Offensive and Defensive Cybersecurity

–

Heap Exploitation

21-22

Heap Introduction

- What's the heap?
  - One or more memory pages used to store data(rw-)
- Why do we need it?
  - For dynamic memory allocation
  - What about alloca, mmap, etc? Ok, but ...
- How do we manage it?
  - Through library functions

Memory Allocations

- **libc**
  - malloc - allocate a chunk of memory
  - calloc - allocate and zero-out memory
  - realloc - change size of an allocation
  - free - free a chunk of memory
- **syscall**
  - mmap (allocate memory page)
  - munmap (deallocate memory page)
  - brk/sbrk (change the location of the program break)

The HEAP Allocators

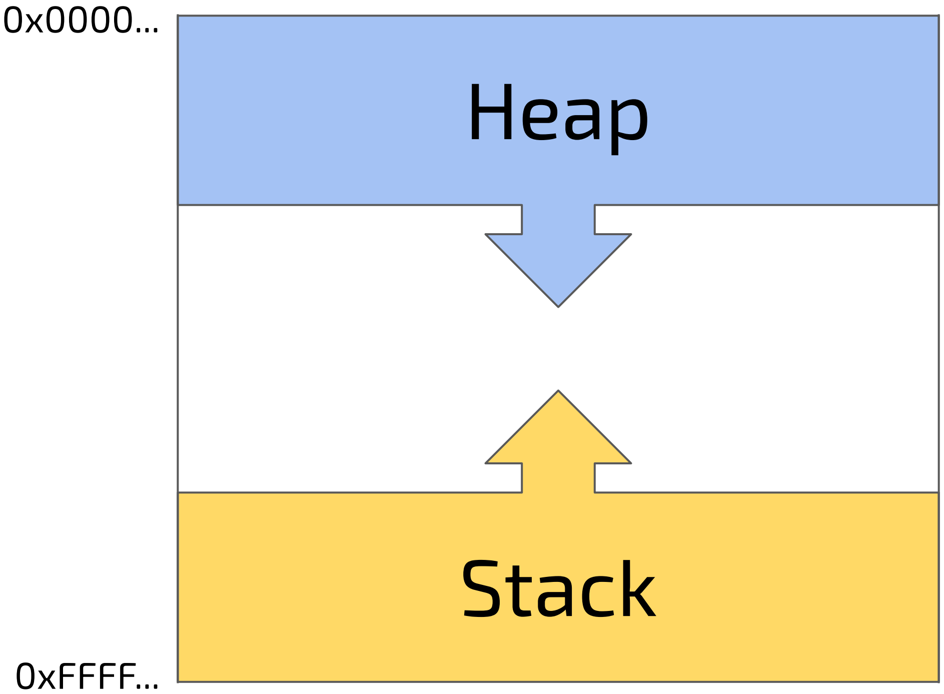
- **ptmalloc** (glibc)
- dlmalloc (was in glibc)
- tcmalloc (chromium)
- jemalloc (FreeBSD, Firefox, Android)

splittings, fits, coalescing, segregations (free list, storage, non determinism)

ptmalloc2 (aka the malloc of glibc)

- **splittings** (how to divide in chunk)
- **fits** (match requested size with)
- **coalescing** (how to merge chunks)
- **segregations** free list
- NO segregations storage
- **deterministic**

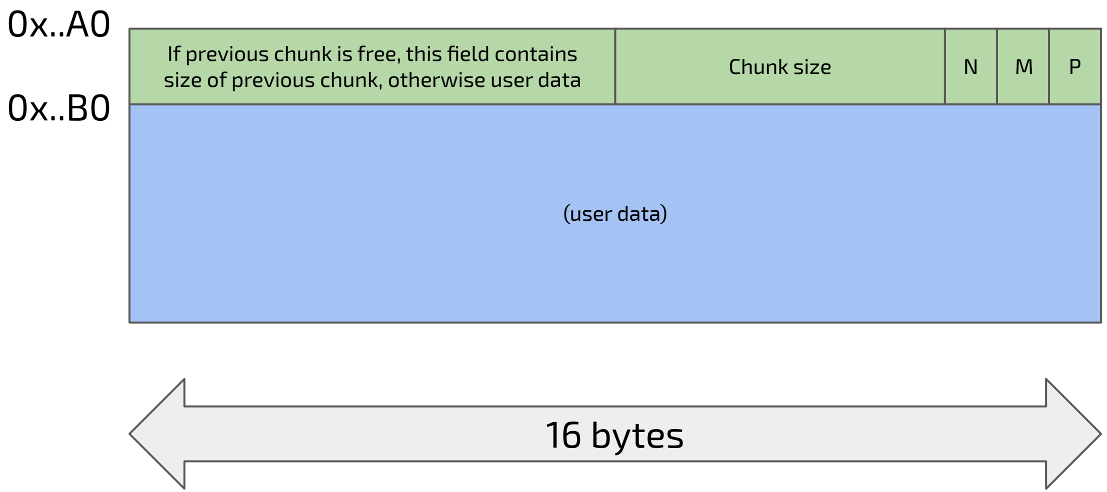
Memory Layout



Memory Allocation: malloc

- void* malloc(size_t size);
- Most known allocation primitive
- Requires allocation size(bytes)
- Returns a “void *” pointer which points the allocated memory(buffer)
- This buffer is a part of struct called chunk

Chunk



- PREV_INUSE (P) – This bit is set when previous chunk is allocated.
- IS_MMAPPED (M) – This bit is set when chunk is mmap'd.
- NON_MAIN_ARENA (N) – This bit is set when this chunk belongs to a thread arena.

Top Chunk

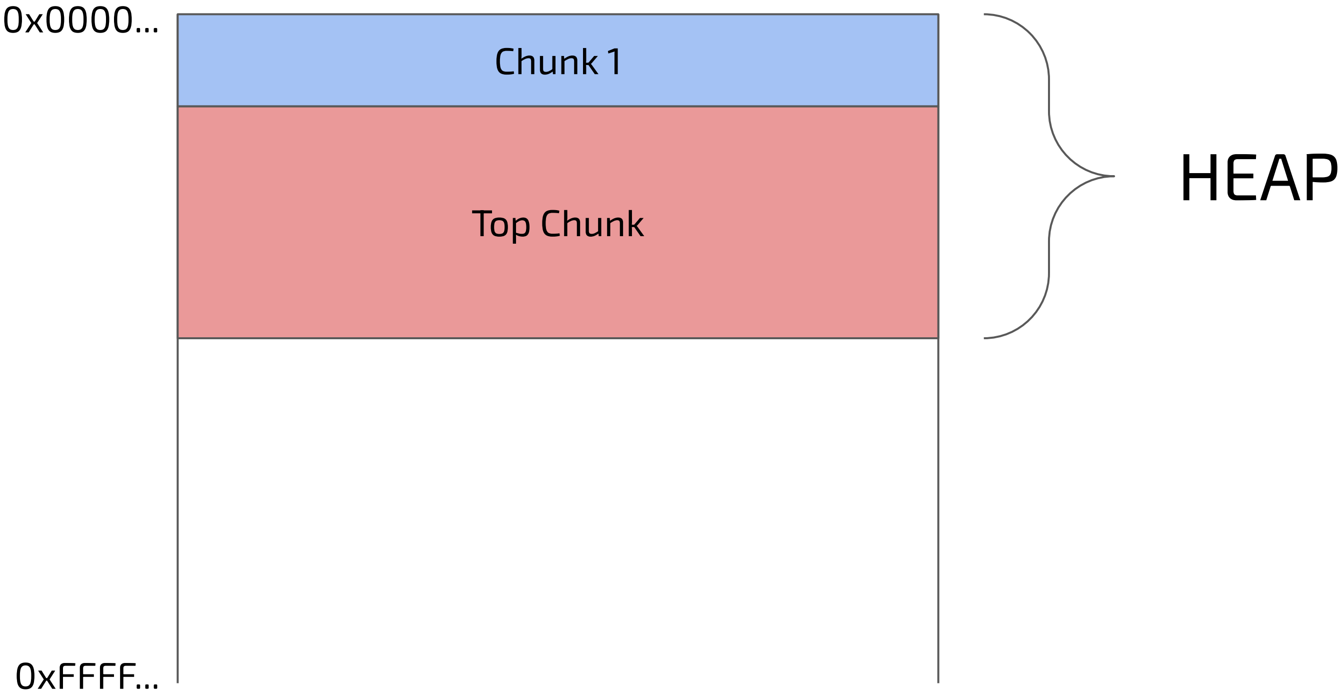
- Special chunk that occupies all the available memory space in the heap
- Every time a malloc is called it might be shrinked
- Once there's no more space on the heap, a brk(void *) is called to allocate more pages to the heap and the top chunk is expanded

Memory Allocation

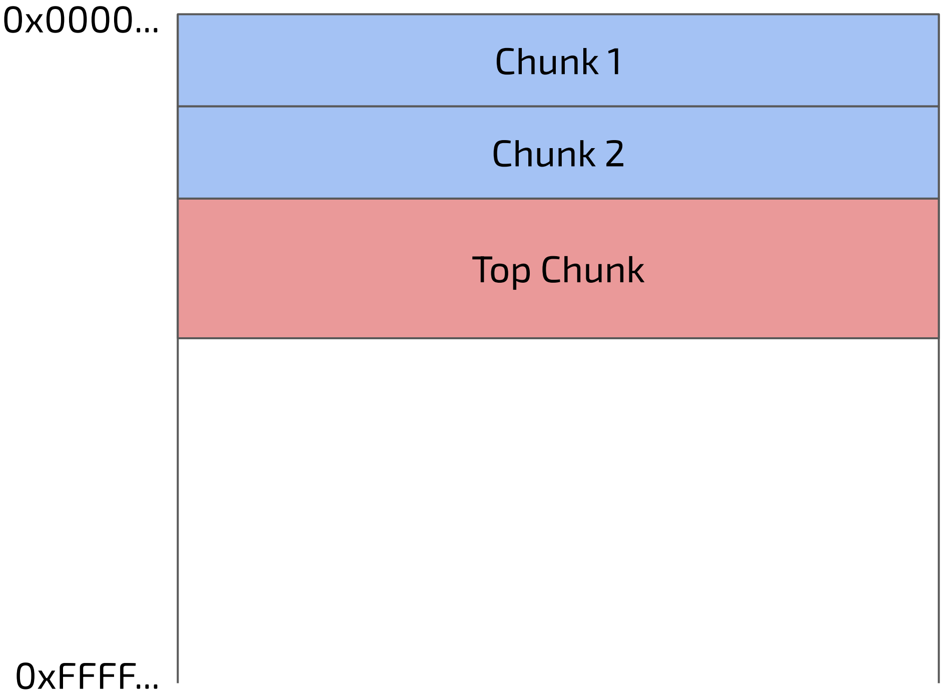
0x0000...

0xFFFF...

