|  |
| --- |
| CupCake  Version 1.0-SNAPSHOT  Code analysis |

|  |
| --- |
| By: Administrator  2019-10-28 |

# Content

[Content 1](#_Toc7772423)

[Introduction 2](#_Toc7772424)

[Configuration 2](#_Toc7772425)

[Synthesis 2](#_Toc7772426)

[Metrics 2](#_Toc7772427)

[Volume 3](#_Toc7772428)

[Issues count by severity and type 3](#_Toc7772429)

[Charts 3](#_Toc7772430)

[Issues 3](#_Toc7772431)

# Introduction

This document contains results of the code analysis of CupCake.

# Configuration

* Quality Profiles
  + Names: Sonar way [Java]; Sonar way [JSP];
  + Files: AW4ScWgUSRN48LnyzVkj.json; AW4ScWtpSRN48LnyzVrm.json;
* Quality Gate
  + Name: Sonar way
  + File: Sonar way.xml

# Synthesis

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Quality Gate | Reliability | Security | Maintainability | Coverage | Duplication |
| OK | E | B | A | 70.6 % | 1.8 % |

# Metrics

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Cyclomatic  Complexity | Cognitive  Complexity | Lines of code per file | Comment  density (%) | Coverage | Duplication (%) |
| Min | 0.0 | 0.0 | 5.0 | 0.0 | 0.0 | 0.0 |
| Max | 291.0 | 180.0 | 2512.0 | 62.5 | 100.0 | 20.0 |

# Volume

|  |  |
| --- | --- |
| Language | Number |
| Java | 1547 |
| JSP | 965 |
| Total | 2512 |

# Issues count by severity and type

|  |  |  |
| --- | --- | --- |
| Type | Severity | Number |
| VULNERABILITY | BLOCKER | 0 |
| VULNERABILITY | CRITICAL | 0 |
| VULNERABILITY | MAJOR | 0 |
| VULNERABILITY | MINOR | 7 |
| VULNERABILITY | INFO | 0 |
| BUG | BLOCKER | 26 |
| BUG | CRITICAL | 0 |
| BUG | MAJOR | 39 |
| BUG | MINOR | 4 |
| BUG | INFO | 0 |
| CODE\_SMELL | BLOCKER | 12 |
| CODE\_SMELL | CRITICAL | 11 |
| CODE\_SMELL | MAJOR | 26 |
| CODE\_SMELL | MINOR | 59 |
| CODE\_SMELL | INFO | 9 |
| SECURITY\_HOTSPOT | BLOCKER | 0 |
| SECURITY\_HOTSPOT | CRITICAL | 0 |
| SECURITY\_HOTSPOT | MAJOR | 0 |
| SECURITY\_HOTSPOT | MINOR | 0 |
| SECURITY\_HOTSPOT | INFO | 0 |

