Intro to Heap Exploitation for CTF

20/07/2023

The Heap

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

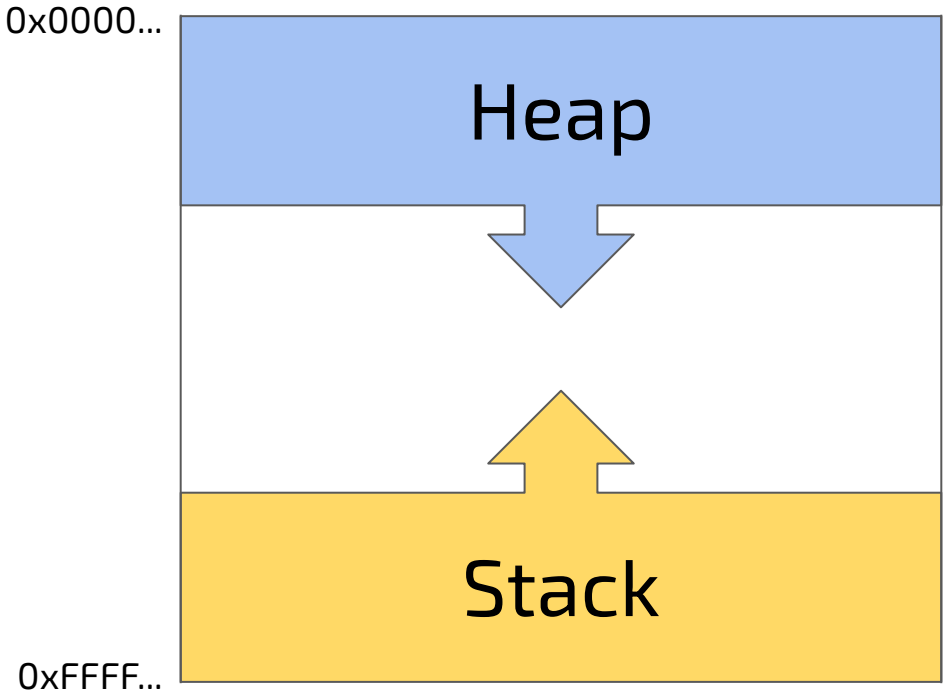
- **ptmalloc** (glibc)
- dlmalloc (was in glibc)
- PartitionAlloc (chromium)
- jemalloc (FreeBSD, Firefox, Android)
- Scudo (Android)

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)
- **coalescing** (how to merge chunks)
- **segregations** free list
- NO segregations storage
- **deterministic**

Reference Memory Layout

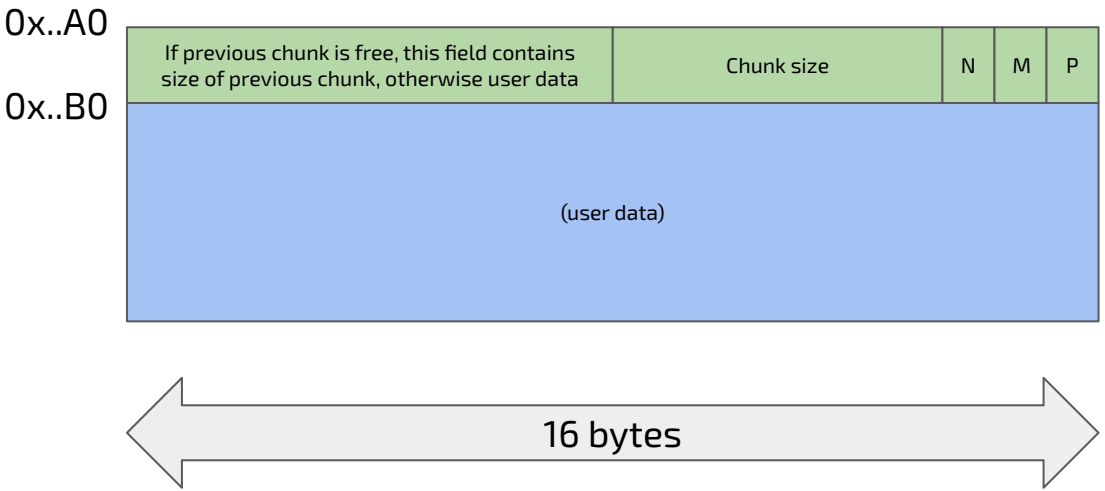


Memory Allocation: malloc

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

- Input: size(bytes)
- Returns a “`void *`” pointer which points the allocated memory(buffer)
- This buffer is a part of struct called chunk

Chunk

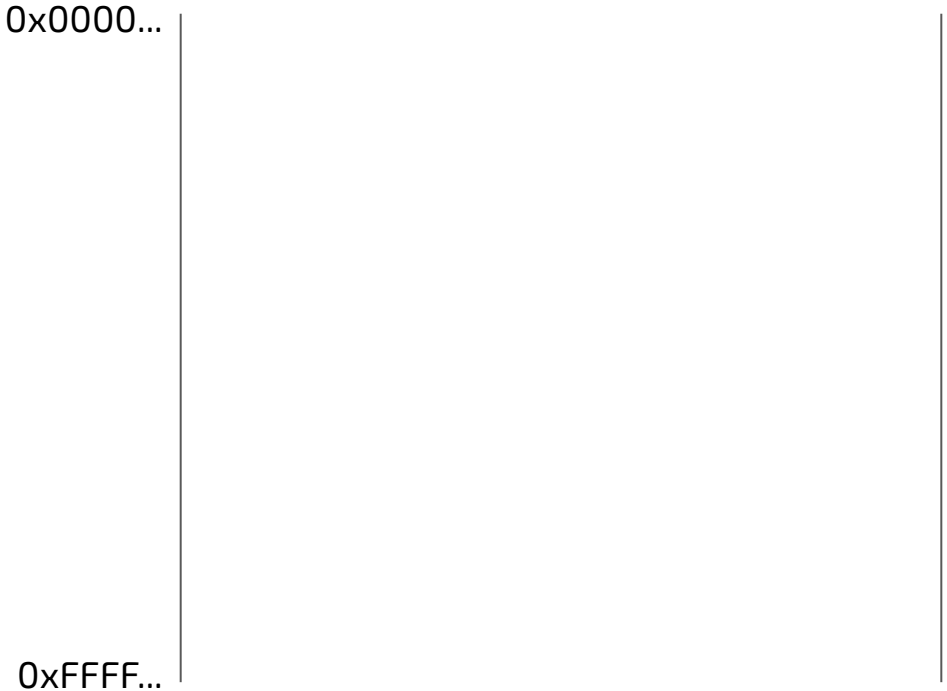


- PREV_INUSE (P) – This bit is set when previous chunk is allocated.
- IS_MMAPPED (M) – This bit is set when chunk is mmap'd.
- NON_MAIN_ARENA (N) – This bit is set when this chunk belongs to a thread arena.

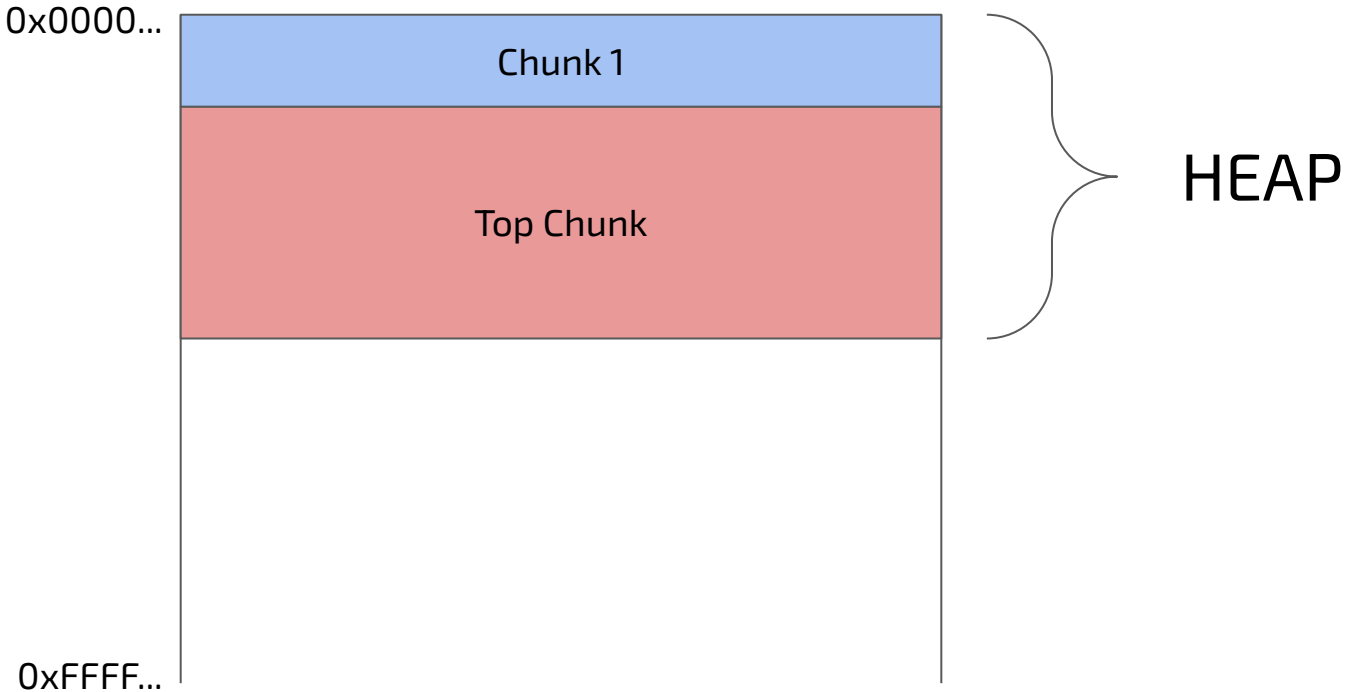
Top Chunk / Wilderness

- Special chunk that occupies all the available memory space in the heap
- When a malloc is called it might shrink
- 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

