# Heap Metadata Exploits

CS-UY 3943-G / CS-GY 9223-H

# Last Time

- Looked at the data structures of the glibc allocator (ptmalloc2)
- Looked at:
  - Heap overflows
  - Use after free
- The exploits we discussed involved corrupting *application* data on the heap

# This Time

- Introduction to heap metadata exploits
- Two case studies:
  - Exploiting unlink()
  - One-byte poison null overwrite

# Metadata Exploits

- Many allocators use *inline metadata* – information about each heap chunk is stored alongside the application data
- The basic idea:
  - Corrupt metadata via heap overflow/type confusion
  - When the allocator interacts with it (e.g. during a malloc() or free()) it will have some controlled effect
    - Often *write-what-where*
  - We can use this as an exploit primitive (GOT overwrite, function pointer overwrite, start a ROP chain, ...)

# General Concept: Bins

- One thing to be aware of is that for efficiency, many allocators have different pools (***bins***) of free memory chunks
- These are usually divided up by size
- Different bins may use different data structures or allocation strategies
- Where your memory comes from when you do a malloc() depends on the size and allocations that have been made so far
- Good resource: https://heap-exploitation.dhavalkapil.com/diving_into_glibc_heap/bins_chunks.html

# Bin Types

- **Fast bins**: 10 singly-linked lists, one each for chunk sizes 16, 24, 32, 40, 48, 56, 64, 72, 80 and 88 bytes
- **Small bins**: 62 doubly-linked lists, again one for each size, where the sizes are 16, 24, ... , 504 bytes
- **Large bins**: 63 doubly-linked lists, one for each size in the ranges [512 - 568], [576 - 632], ...
- **Unsorted bin**: a single bin that stores recently-freed small and large bins. Chunks may end up here temporarily in an attempt to reuse recently-freed allocations quickly.

# More Terminology

- **Top chunk**: the chunk at the top of the arena (i.e., the top of the currently allocated heap area for this thread)
    - If there’s no other space, the allocator will try to use this chunk
    - If there’s still no space, it will grow the heap by extending the top chunk
- **Last remainder chunk**: if a chunk of the exact size requested doesn’t exist, a chunk may be split in two. The unused half will be set as the “last remainder chunk”

# unlink()

- As a warm-up, we'll look at exploiting the way the unlink() function works
- This is an older technique that no longer works due to extra sanity checks added in the allocator
- But the mechanisms are simple so it's still good as a toy example

# Heap Chunk Structure

- For a free chunk, we have:

```
struct malloc_chunk {
  INTERNAL_SIZE_T      mchunk_prev_size;
  INTERNAL_SIZE_T      mchunk_size;
  struct malloc_chunk* fd;
  struct malloc_chunk* bk;
};
```

**Size of previous chunk**

**Size of this chunk**

**Pointer to previous free chunk**

**Pointer to next free chunk**

# Heap Chunk Structure

- For a free chunk, we have:

**Note: prev_size *also* stores a flag that says whether the previous chunk is actually free!**

```
struct malloc_chunk {
  INTERNAL_SIZE_T      mchunk_prev_size;
  INTERNAL_SIZE_T      mchunk_size;
  struct malloc_chunk* fd;
  struct malloc_chunk* bk;
};
```

