Back to Blog

February 18, 2022

JavaScript Obfuscation: The Definitive Guide (2022)

By Jscrambler | 18 min read

This blog post aims to provide the most detailed and comprehensive guide to JavaScript obfuscation in 2022.

Familiarity with JavaScript and npm is a plus, but not necessary to follow this guide.

Let's get into it!

Chapter 1: What is Obfuscation of Code?

To put it simply, obfuscation of code is a technique used to transform plain, easy-to-read code into a new version that is deliberately hard to understand and reverse-engineer—both for humans and machines.

Think of obfuscation like this: you call a friend to schedule a coffee for later (remember when that was a thing?).

A possible reply would be something like *“Hi! Sorry, can’t do it today, I have to watch the kids. Same time tomorrow?”*.

But let’s imagine that your friend decided to obfuscate this a bit, hitting you with a hearty *“Good morrow. I offer thee the sincerest of apologies but, alas, I can't doth t the present day. Haply tom'rrow, equal timeth? Has't to taketh careth of mine own children, I do. Sinc're apologies I offer thee. Fare thee well.”*

Well, that was a mouthful. If you take a closer look at your friend’s Shakespearean reply, it’s clear that the whole thing is unnecessarily complicated. It takes a lot longer to decipher the meaning of the message and there are some redundancies. Plus, your friend added some irrelevant details. Sure, you can bear to decipher this nonsense once. But will you keep calling your friend if this becomes a permanent thing?

As silly of an example as this may seem, it includes the same reasoning as some techniques used in code obfuscation. In the next chapter, we’ll see real examples of obfuscation of code and you’ll hopefully see the resemblance.

While (thankfully) there aren’t many real-life examples of obfuscation in human conversation, obfuscation of code has been around for a long time—there are references to “code obfuscation” in books dating back to 1972.

Obfuscation has been used in several different programming languages, notably in C/C++ (there’s even a competition for obfuscating C code) and Perl. But there’s a language where obfuscation has gained tremendous popularity among developers and business owners alike: JavaScript.

Chapter 2: JavaScript Obfuscation

Why Obfuscate JavaScript Code?

JavaScript has quickly grown into becoming the language of the web. It powers nearly every website in existence and the rise of cross-platform JavaScript frameworks like React Native and Ionic allows developers to create mobile and desktop apps using a shared JS codebase.

97%

Modern websites using JavaScript.

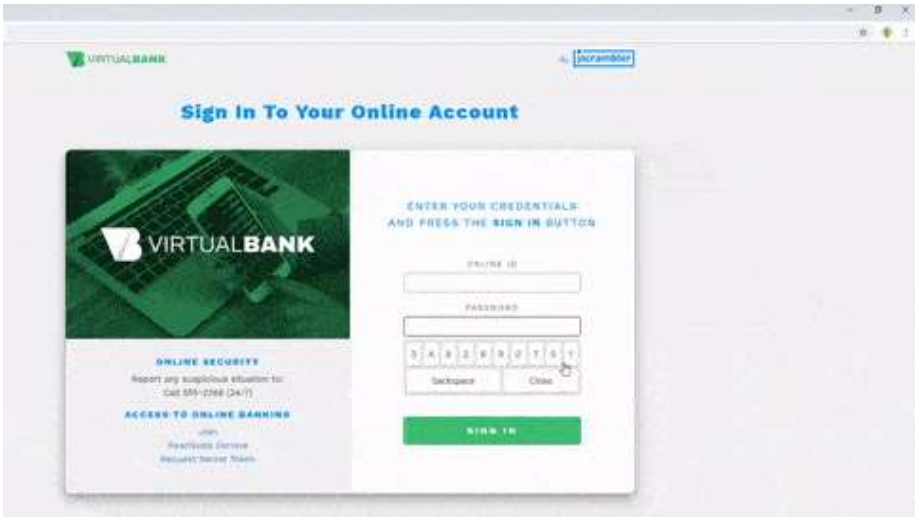
100%

Fortune 500 companies using JavaScript.

With every single Fortune 500 company using JavaScript to develop their apps, today we see JS powering critical applications in various fields like mobile banking, e-commerce, and streaming services.

