|  |
| --- |
| secure-software-development  Version 0.0.1-SNAPSHOT  Code analysis |

|  |
| --- |
| **By: default**  **2024-01-14** |

# 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 secure-software-development.

Secure Software Development

# Configuration

* Quality Profiles
  + Names: Sonar way [Java]; Sonar way [XML];
  + Files: AYzUz5-m4-8Yp4SfXBny.json; AYzUz6C34-8Yp4SfXBwo.json;
* Quality Gate
  + Name: Sonar way
  + File: Sonar way.xml

# Synthesis

## Analysis Status

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

## Quality gate status

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



## Metrics

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Coverage | Duplication | Comment  density | Median number of lines of code per file | Adherence to coding standard |
| 0.0 % | **1.8 %** | **0.8 %** | **38.0** | **98.3 %** |

## Tests

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Total | Success Rate | Skipped | Errors | Failures |
| 1 | **100.0 %** | **0** | **0** | **0** |

## Detailed technical debt

|  |  |  |  |
| --- | --- | --- | --- |
| Reliability | Security | Maintainability | Total |
| 0d 0h 20min | - | 0d 5h 20min | 0d 5h 40min |

## Metrics Range

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Cyclomatic  Complexity | Cognitive  Complexity | Lines of code per file | Comment  density (%) | Coverage | Duplication (%) |
| Min | 1.0 | 0.0 | 10.0 | 0.0 | 0.0 | 0.0 |
| Max | 218.0 | 72.0 | 1486.0 | 19.5 | 0.0 | 33.3 |

## Volume

|  |  |
| --- | --- |
| Language | Number |
| Java | 1486 |
| XML | 90 |
| Total | 1576 |

# Issues

## Charts

## Issues count by severity and type

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Type / Severity | INFO | MINOR | MAJOR | CRITICAL | BLOCKER |
| BUG | 0 | 0 | 1 | 0 | 2 |
| VULNERABILITY | 0 | 0 | 0 | 0 | 0 |
| CODE\_SMELL | 0 | 14 | 31 | 5 | 2 |