**Size of previous chunk**

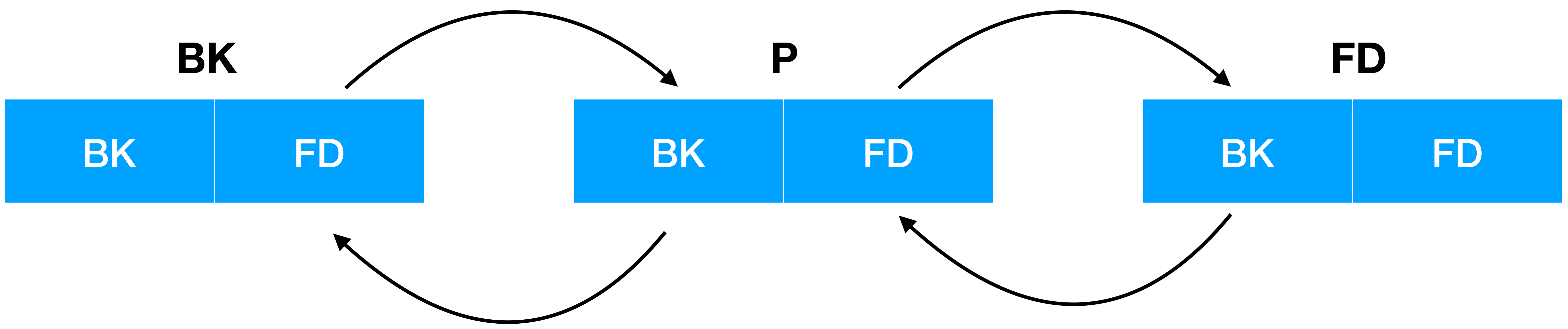
**Size of this chunk**

**Pointer to previous free chunk**

**Pointer to next free chunk**

# unlink() Macro

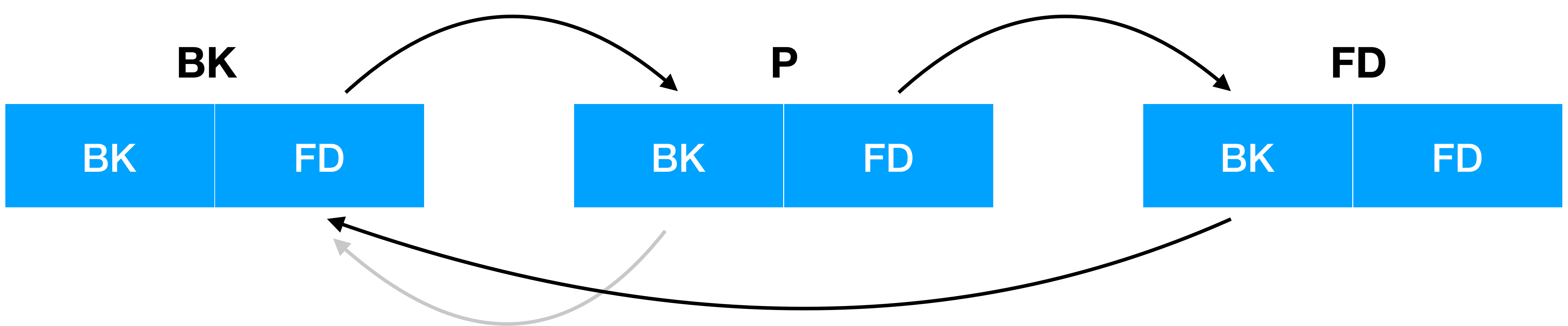
```
#define unlink( P, BK, FD ) {          \
  BK = P->bk;                          \
  FD = P->fd;                          \
  FD->bk = BK;                         \
  BK->fd = FD;                         \
}
```




