Reporte de Reglas de SonarQube

Tabla de Contenido

JAVA

Track uses of "FIXME" tags (java:S1134)

Severidad: MAJOR

Resources:

Documentation

- CWE - CWE-546 - Suspicious Comment

Root_cause:

`FIXME` tags are commonly used to mark places where a bug is suspected, but which the developer wants to deal with later.

Sometimes the developer will not have the time or will simply forget to get back to that tag.

This rule is meant to track those tags and to ensure that they do not go unnoticed.

```
int divide(int numerator, int denominator) {
  return numerator / denominator;              // FIXME denominator value might be  0
}
```

Track uses of "TODO" tags (java:S1135)

Severidad: INFO

Resources:

- CWE - CWE-546 - Suspicious Comment

Root_cause:

Developers often use `TODO` tags to mark areas in the code where additional work or improvements are needed but are not implemented immediately. However, these `TODO` tags sometimes get overlooked or forgotten, leading to incomplete or unfinished code. This rule aims to identify and address unattended `TODO` tags to ensure a clean and maintainable codebase. This description explores why this is a problem and how it can be fixed to improve the overall code quality.

What is the potential impact?

Unattended `TODO` tags in code can have significant implications for the development process and the overall codebase.

Incomplete Functionality: When developers leave `TODO` tags without implementing the corresponding code, it results in incomplete functionality within the software. This can lead to unexpected behavior or missing features, adversely affecting the end-user experience.

Missed Bug Fixes: If developers do not promptly address `TODO` tags, they might overlook critical bug fixes and security updates. Delayed bug fixes can result in more severe issues and increase the effort required to resolve them later.

Impact on Collaboration: In team-based development environments, unattended `TODO` tags can hinder collaboration. Other team members might not be aware of the intended changes, leading to conflicts or redundant efforts in the codebase.

Codebase Bloat: The accumulation of unattended `TODO` tags over time can clutter the codebase and make it difficult to distinguish between work in progress and completed code. This bloat can make it challenging to maintain an organized and efficient codebase.

Addressing this code smell is essential to ensure a maintainable, readable, reliable codebase and promote effective collaboration among developers.

Noncompliant code example

```
void doSomething() {
  // TODO
}
```

Classes and enums with private members should have a constructor (java:S1258)

Severidad: MAJOR

Root_cause:

Non-abstract classes and enums with non-`static`, `private` members should explicitly initialize those members, either in a constructor or with a default value.

Noncompliant code example

```
class A { // Noncompliant
  private int field;
}
```

Compliant solution

```
class A {
  private int field;

  A(int field) {
    this.field = field;
  }
}
```

Exceptions

- Class implementing a Builder Pattern (name ending with "Builder").
- Java EE class annotated with:
  - ManagedBean
  - MessageDriven
  - Singleton
  - Stateful
  - Stateless
  - WebService
  - WebFilter
  - WebServlet
- Class and field annotated with:
  - Plexus Component Annotations
  - Maven Mojo
- Field annotated with:
  - Resource
  - EJB
  - Inject
  - Autowired
- Class annotated with:
  - Lombok Builder

"toString()" and "clone()" methods should not return null (java:S2225)

Severidad: MAJOR

Root_cause:

Calling `toString()` or `clone()` on an object should always return a string or an object. Returning `null` instead contravenes the method’s implicit contract.

Noncompliant code example

```
public String toString () {
  if (this.collection.isEmpty()) {
    return null; // Noncompliant
  } else {
    // ...
```

Compliant solution

```
public String toString () {
  if (this.collection.isEmpty()) {
    return "";
  } else {
    // ...
```

Resources:

- CWE - CWE-476 - NULL Pointer Dereference
- CERT, EXP01-J. - Do not use a null in a case where an object is required

---

Servlets should not have mutable instance fields (java:S2226)

Severidad: MAJOR

How_to_fix:

Noncompliant code example

If the field is never modified, declare it `final`.

```
public class MyServlet extends HttpServlet {
  String apiVersion = "0.9.1"; // Noncompliant, field changes are not thread-safe
}
```

Compliant solution

```
public class MyServlet extends HttpServlet {
  final String apiVersion = "0.9.1"; // Compliant, field cannot be changed
}
```

Noncompliant code example

If a field is modified within instance methods, refactor it into a local variable. That variable can be passed as an argument to other functions if needed.

```
public class MyServlet extends HttpServlet {

  String userName; // Noncompliant, field changes are not thread-safe

  @Override
  public void doGet(HttpServletRequest req, HttpServletResponse resp) throws IOException {
    userName = req.getParameter("userName"); // Different threads may write concurrently to userName
    resp.getOutputStream().print(getGreeting());
  }

  public String getGreeting() { // Unpredictable value in field userName
    return "Hello "+userName+"!";
  }
}
```

Compliant solution

```
public class MyServlet extends HttpServlet {

  @Override
  public void doGet(HttpServletRequest req, HttpServletResponse resp) throws IOException {
    String userName = req.getParameter("userName"); // Compliant, local variable instead instance field
    resp.getOutputStream().print(getGreeting(userName));
  }

  public String getGreeting(String userName) { // Compliant, method argument instead instance field
    return "Hello "+userName+"!";
  }
}
```

Noncompliant code example

If you still prefer instance state over local variables, consider using `ThreadLocal` fields. These fields provide a separate instance of their value for each thread.

```
public class MyServlet extends HttpServlet {

  String userName; // Noncompliant, field changes are not thread-safe

  @Override
  public void doGet(HttpServletRequest req, HttpServletResponse resp) throws IOException {
    userName = req.getParameter("userName"); // Different threads may write concurrently to userName
    resp.getOutputStream().print(getGreeting());
  }
  public String getGreeting() { // Unpredictable value in field userName
    return "Hello "+userName+"!";
  }
}
```

Compliant solution

```
public class MyServlet extends HttpServlet {

  final ThreadLocal<String> userName = new ThreadLocal<>(); // Compliant, field itself does not change

  @Override
  public void doGet(HttpServletRequest req, HttpServletResponse resp) throws IOException {
    userName.set(req.getParameter("userName")); // Compliant, own value provided for every thread
    resp.getOutputStream().print(getGreeting());
  }

  public String getGreeting() {
    return "Hello "+userName.get()+"!"; // Compliant, own value provided for every thread
  }
}
```

Noncompliant code example

If you have a use case that requires a shared instance state between threads, declare the corresponding fields as `static` to indicate your intention and awareness that there is only one instance of the servlet. However, the `static` modifier alone does not ensure thread safety. Make sure also to take countermeasures against possible race conditions.

```
public class MyServlet extends HttpServlet {

  public long timestampLastRequest; // Noncompliant, field changes are not thread-safe

  @Override
  public void doGet(HttpServletRequest req, HttpServletResponse resp) throws IOException {
    timestampLastRequest = System.currentTimeMillis();
    resp.getOutputStream().print(timestampLastRequest); // Race condition
  }
}
```

Compliant solution

```
public class MyServlet extends HttpServlet {

  public static long timestampLastRequest; // Compliant, sharing state is our intention

  @Override
  public void doGet(HttpServletRequest req, HttpServletResponse resp) throws IOException {
    long timestamp;
    synchronized (this) {
      timestamp = timestampLastRequest; // No race condition, synchronized get & set
      timestampLastRequest = System.currentTimeMillis();
    }
    resp.getOutputStream().print(timestamp);
  }
}
```

Resources:

Articles & blog posts

- Nikhil Ranjan: How to make thread safe servlet ?

Standards

- STIG Viewer - Application Security and Development: V-222567 - The application must not be vulnerable to race conditions.

Root_cause:

The `processHttpRequest` method and methods called from it can be executed by multiple threads within the same servlet instance, and state changes to the instance caused by these methods are, therefore, not threadsafe.

This is due to the servlet container creating only one instance of each servlet (`javax.servlet.http.HttpServlet`) and attaching a dedicated thread to each incoming HTTP request. The same problem exists for `org.apache.struts.action.Action` but with different methods.

To prevent unexpected behavior, avoiding mutable states in servlets is recommended. Mutable instance fields should either be refactored into local variables or made immutable by declaring them `final`.

Exceptions

- Fields annotated with `@javax.inject.Inject`, `@javax.ejb.EJB`, `@org.springframework.beans.factory.annotation.Autowired`, `@javax.annotation.Resource`
- Fields initialized in `init()` or `init(ServletConfig config)` methods

Boolean expressions should not be gratuitous (java:S2589)

Severidad: MAJOR

Introduction:

Gratuitous boolean expressions are conditions that do not change the evaluation of a program. This issue can indicate logical errors and affect the correctness of an application, as well as its maintainability.

How_to_fix:

Gratuitous boolean expressions are suspicious and should be carefully removed from the code.

First, the boolean expression in question should be closely inspected for logical errors. If a mistake was made, it can be corrected so the condition is no longer gratuitous.

If it becomes apparent that the condition is actually unnecessary, it can be removed. The associated control flow construct (e.g., the `if`-statement containing the condition) will be adapted or even removed, leaving only the necessary branches.

Noncompliant code example

```
public class MyClass {
    public void doThings(boolean b, boolean c) {
        a = true;
        if (a) {                // Noncompliant
          doSomething();
        }

        if (b && a) {           // Noncompliant; "a" is always "true"
          doSomething();
        }

        if (c || !a) {          // Noncompliant; "!a" is always "false"
          doSomething();
        }

        if (c || (!c && b)) {   // Noncompliant; c || (!c && b) is equal to c || b
          doSomething();
        }
    }
}
```

Compliant solution

```
public class MyClass {
    public void doThings(boolean b, boolean c) {
        a = true;
        if (foo(a)) {
          doSomething();
        }

        if (b) {
          doSomething();
        }

        if (c) {
          doSomething();
        }

        if (c || b) {
          doSomething();
        }
    }
}
```

Root_cause:

Control flow constructs like `if`-statements allow the programmer to direct the flow of a program depending on a boolean expression. However, if the condition is always true or always false, only one of the branches will ever be executed. In that case, the control flow construct and the condition no longer serve a purpose; they become *gratuitous*.

What is the potential impact?

The presence of gratuitous conditions can indicate a logical error. For example, the programmer *intended* to have the program branch into different paths but made a mistake when formulating the branching condition. In this case, this issue might result in a bug and thus affect the reliability of the application. For instance, it might lead to the computation of incorrect results.

Additionally, gratuitous conditions and control flow constructs introduce unnecessary complexity. The source code becomes harder to understand, and thus, the application becomes more difficult to maintain.

Resources:

Articles & blog posts

- CWE - CWE-571 - Expression is Always True
- CWE - CWE-570 - Expression is Always False

Value-based classes should not be used for locking (java:S3436)

Severidad: MAJOR

Root_cause:

According to the documentation,

> A program may produce unpredictable results if it attempts to distinguish two references to equal values of a value-based class, whether directly via reference equality or indirectly via an appeal to synchronization…

This is because value-based classes are intended to be wrappers for value types, which will be primitive-like collections of data (similar to `structs` in other languages) that will come in future versions of Java.

> Instances of a value-based class …
>
> - do not have accessible constructors, but are instead instantiated through factory methods which make no commitment as to the identity of returned instances;

This means that you can’t be sure you’re the only one trying to lock on any given instance of a value-based class, opening your code up to contention and deadlock issues.

Under Java 8 breaking this rule may not actually break your code, but there are no guarantees of the behavior beyond that.

This rule raises an issue when a known value-based class is used for synchronization. That includes all the classes in the `java.time` package except `Clock`; the date classes for alternate calendars, `HijrahDate`, `JapaneseDate`, `MinguoDate`, `ThaiBuddhistDate`; and the optional classes: `Optional`, `OptionalDouble`, `OptionalLong`, `OptionalInt`.

Note that this rule is automatically disabled when the project’s `sonar.java.source` is lower than `8`.

Noncompliant code example

```
Optional<Foo> fOpt = doSomething();
synchronized (fOpt) {  // Noncompliant
  // ...
}
```

Resources:

- Value-based classes

"default" clauses should be last (java:S4524)

Severidad: CRITICAL

Root_cause:

`switch` can contain a `default` clause for various reasons: to handle unexpected values, to show that all the cases were properly considered, etc.

For readability purposes, to help a developer quickly spot the default behavior of a `switch` statement, it is recommended to put the `default` clause at the end of the `switch` statement.

This rule raises an issue if the `default` clause is not the last one of the `switch`'s cases.

```
switch (param) {
  case 0:
    doSomething();
    break;
  default: // Noncompliant: default clause should be the last one
    error();
    break;
  case 1:
    doSomethingElse();
    break;
}
```

Lambdas should not have too many lines (java:S5612)

Severidad: MAJOR

Root_cause:

Lambdas (introduced with Java 8) are a very convenient and compact way to inject a behavior without having to create a dedicated class or method. But those lambdas should be used only if the behavior to be injected can be defined in a few lines of code, otherwise the source code can quickly become unreadable.

Consecutive AssertJ "assertThat" statements should be chained (java:S5853)

Severidad: MINOR

Root_cause:

AssertJ assertions methods targeting the same object can be chained instead of using multiple `assertThat`. It avoids duplication and increases the clarity of the code.

This rule raises an issue when multiples `assertThat` target the same tested value.

Noncompliant code example

```
assertThat(someList).hasSize(3);
assertThat(someList).contains("something");
```

Compliant solution

```
assertThat(someList)
  .hasSize(3)
  .contains("something");
```

Regexes containing characters subject to normalization should use the CANON_EQ flag (java:S5854)

Severidad: MAJOR

Root_cause:

Characters like `'é'` can be expressed either as a single code point or as a cluster of the letter `'e'` and a combining accent mark. Without the `CANON_EQ` flag, a regex will only match a string in which the characters are expressed in the same way.

Noncompliant code example

```
String s = "e\u0300";
Pattern p = Pattern.compile("é|ë|è"); // Noncompliant
System.out.println(p.matcher(s).replaceAll("e")); // print 'è'
```

Compliant solution

```
String s = "e\u0300";
Pattern p = Pattern.compile("é|ë|è", Pattern.CANON_EQ);
System.out.println(p.matcher(s).replaceAll("e")); // print 'e'
```

Regex alternatives should not be redundant (java:S5855)

Severidad: MAJOR

Introduction:

This rule raises an issue when multiple branches of a regex alternative match the same input.

Root_cause:

If an alternative in a regular expression only matches things that are already matched by another alternative, that alternative is redundant and serves no purpose.

In the best case this means that the offending subpattern is merely redundant and should be removed. In the worst case it's a sign that this regex does not match what it was intended to match and should be reworked.

Noncompliant code example

```
"[ab]|a"   // Noncompliant: the "|a" is redundant because "[ab]" already matches "a"
".*|a"     // Noncompliant: .* matches everything, so any other alternative is redundant
```

Compliant solution

```
"[ab]"
".*"
```

Regular expressions should be syntactically valid (java:S5856)

Severidad: CRITICAL

Root_cause:

Regular expressions have their own syntax that is understood by regular expression engines. Those engines will throw an exception at runtime if they are given a regular expression that does not conform to that syntax.

To avoid syntax errors, special characters should be escaped with backslashes when they are intended to be matched literally and references to capturing groups should use the correctly spelled name or number of the group.

To match a literal string instead of a regular expression, either all special characters should be escaped, the `Pattern.LITERAL` flag or methods that don't use regular expressions should be used.

Noncompliant code example

```
Pattern.compile("([");
str.matches("([");
str.replaceAll("([", "{");
str.matches("(\\w+-(\\d+)");
```

Compliant solution

```
Pattern.compile("\\(\\[");
Pattern.compile("([", Pattern.LITERAL);
str.equals("([");
str.replace("([", "{");
str.matches("(\\w+)-(\\d+)");
```

Similar tests should be grouped in a single Parameterized test (java:S5976)

Severidad: MAJOR

Root_cause:

When multiple tests differ only by a few hardcoded values they should be refactored as a single "parameterized" test. This reduces the chances of adding a bug and makes them more readable. Parameterized tests exist in most test frameworks (JUnit, TestNG, etc…).

The right balance needs of course to be found. There is no point in factorizing test methods when the parameterized version is a lot more complex than initial tests.

This rule raises an issue when at least 3 tests could be refactored as one parameterized test with less than 4 parameters. Only test methods which have at least one duplicated statement are considered.

Noncompliant code example

with JUnit 5

```
import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;

public class AppTest
{
    @Test
    void test_not_null1() {  // Noncompliant. The 3 following tests differ only by one hardcoded number.
      setupTax();
      assertNotNull(getTax(1));
    }

    @Test
    void test_not_null2() {
      setupTax();
      assertNotNull(getTax(2));
    }

    @Test
    void test_not_nul3l() {
      setupTax();
      assertNotNull(getTax(3));
    }

    @Test
    void testLevel1() {  // Noncompliant. The 3 following tests differ only by a few hardcoded numbers.
        setLevel(1);
        runGame();
        assertEquals(playerHealth(), 100);
    }

    @Test
    void testLevel2() {  // Similar test
        setLevel(2);
        runGame();
        assertEquals(playerHealth(), 200);
    }

    @Test
    void testLevel3() {  // Similar test
        setLevel(3);
        runGame();
        assertEquals(playerHealth(), 300);
    }
}
```

Compliant solution

```
import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;

public class AppTest
{

   @ParameterizedTest
   @ValueSource(ints = {1, 2, 3})
   void test_not_null(int arg) {
     setupTax();
     assertNotNull(getTax(arg));
   }

    @ParameterizedTest
    @CsvSource({
        "1, 100",
        "2, 200",
        "3, 300",
    })
    void testLevels(int level, int health) {
        setLevel(level);
        runGame();
        assertEquals(playerHealth(), health);
    }
}
```

Resources:

- Modern Best Practices for Testing in Java - Philipp Hauer
- JUnit 5 documentation - Parameterized tests
- Writing Parameterized Tests With JUnit 4
- TestNG documentation - Parameters

Tests should use fixed data instead of randomized data (java:S5977)

Severidad: MAJOR

Resources:

- [Modern Best Practices for Testing in Java - Philipp Hauer]
- [Jqwik test engine]

Root_cause:

Tests should always:

- Make sure that production code behaves as expected, including edge cases.
- Be easy to debug, i.e. understandable and reproducible.

Using random values in tests will not necessarily check edge cases, and it will make test logs a lot harder to read. It is better to use easily readable hardcoded values. If this makes your code bigger you can use helper functions.

There is one valid use case for random data in tests: when testing every value would make tests impractically slow. In this case the best you can do is use random to test every value on the long run. You should however make sure that random values are logged so that you can reproduce failures. Some libraries exist to make all this easier. You can for example use property-based testing libraries such as jqwik.

This rule raises an issue when `new Random()` or `UUID.randomUUID()` are called in test code.

Noncompliant code example

```
int userAge = new Random().nextInt(42);  // Noncompliant
UUID userID = UUID.randomUUID(); // Noncompliant
```

Compliant solution

```
int userAge = 31;
UUID userID = UUID.fromString("00000000-000-0000-0000-000000000001");
```

Exceptions in "throws" clauses should not be superfluous (java:S1130)

Severidad: MINOR

Root_cause:

Superfluous exceptions within `throws` clauses have negative effects on the readability and maintainability of the code. An exception in a `throws` clause is superfluous if it is:

- listed multiple times
- a subclass of another listed exception
- not actually thrown by any execution path of the method

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 boo() throws IOException { // Noncompliant; IOException cannot be thrown
  System.out.println("Hi!");
}
```

Compliant solution

```
void foo() throws MyException {}
void bar() throws Throwable {}
void boo() {
  System.out.println("Hi!");
}
```

Exceptions

The rule will not raise any issue for exceptions that cannot be thrown from the method body:

- in interface `default` methods
- in overriding and implementating 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

```
interface MyInterface {
  default void defaultMethod() throws IOException {
    System.out.println("Hi!");
  }
  void doSomething() throws IOException;
}

class A implements MyInterface {
  @Override
  void doSomething() throws IOException {
    System.out.println("Hi!");
  }

  public void emptyBody() throws IOException {}

  protected void singleThrowStatement() throws IOException {
    throw new UnsupportedOperationException("This method should be implemented in subclasses");
  }

  Object singleReturnStatement() throws IOException {
    return null;
  }

  /**
   * @throws IOException Overriding classes may throw this exception if they print values into a file
   */
  protected void overridable() throws IOException { // no issue, method is overridable and the exception has proper javadoc
    System.out.println("foo");
  }
}
```

Also, the rule will not raise issues on `RuntimeException`, or one of its sub-classes, because documenting runtime exceptions which could be thrown can ultimately help users of the method understand its behavior.

