|  |
| --- |
| omni-svu-connector-all  Version 1.0  Code analysis |

|  |
| --- |
| By: default  2019-08-17 |

# 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 omni-svu-connector-all.

# Configuration

* Quality Profiles
  + Names: Sonar way [Java]; Sonar way [XML];
  + Files: AWxlmWoc5IoKq3LiBrWI.json; AWxlmWzR5IoKq3LiBrcx.json;
* Quality Gate
  + Name: Sonar way
  + File: Sonar way.xml

# Synthesis

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Quality Gate | Reliability | Security | Maintainability | Coverage | Duplication |
| ERROR | E | E | A | 0.0 % | 19.9 % |

# Metrics

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Cyclomatic  Complexity | Cognitive  Complexity | Lines of code per file | Comment  density (%) | Coverage | Duplication (%) |
| Min | 0.0 | 0.0 | 2.0 | 0.0 | 0.0 | 0.0 |
| Max | 13028.0 | 9142.0 | 330034.0 | 69.3 | 0.0 | 88.9 |

# Volume

|  |  |
| --- | --- |
| Language | Number |
| Java | 128256 |
| XML | 298098 |
| Total | 426354 |

# Issues count by severity and type

|  |  |  |
| --- | --- | --- |
| Type | Severity | Number |
| VULNERABILITY | BLOCKER | 1 |
| VULNERABILITY | CRITICAL | 0 |
| VULNERABILITY | MAJOR | 0 |
| VULNERABILITY | MINOR | 253 |
| VULNERABILITY | INFO | 0 |
| BUG | BLOCKER | 4 |
| BUG | CRITICAL | 0 |
| BUG | MAJOR | 47 |
| BUG | MINOR | 2 |
| BUG | INFO | 0 |
| CODE\_SMELL | BLOCKER | 27 |
| CODE\_SMELL | CRITICAL | 166 |
| CODE\_SMELL | MAJOR | 1411 |
| CODE\_SMELL | MINOR | 3172 |
| CODE\_SMELL | INFO | 42 |

# 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 | 4 |
| 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 (int i = 0; i &lt; 10; i++) { // noncompliant, loop only executes once printf("i is %d", i); break; } ... for (int i = 0; i &lt; 10; i++) { // noncompliant, loop only executes once if(i == x) { break; } else { printf("i is %d", i); return; } } Compliant Solution for (int i = 0; i &lt; 10; i++) { printf("i is %d", i); } ... for (int i = 0; i &lt; 10; i++) { if(i == x) { break; } else { printf("i is %d", i); } } | BUG | MAJOR | 3 |
| Null pointers should not be dereferenced | A reference to null should never be dereferenced/accessed. Doing so will cause a NullPointerException to be thrown. At best, such an exception will cause abrupt program termination. At worst, it could expose debugging information that would be useful to an attacker, or it could allow an attacker to bypass security measures. Note that when they are present, this rule takes advantage of @CheckForNull and @Nonnull annotations defined in JSR-305 to understand which values are and are not nullable except when @Nonnull is used on the parameter to equals, which by contract should always work with null. Noncompliant Code Example @CheckForNull String getName(){...} public boolean isNameEmpty() { return getName().length() == 0; // Noncompliant; the result of getName() could be null, but isn't null-checked } Connection conn = null; Statement stmt = null; try{ conn = DriverManager.getConnection(DB\_URL,USER,PASS); stmt = conn.createStatement(); // ... }catch(Exception e){ e.printStackTrace(); }finally{ stmt.close(); // Noncompliant; stmt could be null if an exception was thrown in the try{} block conn.close(); // Noncompliant; conn could be null if an exception was thrown } private void merge(@Nonnull Color firstColor, @Nonnull Color secondColor){...} public void append(@CheckForNull Color color) { merge(currentColor, color); // Noncompliant; color should be null-checked because merge(...) doesn't accept nullable parameters } void paint(Color color) { if(color == null) { System.out.println("Unable to apply color " + color.toString()); // Noncompliant; NullPointerException will be thrown return; } ... } 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 an object is required | BUG | MAJOR | 20 |
| 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(); } int b = a &gt; 12 ? 4 : 4; // 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 | 8 |
| Strings and Boxed types should be compared using "equals()" | It's almost always a mistake to compare two instances of java.lang.String or boxed types like java.lang.Integer using reference equality == or !=, because it is not comparing actual value but locations in memory. Noncompliant Code Example String firstName = getFirstName(); // String overrides equals String lastName = getLastName(); if (firstName == lastName) { ... }; // Non-compliant; false even if the strings have the same value Compliant Solution String firstName = getFirstName(); String lastName = getLastName(); if (firstName != null &amp;&amp; firstName.equals(lastName)) { ... }; See MITRE, CWE-595 - Comparison of Object References Instead of Object Contents MITRE, CWE-597 - Use of Wrong Operator in String Comparison CERT, EXP03-J. - Do not use the equality operators when comparing values of boxed primitives CERT, EXP50-J. - Do not confuse abstract object equality with reference equality | BUG | MAJOR | 16 |
| 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 } } See MISRA C:2012, 17.8 - A function parameter should not be modified | BUG | MINOR | 2 |