# Charts

# Issues

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 it's 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 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 | 25 |
| "PreparedStatement" and "ResultSet" methods should be called with valid indices | The parameters in a PreparedStatement are numbered from 1, not 0, so using any "set" method of a PreparedStatement with a number less than 1 is a bug, as is using an index higher than the number of parameters. Similarly, ResultSet indices also start at 1, rather than 0 Noncompliant Code Example PreparedStatement ps = con.prepareStatement("SELECT fname, lname FROM employees where hireDate &gt; ? and salary &lt; ?"); ps.setDate(0, date); // Noncompliant ps.setDouble(3, salary); // Noncompliant ResultSet rs = ps.executeQuery(); while (rs.next()) { String fname = rs.getString(0); // Noncompliant // ... } Compliant Solution PreparedStatement ps = con.prepareStatement("SELECT fname, lname FROM employees where hireDate &gt; ? and salary &lt; ?"); ps.setDate(1, date); ps.setDouble(2, salary); ResultSet rs = ps.executeQuery(); while (rs.next()) { String fname = rs.getString(1); // ... } | BUG | BLOCKER | 1 |
| "<!DOCTYPE>" declarations should appear before "<html>" tags |  | BUG | MAJOR | 12 |
| "<html>" element should have a language attribute |  | BUG | MAJOR | 12 |
| "<th>" tags should have "id" or "scope" attributes |  | BUG | MAJOR | 11 |
| Non-serializable objects should not be stored in "HttpSession" objects | If you have no intention of writting an HttpSession object to file, then storing non-serializable objects in it may not seem like a big deal. But whether or not you explicitly serialize the session, it may be written to disk anyway, as the server manages its memory use in a process called "passivation". Further, some servers automatically write their active sessions out to file at shutdown &amp; deserialize any such sessions at startup. The point is, that even though HttpSession does not extend Serializable, you must nonetheless assume that it will be serialized, and understand that if you've stored non-serializable objects in the session, errors will result. Noncompliant Code Example public class Address { //... } //... HttpSession session = request.getSession(); session.setAttribute("address", new Address()); // Noncompliant; Address isn't serializable See MITRE, CWE-579 - J2EE Bad Practices: Non-serializable Object Stored in Session | BUG | MAJOR | 4 |
| "<table>" tags should have a description |  | BUG | MINOR | 3 |
| Method parameters, caught exceptions and foreach variables' initial values should not be ignored | While it is technically correct to assign to parameters from within method bodies, doing so before the parameter value is read is likely a bug. Instead, initial values of parameters, caught exceptions, and foreach parameters should be, if not treated as final, then at least read before reassignment. Noncompliant Code Example public void doTheThing(String str, int i, List&lt;String&gt; strings) { str = Integer.toString(i); // Noncompliant for (String s : strings) { s = "hello world"; // Noncompliant } } | BUG | MINOR | 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 | 2 |
| Tests should include assertions |  | CODE\_SMELL | BLOCKER | 10 |
| 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 | 4 |
| 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 | 5 |
| Cognitive Complexity of methods should not be too high | Cognitive Complexity is a measure of how hard the control flow of a method is to understand. Methods with high Cognitive Complexity will be difficult to maintain. See Cognitive Complexity | CODE\_SMELL | CRITICAL | 2 |
| Track uses of "TODO" tags | TODO tags are commonly used to mark places where some more code is required, but which the developer wants to implement later. Sometimes the developer will not have the time or will simply forget to get back to that tag. This rule is meant to track those tags and to ensure that they do not go unnoticed. See MITRE, CWE-546 - Suspicious Comment | CODE\_SMELL | INFO | 4 |
| 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 void doSomething() { // TODO } See MITRE, CWE-546 - Suspicious Comment | CODE\_SMELL | INFO | 5 |
| 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 | 8 |
| 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 (int i = 0; i &lt; 42; i++){} // Empty on purpose or missing piece of code ? Exceptions When a block contains a comment, this block is not considered to be empty unless it is a synchronized block. synchronized blocks are still considered empty even with comments because they can still affect program flow. | 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 |
| 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 |
| Try-catch blocks should not be nested | Nesting try/catch blocks severely impacts the readability of source code because it makes it too difficult to understand which block will catch which exception. | CODE\_SMELL | MAJOR | 3 |