```
class B {
  int possibleDivisionByZero(int a, int b) throws ArithmeticException {
      return a / b;
  }
}
```

Strings literals should be placed on the left side when checking for equality (java:S1132)

Severidad: MINOR

Root_cause:

It is preferable to place string literals on the left-hand side of an `equals()` or `equalsIgnoreCase()` method call.

This prevents null pointer exceptions from being raised, as a string literal can never be null by definition.

Noncompliant code example

```
String myString = null;

System.out.println("Equal? " + myString.equals("foo"));                        // Noncompliant; will raise a NPE
System.out.println("Equal? " + (myString != null && myString.equals("foo")));  // Noncompliant; null check could be removed
```

Compliant solution

```
System.out.println("Equal?" + "foo".equals(myString));                         // properly deals with the null case
```

Deprecated code should be removed (java:S1133)

Severidad: INFO

Root_cause:

This rule is meant to be used as a way to track code which is marked as being deprecated. Deprecated code should eventually be removed.

Noncompliant code example

```
class Foo {
  /**
   * @deprecated
   */
  public void foo() {    // Noncompliant
  }
  @Deprecated            // Noncompliant
  public void bar() {
  }

  public void baz() {    // Compliant
  }
}
```

"Exception" should not be caught when not required by called methods (java:S2221)

Severidad: MINOR

Resources:

- CWE - CWE-396 - Declaration of Catch for Generic Exception

Root_cause:

Catching `Exception` seems like an efficient way to handle multiple possible exceptions. Unfortunately, it traps all exception types, both checked and runtime exceptions, thereby casting too broad a net. Indeed, was it really the intention of developers to also catch runtime exceptions? To prevent any misunderstanding, if both checked and runtime exceptions are really expected to be caught, they should be explicitly listed in the `catch` clause.

This rule raises an issue if `Exception` is caught when it is not explicitly thrown by a method in the `try` block.

Noncompliant code example

```
try {
  // do something that might throw an UnsupportedDataTypeException or UnsupportedEncodingException
} catch (Exception e) { // Noncompliant
  // log exception ...
}
```

Compliant solution

```
try {
  // do something
} catch (UnsupportedEncodingException|UnsupportedDataTypeException|RuntimeException e) {
  // log exception ...
}
```

or if runtime exceptions should not be caught:

```
try {
  // do something
} catch (UnsupportedEncodingException|UnsupportedDataTypeException e) {
  // log exception ...
}
```

Locks should be released on all paths (java:S2222)

Severidad: CRITICAL

Root_cause:

If a lock is acquired and released within a method, then it must be released along all execution paths of that method.

Failing to do so will expose the conditional locking logic to the method’s callers and hence be deadlock-prone.

Noncompliant code example

```
public class MyClass {
  public void doSomething() {
    Lock lock = new Lock();
    lock.lock(); // Noncompliant
    if (isInitialized()) {
      // ...
      lock.unlock();
    }
  }
}
```

Compliant solution

```
public class MyClass {
  public void doSomething() {
    Lock lock = new Lock();
    if (isInitialized()) {
      lock.lock();
      // ...
      lock.unlock();
    }
  }
}
```

Resources:

- CWE - CWE-459 - Incomplete Cleanup

Methods should not call same-class methods with incompatible "@Transactional" values (java:S2229)

Severidad: BLOCKER

Resources:

- Spring Framework 6 API: Enum Class propagation
- Spring Framework 6 API: Annotation Interface Transactional
- Spring 6 Documentation: Transaction Propagation

Articles & blog posts

- Baeldung: Transaction Propagation and Isolation in Spring @Transactional
- DZone: Spring Transaction Propagation in a Nutshell

How_to_fix:

Change the corresponding functions into a compatible propagation type.

Noncompliant code example

```
public void doTheThing() {
  // ...
  actuallyDoTheThing(); // Noncompliant, call from non-transactional to transactional
}

@Transactional
public void actuallyDoTheThing() {
  // ...
}
```

Compliant solution

```
@Transactional
public void doTheThing() {
  // ...
  actuallyDoTheThing(); // Compliant
}

@Transactional
public void actuallyDoTheThing() {
  // ...
}
```

Noncompliant code example

```
@Transactional
public void doTheThing() {
  // ...
  actuallyDoTheThing(); // Noncompliant, call from REQUIRED to REQUIRES_NEW
}

@Transactional(propagation = Propagation.REQUIRES_NEW)
public void actuallyDoTheThing() {
  // ...
}
```

Compliant solution

```
@Transactional
public void doTheThing() {
  // ...
  actuallyDoTheThing(); // Compliant, call from REQUIRED to MANDATORY
}

@Transactional(propagation = Propagation.MANDATORY)
```

```
public void actuallyDoTheThing() {
  // ...
}
```

Root_cause:

Transactional methods have a propagation type parameter in the @Transaction annotation that specifies the requirements about the transactional context in which the method can be called and how it creates, appends, or suspends an ongoing transaction.

When an instance that contains transactional methods is injected, Spring uses proxy objects to wrap these methods with the actual transaction code.

However, if a transactional method is called from another method in the same class, the `this` argument is used as the receiver instance instead of the injected proxy object, which bypasses the wrapper code. This results in specific transitions from one transactional method to another, which are not allowed:

From	To
non-`@Transactional`	MANDATORY, NESTED, REQUIRED, REQUIRES_NEW
MANDATORY	NESTED, NEVER, NOT_SUPPORTED, REQUIRES_NEW
NESTED	NESTED, NEVER, NOT_SUPPORTED, REQUIRES_NEW
NEVER	MANDATORY, NESTED, REQUIRED, REQUIRES_NEW
NOT_SUPPORTED	MANDATORY, NESTED, REQUIRED, REQUIRES_NEW
REQUIRED or `@Transactional`	NESTED, NEVER, NOT_SUPPORTED, REQUIRES_NEW
REQUIRES_NEW	NESTED, NEVER, NOT_SUPPORTED, REQUIRES_NEW
SUPPORTS	MANDATORY, NESTED, NEVER, NOT_SUPPORTED, REQUIRED, REQUIRES_NEW

Conditionally executed code should be reachable (java:S2583)

Severidad: MAJOR

Root_cause:

Conditional expressions which are always `true` or `false` can lead to unreachable code.

Noncompliant code example

```
a = false;
if (a) { // Noncompliant
  doSomething(); // never executed
}

if (!a || b) { // Noncompliant; "!a" is always "true", "b" is never evaluated
  doSomething();
} else {
  doSomethingElse(); // never executed
}
```

Exceptions

This rule will not raise an issue in either of these cases:

- When the condition is a single `final boolean`

```
final boolean debug = false;
//...
if (debug) {
  // Print something
}
```

- When the condition is literally `true` or `false`.

```
if (true) {
  // do something
}
```

In these cases it is obvious the code is as intended.

Resources:

- CWE - CWE-570 - Expression is Always False
- CWE - CWE-571 - Expression is Always True
- CERT, MSC12-C. - Detect and remove code that has no effect or is never executed

Overrides should match their parent class methods in synchronization (java:S3551)

Severidad: MAJOR

Resources:

- CERT, TSM00-J - Do not override thread-safe methods with methods that are not thread-safe

Root_cause:

When `@Overrides` of `synchronized` methods are not themselves `synchronized`, the result can be improper synchronization as callers rely on the thread-safety promised by the parent class.

Noncompliant code example

```
public class Parent {

  synchronized void foo() {
    //...
  }
}

public class Child extends Parent {

 @Override
  public void foo () {  // Noncompliant
    // ...
    super.foo();
  }
}
```

Compliant solution

```
public class Parent {

  synchronized void foo() {
    //...
  }
}

public class Child extends Parent {

  @Override
  synchronized void foo () {
    // ...
    super.foo();
  }
}
```

"Optional" should not be used for parameters (java:S3553)

Severidad: MAJOR

Root_cause:

The Java language authors have been quite frank that `Optional` was intended for use only as a return type, as a way to convey that a method may or may not return a value.

And for that, it’s valuable but using `Optional` on the input side increases the work you have to do in the method without really increasing the value. With an `Optional` parameter, you go from having 2 possible inputs: null and not-null, to three: null, non-null-without-value, and non-null-with-value. Add to that the fact that overloading has long been available to convey that some parameters are optional, and there’s really no reason to have `Optional` parameters.

The rule also checks for Guava’s `Optional`, as it was the inspiration for the JDK `Optional`. Although it is different in some aspects (serialization, being recommended for use as collection elements), using it as a parameter type causes exactly the same problems as for JDK `Optional`.

Noncompliant code example

```
public String sayHello(Optional<String> name) {  // Noncompliant
  if (name == null || !name.isPresent()) {
    return "Hello World";
  } else {
    return "Hello " + name;
  }
}
```

Compliant solution

```
public String sayHello(String name) {
  if (name == null) {
    return "Hello World";
  } else {
    return "Hello " + name;
  }
}
```

Exceptions

No issues will be raised if a method is overriding, as the developer has no control over the signature at this point.

```
@Override
public String sayHello(Optional<String> name) {
  if (name == null || !name.isPresent()) {
    return "Hello World";
  } else {
    return "Hello " + name;
  }
}
```

Alternatives in regular expressions should be grouped when used with anchors (java:S5850)

Severidad: MAJOR

Root_cause:

In regular expressions, anchors (`^`, `$`, `\A`, `\Z` and `\z`) have higher precedence than the `|` operator. So in a regular expression like `^alt1|alt2|alt3$`, `alt1` would be anchored to the beginning, `alt3` to the end and `alt2` wouldn't be anchored at all. Usually the intended behavior is that all alternatives are anchored at both ends. To achieve this, a non-capturing group should be used around the alternatives.

In cases where it is intended that the anchors only apply to one alternative each, adding (non-capturing) groups around the anchors and the parts that they apply to will make it explicit which parts are anchored and avoid readers misunderstanding the precedence or changing it because they mistakenly assume the precedence was not intended.

Noncompliant code example

```
^a|b|c$
```

Compliant solution

```
^(?:a|b|c)$
```

or

```
^a$|^b$|^c$
```

or, if you do want the anchors to only apply to `a` and `c` respectively:

```
(?:^a)|b|(?:c$)
```

Using slow regular expressions is security-sensitive (java:S5852)

Severidad: CRITICAL

Default:

Most of the regular expression engines use `backtracking` to try all possible execution paths of the regular expression when evaluating an input, in some cases it can cause performance issues, called `catastrophic backtracking` situations. In the worst case, the complexity of the regular expression is exponential in the size of the input, this means that a small carefully-crafted input (like 20 chars) can trigger `catastrophic backtracking` and cause a denial of service of the application. Super-linear regex complexity can lead to the same impact too with, in this case, a large carefully-crafted input (thousands chars).

This rule determines the runtime complexity of a regular expression and informs you of the complexity if it is not linear.

Note that, due to improvements to the matching algorithm, some cases of exponential runtime complexity have become impossible when run using JDK 9 or later. In such cases, an issue will only be reported if the project's target Java version is 8 or earlier.

Ask Yourself Whether

- The input is user-controlled.
- The input size is not restricted to a small number of characters.
- There is no timeout in place to limit the regex evaluation time.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

To avoid `catastrophic backtracking` situations, make sure that none of the following conditions apply to your regular expression.

In all of the following cases, catastrophic backtracking can only happen if the problematic part of the regex is followed by a pattern that can fail, causing the backtracking to actually happen. Note that when performing a full match (e.g. using `String.matches`), the end of the regex counts as a pattern that can fail because it will only succeed when the end of the string is reached.

- If you have a non-possessive repetition `r*` or `r*?`, such that the regex `r` could produce different possible matches (of possibly different lengths) on the same input, the worst case matching time can be exponential. This can be the case if `r` contains optional parts, alternations or additional repetitions (but not if the repetition is written in such a way that there's only one way to match it).
    - When using JDK 9 or later an optimization applies when the repetition is greedy and the entire regex does not contain any back references. In that case the runtime will only be polynomial (in case of nested repetitions) or even linear (in case of alternations or optional parts).
- If you have multiple non-possessive repetitions that can match the same contents and are consecutive or are only separated by an optional separator or a separator that can be matched by both of the repetitions, the worst case matching time can be polynomial (O(n^c) where c is the number of problematic repetitions). For example `a*b*` is not a problem because `a*` and `b*` match different things and `a*_a*` is not a problem because the repetitions are separated by a `'_'` and can't match that `'_'`. However, `a*a*` and `.*_.*` have quadratic runtime.
- If you're performing a partial match (such as by using `Matcher.find`, `String.split`, `String.replaceAll` etc.) and the regex is not anchored to the beginning of the string, quadratic runtime is especially hard to avoid because whenever a match fails, the regex engine will try again starting at the next index. This means that any unbounded repetition (even a possessive one), if it's followed by a pattern that can fail, can cause quadratic runtime on some inputs. For example `str.split("\\s*,")` will run in quadratic time on strings that consist entirely of spaces (or at least contain large sequences of spaces, not followed by a comma).

In order to rewrite your regular expression without these patterns, consider the following strategies:

- If applicable, define a maximum number of expected repetitions using the bounded quantifiers, like `{1,5}` instead of `+` for instance.
- Refactor `nested quantifiers` to limit the number of way the inner group can be matched by the outer quantifier, for instance this nested quantifier situation `(ba+)+` doesn't cause performance issues, indeed, the inner group can be matched only if there exists exactly one `b` char per repetition of the group.
- Optimize regular expressions with `possessive quantifiers` and `atomic grouping`.
- Use negated character classes instead of `.` to exclude separators where applicable. For example the quadratic regex `.*_.*` can be made linear by changing it to `[^_]*_.*`

Sometimes it's not possible to rewrite the regex to be linear while still matching what you want it to match. Especially when using partial matches, for which it is quite hard to avoid quadratic runtimes. In those cases consider the following approaches:

- Solve the problem without regular expressions
- Use an alternative non-backtracking regex implementations such as Google's RE2 or RE2/J.
- Use multiple passes. This could mean pre- and/or post-processing the string manually before/after applying the regular expression to it or using multiple regular expressions. One example of this would be to replace `str.split("\\s*,\\s*")` with `str.split(",")` and then trimming the spaces from the strings as a second step.
- When using `Matcher.find()`, it is often possible to make the regex infallible by making all the parts that could fail optional, which will prevent backtracking. Of course this means that you'll accept more strings than intended, but this can be handled by using capturing groups to check whether the optional parts were matched or not and then ignoring the match if they weren't. For example the regex `x*y` could be replaced with `x*(y)?` and then the call to `matcher.find()` could be replaced with `matcher.find() && matcher.group(1) != null`.

Sensitive Code Example

The first regex evaluation will never end in `JDK` <= 9 and the second regex evaluation will never end in any versions of the `JDK`:

```
java.util.regex.Pattern.compile("(a+)+").matcher(
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaa!").matches(); // Sensitive

java.util.regex.Pattern.compile("(h|h|ih(((i|a|c|c|a|i|i|j|b|a|i|b|a|a|j))+h)ahbfhba|c|i)*").matcher(
"hchcchicihcchciiicichhcichcihcchiihichiciiiihhcchi"+
"cchhcihchcihiihciichhccciccichcichiihcchcihhicchcciicchcccihiiihhihihihi"+
"chicihhcciccchihhhcchichchciihiicihciihccciciccicciiiiiiiiicihhhiiiihchccch"+
"chhhhiiihchihcccchhhiiiiiiiicicichicihcciciihichhhhchihciiihhiccccccciciihh"+
"ichicccchhicchicihihccichicciihcichccihhicicccccccichhhhihihhcchchihih"+
"iihhihihihicichihiiiihhhhihhhchhichiicihhiiiiihchccccchichci").matches(); // Sensitive
```

Compliant Solution

Possessive quantifiers do not keep backtracking positions, thus can be used, if possible, to avoid performance issues:

```
java.util.regex.Pattern.compile("(a+)++").matcher(
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaa!").matches(); // Compliant

java.util.regex.Pattern.compile("(h|h|ih(((i|a|c|c|a|i|i|j|b|a|i|b|a|a|j))+h)ahbfhba|c|i)*+").matcher(
"hchcchicihcchciiicichhcichcihcchiihichiciiiihhcchi"+
"cchhcihchcihiihciichhccciccichcichiihcchcihhicchcciicchcccihiiihhihihihi"+
"chicihhcciccchihhhcchichchciihiicihciihccccicicccicciiiiiiiiicihhhiiiihchccch"+
"chhhhiiihchihcccchhhiiiiiiiicicichicihcciciihichhhhchihciiihhiccccccciciihh"+
"ichicccchhicchicihihccichicciihcichccihhicicccccccichhhhihihhcchchihih"+
"iihhihihihicichihiiiihhhhihhhchhichiicihhiiiiihchccccchichci").matches(); // Compliant
```

See

- OWASP - Top 10 2017 Category A1 - Injection
- CWE - CWE-400 - Uncontrolled Resource Consumption
- CWE - CWE-1333 - Inefficient Regular Expression Complexity
- owasp.org - OWASP Regular expression Denial of Service - ReDoS
- stackstatus.net(archived) - Outage Postmortem - July 20, 2016
- regular-expressions.info - Runaway Regular Expressions: Catastrophic Backtracking
- docs.microsoft.com - Backtracking with Nested Optional Quantifiers

Root_cause:

Most of the regular expression engines use `backtracking` to try all possible execution paths of the regular expression when evaluating an input, in some cases it can cause performance issues, called `catastrophic backtracking` situations. In the worst case, the complexity of the regular expression is exponential in the size of the input, this means that a small carefully-crafted input (like 20 chars) can trigger `catastrophic backtracking` and cause a denial of service of the application. Super-linear regex complexity can lead to the same impact too with, in this case, a large carefully-crafted input (thousands chars).

This rule determines the runtime complexity of a regular expression and informs you of the complexity if it is not linear.

Note that, due to improvements to the matching algorithm, some cases of exponential runtime complexity have become impossible when run using JDK 9 or later. In such cases, an issue will only be reported if the project's target Java version is 8 or earlier.

How_to_fix:

Recommended Secure Coding Practices

To avoid `catastrophic backtracking` situations, make sure that none of the following conditions apply to your regular expression.

In all of the following cases, catastrophic backtracking can only happen if the problematic part of the regex is followed by a pattern that can fail, causing the backtracking to actually happen. Note that when performing a full match (e.g. using `String.matches`), the end of the regex counts as a pattern that can fail because it will only succeed when the end of the string is reached.

- If you have a non-possessive repetition `r*` or `r*?`, such that the regex `r` could produce different possible matches (of possibly different lengths) on the same input, the worst case matching time can be exponential. This can be the case if `r` contains optional parts, alternations or additional repetitions (but not if the repetition is written in such a way that there's only one way to match it).
  - When using JDK 9 or later an optimization applies when the repetition is greedy and the entire regex does not contain any back references. In that case the runtime will only be polynomial (in case of nested repetitions) or even linear (in case of alternations or optional parts).
- If you have multiple non-possessive repetitions that can match the same contents and are consecutive or are only separated by an optional separator or a separator that can be matched by both of the repetitions, the worst case matching time can be polynomial (O(n^c) where c is the number of problematic repetitions). For example `a*b*` is not a problem because `a*` and `b*` match different things and `a*_a*` is not a problem because the repetitions are separated by a `'_'` and can't match that `'_'`. However, `a*a*` and `.*_.*` have quadratic runtime.
- If you're performing a partial match (such as by using `Matcher.find`, `String.split`, `String.replaceAll` etc.) and the regex is not anchored to the beginning of the string, quadratic runtime is especially hard to avoid because whenever a match fails, the regex engine will try again starting at the next index. This means that any unbounded repetition (even a possessive one), if it's followed by a pattern that can fail, can cause quadratic runtime on some inputs. For example `str.split("\\s*,")` will run in quadratic time on strings that consist entirely of spaces (or at least contain large sequences of spaces, not followed by a comma).

In order to rewrite your regular expression without these patterns, consider the following strategies:

- If applicable, define a maximum number of expected repetitions using the bounded quantifiers, like `{1,5}` instead of `+` for instance.
- Refactor `nested quantifiers` to limit the number of way the inner group can be matched by the outer quantifier, for instance this nested quantifier situation `(ba+)+` doesn't cause performance issues, indeed, the inner group can be matched only if there exists exactly one `b` char per repetition of the group.
- Optimize regular expressions with `possessive quantifiers` and `atomic grouping`.
- Use negated character classes instead of `.` to exclude separators where applicable. For example the quadratic regex `.*_.*` can be made linear by changing it to `[^_]*_.*`

Sometimes it's not possible to rewrite the regex to be linear while still matching what you want it to match. Especially when using partial matches, for which it is quite hard to avoid quadratic runtimes. In those cases consider the following approaches:

- Solve the problem without regular expressions
- Use an alternative non-backtracking regex implementations such as Google's RE2 or RE2/J.
- Use multiple passes. This could mean pre- and/or post-processing the string manually before/after applying the regular expression to it or using multiple regular expressions. One example of this would be to replace `str.split("\\s*,\\s*")` with `str.split(",")` and then trimming the spaces from the strings as a second step.
- When using `Matcher.find()`, it is often possible to make the regex infallible by making all the parts that could fail optional, which will prevent backtracking. Of course this means that you'll accept more strings than intended, but this can be handled by using capturing groups to check whether the optional parts were matched or not and then ignoring the match if they weren't. For example the regex `x*y` could be replaced with `x*(y)?` and then the call to `matcher.find()` could be replaced with `matcher.find() && matcher.group(1) != null`.

Compliant Solution

Possessive quantifiers do not keep backtracking positions, thus can be used, if possible, to avoid performance issues:

```
java.util.regex.Pattern.compile("(a+)++").matcher(
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaa!").matches(); // Compliant

java.util.regex.Pattern.compile("(h|h|ih(((i|a|c|c|a|i|i|j|b|a|i|b|a|a|j))+h)ahbfhba|c|i)*+").matcher(
"hchcchicihcchciiicichhcichcihcchiihichiciiiihhcchi"+
"cchhcihchcihiihciichhccciccichcichiihcchcihhicchcciicchcccihiiihhihihihi"+
"chicihhcciccchihhhcchichchciihiicihciihcccciciccicciiiiiiiiicihhhiiiihchccch"+
"chhhhiiihchihcccchhhiiiiiiiiicicichicihcciciihichhhhchihciiihhiccccccciciihh"+
"ichicccchhicchicihihccichicciihcichccihhicicccccccichhhihihhcchchihih"+
"iihhihihihicichihiiiihhhhihhhchhichiicihhiiiiihchccccchichci").matches(); // Compliant
```

See

- OWASP - Top 10 2017 Category A1 - Injection
- CWE - CWE-400 - Uncontrolled Resource Consumption
- CWE - CWE-1333 - Inefficient Regular Expression Complexity
- owasp.org - OWASP Regular expression Denial of Service - ReDoS
- stackstatus.net(archived) - Outage Postmortem - July 20, 2016
- regular-expressions.info - Runaway Regular Expressions: Catastrophic Backtracking
- docs.microsoft.com - Backtracking with Nested Optional Quantifiers

Assess_the_problem:

Ask Yourself Whether

- The input is user-controlled.
- The input size is not restricted to a small number of characters.
- There is no timeout in place to limit the regex evaluation time.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

The first regex evaluation will never end in `JDK` <= 9 and the second regex evaluation will never end in any versions of the `JDK`:

```
java.util.regex.Pattern.compile("(a+)+").matcher(
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaa!").matches(); // Sensitive

java.util.regex.Pattern.compile("(h|h|ih(((i|a|c|c|a|i|i|j|b|a|i|b|a|a|j))+h)ahbfhba|c|i)*").matcher(
"hchcchicihcchciiicichhcichcihcchiihichiciiiihhcchi"+
"cchhcihchcihiihciichhccciccichcichiihcchcihhicchcciicchcccihiiihhihihihi"+
"chicihhcciccchihhhcchichchciihiicihciihcccciciccicciiiiiiiiicihhhiiiihchccch"+
"chhhhiiihchihcccchhhiiiiiiiiicicichicihcciciihichhhhchihciiihhiccccccciciihh"+
"ichicccchhicchicihihccichicciihcichccihhicicccccccichhhihihhcchchihih"+
"iihhihihihicichihiiiihhhhihhhchhichiicihhiiiiihchccccchichci").matches(); // Sensitive
```

Character classes should be preferred over reluctant quantifiers in regular expressions (java:S5857)

Severidad: MINOR

Root_cause:

Using reluctant quantifiers (also known as lazy or non-greedy quantifiers) in patterns can often lead to needless backtracking, making the regex needlessly inefficient and potentially vulnerable to catastrophic backtracking. Particularly when using `.*?` or `.+?` to match anything up to some terminating character, it is usually a better idea to instead use a greedily or possessively quantified negated character class containing the terminating character. For example `<.+?>` should be replaced with `<[^>]++>`.

Noncompliant code example

```
<.+?>
".*?"
```

Compliant solution

```
<[^>]++>
"[^"]*+"
```

or

```
<[^>]+>
"[^"]*"
```

Exceptions

This rule only applies in cases where the reluctant quantifier can easily be replaced with a negated character class. That means the repetition has to be terminated by a single character or character class. Patterns such as the following, where the alternatives without reluctant quantifiers are more complicated, are therefore not subject to this rule:

```
<!--.*?-->
/\*.*?\*/
```

Spring's ModelAndViewAssert assertions should be used instead of other assertions (java:S5970)

Severidad: MAJOR

Root_cause:

The Spring framework comes with dedicated classes to help writing better and simpler unit tests. In particular, when testing applications built on top of Spring MVC, it is recommended to use Spring’s `ModelAndViewAssert` assertions class, instead of manually testing MVC’s properties.

This rule raises an issue when Spring’s `ModelAndViewAssert` assertions should be used instead of manual testing.

Noncompliant code example

```
ModelAndView mav = getMyModelAndView();

Assert.assertEquals("register", mav.getViewName());
Assert.assertTrue((Boolean) mav.getModelMap().get("myAttribute"));
Assert.assertFalse((Boolean) mav.getModelMap().get("myAttribute"));
Assert.assertEquals(myObject, mav.getModelMap().get("myAttribute"));
```

Compliant solution

```
ModelAndView mav = getMyModelAndView();

ModelAndViewAssert.assertViewName(mav, "register");
ModelAndViewAssert.assertModelAttributeValue(mav, "myAttribute", Boolean.TRUE);
ModelAndViewAssert.assertModelAttributeValue(mav, "myAttribute", Boolean.FALSE);
ModelAndViewAssert.assertModelAttributeValue(mav, "myAttribute", myObject);
```

Resources:

- Unit Testing Spring MVC
- ModelAndViewAssert Javadoc

Tests should be stable (java:S5973)

Severidad: MAJOR

Resources:

- TestNG documentation - Annotations
- Spotify Engineering - Test Flakiness - Methods for identifying and dealing with flaky tests

Root_cause:

Unstable / flaky tests are tests which sometimes pass and sometimes fail, without any code change. Obviously, they slow down developments when developers have to rerun failed tests. However, the real problem is that you can’t completely trust these tests, they might fail for many different reasons and you don’t know if any of them will happen in production.

Some tools, such as TestNG, enable developers to automatically retry flaky tests. This might be acceptable as a temporary solution, but it should eventually be fixed. The more flaky tests you add, the more chances there are for a bug to arrive in production.

This rule raises an issue when the annotation `org.testng.annotations.Test` is given a `successPercentage` argument with a value lower than `100`.

Noncompliant code example

```
import org.testng.annotations.Test;

public class PercentageTest {
    @Test(successPercentage = 80, invocationCount = 10)  // Noncompliant. The test is allowed to fail 2 times.
    public void flakyTest() {
    }
}
```

Annotated Mockito objects should be initialized (java:S5979)

Severidad: BLOCKER

Resources:

- Mockito documentation - MockitoAnnotations
- Mockito documentation - MockitoRule

Root_cause:

Objects annotated with Mockito annotations `@Mock`, `@Spy`, `@Captor`, or `@InjectMocks` need to be initialized explicitly.

There are several ways to do this:

- Call `MockitoAnnotations.openMocks(this)` or `MockitoAnnotations.initMocks(this)` in a setup method
- Annotate test class with `@RunWith(MockitoJUnitRunner.class)` (JUnit 4)
- Annotate test class with `@ExtendWith(MockitoExtension.class)` (JUnit 5 Jupiter)
- Use `@Rule public MockitoRule rule = MockitoJUnit.rule();`

Test using uninitialized mocks will fail.

Note that this only applies to annotated Mockito objects. It is not necessary to initialize objects instantiated via `Mockito.mock()` or `Mockito.spy()`.

This rule raises an issue when a test class uses uninitialized mocks.

Noncompliant code example

```
public class FooTest { // Noncompliant: Mockito initialization missing
  @Mock private Bar bar;

  @Spy private Baz baz;

  @InjectMocks private Foo fooUnderTest;

  @Test
  void someTest() {
    // test something ...
  }

  @Nested
  public class Nested {
    @Mock
    private Bar bar;
  }
```

Compliant solution

```
@RunWith(MockitoJUnitRunner.class)
public class FooTest {
  @Mock private Bar bar;
  // ...
}

@ExtendWith(MockitoExtension.class)
public class FooTest {
  @Mock private Bar bar;
  // ...
}

public class FooTest {
  @Rule
  public MockitoRule rule = MockitoJUnit.rule();

  @Mock private Bar bar;
  // ...
}

public class FooTest {
  @Mock private Bar bar;
  // ...

  @BeforeEach
  void setUp() {
    MockitoAnnotations.openMocks(this);
  }
  // ...
}

public class FooTest {
  @Mock private Bar bar;
  // ...

  @Before
  void setUp() {
    MockitoAnnotations.initMocks(this);
  }
  // ...
}

@ExtendWith(MockitoExtension.class)
public class FooTest {
  @Nested
  public class Nested {
    @Mock
    private Bar bar;
  }
}
```

"@Autowired" should be used when multiple constructors are provided (java:S6829)

Severidad: MINOR

Introduction:

The `@Autowired` annotation in Spring is used for automatic dependency injection. It allows Spring to resolve and inject the required beans into your bean. For example to inject a `@Repository` object into a `@Service`.

How_to_fix:

Use the `@Autowired` annotation to specify which constructor to use for auto-wiring.

Noncompliant code example

```
@Component
public class ExampleClass { // Noncompliant, multiple constructors present and no @Autowired annotation to specify which one to use

    private final DependencyClass1 dependency1;

    public ExampleClass() {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    public ExampleClass(DependencyClass1 dependency1) {
        this.dependency1 = dependency1;
    }

    // ...
}
```

Compliant solution

```
@Component
public class ExampleClass {

    private final DependencyClass1 dependency1;

    public ExampleClass() {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    @Autowired
    public ExampleClass(DependencyClass1 dependency1) {
        this.dependency1 = dependency1;
    }

    // ...
}
```

Resources:

Documentation

- Spring - Annotation Config: Autowired

Articles & blog posts

- Java Guides - UnsatisfiedDependencyException in Spring Boot

Root_cause:

The Spring dependency injection mechanism cannot identify which constructor to use for auto-wiring when multiple constructors are present in a class. This ambiguity can cause the application to crash at runtime, and it makes the code less clear to understand and more complex to extend and maintain.

What is the potential impact?

- **Incorrect Instantiation**: the wrong constructor is selected for instantiation, leading to a bean not being correctly initialized.
- **Unsatisfied Dependency Exception**: the constructor selected by Spring requires beans that are not available in the Spring context.
- **Non-Deterministic Behavior**: the constructor selected by Spring can vary, based on the number of dependencies that can be satisfied at runtime, leading to unpredictable application behavior.
- **Maintainability Issues**: adding more constructors in the future could lead to further confusion and potential bugs.

Methods should not have too many return statements (java:S1142)

Severidad: MAJOR

Root_cause:

Having too many return statements in a method increases the method's essential complexity because the flow of execution is broken each time a return statement is encountered. This makes it harder to read and understand the logic of the method.

Noncompliant code example

With the default threshold of 3:

```
public boolean myMethod() { // Noncompliant; there are 4 return statements
  if (condition1) {
    return true;
  } else {
    if (condition2) {
      return false;
    } else {
      return true;
    }
  }
  return false;
}
```

Jump statements should not occur in "finally" blocks (java:S1143)

Severidad: CRITICAL

Resources:

- CWE - CWE-584 - Return Inside Finally Block
- CERT, ERR04-J. - Do not complete abruptly from a finally block

Root_cause:

Using `return`, `break`, `throw`, and so on from a `finally` block suppresses the propagation of any unhandled `Throwable` which was thrown in the `try` or `catch` block.

This rule raises an issue when a jump statement (`break`, `continue`, `return`, `throw`, and `goto`) would force control flow to leave a `finally` block.

Noncompliant code example

```
public static void main(String[] args) {
  try {
    doSomethingWhichThrowsException();
    System.out.println("OK");   // incorrect "OK" message is printed
  } catch (RuntimeException e) {
    System.out.println("ERROR");  // this message is not shown
  }
}

public static void doSomethingWhichThrowsException() {
  try {
    throw new RuntimeException();
  } finally {
    for (int i = 0; i < 10; i ++) {
      //...
      if (q == i) {
        break; // ignored
      }
    }

    /* ... */
    return;      // Noncompliant - prevents the RuntimeException from being propagated
  }
}
```

Compliant solution

```
public static void main(String[] args) {
  try {
    doSomethingWhichThrowsException();
    System.out.println("OK");
  } catch (RuntimeException e) {
    System.out.println("ERROR");  // "ERROR" is printed as expected
  }
}

public static void doSomethingWhichThrowsException() {
  try {
    throw new RuntimeException();
  } finally {
    for (int i = 0; i < 10; i ++) {
      //...
      if (q == i) {
        break; // ignored
      }
    }

    /* ... */
  }
}
```

Unused "private" methods should be removed (java:S1144)

Severidad: MAJOR

Introduction:

This rule raises an issue when a private method is never referenced in the code.

Root_cause:

A method that is never called is dead code, and 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.

This rule detects methods that are never referenced from inside a translation unit, and cannot be referenced from the outside.

Noncompliant code example

```
public class Foo implements Serializable
{
  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
{
  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 issues for:

- annotated methods
- methods with parameters that are annotated with `@javax.enterprise.event.Observes`

The 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.

Exit methods should not be called (java:S1147)

Severidad: BLOCKER

Resources:

- CWE - CWE-382 - Use of System.exit()
- CERT, ERR09-J. - Do not allow untrusted code to terminate the JVM

Root_cause:

Calling `System.exit(int status)` or `Rutime.getRuntime().exit(int status)` calls the shutdown hooks and shuts downs the entire Java virtual machine. Calling `Runtime.getRuntime().halt(int)` does an immediate shutdown, without calling the shutdown hooks, and skipping finalization.

Each of these methods should be used with extreme care, and only when the intent is to stop the whole Java process. For instance, none of them should be called from applications running in a J2EE container.

Noncompliant code example

```
System.exit(0);
Runtime.getRuntime().exit(0);
Runtime.getRuntime().halt(0);
```

Exceptions

These methods are ignored inside `main`.

A "while" loop should be used instead of a "for" loop (java:S1264)

Severidad: MINOR

Resources:

- Java SE - The for Statement

How_to_fix:

Noncompliant code example

```
for (;condition;) { /*...*/ } // Noncompliant; only the condition is specified
```

When only the condition expression is defined in a `for` loop, a `while` loop should be used instead to increase readability. A `while` loop consists of a single loop condition and allows a block of code to be executed repeatedly as long as the specified condition is true.

Compliant solution

```
while (condition) { /*...*/ }
```

Root_cause:

A `for` loop is a type of loop construct that allows a block of code to be executed repeatedly for a fixed number of times. The `for` loop is typically used when the number of iterations is known in advance and consists of three parts:

- The initialization statement is executed once at the beginning of the loop. It is used to initialize the loop counter or any other variables that may be used in the loop.
- The loop condition is evaluated at the beginning of each iteration, and if it is `true`, the code inside the loop is executed.
- The update statement is executed at the end of each iteration and is used to update the loop counter or any other variables that may be used in the loop.

```
for (initialization; termination; increment) { /*...*/ }
```

All three statements are optional. However, when the initialization and update statements are not used, it can be unclear to the reader what the loop counter is and how it is being updated. This can make the code harder to understand and maintain.

Reflection should not be used to check non-runtime annotations (java:S2109)

Severidad: MAJOR

Root_cause:

Denoted by the "@" symbol, annotations are metadata that can be added to classes, methods, and variables for various purposes such as documentation, code analysis, and runtime processing.

Annotations have retention policies that determine in which context they are retained and available for use. There are three retention policies for annotations:

- `RetentionPolicy.SOURCE` - Annotations are only available during compilation and code analysis. They are not included in the compiled class file and are not available at runtime. E.G. `@Override`, `@SuppressWarnings`
- `RetentionPolicy.CLASS` - Annotations are included in the compiled class file providing information to the compiler, but they are not retained by the JVM at runtime. This is the default retention policy. E.G. `@PreviewFeature`
- `RetentionPolicy.RUNTIME` - Annotations are included in the compiled class file and available at runtime. They can be accessed and used by the program through reflection. E.G. `@FunctionalInterface`, `@Deprecated`

It is important to understand that only annotations having the `RUNTIME` retention policy can be accessed at runtime using reflection. For example, the following if condition is true when the method argument is the `java.util.function.Function` class:

```
void execute(Class<?> cls) {
  if (cls.isAnnotationPresent(FunctionalInterface.class)) {
    // ...
  }
}
```

Therefore, it is an issue to use reflection in combination with annotations with the `SOURCE` or `CLASS` retention policy because they are not present at runtime. For example, in the JVM source code, the `hashCode()` method of the `Integer` class has the `@Override` annotation. However, the following if condition will always be false even if the method argument is the `Integer#hashCode()` method because `@Override` has the `SOURCE` retention policy:

```
void execute(Method method) {
  if (method.isAnnotationPresent(Override.class)) { // Noncompliant, if condition will always be false because
                                                    // @Override is declared with @Retention(RetentionPolicy.SOURCE)
    // ...
  }
}
```

This rule detects improper reflective access on annotations having the `SOURCE` or `CLASS` retention policy.

Resources:

Documentation

- Oracle SDK - java.lang.annotation.RetentionPolicy

"BigDecimal(double)" should not be used (java:S2111)

Severidad: MAJOR

Root_cause:

The `BigDecimal` is used to represents immutable, arbitrary-precision signed decimal numbers.

Differently from the `BigDecimal`, the `double` primitive type and the `Double` type have limited precision due to the use of double-precision 64-bit IEEE 754 floating point. Because of floating point imprecision, the `BigDecimal(double)` constructor can be somewhat unpredictable.

For example writing `new BigDecimal(0.1)` doesn’t create a BigDecimal which is exactly equal to 0.1, but it is equal to 0.1000000000000000055511151231257827021181583404541015625. This is because 0.1 cannot be represented exactly as a double (or, for that matter, as a binary fraction of any finite length).

Resources:

Documentation

- Oracle - BigDecimal
- CERT, NUM10-J - Do not construct BigDecimal objects from floating-point literals

How_to_fix:

Use `BigDecimal.valueOf`, which uses a string under the covers to eliminate floating point rounding errors, or the constructor that takes a `String` argument.