This brings us to the main question of "why obfuscate JavaScript code?". JavaScript is an interpreted language—so, client-side JavaScript requires an interpreter in the browser to read it, interpret it, and run it. This also means that anyone can use a browser debugger to easily go through the JS code and read or modify it at will.

In the example below, you can see someone easily accessing the code logic behind a virtual keyboard where a bank's clients type their password.



With such easy access to client-side JavaScript code, it's almost effortless for an attacker to take advantage of this security weakness and target any unprotected code.

All of this should seem trivial when we're talking about a simple web application. But companies—notably, the enterprise and Fortune 500—are frequently **storing important business logic on the client-side** of their apps.

If you understand the basics of application security, you know that code secrets should always be kept on trusted execution environments like the backend server. But this is one of those cases where practice takes precedence over theory. When companies store this important

logic on the client-side, they typically do it because they can't feasibly keep this code on the server-side.

A common reason for this is when **there's not a backend** in the first place, as in the case of some mobile applications. Another example is when there's some code that's related to the user experience (like an analytics algorithm) that must run on the client-side. Still, the most common reason is **performance**. Server calls take time and when you have a service where performance is crucial—like a streaming platform or an HTML5 game—storing all the JavaScript on the server is not an option.

Whatever the case, companies usually don't want to expose their proprietary logic. And they definitely never want to expose code secrets. Especially when their competitors can reverse-engineer the code and copy proprietary algorithms.

Besides intellectual property theft, client-side JavaScript can also be targeted in more sophisticated attacks such as automated abuse, piracy, cheating, and data exfiltration (learn more about these here).

It's no wonder that information security standards like ISO 27001 make statements such as:

"Program source code can be vulnerable to attack if not adequately protected and can provide an attacker with a good means to compromise systems in an often covert manner. If the source code is central to the business success its loss can also destroy the business value quickly too."

And OWASP (Open Web Application Security Project) clearly reinforces this recommendation in their Mobile Top 10 Security Risks guide:

"In order to prevent effective reverse engineering, you must use an obfuscation tool."

What is JavaScript Obfuscation?

JavaScript obfuscation is a series of code transformations that turn plain, easy-to-read JS code into a modified version that is extremely hard to understand and reverse-engineer.

Unlike encryption, where you must supply a password used for decryption, **there's no decryption key in JavaScript obfuscation**. In fact, if you encrypt JavaScript on the client-side, that would be a pointless effort—if we had a decryption key we needed to supply to the browser, that key could become compromised and the code could be easily accessed.

So, with obfuscation, the browser can access, read and interpret the obfuscated JavaScript code just as easily as the original, un-obfuscated code. And even though the obfuscated code looks completely different, **it will generate precisely the same output in the browser**.

JavaScript obfuscation is often confused with other techniques like **minification**, **optimization**, and **compression**. Let's quickly look at the differences between them.

Code minifiers remove unnecessary characters in the code (whitespaces, newlines, smaller identifiers, etc.) minimizing the size of the code—but they don't protect the source code.

Code optimizers are mostly used to improve code performance (speed and memory use of the app). Sometimes they can incidentally also make the code harder to read, but this provides no protection (as we'll see later on).

Finally, **code compressors** and packers reduce the code size using encoding and packing techniques but they also don't protect the source code.

Another common misconception is that if you already use SAST or DAST to find vulnerabilities in your JavaScript code and to fix them, this solves all your code problems. While SAST/DAST is useful to fix vulnerabilities, **it doesn't prevent code tampering and reverse engineering**, as vulnerabilities are not required in order to do that. As such, it's advisable to use SAST and DAST **alongside** JavaScript source code protection.

JavaScript Obfuscation Techniques & Targets

Now that it's clear what JS obfuscation is on a broader scope, let's get a bit more technical and look at what it specifically does to the source code.

Since the main objective of obfuscation is to hide JavaScript and parts of the code that could be targeted by attackers or competitors, it's easy to understand that you would want to obfuscate any **data** in the code. So, by concealing things like **variables**, **objects**, and **strings**, you will make it harder for anyone to understand what type of data lies within the code.

