|  |
| --- |
| dynamsoft-dlrjs-src  Version 1.0.0  Code analysis |

|  |
| --- |
| **By: default**  **2023-07-25** |

# Content

[Content 1](#_Toc79573867)

[Introduction 2](#_Toc79573868)

[Configuration 2](#_Toc79573869)

[Synthesis 3](#_Toc79573870)

[Analysis Status 3](#_Toc79573871)

[Quality gate status 3](#_Toc79573872)

[Metrics 3](#_Toc79573873)

[Tests 3](#_Toc79573874)

[Detailed technical debt 3](#_Toc79573875)

[Metrics Range 5](#_Toc79573876)

[Volume 5](#_Toc79573877)

[Issues 6](#_Toc79573878)

[Charts 6](#_Toc79573879)

[Issues count by severity and type 8](#_Toc79573880)

[Issues List 8](#_Toc79573881)

[Security Hotspots 9](#_Toc79573882)

[Security hotspots count by category and priority 9](#_Toc79573883)

[Security hotspots List 9](#_Toc79573884)

# Introduction

This document contains results of the code analysis of dynamsoft-dlrjs-src.

# Configuration

* Quality Profiles
  + Names: Sonar way [TypeScript];
  + Files: AX9sVF1puUAFS9ToL78Z.json;
* Quality Gate
  + Name: Sonar way
  + File: Sonar way.xml

# Synthesis

## Analysis Status

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

## Quality gate status

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

|  |  |
| --- | --- |
| Metric | Value |
| Reliability Rating on New Code | OK |
| Security Rating on New Code | OK |
| Maintainability Rating on New Code | OK |
| Duplicated Lines (%) on New Code | OK |

## Metrics

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Coverage | Duplication | Comment  density | Median number of lines of code per file | Adherence to coding standard |
| 0.0 % | **8.4 %** | **17.7 %** | **10.0** | **98.9 %** |

## Tests

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

## Detailed technical debt

|  |  |  |  |
| --- | --- | --- | --- |
| Reliability | Security | Maintainability | Total |
| 0d 0h 5min | - | 3d 1h 48min | 3d 1h 53min |

## Metrics Range

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Cyclomatic  Complexity | Cognitive  Complexity | Lines of code per file | Comment  density (%) | Coverage | Duplication (%) |
| Min | 0.0 | 0.0 | 4.0 | 0.0 | 0.0 | 0.0 |
| Max | 914.0 | 1517.0 | 3525.0 | 69.0 | 0.0 | 10.9 |

## Volume

|  |  |
| --- | --- |
| Language | Number |
| TypeScript | 3525 |
| Total | 3525 |

# Issues

## Charts

## Issues count by severity and type

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Type / Severity | INFO | MINOR | MAJOR | CRITICAL | BLOCKER |
| BUG | 0 | 0 | 1 | 0 | 0 |
| VULNERABILITY | 0 | 0 | 0 | 0 | 0 |
| CODE\_SMELL | 9 | 52 | 53 | 20 | 0 |

## Issues List

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Description | Type | Severity | Number |
| Loops with at most one iteration should be refactored | A loop with at most one iteration is equivalent to the use of an if statement to conditionally execute one piece of code. No developer expects to find such a use of a loop statement. If the initial intention of the author was really to conditionally execute one piece of code, an if statement should be used instead. At worst that was not the initial intention of the author and so the body of the loop should be fixed to use the nested return, break or throw statements in a more appropriate way. Noncompliant Code Example for (let i = 0; i &lt; 10; i++) { // noncompliant, loop only executes once console.log("i is " + i); break; } ... for (let i = 0; i &lt; 10; i++) { // noncompliant, loop only executes once if (i == x) { break; } else { console.log("i is " + i); return; } } Compliant Solution for (let i = 0; i &lt; 10; i++) { console.log("i is " + i); } ... for (let i = 0; i &lt; 10; i++) { if (i == x) { break; } else { console.log("i is " + i); } } | BUG | MAJOR | 1 |
| 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 | CRITICAL | 1 |
| 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 | 19 |
| 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 | 9 |
| 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 | 1 |
| 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. | CODE\_SMELL | MAJOR | 8 |
| 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 | 20 |
| Two branches in a conditional structure should not have exactly the same implementation |  | 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 | 3 |
| 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 | 11 |
| 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 | 1 |
| 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 | 8 |
| 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() { } | CODE\_SMELL | MINOR | 3 |
| 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 | 1 |
| Local variables should not be declared and then immediately returned or thrown | 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 sufficient for callers to know exactly what will be returned. Noncompliant Code Example function computeDurationInMilliseconds(hours, minutes, seconds) { let duration = (((hours \* 60) + minutes) \* 60 + seconds ) \* 1000; return duration; } Compliant Solution function computeDurationInMilliseconds(hours, minutes, seconds) { return (((hours \* 60) + minutes) \* 60 + seconds ) \* 1000; } | CODE\_SMELL | MINOR | 4 |
| Wrapper objects should not be used for primitive types | The use of wrapper objects for primitive types is gratuitous, confusing and dangerous. If you use a wrapper object constructor for type conversion, just remove the new keyword, and you’ll get a primitive value automatically. If you use a wrapper object as a way to add properties to a primitive, you should re-think the design. Such uses are considered bad practice, and should be refactored. Noncompliant Code Example let x = new Number("0"); if (x) { alert('hi'); // Shows 'hi'. } Compliant Solution let x = Number("0"); if (x) { alert('hi'); } | CODE\_SMELL | MINOR | 1 |
| 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 | 2 |
| "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 | 3 |
| Redundant casts and non-null assertions should be avoided | The TypeScript compiler automatically casts a variable to the relevant type inside conditionals where it is possible to infer the type (because typeof, instanceof, etc was used). This compiler feature makes casts and not-null assertions unnecessary. Noncompliant Code Example function getName(x?: string | UserName) { if (x) { console.log("Getting name for " + x!); // Noncompliant if (typeof x === "string") return (x as string); // Noncompliant else return (x as UserName).name; // Noncompliant } return "NoName"; } Compliant Solution function getName(x?: string | UserName) { if (x) { console.log("Getting name for " + x); if (typeof x === "string") return x; else return x.name; } return "NoName"; } | CODE\_SMELL | MINOR | 10 |
| "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 | 25 |
| 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 | 3 |

# 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 | 0 | 0 | 0 |
| XPath Injection | 0 | 0 | 0 |
| Authentication | 0 | 0 | 0 |
| Weak Cryptography | 0 | 0 | 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 |
| SQL Injection | 0 | 0 | 0 |
| Buffer Overflow | 0 | 0 | 0 |
| File Manipulation | 0 | 0 | 0 |
| Code Injection (RCE) | 0 | 0 | 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 | 0 | 0 | 0 |

## Security hotspots List