REACT

VERSION : 18.2.0 (LATEST)

Source Code Analysis

CONTENT

INTRODUCTION

This document contains results of the code analysis of React.

CONFIGURATION

- Quality Profiles
    - Names: Sonar way [CSS]; Sonar way [JavaScript]; Sonar way [TypeScript]; Sonar way [HTML];
    - Files: AYZmCyZ4WV-BCtvHi3KY.json; AYZmCy_LWV-BCtvHi3Ud.json; AYZmC1WNWV-BCtvHi4Yg.json; AYZmC0UFWV-BCtvHi4Bp.json;
- Quality Gate
    - Name: Sonar way
    - File: Sonar way.xml

SYNTHESIS

ANALYSIS STATUS

Reliability	Security	Security Review	Maintainability
E	D	E	A

QUALITY GATE STATUS

Quality Gate Status	Passed

METRICS

Coverage	Duplication	Comment density	Median number of lines of code per file	Adherence to coding standard
0.0 %	18.8 %	8.2 %	56.0	94.8 %

TESTS

Total	Success Rate	Skipped	Errors	Failures
0	0 %	0	0	0

DETAILED TECHNICAL DEBT

Reliability	Security	Maintainability	Total
4d 0h 44min	0d 2h 10min	109d 0h 3min	**113d 2h 57min**

REACT JS

METRICS RANGE

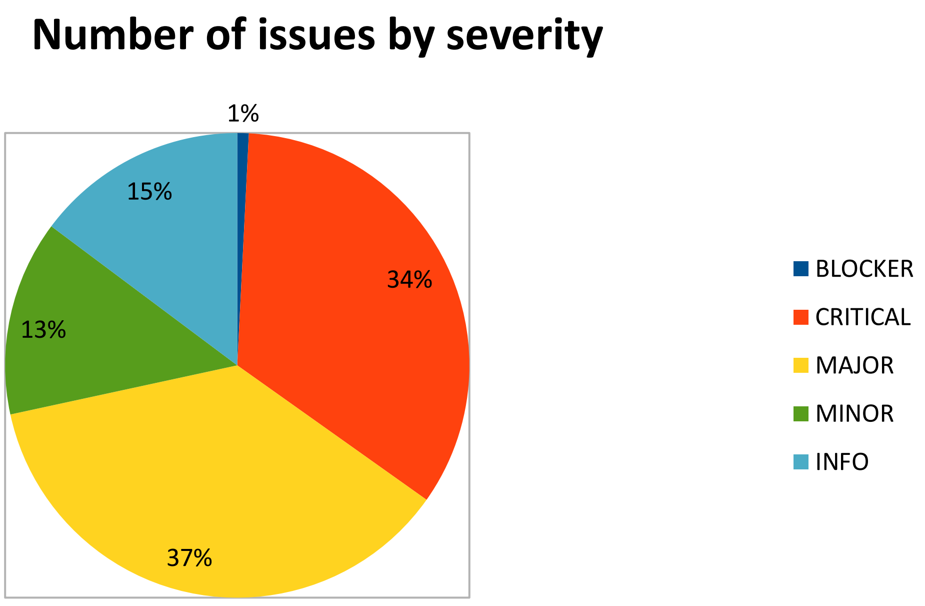
	Cyclomatic Complexity	Cognitive Complexity	Lines of code per file	Comment density (%)	Coverage	Duplication (%)
Min	0.0	0.0	0.0	0.0	0.0	0.0
Max	49392.0	34371.0	298385.0	100.0	0.0	100.0

