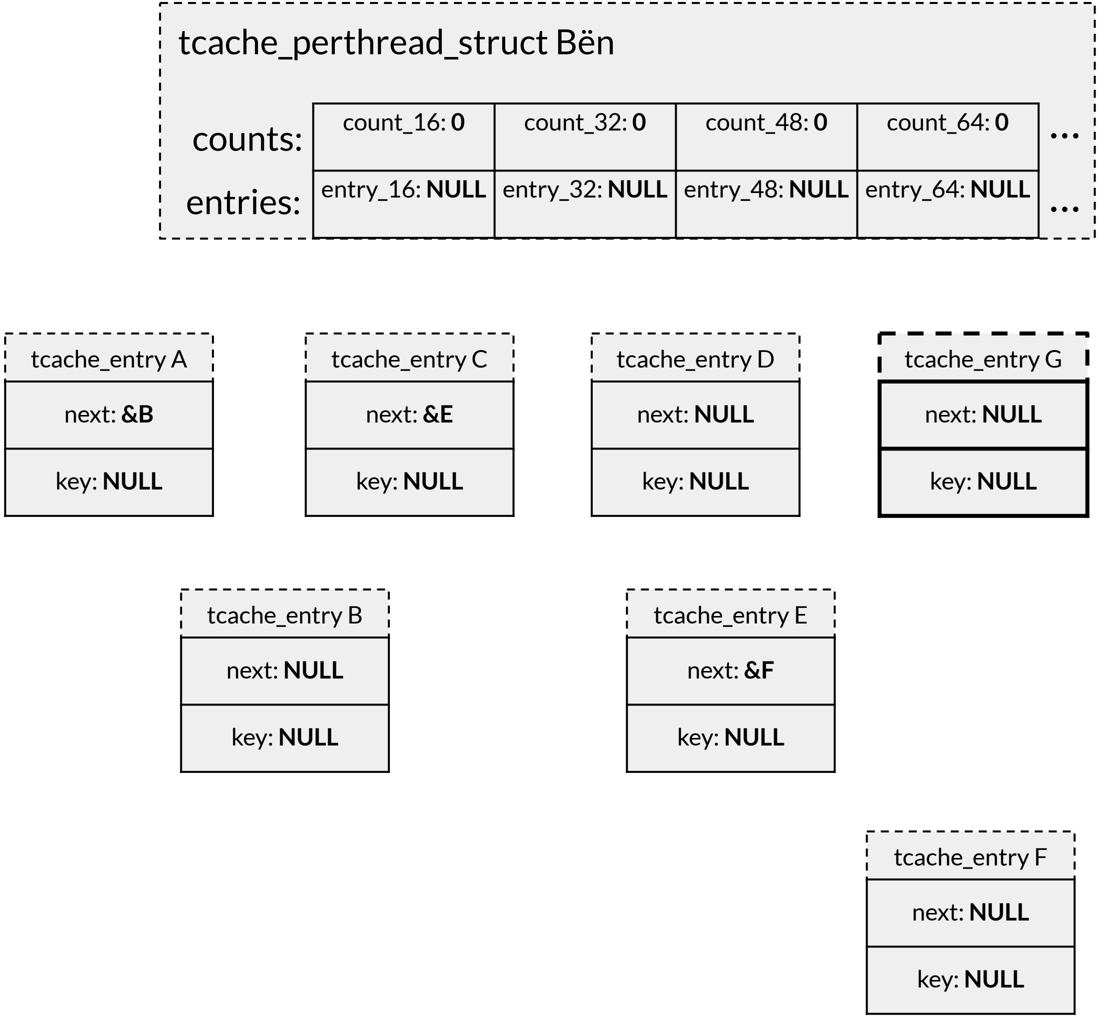
Onward!

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



Onward!

