# Debugging memory leaks in WebAssembly using Emscripten


- Suspicious pattern
- Looking for memory bugs
  - Discovering more issues with sanitizers
- Issues with shared state
- Building a safe wrapper
- Takeaways


Squoosh.app is a PWA that illustrates just how much different image codecs and settings can improve image file size without significantly affecting quality. However, it's also a technical demo showcasing how you can take libraries written in C++ or Rust and bring them to the web.

Being able to port code from existing ecosystems is incredibly valuable, but there are some key differences between those static languages and JavaScript. One of those is in their different approaches to memory management.

While JavaScript is fairly forgiving in cleaning up after itself, such static languages are definitely not. You need to explicitly ask for a new allocated memory and you really need to make sure you give it back afterwards, and never use it again. If that doesn't happen, you get leaks... and it actually happens fairly regularly. Let's take a look at how you can debug those memory leaks and, even better, how you can design your code to avoid them next time.

## Suspicious pattern

Recently, while starting to work on Squoosh, I couldn't help but notice an interesting pattern in C++ codec wrappers. Let's take a look at an ImageQuant wrapper as an example (reduced to show only object creation and deallocation parts):

```
liq_attr* attr;
liq_image* image;
liq_result* res;
uint8_t* result;

RawImage quantize(std::string rawimage,
                  int image_width,
                  int image_height,
                  int num_colors,
                  float dithering) {
  const uint8_t* image_buffer = (uint8_t*)rawimage.c_str();
  int size = image_width * image_height;

  attr = liq_attr_create();
  image = liq_image_create_rgba(attr, image_buffer, image_width, image_height, 0);
  liq_set_max_colors(attr, num_colors);
  liq_image_quantize(image, attr, &res);
  liq_set_dithering_level(res, dithering);
  uint8_t* image8bit = (uint8_t*)malloc(size);
  result = (uint8_t*)malloc(size * 4);

  // …

  free(image8bit);
  liq_result_destroy(res);
  liq_image_destroy(image);
  liq_attr_destroy(attr);

  return {
    val(typed_memory_view(image_width * image_height * 4, result)),
    image_width,
    image_height
  };
}

void free_result() {
  free(result);
}
```

JavaScript (well, TypeScript):

```
export async function process(data: ImageData, opts: QuantizeOptions) {
  if (!emscriptenModule) {
    emscriptenModule = initEmscriptenModule(imagequant, wasmUrl);
  }
  const module = await emscriptenModule;

  const result = module.quantize(/* … */);

  module.free_result();

  return new ImageData(
    new Uint8ClampedArray(result.view),
    result.width,
    result.height
  );
}
```

Do you spot an issue? Hint: it's use-after-free, but in JavaScript!

In Emscripten, `typed_memory_view` returns a JavaScript `Uint8Array` backed by the WebAssembly (Wasm) memory buffer, with `byteOffset` and `byteLength` set to the given pointer and length. The main point is that this is a TypedArray *view* into a WebAssembly memory buffer, rather than a JavaScript-owned copy of the data.

When we call `free_result` from JavaScript, it, in turn, calls a standard C function `free` to mark this memory as available for any future allocations, which means the data that our `Uint8Array` view points to, can be overwritten with arbitrary data by any future call into Wasm.

Or, some implementation of `free` might even decide to zero-fill the freed memory immediately. The `free` that Emscripten uses doesn't do that, but we are relying on an implementation detail here that cannot be guaranteed.

Or, even if the memory behind the pointer gets preserved, new allocation might need to grow the WebAssembly memory. When `WebAssembly.Memory` is grown either via JavaScript API, or corresponding `memory.grow` instruction, it invalidates the existing `ArrayBuffer` and, transitively, any views backed by it.

