I. SECURITY / 2. Real-Life Undetected Damage

How fragile and easy to abuse are the "*most advanced*" memory allocators? Let's see that:

Buffer overflow

```
char sz = 8,  n = 5, *p[n]; // the code below is run twice to do free + malloc
for(int i = 0; i < n; i++) p[i] = malloc(sz); // allocate blocks
memset(p[0], 'A', n * sz);                    // overflow first block
for(int i = 0; i < n; i++) printf(" %d %p '%s'\n", i, p[i], p[i]); // print
for(int i = 0; i < n; i++) free(p[i]);        // free blocks
```

It's a texbook case: we read data (network, database, multimedia file, keyboard, command-line...) that overflows the storage buffer and overwrites what follows – and we call `free()` and `malloc()` again.

If OS/library code can corrupt your data then "*memory-safe*" languages are helpless. And if a block contains a `function pointer`, then this will lead to (probably remote) arbitrary code execution.

GLibC malloc (freelist/blocks corruption detected at free)	Block offset: 16 bytes in 132 KB Area

```
0 0x19b8010 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
1 0x19b8030 'AAAAAAAA'
2 0x19b8050 ''
3 0x19b8070 ''
4 0x19b8090 ''
*** Error in `./oob': free(): invalid next size (fast): 0x00000000019b8010 ***
Aborted
```

GLibC has never segregated its metadata, exposing it to OOB attacks. So, in-place block-accounting is corrupted by the undetected overflow until corrupted metadata is accessed by `free()`, an operation which fails due to corruption, hence the long-awaited overdue `abort()`. This protection can easily be bypassed... but, shockingly, this is much better than another allocator taking more detours to check if its integrity has been compromised – because two other allocators do not even bother to check.

JEmalloc (undetected overflows, blocks corruption)	Block offset: 4 MB in 8 MB Area

```
0 0x7f425a61d000 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
1 0x7f425a61d008 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
2 0x7f425a61d010 'AAAAAAAAAAAAAAAAAAAAAAAA'
3 0x7f425a61d018 'AAAAAAAAAAAAAAAA'
4 0x7f425a61d020 'AAAAAAAA'

0 0x7f425a61d020 'ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ'
1 0x7f425a61d018 'AAAAAAAAZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ'
2 0x7f425a61d010 'AAAAAAAAAAAAAAAAZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ'
3 0x7f425a61d008 'AAAAAAAAAAAAAAAAAAAAAAAAZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ'
4 0x7f425a61d000 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ'
```

JEmalloc "segregates" its block `freelist` so, after a successful and undetected buffer overflow, the second allocation pass has reused the freed blocks in reverse-order – without detecting anything wrong. As the last block of the first allocation-pass became the first re-used by the second pass, the overflow coverage has doubled. Freeing and reallocating overflowed blocks is undetected *by-design* (and by choice since no checks are ever done). Why bother, right?

TCmalloc (undetected overflows, blocks corruption))	Block offset: 8 MB in 9 MB Area

```
0 0x1dce008 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA8��#'
1 0x1dce010 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA8��#'
2 0x1dce018 'AAAAAAAAAAAAAAAAAAAAAAA8��#'
3 0x1dce020 'AAAAAAAAAAAAAAA8��#'
4 0x1dce028 'AAAAAAA8��#'

0 0x1dce008 'ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ8��#'
1 0x1dce010 'ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ8��#'
2 0x1dce018 'ZZZZZZZZZZZZZZZZZZZZZZZ8��#'
3 0x1dce020 'ZZZZZZZZZZZZZZZ8��#'
4 0x1dce028 'ZZZZZZZ8��#'
```

TCmalloc "segregates" its `freelist` metadata and overflows are undetected. It re-uses freed blocks in their natural order so the second buffer overflow replaces the first overflow. The second pass freeing and re-using overflowed blocks is also undetected. There are no integrity checks. Google is not afraid of memory errors, maybe because that's not its data that are compromised [17] [30].

