language: JavaScript

DEOBFUSCATION:

TOOLS:

- open source -

{ relative.im }
[information from GitHub:]
JavaScript cleaner and deobfuscator.
[sources:]
found via www.google.it
https://deobfuscate.relative.im/
https://github.com/relative/synchrony

{ JS Beautifier }
[information from website:]
It is a popular open-source tool for beautifying and analyzing JavaScript code. It can help you reformat and deobfuscate code by improving its readability.
All of the source code is completely free and open, available on GitHub under MIT licence.
[sources:]
found via www.google.it
https://beautifier.io/
https://github.com/beautify-web/js-beautify

{ deobfuscate.io }
[information from GitHub:]
A simple but powerful deobfuscator to remove common JavaScript obfuscation techniques.
Unpacks arrays containing literals (strings, numbers etc) and replaces all references to them.
Removes simple proxy functions (calls to another function), array proxy functions and arithmetic proxy functions (binary expressions).
Simplifies arithmetic expressions.
Simplifies string concatenation.
Renames unreadable hexadecimal identifiers (e.g. _0xca830a).
Converts computed to static member expressions and beautifies the code.
Experimental function evaluation.
[sources:]
found via www.google.it
https://deobfuscate.io/
https://github.com/ben-sb/javascript-deobfuscator
https://obf-io.deobfuscate.io/
https://github.com/ben-sb/obfuscator-io-deobfuscator

{ JACOB (JAvascript COde Bulldozer) }
[information from pdf:]

A deobfuscator/decompiler written in JavaScript for JavaScript "binaries", currently targeted at collina.js from AliExpress.
Reconstructing a flattened CFG, recovering control structures and deobfuscating literals via partial evaluation. In JavaScript.
Code obfuscation may also be adopted by companies for legitimate intellectual-property protection, a dilemma remains on whether a script is harmless or malignant, if not criminal, JACOB help analysts deal with such a dilemma.
Based on five steps, namely: (1) source code parsing, (2) control flow graph recovery, (3) region identification, (4) code structuring, and (5) partial evaluation.
(1) This phase aims to create a standardized interface allowing the subsequent steps to easily retrieve basic blocks as abstract syntax tree chunks which can be easily queried and traversed, while retaining important information about code structure.
(2) The dispatcher is connected to every node, but has no knowledge about the application logic: determining how execution shall proceed is demanded to basic blocks. Once the specific dispatcher implementation has been understood, it is possible to reconnect consequent basic blocks.
(3) To recover the original control structures, basic blocks are gathered into groups, called regions, pertaining to any of the three classes proposed in:• Iteration • Selection • Sequence. JACOB takes into account only two types of regions: cyclic (iteration) and acyclic (selection, sequence).
(4) The code structuring process addresses the reconstruction of different control flow structures, including if-then, if-else, and while loops. Owing to the previously achieved hierarchical scheme, at every level the correct control structure depends solely on the topological shape of the current graph.
(5) A technique for program optimization through specialization. In theory, a partial evaluator accepts in input a program and outputs a specialized version that is equivalent in behavior, where all the static inputs have been precomputed. Conceived for optimization purposes, a partial evaluator elegantly defeats data splitting as well. Reconstructing data at runtime is inherently inefficient, thus optimizing the code (by precomputing static inputs) results in a program where data reconstruction code is replaced with the final value of data itself.
Although code obfuscation can be legitimately adopted for intellectual-property protection, it can also be exploited for illegal/unwanted actions (fingerprinting included), as well as for hiding criminal intents. As many other things, code obfuscation is a double-edged sword, so it is paramount to know which edge of the sword is actually being wielded: JACOB was designed precisely to serve this purpose.
[sources:]
found via professor
https://www.sciencedirect.com/science/article/pii/S1877050922010249
https://github.com/ceres-c/bulldozer

{ de4js }
[information from GitHub:]
JavaScript Deobfuscator and Unpacker
features: Works offline; Source code beautifier / syntax highlighter; Makes obfuscated code readable.
This repository has been archived by the owner on Dec 7, 2021. It is now read-only.
[sources:]
found via www.google.it

https://github.com/lelinhtinh/de4js

{ de4dot }

[information from website:]

Is a .NET deobfuscator and unpacker. It will try its best to restore a packed and obfuscated assembly to almost the original assembly.
Most of the obfuscation can be completely restored (eg. string encryption), but symbol renaming is impossible to restore since the original names aren't (usually) part of the obfuscated assembly.