Let me use the DevTools (or Node.js) console to demonstrate this behavior:

```
> memory = new WebAssembly.Memory({ initial: 1 })
Memory {}

> view = new Uint8Array(memory.buffer, 42, 10)
```

```
Uint8Array(10) [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
// ^ all good, we got a 10 bytes long view at address 42

> view.buffer
ArrayBuffer(65536) {}
// ^ its buffer is the same as the one used for WebAssembly memory
//   (the size of the buffer is 1 WebAssembly "page" == 64KB)

> memory.grow(1)
1
// ^ let's say we grow Wasm memory by +1 page to fit some new data

> view
Uint8Array []
// ^ our original view is no longer valid and looks empty!

> view.buffer
ArrayBuffer(0) {}
// ^ its buffer got invalidated as well and turned into an empty one
```

Finally, even if we don't explicitly call into Wasm again between `free_result` and `new Uint8ClampedArray`, at some point we might add multithreading support to our codecs. In that case it could be a completely different thread that overwrites the data just before we manage to clone it.

# Looking for memory bugs

Just in case, I've decided to go further and check if this code exhibits any issues in practice. This seems like a perfect opportunity to try out the new(ish) Emscripten sanitizers support that was added last year and presented in our WebAssembly talk at the Chrome Dev Summit:

https://www.youtube.com/watch?v=kZrl91SPSpc

Sanitizers are special tools that instrument the code with auto-generated checks during compilation, which can then help catch common bugs during runtime. Since they introduce runtime overhead, they're primarily used during development, although in some critical applications they're sometimes enabled on a [subset of] production environments as well.

In this case, we're interested in the AddressSanitizer, which can detect various pointer- and memory-related issues. To use it, we need to recompile our codec with `-fsanitize=address`:

```
emcc \
  --bind \
  ${OPTIMIZE} \
  --closure 1 \
  -s ALLOW_MEMORY_GROWTH=1 \
  -s MODULARIZE=1 \
  -s 'EXPORT_NAME="imagequant"' \
  -I node_modules/libimagequant \
  -o ./imagequant.js \
  --std=c++11 \
  imagequant.cpp \
  -fsanitize=address \
  node_modules/libimagequant/libimagequant.a
```

This will automatically enable pointer safety checks, but we also want to find potential memory leaks. Since we're using ImageQuant as a library rather than a program, there is no "exit point" at which Emscripten could automatically validate that all memory has been freed.

Instead, for such cases the LeakSanitizer (included in the AddressSanitizer) provides the functions `__lsan_do_leak_check` and `__lsan_do_recoverable_leak_check`, which can be manually invoked whenever we expect all memory to be freed and want to validate that assumption. `__lsan_do_leak_check` is meant to be used at the end of a running application, when you want to abort the process in case any leaks are detected, while `__lsan_do_recoverable_leak_check` is more suitable for library use-cases like ours, when you want to print leaks to the console, but keep the application running regardless.

Let's expose that second helper via Embind so that we can call it from JavaScript at any time:

```
#include <sanitizer/lsan_interface.h>

// …

void free_result() {
  free(result);
}

EMSCRIPTEN_BINDINGS(my_module) {
  function("zx_quantize", &zx_quantize);
  function("version", &version);
  function("free_result", &free_result);
  function("doLeakCheck", &__lsan_do_recoverable_leak_check);
```

```
}
```

And invoke it from the JavaScript side once we're done with the image. Doing this from the JavaScript side, rather than the C++ one, helps to ensure that all the scopes have been exited and all the temporary C++ objects were freed by the time we run those checks:

```
  // …

  const result = opts.zx
    ? module.zx_quantize(data.data, data.width, data.height, opts.dither)
    : module.quantize(data.data, data.width, data.height, opts.maxNumColors, opts.dither);

  module.free_result();
  module.doLeakCheck();

  return new ImageData(
    new Uint8ClampedArray(result.view),
    result.width,
    result.height
  );
}
```

This gives us a report like the following in the console:

*Screenshot of a message "Direct leak of 12 bytes" coming from an anonymous wasm-function[...].*

