Software Security 1

Administrative

Kevin Borgolte kevin.borgolte@rub.de

Tentative Lecture Schedule / Deadlines

Lectures
Wednesday 10-12

1. Oct 9 – Lecture	
2. Oct 16 – Flipped classroom	
3. Oct 23 – Lecture	← First assignment due
4. Oct 30 – Flipped classroom	
5. Nov 6 – Lecture	← Second assignment due
6. Nov 13 – Flipped classroom	
7. Nov 20 – Lecture	← Third assignment due
8. Nov 27 – Flipped classroom	
9. Dec 4 – Lecture	← Fourth assignment due
10. Dec 11 – Flipped classroom	
11. Dec 18 – Guest? Lecture	← Fifth assignment due
12. Jan 8 – Lecture	
13. Jan 15 – Flipped classroom	
14. Jan 22 – Lecture	← Sixth assignment due
15. Jan 29 – Flipped classroom	

Tentative

(will probably not change anymore)

Assignments

- Questions
  - Moodle or email us (softsec+teaching@rub.de)
  - If you run into issues, please report your OS, Docker version, etc.
- Assignment 3
  - 5 tasks (boromir, tarzan, srop, magic8ball, phonebook)
    - Looking at the scoreboard progress, they seemed a bit easy?
  - Due: Midnight this evening! (November 21st, 0:00 Bochum time)
- Assignment 4
  - 5 tasks, may NOT be released tomorrow morning
  - Due: December 5th, 0:00 Bochum time

Reverse Engineering

- So far, you got source code and binary executable
- For one challenge assignment 4, you will not get source code
  - You will need to analyze the disassembly or decompile it
  - Future assignments will have more challenges without source code
- There are a variety of open source tools you can use for reversing, but Hex Rays also has given us ~50 licenses for IDA Classroom Free
  - They are named 1 year licenses and need to be requested individually
    - You need to have solved some assignments to request a license
    - We will post instructions how to request one on Moodle later today
- We will show by example how to reverse in the exercises (starting tomorrow) and next week in the lecture

Exam Setup

- Exam dates (you need to attend both)
  - Monday, March 3rd
  - Wednesday, March 5th
- Exam setup will be the same as MC 5/222
  - Tentative: With Internet iff screen recording works for all machines
    - We will not record you + our machines
  - Please use the image extensively and help us identify issues
  - There will be a dedicated test day in late January/early February with the final exam set up, exact date TBD

Topics Today

- Introduction to Heap Attacks

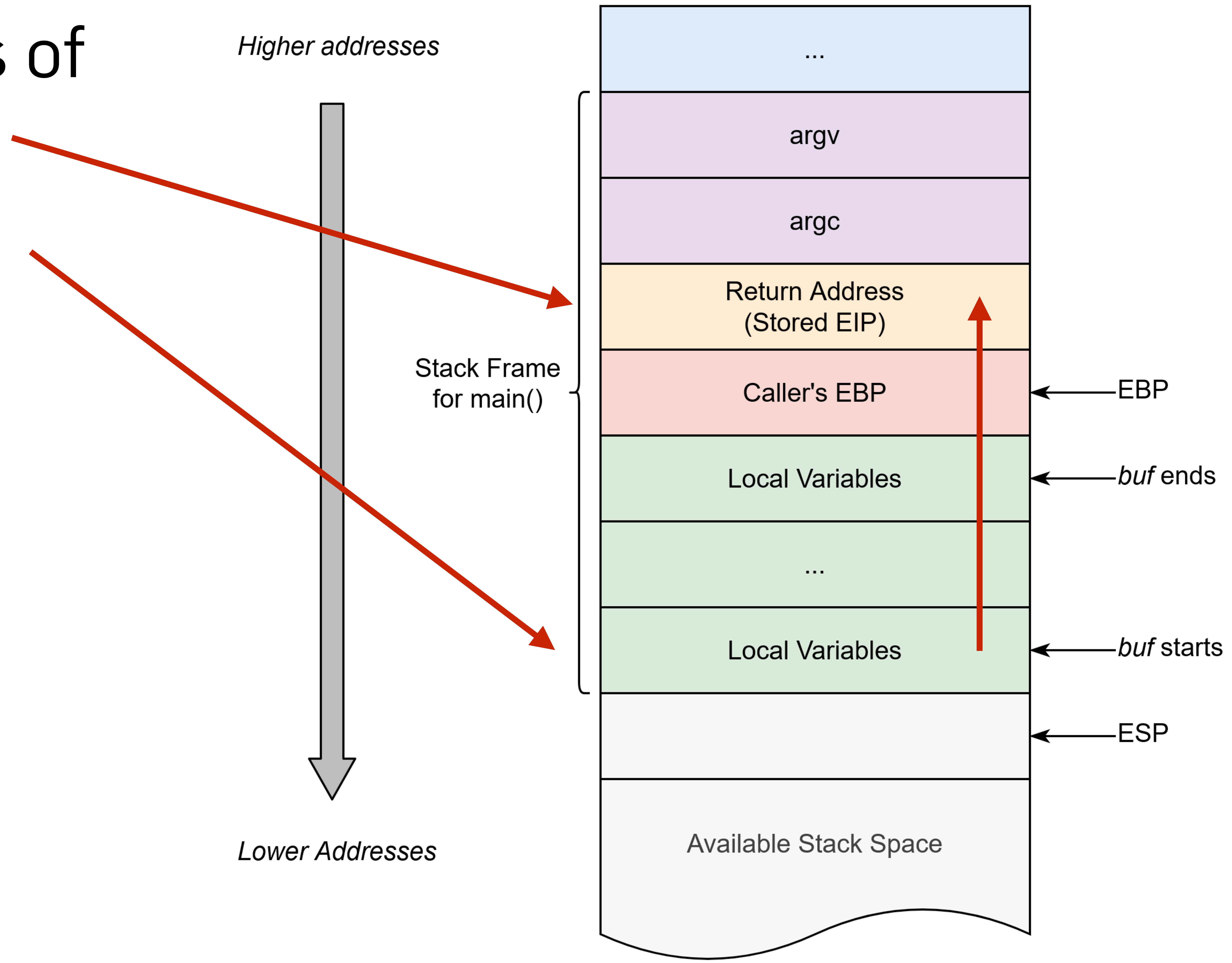
Software Security

Introduction to Heap Attacks

Kevin Borgolte kevin.borgolte@rub.de

Recap: Stack-based Buffer Overflow

- Overwrite return address of a function in a program
  - Some buffer is on the stack
  - Stack grows down
  - Buffer is at a lower address than return address
  - Writing more means overwriting return address
  - Code execution



Recap: Memory Areas

- Memory areas/regions we have seen so far
  - ELF section headers
    - .text: code
    - .plt: library function stubs
    - .got: pointers to imported symbols
    - .bss: uninitialized global writable data
    - .data: pre-initialized global writable data
    - .rodata: global read-only data
  - Stack: local variables, temporary storage, call stack metadata

Recap: Memory Areas

- Memory areas/regions we have seen so far
  - ELF section headers
    - .text: code
    - .plt: library function stubs
    - .got: pointers to imported symbols
    - .bss: uninitialized global writable data
    - .data: pre-initialized global writable data
    - .rodata: global read-only data
  - Stack: local variables, temporary storage, call stack metadata

This might not be enough. We might not know at compile-time how much memory we might need exactly, and the stack is short-lived. We need long-lived dynamic memory.

Dynamic Memory

- We can use mmap() to calculate dynamic memory as needed
  - Relatively inflexible: Allocation size must be pages (4K or more)
  - Slow: Calls to the kernel

Dynamic Memory

- We can use mmap() to calculate dynamic memory as needed
  - Relatively inflexible: Allocation size must be pages (4K or more)
  - Slow: Calls to the kernel
- Better: Wrap mmap() in a library that manages memory
  - Flexible: Program can give out smaller amounts of memory by itself
  - Faster: Only needs to call to the kernel when it needs more space

Dynamic Memory Allocators

- Wrap mmap() in a library that manages memory is what dynamic memory allocators (DMA) do, like
  - dlmalloc, ptmalloc
  - jemalloc (FreeBSD, Firefox, Android)
  - Segment Heap, NT Heap (Windows)
  - kmalloc (Linux kernel)
  - kalloc (Apple iOS kernel)
  - glibc
- The memory space managed by the DMA is called the **heap**

Heap: Management

- Usually implemented via some variants of functions:
  - malloc(): allocate some memory and claim it
  - free(): return the allocated memory to the DMA
  - realloc(): change the size of a reallocation
  - calloc(): allocate a contiguous amount of space for *count* objects
- We briefly talked about this in a side note before, their behavior might differ (quite a bit) based on the DMA
- They are used by basically every software

Heap: Inner Workings

- Not all DMAs use mmap(), or use mmap() always
  - Small allocations might be in the .data segment
    - Mostly historic: syscalls sbrk and brk can be used to extend .data
      - Even the manpage warns: "The brk and sbrk functions are historical curiosities left over from earlier days before the advent of virtual memory management."
  - Larger allocations are almost always mmap()'d directly
  - Actual behavior differs based on the used DMA (its version, etc.)

Heap: Issues

- Developers are human and make mistakes
  - Forget to free() memory
  - Forget what memory they have free()'d
  - Forget dangling pointers (e.g., pointers to objects they free'd)
- DMAs are libraries that need to be **high performance / fast**
  - If allocations/deallocations are not fast, then the interesting computations and program behavior will be slow (one of the reasons memory is often directly reused for high-performance computations)
  - Optimizations aim for performance, rarely for security
- DMA assumes the developer is using the DMA properly and safely, it makes assumptions based on it, security issues arise if they do not

Heap: Detecting Issues

- Some tools exist to detect heap misuse
  - valgrind
  - AddressSanitizer (ASAN)
  - glibc has hardening techniques that can be enabled: MALLOC_CHECK, MALLOC_PERTURB, MALLOC_MMAP_THRESHOLD

"AddressSanitizer: A Fast Address Sanity Checker", Serebryany et al., https://www.usenix.org/system/files/conference/atc12/atc12-final39.pdf 15

Heap: Detecting Issues

- Some tools exist to detect heap misuse
  - valgrind
  - AddressSanitizer (ASAN)
  - glibc has hardening techniques that can be enabled: MALLOC_CHECK, MALLOC_PERTURB, MALLOC_MMAP_THRESHOLD
- These come with high performance overheads, making it very challenging to run in production
  - AddressSanitizer:
    - 1.73x runtime (73% longer)
    - 3.37x memory (240% more)

"AddressSanitizer: A Fast Address Sanity Checker", Serebryany et al., https://www.usenix.org/system/files/conference/atc12/atc12-final39.pdf

Heap: Security Issues

- Forget to free() memory
  - Resource exhaustion, denial of service (memory usage running wild, program slowing down, etc.)

Heap: Security Issues

- Forget to free() memory
    - Resource exhaustion, denial of service (memory usage running wild, program slowing down, etc.)
- Forget what memory they have free()'d
    - Using free memory
    - Freeing free memory (again)

Heap: Security Issues

- Forget to free() memory
  - Resource exhaustion, denial of service (memory usage running wild, program slowing down, etc.)
- Forget what memory they have free()'d
  - Using free memory
  - Freeing free memory (again)
- Corrupt metadata the DMA uses to keep track of the heap
  - Conceptually very similar to corrupting the function state on the stack

Heap: Memory Leaks

- Allocated memory must be explicitly freed

```
void foo() {
    char *bar = malloc(1024);
    /* do stuff with bar */
    return 1;
}
```

Why is this an issue?

Heap: Memory Leaks

- Allocated memory must be explicitly freed

```
void foo() {
    char *bar = malloc(1024);
    /* do stuff with bar */
    return 1;
}
```

Why is this an issue?

We might not know anymore that bar was allocated at all (we return from the function and the pointer is on the stack, but the memory area it points to is not).

Heap: Use After Free (UAF)

- Pointers to an allocation remain valid after being free()'d, they could be used after

```c
void main() {
  char *input = malloc(8); // assuming not NULL
  printf("Name? ");
  scanf("%7s", input);
  printf("Hello %s!\n", input);
  free(input);

  long *authenticated = malloc(8); // assuming not NULL
  *authenticated = 0;

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

  if (getuid() == 0 || strcmp(input, "foobar") == 0) {
    *authenticated = 1;
  }

  if (*authenticated) {
    endfile(0, open("/flag", 0), 0, 128);
  }
}
```

Heap: Use After Free (UAF)

- Pointers to an allocation remain valid after being free()'d, they could be used after

input has been free()'d

```
void main() {
  char *input = malloc(8); // assuming not NULL
  printf("Name? ");
  scanf("%7s", input);
  printf("Hello %s!\n", input);
  free(input);

  long *authenticated = malloc(8); // assuming not NULL
  *authenticated = 0;

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

  if (getuid() == 0 || strcmp(input, "foobar") == 0) {
    *authenticated = 1;
  }

  if (*authenticated) {
    endfile(0, open("/flag", 0), 0, 128);
  }
}
```

Heap: Use After Free (UAF)

- Pointers to an allocation remain valid after being free()'d, they could be used after

input has been free()'d

but is reused

```
void main() {
  char *input = malloc(8); // assuming not NULL
  printf("Name? ");
  scanf("%7s", input);
  printf("Hello %s!\n", input);
  free(input);

  long *authenticated = malloc(8); // assuming not NULL
  *authenticated = 0;

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

  if (getuid() == 0 || strcmp(input, "foobar") == 0) {
    *authenticated = 1;
  }

  if (*authenticated) {
    endfile(0, open("/flag", 0), 0, 128);
  }
}
```

Heap: Use After Free (UAF)

- Pointers to an allocation remain valid after being free()'d, they could be used after

input has been free()'d

but is reused

and authenticated might point to the same location input pointed to

😳

```
void main() {
  char *input = malloc(8); // assuming not NULL
  printf("Name? ");
  scanf("%7s", input);
  printf("Hello %s!\n", input);
  free(input);

  long *authenticated = malloc(8); // assuming not NULL
  *authenticated = 0;

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

  if (getuid() == 0 || strcmp(input, "foobar") == 0) {
    *authenticated = 1;
  }

  if (*authenticated) {
    endfile(0, open("/flag", 0), 0, 128);
  }
}
```

Heap: Memory Disclosure

- Like UAF, but we are reading from memory instead of writing to it

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

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

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

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

Heap: Memory Disclosure

- Like UAF, but we are reading from memory instead of writing to it

name has been free()'d

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

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

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

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

Heap: Memory Disclosure

- Like UAF, but we are reading from memory instead of writing to it

name has been free()'d

but is reused

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

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

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

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

Heap: Memory Disclosure

- Like UAF, but we are reading from memory instead of writing to it

name has been free()'d

but is reused

- Between the free() and printf(), other parts of the program might have used the same memory area (including some DMAs using it for metadata)

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

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

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

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

Heap: Metadata Corruption

- Allocator metadata can not only be read via memory disclosure, but also written via UAF, leading to **heap exploits**
- Idea: Confuse the DMA into allocating **overlapping memory**, and then abuse that two pointers sharing memory
  - First pointer is security-relevant
  - Second pointer might be readable/writable by the attacker

Heap: Metadata Corruption

- Allocator metadata can not only be read via memory disclosure, but also written via UAF, leading to **heap exploits**
- Idea: Confuse the DMA into allocating **overlapping memory**, and then abuse that two pointers sharing memory
  - First pointer is security-relevant
  - Second pointer might be readable/writable by the attacker
- Security-relevant?
  - Authentication structures (e.g., is administrator?)
  - Sensitive data (e.g., a password or the flag)
  - Function pointers (e.g., classes or where to return to)
  - Metadata (e.g., length of a string, to induce other memory errors)

ptmalloc

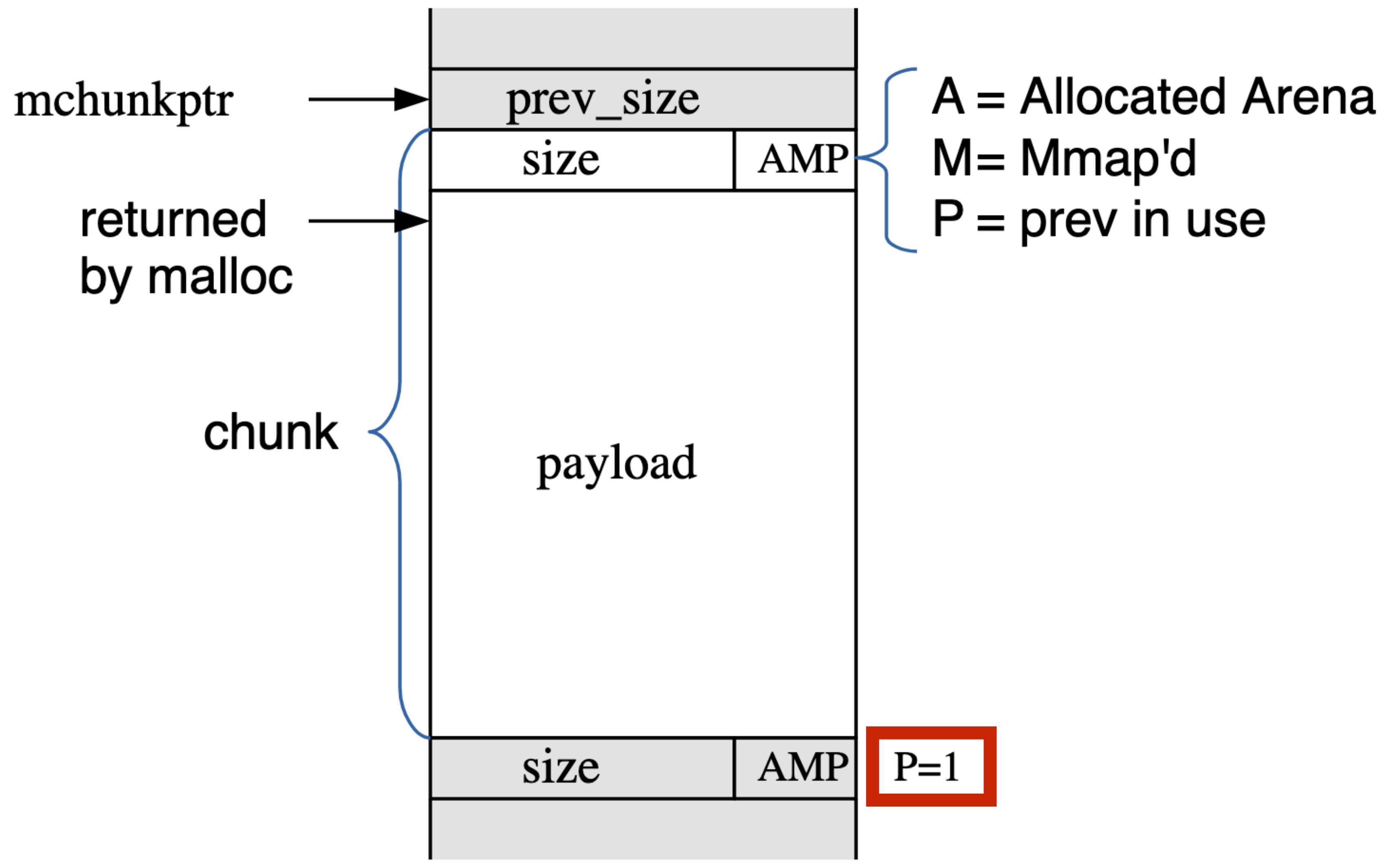
- Default DMA for most Linux applications (part of glibc)
- Allocates some memory from a large contiguous memory area and manage them.
    - Aims to reduce wastage in the form of unusable chunks.
- Traditionally used one large memory area, but now often maintains multiple areas to optimize their use in multi-threaded applications.
    - Idea #1: Keeping related memory areas close to reduce page faults.
    - Idea #2: DMA operations in one thread do not block others.

ptmalloc Terminology

- Arena
  - Data structure shared among threads that owns one or more heaps from which memory is being allocated.
- Heap
  - Contiguous memory region that is divided into chunks. Belongs to exactly one arena.
- Chunk
  - A (small) range of memory that can be allocate'd, free'd, or merged with adjacent chunks into a larger chunk. Belongs to exactly one heap.

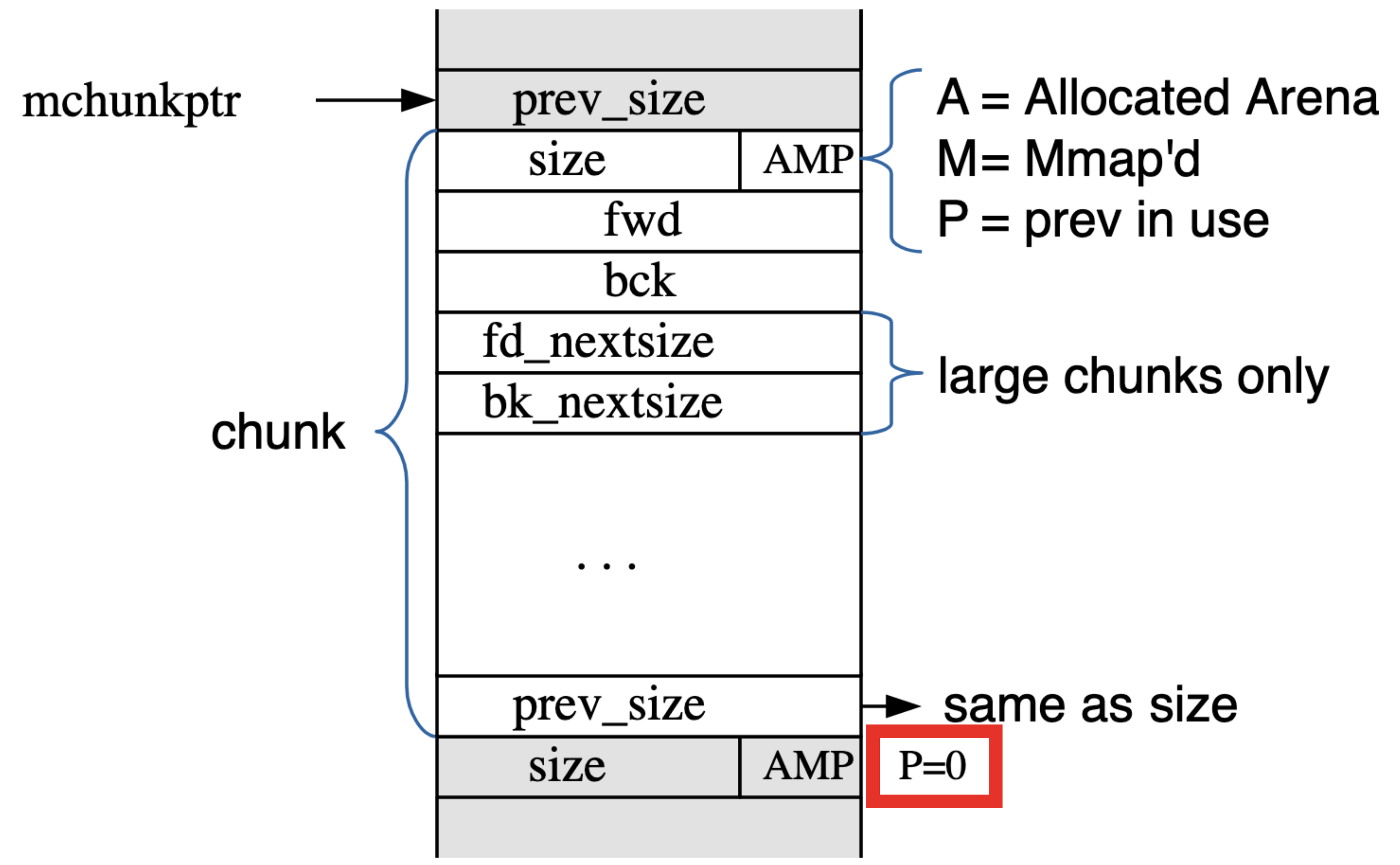
Chunks: Allocated and In-use

- Each chunk has meta-data, about its size
  - The chunk size is a multiple of 8 bytes, which means you can use the least significant 3 bits for additional flags
    - A, Allocated arena:
      If set to 0, the chunk is from the main arena and main heap (the heap that you get when loading the program).
      If set to 1, then it comes for mmap()'d memory (this can be a heap).
    - M, mmap'd:
      If set to 0, the chunk is part of a heap.
      If set to 1, then it was allocated in a mmap() call.
    - P, previous in use:
      If set to 0, the previous chunk can be merged with (to form a bigger memory chunk, for example for realloc() calls).
      If set to 1, then it is in use and cannot be merged with (it is considered in-use).



