I've recently been working on JavaScript Obfuscation. I've read as much as I can from the internet about options and capabilities. It is clear there is one winner out of all the offerings available.

JScrambler Review

Nicholas Starke | https://twitter.com/nstarke | https://github.com/nstarke

JScrambler (https://jscrambler.com/) is a paid product featuring JavaScript Obfuscation capabilities. When it comes to obfuscating JavaScript, it is the gold standard.

This is what the internet proclaimed as I read it [1]. However, it was truly difficult to assess how accurate these claims are; essentially the only public obfuscation examples they provide are:

https://jscrambler.com/products/code-integrity/javascript-obfuscation

A single, ten line example. This was not enough for me to make any decisions or really come up with any valuable insight with.

I needed more. So I thought to myself, **Where can I find more???**

Turns out like any good technology company, **JScrambler** dogfoods their own product. So I present to you more fully-featured examples of their obfuscation prowess:

- https://jscrambler.com/bundle.js [2] - This is the primary marketing site JavaScript - run thru their own obfuscation technology, presumably.
- https://jscrambler.com/vendor.bundle.js [3] - This is the vendor dependencies that the client side application relies on to do its thing.

Now as of this writing the first is 81289 lines of code *prettified* and the second is 66944 *prettified*.

That is some serious obfuscation! Now I can begin more nuanced evaluation of their obfuscation technology (and so can you!), but if you would like more examples they have bountiful subdomains hosting obfuscated code!

- https://app.jscrambler.com/bundle.5.6.11.js [4] - 435995 lines *prettified*.

First Things First

Let's get our source material in more *readable* form. I used https://beautifier.io/ and it worked like a champ in *prettifying* the source material. This only makes it slight more readable, but anything other than a single line of JS is preferable to the alternative.

I will note that Chrome struggled a bit trying to export the prettified source code. I used **DevTools** to *prettify* the source right in the browser, but when I clicked `Save As` I was greeted with an unresponsive browser.

Right off the bat I notice a few things:

- The control flow flattening is amazing
- There is no string obfuscation / encryption.

This second point is more important than one might imagine. Just like in binary reverse engineering, strings serve as a sort of guide through an application, and having those symbols readily available makes reverse engineering much easier.

Also if one has somewhat sensitive data in a client side application, the lack of string protection is a big deal, as i was able to find the jscrambler gitlab server address and application path easily despite the obfuscation. Ditto for the host name of the jscrambler npm mirror.

Now lets talk about the first point. Control flow flattening hides the business logic of an application, and jscrambler does a great job here. Static analysis isn't going to help here, we will need a full debugger. All logic here is written as `switch` statements, which effectively masks the underlying, *unobfuscated* logic. In addition, jscrambler adds `case` statements that do not correspond to anything in the original code, a feat known as *dead code injection*.

Using a technique I developed previously [5] I exract all of the strings in the `bundle.6.5.11-beautified.js` file.

Let's take a look at some of the things that pop out:

- `"SECRET_DO_NOT_PASS_THIS_OR_YOU_WILL_BE_FIRED"`

Well that certainly looks interesting, you should check it out if you are interested.

I Believe

Now is as good of a time as any to say that I believe in obfuscation as a strategy and deterent against reverse engineering, as well as a means for bypassing antivirus (we won't go into that today though). More importantly, bad guys certainly believe in obfuscation because they use it all the time to try to deter reverse engineering by white hat malware analysts who catalog these things legitimately to protect businesses, individuals, and governments.

It is important to recognize the limitations though of this technique. Secrets, such as cryptographic keys or passphrases, will never be safe on the client and obfuscation should not be used to try to make them so. Any mission critical proprietary software should stay on the server.

I say all of this because there are a lot of people [6] [7] [8] [9] who do not think JavaScript obfuscation has any valid purpose.

I should also probably note that I have never signed up for a Jscrambler account and am thus not bound by their terms of service in any way. You can read the terms of service here [10], but I'm not seeing anything related to reverse engineering in there. I think this is a net positive and demonstrates that the company stands by its product and its capabilities. However I am not a lawyer and if you have any questions on their terms of service you should consult a lawyer and **not me**.

Along those lines its also worth mentioning that I have no relationship with Jscrambler, any of its executives, investors, or employees, and am not being paid by **anyone** for this analysis.

Back to regularly obfuscated programming

Now that that is out of the way, let's continue our journey. Harking back to the unencrypted / obfuscated strings issue, we can compile a list of api endpoints by some string analysis on the app. I won't document

them exhaustively here, but if you are following along, you can search your *prettified* code for strings like

- `api`
- `application`

It is even possible to discern what HTTP method is used for each route.

Other things to note is that browser defined APIs - such as everything that hangs off the `window` object, are not obfuscated either. There are definitely ways of doing this and I was disappointed to see that certain object functions and properties were unobfuscated throughout the obfuscated files.

Summary

It is clear to me that the magic that transforms the input code all occurs on the server side and nothing of that nature is included in the obfuscated client-side files available currently. Overall I would say any given portion of the application is essentially non-understandable by simply reading the code (without debugging through it). However, it is possible to get an overall idea of what the application does, maybe not in complete detail, because strings are not obfuscated or encrypted in any way. I do think that obfuscated software deters casual reverse engineering, but whether it would deter let's say a nation-state actor with nation-state resources, the jury is out on that one.

Sources:

[1] https://www.huffpost.com/entry/the-integrity-of-your-jav_b_10218176

[2] https://web.archive.org/web/20190427002248/https://jscrambler.com/bundle.js

[3] https://web.archive.org/web/20190427001957/https://jscrambler.com/vendor.bundle.js

[4] https://web.archive.org/web/20190505170802/https://app.jscrambler.com/bundle.5.6.11.js

[5] https://gist.github.com/nstarke/4f0eba4d1765b9d48fe884301cd5aedf

[6] https://www.reddit.com/r/javascript/comments/3b9zg6/how_to_secure_js_source_for_packaged_apps/

[7] https://www.reddit.com/r/javascript/comments/5v5qv1/looking_for_a_js_obfuscator_which_renames/

[8] https://www.reddit.com/r/javascript/comments/ug50h/horriblejs_a_javascript_code_obfuscator_that/

[9] https://www.reddit.com/r/javascript/comments/88pavs/is_javascript_obfuscation_useless/

[10] https://docs.jscrambler.com/privacy-and-security/terms-of-service