Uh-oh, there are some small leaks, but the stacktrace is not very helpful as all the function names are mangled. Let's recompile with a basic debugging info to preserve them:

```
emcc \
  --bind \
  ${OPTIMIZE} \
  --closure 1 \
  -s ALLOW_MEMORY_GROWTH=1 \
  -s MODULARIZE=1 \
  -s 'EXPORT_NAME="imagequant"' \
  -I node_modules/libimagequant \
  -o ./imagequant.js \
  --std=c++11 \
  imagequant.cpp \
  -fsanitize=address \
```

```
  -g2 \
  node_modules/libimagequant/libimagequant.a
```

This looks much better:

*Screenshot of a message "Direct leak of 12 bytes" coming from a `GenericBindingType<RawImage>::toWireType` function*

Some parts of the stacktrace still look obscure as they point to Emscripten internals, but we can tell that the leak is coming from a `RawImage` conversion to "wire type" (to a JavaScript value) by Embind. Indeed, when we look at the code, we can see that we return `RawImage` C++ instances to JavaScript, but we never free them on either side.

As a reminder, currently there is no garbage collection integration between JavaScript and WebAssembly, although one is being developed. Instead, you have to manually free any memory and call destructors from the JavaScript side once you're done with the object. For Embind specifically, the official docs suggest to call a `.delete()` method on exposed C++ classes:

> JavaScript code must explicitly delete any C++ object handles it has received, or the Emscripten heap will grow indefinitely.
>
> ```
> var x = new Module.MyClass;
> x.method();
> x.delete();
> ```

Indeed, when we do that in JavaScript for our class:

```
// …

const result = opts.zx
  ? module.zx_quantize(data.data, data.width, data.height, opts.dither)
  : module.quantize(data.data, data.width, data.height, opts.maxNumColors, opts.dither);

module.free_result();
result.delete();
module.doLeakCheck();

return new ImageData(
  new Uint8ClampedArray(result.view),
  result.width,
```

```
    result.height
  );
}
```

The leak goes away as expected.

# Discovering more issues with sanitizers

Building other Squoosh codecs with sanitizers reveals both similar as well as some new issues. For example, I've got this error in MozJPEG bindings:

*Screenshot of a message "memory access out of bounds" coming from a `jpeg_start_compress` function*

Here, it's not a leak, but us writing to a memory outside of the allocated boundaries □□

Digging into the code of MozJPEG, we find that the problem here is that `jpeg_mem_dest`—the function that we use to allocate a memory destination for JPEG—reuses existing values of `outbuffer` and `outsize` when they're non-zero:

```
if (*outbuffer == NULL || *outsize == 0) {
  /* Allocate initial buffer */
  dest->newbuffer = *outbuffer = (unsigned char *) malloc(OUTPUT_BUF_SIZE);
  if (dest->newbuffer == NULL)
    ERREXIT1(cinfo, JERR_OUT_OF_MEMORY, 10);
  *outsize = OUTPUT_BUF_SIZE;
}
```

However, we invoke it without initialising either of those variables, which means MozJPEG writes the result into a potentially random memory address that happened to be stored in those variables at the time of the call!

```
uint8_t* output;
unsigned long size;
// …
jpeg_mem_dest(&cinfo, &output, &size);
```

Zero-initialising both variables before the invocation solves this issue, and now the code reaches a memory leak check instead. Luckily, the check passes successfully, indicating that we don't have any leaks in this codec.

# Issues with shared state

…Or do we?

We know that our codec bindings store some of the state as well as results in global static variables, and MozJPEG has some particularly complicated structures.

```
uint8_t* last_result;
struct jpeg_compress_struct cinfo;


val encode(std::string image_in, int image_width, int image_height, MozJpegOptions opts) {
  // …
}
```

What if some of those get lazily initialized on the first run, and then improperly reused on future runs? Then a single call with a sanitizer would not report them as problematic.