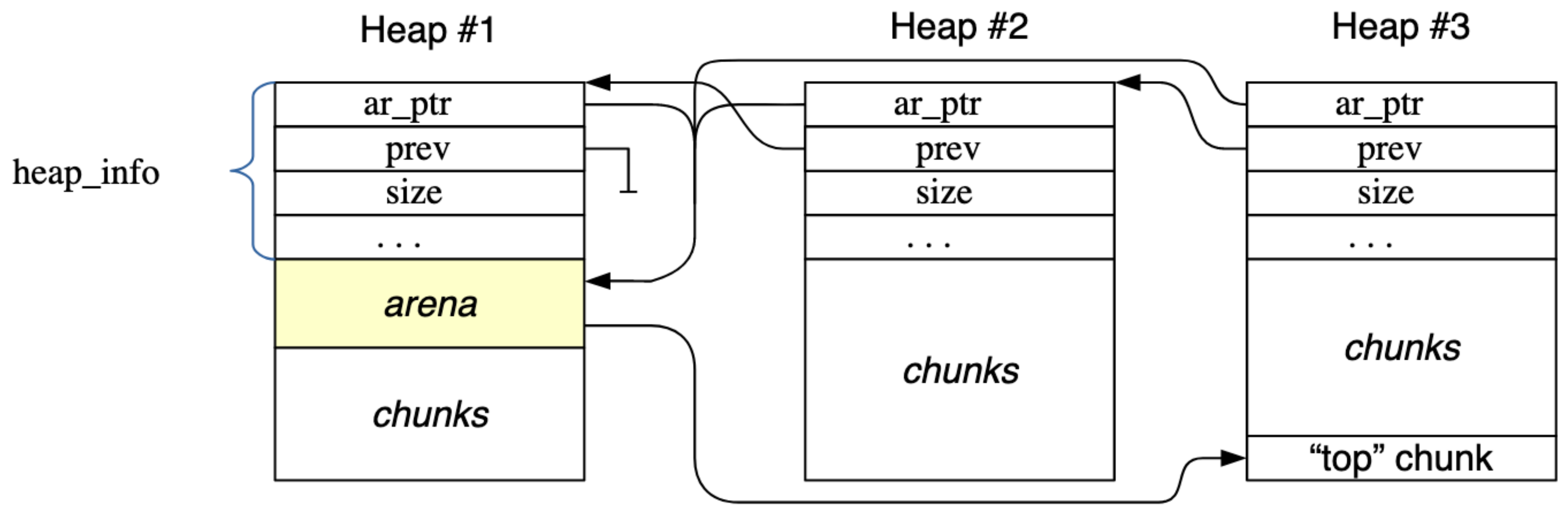
Chunks: Free

- To find free memory chunks that we can allocate, we can:
  - Iterate through all chunks starting at the beginning and checking size and flags.
  - Re-use the memory of the chunks and maintain a doubly-linked list of free chunks.
    - Advantage: No additional memory needed, and quicker
    - Disadvantage: Mixing data and control information 😬



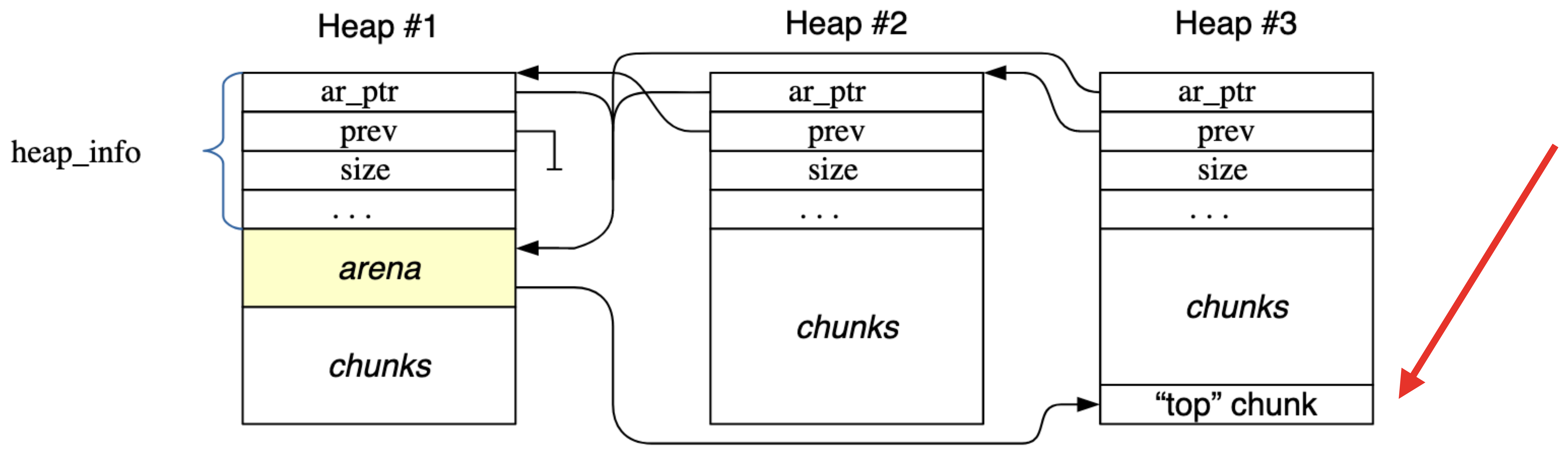
Arenas and Heaps

- Idea: More than one memory region can be active at the same time, so that multiple threads do not collide in their allocations.
- The memory for an arena is on the initial heap for that arena, for example, this arena with three heaps:



Arenas and Heaps

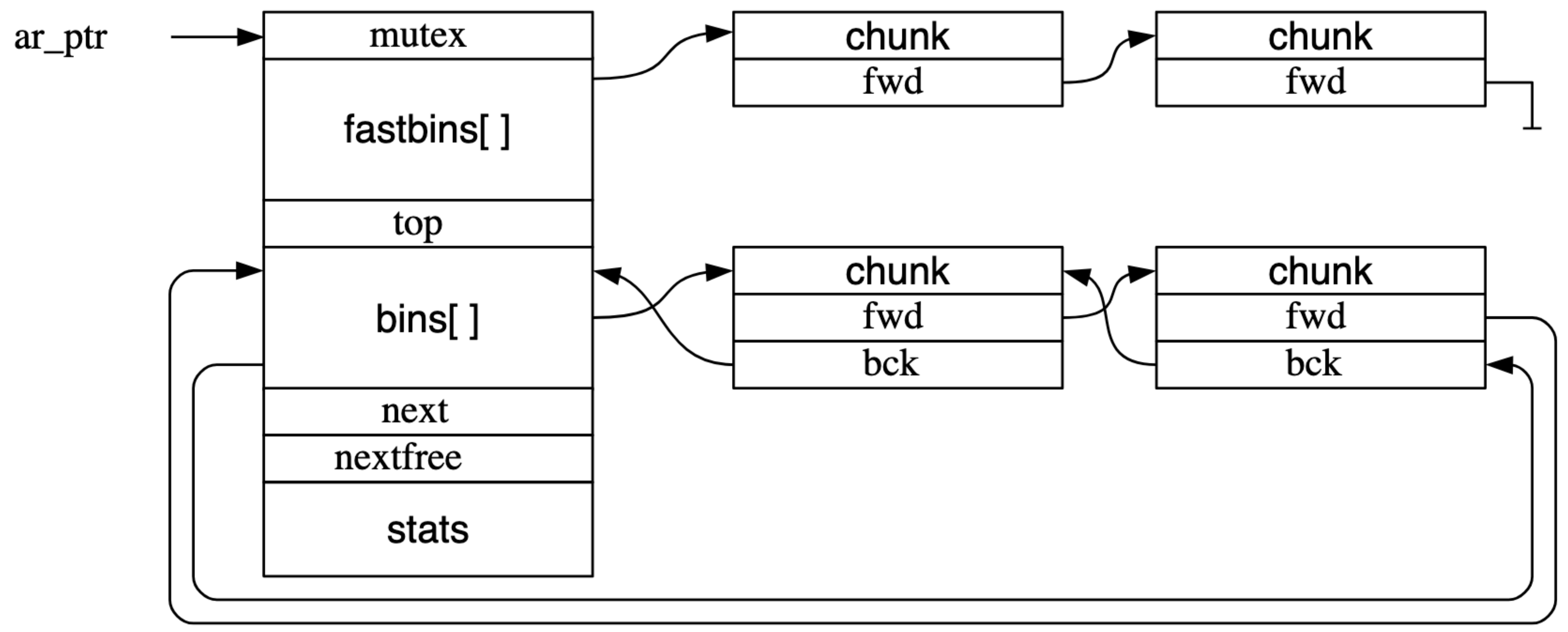
- Idea: More than one memory region can be active at the same time, so that multiple threads do not collide in their allocations.
- The memory for an arena is on the initial heap for that arena, for example, this arena with three heaps:



Typically the largest available chunk on the most recent heap.

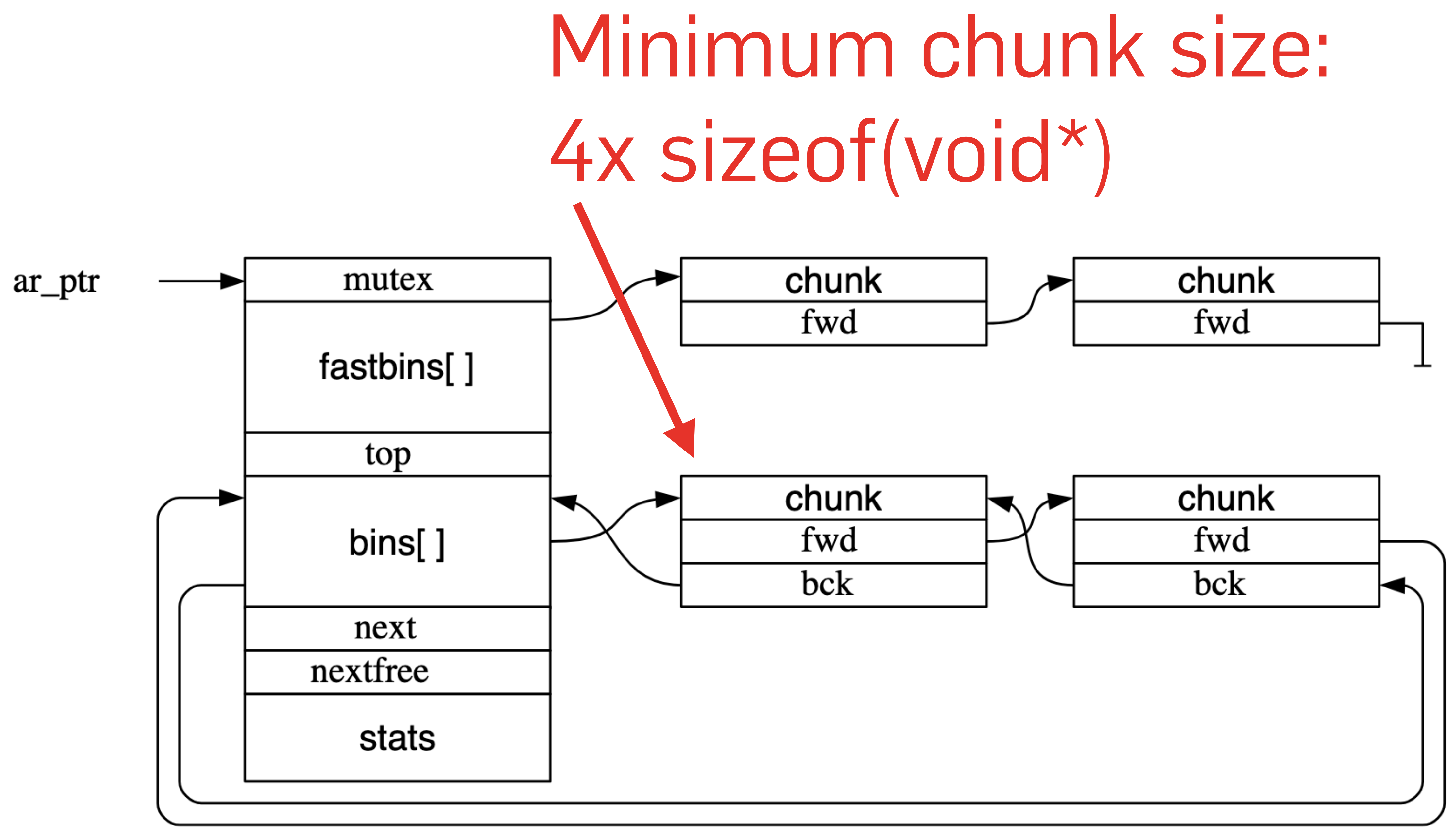
Arenas

- **Free chunks** are tracked by linked lists and binned:
  - Fast
    - Never combined with other chunks
    - Singly-linked list, never go back
    - All chunks have the same size
    - Multiple fastbins of different sizes
  - Unsorted
    - Free'd chunks that need to be moved to one of the bins
  - Small
    - All chunks in a small bin have the same size, you can just use the first chunk (you may have multiple small bins)
  - Large
    - Chunks have different sizes, you need to find the best fitting chunk (and possibly split it into two)
- Generally, whenever you add a chunk to any of these lists, you merge them with the chunks that immediately precede or succeed the chunk in memory (and then you might move these new chunks to another bin).

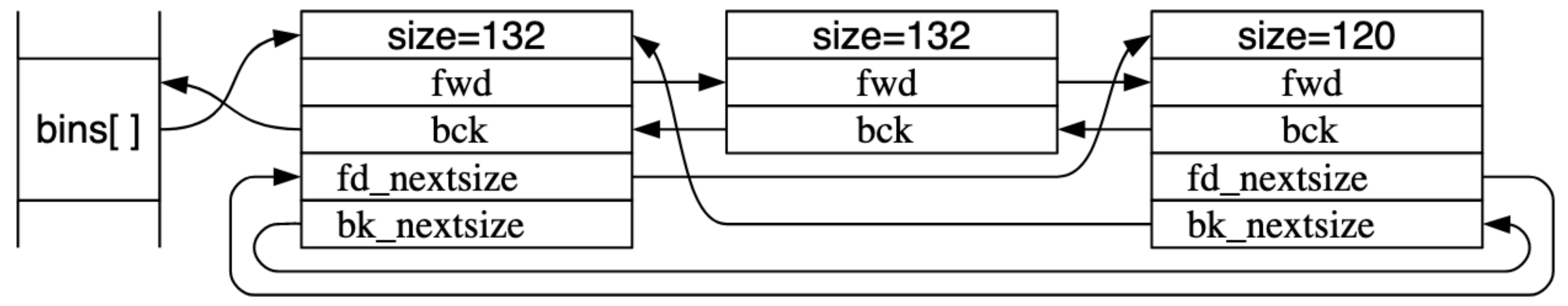


Arenas

- **Free chunks** are tracked by linked lists and binned:
  - Fast
    - Never combined with other chunks
    - Singly-linked list, never go back
    - All chunks have the same size
    - Multiple fastbins of different sizes
  - Unsorted
    - Free'd chunks that need to be moved to one of the bins
  - Small
    - All chunks in a small bin have the same size, you can just use the first chunk (you may have multiple small bins)
  - Large
    - Chunks have different sizes, you need to find the best fitting chunk (and possibly split it into two)
- Generally, whenever you add a chunk to any of these lists, you merge them with the chunks that immediately precede or succeed the chunk in memory (and then you might move these new chunks to another bin).



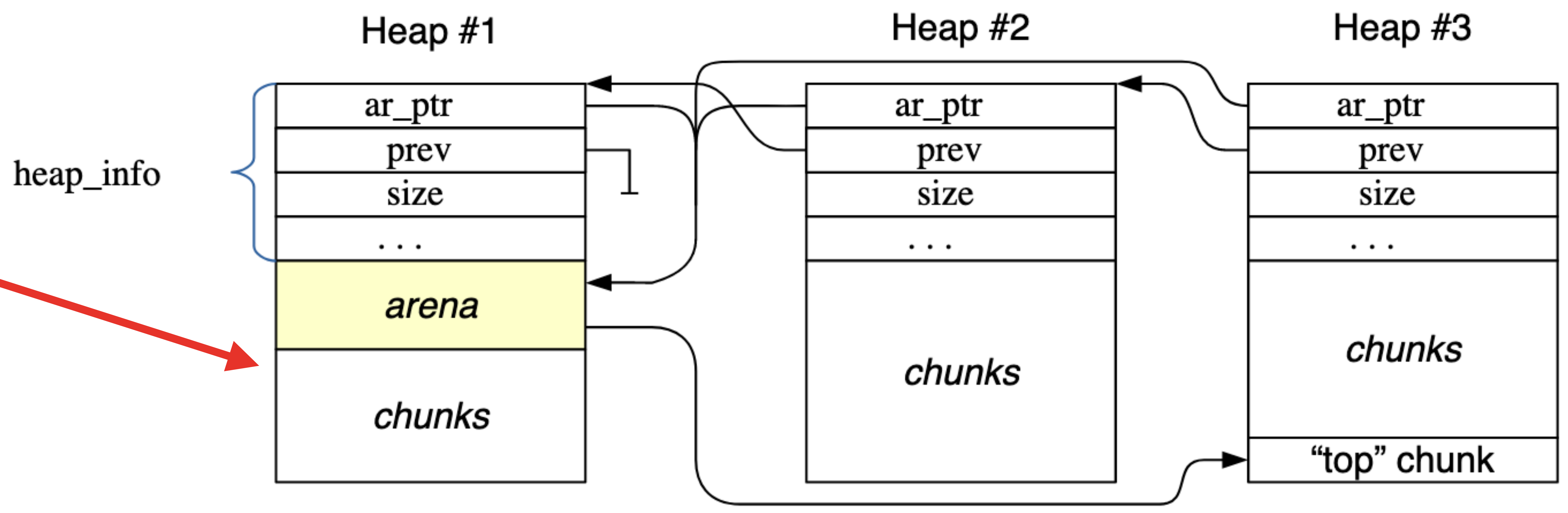
Arena: Large Bin



- If there is no "small" bin that has a chunk of sufficient size, we need to find a chunk
- Traverse the linked list of large chunks based on fd_nextsize/ bk_nextsize pointers to fit a large-enough chunk
- The list is sorted by size, largest to smallest
    - If the size of the next chunk is too small, take the current chunk
    - To make it faster, take the second chunk of the size that fits (we only need to update fwd/bck, but not fd_nextsize/bk_nextsize)

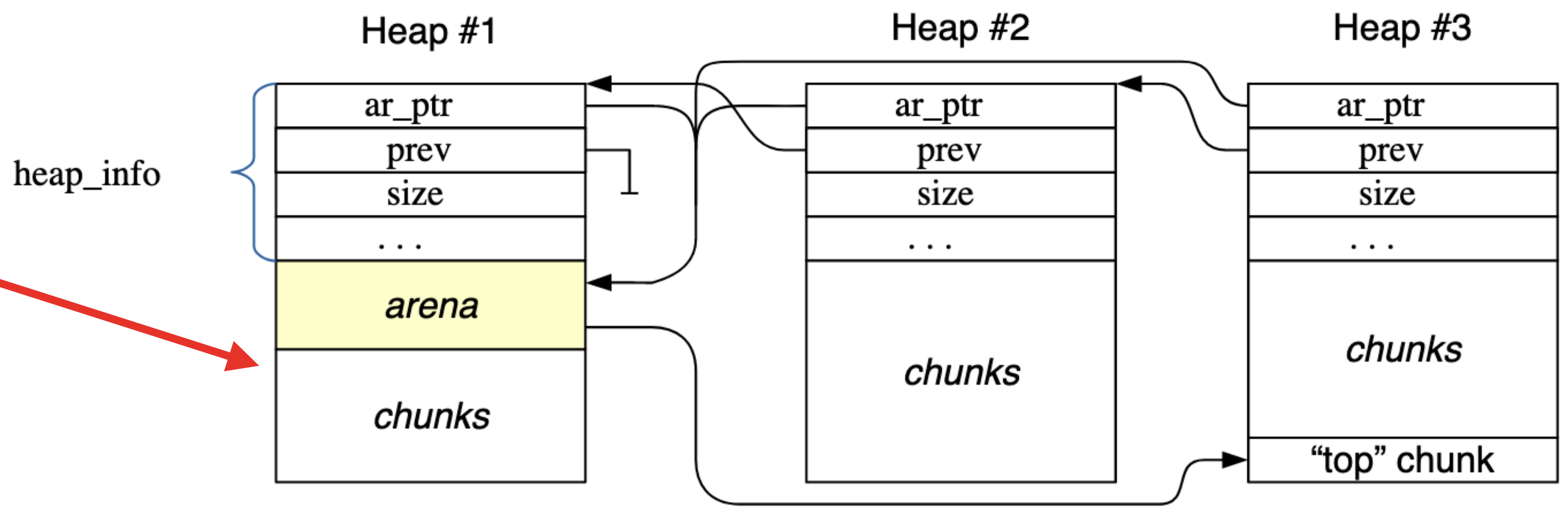
Issues

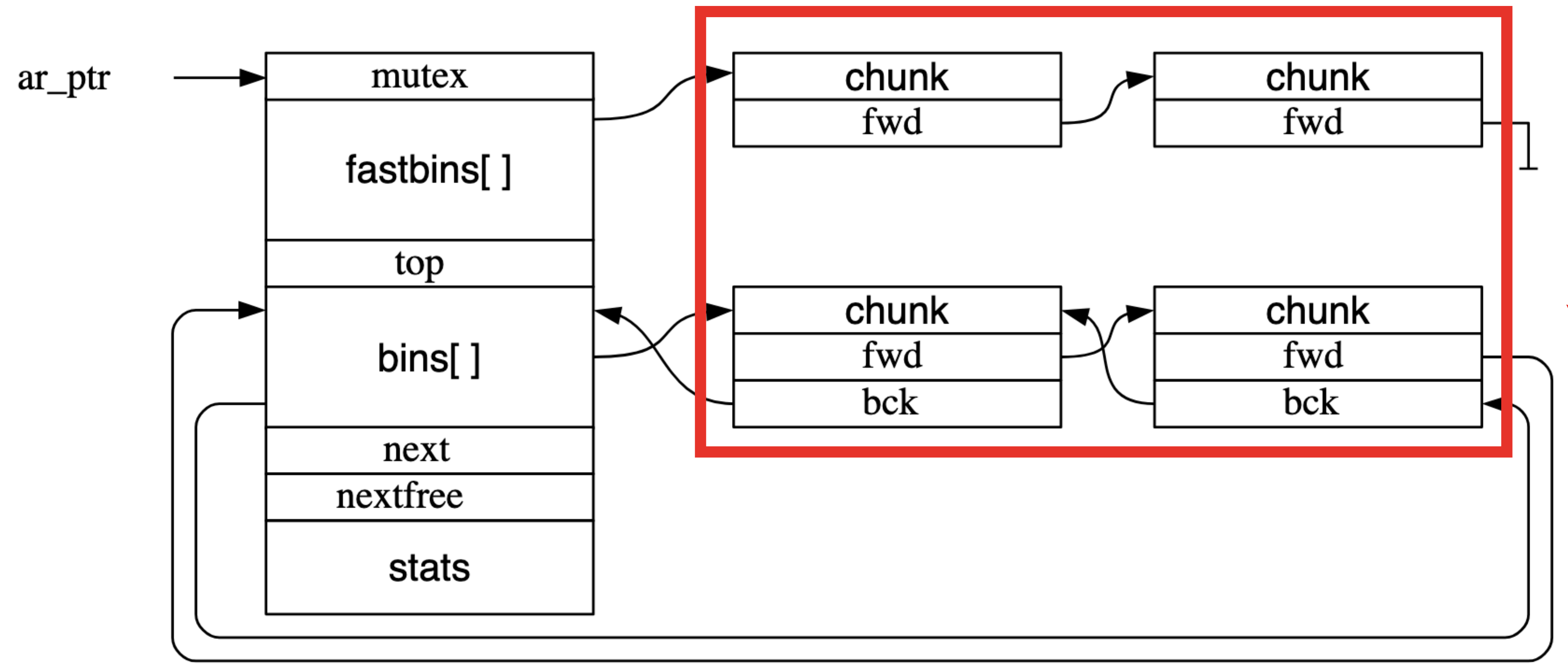
Mixing data and metadata
p = malloc()
p + some (negative) offset points into arena and heap meta data

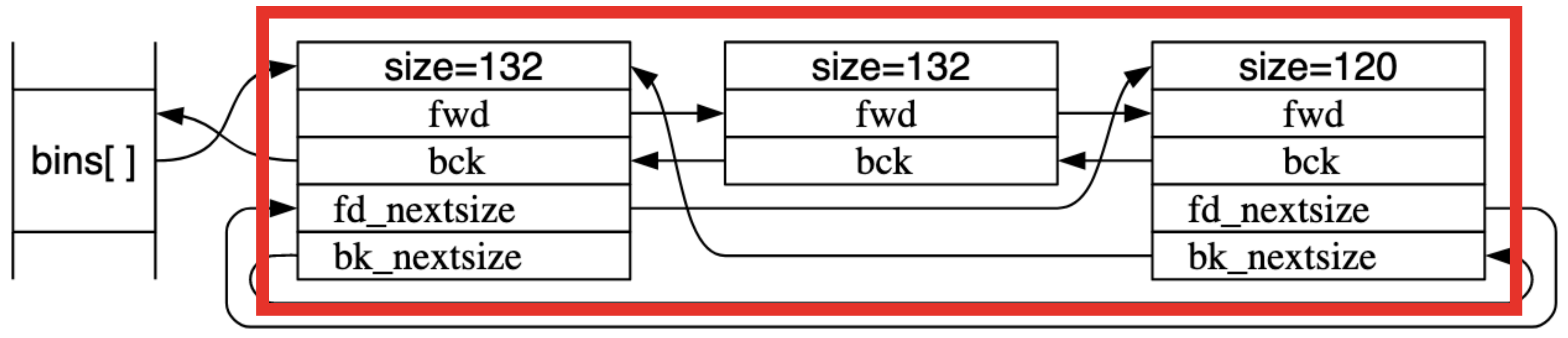


Issues

Mixing data and metadata
p = malloc()
p + some (negative) offset
points into arena and heap
meta data