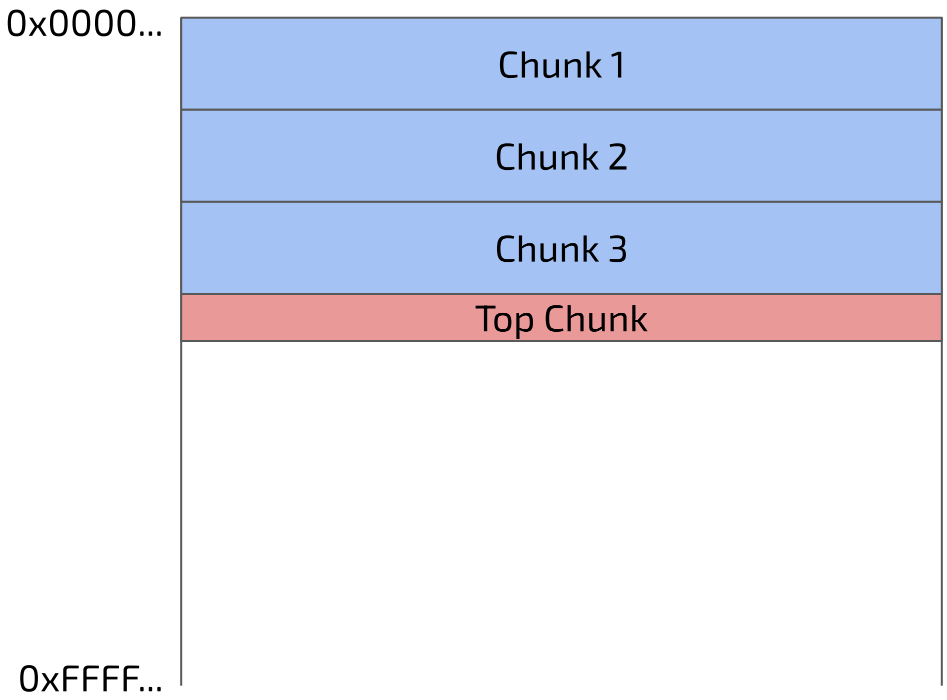
Memory Allocation



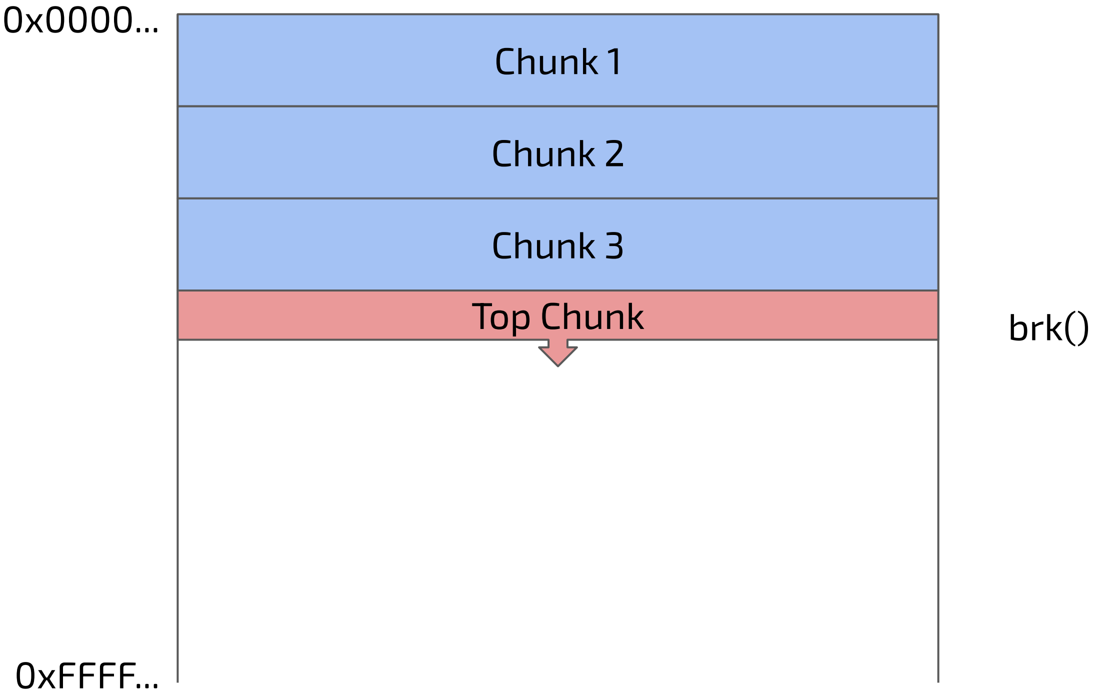
Memory Allocation



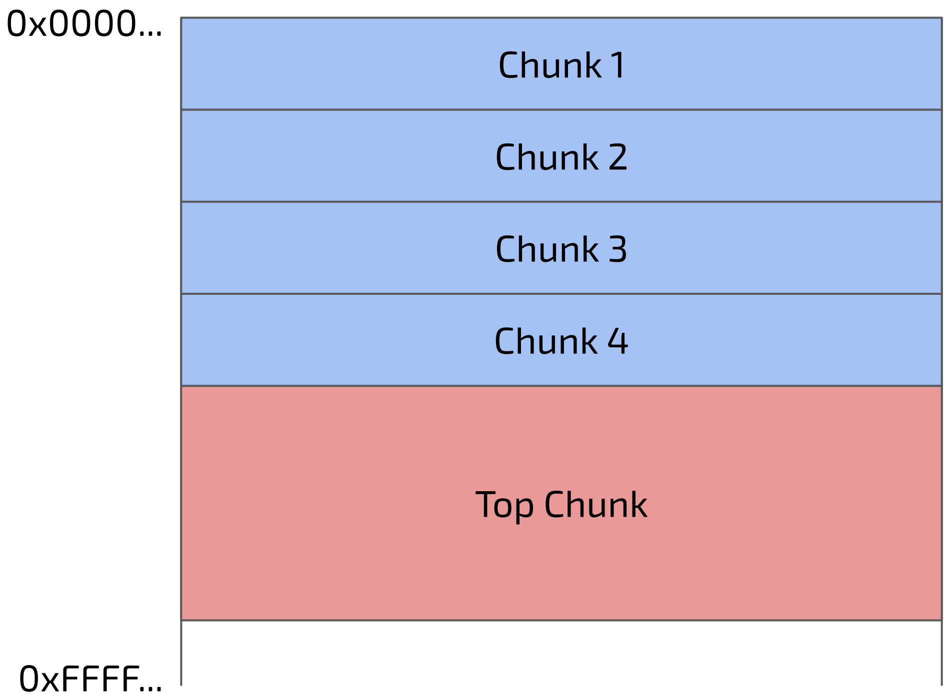
Memory Allocation



Memory Allocation



Memory Allocation

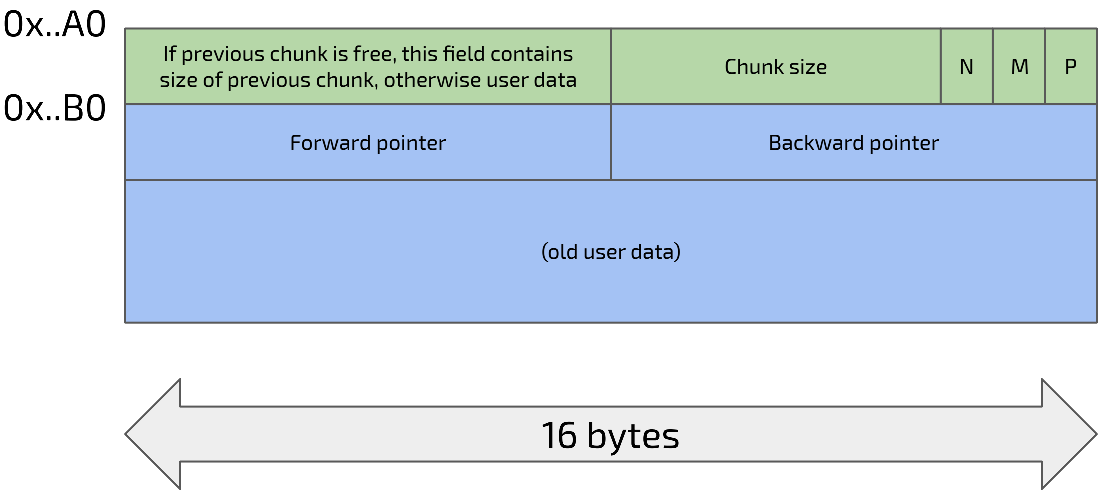


Let's see it in memory!

Memory Deallocation: free

- void free(void* ptr);
- Most known deallocation primitive
- Requires a pointer to a memory buffer previously allocated with a function for memory allocation (e.g. malloc)
- Freed chunks could be consolidated with other freed chunks (also with the top chunks)
- If not they are inserted in lists called bins

Free Chunk



- PREV_INUSE (P) – This bit is set when previous chunk is allocated.
- IS_MMAPPED (M) – This bit is set when chunk is mmap'd.
- NON_MAIN_ARENA (N) – This bit is set when this chunk belongs to a thread arena.

Bins

- Lists of freed chunks of a specific size
- Heads of the lists are located in the .bss of the libc (main_arena)
- Lists can be single or double linked
- 4 types of bins:
  - Fast bins – 8 Linked lists
  - Unsorted bin – 1 Double linked list
  - Small bins – 62 Double linked lists
  - Large bins – 62 Double linked lists

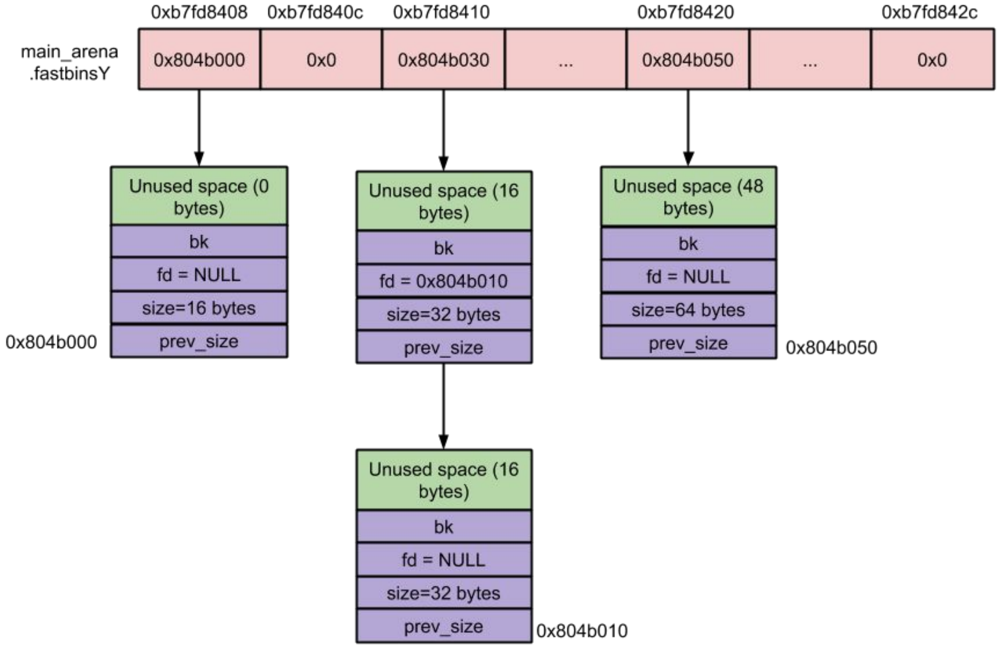
Bins

- t-cache
- Fast bin (0x20 to 0x90 bytes)
- Unsorted bin
- Small bin (< 512 bytes)
- Large bin (>= 512 bytes)
- top-chunk

Fast Bins

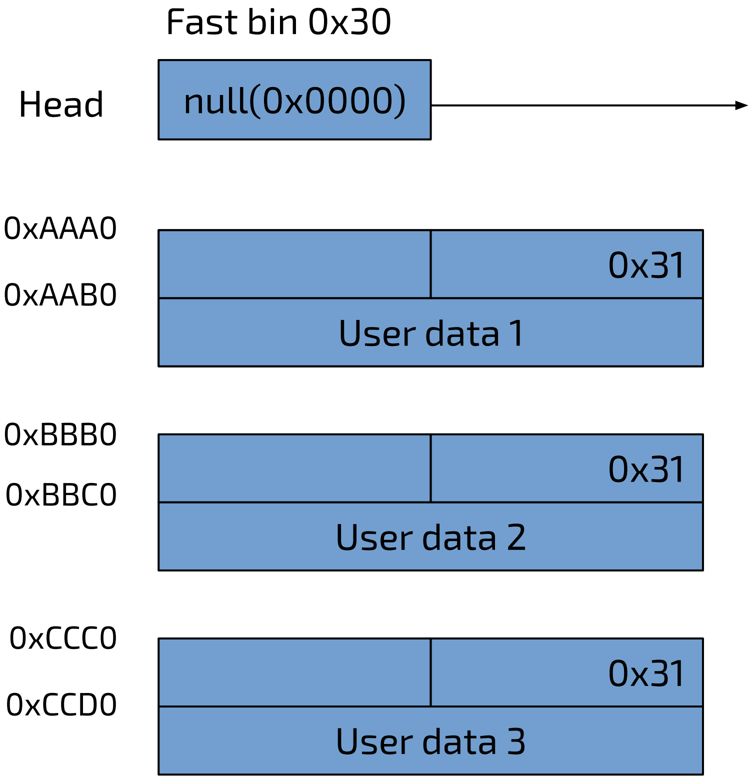
- Optimized bins for tiny freed chunks
- Managed as LIFO linked lists (backward pointer not used)
- Better performance, less checks and operations
- Freed chunks are never consolidate with any other freed chunk
- Freed chunks in fast bins act as non freed chunks (P flag of next chunk is not set to 0)

Fast Bins

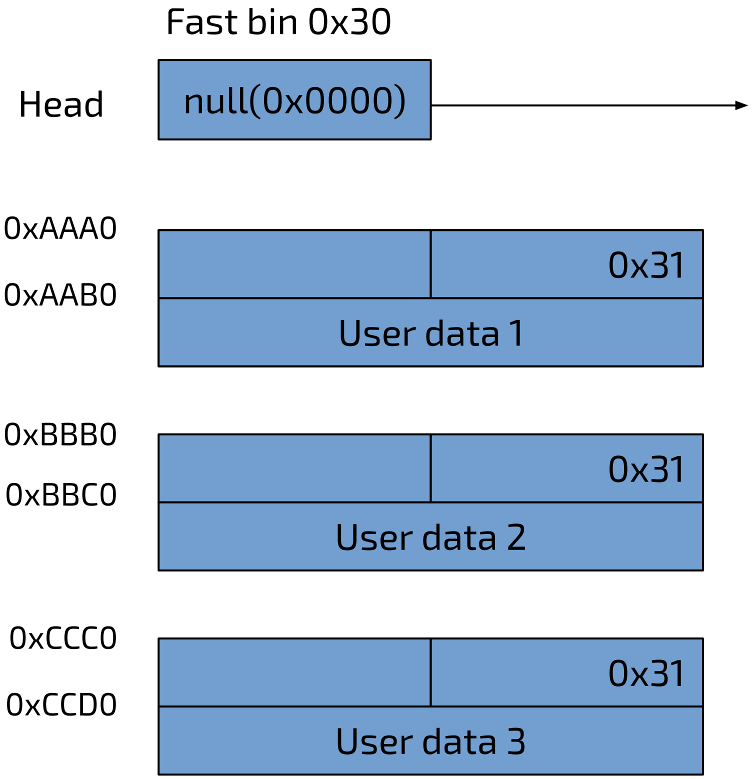


Fast Bin Snapshot

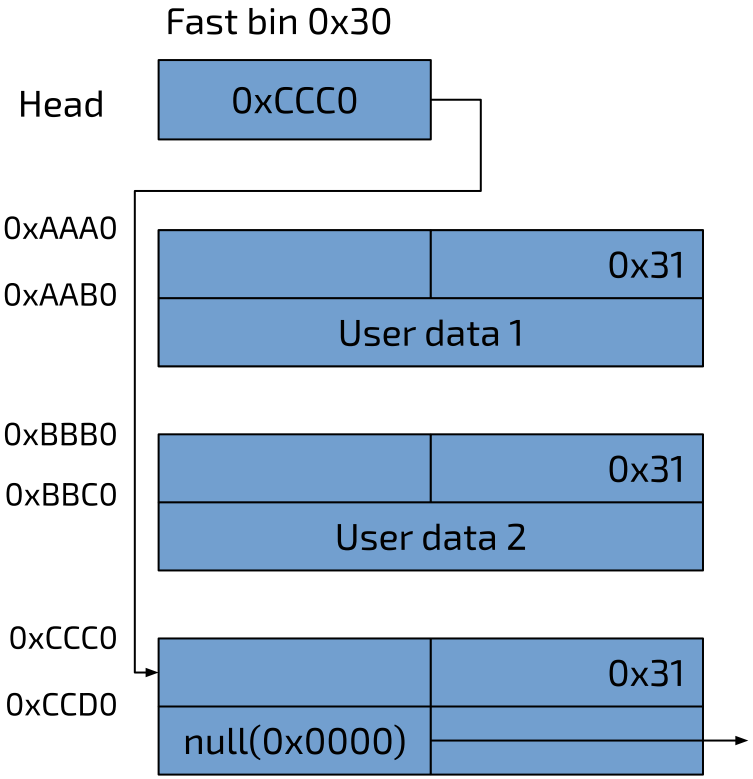
Fast Bins


Fast bin 0x30
Head
null(0x0000)
0xAAA0
0xAAB0
0x31
User data 1
0xBBB0
0xBBC0
0x31
User data 2
0xCCC0
0xCCD0
0x31
User data 3