Sidenote: relying on obfuscation **alone** to protect sensitive data in your code is a bad practice and the reason why you will probably hear someone say "obscurity isn't security". Depending on your use case, you should always use obfuscation **in addition** to good security practices. Think of it like this: if you wanted to keep a pile of cash secure, you'd probably put it in a safe. But instead of leaving that safe completely exposed on your front porch, you'd probably also hide it somewhere to minimize the likelihood of someone finding it and trying to break it open.

But concealing data is just one of several dimensions of JS obfuscation. Strong obfuscation will also obfuscate the **layout** and **program control flow**, as well as include several **optimization** techniques. Typically, it will target:

- Identifiers;
- Booleans;
- Functions;
- Numbers;
- Predicates;
- Regular expressions;
- Statements;
- Program control flow.

The most common JavaScript obfuscation techniques are **reordering**, **encoding**, **splitting**, **renaming**, and **logic concealing** techniques. Understanding each technique in-depth is out of the scope of this guide, but their names are already pretty self-explanatory. If you'd like to learn more about each possible technique, check this documentation.

However, Control-flow obfuscation is worthy of a deeper explanation, as it is an especially useful technique. It makes the program flow significantly harder to follow by **removing the natural conditional constructs** that make the code easier to read.

From a technical perspective, it splits all the source code's basic blocks — such as function body, loops, and conditional branches — and puts them all inside a single infinite loop with a switch statement that controls the program flow.

It can also include clones (semantically equivalent copies of basic blocks that can be executed interchangeably with their original basic blocks), dead clones (dummy copies of basic blocks that are never executed, but mimic and can be confused with the code that will be executed), and opaque steps (which obfuscate the switching variable, making it harder to understand what's the next switch case that'll be executed). The combination of these techniques adds up to the overall complexity of the obfuscated code.

Another noteworthy approach to obfuscation is the use of polymorphism. Polymorphic JavaScript obfuscation is a unique technique used by Jscrambler that ensures that every new code obfuscation results in completely different code.

Let's look at an example. Imagine you are deploying obfuscated code builds once per week. Attackers may start trying to de-obfuscate the code as soon as you ship a new version. Assuming they have made some progress before you release a new build, if the obfuscated code of the new build is similar to the previous one, attackers can leverage most of their progress to continue their reverse engineering. With polymorphic obfuscation, **the new build is completely different**, which means that most (if not all) of the previous de-obfuscation progress becomes useless.

JavaScript Obfuscation Example

Ok, time to put the theory on hold for now, and let's jump right into an actual JavaScript obfuscation example.

Let's consider the code snippet below, which is an algorithm that is used to recommend products to the shoppers of an e-commerce website. It generates a list of product recommendations for a given customer based on that customer's history of previous purchases.

```
function getRecommendations(products, numRecommendations) {
  const weights = [];
  for (let i = 0; i < products.length; i++) {
    const product = products[i];
    const weight = getWeight(product);
    weights.push({_id: product.id, weight});
  }

  weights.sort((recommendation1, recommendation2) =>
    recommendation2.weight - recommendation1.weight
  );

  return weights.slice(0, numRecommendations);
}
```

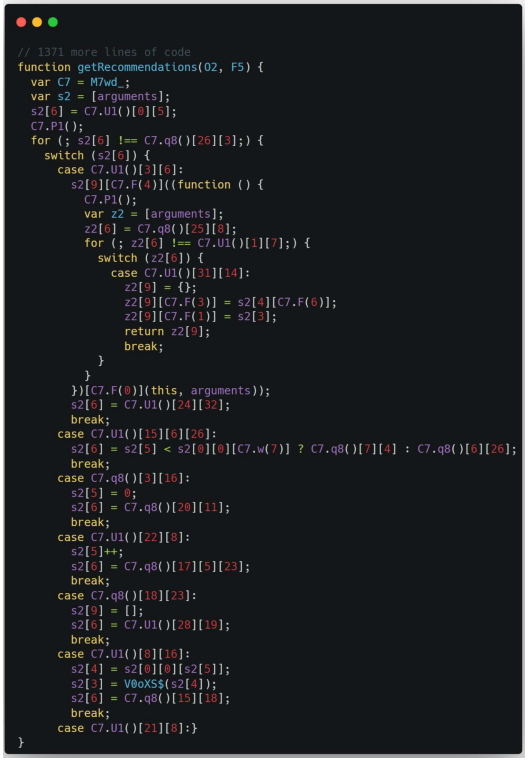
This seems like pretty ordinary code, but let's imagine that this is a proprietary algorithm developed by this company. If we were a competitor visiting their website, we could quickly find this code and do what we wanted with it.

