jscrambler

JavaScript obfuscation checklist

Checklist for JavaScript obfuscation

JavaScript: A ubiquitous language

JavaScript has changed drastically since it was introduced, especially when Node.js entered the scene, along with JS frameworks such as jQuery and React. Today, JavaScript is the backbone of Web, mobile, and even desktop apps. It paved the way to highly advanced services such as streaming and online banking and has unlocked new business models in sectors like Finance, eCommerce, and Healthcare.

97% Modern web apps using JavaScript.	**100%** Fortune 500 companies using JavaScript.	**>55%** Mobile apps using JavaScript.

Today, with cross-platform frameworks like React Native and Ionic, companies can cut their development time and cost significantly by reusing a shared codebase for Web, mobile, and desktop platforms.

But even with all its numerous advantages and business value, we must still consider the changes in our threat model when using JavaScript-based web and mobile apps.

The threat of exposed JavaScript

Because client-side JavaScript must be executed by a browser in order to work, it is exposed and **anyone can access, read, and modify it**. However, while it's advisable to keep important logic on secure execution environments (i.e. the backend), it's often infeasible to keep this logic out of the client-side - for example, due to the inexistence of a back-end (mobile apps) or the need to avoid performance losses.

As a result, companies' proprietary algorithms and logic end up running in an adversarial environment, which opens the door to a series of attacks, such as **automated abuse, piracy, intellectual property theft, and data exfiltration**.

JavaScript obfuscation

The security threats of unprotected JavaScript source code have widely different degrees of complexity and impact on the business. To mitigate these threats, a suitable approach is **protecting the code.**

Source code protection is mentioned in some of the most relevant guides for application security. Namely, the **ISO 27001** information security standard states: ***"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."***

OWASP also advises this type of protection in its **Mobile Top 10 Security Risks** guide:
"An attacker may exploit reverse engineering to (...) Reveal information about back end servers; Reveal cryptographic constants and ciphers; Steal intellectual property; Perform attacks against back end systems; or Gain intelligence needed to perform subsequent code modification. In order to prevent effective reverse engineering, you must use an obfuscation tool."

JavaScript obfuscation can be used to transform the code into a new version that is extremely hard to understand and reverse-engineer, but that preserves its original functionality. Typically, JavaScript obfuscation should include a combination of several different transformations and can be measured using three different metrics:

→ **Potency:** How hard it is for a human to understand the obfuscated code.

→ **Resilience:** How hard it is to revert the transformed code to its original form.

→ **Cost:** Impact on the application's file size and execution time.

JavaScript obfuscation checklist

There are many different possible approaches to JavaScript obfuscation. Below, we highlight the several verifications that should be done to evaluate a suitable JavaScript obfuscation product.

Obfuscation Verifications

#	ACTION
1.1	Verify that the identifiers of the application (e.g. variables and function names) and of native APIs (DOM, Node.js API) are being renamed.
1.2	Verify that identifiers are replaced across multiple files types (e.g. HTML and JavaScript files).
1.3	Verify that, when eval or eval-like expressions are being used, names contained in the evaled expression have also been renamed accordingly.
1.4	Verify that dead code is added to the application to increase confusion in the program analysis.
1.5	Verify that the application's control flow is obfuscated and flattened and that it's no longer trivial to distinguish between if's, Else's, While's and For's and where the control flow is going next.
1.6	Verify that the control flow obfuscation makes intra-function/intra-module control flow become inter-function (e.g. by outlining functions).
1.7	Verify that the control flow obfuscation is protected by resilient and opaque predicates that are not easily understood by a human or easily deobfuscated using static analysis.
1.8	Verify that the control flow obfuscation generates alternative branches that are selected at runtime.
1.9	Verify that the control flow obfuscation makes inter-function/inter-module control flow become intra-function (e.g. by inlining functions/modules)
1.10	Verify that booleans, numbers, objects, arrays, strings, and regex expressions become invisible to humans or are otherwise encoded beyond recognition.
1.11	Verify that, in scenarios where sensitive data is being exchanged, data files (e.g. JSON, images) or streams are encoded or encrypted, only being decoded or decrypted in runtime.
1.12	Verify that each new code protection outputs different obfuscation results, with unpredictable changes to their order and frequency.

#	ACTION
1.13	Verify that any dead code injected is diverse across different protections
1.14	Verify that the additional diversity can be obtained by changing the order or frequency of the protection transformations.
1.15	Verify that the order of the functions inside each file is different across different protections.
1.16	Verify that statements inside a certain scope change their relative position across protections.
1.17	Verify that predicates are resilient and opaque predicates that are not easily understood by a human nor easily reversed using static analysis.
1.18	Verify that the resulting obfuscation cannot be reversed using reverse engineering or code optimization tools.
1.19	Verify that the protection is resilient against partial evaluation and symbolic execution-based reverse engineering tools and techniques, and that their resulting code and its control flow is not simpler to read and to understand.
1.20	Verify that the app detects the presence of code injection tools, hooking frameworks, and debugging servers.