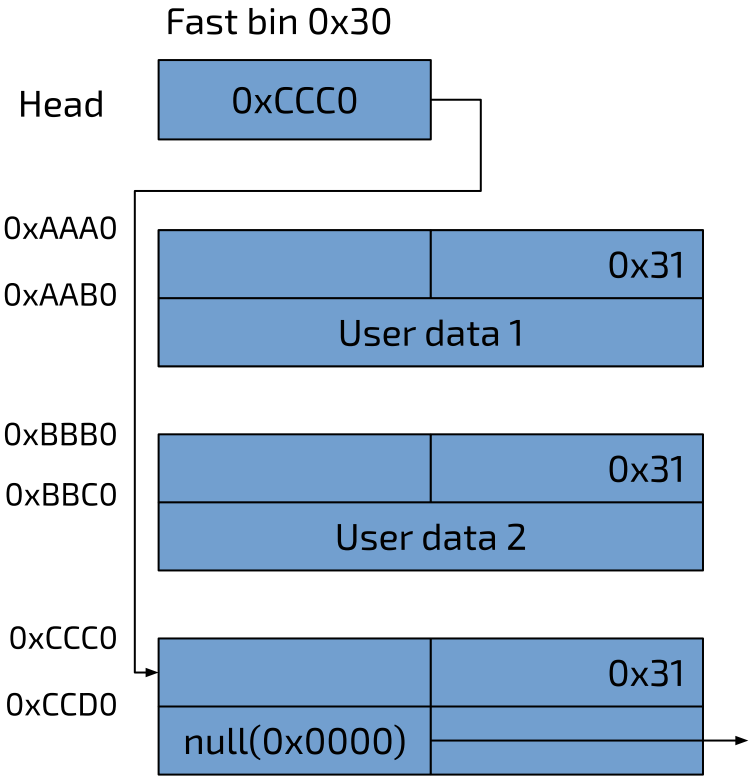
Fast Bins: free(0xCCD0) - Before



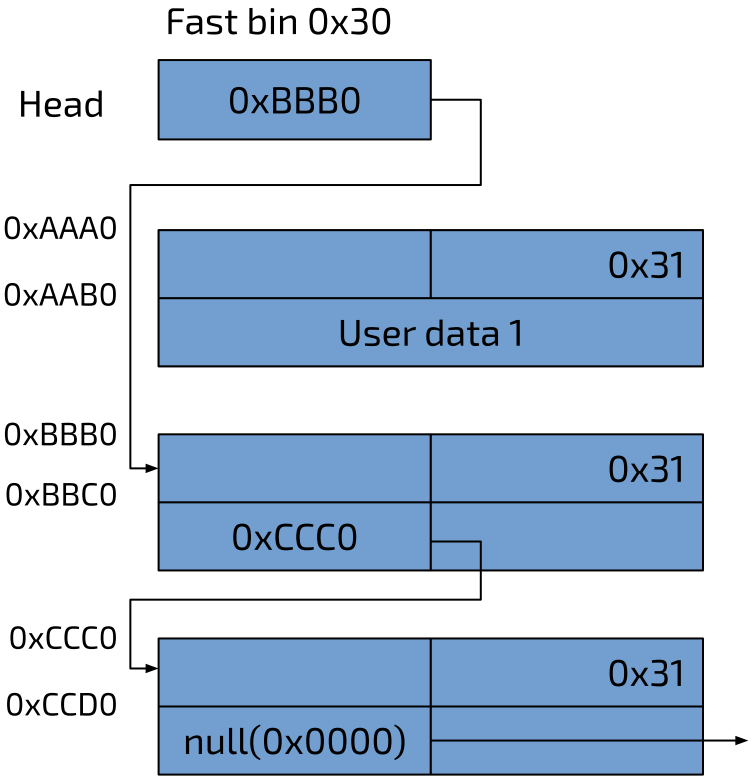
Fast Bins: free(0xCCD0) - After



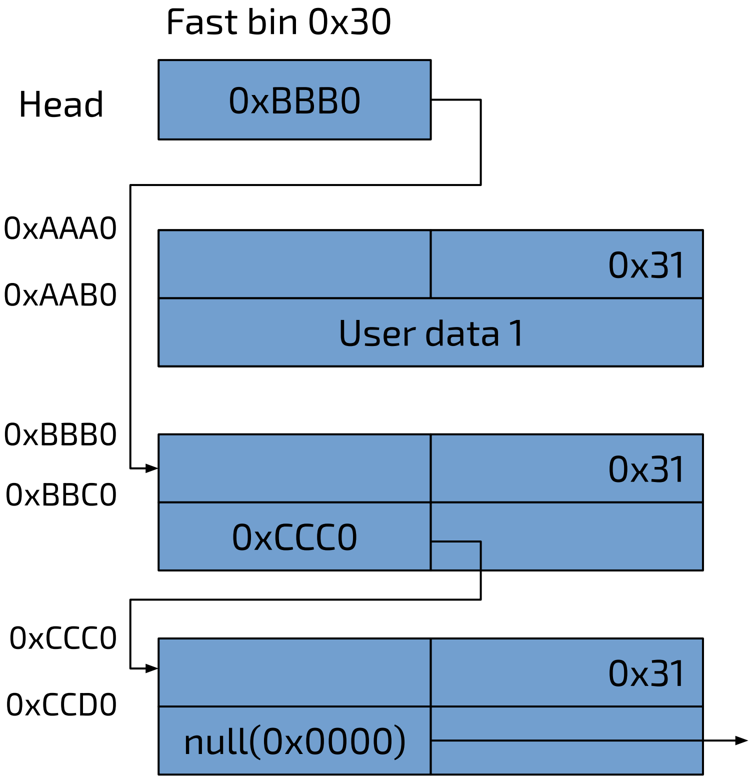
Fast Bins: free(0xBBC0) - Before



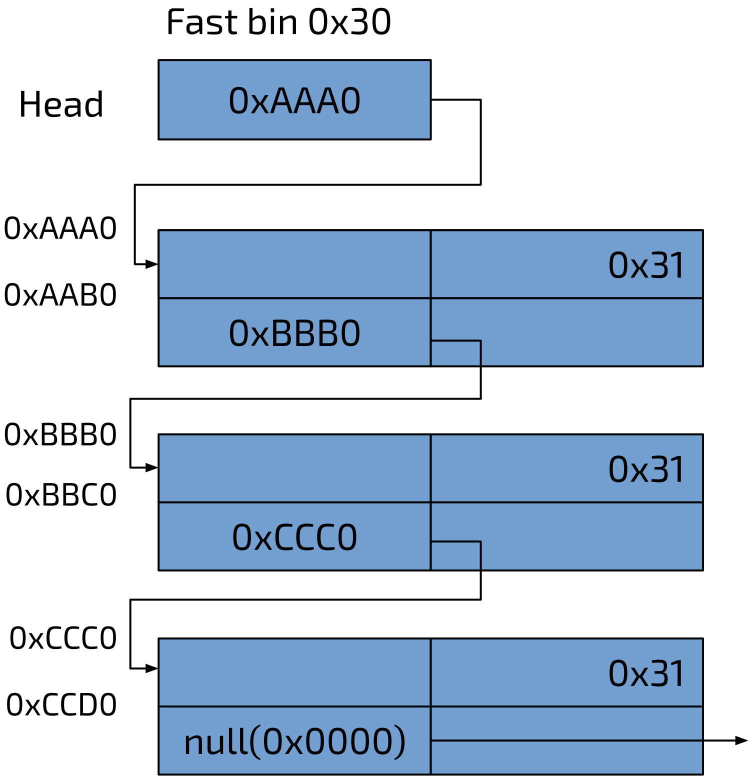
Fast Bins: free(0xBBC0) - After



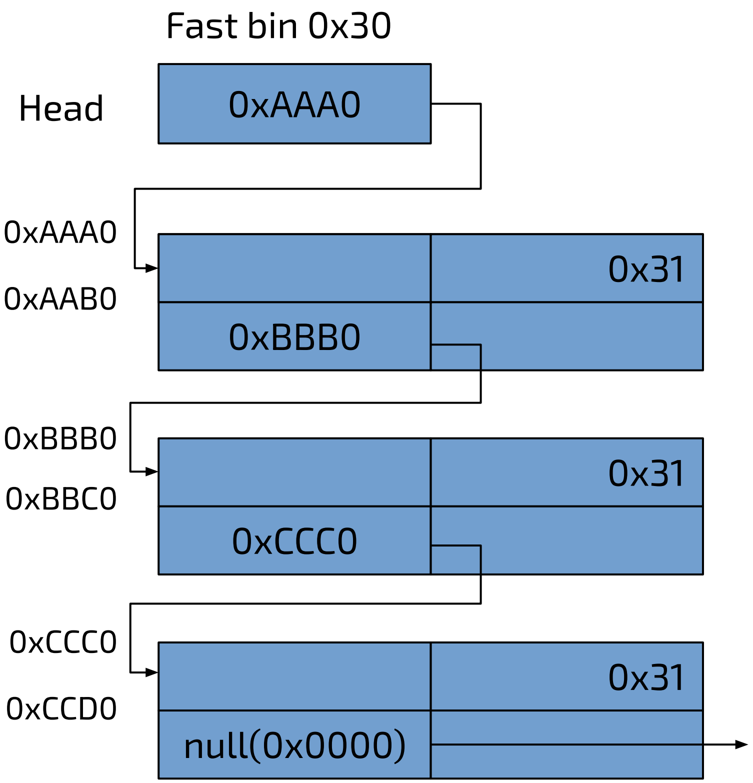
Fast Bins: free(0xAAB0) - Before



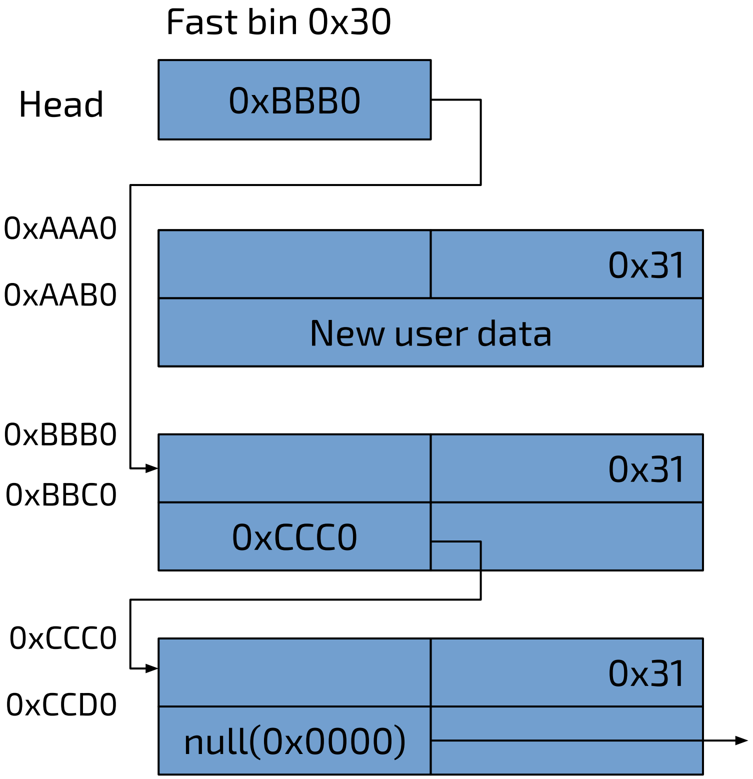
Fast Bins: free(0xAAB0) - After



Fast Bins: malloc(0x20) - Before



Fast Bins: malloc(0x20) - After

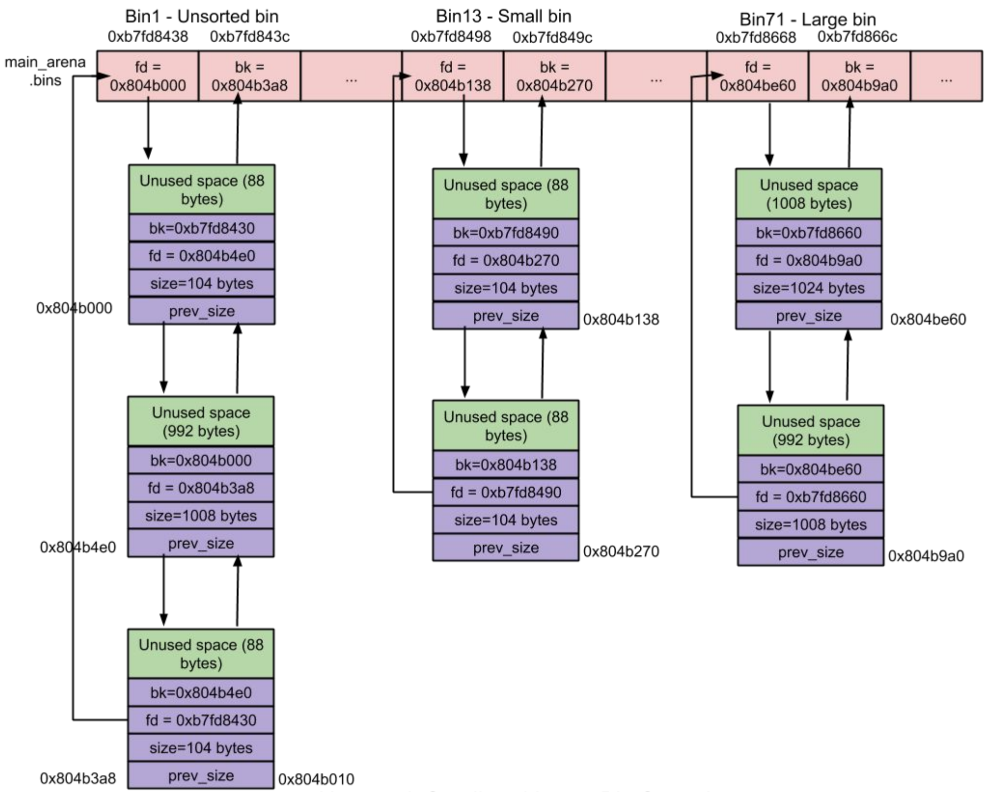


Let’s see it in memory!

Unsorted Bin

- Any freed chunk with size >= 0xA0 ends up in the unsorted bin
- Managed as a double linked list
- When a chunk in the unsorted bin is not able to satisfy a malloc request (e.g., malloc(0x200) but the freed chunk has size 0x100), the chunk in the unsorted bin is moved to the proper small or large bin
- Unsorted bin is like a middle ground

Bins (Unsorted, Small, Large)



Unsorted, Small and Large Bin Snapshot

Let's see it in memory!

Vulnerability after Allocation

- Good old buffer overflow :D
- Overflows on:
  - Metadata and content of the next chunks (in memory)
  - Top chunk (House of force)
- Potential leaks if the buffer is not memset to 0 (calloc solves this problem)

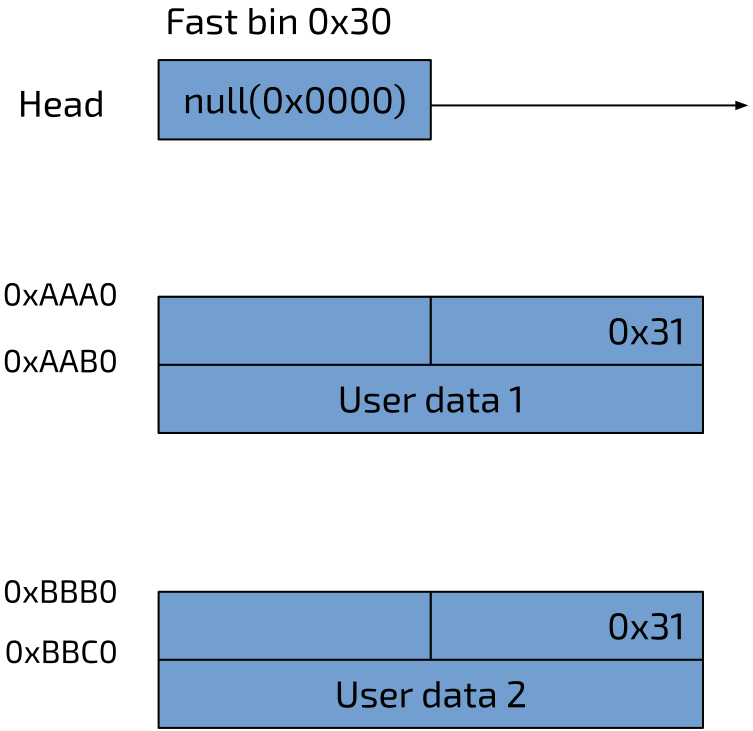
Vulnerability after Deallocation

- Pointer should be set to 0 after free, otherwise it may occur:
  - Leakage of the bins' pointers
  - Corruption of the bins' pointers
  - Multiple pointers to the same chunk in memory
  - Double free (Fastbin attack)