| Child class fields should not shadow parent class fields | Having a variable with the same name in two unrelated classes is fine, but do the same thing within a class hierarchy and you'll get confusion at best, chaos at worst. Noncompliant Code Example public class Fruit { protected Season ripe; protected Color flesh; // ... } public class Raspberry extends Fruit { private boolean ripe; // Noncompliant private static Color FLESH; // Noncompliant } Compliant Solution public class Fruit { protected Season ripe; protected Color flesh; // ... } public class Raspberry extends Fruit { private boolean ripened; private static Color FLESH\_COLOR; } Exceptions This rule ignores same-name fields that are static in both the parent and child classes. This rule ignores private parent class fields, but in all other such cases, the child class field should be renamed. public class Fruit { private Season ripe; // ... } public class Raspberry extends Fruit { private Season ripe; // Compliant as parent field 'ripe' is anyway not visible from Raspberry // ... } | CODE\_SMELL | BLOCKER | 25 |
| Methods returns should not be invariant | When a method 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 surely a bug. This rule raises an issue when a method contains several return statements that all return the same value. Noncompliant Code Example int foo(int a) { int b = 12; if (a == 1) { return b; } return b; // Noncompliant } | CODE\_SMELL | BLOCKER | 2 |
| 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 | 8 |
| 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 | 15 |
| 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 | 141 |
| A conditionally executed single line should be denoted by indentation | In the absence of enclosing curly braces, the line immediately after a conditional is the one that is conditionally executed. By both convention and good practice, such lines are indented. In the absence of both curly braces and indentation the intent of the original programmer is entirely unclear and perhaps not actually what is executed. Additionally, such code is highly likely to be confusing to maintainers. Noncompliant Code Example if (condition) // Noncompliant doTheThing(); doTheOtherThing(); somethingElseEntirely(); foo(); Compliant Solution if (condition) doTheThing(); doTheOtherThing(); somethingElseEntirely(); foo(); | CODE\_SMELL | CRITICAL | 2 |
| Track uses of "TODO" tags | TODO tags are commonly used to mark places where some more code is required, but which the developer wants to implement later. Sometimes the developer will not have the time or will simply forget to get back to that tag. This rule is meant to track those tags and to ensure that they do not go unnoticed. Noncompliant Code Example void doSomething() { // TODO } See MITRE, CWE-546 - Suspicious Comment | CODE\_SMELL | INFO | 42 |
| 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 | 768 |
| 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. See MISRA C:2004, 2.4 - Sections of code should not be "commented out". MISRA C++:2008, 2-7-2 - Sections of code shall not be "commented out" using C-style comments. MISRA C++:2008, 2-7-3 - Sections of code should not be "commented out" using C++ comments. MISRA C:2012, Dir. 4.4 - Sections of code should not be "commented out" | CODE\_SMELL | MAJOR | 320 |
| 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 | 3 |
| Methods 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: public void doSomething(int param1, int param2, int param3, String param4, long param5) { ... } Compliant Solution public void doSomething(int param1, int param2, int param3, String param4) { ... } Exceptions Methods annotated with Spring's @RequestMapping (and related shortcut annotations, like @GetRequest) or @JsonCreator may have a lot of parameters, encapsulation being possible. Such methods are therefore ignored. | CODE\_SMELL | MAJOR | 6 |
| 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 | 2 |
| 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 | 25 |
| Standard outputs should not be used directly to log anything | When logging a message there are several important requirements which must be fulfilled: The user must be able to easily retrieve the logs The format of all logged message must be uniform to allow the user to easily read the log Logged data must actually be recorded Sensitive data must only be logged securely If a program directly writes to the standard outputs, there is absolutely no way to comply with those requirements. That's why defining and using a dedicated logger is highly recommended. Noncompliant Code Example System.out.println("My Message"); // Noncompliant Compliant Solution logger.log("My Message"); See CERT, ERR02-J. - Prevent exceptions while logging data | CODE\_SMELL | MAJOR | 16 |
| Collapsible "if" statements should be merged | Merging collapsible if statements increases the code's readability. Noncompliant Code Example if (file != null) { if (file.isFile() || file.isDirectory()) { /\* ... \*/ } } Compliant Solution if (file != null &amp;&amp; isFileOrDirectory(file)) { /\* ... \*/ } private static boolean isFileOrDirectory(File file) { return file.isFile() || file.isDirectory(); } | CODE\_SMELL | MAJOR | 22 |