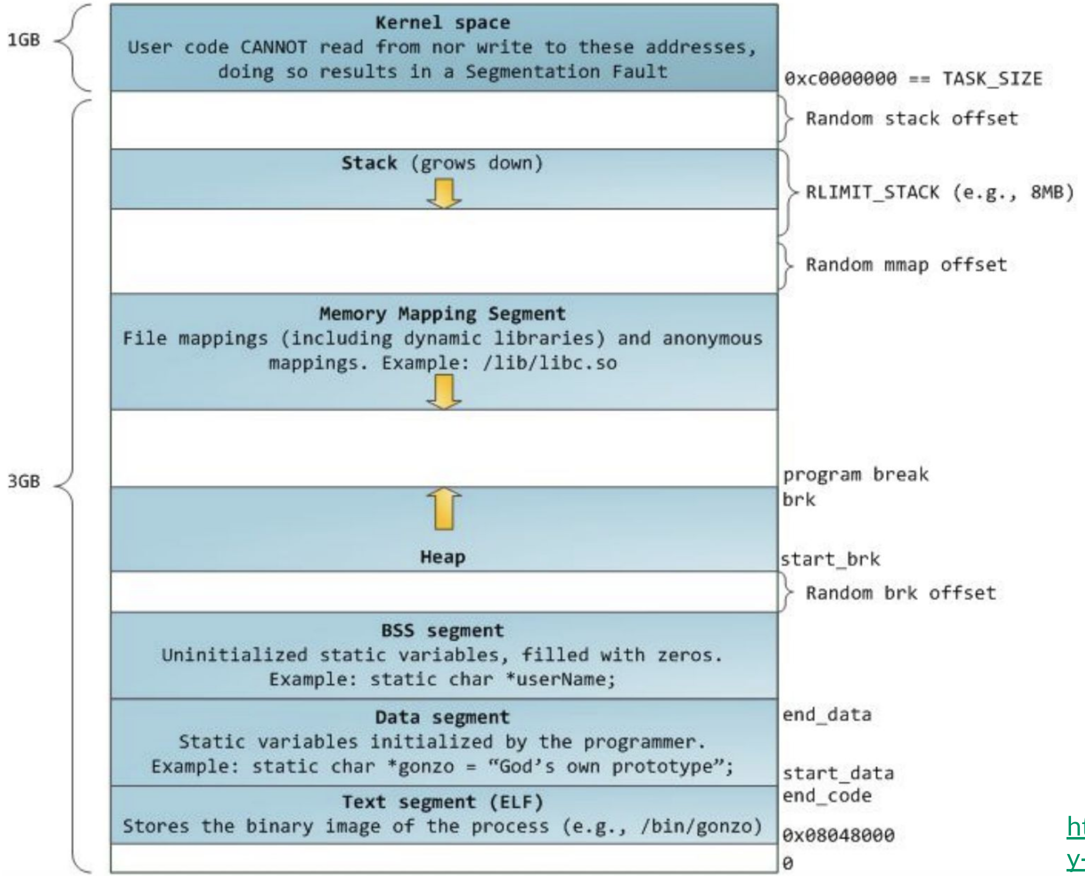
malloc(16) == a
malloc(32) == c
malloc(32) == e
malloc(48) == d
malloc(16) == b
malloc(32) == f
malloc(64) == g



code/heapsizes

```
int main()
{
        unsigned int lengths[] = {32, 4, 20, 0, 64, 32, 32, 32, 32, 32};
        unsigned int * ptr[10];

        int i;
        for(i = 0; i < 10; i++)
                ptr[i] = malloc(lengths[i]);
        for(i = 0; i < 9; i++)
                printf("malloc(%2d) is at 0x%08x, %3d bytes to the next pointer\n",
                lengths[i],
                (unsigned int)ptr[i],
                (ptr[i+1]-ptr[i])*sizeof(unsigned int));
        return 0;
}
```

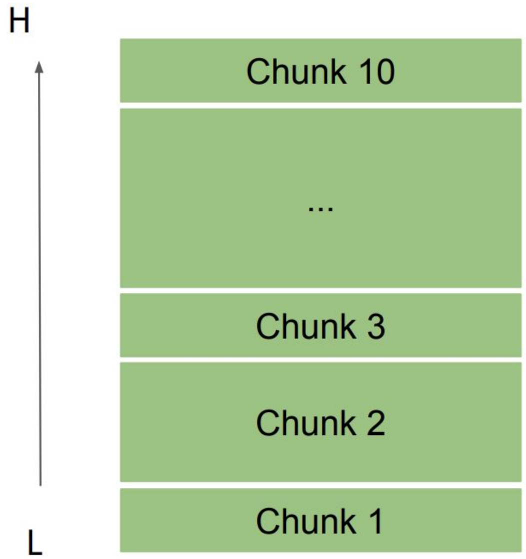
Heap goes from low address to high address