[sources:]

found via www.google.it and https://www.kali.org/tools/
https://www.kali.org/tools/de4dot/
https://salsa.debian.org/pkg-security-team/de4dot

- not open source -

{ Code Amaze }

[information from website:]

Javascript Deobfuscator/ Unpacker is an online tool that converts the obfuscated/ packed JavaScript source code into readable form.

[sources:]

found via www.google.it
https://codeamaze.com/code-beautifier/javascript-deobfuscator

{ Unminify }

[information from website:]

Free tool to unminify (unpack, deobfuscate) JavaScript, CSS, HTML, XML and JSON code, making it readable and pretty.
This unminifying tool will take minified code and expand it so it is easier for humans to read. It can do this with files or with copied code snippets. It supports JavaScript (JS), CSS, HTML, XML, and JSON code.
This unminification happens within your browser itself, so you don't need to worry about a server going through your private or proprietary code. All your code and files stay local.
However, this unminifying tool will only parse out the code based on the minified code. It will not restore a minified file to its original state. In other words, if the person who wrote the code added lots of notes and superfluous information in the original file, this will not restore those notes as they would have been discarded during the minification process. It will only take the code you give it and deobfuscate or unpack it.

[sources:]

found via www.google.it
https://unminify.com/

(NO, it doesn't seem reliable.) JSDetox

[information from GitHub and website:]

It is a powerful JavaScript deobfuscation tool that can automatically reverse engineer and remove obfuscation techniques such as string encryption, code splitting, etc.

JSDetox is a Javascript malware analysis tool using static analysis / deobfuscation techniques and an execution engine featuring HTML DOM (Document Object Model) emulation.
A tool to support the manual analysis of malicious Javascript code.
While it does use the browser as user interface, the whole analysis/execution of the javascript code is done in the backend.
[sources:]
found via www.google.it
https://github.com/svent/jsdetox
https://svent.dev/projects/jsdetox/

PUBLIC DATASETS:
Not found.

MAIN METHODOLOGIES:

Deobfuscate Manually:
A Tutorial About Dealing With an Obfuscated Code.
https://infosecwriteups.com/deobfuscation-for-beginners-944947ee2b9f
[sources:]
found via www.google.it

[In the tools_table document, in the table of deobfuscating tools, other methodologies are listed and are also briefly explained below the table.]

[The next methodologies, from here, found through a search tool that works through artificial intelligence called: PizzaGPT (https://www.pizzagpt.it/):]
Static Analysis:
Analyzing the code without executing it to understand its structure, identify patterns, and locate obfuscated sections. This involves examining the code manually or using automated tools like disassemblers, decompilers, and code analyzers.

Dynamic Analysis:
Running the obfuscated code in a controlled environment, such as a virtual machine or sandbox, to observe its behavior during execution. This helps to uncover hidden functionality, such as code that is only executed under specific conditions.

Code Decompilation:
Transforming lower-level code, such as assembly language or bytecode, into higher-level code, such as C or Java, to enhance readability and understand the obfuscated logic. Decompilers can provide valuable insights into the code's functionality and structure.

Code Inspection:
Thoroughly examining the obfuscated code, looking for patterns, strings, or API calls that indicate suspicious or malicious behavior. This involves identifying keywords, function names, or other recognizable patterns that may reveal the code's purpose.

Symbolic Execution:
Using symbolic values instead of concrete inputs during code analysis to explore different execution paths. This helps to understand how the code operates under various conditions, unveiling hidden behaviors and potential vulnerabilities.

Control Flow Analysis:
Analyzing the program's control flow, such as loops, branches, and conditional statements, to identify complex control structures. Understanding the control flow helps to grasp the intended behavior of the obfuscated code.

Behavioral Profiling:
Monitoring the runtime behavior of the obfuscated code to identify differences in resource usage, network communication, or API calls between benign and obfuscated versions. Comparing the results helps to understand the code's purpose and its potential impact.

Collaborative Analysis:
Collaborating with other researchers or experts in the field to share knowledge, insights, and techniques for deobfuscation. By leveraging collective expertise, researchers can often overcome individual challenges in understanding obfuscated code.