| 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 | 17 |
| 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 | 5 |
| 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 | 27 |
| Synchronized classes Vector, Hashtable, Stack and StringBuffer should not be used | Early classes of the Java API, such as Vector, Hashtable and StringBuffer, were synchronized to make them thread-safe. Unfortunately, synchronization has a big negative impact on performance, even when using these collections from a single thread. It is better to use their new unsynchronized replacements: ArrayList or LinkedList instead of Vector Deque instead of Stack HashMap instead of Hashtable StringBuilder instead of StringBuffer Noncompliant Code Example Vector cats = new Vector(); Compliant Solution ArrayList cats = new ArrayList(); Exceptions Use of those synchronized classes is ignored in the signatures of overriding methods. @Override public Vector getCats() {...} | CODE\_SMELL | MAJOR | 2 |
| "@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 |
| 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 | 5 |
| 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 MISRA C++:2008, 0-1-11 - There shall be no unused parameters (named or unnamed) in nonvirtual functions. MISRA C:2012, 2.7 - There should be no unused parameters in functions CERT, MSC12-C. - Detect and remove code that has no effect or is never executed | CODE\_SMELL | MAJOR | 39 |
| 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 | 1 |
| 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 | 63 |
| Two branches in a conditional structure should not have exactly the same implementation |  | CODE\_SMELL | MAJOR | 2 |
| 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 a = true; if (a) { // Noncompliant doSomething(); } if (b &amp;&amp; a) { // Noncompliant; "a" is always "true" doSomething(); } if (c || !a) { // Noncompliant; "!a" is always "false" doSomething(); } Compliant Solution a = true; if (foo(a)) { doSomething(); } if (b) { doSomething(); } if (c) { doSomething(); } See MISRA C:2004, 13.7 - Boolean operations whose results are invariant shall not be permitted. MISRA C:2012, 14.3 - Controlling expressions shall not be invariant MITRE, CWE-571 - Expression is Always True MITRE, CWE-570 - Expression is Always False MITRE, CWE-489 - Leftover Debug Code CERT, MSC12-C. - Detect and remove code that has no effect or is never executed | CODE\_SMELL | MAJOR | 5 |
| 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. Noncompliant Code Example if (condition) firstActionInBlock(); secondAction(); // Noncompliant; executed unconditionally thirdAction(); if (condition) firstActionInBlock(); secondAction(); // Noncompliant; secondAction executed unconditionally if (condition) firstActionInBlock(); // Noncompliant secondAction(); // Executed unconditionally if (condition); secondAction(); // Noncompliant; secondAction executed unconditionally String str = null; for (int 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(); String str = null; for (int i = 0; i &lt; array.length; i++) { str = array[i]; doTheThing(str); } See MITRE, CWE-483 - Incorrect Block Delimitation CERT, EXP52-J. - Use braces for the body of an if, for, or while statement | CODE\_SMELL | MAJOR | 1 |
| 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 |
| String function use should be optimized for single characters | An indexOf or lastIndexOf call with a single letter String can be made more performant by switching to a call with a char argument. Noncompliant Code Example String myStr = "Hello World"; // ... int pos = myStr.indexOf("W"); // Noncompliant // ... int otherPos = myStr.lastIndexOf("r"); // Noncompliant // ... Compliant Solution String myStr = "Hello World"; // ... int pos = myStr.indexOf('W'); // ... int otherPos = myStr.lastIndexOf('r'); // ... | CODE\_SMELL | MAJOR | 3 |
| Methods should not have identical implementations | When two methods 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, one implementation should invoke the other. Numerical and string literals are not taken into account. Noncompliant Code Example private final static String CODE = "bounteous"; public String calculateCode() { doTheThing(); return CODE; } public String getName() { // Noncompliant doTheThing(); return CODE; } Compliant Solution private final static String CODE = "bounteous"; public String getCode() { doTheThing(); return CODE; } public String getName() { return getCode(); } Exceptions Methods that are not accessors (getters and setters), with fewer than 2 statements are ignored. | CODE\_SMELL | MAJOR | 1 |