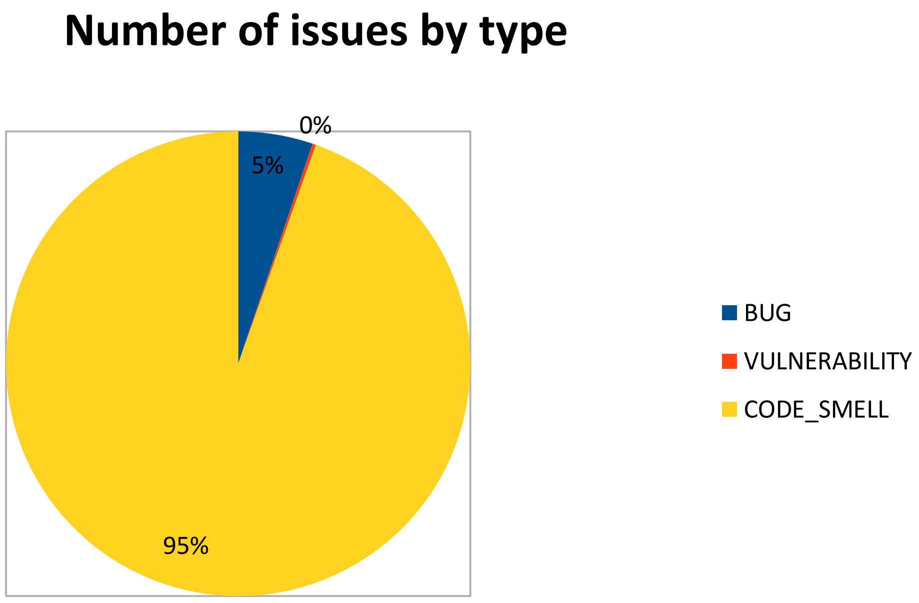
VOLUME

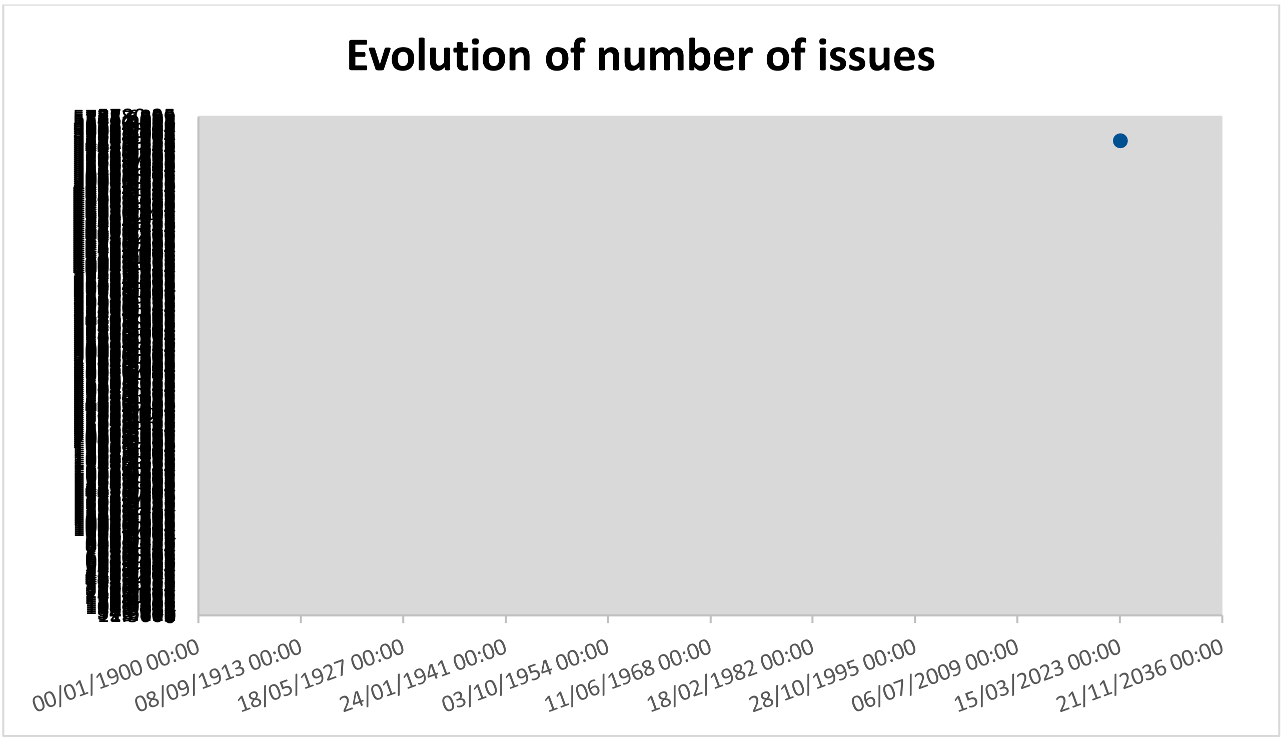
Language	Number
CSS	3238
JavaScript	345738
TypeScript	616
HTML	3388
Total	352980

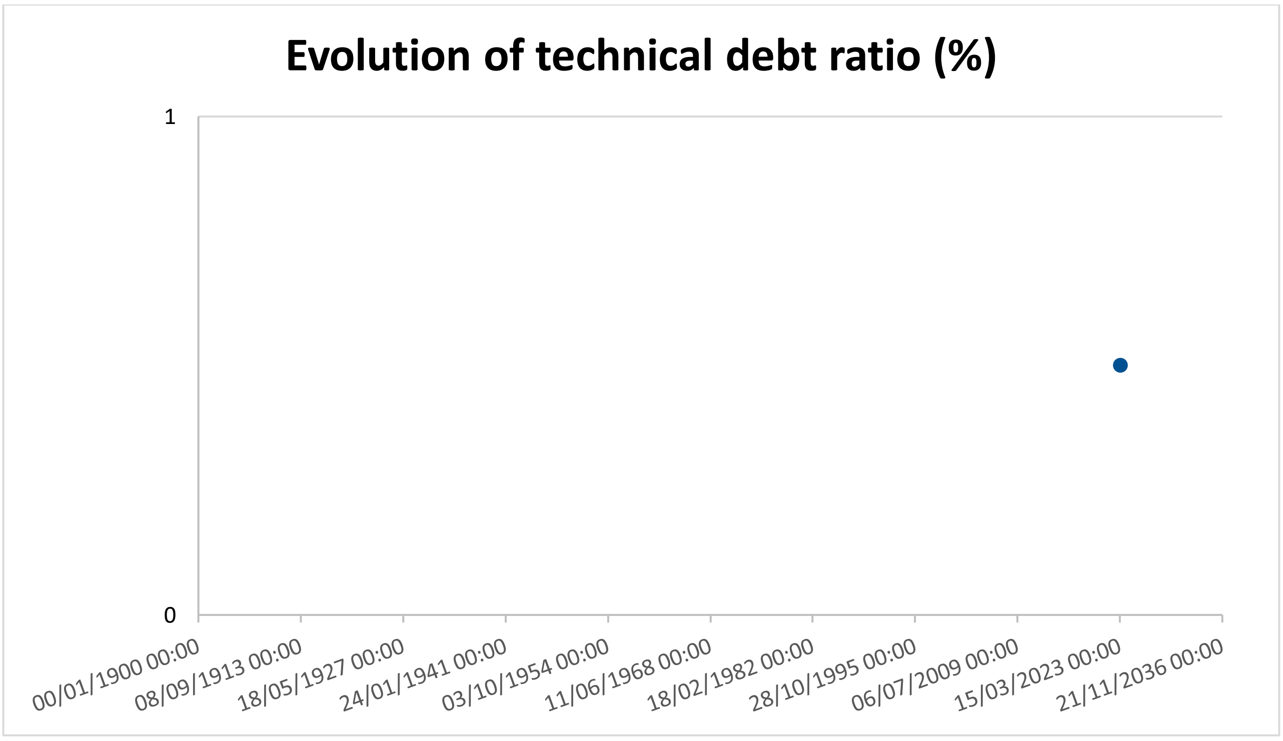
ISSUES

CHARTS






Evolution of number of issues
00/01/1900 00:00
08/09/1913 00:00
18/05/1927 00:00
24/01/1941 00:00
03/10/1954 00:00
11/06/1968 00:00
18/02/1982 00:00
28/10/1995 00:00
06/07/2009 00:00
15/03/2023 00:00
21/11/2036 00:00