## Issues List

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Description | Type | Severity | Number |
| Resources should be closed | Connections, streams, files, and other classes that implement the Closeable interface or its super-interface, AutoCloseable, needs to be closed after use. Further, that close call must be made in a finally block otherwise an exception could keep the call from being made. Preferably, when class implements AutoCloseable, resource should be created using "try-with-resources" pattern and will be closed automatically. Failure to properly close resources will result in a resource leak which could bring first the application and then perhaps the box the application is on to their knees. Noncompliant Code Example private void readTheFile() throws IOException { Path path = Paths.get(this.fileName); BufferedReader reader = Files.newBufferedReader(path, this.charset); // ... reader.close(); // Noncompliant // ... Files.lines("input.txt").forEach(System.out::println); // Noncompliant: The stream needs to be closed } private void doSomething() { OutputStream stream = null; try { for (String property : propertyList) { stream = new FileOutputStream("myfile.txt"); // Noncompliant // ... } } catch (Exception e) { // ... } finally { stream.close(); // Multiple streams were opened. Only the last is closed. } } Compliant Solution private void readTheFile(String fileName) throws IOException { Path path = Paths.get(fileName); try (BufferedReader reader = Files.newBufferedReader(path, StandardCharsets.UTF\_8)) { reader.readLine(); // ... } // .. try (Stream&lt;String&gt; input = Files.lines("input.txt")) { input.forEach(System.out::println); } } private void doSomething() { OutputStream stream = null; try { stream = new FileOutputStream("myfile.txt"); for (String property : propertyList) { // ... } } catch (Exception e) { // ... } finally { stream.close(); } } Exceptions Instances of the following classes are ignored by this rule because close has no effect: java.io.ByteArrayOutputStream java.io.ByteArrayInputStream java.io.CharArrayReader java.io.CharArrayWriter java.io.StringReader java.io.StringWriter Java 7 introduced the try-with-resources statement, which implicitly closes Closeables. All resources opened in a try-with-resources statement are ignored by this rule. try (BufferedReader br = new BufferedReader(new FileReader(fileName))) { //... } catch ( ... ) { //... } See MITRE, CWE-459 - Incomplete Cleanup MITRE, CWE-772 - Missing Release of Resource after Effective Lifetime CERT, FIO04-J. - Release resources when they are no longer needed CERT, FIO42-C. - Close files when they are no longer needed Try With Resources | BUG | BLOCKER | 2 |
| Optional value should only be accessed after calling isPresent() |  | BUG | MAJOR | 1 |
| Methods and field names should not be the same or differ only by capitalization | Looking at the set of methods in a class, including superclass methods, and finding two methods or fields that differ only by capitalization is confusing to users of the class. It is similarly confusing to have a method and a field which differ only in capitalization or a method and a field with exactly the same name and visibility. In the case of methods, it may have been a mistake on the part of the original developer, who intended to override a superclass method, but instead added a new method with nearly the same name. Otherwise, this situation simply indicates poor naming. Method names should be action-oriented, and thus contain a verb, which is unlikely in the case where both a method and a member have the same name (with or without capitalization differences). However, renaming a public method could be disruptive to callers. Therefore renaming the member is the recommended action. Noncompliant Code Example public class Car{ public DriveTrain drive; public void tearDown(){...} public void drive() {...} // Noncompliant; duplicates field name } public class MyCar extends Car{ public void teardown(){...} // Noncompliant; not an override. It it really what's intended? public void drivefast(){...} public void driveFast(){...} //Huh? } Compliant Solution public class Car{ private DriveTrain drive; public void tearDown(){...} public void drive() {...} // field visibility reduced } public class MyCar extends Car{ @Override public void tearDown(){...} public void drivefast(){...} public void driveReallyFast(){...} } | CODE\_SMELL | BLOCKER | 1 |
| Tests should include assertions |  | CODE\_SMELL | BLOCKER | 1 |
| Methods should not be empty | There are several reasons for a method not to have a method 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 UnsupportedOperationException should be thrown. The method is an intentionally-blank override. In this case a nested comment should explain the reason for the blank override. Noncompliant Code Example public void doSomething() { } public void doSomethingElse() { } Compliant Solution @Override public void doSomething() { // Do nothing because of X and Y. } @Override public void doSomethingElse() { throw new UnsupportedOperationException(); } Exceptions Default (no-argument) constructors are ignored when there are other constructors in the class, as are empty methods in abstract classes. public abstract class Animal { void speak() { // default implementation ignored } } | CODE\_SMELL | CRITICAL | 2 |
| String literals should not be duplicated | Duplicated string literals make the process of refactoring error-prone, since you must be sure to update all occurrences. On the other hand, constants can be referenced from many places, but only need to be updated in a single place. Noncompliant Code Example With the default threshold of 3: public void run() { prepare("action1"); // Noncompliant - "action1" is duplicated 3 times execute("action1"); release("action1"); } @SuppressWarning("all") // Compliant - annotations are excluded private void method1() { /\* ... \*/ } @SuppressWarning("all") private void method2() { /\* ... \*/ } public String method3(String a) { System.out.println("'" + a + "'"); // Compliant - literal "'" has less than 5 characters and is excluded return ""; // Compliant - literal "" has less than 5 characters and is excluded } Compliant Solution private static final String ACTION\_1 = "action1"; // Compliant public void run() { prepare(ACTION\_1); // Compliant execute(ACTION\_1); release(ACTION\_1); } Exceptions To prevent generating some false-positives, literals having less than 5 characters are excluded. | CODE\_SMELL | CRITICAL | 3 |