https://manybutfinite.com/post/anatomy-of-a-program-in-memory/

code/heapsizes

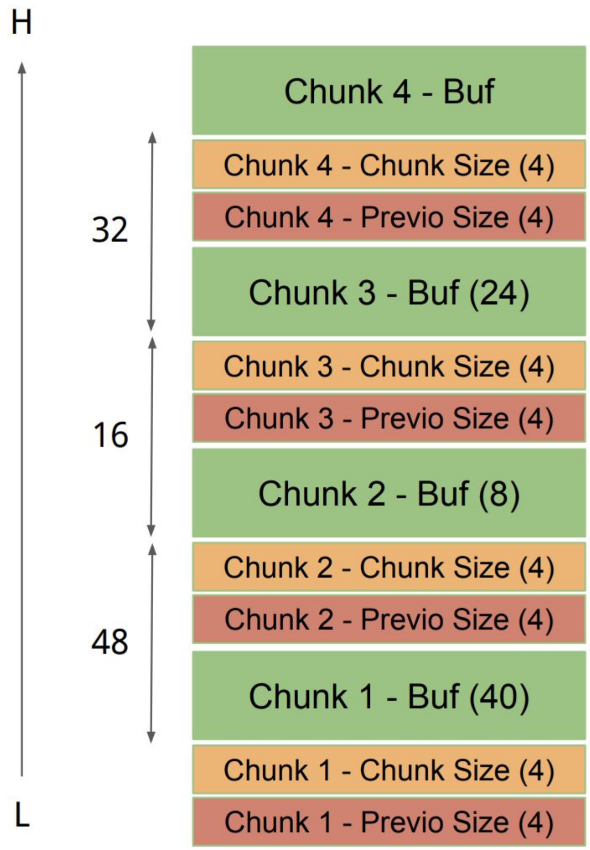
```
int main()
{
        unsigned int lengths[] = {32, 4, 20, 0, 64, 32, 32, 32, 32, 32};
        unsigned int * ptr[10];

        int i;
        for(i = 0; i < 10; i++)
                ptr[i] = malloc(lengths[i]);
        for(i = 0; i < 9; i++)
                printf("malloc(%2d) is at 0x%08x, %3d bytes to the next
        pointer\n",
                lengths[i],
                (unsigned int)ptr[i],
                (ptr[i+1]-ptr[i])*sizeof(unsigned int));
        return 0;
}
```



code/heapsizes 32bit

chunk_sizes = {32, 4, 20, 0, 64, 32, 32, 32, 32, 32};

```
ctf@heapexploitation_heapsizes_32:/$ ./heapexploitation_heapsizes_32
The size of size_t is 4
The size of a pointer is 4
malloc(32) is at 0x5655a5b0,  48 bytes to the next pointer
malloc( 4) is at 0x5655a5e0,  16 bytes to the next pointer
malloc(20) is at 0x5655a5f0,  32 bytes to the next pointer
malloc( 0) is at 0x5655a610,  16 bytes to the next pointer
malloc(64) is at 0x5655a620,  80 bytes to the next pointer
malloc(32) is at 0x5655a670,  48 bytes to the next pointer
malloc(32) is at 0x5655a6a0,  48 bytes to the next pointer
malloc(32) is at 0x5655a6d0,  48 bytes to the next pointer
malloc(32) is at 0x5655a700,  48 bytes to the next pointer
```

