The Block That Was Too Small to Free

A post-mortem on custom_mem_alloc: how an intrusive free list corrupts the block next door, and why the crash never happens where the bug is.

Abir Deol · University of Alberta, Computing Science · July 2026

Abstract. custom_mem_alloc is a thread-safe memory allocator written in C: it requests one contiguous region from the kernel with mmap and sub-allocates out of it using an intrusive free list. This is a post-mortem of the bug that cost me the most hours in that project. It presented as a double free, it was reported by the allocator at a point in the program unrelated to the actual fault, and the root cause was that I had never asked what happens when a caller requests fewer bytes than my own bookkeeping needs to occupy. The fix is two lines. Finding it was not.

1. The system

The allocator claims one large region up front with mmap and never returns it, which removes the kernel from the hot path entirely. Everything after that is bookkeeping in userspace. Each block carries a small header with its size and a free flag, allocation walks a free list looking for a block large enough to satisfy the request, and free pushes the block back onto that list.

The list is *intrusive*, which is the design decision the entire bug hangs on. There is no separate array of list nodes. When a block is free, nobody is using its payload, so the next pointer is stored inside the payload itself. A free block spends its retirement holding the address of the next free block in the memory the caller used to own. This is the standard trick and it costs zero extra space, which is exactly why it is standard.

```
typedef struct block_header {
    size_t size;                /* payload bytes, excluding this header */
    int    free;
} block_header;

/* when a block is free, the payload area is reinterpreted as: */
typedef struct free_node {
    struct free_node *next;     /* 8 bytes on x86-64 */
} free_node;
```

Read those two declarations together and the bug is already visible. I did not see it for a long time.

2. The symptom

A test program that allocated and freed a mixture of sizes in a loop would segfault. Not immediately, and not at the same iteration twice. Sometimes it ran to completion. When it did crash, the stack trace pointed inside my free list traversal, on the line that follows a next pointer.

My first reading was a double free. That is the obvious hypothesis when a free list walk explodes: some block got pushed twice, the list now contains a cycle or a stale address, and traversal eventually walks off

into memory that was never mapped. I added a guard that refused to free a block whose header already said free. It changed nothing, which should have been informative and instead just made me suspicious of the guard.

What made this genuinely difficult is that the crash location and the fault location had nothing to do with each other. The traversal that segfaults is reading a pointer that some earlier, unrelated call corrupted. By the time the program dies, the instruction that actually did the damage has been retired for thousands of cycles. The allocator was the messenger.

3. The hunt

The thing that eventually broke it open was dumping the free list on every operation instead of only on the crash. Printing every header address, size, free flag and next pointer after each call turns an intermittent segfault into a diff you can read. Two runs, one that crashed and one that did not, and the divergence point is right there.

The corrupted entry was always a block whose *neighbour* had been small. Not the block that crashed. The one physically before it in the region. And the corruption was always in the first eight bytes of that neighbour, which is where I had put the size field of the header.

That is when the two declarations above stopped being separate facts.

4. Root cause

A caller asks for four bytes. The allocator finds a block, splits it, writes a header saying the payload is four bytes, and hands back the pointer. Everything is correct so far, and the caller uses their four bytes without incident.

Then they free it. free takes the payload pointer, casts it to a free_node pointer, and writes the address of the current list head into it. That write is eight bytes wide. The payload is four bytes wide. The remaining four bytes land in whatever is physically adjacent, which is the header of the next block.

```
[hdr A][ payload A: 4 bytes ][hdr B][ payload B ...
              ^
              free(A) writes an 8-byte pointer starting here
              |<-- 4 bytes -->|<-- 4 bytes of hdr B destroyed -->|
```

So the block never gets corrupted by its own free. It corrupts its neighbour. And the neighbour does not notice until something traverses it, which might be the next allocation or might be a thousand operations later, depending on where it sits in the list. Hence the intermittency, hence the crash landing nowhere near the cause.

The design error, stated plainly: an intrusive free list imposes a floor on block size that has nothing to do with what the caller asked for. My bookkeeping needs sizeof(free_node) bytes of payload to exist in. I had written an allocator that could hand out blocks too small to ever be freed safely.

5. The fix

Enforce a floor at the top of allocation, before any splitting decision is made.

```
#define MIN_PAYLOAD  (sizeof(free_node))

static size_t normalize(size_t req) {
    if (req < MIN_PAYLOAD) req = MIN_PAYLOAD;   /* room for the next ptr */
    return ALIGN_UP(req, ALIGNMENT);            /* then the 8-byte rule */
}
```

Order matters here and it is worth being explicit about why. Rounding up to the alignment boundary first and then checking the minimum would work for these particular numbers, because eight byte alignment happens to equal the pointer size on this target. That is a coincidence, not a guarantee. Raise the floor first, then align, and the code stays correct if either constant changes.

The same reasoning applies to splitting. A block should only be split if the remainder is still large enough to be a legal free block, header plus MIN_PAYLOAD. Otherwise the remainder is left attached to the allocation as internal fragmentation, which wastes a few bytes and is strictly better than manufacturing a block that cannot be freed.

6. What it changed

The specific lesson is narrow: intrusive containers impose a minimum size on their elements, and that minimum is a property of the container, not of the caller. Anywhere you overlay bookkeeping onto user memory, that constraint exists and needs to be enforced at the boundary.

The general lesson is the one I actually use. Memory corruption reports the wrong location by design. The crash tells you which pointer was read, not which write poisoned it, and those are usually separated by a long stretch of correct execution. Chasing the crash site is chasing the symptom. What worked was making the invariant checkable at every step, so the first violation surfaces instead of the first consequence.

- Validate the structure after every mutation while debugging, not only when it fails. A cheap full traversal in a debug build turns a rare segfault into a reproducible diff.
- When a crash points at a read, ask what wrote there. The interesting instruction is almost never the one holding the gun.
- Corruption in the first bytes of a header means an overrun from the block below it. Physical adjacency is a debugging signal and I now read the region layout first.

There is also an honest note to make about testing. Every allocation size in my original tests was comfortably above eight bytes, because I was thinking in terms of structs and buffers. The bug needed a four byte request to appear. Test inputs drawn from what you imagine a caller does will not find the cases where your own assumptions live.