Nylas

Back to Nylas Blogs

The Nylas Engineering Blog

Low Level Electron Debugging

Looking under the hood

By: Tomasz Finc

March 23, 2017

We're big fans of Electron here at Nylas. It's allowed us to iterate quickly across platforms using the best of modern web standards. Since Electron is built on top of Chromium we get some great debugging tooling from Chrome Developer Tools. Usually these are enough for our purposes, but when bugs get weird, we need to go one level down, and look under the hood.

This post will focus on a recent bug with SQLite and how we utilized LLDB to find the root cause of Nylas Mail mysteriously and intermittently crashing.

Setting up Electron

First you need Electron without symbols stripped. If you run `lldb` on the prebuilt downloaded Electron, you'll get hex garbage in your stacks.

Start by building Electron from source. This isn't as bad as it sounds! Follow the instructions for Mac, Win, or Linux.

On Mac it's as simple as: `script/bootstrap.py -v && script/build.py -c D` . Just make sure you're on the latest OSX and have the latest XCode properly installed.

Once you have Electron built, you can forevermore use your new debug executable to launch your app instead of a precompiled one.

Setting up native Node modules

If you have plain javascript dependencies, you can debug them as normal through the inspector panel. However, if you're using native modules, sometimes the issue can be deep inside the compiled code of that module.

In our case, we had a strong hunch that the source of our bug was in SQLite. Code executing here doesn't show up in the inspector console (except as an unhelpful grey bar). Even with our debug version of Electron, we need to make sure that our native node module also has symbols. By default when you `npm install` native node modules, they'll strip symbols making low-level debugging almost impossible.

You need to rebuild native modules with debug flags.

Here's how we did it for SQLite:

```
$ cd node_modules/sqlite3
$ [vi|emacs|nano] package.json

And change
"install": "node-pre-gyp install --fallback-to-build"
to
"install": "node-pre-gyp install --debug --fallback-to-build"

$ NPM_CONFIG_TARGET=1.4.15 NPM_CONFIG_ARCH_x64=NPM_CONFIG_TARGE_ARCH=x64 NPM_CONFIG_DI
```

All those environment variables we set before the `npm install` are necessary to make sure we're using Electron's headers. Please read up about Using Native Node Modules if that's foreign to you.

Now we're ready to launch the app & have all debug symbols available!

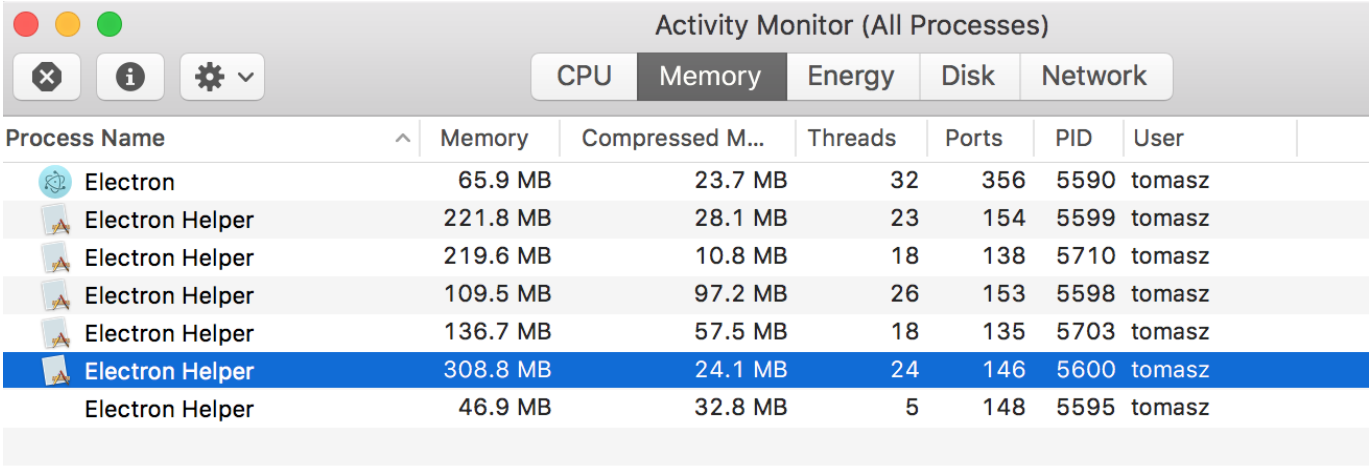
Start Debugging (Now with more symbols)

Launch your app with the debug version of Electron we previously built:

```
$ electron/out/D/Electron.app/Contents/MacOS/Electron myElectronAppFolder
```

Now let's attach `lldb` or `gdb` to your app!