Evolution of technical debt ratio (%)
1
0
00/01/1900 00:00
08/09/1913 00:00
18/05/1927 00:00
24/01/1941 00:00
03/10/1954 00:00
11/06/1968 00:00
18/02/1982 00:00
28/10/1995 00:00
06/07/2009 00:00
15/03/2023 00:00
21/11/2036 00:00

REACT JS

ISSUES COUNT BY SEVERITY AND TYPE

Type / Severity	INFO	MINOR	MAJOR	CRITICAL	BLOCKER
BUG	0	79	144	45	18
VULNERABILITY	0	0	0	13	0
CODE_SMELL	814	670	1881	1815	26

ISSUES LIST

Name	Description	Type	Severity	Number
Loops should not be infinite	An infinite loop is one that will never end while the program is running, i.e., you have to kill the program to get out of the loop. Whether it is by meeting the loop's end condition or via a break, every loop should have an end condition. Known Limitations False positives: when yield is used - Issue #674. False positives: when an exception is raised by a function invoked within the loop. False negatives: when a loop condition is based on an element of an array or object. Noncompliant Code Example for (;;) { // Noncompliant; end condition omitted // ... } var j = 0; while (true) { // Noncompliant; constant end condition j++; } var k; var b = true; while (b) { // Noncompliant; constant end condition k++; } Compliant Solution while (true) { // break will potentially allow leaving the loop if (someCondition) { break; } } var k; var b = true; while (b) { k++; b = k &lt; 10; } outer: while(true) { while(true) { break outer; } } See CERT, MSC01-J. - Do not use an empty infinite loop	BUG	BLOCKER	18
Shorthand properties that override related longhand properties should be avoided	A shorthand property defined after a longhand property will completely override the value defined in the longhand property making the longhand one useless. The code should be refactored to consider the longhand property or to remove it completely. Noncompliant Code Example a { padding-left: 10px; padding: 20px; /* Noncompliant; padding is overriding padding-left making it useless */ } Compliant Solution a { padding: 10px; /* Compliant; padding is defining a general behaviour and padding-left, just after, is precising the left case */ padding-left: 20px; } See Mozilla Web Technology for Developers - Shorthand properties	BUG	CRITICAL	1

Jump statements should not occur in "finally" blocks	Using return, break, throw, and continue from a finally block overwrites similar statements from the suspended try and catch blocks. This rule raises an issue when a jump statement (break, continue, return and throw) would force control flow to leave a finally block. Noncompliant Code Example function foo() { try { return 1; // We expect 1 to be returned } catch(err) { return 2; // Or 2 in cases of error } finally { return 3; // Noncompliant: 3 is returned before 1, or 2, which we did not expect } } Compliant Solution function foo() { try { return 1; // We expect 1 to be returned } catch(err) { return 2; // Or 2 in cases of error } } See MITRE, CWE-584 - Return Inside Finally Block CERT, ERR04-J. - Do not complete abruptly from a finally block	BUG	CRITICAL	28
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 MISRA C:2004, 16.6 - The number of arguments passed to a function shall match the number of parameters. MITRE, CWE-628 - Function Call with Incorrectly Specified Arguments CERT, DCL07-C. - Include the appropriate type information in function declarators CERT, EXP37-C. - Call functions with the correct number and type of arguments	BUG	CRITICAL	16
"<!DOCTYPE>" declarations should appear before "<html>" tags	The &lt;!DOCTYPE&gt; declaration tells the web browser which (X)HTML version is being used on the page, and therefore how to interpret the various elements. Validators also rely on it to know which rules to enforce. It should always preceed the &lt;html&gt; tag. Noncompliant Code Example &lt;html&gt; &lt;!-- Noncompliant --&gt; ... &lt;/html&gt; Compliant Solution &lt;!DOCTYPE html&gt; &lt;html&gt; &lt;!-- Compliant --&gt; ... &lt;/html&gt;	BUG	MAJOR	25
"<title>" should be present in all pages	Titles are important because they are displayed in search engine results as well as the browser's toolbar. This rule verifies that the &lt;head&gt; tag contains a &lt;title&gt; one, and the &lt;html&gt;	BUG	MAJOR	26