MIMalloc (undetected OOB, blocks+freelist corruption)	Block offset: 128 KB in 64 MB Area

```
0 0x4b040020080 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
1 0x4b040020088 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
2 0x4b040020090 'AAAAAAAAAAAAAAAAAAAAAAA'
3 0x4b040020098 'AAAAAAAAAAAAAAAA'
4 0x4b0400200a0 'AAAAAAAA'

0 0x4b0400200a8 'ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ'
1 0x4b0400200b0 'ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ'
2 0x4b0400200b8 'ZZZZZZZZZZZZZZZZZZZZZZZ'
3 0x4b0400200c0 'ZZZZZZZZZZZZZZZZ'
4 0x4b0400200c8 'ZZZZZZZZ'
```

MIMalloc keeps using in-place `freelist` metadata (like GLibC). No freed block are re-used because it keeps allocating new space for (much) longer than our loop. Not reallocating blocks is why the two overflows are undetected even after two free/malloc loops – and despite the fact that MIMalloc uses compiler options designed to catch buffer overflows [35] as recommended by the NSA [28].

Enabling MIMalloc "secure" options (encoded `freelist`, random blocks) requires a recompilation:

MIM "secure" (undetected OOB, blocks/free corruption)	Block offset: 0-4 KB in 63 MB Area

```
0 0x2bbec020580 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAZU"
1 0x2bbec020500 ''
2 0x2bbec020940 ''
3 0x2bbec0207c0 ''
4 0x2bbec020400 ''

0 0x2bbec020e00 'ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZuU"
1 0x2bbec020dc0 ''
2 0x2bbec020a40 ''
3 0x2bbec020280 ''
4 0x2bbec020a00 ''
```

Overflows and `freelist` corruptions are also undetected by MIMalloc "secure". Like with GLibC, metadata corruption will cause an `abort()` only after a new allocation meets a corrupted `freelist`. Microsoft just delays the checks made by `free()` and `malloc()` with a pseudo-random numbers generator to pick a new random block when `malloc()` is invoked. Detecting corruption long after damage has been done lets attackers to their job and hide their tracks to stay undetected.

All these allocators could detect corruption with instant integrity checks. They just don't do it. As a result, application/system buffer overflows and metadata corruption are... undetected.

Memory corruption can generate immediate consequences (crash, control-flow hijacking) or long-delayed visible effects (crash, garbage state and output). Primary causes are increasingly more difficult to find as they get older – hidden behind smoke and mirrors ("*garbage in, garbage out*").

Then developers are blamed. In reality, they might be totally innocent – the OS allocator is guilty:

SLIMalloc (on-the-fly detection, blocking and reporting – no overflow, no corruption)

```
> OOB: memset() accessed:40 block-size:8 OOB:32
  caller ./oob.c:9 main()
0 0x41e254c6a700 'AAAAAAAA'
1 0x41e254c6a708 ''
2 0x41e254c6a710 ''
3 0x41e254c6a718 ''
4 0x41e254c6a720 ''

> OOB: memset() accessed:40 block-size:8 OOB:32
  caller ./oob.c:14 main()
0 0x41e254c6a720 'ZZZZZZZZ'
1 0x41e254c6a718 ''
2 0x41e254c6a710 ''
3 0x41e254c6a708 ''
4 0x41e254c6a700 ''
```

```
Here heap->opt.rnd_block
= false so block-address
randomization is not
active (all blocks are
allocated with their
natural order and
increment - instead of
at random addresses) but
the result is the same
in both cases.

Changing options is done
on-the-fly, on a per-
heap basis, without
recompilation.
```

SLIMalloc is the only allocator to (1) traverse the first allocation/free pass unharmed, and to (2) complete the second allocation/free pass without damage, by (3) detecting, blocking, locating and reporting the exact location and precise cause of the OOB memory access violation on-the-fly.

JE/TC/MIMalloc "segregate" metadata in their block-area... exposed to undetected OOBs, defeating the very purpose of segregated metadata, which was initially to serve security. [14]

50 years ago, C/*asm* programmers had the technical background to understand the implications:

> "*If an error in an operating system program allows a penetration program to work, that program will work every time it is executed – without detection.*" [6]
> – Electronics Systems Division, Air Force Systems Command, Computer Security Developments Summary, report MCI-74-1 (1973)

The flawed memory-allocator vendors (claiming to be the root cause of 70-90% of all vulnerabilities for decades) now want to secure the world with unsafe [34] "*memory-safe*" languages. Ahem, really?