Deobfuscation often requires a combination of different techniques and expertise to successfully reverse engineer and understand obfuscated code.

Manual Analysis:
This method involves manually inspecting the code, looking for patterns, analyzing the flow of execution, and identifying obfuscated portions. It requires a deep understanding of programming languages, algorithms, and various obfuscation techniques. Manual analysis can be time-consuming but sometimes necessary for small and complex codebases.

Automated Tools:
There are several static analysis tools available that can help automate the deobfuscation process. These tools utilize various algorithms and techniques to detect and uncover the obfuscated parts of the code. While they can significantly speed up the process, they may not always be able to handle complex obfuscation techniques or unknown patterns.

Symbolic Execution:
Symbolic execution is a technique that involves executing the code on symbolic inputs instead of concrete values. It helps to analyze the different program paths and resolve conditions. By applying symbolic execution to obfuscated code, it becomes easier to identify how the code obfuscation affects the control flow and modify it to obtain the original functionality.

Abstract Interpretation:
Abstract interpretation is a technique used to analyze the behavior and properties of programs at a higher level of abstraction. By abstracting the code and analyzing its abstract representation, certain aspects of the obfuscation can be deciphered. It can help in detecting patterns, removing irrelevant code, and extracting useful information.

Dynamic Analysis:
Dynamic analysis involves running the code and analyzing its behavior during runtime. This approach can help in understanding the code's purpose, revealing hidden functionality, and identifying any anti-analysis methods. By monitoring function calls, network communications, and system interactions, dynamic analysis provides insights into the obfuscated code's behavior.

Code Instrumentation:
Code instrumentation involves modifying the code to insert additional instructions or code snippets that aid in the deobfuscation process. It can include adding debugging code, logging data, or hooks that allow the analysis of obfuscated operations and intermediate results.

Collaboration and Community Support:
Deobfuscation can be a challenging task, especially when dealing with unknown or novel obfuscation techniques. Community support, forums, and collaboration with other experts could provide insights, shared experiences, and even pre-existing deobfuscated versions of similar obfuscation methods.
[to here]

Deobfuscation, unpacking, and decoding of obfuscated malicious JavaScript for machine learning models detection performance improvement.

To effectively conduct feature learning on such obfuscated JS codes, it is essential to deobfuscate, unpack, and decode the obfuscated JS codes. We propose the use of a JS code beautifier, formatter, VM, JS code editor, and a python *int_to_str()* function to obtain a Plain-JS code from an obfuscated one and the FastText model for feature learning coupled with a classifier for the prediction task.

[sources:]
found via Google Scholar
https://ietresearch.onlinelibrary.wiley.com/doi/full/10.1049/trit.2020.0026

Android application forensics: A survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations.

[sources:]
found via Google Scholar
https://www.sciencedirect.com/science/article/pii/S2666281721002031#sec7

Deobfuscation is in NP.

Under some very general assumptions about how a program-obfuscation works, deobfuscation is NP-easy.
While this does not immediately lead to a practical deobfuscation algorithm, it shows that deobfuscation is much easier than most other exact program analyses, such as the related problem of program optimization.

[sources:]
found via Google Scholar
https://www.cs.princeton.edu/~appel/papers/deobfus.pdf

Deobfuscation Reverse Engineering Obfuscated Code.
This section describes a number of analyses and program transformations that we have found useful for reverse engineering obfuscated code.
[sources:]
found via Google Scholar
http://www2.cs.arizona.edu/~debray/Publications/unflatten.pdf

Code deobfuscation by program synthesis-aided simplification of Mixed Boolean-Arithmetic expressions.
[sources:]
found via www.google.it
https://diposit.ub.edu/dspace/bitstream/2445/176925/3/176925.pdf

JavaScript Deobfuscation.
[sources:]
found via www.google.it
https://dhanushnehru.medium.com/javascript-deobfuscation-7da22f0b98fa

Automated (simple) deobfuscation of .NET malware.
Obfuscation techniques can make the analysis of obfuscated .NET malware challenging, even though .NET binaries can be easily decompiled. Manually analyzing such obfuscated malware can be a tedious process. In this article, we will explore how to automate the deobfuscation of a SolarMarker malware sample using the de4dot and MSIL tools.
[sources:]
found via www.google.it
https://medium.com/@upsidedwn/automated-simple-deobfuscation-of-net-malware-8724d36710be
https://github.com/NotPrab/.NET-Deobfuscator