| Source files should not have any duplicated blocks | An issue is created on a file as soon as there is at least one block of duplicated code on this file | CODE\_SMELL | MAJOR | 2 |
| Unused "private" fields should be removed | If a private field is declared but not used in the program, it can be considered dead code and should therefore be removed. This will improve maintainability because developers will not wonder what the variable is used for. Note that this rule does not take reflection into account, which means that issues will be raised on private fields that are only accessed using the reflection API. Noncompliant Code Example public class MyClass { private int foo = 42; public int compute(int a) { return a \* 42; } } Compliant Solution public class MyClass { public int compute(int a) { return a \* 42; } } Exceptions The Java serialization runtime associates with each serializable class a version number, called serialVersionUID, which is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization. A serializable class can declare its own serialVersionUID explicitly by declaring a field named serialVersionUID that must be static, final, and of type long. By definition those serialVersionUID fields should not be reported by this rule: public class MyClass implements java.io.Serializable { private static final long serialVersionUID = 42L; } Moreover, this rule doesn't raise any issue on annotated fields. | CODE\_SMELL | MAJOR | 19 |
| Local variables should not shadow class fields | 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. Noncompliant Code Example class Foo { public int myField; public void doSomething() { int myField = 0; ... } } 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 | 1 |
| Utility classes should not have public constructors | Utility classes, which are collections of static members, are not meant to be instantiated. Even abstract utility classes, which can be extended, should not have public constructors. Java adds an implicit public constructor to every class which does not define at least one explicitly. Hence, at least one non-public constructor should be defined. Noncompliant Code Example class StringUtils { // Noncompliant public static String concatenate(String s1, String s2) { return s1 + s2; } } Compliant Solution class StringUtils { // Compliant private StringUtils() { throw new IllegalStateException("Utility class"); } public static String concatenate(String s1, String s2) { return s1 + s2; } } Exceptions When class contains public static void main(String[] args) method it is not considered as utility class and will be ignored by this rule. | CODE\_SMELL | MAJOR | 1 |
| Generic exceptions should never be thrown | Using such generic exceptions as Error, RuntimeException, Throwable, and Exception prevents calling methods from handling true, system-generated exceptions differently than application-generated errors. Noncompliant Code Example public void foo(String bar) throws Throwable { // Noncompliant throw new RuntimeException("My Message"); // Noncompliant } Compliant Solution public void foo(String bar) { throw new MyOwnRuntimeException("My Message"); } Exceptions Generic exceptions in the signatures of overriding methods are ignored, because overriding method has to follow signature of the throw declaration in the superclass. The issue will be raised on superclass declaration of the method (or won't be raised at all if superclass is not part of the analysis). @Override public void myMethod() throws Exception {...} Generic exceptions are also ignored in the signatures of methods that make calls to methods that throw generic exceptions. public void myOtherMethod throws Exception { doTheThing(); // this method throws Exception } See MITRE, CWE-397 - Declaration of Throws for Generic Exception CERT, ERR07-J. - Do not throw RuntimeException, Exception, or Throwable | CODE\_SMELL | MAJOR | 1 |
| "@Override" should be used on overriding and implementing methods | Using the @Override annotation is useful for two reasons : It elicits a warning from the compiler if the annotated method doesn't actually override anything, as in the case of a misspelling. It improves the readability of the source code by making it obvious that methods are overridden. Noncompliant Code Example class ParentClass { public boolean doSomething(){...} } class FirstChildClass extends ParentClass { public boolean doSomething(){...} // Noncompliant } Compliant Solution class ParentClass { public boolean doSomething(){...} } class FirstChildClass extends ParentClass { @Override public boolean doSomething(){...} // Compliant } Exceptions This rule is relaxed when overriding a method from the Object class like toString(), hashCode(), ... | CODE\_SMELL | MAJOR | 2 |
| A field should not duplicate the name of its containing class | It's confusing to have a class member with the same name (case differences aside) as its enclosing class. This is particularly so when you consider the common practice of naming a class instance for the class itself. Best practice dictates that any field or member with the same name as the enclosing class be renamed to be more descriptive of the particular aspect of the class it represents or holds. Noncompliant Code Example public class Foo { private String foo; public String getFoo() { } } Foo foo = new Foo(); foo.getFoo() // what does this return? Compliant Solution public class Foo { private String name; public String getName() { } } //... Foo foo = new Foo(); foo.getName() Exceptions When the type of the field is the containing class and that field is static, no issue is raised to allow singletons named like the type. public class Foo { ... private static Foo foo; public Foo getInstance() { if(foo==null) { foo = new Foo(); } return foo; } ... } | 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, null, true, false and "". 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 | 1 |
| 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 |