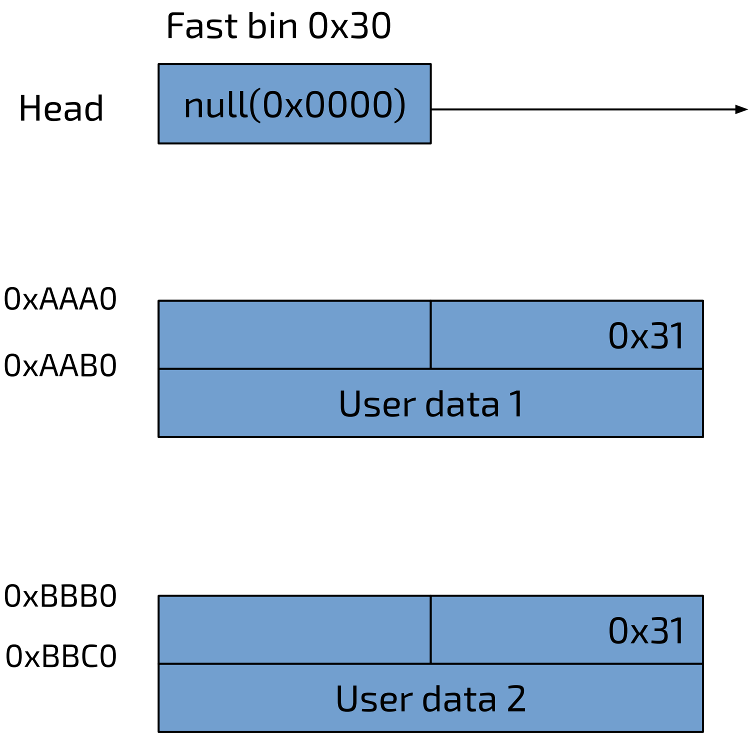
Fast Bin Attack

- Allocate twice the same chunk
- Allocate an almost arbitrary chunk

Fast Bin Attack

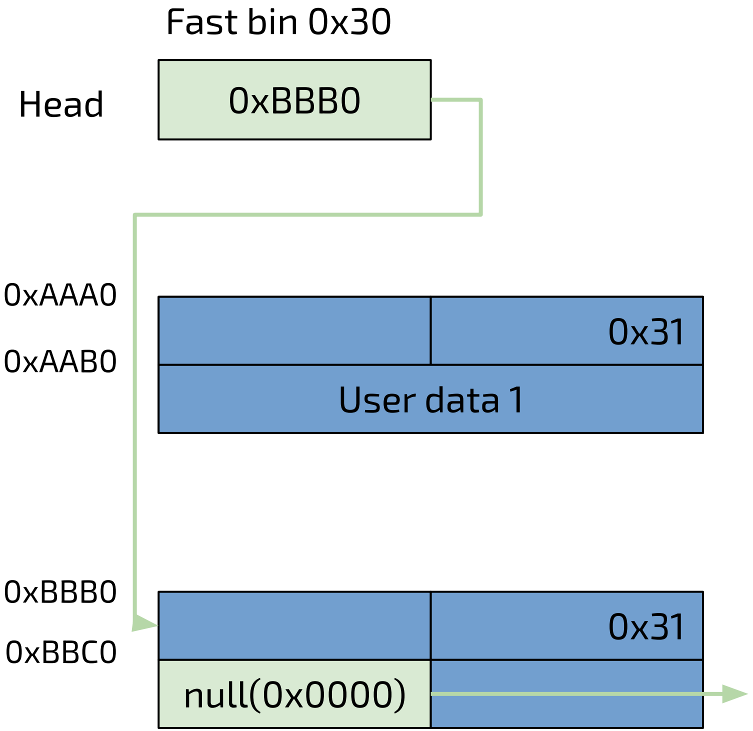


Fast Bin Attack



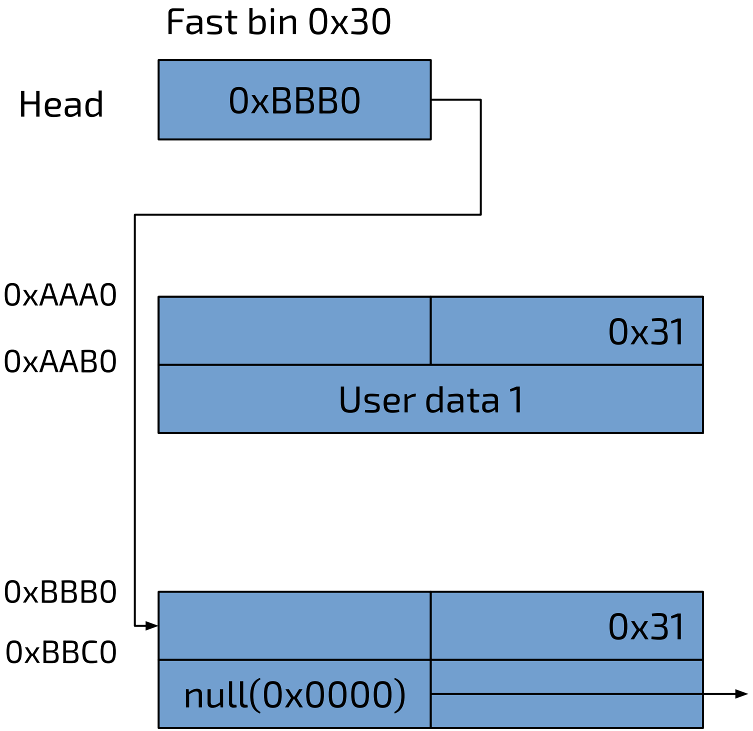

```
free(0xBBC0)
```

Fast Bin Attack



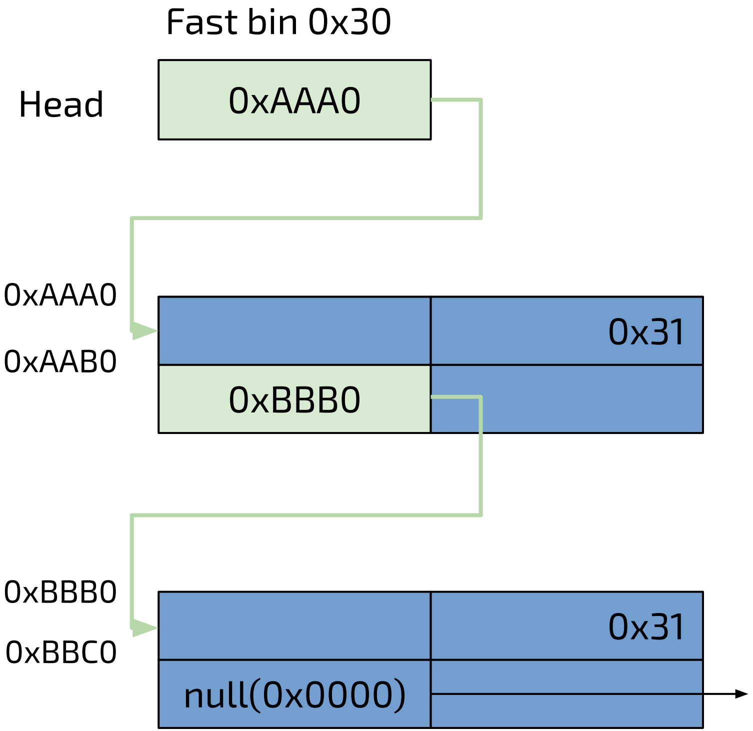
`free(0xBBC0)`

Fast Bin Attack



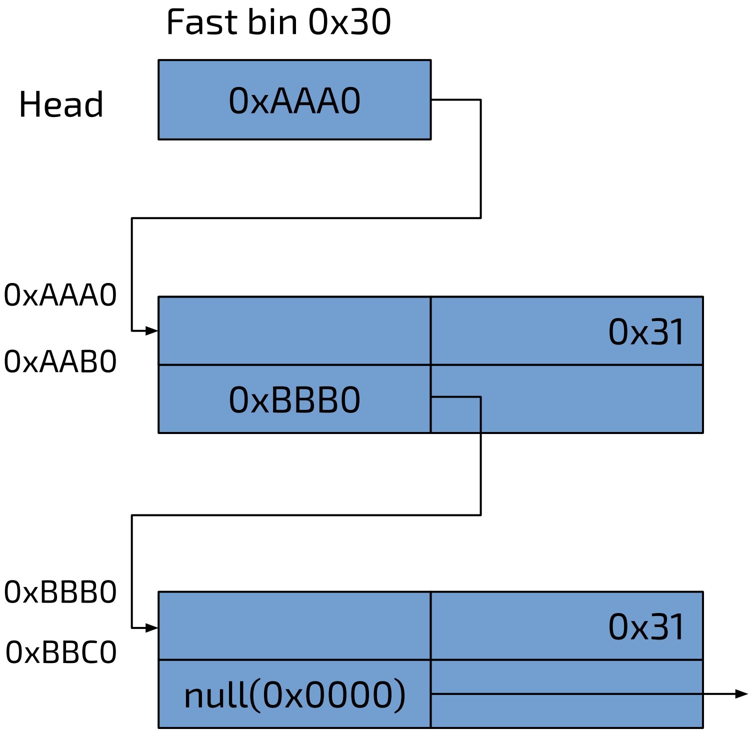

```
free(0xAAB0)
```

Fast Bin Attack



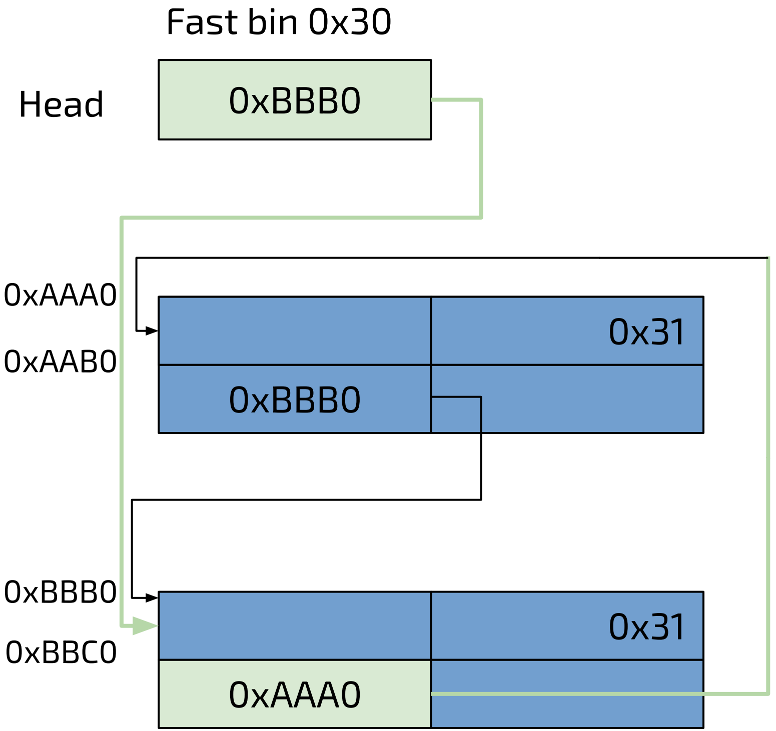
free(0xAAB0)

Fast Bin Attack



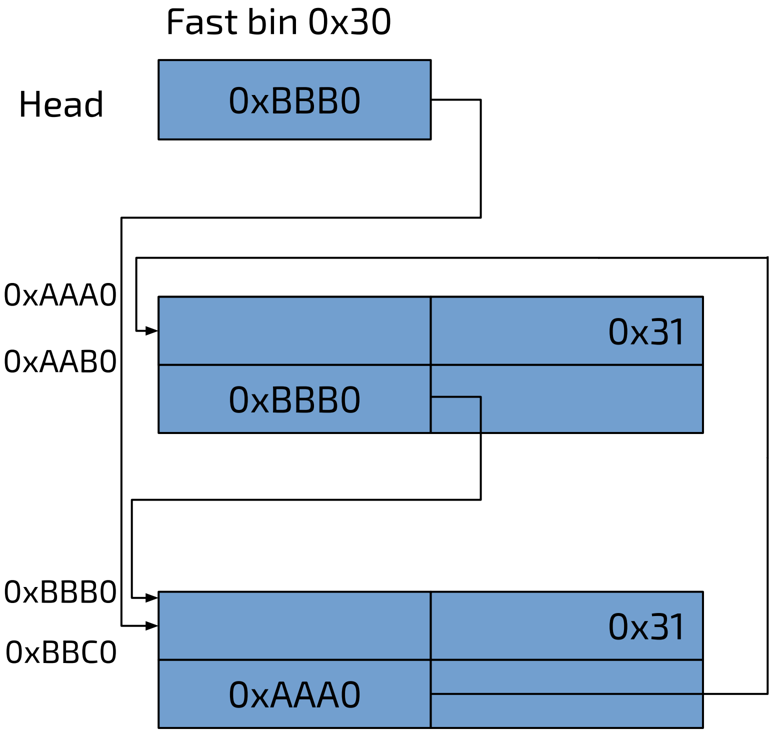
`free(0xBBC0)`

Fast Bin Attack



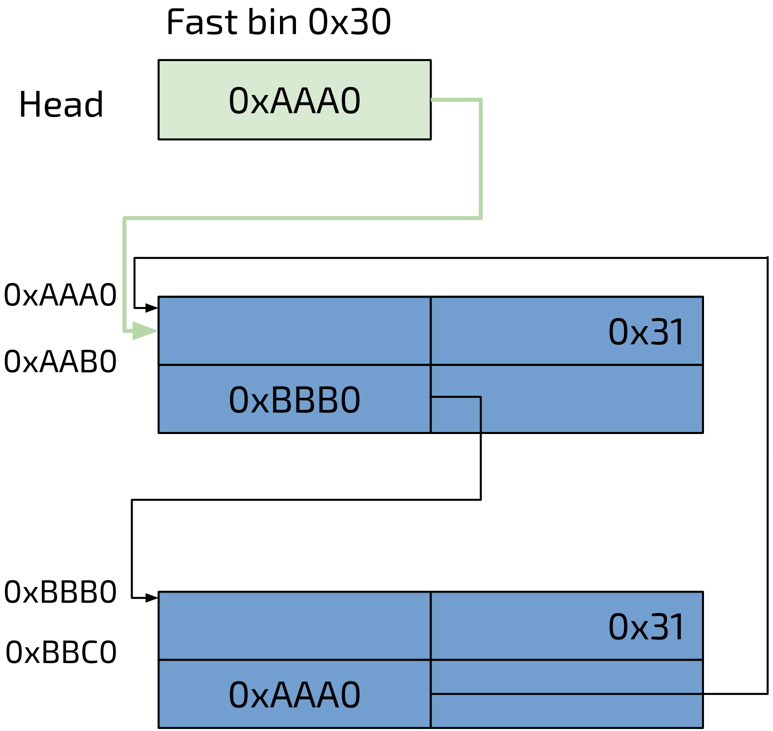
`free(0xBBC0)`

Fast Bin Attack



`malloc(0x20)`

Fast Bin Attack

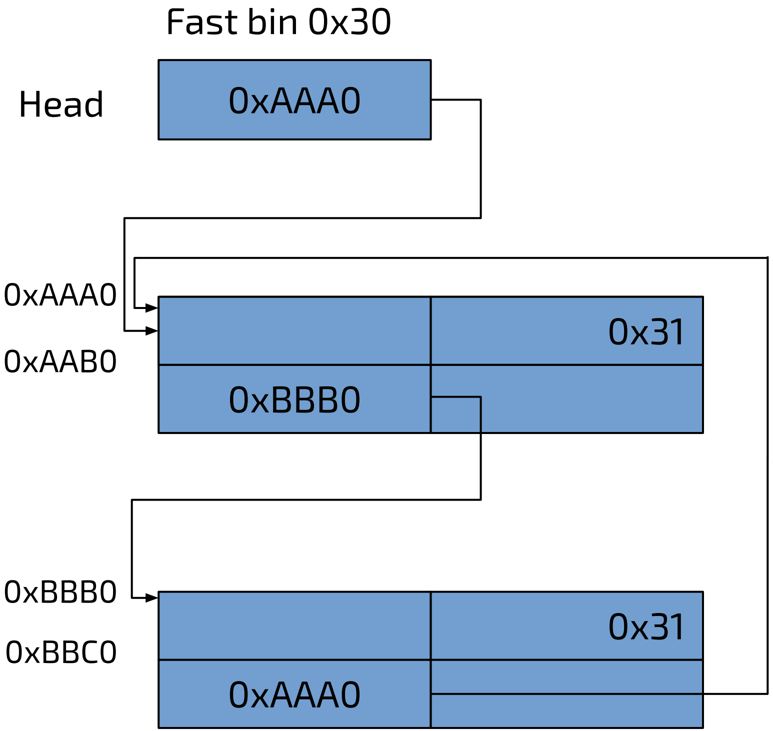


`malloc(0x20)`

0xBBC0 is returned by the malloc.