Noncompliant code example

```
double d = 1.1;

BigDecimal bd1 = new BigDecimal(d);   // Noncompliant
BigDecimal bd2 = new BigDecimal(1.1); // Noncompliant
```

Compliant solution

```
double d = 1.1;

BigDecimal bd1 = BigDecimal.valueOf(d); // Compliant
BigDecimal bd2 = new BigDecimal("1.1"); // Compliant
```

"URL.hashCode" and "URL.equals" should be avoided (java:S2112)

Severidad: MAJOR

How_to_fix:

Use the `URI` class until access to the resource is actually needed.

Noncompliant code example

```
public void checkUrl(URL url) {
  Set<URL> sites = new HashSet<URL>();               // Noncompliant

  URL homepage = new URL("http://sonarsource.com");  // Compliant
  if (homepage.equals(url)) {                        // Noncompliant
    // ...
  }
}
```

Compliant solution

```
public void checkUrl(URL url) {
  Set<URI> sites = new HashSet<URI>();               // Compliant

  URI homepage = new URI("http://sonarsource.com");  // Compliant
  URI uri = url.toURI();
  if (homepage.equals(uri)) {                        // Compliant
    // ...
  }
}
```

Resources:

- Oracle Java SE - java.net.URL

Root_cause:

The `equals` and `hashCode` methods of `java.net.URL` may trigger a name service lookup (typically DNS) to resolve the hostname or IP address. Depending on the configuration, and network status, this lookup can be time-consuming.

On the other hand, the `URI` class does not perform such lookups and is a better choice unless you specifically require the functionality provided by `URL`.

In general, it is better to use the `URI` class until access to the resource is actually needed, at which point you can convert the `URI` to a `URL` using `URI.toURL()`.

This rule checks for uses of `URL` 's in `Map` and `Set` , and for explicit calls to the `equals` and `hashCode` methods. It suggests reconsidering the use of `URL` in such scenarios to avoid potential performance issues related to name service lookups.

Collections should not be passed as arguments to their own methods (java:S2114)

Severidad: MAJOR

Root_cause:

Passing a collection as an argument to the collection’s own method is either an error - some other argument was intended - or simply nonsensical code.

Further, because some methods require that the argument remain unmodified during the execution, passing a collection to itself can result in undefined behavior.

Noncompliant code example

```
List <Object> objs = new ArrayList<Object>();
objs.add("Hello");

objs.add(objs); // Noncompliant; StackOverflowException if objs.hashCode() called
objs.addAll(objs); // Noncompliant; behavior undefined
objs.containsAll(objs); // Noncompliant; always true
objs.removeAll(objs); // Noncompliant; confusing. Use clear() instead
objs.retainAll(objs); // Noncompliant; NOOP
```

A secure password should be used when connecting to a database (java:S2115)

Severidad: BLOCKER

Introduction:

When accessing a database, an empty password should be avoided as it introduces a weakness.

How_to_fix:

The following code uses an empty password to connect to a Postgres database.

The vulnerability can be fixed by using a strong password retrieved from Properties. This `database.password` property is set during deployment. Its value should be strong and different for each database.

Noncompliant code example

```
Connection conn = DriverManager.getConnection("jdbc:derby:memory:myDB;create=true", "login", ""); // Noncompliant
```

Compliant solution

```
String password = System.getProperty("database.password");
Connection conn = DriverManager.getConnection("jdbc:derby:memory:myDB;create=true", "login", password);
```

Pitfalls

Hard-coded passwords

It could be tempting to replace the empty password with a hard-coded one. Hard-coding passwords in the code can pose significant security risks. Here are a few reasons why it is not recommended:

1. Security Vulnerability: Hard-coded passwords can be easily discovered by anyone who has access to the code, such as other developers or attackers. This can lead to unauthorized access to the database and potential data breaches.
2. Lack of Flexibility: Hard-coded passwords make it difficult to change the password without modifying the code. If the password needs to be updated, it would require recompiling and redeploying the code, which can be time-consuming and error-prone.
3. Version Control Issues: Storing passwords in code can lead to version control issues. If the code is shared or stored in a version control system, the password will be visible to anyone with access to the repository, which is a security risk.

To mitigate these risks, it is recommended to use secure methods for storing and retrieving passwords, such as using environment variables, configuration files, or secure key management systems. These methods allow for better security, flexibility, and separation of sensitive information from the codebase.

Resources:

- Java Properties

Standards

- OWASP - Top 10 2021 Category A7 - Identification and Authentication Failures
- OWASP - Top 10 2017 Category A2 - Broken Authentication
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- CWE - CWE-521 - Weak Password Requirements

Root_cause:

When a database does not require a password for authentication, it allows anyone to access and manipulate the data stored within it. Exploiting this vulnerability typically involves identifying the target database and establishing a connection to it without the need for any authentication credentials.

What is the potential impact?

Once connected, an attacker can perform various malicious actions, such as viewing, modifying, or deleting sensitive information, potentially leading to data breaches or unauthorized access to critical systems. It is crucial to address this vulnerability promptly to ensure the security and integrity of the database and the data it contains.

Unauthorized Access to Sensitive Data

When a database lacks a password for authentication, it opens the door for unauthorized individuals to gain access to sensitive data. This can include personally identifiable information (PII), financial records, intellectual property, or any other confidential information stored in the database. Without proper access controls in place, malicious actors can exploit this vulnerability to retrieve sensitive data, potentially leading to identity theft, financial loss, or reputational damage.

Compromise of System Integrity

Without a password requirement, unauthorized individuals can gain unrestricted access to a database, potentially compromising the integrity of the entire system. Attackers can inject malicious code, alter configurations, or manipulate data within the database, leading to system malfunctions, unauthorized system access, or even complete system compromise. This can disrupt business operations, cause financial losses, and expose the organization to further security risks.

Unwanted Modifications or Deletions

The absence of a password for database access allows anyone to make modifications or deletions to the data stored within it. This poses a significant risk, as unauthorized changes can lead to data corruption, loss of critical information, or the introduction of malicious content. For example, an attacker could modify financial records, tamper with customer orders, or delete important files, causing severe disruptions to business processes and potentially leading to financial and legal consequences.

Overall, the lack of a password configured to access a database poses a serious security risk, enabling unauthorized access, data breaches, system compromise, and unwanted modifications or deletions. It is essential to address this vulnerability promptly to safeguard sensitive data, maintain system integrity, and protect the organization from potential harm.

"hashCode" and "toString" should not be called on array instances (java:S2116)

Severidad: MAJOR

How_to_fix:

Use relevant static `Arrays` method.

- `Arrays.hashCode` or `Arrays.deepHashCode`
- `Arrays.toString` or `Arrays.deepToString`

Noncompliant code example

```
public static void main(String[] args) {
    String argStr = args.toString();       // Noncompliant
    int argHash = args.hashCode();         // Noncompliant
}
```

Compliant solution

```
public static void main(String[] args) {
    String argStr = Arrays.toString(args); // Compliant
    int argHash = Arrays.hashCode(args);   // Compliant
}
```

Root_cause:

The purpose of the `hashCode` method is to return a hash code based on the contents of the object. Similarly, the purpose of the `toString` method is to provide a textual representation of the object's contents.

Calling `hashCode()` and `toString()` directly on array instances should be avoided because the default implementations provided by the `Object` class do not provide meaningful results for arrays. `hashCode()` returns the array's "identity hash code", and `toString()` returns nearly the same value. Neither method's output reflects the array's contents.

Resources:

Documentation

- Oracle Java SE - Arrays API

Articles & blog posts

- Error Prone - ArrayHashCode

Methods with Spring proxying annotations should be public (java:S2230)

Severidad: MAJOR

Resources:

Documentation

- Spring Framework API - Annotation Interface Async
- Spring Framework API - Annotation Interface Transactional

Articles & blog posts

- Baeldung - How To Do @Async in Spring
- Stack Overflow - Spring @Async ignored
- Stack Overflow - Does Spring @Transactional attribute work on a private method?

How_to_fix:

Declare the method `public`. Note that this action alone does not resolve the issue of direct instance calls from within the same class (see rule S6809), but it is a required precondition to fix it.

Noncompliant code example

```
@Async
private Future<String> asyncMethodWithReturnType() { // Noncompliant, no proxy generated and
    return "Hellow, world!";                         // can only be invoked from same class
}
```

Compliant solution

```
@Async
public Future<String> asyncMethodWithReturnType() { // Compliant
    return "Hellow, world!";
}
```

Root_cause:

Marking a non-public method `@Async` or `@Transactional` is misleading because Spring does not recognize non-public methods, and so makes no provision for their proper invocation. Nor does Spring make provision for the methods invoked by the method it called.

Therefore marking a private method, for instance, @Transactional can only result in a runtime error or exception if the method is annotated as @Transactional.

"ResultSet.isLast()" should not be used (java:S2232)

Severidad: MAJOR

Resources:

Documentation

- [Java SE 8 API Specification: ResultSet.isLast()]

How_to_fix:

Refactor your code to use `ResultSet.next()` instead of `ResultSet.isLast()`. Be cautious of its different semantics and side effects on cursor positioning in the result set. Verify that your program logic is still valid under these side effects and otherwise adjust it.

Noncompliant code example

```
ResultSet results = stmt.executeQuery("SELECT name, address FROM PERSON");
StringBuilder sb = new StringBuilder();
while (results.next() && !results.isLast()) { // Noncompliant
  sb.append(results.getString("name") + ", ");
}
sb.append(results.getString("name"));
String formattedNames = sb.toString();
```

Compliant solution

```
ResultSet results = stmt.executeQuery("SELECT name, address FROM PERSON");
List<String> names = new ArrayList<>();
while (results.next()) { // Compliant, and program logic refactored
  names.add(results.getString("name"));
}
String formattedNames =  names.stream().collect(Collectors.joining(", "));
```

Root_cause:

There are several reasons to avoid using this method:

1. It is optionally available only for result sets of type `ResultSet.TYPE_FORWARD_ONLY`. Database drivers will throw an exception if not supported.
2. The method can be expensive to execute as the database driver may need to fetch ahead one row to determine whether the current row is the last in the result set. The documentation of the method explicitly mentions this fact.
3. What "the cursor is on the last row" means for an empty `ResultSet` is unclear. Database drivers may return `true` or `false` in this case .

`ResultSet.next()` is a good alternative to `ResultSet.isLast()` as it does not have the mentioned issues. It is always supported and, as per specification, returns `false` for empty result sets.

Parameters should be passed in the correct order (java:S2234)

Severidad: MAJOR

Root_cause:

When the names of parameters in a method call match the names of the method arguments, it contributes to clearer, more readable code. However, when the names match, but are passed in a different order than the method arguments, it indicates a mistake in the parameter order which will likely lead to unexpected results.

Noncompliant code example

```
public double divide(int divisor, int dividend) {
  return divisor/dividend;
}

public void doTheThing() {
  int divisor = 15;
  int dividend = 5;

  double result = divide(dividend, divisor);  // Noncompliant; operation succeeds, but result is unexpected
  //...
}
```

Compliant solution

```
public double divide(int divisor, int dividend) {
  return divisor/dividend;
}

public void doTheThing() {
  int divisor = 15;
  int dividend = 5;

  double result = divide(divisor, dividend);
  //...
}
```

"IllegalMonitorStateException" should not be caught (java:S2235)

Severidad: CRITICAL

Root_cause:

The `IllegalMonitorStateException` is an exception that occurs when a thread tries to perform an operation on an object's monitor that it does not own. This exception is typically thrown when a method like `wait()`, `notify()`, or `notifyAll()` is called outside a synchronized block or method.

`IllegalMonitorStateException` is specifically designed to be an unchecked exception to point out a programming mistake. This exception serves as a reminder for developers to rectify their code by correctly acquiring and releasing locks using synchronized blocks or methods. It also emphasizes the importance of calling monitor-related methods on the appropriate objects to ensure proper synchronization.

Catching and handling this exception can mask underlying synchronization issues and lead to unpredictable behavior.

Noncompliant code example

```
public void doSomething() {
  try {
    anObject.notify();
  } catch(IllegalMonitorStateException e) { // Noncompliant
  }
}
```

Compliant solution

```
public void doSomething() {
  synchronized(anObject) {
    anObject.notify();
  }
}
```

Resources:

- Oracle Java SE - IllegalMonitorStateException

Methods "wait(...)", "notify()" and "notifyAll()" should not be called on Thread instances (java:S2236)

Severidad: BLOCKER

Resources:

- Oracle Java SE - Thread
- Oracle Java SE - Object

Root_cause:

In Java, the `Thread` class represents a thread of execution. Synchronization between threads is typically achieved using objects or shared resources.

The methods `wait(…)`, `notify()`, and `notifyAll()` are related to the underlying object's monitor and are designed to be called on objects that act as locks or monitors for synchronization. These methods are available on Java `Object` and, therefore, automatically inherited by all objects, including `Thread.`

Calling these methods on a `Thread` may corrupt the behavior of the JVM, which relies on them to change the state of the thread (`BLOCKED, WAITING,`…).

Noncompliant code example

```
Thread myThread = new Thread(new RunnableJob());
...
myThread.wait(); // Noncompliant
```

Whitespace and control characters in literals should be explicit (java:S2479)

Severidad: CRITICAL

Root_cause:

Non-encoded control characters and whitespace characters are often injected in the source code because of a bad manipulation. They are either invisible or difficult to recognize, which can result in bugs when the string is not what the developer expects. If you actually need to use a control character use their encoded version (ex: ASCII `\n,\t,`… or Unicode `U+000D, U+0009,`…).

This rule raises an issue when the following characters are seen in a literal string:

- ASCII control character. (character index < 32 or = 127)
- Unicode whitespace characters.
- Unicode C0 control characters
- Unicode characters `U+200B, U+200C, U+200D, U+2060, U+FEFF, U+2028, U+2029`

No issue will be raised on the simple space character. Unicode `U+0020`, ASCII 32.

Noncompliant code example

```
String tabInside = "A   B";  // Noncompliant, contains a tabulation
String zeroWidthSpaceInside = "foobar"; // Noncompliant, it contains a U+200B character inside
char tab = '    ';
```

Compliant solution

```
String tabInside = "A\tB";  // Compliant, uses escaped value
String zeroWidthSpaceInside = "foo\u200Bbar";  // Compliant, uses escaped value
char tab = '\t';
```

Value-based objects should not be serialized (java:S3437)

Severidad: MINOR

Root_cause:

According to the documentation,

> A program may produce unpredictable results if it attempts to distinguish two references to equal values of a value-based class, whether directly via reference equality or indirectly via an appeal to synchronization, identity hashing, serialization…

For example (credit to Brian Goetz), imagine Foo is a value-based class:

```
Foo[] arr = new Foo[2];
arr[0] = new Foo(0);
arr[1] = new Foo(0);
```

Serialization promises that on deserialization of arr, elements 0 and 1 will not be aliased. Similarly, in:

```
Foo[] arr = new Foo[2];
arr[0] = new Foo(0);
arr[1] = arr[0];
```

Serialization promises that on deserialization of `arr`, elements 0 and 1 **will** be aliased.

While these promises are coincidentally fulfilled in current implementations of Java, that is not guaranteed in the future, particularly when true value types are introduced in the language.

This rule raises an issue when a `Serializable` class defines a non-transient, non-static field field whose type is a known serializable value-based class. Known serializable value-based classes are: all the classes in the `java.time` package except `Clock`; the date classes for alternate calendars: `HijrahDate`, `JapaneseDate`, `MinguoDate`, `ThaiBuddhistDate`.

Noncompliant code example

```
class MyClass implements Serializable {
  private HijrahDate date;  // Noncompliant; mark this transient
  // ...
}
```

Compliant solution

```
class MyClass implements Serializable {
  private transient HijrahDate date;
  // ...
}
```

Resources:

- Value-based classes

Track uses of disallowed classes (java:S3688)

Severidad: INFO

Root_cause:

This rule allows banning certain classes.

Noncompliant code example

Given parameters:

- className:java.lang.String

```
String name;  // Noncompliant
```

"@Deprecated" code marked for removal should never be used (java:S5738)

Severidad: MAJOR

How_to_fix:

Usage of deprecated classes, interfaces, and their methods explicitly marked for removal is discouraged. A developer should either migrate to alternative methods or refactor the code to avoid the deprecated ones.

Noncompliant code example

```
/**
 * @deprecated As of release 1.3, replaced by {@link #Fee}. Will be dropped with release 1.4.
 */
@Deprecated(forRemoval=true)
public class Foo { ... }

public class Bar {
  /**
   * @deprecated  As of release 1.7, replaced by {@link #doTheThingBetter()}
   */
  @Deprecated(forRemoval=true)
  public void doTheThing() { ... }

  public void doTheThingBetter() { ... }

  /**
   * @deprecated As of release 1.14 due to poor performances.
   */
  @Deprecated(forRemoval=false)
  public void doTheOtherThing() { ... }
}

public class Qix extends Bar {
  @Override
  public void doTheThing() { ... } // Noncompliant; don't override a deprecated method marked for removal
}

public class Bar extends Foo {  // Noncompliant; Foo is deprecated and will be removed

  public void myMethod() {
    Bar bar = new Bar();  // okay; the class isn't deprecated
    bar.doTheThing();  // Noncompliant; doTheThing method is deprecated and will be removed

    bar.doTheOtherThing(); // Okay; deprecated, but not marked for removal
  }
}
```

Root_cause:

With the introduction of Java 9, the standard annotation class `java.lang.Deprecated` has been updated with new parameters. Notably, a boolean parameter `forRemoval` has been added to clearly signify whether the deprecated code is intended to be removed in the future. This is indicated with `forRemoval=true`.

The javadoc of the annotation explicitly mentions the following:

> This annotation type has a boolean-valued element `forRemoval`. A value of `true` indicates intent to remove the annotated program element in a future version. A value of `false` indicates that use of the annotated program element is discouraged, but at the time the program element was annotated, there was no specific intent to remove it.

While it is generally recommended for developers to steer clear of using deprecated classes, interfaces, and their deprecated members, those already marked for removal will surely block you from upgrading your dependency. Usage of deprecated code should be avoided or eliminated as soon as possible to prevent accumulation and allow a smooth upgrade of dependencies.

The deprecated code is usually no longer maintained, can contain some bugs or vulnerabilities, and usually indicates that there is a better way to do the same thing. Removing it can even lead to significant improvement of your software.

Resources:

- CWE - CWE-477 - Use of Obsolete Functions
- CERT, MET02-J. - Do not use deprecated or obsolete classes or methods
- RSPEC-1874 for standard deprecation use

Names of regular expressions named groups should be used (java:S5860)

Severidad: MAJOR

Root_cause:

Why use named groups only to never use any of them later on in the code?

This rule raises issues every time named groups are:

- defined but never called anywhere in the code through their name;
- defined but called elsewhere in the code by their number instead;
- referenced while not defined.

Noncompliant code example

```
String date = "01/02";

Pattern datePattern = Pattern.compile("(?<month>[0-9]{2})/(?<year>[0-9]{2})");
Matcher dateMatcher = datePattern.matcher(date);

if (dateMatcher.matches()) {
  checkValidity(dateMatcher.group(1), dateMatcher.group(2));  // Noncompliant - numbers instead of names of groups are used
  checkValidity(dateMatcher.group("day")); // Noncompliant - there is no group called "day"
}

// ...

String score = "14:1";

Pattern scorePattern = Pattern.compile("(?<player1>[0-9]+):(?<player2>[0-9]+)"); // Noncompliant - named groups are never used
Matcher scoreMatcher = scorePattern.matcher(score);

if (scoreMatcher.matches()) {
  checkScore(score);
}
```

Compliant solution

```
String date = "01/02";

Pattern datePattern = Pattern.compile("(?<month>[0-9]{2})/(?<year>[0-9]{2})");
Matcher dateMatcher = datePattern.matcher(date);

if (dateMatcher.matches()) {
  checkValidity(dateMatcher.group("month"), dateMatcher.group("year"));
}

// ...

String score = "14:1";

Pattern scorePattern = Pattern.compile("(?<player1>[0-9]+):(?<player2>[0-9]+)");
Matcher scoreMatcher = scorePattern.matcher(score);

if (scoreMatcher.matches()) {
  checkScore(scoreMatcher.group("player1"));
  checkScore(scoreMatcher.group("player2"));
}
```

Or, using dedicated variables instead of group names:

```
String score = "14:1";

String player = "([0-9]+)";
String gameScore = player + ":" + player;

Pattern scorePattern = Pattern.compile(gameScore);
Matcher scoreMatcher = scorePattern.matcher(score);

if (scoreMatcher.matches()) {
  checkScore(score);
}
```

Case insensitive Unicode regular expressions should enable the "UNICODE_CASE" flag (java:S5866)

Severidad: MAJOR

Root_cause:

By default case insensitivity only affects letters in the ASCII range. This can be changed by either passing `Pattern.UNICODE_CASE` or `Pattern.UNICODE_CHARACTER_CLASS` as an argument to `Pattern.compile` or using `(?u)` or `(?U)` within the regex.