	tag a &lt;head&gt; one. Noncompliant Code Example &lt;html&gt; &lt;!-- Non-Compliant --&gt; &lt;body&gt; ... &lt;/body&gt; &lt;/html&gt; Compliant Solution &lt;html&gt; &lt;!-- Compliant --&gt; &lt;head&gt; &lt;title&gt;Some relevant title&lt;/title&gt; &lt;/head&gt; &lt;body&gt; ... &lt;/body&gt; &lt;/html&gt;			
"<html>" element should have a language attribute	The &lt;html&gt;&nbsp;element should provide the lang and/or xml:lang attribute in order to identify the default language of a document. It enables assistive technologies, such as screen readers,&nbsp;to provide a comfortable reading experience by adapting the pronunciation and accent to the language. It also helps braille translation software, telling it to switch the control codes for accented characters for instance. Other benefits of marking the language include: assisting user agents in providing dictionary definitions or helping users benefit from translation tools. improving search engine ranking. Both the lang and the xml:lang attributes can take only one value. &nbsp; Noncompliant Code Example &lt;!DOCTYPE html&gt; &lt;html&gt; &lt;!-- Noncompliant --&gt; &nbsp;&nbsp;&nbsp; &lt;head&gt; &nbsp;&nbsp;&nbsp;&nbsp; &nbsp;&nbsp;&nbsp; &lt;title&gt;A page written in english&lt;/title&gt; &nbsp;&nbsp;&nbsp;&nbsp; &nbsp;&nbsp;&nbsp; &lt;meta content="text/html; charset=utf-8" /&gt; &nbsp;&nbsp;&nbsp; &lt;/head&gt; &nbsp; &nbsp;&nbsp;&nbsp; &lt;body&gt; &nbsp;&nbsp;&nbsp;&nbsp; &nbsp;&nbsp;&nbsp; ... &nbsp;&nbsp; &nbsp;&nbsp;&nbsp; &lt;/body&gt; &lt;/html&gt; Compliant Solution &lt;!DOCTYPE html&gt; &lt;html lang="en"&gt; &nbsp;&nbsp;&nbsp; &lt;head&gt; &nbsp;&nbsp;&nbsp;&nbsp; &nbsp;&nbsp;&nbsp; &lt;title&gt;A page written in english&lt;/title&gt; &nbsp;&nbsp;&nbsp;&nbsp; &nbsp;&nbsp;&nbsp; &lt;meta content="text/html; charset=utf-8" /&gt; &nbsp;&nbsp;&nbsp; &lt;/head&gt; &nbsp; &nbsp;&nbsp;&nbsp; &lt;body&gt; &nbsp;&nbsp;&nbsp;&nbsp; &nbsp;&nbsp;&nbsp; ... &nbsp;&nbsp; &nbsp;&nbsp;&nbsp; &lt;/body&gt; &lt;/html&gt; &lt;!DOCTYPE html&gt; &lt;html lang="en" xml:lang="en"&gt; &nbsp;&nbsp;&nbsp; &lt;head&gt; &nbsp;&nbsp;&nbsp;&nbsp; &nbsp;&nbsp;&nbsp; &lt;title&gt;A page written in english&lt;/title&gt; &nbsp;&nbsp;&nbsp;&nbsp; &nbsp;&nbsp;&nbsp; &lt;meta content="text/html; charset=utf-8" /&gt; &nbsp;&nbsp;&nbsp; &lt;/head&gt; &nbsp;	BUG	MAJOR	62

	&nbsp;&nbsp;&nbsp; &lt;body&gt; &nbsp;&nbsp;&nbsp;&nbsp; &nbsp;&nbsp;&nbsp; ... &nbsp;&nbsp; &nbsp;&nbsp;&nbsp; &lt;/body&gt; &lt;/html&gt; See WCAG2, H57 - Using language attributes on the html element WCAG2, 3.1.1 - Language of Page			
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	1
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	3
Bitwise operators should not be used in boolean contexts	The bitwise operators &amp;, | can be mistaken for the boolean operators &amp;&amp; and ||. This rule raises an issue when &amp; or | is used in a boolean context. Noncompliant Code Example if (a &amp; b) { ... } // Noncompliant; &amp; used in error Compliant Solution if (a &amp;&amp; b) { ... } Exceptions When a file contains other bitwise operations, (^, &lt;&lt;, &gt;&gt;&gt;, &gt;&gt;, ~, &amp;=, ^=, |=, &lt;&lt;=, &gt;&gt;=, &gt;&gt;&gt;= and &amp; or | used with a numeric literal as the right operand) all issues in the file are ignored, because it is evidence that bitwise operations are truly intended in the file.	BUG	MAJOR	2
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	BUG	MAJOR	3

	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; } Exceptions This rule ignores unreachable break statements in switch clauses. switch (x) { case 42: return 43; break; // Compliant default: doSomething(); } Hoisted variables declarations without initialization are always considered reachable. function bar() { return x = function() { x.foo = 42; } var x; } See MITRE, CWE-561 - Dead Code CERT, MSC56-J. - Detect and remove superfluous code and values CERT, MSC12-C. - Detect and remove code that has no effect or is never executed			
Identical expressions should not be used on both sides of a binary operator	Using the same value on either side of a binary operator is almost always a mistake. In the case of logical operators, it is either a copy/paste error and therefore a bug, or it is simply wasted code, and should be simplified. In the case of bitwise operators and most binary mathematical operators, having the same value on both sides of an operator yields predictable results, and should be simplified. Noncompliant Code Example if (a == b &amp;&amp; a == b) { // if the first one is true, the second one is too doX(); } if (a &gt; a) { // always false doW(); } var j = 5 / 5; //always 1 var k = 5 - 5; //always 0 Exceptions The specific case of testing one variable against itself is a valid test for NaN and is therefore ignored. Similarly, left-shifting 1 onto 1 is common in the construction of bit masks, and is ignored. Moreover comma operator , and instanceof operator are ignored as there are use-cases when there usage is valid. if (f !== f) { // test for NaN value console.log("f is NaN"); } var i = 1 &lt;&lt; 1; // Compliant var j = a &lt;&lt; a; // Noncompliant See CERT, MSC12-C. - Detect and remove code that has no effect or is never executed S1656 - Implements a check on =.	BUG	MAJOR	4
Properties of variables with "null" or "undefined" values should not be accessed	When a variable is assigned an undefined or null value, it has no properties. Trying to access properties of such a variable anyway results in a TypeError, causing abrupt termination of the script if the error is not caught in a catch block. But instead of catch-ing this condition, it is best to avoid it altogether. Noncompliant Code Example if (x === undefined) { console.log(x.length); // Noncompliant; TypeError will be thrown } See MITRE, CWE-476 - NULL Pointer Dereference CERT, EXP34-C. - Do not dereference null pointers CERT, EXP01-J. - Do not use a null in a case where	BUG	MAJOR	8