- Buffer 1: 0xBBC0

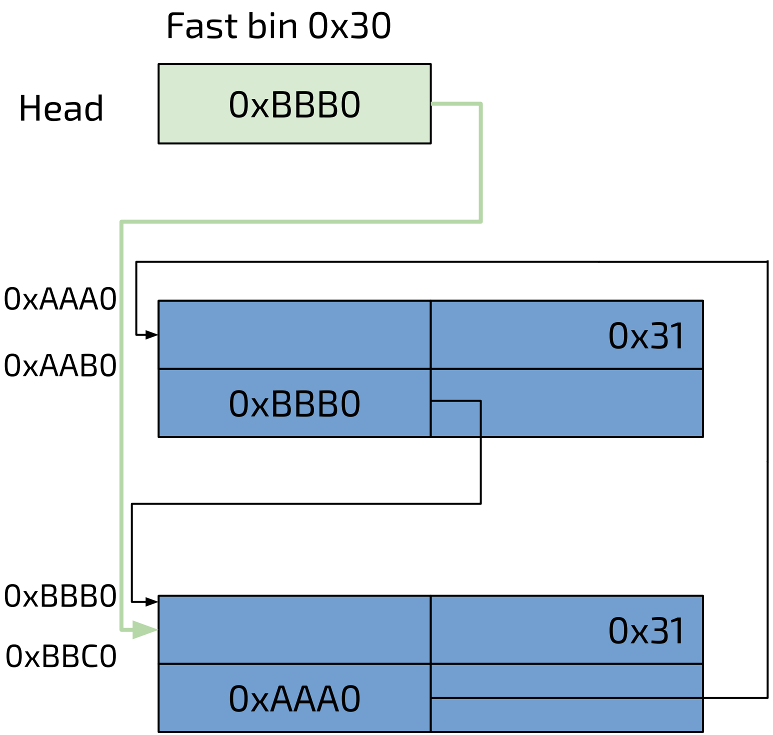
Fast Bin Attack




```
malloc(0x20)
```

- Buffer 1: 0xBBC0

Fast Bin Attack

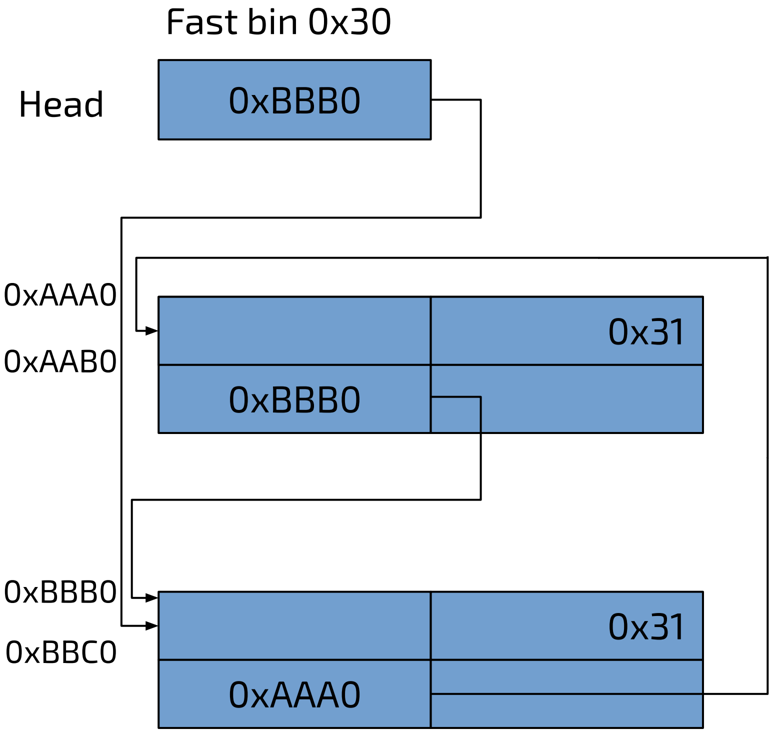


`malloc(0x20)`

0xAAB0 is returned by the malloc.

- Buffer 1: 0xBBC0
- Buffer 2: 0xAAB0

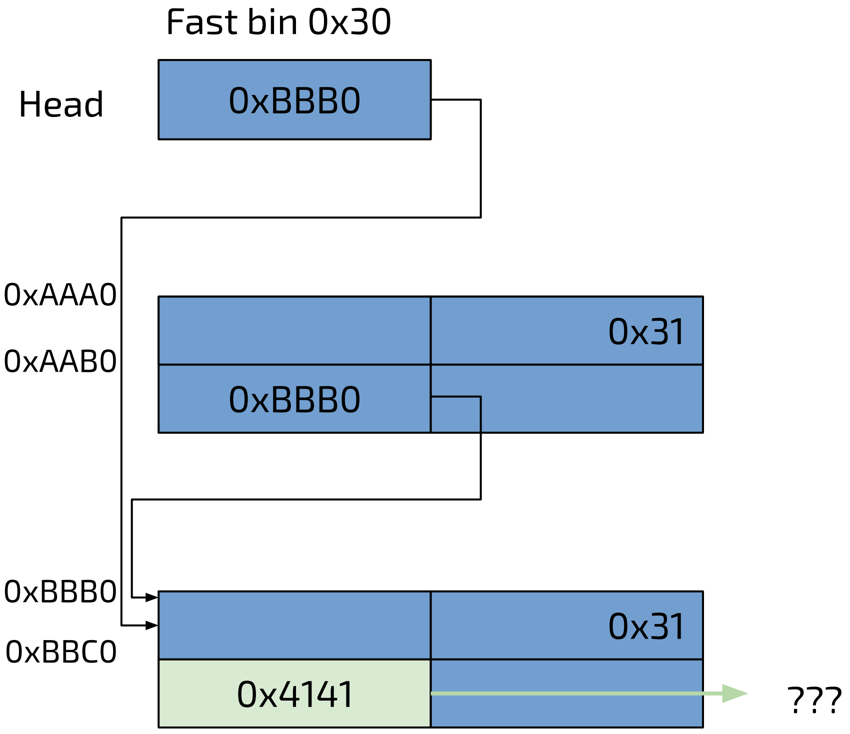
Fast Bin Attack



write Buffer 1

- Buffer 1: 0xBBC0
- Buffer 2: 0xAAB0

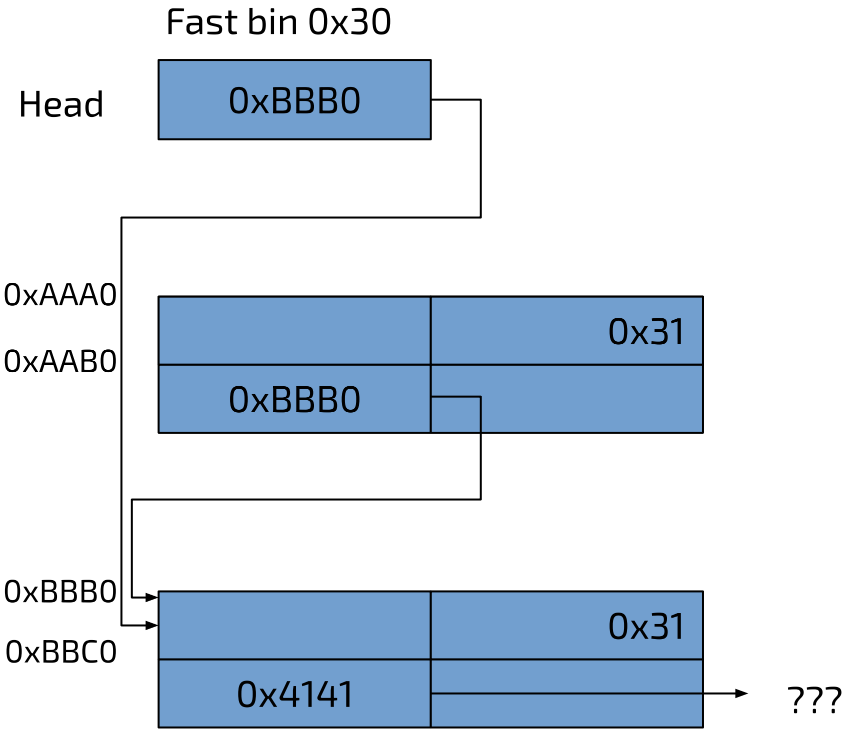
Fast Bin Attack



write Buffer 1

- Buffer 1: 0xBBC0
- Buffer 2: 0xAAB0

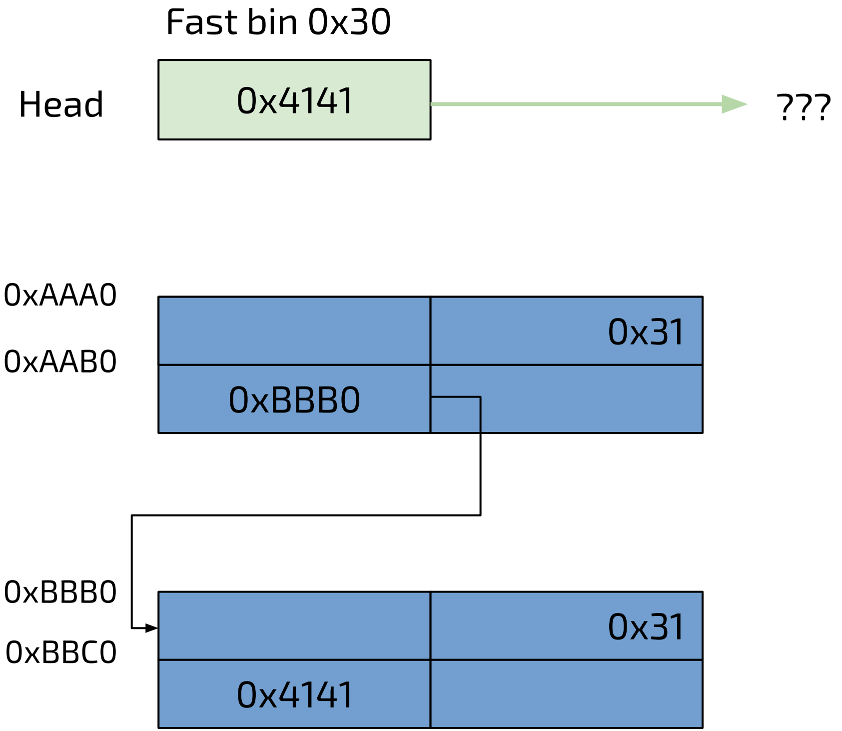
Fast Bin Attack



`malloc(0x20)`

- Buffer 1: 0xBBC0
- Buffer 2: 0xAAB0

Fast Bin Attack: malloc(0x20) - After



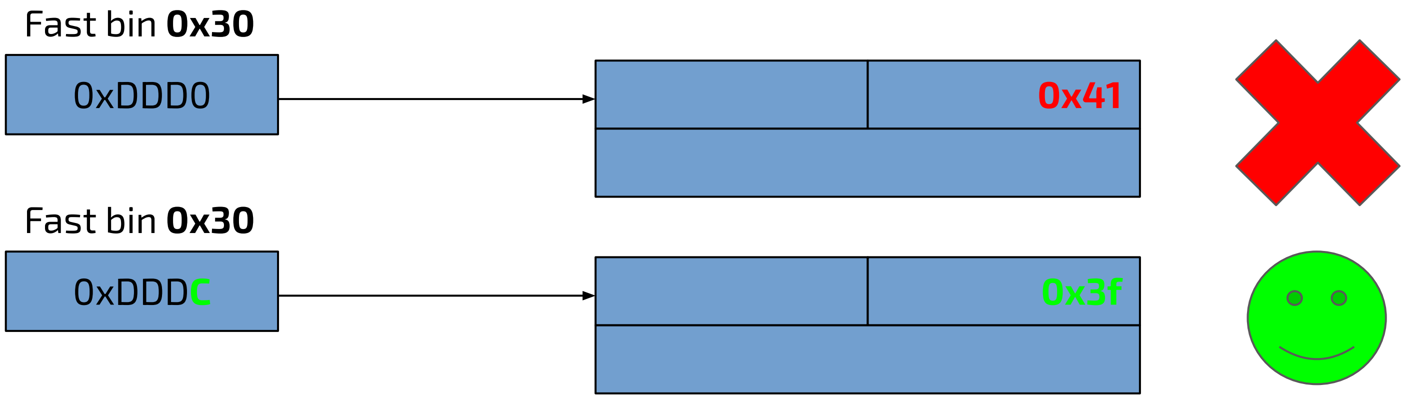
`malloc(0x20)`

0xBBC0 is returned by the malloc.

- Buffer 1: 0xBBC0
- Buffer 2: 0xAAB0
- Buffer 3: 0xBBC0

Notes

- The arbitrary chunk will be returned if:
  - The memory is mapped (otherwise SEG FAULT)
  - The size of the fake chunk matches with the bin size
- The first 4 bits of the size are not considered
- No requirements on the alignment of the chunk



Load another Library (libc-2.xx.so)

- **env LD_PRELOAD**
  - `LD_PRELOAD=./libc-2.23.so ./binary`
- **`ld.so`**
  - `./ld-2.23.so --library-path ./lib ./binary`
  - `lib contains libc.so.6`
- **`patchelf`** (https://github.com/NixOS/patchelf)
  - `patchelf --set-interpreter ./ld-2.23.so --replace-needed libc.so.6 ./libc-2.23.so ./binary`
- **YOLO** `(Do not use this!)`
  - `Replace system library`
- **`Docker/Virtual Machine`**

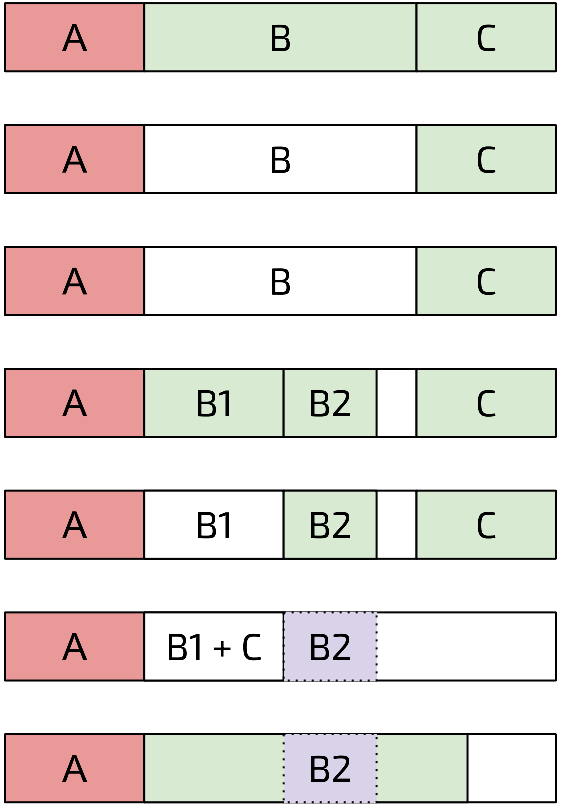
Poison Null Byte

- Allocate two chunks that overlap

Poison Null Byte

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

Poison Null Byte



Initial Setup

Free(B)