**Intended Use**

# unlink() Macro

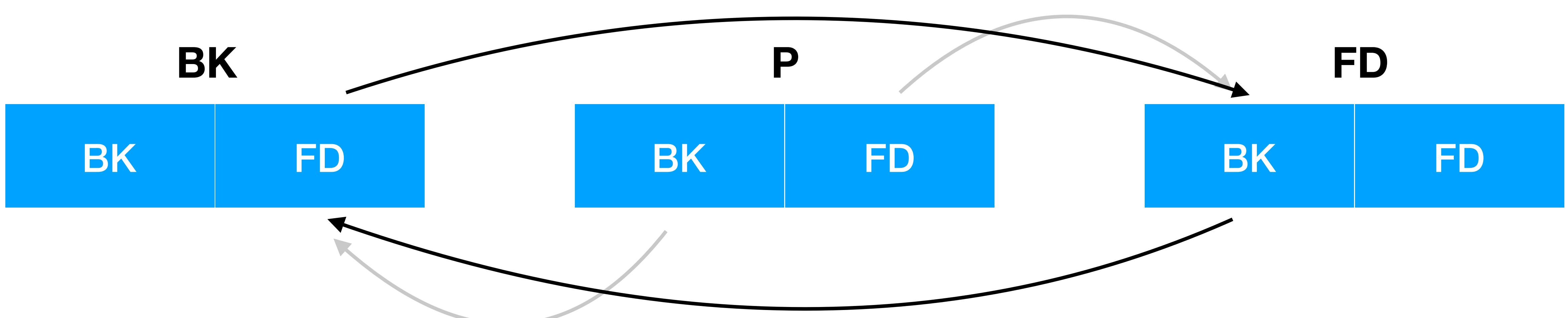
```
#define unlink( P, BK, FD ) {          \
  BK = P->bk;                          \
  FD = P->fd;                          \
  FD->bk = BK;                         \
  BK->fd = FD;                         \
}
```




**Intended Use**

# unlink() Macro

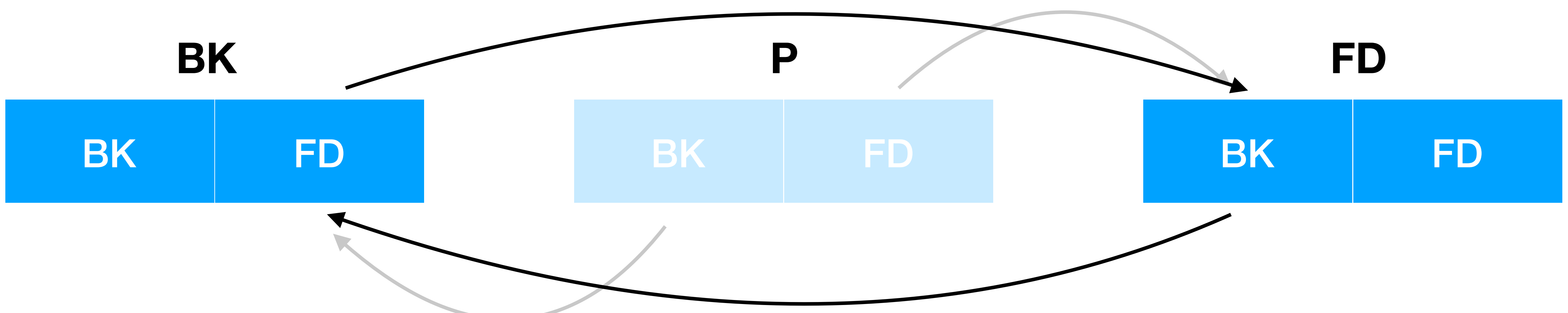
```
#define unlink( P, BK, FD ) {            \
  BK = P->bk;                            \
  FD = P->fd;                            \
  FD->bk = BK;                           \
  BK->fd = FD;                           \
}
```




**Intended Use**

# unlink() Macro

```
#define unlink( P, BK, FD ) {          \
  BK = P->bk;                          \
  FD = P->fd;                          \
  FD->bk = BK;                         \
  BK->fd = FD;                         \
}
```




**Intended Use**

# unlink() operations

- Suppose control the content of the heap chunk **P**
- What do the unlink operations do?
- Suppose:
  - chunk.BK is at offset 12 in the struct
  - chunk.FD is at offset 8 in the struct

# unlink() operations

- Then:
  - FD->bk = BK    is the same as    *(FD+12) = BK
    - “Write BK to the address FD + 12”
  - BK->fd = FD    is the same as    *(BK+8) = FD
    - “Write FD to the address BK + 8”
- Since we control both BK and FD, this is a **write-what-where**