	an object is required			
Setters should not return values	Functions declared with the set keyword will automatically return the values they were passed. Thus any value explicitly returned from a setter will be ignored, and explicitly returning a value is an error. Noncompliant Code Example var person = { // ... set name(name) { this.name = name; return 42; // Noncompliant } } Compliant Solution var person = { // ... set name(name) { this.name = name; } }	BUG	MAJOR	1
Strict equality operators should not be used with dissimilar types	Comparing dissimilar types using the strict equality operators === and !== will always return the same value, respectively false and true, because no type conversion is done before the comparison. Thus, such comparisons can only be bugs. Noncompliant Code Example var a = 8; var b = "8"; if (a === b) { // Noncompliant; always false // ... } Compliant Solution var a = 8; var b = "8"; if (a == b) { // ... } or var a = 8; var b = "8"; if (a === Number(b)) { // ... }	BUG	MAJOR	2
All branches in a conditional structure should not have exactly the same implementation	Having all branches in a switch or if chain with the same implementation is an error. Either a copy-paste error was made and something different should be executed, or there shouldn't be a switch/if chain at all. Noncompliant Code Example if (b == 0) { // Noncompliant doOneMoreThing(); } else { doOneMoreThing(); } let a = (b == 0) ? getValue() : getValue(); // Noncompliant switch (i) { // Noncompliant case 1: doSomething(); break; case 2: doSomething(); break; case 3: doSomething(); break; default: doSomething(); } Exceptions This rule does not apply to if chains without else-s, or to switch-es without default clauses. if(b == 0) { //no issue, this could have been done on purpose to make the code more readable doSomething(); } else if(b == 1) { doSomething(); }	BUG	MAJOR	4
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	BUG	MAJOR	3

	Assigning			
"<strong>" and "<em>" tags should be used	The &lt;strong&gt;/&lt;b&gt; and &lt;em&gt;/&lt;i&gt; tags have exactly the same effect in most web browsers, but there is a fundamental difference between them: &lt;strong&gt; and &lt;em&gt; have a semantic meaning whereas &lt;b&gt; and &lt;i&gt; only convey styling information like CSS. While &lt;b&gt; can have simply no effect on a some devices with limited display or when a screen reader software is used by a blind person, &lt;strong&gt; will: Display the text bold in normal browsers Speak with lower tone when using a screen reader such as Jaws Consequently: in order to convey semantics, the &lt;b&gt; and &lt;i&gt; tags shall never be used, in order to convey styling information, the &lt;b&gt; and &lt;i&gt; should be avoided and CSS should be used instead. Noncompliant Code Example &lt;i&gt;car&lt;/i&gt; &lt;!-- Noncompliant --&gt; &lt;b&gt;train&lt;/b&gt; &lt;!-- Noncompliant --&gt; Compliant Solution &lt;em&gt;car&lt;/em&gt; &lt;strong&gt;train&lt;/strong&gt; Exceptions This rule is relaxed in case of icon fonts usage. &lt;i class="..." aria-hidden="true" /&gt; &lt;!-- Compliant icon fonts usage --&gt;	BUG	MINOR	31
"<frames>" should have a "title" attribute	Frames allow different web pages to be put together on the same visual space. Users without disabilities can easily scan the contents of all frames at once. However, visually impaired users using screen readers hear the page content linearly. The title attribute is used to list all the page's frames, enabling those users to easily navigate among them. Therefore, the &lt;frame&gt; and &lt;iframe&gt; tags should always have a title attribute. Noncompliant Code Example &lt;frame src="index.php?p=menu"&gt; &lt;-- Non-Compliant --&gt; &lt;frame src="index.php?p=home" name="contents"&gt; &lt;-- Non-Compliant --&gt; Compliant Solution &lt;frame src="index.php?p=menu" title="Navigation menu"&gt; &lt;-- Compliant --&gt; &lt;frame src="index.php?p=home" title="Main content" name="contents"&gt; &lt;-- Compliant --&gt;	BUG	MINOR	43
Function parameters, caught	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	BUG	MINOR	1

exceptions and foreach variables' initial values should not be ignored	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 } }			
Empty collections should not be accessed or iterated	When a collection is empty it makes no sense to access or iterate it. Doing so anyway is surely an error; either population was accidentally omitted or the developer doesn't understand the situation. Noncompliant Code Example let strings = []; if (strings.includes("foo")) {} // Noncompliant for (str of strings) {} // Noncompliant strings.forEach(str =&gt; doSomething(str)); // Noncompliant	BUG	MINOR	4
"switch" statements should not contain non-case labels	Even if it is legal, mixing case and non-case labels in the body of a switch statement is very confusing and can even be the result of a typing error. Noncompliant Code Example Case 1, the code is syntactically correct but the behavior is not the expected one switch (day) { case MONDAY: case TUESDAY: WEDNESDAY: // instead of "case WEDNESDAY" doSomething(); break; ... } Case 2, the code is correct and behaves as expected but is hardly readable switch (day) { case MONDAY: break; case TUESDAY: foo:for(i = 0 ; i &lt; X ; i++) { /* ... */ break foo; // this break statement doesn't relate to the nesting case TUESDAY /* ... */ } break; /* ... */ } Compliant Solution Case 1 switch (day) { case MONDAY: case TUESDAY: case WEDNESDAY: doSomething(); break; ... } Case 2 switch (day) { case MONDAY: break; case TUESDAY: compute(args); // put the content of the labelled "for" statement in a dedicated method break; /* ... */ }	CODE_SMELL	BLOCKER	1
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	CODE_SMELL	BLOCKER	19

