§ THE NODEBOOK · CHAPTER 6

Buffer *allocation patterns.*

Three ways to make a Buffer, three different cost profiles. This chapter covers `alloc`, `allocUnsafe`, the four `from` overloads, the 8 KB internal pool, and the fragmentation cliffs that bite at scale.

CHAPTER 6 · ALLOC · ALLOCUNSAFE · POOL

§ SECTION OPENER

Three ways to make a *Buffer.*

Same surface area, very different contracts. One is safe and slow, one is fast and dangerous, one is a shape-shifter whose behavior depends on what you hand it.

`BUFFER.ALLOC(SIZE)`	requests memory, then writes 0x00 over every byte
`BUFFER.ALLOCUNSAFE(SIZE)`	requests memory, hands it back as-is, contents are whatever was there
`BUFFER.FROM(SOURCE)`	copies or views, depending on the source type

§ DEFAULT

alloc gives you a *known-good buffer.*

Two operations: ask the OS for the bytes, then iterate and write a zero to every one. Predictable, secure, blank slate. Whatever the process was doing a millisecond before doesn't matter.

JS

```
const buf = Buffer.alloc(10);

console.log(buf);
// <Buffer 00 00 00 00 00 00 00 00 00 00>
```

§ THE TRADE

Zero-fill is a *memset loop.*

That second step isn't free. It's a C++ memset down to memory. Negligible for small buffers, lost in the noise of any I/O bound handler. In a tight loop on a hot path, it can pin a CPU.

JS

```
function processChunk(chunk) {
  const workBuffer = Buffer.alloc(chunk.length);
  chunk.copy(workBuffer);
  // ... watermarking logic
}
```

WARNING

Thousands of chunks per second through this and the profiler will point straight at the alloc call, not the image work.

§ ALLOCUNSAFE

Skip the fill, *keep the bytes.*

allocUnsafe asks for memory and hands you the pointer. No zero pass. Faster because it does less work. The contents are whatever was sitting in that region: zeros, junk, or fragments of recently freed buffers.

JS

```
const buf = Buffer.allocUnsafe(10);
// contents are uninitialized memory
```

X PITFALL

"Unsafe" doesn't mean "might throw". It means uninitialized memory. The OS or the buffer pool gives you a pointer to a location it hasn't bothered to clean.

§ PATTERN TO AVOID

Allocate large, *write small.*

Classic leak shape. Estimate the size too high, write the real payload into the front, never touch the rest. Send the whole buffer downstream. The tail bytes can be anything the pool last held: a JWT, a session id, a key.

JS

```
function handleReport(req, res) {
  const reportBuffer = Buffer.allocUnsafe(1024);

  const reportData = generateReportData(); // 500 bytes
  reportData.copy(reportBuffer, 0);

  // remaining 524 bytes are uninitialized
  res.send(reportBuffer);
}
```

X PITFALL

The bug isn't loud. No crash, no error, just a tail of leftover memory tagged onto every response.

§ MEMORY ARCHITECTURE

Buffer bytes live *off the V8 heap.*

The JS-side Buffer is a small handle on the V8 heap. The bytes are raw memory Node requests directly from the OS through its C++ core. V8 never walks those bytes during GC.

1. Node C++ asks the OS for N bytes
2. The OS returns a memory address
3. C++ wraps the address with length metadata
4. JS gets a small Buffer object holding a pointer to that memory

§ EXTERNAL VS HEAPUSED

50 MB shows up in *external.*

Allocate 50 MB of Buffer, then read process.memoryUsage(). The V8 heap stays tiny. The bytes are accounted for under external and arrayBuffers.

JS

```
const bigBuffer = Buffer.alloc(50 * 1024 * 1024);
console.log(process.memoryUsage());

// {
//   rss: 39845888,
//   heapTotal: 5341184,
//   heapUsed: 3638280,
//   external: 53790468,
//   arrayBuffers: 52439315
// }
```

§ POOLING

Small buffers come from *an 8 KB slab.*

malloc has overhead. Node pre-allocates an 8 KB pool and slices small buffers off it. allocUnsafe and from both pull from this pool. alloc can use it too, but still has to zero what it gets.

POOL SIZE	8 KB by default (Buffer.poolSize)
WHO POOLS	allocUnsafe, Buffer.from for small inputs
ON FREE	slice marked available, bytes left in place
LARGER THAN 8 KB	goes straight to the OS, never touches the pool

§ FAILURE MODE

The pool recycles *your own secrets.*

The data inside an allocUnsafe buffer is rarely random. It's recently freed bytes from your own process, often structured data from another request handler.

1. Request 1: allocate 200 B from the pool, store a JWT
2. Request 1 ends: slice marked free, JWT bytes still there
3. Request 2: allocUnsafe(500), reuses the same slot plus 300 more
4. First 200 B of the new buffer are User A's session data

X PITFALL

Not theoretical. The pool turns the address space into a high-speed recycler of recently-touched secrets.

§ THE RULE

Only safe with an *immediate full overwrite.*

One acceptable shape. Allocate, then immediately hand the buffer to a syscall that fills every byte from start to finish. The window where uninitialized data exists is closed by the next instruction.

JS

```
const fd = fs.openSync("script.js", "r");
const size = fs.fstatSync(fd).size;

const buf = Buffer.allocUnsafe(size);
const bytesRead = fs.readSync(fd, buf, 0, size, 0);
```

⚠ WARNING

Any if branch, early return, or try/catch between the alloc and the overwrite turns this into a leak waiting to happen.

§ BUFFER.FROM(STRING)

Encodes, then *copies.*

Most common use. Node walks the string, transcodes characters into the requested encoding, allocates a new buffer, copies the bytes in. Fast for UTF-8, but it is a copy, not a view.