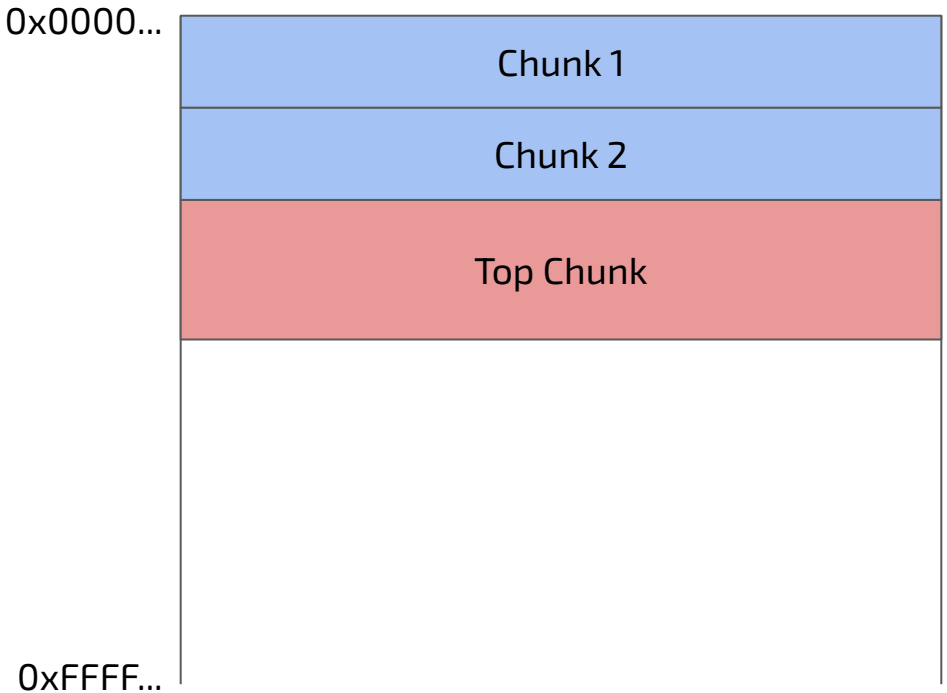
Memory Allocation



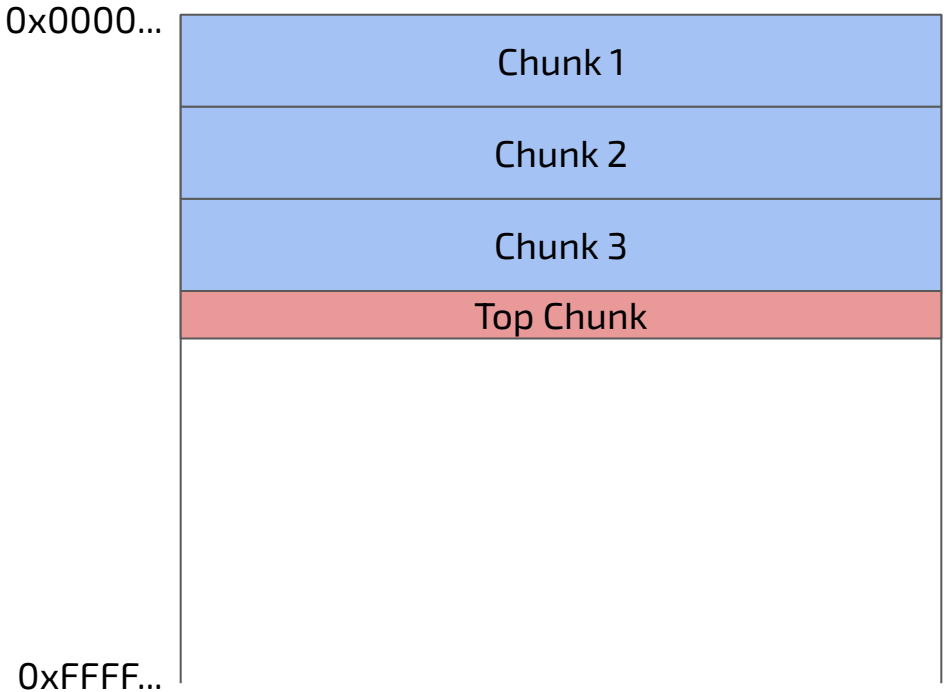
Memory Allocation



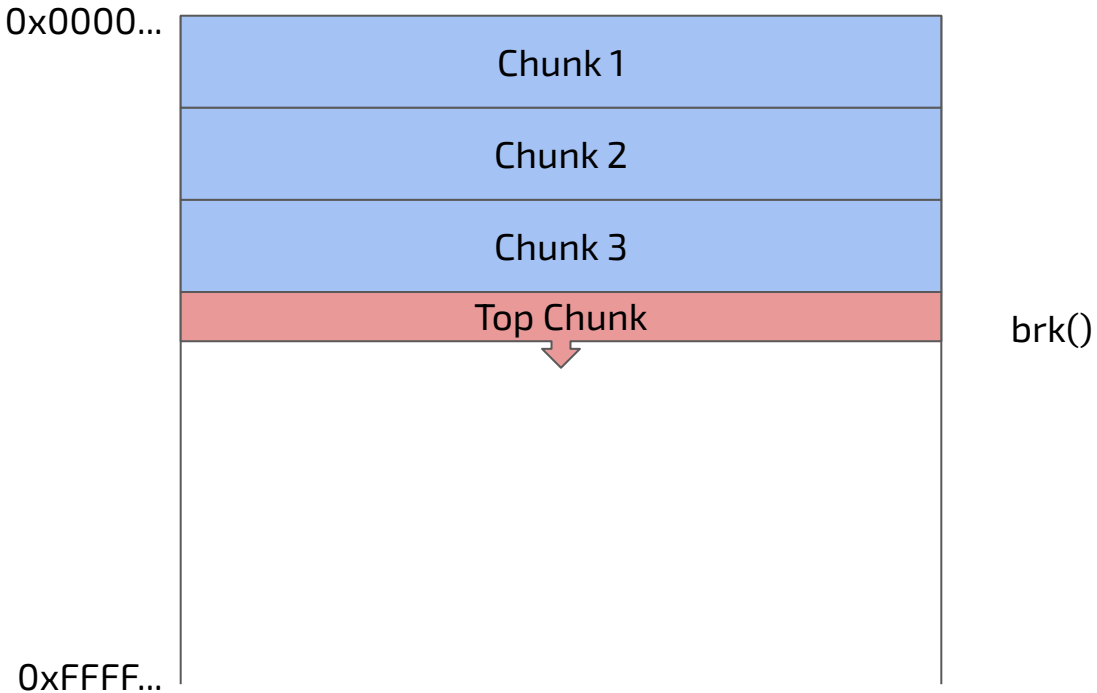
Memory Allocation



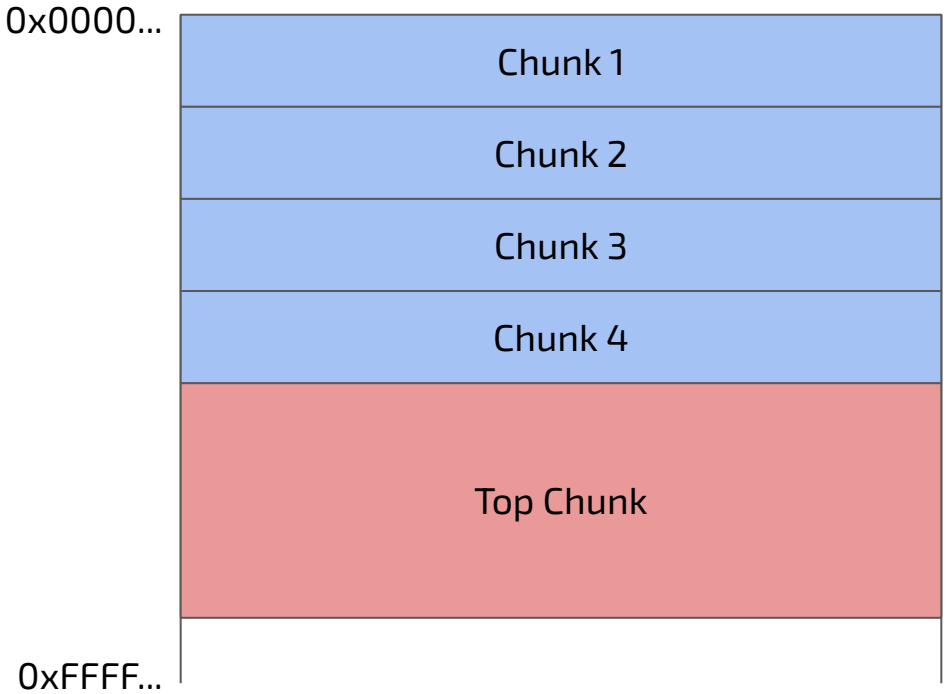
Memory Allocation



Memory Allocation



Memory Allocation



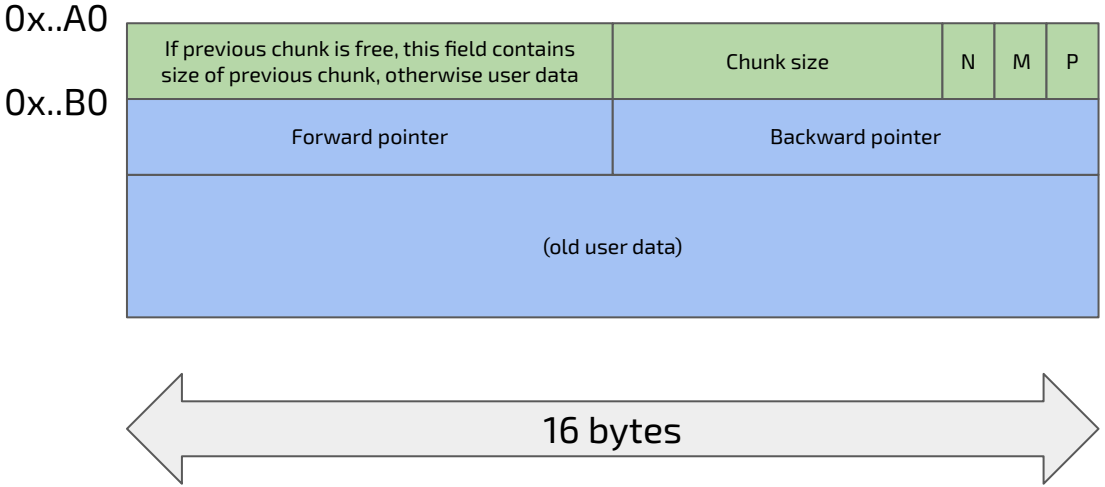
Let’s see it in memory!

Memory Deallocation: free

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

- Input: 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 **free 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

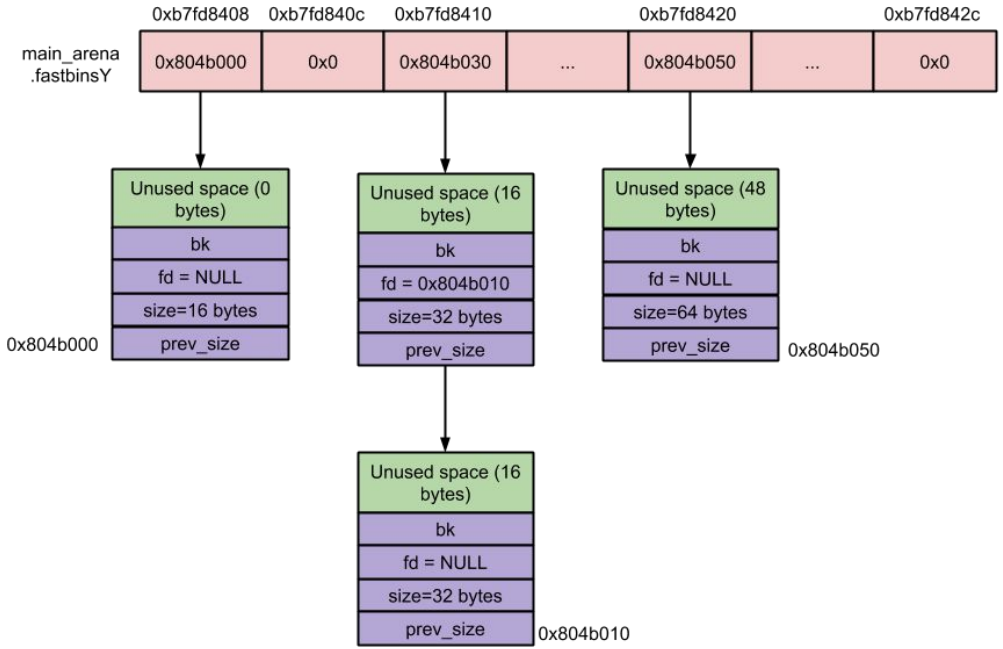
Bins

- t-cache (< 0x500)
- Fast bin (0x20 to 0x90 bytes)
- Unsorted bin
- Small bin (< 0x400 bytes)
- Large bin (>= 0x400 bytes)
- top-chunk

Fast Bins

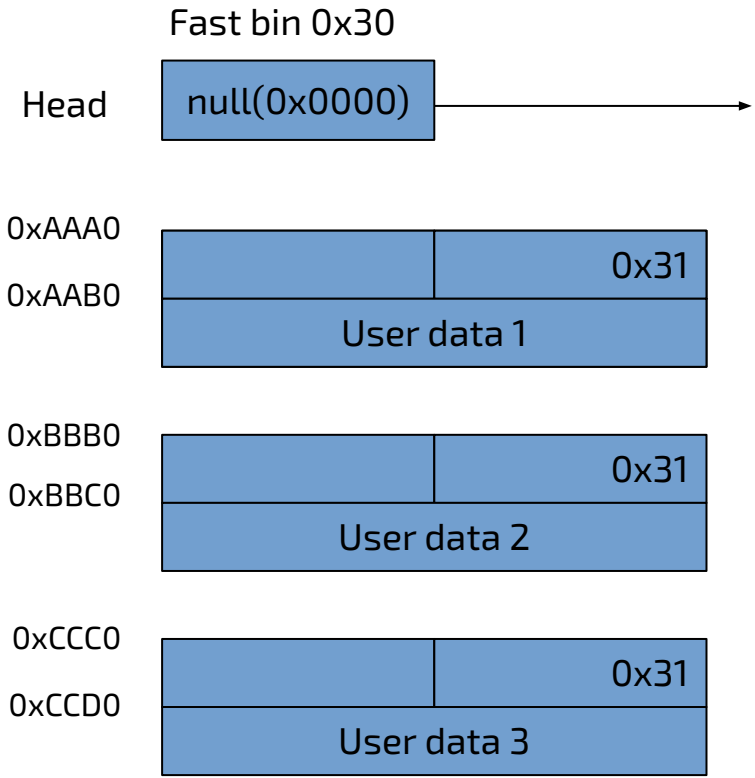
- Optimized bins for tiny freed chunks
- Managed as **LIFO** linked lists (backward pointer not used)
- Better performance, less checks and operations
- Freed chunks are **never consolidate** with any other freed chunk (not really)
- Freed chunks in fast bins act as non freed chunks (**P flag** of next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