Overflow into B. Sizes goes from 0x208 to 0x200.
prev_size is not update

Allocate two chunks into old B. First not a fastbin

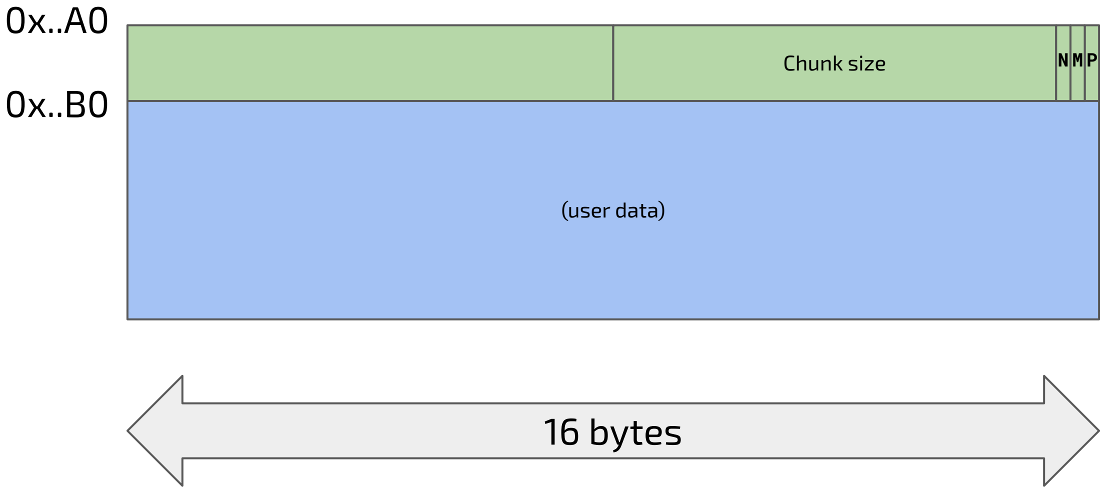
Free(B1)

Free(C) trigger the merge with the previous chunk

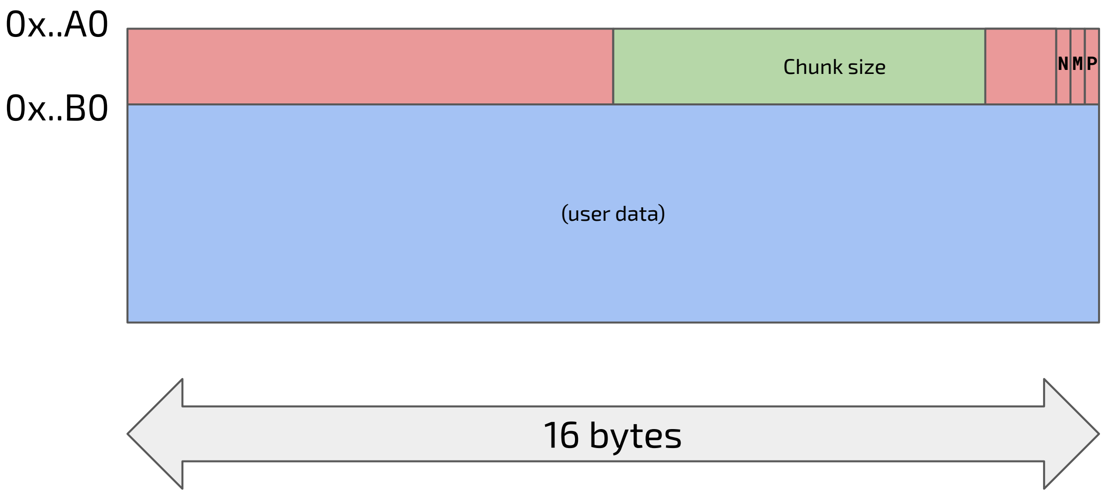
the next allocation will overlap with B2

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

Chunk



Null Byte



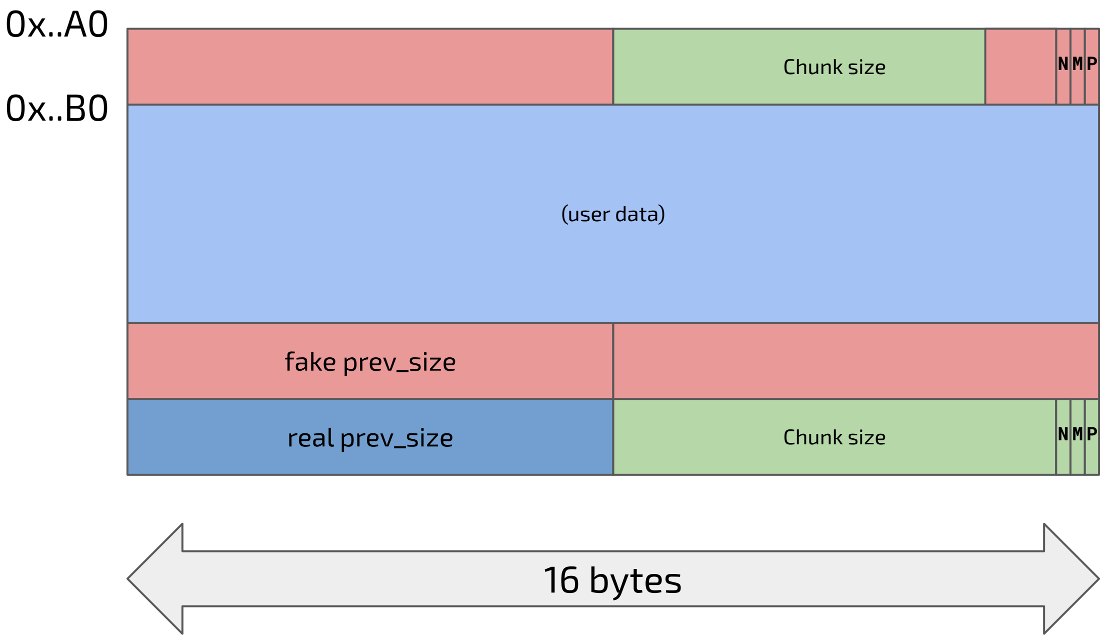
Poison Null Byte “FIX”

```
if (__builtin_expect (chunksize(P) != prev_size (next_chunk(P)), 0))
      malloc_printerr ("corrupted size vs. prev_size");
```

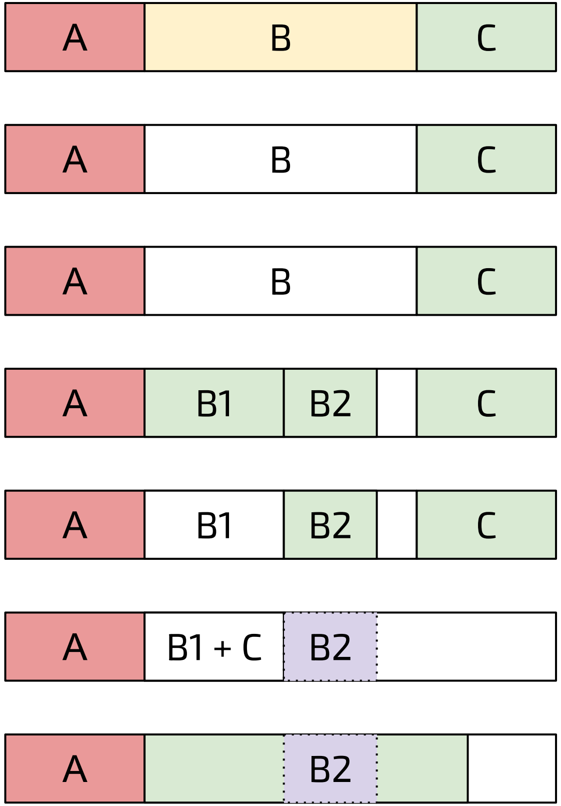
Poison Null Byte “FIX”

```
if (__builtin_expect (chunksize(P) != prev_size (next_chunk(P)), 0))
      malloc_printerr ("corrupted size vs. prev_size");


/* Size of the chunk below P.  Only valid if !prev_inuse (P).  */
 #define prev_size(p) ((p)->mchunk_prev_size)

/* Ptr to next physical malloc_chunk. */
 #define next_chunk(p) ((mchunkptr) (((char *) (p)) + chunksize (p)))
```

fake prev_size



Poison Null Byte Fixed



Initial Setup (B chunk needs to contain fake prev_size)

Free(B)

Overflow into B. Sizes goes from 0x208 to 0x200.
prev_size is not update

Allocate two chunks into old B. First not a fastbin

Free(B1)

Free(C) trigger the merge with the previous chunk

the next allocation will overlap with B2

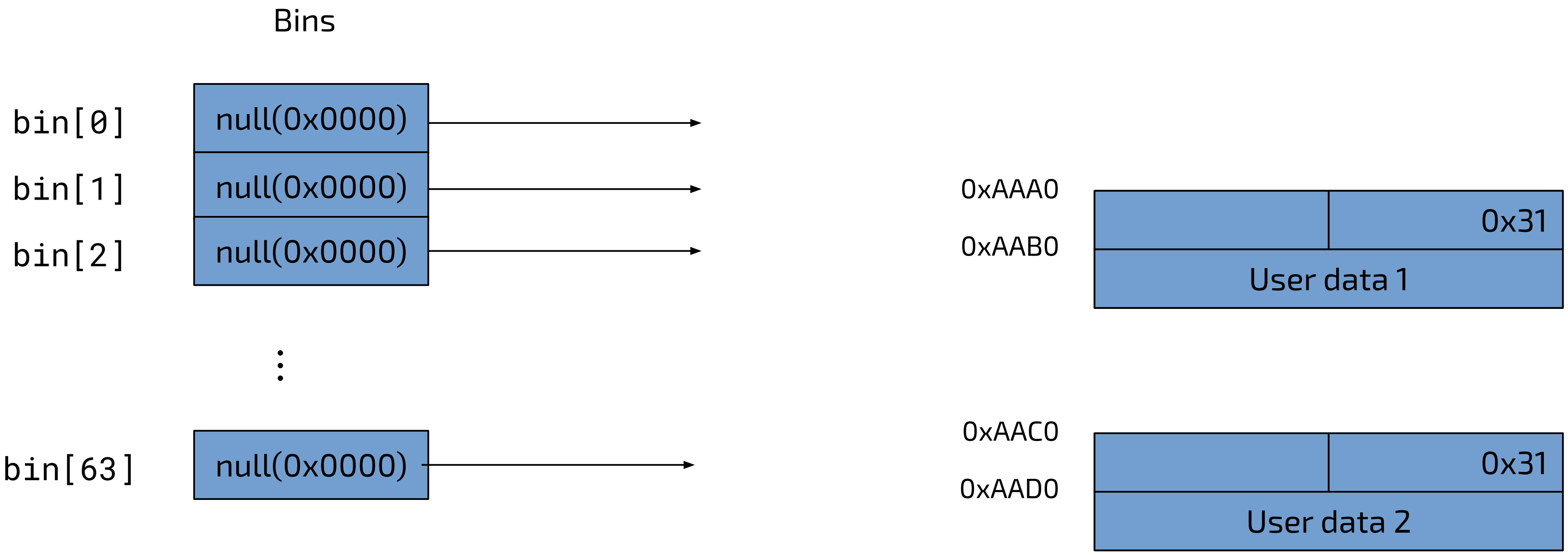
https://heap-exploitation.dhavalkapil.com/attacks/shrinking_free_chunks.html

T-Cache

T-Cache

- **Cache** for chunk size **< 0x500**
- **LIFO**
- **New Attack** Vector
- Need to **bypass** it for attacks on other bins
- You can exploit T-Cache for better **HEAP Manipulation**
- **64 bins**
- **7** chunks per bin as cache

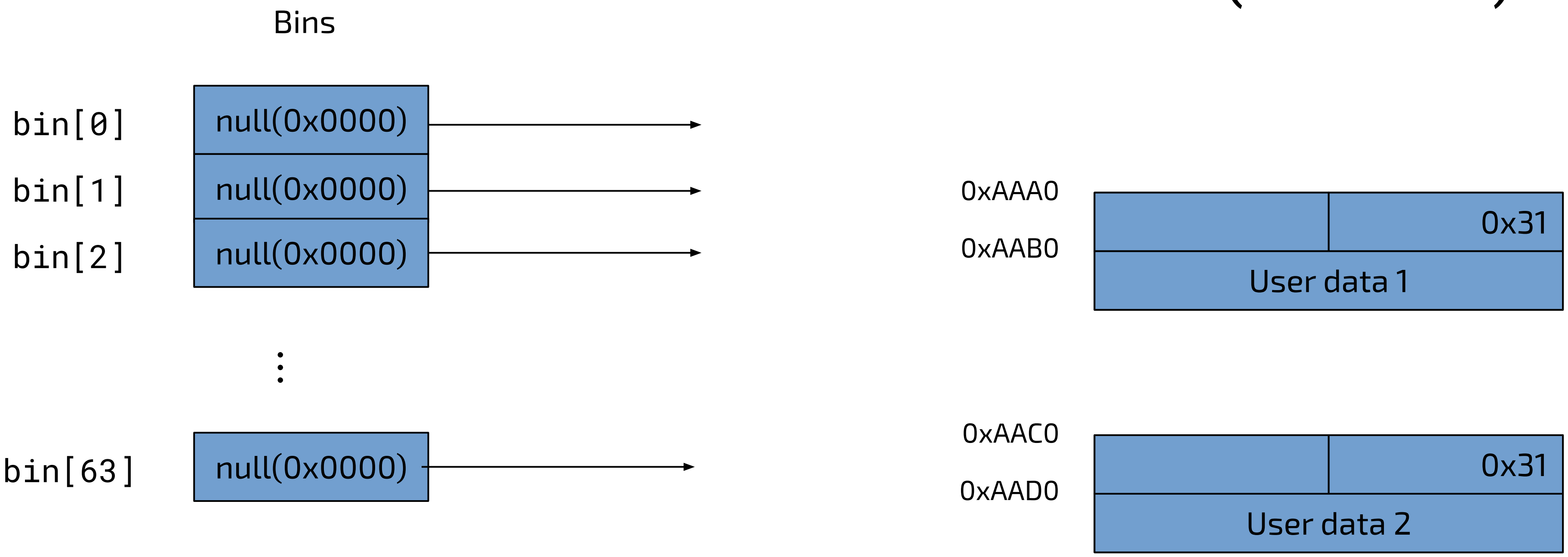
T-Cache



Bins
bin[0]
null(0x0000)
bin[1]
null(0x0000)
bin[2]
null(0x0000)
bin[63]
null(0x0000)
0xAAA0
0xAAB0
0x31
User data 1
0xAAC0
0xAAD0
0x31
User data 2

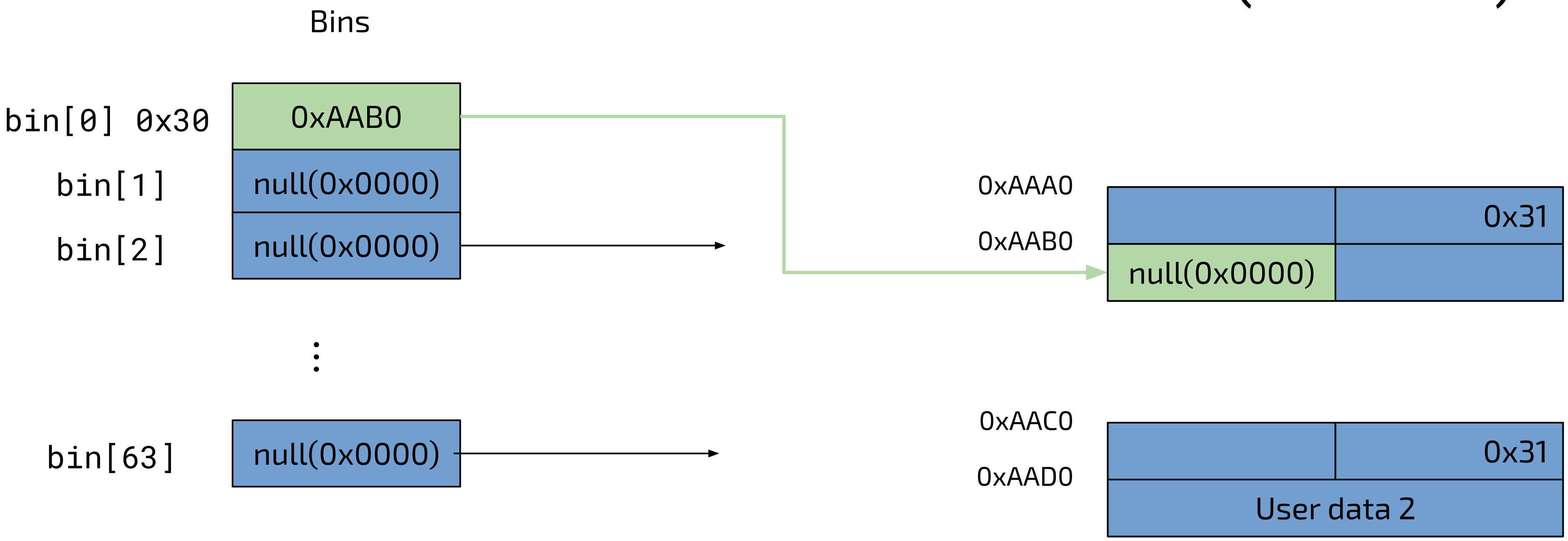
T-Cache

free(0xAAB0)