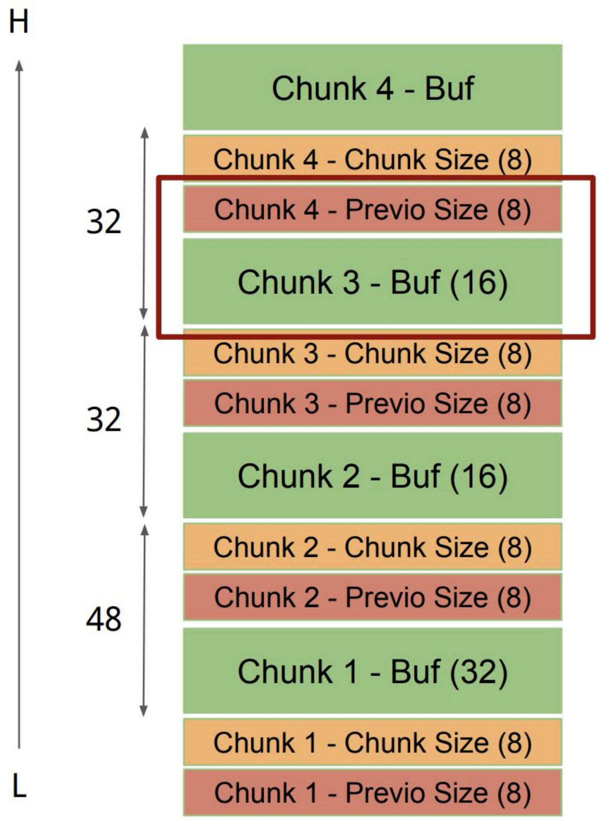


Alignment is at least defined as 2 * (sizeof(size_t)) = 8

code/heapsizes 64bit

chunk_sizes = {32, 4, **20**, 0, 64, 32, 32, 32, 32, 32};

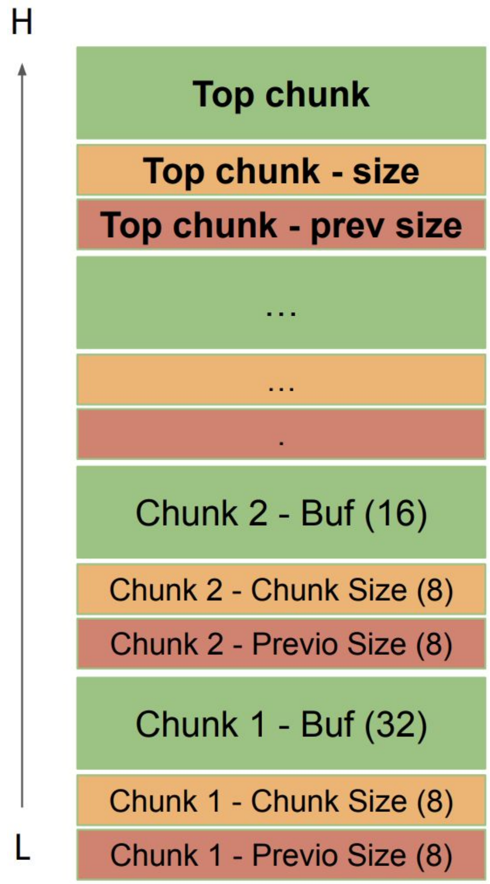
```
ctf@heapexploitation_heapsizes_64:/$ ./heapexploitation_heapsizes_64
The size of size_t is 8
The size of a pointer is 8
malloc(32) is at 0x555596b0,  48 bytes to the next pointer
malloc( 4) is at 0x555596e0,  32 bytes to the next pointer
malloc(20) is at 0x55559700,  32 bytes to the next pointer
malloc( 0) is at 0x55559720,  32 bytes to the next pointer
malloc(64) is at 0x55559740,  80 bytes to the next pointer
malloc(32) is at 0x55559790,  48 bytes to the next pointer
malloc(32) is at 0x555597c0,  48 bytes to the next pointer
malloc(32) is at 0x555597f0,  48 bytes to the next pointer
malloc(32) is at 0x55559820,  48 bytes to the next pointer
```



Alignment is defined as 2 * (sizeof(size_t)) = 16

Chunk4's previous size field is used by Chunk3

Top chunk a.k.a. wilderness



The topmost chunk is also known as the 'wilderness'.

It borders the end of the heap (i.e. it is at the maximum address within the heap/arena) and is not present in any bin.

It follows the same format of the chunk structure.

While servicing 'malloc' requests, it is used as the last resort. If more size is required, it can grow using the sbrk() system call. The **PREV_INUSE** flag is always set for the top chunk.

In the beginning, this is the only chunk existing and malloc first makes allocated chunks by splitting the wilderness chunk.