As the owners of this code, we understand this risk and want to protect it. Before we get into actual JS obfuscation, let's see what minification would do to the code.

```
function getRecomendations(products,purchasedProducts,numRecommendations){var weights=
[];for(var i=0;i < products.length;i++){var product=products[i];var
weight=getWeight(product);weights.push({id:product.id,weight});}weights.sort((recommendation
1,recommendation2)=>recommendation2.weight - recommendation1.weight);return
weights.slice(0,Math.min(numRecommendations,weights.length));}
```

At first glance, you'd say it's harder to read the code. But it just takes a second to realize that all our functions, objects, and variables are there in plain sight. Again, minification doesn't offer any sort of code protection.

Let's now see what the code looks like after we add a single obfuscation technique.



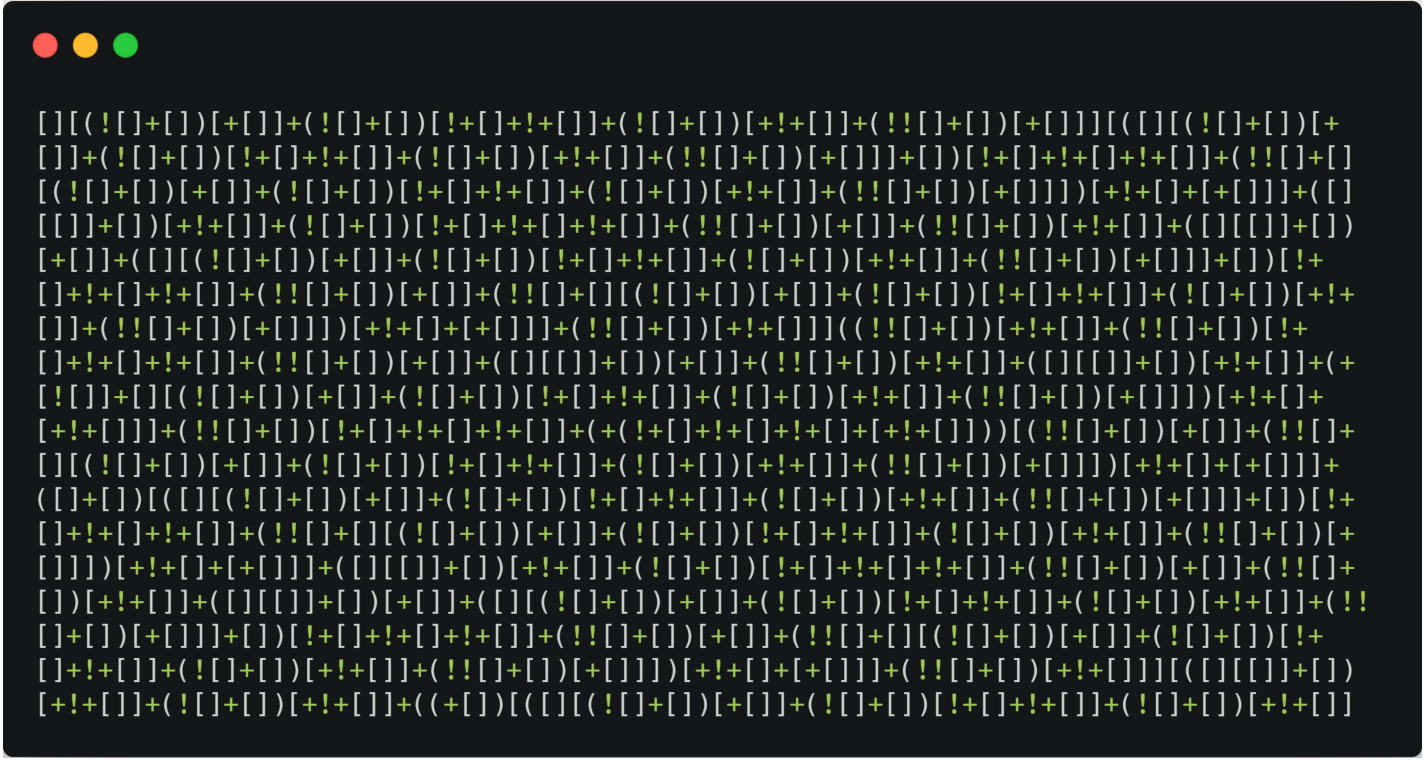
```
// 1371 more lines of code
function getRecommendations(O2, F5) {
  var C7 = M7wd_;
  var s2 = [arguments];
  s2[6] = C7.U1()[8][5];
  C7.P1();
  for (; s2[6] !== C7.q8()[26][3];) {
    switch (s2[6]) {
      case C7.U1()[3][6]:
        s2[9][C7.F(4)]((function () {
          C7.P1();
          var z2 = [arguments];
          z2[6] = C7.q8()[25][8];
          for (; z2[6] !== C7.U1()[1][7];) {
            switch (z2[6]) {
              case C7.U1()[31][14]:
                z2[9] = {};
                z2[9][C7.F(3)] = s2[4][C7.F(6)];
                z2[9][C7.F(1)] = s2[3];
                return z2[9];
                break;
            }
          }
        })[C7.F(0)](this, arguments));
        s2[6] = C7.U1()[24][32];
        break;
      case C7.U1()[15][6][26]:
        s2[6] = s2[5] < s2[0][0][C7.w(7)] ? C7.q8()[7][4] : C7.q8()[6][26];
        break;
      case C7.q8()[3][16]:
        s2[5] = 0;
        s2[6] = C7.q8()[20][11];
        break;
      case C7.U1()[22][8]:
        s2[5]++;
        s2[6] = C7.q8()[17][5][23];
        break;
      case C7.q8()[18][23]:
        s2[9] = [];
        s2[6] = C7.U1()[28][19];
        break;
      case C7.U1()[8][16]:
        s2[4] = s2[0][0][s2[5]];
        s2[3] = V0oXS$(s2[4]);
        s2[6] = C7.q8()[15][18];
        break;
      case C7.U1()[21][8]:}
}
```