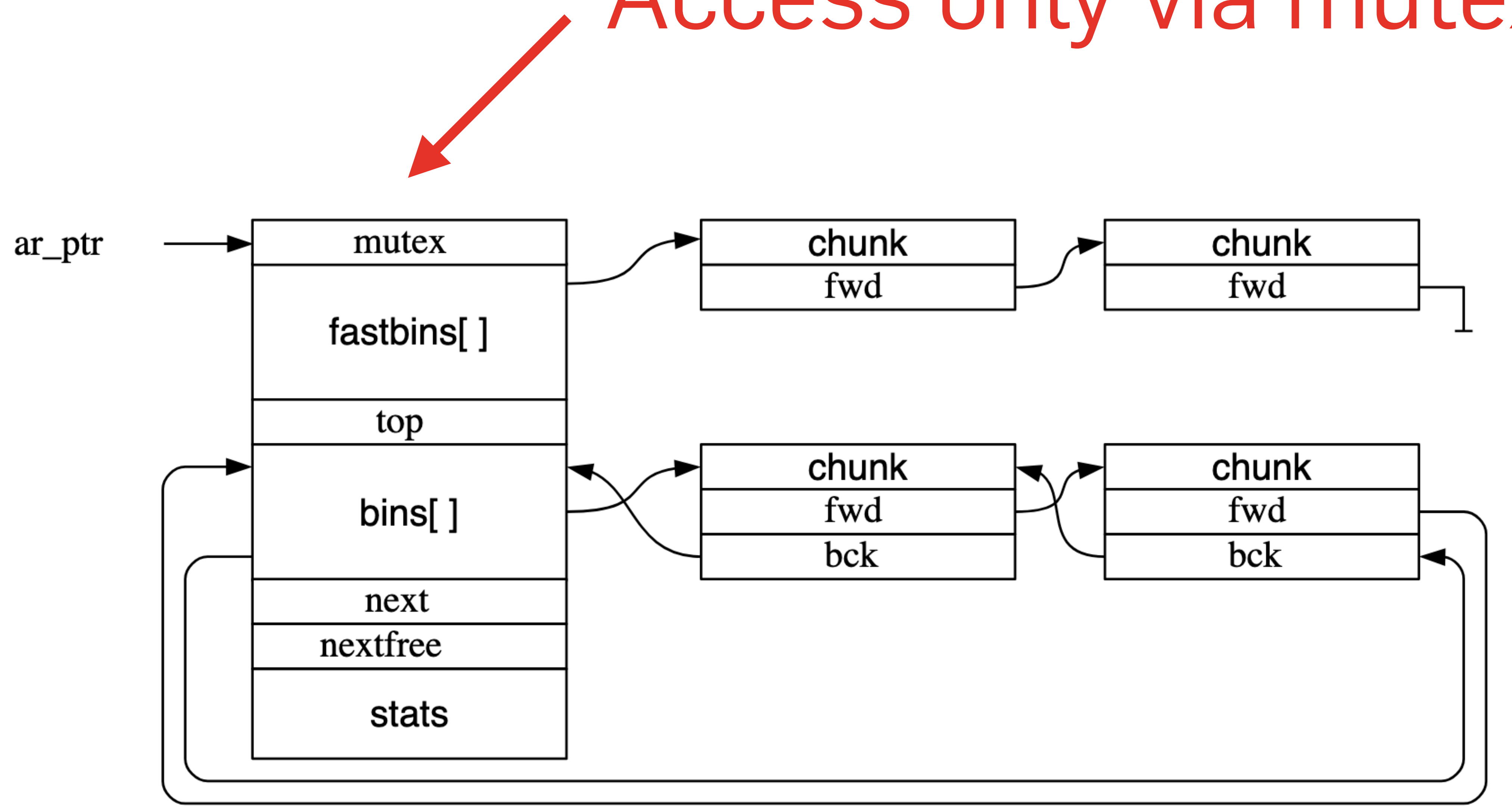


These are memory regions
the program might have free()'d.
If the program uses it after the free,
there is DMA metadata there!

tcache

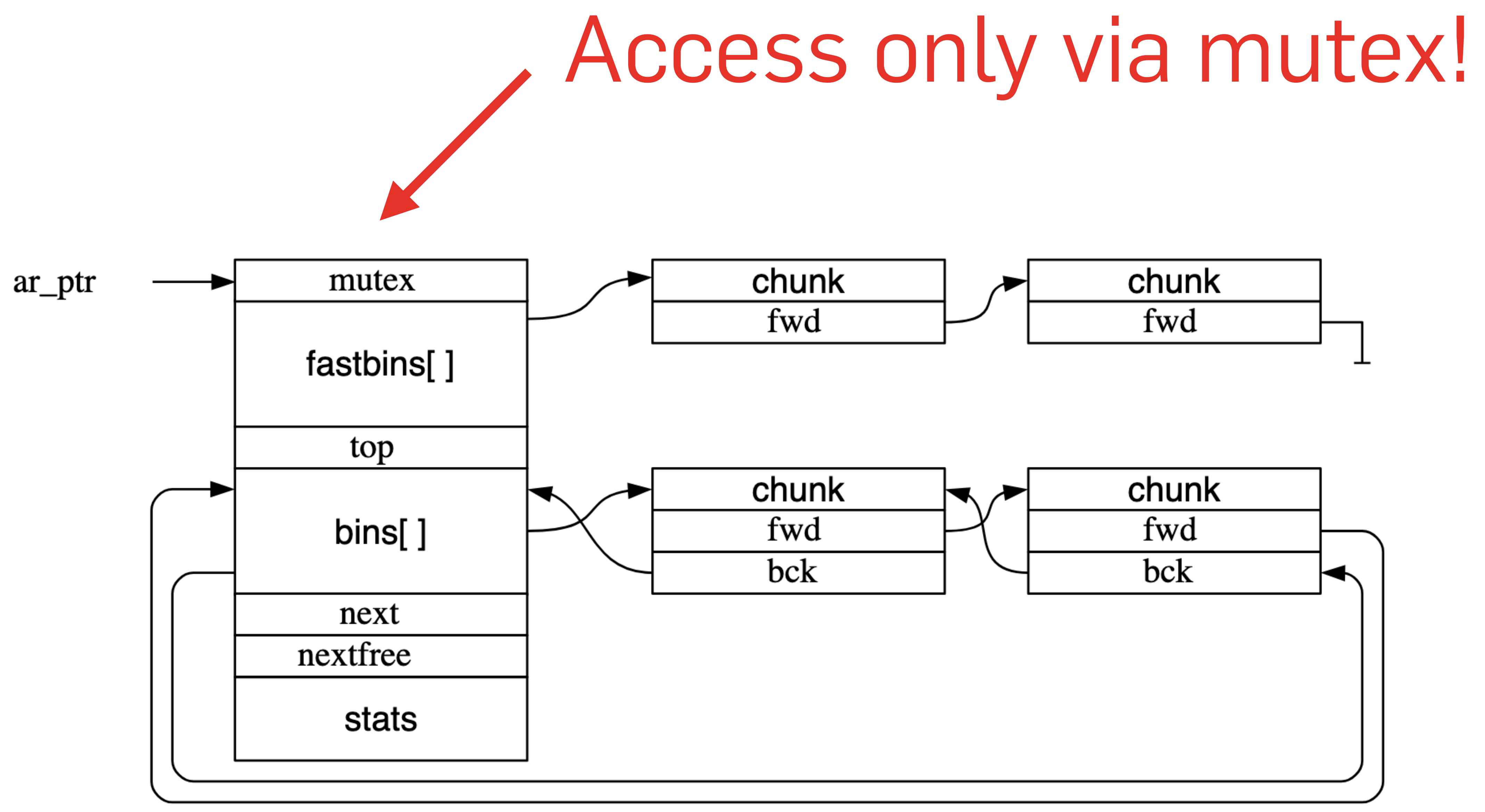
- Threads may use any arena for allocations
  - Can re-use existing arena, use an unused one, or create new one
  - Access is mutex'd = blocking
    - This is slow 🐢

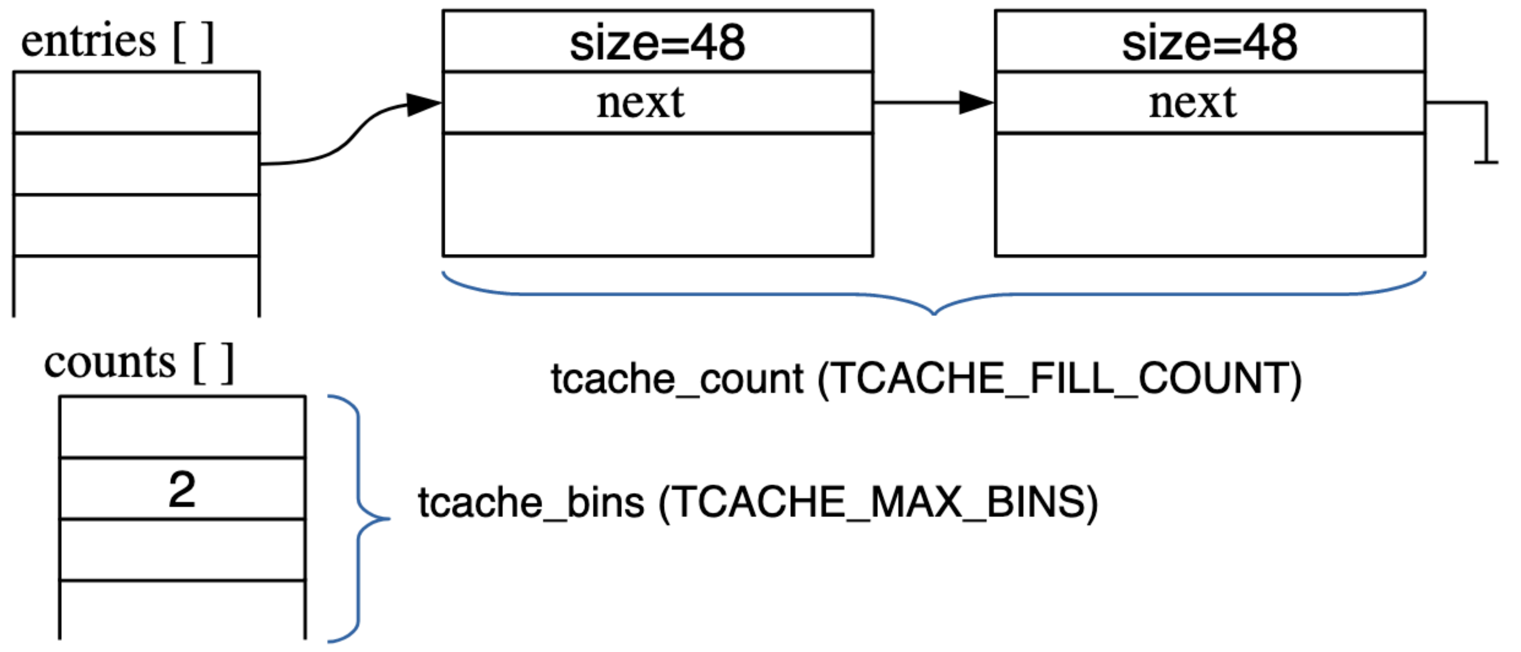
Access only via mutex!



tcache

- Threads may use any arena for allocations
  - Can re-use existing arena, use an unused one, or create new one
  - Access is mutex'd = blocking
    - This is slow 🐢
- Thread Local Caching (tcache)
  - Idea: Give each thread its own cache to speed up small allocations (which are common) without needing to lock arena for exclusive access
  - Own type of bins, similar to fastbins
  - Strict limit on how many entries (falls back to normal malloc if no more entries, does not pick a larger chunk to prevent fragmentation)





entries [ ]
size=48
next
size=48
next
tcache_count (TCACHE_FILL_COUNT)
counts [ ]
2
tcache_bins (TCACHE_MAX_BINS)

ptmalloc: malloc()

1. Can I use tcache (exact match only)?

ptmalloc: malloc()

2. Is the malloc() large enough for mmap()?
   Threshold often determined dynamically

ptmalloc: malloc()

3. Can I use a fastbin?
   If yes, also pre-fill tcache a bit, in case of more allocations of this size

ptmalloc: malloc()

4. Can I use a small bin?
   If yes, also pre-fill tcache bit

ptmalloc: malloc()

5. Ok, I need a large bin, might as well do some house cleaning.
   Move all fastbins to unsorted and merge contiguous chunks

ptmalloc: malloc()

6. Oh that's a lot of unsorted chunks, better start sorting them into small/large and merging them. Oh, there is one of the right size? Might as well use that one!
   We do this "on the go", we might find a right chunk before we sorted/merged all.

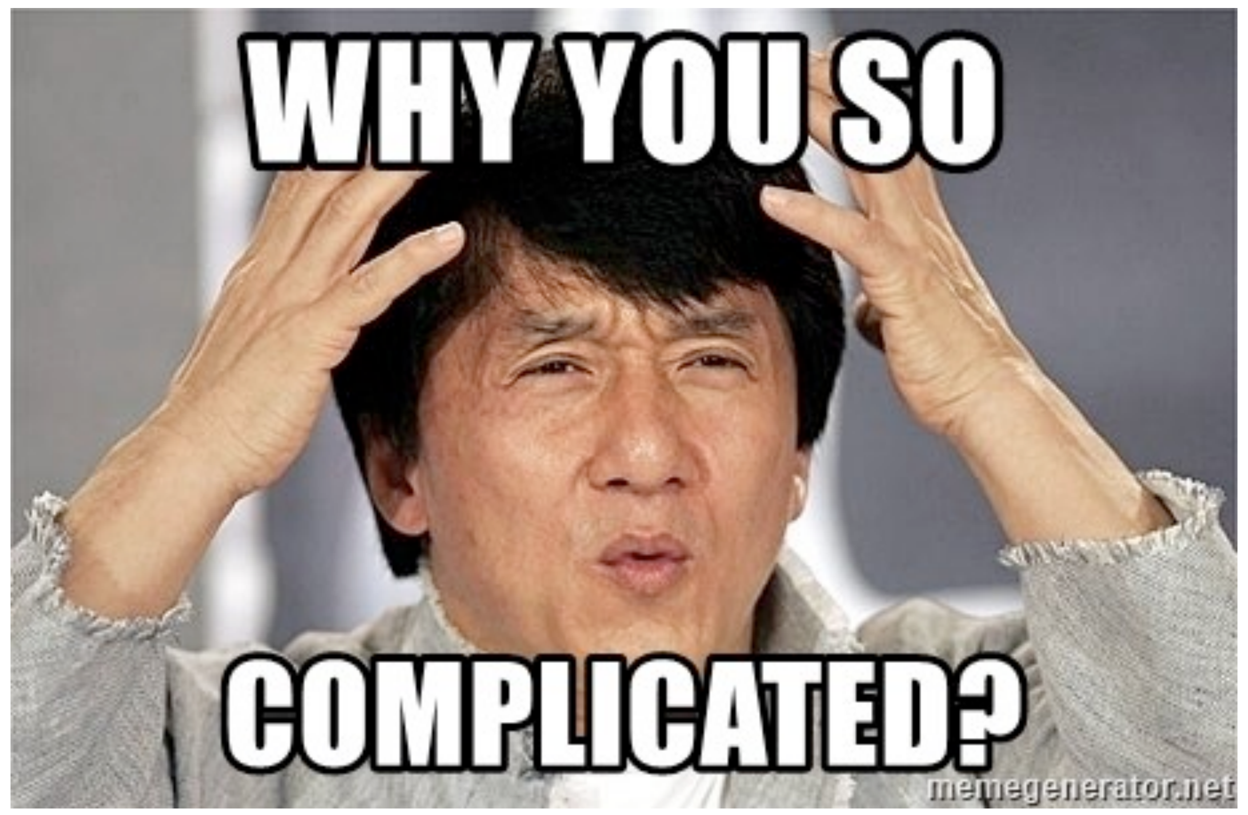
ptmalloc: malloc()

7. Nothing yet? Let's check the large bins. If there a large-enough chunk, split it into our allocation.

ptmalloc: malloc()

8. If the allocation fit a small bin, consolidate fastbins and perform steps 6 and 7.

ptmalloc: malloc()



9. If we couldn't fit the allocation anywhere, use the "top chunk" (possibly making it larger before).

ptmalloc: free()

1. If there is room the in tcache, put the chunk there.

ptmalloc: free()

2. If the chunk fits a fastbin, put it there.

ptmalloc: free()

3. If the chunk was mmap'd(), munmap() it.

ptmalloc: free()

4. Is the chunk next to another free chunk? Merge them.

ptmalloc: free()

5. Place the chunk in the unsorted list.

ptmalloc: free()

6. If the chunk is large, consider returning the top chunk back to the operating system.
   Note: This might not actually happen on the call to free(), but might be delayed to some other calls for performance reasons.

ptmalloc(): Heap Corruptions

- ptmalloc mixes data and control information
- ptmalloc uses some memory alignments
    - Chunk sizes are multiple of 8 bytes
      (so the least significant 3 bits can be used for the flags AMP)
    - Heaps are always aligned at power of two addresses
      (so you can compute the heap address based on the chunk address)
    - ptmalloc checks with heuristics that pointers satisfy these alignments
        - Your program will fail with a heap corruption message if you overwrite heap metadata with obviously invalid pointers, but they are heuristics and can be easily fooled

tcache: Example

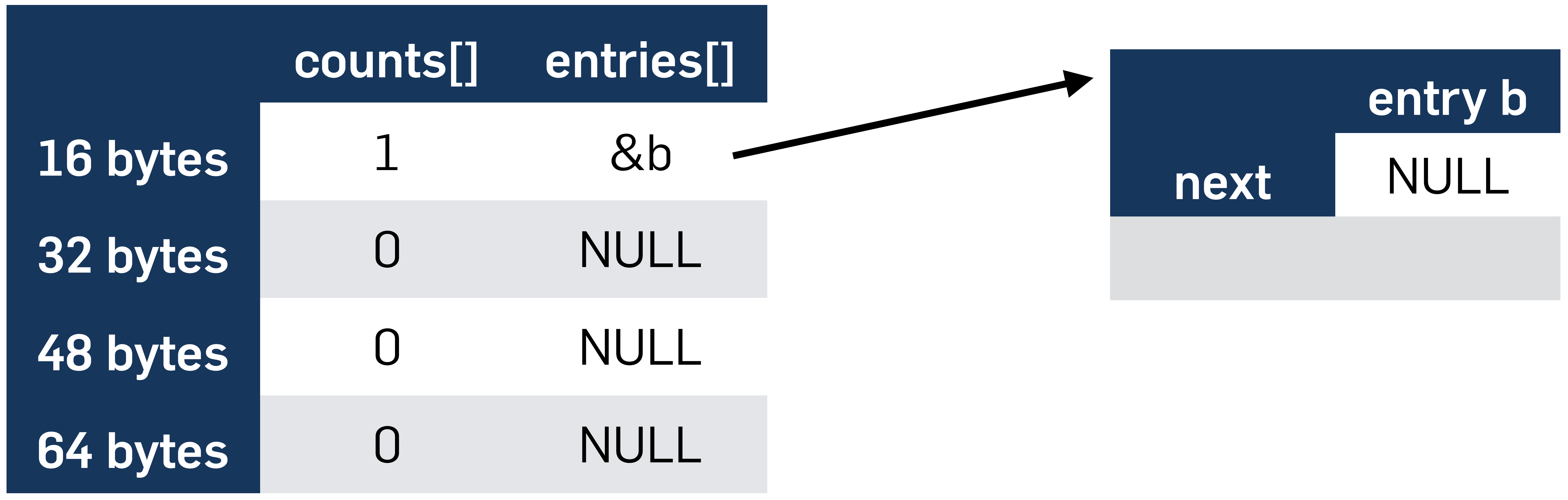
	counts[]	entries[]
16 bytes	0	NULL
32 bytes	0	NULL
48 bytes	0	NULL
64 bytes	0	NULL

tcache: Example

	counts[]	entries[]
16 bytes	0	NULL
32 bytes	0	NULL
48 bytes	0	NULL
64 bytes	0	NULL

```
a = malloc(16);
b = malloc(16);
c = malloc(32);
d = malloc(48);
e = malloc(32);
f = malloc(32);
x = malloc(64);
y = malloc(64);
z = malloc(64);
```

tcache: Example

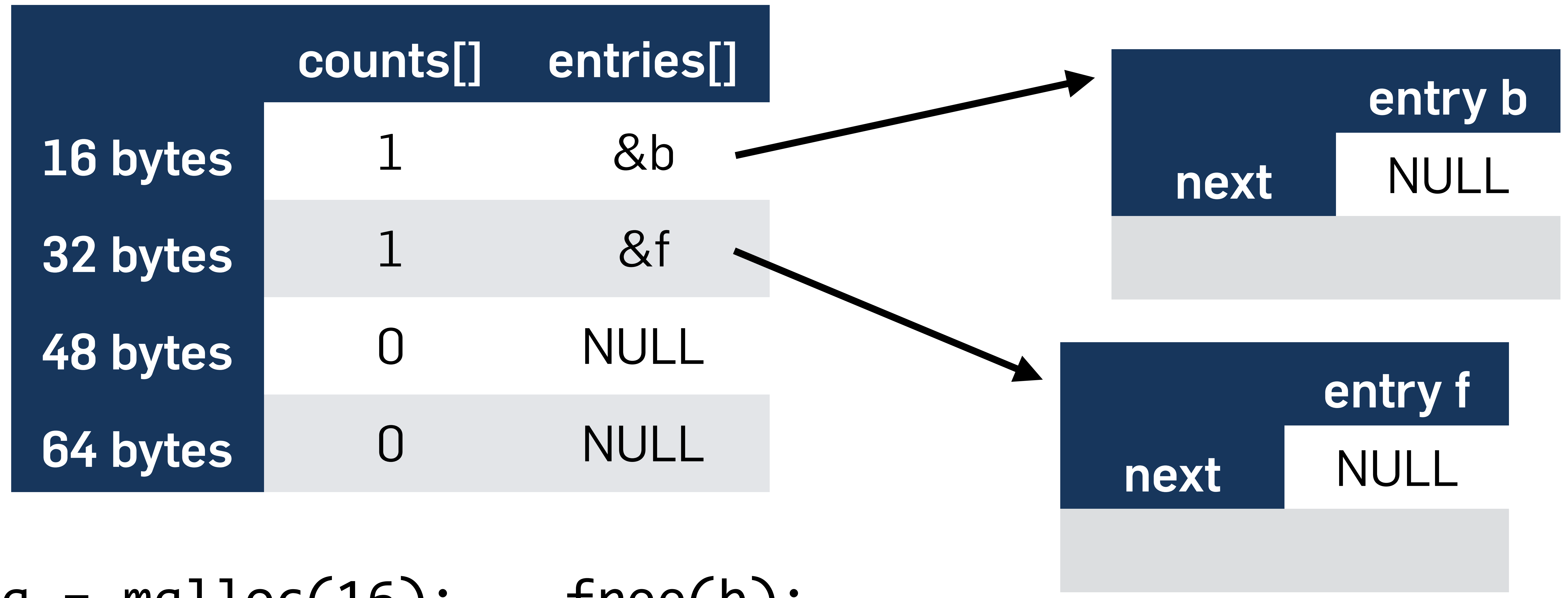


	counts[]	entries[]
16 bytes	1	&b
32 bytes	0	NULL
48 bytes	0	NULL
64 bytes	0	NULL

	entry b
next	NULL

```
a = malloc(16);    free(b);
b = malloc(16);
c = malloc(32);
d = malloc(48);
e = malloc(32);
f = malloc(32);
x = malloc(64);
y = malloc(64);
z = malloc(64);
```

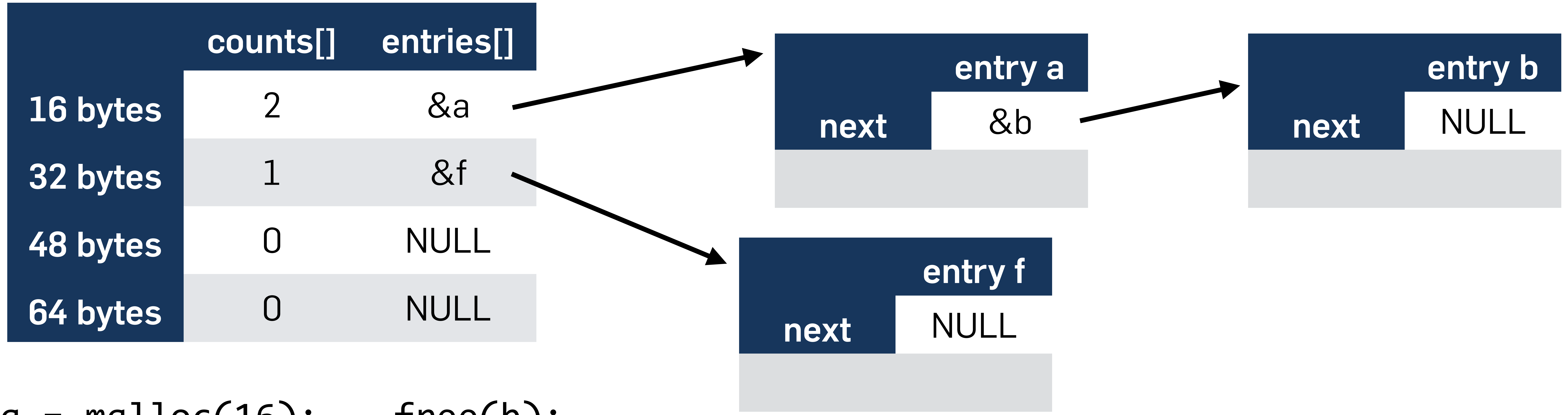
tcache: Example



	counts[]	entries[]
16 bytes	1	&b
32 bytes	1	&f
48 bytes	0	NULL
64 bytes	0	NULL

```
a = malloc(16);    free(b);
b = malloc(16);    free(f);
c = malloc(32);
d = malloc(48);
e = malloc(32);
f = malloc(32);
x = malloc(64);
y = malloc(64);
z = malloc(64);
```

