DEOBFUSCATION	
Tool	**Methodologies**
open source	

relative.im
https://deobfuscate.relative.im/
https://github.com/relative/synchrony
	array map; control flow; dead code;
	demangle; desequence; literal map;
	member expression cleaner; rename;
	simplify; string decoder.

JS Beautifier https://beautifier.io/ https://github.com/beautify-web/js-beautify	Restores indentation, spaces, newlines and control flow.
deobfuscate.io https://deobfuscate.io/ https://github.com/ben-sb/javascript-deobfuscator https://obf-io.deobfuscate.io/ https://github.com/ben-sb/obfuscator-io-deobfuscator https://www.npmjs.com/package/obfuscator-io	Unpacks arrays containing literals and replaces all references to them; Removes simple proxy functions, array proxy functions and arithmetic proxy functions; Simplifies arithmetic expressions; Simplifies string concatenation; Renames unreadable hexadecimal identifiers; Converts computed to static member expressions and beautifies the code; Experimental function evaluation.

JACOB
https://www.sciencedirect.com/science/article/pii/S1877050922010249
https://github.com/ceres-c/bulldozer
	source code parsing;
	control flow graph recovery;
	region identification;
	code structuring;
	partial evaluation.

de4js https://github.com/lelinhtinh/de4js	array decode; restore strings, control flow, indentation, spaces and newlines.

de4dot
https://www.kali.org/tools/de4dot/
https://salsa.debian.org/pkg-security-team/de4dot
	Decrypt strings, constants, methods;
	remove proxy methods; rename symbols;
	devirtualize code;
	decrypt resources, embedded files;
	remove tamper detection code;
	remove anti-debug code;
	control flow deobfuscation;
	restore class fields;
	convert PE exe to .NET exe; removes junk classes;
	fixes peverify errors;
	restore types of methods parameters and fields.

not open source	
Code Amaze https://codeamaze.com/code-beautifier/javascript-deobfuscator	Restores indentation, spaces, newlines and control flow.
Unminify https://unminify.com/	Restores indentation, spaces, newlines and control flow.

<u>**Deobfuscation tools and methodologies:**</u>

{ relative.im: }
In the src folder and then in the transformers folder:

[array map]
Maps arrays trying to understand what they are made up of. Array mapping typically involves encoding data or instructions as elements in an array, where the order and values of the array elements are used as a key to map to the original data or instructions. During deobfuscation, reverse engineers need to analyze how the array is constructed and how the mapping is done in order to understand the original data or instructions.

[control flow]
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.

[dead code]
Identifies and removes unused code inserted only to confuse. Dead code can be identified by analyzing the program's control flow and data dependencies. Once identified, the dead code can be safely removed from the program without affecting its functionality. This can help simplify the code and make it easier to understand and maintain.

[demangle]
Demangles proxy and string functions to understand usage. Demangling is the process of converting mangled or obfuscated function names back to their original human-readable form. Demangling is typically done using a demangler tool or library that can parse the mangled name and extract the original function name, along with any additional information such as parameters and return types.

[desequence]
Try to re-establish the original code sequence. "Desequencing" refers to the process of identifying and reconstructing the original sequence of instructions, statements, or operations that were manipulated or obscured in the obfuscated code.

[literal map]
Maps literals trying to understand what they are made up of. Mapping literals during code deobfuscation involves identifying and translating the obfuscated or encoded values back to their original form. This process typically involves creating a mapping table or dictionary that pairs the obfuscated values with their corresponding original values.

[member expression cleaner]
Tries to restore the original members of an expression. Analyze and simplify complex or obfuscated code by identifying and removing unnecessary or redundant expressions. Works by analyzing the code and identifying expressions that are not necessary for the overall functionality of the program. These expressions could be used by the developer to obfuscate the code and make it more difficult to understand, but they do not contribute to the core

functionality of the program. Once these unnecessary expressions are identified, it will remove or simplify them, making the code cleaner and easier to understand. This process can help in revealing the true purpose and logic of the code, making it easier for developers to analyze, debug, and modify the code as needed.