First off, this doesn't even seem like recognizable JavaScript code. It has been obfuscated with something called control-flow flattening—a unique Jscrambler transformation that flattens the program flow and conceals every single natural conditional construct that would otherwise make the code easier to read.

Sidenote: if you want to test this obfuscation transformation in your own code to see what the output looks like, you just need to create a free Jscrambler account.

The snippet above just shows the first few lines of code but the whole thing is almost **700 lines long**. And if we run this code, the browser will run it just like the original thing.

Now let's look at an example of extreme obfuscation:



This is a piece of code that has **non-alphanumeric obfuscation**, which you don't often find in the wild. To the human eye, this seems impossible to reverse-engineer. But if we run this code through an automated reverse-engineering tool, we would get the original code almost immediately.

This seemingly extreme obfuscation is, in fact, a great example of what **weak obfuscation** can look like.

So, how can we distinguish between weak and strong obfuscation? To answer that, we need to understand the **obfuscation metrics**.

JS Obfuscation Metrics

One of the clearest interpretations of JS obfuscation metrics is provided by Collberg *et al.* in their paper "A Taxonomy of Obfuscating Transformations".

As these researchers put it, there are 3 key metrics: **potency**, **resilience**, and **cost**.

Potency

Potency is a metric that answers the question "To what degree is a human reader confused?". Looking back at our 3 previous examples, we can confidently say that example #1 (minification) has low potency, while example #2 has high potency and example #3 has extremely high potency.