| Empty arrays and collections should be returned instead of null | Returning null instead of an actual array or collection forces callers of the method to explicitly test for nullity, making them more complex and less readable. Moreover, in many cases, null is used as a synonym for empty. Noncompliant Code Example public static List&lt;Result&gt; getResults() { return null; // Noncompliant } public static Result[] getResults() { return null; // Noncompliant } public static void main(String[] args) { Result[] results = getResults(); if (results != null) { // Nullity test required to prevent NPE for (Result result: results) { /\* ... \*/ } } } Compliant Solution public static List&lt;Result&gt; getResults() { return Collections.emptyList(); // Compliant } public static Result[] getResults() { return new Result[0]; } public static void main(String[] args) { for (Result result: getResults()) { /\* ... \*/ } } See CERT, MSC19-C. - For functions that return an array, prefer returning an empty array over a null value CERT, MET55-J. - Return an empty array or collection instead of a null value for methods that return an array or collection | CODE\_SMELL | MAJOR | 2 |
| Unused method parameters should be removed | Unused parameters are misleading. Whatever the values passed to such parameters, the behavior will be the same. Noncompliant Code Example void doSomething(int a, int b) { // "b" is unused compute(a); } Compliant Solution void doSomething(int a) { compute(a); } Exceptions The rule will not raise issues for unused parameters: that are annotated with @javax.enterprise.event.Observes in overrides and implementation methods in interface default methods in non-private methods that only throw or that have empty bodies in annotated methods, unless the annotation is @SuppressWarning("unchecked") or @SuppressWarning("rawtypes"), in which case the annotation will be ignored in overridable methods (non-final, or not member of a final class, non-static, non-private), if the parameter is documented with a proper javadoc. @Override void doSomething(int a, int b) { // no issue reported on b compute(a); } public void foo(String s) { // designed to be extended but noop in standard case } protected void bar(String s) { //open-closed principle } public void qix(String s) { throw new UnsupportedOperationException("This method should be implemented in subclasses"); } /\*\* \* @param s This string may be use for further computation in overriding classes \*/ protected void foobar(int a, String s) { // no issue, method is overridable and unused parameter has proper javadoc compute(a); } See CERT, MSC12-C. - Detect and remove code that has no effect or is never executed | CODE\_SMELL | MAJOR | 2 |
| Dead stores 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 | 7 |
| Static fields should not be updated in constructors | Assigning a value to a static field in a constructor could cause unreliable behavior at runtime since it will change the value for all instances of the class. Instead remove the field's static modifier, or initialize it statically. Noncompliant Code Example public class Person { static Date dateOfBirth; static int expectedFingers; public Person(date birthday) { dateOfBirth = birthday; // Noncompliant; now everyone has this birthday expectedFingers = 10; // Noncompliant } } Compliant Solution public class Person { Date dateOfBirth; static int expectedFingers = 10; public Person(date birthday) { dateOfBirth = birthday; } } | CODE\_SMELL | MAJOR | 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 |
| "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 a RuntimeException, or one of its descendants 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 void baz() throws RuntimeException {} // Noncompliant; RuntimeException can always be thrown Compliant Solution void foo() throws MyException {} void bar() throws Throwable {} void baz() {} 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. class A extends B { @Override void doSomething() throws IOException { compute(a); } public void foo() throws IOException {} 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 | 9 |
| 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]\*$: class MyClass { private int my\_field; } Compliant Solution class MyClass { private int myField; } | CODE\_SMELL | MINOR | 3 |
| Local variable and method parameter names should comply with a naming convention | Shared naming conventions allow teams to collaborate effectively. This rule raises an issue when a local variable or function parameter name does not match the provided regular expression. Noncompliant Code Example With the default regular expression ^[a-z][a-zA-Z0-9]\*$: public void doSomething(int my\_param) { int LOCAL; ... } Compliant Solution public void doSomething(int myParam) { int local; ... } Exceptions Loop counters are ignored by this rule. for (int i\_1 = 0; i\_1 &lt; limit; i\_1++) { // Compliant // ... } as well as one-character catch variables: try { //... } catch (Exception e) { // Compliant } | CODE\_SMELL | MINOR | 5 |