| Unused "private" methods should be removed | private methods that are never executed are dead code: unnecessary, inoperative code that should be removed. Cleaning out dead code decreases the size of the maintained codebase, making it easier to understand the program and preventing bugs from being introduced. Note that this rule does not take reflection into account, which means that issues will be raised on private methods that are only accessed using the reflection API. Noncompliant Code Example public class Foo implements Serializable { private Foo(){} //Compliant, private empty constructor intentionally used to prevent any direct instantiation of a class. public static void doSomething(){ Foo foo = new Foo(); ... } private void unusedPrivateMethod(){...} private void writeObject(ObjectOutputStream s){...} //Compliant, relates to the java serialization mechanism private void readObject(ObjectInputStream in){...} //Compliant, relates to the java serialization mechanism } Compliant Solution public class Foo implements Serializable { private Foo(){} //Compliant, private empty constructor intentionally used to prevent any direct instantiation of a class. public static void doSomething(){ Foo foo = new Foo(); ... } private void writeObject(ObjectOutputStream s){...} //Compliant, relates to the java serialization mechanism private void readObject(ObjectInputStream in){...} //Compliant, relates to the java serialization mechanism } Exceptions This rule doesn't raise any issue on annotated methods. | CODE\_SMELL | MAJOR | 74 |
| 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. See MISRA C:2004, 2.4 - Sections of code should not be "commented out". MISRA C++:2008, 2-7-2 - Sections of code shall not be "commented out" using C-style comments. MISRA C++:2008, 2-7-3 - Sections of code should not be "commented out" using C++ comments. MISRA C:2012, Dir. 4.4 - Sections of code should not be "commented out" | CODE\_SMELL | MAJOR | 1 |
| "@Deprecated" code 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 /\*\* \* @deprecated As of release 1.3, replaced by {@link #Fee} \*/ @Deprecated public class Fum { ... } public class Foo { /\*\* \* @deprecated As of release 1.7, replaced by {@link #doTheThingBetter()} \*/ @Deprecated public void doTheThing() { ... } public void doTheThingBetter() { ... } } public class Bar extends Foo { public void doTheThing() { ... } // Noncompliant; don't override a deprecated method or explicitly mark it as @Deprecated } public class Bar extends Fum { // Noncompliant; Fum is deprecated public void myMethod() { Foo foo = new Foo(); // okay; the class isn't deprecated foo.doTheThing(); // Noncompliant; doTheThing method is deprecated } } See MITRE, CWE-477 - Use of Obsolete Functions CERT, MET02-J. - Do not use deprecated or obsolete classes or methods | CODE\_SMELL | MINOR | 10 |
| 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 MISRA C:2004, 14.3 - Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment provided that the first character following the null statement is a white-space character. MISRA C++:2008, 6-2-3 - Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character. 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 | 3 |
| Modifiers should be declared in the correct order | The Java Language Specification recommends listing modifiers in the following order: 1. Annotations 2. public 3. protected 4. private 5. abstract 6. static 7. final 8. transient 9. volatile 10. synchronized 11. native 12. strictfp Not following this convention has no technical impact, but will reduce the code's readability because most developers are used to the standard order. Noncompliant Code Example static public void main(String[] args) { // Noncompliant } Compliant Solution public static void main(String[] args) { // Compliant } | CODE\_SMELL | MINOR | 561 |
| "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 | 7 |
| 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]\*$: public int DoSomething(){...} Compliant Solution public int doSomething(){...} Exceptions Overriding methods are excluded. @Override public int Do\_Something(){...} | CODE\_SMELL | MINOR | 139 |
| Field names should comply with a naming convention | Sharing some naming conventions is a key point to make it possible for a team to efficiently collaborate. This rule allows to check that field names match a provided regular expression. Noncompliant Code Example With the default regular expression ^[a-z][a-zA-Z0-9]\*$: class MyClass { private int my\_field; } Compliant Solution class MyClass { private int myField; } | CODE\_SMELL | MINOR | 102 |
| 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 | 85 |
| 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. Similarly, method invocations wrapped into if-then-else differing only from boolean literals should be simplified into a single invocation. Noncompliant Code Example boolean foo(Object param) { if (expression) { // Noncompliant bar(param, true, "qix"); } else { bar(param, false, "qix"); } if (expression) { // Noncompliant return true; } else { return false; } } Compliant Solution boolean foo(Object param) { bar(param, expression, "qix"); return expression; } | CODE\_SMELL | MINOR | 5 |
| String.valueOf() should not be appended to a String | Appending String.valueOf() to a String decreases the code readability. The argument passed to String.valueOf() should be directly appended instead. Noncompliant Code Example public void display(int i){ System.out.println("Output is " + String.valueOf(i)); // Noncompliant } Compliant Solution public void display(int i){ System.out.println("Output is " + i); // Compliant } | CODE\_SMELL | MINOR | 15 |
| 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 | 15 |