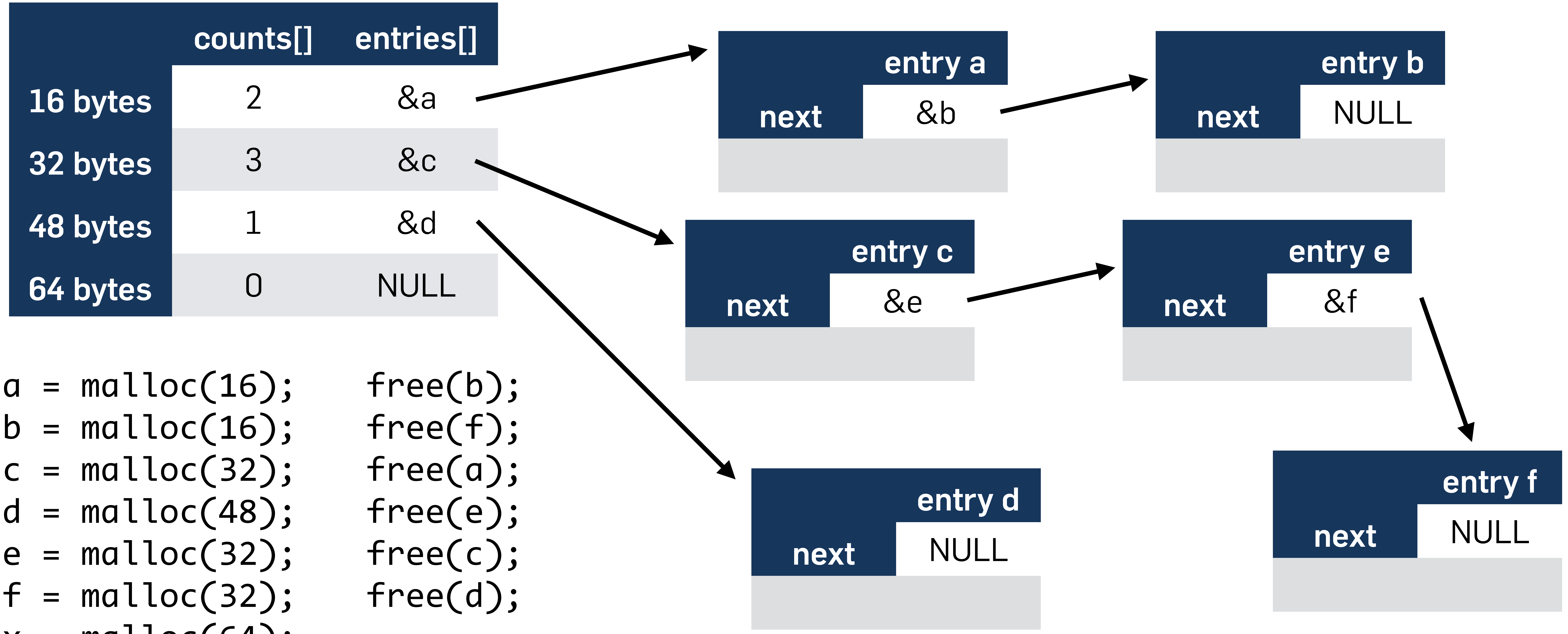
tcache: Example



```
a = malloc(16);     free(b);
b = malloc(16);     free(f);
c = malloc(32);     free(a);
d = malloc(48);
e = malloc(32);
f = malloc(32);
x = malloc(64);
y = malloc(64);
z = malloc(64);
```

For ptmalloc(), you add new free's to the head of the linked list.

tcache: Example

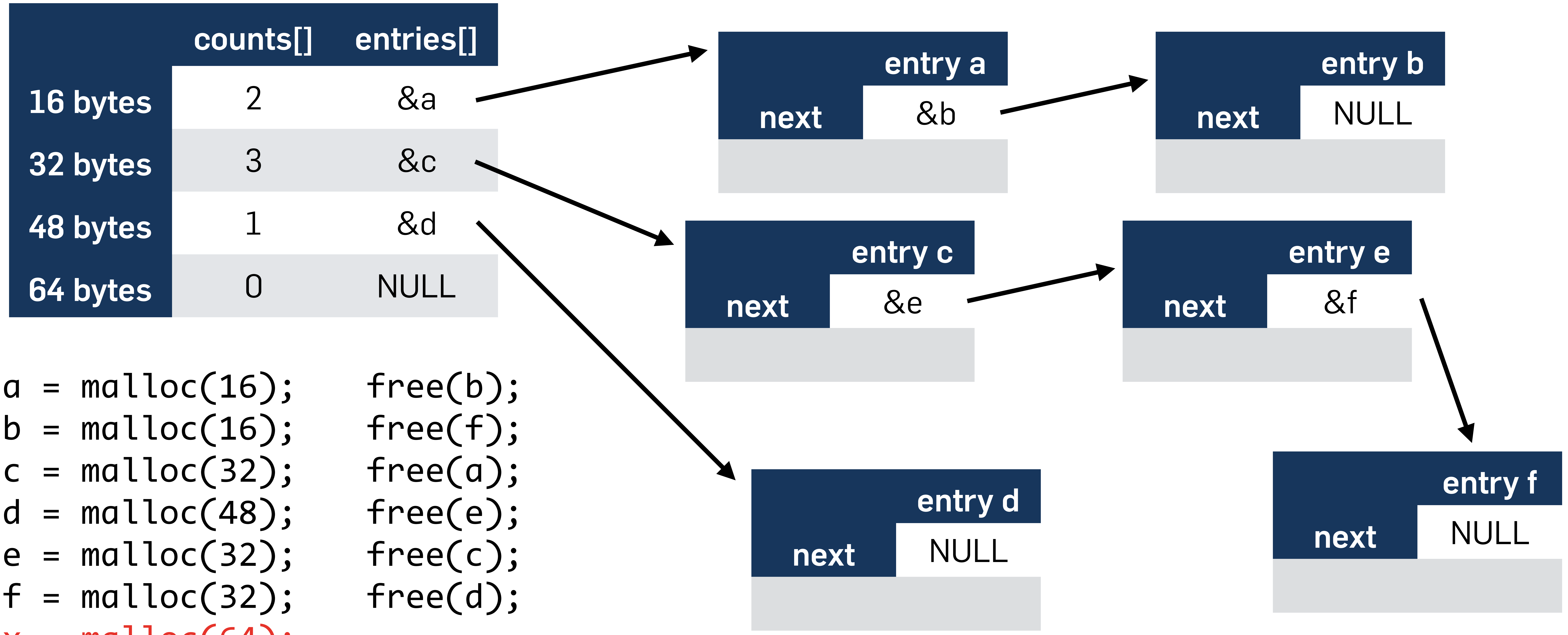


	counts[]	entries[]
16 bytes	2	&a
32 bytes	3	&c
48 bytes	1	&d
64 bytes	0	NULL

```
a = malloc(16);    free(b);
b = malloc(16);    free(f);
c = malloc(32);    free(a);
d = malloc(48);    free(e);
e = malloc(32);    free(c);
f = malloc(32);    free(d);
x = malloc(64);
y = malloc(64);
z = malloc(64);
```

For ptmalloc(), you add new free's to the head of the linked list.

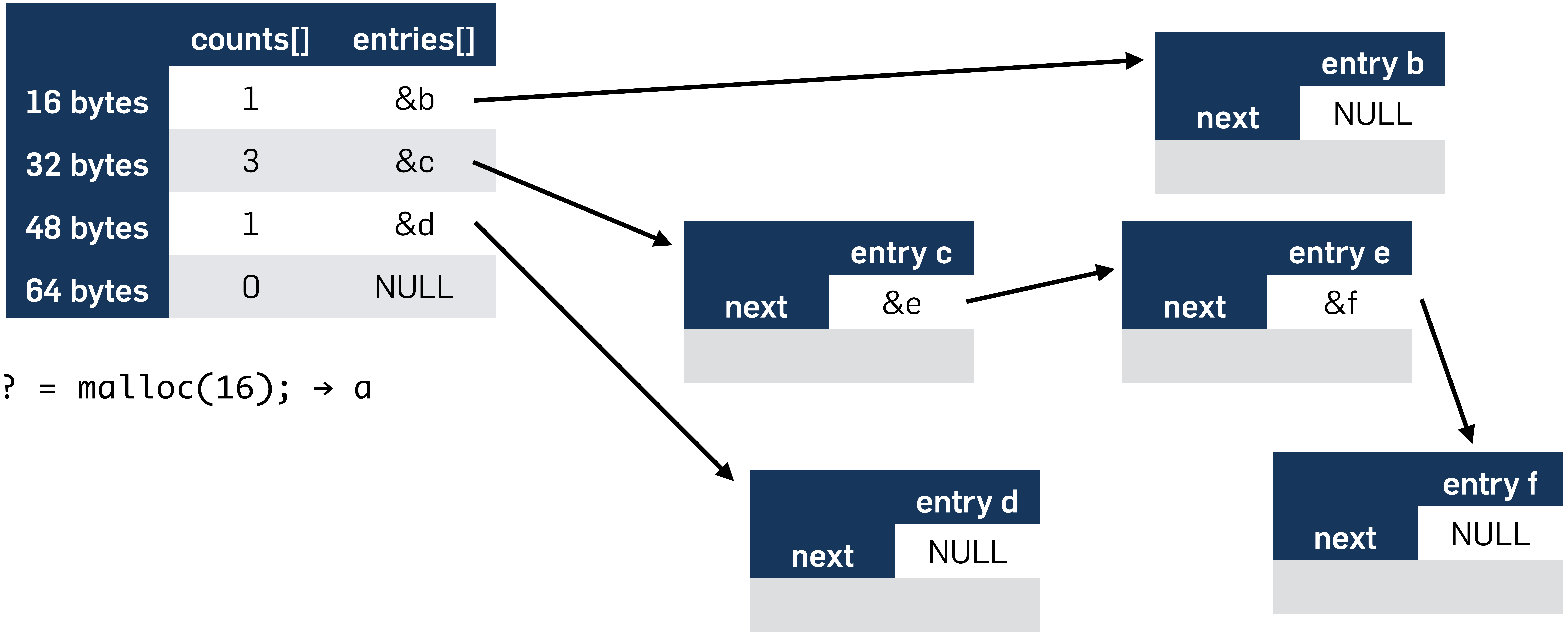
tcache: Example



```
a = malloc(16);     free(b);
b = malloc(16);     free(f);
c = malloc(32);     free(a);
d = malloc(48);     free(e);
e = malloc(32);     free(c);
f = malloc(32);     free(d);
x = malloc(64);
y = malloc(64);
z = malloc(64);
```

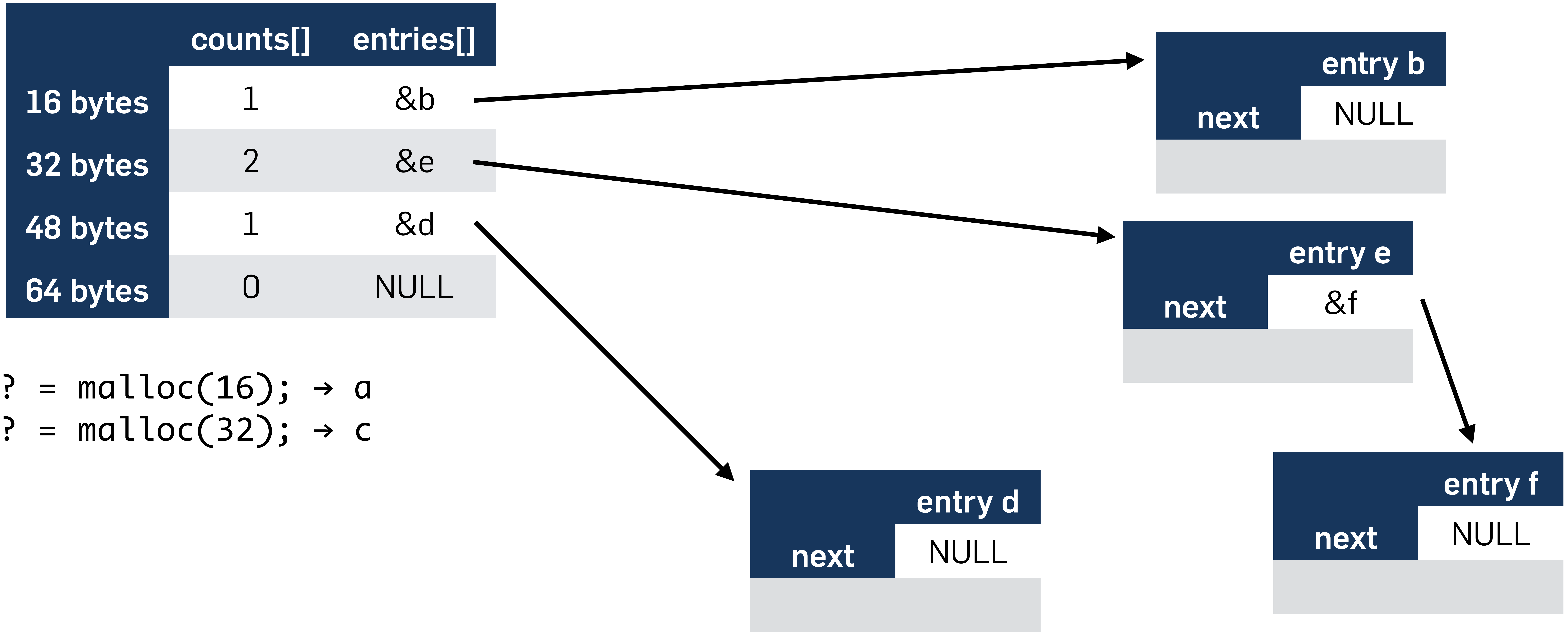
Memory that is not free()'d will not be in the tcache.

tcache: Example



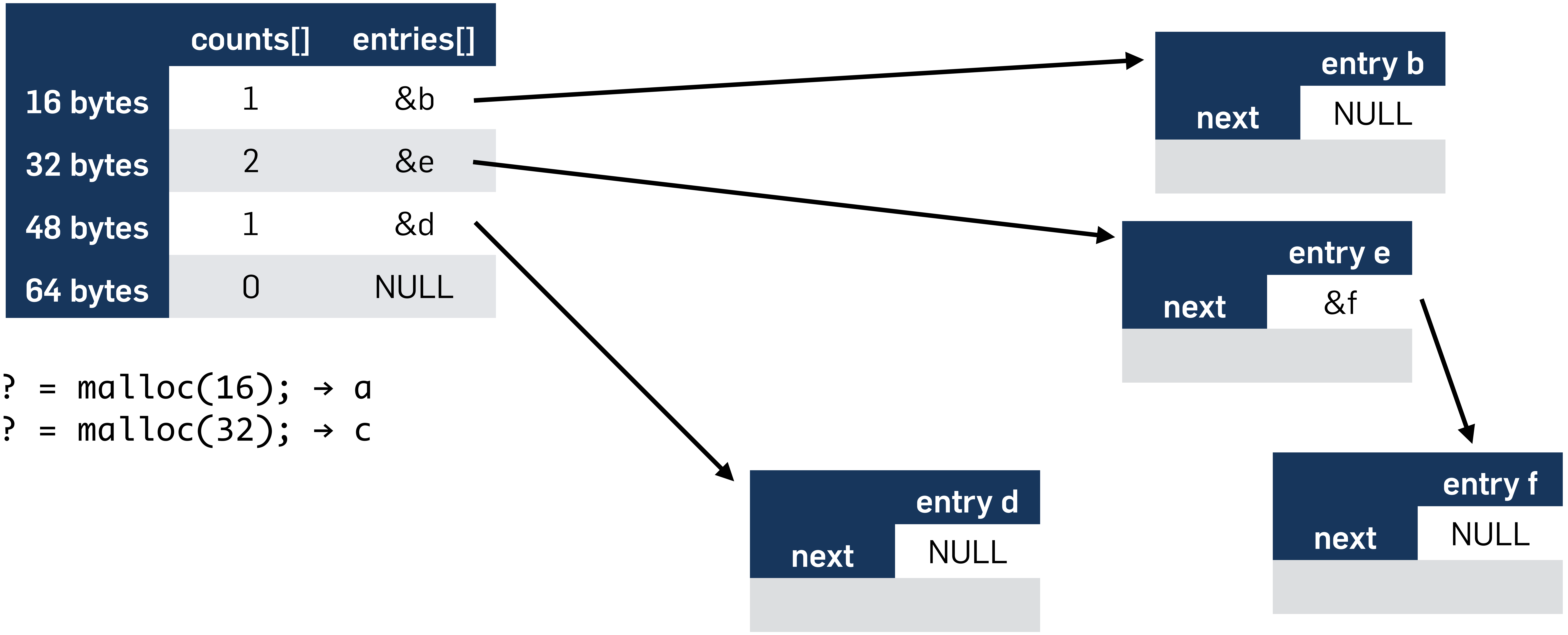
```
? = malloc(16); → a
```

tcache: Example



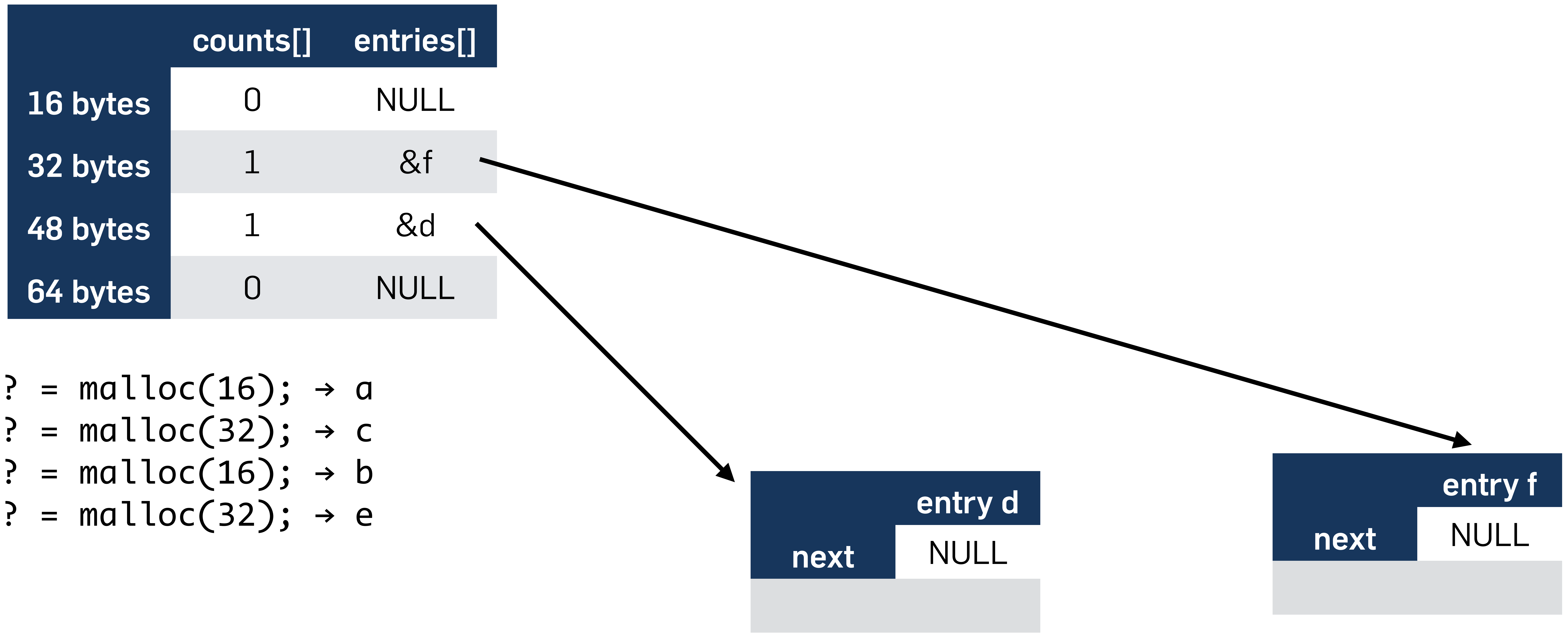
```
? = malloc(16); → a
? = malloc(32); → c
```

tcache: Example



```
? = malloc(16); → a
? = malloc(32); → c
```

tcache: Example



	counts[]	entries[]
16 bytes	0	NULL
32 bytes	1	&f
48 bytes	1	&d
64 bytes	0	NULL

```
? = malloc(16); → a
? = malloc(32); → c
? = malloc(16); → b
? = malloc(32); → e
```

tcache: Example

	counts[]	entries[]
16 bytes	0	NULL
32 bytes	0	NULL
48 bytes	0	NULL
64 bytes	0	NULL

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

Take the head of the list and use it for the allocation (last in, first out; LIFO).

tcache: Implementation Details

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

- On allocation, tcache reuses the head of the list
  - It does **NOT**
    - Clear sensitive pointers that might are stored in the tcache (*next is not cleared, only *key is cleared)
    - Check if *next actually makes sense
- On free, tcache checks if *key has been set (and points to the owning tcache structure in memory) to prevent double free()
  - And sets *key if it is not a double free
- glibc 2.32 protects/mangles pointers to prevent plain overwrites

Heap: Double Free

- Memory that we allocated via the DMA and free()'d is not actually returned to the operating system. It remains valid.
  - We can still read and write to it, and we might free() it again (this does violate the assumptions of the DMA)
  - New versions of glibc/ptmalloc introduced the check for *key

Heap: Double Free

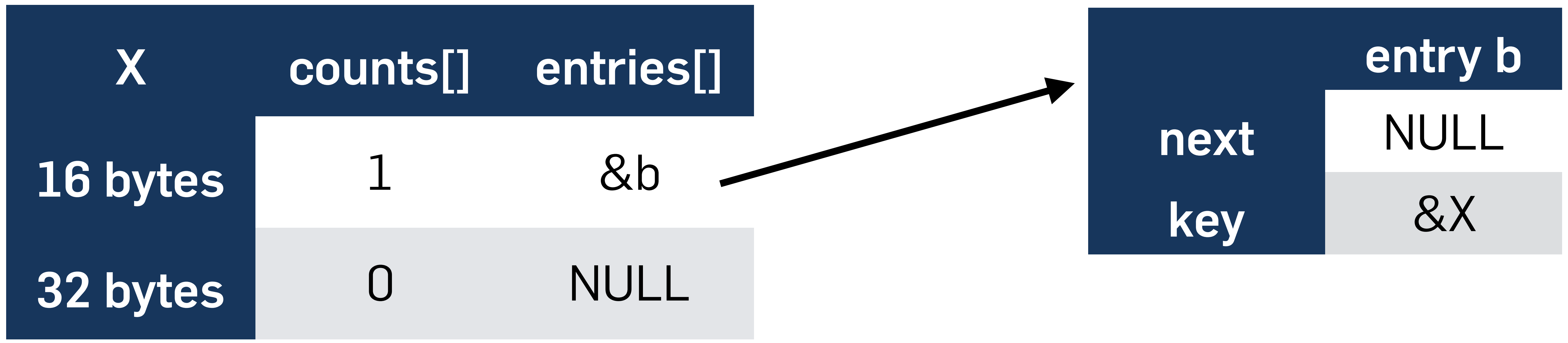
- Memory that we allocated via the DMA and free()'d is not actually returned to the operating system. It remains valid.
  - We can still read and write to it, and we might free() it again (this does violate the assumptions of the DMA)
  - New versions of glibc/ptmalloc introduced the check for *key

But, we could just overwrite *key after free()'ing our allocation?!

Heap: Double Free

- Memory that we allocated via the DMA and free()'d is not actually returned to the operating system. It remains valid.
  - We can still read and write to it, and we might free() it again (this does violate the assumptions of the DMA)
  - New versions of glibc/ptmalloc introduced the check for *key

But, we could just overwrite *key after free()'ing our allocation?!



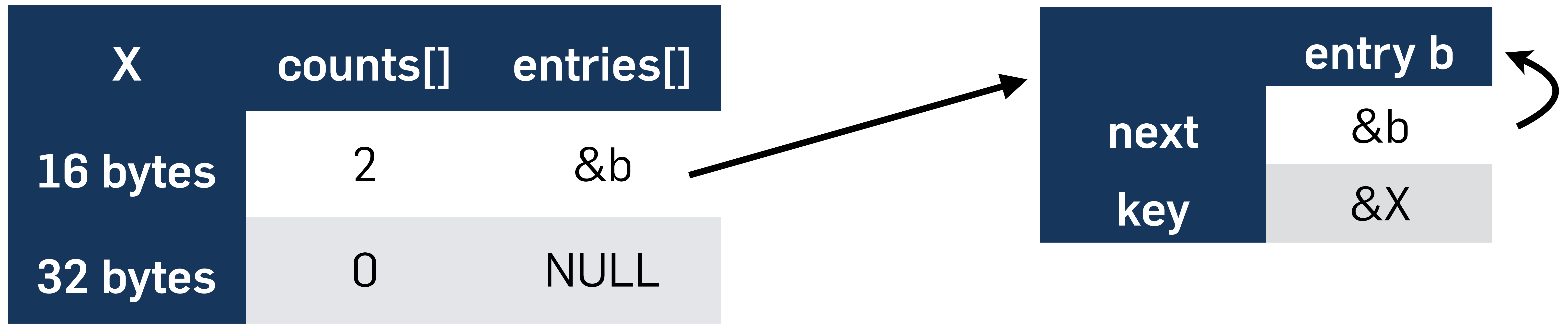
X	counts[]	entries[]
16 bytes	1	&b
32 bytes	0	NULL

	entry b
next	NULL
key	&X

Heap: Double Free

- Memory that we allocated via the DMA and free()'d is not actually returned to the operating system. It remains valid.
  - We can still read and write to it, and we might free() it again (this does violate the assumptions of the DMA)
  - New versions of glibc/ptmalloc introduced the check for *key

But, we could just overwrite *key after free()'ing our allocation?!