The first trick is finding the correct process ID to attach to. Your app will likely have two or more processes. In our case, we knew our bug was coming from a particular process because it would blow the memory sky high.



Process Name	Memory	Compressed M...	Threads	Ports	PID	User
Electron	65.9 MB	23.7 MB	32	356	5590	tomasz
Electron Helper	221.8 MB	28.1 MB	23	154	5599	tomasz
Electron Helper	219.6 MB	10.8 MB	18	138	5710	tomasz
Electron Helper	109.5 MB	97.2 MB	26	153	5598	tomasz
Electron Helper	136.7 MB	57.5 MB	18	135	5703	tomasz
Electron Helper	308.8 MB	24.1 MB	24	146	5600	tomasz
Electron Helper	46.9 MB	32.8 MB	5	148	5595	tomasz

Now start lldb:

```
$ lldb -p 5600
```

Once attached, it'll bring your process to a screeching halt. At this point you can look at the backtrace, explore stack frames, and much more. Read the full LLDB documentation to find out everything you can do.

Catching Trouble

The trick was to have `lldb` stop at just the right time when our bug happened. For intermittent bugs this is frequently difficult. While you can use a combination of chrome inspector breakpoints and lldb breakpoints, our bug had an (un)fortunate property of causing the whole app to mysteriously crash when it reared its head. When this happened, we attached lldb.

Now that we're in lldb, attached to the right process, and stopped in the middle of our mysterious crash, let's look around.

```
$ (lldb) thread backtrace all
```

Since we rebuilt SQLite with debug flags, we now get obviously sqlite-related stacktraces in some of our threads and frames. Let's dive into those further:

```
(lldb) thread backtrace
* thread #1: tid = 0xbc0d2e,
...
frame #11: 0x0000000116334d25 node_sqlite3.node`Nan::imp::Factory<v8::String>::New(val
    frame #12: 0x0000000116348a31 node_sqlite3.node`Nan::imp::Factory<v8::String>::ret
...
```

Next we pick the thread and frame that has sqlite in it:

```
$ (lldb) thread select 1
..
$ (lldb) thread backtrace
..
```

```
$ (lldb) frame select 14
frame #14: 0x00000001163460df node_sqlite3.node`node_sqlite3::Statement::Work_AfterAll
   548                         Rows::const_iterator it = baton->rows.begin();
   549                         Rows::const_iterator end = baton->rows.end();
   550                         for (int i = 0; it < end; ++it, i++) {
-> 551                             Nan::Set(result, i, RowToJS(*it));
   552                             delete *it;
   553                         }
   554
$ (lldb)  l
   555                         Local<Value> argv[] = { Nan::Null(), result };
   556                         TRY_CATCH_CALL(stmt->handle(), cb, 2, argv);
   557                     }
   558                     else {
   559                         // There were no result rows.
   560                         Local<Value> argv[] = {
   561                             Nan::Null(),
```

We zero in on: `node_sqlite3::Statement::Work_AfterAll` . By taking a quick look through the sqlite source code, that function stood out as one that likely has frame variables that can tell us what we want.

Finally, we use the fact that lldb is fully interactive and use existing sqlite functions to print out the value of suspicious variables. In our case we wanted to know what query was running when the app hung.

```
$ (lldb) print sqlite3_sql(stmt->_handle)
(const char *) $0 = 0x00007fcf95498720 "SELECT * FROM `messages`;"
```

AH HA! That'll do it… Selecting several GB of message data at once will hang sqlite and crash the app when it runs out of memory. Some piece of our code unexpectedly, and intermittently, queried sqlite with invalid limits.

Final Thoughts

LLDB, chrome developer tools, and sound development practices are all parts of our toolkit. Each one serves a slightly different purpose and we're always learning something new about how to improve our debugging skills and the improving the

quality of Nylas Mail. Going forward we have a couple of ideas on where to take this next.

How to connect Xcode Instruments

Instruments is a very powerful debugging tool. It's possible to connect this same electron stack and get timelines of memory allocation, disk access, and much more.

Could we build an Electron debugger?

All of this setup could be automated and generalized for any Electron app. Building an Electron debugger wouldn't be too far off. Imagine having a tool like Instruments but specifically built for Electron Apps. We think that would be amazing.

Thanks to Mark Hahnenberg for contributing the lldb expertise for this work!