| Exception classes should be immutable | Exceptions are meant to represent the application's state at the point at which an error occurred. Making all fields in an Exception class final ensures that this state: Will be fully defined at the same time the Exception is instantiated. Won't be updated or corrupted by a questionable error handler. This will enable developers to quickly understand what went wrong. Noncompliant Code Example public class MyException extends Exception { private int status; // Noncompliant public MyException(String message) { super(message); } public int getStatus() { return status; } public void setStatus(int status) { this.status = status; } } Compliant Solution public class MyException extends Exception { private final int status; public MyException(String message, int status) { super(message); this.status = status; } public int getStatus() { return status; } } | CODE\_SMELL | MINOR | 5 |
| Public constants and fields initialized at declaration should be "static final" rather than merely "final" | Making a public constant just final as opposed to static final leads to duplicating its value for every instance of the class, uselessly increasing the amount of memory required to execute the application. Further, when a non-public, final field isn't also static, it implies that different instances can have different values. However, initializing a non-static final field in its declaration forces every instance to have the same value. So such fields should either be made static or initialized in the constructor. Noncompliant Code Example public class Myclass { public final int THRESHOLD = 3; } Compliant Solution public class Myclass { public static final int THRESHOLD = 3; // Compliant } Exceptions No issues are reported on final fields of inner classes whose type is not a primitive or a String. Indeed according to the Java specification: An inner class is a nested class that is not explicitly or implicitly declared static. Inner classes may not declare static initializers (§8.7) or member interfaces. Inner classes may not declare static members, unless they are compile-time constant fields (§15.28). | CODE\_SMELL | MINOR | 1 |
| Array designators "[]" should be on the type, not the variable | Array designators should always be located on the type for better code readability. Otherwise, developers must look both at the type and the variable name to know whether or not a variable is an array. Noncompliant Code Example int matrix[][]; // Noncompliant int[] matrix[]; // Noncompliant Compliant Solution int[][] matrix; // Compliant | CODE\_SMELL | MINOR | 2 |
| The default unnamed package should not be used | According to the Java Language Specification: Unnamed packages are provided by the Java platform principally for convenience when developing small or temporary applications or when just beginning development. To enforce this best practice, classes located in default package can no longer be accessed from named ones since Java 1.4. Noncompliant Code Example public class MyClass { /\* ... \*/ } Compliant Solution package org.example; public class MyClass{ /\* ... \*/ } | CODE\_SMELL | MINOR | 4 |
| "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(); } See MISRA C:2012, 16.6 - Every switch statement shall have at least two switch-clauses | CODE\_SMELL | MINOR | 1 |
| Declarations should use Java collection interfaces such as "List" rather than specific implementation classes such as "LinkedList" |  | CODE\_SMELL | MINOR | 8 |
| Loops should not contain more than a single "break" or "continue" statement | Restricting the number of break and continue statements in a loop is done in the interest of good structured programming. One break and continue statement is acceptable in a loop, since it facilitates optimal coding. If there is more than one, the code should be refactored to increase readability. Noncompliant Code Example for (int i = 1; i &lt;= 10; i++) { // Noncompliant - 2 continue - one might be tempted to add some logic in between if (i % 2 == 0) { continue; } if (i % 3 == 0) { continue; } System.out.println("i = " + i); } | CODE\_SMELL | MINOR | 19 |
| Private fields only used as local variables in methods should become local variables | When the value of a private field is always assigned to in a class' methods before being read, then it is not being used to store class information. Therefore, it should become a local variable in the relevant methods to prevent any misunderstanding. Noncompliant Code Example public class Foo { private int a; private int b; public void doSomething(int y) { a = y + 5; ... if(a == 0) { ... } ... } public void doSomethingElse(int y) { b = y + 3; ... } } Compliant Solution public class Foo { public void doSomething(int y) { int a = y + 5; ... if(a == 0) { ... } } public void doSomethingElse(int y) { int b = y + 3; ... } } Exceptions This rule doesn't raise any issue on annotated field. | CODE\_SMELL | MINOR | 19 |
| 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 | 61 |
| 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 | 32 |