| 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 |
| Declarations should use Java collection interfaces such as "List" rather than specific implementation classes such as "LinkedList" |  | CODE\_SMELL | MINOR | 6 |
| 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 | 9 |
| 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 | 2 |
| Multiple variables should not be declared on the same line | Declaring multiple variables on one line is difficult to read. Noncompliant Code Example class MyClass { private int a, b; public void method(){ int c; int d; } } Compliant Solution class MyClass { private int a; private int b; public void method(){ int c; int d; } } See CERT, DCL52-J. - Do not declare more than one variable per declaration CERT, DCL04-C. - Do not declare more than one variable per declaration | CODE\_SMELL | MINOR | 5 |
| 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 | 18 |
| Formatting SQL queries is security-sensitive | Formatting strings used as SQL queries is security-sensitive. It has led in the past to the following vulnerabilities: CVE-2018-9019 CVE-2018-7318 CVE-2017-5611 SQL queries often need to use a hardcoded SQL string with a dynamic parameter coming from a user request. Formatting a string to add those parameters to the request is a bad practice as it can result in an SQL injection. The safe way to add parameters to a SQL query is to use SQL binding mechanisms. This rule raises an issue when an SQL query is built by formatting Strings, even if there is no injection. This rule does not detect SQL injections. The goal is to guide security code reviews and to prevent a common bad practice. The following method signatures from Java JDBC, JPA, JDO, Hibernate and Spring are tested: org.hibernate.Session.createQuery org.hibernate.Session.createSQLQuery java.sql.Statement.executeQuery java.sql.Statement.execute java.sql.Statement.executeUpdate java.sql.Statement.executeLargeUpdate java.sql.Statement.addBatch java.sql.Connection.prepareStatement java.sql.Connection.prepareCall java.sql.Connection.nativeSQL javax.persistence.EntityManager.createNativeQuery javax.persistence.EntityManager.createQuery org.springframework.jdbc.core.JdbcOperations.batchUpdate org.springframework.jdbc.core.JdbcOperations.execute org.springframework.jdbc.core.JdbcOperations.query org.springframework.jdbc.core.JdbcOperations.queryForList org.springframework.jdbc.core.JdbcOperations.queryForMap org.springframework.jdbc.core.JdbcOperations.queryForObject org.springframework.jdbc.core.JdbcOperations.queryForRowSet org.springframework.jdbc.core.JdbcOperations.queryForInt org.springframework.jdbc.core.JdbcOperations.queryForLong org.springframework.jdbc.core.JdbcOperations.update org.springframework.jdbc.core.PreparedStatementCreatorFactory.&lt;init&gt; org.springframework.jdbc.core.PreparedStatementCreatorFactory.newPreparedStatementCreator javax.jdo.PersistenceManager.newQuery javax.jdo.Query.setFilter javax.jdo.Query.setGrouping If a method is defined in an interface, implementations are also tested. For example this is the case for org.springframework.jdbc.core.JdbcOperations , which is usually used as org.springframework.jdbc.core.JdbcTemplate). Ask Yourself Whether the SQL query is built using string formatting technics, such as concatenating variables. some of the values are coming from an untrusted source and are not sanitized. You may be at risk if you answered yes to this question. Recommended Secure Coding Practices Avoid building queries manually using formatting technics. If you do it anyway, do not include user input in this building process. Use parameterized queries, prepared statements, or stored procedures whenever possible. You may also use ORM frameworks such as Hibernate which, if used correctly, reduce injection risks. Avoid executing SQL queries containing unsafe input in stored procedures or functions. Sanitize every unsafe input. You can also reduce the impact of an attack by using a database account with low privileges. Sensitive Code Example public User getUser(Connection con, String user) throws SQLException { Statement stmt1 = null; Statement stmt2 = null; PreparedStatement pstmt; try { stmt1 = con.createStatement(); ResultSet rs1 = stmt1.executeQuery("GETDATE()"); // No issue; hardcoded query stmt2 = con.createStatement(); ResultSet rs2 = stmt2.executeQuery("select FNAME, LNAME, SSN " + "from USERS where UNAME=" + user); // Sensitive pstmt = con.prepareStatement("select FNAME, LNAME, SSN " + "from USERS where UNAME=" + user); // Sensitive ResultSet rs3 = pstmt.executeQuery(); //... } public User getUserHibernate(org.hibernate.Session session, String data) { org.hibernate.Query query = session.createQuery( "FROM students where fname = " + data); // Sensitive // ... } Compliant Solution public User getUser(Connection con, String user) throws SQLException { Statement stmt1 = null; PreparedStatement pstmt = null; String query = "select FNAME, LNAME, SSN " + "from USERS where UNAME=?" try { stmt1 = con.createStatement(); ResultSet rs1 = stmt1.executeQuery("GETDATE()"); pstmt = con.prepareStatement(query); pstmt.setString(1, user); // Good; PreparedStatements escape their inputs. ResultSet rs2 = pstmt.executeQuery(); //... } } public User getUserHibernate(org.hibernate.Session session, String data) { org.hibernate.Query query = session.createQuery("FROM students where fname = ?"); query = query.setParameter(0,data); // Good; Parameter binding escapes all input org.hibernate.Query query2 = session.createQuery("FROM students where fname = " + data); // Sensitive // ... See OWASP Top 10 2017 Category A1 - Injection MITRE, CWE-89 - Improper Neutralization of Special Elements used in an SQL Command MITRE, CWE-564 - SQL Injection: Hibernate MITRE, CWE-20 - Improper Input Validation MITRE, CWE-943 - Improper Neutralization of Special Elements in Data Query Logic CERT, IDS00-J. - Prevent SQL injection SANS Top 25 - Insecure Interaction Between Components Derived from FindSecBugs rules Potential SQL/JPQL Injection (JPA), Potential SQL/JDOQL Injection (JDO), Potential SQL/HQL Injection (Hibernate) | SECURITY\_HOTSPOT | CRITICAL | 5 |
| Hashing data is security-sensitive | Hashing data is security-sensitive. It has led in the past to the following vulnerabilities: CVE-2018-9233 CVE-2013-5097 CVE-2007-1051 Cryptographic hash functions are used to uniquely identify information without storing their original form. When not done properly, an attacker can steal the original information by guessing it (ex: with a rainbow table), or replace the original data with another one having the same hash. This rule flags code that initiates hashing. Ask Yourself Whether the hashed value is used in a security context. the hashing algorithm you are using is known to have vulnerabilities. salts are not automatically generated and applied by the hashing function. any generated salts are cryptographically weak or not credential-specific. You are at risk if you answered yes to the first question and any of the following ones. Recommended Secure Coding Practices for security related purposes, use only hashing algorithms which are currently known to be strong. Avoid using algorithms like MD5 and SHA1 completely in security contexts. do not define your own hashing- or salt algorithms as they will most probably have flaws. do not use algorithms that compute too quickly, like SHA256, as it must remain beyond modern hardware capabilities to perform brute force and dictionary based attacks. use a hashing algorithm that generate its own salts as part of the hashing. If you generate your own salts, make sure that a cryptographically strong salt algorithm is used, that generated salts are credential-specific, and finally, that the salt is applied correctly before the hashing. save both the salt and the hashed value in the relevant database record; during future validation operations, the salt and hash can then be retrieved from the database. The hash is recalculated with the stored salt and the value being validated, and the result compared to the stored hash. the strength of hashing algorithms often decreases over time as hardware capabilities increase. Check regularly that the algorithms you are using are still considered secure. If needed, rehash your data using a stronger algorithm. Sensitive Code Example // === MessageDigest === import java.security.MessageDigest; import java.security.Provider; class A { void foo(String algorithm, String providerStr, Provider provider) throws Exception { MessageDigest.getInstance(algorithm); // Sensitive MessageDigest.getInstance(algorithm, providerStr); // Sensitive MessageDigest.getInstance(algorithm, provider); // Sensitive } } Regarding SecretKeyFactory. Any call to SecretKeyFactory.getInstance("...") with an argument starting by "PBKDF2" will be highlighted. See OWASP guidelines, list of standard algorithms and algorithms on android. // === javax.crypto === import javax.crypto.spec.PBEKeySpec; import javax.crypto.SecretKeyFactory; class A { void foo(char[] password, byte[] salt, int iterationCount, int keyLength) throws Exception { // Sensitive. Review this, even if it is the way recommended by OWASP SecretKeyFactory factory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA512"); PBEKeySpec spec = new PBEKeySpec(password, salt, iterationCount, keyLength); factory.generateSecret(spec).getEncoded(); } } Regarding Guava, only the hashing functions which are usually misused for sensitive data will raise an issue, i.e. md5 and sha\*. // === Guava === import com.google.common.hash.Hashing; class A { void foo() { Hashing.md5(); // Sensitive Hashing.sha1(); // Sensitive Hashing.sha256(); // Sensitive