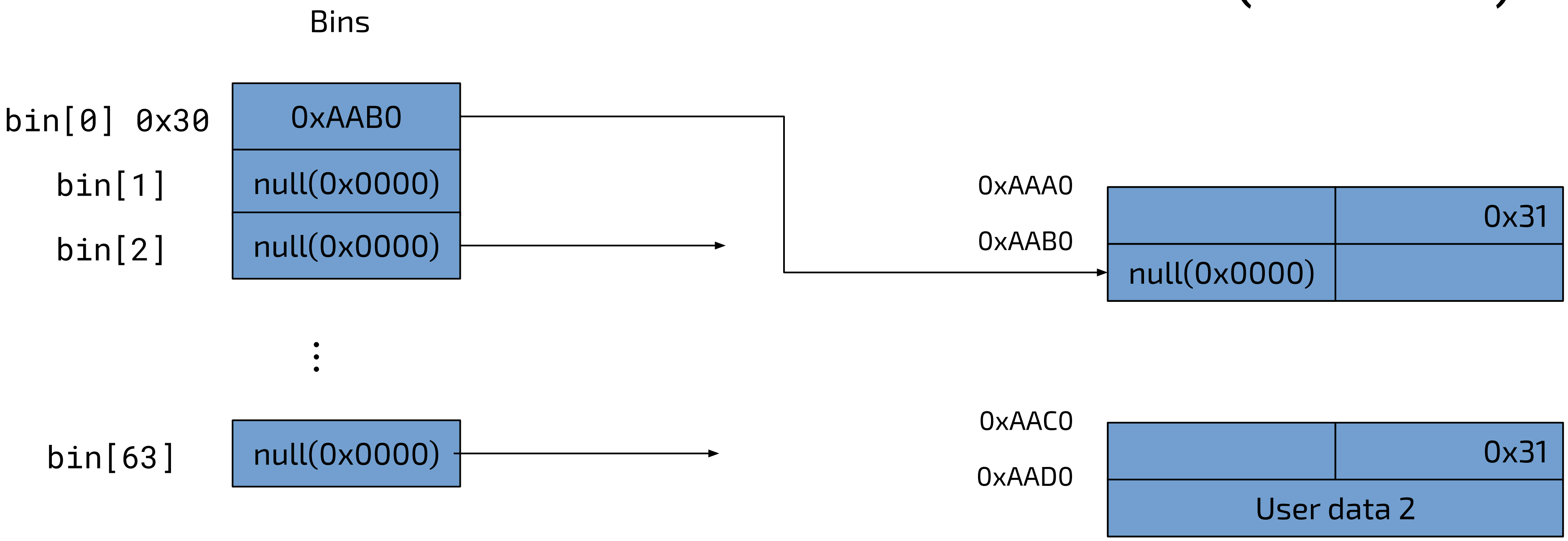
T-Cache

free(0xAAB0)

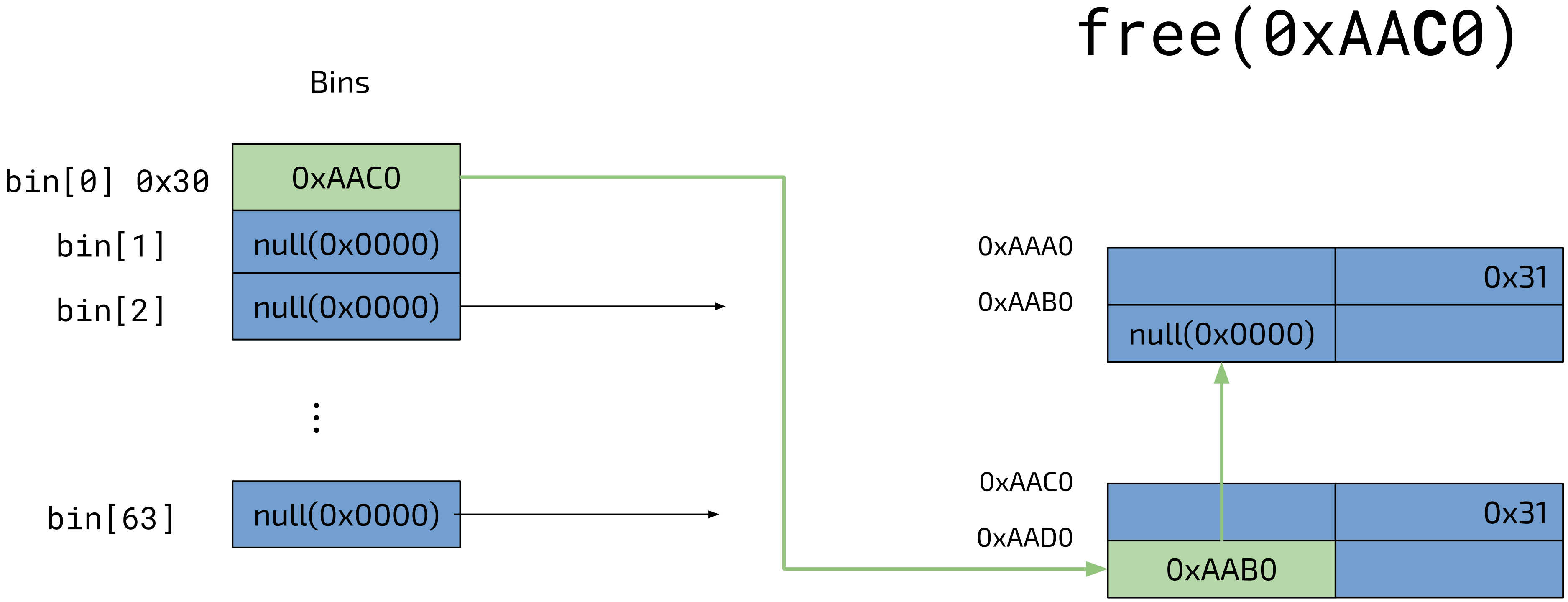


T-Cache

free(0xAAC0)



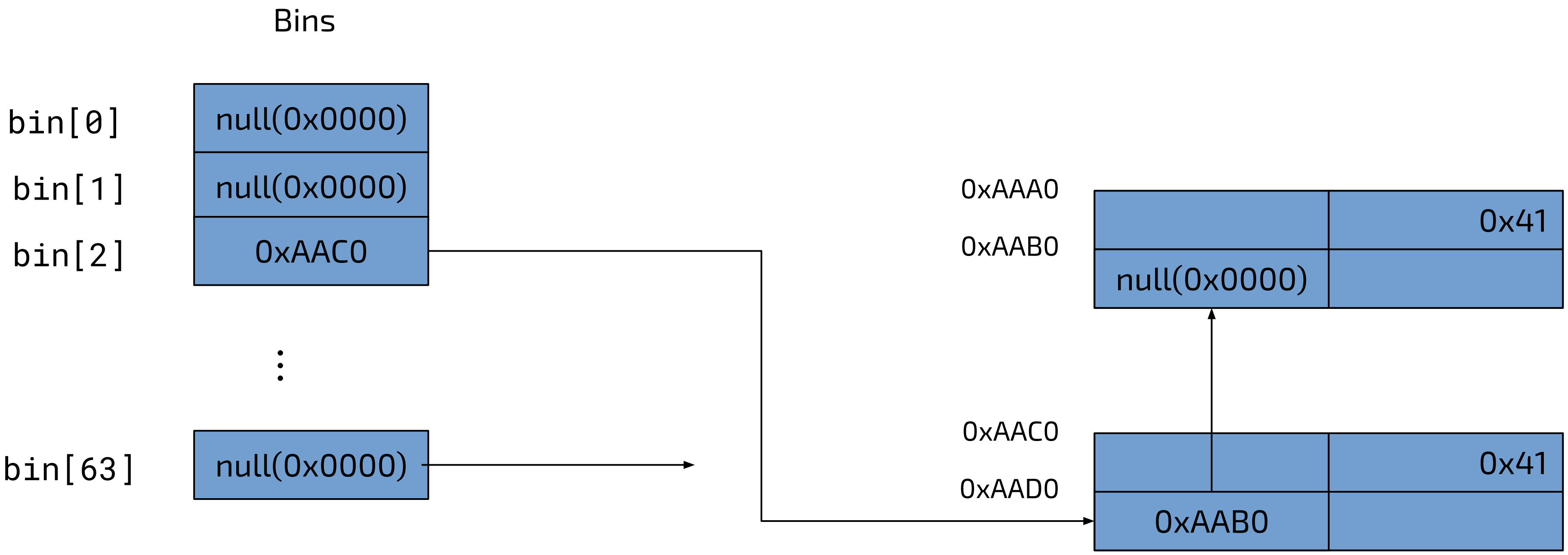
T-Cache


free(0xAAC0)
Bins
bin[0] 0x30
0xAAC0
bin[1]
null(0x0000)
bin[2]
null(0x0000)
bin[63]
null(0x0000)
0xAAA0
0xAAB0
0x31
null(0x0000)
0xAAC0
0xAAD0
0x31
0xAAB0

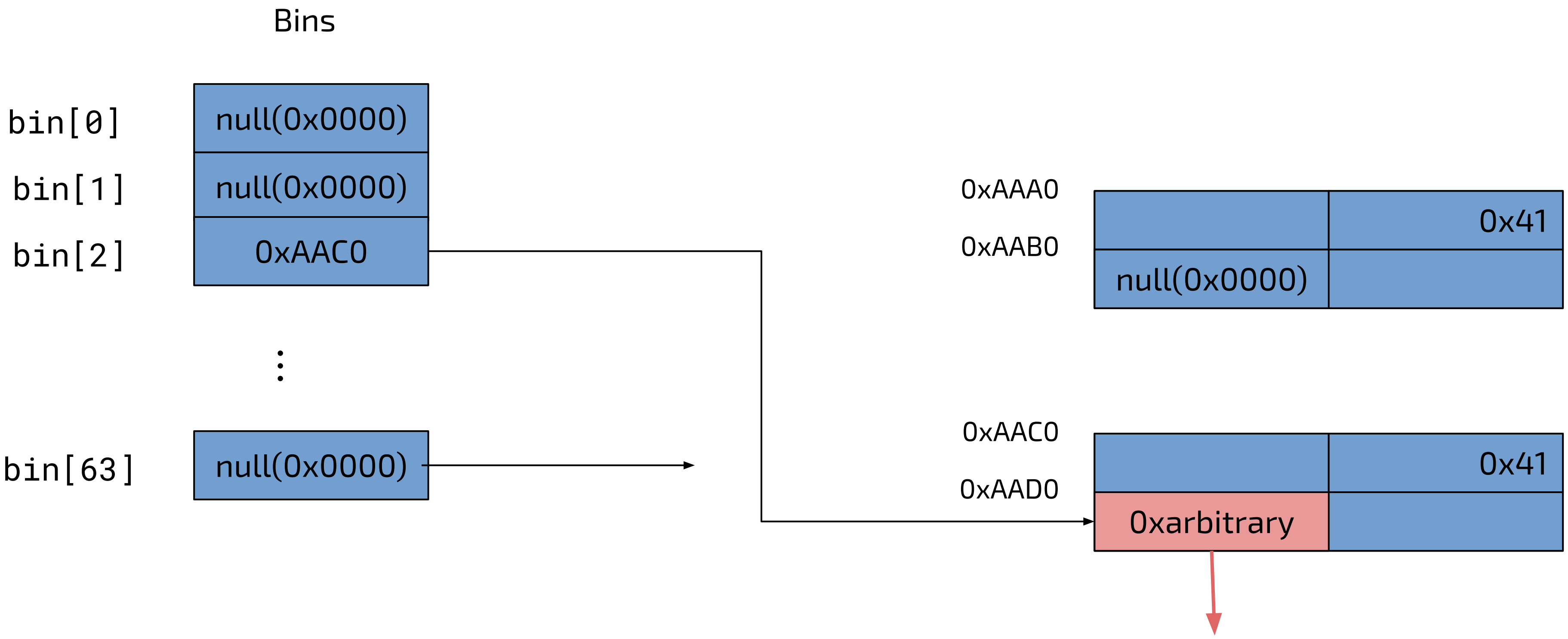
T-Cache Poison

- Allocate an arbitrary chunk

T-Cache (need at least 2 elements in the list)

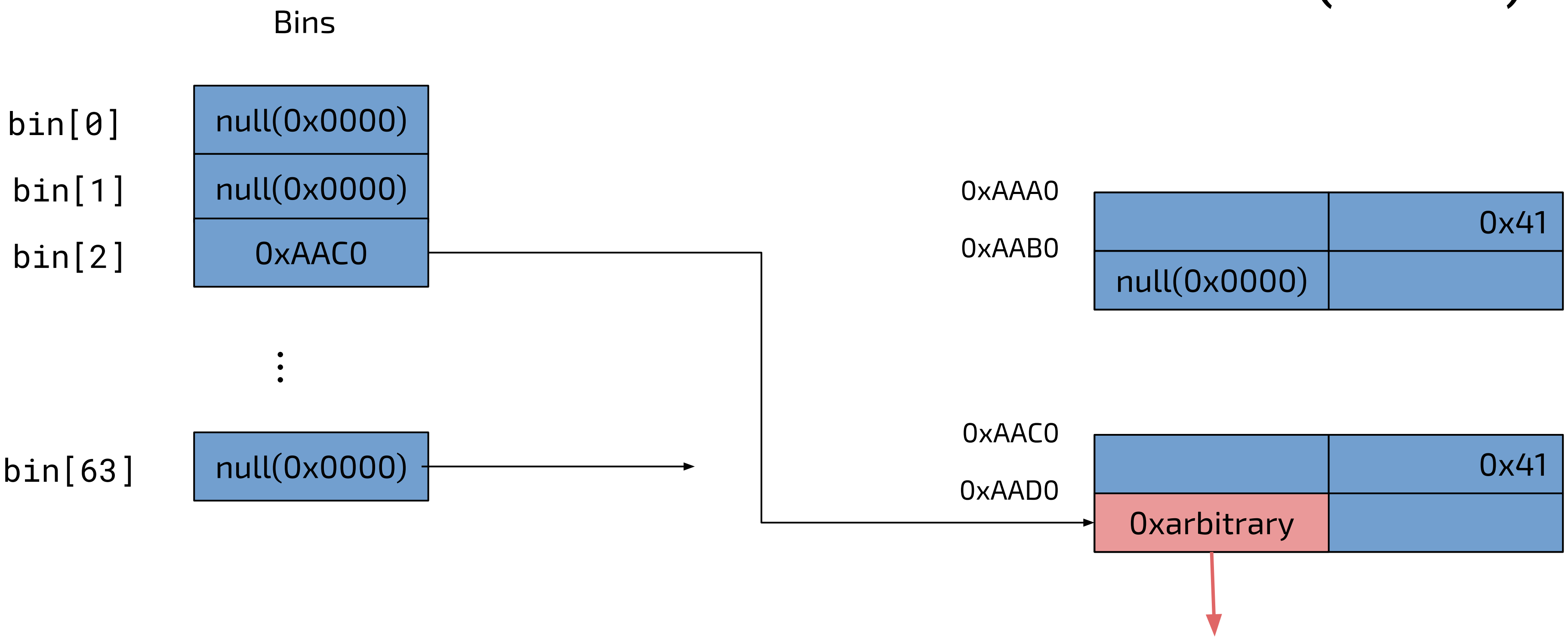


T-Cache


Bins
bin[0]
null(0x0000)
bin[1]
null(0x0000)
bin[2]
0xAAC0
bin[63]
null(0x0000)
0xAAA0
0xAAB0
0x41
null(0x0000)
0xAAC0
0xAAD0
0x41
0xarbitrary