POTENTIAL NOVEL METHODS:

[Found through a search tool that works through artificial intelligence called: PizzaGPT (https://www.pizzagpt.it/):]
Based on neural networks:
Neural networks can be used for deobfuscating code by learning the patterns of obfuscated code and predicting the original, clean version. Here are some common approaches:
1. Supervised learning: In this approach, a neural network is trained using a dataset of obfuscated code snippets paired with their corresponding clean versions. The network learns to extract features from the obfuscated code and predict the original code. The dataset can be created by manually obfuscating clean code or by using existing obfuscated code samples.
2. Reinforcement learning: In this approach, the neural network is trained to deobfuscate code by interacting with an environment and receiving rewards for successfully predicting the clean code. The environment can simulate different levels of obfuscation, and the network learns to optimize its predictions by learning from the rewards received.

3. Unsupervised learning: This approach involves training the neural network on a dataset of obfuscated code without paired clean versions. The network learns to discover hidden structures, patterns, and relationships within the obfuscated code, which can help in deobfuscation. However, this approach may require large amounts of data for effective training.
4. Transfer learning: Instead of training from scratch, a pre-trained neural network, such as a language model or a code generator, can be fine-tuned and adapted for deobfuscation tasks. This can be done by training the network on a relatively small dataset of paired obfuscated and clean code snippets.
It's important to note that while neural networks can be effective in deobfuscation tasks, they are not a silver bullet and may not work well with all types of obfuscation techniques. Code obfuscation can employ a wide variety of techniques, such as renaming variables, inserting dead code, or applying complex encoding schemes, which can pose challenges for deobfuscation methods. Hence, a combination of different techniques and approaches may be required to successfully deobfuscate code.

OBFUSCATION:

TOOLS:

- open source -

{ obfuscator.io }

[information from GitHub:]

JavaScript Obfuscator is a free online tool that obfuscates your source code, preventing it from being stolen and used without permission.
JavaScript Obfuscator is a powerful free obfuscator for JavaScript, containing a variety of features which provide protection for your source code.
Key features: variables renaming; strings extraction and encryption; dead code injection; control flow flattening; various code transformations and more...

[sources:]

found via www.google.it
https://obfuscator.io/
https://github.com/javascript-obfuscator/javascript-obfuscator

{ javascriptobfuscator }

[information from website:]

Prevent people from stealing your JavaScript code directly from the browser. It is specially useful for games, template developers that sell their work on ThemeForest or related template marketplaces. Show your work to the client without giving it away. This tool allows you to simply paste your code in the editor or upload the file from your device.
The obfuscator runs in your own machine through a JavaScript worker, so if there's a lot of code to handle, the user interface won't freeze. Your code stays on your machine, there's no server-side processing.

[sources:]

found via www.google.it

https://javascriptobfuscator.dev/ (refer to obfuscator.io GitHub)
https://github.com/javascript-obfuscator/javascript-obfuscator

- not open source -

{ ByteHide }

[information from website:]

Free JavaScript Obfuscator Online
Welcome to the ultimate online JavaScript obfuscator. User-friendly tool that helps you protect your code by making it unreadable to others, ensuring your intellectual property stays safe. With features like string obfuscation and hex obfuscation, it's never been easier to secure your JavaScript code!

[sources:]

found via www.google.it
https://www.bytehide.com/products/shield-obfuscator/javascript

{ Code Beautify }

[information from website:]

Javascript Obfuscator is a free tool that can be used for obfuscating JavaScript code. It is mainly used to protect the source code of client-side web applications from reverse engineering and tampering. Copy, Paste, and Obfuscator.
Javascript Obfuscator can be used to protect the source code of client-side web applications by generating code that is difficult to read, understand, and modify. This improves the security of any application which is exposed to the internet in any way.