Fast Bins

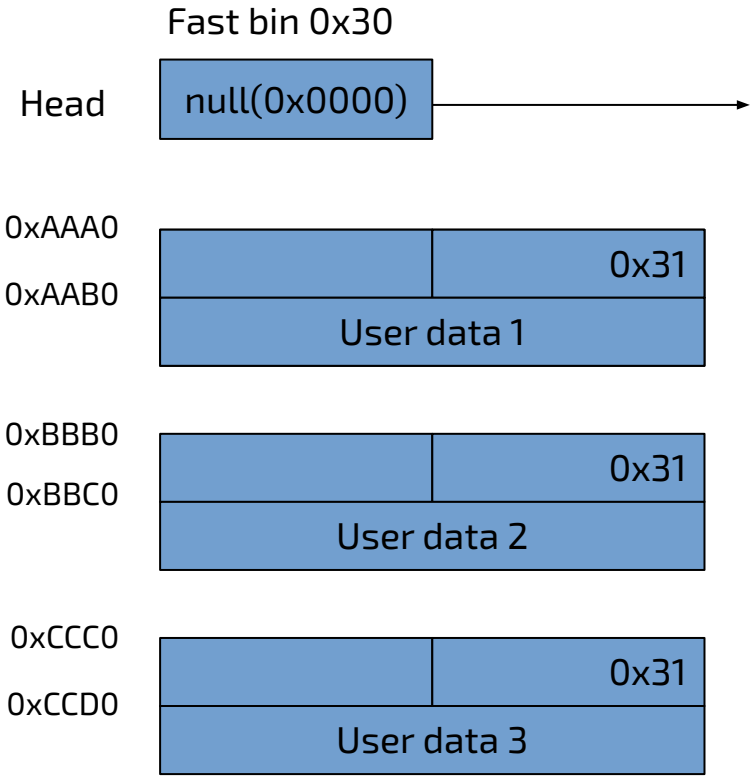


Fast Bin Snapshot

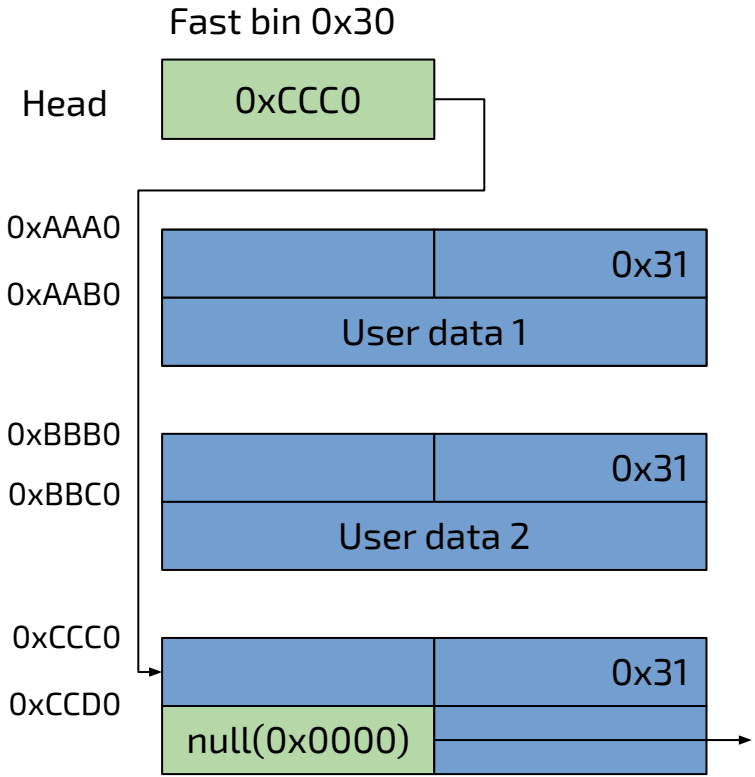
Fast Bins


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

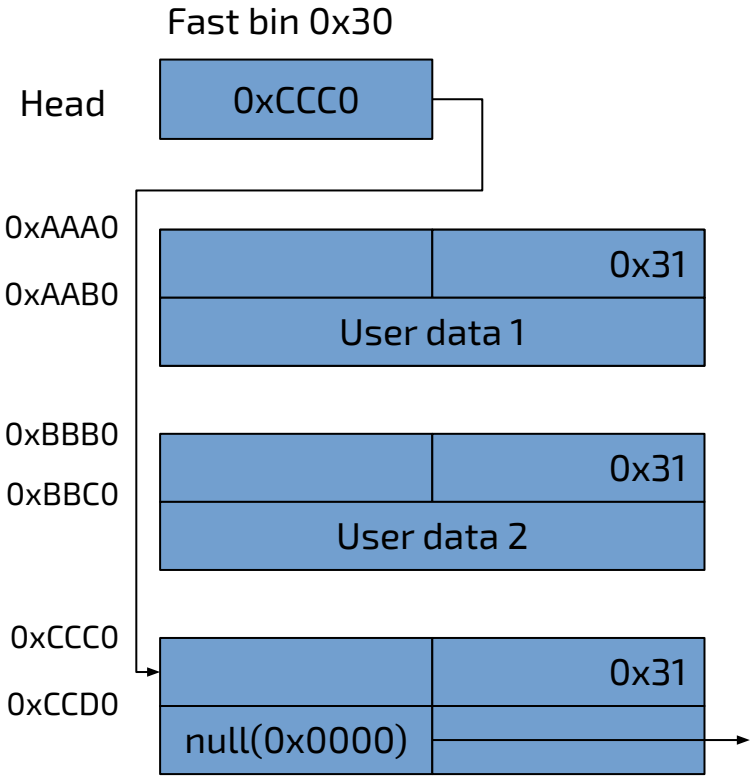
Fast Bins: free(0xCCD0) - Before



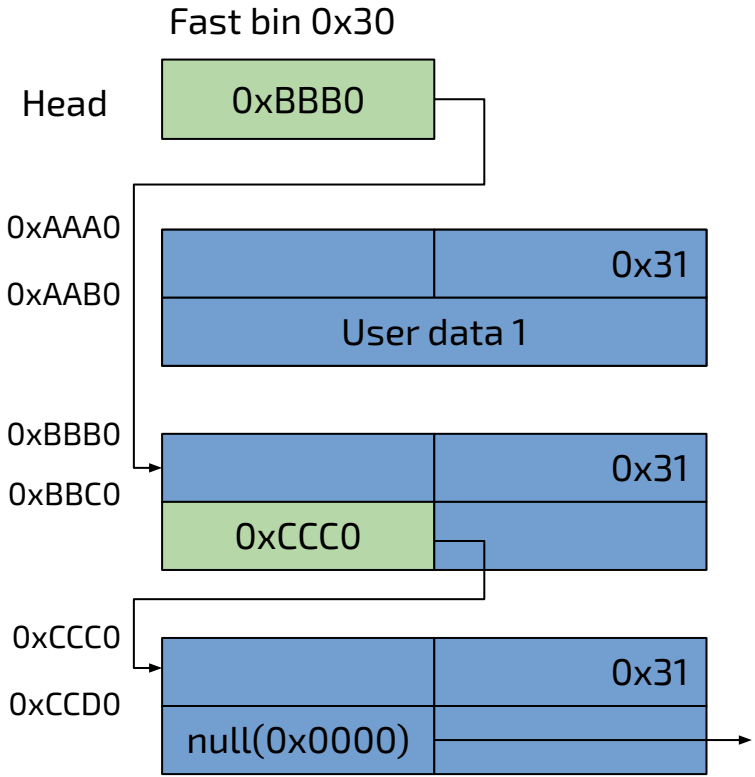
Fast Bins: free(0xCCD0) - After



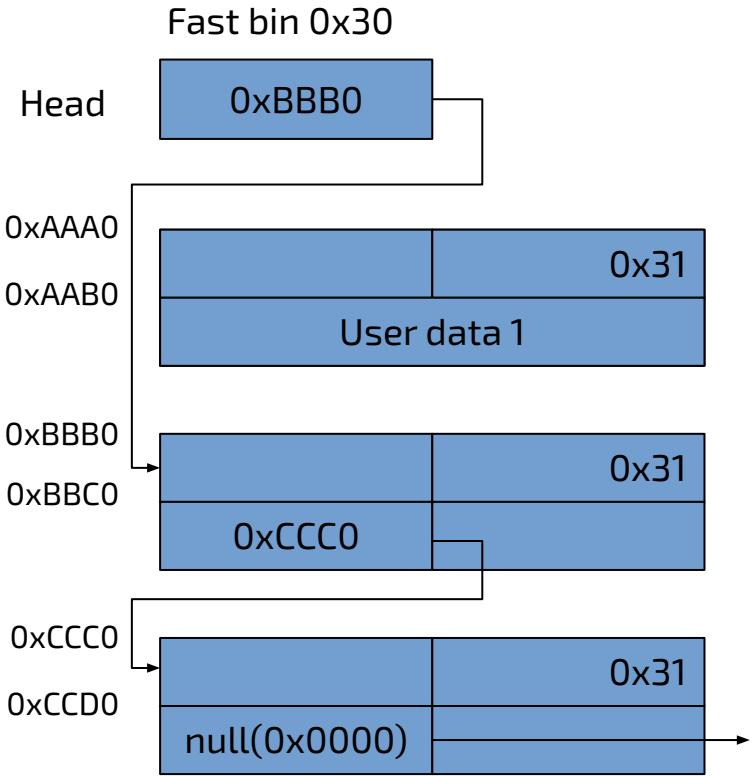
Fast Bins: free(0xBBC0) - Before



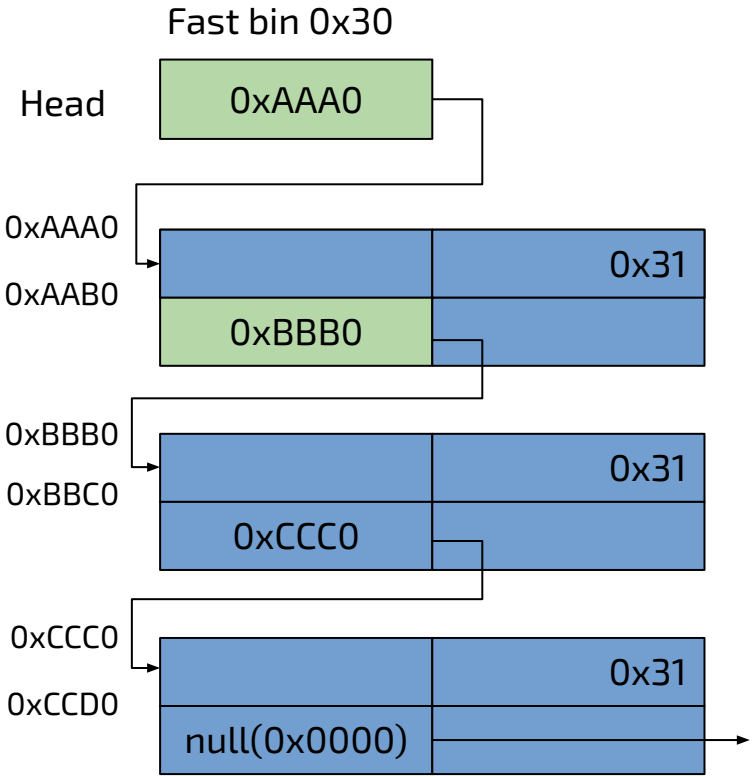
Fast Bins: free(0xBBC0) - After



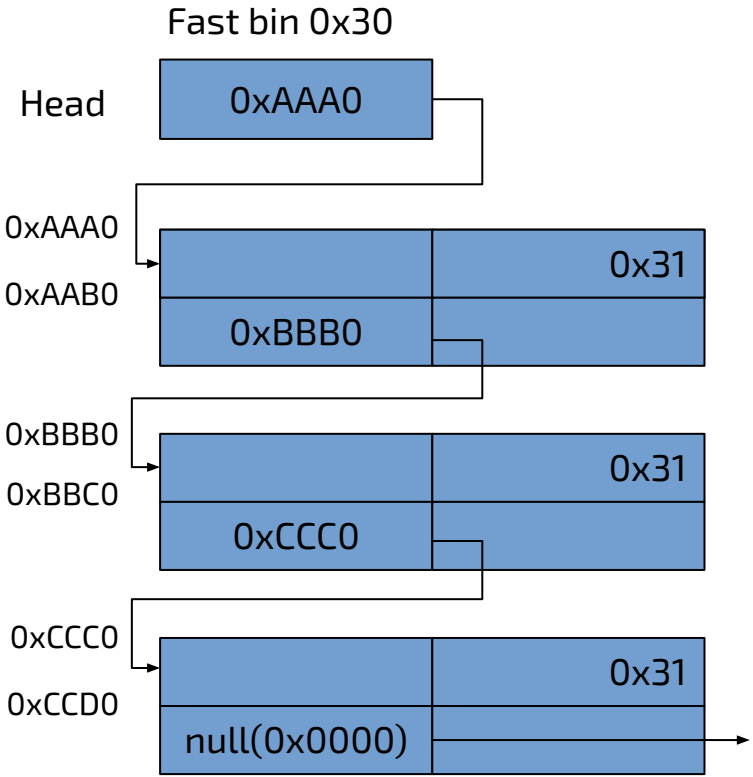
Fast Bins: free(0xAAB0) - Before



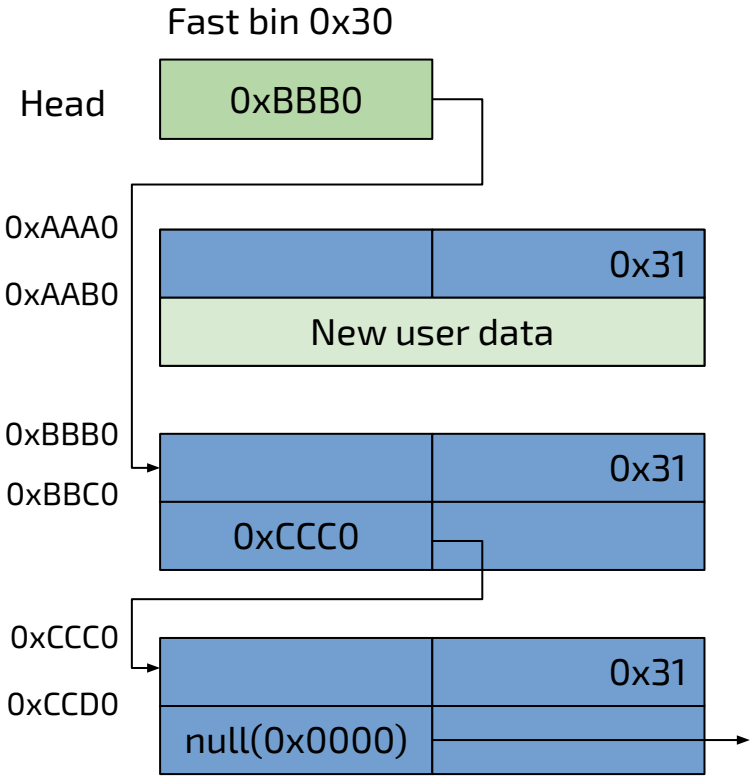
Fast Bins: free(0xAAB0) - After



Fast Bins: malloc(0x20) - Before



Fast Bins: malloc(0x20) - After

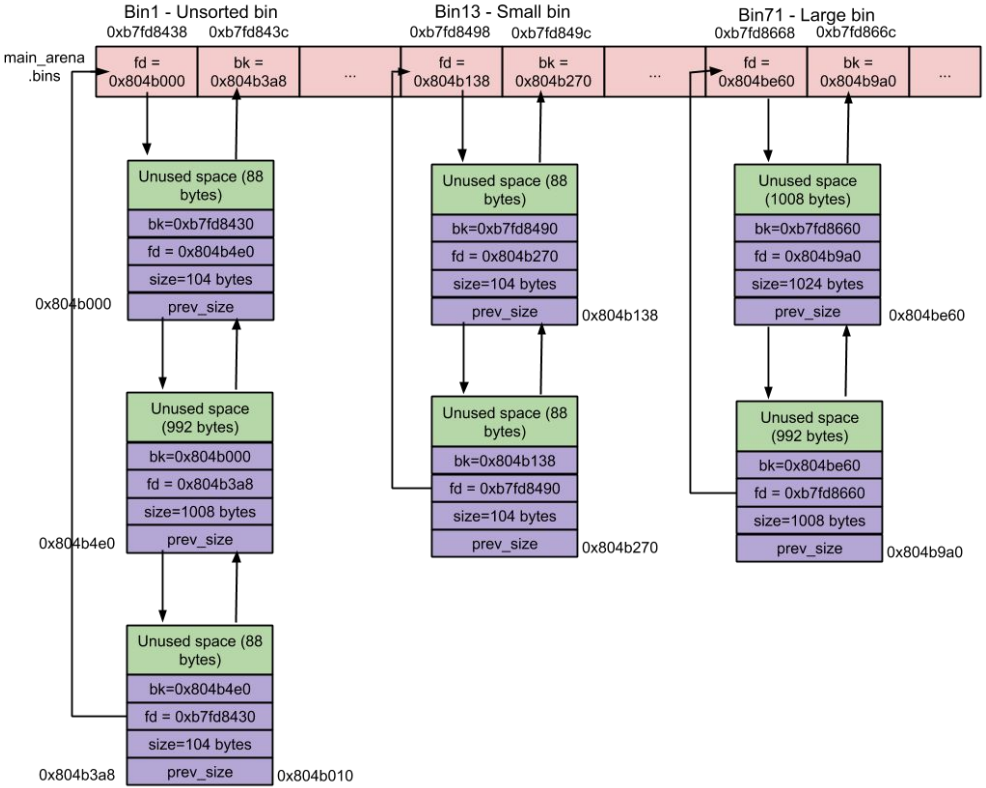


Let’s see it in memory!

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!

Heap Vulnerabilities

- **Double Free**
- **Use after Free**
- **Heap overflow**
- **Arbitrary Free**

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)

Interesting Stuff for **Code Execution**

.got: library functions

`puts("/bin/sh") one_gadget`

Free/Malloc Hook: function pointers executed instead of

`free("/bin/sh") malloc(0x7f6723a7a) one_gadget`

.bss libc: vtable, function ptr …

`one_gadget crafted_vtable, FSROP` (https://blog.kylebot.net/2022/10/22/angry-FSROP/), non-libc ptr (https://github.com/untangle-tool/untangle)

Stack: return address

`one_gadget, rop, environ in libc or ld.so`

Heap: function pointers

Interesting Stuff for **Code Execution**

~~**.got**: library functions~~

~~`puts("/bin/sh") one_gadget`~~

~~**Free/Malloc Hook**: function pointers executed instead of~~

~~`free("/bin/sh") malloc(0x7f6723a7a) one_gadget`~~

.bss libc: vtable, function ptr …

`one_gadget crafted_vtable, FSROP` (https://blog.kylebot.net/2022/10/22/angry-FSROP/), non-libc ptr (https://github.com/untangle-tool/untangle)

Stack: return address

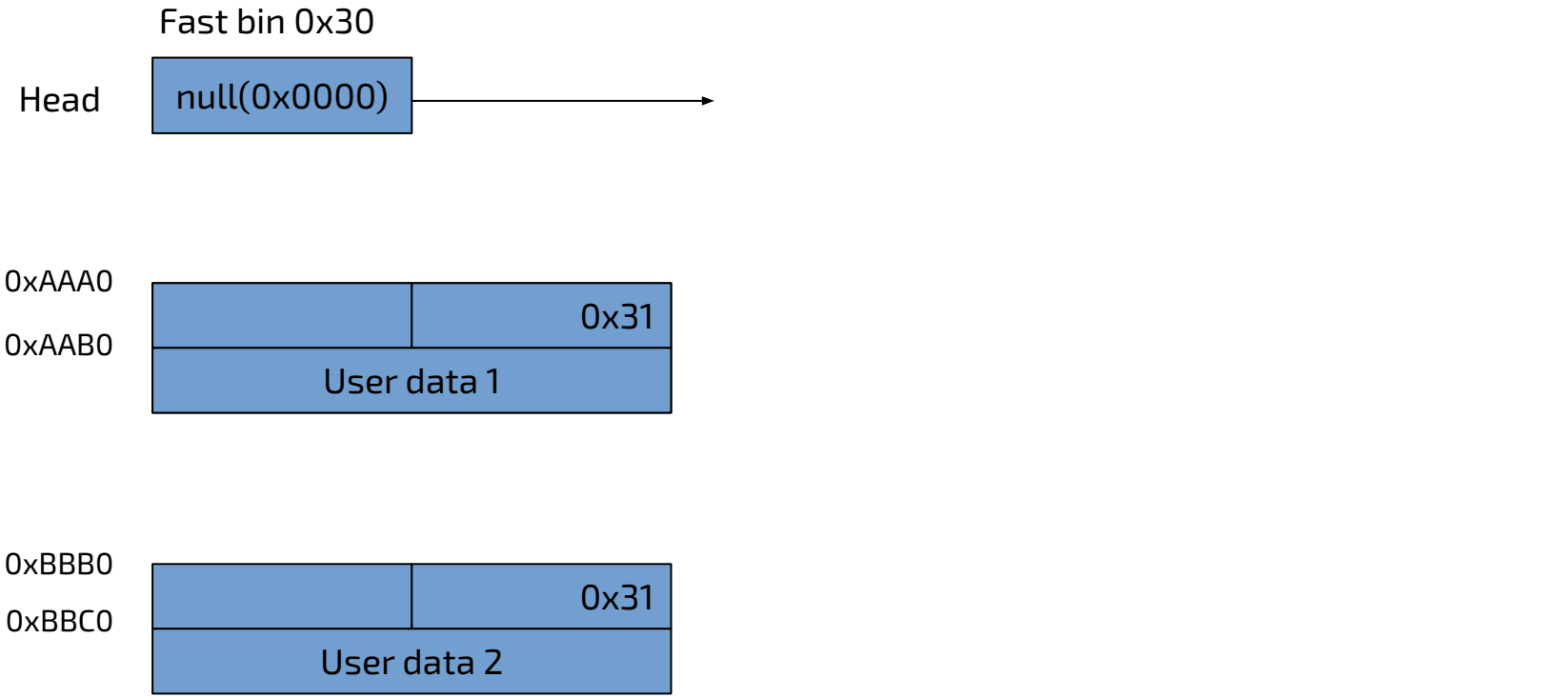
`one_gadget, rop, environ in libc or ld.so`