	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 CERT, MSC17-C. - Finish every set of statements associated with a case label with a break statement CERT, MSC52-J. - Finish every set of statements associated with a case label with a break statement			
Variables should be declared explicitly	JavaScript variable scope can be particularly difficult to understand and get right. The situation gets even worse when you consider the accidental creation of global variables, which is what happens when you declare a variable inside a function or the for clause of a for-loop without using the let, const or var keywords. let and const were introduced in ECMAScript 2015, and are now the preferred keywords for variable declaration. Noncompliant Code Example function f(){ i = 1; // Noncompliant; i is global for (j = 0; j &lt; array.length; j++) { // Noncompliant; j is global now too // ... } } Compliant Solution function f(){ var i = 1; for (let j = 0; j &lt; array.length; j++) { // ... } }	CODE_SMELL	BLOCKE R	4
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	BLOCKE R	2
Functions should not be	There are several reasons for a function not to have a function body: It is an unintentional omission, and	CODE_SMELL	CRITICA	930

empty	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(); };		L	
"for" loop increment clauses should modify the loops' counters	It can be extremely confusing when a for loop's counter is incremented outside of its increment clause. In such cases, the increment should be moved to the loop's increment clause if at all possible. Noncompliant Code Example for (i = 0; i &lt; 10; j++) { // Noncompliant // ... i++; } Compliant Solution for (i = 0; i &lt; 10; i++, j++) { // ... } Or for (i = 0; i &lt; 10; i++) { // ... j++; }	CODE_SMELL	CRITICAL	2
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	22
"void" should not be used	The void operator evaluates its argument and unconditionally returns undefined. It can be useful in pre-ECMAScript 5 environments, where undefined could be reassigned, but generally, its use makes code harder to understand. Noncompliant Code Example void doSomething(); Compliant Solution doSomething(); Exceptions No issue is raised when void 0 is used in place of undefined. if (parameter === void 0) {...} No issue is also raised when void is used before immediately invoked function expressions. void (function() { ... }());	CODE_SMELL	CRITICAL	32
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	553

"await" should only be used with promises	It is possible to use await on values which are not Promises, but it's useless and misleading. The point of await is to pause execution until the Promise's asynchronous code has run to completion. With anything other than a Promise, there's nothing to wait for. This rule raises an issue when an awaited value is guaranteed not to be a Promise. Noncompliant Code Example let x = 42; await x; // Noncompliant Compliant Solution let x = new Promise(resolve =&gt; resolve(42)); await x; let y = p ? 42 : new Promise(resolve =&gt; resolve(42)); await y;	CODE_SMELL	CRITICA L	275
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(); };	CODE_SMELL	CRITICA L	1
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	813
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	1
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 { }	CODE_SMELL	MAJOR	6

	Compliant Solution a { color: pink; }			
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	5
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) { ... }	CODE_SMELL	MAJOR	12
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	2
Variables should not be shadowed	Overriding or shadowing a variable declared in an outer scope can strongly impact the readability, and therefore the maintainability, of a piece of code. Further, it could lead maintainers to introduce bugs because they think they're using one variable but are really using another. See CERT, DCL01-C. - Do not reuse variable names in subscopes CERT, DCL51-J. - Do not shadow or obscure identifiers in subscopes	CODE_SMELL	MAJOR	195
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	19

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. 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: assignments at declaration-level: let a = b = 0; 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 CERT, EXP45-C. - Do not perform assignments in selection statements CERT, EXP51-J. - Do not perform assignments in conditional expressions	CODE_SMELL	MAJOR	36
Track uses of "FIXME" tags	FIXME tags are commonly used to mark places where a bug is suspected, but which the developer wants to deal with 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 divide(numerator, denominator) { return numerator / denominator; // FIXME denominator value might be 0 } See MITRE, CWE-546 - Suspicious Comment	CODE_SMELL	MAJOR	12
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	63
Only "while", "do", "for" and "switch" statements should be labelled	Any statement or block of statements can be identified by a label, but those labels should be used only on while, do-while, for and switch statements. Using labels in any other context leads to unstructured, confusing code. Noncompliant Code Example myLabel: if (i % 2 == 0) { // Noncompliant if (i == 12) { console.log("12"); break myLabel; } console.log("Odd number, but not 12"); } Compliant Solution myLabel: for (i = 0; i &lt; 10; i++) { // Compliant console.log("Loop"); break myLabel; }	CODE_SMELL	MAJOR	2
Function parameters with	The ability to define default values for function parameters can make a function easier to use.	CODE_SMELL	MAJOR	5

default values should be last	Default parameter values allow callers to specify as many or as few arguments as they want while getting the same functionality and minimizing boilerplate, wrapper code. But all function parameters with default values should be declared after the function parameters without default values. Otherwise, it makes it impossible for callers to take advantage of defaults; they must re-specify the defaulted values or pass undefined in order to "get to" the non-default parameters. Noncompliant Code Example function multiply(a = 1, b) { // Noncompliant return a*b; } var x = multiply(42); // returns NaN as b is undefined Compliant Solution function multiply(b, a = 1) { return a*b; } var x = multiply(42); // returns 42 as expected			
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, null, undefined, [], {}, true, false and "". Variables that start with an underscore (e.g. '_unused') are ignored. 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') CERT, MSC13-C. - Detect and remove unused values CERT, MSC56-J. - Detect and remove superfluous code and values	CODE_SMELL	MAJOR	133
Two branches in a conditional structure should not have exactly the same implementation	Having two cases in a switch statement or two branches in an if chain with the same implementation is at best duplicate code, and at worst a coding error. If the same logic is truly needed for both instances, then in an if chain they should be combined, or for a switch, one should fall through to the other. Noncompliant Code Example switch (i) { case 1: doFirstThing(); doSomething(); break; case 2: doSomethingDifferent(); break; case 3: // Noncompliant; duplicates case 1's implementation	CODE_SMELL	MAJOR	23