[sources:]

found via www.google.it
https://codebeautify.org/javascript-obfuscator#

{ Clean CSS }

[information from website:]

Obfuscation provides a way to protect your code by making it unreadable using advanced algorithms and also reduces the size of your files for speed. Don't worry your code will still work like normal, it just won't be readable and you can convert back at any time.
Obfuscator tool replaces symbol names with non-meaningfull ones (eg hello_moto will be converted to 1cd5dg4g1gf); replaces numeric constants with expressions (eg 232 will be converted to 0x29b9+2011-0x2d25); replaces characters in strings with their hex escapes (eg string "noob" will be converted to "`vcD"); removes comments completely (if selected, otherwise obfuscates comments); removes spaces and tabs and blank lines in the code; joins all the lines of code together; encodes the previous stages using advances algorithms.

[sources:]

found via www.google.it
https://www.cleancss.com/javascript-obfuscate/index.php

{ LambdaTest }

[information from website:]

Use this free online tool to obscure or conceal your JavaScript code for security purposes, to make it difficult to read and understand for humans but still able to execute normally by computers.

JavaScript Obfuscator is an online free tool used to convert JavaScript source code into a format that is more difficult to understand and reverse-engineer. The goal of obfuscation is to make the code harder to read and understand without affecting its functionality. It is done to protect intellectual property, and hide algorithms, making it difficult for hackers to reverse-engineer code.

[sources:]
found via www.google.it
https://www.lambdatest.com/free-online-tools/js-obfuscator

{ javascriptobfuscator.com }

[information from website:]
Javascript Obfuscator can protect and defend JavaScript Code from reverse engineering and tampering. JavaScript Obfuscator is fully compatible with ECMAScript latest versions. Javascript Obfuscator converts the JavaScript source code into obfuscated and completely unreadable form, preventing it from analyzing and theft. It's a 100% safe JavaScript minifier and the best JavaScript compressor.
There is a premium membership.

[sources:]
found via www.google.it
https://www.javascriptobfuscator.com/Javascript-Obfuscator.aspx

{ JavaScript Obfuscator Tool }

[information from website:]
JavaScript obfuscation is like putting a secret disguise on your code. It takes your readable JavaScript code and transforms it into a jumbled mess that's tough for humans to decipher. But don't worry, browsers and interpreters can still run the obfuscated code without a problem.

[sources:]
found via www.google.it and
https://stackoverflow.com/questions/69493470/obfuscate-compress-protect-javascript-code
https://developer.wikimint.com/p/javascript-obfuscator.html

{ JSDefender }

[information from website:]
Professional obfuscation and in-app protection for over 20 years.
Powerful Protection: Cutting-edge JavaScript obfuscation techniques with control-flow flattening, tamper detection and other in-app protection transforms.

[sources:]
found via www.google.it and
https://stackoverflow.com/questions/69493470/obfuscate-compress-protect-javascript-code
https://www.preemptive.com/products/jsdefender/

{ verimatrix }

[information from website:]
Implement a range of obfuscation methods.
control flow obfuscation

Rather than simply renaming functions, methods, and classes, control flow goes further to obfuscate your app logic and deter hackers.
symbol obfuscation
Rename telltale identifiers in your code - like methods and package names - to something meaningless that an attacker wouldn't understand.
string obfuscation
Obfuscate hardcoded information like character strings and other literal data to protect clues to the functionality of your code as it executes.
arithmetic obfuscation
Further confuse a would-be attacker by making it more difficult to decipher the result of arithmetic and logical instructions in your app.
[sources:]
found via www.google.it and
https://www.verimatrix.com/cybersecurity/code-obfuscation/

(NO) FREE Javascript Obfuscator
[information from website:] !probably unsafe site!
100% free one-way code blending & code minimizing. Just insert your code and obfuscate.
They also offer custom-private solutions: freejsobfuscator@gmail.com
[sources:]
found via features on https://github.com/lelinhtinh/de4js
http://www.freejsobfuscator.com/

PUBLIC DATASETS:
Not found.

MAIN METHODOLOGIES:

[In the tools_table document, in the table of obfuscating tools, other methodologies are listed and are also briefly explained below the table.]

[The next methodologies, from here, found through a search tool that works through artificial intelligence called: PizzaGPT (https://www.pizzagpt.it/):]
Code restructuring:
This involves modifying the structure and format of the code to make it harder to understand. This may include techniques such as renaming variables and functions with arbitrary or meaningless names, rearranging code blocks, adding unnecessary code snippets, and minimizing code comments.