| Method names should comply with a naming convention | Shared naming conventions allow teams to collaborate efficiently. This rule checks that all method names match a provided regular expression. Noncompliant Code Example With default provided regular expression ^[a-z][a-zA-Z0-9]\*$: public int DoSomething(){...} Compliant Solution public int doSomething(){...} Exceptions Overriding methods are excluded. @Override public int Do\_Something(){...} | CODE\_SMELL | MINOR | 1 |
| Empty statements should be removed | Empty statements, i.e. ;, 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. ;;. Noncompliant Code Example void doSomething() { ; // Noncompliant - was used as a kind of TODO marker } void doSomethingElse() { System.out.println("Hello, world!");; // Noncompliant - double ; ... } Compliant Solution void doSomething() {} void doSomethingElse() { System.out.println("Hello, world!"); ... for (int i = 0; i &lt; 3; i++) ; // compliant if unique statement of a loop ... } 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 |
| Unnecessary imports should be removed | The imports part of a file should be handled by the Integrated Development Environment (IDE), not manually by the developer. Unused and useless imports should not occur if that is the case. Leaving them in reduces the code's readability, since their presence can be confusing. Noncompliant Code Example package my.company; import java.lang.String; // Noncompliant; java.lang classes are always implicitly imported import my.company.SomeClass; // Noncompliant; same-package files are always implicitly imported import java.io.File; // Noncompliant; File is not used import my.company2.SomeType; import my.company2.SomeType; // Noncompliant; 'SomeType' is already imported class ExampleClass { public String someString; public SomeType something; } Exceptions Imports for types mentioned in comments, such as Javadocs, are ignored. | CODE\_SMELL | MINOR | 6 |
| "throws" declarations should not be superfluous | An exception in a throws declaration in Java is superfluous if it is: listed multiple times a subclass of another listed exception completely unnecessary because the declared exception type cannot actually be thrown Noncompliant Code Example void foo() throws MyException, MyException {} // Noncompliant; should be listed once void bar() throws Throwable, Exception {} // Noncompliant; Exception is a subclass of Throwable Compliant Solution void foo() throws MyException {} void bar() throws Throwable {} Exceptions The rule will not raise any issue for exceptions that cannot be thrown from the method body: in overriding and implementation methods in interface default methods in non-private methods that only throw, have empty bodies, or a single return statement. in overridable methods (non-final, or not member of a final class, non-static, non-private), if the exception is documented with a proper JavaDoc Also, the rule won't raise issues on RuntimeException, or one of its descendants, because explicating runtime exceptions which could be thrown can ultimately help the method's users, and can even be considered as good practice. class A extends B { @Override void doSomething() throws IOException { compute(a); } public void foo() throws IOException {} public void qix() throws MyRuntimeException {} protected void bar() throws IOException { throw new UnsupportedOperationException("This method should be implemented in subclasses"); } Object foobar(String s) throws IOException { return null; } /\*\* \* @throws IOException Overriding classes may throw this exception if they print values into a file \*/ protected void print() throws IOException { // no issue, method is overridable and the exception has proper javadoc System.out.println("foo"); } } | CODE\_SMELL | MINOR | 1 |
| Collection.isEmpty() should be used to test for emptiness | Using Collection.size() to test for emptiness works, but using Collection.isEmpty() makes the code more readable and can be more performant. The time complexity of any isEmpty() method implementation should be O(1) whereas some implementations of size() can be O(n). Noncompliant Code Example if (myCollection.size() == 0) { // Noncompliant /\* ... \*/ } Compliant Solution if (myCollection.isEmpty()) { /\* ... \*/ } | CODE\_SMELL | MINOR | 1 |
| Field names should comply with a naming convention | Sharing some naming conventions is a key point to make it possible for a team to efficiently collaborate. This rule allows to check that field names match a provided regular expression. Noncompliant Code Example With the default regular expression ^[a-z][a-zA-Z0-9]\*$: class MyClass { private int my\_field; } Compliant Solution class MyClass { private int myField; } | CODE\_SMELL | MINOR | 1 |
| Unused local variables should be removed | If a local variable 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 is used for. Noncompliant Code Example public int numberOfMinutes(int hours) { int seconds = 0; // seconds is never used return hours \* 60; } Compliant Solution public int numberOfMinutes(int hours) { return hours \* 60; } | 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 public long computeDurationInMilliseconds() { long duration = (((hours \* 60) + minutes) \* 60 + seconds ) \* 1000 ; return duration; } public void doSomething() { RuntimeException myException = new RuntimeException(); throw myException; } Compliant Solution public long computeDurationInMilliseconds() { return (((hours \* 60) + minutes) \* 60 + seconds ) \* 1000 ; } public void doSomething() { throw new RuntimeException(); } | CODE\_SMELL | MINOR | 1 |
| Lambdas should be replaced with method references |  | CODE\_SMELL | MINOR | 1 |

# 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 | 21 | 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 | 1 |
| Open Redirect | 0 | 0 | 0 |
| Permission | 0 | 0 | 0 |
| SQL Injection | 0 | 0 | 21 |
| 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 | 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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Category | Name | Priority | Severity | Count |
| Insecure Configuration | Delivering code in production with debug features activated is security-sensitive | LOW | MINOR | 21 |
| SQL Injection | Formatting SQL queries is security-sensitive | HIGH | MAJOR | 21 |
| Cross-Site Request Forgery (CSRF) | Disabling CSRF protections is security-sensitive | HIGH | CRITICAL | 1 |