	doFirstThing(); doSomething(); break; default: doTheRest(); } if (a &gt;= 0 &amp;&amp; a &lt; 10) { doFirstThing(); doTheThing(); } else if (a &gt;= 10 &amp;&amp; a &lt; 20) { doTheOtherThing(); } else if (a &gt;= 20 &amp;&amp; a &lt; 50) { doFirstThing(); doTheThing(); // Noncompliant; duplicates first condition } else { doTheRest(); } Exceptions Blocks in an if chain that contain a single line of code are ignored, as are blocks in a switch statement that contain a single line of code with or without a following break. if (a == 1) { doSomething(); //no issue, usually this is done on purpose to increase the readability } else if (a == 2) { doSomethingElse(); } else { doSomething(); } But this exception does not apply to if chains without else-s, or to switch-es without default clauses when all branches have the same single line of code. In case of if chains with else-s, or of switch-es with default clauses, rule S3923 raises a bug. if (a == 1) { doSomething(); //Noncompliant, this might have been done on purpose but probably not } else if (a == 2) { doSomething(); }			
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	4
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)	CODE_SMELL	MAJOR	42

	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			
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	3
Ternary operators should not be nested	Just because you can do something, doesn't mean you should, and that's the case with nested ternary operations. Nesting ternary operators results in the kind of code that may seem clear as day when you write it, but six months later will leave maintainers (or worse - future you) scratching their heads and cursing. Instead, err on the side of clarity, and use another line to express the nested operation as a separate statement. Noncompliant Code Example function getReadableStatus(job) { return job.isRunning() ? "Running" : job.hasErrors() ? "Failed" : "Succeeded "; // Noncompliant } Compliant Solution function getReadableStatus(job) { if (job.isRunning()) { return "Running"; } return job.hasErrors() ? "Failed" : "Succeeded"; }	CODE_SMELL	MAJOR	77

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	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	11
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	7
Collection and array contents should be used	When a collection is populated but its contents are never used, then it is surely some kind of mistake. Either refactoring has rendered the collection moot, or an access is missing. This rule raises an issue when no methods are called on a collection other than those that add or remove values. Noncompliant Code Example function getLength(a, b, c) { const strings = []; // Noncompliant strings.push(a); strings.push(b); strings.push(c); return a.length + b.length + c.length; } Compliant Solution function getLength(a, b, c) { return a.length + b.length + c.length; }	CODE_SMELL	MAJOR	9

Functions should not have identical implementations	When two functions have the same implementation, either it was a mistake - something else was intended - or the duplication was intentional, but may be confusing to maintainers. In the latter case, the code should be refactored. Noncompliant Code Example function calculateCode() { doTheThing(); doOtherThing(); return code; } function getName() { // Noncompliant doTheThing(); doOtherThing(); return code; } Compliant Solution function calculateCode() { doTheThing(); doOtherThing(); return code; } function getName() { return calculateCode(); } Exceptions Functions with fewer than 3 lines are ignored.	CODE_SMELL	MAJOR	1187
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	3
Template literals should not be nested	Template literals (previously named "template strings") are an elegant way to build a string without using the + operator to make strings concatenation more readable. However, it's possible to build complex string literals by nesting together multiple template literals, and therefore lose readability and maintainability. In such situations, it's preferable to move the nested template into a separate statement. Noncompliant Code Example let color = "red"; let count = 3; let message = `I have ${color ? `${count} ${color}` : count} apples`; // Noncompliant; nested template strings not easy to read Compliant Solution let color = "red"; let count = 3; let apples = color ? `${count} ${color}` : count; let message = `I have ${apples} apples`;	CODE_SMELL	MAJOR	13
Shorthand promises should be used	When a Promise needs to only "resolve" or "reject", it's more efficient and readable to use the methods specially created for such use cases: Promise.resolve(value) and Promise.reject(error). Noncompliant Code Example let fulfilledPromise = new Promise(resolve =&gt; resolve(42)); let rejectedPromise = new Promise(function(resolve, reject) { reject('fail'); }); Compliant Solution let fulfilledPromise = Promise.resolve(42); let rejectedPromise = Promise.reject('fail');	CODE_SMELL	MAJOR	4

Functions should not have identical implementations	When two functions have the same implementation, either it was a mistake - something else was intended - or the duplication was intentional, but may be confusing to maintainers. In the latter case, the code should be refactored. Noncompliant Code Example function calculateCode() { doTheThing(); doOtherThing(); return code; } function getName() { // Noncompliant doTheThing(); doOtherThing(); return code; } Compliant Solution function calculateCode() { doTheThing(); doOtherThing(); return code; } function getName() { return calculateCode(); } Exceptions Functions with fewer than 3 lines are ignored.	CODE_SMELL	MAJOR	7
Class names should comply with a naming convention	Shared coding conventions allow teams to collaborate effectively. This rule allows to check that all class names (and interfaces for TypeScript) match a provided regular expression. Noncompliant Code Example With default provided regular expression ^[A-Z][a-zA-Z0-9]*$: class my_class {...} Compliant Solution class MyClass {...}	CODE_SMELL	MINOR	42
Extra semicolons should be removed	Extra semicolons (;) are usually introduced by mistake, for example because: It was meant to be replaced by an actual statement, but this was forgotten. There was a typo which lead the semicolon to be doubled, i.e. ;;. There was a misunderstanding about where semicolons are required or useful. Noncompliant Code Example var x = 1;; // Noncompliant function foo() { }; // Noncompliant Compliant Solution var x = 1; function foo() { } See CERT, MSC12-C. - Detect and remove code that has no effect or is never executed CERT, MSC51-J. - Do not place a semicolon immediately following an if, for, or while condition CERT, EXP15-C. - Do not place a semicolon on the same line as an if, for, or while statement	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	1

Return of boolean expressions should not be wrapped into an "if-then-else" statement	Return of boolean literal statements wrapped into if-then-else ones 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. Noncompliant Code Example if (expression) { return true; } else { return false; } Compliant Solution return expression; or return !!expression;	CODE_SMELL	MINOR	2
Unnecessary imports should be removed	There's no reason to import modules you don't use; and every reason not to: doing so needlessly increases the load. Noncompliant Code Example import A from 'a'; // Noncompliant, A isn't used import { B1 } from 'b'; console.log(B1); Compliant Solution import { B1 } from 'b'; console.log(B1);	CODE_SMELL	MINOR	6
A "while" loop should be used instead of a "for" loop	When only the condition expression is defined in a for loop, and the initialization and increment expressions are missing, a while loop should be used instead to increase readability. Noncompliant Code Example for (;condition;) { /*...*/ } Compliant Solution while (condition) { /*...*/ }	CODE_SMELL	MINOR	1
"switch" statements should have at least 3 "case" clauses	switch statements are useful when there are many different cases depending on the value of the same expression. For just one or two cases however, the code will be more readable with if statements. Noncompliant Code Example switch (variable) { case 0: doSomething(); break; default: doSomethingElse(); break; } Compliant Solution if (variable == 0) { doSomething(); } else { doSomethingElse(); }	CODE_SMELL	MINOR	35
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	44
Local variables should not be declared and then immediately returned or	Declaring a variable only to immediately return or throw it is a bad practice. Some developers argue that the practice improves code readability, because it enables them to explicitly name what is being returned. However, this variable is an internal implementation detail that is not exposed to the callers of the method. The method name should be	CODE_SMELL	MINOR	93

thrown	sufficient for callers to know exactly what will be returned. Noncompliant Code Example function computeDurationInMilliseconds() { var duration = (((hours * 60) + minutes) * 60 + seconds) * 1000 ; return duration; } Compliant Solution function computeDurationInMilliseconds() { return (((hours * 60) + minutes) * 60 + seconds) * 1000 ; }			
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 CERT, MET02-J. - Do not use deprecated or obsolete classes or methods	CODE_SMELL	MINOR	28
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	4
Default export names and file names should match	By convention, a file that exports only one class, function, or constant should be named for that class, function or constant. Anything else may confuse maintainers. Noncompliant Code Example // file path: myclass.js -- Noncompliant class MyClass { // ... } export default MyClass; Compliant Solution // file path: MyClass.js class MyClass { // ... } export default MyClass; Exceptions Case, underscores (_) and dashes (-) are ignored from the name comparison.	CODE_SMELL	MINOR	12
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	CODE_SMELL	MINOR	6

	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.			
Imports from the same modules should be merged	Multiple imports from the same module should be merged together to improve readability. Noncompliant Code Example import { B1 } from 'b'; import { B2 } from 'b'; // Noncompliant Compliant Solution import { B1, B2 } from 'b';	CODE_SMELL	MINOR	115
"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. 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	227
"await" should not be used redundantly	An async function always wraps the return value in a Promise. Using return await is therefore redundant. Noncompliant Code Example async function foo() { // ... } async function bar() { // ... return await foo(); // Noncompliant } Compliant Solution async function foo() { // ... } async function bar() { // ... return foo(); }	CODE_SMELL	MINOR	53
Origins should be verified during cross-origin communications	Browsers allow message exchanges between Window objects of different origins. Because any window can send / receive messages from other window it is important to verify the sender's / receiver's identity: When sending message with postMessage method, the identity's receiver should be defined (the wildcard keyword (*) should not be used). When receiving message with message event, the sender's identity should be verified using the origin and possibly source properties. Noncompliant Code Example When sending message: var iframe = document.getElementById("testiframe"); iframe.contentWindow.postMessage("secret", "*"); // Noncompliant: * is used When receiving message: window.addEventListener("message", function(event) { // Noncompliant: no checks are done on the origin property. console.log(event.data); }); Compliant Solution	VULNERABILITY	CRITICAL	13

	When sending message: var iframe = document.getElementById("testsecureiframe"); iframe.contentWindow.postMessage("hello", "https://secure.example.com"); // Compliant When receiving message: window.addEventListener("message", function(event) { if (event.origin !== "http://example.org") // Compliant return; console.log(event.data) }); See OWASP Top 10 2017 Category A3 - Broken Authentication and Session Management developer.mozilla.org - postMessage API

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	1	0	0
XPath Injection	0	0	0
Authentication	0	0	0
Weak Cryptography	0	15	0
Denial of Service (DoS)	0	0	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	0
Encryption of Sensitive Data	0	0	0
Traceability	0	0	0
Buffer Overflow	0	0	0
File Manipulation	0	0	0
Code Injection (RCE)	0	4	0

REACT JS

Cross-Site Scripting (XSS)	0	0	0
Command Injection	0	0	2
Path Traversal Injection	0	0	0
HTTP Response Splitting	0	0	0
Others	119	0	0

SECURITY HOTSPOTS LIST

Category	Name	Priority	Severity	Count
Weak Cryptography	Using pseudorandom number generators (PRNGs) is security-sensitive	MEDIUM	CRITICAL	15
Others	Using clear-text protocols is security-sensitive	LOW	CRITICAL	26
Insecure Configuration	Having a permissive Cross-Origin Resource Sharing policy is security-sensitive	LOW	MINOR	1
Others	Disabling resource integrity features is security-sensitive	LOW	MINOR	88
Code Injection (RCE)	Dynamically executing code is security-sensitive	MEDIUM	CRITICAL	4
Command Injection	Using shell interpreter when executing OS commands is security-sensitive	HIGH	MAJOR	2
Others	Disclosing fingerprints from web application technologies is security-sensitive	LOW	MINOR	5