# Exploit Example

- Suppose we want to overwrite a GOT pointer at 0x602020 (puts) with the address 0x4006e0
- We set FD = 0x602020 - 12 = 0x602014
- We set BK = 0x4006e0
- Then when we unlink(), FD->bk = BK will do *(FD + 12) = *(0x602020) = 0x4006e0

# Exploit Example

- Suppose we want to overwrite a GOT pointer at 0x602020 (puts) with the address 0x4006e0
- We set FD = 0x602020 - 12 = 0x602014
- We set BK = 0x4006e0
- Then when we unlink(), FD->bk = BK will do *(FD + 12) = *(0x602020) = 0x4006e0

**Note: this doesn't *quite* work because right afterward BK->fd = FD will try to write to a read-only code page!**

# Getting to unlink()

- Now that we understand the basic primitive, we still need to get the allocator to call **unlink**() on data we control
- Where is unlink() called?
  - Inside malloc(), to grab a chunk from the free list and use it for an allocation
  - Inside free(), to consolidate adjacent free chunks

# Getting to unlink()

- Now that we understand the basic primitive, we still need to get the allocator to call **unlink**() on data we control
- Where is unlink() called?
    - Inside malloc(), to grab a chunk from the free list and use it for an allocation
    - **Inside free(), to consolidate adjacent free chunks**

# Chunk Consolidation

- The code to consolidate a chunk with the previous one looks like:

```
    /* consolidate backward */
    if (!prev_inuse(p)) {
      prevsize = prev_size (p);
      size += prevsize;
      p = chunk_at_offset(p, -((long) prevsize));
      unlink(p, bck, fwd);
    }
```

# Chunk Consolidation

- The code to consolidate a chunk with the previous one looks like:

**Checks “in use” bit**

**Gets address of previous chunk**

```
/* consolidate backward */
if (!prev_inuse(p)) {
  prevsize = prev_size (p);
  size += prevsize;
  p = chunk_at_offset(p, -((long) prevsize));
  unlink(p, bck, fwd);
}
```

**unlink()**

# Exploit Scenario

- Suppose we can set up the heap to look like:


- And there is an overflow that lets us write past the end of Chunk 1, letting us overwrite Metadata and Chunk 2:

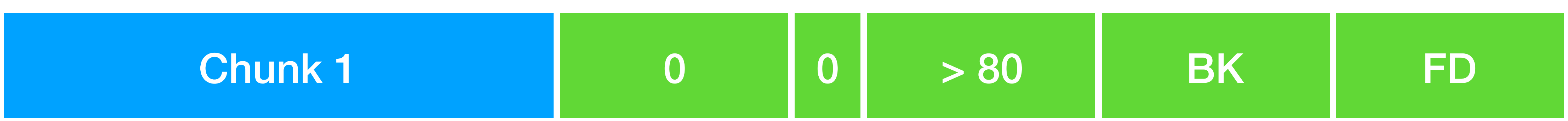

- Then if the program tries to free() Chunk 2, and we can create **fake** metadata that makes the allocator Chunk 1 is also free, we can get it to call unlink() on data we control

# Fake Chunk

- Interpreted as a free chunk, the fields we control look like


- We want to set them to values that will result in the allocator calling unlink() on our fake chunk

- We can do this by setting “in-use” (U) to 0, prev_size to 0, and size large enough that it will not go in the fast bins



# Consolidation

- When free() tries to consolidate the chunk, let's look at what the code sees now:

**"In use" is 0 so we will try to consolidate**

```
/* consolidate backward */
if (!prev_inuse(p)) {
  prevsize = prev_size (p);
  size += prevsize;
  p = chunk_at_offset(p, -((long) prevsize));
  unlink(p, bck, fwd);
}
```

**Since prev_size is 0, the address of the "previous" chunk will end up being the start of our fake chunk**

**Allocator will call unlink() on our fake chunk**
**Assuming we set FD and BK appropriately, this will execute our write-what-where!**

