JavaScript Software Protections Security Checklist

The JavaScript Software Protection Security Checklist is a vendor-independent effort that aims at helping Security Practitioners analyzing the potency and resilience of a JavaScript Software Protection.

The checklist follows the OWASP MASVS [1] organization model of enumerating different verification requirements across a number of different categories. It builds upon its *V8: Resilience Requirements* section, further expanding it and adjusting it to a different scope: JavaScript Software Protections - which is not only applicable to Mobile Applications having JavaScript but, in fact, to any other kind of JavaScript-based application (e.g. Web, Node.js) for which one may consider using software protections. Some inspiration was also drawn from the OWASP Mobile Security Testing Guide [2].

Verification requirements were written, as much as possible, to be easy for anyone to follow and independently verify them. However, we recognize that some specific requirements might be harder to verify without 1) using specialized troubleshooting features of the software protection solution; 2) needing help from the software protection vendor, or, 3) without a significant amount of time invested.

These verification requirements do not aim to verify if some feature is available in the vendor solution, if the solution is reliable (i.e. does not break the code), or if it is compliant with all JavaScript engines. Even though those are important matters, they are considered out of scope for this checklist.

V1: Symbol Renaming

#	Description	1	2	3
1.1	Verify that the source application identifiers (e.g. variables and function names, namespace identifiers like foo.bar or foo.bar.baz) are being renamed	✓	✓	✓
1.2	Verify that identifiers are replaced across multiple files types (e.g. HTML and JavaScript files)	✓	✓	✓
1.3	Verify that native API (DOM, Node.js API) identifiers (e.g. window, window.location, fs.readFile) are being renamed		✓	✓
1.4	Verify that, when eval or eval-like expressions are being used, names contained in the evaled expression have also been renamed accordingly			✓

V2: Control Flow

#	Description	1	2	3
2.1	Verify that new predicates are being injected with dead code in order to create further confusion in the program analysis	✓	✓	✓
2.2	Verify that the control flow obfuscation makes intra-function/intra-module control flow become inter-function (e.g. by outlining functions)		✓	✓
2.3	Verify that the control flow is 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		✓	✓
2.4	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		✓	✓
2.5	Verify that the control flow obfuscation generates alternative branches that are selected in runtime			✓
2.6	Verify that the control flow obfuscation makes inter-function/inter-module control flow become intra-function (e.g. by inlining functions/modules)			✓

V3: Data Obfuscation

#	Description	1	2	3
3.1	Verify that booleans and numbers originally present in the source code are no longer visible	✓	✓	✓
3.2	Verify that is statically difficult to tell which values are stored in JavaScript arrays and objects	✓	✓	✓
3.3	Verify that there are transformations capable of encoding or encrypting strings, that become invisible to humans, and in a way that is not reversible automatically using static code analysis or code optimization tools	✓	✓	✓
3.4	Verify that regex expressions are obfuscated		✓	✓
3.5	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			✓
3.6	Verify that, in scenarios where the goal of protection is to protect the secrecy of a cryptographic key, an obfuscation scheme is used that makes retrieving the original cryptographic key very costly			✓

V4: Data and Code Integrity

#	Description	1	2	3
4.1	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 significant manual effort to be able to tamper with the code or data	✓	✓	✓
4.2	Verify that the integrity checks have good coverage of all the JavaScript in the application, including inline JavaScript		✓	✓
4.3	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		✓	✓
4.4	Verify the presence of integrity checks that are resilient to code poisoning			✓
4.5	Verify that native API calls (e.g. Web Cryptography API, DOM, Node.js) are also subject to integrity checks			✓

V5: Runtime defenses

#	Description	1	2	3
5.1	Verify that there are multiple functionally independent debugging defenses that, in the context of the overall protection scheme, forces adversaries to invest significant manual effort to enable debugging	✓	✓	✓
5.2	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		✓	✓
5.3	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		✓	✓
5.4	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		✓	✓
5.5	Verify that logs, debug messages and stack traces have been eliminated, making the debugging activity significantly harder			✓
5.6	Verify that the protection, upon the detection of an attack (e.g. running in a unauthorized environment, code tampering), can optionally execute a custom callback (e.g. terminate session, remove file, send request to a remote API)			✓

V6: Diversity

#	Description	1	2	3
6.1	Verify that symbols that are renamed always get different names across protections	✓	✓	✓
6.2	Verify that any dead code injected is diverse across different protections	✓	✓	✓
6.3	Verify that the additional diversity can be obtained by changing the order or frequency of the protection transformations		✓	✓
6.4	Verify that the order of the functions inside each file is different across different protections		✓	✓
6.5	Verify that statements inside a certain scope change their relative position across protections		✓	✓
6.6	Verify that the integrity checks are diverse across different protections			✓
6.7	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			✓

V7: Resilience

#	Description	1	2	3
7.1	Verify that predicates are resilient and opaque predicates that are not easily understood by a human nor easily reversed using static analysis	✓	✓	✓
7.2	Verify that any encryption or encoding, if used, cannot be easily reversed by using automated code optimization tools	✓	✓	✓
7.3	Verify that the app detects, and responds to, being run in an emulator using any method		✓	✓
7.4	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			✓
7.5	Verify that the app detects the presence of code injection tools, hooking frameworks and debugging servers			✓
7.6	Verify that, if the goal of obfuscation is to protect sensitive computations, an obfuscation scheme is used that is both appropriate for the particular task and robust against manual and automated de-obfuscation methods, considering currently published research. The effectiveness of the obfuscation scheme must be verified through manual testing. Note that hardware-based isolation features are prefered over obfuscation whenever possible			✓

References

[1] OWASP Mobile Application Security Verification Standard (MASVS): https://github.com/OWASP/owasp-masvs
[2] OWASP Mobile Security Testing Guide: https://www.owasp.org/index.php/OWASP_Mobile_Security_Testing_Guide