Heap: function pointers

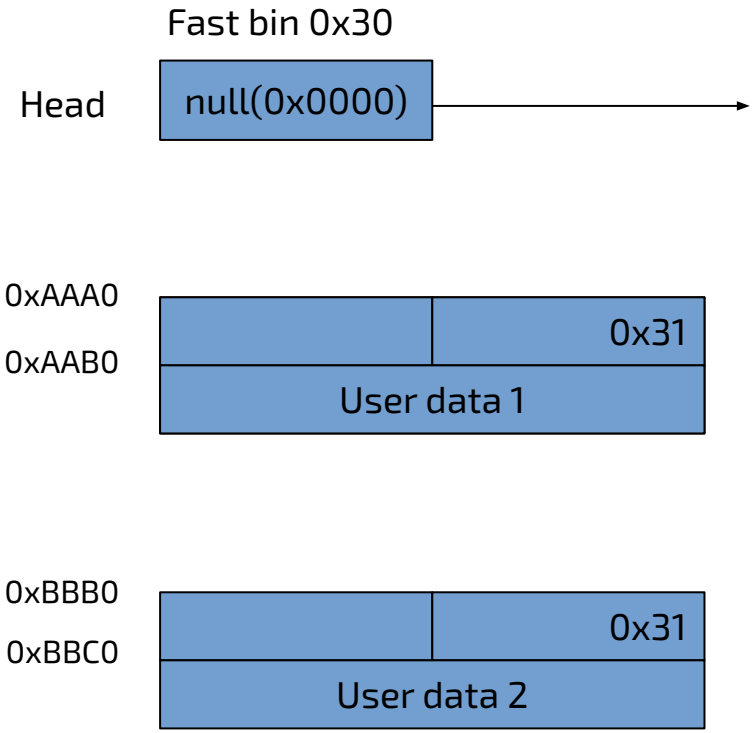
Fast Bin Attack

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

Fast Bin Attack

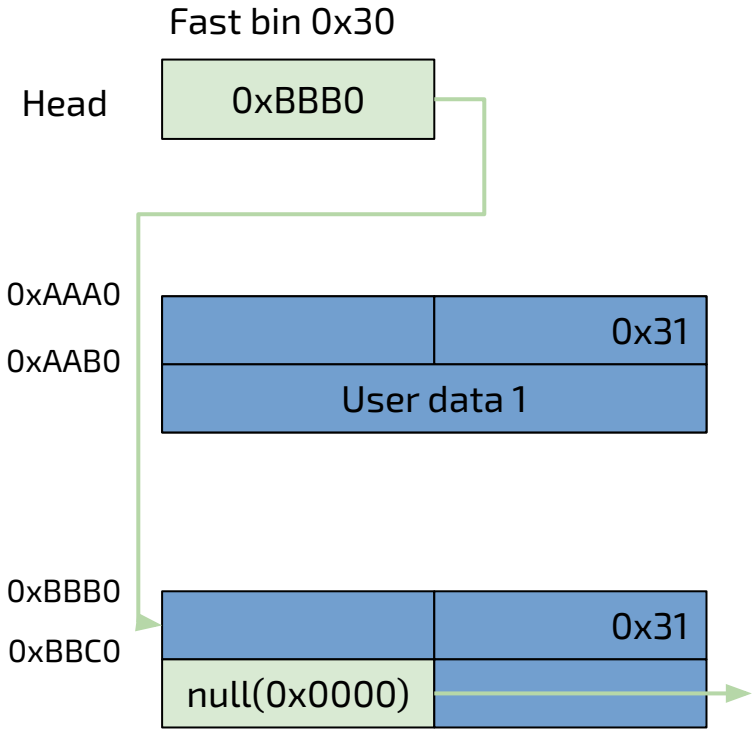


Fast Bin Attack



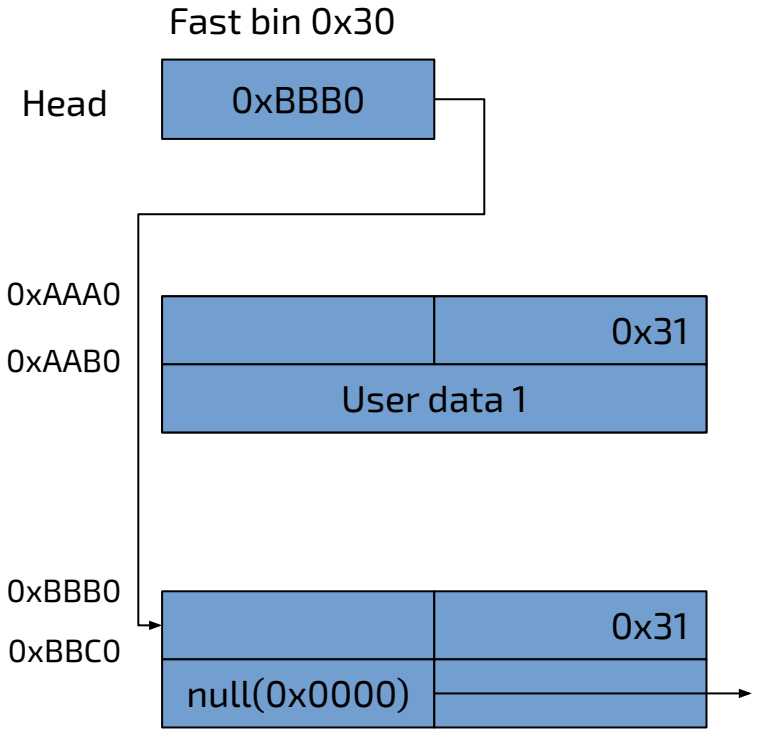

```
free(0xBBC0)
```

Fast Bin Attack



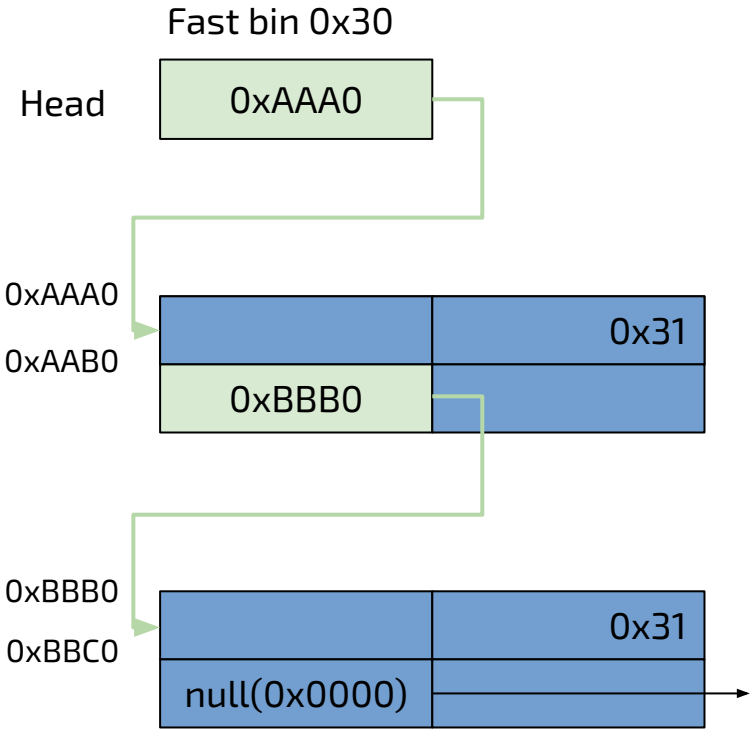

```
free(0xBBC0)
```

Fast Bin Attack



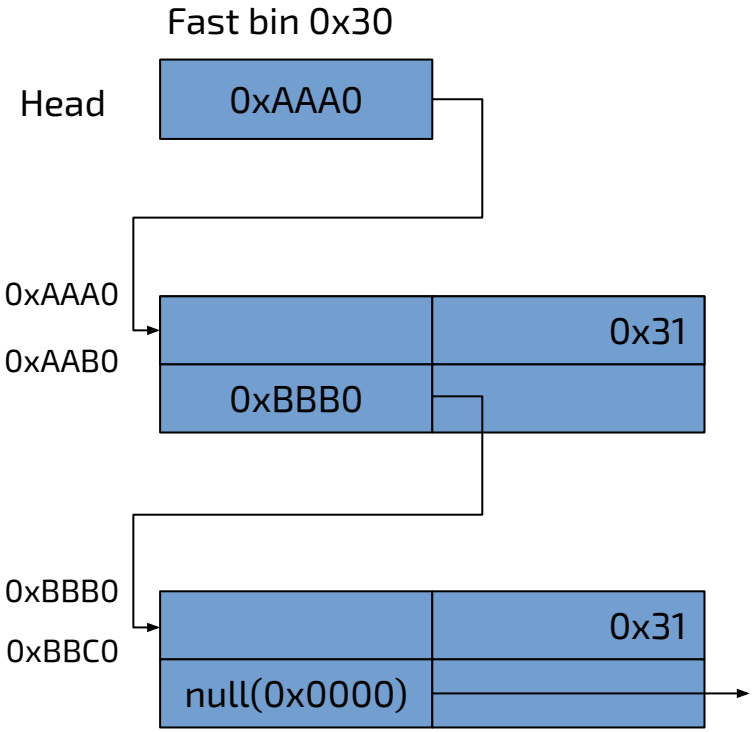

```
free(0xAAB0)
```

Fast Bin Attack



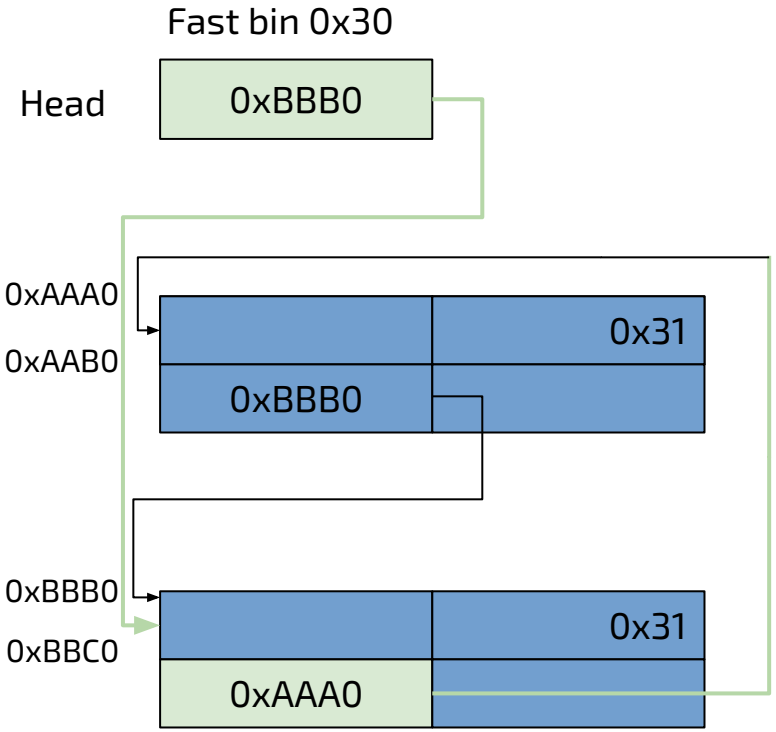

```
free(0xAAB0)
```

Fast Bin Attack



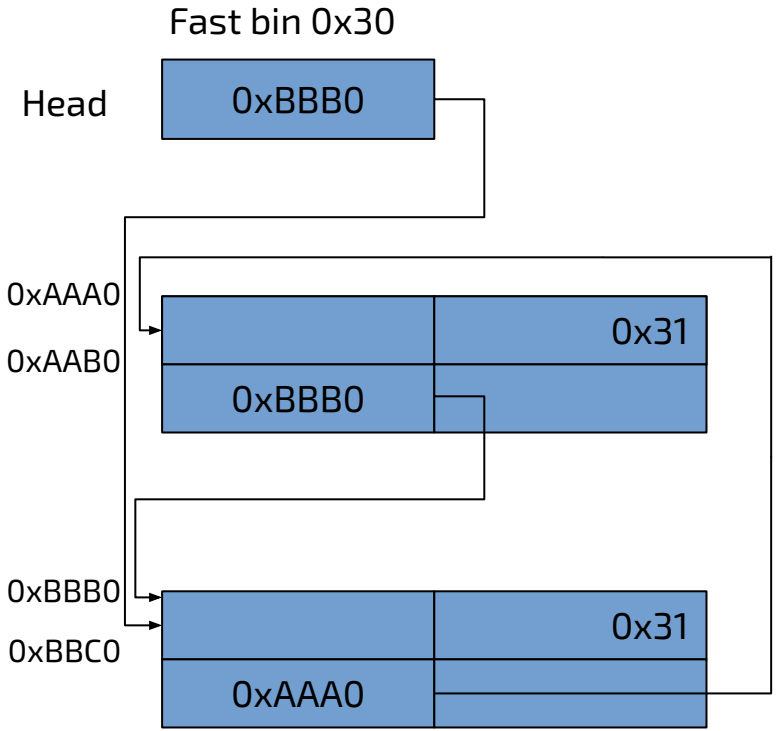

```
free(0xBBC0)
```

Fast Bin Attack



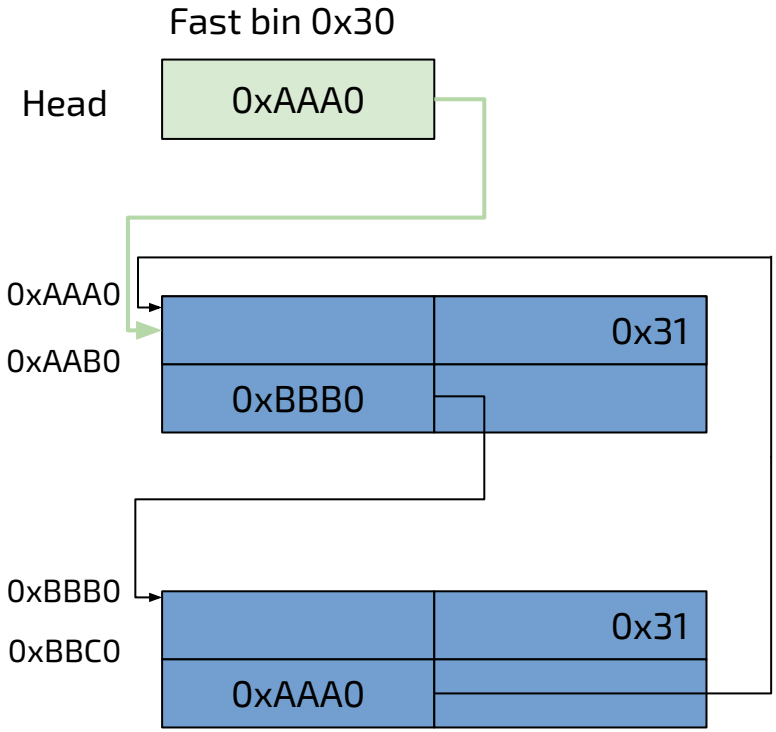

```
free(0xBBC0)
```

Fast Bin Attack




```
malloc(0x20)
```

Fast Bin Attack

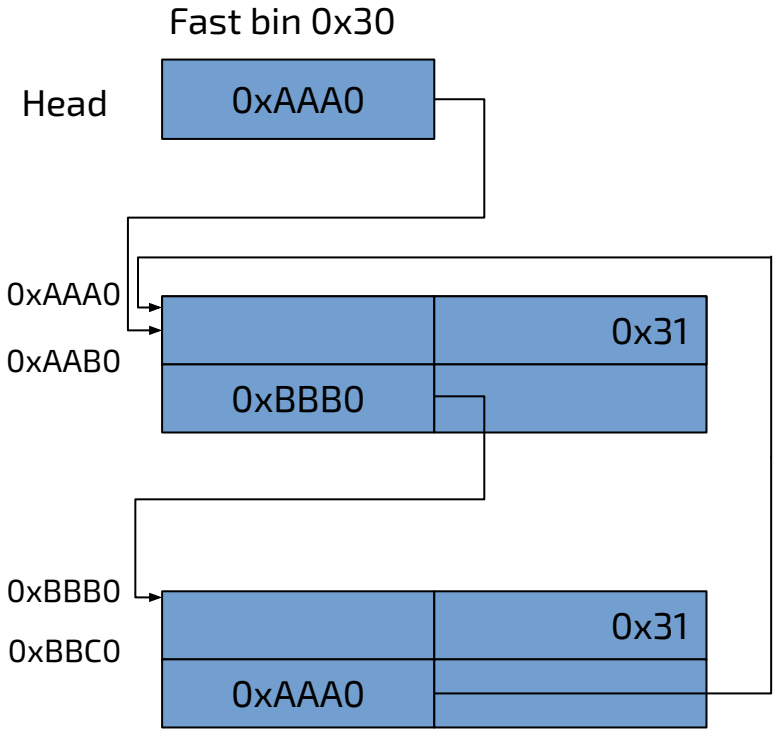


malloc(0x20)

0xBBC0 is returned by the malloc.

- Buffer 1: 0xBBC0

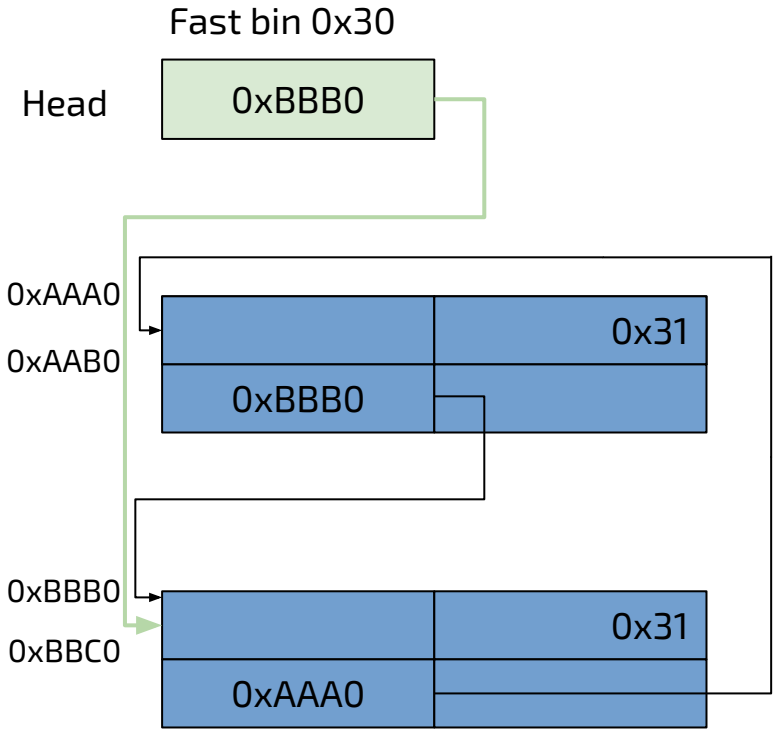
Fast Bin Attack




```
malloc(0x20)
```

- Buffer 1: 0xBBC0

Fast Bin Attack

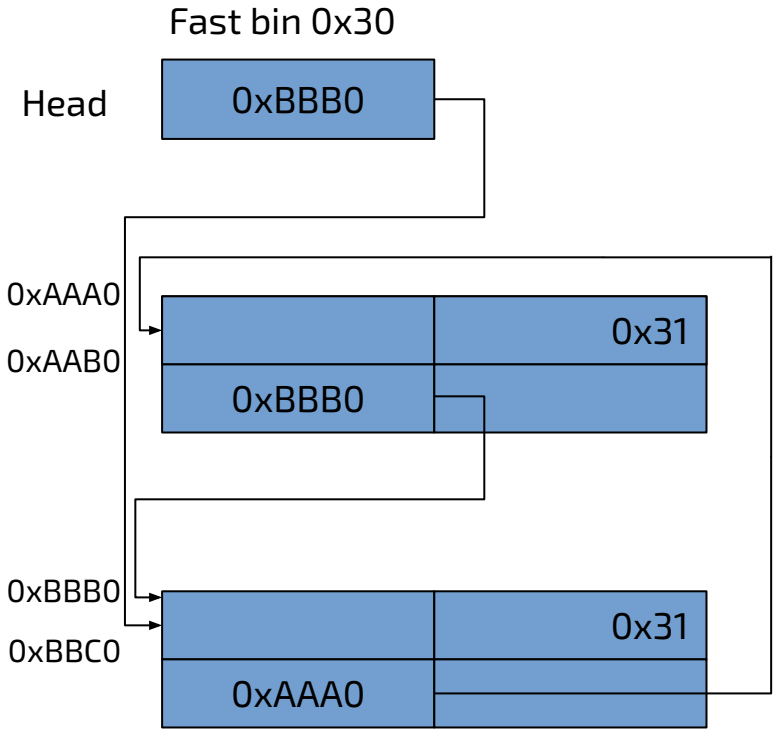


`malloc(0x20)`

0xAAB0 is returned by the malloc.