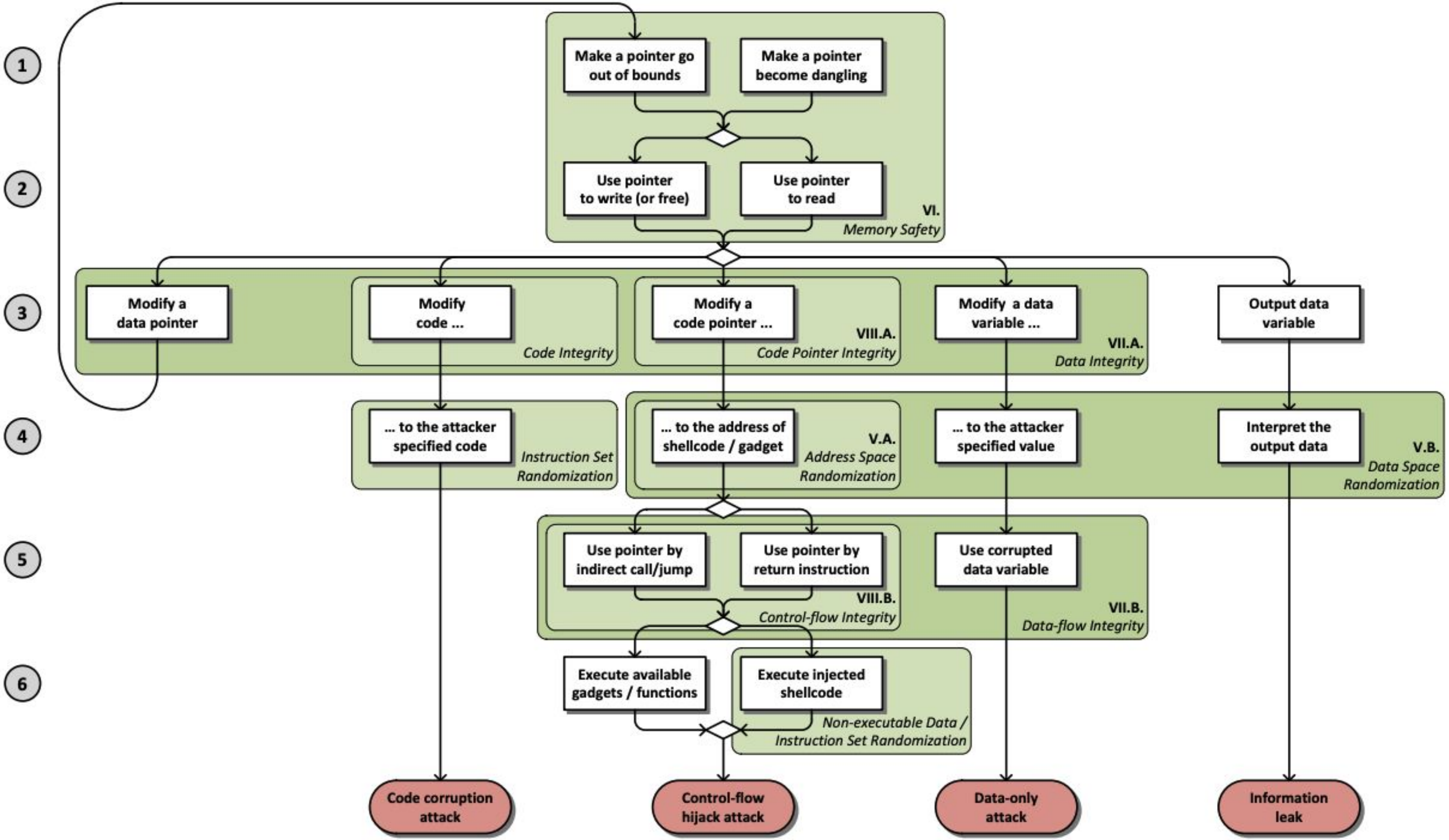
code/chunk_sizes

```
/*
  malloc(size_t n)
  Returns a pointer to a newly allocated chunk of at least n
  bytes, or null if no space is available. Additionally, on
  failure, errno is set to ENOMEM on ANSI C systems.

  If n is zero, malloc returns a minimum-sized chunk. (The
  minimum size is 16 bytes on most 32bit systems, and 24 or 32
  bytes on 64bit systems.)  On most systems, size_t is an unsigned
  type, so calls with negative arguments are interpreted as
  requests for huge amounts of space, which will often fail. The
  maximum supported value of n differs across systems, but is in
  all cases less than the maximum representable value of a
  size_t.
*/
```

Malloc Trivia

How many bytes on the heap are your ***malloc chunks*** really taking up?

- malloc(32); 48 bytes (32bit/64bit)
- malloc(4); 16 bytes (32bit) / 32 bytes (64bit)
- malloc(20); 32 bytes (32bit/64bit [Prev Size field reused])
- malloc(0); 16 bytes (32bit) / 32 bytes (64bit)



SoK: Eternal War in Memory. IEEE S&P 2013