If not done, regular expressions involving non-ASCII letters will still handle those letters as being case sensitive.

Noncompliant code example

```
Pattern.compile("söme pättern", Pattern.CASE_INSENSITIVE);
str.matches("(?i)söme pättern");
str.matches("(?i:söme) pättern");
```

Compliant solution

```
Pattern.compile("söme pättern", Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE);
str.matches("(?iu)söme pättern");
str.matches("(?iu:söme) pättern");

// UNICODE_CHARACTER_CLASS implies UNICODE_CASE
Pattern.compile("söme pättern", Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CHARACTER_CLASS);
str.matches("(?iU)söme pättern");
str.matches("(?iU:söme) pättern");
```

Unicode-aware versions of character classes should be preferred (java:S5867)

Severidad: MINOR

Root_cause:

When using POSIX classes like `\p{Alpha}` without the `UNICODE_CHARACTER_CLASS` flag or when using hard-coded character classes like `"[a-zA-Z]"`, letters outside of the ASCII range, such as umlauts, accented letters or letter from non-Latin languages, won't be matched. This may cause code to incorrectly handle input containing such letters.

To correctly handle non-ASCII input, it is recommended to use Unicode classes like `\p{IsAlphabetic}`. When using POSIX classes, Unicode support should be enabled by either passing `Pattern.UNICODE_CHARACTER_CLASS` as a flag to `Pattern.compile` or by using `(?U)` inside the regex.

Noncompliant code example

```
Pattern.compile("[a-zA-Z]");
Pattern.compile("\\p{Alpha}");
```

Compliant solution

```
Pattern.compile("\\p{IsAlphabetic}"); // matches all letters from all languages
Pattern.compile("\\p{IsLatin}"); // matches latin letters, including umlauts and other non-ASCII variations
Pattern.compile("\\p{Alpha}", Pattern.UNICODE_CHARACTER_CLASS);
Pattern.compile("(?U)\\p{Alpha}");
```

Bean names should adhere to the naming conventions (java:S6830)

Severidad: MINOR

Resources:

Documentation

- Spring Framework Documentation - 3.3 Bean overview

Articles & blog posts

- Java Guides - Spring Boot Best Practices

How_to_fix:

Change the bean’s name to adhere to the naming conventions. Names should be camel-cased and start with a lowercase letter, for example, `myBean`.

Noncompliant code example

```
@Bean(name = "MyBean") // Noncompliant, the first letter of the name should be lowercase
public MyBean myBean() {
    ...
```

Compliant solution

```
@Bean(name = "myBean") // Compliant
public MyBean myBean() {
    ...
```

Noncompliant code example

```
@Service("my_service") // Noncompliant, the name should be camel-cased
public class MyService {
    ...
```

Compliant solution

```
@Service("myService") // Compliant
public class MyService {
    ...
```

Root_cause:

Consistent naming of beans is important for the readability and maintainability of the code. More precisely, according to the Spring documentation:

```
Naming beans consistently makes your configuration easier to read and understand. Also, if you use Spring AOP, it helps a lot when applying
```

Not following accepted conventions can introduce inconsistent naming, especially when multiple developers work on the same project, leading to technical debt.

The spring documentation establishes a naming convention that consists of camel-cased names with a leading lowercase letter.

This rule raises an issue when a bean name defined in one of the following annotations does not adhere to the naming convention:

- `@Bean`
- `@Configuration`
- `@Controller`
- `@Component`
- `@Qualifier`
- `@Repository`
- `@Service`

Non-singleton Spring beans should not be injected into singleton beans (java:S6832)

Severidad: MAJOR

Root_cause:

In Spring, singleton beans and their dependencies are initialized when the application context is created.

If a `Singleton` bean depends on a bean with a shorter-lived scope (like `Request` or `Session` beans), it retains the same instance of that bean, even when new instances are created for each Request or Session. This mismatch can cause unexpected behavior and bugs, as the Singleton bean doesn’t interact correctly with the new instances of the shorter-lived bean.

This rule raises an issue when non-singleton beans are injected into a singleton bean.

What is the potential impact?

When a `Singleton` bean has a dependency on a bean with a shorter-lived scope, it can lead to the following issues:

- **Data inconsistency**: any state change in the shorter-lived bean will not be reflected in the Singleton bean.
- **Incorrect behavior**: using the same instance of the shorter-lived bean, when a new instance is supposed to be created for each new request or session.
- **Memory leaks**: preventing garbage collection of a shorter-lived bean that allocates a significant amount of data over time.

Introduction:

In Spring the scope of a bean defines the lifecycle and visibility of that bean in the Spring container. There are six scopes:

- **Singleton**: default, one instance per Spring container
- **Prototype**: a new instance per bean request
- **Request**: a new instance per HTTP request
- **Session**: a new instance per HTTP session
- **Application**: a new instance per ServletContext
- **Websocket**: a new instance per Websocket session

The last four scopes mentioned, request, session, application and websocket, are only available in a web-aware application.

How_to_fix:

Inject a shorter-lived bean into a `Singleton` bean using **ApplicationContext**, **Factories** or **Providers**.

Noncompliant code example

When a `Singleton` bean auto-wires a `Request` bean, the dependency is resolved at instantiation time and thus the same instance is used for each HTTP request.

```
@Component
@Scope(value = WebApplicationContext.SCOPE_REQUEST, proxyMode = ScopedProxyMode.TARGET_CLASS)
public class RequestBean {
    //...
}

public class SingletonBean {
    @Autowired
    private final RequestBean requestBean; // Noncompliant, the same instance of RequestBean is used for each HTTP request.

    public RequestBean getRequestBean() {
        return requestBean;
    }
}
```

Compliant solution

Instead, use a `ObjectFactory<RequestBean>`, `ObjectProvider<RequestBean>`, or `Provider<RequestBean>` as injection point (as for JSR-330).

Such a dependency is resolved at runtime, allowing for actual injection of a new instance of the shorter-lived bean on each HTTP request.

```
@Component
@Scope(value = WebApplicationContext.SCOPE_REQUEST, proxyMode = ScopedProxyMode.TARGET_CLASS)
public class RequestBean {
    //...
}

public class SingletonBean {
    private final ObjectFactory<RequestBean> requestBeanFactory;

    @Autowired
    public SingletonBean(ObjectFactory<RequestBean> requestBeanFactory) {
        this.requestBeanFactory = requestBeanFactory;
    }

    public RequestBean getRequestBean() {
        return requestBeanFactory.getObject();
    }
}
```

Noncompliant code example

When a `Singleton` bean auto-wires a `Prototype` bean, the dependency is resolved at instantiation time and thus the same instance is used for each bean request.

```
@Component
@Scope("prototype")
public class PrototypeBean {
    public Object execute() {
      //...
    }
}

public class SingletonBean {
    private PrototypeBean prototypeBean;

    @Autowired
    public SingletonBean(PrototypeBean prototypeBean) { // Noncompliant, the same instance of PrototypeBean is used for each bean request.
      this.prototypeBean = prototypeBean;
    }

    public Object process() {
        return prototypeBean.execute();
    }
}
```

Compliant solution

Using the `ApplicationContext` to retrieve a new instance of a `Prototype` bean on each bean request.

```
@Component
@Scope("prototype")
public class PrototypeBean {
    public Object execute() {
      //...
    }
}

public class SingletonBean implements ApplicationContextAware {
    private ApplicationContext applicationContext;

    @Autowired
    public SingletonBean(ApplicationContext applicationContext) {
      this.applicationContext = applicationContext;
    }

    public Object process() {
        PrototypeBean prototypeBean = createPrototypeBean();
        return prototypeBean.execute();
    }

    protected PrototypeBean createPrototypeBean() {
        return this.applicationContext.getBean("prototypeBean", PrototypeBean.class);
    }
}
```

Resources:

Documentation

- Spring Framework - Factory Scopes
- Spring Framework - Beans Inject Named
- Spring Framework - Method Injection

Articles & blog posts

- Baeldung - Spring Bean Scopes

"@Controller" should be replaced with "@RestController" (java:S6833)

Severidad: MAJOR

Root_cause:

Classes annotated as `@Controller` in Spring are responsible for handling incoming web requests. When annotating methods or the entire controller with `@ResponseBody`, the return value of said methods will be serialized and set as the response body. In other words, it tells the Spring framework that this method does not produce a view. This mechanism is commonly used to create API endpoints.

Spring provides `@RestController` as a convenient annotation to replace the combination of `@Controller` and `@ResponseBody`. The two are functionally identical, so the single annotation approach is preferred.

This rule will raise an issue on a class that is annotated with `@Controller` if:

- the class is also annotated with `@ResponseBody` or

- all methods in said class are annotated with `@ResponseBody`.

Resources:

Articles & blog posts

- Spring Guides - Building a RESTful Web Service
- Baeldung - The Spring @Controller and @RestController Annotations
- Baeldung - Spring's RequestBody and ResponseBody Annotations

How_to_fix:

Replace the `@Controller` annotation with the `@RestController` annotation and remove all `@ResponseBody` annotations from the class and its methods.

Noncompliant code example

```
@Controller
@ResponseBody
public class MyController {
    @GetMapping("/hello")
    public String hello() {
        return "Hello World!";
    }
}
```

Compliant solution

```
@RestController
public class MyController {
    @GetMapping("/hello")
    public String hello() {
        return "Hello World!";
    }
}
```

Noncompliant code example

```
@Controller
public class MyController {
    @ResponseBody
    @GetMapping("/hello")
    public String hello() {
        return "Hello World!";
    }

    @ResponseBody
    @GetMapping("/foo")
    public String foo() {
        return "Foo";
    }
}
```

Compliant solution

```
@RestController
public class MyController {
    @GetMapping("/hello")
    public String hello() {
        return "Hello World!";
    }

    @GetMapping("/foo")
    public String foo() {
        return "Foo";
    }
}
```

Limited dependence should be placed on operator precedence (java:S864)

Severidad: MAJOR

Resources:

- CERT, EXP00-C. - Use parentheses for precedence of operation
- CERT, EXP53-J. - Use parentheses for precedence of operation
- CWE - CWE-783 - Operator Precedence Logic Error

Root_cause:

The rules of operator precedence are complicated and can lead to errors. For this reason, parentheses should be used for clarification in complex statements. However, this does not mean that parentheses should be gratuitously added around every operation.

This rule raises issues when && and `||` are used in combination, when assignment and equality or relational operators are used together in a condition, and for other operator combinations according to the following table:

	+, -, *, /, %	**<<, >>, >>>**	**&**	**^**	**|**
+, -, *, /, %		x	x	x	x
<<, >>, >>>	x		x	x	x
&	x	x		x	x
^	x	x	x		x
|	x	x	x	x	

This rule also raises an issue when the "true" or "false" expression of a ternary operator is not trivial and not wrapped inside parentheses.

Noncompliant code example

```
x = a + b - c;
x = a + 1 << b;  // Noncompliant
y = a == b ? a * 2 : a + b;  // Noncompliant

if ( a > b || c < d || a == d) {...}
if ( a > b && c < d || a == b) {...}  // Noncompliant
if (a = f(b,c) == 1) { ... } // Noncompliant; == evaluated first
```

Compliant solution

```
x = a + b - c;
x = (a + 1) << b;
y = a == b ? (a * 2) : (a + b);

if ( a > b || c < d || a == d) {...}
if ( (a > b && c < d) || a == b) {...}
if ( (a = f(b,c)) == 1) { ... }
```

Try-catch blocks should not be nested (java:S1141)

Severidad: MAJOR

Root_cause:

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.

Synchronized classes "Vector", "Hashtable", "Stack" and "StringBuffer" should not be used (java:S1149)

Severidad: MAJOR

Root_cause:

Early classes of the Java API, such as `Vector`, `Hashtable` and `StringBuffer`, were synchronized to make them thread-safe. However, synchronization has a significant negative impact on performance, even when using these collections from a single thread.

It is often best to use their non-synchronized counterparts:

- `ArrayList` or `LinkedList` instead of `Vector`
- `Deque` instead of `Stack`
- `HashMap` instead of `Hashtable`
- `StringBuilder` instead of `StringBuffer`

Even when used in synchronized contexts, you should think twice before using their synchronized counterparts, since their usage can be costly. If you are confident the usage is legitimate, you can safely ignore this warning.

Noncompliant code example

```
Vector<Cat> cats = new Vector<>();
```

Compliant solution

```
ArrayList<Cat> cats = new ArrayList<>();
```

Exceptions

Usage of these synchronized classes is ignored in the signatures of overriding methods.

```
@Override
public Vector getCats() {...} // Compliant
```

Package declaration should match source file directory (java:S1598)

Severidad: CRITICAL

Resources:

Articles & blog posts

- Baeldung - Fixing the "Declared package does not match the expected package" Error
- Stackoverflow - Why do java source files require package declarations?
- tutorialspoint - What are the best practices to keep in mind while using packages in Java?

How_to_fix:

Either move the source file so that the relative file path within the source directory matches the package name, or change the package name so that it matches the relative file path.

Noncompliant code example

```
// file: src/main/foo/Fubar.java
package com.foo.bar;

class Fubar {
}
```

Compliant solution

```
// file: src/main/com/foo/bar/Fubar.java
package com.foo.bar;

class Fubar {
}

// file: src/main/foo/Fubar.java
package foo;

class Fubar {
}
```

Root_cause:

The purpose of Java packages is to give structure to your project. A structure helps to mentally break down a project into smaller parts, simplifying readers' understanding of how components are connected and how they interact.

By convention, the source files' directory structure should replicate the project's package structure. This is for the following reasons:

1. The mapping between the package name and the location of the source file of a class is straightforward. That is, the path to the source file is easier to find for a given fully qualified class name.
2. If two different structures are applied to the same project - one to the packages but another to the source file directories - this confuses developers while not providing any benefit.
3. The directory structure of the class files generated by the compiler will match the package structure, no matter the source file's directory. It would not make sense to have one directory structure for the generated class files but a different one for the associated source files.

Similarly, a source directory should not have the character `.` in its name, as this would make it impossible to match the directory to the package structure.

Invalid "Date" values should not be used (java:S2110)

Severidad: MAJOR

Root_cause:

Whether the valid value ranges for `Date` fields start with 0 or 1 varies by field. For instance, month starts at 0, and day of month starts at 1. Enter a date value that goes past the end of the valid range, and the date will roll without error or exception. For instance, enter 12 for month, and you'll get January of the following year.

This rule checks for bad values used in conjunction with `java.util.Date`, `java.sql.Date`, and `java.util.Calendar`. Specifically, values outside of the valid ranges:

Field	Valid
month	0-11
date (day)	0-31
hour	0-23
minute	0-60
second	0-61

Note that this rule does not check for invalid leap years, leap seconds (second = 61), or invalid uses of the 31st day of the month.

Noncompliant code example

```
Date d = new Date();
d.setDate(25);
d.setYear(2014);
d.setMonth(12);  // Noncompliant; rolls d into the next year

Calendar c = new GregorianCalendar(2014, 12, 25);  // Noncompliant
if (c.get(Calendar.MONTH) == 12) {  // Noncompliant; invalid comparison
  // ...
}
```

Compliant solution

```
Date d = new Date();
d.setDate(25);
d.setYear(2014);
d.setMonth(11);

Calendar c = new Gregorian Calendar(2014, 11, 25);
if (c.get(Calendar.MONTH) == 11) {
  // ...
}
```

"writeObject" argument must implement "Serializable" (java:S2118)

Severidad: MAJOR

Resources:

- Oracle Java SE - Serializable
- Oracle Java SE - ObjectOutputStream

Root_cause:

Serialization is a platform-independent mechanism for writing the state of an object into a byte-stream. For serializing the object, we call the `writeObject()` method of `java.io.ObjectOutputStream` class. Only classes that implement `Serializable` or extend a class that does it can successfully be serialized (or de-serialized).

Attempting to write a class with the `writeObject` method of the `ObjectOutputStream` class that does not implement `Serializable` or extends a class that implements it, will throw an `IOException`.

How_to_fix:

The object class passed as an argument to the `writeObject` must implement `Serializable`.

Noncompliant code example

```
public class Vegetable {
  // ...
}

public class Menu {
  public void meal(ObjectOutputStream oos) throws IOException {
    Vegetable veg = new Vegetable();
    oos.writeObject(veg);  // Noncompliant
  }
}
```

Compliant solution

```
public class Vegetable implements Serializable {
  // ...
}

public class Menu {
  public void meal(ObjectOutputStream oos) throws IOException {
    Vegetable veg = new Vegetable();
    oos.writeObject(veg);
  }
}
```

"Random" objects should be reused (java:S2119)

Severidad: CRITICAL

How_to_fix:

Define and reuse the `Random` object.

Noncompliant code example

```
class MyClass {

    public void doSomethingCommon() {
      Random random = new Random();        // Noncompliant - new instance created with each invocation
      int rValue = random.nextInt();
    }
}
```

Compliant solution

```
class MyClass {
    private Random random = new Random();  // Compliant

    public void doSomethingCommon() {
      int rValue = this.random.nextInt();
    }
}
```

Root_cause:

Creating a new `Random` object each time a random value is needed is inefficient and may produce numbers that are not random, depending on the JDK. For better efficiency and randomness, create a single `Random`, store it, and reuse it.

The `Random()` constructor tries to set the seed with a distinct value every time. However, there is no guarantee that the seed will be randomly or uniformly distributed. Some JDK will use the current time as seed, making the generated numbers not random.

This rule finds cases where a new `Random` is created each time a method is invoked.

Exceptions

This rule doesn't apply to classes that use a `Random` in their constructors or the static `main` function and nowhere else.

Resources:

Documentation

- Oracle Java SE - Random

Articles & blog posts

- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- Baeldung - generating random number

"collect" should be used with "Streams" instead of "list::add" (java:S2203)

Severidad: MINOR

Root_cause:

While you can use either `forEach(list::add)` or `collect` with a `Stream`, `collect` is by far the better choice because it's automatically thread-safe and parallellizable.

Noncompliant code example

```
List<String> bookNames = new ArrayList<>();
books.stream().filter(book -> book.getIsbn().startsWith("0"))
                .map(Book::getTitle)
                .forEach(bookNames::add);  // Noncompliant
```

Compliant solution

```
List<String> bookNames = books.stream().filter(book -> book.getIsbn().startsWith("0"))
                .map(Book::getTitle)
                .collect(Collectors.toList());
```

"private" and "final" methods that don't access instance data should be "static" (java:S2325)

Severidad: MINOR

Root_cause:

Non-overridable methods (`private` or `final`) that don’t access instance data can be `static` to prevent any misunderstanding about the contract of the method.

Noncompliant code example

```
class Utilities {
  private static String magicWord = "magic";

  private String getMagicWord() { // Noncompliant
    return magicWord;
  }

  private void setMagicWord(String value) { // Noncompliant
    magicWord = value;
  }

}
```

Compliant solution

```
class Utilities {
  private static String magicWord = "magic";

  private static String getMagicWord() {
    return magicWord;
  }

  private static void setMagicWord(String value) {
    magicWord = value;
  }

}
```

Exceptions

When `java.io.Serializable` is implemented the following three methods are excluded by the rule:

- `private void writeObject(java.io.ObjectOutputStream out) throws IOException;`
- `private void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException;`
- `private void readObjectNoData() throws ObjectStreamException;`

Blocks should be synchronized on "private final" fields (java:S2445)

Severidad: MAJOR

Resources:

- CWE - CWE-412 - Unrestricted Externally Accessible Lock
- CWE - CWE-413 - Improper Resource Locking
- CERT, LCK00-J. - Use private final lock objects to synchronize classes that may interact with untrusted code

Root_cause:

Synchronizing on a class field synchronizes not on the field itself, but on the object assigned to it. So synchronizing on a non-`final` field makes it possible for the field’s value to change while a thread is in a block synchronized on the old value. That would allow a second thread, synchronized on the new value, to enter the block at the same time.