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

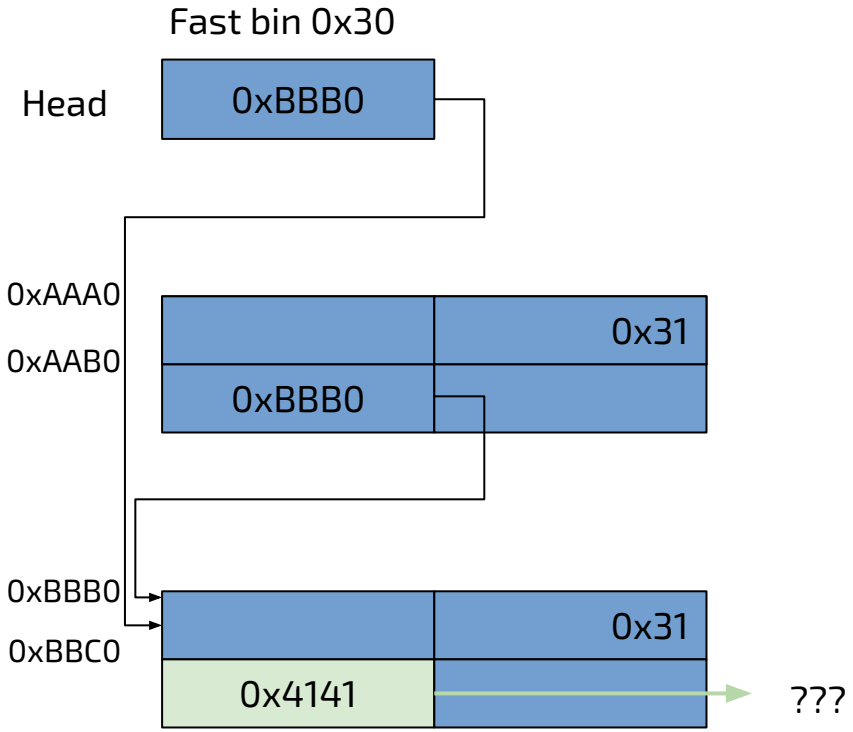
Fast Bin Attack



write Buffer 1

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

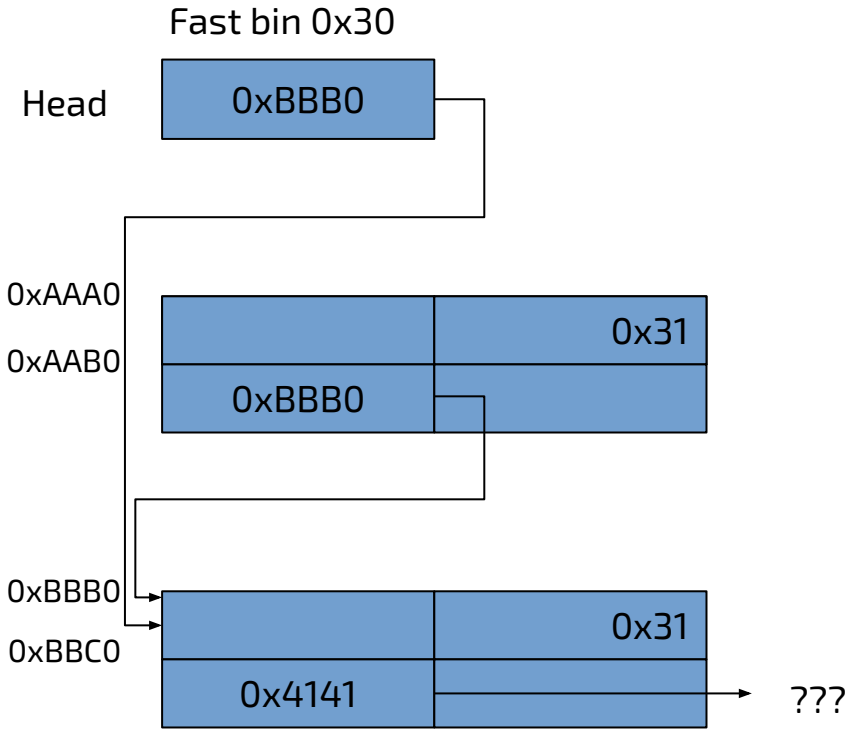
Fast Bin Attack



write Buffer 1

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

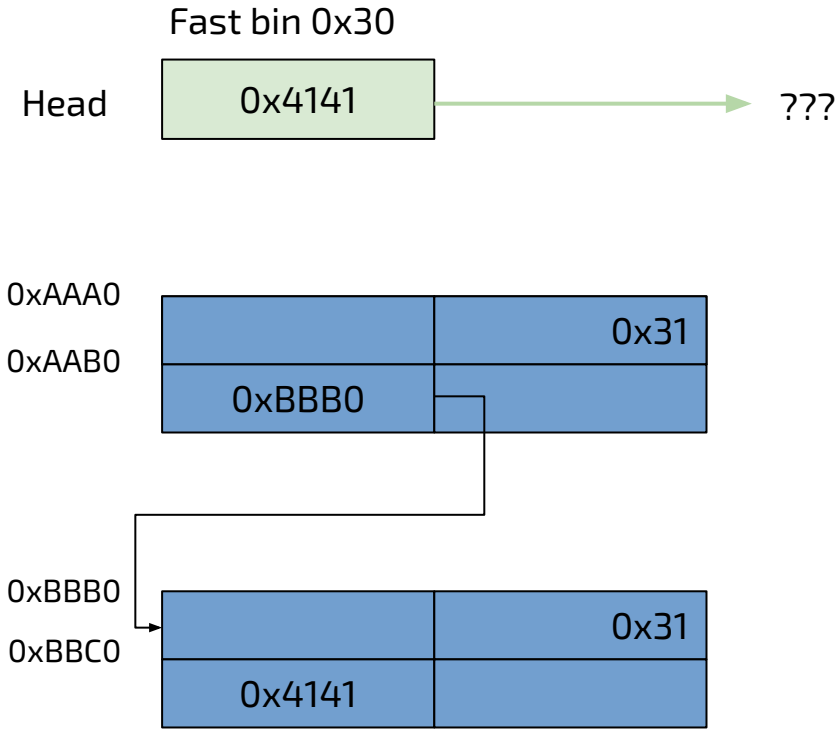
Fast Bin Attack



`malloc(0x20)`

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

Fast Bin Attack: malloc(0x20) - After



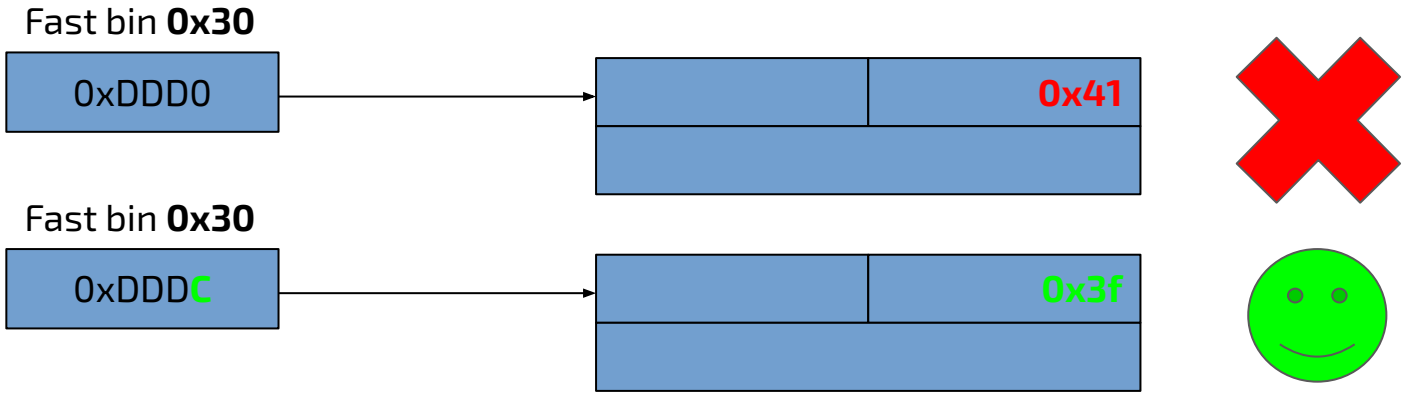
`malloc(0x20)`

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 last 4 bits of the size are not considered
- No requirements on the alignment of the chunk

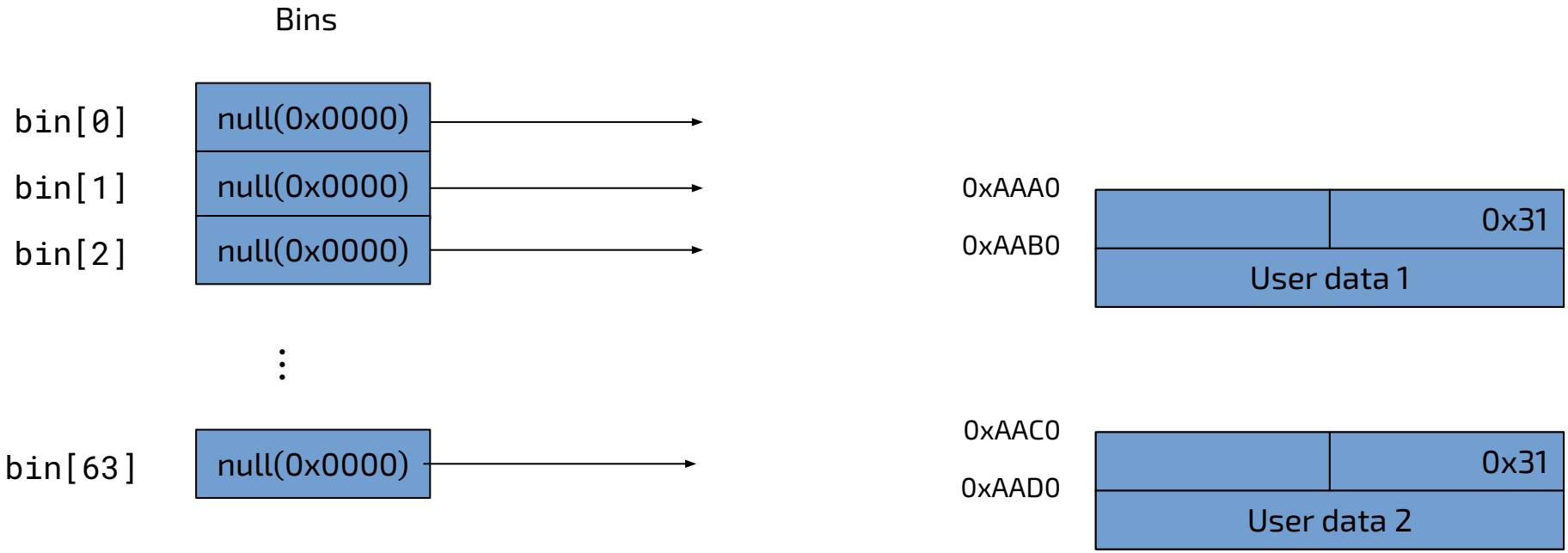


T-Cache

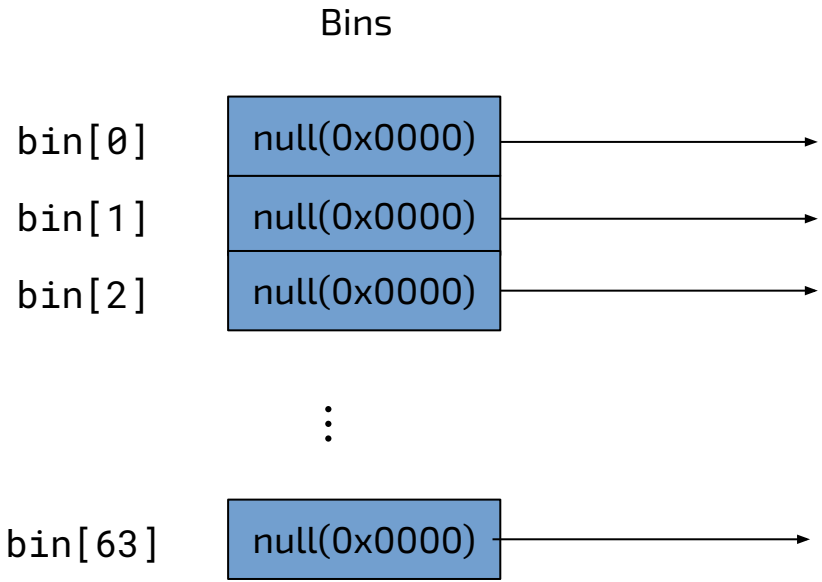
T-Cache

- **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

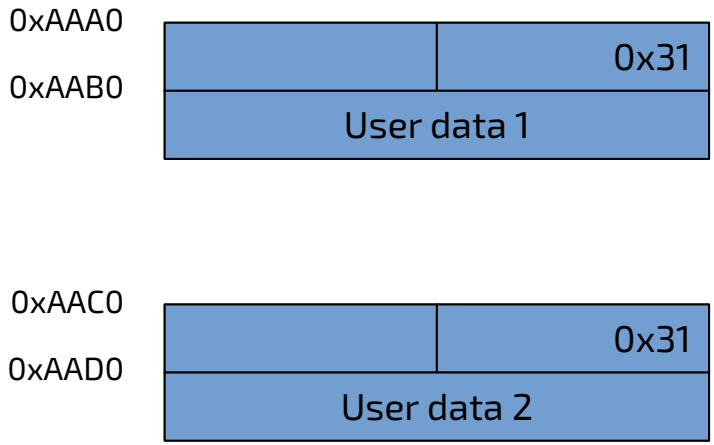
T-Cache


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

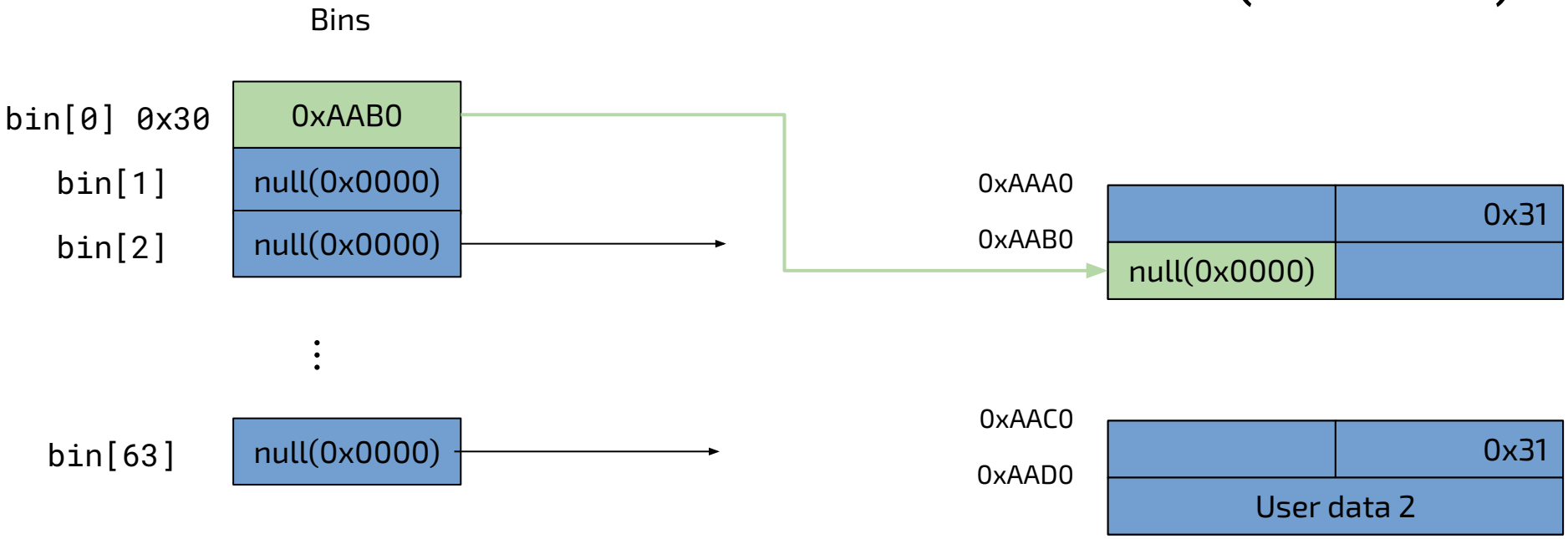


`free(0xAAB0)`



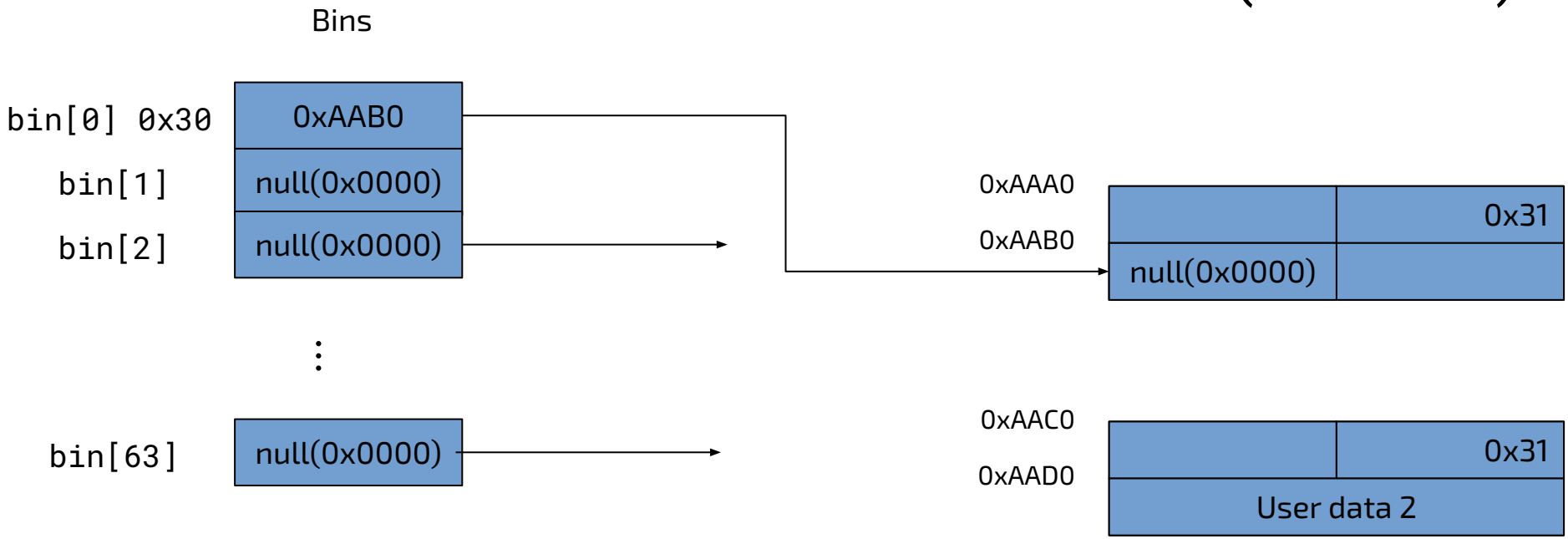
T-Cache

free(0xAAB0)