You might be wondering: how do I calculate the potency metric? Well, potency is typically measured using Software Complexity Metrics such as Halstead's Metrics. So, you typically won't be calculating potency yourself.

That being said, there are some specific characteristics of the transformation that you can use to more easily evaluate its potency. So, a high potency transformation typically:

- hides constants and names;
- makes it difficult to understand the order in which the code is executed;
- makes it difficult to understand what the relevant code is;
- increases overall program size and introduces new classes and methods;
- introduces new predicates and rewrites the conditional and looping constructs;
- increases long-range variable dependencies.

However, one key mistake when evaluating obfuscated JavaScript code is only considering its potency. And as we saw before, a high potency transformation can be very easy to defeat. That's why we must also consider another metric: resilience.

Resilience

The resilience metric answers the question "How well are automatic deobfuscation attacks resisted?".

For example, we can add an if statement that introduces a dummy variable into our code. It may take a while for a human to identify the code as dummy code, but a deobfuscator would immediately remove the statement.

This is why resilience is calculated by considering two different aspects:

- the amount of time required to develop a deobfuscator capable of reverting a transformation's result;
- the required execution time and space by a deobfuscator to effectively revert the transformation.

This is the metric where most obfuscation tools fail, especially free JS obfuscators. They may output what looks like highly obfuscated code, but it's typically quite simple to de-obfuscate it using readily available tools. **When comparing different obfuscation results, we can't simply trust our own eyes and perception.**

Jscrambler's transformations, however, are built to achieve maximum resilience whenever possible. Specifically, Jscrambler includes a Code Hardening feature that is built into every code obfuscation. This feature provides the code with guaranteed up-to-date resilience against all automated reverse engineering tools and techniques. So, when these tools attempt to reverse code protected by Jscrambler, they will typically time out or hang, forcing attackers to go manual and face the dreaded high-potency transformations by hand.

Cost

Finally, we have the cost metric, which represents the impact of a transformation in the execution time of a transformed application as well as the impact on the application's file size.

This is important because you wouldn't want your application performance to be ruined due to obfuscation, especially when you have a client-facing app and could be losing money if the app starts running slowly.

A good obfuscation tool should always provide specific features to minimize performance hits, and also allow you to fine-tune the transformations throughout your code. This is yet another shortcoming of free JS obfuscators, which typically provide little to no capabilities to fine-tune the protection.

In contrast, using Jscrambler you'll find several features that automatically fine-tune the protection to maximize performance, such as Profiling and App Classification.

Understanding these three obfuscation metrics is crucial to ensure that your code is actually protected and doesn't just look like so.

Chapter 3: Obfuscation & The SDLC

JavaScript obfuscation shouldn't result in process overhead and over-complicate your SDLC. To make sure that doesn't happen, it's critical to address two dimensions: compatibility and integrations.

Compatibility of Obfuscated JavaScript Code

When it comes to compatibility, first there's the matter of understanding if your own source code is compatible with a specific obfuscation tool. Some JS obfuscators lack compatibility with some ECMAScript versions and may require you to transpile the code as an extra step before

protecting it. More frequently, they may lack compatibility with certain JS libraries and frameworks, requiring substantial changes to enable code protection.

Another important aspect is the compatibility of the obfuscated code. Going back to our original definition of JavaScript obfuscation, it is "used to **transform** (...) code". While your obfuscated code should always run just like the original code, obfuscation can result in some compatibility changes, namely with specific browser versions.

As an enterprise product, Jscrambler ensures compatibility with all ECMAScript versions and provides features like Browser Compatibility to give visibility and control over the compatibility of the protected code. So, you can always ensure that the protected code will be compatible with the target browser versions. Plus, it ensures compatibility with all the main JS libraries and frameworks.

Obfuscation, CI/CD Integration, and Making Engineers Happy

In case you want to ensure that all your app deployments are obfuscated, you will likely want to automate this process. Here, it's especially important to consider what JavaScript frameworks you're using and how your build process is structured.

As mentioned before, several obfuscators offer very limited compatibility with JavaScript frameworks, especially with React Native and Ionic. So, they will typically fail to obfuscate the code altogether.