Let's try and process the image a couple of times by randomly clicking at different quality levels in the UI. Indeed, now we get the following report:

*Screenshot of a message "Direct leak of 262144 bytes" coming from a `jpeg_finish_compress` function*

262,144 bytes—looks like the whole sample image is leaked from `jpeg_finish_compress`!

After checking out the docs and the official examples, it turns out that `jpeg_finish_compress` doesn't free the memory allocated by our earlier `jpeg_mem_dest` call—it only frees the compression structure, even though that compression structure already knows about our memory destination… Sigh.

We can fix this by freeing the data manually in the `free_result` function:

```
void free_result() {
  /* This is an important step since it will release a good deal of memory. */
  free(last_result);
  jpeg_destroy_compress(&cinfo);
}
```

I could keep hunting those memory bugs one by one, but I think by now it's clear enough that the current approach to memory management leads to some nasty systematic issues.

Some of them can be caught by the sanitizer right away. Others require intricate tricks to be caught. Finally, there are issues like in the beginning of the post that, as we can see from the logs, aren't caught by the sanitizer at all. The reason is that the actual mis-use happens on the JavaScript side, into which the sanitizer has no visibility. Those issues will reveal themselves only in production or after seemingly unrelated changes to the code in the future.

# Building a safe wrapper

Let's take a couple of steps back, and instead fix all of these problems by restructuring the code in a safer way. I'll use ImageQuant wrapper as an example again, but similar refactoring rules apply to all the codecs, as well as other similar codebases.

First of all, let's fix the use-after-free issue from the beginning of the post. For that, we need to clone the data from the WebAssembly-backed view before marking it as free on the JavaScript side:

```
  // …

  const result = /* … */;

  const imgData = new ImageData(
    new Uint8ClampedArray(result.view),
    result.width,
    result.height
  );

  module.free_result();
  result.delete();
  module.doLeakCheck();

  return new ImageData(
    new Uint8ClampedArray(result.view),
    result.width,
    result.height
  );
  return imgData;
}
```

Now, let's make sure that we don't share any state in global variables between invocations. This will both fix some of the issues we've already seen, as well as will make it easier to use our codecs in a multithreaded environment in the future.

To do that, we refactor the C++ wrapper to make sure that each call to the function manages its own data using local variables. Then, we can change the signature of our `free_result` function to accept the pointer back:

```
liq_attr* attr;
liq_image* image;
liq_result* res;
uint8_t* result;

RawImage quantize(std::string rawimage,
                  int image_width,
                  int image_height,
                  int num_colors,
                  float dithering) {
  const uint8_t* image_buffer = (uint8_t*)rawimage.c_str();
  int size = image_width * image_height;

  attr = liq_attr_create();
  image = liq_image_create_rgba(attr, image_buffer, image_width, image_height, 0);
  liq_attr* attr = liq_attr_create();
  liq_image* image = liq_image_create_rgba(attr, image_buffer, image_width, image_height, 0);
  liq_set_max_colors(attr, num_colors);
  liq_result* res = nullptr;
  liq_image_quantize(image, attr, &res);
  liq_set_dithering_level(res, dithering);
  uint8_t* image8bit = (uint8_t*)malloc(size);
  result = (uint8_t*)malloc(size * 4);
  uint8_t* result = (uint8_t*)malloc(size * 4);

  // …
}

void free_result() {
void free_result(uint8_t *result) {
  free(result);
}
```

But, since we're already using Embind in Emscripten to interact with JavaScript, we might as well make the API even safer by hiding C++ memory management details altogether!