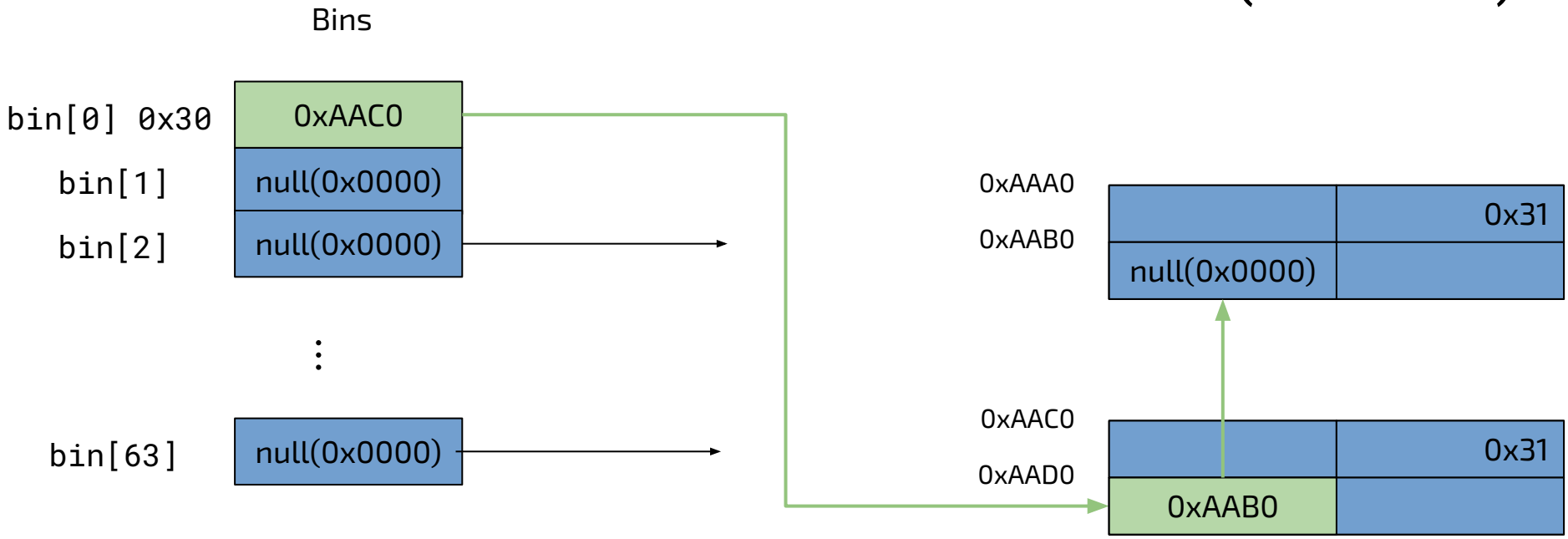
T-Cache

`free(0xAAC0)`



T-Cache

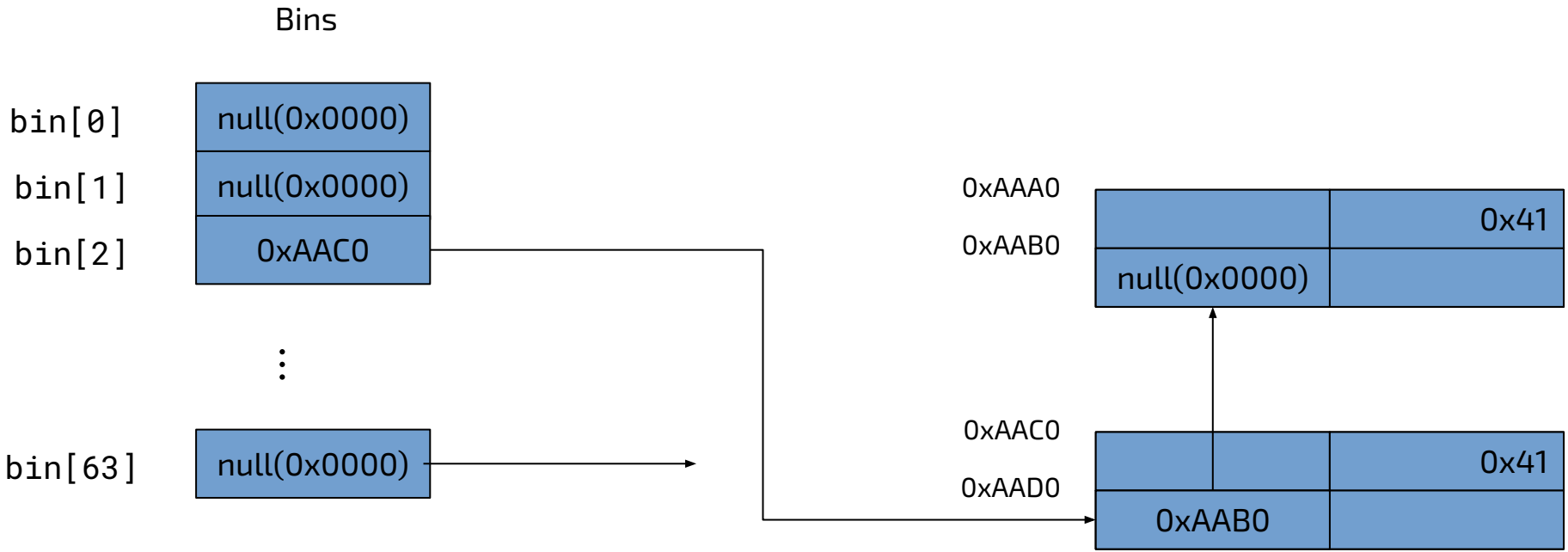
`free(0xAAC0)`



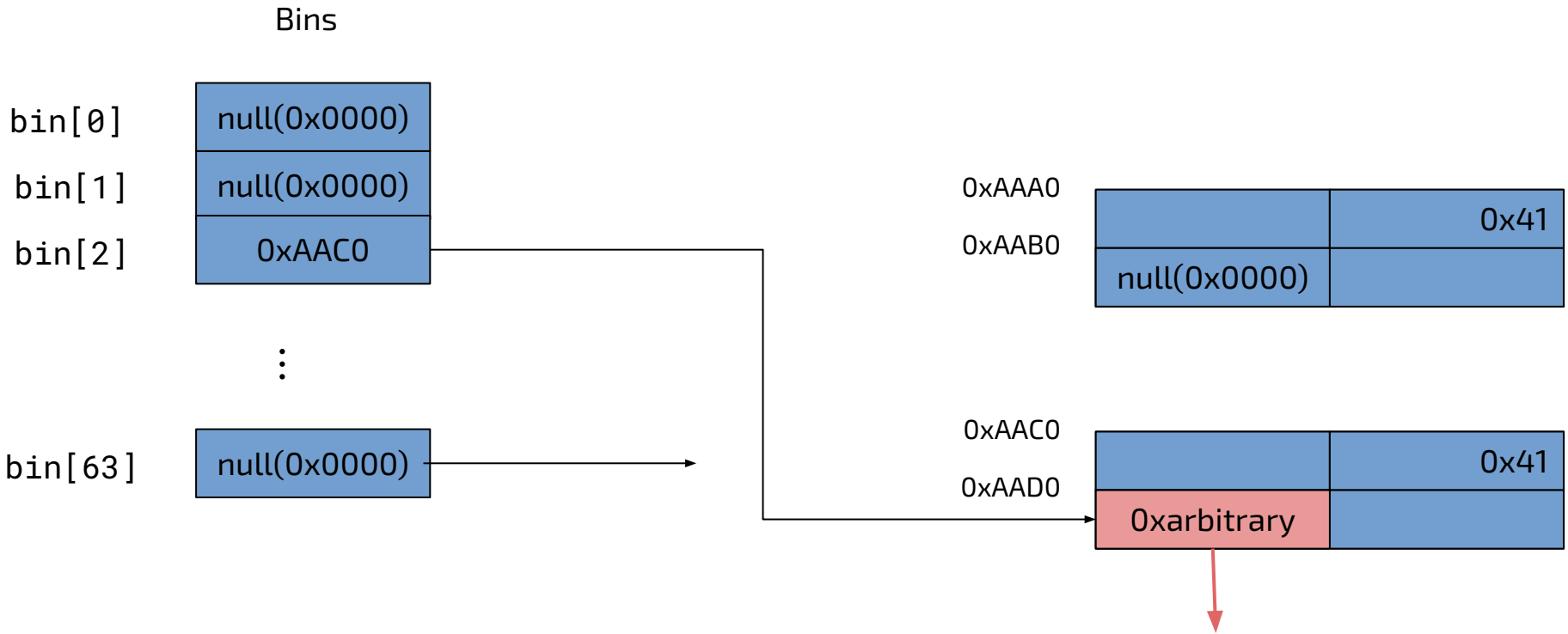
T-Cache Poison

- Allocate an arbitrary chunk

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

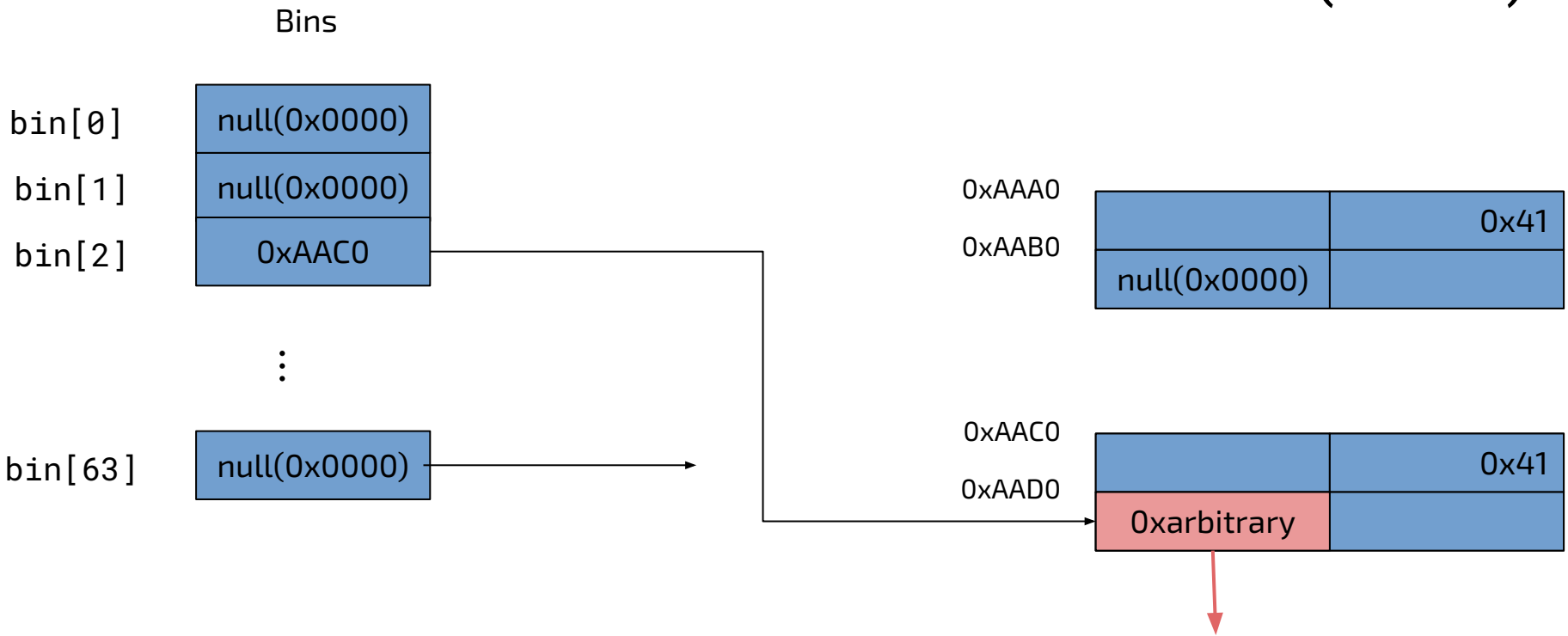


T-Cache

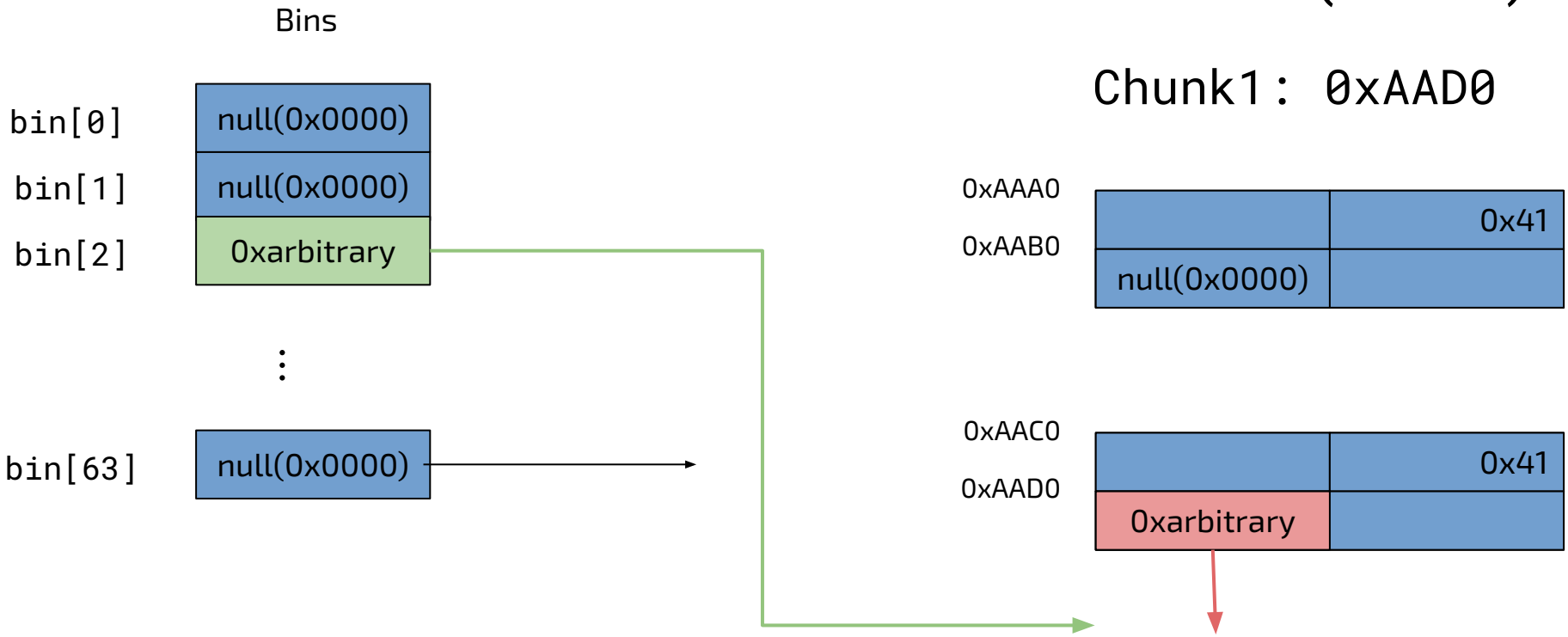

Bins
bin[0]
bin[1]
bin[2]
bin[63]
null(0x0000)
null(0x0000)
0xAAC0
null(0x0000)
0xAAA0
0xAAB0
0x41
null(0x0000)
0xAAC0
0xAAD0
0x41
0xarbitrary