The story is very similar for synchronizing on parameters; two different threads running the method in parallel could pass two different object instances in to the method as parameters, completely undermining the synchronization.

Noncompliant code example

```
private String color = "red";

private void doSomething(){
  synchronized(color) {  // Noncompliant; lock is actually on object instance "red" referred to by the color variable
    //...
    color = "green"; // other threads now allowed into this block
    // ...
  }
  synchronized(new Object()) { // Noncompliant this is a no-op.
     // ...
  }
}
```

Compliant solution

```
private String color = "red";
private final Object lockObj = new Object();

private void doSomething(){
  synchronized(lockObj) {
    //...
    color = "green";
    // ...
  }
}
```

"notifyAll()" should be preferred over "notify()" (java:S2446)

Severidad: MAJOR

Root_cause:

`notify` and `notifyAll` both wake up sleeping threads waiting on the object’s monitor, but `notify` only wakes up one single thread, while `notifyAll` wakes them all up. Unless you do not care which specific thread is woken up, `notifyAll` should be used instead.

Noncompliant code example

```
class MyThread implements Runnable {
  Object lock = new Object();

  @Override
  public void run() {
    synchronized(lock) {
      // ...
      lock.notify();  // Noncompliant
    }
  }
}
```

Compliant solution

```
class MyThread implements Runnable {
  Object lock = new Object();

  @Override
  public void run() {
    synchronized(lock) {
      // ...
      lock.notifyAll();
    }
  }
}
```

Resources:

- CERT, THI02-J. - Notify all waiting threads rather than a single thread

---

Cipher Block Chaining IVs should be unpredictable (java:S3329)

Severidad: CRITICAL

Introduction:

This vulnerability exposes encrypted data to a number of attacks whose goal is to recover the plaintext.

Resources:

Standards

- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- CWE - CWE-329 - Not Using an Unpredictable IV with CBC Mode
- CWE - CWE-780 - Use of RSA Algorithm without OAEP
- NIST, SP-800-38A - Recommendation for Block Cipher Modes of Operation
- OWASP - Mobile AppSec Verification Standard - Cryptography Requirements
- OWASP - Mobile Top 10 2016 Category M5 - Insufficient Cryptography

How_to_fix:

Noncompliant code example

```
import java.nio.charset.StandardCharsets;
import java.security.NoSuchAlgorithmException;
import java.security.InvalidKeyException;
import java.security.InvalidAlgorithmParameterException;
import javax.crypto.Cipher;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;
import javax.crypto.NoSuchPaddingException;

public void encrypt(String key, String plainText) {

    byte[] RandomBytes = "7cVgr5cbdCZVw5WY".getBytes(StandardCharsets.UTF_8);

    IvParameterSpec iv   = new IvParameterSpec(RandomBytes);
    SecretKeySpec keySpec = new SecretKeySpec(key.getBytes(StandardCharsets.UTF_8), "AES");

    try {
        Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
        cipher.init(Cipher.ENCRYPT_MODE, keySpec, iv);

    } catch(NoSuchAlgorithmException|InvalidKeyException|
            NoSuchPaddingException|InvalidAlgorithmParameterException e) {
        // ...
    }
}
```

Compliant solution

In this example, the code explicitly uses a number generator that is considered **strong**.

```
import java.nio.charset.StandardCharsets;
import java.security.SecureRandom;
import java.security.NoSuchAlgorithmException;
import java.security.InvalidKeyException;
import java.security.InvalidAlgorithmParameterException;
import javax.crypto.Cipher;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;
import javax.crypto.NoSuchPaddingException;

public void encrypt(String key, String plainText) {

    SecureRandom random = new SecureRandom();
    byte[] randomBytes  = new byte[128];
    random.nextBytes(randomBytes);

    IvParameterSpec iv   = new IvParameterSpec(randomBytes);
    SecretKeySpec keySpec = new SecretKeySpec(key.getBytes(StandardCharsets.UTF_8), "AES");

    try {
        Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
        cipher.init(Cipher.ENCRYPT_MODE, keySpec, iv);

    } catch(NoSuchAlgorithmException|InvalidKeyException|
            NoSuchPaddingException|InvalidAlgorithmParameterException e) {
        // ...
    }
}
```

How does this work?

Use unique IVs

To ensure high security, initialization vectors must meet two important criteria:

- IVs must be unique for each encryption operation.
- For CBC and CFB modes, a secure FIPS-compliant random number generator should be used to generate unpredictable IVs.

The IV does not need be secret, so the IV or information sufficient to determine the IV may be transmitted along with the ciphertext.

In the previous non-compliant example, the problem is not that the IV is hard-coded.
It is that the same IV is used for multiple encryption attempts.

Root_cause:

Encryption algorithms are essential for protecting sensitive information and ensuring secure communications in a variety of domains. They are used for several important reasons:

- Confidentiality, privacy, and intellectual property protection
- Security during transmission or on storage devices
- Data integrity, general trust, and authentication

When selecting encryption algorithms, tools, or combinations, you should also consider two things:

1. No encryption is unbreakable.
2. The strength of an encryption algorithm is usually measured by the effort required to crack it within a reasonable time frame.

In the mode Cipher Block Chaining (CBC), each block is used as cryptographic input for the next block. For this reason, the first block requires an initialization vector (IV), also called a "starting variable" (SV).

If the same IV is used for multiple encryption sessions or messages, each new encryption of the same plaintext input would always produce the same ciphertext output. This may allow an attacker to detect patterns in the ciphertext.

What is the potential impact?

After retrieving encrypted data and performing cryptographic attacks on it on a given timeframe, attackers can recover the plaintext that encryption was supposed to protect.

Depending on the recovered data, the impact may vary.

Below are some real-world scenarios that illustrate the potential impact of an attacker exploiting the vulnerability.

Additional attack surface

By modifying the plaintext of the encrypted message, an attacker may be able to trigger additional vulnerabilities in the code. An attacker can further exploit a system to obtain more information.
Encrypted values are often considered trustworthy because it would not be possible for a third party to modify them under normal circumstances.

Breach of confidentiality and privacy

When encrypted data contains personal or sensitive information, its retrieval by an attacker can lead to privacy violations, identity theft, financial loss, reputational damage, or unauthorized access to confidential systems.

In this scenario, a company, its employees, users, and partners could be seriously affected.

The impact is twofold, as data breaches and exposure of encrypted data can undermine trust in the organization, as customers, clients and stakeholders may lose confidence in the organization’s ability to protect their sensitive data.

Legal and compliance issues

In many industries and locations, there are legal and compliance requirements to protect sensitive data. If encrypted data is compromised and the plaintext can be recovered, companies face legal consequences, penalties, or violations of privacy laws.

Tests should be kept in a dedicated source directory (java:S3414)

Severidad: MAJOR

Root_cause:

It is a good practice to isolate test classes in a separate package so that what is shipped to production is neither polluted by nor bloated with them. Further, including unit tests in code assemblies could affect build processes.

This rule raises an issue when test classes are found in projects containing non-test-related code.

Optional value should only be accessed after calling isPresent() (java:S3655)

Severidad: MAJOR

Resources:

- CWE - [CWE-476 - NULL Pointer Dereference]

Root_cause:

`Optional` value can hold either a value or not. The value held in the `Optional` can be accessed using the `get()` method, but it will throw a

`NoSuchElementException` if there is no value present. To avoid the exception, calling the `isPresent()` or `! isEmpty()` method should always be done before any call to `get()`.

Alternatively, note that other methods such as `orElse(...)`, `orElseGet(...)` or `orElseThrow(...)` can be used to specify what to do with an empty `Optional`.

Noncompliant code example

```
Optional<String> value = this.getOptionalValue();

// ...

String stringValue = value.get(); // Noncompliant
```

```
if (methodThatReturnsOptional().isEmpty()) {
  throw new NotFoundException();
}
String value = methodThatReturnsOptional().get(); // Noncompliant: indirect access, we consider that two consecutive calls can return diffe
```

Compliant solution

```
this.getOptionalValue().ifPresent(stringValue ->
  // Do something with stringValue
);
```

or

```
Optional<String> value = this.getOptionalValue();

// ...

if (value.isPresent()) {
  String stringValue = value.get();
}
```

or

```
Optional<String> value = this.getOptionalValue();

// ...

String stringValue = value.orElse("default");
```

```
Optional<String> optional = methodThatReturnsOptional();
if (optional.isEmpty()) {
  throw new NotFoundException();
}
String value = optional.get();
```

Cognitive Complexity of methods should not be too high (java:S3776)

Severidad: CRITICAL

Introduction:

This rule raises an issue when the code cognitive complexity of a function is above a certain threshold.

Root_cause:

Cognitive Complexity is a measure of how hard it is to understand the control flow of a unit of code. Code with high cognitive complexity is hard to read, understand, test, and modify.

As a rule of thumb, high cognitive complexity is a sign that the code should be refactored into smaller, easier-to-manage pieces.

Which syntax in code does impact cognitive complexity score?

Here are the core concepts:

- **Cognitive complexity is incremented each time the code breaks the normal linear reading flow.**
  This concerns, for example, loop structures, conditionals, catches, switches, jumps to labels, and conditions mixing multiple operators.
- **Each nesting level increases complexity.**
  During code reading, the deeper you go through nested layers, the harder it becomes to keep the context in mind.
- **Method calls are free**
  A well-picked method name is a summary of multiple lines of code. A reader can first explore a high-level view of what the code is performing then go deeper and deeper by looking at called functions content.
  Note: This does not apply to recursive calls, those will increment cognitive score.

The method of computation is fully detailed in the pdf linked in the resources.

Exceptions

`equals` and `hashCode` methods are ignored because they might be automatically generated and might end up being difficult to understand, especially in the presence of many fields.

Resources:

Documentation

- Sonar - Cognitive Complexity

Articles & blog posts

- Sonar Blog - 5 Clean Code Tips for Reducing Cognitive Complexity

How_to_fix:

Reducing cognitive complexity can be challenging.
Here are a few suggestions:

- **Extract complex conditions in a new function.**
  Mixed operators in condition will increase complexity. Extracting the condition in a new function with an appropriate name will reduce cognitive load.
- **Break down large functions.**
  Large functions can be hard to understand and maintain. If a function is doing too many things, consider breaking it down into smaller, more manageable functions. Each function should have a single responsibility.
- **Avoid deep nesting by returning early.**
  To avoid the nesting of conditions, process exceptional cases first and return early.

Extraction of a complex condition in a new function.

Noncompliant code example

The code is using a complex condition and has a cognitive cost of 3.

```
double calculateFinalPrice(User user, Cart cart) {
  double total = calculateTotal(cart);
  if (user.hasMembership()                           // +1 (if)
    && user.ordersCount() > 10                       // +1 (more than one condition)
    && user.isAccountActive()
    && !user.hasDiscount()
    || user.ordersCount() == 1) {                   // +1 (change of operator in condition)
    total = applyDiscount(user, total);
  }
  return total;
}
```

Compliant solution

Even if the cognitive complexity of the whole program did not change, it is easier for a reader to understand the code of the `calculateFinalPrice` function, which now only has a cognitive cost of 1.

```
double calculateFinalPrice(User user, Cart cart) {
  double total = calculateTotal(cart);
  if (isEligibleForDiscount(user)) {                 // +1 (if)
    total = applyDiscount(user, total);
  }
  return total;
}

boolean isEligibleForDiscount(User user) {
  return user.hasMembership()
    && user.ordersCount() > 10                       // +1 (more than one condition)
    && user.isAccountActive()
    && !user.hasDiscount()
    || user.ordersCount() == 1;                      // +1 (change of operator in condition)
}
```

Break down large functions.

Noncompliant code example

For example, consider a function that calculates the total price of a shopping cart, including sales tax and shipping.
Note: The code is simplified here, to illustrate the purpose. Please imagine there is more happening in the `for` loops.

```
double calculateTotal(Cart cart) {
  double total = 0;
  for (Item item : cart.items()) {       // +1 (for)
    total += item.price;
  }

  // calculateSalesTax
  for (Item item : cart.items()) {       // +1 (for)
    total += 0.2 * item.price;
  }

  //calculateShipping
  total += 5 * cart.items().size();

  return total;
}
```

This function could be refactored into smaller functions: The complexity is spread over multiple functions and the complex `calculateTotal` has now a complexity score of zero.

Compliant solution

```
double calculateTotal(Cart cart) {
  double total = 0;
  total = calculateSubtotal(cart, total);
  total += calculateSalesTax(cart, total);
  total += calculateShipping(cart, total);

  return total;
}

double calculateShipping(Cart cart, double total) {
  total += 5 * cart.items().size();
  return total;
}

double calculateSalesTax(Cart cart, double total) {
  for (Item item : cart.items()) {       // +1 (for)
    total += 0.2 * item.price;
  }
  return total;
}

double calculateSubtotal(Cart cart, double total) {
  for (Item item : cart.items()) {       // +1 (for)
    total += item.price;
  }
  return total;
}
```

Avoid deep nesting by returning early.

Noncompliant code example

The below code has a cognitive complexity of 6.

```
double calculateDiscount(double price, User user) {
  if (isEligibleForDiscount(user)) {      // +1 ( if )
    if (user.hasMembership()) {           // +2 ( nested if )
      return price * 0.9;
    } else if (user.ordersCount() == 1) { // +1 ( else )
      return price * 0.95;
    } else {                              // +1 ( else )
      return price;
    }
  } else {                                // +1 ( else )
    return price;
  }
}
```

Compliant solution

Checking for the edge case first flattens the `if` statements and reduces the cognitive complexity to 3.

```
double calculateDiscount(double price, User user) {
  if (!isEligibleForDiscount(user)) {     // +1 ( if )
    return price;
  }
  if (user.hasMembership()) {             // +1
    return price * 0.9;
  }
  if (user.ordersCount() == 1) {          // +1 ( if )
    return price * 0.95;
  }
  return price;
}
```

Pitfalls

As this code is complex, ensure that you have unit tests that cover the code before refactoring.

Assertions should not compare an object to itself (java:S5863)

Severidad: MAJOR

Root_cause:

Assertions comparing an object to itself are more likely to be bugs due to developer’s carelessness.

This rule raises an issue when the actual expression matches the expected expression.

Noncompliant code example

```
assertThat(actual).isEqualTo(actual); // Noncompliant
```

Compliant solution

```
assertThat(actual).isEqualTo(expected);
```

Exceptions

In a unit test validating the `equals(...)` and `hashCode()` methods, it’s legitimate to compare an object to itself. This rule does not raise an issue for `isEqualTo`, `assertEquals` or `hasSameHashCodeAs` when the unit test name contains (case insensitive): `equal`, `hash_?code`, `object_?method`. For example, in tests with the following names: `test_equals`, `testEqual`, `test_hashCode`, `test_hash_code`, `test_object_methods`.

```
class MyClassTest {
  @Test
  void test_equals_and_hash_code() {
    MyClass obj = new MyClass();
    assertThat(obj).isEqualTo(obj); // Compliant
    assertThat(obj).hasSameHashCodeAs(obj); // Compliant
  }
}
```

Unicode Grapheme Clusters should be avoided inside regex character classes (java:S5868)

Severidad: MAJOR

Root_cause:

When placing Unicode Grapheme Clusters (characters which require to be encoded in multiple Code Points) inside a character class of a regular expression, this will likely lead to unintended behavior.

For instance, the grapheme cluster c̈ requires two code points: one for `'c'`, followed by one for the *umlaut* modifier `'\u{0308}'`. If placed within a character class, such as `[c̈]`, the regex will consider the character class being the enumeration `[c\u{0308}]` instead. It will, therefore, match every `'c'` and every *umlaut* that isn't expressed as a single codepoint, which is extremely unlikely to be the intended behavior.

This rule raises an issue every time Unicode Grapheme Clusters are used within a character class of a regular expression.

Noncompliant code example

```
"cc̈d̈d".replaceAll("[c̈d̈]", "X"); // Noncompliant, print "XXXXXX" instead of expected "cXXd".
```

Compliant solution

```
"cc̈d̈d".replaceAll("c̈|d̈", "X"); // print "cXXd"
```

Character classes in regular expressions should not contain the same character twice (java:S5869)

Severidad: MAJOR

Root_cause:

Character classes in regular expressions are a convenient way to match one of several possible characters by listing the allowed characters or ranges of characters. If the same character is listed twice in the same character class or if the character class contains overlapping ranges, this has no effect.

Thus duplicate characters in a character class are either a simple oversight or a sign that a range in the character class matches more than is intended or that the author misunderstood how character classes work and wanted to match more than one character. A common example of the latter mistake is trying to use a range like `[0-99]` to match numbers of up to two digits, when in fact it is equivalent to `[0-9]`. Another common cause is forgetting to escape the `-` character, creating an unintended range that overlaps with other characters in the character class.

Noncompliant code example

```
str.matches("[0-99]") // Noncompliant, this won't actually match strings with two digits
str.matches("[0-9.-_]") // Noncompliant, .-_ is a range that already contains 0-9 (as well as various other characters such as capital lett
```

Compliant solution

```
str.matches("[0-9]{1,2}")
str.matches("[0-9.\\-_]")
```

"@Qualifier" should not be used on "@Bean" methods (java:S6831)

Severidad: MAJOR

Resources:

Documentation

- Spring Framework - Using the @Bean Annotation
- Spring Framework - Using @Qualifier

Articles & blog posts

- Baeldung - Spring @Qualifier Annotation
- Baeldung - Spring Bean Annotations

Root_cause:

In Spring Framework, the `@Qualifier` annotation is typically used to disambiguate between multiple beans of the same type when auto-wiring dependencies. It is not necessary to use `@Qualifier` when defining a bean using the `@Bean` annotation because the bean's name can be explicitly specified using the `name` attribute or derived from the method name. Using `@Qualifier` on `@Bean` methods can lead to confusion and redundancy. Beans should be named appropriately using either the `name` attribute of the `@Bean` annotation or the method name itself.

Noncompliant code example

```
@Configuration
public class MyConfiguration {
  @Bean
  @Qualifier("myService")
  public MyService myService() {
    // ...
    return new MyService();
  }

  @Bean
  @Qualifier("betterService")
  public MyService aBetterService() {
    // ...
    return new MyService();
  }

  @Bean
  @Qualifier("evenBetterService")
  public MyService anEvenBetterService() {
    // ...
    return new MyService();
  }

  @Bean
  @Qualifier("differentService")
  public MyBean aDifferentService() {
    // ...
    return new MyBean();
  }
}
```

Compliant solution

```
@Configuration
public class MyConfiguration {
  @Bean
  public MyService myService() {
    // ...
    return new MyService();
  }

  @Bean(name="betterService")
  public MyService aBetterService() {
    // ...
    return new MyService();
  }

  @Bean(name="evenBetterService")
  public MyService anEvenBetterService() {
    // ...
    return new MyService();
  }

  @Bean(name="differentService")
  public MyBean aDifferentService() {
    // ...
    return new MyBean();
  }
}
```

Superfluous "@ResponseBody" annotations should be removed (java:S6837)

Severidad: MAJOR

How_to_fix:

Remove the `@ResponseBody` annotation from the class or method.