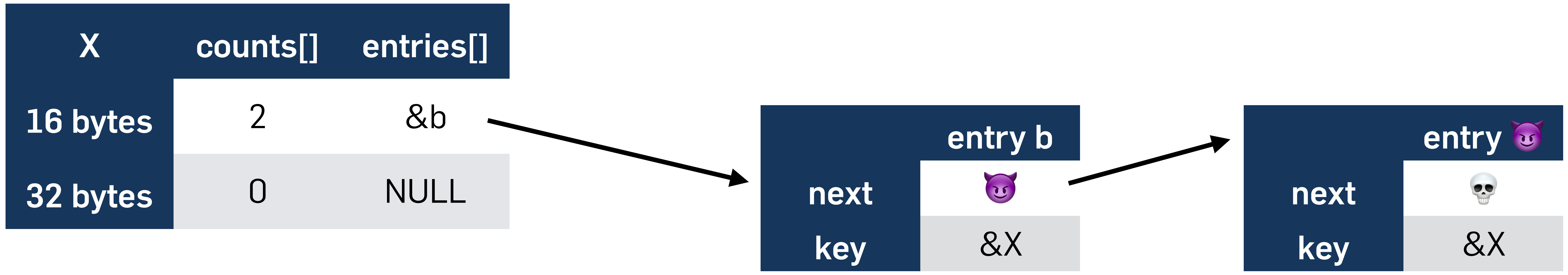


X	counts[]	entries[]
16 bytes	2	&b
32 bytes	0	NULL

	entry b
next	&b
key	&X

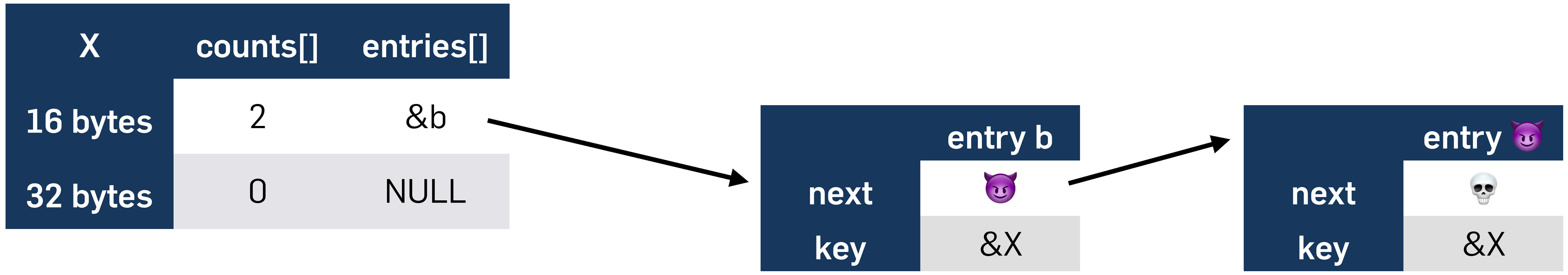
Heap: Poisoning *next

- What if we overwrite *next instead?



Heap: Poisoning *next

- What if we overwrite *next instead?



We can now control where the second malloc() of this size will be at!

tcache: Summary

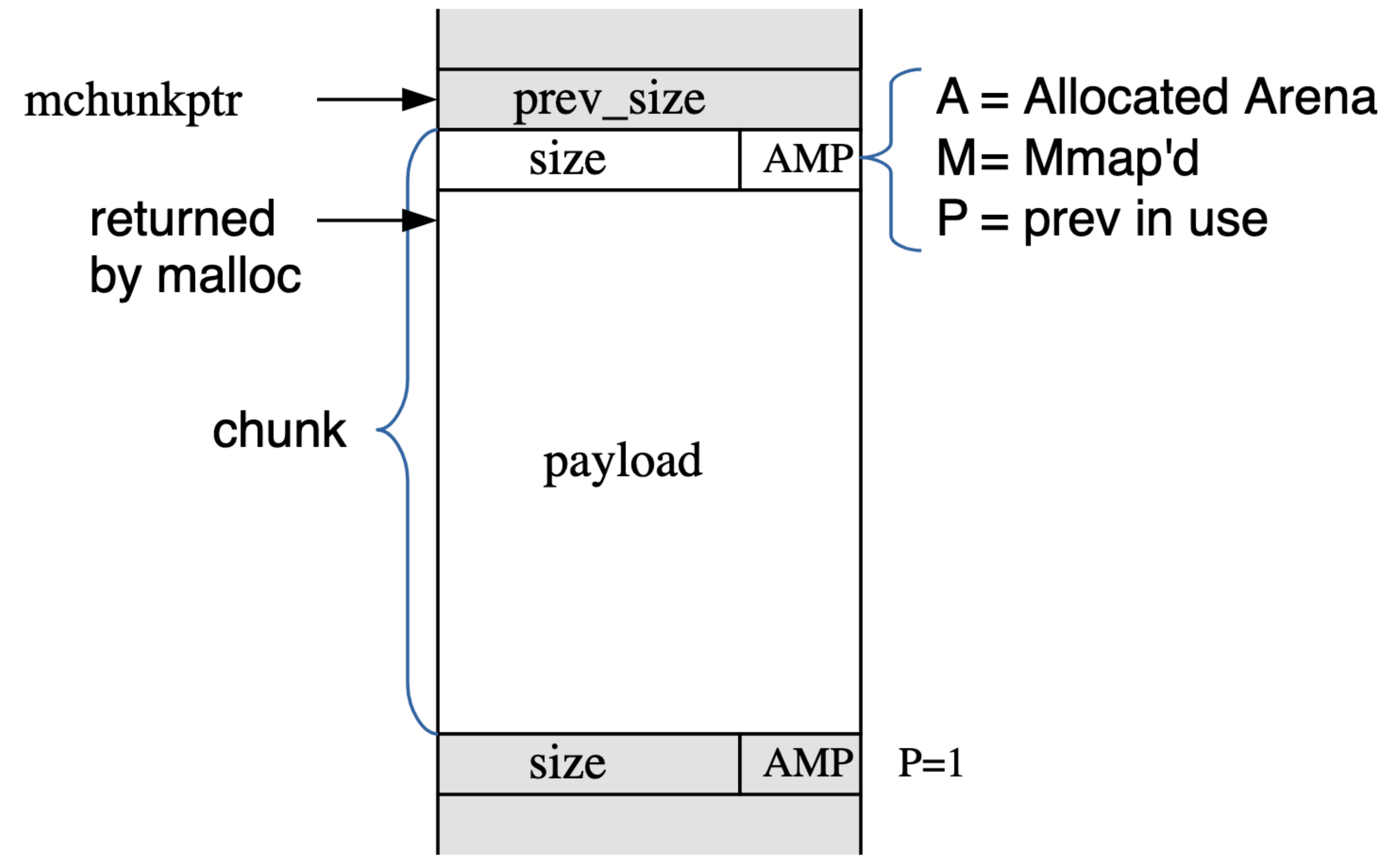
- tcache is a caching layer in ptmalloc for small allocations
  - Size of bins depends on architecture, version, etc.
- Has basically no security checks
- Can be (easily) tricked to behave "weirdly" if memory is being used after free
  - Reason: Memory is not returned to the operating system, but kept in the application and even reused for metadata!
- Basic idea of what we saw for tcache also works for other glibc bins (fastbins etc.) and other DMAs

Examples

- tcache_1: free → free
- tcache_2: free → free
- tcache_3: free → write → free
- tcache_4: free → write → free → malloc

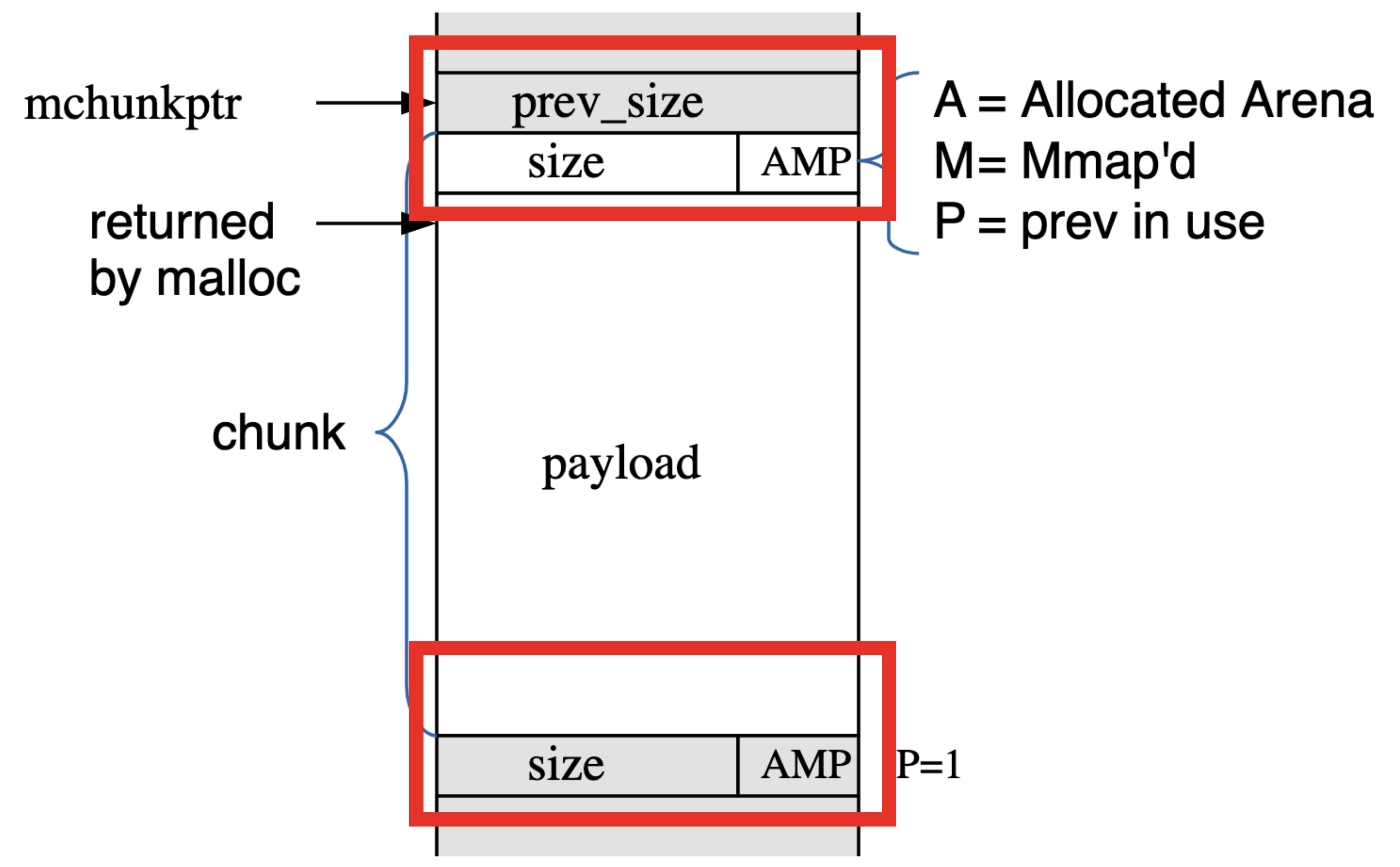
What About the Other Metadata?

- Recall how ptmalloc chunks look like



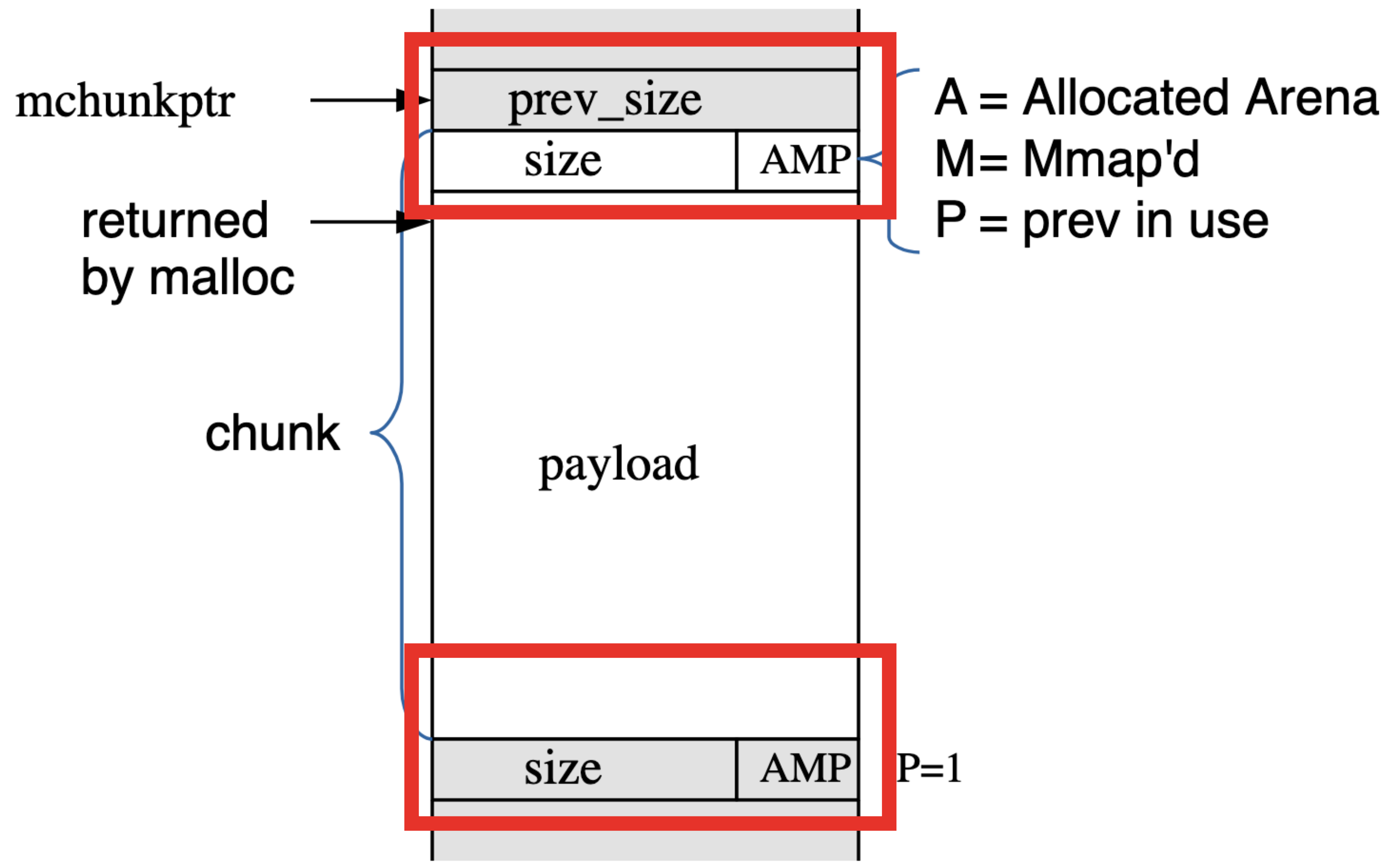
What About the Other Metadata?

- Recall how ptmalloc chunks look like



What About the Other Metadata?

- Recall how ptmalloc chunks look like

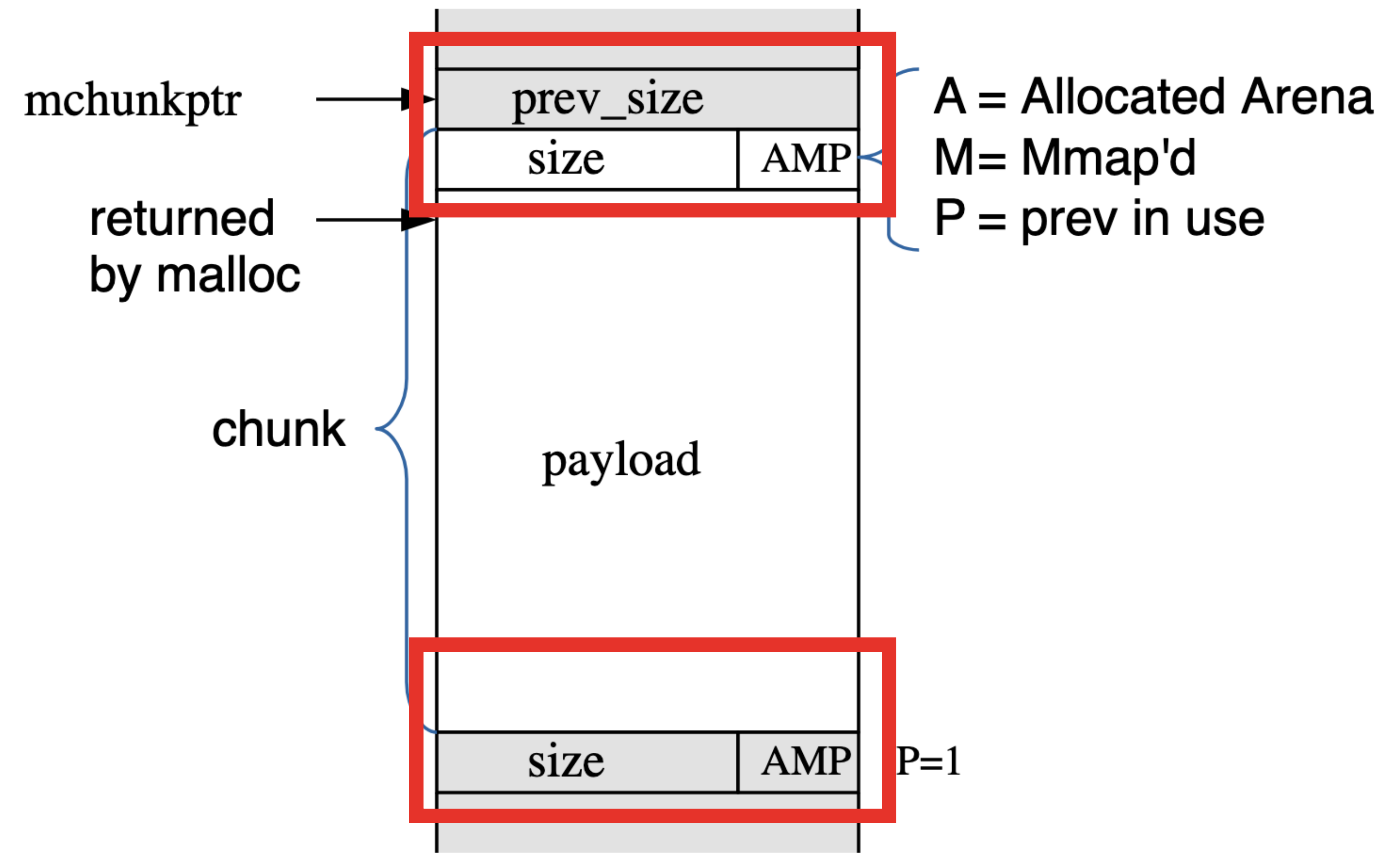


Minimum size:

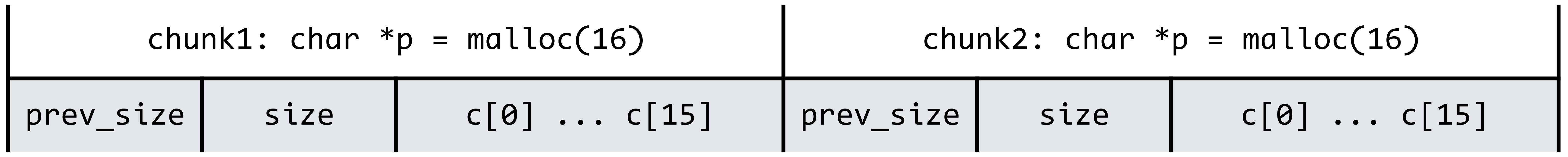
chunk1: char *p = malloc(16)			chunk2: char *p = malloc(16)		
prev_size	size	c[0] ... c[15]	prev_size	size	c[0] ... c[15]

What About the Other Metadata?

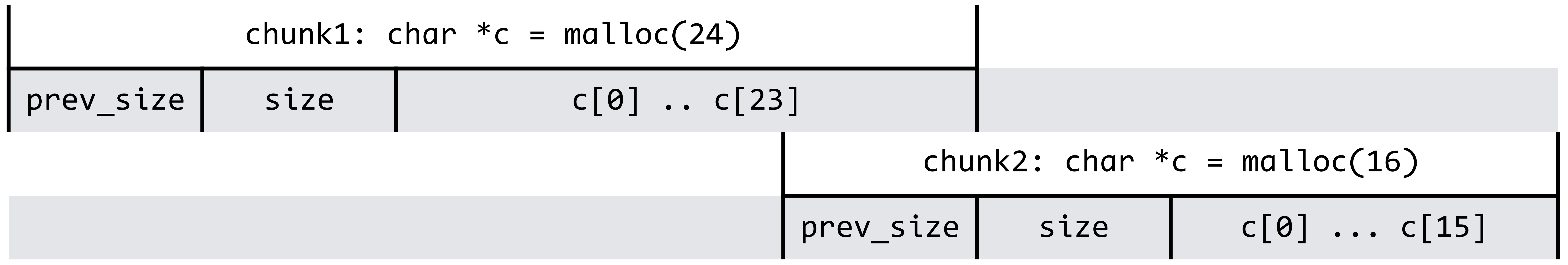
- Recall how ptmalloc chunks look like



Minimum size:

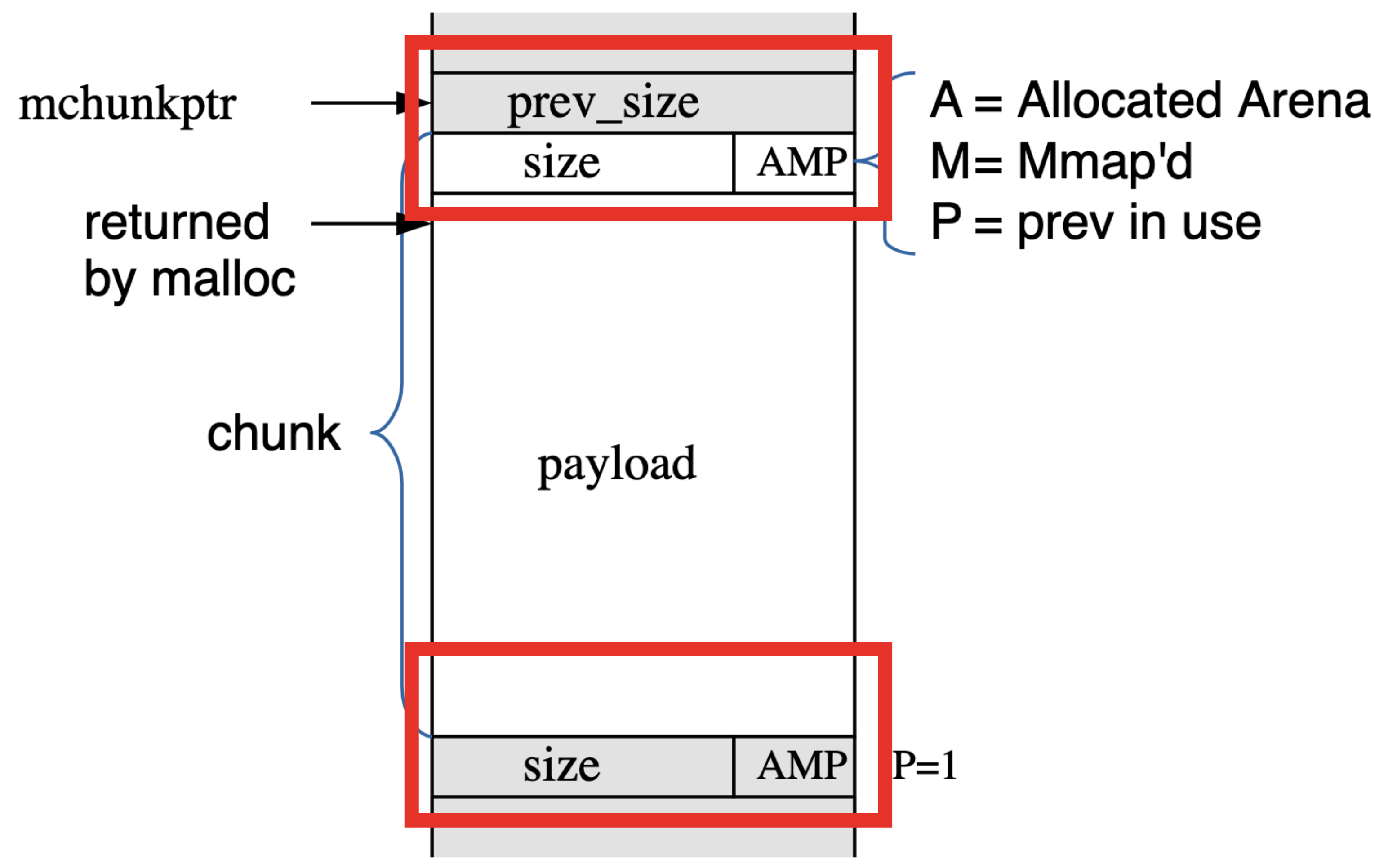


Larger size:

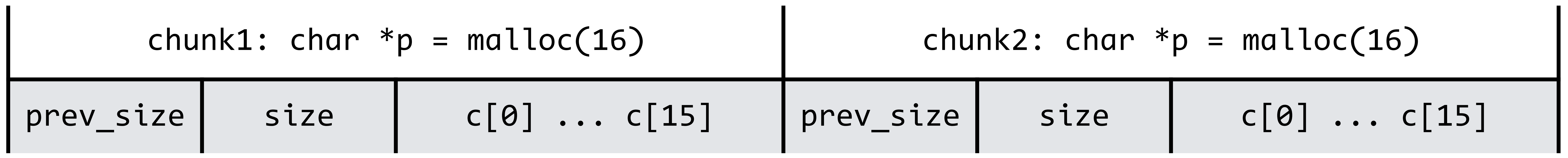


What About the Other Metadata?