[rename]
Identifies possible identifiers or names of variables or functions and tries to rename them giving names that reflect the task they perform. The deobfuscator typically looks for patterns in the code that indicate the purpose of each element. For example, if a variable is used to store a user's name, it may be renamed to "userName" for clarity. Similarly, functions can be renamed based on their functionality, such as renaming a function that calculates the total price of items in a shopping cart to "calculateTotalPrice".

[simplify]
Simplifies the expressions it identifies, making them more understandable. Involves breaking down complex and convoluted code into a more understandable and readable form. This often involves identifying redundant or unnecessary operations, constants, and variables, and simplifying them to make the code more concise and easier to understand.

[string decoder]
Identifies strings and tries to recognize the meaning of the parts that form them. The decoder may use various techniques such as pattern recognition, string manipulation functions, and regular expressions to identify and decode the obfuscated strings. Once the strings are decoded, they can be analyzed and understood more easily, making the code clearer and easier to work with.

{ JS Beautifier: }
[Restores indentation, spaces, newlines and control flow]
The result is more understandable and it is easier to identify the use of certain parts of the code.

{ deobfuscate.io: }
[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). Involves identifying and analyzing the code to determine where proxy functions are being used and how they are being called. Simple proxy functions: These are functions that simply call another function within their body. To remove these proxy functions, the deobfuscation process involves identifying the proxy function calls and replacing them with direct calls to the original function being proxied. Array proxy functions: These functions involve using arrays to store function names or references and then calling those functions dynamically at runtime. Deobfuscation of array proxy functions requires identifying the array structure and the logic used to select and call the proxied functions. This may involve extracting the array contents and replacing the dynamic function calls with direct calls to the selected functions. Arithmetic proxy functions: These functions involve using arithmetic expressions to determine which function to call based on certain conditions or calculations. Deobfuscation of arithmetic proxy functions requires analyzing the arithmetic expressions and conditions to determine the logic used to select and call the proxied functions. This may involve simplifying the arithmetic expressions and replacing the dynamic function calls with direct calls to the selected functions.

[Simplifies arithmetic expressions and string concatenation.]
Involves breaking down complex expressions into simpler components and evaluating them, as well as identifying and combining concatenated string literals. This process helps in understanding the code and making it more readable.
For example, consider the following obfuscated code snippet:
var x = ((5 * 3) + 2) * 4;
var y = "Hello" + " " + "World";
During deobfuscation, the arithmetic expression can be simplified as follows:
var x = ((5 * 3) + 2) * 4; // Simplified to: var x = (15 + 2) * 4;
// Further simplified to: var x = 17 * 4; // Result: var x = 68;
Similarly, the string concatenation can be simplified as follows:
var y = "Hello" + " " + "World"; // Simplified to: var y = "Hello World";}

[Renames] unreadable hexadecimal identifiers (e.g. _0xca830a).

[Converts] computed to static member expressions and [beautifies] the code.

[Experimental function evaluation.]

BE CAREFUL when using function evaluation, this executes whatever functions you specify on your local machine so make sure those functions are not doing anything malicious.
This feature is still somewhat experimental, it's probably easier to use via the CLI as it's easier to find errors than the online version.
If the function is not a function declaration (i.e. a function expression or an arrow function expression) then the deobfuscator will not be able to detect the name of it automatically. To provide it use "#execute[name=FUNC_NAME]" directive.
You may need to modify the function to ensure it relies on no external variables (i.e. move a string array declaration inside the function) and handle any extra logic like string array rotation first.
You must first remove any anti tampering mechanisms before using function evaluation, otherwise it may cause an infinite loop.

{ JACOB: }
[source code parsing]
The first step is to identify code sections implementing application logic, ignoring ancillary structures introduced by the obfuscation process. In many languages, the execution flow can be redirected at will via jump-like statements. Since JavaScript lacks similar unconditional branching instructions, they need to be emulated by the authors of the obfuscator. Of course, thanks to the flexibility of the language, many solutions can be designed to achieve this goal, and implementations can drastically vary among different obfuscators, requiring ad-hoc parsing.
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.

[control flow graph recovery]
The control flow graph of a control-flow flattened program is flat, and the execution flow returns to the dispatcher after every basic block. To allow further analysis and code reconstruction, the original CFG must be recovered, restoring direct edges between nodes and removing the intermediary dispatcher. This process is called unflattening.
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. From a practical standpoint, this process could be as simple as tracking write accesses to a specific variable which controls the execution flow (e.g. a pseudo Program Counter). Or it might even be necessary to deal with some kind of dynamic execution routing which cannot be inferred statically, thus requiring a theorem prover to identify the next successor.
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.

[region identification]
JavaScript is a block-structured programming language, denoting that control structures rely on source code being organized in lexical blocks. Considering an if-else structure, each of the two branches is delimited by separators defining a block (curly brackets). The separators identify execution boundaries: statements in a block are executed until the block is consumed.
In source code, blocks are explicitly organized hierarchically; for example, an if-then can be nested within a while loop. On the other hand, basic blocks are minimal structures which can not contain, per definition, other basic blocks: this means they can not be organized hierarchically.
When going from a structured to an unstructured code representation (like when compiling C source into Assembly code), the hierarchical structure is lost. Nonetheless, the CFG of the unstructured program still contains, implicitly, information on the original block hierarchy. To recover the original control structures, basic blocks are gathered into groups, called regions, pertaining to any of the three classes: Iteration; Selection; Sequence.
JACOB takes into account only two types of regions: cyclic (iteration) and acyclic (selection, sequence).
A single-entry, single-exit property is enforced, that is all the edges entering a region (inedges) must end at the same node inside the region and all the edges leaving a region (outedges) must end on the same node outside the region.
Based on this property, once a region is identified, it can be extracted from the CFG and replaced with a single node.

[code structuring]
The region identification process carried out in the previous step inferred hierarchical information from the CFG, highlighting logical code blocks boundaries. At this point, for example, the graph of an if clause body will be a region by itself, while the body of the else branch will be in a different region. This partitioning allows code generation via a simple substitution process, where every region is replaced with a node containing the equivalent JavaScript code, already wrapped in the appropriate control structure.

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. It is possible to discern these three structures thanks to their characteristic shape, which directly correlates to a specific meaning: The if-then structure implies the body of the if might be executed before reaching the end node. The end might as well be executed directly, skipping the if body. This semantics can be represented by a triangle, where the head is connected to the body, which is then connected to the end, and directly to the end.
The if-else structure implies that either side of the branch has to be executed, and only then the end can be reached. This means that the head must be connected with the two candidate code blocks, which are then connected with the same end node. These connections generate a diamond-shaped graph.
The while structure, as previously mentioned, is characterized by a backedge aiming at the head of the region, which can leave the region. Loops are cyclic regions of the graph.
Much like region identification, the code structuring process “bubbles up” a graph-to-code conversion, starting from the innermost region. The final result is a single node embodying unflattened and structured code for the whole program. The output is valid, goto-free JavaScript code semantically equivalent to the original obfuscated input.

[partial evaluation]
The output of code structuring is guaranteed to be consequential code, which means basic blocks to be executed one after another are sequential, thereby allowing a human reader to follow the execution flow easily. Still, any obfuscation technique other than control flow flattening has not been addressed yet. Data splitting, for example, often accompanies and greatly benefits from control flow flattening: scattering basic blocks through the codebase makes it difficult to deduce the value of a variable at a specific code location when its final value is dynamically generated. Moreover, JavaScript programs heavily rely on strings to perform their tasks, making string splitting actively used.
JACOB involves then a final phase to attempt data reconstruction via partial evaluation, 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.

{ de4js: }
[array decode]
Identifies the different parts of the array trying to reveal the purpose transforming them to the original one. Array decoding involves applying some kind of algorithm or function to an encoded array in order to decrypt or decode the original data or instructions. During deobfuscation, reverse engineers have to identify the decoding algorithm or function used and apply it to the encoded array in order to reveal the original data or instructions.

[restore strings]
Identifies different parts of the string transforming them to the most understandable original one.

The decoder may use various techniques such as pattern recognition, string manipulation functions, and regular expressions to identify and decode the obfuscated strings. Once the strings are decoded, they can be analyzed and understood more easily, making the code clearer and easier to work with.

[restore control flow, indentation, spaces and newlines]
The result is more understandable and it is easier to identify the use of certain parts of the code.

{ de4dot: }
(.NET deobfuscator, not javascript)

Here's a pseudo random list of the things it will do depending on what obfuscator was used to obfuscate an assembly:

Inline methods. Some obfuscators move small parts of a method to another static method and calls it.

- [Decrypt strings] statically or dynamically
- [Decrypt other constants]. Some obfuscators can also encrypt other constants, such as all integers, all doubles, etc.
- [Decrypt methods] statically or dynamically
- [Remove proxy methods]. Many obfuscators replace most/all call instructions with a call to a delegate. This delegate in turn calls the real method.
- [Rename symbols]. Even though most symbols can't be restored, it will rename them to human readable strings. Sometimes, some of the original names can be restored, though.
- [Devirtualize] virtualized code
- [Decrypt resources]. Many obfuscators have an option to encrypt .NET resources.
- [Decrypt embedded files]. Many obfuscators have an option to embed and possibly encrypt/compress other assemblies.
- [Remove] tamper detection code
- [Remove] anti-debug code
- [Control flow deobfuscation]. Many obfuscators modify the IL code so it looks like spaghetti code making it very difficult to understand the code.
- [Restore class fields]. Some obfuscators can move fields from one class to some other obfuscator created class.
- [Convert] a PE exe to a .NET exe. Some obfuscators wrap a .NET assembly inside a Win32 PE so a .NET decompiler can't read the file.
- [Removes] most/all junk classes added by the obfuscator.
- [Fixes some peverify errors]. Many of the obfuscators are buggy and create unverifiable code by mistake.
- [Restore] the types of method parameters and fields.

[Decrypts the encrypted parts of the code] Encrypted strings, constants, methods, resources, embedded files return to the plain and understandable form. This involves identifying the encryption algorithm and key used, and then applying the decryption process to reveal the original content. For strings, constants, and other data embedded in the code, the decryption process typically involves identifying the encryption algorithm, key, and any other necessary parameters (such as IVs for block ciphers) and applying them to decrypt the data back to its original form. For methods and resources that have been obfuscated, it may involve decompiling the obfuscated code back to its original source code form, and then analyzing and understanding the logic and functionality of the code to reverse any obfuscation techniques used.

[remove] proxy methods (Many obfuscators replace most/all call instructions with a call to a delegate. This delegate in turn calls the real method.), tamper detection code, anti-debug code and most/all junk classes to make code more accessible.The process of removing proxy methods may involve identifying the delegate objects used by the obfuscator, understanding how these delegates are invoked, and determining the mapping between the proxy methods and the original methods. By analyzing the code and identifying these relationships, the deobfuscator can effectively remove the proxy methods and restore the original functionality of the code. The tamper detection code needs to be identified and carefully removed to prevent any unintended consequences. This process involves understanding how the tamper detection code works and finding ways to bypass or disable it. Removing anti-debugging code during code deobfuscation involves identifying and neutralizing the various techniques used to detect and thwart debugging and analysis tools. To remove most/all junk classes analyze the codebase to identify classes that are not being used or serving any meaningful purpose and then remove them manually or using a tool to automatically remove them.

{ Code Amaze: }
[Restores indentation, spaces, newlines and control flow]
The result is more understandable and it is easier to identify the use of certain parts of the code.

{ Unminify: }
[Restores indentation, spaces, newlines and control flow]
The result is more understandable and it is easier to identify the use of certain parts of the code.

OBFUSCATION	
Tool	**Methodologies**
open source	

obfuscator.io
https://obfuscator.io/
https://github.com/javascript-obfuscator/javascript-obfuscator
	variables renaming; strings extraction and encryption;
	dead code injection; control flow flattening;
	various code transformations; debug protection;
	disable console output; domain lock.

javascriptobfuscator https://javascriptobfuscator.dev/ (refers to obfuscator.io GitHub) https://github.com/javascript-obfuscator/javascript-obfuscator	remove indentation, spaces, newlines; control flow fattening; adding meaningless code; rename variables and functions; comments obfuscation.
not open source	
ByteHide https://www.bytehide.com/products/shield-obfuscator/javascript	remove indentation, spaces, newlines; control flow fattening; adding meaningless code; rename variables and functions; comments obfuscation.
Code Beautify https://codebeautify.org/javascript-obfuscator#	remove indentation, spaces, newlines; control flow flattening; dead code injection; variables renaming; comments obfuscation.
Clean CSS https://www.cleancss.com/javascript-obfuscate/index.php	replace names; replace numeric constants; replace strings characters with their hex escapes; remove or obfuscate comments; remove spaces, tabs and blank lines; encode part of the code.
LambdaTest https://www.lambdatest.com/free-online-tools/js-obfuscator	renaming variables and functions; adding meaningless code; encrypting strings; replacing function calls with equivalent but less readable code; remove indentation, spaces, newlines; control flow flattening; comments obfuscation.

javascriptobfuscator.com
https://www.javascriptobfuscator.com/Javascript-Obfuscator.aspx
	remove linefeeds and indentations;
	encode strings; move strings;
	name obfuscation; code flow obfuscation; minification and compression;
	code transposition; dead-code insertion;
	replace globals; remove header comment;
	move members; string encryption;
	protect members; flat transform; powerful locking.

Javascript Obfuscator Tool https://developer.wikimint.com/p/javascript-obfuscator.html	adding meaningless code; remove indentation, spaces, newlines; control flow fattening; rename variables and functions; comments obfuscation.
JSDefender https://www.preemptive.com/products/jsdefender/	Being a non-open source obfuscator, the code is not shown and the obfuscation techniques used are not shown.
verimatrix https://www.verimatrix.com/cybersecurity/code-obfuscation/	control flow obfuscation; symbol obfuscation; string obfuscation; arithmetic obfuscation.

<u>Obfuscation tools and methodologies:</u>

{ obfuscator.io: }
[variables renaming]
Through various techniques the variables are renamed to make their use incomprehensible.
Variable renaming involves changing the names of variables in the code to random characters or short, meaningless names. This makes it harder for someone to understand the purpose of each variable and makes the code more difficult to follow.

[strings extraction and encryption]
Identifies strings and encrypts them to make their usage difficult to understand.
String extraction involves moving strings (such as API keys, URLs, or sensitive data) out of the code and storing them separately. This makes it harder for attackers to find and understand these important strings by looking at the code itself.
Encryption involves encrypting strings before storing them in the code. The encrypted strings are decrypted at runtime when they are needed. This makes it difficult for attackers to extract and understand the sensitive information without the decryption key.

[dead code injection]
With this option, random blocks of dead code will be added to the obfuscated code.
During dead code injection, obfuscation tools insert random or irrelevant functions, variables, statements, or comments into the code. These pieces of code have no impact on the program's functionality and are never executed during runtime. They are simply included to confuse and complicate the understanding of the code.

[control flow flattening]
Enables code control flow flattening. Control flow flattening is a structure transformation of the source code that hinders program comprehension.
During control flow flattening, the obfuscation tool will introduce a new set of instructions that determine the flow of execution. These instructions can include loops, conditionals, and jumps that are not present in the original code. Additionally, the obfuscator may introduce fake code paths and dead ends to confuse reverse engineers.

[various code transformations]
Some code part format is modified using different techniques to make it incomprehensible.

[debug protection]
This option makes it almost impossible to use the debugger function of the Developer Tools (both on WebKit-based and Mozilla Firefox).

[disable console output]
Disables the use of console.log, console.info, console.error, console.warn, console.debug, console.exception and console.trace by replacing them with empty functions. This makes the use of the debugger harder.

[domain lock]

Allows to run the obfuscated source code only on specific domains and/or sub-domains. This makes really hard for someone to just copy and paste your source code and run it elsewhere.
If the source code isn't run on the domains specified by this option, the browser will be redirected to a passed to the domainLockRedirectUrl option URL.

{ javascriptobfuscator }
[remove indentation, spaces, newlines]
Results confusing code, in which it is complex to understand the use of the different parts.

[control flow fattening]
Enables code control flow flattening. Control flow flattening is a structure transformation of the source code that hinders program comprehension.
During control flow flattening, the obfuscation tool will introduce a new set of instructions that determine the flow of execution. These instructions can include loops, conditionals, and jumps that are not present in the original code. Additionally, the obfuscator may introduce fake code paths and dead ends to confuse reverse engineers.

[adding meaningless code]
Unuseful code is added to the original code to confuse the real intentions of the code used. Obfuscation tools insert random or irrelevant functions, variables, statements, or comments into the code. These pieces of code have no impact on the program's functionality and are never executed during runtime. They are simply included to confuse and complicate the understanding of the code.

[rename variables and functions]
Through various techniques the variables and functions are renamed to make their use incomprehensible.
Variable renaming involves changing the names of variables in the code to random characters or short, meaningless names. This makes it harder for someone to understand the purpose of each variable and makes the code more difficult to follow.
Function renaming involves changing the names of functions in the code to obscure or arbitrary names. This can also make the code harder to understand, as it may not be immediately clear what each function does based on its name.

[comments obfuscation]
Comments can be very useful to understand the use of a part of the code and so it is important to hide them. Comments may be obfuscated along with the rest of the code, either by removing them entirely or by transforming them into obfuscated text. This can help to further obscure the code and make it even harder to understand.

{ ByteHide: }
[remove indentation, spaces, newlines]
Results confusing code, in which it is complex to understand the use of the different parts.

[control flow fattening]
Enables code control flow flattening. Control flow flattening is a structure transformation of the source code that hinders program comprehension.

During control flow flattening, the obfuscation tool will introduce a new set of instructions that determine the flow of execution. These instructions can include loops, conditionals, and jumps that are not present in the original code. Additionally, the obfuscator may introduce fake code paths and dead ends to confuse reverse engineers.

[adding meaningless code]
Unuseful code is added to the original code to confuse the real intentions of the code used. Obfuscation tools insert random or irrelevant functions, variables, statements, or comments into the code. These pieces of code have no impact on the program's functionality and are never executed during runtime. They are simply included to confuse and complicate the understanding of the code.

[rename variables and functions]
Through various techniques the variables and functions are renamed to make their use incomprehensible.
Variable renaming involves changing the names of variables in the code to random characters or short, meaningless names. This makes it harder for someone to understand the purpose of each variable and makes the code more difficult to follow.
Function renaming involves changing the names of functions in the code to obscure or arbitrary names. This can also make the code harder to understand, as it may not be immediately clear what each function does based on its name.

[comments obfuscation]
Comments can be very useful to understand the use of a part of the code and so it is important to hide them. Comments may be obfuscated along with the rest of the code, either by removing them entirely or by transforming them into obfuscated text. This can help to further obscure the code and make it even harder to understand.

{ Code Beautify: }
[remove indentation, spaces, newlines]
Results confusing code, in which it is complex to understand the use of the different parts.

[control flow flattening]
Enables code control flow flattening. Control flow flattening is a structure transformation of the source code that hinders program comprehension.
During control flow flattening, the obfuscation tool will introduce a new set of instructions that determine the flow of execution. These instructions can include loops, conditionals, and jumps that are not present in the original code. Additionally, the obfuscator may introduce fake code paths and dead ends to confuse reverse engineers.

[dead code injection]
Unuseful code is added to the original code to confuse the real intentions of the code used. Obfuscation tools insert random or irrelevant functions, variables, statements, or comments into the code. These pieces of code have no impact on the program's functionality and are never executed during runtime. They are simply included to confuse and complicate the understanding of the code.

[variables renaming]

Through various techniques the variables are renamed to make their use incomprehensible. Variable renaming involves changing the names of variables in the code to random characters or short, meaningless names. This makes it harder for someone to understand the purpose of each variable and makes the code more difficult to follow.

[comments obfuscation]
Comments can be very useful to understand the use of a part of the code and so it is important to hide them. Comments may be obfuscated along with the rest of the code, either by removing them entirely or by transforming them into obfuscated text. This can help to further obscure the code and make it even harder to understand.

{ Clean CSS: }
[replaces symbol names] with non-meaningfull ones (eg hello_moto will be converted to 1cd5dg4g1gf). A tool or script can be used to automatically rename the symbols in the code. This tool will generate random or nonsensical names for each symbol based on a specific naming scheme. For example, a variable named "counter" might be renamed to something like "a4531s" and a function named "calculateTotal" might be renamed to "x9b5s". These meaningless names make it harder for someone to understand the purpose of the symbols and follow the code logic.

[replaces numeric constants] with expressions (eg 232 will be converted to 0x29b9+2011-0x2d25). A technique that involves transforming static numeric values in the code into dynamic expressions that compute the same value at runtime. For example, instead of having a constant value like 5 in the code, it may be replaced with an expression like 2 + 3 so that when the code is executed, the value of 5 is computed at runtime. This technique can be implemented using various methods such as using arithmetic operations, bitwise operations, or algorithmic calculations to generate the same result as the original numeric constants. By doing this, the code becomes more complex and less transparent, making it more difficult for someone to analyze and understand the code.

[replaces characters in strings] with their hex escapes (eg string "noob" will be converted to "`vcD"). This is done by converting the characters to their hexadecimal representation and adding a backslash and "x" before the hex value. For example, the string "Hello, World!" could be obfuscated by replacing it with "\x48\x65\x6c\x6c\x6f\x2c\x20\x57\x6f\x72\x6c\x64\x21". This representation is harder for a human to interpret at a glance, making the code more challenging to reverse engineer.

[removes comments] completely (if selected, otherwise obfuscates comments). Comments can be very useful to understand the use of a part of the code and so it is important to hide them. Comments may be obfuscated along with the rest of the code, either by removing them entirely or by transforming them into obfuscated text. This can help to further obscure the code and make it even harder to understand.

[removes spaces and tabs and blank lines] in the code. Results confusing code, in which it is complex to understand the use of the different parts.
[joins all the lines of code together]
[encodes] the previous stages using advanced algorithms.

{ LambdaTest: }
[renaming variables and functions]
Through various techniques the variables and functions are renamed to make their use incomprehensible.
Variable renaming involves changing the names of variables in the code to random characters or short, meaningless names. This makes it harder for someone to understand the purpose of each variable and makes the code more difficult to follow.
Function renaming involves changing the names of functions in the code to obscure or arbitrary names. This can also make the code harder to understand, as it may not be immediately clear what each function does based on its name.

[adding meaningless code]
Unuseful code is added to the original code to confuse the real intentions of the code used. Obfuscation tools insert random or irrelevant functions, variables, statements, or comments into the code. These pieces of code have no impact on the program's functionality and are never executed during runtime. They are simply included to confuse and complicate the understanding of the code.

[encrypting strings]
Encrypted strings are incomprehensible and their usage is hidden. Encryption involves encrypting strings before storing them in the code. The encrypted strings are decrypted at runtime when they are needed.

[replacing function calls with equivalent but less readable code]
Functions format is transformed to make it incomprehensible and to hide the usage.
This can be achieved by taking the functionality of the original function and implementing it using different constructs or methods.
For example, a simple function call like:
```
result = addNumbers(x, y);
```
Could be replaced with:
```
result = x + y;
```

[remove indentation, spaces, newlines]
Results confusing code, in which it is complex to understand the use of the different parts.

[control flow flattening]
Enables code control flow flattening. Control flow flattening is a structure transformation of the source code that hinders program comprehension.
During control flow flattening, the obfuscation tool will introduce a new set of instructions that determine the flow of execution. These instructions can include loops, conditionals, and jumps that are not present in the original code. Additionally, the obfuscator may introduce fake code paths and dead ends to confuse reverse engineers.

[comments obfuscation]
Comments can be very useful to understand the use of a part of the code and so it is important to hide them. Comments may be obfuscated along with the rest of the code, either by removing them entirely or by transforming them into obfuscated text. This can help to further obscure the code and make it even harder to understand.

{ javascriptobfuscator.com: }
[remove linefeeds and indentations]
Removing them makes the code messy and confusing and therefore incomprehensible.

[encode strings]
The encoded strings are incomprehensible and their usage is hidden. Strings can be encoded using techniques like base64 encoding or URL encoding, making them harder to recognize.

[move strings]
Changing strings position in the code increases the misunderstanding of their use. This involves hiding or rearranging the sequences of characters that represent string literals within the code to make it harder for someone to understand or reverse engineer the code. This can be achieved through techniques such as encrypting the strings, splitting them into multiple parts, or storing them in different locations within the code. By moving strings around in this way, it becomes more challenging for someone to easily extract and identify the sensitive information or logic embedded in the strings.

[name obfuscation]
The process of replacing the identifiers with meaningless sequences of characters. Of course, a name obfuscator must process the entire application to ensure consistency of name changes across all files.