Noncompliant code example

```
@RestController
public class MyController {
  @ResponseBody // Noncompliant, the @RestController annotation already implies @ResponseBody
  @RequestMapping("/hello")
  public String hello() {
    return "Hello World!";
  }
}
```

Compliant solution

```
@RestController
public class MyController {
  @RequestMapping("/hello")
  public String hello() {
    return "Hello World!";
  }
}
```

Noncompliant code example

```
@RestController
@ResponseBody // Noncompliant, the @RestController annotation already implies @ResponseBody
public class MyController {
  @RequestMapping("/hello")
  public String hello() {
    return "Hello World!";
  }
}
```

Compliant solution

```
@RestController
public class MyController {
  @RequestMapping("/hello")
  public String hello() {
    return "Hello World!";
  }
}
```

Root_cause:

The Spring framework’s `@RestController` annotation is equivalent to using the `@Controller` and `@ResponseBody` annotations together. As such, it is redundant to add a `@ResponseBody` annotation when the class is already annotated with `@RestController`.

Resources:

Articles & blog posts

- Spring Guides - Building a RESTful Web Service
- Baeldung - The Spring @Controller and @RestController Annotations
- Baeldung - Spring’s RequestBody and ResponseBody Annotations

---

"@Bean" methods for Singleton should not be invoked in "@Configuration" when proxyBeanMethods is false (java:S6838)

Severidad: MAJOR

Resources:

Documentation

- Spring - Configuration - proxyBeanMethods
- Spring - Proxying Mechanisms
- Spring - Bean Annotation - Dependencies
- GitHub - CGLIB

Articles & blog posts

- Medium - Demystifying Proxy in Spring

Introduction:

Spring proxies are based on the **Proxy design pattern** and serve as intermediaries to other resources, offering extra features at a slight performance penalty. For example, they facilitate lazy resource initialization and data caching.

The `@Configuration` annotation enables this mechanism by default through the `proxyBeanMethods` attribute set to `true`. This ensures that the `@Bean` methods are proxied in order to enforce bean lifecycle behavior, e.g. to return shared singleton bean instances even in case of direct `@Bean` method calls in user code. This functionality is achieved via method interception, implemented through a runtime-generated **CGLIB** subclass.

Root_cause:

When setting the `proxyBeanMethods` attribute to `false` the `@Bean` methods are not proxied and this is similar to removing the `@Configuration` stereotype. In this scenario, `@Bean` methods within the `@Configuration` annotated class operate in *lite mode*, resulting in a new bean creation each time the method is invoked.

For `Singleton` beans, this could cause unexpected outcomes as the bean is created multiple times instead of being created once and cached.

The rule raises an issue when the `proxyBeanMethods` attribute is set to `false` and the `@Bean` method of a `Singleton` bean is directly invoked in the `@Configuration` annotated class code.

How_to_fix:

The issue can be fixed in the following ways:

- Not invoking the `@Bean` method directly, but rather injecting the bean in the context and using it, by means of `@Bean` method parameters.
- If the performance penalty is negligible, consider not disabling the `proxyBeanMethods` attribute, so that the `@Bean` methods are proxied and the bean lifecycle is enforced.

Noncompliant code example

In the example below, every instance of `PrototypeBean` will have a different instance of `SingletonBean`, as `singletonBean()` is called directly from `prototypeBean()`.

```
@Configuration(proxyBeanMethods = false)
class ConfigurationExample {
  @Bean
  public SingletonBean singletonBean() {
    return new SingletonBean();
  }

  @Bean
  @Scope("prototype")
  public PrototypeBean prototypeBean() {
    return new PrototypeBean(singletonBean()); // Noncompliant, the singletonBean is created every time a prototypeBean is created
  }

  class SingletonBean {
    // ...
  }

  class PrototypeBean {
    // ...

    public PrototypeBean(SingletonBean singletonBean) {
      // ...
    }

    // ...
  }
}
```

Compliant solution

The compliant solution relies on the `@Bean` method parameter to automatically inject the `SingletonBean` from the `ApplicationContext`. This way every instance of `PrototypeBean` will have the same instance of `SingletonBean`.

```
@Configuration(proxyBeanMethods = false)
class ConfigurationExample {
  @Bean
  public SingletonBean singletonBean() {
    return new SingletonBean();
  }

  @Bean
  @Scope("prototype")
  public PrototypeBean prototypeBean(SingletonBean singletonBean) { // Compliant, the singletonBean is injected in the context and used by
    return new PrototypeBean(singletonBean);
  }

  class SingletonBean {
    // ...
  }

  class PrototypeBean {
    // ...

    public PrototypeBean(SingletonBean singletonBean) {
      // ...
    }

    // ...
  }
}
```

Increment (++) and decrement (--) operators should not be used in a method call or mixed with other operators in an expression (java:S881)

Severidad: MAJOR

Root_cause:

The use of increment and decrement operators in method calls or in combination with other arithmetic operators is not recommended, because:

- It can significantly impair the readability of the code.
- It introduces additional side effects into a statement, with the potential for undefined behavior.
- It is safer to use these operators in isolation from any other arithmetic operators.

Noncompliant code example

```
u8a = ++u8b + u8c--;
foo = bar++ / 4;
```

Compliant solution

The following sequence is clearer and therefore safer:

```
++u8b;
u8a = u8b + u8c;
u8c--;
foo = bar / 4;
bar++;
```

Resources:

- CERT, EXP30-C. - Do not depend on the order of evaluation for side effects
- CERT, EXP50-CPP. - Do not depend on the order of evaluation for side effects
- CERT, EXP05-J. - Do not follow a write by a subsequent write or read of the same object within an expression

---

Track uses of "NOSONAR" comments (java:NoSonar)

Severidad: MAJOR

Root_cause:

Any issue to quality rule can be deactivated with the `NOSONAR` marker. This marker is pretty useful to exclude false-positive results but it can also be used abusively to hide real quality flaws.

This rule raises an issue when `NOSONAR` is used.

Redundant pairs of parentheses should be removed (java:S1110)

Severidad: MAJOR

Root_cause:

Parentheses can disambiguate the order of operations in complex expressions and make the code easier to understand.

```
a = (b * c) + (d * e); // Compliant: the intent is clear.
```

Redundant parentheses are parenthesis that do not change the behavior of the code, and do not clarify the intent. They can mislead and complexify the code. They should be removed.

Noncompliant code example

```
int x = ((y / 2 + 1)); // Noncompliant

if (a && ((x + y > 0))) { // Noncompliant
  return ((x + 1)); // Noncompliant
}
```

Compliant solution

```
int x = (y / 2 + 1);

if (a && (x + y > 0)) {
  return (x + 1);
}
```

The "Object.finalize()" method should not be called (java:S1111)

Severidad: MAJOR

Root_cause:

Before it reclaims storage from an object that is no longer referenced, the garbage collector calls `finalize()` on the object.

This is a good time to release resources held by the object.

Because the general contract is that the `finalize` method should only be called once per object, calling this method explicitly is misleading and does not respect this contract.

What is the potential impact?

An explicit call to an object’s finalize method will perform operations that most likely were supposed to be performed only when the object was not referenced anymore by any thread.

Since it is an acceptable practice to override the finalize method in any subclass of `Object`, by invoking it explicitly, we will run code that was designed to only be ran at a different time.

Noncompliant code example

```
public void dispose() throws Throwable {
  this.finalize();                       // Noncompliant
}
```

Resources:

- docs.oracle.com - Finalization of Class Instances
- CWE - CWE-586 - Explicit Call to Finalize()
- CERT, MET12-J. - Do not use finalizers

The "Object.finalize()" method should not be overridden (java:S1113)

Severidad: CRITICAL

Root_cause:

Before it reclaims storage from an object that is no longer referenced, the garbage collector calls `finalize()` on the object.

But there is no guarantee that this method will be called as soon as the last references to the object are removed.

It can be few microseconds to few minutes later.

For this reason relying on overriding the `finalize()` method to release resources or to update the state of the program is highly discouraged.

What is the potential impact?

More unexpected issues can be caused by relying on the `finalize()` method to perform important operations on the application state:

- The JVM might terminate without ever calling this method on a particular object, leaving an unexpected or incomplete state of the program
- Uncaught exceptions will be ignored inside this method, making it harder to detect issues that could have been logged otherwise
- Finalizer methods can also be invoked concurrently, even on single-threaded applications, making it hard to maintain desired program invariants

Noncompliant code example

```
public class MyClass {

  @Override
  protected void finalize() { // Noncompliant
    releaseSomeResources();
  }

}
```

Exceptions

It is allowed to override the `finalize()` method as `final` method with an empty body, to prevent the *finalizer attack* as described in *MET12-J-EX1*.

Resources:

- docs.oracle.com - Finalization of Class Instances
- CERT, MET12-J. - Do not use finalizers

Local variables should not shadow class fields (java:S1117)

Severidad: MAJOR

Root_cause:

Shadowing occurs when a local variable has the same name as a variable or a field in an outer scope.

This can lead to three main problems:

- Confusion: The same name can refer to different variables in different parts of the scope, making the code hard to read and understand.
- Unintended Behavior: You might accidentally use the wrong variable, leading to hard-to-detect bugs.
- Maintenance Issues: If the inner variable is removed or renamed, the code's behavior might change unexpectedly because the outer variable is now being used.

To avoid these problems, rename the shadowing, shadowed, or both identifiers to accurately represent their purpose with unique and meaningful names.

This rule focuses on variables in methods that shadow a field.

Noncompliant code example

```
class Foo {
  public int myField;

  public void doSomething() {
    int myField = 0; // Noncompliant
    // ...
  }
}
```

Resources:

Documentation

- CERT - DCL51-J. Do not shadow or obscure identifiers in subscopes

Related rules

- S2176 - Class names should not shadow interfaces or superclasses
- S2387 - Child class fields should not shadow parent class fields
- S4977 - Type parameters should not shadow other type parameters

---

Utility classes should not have public constructors (java:S1118)

Severidad: MAJOR

How_to_fix:

To prevent the class from being instantiated, you should define a non-public constructor. This will prevent the compiler from implicitly generating a public parameterless constructor.

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;
  }

}
```

Root_cause:

Whenever there are portions of code that are duplicated and do not depend on the state of their container class, they can be centralized inside a "utility class". A utility class is a class that only has static members, hence it should not be instantiated.

Exceptions

When a class contains `public static void main(String[] args)` method it is not considered as a utility class and will be ignored by this rule.

Labels should not be used (java:S1119)

Severidad: MAJOR

Root_cause:

Labels are not commonly used in Java, and many developers do not understand how they work. Moreover, their usage makes the control flow harder to follow, which reduces the code’s readability.

Noncompliant code example

```
int matrix[][] = {
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9}
};

outer: for (int row = 0; row < matrix.length; row++) {   // Non-Compliant
  for (int col = 0; col < matrix[row].length; col++) {
    if (col == row) {
      continue outer;
    }
    System.out.println(matrix[row][col]);                // Prints the elements under the diagonal, i.e. 4, 7 and 8
  }
}
```

Compliant solution

```
for (int row = 1; row < matrix.length; row++) {          // Compliant
  for (int col = 0; col < row; col++) {
    System.out.println(matrix[row][col]);                // Also prints 4, 7 and 8
  }
}
```

"Collections.EMPTY_LIST", "EMPTY_MAP", and "EMPTY_SET" should not be used (java:S1596)

Severidad: MINOR

Resources:

Documentation

- Oracle - Java™ Platform, Standard Edition 8 API Specification, Class Collections
- Oracle - The Java™ Tutorials - Raw Types

Articles & blog posts

- Baeldung - The Basics of Java Generics

Introduction:

This rule raises an issue when the `Collections.EMPTY_*` fields are used instead of the `Collections.empty*()` methods.

How_to_fix:

Use:

- `Collections.emptyList()` instead of `Collections.EMPTY_LIST`
- `Collections.emptySet()` instead of `Collections.EMPTY_SET`
- `Collections.emptyMap()` instead of `Collections.EMPTY_MAP`

In addition, there are variants of `Collections.empty*()` available also for other collection interfaces, such as `Collections.emptyIterator()`, `Collections.emptyNavigableMap()`, `Collections.emptySortedSet()`.

Noncompliant code example

```
List<String> collection1 = Collections.EMPTY_LIST;     // Noncompliant, raw List
Set<Float> collection2 = Collections.EMPTY_SET;        // Noncompliant, raw Set
Map<Int, String> collection3 = Collections.EMPTY_MAP;  // Noncompliant, raw Map
```

Compliant solution

```
List<String> collection1 = Collections.emptyList();    // Compliant, List<String>
Set<Float> collection2 = Collections.emptySet();       // Compliant, Set<Float>
Map<Int, String> collection3 = Collections.emptyMap(); // Compliant, Map<Int, String>
```

Root_cause:

Generic types (types with type parameters) have been introduced into Java with language version 1.5. If type parameters are specified for a class or method, it is still possible to ignore them to keep backward compatibility with older code, which is called the *raw type* of the class or interface.

Using raw type expressions is highly discouraged because the compiler cannot perform static type checking on them. This means that the compiler will not report typing errors about them at compile time, but a `ClassCastException` will be thrown during runtime.

In Java 1.5, generics were also added to the Java collections API, and the data structures in `java.util`, such as `List`, `Set`, or `Map`, now feature type parameters. `Collections.EMPTY_LIST`, `Collections.EMPTY_SET`, and `Collections.EMPTY_MAP` are relics from before generics, and they return raw lists, sets, or maps, with the limitations mentioned above.

"compareTo" results should not be checked for specific values (java:S2200)

Severidad: MINOR

Root_cause:

Assuming that a comparator or `compareTo` method always returns -1 or 1 if the first operand is less than or greater than the second is incorrect.

The specifications for both methods, `Comparator.compare` and `Comparable.compareTo`, state that their return value is "a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object." Even if a specific comparator always returns -1, 0, or 1, this is only an implementation detail, not the API contract developers can rely on.

How_to_fix:

Replace

- `== -1` with `< 0` (the first operand is less than the second)
- `== 1` with `> 0` (the first operand is greater than the second)
- `!= -1` with `>= 0` (the first operand is greater than or equal to the second)
- `!= 1` with `<= 0` (the first operand is less than or equal to the second)

Noncompliant code example

```
public class Main {

  boolean isAGreaterThanB(Comparable<Integer> a, Integer b) {
    return a.compareTo(b) == 1; // Noncompliant, check for constant return value
  }

  public static void main(String[] args) {
    ByteComparator comparator = new ByteComparator();
    if (comparator.compare((byte) 23, (byte) 42) == -1) { // Noncompliant, check for constant return value
      System.out.println("23 < 42");
    } else {
      System.out.println("23 >= 42");
    }
  }

  static class ByteComparator implements Comparator<Byte> {
    @Override
    public int compare(Byte a, Byte b) {
      return a - b;
    }
  }
}
```

Compliant solution

```
public class Main {

  boolean isAGreaterThanB(Comparable<Integer> a, Integer b) {
    return a.compareTo(b) > 0; // Compliant, check for positive return value
  }

  public static void main(String[] args) {
    ByteComparator comparator = new ByteComparator();
    if (comparator.compare((byte) 23, (byte) 42) < 0) { // Compliant, check for negative return value
      System.out.println("23 < 42");
    } else {
      System.out.println("23 >= 42");
    }
  }

  static class ByteComparator implements Comparator<Byte> {
    @Override
    public int compare(Byte a, Byte b) {
      return a - b;
    }
  }
}
```

Resources:

Documentation

- Java SE 8 API Specification: Comparable.compareTo
- Java SE 8 API Specification: Comparator.compare

Return values from functions without side effects should not be ignored (java:S2201)

Severidad: MAJOR

Root_cause:

When the call to a function doesn't have any side effects, what is the point of making the call if the results are ignored? In such case, either the function call is useless and should be dropped or the source code doesn't behave as expected.

To prevent generating any false-positives, this rule triggers an issue only on the following predefined list of immutable classes in the Java API :

- `java.lang.String`
- `java.lang.Boolean`
- `java.lang.Integer`
- `java.lang.Double`
- `java.lang.Float`
- `java.lang.Byte`
- `java.lang.Character`
- `java.lang.Short`
- `java.lang.StackTraceElement`
- `java.time.DayOfWeek`
- `java.time.Duration`
- `java.time.Instant`
- `java.time.LocalDate`
- `java.time.LocalDateTime`
- `java.time.LocalTime`
- `java.time.Month`
- `java.time.MonthDay`
- `java.time.OffsetDateTime`
- `java.time.OffsetTime`
- `java.time.Period`
- `java.time.Year`
- `java.time.YearMonth`
- `java.time.ZonedDateTime`
- `java.math.BigInteger`

- `java.math.BigDecimal`
- `java.util.Optional`

As well as methods of the following classes:

- `java.util.Collection`:
  - `size()`
  - `isEmpty()`
  - `contains(...)`
  - `containsAll(...)`
  - `iterator()`
  - `toArray()`
- `java.util.Map`:
  - `size()`
  - `isEmpty()`
  - `containsKey(...)`
  - `containsValue(...)`
  - `get(...)`
  - `getOrDefault(...)`
  - `keySet()`
  - `entrySet()`
  - `values()`
- `java.util.stream.Stream`
  - `toArray`
  - `reduce`
  - `collect`
  - `min`
  - `max`
  - `count`
  - `anyMatch`
  - `allMatch`
  - `noneMatch`
  - `findFirst`
  - `findAny`
  - `toList`

Noncompliant code example

```
public void handle(String command){
  command.toLowerCase(); // Noncompliant; result of method thrown away
  ...
}
```

Compliant solution

```
public void handle(String command){
  String formattedCommand = command.toLowerCase();
  ...
}
```

Exceptions

This rule will not raise an issue when both these conditions are met:

- The method call is in a `try` block with an associated `catch` clause.
- The method name starts with "parse", "format", "decode" or "valueOf" or the method is `String.getBytes(Charset)`.