String encryption:
In this technique, sensitive strings such as API keys, URLs, and database credentials are encrypted and decrypted at runtime. It makes it difficult for an attacker to extract valuable information from the source code directly.
Strings, such as API keys or sensitive data, can be encrypted and stored as ciphertext in the code. Decryption routines are then added to the code to decrypt the strings during runtime, making it harder for an attacker to uncover sensitive information.

Code encryption:
The entire code or specific critical sections are encrypted to prevent reverse engineering. The encrypted code is then decrypted at runtime, making it more challenging for an attacker to analyze and understand the logic.

Control flow obfuscation:
This technique alters the flow of the code execution, making it harder to follow. Techniques such as adding dead code, conditional branches, and code block rearrangements can be used to confuse attackers.
In this technique, the order and structure of the code are manipulated to confuse the flow of execution. This can be achieved through techniques like code-splitting, useless code insertion, and jumping through conditional statements.

Code obfuscation tools:
There are numerous code obfuscation tools available that can automatically apply various obfuscation techniques. These tools often combine multiple methodologies to provide a stronger level of protection.

Anti-debugging techniques:
These techniques make it difficult for attackers to use debuggers to analyze and understand the code. This may include adding checks for debugging tools, intentional code errors to confuse debuggers, or employing anti-debugging libraries.
To prevent code analysis and debugging, various anti-debugging techniques can be implemented, such as detecting debuggers or using anti-analysis tricks in the code. These techniques make it harder for attackers to trace and understand the code during runtime.

Dynamic code generation:
This technique involves generating code dynamically at runtime instead of having everything pre-defined in the source code. By generating code on-the-fly, it becomes harder for attackers to understand the logic and manipulate the code.

Code-based vulnerabilities:
Creating obfuscated code is not just about making it harder to read, but also ensuring the code is resistant to common vulnerabilities. Techniques such as input validation, secure coding practices, and avoiding hardcoded secrets help protect the code from common attack vectors.

Name Obfuscation:
This involves replacing meaningful variable and function names with randomly generated or encrypted names. This makes it harder to understand the code logic by obscuring the purpose and relationships between different parts of the code.

Code Polymorphism:
This technique involves creating multiple versions of the same code logic with slight variations. These variations are achieved by using alternative coding styles, different libraries, or syntax variations. This adds complexity to the code and makes it more challenging to understand and analyze.

Code Compression:
Code can be compressed or packed using algorithms like gzip or custom compression techniques. This reduces the size of the code and makes it more difficult to read and analyze.

Dead Code Insertion:
In this technique, non-functional and unused code snippets are inserted into the program. This can include fake functions, unreachable branches, and unused variables. Such additions confuse the reverse engineer and make the codebase more convoluted.
[to here]

[source of the one below from here: JACOB pdf]
automated code obfuscators rely on established techniques, among which:

• Control logic obfuscation. The logic ruling branching and decision-making in the code is obfuscated by different techniques: (a) control flow flattening, where the control flow does not develop in depth, because every basic block is redirected to a main dispatcher, which steers execution from one basic block to the subsequent; (b) bogus control flow, where useless control flow paths that cannot possibly be followed are deliberately inserted into the code, thereby undermining a purely static analysis, owing to a possibly overwhelming magnified code; the complexity can be increased even further, via opaque predicates, which may prevent both software and humans from inferring the actual execution flow; and (c) probabilistic execution, where blocks with different syntax, but equivalent behavior, are executed in an apparently arbitrary way.

• Layout obfuscation. The layout of the code is obfuscated while retaining the original syntax: (a) identifier replacement, where original variable identifiers are replaced with meaningless names such as one-letter identifiers or purposefully confusing names (characters like O, o, 0); to generate even more confusion, identifiers may be reused in different scopes; (b) dead code injection, where useless statements are inserted; and (c) symbol stripping, where unnecessary symbols, including debugging information, are removed.

• Data obfuscation. The referenced data is obfuscated: (a) data encoding, where data is encoded with a compression or encryption algorithm and decoded at runtime; (b) data splitting, where contiguous data is split and relocated appropriately in order to appear scrambled; and (c) literal to function-call conversion, where literals are replaced with calls to a function that returns the literal, given some specific arguments.

• Anti debugging, where the software, noticing it is being analyzed with a debugging tool, refuses to execute or takes specific code paths hiding relevant information.
[to here]