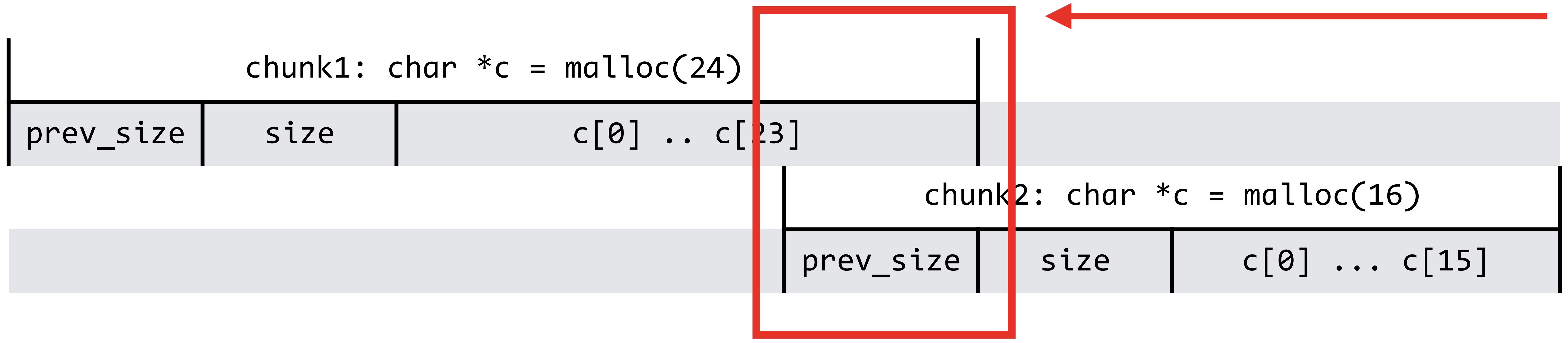
- Recall how ptmalloc chunks look like



Minimum size:



Larger size:



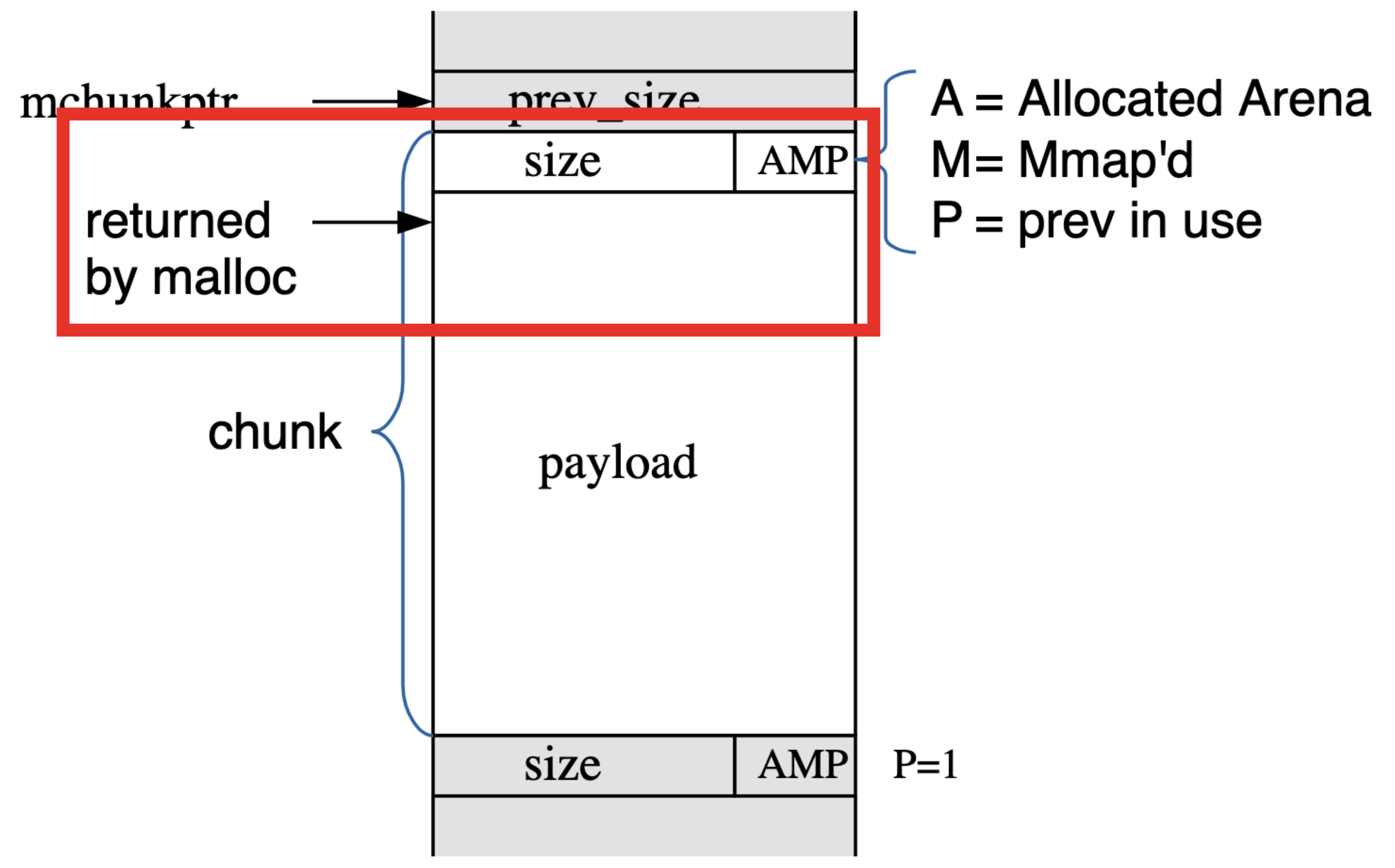
prev_size is actually part of of the memory given to the application as part of the previous chunk

Heap Attacks

- Possible because of performance optimizations and violating the DMAs assumptions of how the program should be using it
- Again leads to a weird machine that you can program
  - Primary primitives
    - How the DMA works (how malloc() and free() work internally)
    - How the program uses malloc() and free(), and whether this can violate the program's <u>intended</u> state machine
- Allocators differ
  - Bins, tcache etc. are specific to ptmalloc (glibc)
  - Different versions behave differently (and can be misused differently)
  - Some do not have inline metadata (e.g., jemalloc)
  - Windows 10 has two heaps (Segment and NT Heap)

Heap Attacks: House of Spirit

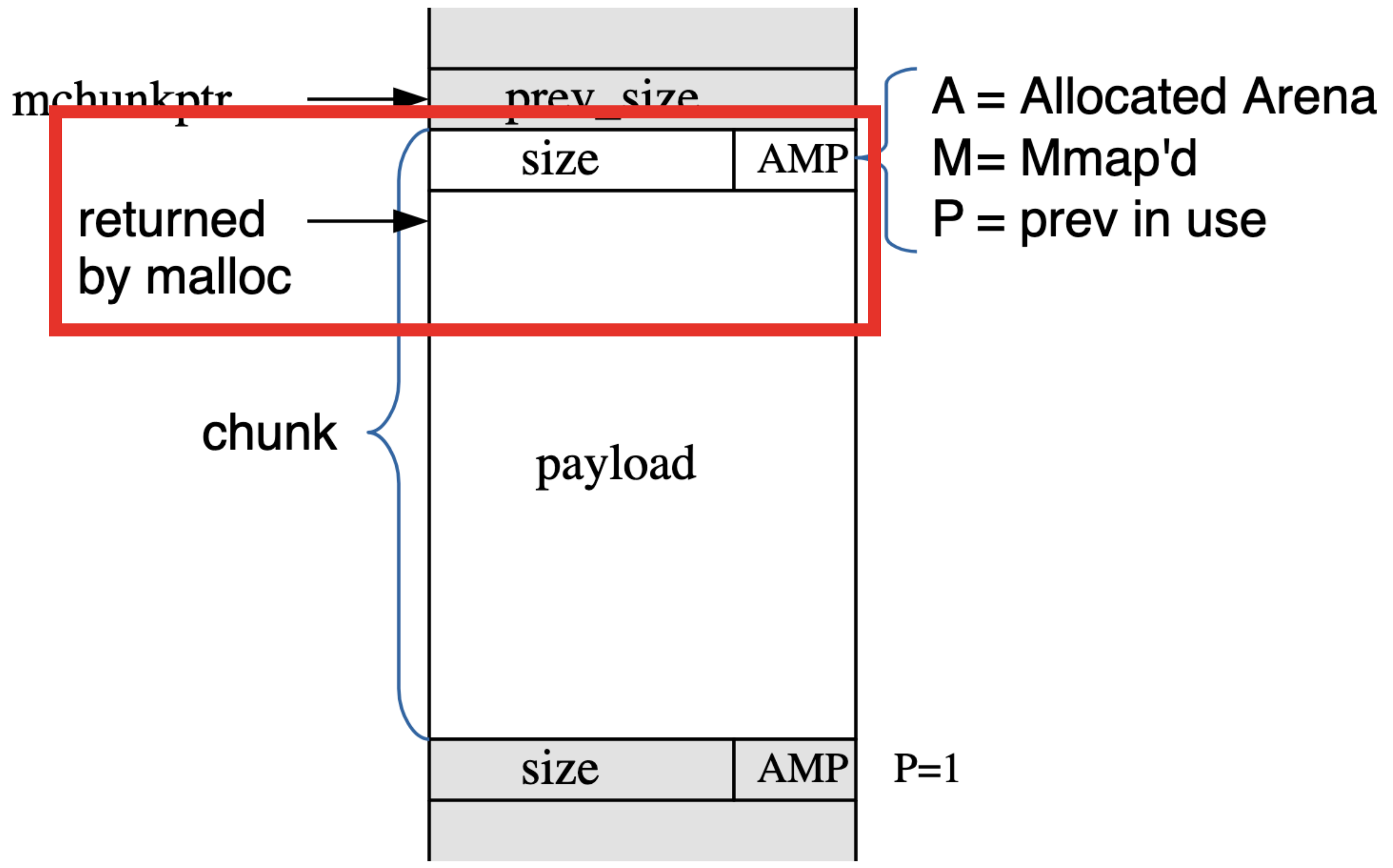
- ptmalloc does not actually track what memory is allocated (only what is free)
- free() does not do a lot of checks



© Daniel Shaw

Heap Attacks: House of Spirit

- ptmalloc does not actually track what memory is allocated (only what is free)
- free() does not do a lot of checks

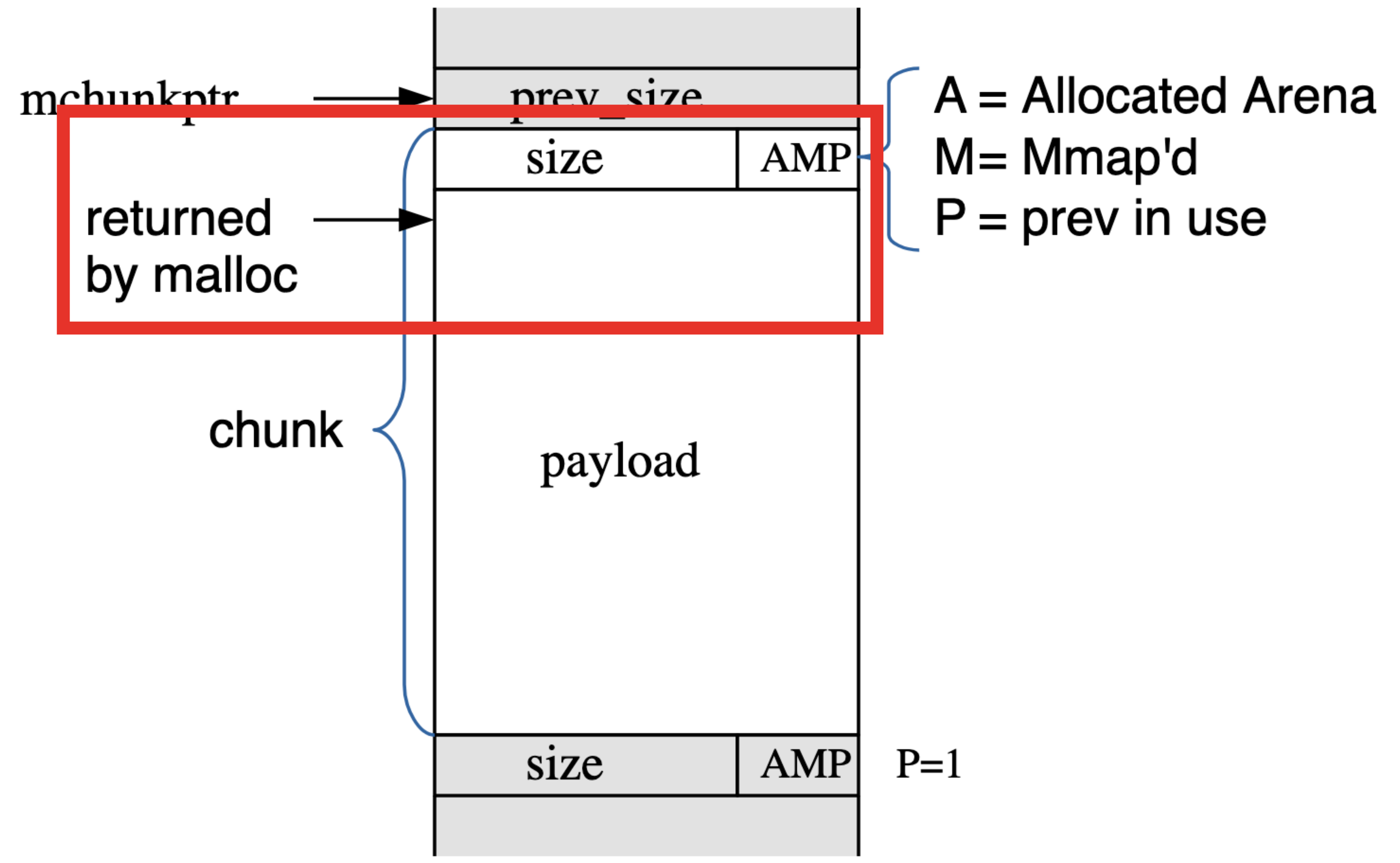


Can we just forge some chunk in memory, that looks like a chunk, and free() it?

© Daniel Shaw

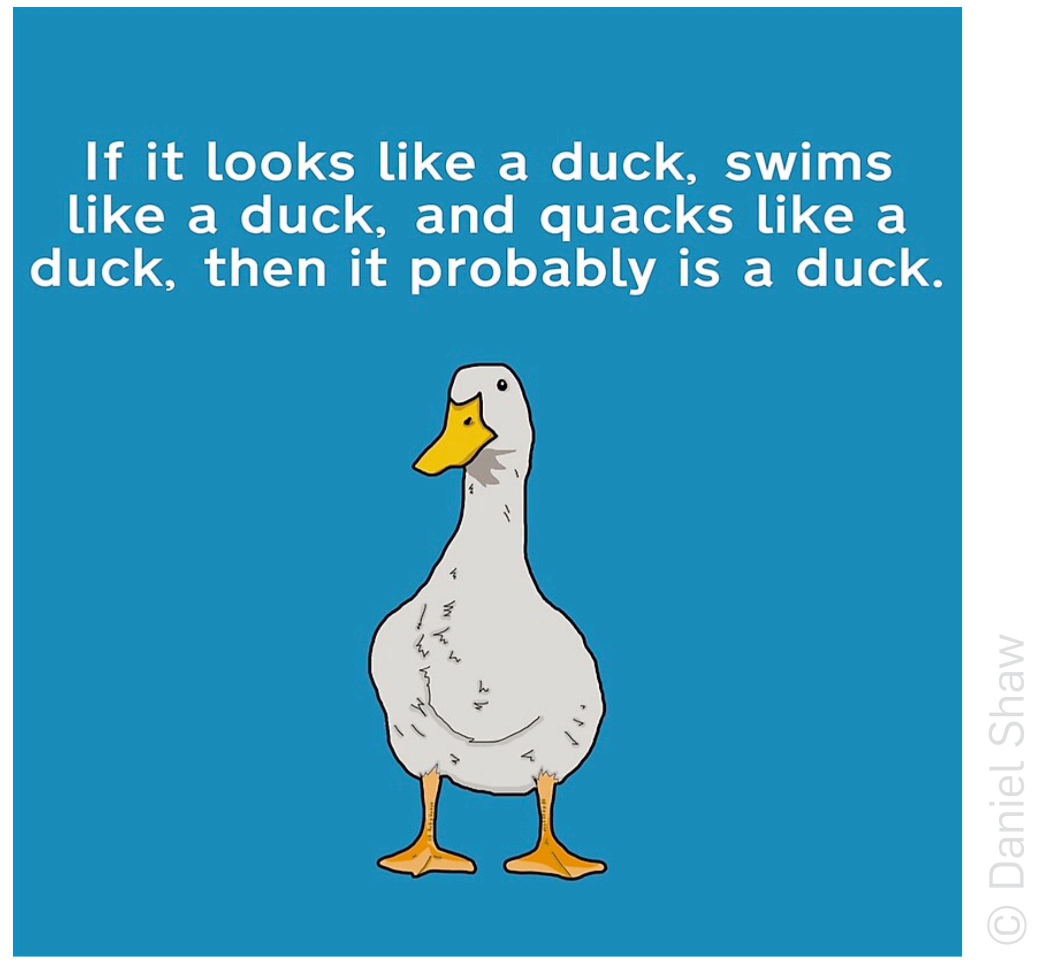
Heap Attacks: House of Spirit

- ptmalloc does not actually track what memory is allocated (only what is free)
- free() does not do a lot of checks



Can we just forge some chunk in memory, that looks like a chunk, and free() it?

Yep! It will be added to the free list/bins, and the next malloc() call will return it.


If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.
© Daniel Shaw

No malloc() or free()?

- If there are no direct calls to malloc() / free() that are controllable, this might seem like a problem

No malloc() or free()?

- If there are no direct calls to malloc() / free() that are controllable, this might seem like a problem
- Luckily there are other functions that use malloc() and free()
  - printf(), scanf(), etc. (which we might be able to control)
    - Side note: Input/output buffering affects how exactly this works
      - You can disable it, but it comes at (some) performance costs:

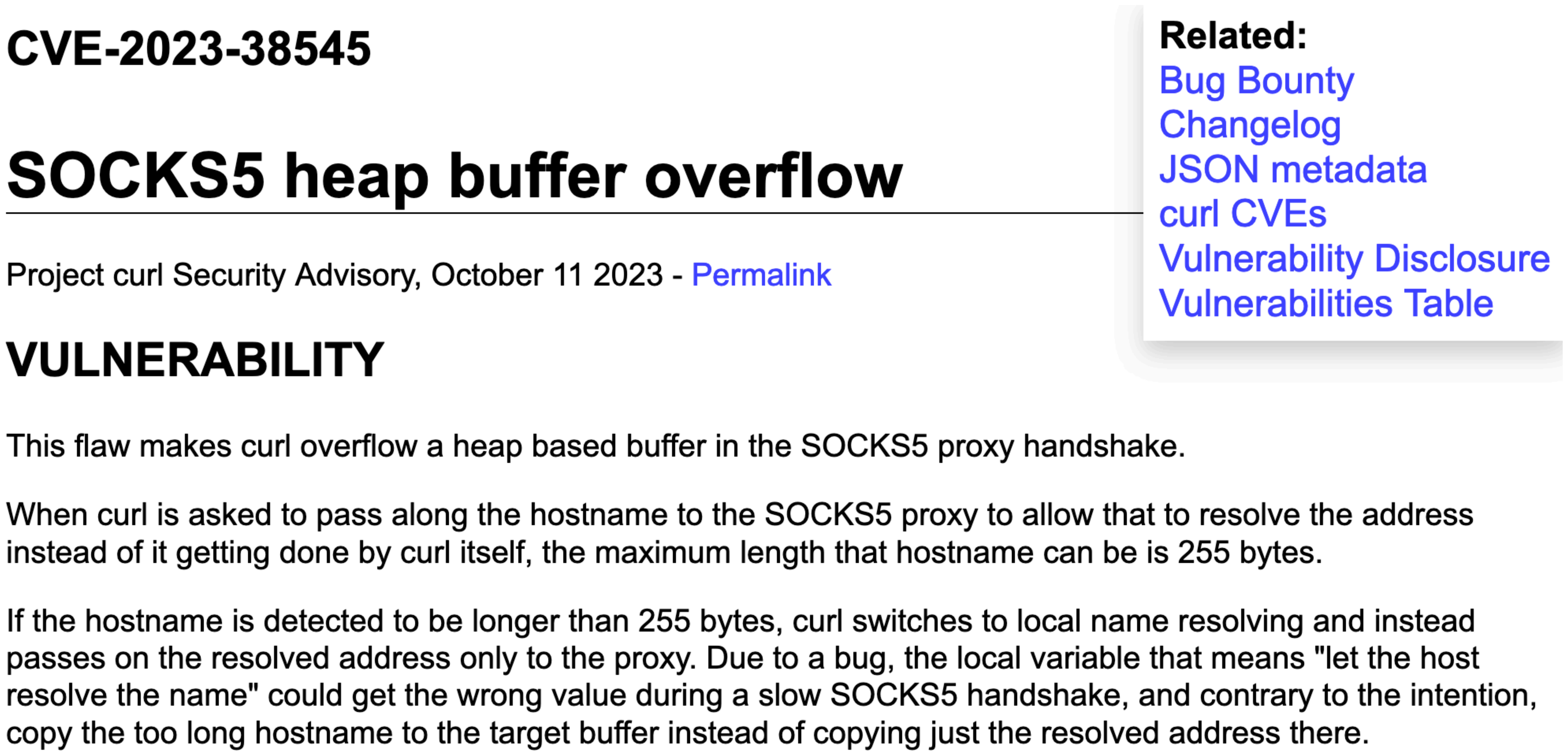
        ```
        setbuf(stdout, NULL);
        setbuf(stdin, NULL);
        ```

Heap: Insightful Resources

- "The Malloc Maleficarum", Phantasmal Phantasmagoria
  https://seclists.org/bugtraq/2005/Oct/118
  - Now extended to an entire "collection" of houses (see how2heap)
- how2heap, Shellphish
  https://github.com/shellphish/how2heap/
- "JPEG COM Marker Processing Vulnerability", Solar Designer
  https://www.openwall.com/articles/JPEG-COM-Marker-Vulnerability
- "Vudo malloc tricks", Michel Kaempf
  http://phrack.org/issues/57/8.html
- "Once upon a free()", anonymous
  http://phrack.org/issues/57/9.html

curl: CVE-2023-38545

Remember this vulnerability?



CVE-2023-38545

SOCKS5 heap buffer overflow

Project curl Security Advisory, October 11 2023 - Permalink

VULNERABILITY

This flaw makes curl overflow a heap based buffer in the SOCKS5 proxy handshake.

When curl is asked to pass along the hostname to the SOCKS5 proxy to allow that to resolve the address instead of it getting done by curl itself, the maximum length that hostname can be is 255 bytes.

If the hostname is detected to be longer than 255 bytes, curl switches to local name resolving and instead passes on the resolved address only to the proxy. Due to a bug, the local variable that means "let the host resolve the name" could get the wrong value during a slow SOCKS5 handshake, and contrary to the intention, copy the too long hostname to the target buffer instead of copying just the resolved address there.

Related:
Bug Bounty
Changelog
JSON metadata
curl CVEs
Vulnerability Disclosure
Vulnerabilities Table

curl: CVE-2023-38545

- Severity: High (curl authors) / CVE 9.8/10 Critical
  - Side note: Do not take CVE severity too serious, it is quite random
- Heap overwrite in SOCKS5 proxy protocol code
- Occurred during a performance rewrite
  - "to convert the function that connects to a SOCKS5 proxy from a blocking call into a non-blocking state machine."
  - "shipped in 7.69.0 as the first release featuring this enhancement. And by extension also the first release vulnerable to CVE-2023-38545."

curl: CVE-2023-38545

- Severity: High (curl authors) / CVE 9.8/10 Critical
  - Side note: Do not take CVE severity too serious, it is quite random
- Heap overwrite in SOCKS5 proxy protocol code
- Occurred during a performance rewrite
  - "to convert the function that connects to a SOCKS5 proxy from a blocking call into a non-blocking state machine."
  - "shipped in 7.69.0 as the first release featuring this enhancement. And by extension also the first release vulnerable to CVE-2023-38545."