| Strings should not be concatenated using '+' in a loop | Strings are immutable objects, so concatenation doesn't simply add the new String to the end of the existing string. Instead, in each loop iteration, the first String is converted to an intermediate object type, the second string is appended, and then the intermediate object is converted back to a String. Further, performance of these intermediate operations degrades as the String gets longer. Therefore, the use of StringBuilder is preferred. Noncompliant Code Example String str = ""; for (int i = 0; i &lt; arrayOfStrings.length ; ++i) { str = str + arrayOfStrings[i]; } Compliant Solution StringBuilder bld = new StringBuilder(); for (int i = 0; i &lt; arrayOfStrings.length; ++i) { bld.append(arrayOfStrings[i]); } String str = bld.toString(); | 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 MISRA C++:2008, 8-0-1 - An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively 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 | 2 |
| "toString()" should never be called on a String object | Invoking a method designed to return a string representation of an object which is already a string is a waste of keystrokes. This redundant construction may be optimized by the compiler, but will be confusing in the meantime. Noncompliant Code Example String message = "hello world"; System.out.println(message.toString()); // Noncompliant; Compliant Solution String message = "hello world"; System.out.println(message); | CODE\_SMELL | MINOR | 9 |
| Classes should not be empty | There is no good excuse for an empty class. If it's being used simply as a common extension point, it should be replaced with an interface. If it was stubbed in as a placeholder for future development it should be fleshed-out. In any other case, it should be eliminated. Noncompliant Code Example public class Nothing { // Noncompliant } Compliant Solution public interface Nothing { } Exceptions Empty classes can be used as marker types (for Spring for instance), therefore empty classes that are annotated will be ignored. @Configuration @EnableWebMvc public final class ApplicationConfiguration { } | CODE\_SMELL | MINOR | 1 |
| Parsing should be used to convert "Strings" to primitives | Rather than creating a boxed primitive from a String to extract the primitive value, use the relevant parse method instead. It will be clearer and more efficient. Noncompliant Code Example String myNum = "12.2"; float f = (new Float(myNum)).floatValue(); // Noncompliant; creates &amp; discards a Float Compliant Solution String myNum = "12.2"; float f = Float.parseFloat(myNum); | CODE\_SMELL | MINOR | 1 |
| Catches should be combined | Since Java 7 it has been possible to catch multiple exceptions at once. Therefore, when multiple catch blocks have the same code, they should be combined for better readability. Note that this rule is automatically disabled when the project's sonar.java.source is lower than 7. Noncompliant Code Example catch (IOException e) { doCleanup(); logger.log(e); } catch (SQLException e) { // Noncompliant doCleanup(); logger.log(e); } catch (TimeoutException e) { // Compliant; block contents are different doCleanup(); throw e; } Compliant Solution catch (IOException|SQLException e) { doCleanup(); logger.log(e); } catch (TimeoutException e) { doCleanup(); throw e; } | CODE\_SMELL | MINOR | 2 |
| The diamond operator ("<>") should be used |  | CODE\_SMELL | MINOR | 1869 |
| Static non-final field names should comply with a naming convention | Shared naming conventions allow teams to collaborate efficiently. This rule checks that static non-final field names match a provided regular expression. Noncompliant Code Example With the default regular expression ^[a-z][a-zA-Z0-9]\*$: public final class MyClass { private static String foo\_bar; } Compliant Solution class MyClass { private static String fooBar; } | CODE\_SMELL | MINOR | 129 |
| Methods should not return constants | There's no point in forcing the overhead of a method call for a method that always returns the same constant value. Even worse, the fact that a method call must be made will likely mislead developers who call the method thinking that something more is done. Declare a constant instead. This rule raises an issue if on methods that contain only one statement: the return of a constant value. Noncompliant Code Example int getBestNumber() { return 12; // Noncompliant } Compliant Solution static int bestNumber = 12; Exceptions Methods with annotations, such as @Override and Spring's @RequestMapping, are ignored. | CODE\_SMELL | MINOR | 2 |
| Jump statements should not be redundant | Jump statements such as return 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 public void foo() { while (condition1) { if (condition2) { continue; // Noncompliant } else { doTheThing(); } } return; // Noncompliant; this is a void method } Compliant Solution public void foo() { while (condition1) { if (!condition2) { doTheThing(); } } } | CODE\_SMELL | MINOR | 5 |