T-Cache

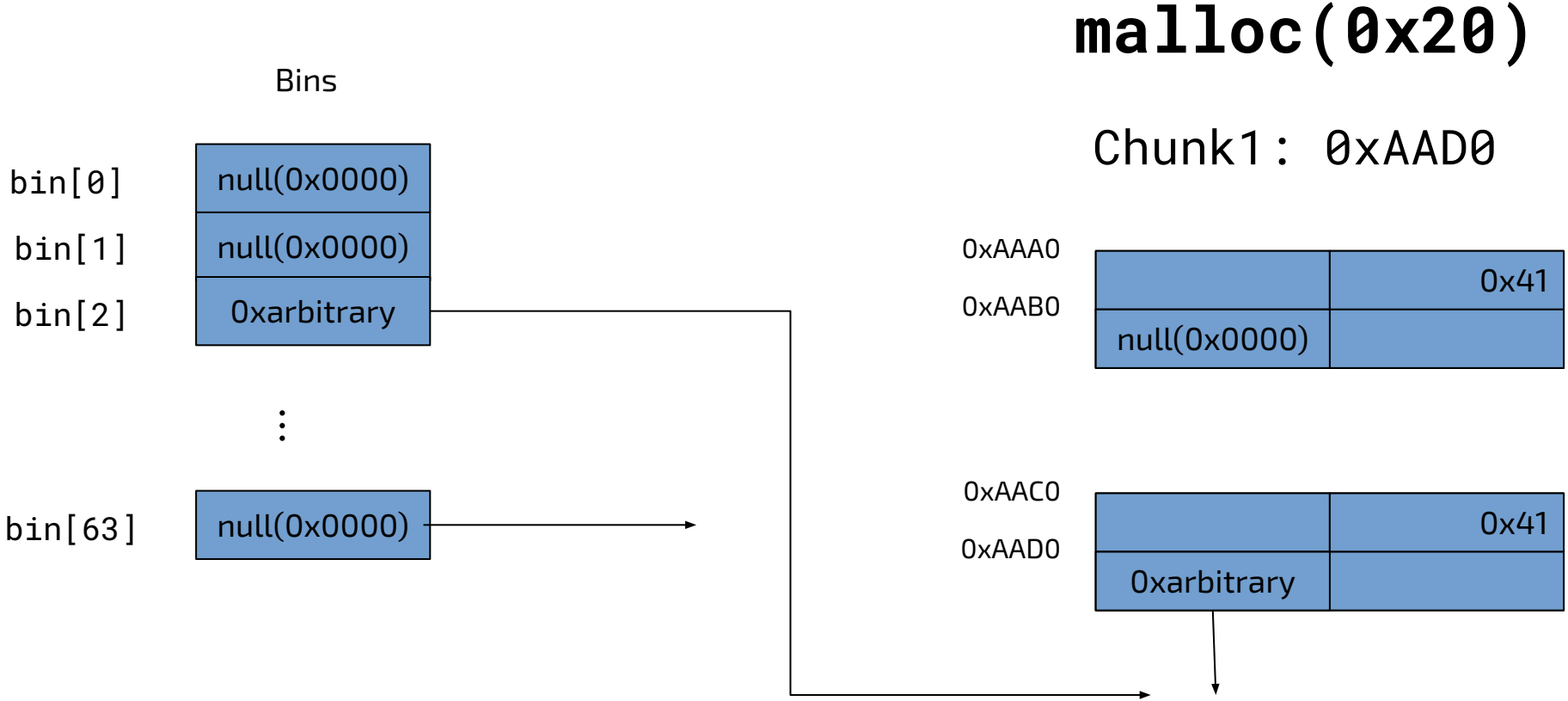
malloc(0x20)



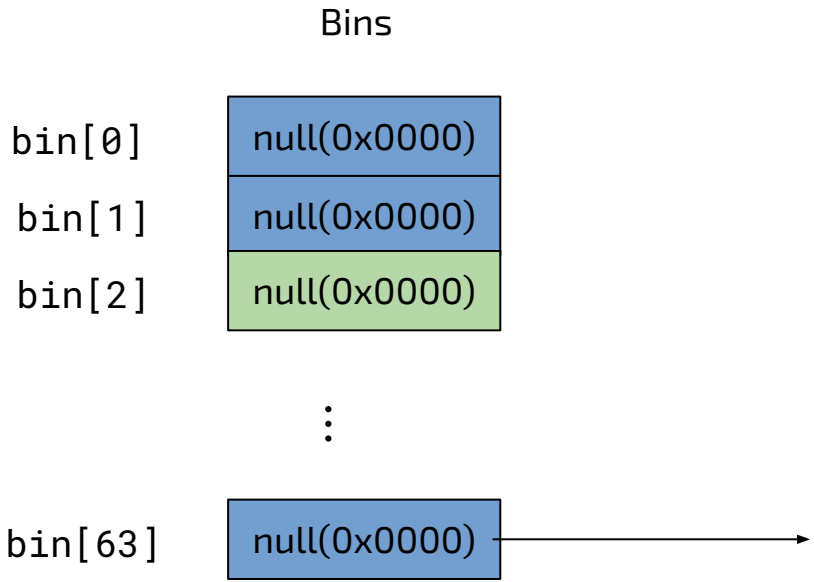
T-Cache


malloc(0x20)
Chunk1: 0xAAD0
Bins
bin[0]
bin[1]
bin[2]
bin[63]
null(0x0000)
null(0x0000)
0xarbitrary
null(0x0000)
0xAAA0
0xAAB0
0xAAC0
0xAAD0
0x41
null(0x0000)
0x41
0xarbitrary

T-Cache


malloc(0x20)
Bins
bin[0]
null(0x0000)
bin[1]
null(0x0000)
bin[2]
0xarbitrary
bin[63]
null(0x0000)
Chunk1: 0xAAD0
0xAAA0
0xAAB0
0x41
null(0x0000)
0xAAC0
0xAAD0
0x41
0xarbitrary

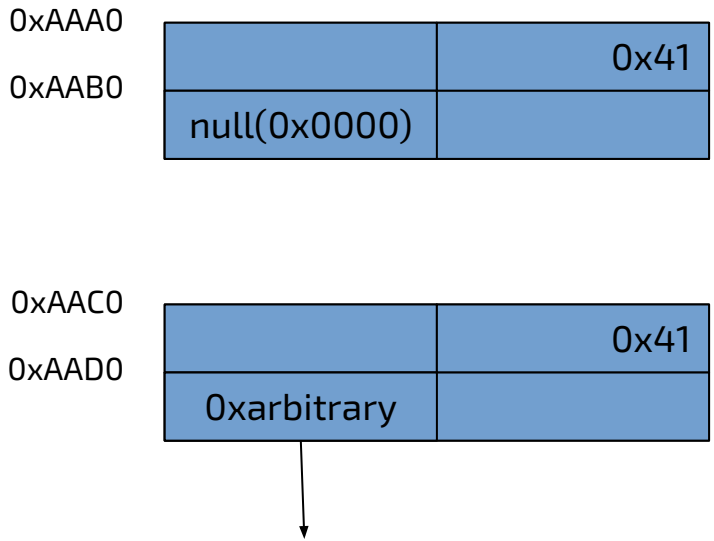
T-Cache



malloc(0x20)

Chunk1: 0xAAD0

Chunk2: 0xarbitrary



T-Cache Key (Stop Double Free) > glibc 2.29

glibc 2.34

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

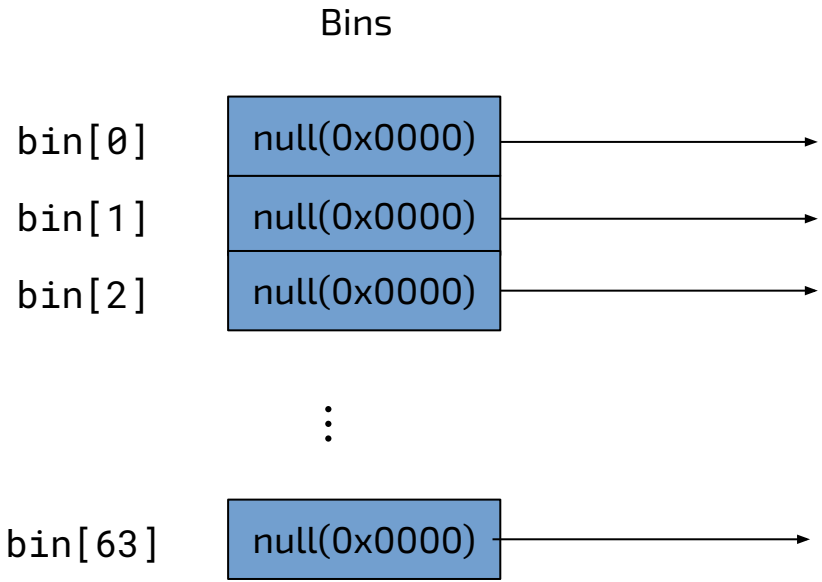
glibc 2.29 - 2.33

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

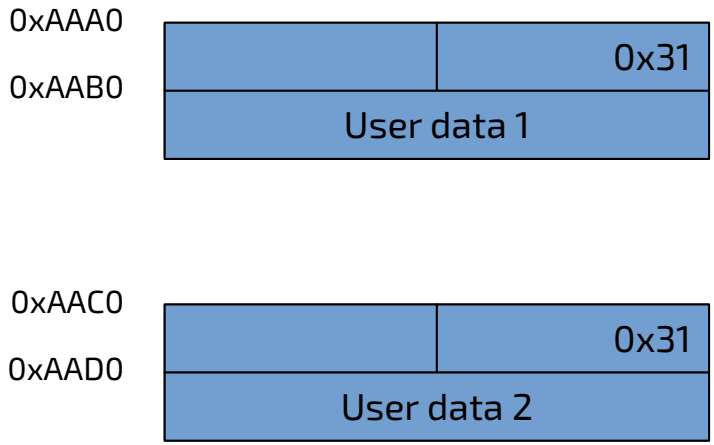
free

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

T-Cache

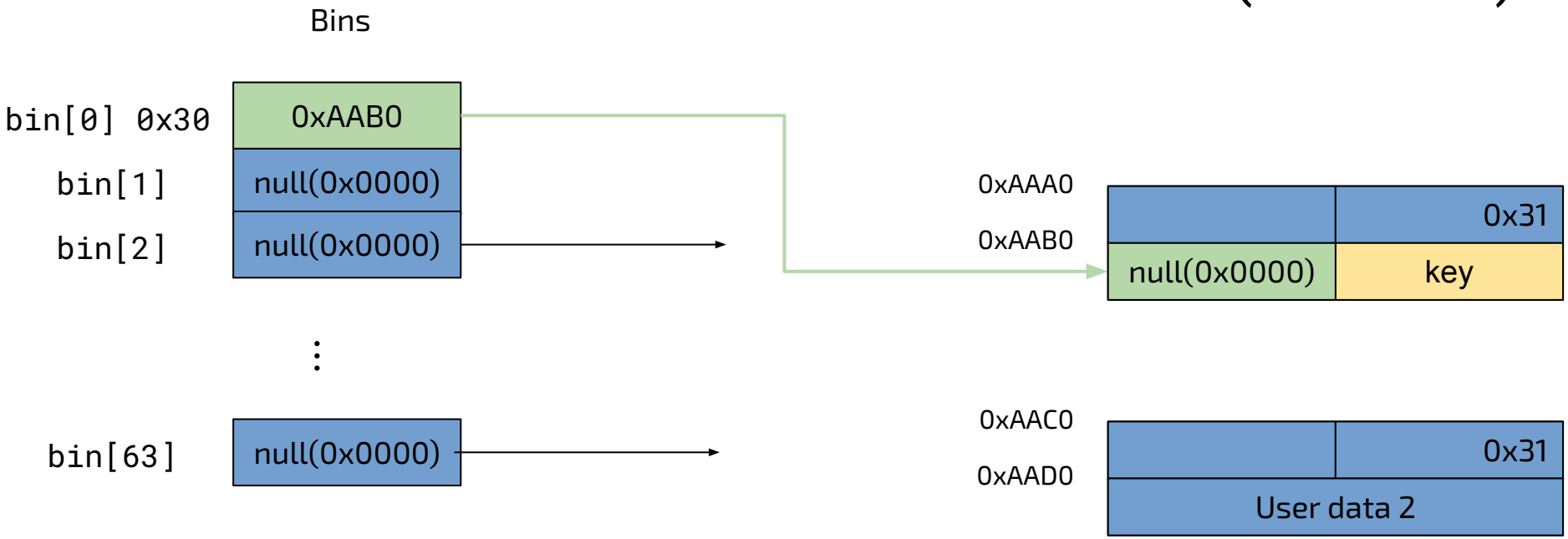


`free(0xAAB0)`



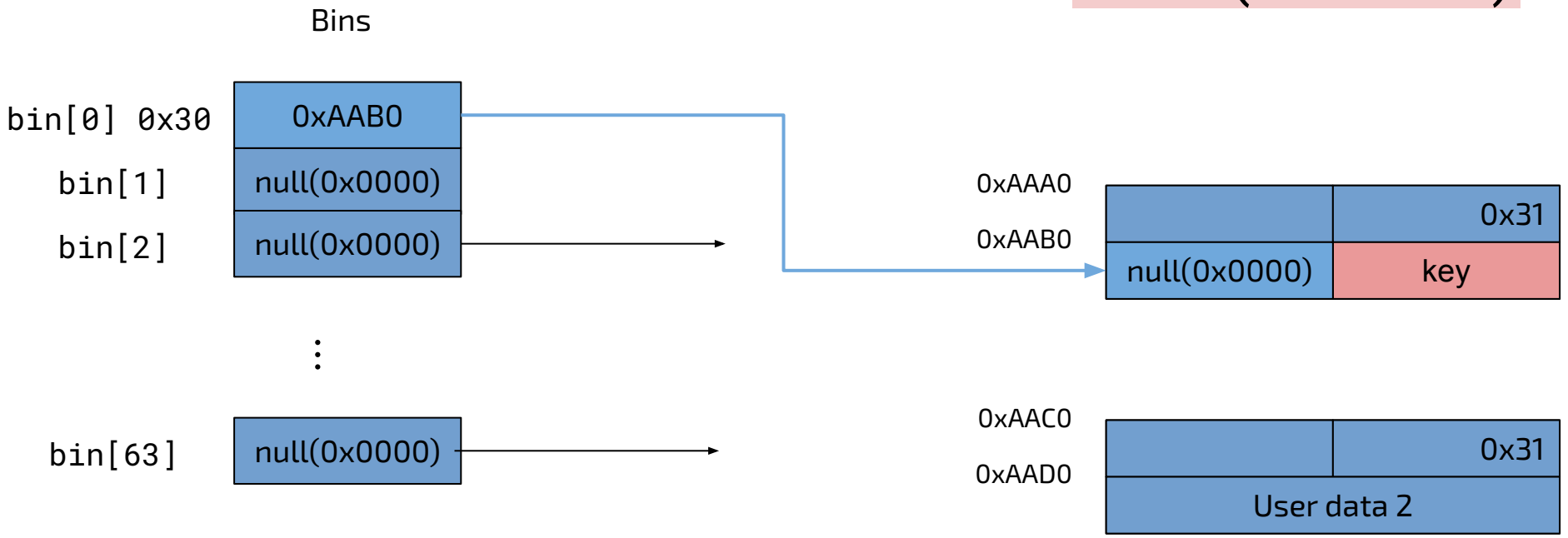
T-Cache

free(0xAAB0)



T-Cache

```
free(0xAAB0)
```



T-Cache PTR Protection (Protect from Partial Overflow)

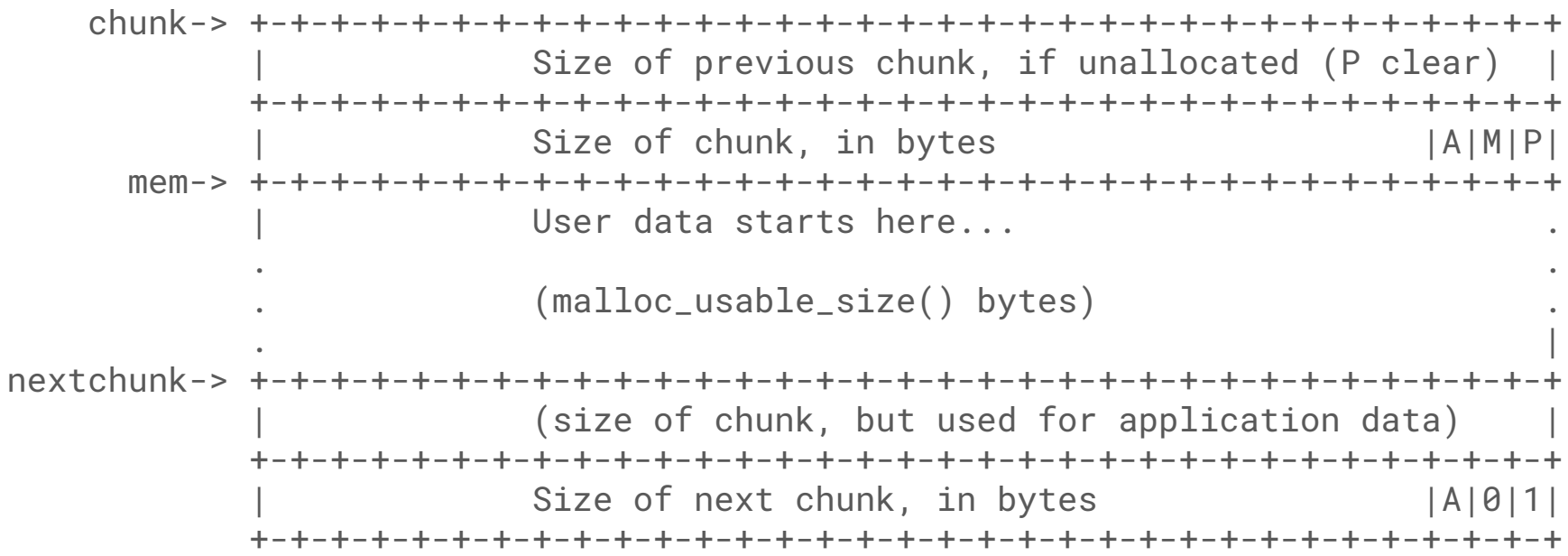
glibc >= 2.32

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

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

- You Known last 3 nimbles (12 bits) you can “**decrypt**”!
  https://github.com/shellphish/how2heap/blob/master/glibc_2.32/decrypt_safe_linking.c

Best documentation is source code.

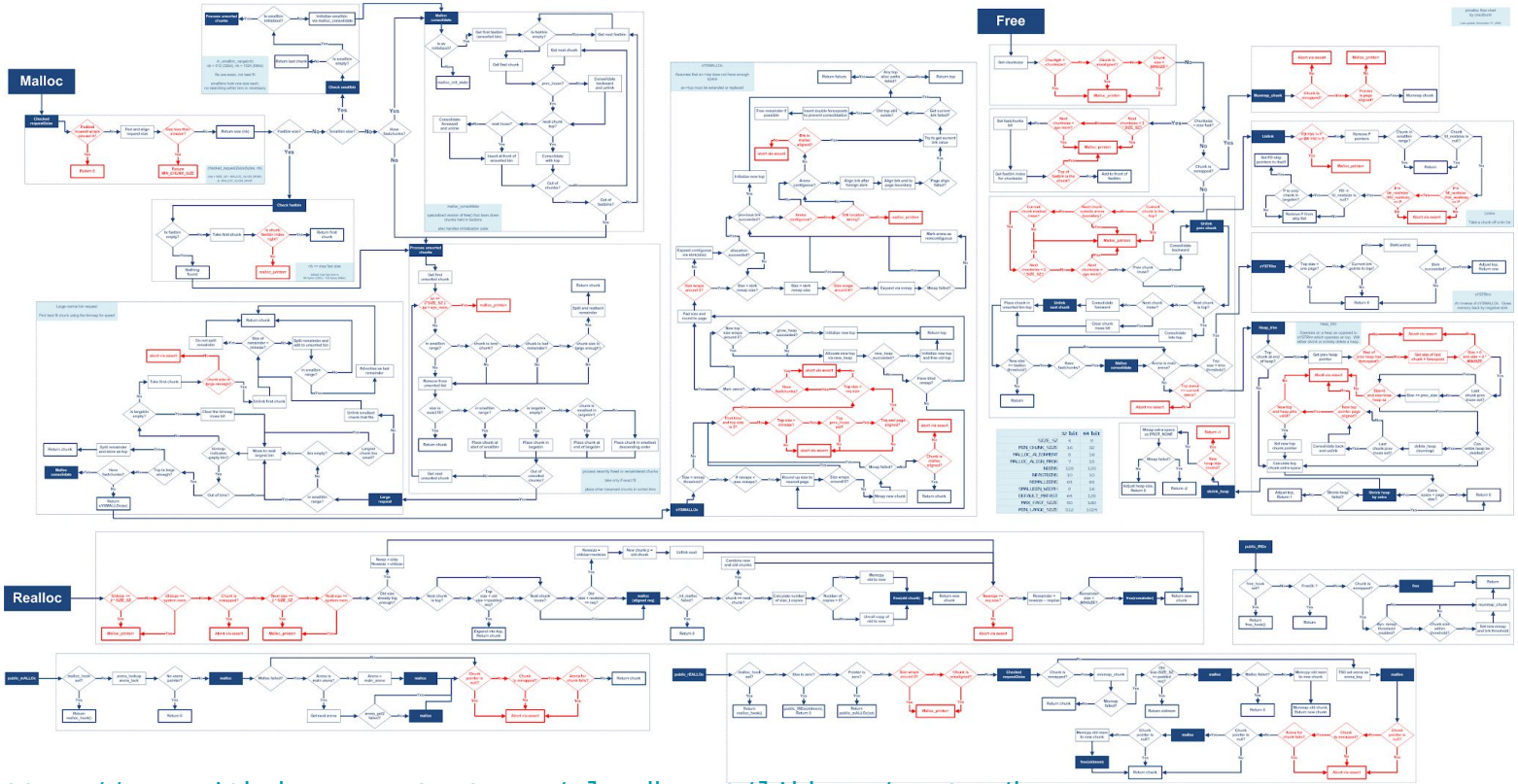


Best documentation is ~~source code~~. binary code

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)
- Source Code: https://elixir.bootlin.com/glibc/glibc-2.27/source/malloc/malloc.c

Setup

Debugging Challenges with GDB

Host the challenge:

```
socat TCP-LISTEN:4000,reuseaddr,fork EXEC:"./challenge"
```

Connect your script. (NB You script should wait.)

`python x.py` (OR `ncat 127.0.0.1 4000`)

Attach with gdb:

```
ps aux | grep challenge
```

```
sudo gdb attach 25209
```

Debugging Challenges with GDB the pwntools way

```
1. context.terminal = ['tmux', 'splitw', '-h']
2. # ssh = ssh("jinblack", "192.168.56.102")
3. r = ssh.process("./multistage")
4. gdb.attach(r,'''
5. # b * 0x004000b0
6. # b *0x4000DD
7. ''')
8. input("wait")
```

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**

If you do NOT have LibC

- **Standard** LibC
  - Two Symbols
  - LibC DB: https://libc.blukat.me/
- **Custom** LibC (needs a leak)
  - Read Out Libc
  - pwntools dynelf

pwndbg Heap Inspection

```
pwndbg> heap
Top Chunk: 0x6020a0
Last Remainder: 0

0x602000 PREV_INUSE
{
  prev_size = 0x0,
  size = 0x31,
  fd = 0xaaaaaaaaaaaaaaaa,
  bk = 0xaaaaaaaaaaaaaaaa,
  fd_nextsize = 0xaaaaaaaaaaaaaaaa,
  bk_nextsize = 0xaaaaaaaaaaaaaaaa
}
0x602030 PREV_INUSE
{
  prev_size = 0x0,
  size = 0x21,
  fd = 0x0,
  bk = 0x5555555555555555,
  fd_nextsize = 0x0,
  bk_nextsize = 0x51
}
```

```
pwndbg> bins
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x606850 ← 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x602070 ← 0x7ffff7dd37b8
smallbins
0x190: 0x602150 ← 0x7ffff7dd3938
0x210: 0x602320 ← 0x7ffff7dd39b8
0x290: 0x602570 ← 0x7ffff7dd3a38
largebins
0x3000: 0x602d50 ← 0x7ffff7dd3eb8
```

libc Debugging Symbols

- `sudo apt install libc6-dbg`
- **`pwninit/spwn`** (https://github.com/io12/pwninit https://github.com/MarcoMeinardi/spwn)
- **`eu-unstrip`** `libc-2.23.so libc-2.23.so.dbg` from the `elfutils` package
- Load it in GDB:

```
(gdb) add-symbol-file ./libc-2.23.so.debug -o 0x7ffff7a0d000 0x7ffff7a0d000
```

- GDB Auto Load:

https://sourceware.org/gdb/onlinedocs/gdb/Separate-Debug-Files.html

Get libc Debugging Symbols

```
file libc-2.23.so -> BuildID

curl -X POST -H 'Content-Type: application/json' --data \
    '{"buildid": "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"}' \
    'https://libc.rip/api/find'
```

http://archive.ubuntu.com/ubuntu/pool/main/g/glibc/libc6-dbg_2.23-0ubuntu11.3_amd64.deb

```
data.tar.xz
usr/lib/debug/.build-id/ or usr/lib/debug/lib/
```

Get libc Debugging Symbols - DEBUGINFOD

A online DB of debugging symbol from linux distros.
https://sourceware.org/elfutils/Debuginfod.html

Automatically (or manually) download debugging symbols.

Get libc Debugging Symbols - DEBUGINFOD

A online DB of debugging symbol from linux distros.
https://sourceware.org/elfutils/Debuginfod.html

```
export DEBUGINFOD_URLS="https://debuginfod.elfutils.org/"
(or setup /etc/debuginfod/)

file libc-2.23.so -> BuildID

debuginfod-find -v debuginfo BuildID
```

from **gdb** >10.1 autoload debuginfo:

```
set debuginfod enabled on
```

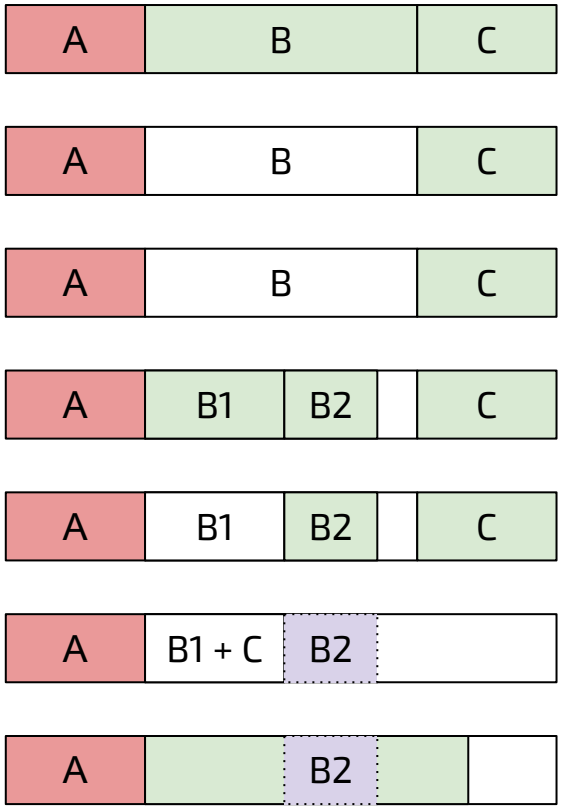
Poison Null Byte

- 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

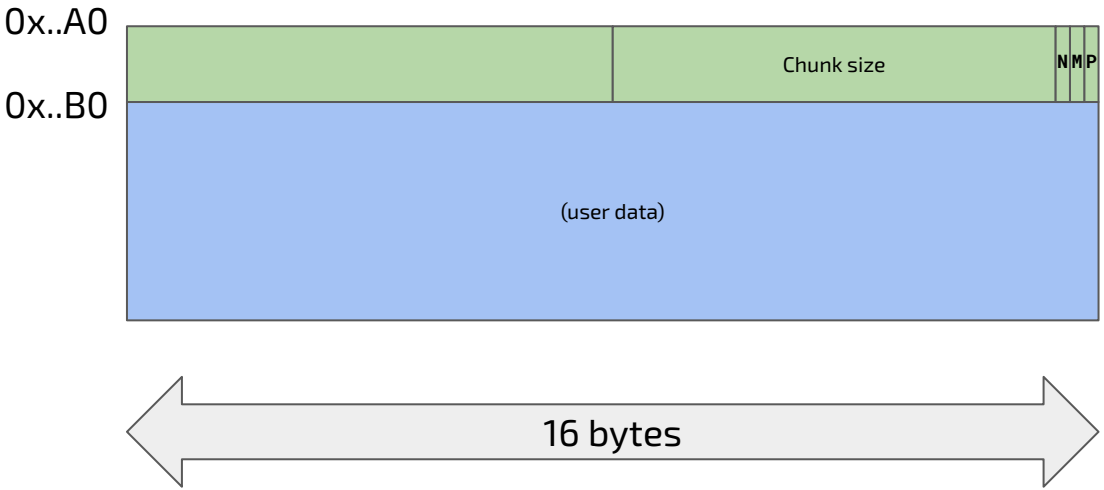
Free(B1)

Free(C) trigger the merge with the previous chunk

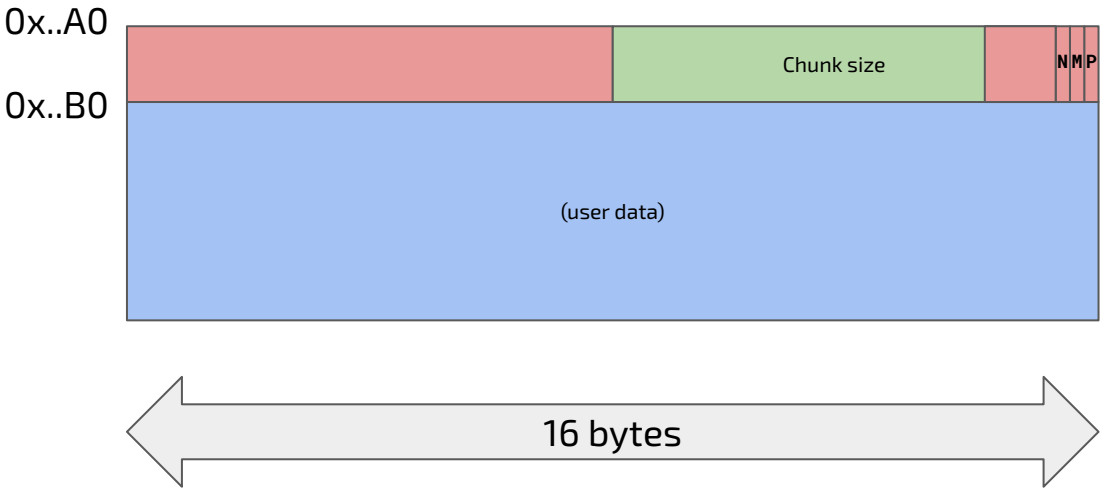
the next allocation will overlap with B2

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

Chunk



Null Byte



Poison Null Byte “FIX” - unlink

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

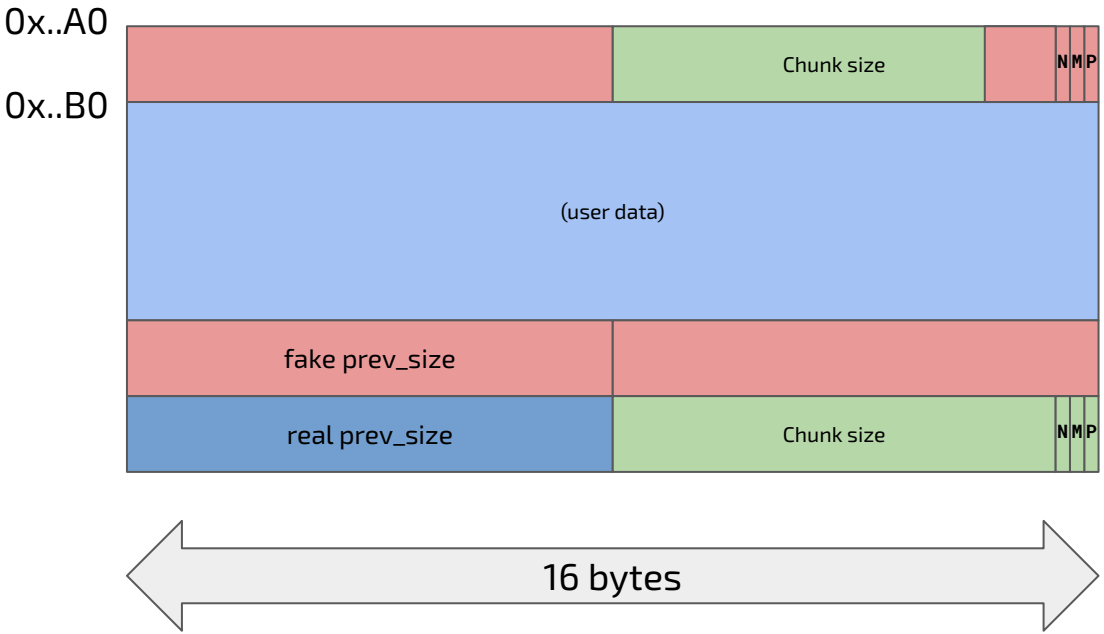
Poison Null Byte “FIX” - unlink

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

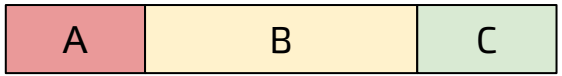
```
/* Size of the chunk below P.  Only valid if !prev_inuse (P).  */
 #define prev_size(p) ((p)->mchunk_prev_size)

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

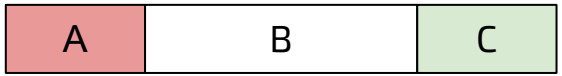
fake prev_size



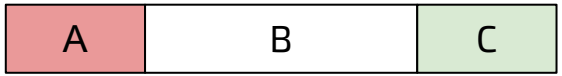
Poison Null Byte Fixed



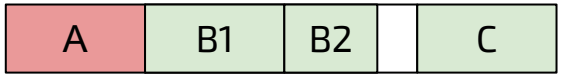
Initial Setup (B chunk needs to contain fake prev_size)



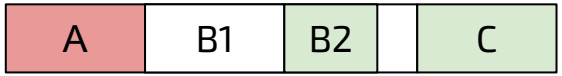
Free(B)



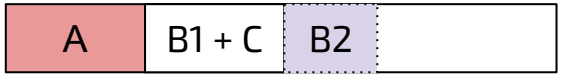
Overflow into B. Sizes goes from 0x208 to 0x200.
prev_size is not update



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