Data and Code Integrity Verifications

Although JavaScript obfuscation brings an important level of security to web and mobile applications, additional protective layers should be added, **especially in applications that handle sensitive data and operations.**

Specific verifications should be made to attest to the level of integrity of the code and data, as outlined below.

#	ACTION
2.1	Verify that the protection injects multiple functionality independent integrity checks throughout the protected code that, in the context of the overall protection scheme, forces adversaries to invest a significant manual effort to be able to tamper with the code or data.
2.2	Verify that the integrity checks have good coverage of all the JavaScript in the application, including inline JavaScript.

#	ACTION
2.3	Verify that no checksums or encryption keys can easily be found in the code, namely in visible strings or inside objects, using static analysis tools.
2.4	Verify the presence of integrity checks that are resilient to code poisoning.
2.5	Verify that native API calls (e.g. Web Cryptography API, DOM, Node.js) are also subject to integrity checks.
2.6	Verify that the integrity checks are diverse across different protections.

Runtime Protection Verifications

Even with the security layers presented before, a relentless attacker may try to understand the code logic of obfuscated code by debugging it and experimenting with it at runtime. Runtime protection techniques should be added to the code in order to give it **anti-debugging and anti-tampering capabilities, as outlined below.**

#	ACTION
3.1	Verify that there are multiple functionally independent debugging defenses that, in the context of the overall protection scheme, force adversaries to invest a significant manual effort to enable debugging.
3.2	Verify that, if the goal of obfuscation is to lock the code to a certain environment (e.g. OS, Browser, Domain) and that it takes a significant amount of manual work to remove all the checks.
3.3	Verify that, if the goal of obfuscation is to lock the code to a date interval and that it takes a significant amount of manual work to remove all the checks.
3.4	Verify that the app implements a 'device binding' functionality when a mobile device is treated as being trusted. Verify that the device fingerprint is derived from multiple device properties.
3.5	Verify that logs, debug messages and stack traces have been eliminated, making the debugging activity significantly harder.
3.6	Verify that the protection, upon the detection of an attack (e.g. running in an unauthorized environment, code tampering), can optionally execute a custom callback (e.g. terminate the session, remove a file, send a request to a remote API).

#	ACTION
3.7	Verify that the protection, upon the detection of an attack (e.g. running in an unauthorized environment, code tampering), can optionally trigger a real-time notification with details about the attack.
3.8	Verify that the app detects, and responds to, being run in an emulator using any method.
3.9	Verify that the detection mechanisms (including responses to tampering, debugging, and emulation) trigger responses of different types, including delayed and stealthy responses that don't simply terminate the app.

Frequently Asked Questions

Can’t I just encrypt JavaScript?

Encrypting JavaScript doesn’t work because the browser always needs to render JS so that it works. If we have a decryption key we need to supply to the browser, that key can be compromised and the code easily accessed.

Isn’t it better to avoid having client-side JavaScript?

As a rule of thumb, if JavaScript has sensitive data that shouldn’t be accessed by third-parties, it should be kept confined to trusted execution environments such as the backend servers. However, this often is not a possibility - like in cases where there’s no backend.

Another common issue is that server calls take time and, in services where performance is crucial - such as streaming, e-commerce, or gaming - storing all JavaScript on the server is not an option.

Doesn’t SAST/DAST protect JavaScript?

Security testing tools like SAST and DAST are widely used to inspect the application’s source code, check if it contains any vulnerabilities, and attempt to fix them.

Considering that the typical JavaScript application today has well over 1000 code dependencies and an absolute mess of sub-dependencies, development teams need SAST and DAST to gain visibility over potentially insecure code.

However, even if we find and fix every single vulnerability in our JavaScript code, at the end of the day our JavaScript is still plain, easy to understand code.

Why Choose Jscrambler?

Expertise: Jscrambler started working on JavaScript code protection in 2009, more than 10 years ago. Many innovative features and patents in this area were introduced by Jscrambler's continuous R&D efforts. This R&D team continuously updates Jscrambler to resist all reverse engineering tools and techniques.

Unmatched capabilities: Jscrambler Code Integrity has both the largest and most powerful set of JavaScript code transformations, many of which are unique, like Jscrambler's Control Flow Flattening, Self-Healing, and robust source maps. Other code protection tools often break the code and are far less resilient.

JavaScript threat monitoring: This innovative new feature actively monitors any attempts to attack protected code and shows real-time alerts on the Jscrambler dashboard when its anti-debugging and anti-tampering features stop an attack.

Maturity: With a sophisticated testing process, Jscrambler ensures every new feature is compatible with all browsers, JS frameworks, and libraries. If a single test fails, our build fails. This means that only pristine releases reach Jscrambler clients.

Proven in demanding scenarios: Jscrambler Code Integrity is trusted by the Fortune 500 and thousands of businesses globally, totaling over 500,000 code builds protected by Jscrambler.

Endorsed by the market: Gartner, a global research and advisory firm, has consistently recognized Jscrambler in the Market Guides for In-App Protection and Online Fraud Prevention.

If you want to know more about how Jscrambler can help you prevent client-side attacks, don't hesitate to contact us.

hello@jscrambler.com | +1 650 999 0010