Hashing.sha384(); // Sensitive Hashing.sha512(); // Sensitive } } // === org.apache.commons === import org.apache.commons.codec.digest.DigestUtils; class A { void foo(String strName, byte[] data, String str, java.io.InputStream stream) throws Exception { new DigestUtils(strName); // Sensitive new DigestUtils(); // Sensitive DigestUtils.getMd2Digest(); // Sensitive DigestUtils.getMd5Digest(); // Sensitive DigestUtils.getShaDigest(); // Sensitive DigestUtils.getSha1Digest(); // Sensitive DigestUtils.getSha256Digest(); // Sensitive DigestUtils.getSha384Digest(); // Sensitive DigestUtils.getSha512Digest(); // Sensitive DigestUtils.md2(data); // Sensitive DigestUtils.md2(stream); // Sensitive DigestUtils.md2(str); // Sensitive DigestUtils.md2Hex(data); // Sensitive DigestUtils.md2Hex(stream); // Sensitive DigestUtils.md2Hex(str); // Sensitive DigestUtils.md5(data); // Sensitive DigestUtils.md5(stream); // Sensitive DigestUtils.md5(str); // Sensitive DigestUtils.md5Hex(data); // Sensitive DigestUtils.md5Hex(stream); // Sensitive DigestUtils.md5Hex(str); // Sensitive DigestUtils.sha(data); // Sensitive DigestUtils.sha(stream); // Sensitive DigestUtils.sha(str); // Sensitive DigestUtils.shaHex(data); // Sensitive DigestUtils.shaHex(stream); // Sensitive DigestUtils.shaHex(str); // Sensitive DigestUtils.sha1(data); // Sensitive DigestUtils.sha1(stream); // Sensitive DigestUtils.sha1(str); // Sensitive DigestUtils.sha1Hex(data); // Sensitive DigestUtils.sha1Hex(stream); // Sensitive DigestUtils.sha1Hex(str); // Sensitive DigestUtils.sha256(data); // Sensitive DigestUtils.sha256(stream); // Sensitive DigestUtils.sha256(str); // Sensitive DigestUtils.sha256Hex(data); // Sensitive DigestUtils.sha256Hex(stream); // Sensitive DigestUtils.sha256Hex(str); // Sensitive DigestUtils.sha384(data); // Sensitive DigestUtils.sha384(stream); // Sensitive DigestUtils.sha384(str); // Sensitive DigestUtils.sha384Hex(data); // Sensitive DigestUtils.sha384Hex(stream); // Sensitive DigestUtils.sha384Hex(str); // Sensitive DigestUtils.sha512(data); // Sensitive DigestUtils.sha512(stream); // Sensitive DigestUtils.sha512(str); // Sensitive DigestUtils.sha512Hex(data); // Sensitive DigestUtils.sha512Hex(stream); // Sensitive DigestUtils.sha512Hex(str); // Sensitive } } See OWASP Top 10 2017 Category A3 - Sensitive Data Exposure OWASP Top 10 2017 Category A6 - Security Misconfiguration MITRE, CWE-916 - Use of Password Hash With Insufficient Computational Effort MITRE, CWE-759 - Use of a One-Way Hash without a Salt MITRE, CWE-760 - Use of a One-Way Hash with a Predictable Salt SANS Top 25 - Porous Defenses | SECURITY\_HOTSPOT | CRITICAL | 1 |
| Throwable.printStackTrace(...) should not be called | Throwable.printStackTrace(...) prints a Throwable and its stack trace to some stream. By default that stream System.Err, which could inadvertently expose sensitive information. Loggers should be used instead to print Throwables, as they have many advantages: Users are able to easily retrieve the logs. The format of log messages is uniform and allow users to browse the logs easily. This rule raises an issue when printStackTrace is used without arguments, i.e. when the stack trace is printed to the default stream. Noncompliant Code Example try { /\* ... \*/ } catch(Exception e) { e.printStackTrace(); // Noncompliant } Compliant Solution try { /\* ... \*/ } catch(Exception e) { LOGGER.log("context", e); } See OWASP Top 10 2017 Category A3 - Sensitive Data Exposure MITRE, CWE-489 - Leftover Debug Code | VULNERABILITY | MINOR | 3 |
| Exceptions should not be thrown from servlet methods | Even though the signatures for methods in a servlet include throws IOException, ServletException, it's a bad idea to let such exceptions be thrown. Failure to catch exceptions in a servlet could leave a system in a vulnerable state, possibly resulting in denial-of-service attacks, or the exposure of sensitive information because when a servlet throws an exception, the servlet container typically sends debugging information back to the user. And that information could be very valuable to an attacker. This rule checks all exceptions in methods named "do\*" are explicitly handled in servlet classes. Noncompliant Code Example public void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException, ServletException { String ip = request.getRemoteAddr(); InetAddress addr = InetAddress.getByName(ip); // Noncompliant; getByName(String) throws UnknownHostException //... } Compliant Solution public void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException, ServletException { try { String ip = request.getRemoteAddr(); InetAddress addr = InetAddress.getByName(ip); //... } catch (UnknownHostException uhex) { //... } } See OWASP Top 10 2017 Category A3 - Sensitive Data Exposure MITRE, CWE-600 - Uncaught Exception in Servlet CERT, ERR01-J. - Do not allow exceptions to expose sensitive information | VULNERABILITY | MINOR | 4 |