| "StandardCharsets" constants should be preferred | JDK7 introduced the class java.nio.charset.StandardCharsets. It provides constants for all charsets that are guaranteed to be available on every implementation of the Java platform. ISO\_8859\_1 US\_ASCII UTF\_16 UTF\_16BE UTF\_16LE UTF\_8 These constants should be preferred to: - the use of a String such as "UTF-8" which has the drawback of requiring the catch/throw of an UnsupportedEncodingException that will never actually happen - the use of Guava’s Charsets class, which has been obsolete since JDK7 Noncompliant Code Example try { byte[] bytes = string.getBytes("UTF-8"); // Noncompliant; use a String instead of StandardCharsets.UTF\_8 } catch (UnsupportedEncodingException e) { throw new AssertionError(e); } // ... byte[] bytes = string.getBytes(Charsets.UTF\_8); // Noncompliant; Guava way obsolete since JDK7 Compliant Solution byte[] bytes = string.getBytes(StandardCharsets.UTF\_8) | CODE\_SMELL | MINOR | 15 |
| 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 | 41 |
| 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. Questionable 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); // Questionable MessageDigest.getInstance(algorithm, providerStr); // Questionable MessageDigest.getInstance(algorithm, provider); // Questionable } } 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 { // Questionable. 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(); // Questionable Hashing.sha1(); // Questionable Hashing.sha256(); // Questionable Hashing.sha384(); // Questionable Hashing.sha512(); // Questionable } } // === 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); // Questionable new DigestUtils(); // Questionable DigestUtils.getMd2Digest(); // Questionable DigestUtils.getMd5Digest(); // Questionable DigestUtils.getShaDigest(); // Questionable DigestUtils.getSha1Digest(); // Questionable DigestUtils.getSha256Digest(); // Questionable DigestUtils.getSha384Digest(); // Questionable DigestUtils.getSha512Digest(); // Questionable DigestUtils.md2(data); // Questionable DigestUtils.md2(stream); // Questionable DigestUtils.md2(str); // Questionable DigestUtils.md2Hex(data); // Questionable DigestUtils.md2Hex(stream); // Questionable DigestUtils.md2Hex(str); // Questionable DigestUtils.md5(data); // Questionable DigestUtils.md5(stream); // Questionable DigestUtils.md5(str); // Questionable DigestUtils.md5Hex(data); // Questionable DigestUtils.md5Hex(stream); // Questionable DigestUtils.md5Hex(str); // Questionable DigestUtils.sha(data); // Questionable DigestUtils.sha(stream); // Questionable DigestUtils.sha(str); // Questionable DigestUtils.shaHex(data); // Questionable DigestUtils.shaHex(stream); // Questionable DigestUtils.shaHex(str); // Questionable DigestUtils.sha1(data); // Questionable DigestUtils.sha1(stream); // Questionable DigestUtils.sha1(str); // Questionable DigestUtils.sha1Hex(data); // Questionable DigestUtils.sha1Hex(stream); // Questionable DigestUtils.sha1Hex(str); // Questionable DigestUtils.sha256(data); // Questionable DigestUtils.sha256(stream); // Questionable DigestUtils.sha256(str); // Questionable DigestUtils.sha256Hex(data); // Questionable DigestUtils.sha256Hex(stream); // Questionable DigestUtils.sha256Hex(str); // Questionable DigestUtils.sha384(data); // Questionable DigestUtils.sha384(stream); // Questionable DigestUtils.sha384(str); // Questionable DigestUtils.sha384Hex(data); // Questionable DigestUtils.sha384Hex(stream); // Questionable DigestUtils.sha384Hex(str); // Questionable DigestUtils.sha512(data); // Questionable DigestUtils.sha512(stream); // Questionable DigestUtils.sha512(str); // Questionable DigestUtils.sha512Hex(data); // Questionable DigestUtils.sha512Hex(stream); // Questionable DigestUtils.sha512Hex(str); // Questionable } } | SECURITY\_HOTSPOT | CRITICAL | 3 |
| Using command line arguments is security-sensitive |  | SECURITY\_HOTSPOT | CRITICAL | 3 |
| Expanding archive files is security-sensitive | Expanding archive files is security-sensitive. For example, expanding archive files has led in the past to the following vulnerabilities: CVE-2018-1263 CVE-2018-16131 Applications that expand archive files (zip, tar, jar, war, 7z, ...) should verify the path where the archive's files are expanded and not trust blindly the content of the archive. Archive's files should not be expanded outside of the root directory where the archive is supposed to be expanded. Also, applications should control the size of the expanded data to not be a victim of Zip Bomb attack. Failure to do so could allow an attacker to use a specially crafted archive that holds directory traversal paths (e.g. ../../attacker.sh) or the attacker could overload the file system, processors or memory of the operating system where the archive is expanded making the target OS completely unusable. This rule raises an issue when code handle archives. The goal is to guide security code reviews. Ask Yourself Whether there is no validation of the name of the archive entry there is no validation of the effective path where the archive entry is going to be expanded there is no validation of the size of the expanded archive entry there is no validation of the ratio between the compressed and uncompressed archive entry You are at risk if you answered yes to any of those questions. Recommended Secure Coding Practices Validate the full path of the extracted file against the full path of the directory where files are