What's the issue?

curl: CVE-2023-38545

- Function do_SOCKS5 implements a state machine
- SOCKS5 allows you to resolve network names locally or pass the name to the proxy for resolving
    - Side note: Local resolving leaks to your DNS recursor what names you are trying to resolve, which you then access via the proxy!

curl: CVE-2023-38545

- Function do_SOCKS5 implements a state machine
- SOCKS5 allows you to resolve network names locally or pass the name to the proxy for resolving
  - Side note: Local resolving leaks to your DNS recursor what names you are trying to resolve, which you then access via the proxy!
  - curl checked this via at the beginning of every call to do_SOCKS5

```
bool socks5_resolve_local =
  (conn->socks_proxy.proxytype == CURLPROXY_SOCKS5) ? TRUE : FALSE;
```

curl: CVE-2023-38545

- Function do_SOCKS5 implements a state machine
- SOCKS5 allows you to resolve network names locally or pass the name to the proxy for resolving
  - Side note: Local resolving leaks to your DNS recursor what names you are trying to resolve, which you then access via the proxy!
  - curl checked this via at the beginning of every call to do_SOCKS5

```
bool socks5_resolve_local =
  (conn->socks_proxy.proxytype == CURLPROXY_SOCKS5) ? TRUE : FALSE;
```

  - SOCKS5 limits names to 255 bytes, if the name is longer, then curl decided to instead resolve it locally (DNS limits it to 253 bytes, but curl's URL parser accepts names up to 65535 bytes)

```
/* RFC1928 chapter 5 specifies max 255 chars for domain name in packet */
if(!socks5_resolve_local && hostname_len > 255) {
  infof(data, "SOCKS5: server resolving disabled for hostnames of "
        "length > 255 [actual len=%zu]", hostname_len);
  socks5_resolve_local = TRUE;
}
```

curl: CVE-2023-38545

- If the name is too long for SOCKS5, then curl switches to local resolving
  - This is bad because it ignores user preferences and curl should fail instead, but this is how it worked before, so the performance rewrite maintained it

curl: CVE-2023-38545

- If the name is too long for SOCKS5, then curl switches to local resolving
  - This is bad because it ignores user preferences and curl should fail instead, but this is how it worked before, so the performance rewrite maintained it
- If there is no more data at the SOCKS proxy, for example, because it is too slow, then do_SOCKS5 returns and needs to be called again

curl: CVE-2023-38545

- If the name is too long for SOCKS5, then curl switches to local resolving
    - This is bad because it ignores user preferences and curl should fail instead, but this is how it worked before, so the performance rewrite maintained it
- If there is no more data at the SOCKS proxy, for example, because it is too slow, then do_SOCKS5 returns and needs to be called again
- But at the beginning of do_SOCKS5, curl sets the value to resolve via the proxy, ignoring the length of the name in that decision

curl: CVE-2023-38545

- If the name is too long for SOCKS5, then curl switches to local resolving
  - This is bad because it ignores user preferences and curl should fail instead, but this is how it worked before, so the performance rewrite maintained it
- If there is no more data at the SOCKS proxy, for example, because it is too slow, then do_SOCKS5 returns and needs to be called again
- But at the beginning of do_SOCKS5, curl sets the value to resolve via the proxy, ignoring the length of the name in that decision
- Then, curl assembles the SOCKS5 protocol frame in memory, assuming that it should copy the too-long hostname, which in turn overwrites other fields in the protocol frame

curl: CVE-2023-38545

- do_SOCKS5 is called twice
  - It initializes the how-to-resolve setting
  - This value is updated for when the name is too long, but then reset on the next do_SOCKS5 call
  - curl finally copies the too-long name into a buffer that may be too small (depending on other settings), overwriting metadata

curl: CVE-2023-38545

- do_SOCKS5 is called twice
  - It initializes the how-to-resolve setting
  - This value is updated for when the name is too long, but then reset on the next do_SOCKS5 call
  - curl finally copies the too-long name into a buffer that may be too small (depending on other settings), overwriting metadata
- An attacker can remotely exploit against a client that uses curl with a SOCKS5 proxy and resolving via the proxy, by sending a HTTP redirect to a domain name that is an invalid DNS name (>253 bytes), but a valid name for curl's parser, like: `https://[64000 times A]/`
  - POC: https://gist.github.com/xen0bit/0dccb11605abbeb6021963e2b1a811d3

Looney Tunables: CVE-2023-4911

- Local Privilege Escalation in the glibc's ld.so (loader!)
  - via glibc tunables (settings you can change at load time)
- Discovered by Qualys, extensive + detailed write-up available
  - https://www.qualys.com/2023/10/03/cve-2023-4911/looney-tunables-local-privilege-escalation-glibc-ld-so.txt

Looney Tunables: CVE-2023-4911

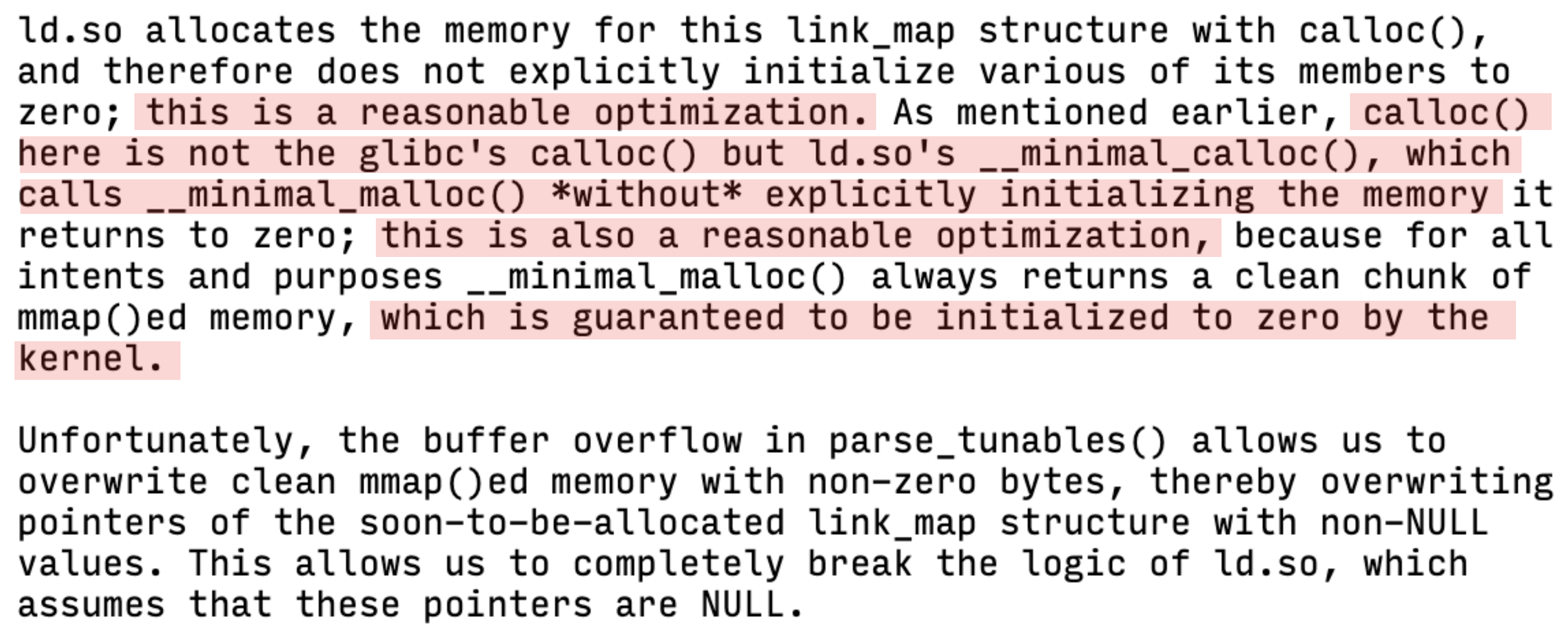
- Local Privilege Escalation in the glibc's ld.so (loader!)
  - via glibc tunables (settings you can change at load time)
- Discovered by Qualys, extensive + detailed write-up available
  - https://www.qualys.com/2023/10/03/cve-2023-4911/looney-tunables-local-privilege-escalation-glibc-ld-so.txt
- An interesting snippet:

```
ld.so allocates the memory for this link_map structure with calloc(),
and therefore does not explicitly initialize various of its members to
zero; this is a reasonable optimization. As mentioned earlier, calloc()
here is not the glibc's calloc() but ld.so's __minimal_calloc(), which
calls __minimal_malloc() *without* explicitly initializing the memory it
returns to zero; this is also a reasonable optimization, because for all
intents and purposes __minimal_malloc() always returns a clean chunk of
mmap()ed memory, which is guaranteed to be initialized to zero by the
kernel.

Unfortunately, the buffer overflow in parse_tunables() allows us to
overwrite clean mmap()ed memory with non-zero bytes, thereby overwriting
pointers of the soon-to-be-allocated link_map structure with non-NULL
values. This allows us to completely break the logic of ld.so, which
assumes that these pointers are NULL.
```

Looney Tunables: CVE-2023-4911

- Local Privilege Escalation in the glibc's ld.so (loader!)
  - via glibc tunables (settings you can change at load time)
- Discovered by Qualys, extensive + detailed write-up available
  - https://www.qualys.com/2023/10/03/cve-2023-4911/looney-tunables-local-privilege-escalation-glibc-ld-so.txt
- An interesting snippet:

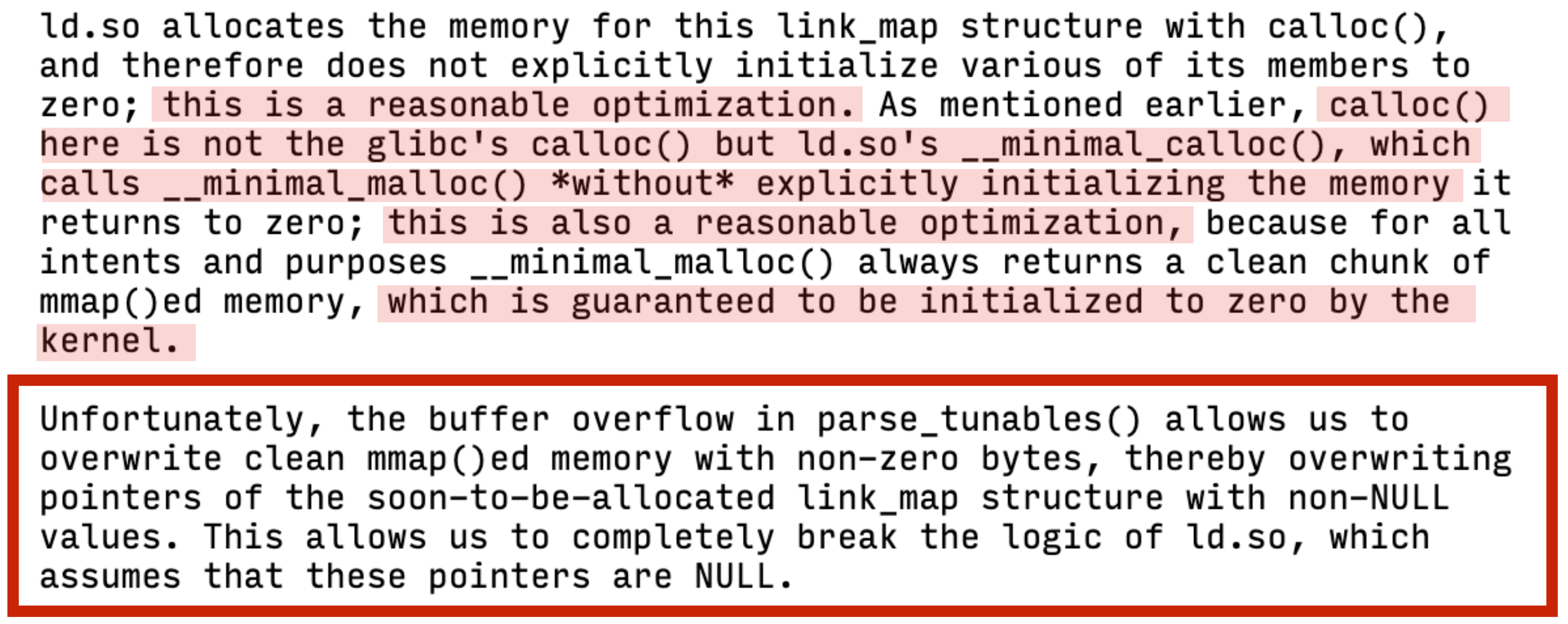


```
ld.so allocates the memory for this link_map structure with calloc(),
and therefore does not explicitly initialize various of its members to
zero; this is a reasonable optimization. As mentioned earlier, calloc()
here is not the glibc's calloc() but ld.so's __minimal_calloc(), which
calls __minimal_malloc() *without* explicitly initializing the memory it
returns to zero; this is also a reasonable optimization, because for all
intents and purposes __minimal_malloc() always returns a clean chunk of
mmap()ed memory, which is guaranteed to be initialized to zero by the
kernel.

Unfortunately, the buffer overflow in parse_tunables() allows us to
overwrite clean mmap()ed memory with non-zero bytes, thereby overwriting
pointers of the soon-to-be-allocated link_map structure with non-NULL
values. This allows us to completely break the logic of ld.so, which
assumes that these pointers are NULL.
```

Looney Tunables: CVE-2023-4911

- Local Privilege Escalation in the glibc's ld.so (loader!)
  - via glibc tunables (settings you can change at load time)
- Discovered by Qualys, extensive + detailed write-up available
  - https://www.qualys.com/2023/10/03/cve-2023-4911/looney-tunables-local-privilege-escalation-glibc-ld-so.txt
- An interesting snippet:

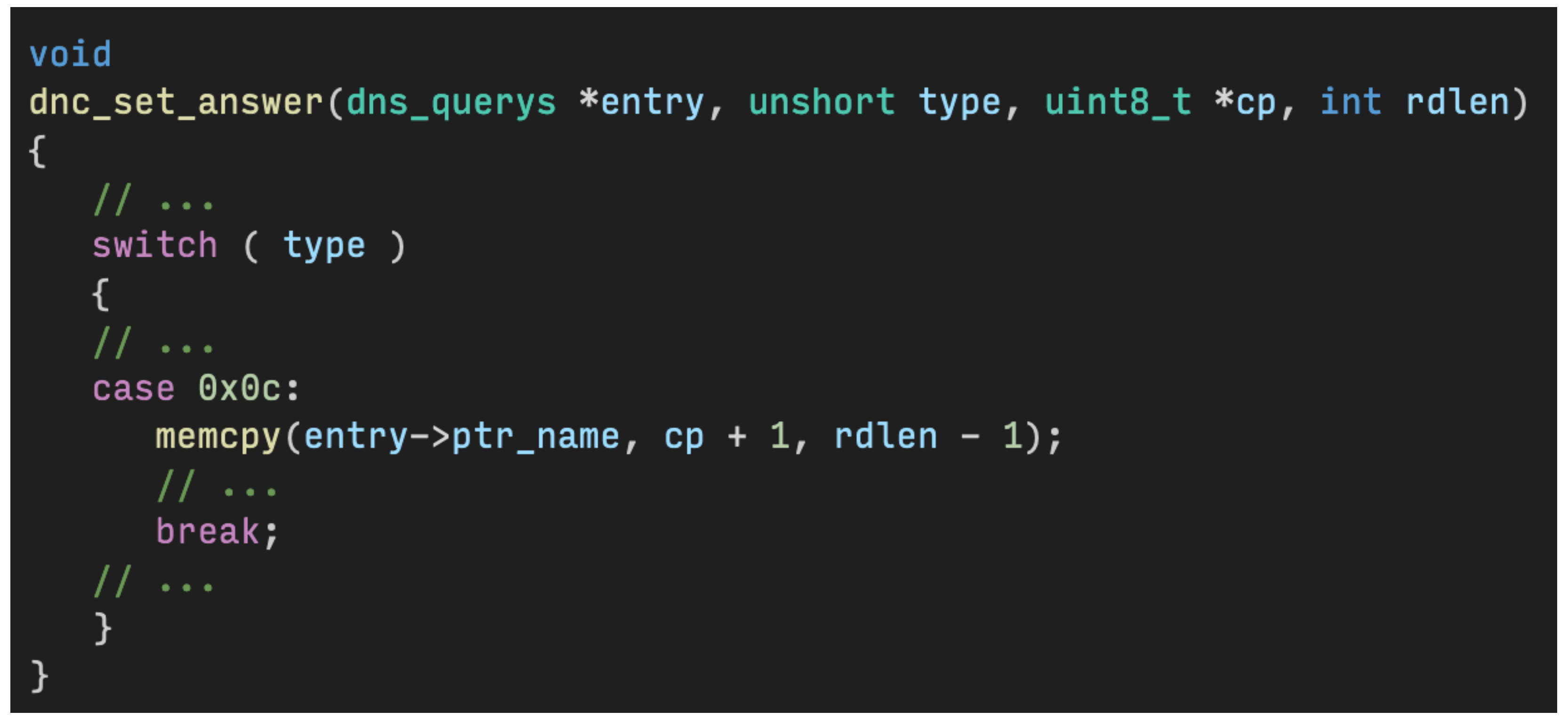


```
ld.so allocates the memory for this link_map structure with calloc(),
and therefore does not explicitly initialize various of its members to
zero; this is a reasonable optimization. As mentioned earlier, calloc()
here is not the glibc's calloc() but ld.so's __minimal_calloc(), which
calls __minimal_malloc() *without* explicitly initializing the memory it
returns to zero; this is also a reasonable optimization, because for all
intents and purposes __minimal_malloc() always returns a clean chunk of
mmap()ed memory, which is guaranteed to be initialized to zero by the
kernel.

Unfortunately, the buffer overflow in parse_tunables() allows us to
overwrite clean mmap()ed memory with non-zero bytes, thereby overwriting
pointers of the soon-to-be-allocated link_map structure with non-NULL
values. This allows us to completely break the logic of ld.so, which
assumes that these pointers are NULL.
```

Nintendo Wii U: CVE-2020-25928

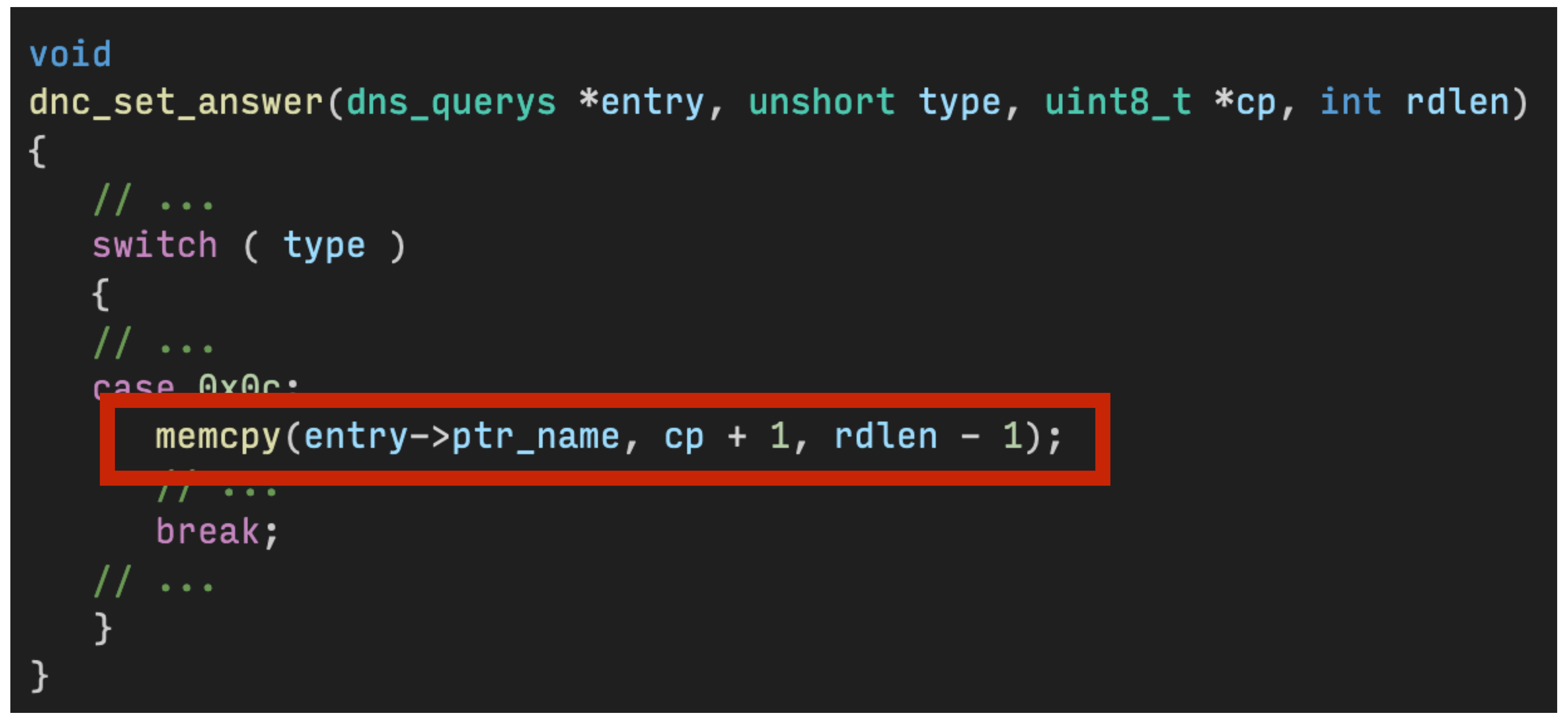
- Wii U uses InterNiche's NicheStack for TCP/IP
- NicheStack TCP/IP has a buffer overflow in a DNS feature



```
void
dnc_set_answer(dns_querys *entry, unshort type, uint8_t *cp, int rdlen)
{
   // ...
   switch ( type )
   {
   // ...
   case 0x0c:
      memcpy(entry->ptr_name, cp + 1, rdlen - 1);
      // ...
      break;
   // ...
   }
}
```

Nintendo Wii U: CVE-2020-25928

- Wii U uses InterNiche's NicheStack for TCP/IP
- NicheStack TCP/IP has a buffer overflow in a DNS feature

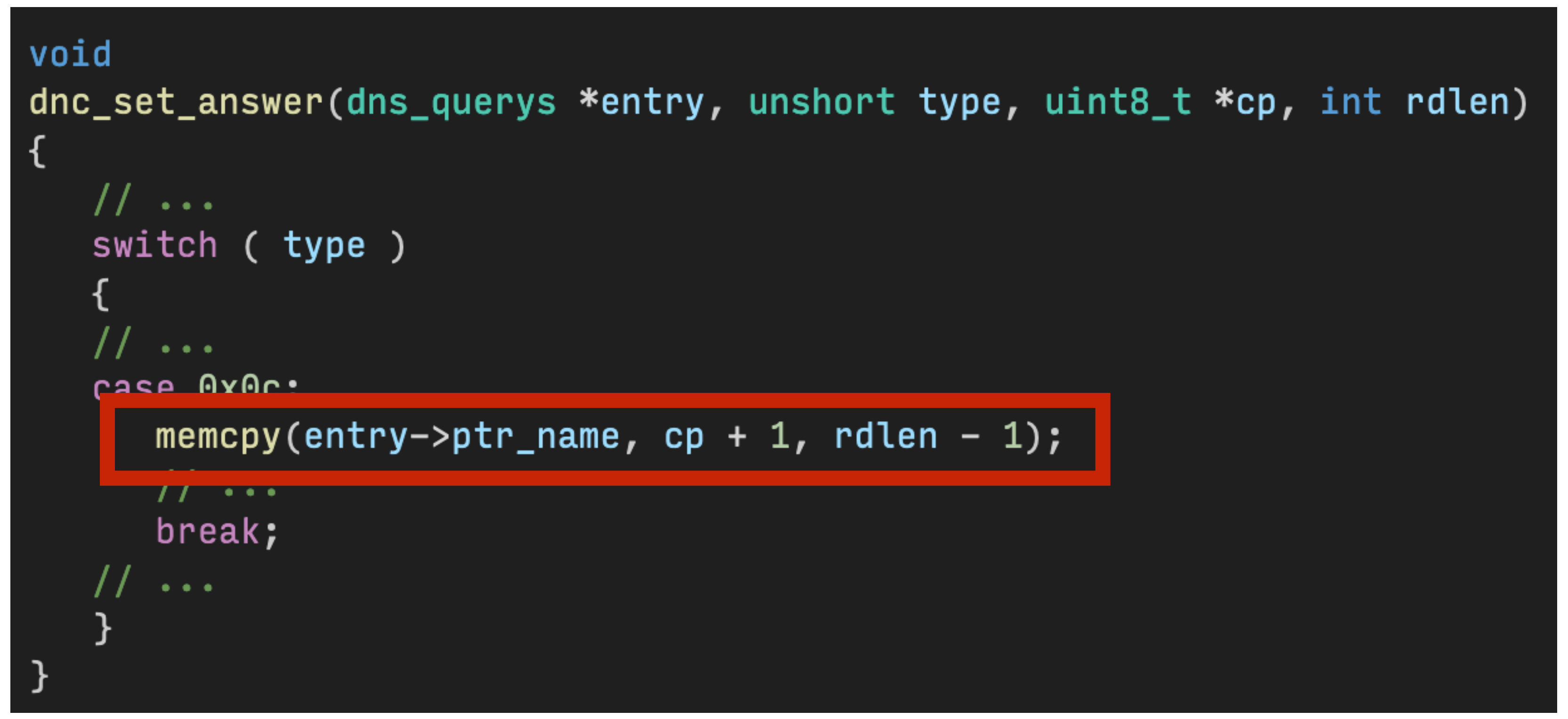


```
void
dnc_set_answer(dns_querys *entry, unshort type, uint8_t *cp, int rdlen)
{
   // ...
   switch ( type )
   {
   // ...
   case 0x0c:
      memcpy(entry->ptr_name, cp + 1, rdlen - 1);
      // ...
      break;
   // ...
   }
}
```

DNS client copies the response
to a fixed size on the heap,
without checking the size