In the case of Jscrambler, the obfuscation process is done at build-time and is fully compatible with every main JavaScript framework. Jscrambler can be easily integrated into the build process of React, Angular, Vue, Node.js, React Native, Ionic, NativeScript, and many other frameworks. Integrating Jscrambler into your CI/CD pipeline is simple and there's even integrations for specific build processes: you just need to call the Jscrambler API and get a protected version of your application. This protected version is the one you should deploy.

A smooth CI/CD integration will certainly put a smile on your engineer's faces, but there's still another "quality of life" feature that is especially relevant when it comes to obfuscation. After seeing the previous examples of obfuscated code, you might have wondered "how do I debug this protected code?". Since the goal of obfuscation is to make it harder to go through the

code, it may make the lives of your developers a living nightmare when they have to debug a bug in production. Hence the importance of **source maps**.

While many obfuscation tools do not provide comprehensive source maps, Jscrambler Source Maps enable easily mapping the obfuscated code back to its original source code—both through the web app and through the Jscrambler CLI.

Support and Trust

As with all things related to security, obfuscation is a high-stakes process. Just like using a weak JS obfuscator can provide a wrong and dangerous sense of security, misconfiguring any obfuscation tool can result in serious problems that jeopardize the overall security and usability of the application.

So, if you're not a JS obfuscation expert, how can you navigate this configuration and avoid any pitfalls?

To prevent being blindsided by poor configuration, make sure that you're using an obfuscation tool that provides comprehensive documentation along with priority support. Every app is different, and obfuscation is surely not a one-size-fits-all solution. By counting on a dedicated support team, you can more easily fine-tune the obfuscation to match your specific use case and avoid common pitfalls that can degrade the usability of your app.

There's no way around it: security is trust. Just like you wouldn't give your source code (especially if it contains sensitive information) to any random person, you likely won't want to blindly trust any JS obfuscator with it. A particular thing about obfuscation is that it's seriously difficult to vet the end-result. There have been some cases where free obfuscators added malware/spyware to the source code before obfuscating it. It's extremely important to exercise due diligence on the tool you'll be using to avoid any unpleasant surprises. Look for market recognition, client testimonials, and the maturity of the company/technology.

Chapter 4: Beyond Obfuscation, JavaScript Protection

Usually, most guides on JavaScript obfuscation would end right about here. But this bonus chapter is a must-read because it will explain why JS obfuscation is frequently not enough to cover some use cases.

While obfuscation should provide a good way of preventing reverse-engineering and making it extremely difficult for anyone (including attackers) to understand, target, and potentially steal

the logic of your app, more advanced threats like code tampering, data exfiltration, piracy, and automated abuse require advanced JavaScript protection.

JavaScript Protection: Environment Checks/Locks

One important type of JavaScript protection is the so-called environment checks or code locks. These allow locking the JavaScript code to only run in specific allowed environments.

These environments typically include operating systems, browsers, domains, dates, or certain types of devices, like mobile phones that haven't been rooted or jailbroken.

Each of these locks can help accomplish different requirements. For example, if you have an app that deals with very sensitive data or performs critical tasks, you can prevent it from running on rooted or jailbroken devices because these are more vulnerable to attacks. And if you want to enforce licensing agreements, you can deliver a product demo to a client and have that code locked to the client's domain and automatically expire after a specific date.

Usually, whenever a lock violation occurs, the application will break. So, these locks can be especially useful against piracy and license violations. But there's another type of JS protection that is very useful in almost every single use case: runtime protection.