# Mitigations

- The unlink() trick described here no longer works
- Two checks were added:
  - Checking that the next chunk’s prev_size equals this chunk’s size
  - Checking the linked list pointers:
    - FD->bk points to current chunk
    - BK->fd points to current chunk

# Off-by-One Overwrites

- In many programs you will find “off-by-one” vulnerabilities
    - Usually from calculating the size of a buffer wrong, e.g. by forgetting about the byte for the terminating null
- This may seem fairly useless – what could you do with just a single byte overwrite?
- Even more restriction: very often the overwrite will only let you write a “0” (for the null terminator)

# Poison NULL

- Back in 1998, Olaf Kirch showed that even one-byte NULL overwrites could be dangerous
  - Demonstrated a one-byte NULL overwrite that modified the saved base pointer on the stack, eventually leading to full code execution
- In 2014, Tavis Ormandy of Google Project Zero showed that a one-byte NULL overwrite could *also* be used for a heap metadata exploit in glibc

# glibc Poison NULL

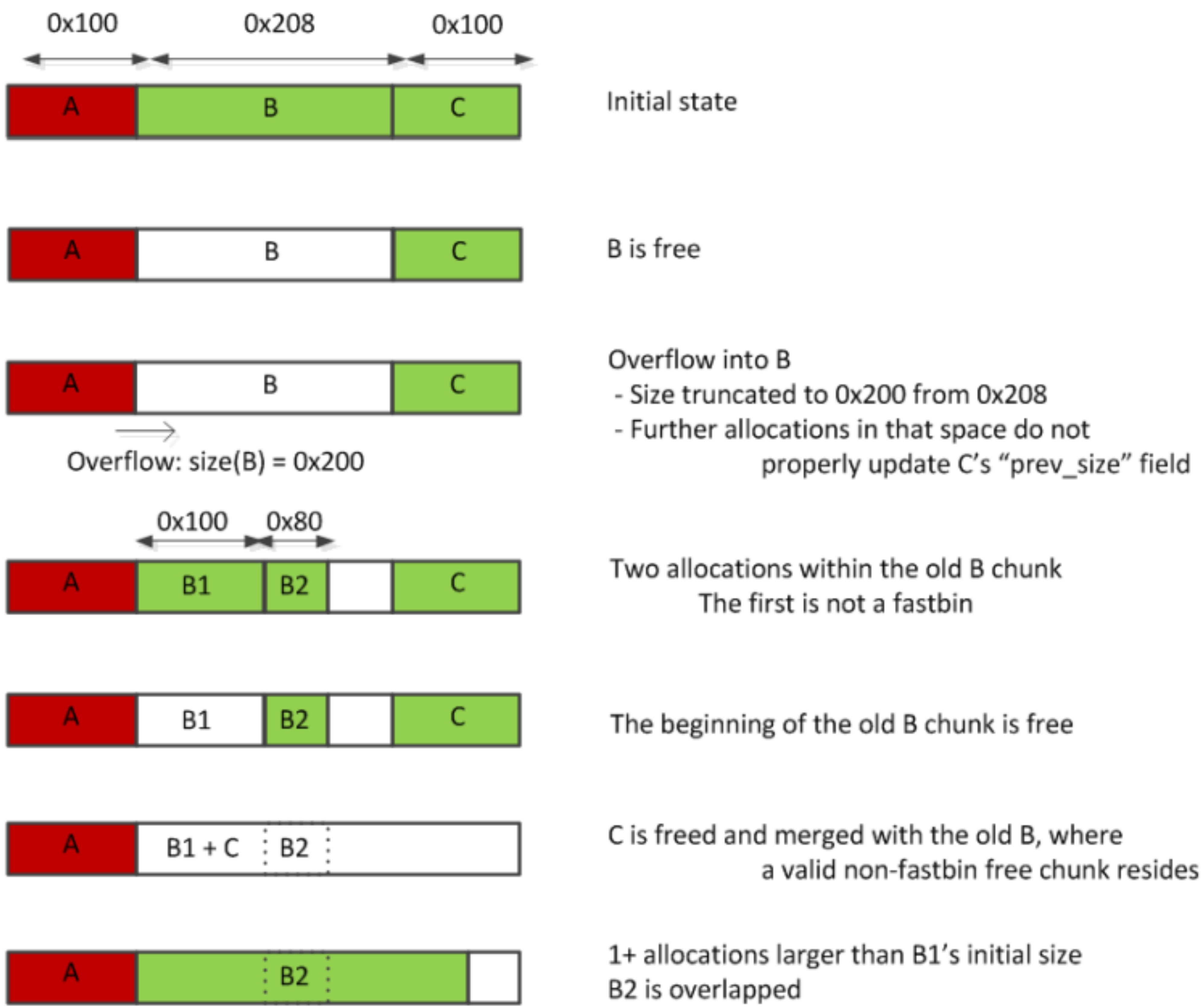
- The basic trick is that a one-byte overflow that writes a 0 will have the effect of setting the least significant byte of the next chunk’s size field to 0
- This will effectively *shrink* the size of that chunk from the allocator’s point of view
  - Any calculations done by the allocator with this incorrect size will be affected!
- We’re going to use this to eventually create two *overlapping chunks*

# Shrinking a Chunk

- Allocate three consecutive chunks: A, B, C
- Free B
- Overflow from A, making B.size smaller
- Allocate two new chunks, B1 and B2 in the free space
    - **C's prev_size is not updated correctly (why?)**

# Shrinking a Chunk

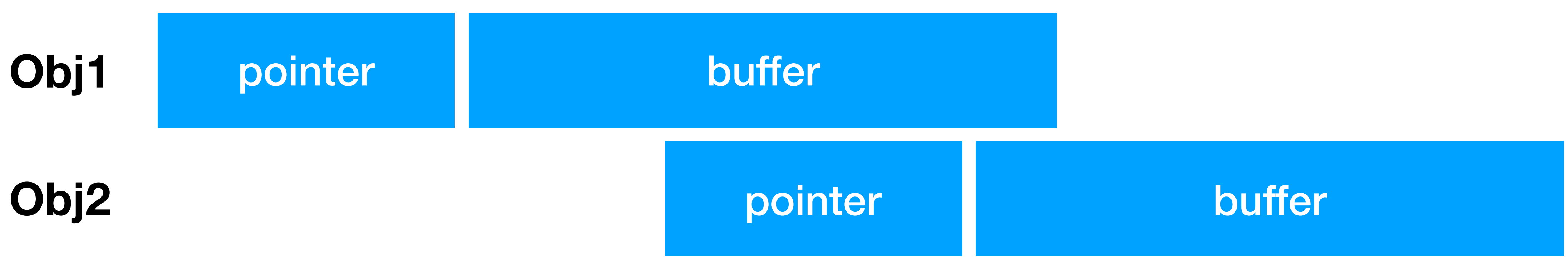
- Free B1 and C
    - When we free C, the allocator thinks B is still free because C's prev_size was not updated!
    - **The allocator will merge C and B to create one large free area**
- Now we allocate a final object. It will be placed at the start of **B** – *overlapping with **B2***




**Source: https://www.contextis.com/media/downloads/Glibc_Adventures__The_forgotten_chunks.pdf**

# Using the Overlap

- What can we do with two overlapping heap chunks?
- If we can make some (allowed) modification to a field of one of the overlapping objects, we can change a *different* field in the other




- By editing Obj1.buffer, we can set Obj2.pointer to whatever we want!

# Lots More Metadata Attacks

- There are many more types of heap metadata exploit techniques, even just with glibc
  - House of _____ (force, spirit, Einherjar, orange, lore, ...)
  - Unsorted bin attacks
  - Exploits using the “tcache” feature of glibc
- Details: https://github.com/shellphish/how2heap