For that, let's move the `new Uint8ClampedArray(…)` part from JavaScript to the C++ side with Embind. Then, we can use it to clone the data into the JavaScript memory even *before* returning from the function:

```
class RawImage {
 public:
  val buffer;
  int width;
  int height;

  RawImage(val b, int w, int h) : buffer(b), width(w), height(h) {}
};
thread_local const val Uint8ClampedArray = val::global("Uint8ClampedArray");

RawImage quantize(/* … */) {
val quantize(/* … */) {
  // …
  return {
    val(typed_memory_view(image_width * image_height * 4, result)),
    image_width,
    image_height
  };
  val js_result = Uint8ClampedArray.new_(typed_memory_view(
    image_width * image_height * 4,
    result
  ));
  free(result);
  return js_result;
}
```

In this case we can return `Uint8ClampedArray`, because JavaScript already knows the width and height of the resulting image. If this wasn't the case, then we could return an `ImageData` instance instead, which is functionally equivalent to our previous `RawImage` wrapper, but is a standard JavaScript-owned object:

```
  // …
  val js_result = Uint8ClampedArray.new_(typed_memory_view(
    image_width * image_height * 4,
    result
  ));
  free(result);
```

```
  return ImageData.new_(js_result, image_width, image_height);
}
```

Note how, with a single change, we both ensure that the resulting byte array is owned by JavaScript and not backed by WebAssembly memory, ***and*** get rid of the previously leaked `RawImage` wrapper too.

Now JavaScript doesn't have to worry about freeing data at all anymore, and can use the result like any other garbage-collected object:

```
  // …

  const result = /* … */;

  const imgData = new ImageData(
    new Uint8ClampedArray(result.view),
    result.width,
    result.height
  );

  module.free_result();
  result.delete();
  // module.doLeakCheck();

  return imgData;
  return new ImageData(result, result.width, result.height);
}
```

This also means we no longer need a custom `free_result` binding on the C++ side:

```
void free_result(uint8_t* result) {
  free(result);
}

EMSCRIPTEN_BINDINGS(my_module) {
  class_<RawImage>("RawImage")
      .property("buffer", &RawImage::buffer)
      .property("width", &RawImage::width)
      .property("height", &RawImage::height);

  function("quantize", &quantize);
```

```
  function("zx_quantize", &zx_quantize);
  function("version", &version);
  function("free_result", &free_result, allow_raw_pointers());
}
```

All in all, our wrapper code became both cleaner and safer at the same time.

After this I went through some further minor improvements to the code of the ImageQuant wrapper and replicated similar memory management fixes for other codecs. If you're interested in more details, you can see the resulting PR here: Memory fixes for C++ codecs.

# Takeaways

What lessons can we learn and share from this refactoring that could be applied to other codebases?

- Don't use memory views backed by WebAssembly—no matter which language it's built from—beyond a single invocation. You can't rely on them surviving any longer than that, and you won't be able to catch these bugs by conventional means, so if you need to store the data for later, copy it to the JavaScript side and store it there.
- If possible, use a safe memory management language or, at least, safe type wrappers, instead of operating on raw pointers directly. This won't save you from bugs on the JavaScript ↔ WebAssembly boundary, but at least it will reduce the surface for bugs self-contained by the static language code.
- No matter which language you use, run code with sanitizers during development—they can help to catch not only problems in the static language code, but also some issues across the JavaScript ↔ WebAssembly boundary, such as forgetting to call `.delete()` or passing in invalid pointers from the JavaScript side.
- If possible, avoid exposing unmanaged data and objects from WebAssembly to JavaScript altogether. JavaScript is a garbage-collected language, and manual memory management is not common in it. This can be considered an abstraction leak of the memory model of the language your WebAssembly was built from, and incorrect management is easy to overlook in a JavaScript codebase.
- This might be obvious, but, like in any other codebase, avoid storing mutable state in global variables. You don't want to debug issues with its reuse across various invocations or even threads, so it's best to keep it as self-contained as possible.

---

□□□ #3
□ □□□ □□□ 2026-10-31 22:27:41 UTC
□ cocobook-sync □□□ 2026-11-04 17:38:27 UTC