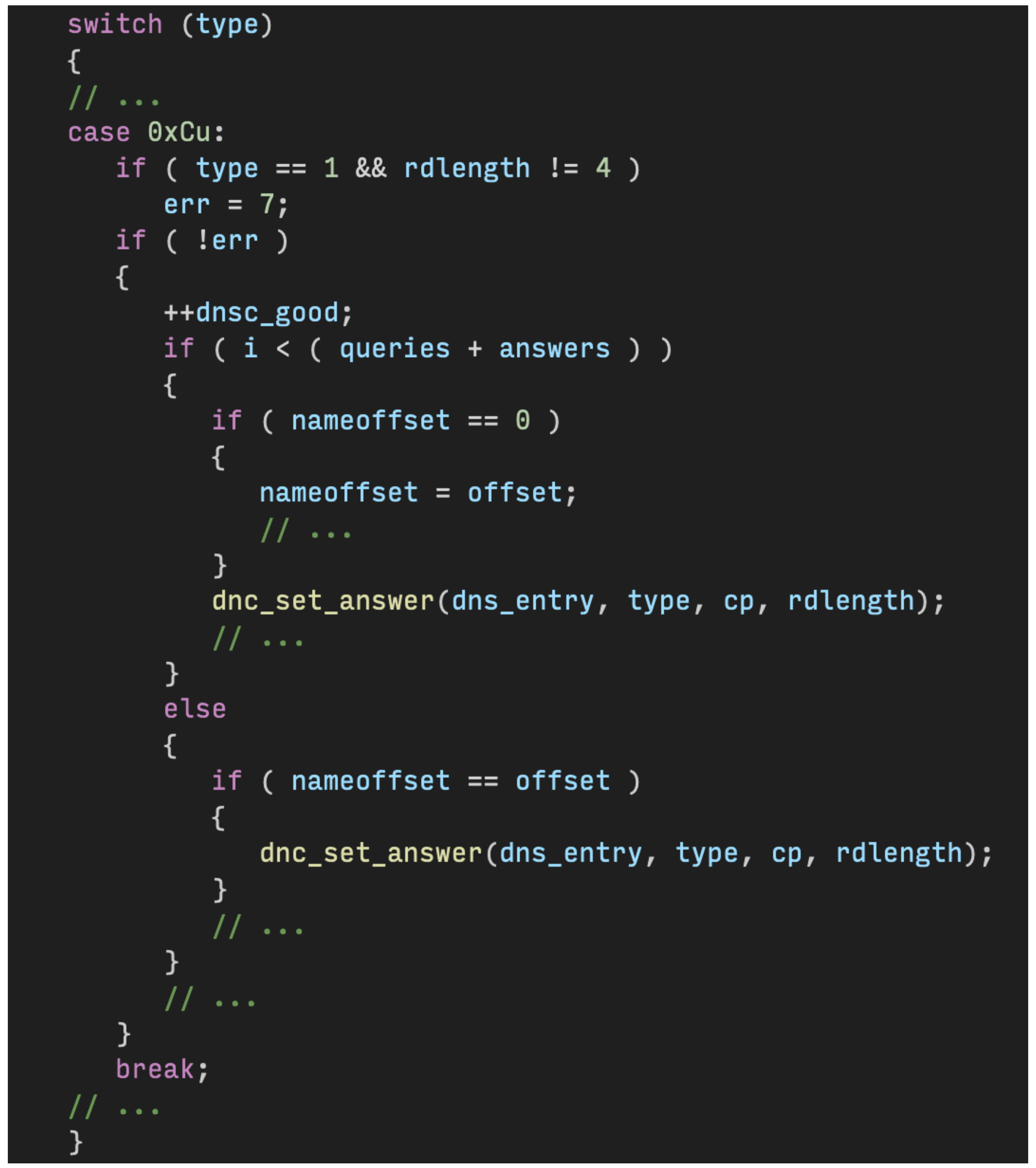
Nintendo Wii U: CVE-2020-25928

- Wii U uses InterNiche's NicheStack for TCP/IP
- NicheStack TCP/IP has a buffer overflow in a DNS feature



```
void
dnc_set_answer(dns_querys *entry, unshort type, uint8_t *cp, int rdlen)
{
   // ...
   switch ( type )
   {
   // ...
   case 0x0c:
      memcpy(entry->ptr_name, cp + 1, rdlen - 1);
      // ...
      break;
   // ...
   }
}
```

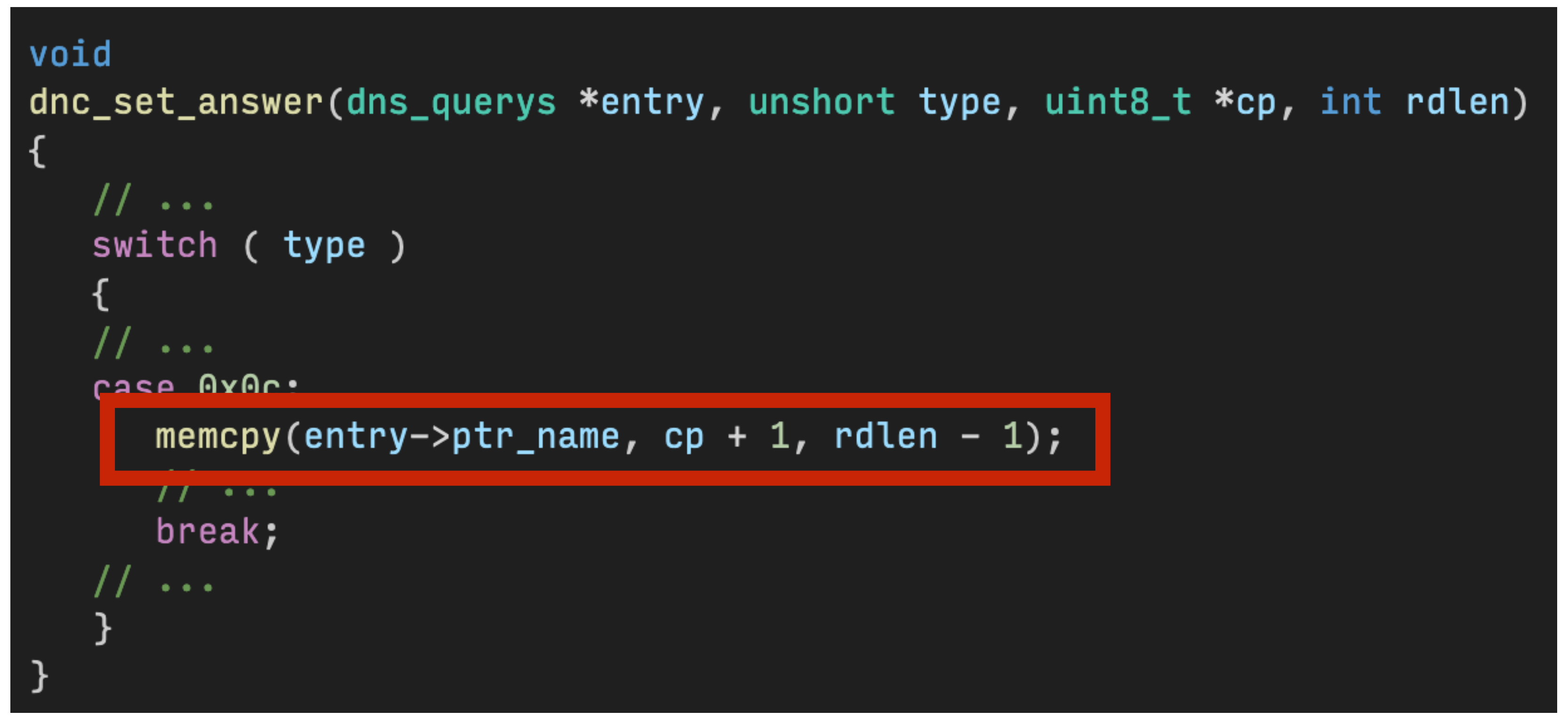
DNS client copies the response to a fixed size on the heap, without checking the size



```
switch (type)
{
// ...
case 0xCu:
   if ( type == 1 && rdlength != 4 )
      err = 7;
   if ( !err )
   {
      ++dnsc_good;
      if ( i < ( queries + answers ) )
      {
         if ( nameoffset == 0 )
         {
            nameoffset = offset;
            // ...
         }
         dnc_set_answer(dns_entry, type, cp, rdlength);
         // ...
      }
      else
      {
         if ( nameoffset == offset )
         {
            dnc_set_answer(dns_entry, type, cp, rdlength);
         }
         // ...
      }
      // ...
   }
   break;
// ...
}
```

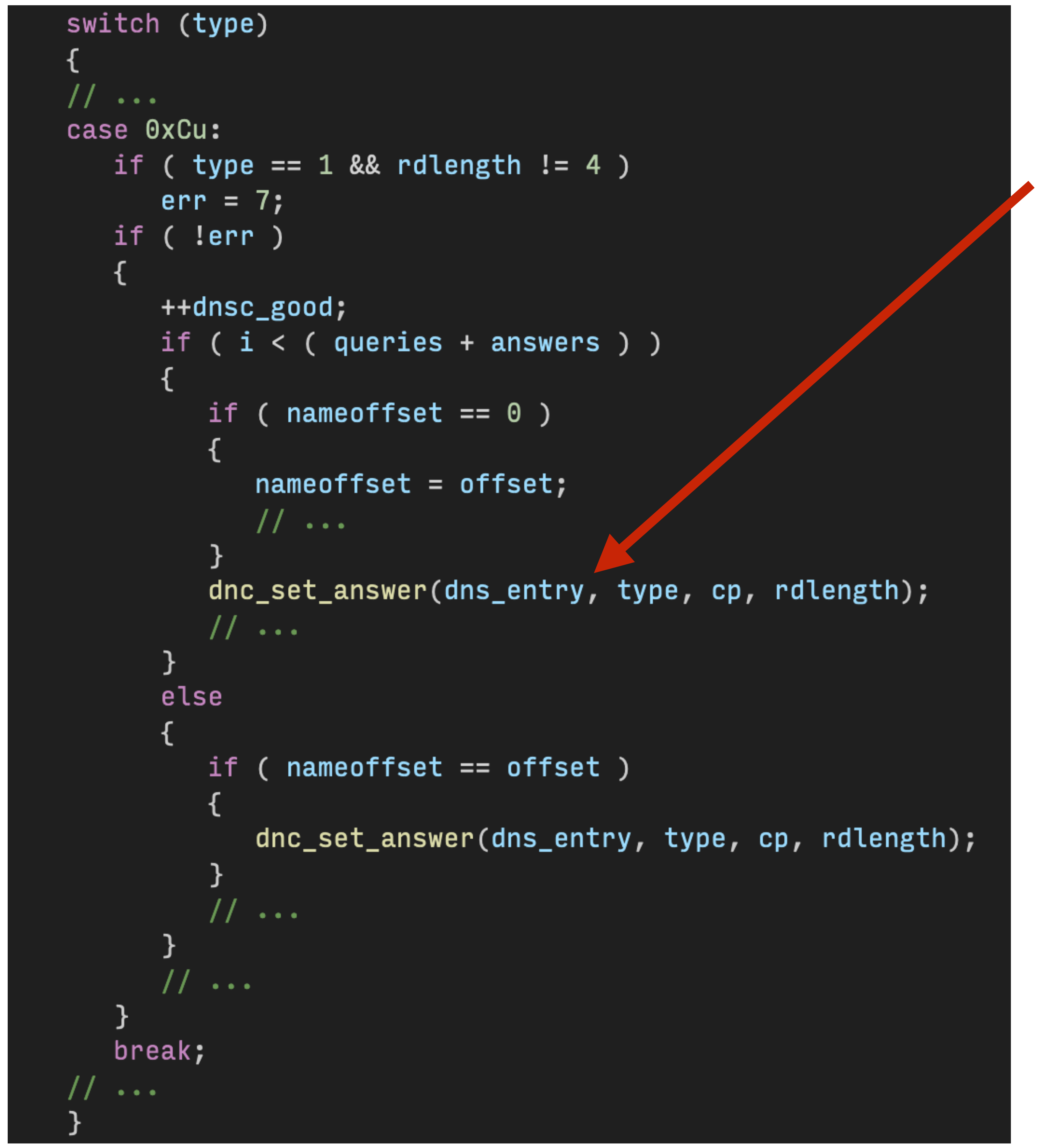
Nintendo Wii U: CVE-2020-25928

- Wii U uses InterNiche's NicheStack for TCP/IP
- NicheStack TCP/IP has a buffer overflow in a DNS feature

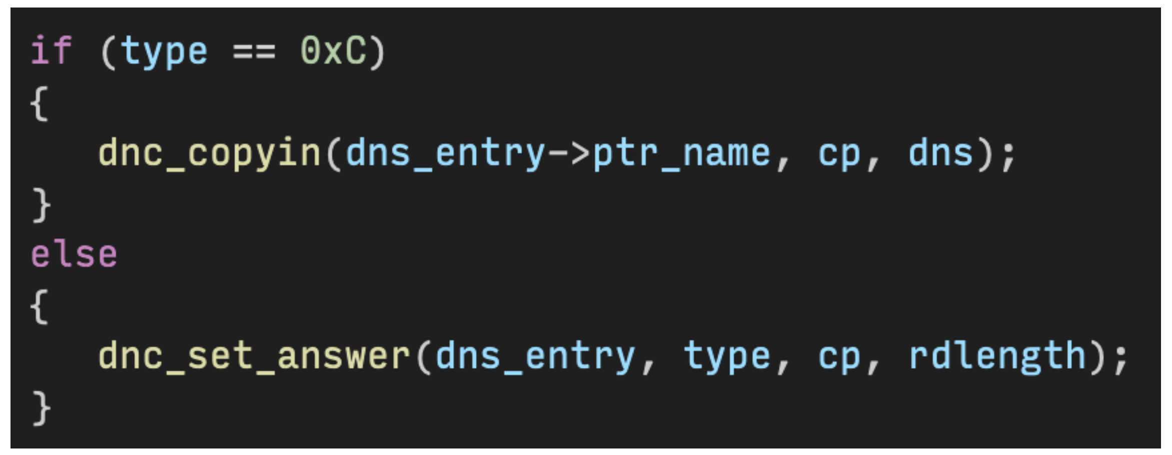


```
void
dnc_set_answer(dns_querys *entry, unshort type, uint8_t *cp, int rdlen)
{
   // ...
   switch ( type )
   {
   // ...
   case 0x0c:
      memcpy(entry->ptr_name, cp + 1, rdlen - 1);
      // ...
      break;
   // ...
   }
}
```

DNS client copies the response to a fixed size on the heap, without checking the size



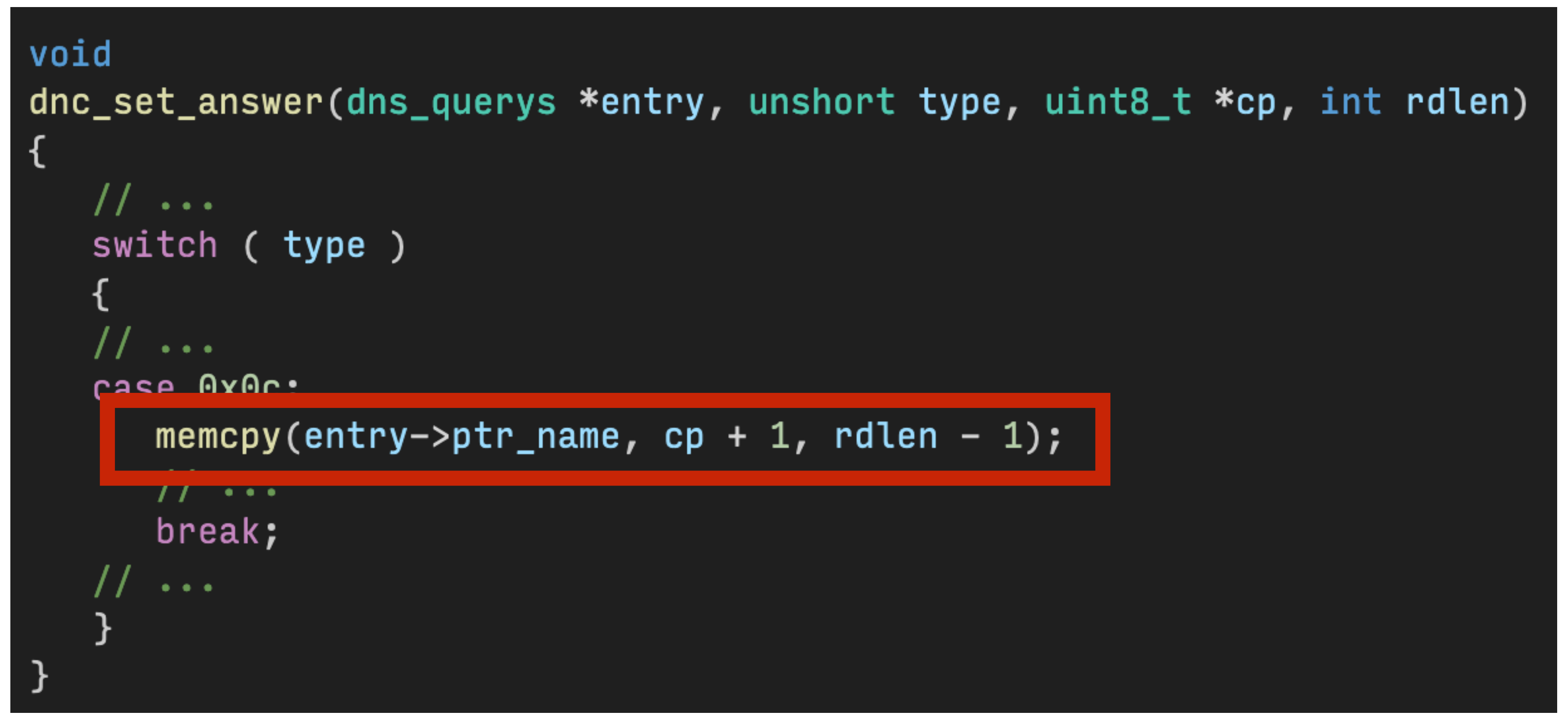
```
switch (type)
{
// ...
case 0xCu:
   if ( type == 1 && rdlength != 4 )
      err = 7;
   if ( !err )
   {
      ++dnsc_good;
      if ( i < ( queries + answers ) )
      {
         if ( nameoffset == 0 )
         {
            nameoffset = offset;
            // ...
         }
         dnc_set_answer(dns_entry, type, cp, rdlength);
         // ...
      }
      else
      {
         if ( nameoffset == offset )
         {
            dnc_set_answer(dns_entry, type, cp, rdlength);
         }
         // ...
      }
      // ...
   }
   break;
// ...
}
```



```
if (type == 0xC)
{
   dnc_copyin(dns_entry->ptr_name, cp, dns);
}
else
{
   dnc_set_answer(dns_entry, type, cp, rdlength);
}
```

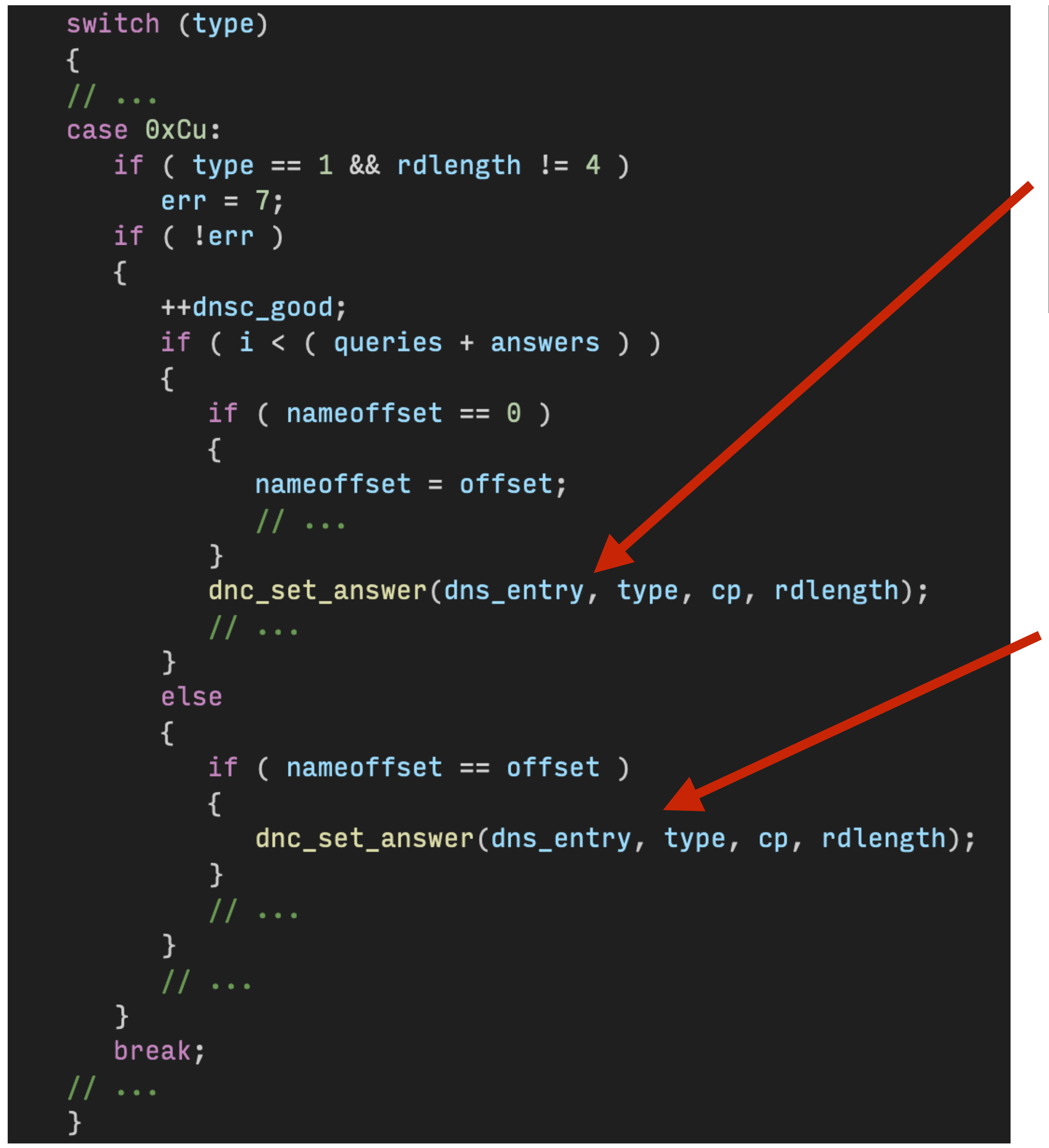
Nintendo Wii U: CVE-2020-25928

- Wii U uses InterNiche's NicheStack for TCP/IP
- NicheStack TCP/IP has a buffer overflow in a DNS feature

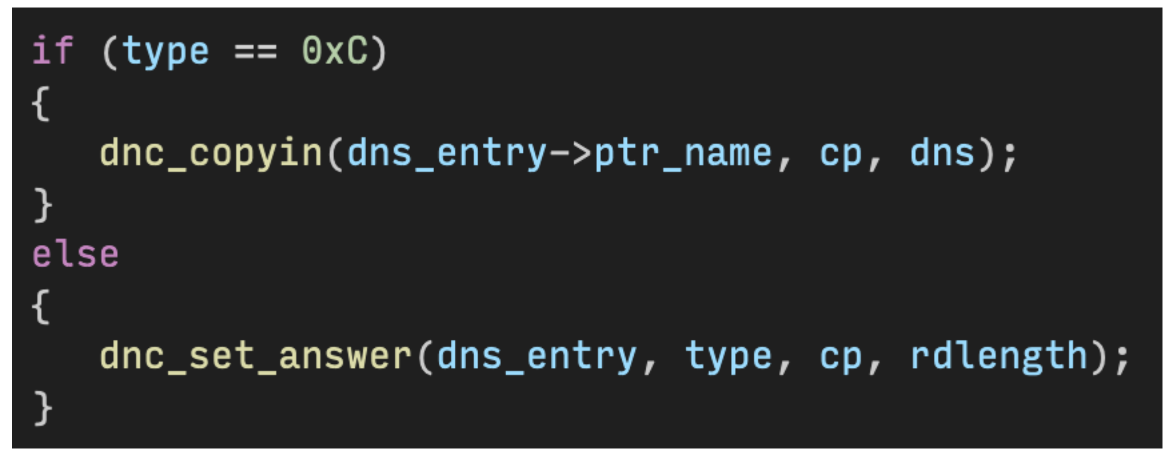


```c
void
dnc_set_answer(dns_querys *entry, unshort type, uint8_t *cp, int rdlen)
{
   // ...
   switch ( type )
   {
   // ...
   case 0x0c:
      memcpy(entry->ptr_name, cp + 1, rdlen - 1);
      // ...
      break;
   // ...
   }
}
```

DNS client copies the response to a fixed size on the heap, without checking the size



```c
switch (type)
{
// ...
case 0xCu:
   if ( type == 1 && rdlength != 4 )
      err = 7;
   if ( !err )
   {
      ++dnsc_good;
      if ( i < ( queries + answers ) )
      {
         if ( nameoffset == 0 )
         {
            nameoffset = offset;
            // ...
         }
         dnc_set_answer(dns_entry, type, cp, rdlength);
         // ...
      }
      else
      {
         if ( nameoffset == offset )
         {
            dnc_set_answer(dns_entry, type, cp, rdlength);
         }
         // ...
      }
      // ...
   }
   break;
// ...
}
```



```c
if (type == 0xC)
{
   dnc_copyin(dns_entry->ptr_name, cp, dns);
}
else
{
   dnc_set_answer(dns_entry, type, cp, rdlength);
}
```

Nintendo fixed it for the first case (which are the queries and answers of a DNS reply, and typically the only ones set for PTR responses), but an attacker could also have responses in the additional section

Nintendo Wii U: CVE-2020-25928

- DNS additional section?
  - DNS RFC 1035: "the additional records section contains RRs which relate to the query, but are not strictly answers for the question."

Nintendo Wii U: CVE-2020-25928

- DNS additional section?
  - DNS RFC 1035: "the additional records section contains RRs which relate to the query, but are not strictly answers for the question."
- Additional section for PTRs?
  - DNS RFC 1035: "PTR records cause no additional section processing."
    - Still processed here, and leading to a heap buffer overflow (because the dns_querys struct is on the heap)

Nintendo Wii U: CVE-2020-25928

- DNS additional section?
  - DNS RFC 1035: "the additional records section contains RRs which relate to the query, but are not strictly answers for the question."
- Additional section for PTRs?
  - DNS RFC 1035: "PTR records cause no additional section processing."
    - Still processed here, and leading to a heap buffer overflow (because the dns_querys struct is on the heap)
- Exploiting this on the Wii U is a bit more difficult, check out the write-up
  - The code runs on the ARM co-processor, but is later copied to the PowerPC side, requiring to spoof a struct on where to copy (requiring an address)
  - Using DNS over TCP and additional fields it uses, you get an arbitrary 256 bytes write
  - This is quite limited, but sufficient for a stage 1 ROP chain to read more data from a socket

https://garyodernichts.blogspot.com/2023/10/exploiting-dns-response-parsing-on-wii-u.html

Takeaways

- Heap exploitation is **non-trivial**
  - It requires you to precisely modify the data structures on the heap, to make the DMA do something that the program is not expecting, which you can then leverage
- At the same time, DMA is constantly modifying it
  - Think multi-threading, race conditions, etc.
- Further reading/watching:
  - "Automatic Heap Layout Manipulation for Exploitation", Heelan et al. https://www.usenix.org/conference/usenixsecurity18/presentation/heelan
  - "HeapHopper: Bringing Bounded Model Checking to Heap Implementation Security", Eckert et al., https://www.usenix.org/conference/usenixsecurity18/presentation/eckert
  - how2heap, https://github.com/shellphish/how2heap
  - heap-exploitation, https://heap-exploitation.dhavalkapil.com