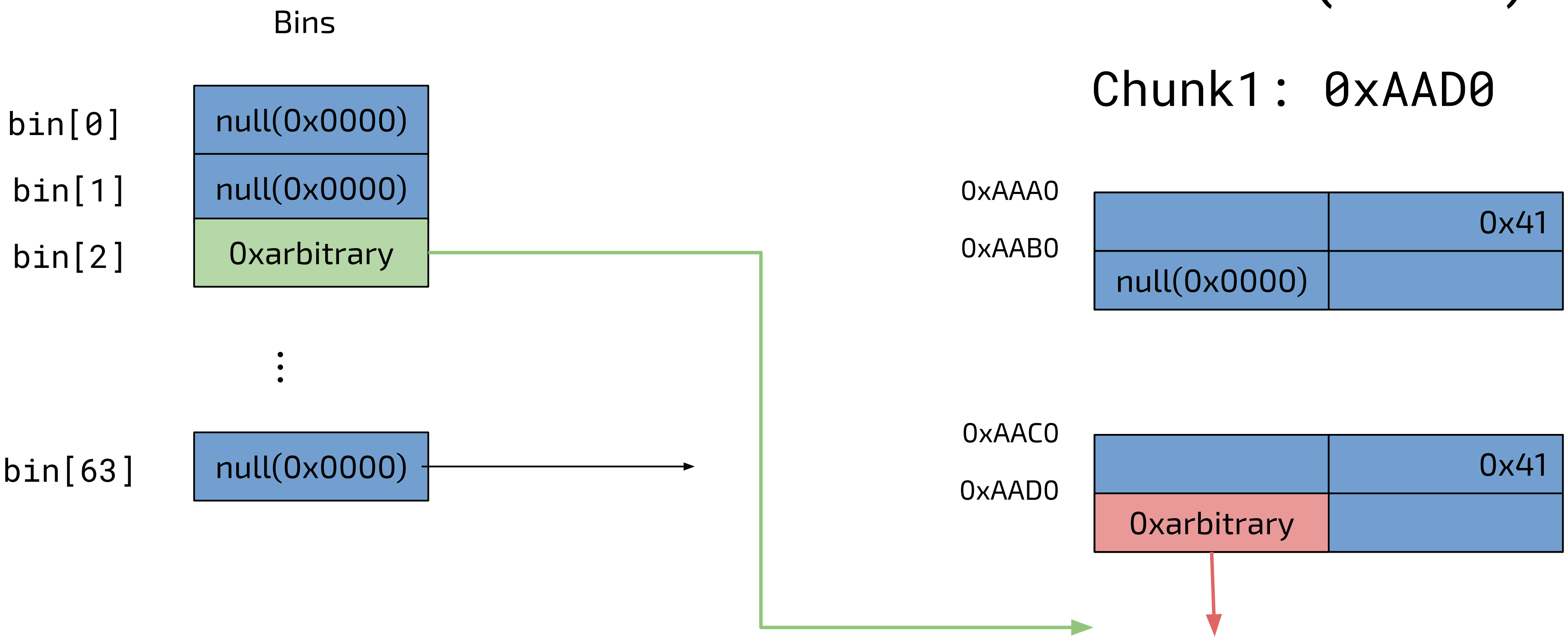
T-Cache

`malloc(0x20)`



T-Cache

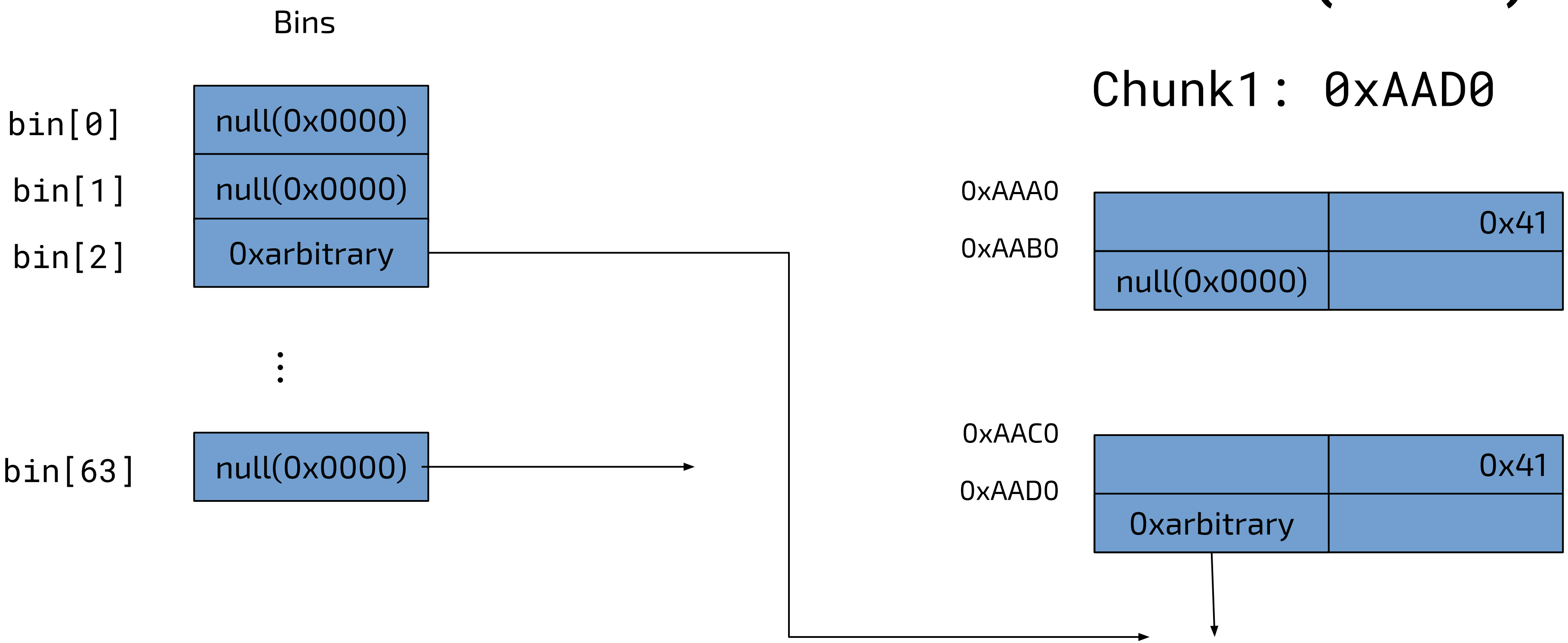
malloc(0x20)



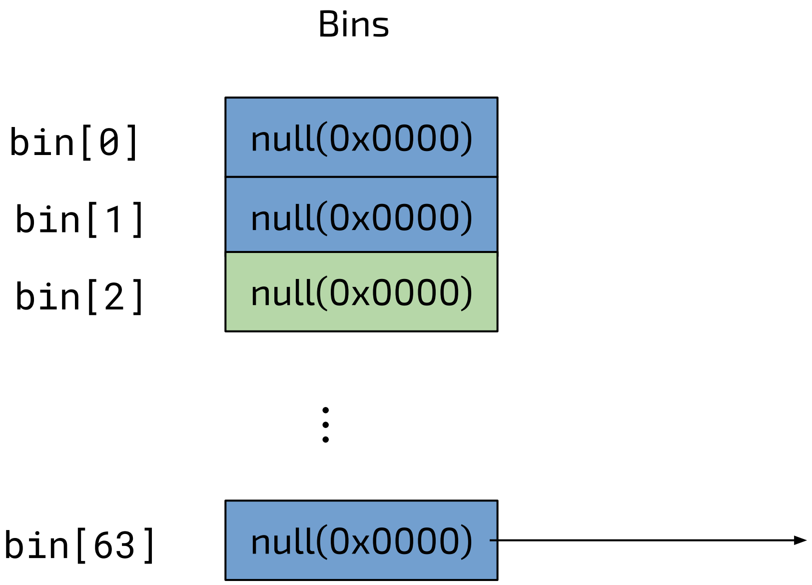
T-Cache

malloc(0x20)

Chunk1: 0xAAD0

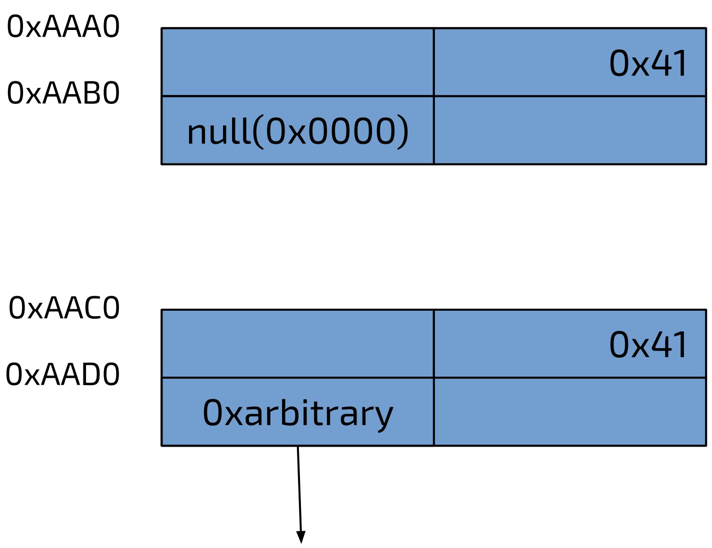


T-Cache



malloc(0x20)

Chunk1: 0xAAD0
Chunk2: **0xarbitrary**



T-Cache Key > glibc 2.29

glibc 2.34

```
tcache_key = random_bits ();
tcache_key = (tcache_key << 32) | random_bits ();
e->key = tcache_key;
```

glibc 2.29 - 2.33

```
e->key = tcache;
```

free

```
if (__glibc_unlikely (e->key == tcache)){
/* very likely a problem make extra checks*/
}
```

T-Cache PTR Protection

glibc >= 2.32

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

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

Best documentation is source code.

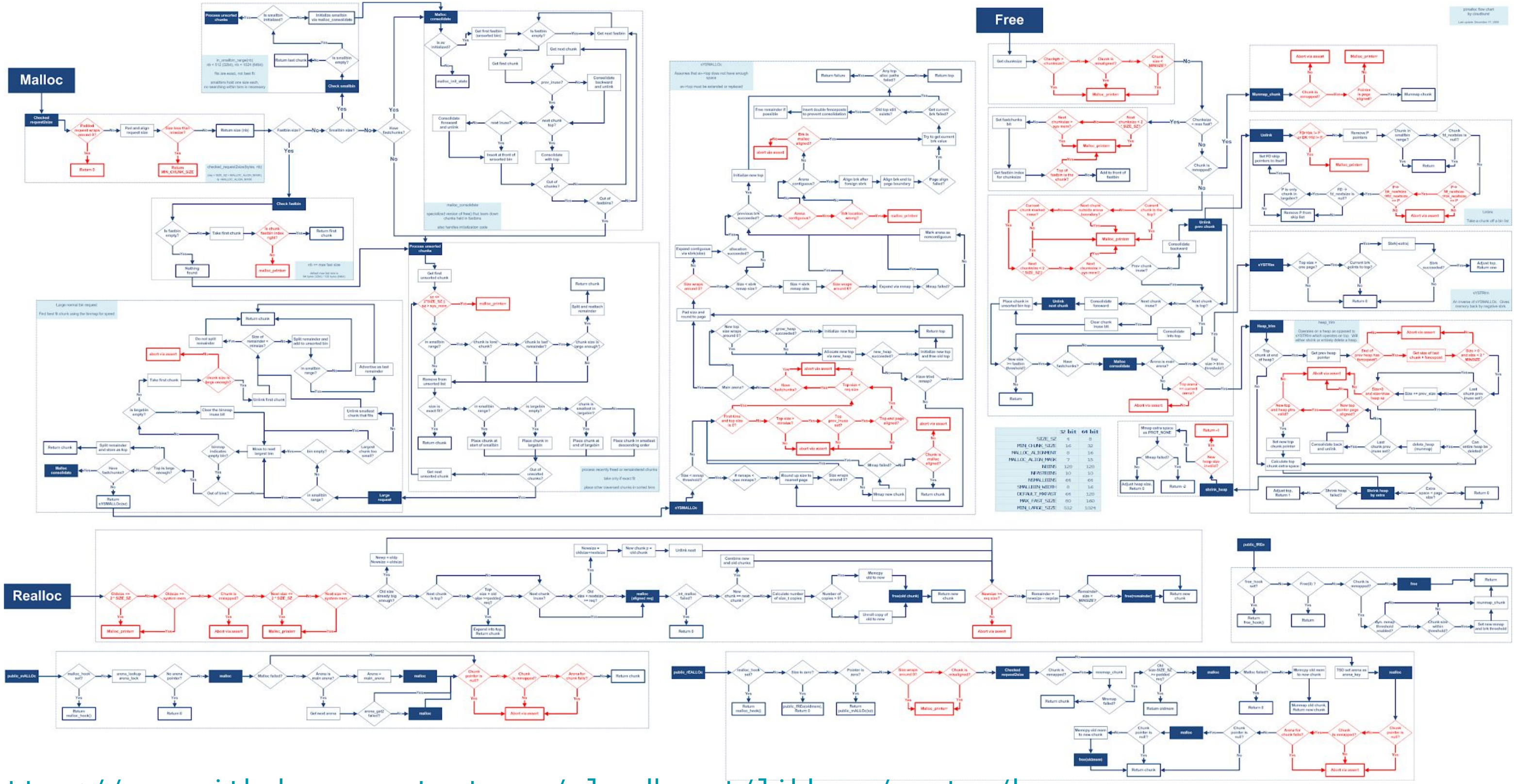
```
    chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of previous chunk, if unallocated (P clear)  |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of chunk, in bytes                     |A|M|P|
      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|
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Best documentation is ~~source code~~. binary code

Ubuntu Backports - glibc 2.27

```
@@ -2942,6 +2951,7 @@ tcache_get (size_t tc_idx)
   assert (tcache->entries[tc_idx] > 0);
   tcache->entries[tc_idx] = e->next;
   --(tcache->counts[tc_idx]);
+  e->key = NULL;
   return (void *) e;
 }
```

Algorithm



https://raw.githubusercontent.com/cloudburst/libheap/master/heap.png

Useful Links / Reading Material

- https://github.com/shellphish/how2heap

- https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/

- https://heap-exploitation.dhavalkapil.com

- https://www.usenix.org/conference/usenixsecurity18/presentation/heelan (Automatic Heap Manipulation)