JavaScript Protection: Runtime Protection

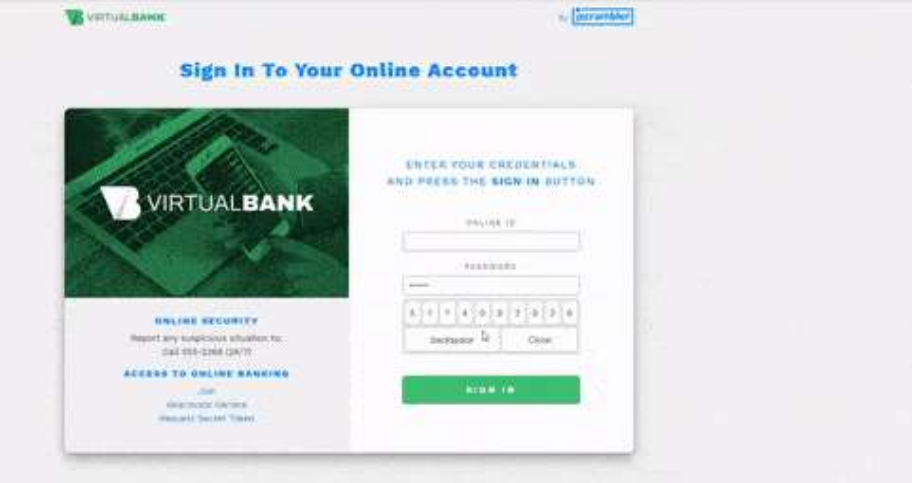
Motivated attackers may not be easily dissuaded by obfuscation alone. In certain types of attacks, such as data exfiltration and automated abuse, the potential gains of a successful attack may justify an extensive effort to reverse-engineer the code.

The most common first step of reverse engineering is to try to understand the logic of the obfuscated code by debugging it and experimenting with it at runtime to gradually understand portions of the code.

A methodical approach may eventually yield some results (which will greatly vary depending on the tool that was used to obfuscate the code and the usage of polymorphic obfuscation). Runtime protection can make this reverse engineering process much harder by preventing any type of debugging or tampering of the protected code.

From a technical perspective, this is achieved by scattering integrity checks and anti-debugging traps throughout the source code.

As a first step to understanding the logic of the protected code, attackers will normally use a debugger and perform a step-by-step inspection of the code. If the code has been instrumented with anti-debugging traps, whenever attackers attempt to use a debugger, the traps will be triggered, breaking the application on purpose and getting attackers stuck in an infinite debugger loop, as we can see below.



When attackers cannot inspect the code dynamically by debugging it, then their next best step is downloading the code to statically analyze and modify it. Successfully modifying code is an essential step for anyone reversing or tampering with an application. However, if the source code contains integrity checks, once any changes are made (like simply changing a single character), these checks will be triggered, also breaking the code to prevent the attack from being successful.

As you may expect, all these locks and checks will begin to frustrate attackers, especially when they are coupled with additional countermeasures.

JavaScript Protection: Countermeasures

Usually, whenever there's a violation of a code lock or when anti-debugging traps or integrity checks are triggered, the default response is to derail the app execution to contain a possible threat.

However, breaking the app is only an example of several possible countermeasures. Other possibilities include:

- redirecting attackers to another page, to make them lose all progress;
- deleting the cookies, namely as a countermeasure to thwart scraping attacks;
- sending a real-time notification to a dashboard with the full details of the incident;
- destroying the environment of the attacker, by crashing the memory, destroying the session, and destroying objects;

- triggering a custom callback function for complete flexibility and control over the intended reaction.

This level of customization can definitely help fine-tune the overall code protection to match your specific use case.

Security In-Depth

While JavaScript obfuscation is often the entry point for those looking for some degree of source code protection, the bottom line is that obfuscation is usually a means to an end.

While developing your application's threat model, it's important to understand the risks posed by unprotected JavaScript code. Answering these security concerns always calls for a security in-depth approach, meaning incorporating source code protection into a robust client-side security strategy.

When it comes to ensuring the maximum level of protection of JavaScript source code, the best answer is to rely on a trustworthy vendor that provides resilient and potent obfuscation, along with runtime protection and a broad range of integrations.

For over 10 years, Jscrambler has been the leading JavaScript obfuscation and protection technology. With a strong investment in R&D, Jscrambler has introduced most of the innovative features and patents when it comes to JavaScript protection, including features like Self-Healing, Self-Defending and Control-Flow Flattening.

After over 700,000 protected code builds and recognition by advisory firms like Gartner, Jscrambler is the trusted choice of the Fortune 500 and thousands of companies globally, whose feedback is excellent: