|  |
| --- |
| TestVuln  Version not provided  Code analysis |

|  |
| --- |
| **By: Administrator**  **2022-10-22** |

# Content

[Content 1](#_Toc97156258)

[Introduction 2](#_Toc97156259)

[Configuration 2](#_Toc97156260)

[Synthesis 3](#_Toc97156261)

[Analysis Status 3](#_Toc97156262)

[Quality gate status 3](#_Toc97156263)

[Metrics 3](#_Toc97156264)

[Tests 3](#_Toc97156265)

[Detailed technical debt 3](#_Toc97156266)

[Metrics Range 5](#_Toc97156267)

[Volume 5](#_Toc97156268)

[Issues 6](#_Toc97156269)

[Charts 6](#_Toc97156270)

[Issues count by severity and type 8](#_Toc97156271)

[Issues List 8](#_Toc97156272)

[Security Hotspots 9](#_Toc97156273)

[Security hotspots count by category and priority 9](#_Toc97156274)

[Security hotspots List 9](#_Toc97156275)

# Introduction

This document contains results of the code analysis of TestVuln.

# Configuration

* Quality Profiles
  + Names: Sonar way [CSS]; Sonar way [Java]; Sonar way [JavaScript]; Sonar way [JSP]; Sonar way [HTML]; Sonar way [XML];
  + Files: AYP5taVi9kzJocWXodGE.json; AYP5tamD9kzJocWXod9O.json; AYP5tabp9kzJocWXodT5.json; AYP5taXw9kzJocWXodIC.json; AYP5tao\_9kzJocWXoeFp.json; AYP5taqg9kzJocWXoeG5.json;
* Quality Gate
  + Name: Sonar way
  + File: Sonar way.xml

# Synthesis

## Analysis Status

|  |  |  |  |
| --- | --- | --- | --- |
| Reliability | Security | Security Review | Maintainability |
| E.png | **E.png** | **E.png** | **A.png** |

## Quality gate status

|  |  |
| --- | --- |
| Quality Gate Status | **ERROR.png** |

|  |  |
| --- | --- |
| Metric | Value |
| Reliability Rating on New Code | ERROR (E is worse than A) |
| Security Rating on New Code | ERROR (D is worse than A) |
| Maintainability Rating on New Code | OK |
| Security Hotspots Reviewed on New Code | ERROR (0.0% is less than 100%) |

## Metrics

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Coverage | Duplication | Comment  density | Median number of lines of code per file | Adherence to coding standard |
| 0.0 % | **36.2 %** | **6.5 %** | **106.0** | **96.2 %** |

## Tests

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Total | Success Rate | Skipped | Errors | Failures |
| 0 | **0 %** | **0** | **0** | **0** |

## Detailed technical debt

|  |  |  |  |
| --- | --- | --- | --- |
| Reliability | Security | Maintainability | Total |
| 16d 4h 11min | 0d 1h 35min | 1084d 6h 52min | 1101d 4h 38min |

## Metrics Range

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Cyclomatic  Complexity | Cognitive  Complexity | Lines of code per file | Comment  density (%) | Coverage | Duplication (%) |
| Min | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| Max | 86627.0 | 172982.0 | 439383.0 | 100.0 | 0.0 | 100.0 |

## Volume

|  |  |
| --- | --- |
| Language | Number |
| CSS | 64143 |
| Java | 98621 |
| JavaScript | 346438 |
| JSP | 17405 |
| HTML | 45 |
| XML | 4701 |
| Total | 531353 |

# Issues

## Charts

## Issues count by severity and type

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Type / Severity | INFO | MINOR | MAJOR | CRITICAL | BLOCKER |
| BUG | 0 | 10 | 62 | 1 | 6 |
| VULNERABILITY | 0 | 0 | 0 | 0 | 0 |
| CODE\_SMELL | 29 | 266 | 887 | 8728 | 11 |

## Issues List

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Description | Type | Severity | Number |
| CSS properties should be valid | The W3C specifications define the valid CSS properties. Only the official and browser-specific properties should be used to get the expected impact in the final rendering. This rule ignores: $sass, @less, and var(--custom-property) variable syntaxes. vendor-prefixed properties (e.g., -moz-align-self, -webkit-align-self). Noncompliant Code Example a { colour: blue; /\* Noncompliant; colour is not part of the specifications \*/ } Compliant Solution a { color: blue; } | BUG | BLOCKER | 6 |
| Function calls should not pass extra arguments | You can easily call a JavaScript function with more arguments than the function needs, but the extra arguments will be just ignored by function execution. Noncompliant Code Example function say(a, b) { print(a + " " + b); } say("hello", "world", "!"); // Noncompliant; last argument is not used Exceptions No issue is reported when arguments is used in the body of the function being called. function doSomething(a, b) { compute(arguments); } doSomething(1, 2, 3) // Compliant See MITRE, CWE-628 - Function Call with Incorrectly Specified Arguments | BUG | CRITICAL | 1 |
| Font declarations should contain at least one generic font family | If none of the font names defined in a font or font-family declaration are available on the browser of the user, the browser will display the text using its default font. It’s recommended to always define a generic font family for each declaration of font or font-family to get a less degraded situation than relying on the default browser font. All browsers should implement a list of generic font matching these families: Serif, Sans-serif, cursive, fantasy, Monospace. Noncompliant Code Example a { font-family: Helvetica, Arial, Verdana, Tahoma; /\* Noncompliant; there is no generic font family in the list \*/ } Compliant Solution a { font-family: Helvetica, Arial, Verdana, Tahoma, sans-serif; } See CSS Specification - Generic font families | BUG | MAJOR | 16 |
| Properties should not be duplicated | CSS allows duplicate property names but only the last instance of a duplicated name determines the actual value that will be used for it. Therefore, changing values of other occurrences of a duplicated name will have no effect and may cause misunderstandings and bugs. This rule ignores $sass, @less, and var(--custom-property) variable syntaxes. Noncompliant Code Example a { color: pink; background: orange; color: orange } Compliant Solution a { color: pink; background: orange } | BUG | MAJOR | 11 |
| All code should be reachable | Jump statements (return, break and continue) and throw expressions move control flow out of the current code block. So any statements that come after a jump are dead code. Noncompliant Code Example function fun(a) { var i = 10; return i + a; i++; // Noncompliant; this is never executed } Compliant Solution function fun(int a) { var i = 10; return i + a; } See MITRE, CWE-561 - Dead Code | BUG | MAJOR | 4 |
| Objects should not be created to be dropped immediately without being used | There is no good reason to create a new object to not do anything with it. Most of the time, this is due to a missing piece of code and so could lead to an unexpected behavior in production. If it was done on purpose because the constructor has side-effects, then that side-effect code should be moved into a separate method and called directly. Noncompliant Code Example new MyConstructor(); // Non-Compliant Compliant Solution var something = new MyConstructor(); // Compliant Exceptions Immediately dropped new objects inside try-statements are ignored. try { new MyConstructor(); } catch (e) { /\* ... \*/ } | BUG | MAJOR | 4 |
| Special identifiers should not be bound or assigned | JavaScript has special identifiers that, while not reserved, still should not be used as identifiers. They include: eval - evaluates a string as JavaScript code arguments - used to access function arguments through indexed properties. undefined - returned for values and properties that have not yet been assigned NaN - Not a Number; returned when math functions fail. Infinity - when a number exceeds the upper limit of the floating point numbers These words should not be bound or assigned, because doing so would overwrite the original definitions of these identifiers. What’s more, assigning or binding some of these names will generate an error in JavaScript strict mode code. Noncompliant Code Example eval = 17; // Noncompliant arguments++; // Noncompliant ++eval; // Noncompliant var obj = { set p(arguments) { } }; // Noncompliant var eval; // Noncompliant try { } catch (arguments) { } // Noncompliant function x(eval) { } // Noncompliant function arguments() { } // Noncompliant var y = function eval() { }; // Noncompliant var f = new Function("arguments", "return 17;"); // Noncompliant function fun() { if (arguments.length == 0) { // Compliant // do something } } Compliant Solution result = 17; args++; ++result; var obj = { set p(arg) { } }; var result; try { } catch (args) { } function x(arg) { } function args() { } var y = function fun() { }; var f = new Function("args", "return 17;"); function fun() { if (arguments.length == 0) { // do something } } | BUG | MAJOR | 2 |
| Return values from functions without side effects should not be ignored | When the call to a function doesn’t have any side effects, what is the point of making the call if the results are ignored? In such case, either the function call is useless and should be dropped or the source code doesn’t behave as expected. To prevent generating any false-positives, this rule triggers an issues only on a predefined list of known objects &amp; functions. Noncompliant Code Example 'hello'.lastIndexOf('e'); // Noncompliant Compliant Solution let char = 'hello'.lastIndexOf('e'); | BUG | MAJOR | 5 |
| Alternatives in regular expressions should be grouped when used with anchors | In regular expressions, anchors ^ and $ have higher precedence than the | operator. So in a regular expression like ^alt1|alt2|alt3$, alt1 would be anchored to the beginning, alt3 to the end and alt2 wouldn’t be anchored at all. Usually the intended behavior is that all alternatives are anchored at both ends. To achieve this, a non-capturing group should be used around the alternatives. In cases where it is intended that the anchors only apply to one alternative each, adding (non-capturing) groups around the anchors and the parts that they apply to will make it explicit which parts are anchored and avoid readers misunderstanding the precedence or changing it because they mistakenly assume the precedence was not intended. Noncompliant Code Example ^a|b|c$ Compliant Solution ^(?:a|b|c)$ or ^a$|^b$|^c$ or, if you do want the anchors to only apply to a and c respectively: (?:^a)|b|(?:c$) | BUG | MAJOR | 4 |
| Unicode Grapheme Clusters should be avoided inside regex character classes | When placing Unicode Grapheme Clusters (characters which require to be encoded in multiple Code Points) inside a character class of a regular expression, this will likely lead to unintended behavior. For instance, the grapheme cluster c̈ requires two code points: one for 'c', followed by one for the umlaut modifier '\u{0308}'. If placed within a character class, such as [c̈], the regex will consider the character class being the enumeration [c\u{0308}] instead. It will, therefore, match every 'c' and every umlaut that isn’t expressed as a single codepoint, which is extremely unlikely to be the intended behavior. This rule raises an issue every time Unicode Grapheme Clusters are used within a character class of a regular expression. Noncompliant Code Example "cc̈d̈d".replace(/[c̈d̈]/g, "X"); // result is "XXXXXX" and not expected "cXXd" Compliant Solution "cc̈d̈d".replace(/c̈|d̈/g, "X"); // result is "cXXd" | BUG | MAJOR | 1 |
| Regular expressions should not contain control characters | Entries in the ASCII table below code 32 are known as control characters or non-printing characters. As they are not common in JavaScript strings, using these invisible characters in regular expressions is most likely a mistake. Noncompliant Code Example const pattern1 = /\x1a/; const pattern2 = new RegExp('\x1a'); Compliant Solution const pattern1 = /\x20/; const pattern2 = new RegExp('\x20'); | BUG | MAJOR | 11 |
| Non-empty statements should change control flow or have at least one side-effect | Any statement (other than a null statement, which means a statement containing only a semicolon ;) which has no side effect and does not result in a change of control flow will normally indicate a programming error, and therefore should be refactored. Noncompliant Code Example a == 1; // Noncompliant; was assignment intended? var msg = "Hello, " "World!"; // Noncompliant; have we forgotten '+' operator on previous line? See MITRE, CWE-482 - Comparing instead of Assigning | BUG | MAJOR | 4 |
| Function parameters, caught exceptions and foreach variables' initial values should not be ignored | While it is technically correct to assign to parameters from within function bodies, it reduces code readability because developers won’t be able to tell whether the original parameter or some temporary variable is being accessed without going through the whole function. Moreover, some developers might also expect assignments of function parameters to be visible to callers, which is not the case, and this lack of visibility could confuse them. Instead, all parameters, caught exceptions, and foreach parameters should be treated as constants. Noncompliant Code Example function MyClass(name, strings) { name = foo; // Noncompliant for (var str of strings) { str = ""; // Noncompliant } } Exceptions There is a common pattern in JavaScript to overwrite certain parameters depending on other parameters that are optional. For example, a callback is by convention always passed in the last position. If a parameter in a previous position was not passed, the callback will be passed in its position instead. Therefore, the rule ignores parameter reassignments that are inside an if statement block. function f(param, optionalParam, cb) { if (typeof optionalParam === 'function') { cb = optionalParam; optionalParam = {}; } } | BUG | MINOR | 5 |
| "delete" should be used only with object properties | The semantics of the delete operator are a bit tricky, and it can only be reliably used to remove properties from objects. Pass anything else to it, and you may or may not get the desired result. Noncompliant Code Example var x = 1; delete x; // Noncompliant function foo(){ .. } delete foo; // Noncompliant Compliant Solution var obj = { x:1, foo: function(){ ... } }; delete obj.x; delete obj.foo; | BUG | MINOR | 1 |
| Repeated patterns in regular expressions should not match the empty string | A regex should never include a repetitive pattern whose body would match the empty string. This is usually a sign that a part of the regex is redundant or does not do what the author intended. Noncompliant Code Example /(?:)\*/ // same as the empty regex, the '\*' accomplishes nothing /(?:|x)\*/ // same as the empty regex, the alternative has no effect /(?:x|)\*/ // same as 'x\*', the empty alternative has no effect /(?:x\*|y\*)\*/ // same as 'x\*', the first alternative would always match, y\* is never tried /(?:x?)\*/ // same as 'x\*' /(?:x?)+/ // same as 'x\*' Compliant Solution /x\*/ | BUG | MINOR | 4 |
| Switch cases should end with an unconditional "break" statement | When the execution is not explicitly terminated at the end of a switch case, it continues to execute the statements of the following case. While this is sometimes intentional, it often is a mistake which leads to unexpected behavior. Noncompliant Code Example switch (myVariable) { case 1: foo(); break; case 2: // Both 'doSomething()' and 'doSomethingElse()' will be executed. Is it on purpose ? doSomething(); default: doSomethingElse(); break; } Compliant Solution switch (myVariable) { case 1: foo(); break; case 2: doSomething(); break; default: doSomethingElse(); break; } Exceptions This rule is relaxed in the following cases: switch (myVariable) { case 0: // Empty case used to specify the same behavior for a group of cases. case 1: doSomething(); break; case 2: // Use of return statement return; case 3: // Ends with comment when fall-through is intentional console.log("this case falls through") // fall through case 4: // Use of throw statement throw new IllegalStateException(); case 5: // Use of continue statement continue; default: // For the last case, use of break statement is optional doSomethingElse(); } See MITRE, CWE-484 - Omitted Break Statement in Switch | CODE\_SMELL | BLOCKER | 8 |
| Function returns should not be invariant | When a function is designed to return an invariant value, it may be poor design, but it shouldn’t adversely affect the outcome of your program. However, when it happens on all paths through the logic, it is likely a mistake. This rule raises an issue when a function contains several return statements that all return the same value. Noncompliant Code Example function foo(a) { // Noncompliant let b = 12; if (a) { return b; } return b; } | CODE\_SMELL | BLOCKER | 3 |
| Functions should not be empty | There are several reasons for a function not to have a function body: It is an unintentional omission, and should be fixed to prevent an unexpected behavior in production. It is not yet, or never will be, supported. In this case an exception should be thrown in languages where that mechanism is available. The method is an intentionally-blank override. In this case a nested comment should explain the reason for the blank override. Noncompliant Code Example function foo() { } var foo = () =&gt; {}; Compliant Solution function foo() { // This is intentional } var foo = () =&gt; { do\_something(); }; Exceptions This rule does not apply to function expressions and arrow functions as they can denote default values. static defaultProps = { listStyle: () =&gt; {} }; The rule allows for empty functions with a name starting with the prefix on like onClick. function onClick() { } | CODE\_SMELL | CRITICAL | 35 |
| Loop counters should not be assigned to from within the loop body | Loop counters should not be modified in the body of the loop. However other loop control variables representing logical values may be modified in the loop, for example a flag to indicate that something has been completed, which is then tested in the for statement. Noncompliant Code Example var names = [ "Jack", "Jim", "", "John" ]; for (var i = 0; i &lt; names.length; i++) { if (!names[i]) { i = names.length; // Non-Compliant } else { console.log(names[i]); } } Compliant Solution var names = [ "Jack", "Jim", "", "John" ]; for (var name of names) { if (!name) { break; // Compliant } else { console.log(name); } } | CODE\_SMELL | CRITICAL | 31 |
| Variables should be declared with "let" or "const" | ECMAScript 2015 introduced the let and const keywords for block-scope variable declaration. Using const creates a read-only (constant) variable. The distinction between the variable types created by var and by let is significant, and a switch to let will help alleviate many of the variable scope issues which have caused confusion in the past. Because these new keywords create more precise variable types, they are preferred in environments that support ECMAScript 2015. However, some refactoring may be required by the switch from var to let, and you should be aware that they raise SyntaxErrors in pre-ECMAScript 2015 environments. This rule raises an issue when var is used instead of const or let. Noncompliant Code Example var color = "blue"; var size = 4; Compliant Solution const color = "blue"; let size = 4; | CODE\_SMELL | CRITICAL | 8371 |
| Cognitive Complexity of functions should not be too high | Cognitive Complexity is a measure of how hard the control flow of a function is to understand. Functions with high Cognitive Complexity will be difficult to maintain. See Cognitive Complexity | CODE\_SMELL | CRITICAL | 289 |
| Equality operators should not be used in "for" loop termination conditions | Testing for loop termination using an equality operator (== and !=) is dangerous, because it could set up an infinite loop. Using a broader relational operator instead casts a wider net, and makes it harder (but not impossible) to accidentally write an infinite loop. Noncompliant Code Example for (var i = 1; i != 10; i += 2) // Noncompliant. Infinite; i goes from 9 straight to 11. { //... } Compliant Solution for (var i = 1; i &lt;= 10; i += 2) // Compliant { //... } Exceptions Equality operators are ignored if the loop counter is not modified within the body of the loop and either: starts below the ending value and is incremented by 1 on each iteration. starts above the ending value and is decremented by 1 on each iteration. Equality operators are also ignored when the test is against null. for (var i = 0; arr[i] != null; i++) { // ... } for (var i = 0; (item = arr[i]) != null; i++) { // ... } See MITRE, CWE-835 - Loop with Unreachable Exit Condition ('Infinite Loop') | CODE\_SMELL | CRITICAL | 2 |
| Track uses of "TODO" tags | TODO tags are commonly used to mark places where some more code is required, but which the developer wants to implement later. Sometimes the developer will not have the time or will simply forget to get back to that tag. This rule is meant to track those tags and to ensure that they do not go unnoticed. Noncompliant Code Example function doSomething() { // TODO } See MITRE, CWE-546 - Suspicious Comment | CODE\_SMELL | INFO | 29 |
| Empty blocks should be removed | Leftover empty blocks are usually introduced by mistake. They are useless and prevent readability of the code. They should be removed or completed with real code. Noncompliant Code Example a { } Compliant Solution a { color: pink; } | CODE\_SMELL | MAJOR | 9 |
| Selectors should not be duplicated | Duplication of selectors might indicate a copy-paste mistake. The rule detects the following kinds of duplications: within a list of selectors in a single rule set for duplicated selectors in different rule sets within a single stylesheet. Noncompliant Code Example .foo, .bar, .foo { ... } /\* Noncompliant \*/ .class1 { ... } .class1 { ... } /\* Noncompliant \*/ Compliant Solution .foo, .bar { ... } .class1 { ... } .class2 { ... } | CODE\_SMELL | MAJOR | 17 |
| CSS files should not be empty | This rule raises an issue when a CSS file is empty (ie: containing only spaces). | CODE\_SMELL | MAJOR | 3 |
| Functions should not have too many parameters | A long parameter list can indicate that a new structure should be created to wrap the numerous parameters or that the function is doing too many things. Noncompliant Code Example With a maximum number of 4 parameters: function doSomething(param1, param2, param3, param4, param5) { ... } Compliant Solution function doSomething(param1, param2, param3, param4) { ... } Exceptions The rule ignores constructors where parameters are all parameter properties: class C { constructor( private param1: number, private param2: boolean, private param3: string, private param4: string[], private param5: number | string ) {} } The rule also ignores Angular component constructors: import { Component } from '@angular/core'; @Component({/\* ... \*/}) class Component { constructor(p1, p2, p3, p4, p5) {} } | CODE\_SMELL | MAJOR | 1 |
| Nested blocks of code should not be left empty | Most of the time a block of code is empty when a piece of code is really missing. So such empty block must be either filled or removed. Noncompliant Code Example for (var i = 0; i &lt; length; i++) {} // Empty on purpose or missing piece of code ? Exceptions When a block contains a comment, this block is not considered to be empty. Moreover catch blocks are ignored. | CODE\_SMELL | MAJOR | 4 |
| Labels should not be used | Labels are not commonly used, and many developers do not understand how they work. Moreover, their usage makes the control flow harder to follow, which reduces the code’s readability. Noncompliant Code Example myLabel: { let x = doSomething(); if (x &gt; 0) { break myLabel; } doSomethingElse(); } Compliant Solution let x = doSomething(); if (x &lt;= 0) { doSomethingElse(); } &nbsp; | CODE\_SMELL | MAJOR | 3 |
| Assignments should not be made from within sub-expressions | Assignments within sub-expressions are hard to spot and therefore make the code less readable. Ideally, sub-expressions should not have side-effects. Moreover, using chained assignment in declarations is also dangerous because one may accidentally create global variables, e.g. in let x = y = 1;, if y is not declared, it will be hoisted as global. Noncompliant Code Example if (val = value() &amp;&amp; check()) { // Noncompliant // ... } Compliant Solution val = value(); if (val &amp;&amp; check()) { // ... } Exceptions The rule does not raise issues for the following patterns: chained assignments: a = b = c = 0; relational assignments: (a = 0) != b sequential assignments: a = 0, b = 1, c = 2 assignments in lambda body: () =&gt; a = 0 conditional assignment idiom: a || (a = 0) assignments in (do-)while conditions: while (a = 0); See MITRE, CWE-481 - Assigning instead of Comparing | CODE\_SMELL | MAJOR | 46 |
| Sections of code should not be commented out | Programmers should not comment out code as it bloats programs and reduces readability. Unused code should be deleted and can be retrieved from source control history if required. | CODE\_SMELL | MAJOR | 3 |
| Unused assignments should be removed | A dead store happens when a local variable is assigned a value that is not read by any subsequent instruction. Calculating or retrieving a value only to then overwrite it or throw it away, could indicate a serious error in the code. Even if it’s not an error, it is at best a waste of resources. Therefore all calculated values should be used. Noncompliant Code Example i = a + b; // Noncompliant; calculation result not used before value is overwritten i = compute(); Compliant Solution i = a + b; i += compute(); Exceptions This rule ignores initializations to -1, 0, 1, undefined, [], {}, true, false and "". Variables that start with an underscore (e.g. '\_unused') are ignored. Assignment of null is ignored because it is sometimes used to help garbage collection Increment and decrement expressions are ignored because they are often used idiomatically instead of x+1 This rule also ignores variables declared with object destructuring using rest syntax (used to exclude some properties from object): let {a, b, ...rest} = obj; // 'a' and 'b' are ok doSomething(rest); let [x1, x2, x3] = arr; // but 'x1' is noncompliant, as omitting syntax can be used: "let [, x2, x3] = arr;" doSomething(x2, x3); See MITRE, CWE-563 - Assignment to Variable without Use ('Unused Variable') | CODE\_SMELL | MAJOR | 91 |
| Two branches in a conditional structure should not have exactly the same implementation |  | CODE\_SMELL | MAJOR | 3 |
| Parameters should be passed in the correct order | When the names of arguments in a function call match the names of the function parameters, it contributes to clearer, more readable code. However, when the names match, but are passed in a different order than the function parameters, it indicates a mistake in the parameter order which will likely lead to unexpected results. Noncompliant Code Example function divide(divisor, dividend) { return divisor/dividend; } function doTheThing() { var divisor = 15; var dividend = 5; var result = divide(dividend, divisor); // Noncompliant; operation succeeds, but result is unexpected //... } Compliant Solution function divide(divisor, dividend) { return divisor/dividend; } function doTheThing() { var divisor = 15; var dividend = 5; var result = divide(divisor, dividend); //... } Exceptions Swapped arguments that are compared beforehand in an enclosing if-statement are ignored: function divide(divisor, dividend) { return divisor/dividend; } function doTheThing() { var divisor = 15; var dividend = 5; if (divisor &gt; dividend) { var result = divide(dividend, divisor); //... } } | CODE\_SMELL | MAJOR | 5 |
| Variables should be used in the blocks where they are declared |  | CODE\_SMELL | MAJOR | 210 |
| Boolean expressions should not be gratuitous | If a boolean expression doesn’t change the evaluation of the condition, then it is entirely unnecessary, and can be removed. If it is gratuitous because it does not match the programmer’s intent, then it’s a bug and the expression should be fixed. Noncompliant Code Example if (a) { if (a) { // Noncompliant doSomething(); } } Compliant Solution if (a) { if (b) { doSomething(); } } // or if (a) { doSomething(); } See MITRE, CWE-571 - Expression is Always True MITRE, CWE-570 - Expression is Always False | CODE\_SMELL | MAJOR | 1 |
| Multiline blocks should be enclosed in curly braces | Curly braces can be omitted from a one-line block, such as with an if statement or for loop, but doing so can be misleading and induce bugs. This rule raises an issue when the whitespacing of the lines after a one line block indicates an intent to include those lines in the block, but the omission of curly braces means the lines will be unconditionally executed once. Note that this rule considers tab characters to be equivalent to 1 space. If you mix spaces and tabs you will sometimes see issues in code which looks fine in your editor but is confusing when you change the size of tabs. Noncompliant Code Example if (condition) firstActionInBlock(); secondAction(); // Noncompliant; executed unconditionally thirdAction(); if (condition) firstActionInBlock(); secondAction(); // Noncompliant; secondAction executed unconditionally if (condition) firstActionInBlock(); // Noncompliant secondAction(); // Executed unconditionally if (condition); secondAction(); // Noncompliant; secondAction executed unconditionally let str = undefined; for (let i = 0; i &lt; array.length; i++) str = array[i]; doTheThing(str); // Noncompliant; executed only on last array element Compliant Solution if (condition) { firstActionInBlock(); secondAction(); } thirdAction(); let str = undefined; for (let i = 0; i &lt; array.length; i++) { str = array[i]; doTheThing(str); } See MITRE, CWE-483 - Incorrect Block Delimitation | CODE\_SMELL | MAJOR | 3 |
| Variables and functions should not be redeclared | This rule checks that a declaration doesn’t use a name that is already in use. Indeed, it is possible to use the same symbol multiple times as either a variable or a function, but doing so is likely to confuse maintainers. Further it’s possible that such reassignments are made in error, with the developer not realizing that the value of the variable is overwritten by the new assignment. This rule also applies to function parameters. Noncompliant Code Example var a = 'foo'; function a() {} // Noncompliant console.log(a); // prints "foo" function myFunc(arg) { var arg = "event"; // Noncompliant, argument value is lost } fun(); // prints "bar" function fun() { console.log("foo"); } fun(); // prints "bar" function fun() { // Noncompliant console.log("bar"); } fun(); // prints "bar" Compliant Solution var a = 'foo'; function otherName() {} console.log(a); function myFunc(arg) { var newName = "event"; } fun(); // prints "foo" function fun() { print("foo"); } fun(); // prints "foo" function printBar() { print("bar"); } printBar(); // prints "bar" | CODE\_SMELL | MAJOR | 239 |
| Ternary operators should not be nested |  | CODE\_SMELL | MAJOR | 26 |
| Array indexes should be numeric | Associative arrays allow you to store values in an array with either numeric or named indexes. But creating and populating an object is just as easy as an array, and more reliable if you need named members. Noncompliant Code Example let arr = []; arr[0] = 'a'; arr['name'] = 'bob'; // Noncompliant arr[1] = 'foo'; Compliant Solution let obj = { name: 'bob', arr: ['a', 'foo'] }; | CODE\_SMELL | MAJOR | 2 |
| Functions should not be called both with and without "new" | Constructor functions, which create new object instances, must only be called with new. Non-constructor functions must not. Mixing these two usages could lead to unexpected results at runtime. Noncompliant Code Example function getNum() { return 5; } function Num(numeric, alphabetic) { this.numeric = numeric; this.alphabetic = alphabetic; } var myFirstNum = getNum(); var my2ndNum = new getNum(); // Noncompliant. An empty object is returned, NOT 5 var myNumObj1 = new Num(); var myNumObj2 = Num(); // Noncompliant. undefined is returned, NOT an object | CODE\_SMELL | MAJOR | 1 |
| Literals should not be thrown | It is a bad practice to throw something that’s not derived at some level from Error. If you can’t find an existing Error type that suitably conveys what you need to convey, then you should extend Error to create one. Specifically, part of the point of throwing Errors is to communicate about the conditions of the error, but literals have far less ability to communicate meaningfully than Errors because they don’t include stacktraces. Noncompliant Code Example throw 404; // Noncompliant throw "Invalid negative index."; // Noncompliant Compliant Solution throw new Error("Status: " + 404); throw new Error("Invalid negative index.");{code} | CODE\_SMELL | MAJOR | 5 |
| Functions should always return the same type | Unlike strongly typed languages, JavaScript does not enforce a return type on a function. This means that different paths through a function can return different types of values, which can be very confusing to the user and significantly harder to maintain. Noncompliant Code Example function foo(a) { // Noncompliant if (a === 1) { return true; } return 3; } Compliant Solution function foo(a) { if (a === 1) { return true; } return false; } Exceptions Functions returning this are ignored. function foo() { // ... return this; } Functions returning expressions having type any are ignored. | CODE\_SMELL | MAJOR | 14 |
| Array-mutating methods should not be used misleadingly | Many of JavaScript’s Array methods return an altered version of the array while leaving the source array intact. reverse and sort do not fall into this category. Instead, they alter the source array in addition to returning the altered version, which is likely not what was intended. This rule raises an issue when the return values of these methods are assigned, which could lead maintainers to overlook the fact that the original value is altered. Noncompliant Code Example var b = a.reverse(); // Noncompliant var d = c.sort(); // Noncompliant Compliant Solution var b = [...a].reverse(); // de-structure and create a new array, so reverse doesn't impact 'a' a.reverse(); c.sort(); // this sorts array in place | CODE\_SMELL | MAJOR | 3 |
| Assignments should not be redundant | The transitive property says that if a == b and b == c, then a == c. In such cases, there’s no point in assigning a to c or vice versa because they’re already equivalent. This rule raises an issue when an assignment is useless because the assigned-to variable already holds the value on all execution paths. Noncompliant Code Example a = b; c = a; b = c; // Noncompliant: c and b are already the same Compliant Solution a = b; c = a; | CODE\_SMELL | MAJOR | 4 |
| Regular expressions should not be too complicated | Overly complicated regular expressions are hard to read and to maintain and can easily cause hard-to-find bugs. If a regex is too complicated, you should consider replacing it or parts of it with regular code or splitting it apart into multiple patterns at least. The complexity of a regular expression is determined as follows: Each of the following operators increases the complexity by an amount equal to the current nesting level and also increases the current nesting level by one for its arguments: | - when multiple | operators are used together, the subsequent ones only increase the complexity by 1 Quantifiers (\*, +, ?, {n,m}, {n,} or {n}) Lookahead and lookbehind assertions Additionally, each use of the following features increase the complexity by 1 regardless of nesting: character classes back references If a regular expression is split among multiple variables, the complexity is calculated for each variable individually, not for the whole regular expression. Noncompliant Code Example const datePattern = /^(?:(?:31(\/|-|\.)(?:0?[13578]|1[02]))\1|(?:(?:29|30)(\/|-|\.)(?:0?[13-9]|1[0-2])\2))(?:(?:1[6-9]|[2-9]\d)?\d{2})$|^(?:29(\/|-|\.)0?2\3(?:(?:(?:1[6-9]|[2-9]\d)?(?:0[48]|[2468][048]|[13579][26])|(?:(?:16|[2468][048]|[3579][26])00))))$|^(?:0?[1-9]|1\d|2[0-8])(\/|-|\.)(?:(?:0?[1-9])|(?:1[0-2]))\4(?:(?:1[6-9]|[2-9]\d)?\d{2})$/; if (dateString.match(datePattern)) { handleDate(dateString); } Compliant Solution const datePattern = /^\d{1,2}([-/.])\d{1,2}\1\d{1,4}$/; if (dateString.match(datePattern)) { const dateParts = dateString.split(/[-/.]/); const day = parseInt(dateParts[0]); const month = parseInt(dateParts[1]); const year = parseInt(dateParts[2]); // Put logic to validate and process the date based on its integer parts here } | CODE\_SMELL | MAJOR | 71 |
| Character classes in regular expressions should not contain the same character twice | Character classes in regular expressions are a convenient way to match one of several possible characters by listing the allowed characters or ranges of characters. If the same character is listed twice in the same character class or if the character class contains overlapping ranges, this has no effect. Thus duplicate characters in a character class are either a simple oversight or a sign that a range in the character class matches more than is intended or that the author misunderstood how character classes work and wanted to match more than one character. A common example of the latter mistake is trying to use a range like [0-99] to match numbers of up to two digits, when in fact it is equivalent to [0-9]. Another common cause is forgetting to escape the - character, creating an unintended range that overlaps with other characters in the character class. Noncompliant Code Example /[0-99]/ // Noncompliant, this won't actually match strings with two digits /[0-9.-\_]/ // Noncompliant, .-\_ is a range that already contains 0-9 (as well as various other characters such as capital letters) Compliant Solution /[0-9]{1,2}/ /[0-9.\-\_]/ | CODE\_SMELL | MAJOR | 6 |
| Reluctant quantifiers in regular expressions should be followed by an expression that can't match the empty string | When a reluctant (or lazy) quantifier is followed by a pattern that can match the empty string or directly by the end of the regex, it will always match zero times for \*? or one time for +?. If a reluctant quantifier is followed directly by the end anchor ($), it behaves indistinguishably from a greedy quantifier while being less efficient. This is likely a sign that the regex does not work as intended. Noncompliant Code Example str.split(/.\*?x?/); // Noncompliant, this will behave just like "x?" /^.\*?$/.test(str); // Noncompliant, replace with ".\*" Compliant Solution str.split(/.\*?x/); /^.\*$/.test(str); | CODE\_SMELL | MAJOR | 68 |
| Single-character alternations in regular expressions should be replaced with character classes | When an alternation contains multiple alternatives that consist of a single character, it can be rewritten as a character class. This should be preferred because it is more efficient and can even help prevent stack overflows when used inside a repetition (see rule S5998). Noncompliant Code Example /a|b|c/; // Noncompliant Compliant Solution /[abc]/; // or /[a-c]/; | CODE\_SMELL | MAJOR | 39 |
| Regular expressions should not contain multiple spaces | Multiple spaces in a regular expression can make it hard to tell how many spaces should be matched. It’s more readable to use only one space and then indicate with a quantifier how many spaces are expected. Noncompliant Code Example const pattern = /Hello, world!/; Compliant Solution const pattern = /Hello, {3}world!/; | CODE\_SMELL | MAJOR | 1 |
| Regular expressions should not contain empty groups | There are several reasons to use a group in a regular expression: to change the precedence (e.g. do(g|or) will match 'dog' and 'door') to remember parenthesised part of the match in the case of capturing group to improve readability In any case, having an empty group is most probably a mistake. Either it is a leftover after refactoring and should be removed, or the actual parentheses were intended and were not escaped. Noncompliant Code Example const dateRegex = /^(?:0[1-9]|[12][0-9]|3[01])[- /.](?:0[1-9]|1[012])[- /.](?:19|20)\d\d(?:)$/; // Noncompliant, useless group at the end const methodCallRegex = /foo()/; // Noncompliant, will match only 'foo' Compliant Solution const dateRegex = /^(?:0[1-9]|[12][0-9]|3[01])[- /.](?:0[1-9]|1[012])[- /.](?:19|20)\d\d$/; const methodCallRegex = /foo\(\)/; // OK, matches 'foo()' | CODE\_SMELL | MAJOR | 1 |
| Character classes in regular expressions should not contain only one character | Character classes in regular expressions are a convenient way to match one of several possible characters by listing the allowed characters or ranges of characters. If a character class contains only one character, the effect is the same as just writing the character without a character class. Thus, having only one character in a character class is usually a simple oversight that remained after removing other characters of the class. Noncompliant Code Example /a[b]c/ /[\^]/ Compliant Solution /abc/ /\^/ /a[\*]c/ // Compliant, see Exceptions Exceptions This rule does not raise when the character inside the class is a metacharacter. This notation is sometimes used to avoid escaping (e.g., [.]{3} to match three dots). | CODE\_SMELL | MAJOR | 8 |
| Extra semicolons should be removed | Extra semicolons are usually introduced by mistake, for example because: It was meant to be replaced by one more property declaration, but this was forgotten. There was a typo which lead the semicolon to be doubled, i.e. ;;. | CODE\_SMELL | MINOR | 1 |
| Boolean literals should not be used in comparisons | Boolean literals should be avoided in comparison expressions == and != to improve code readability. This rule also reports on redundant boolean operations. Noncompliant Code Example let someValue = "0"; // ... if (someValue == true) { /\* ... \*/ } if (someBooleanValue != true) { /\* ... \*/ } doSomething(!false); Compliant Solution if (someValue &amp;&amp; someValue != "0") { /\* ... \*/ } if (!someBooleanValue) { /\* ... \*/ } doSomething(true); | CODE\_SMELL | MINOR | 23 |
| Return of boolean expressions should not be wrapped into an "if-then-else" statement | Return of boolean literal statements wrapped into if-then-else flow should be simplified. Note that if the result of the expression is not a boolean but for instance an integer, then double negation should be used for proper conversion. if (expression) { return true; } else { return false; } or if (expression) { return true; } return false; Compliant Solution return expression; or return !!expression; | CODE\_SMELL | MINOR | 6 |
| Unused local variables and functions should be removed | If a local variable or a local function is declared but not used, it is dead code and should be removed. Doing so will improve maintainability because developers will not wonder what the variable or function is used for. Noncompliant Code Example function numberOfMinutes(hours) { var seconds = 0; // seconds is never used return hours \* 60; } Compliant Solution function numberOfMinutes(hours) { return hours \* 60; } | CODE\_SMELL | MINOR | 84 |
| Multiline string literals should not be used | Continuing a string across a linebreak is supported in most script engines, but it is not a part of ECMAScript. Additionally, the whitespace at the beginning of each line can’t be safely stripped at compile time, and any whitespace after the slash will result in tricky errors. Noncompliant Code Example var myString = 'A rather long string of English text, an error message \ actually that just keeps going and going -- an error \ message to make the Energizer bunny blush (right through \ those Schwarzenegger shades)! Where was I? Oh yes, \ you\'ve got an error and all the extraneous whitespace is \ just gravy. Have a nice day.'; // Noncompliant Compliant Solution var myString = 'A rather long string of English text, an error message ' + 'actually that just keeps going and going -- an error ' + 'message to make the Energizer bunny blush (right through ' + 'those Schwarzenegger shades)! Where was I? Oh yes, ' + 'you\'ve got an error and all the extraneous whitespace is ' + 'just gravy. Have a nice day.'; | CODE\_SMELL | MINOR | 21 |
| Deprecated APIs should not be used | Once deprecated, classes, and interfaces, and their members should be avoided, rather than used, inherited or extended. Deprecation is a warning that the class or interface has been superseded, and will eventually be removed. The deprecation period allows you to make a smooth transition away from the aging, soon-to-be-retired technology. Noncompliant Code Example export interface LanguageService { /\*\* \* @deprecated Use getEncodedSyntacticClassifications instead. \*/ getSyntacticClassifications(fileName: string, span: TextSpan): ClassifiedSpan[]; } const syntacticClassifications = getLanguageService().getSyntacticClassifications(file, span); // Noncompliant See MITRE, CWE-477 - Use of Obsolete Functions | CODE\_SMELL | MINOR | 23 |
| Boolean checks should not be inverted | It is needlessly complex to invert the result of a boolean comparison. The opposite comparison should be made instead. Noncompliant Code Example if (!(a === 2)) { ... } // Noncompliant Compliant Solution if (a !== 2) { ... } | CODE\_SMELL | MINOR | 3 |
| Jump statements should not be redundant | Jump statements, such as return, break and continue let you change the default flow of program execution, but jump statements that direct the control flow to the original direction are just a waste of keystrokes. Noncompliant Code Example function redundantJump(x) { if (x == 1) { console.log("x == 1"); return; // Noncompliant } } Compliant Solution function redundantJump(x) { if (x == 1) { console.log("x == 1"); } } Exceptions break and return inside switch statement are ignored, because they are often used for consistency. continue with label is also ignored, because label is usually used for clarity. Also a jump statement being a single statement in a block is ignored. | CODE\_SMELL | MINOR | 1 |
| "for of" should be used with Iterables | If you have an iterable, such as an array, set, or list, your best option for looping through its values is the for of syntax. Use a counter, and …​ well you’ll get the right behavior, but your code just isn’t as clean or clear. In a browser environment, NodeList and other array-like collections should work by default. If you are using TypeScript and seeing a type error, make sure your configuration is correct. Noncompliant Code Example const arr = [4, 3, 2, 1]; for (let i = 0; i &lt; arr.length; i++) { // Noncompliant console.log(arr[i]); } Compliant Solution const arr = [4, 3, 2, 1]; for (let value of arr) { console.log(value); } | CODE\_SMELL | MINOR | 75 |
| Regular expression literals should be used when possible | Regular expression literals should be preferred over the RegExp constructor calls when the pattern is a literal. Simply using a regular expression literal is more concise and easier to read and does not require escaping like a string literal does. Using the RegExp constructor is suitable when the pattern is computed dynamically, e.g. when it is provided by the user. Noncompliant Code Example new RegExp(/foo/); new RegExp('bar'); new RegExp('baz', 'i'); new RegExp("\\d+"); new RegExp(`qux|quuz`); Compliant Solution /foo/; /bar/; /baz/i; /\d+/; /qux|quuz/; new RegExp(`Dear ${title},`); | CODE\_SMELL | MINOR | 1 |
| Regular expression quantifiers and character classes should be used concisely | With regular expressions syntax, it’s possible to express the same thing in many ways. For example, to match a two-digit number, one could write [0-9]{2,2} or \d{2}. Latter is not only shorter in terms of expression length, but also easier to read and thus to maintain. This rule recommends to replace some bulky quantifiers and character classes with more concise equivalents: \d for [0-9] and \D for [^0-9] \w for [A-Za-z0-9\_] and \W for [^A-Za-z0-9\_] . for character classes matching everything (e.g. [\w\W], [\d\D], or [\s\S] with s flag) x? for x{0,1}, x\* for x{0,}, x+ for x{1,}, x{N} for x{N,N} Noncompliant Code Example /a{1,}/; // Noncompliant, '{1,}' quantifier is the same as '+' /[A-Za-z0-9\_]/; // Noncompliant, '\w' is equivalent Compliant Solution /a+/; /\w/; | CODE\_SMELL | MINOR | 28 |

# Security Hotspots

## Security hotspots count by category and priority

|  |  |  |  |
| --- | --- | --- | --- |
| Category / Priority | LOW | MEDIUM | HIGH |
| LDAP Injection | 0 | 0 | 0 |
| Object Injection | 0 | 0 | 0 |
| Server-Side Request Forgery (SSRF) | 0 | 0 | 0 |
| XML External Entity (XXE) | 0 | 0 | 0 |
| Insecure Configuration | 38 | 0 | 0 |
| XPath Injection | 0 | 0 | 0 |
| Authentication | 0 | 0 | 5 |
| Weak Cryptography | 0 | 17 | 0 |
| Denial of Service (DoS) | 0 | 574 | 0 |
| Log Injection | 0 | 0 | 0 |
| Cross-Site Request Forgery (CSRF) | 0 | 0 | 0 |
| Open Redirect | 0 | 0 | 0 |
| Permission | 0 | 0 | 0 |
| SQL Injection | 0 | 0 | 2 |
| Encryption of Sensitive Data | 3 | 0 | 0 |
| Traceability | 0 | 0 | 0 |
| Buffer Overflow | 0 | 0 | 0 |
| File Manipulation | 0 | 0 | 0 |
| Code Injection (RCE) | 0 | 35 | 0 |
| Cross-Site Scripting (XSS) | 0 | 0 | 0 |
| Command Injection | 0 | 0 | 0 |
| Path Traversal Injection | 0 | 0 | 0 |
| HTTP Response Splitting | 0 | 0 | 0 |
| Others | 21 | 0 | 0 |

## Security hotspots List

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Category | Name | Priority | Severity | Count |
| Denial of Service (DoS) | Using slow regular expressions is security-sensitive | MEDIUM | CRITICAL | 565 |
| Weak Cryptography | Using pseudorandom number generators (PRNGs) is security-sensitive | MEDIUM | CRITICAL | 15 |
| Insecure Configuration | Delivering code in production with debug features activated is security-sensitive | LOW | MINOR | 38 |
| Encryption of Sensitive Data | Using clear-text protocols is security-sensitive | LOW | CRITICAL | 3 |
| Authentication | Hard-coded credentials are security-sensitive | HIGH | BLOCKER | 4 |
| Code Injection (RCE) | Dynamically executing code is security-sensitive | MEDIUM | CRITICAL | 35 |
| Others | Authorizing an opened window to access back to the originating window is security-sensitive | LOW | MINOR | 17 |
| Others | Using weak hashing algorithms is security-sensitive | LOW | CRITICAL | 3 |
| Others | Using publicly writable directories is security-sensitive | LOW | CRITICAL | 1 |
| SQL Injection | Formatting SQL queries is security-sensitive | HIGH | MAJOR | 2 |
| Denial of Service (DoS) | Using slow regular expressions is security-sensitive | MEDIUM | CRITICAL | 9 |
| Authentication | Hard-coded passwords are security-sensitive | HIGH | BLOCKER | 1 |
| Weak Cryptography | Using pseudorandom number generators (PRNGs) is security-sensitive | MEDIUM | CRITICAL | 2 |