JS

```
const buf = Buffer.from("hello world", "utf8");
// <Buffer 68 65 6c 6c 6f 20 77 6f 72 6c 64>
```

§ BUFFER.FROM(BUFFER)

Buffer source means *full copy.*

Pass an existing Buffer and you get a fresh allocation of the same size with the bytes copied over. Mutations on one don't reach the other. Predictable, but watch the cost on large copies.

JS

```
const buf1 = Buffer.from("learn_node");
const buf2 = Buffer.from(buf1);

buf2[0] = 0x6e; // 'n'

console.log(buf1.toString()); // 'learn_node'
console.log(buf2.toString()); // 'nearn_node'
```

§ THE TRAP

ArrayBuffer source is a *shared view.*

Pass an ArrayBuffer and Buffer.from creates a view over that same memory. No copy. If anything else writes to the underlying ArrayBuffer, your "buffer" silently changes.

JS

```
const arrayBuffer = getUploadAsArrayBuffer();

// might NOT be a copy, may share memory with arrayBuffer
const headerBuffer = Buffer.from(arrayBuffer, 0, 16);

sanitizeFileInMemory(arrayBuffer); // mutates the same memory
```

✕ PITFALL

The headerBuffer reads correctly the first time, then changes under you when something else touches the underlying ArrayBuffer.

§ THE TRIGGER

Race window, *silent corruption.*

The bug only shows up when the sanitizer runs in the gap between the header check and the second read. Async code yields, another task gets scheduled, the shared memory mutates.

1. Read first 16 bytes from headerBuffer, confirm PNG
2. await db lookup for permissions, control yields
3. sanitizeFileInMemory runs, scrubs byte 10 of arrayBuffer
4. db query resolves, code reads headerBuffer again, byte 10 differs

§ THE FIX

Allocate fresh, *copy explicitly.*

When you don't own the source memory and need stability, don't trust Buffer.from to do the right thing. Allocate your own buffer, view the source, copy across.

JS

```js
const arrayBuffer = getUploadAsArrayBuffer();

const headerBuffer = Buffer.alloc(16);
const sourceView = Buffer.from(arrayBuffer, 0, 16);
sourceView.copy(headerBuffer);

// headerBuffer is now decoupled from arrayBuffer
sanitizeFileInMemory(arrayBuffer);
```

§ MEASUREMENTS

alloc vs allocUnsafe, *10k iterations.*

Same loop, three sizes. Below the pool, allocUnsafe wins on raw speed. Just above the pool, the malloc overhead can flip the result. At 1 MB, zero-fill dominates.

SIZE	ALLOC	ALLOCUNSAFE	RATIO
100 B (pooled)	3.11 ms	1.23 ms	~2.5x faster
10 KB (non-pooled)	9.65 ms	12.84 ms	1.3x slower
1 MB (non-pooled)	1151 ms	988 ms	1.2x faster

§ DOS VIA FROM

Decoded size, not *payload size.*

Buffer.from(string, "base64") allocates based on the decoded length, not the request body length. A small request that decodes huge gives an attacker a cheap memory amplifier.

JS

```
const body = JSON.parse(req.body);
const dataBuffer = Buffer.from(body.data, "base64");
```

⚠ WARNING

Cap the input string length before you call Buffer.from. The convenience hides an unbounded allocation behind a one-liner.

§ ALLOCATION STRATEGY

Buffer.concat in a loop *thrashes the GC.*

Each concat allocates a new buffer big enough to hold both inputs, copies both in, then drops the old one. 100 chunks = 100 allocations and 99 throwaway buffers. The function is right, the loop is wrong.

JS

```
let internalBuffer = Buffer.alloc(0);

function handleData(chunk) {
  internalBuffer = Buffer.concat([internalBuffer, chunk]);
  // ... try to parse messages from internalBuffer
}
```

⚠ WARNING

Manage a single growing buffer with a write offset, or stream parse without buffering. Don't let the strategy turn O(n) work into O(n^2).

§ FRAGMENTATION

Free in the middle, *big alloc fails.*

Long-running processes that allocate and free large variable-size buffers leave holes. Total free memory looks fine, but no single contiguous block is large enough for the next request.

1. alloc 1 MB (A), 2 MB (B), 1 MB (C). Layout: [A][B][C]
2. free B. Layout: [A][----2 MB hole----][C]
3. alloc 3 MB. 2 MB free isn't contiguous, request goes to OS for more
4. rss creeps up over time, heapUsed and external stay stable

§ DECISION FRAMEWORK

Three questions before you *type Buffer.*

A short flowchart that keeps you out of trouble. Default to safety. Demand evidence before optimizing. Demand a proof of total overwrite before going unsafe.

1. 1. Default to Buffer.alloc(size). Always.
2. 2. Only deviate with a CPU profile pointing at this exact line
3. 3. allowed if the next ops unconditionally fill every byte 0..size-1
4. Otherwise: rethink the algorithm, don't reach for allocUnsafe

§ MIGRATION

new Buffer is *deprecated.*

The old constructor is unsafe and behavior changes by argument type. With a number, it skips zero-fill. Replace every call site. Most Node versions issue a runtime deprecation warning.

```
- const unsafeBuf = new Buffer(1024);
+ const safeBuf = Buffer.alloc(1024);
- const oldWay = new Buffer("hello", "utf8");
+ const newWay = Buffer.from("hello", "utf8");
```

§ RECAP

One sentence to *hold onto.*

alloc is the safe default. allocUnsafe is a profiler-justified, fully-overwritten exception. from copies for strings, arrays, and Buffers, but views ArrayBuffers, so copy explicitly when you need stability.

Default to alloc. Earn allocUnsafe with a profile and a complete overwrite. Triple-check what you pass to from.