[code flow obfuscation]
It is the process to change the control flow in a software application. The changed control flow must lead to the same results as the initial one, but produces spaghetti logic that can be very difficult for a cracker to analyze. Obfuscation tool will introduce a new set of instructions that determine the flow of execution. These instructions can include loops, conditionals, and jumps that are not present in the original code. Additionally, the obfuscator may introduce fake code paths and dead ends to confuse reverse engineers.

[minification and compression]
Reduces the size of JavaScript files and makes them more efficient, helping applications to load faster and reducing bandwidth consumption.

[code transposition]
Rearranging the instructions of a piece of code without changing its behavior. During code transposition, sections of the code are rearranged or shuffled in a random or specific order, making it harder for someone to understand the logic and flow of the program. This can involve moving statements or functions around, changing the order of loops or conditions, or altering the flow of control structures.

[dead-code insertion]
Inserts code that is executed when the program is run but does not affect the semantics of the program, making any disassembled code more difficult to analyze. Obfuscation tools insert random or irrelevant functions, variables, statements, or comments into the code. These pieces of code have no impact on the program's functionality and are never executed

during runtime. They are simply included to confuse and complicate the understanding of the code.

[replace globals]
Global variables are replaced with names inconsistent with their use.

[remove header comment]
Header comments are removed to not reveal what is the purpose of that piece of code.

[move and protect members]
Code members are moved and transformed to confuse and hide their purpose.

[string encryption]
Crackers will frequently search for specific strings in your code to locate strategic logic. String Encryption makes this much more difficult to do, because the attacker's search will come up empty. The original string is nowhere to be found in the code. Only its encrypted version is present.

[flat transform]
Makes the code confused and incomprehensible. During the flat transform process, the structure of the program is modified to remove nested loops, conditional statements, and other control flow structures that can make the code more complex and harder to analyze. This can involve converting nested loops into a series of linear loops, replacing conditional statements with jump instructions, or restructuring the code in other ways to reduce branching and make the code more linear.

[powerful locking]
Lock your code based on IP address, domain name and much more. Create trial versions with time limiting and many other features.

{ JavaScript Obfuscator Tool: }
[adding meaningless code]
Unuseful code is added to the original code to confuse the real intentions of the code used. Obfuscation tools insert random or irrelevant functions, variables, statements, or comments into the code. These pieces of code have no impact on the program's functionality and are never executed during runtime. They are simply included to confuse and complicate the understanding of the code.

[remove indentation, spaces, newlines]
Results confusing code, in which it is complex to understand the use of the different parts.

[control flow fattening]
Enables code control flow flattening. Control flow flattening is a structure transformation of the source code that hinders program comprehension. During control flow flattening, the obfuscation tool will introduce a new set of instructions that determine the flow of execution. These instructions can include loops, conditionals, and jumps that are not present in the original code. Additionally, the obfuscator may introduce fake code paths and dead ends to confuse reverse engineers.

[rename variables and functions]
Through various techniques the variables and functions are renamed to make their use incomprehensible. Variable renaming involves changing the names of variables in the code to random characters or short, meaningless names. This makes it harder for someone to understand the purpose of each variable and makes the code more difficult to follow. Function renaming involves changing the names of functions in the code to obscure or arbitrary names. This can also make the code harder to understand, as it may not be immediately clear what each function does based on its name.

[comments obfuscation]
Comments can be very useful to understand the use of a part of the code and so it is important to hide them. Comments may be obfuscated along with the rest of the code, either by removing them entirely or by transforming them into obfuscated text. This can help to further obscure the code and make it even harder to understand.

{ JSDefender: }
Being a non-open source obfuscator, the code is not shown and the obfuscation techniques used are not shown.

{verimatrix}
[control flow obfuscation]
Rather than simply renaming functions, methods, and classes, control flow goes further to obfuscate your app logic and deter hackers. During control flow flattening, the obfuscation tool will introduce a new set of instructions that determine the flow of execution. These instructions can include loops, conditionals, and jumps that are not present in the original code. Additionally, the obfuscator may introduce fake code paths and dead ends to confuse reverse engineers.

[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. This involves changing the way arithmetic operations are performed in a code in order to obscure the original logic and make it harder for someone to understand the code. This can include changing the order of operations, introducing unnecessary calculations, or using different data types. For example, a simple arithmetic operation like "x = a + b * c" could be obfuscated by rewriting it as "x = (a * b) + c + (b - a)".