uncompressed. the canonical path of the uncompressed file must start with the canonical path of the directory where files are extracted. the name of the archive entry must not contain "..", i.e. reference to a parent directory. String canonicalDirPath = outputDir.getCanonicalPath(); String canonicalDestPath = targetFile.getCanonicalPath(); if (!canonicalDestPath.startsWith(canonicalDirPath + File.separator)) { // Sanitizer throw new ArchiverException("Entry is trying to leave the target dir: " + zipEntry.getName()); } Stop extracting the archive if any of its entries has been tainted with a directory traversal path. Define and control the ratio between compressed and uncompress bytes. Define and control the maximum allowed uncompressed file size. Count the number of file entries extracted from the archive and abort the extraction if their number is greater than a predefined threshold. Sensitive Code Example java.util.zip.ZipFile zipFile = new ZipFile(zipFileName); Enumeration&lt;? extends ZipEntry&gt; entries = zipFile.entries(); while (entries.hasMoreElements()) { ZipEntry e = entries.nextElement(); // Questionable File f = new File(outputDir, e.getName()); InputStream input = zipFile.getInputStream(e); extractFile(new ZipInputStream(input), outputDir, e.getName()); } Exceptions This rule doesn't raise an issue when a ZipEntry or a ArchiveEntry: is declared as a class field is a parameter of an abstract method of an interface or abstract class See OWASP Top 10 2017 Category A1 - Injection MITRE, CWE-409 - Improper Handling of Highly Compressed Data (Data Amplification) CERT, IDS04-J. - Safely extract files from ZipInputStream Snyk Research Team: Zip Slip Vulnerability | SECURITY\_HOTSPOT | CRITICAL | 2 |
| Credentials should not be hard-coded | Because it is easy to extract strings from a compiled application, credentials should never be hard-coded. Do so, and they're almost guaranteed to end up in the hands of an attacker. This is particularly true for applications that are distributed. Credentials should be stored outside of the code in a strongly-protected encrypted configuration file or database. It's recommended to customize the configuration of this rule with additional credential words such as "oauthToken", "secret", ... Noncompliant Code Example Connection conn = null; try { conn = DriverManager.getConnection("jdbc:mysql://localhost/test?" + "user=steve&amp;password=blue"); // Noncompliant String uname = "steve"; String password = "blue"; conn = DriverManager.getConnection("jdbc:mysql://localhost/test?" + "user=" + uname + "&amp;password=" + password); // Noncompliant java.net.PasswordAuthentication pa = new java.net.PasswordAuthentication("userName", "1234".toCharArray()); // Noncompliant Compliant Solution Connection conn = null; try { String uname = getEncryptedUser(); String password = getEncryptedPass(); conn = DriverManager.getConnection("jdbc:mysql://localhost/test?" + "user=" + uname + "&amp;password=" + password); See OWASP Top 10 2017 Category A2 - Broken Authentication MITRE, CWE-798 - Use of Hard-coded Credentials MITRE, CWE-259 - Use of Hard-coded Password CERT, MSC03-J. - Never hard code sensitive information SANS Top 25 - Porous Defenses Derived from FindSecBugs rule Hard Coded Password | VULNERABILITY | BLOCKER | 1 |
| Class variable fields should not have public accessibility | Public class variable fields do not respect the encapsulation principle and has three main disadvantages: Additional behavior such as validation cannot be added. The internal representation is exposed, and cannot be changed afterwards. Member values are subject to change from anywhere in the code and may not meet the programmer's assumptions. By using private attributes and accessor methods (set and get), unauthorized modifications are prevented. Noncompliant Code Example public class MyClass { public static final int SOME\_CONSTANT = 0; // Compliant - constants are not checked public String firstName; // Noncompliant } Compliant Solution public class MyClass { public static final int SOME\_CONSTANT = 0; // Compliant - constants are not checked private String firstName; // Compliant public String getFirstName() { return firstName; } public void setFirstName(String firstName) { this.firstName = firstName; } } Exceptions Because they are not modifiable, this rule ignores public final fields. See MITRE, CWE-493 - Critical Public Variable Without Final Modifier | VULNERABILITY | MINOR | 124 |
| 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 | 16 |
| "public static" fields should be constant | There is no good reason to declare a field "public" and "static" without also declaring it "final". Most of the time this is a kludge to share a state among several objects. But with this approach, any object can do whatever it wants with the shared state, such as setting it to null. Noncompliant Code Example public class Greeter { public static Foo foo = new Foo(); ... } Compliant Solution public class Greeter { public static final Foo FOO = new Foo(); ... } See MITRE, CWE-500 - Public Static Field Not Marked Final CERT OBJ10-J. - Do not use public static nonfinal fields | VULNERABILITY | MINOR | 113 |