```
private boolean textIsInteger(String textToCheck) {

    try {
        Integer.parseInt(textToCheck, 10); // OK
        return true;
    } catch (NumberFormatException ignored) {
        return false;
    }
}
```

Resources:

- CERT, EXP00-J. - Do not ignore values returned by methods

".equals()" should not be used to test the values of "Atomic" classes (java:S2204)

Severidad: MAJOR

Resources:

Documentation

- [Java SE 8 API Specification: Package "java.util.concurrent.atomic"]

Articles & blog posts

- [Programming.Guide: AtomicInteger and equals / Comparable]

How_to_fix:

- To compare the number value of two instances `a` and `b` of `AtomicInteger` or `AtomicLong`, use `a.get() == b.get()` instead of `a.equals(b)`.
- If you want to check for object identity, use `a == b` instead of `a.equals(b)` to clarify your intention.

Noncompliant code example

```
Boolean isSameNumberValue(AtomicLong a, AtomicLong b) {
  return a.equals(b); // Noncompliant, this is true only if a == b
}

Boolean isSameReference(AtomicLong a, AtomicLong b) {
  return a.equals(b); // Noncompliant, because misleading
}
```

Compliant solution

```
Boolean isSameNumberValue(AtomicLong a, AtomicLong b) {
  return a.get() == b.get(); // Compliant
}

Boolean isSameReference(AtomicLong a, AtomicLong b) {
  return a == b; // Compliant
}
```

Root_cause:

The `equals` method in `AtomicInteger` and `AtomicLong` returns `true` only if two instances are identical, not if they represent the same number value.

This is because `equals` is not part of the API contract of these classes, and they do not override the method inherited from `java.lang.Object`. Although both classes implement the `Number` interface, assertions about `equals` comparing number values are not part of that interface either. Only the API contract of implementing classes like `Integer`, `Long`, `Float`, `BigInteger`, etc., provides such assertions.

Wildcard imports should not be used (java:S2208)

Severidad: CRITICAL

Root_cause:

Using wildcards in imports may look cleaner as it reduces the number of lines in the import section and simplifies the code.
On the other hand, it makes the code harder to maintain:

- It reduces code readability as developers will have a hard time knowing where names come from.
- It could lead to conflicts between names defined locally and the ones imported.
- It could later raise conflicts on dependency upgrade or Java version migration, as a wildcard import that works today might be broken tomorrow.

That is why it is better to import only the specific classes or modules you need.

Exceptions

Static imports are ignored by this rule. For example:

```
import static java.lang.Math.*;
```

will not raise an issue;

How_to_fix:

Depending on your IDE you can solve this issue almost **automatically**:
Look for **Organize/Optimize imports** actions. These actions can also often be applied automatically on save.
Note: To make this work properly, you must adjust IDE settings to use a very high `allowed class count usage` before using wildcards.

Resolving this issue **manually** will require a step-by-step approach:

1. Remove one wildcard import and note down compilation failures.
2. For each missing class, import it back with the package prefix.
3. When the code compiles again, proceed with the next wildcard import.

Noncompliant code example

```
import java.sql.*; // Noncompliant
import java.util.*; // Noncompliant

private Date date; // Date class exists in java.sql and java.util. Which one is this?
```

Compliant solution

```
import java.sql.Date;
import java.util.List;
import java.util.ArrayList;

private Date date;
```

"static" members should be accessed statically (java:S2209)

Severidad: MAJOR

Root_cause:

While it is *possible* to access `static` members from a class instance, it’s bad form, and considered by most to be misleading because it implies to the readers of your code that there’s an instance of the member per class instance.

Noncompliant code example

```
public class A {
  public static int counter = 0;
}

public class B {
  private A first = new A();
  private A second = new A();

  public void runUpTheCount() {
    first.counter ++;  // Noncompliant
    second.counter ++;  // Noncompliant. A.counter is now 2, which is perhaps contrary to expectations
  }
}
```

Compliant solution

```
public class A {
  public static int counter = 0;
}

public class B {
  private A first = new A();
  private A second = new A();

  public void runUpTheCount() {
    A.counter ++;  // Compliant
    A.counter ++;  // Compliant
  }
}
```

Unused type parameters should be removed (java:S2326)

Severidad: MAJOR

Root_cause:

Type parameters that aren’t used are dead code, which can only distract and possibly confuse developers during maintenance. Therefore, unused type parameters should be removed.

Noncompliant code example

```
int <T> Add(int a, int b) // Noncompliant; <T> is ignored
{
  return a + b;
}
```

Compliant solution

```
int Add(int a, int b)
{
  return a + b;
}
```

Classes with only "static" methods should not be instantiated (java:S2440)

Severidad: MAJOR

Root_cause:

`static` methods can be accessed without an instance of the enclosing class, so there’s no reason to instantiate a class that has only `static` methods.

Noncompliant code example

```
public class TextUtils {
  public static String stripHtml(String source) {
    return source.replaceAll("<[^>]+>", "");
  }
}

public class TextManipulator {

  // ...

  public void cleanText(String source) {
    TextUtils textUtils = new TextUtils(); // Noncompliant

    String stripped = textUtils.stripHtml(source);

    //...
  }
}
```

Compliant solution

```
public class TextUtils {
  public static String stripHtml(String source) {
    return source.replaceAll("<[^>]+>", "");
  }
}

public class TextManipulator {

  // ...

  public void cleanText(String source) {
    String stripped = TextUtils.stripHtml(source);

    //...
  }
}
```

Resources:

Related rules

- S1118 - Utility classes should not have public constructors

---

Non-serializable objects should not be stored in "javax.servlet.http.HttpSession" instances (java:S2441)

Severidad: MAJOR

Resources:

- OWASP - Top 10 2021 Category A4 - Insecure Design
- CWE - CWE-579 - J2EE Bad Practices: Non-serializable Object Stored in Session

Root_cause:

`HttpSession` s are managed by web servers and can be serialized and stored on disk as the server manages its memory use in a process called "passivation" (and later restored during "activation").

Even though `HttpSession` does not extend `Serializable`, you must nonetheless assume that it will be serialized. If non-serializable objects are stored in the session, serialization might fail.

Noncompliant code example

```
public class Address {
  //...
}

HttpSession session = request.getSession();
session.setAttribute("address", new Address());  // Noncompliant; Address isn't serializable
```

Compliant solution

```
public class Address implements Serializable {
  //...
}

HttpSession session = request.getSession();
session.setAttribute("address", new Address());
```

Synchronizing on a "Lock" object should be avoided (java:S2442)

Severidad: MAJOR

Root_cause:

`java.util.concurrent.locks.Lock` offers far more powerful and flexible locking operations than are available with `synchronized` blocks. So synchronizing on a `Lock` instance throws away the power of the object, as it overrides its better locking mechanisms. Instead, such objects should be locked and unlocked using one of their `lock` and `unlock` method variants.

Noncompliant code example

```
Lock lock = new MyLockImpl();
synchronized(lock) {  // Noncompliant
  // ...
}
```

Compliant solution

```
Lock lock = new MyLockImpl();
if (lock.tryLock()) {
  try {
    // ...
  } finally {
    lock.unlock();
  }
}
```

Resources:

- CERT, LCK03-J. - Do not synchronize on the intrinsic locks of high-level concurrency objects

---

Lazy initialization of "static" fields should be "synchronized" (java:S2444)

Severidad: CRITICAL

Root_cause:

In a multi-threaded situation, un-`synchronized` lazy initialization of static fields could mean that a second thread has access to a half-initialized object while the first thread is still creating it. Allowing such access could cause serious bugs. Instead. the initialization block should be `synchronized`.

Similarly, updates of such fields should also be `synchronized`.

This rule raises an issue whenever a lazy static initialization is done on a class with at least one `synchronized` method, indicating intended usage in multi-threaded applications.

Noncompliant code example

```
private static Properties fPreferences = null;

private static Properties getPreferences() {
        if (fPreferences == null) {
            fPreferences = new Properties(); // Noncompliant
            fPreferences.put("loading", "true");
            fPreferences.put("filterstack", "true");
            readPreferences();
        }
        return fPreferences;
    }
}
```

Compliant solution

```
private static Properties fPreferences = null;

private static synchronized Properties getPreferences() {
        if (fPreferences == null) {
            fPreferences = new Properties();
            fPreferences.put("loading", "true");
            fPreferences.put("filterstack", "true");
            readPreferences();
        }
        return fPreferences;
    }
}
```

Multiline blocks should be enclosed in curly braces (java:S2681)

Severidad: MAJOR

Root_cause:

Having inconsistent indentation and omitting curly braces from a control structure, such as an `if` statement or `for` loop, is misleading and can induce bugs.

This rule raises an issue when the indentation of the lines after a control structure indicates an intent to include those lines in the block, but the omission of curly braces means the lines will be unconditionally executed once.

The following patterns are recognized:

```
if (condition)
  firstActionInBlock();
  secondAction();  // Noncompliant: secondAction is executed unconditionally
thirdAction();

if (condition) firstActionInBlock(); secondAction();  // Noncompliant: secondAction is executed unconditionally

if (condition) firstActionInBlock();
  secondAction();  // Noncompliant: secondAction is executed unconditionally

if (condition); secondAction();  // Noncompliant: secondAction is executed unconditionally

for (int i = 0; i < array.length; i++)
  str = array[i];
  doTheThing(str);  // Noncompliant: executed only on the last element
```

Note that this rule considers tab characters to be equivalent to 1 space. When mixing spaces and tabs, a code may look fine in one editor but be confusing in another configured differently.

Resources:

- CWE - CWE-483 - Incorrect Block Delimitation
- CERT, EXP52-J. - Use braces for the body of an if, for, or while statement

---

Files opened in append mode should not be used with "ObjectOutputStream" (java:S2689)

Severidad: BLOCKER

Root_cause:

An `ObjectOutputStream` writes primitive data types and graphs of Java objects to an `OutputStream`. The objects can be read (reconstituted) using an `ObjectInputStream`.

When `ObjectOutputStream` is used with files opened in append mode, it can cause data corruption and unexpected behavior. This is because when `ObjectOutputStream` is created, it writes metadata to the output stream, which can conflict with the existing metadata when the file is opened in append mode. This can lead to errors and data loss.

When used with serialization, an `ObjectOutputStream` first writes the serialization stream header. This header should appear once per file at the beginning. When you're trying to read your object(s) back from the file, only the first one will be read successfully, and a `StreamCorruptedException` will be thrown after that.

Resources:

Articles & blog posts

- JBoss - AppendingObjectOutputStream

Documentation

- Oracle SE 20 - ObjectOutputStream

How_to_fix:

Open the file to use the default action (writes stream header).

Noncompliant code example

```
FileOutputStream fos = new FileOutputStream(fileName , true);  // fos opened in append mode
ObjectOutputStream out = new ObjectOutputStream(fos);  // Noncompliant
```

Compliant solution

```
FileOutputStream fos = new FileOutputStream(fileName);
ObjectOutputStream out = new ObjectOutputStream(fos);
```

Disabling CSRF protections is security-sensitive (java:S4502)

Severidad: CRITICAL

Root_cause:

A cross-site request forgery (CSRF) attack occurs when a trusted user of a web application can be forced, by an attacker, to perform sensitive actions that he didn't intend, such as updating his profile or sending a message, more generally anything that can change the state of the application.

The attacker can trick the user/victim to click on a link, corresponding to the privileged action, or to visit a malicious web site that embeds a hidden web request and as web browsers automatically include cookies, the actions can be authenticated and sensitive.

How_to_fix:

Recommended Secure Coding Practices

- Protection against CSRF attacks is strongly recommended:
  - to be activated by default for all unsafe HTTP methods.
  - implemented, for example, with an unguessable CSRF token
- Of course all sensitive operations should not be performed with safe HTTP methods like `GET` which are designed to be used only for information retrieval.

Compliant Solution

Spring Security CSRF protection is enabled by default, do not disable it:

```
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

  @Override
  protected void configure(HttpSecurity http) throws Exception {
    // http.csrf().disable(); // Compliant
  }
}
```

See

- OWASP - Top 10 2021 Category A1 - Broken Access Control
- CWE - CWE-352 - Cross-Site Request Forgery (CSRF)
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration

- OWASP - Cross-Site Request Forgery
- STIG Viewer - Application Security and Development: V-222603 - The application must protect from Cross-Site Request Forgery (CSRF) vulnerabilities.
- PortSwigger - Web storage: the lesser evil for session tokens

Assess_the_problem:

Ask Yourself Whether

- The web application uses cookies to authenticate users.
- There exist sensitive operations in the web application that can be performed when the user is authenticated.
- The state / resources of the web application can be modified by doing HTTP POST or HTTP DELETE requests for example.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

Spring Security provides by default a protection against CSRF attacks which can be disabled:

```
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http.csrf().disable(); // Sensitive: csrf protection is entirely disabled
   // or
    http.csrf().ignoringAntMatchers("/route/"); // Sensitive: csrf protection is disabled for specific routes
  }
}
```

Default:

A cross-site request forgery (CSRF) attack occurs when a trusted user of a web application can be forced, by an attacker, to perform sensitive actions that he didn't intend, such as updating his profile or sending a message, more generally anything that can change the state of the application.

The attacker can trick the user/victim to click on a link, corresponding to the privileged action, or to visit a malicious web site that embeds a hidden web request and as web browsers automatically include cookies, the actions can be authenticated and sensitive.

Ask Yourself Whether

- The web application uses cookies to authenticate users.
- There exist sensitive operations in the web application that can be performed when the user is authenticated.
- The state / resources of the web application can be modified by doing HTTP POST or HTTP DELETE requests for example.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- Protection against CSRF attacks is strongly recommended:
  - to be activated by default for all unsafe HTTP methods.
  - implemented, for example, with an unguessable CSRF token
- Of course all sensitive operations should not be performed with safe HTTP methods like `GET` which are designed to be used only for information retrieval.

Sensitive Code Example

Spring Security provides by default a protection against CSRF attacks which can be disabled:

```
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http.csrf().disable(); // Sensitive: csrf protection is entirely disabled
   // or
    http.csrf().ignoringAntMatchers("/route/"); // Sensitive: csrf protection is disabled for specific routes
  }
}
```

Compliant Solution

Spring Security CSRF protection is enabled by default, do not disable it:

```
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

  @Override
  protected void configure(HttpSecurity http) throws Exception {
    // http.csrf().disable(); // Compliant
  }
}
```

See

- OWASP - Top 10 2021 Category A1 - Broken Access Control
- CWE - CWE-352 - Cross-Site Request Forgery (CSRF)
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- OWASP - Cross-Site Request Forgery
- STIG Viewer - Application Security and Development: V-222603 - The application must protect from Cross-Site Request Forgery (CSRF) vulnerabilities.
- PortSwigger - Web storage: the lesser evil for session tokens

---

AssertJ configuration should be applied (java:S5831)

Severidad: MAJOR

Root_cause:

A `org.assertj.core.configuration.Configuration` will be effective only once you call `Configuration.apply()` or `Configuration.applyAndDisplay()`.

This rule raises an issue when configurations are set without the appropriate call to apply them.

Noncompliant code example

```
Configuration configuration = new Configuration(); // Noncompliant, this configuration will not be applied.
configuration.setComparingPrivateFields(true);
```

Compliant solution

```
Configuration configuration = new Configuration();
configuration.setComparingPrivateFields(true);
configuration.applyAndDisplay();
// Alternatively: configuration.apply();
```

Resources:

- AssertJ configuration documentation

AssertJ methods setting the assertion context should come before an assertion (java:S5833)

Severidad: MAJOR

Root_cause:

Describing, setting error message or adding a comparator in AssertJ must be done before calling the assertion, otherwise, settings will not be taken into account.

This rule raises an issue when one of the method (with all similar methods):

- `as`
- `describedAs`
- `withFailMessage`
- `overridingErrorMessage`
- `usingComparator`
- `usingElementComparator`
- `extracting`
- `filteredOn`

is called without calling an AssertJ assertion afterward.

Noncompliant code example

```
assertThat(actual).isEqualTo(expected).as("Description"); // Noncompliant
assertThat(actual).isEqualTo(expected).withFailMessage("Fail message"); // Noncompliant
assertThat(actual).isEqualTo(expected).usingComparator(new CustomComparator()); // Noncompliant
```

Compliant solution

```
assertThat(actual).as("Description").isEqualTo(expected);
assertThat(actual).withFailMessage("Fail message").isEqualTo(expected);
assertThat(actual).usingComparator(new CustomComparator()).isEqualTo(expected);
```

Resources:

- AssertJ incorrect usage documentation

"@Value" annotation should inject property or SpEL expression (java:S6804)

Severidad: MAJOR

Root_cause:

The purpose of the `@Value` annotation in `org.springframework.beans.factory.annotation` is to inject a value into a field or method based on the Spring context after it has been established.

If the annotation does not include an expression (either Spring Expression Language or a property injection), the injected value is a simple constant that does not depend on the Spring context, making the annotation replaceable with a standard field initialization statement.

This not only implies the redundant use of `@Value`, but could also indicate an error where the expression indicators (`#`, `$`) were omitted by mistake.

Exceptions

This rule does not raise an issue if `@Value` is applied to a method or method argument, because the annotation has the side effect that the method is called.

How_to_fix:

- If a property is to be injected, use `${propertyName}` instead of `propertyName`.
- If a SpEL expression is to be evaluated, use `#{expression}` instead of `expression`.
- If you intend to initialize a field with a simple value or with an expression that does not depend on the Spring context, use a standard field initialization statement.

Noncompliant code example

```
@Value("catalog.name") // Noncompliant, this will not inject the property
String catalog;
```

Compliant solution

```
@Value("${catalog.name}") // Compliant
String catalog;
```

Noncompliant code example

```
@Value("book.topics[0]") // Noncompliant, this will not evaluate the expression
Topic topic;
```

Compliant solution

```
@Value("#{book.topics[0]}") // Compliant
Topic topic;
```

Noncompliant code example

```
@Value("Hello, world!") // Noncompliant, this use of @Value is redundant
String greeting;
```

Compliant solution

```
String greeting = "Hello, world!"; // Compliant
```

Resources:

Documentation

- Spring Framework API - Annotation Interface Value

Articles & blog posts

- Baeldung - A Quick Guide to Spring @Value
- DigitalOcean - Spring @Value Annotation

Introduction:

This rule reports when the Spring `@Value` annotation injects a simple value that does not contain an expression.

Methods with Spring proxy should not be called via "this" (java:S6809)

Severidad: MAJOR

How_to_fix:

Replace calls to `@Async`, `@Cacheable` or `@Transactional` methods via `this` with calls on an instance that was injected by Spring (`@Autowired`, `@Resource` or `@Inject`). The injected instance is a proxy on which the methods can be invoked safely.

Noncompliant code example

```
@Service
public class AsyncNotificationProcessor implements NotificationProcessor {

  @Override
  public void process(Notification notification) {
    processAsync(notification); // Noncompliant, call bypasses proxy
    retrieveNotification(notification.id); // Noncompliant, call bypasses proxy and will not be cached
  }

  @Async
  public processAsync(Notification notification) {
    // ...
  }

  @Cacheable
  public Notification retrieveNotification(Long id) {
    // ...
  }

}
```

Compliant solution

```
@Service
public class AsyncNotificationProcessor implements NotificationProcessor {

  @Resource
  private AsyncNotificationProcessor

  @Override
  public void process(Notification notification) {
    asyncNotificationProcessor.processAsync(notification); // Compliant, call via injected proxy
    asyncNotificationProcessor.retrieveNotification(notification.id); // Compliant, the call will be cached
  }

  @Async
  public processAsync(Notification notification) {
    // ...
  }

  @Cacheable
  public Notification retrieveNotification(Long id) {
    // ...
  }
}
```

Root_cause:

A method annotated with Spring’s `@Async`, `@Cacheable` or `@Transactional` annotations will not work as expected if invoked directly from within its class.

This is because Spring generates a proxy class with wrapper code to manage the method’s asynchronicity (`@Async`), to cache methods invocations (`@Cacheable`), or to handle the transaction (`@Transactional`). However, when called using `this`, the proxy instance is bypassed, and the method is invoked directly without the required wrapper code.

Resources:

Documentation

- Spring Framework API - Annotation Interface Async
- Spring Framework API - Annotation Interface Transactional
- Spring Framework API - Annotation Interface Cacheable

Articles & blog posts

- Baeldung - How To Do @Async in Spring
- Stack Overflow - Spring @Async ignored
- Stack Overflow - Does Spring @Transactional attribute work on a private method?
- Spring docs, The @Cacheable Annotation

Motion Sensor should not use gyroscope (java:S6923)

Severidad: MAJOR

Introduction:

The `android.hardware.SensorManager#getDefaultSensor` offers two types of Motion Sensors:

- `TYPE_ROTATION_VECTOR`: a combination of the gyroscope, accelerometer, and magnetometer.
- `TYPE_GEOMAGNETIC_ROTATION_VECTOR`: a combination of the accelerometer and magnetometer.

How_to_fix:

Replace `TYPE_ROTATION_VECTOR` with `TYPE_GEOMAGNETIC_ROTATION_VECTOR` when retrieving the Motion Sensor.

Noncompliant code example

```
public class BackGroundActivity extends Activity {

    private Sensor motionSensor;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        SensorManager sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
        motionSensor = sensorManager.getDefaultSensor(Sensor.TYPE_ROTATION_VECTOR); // Noncompliant
        // ..
    }
    //..
}
```

Compliant solution

```
public class BackGroundActivity extends Activity {

    private Sensor motionSensor;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        SensorManager sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
        motionSensor = sensorManager.getDefaultSensor(Sensor.TYPE_GEOMAGNETIC_ROTATION_VECTOR); // Compliant
        // ..
    }
    //..
}
```

Resources:

Documentation

- Android - Low Power Sensors
- Android - Motion Sensors

Root_cause:

The battery life is a major concern for mobile devices and choosing the right Sensor is very important to reduce the power usage and extend the battery life.

It is recommended, for reducing the power usage, to use `TYPE_GEOMAGNETIC_ROTATION_VECTOR` for *background tasks*, *long-running tasks* and other tasks not requiring accurate motion detection.

The rule reports an issue when `android.hardware.SensorManager#getDefaultSensor` uses `TYPE_ROTATION_VECTOR` instead of `TYPE_GEOMAGNETIC_ROTATION_VECTOR`.

Empty statements should be removed (java:S1116)

Severidad: MINOR

Resources:

Documentation

- 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

Root_cause:

Empty statements represented by a semicolon `;` are statements that do not perform any operation. They are often the result of a typo or a misunderstanding of the language syntax. It is a good practice to remove empty statements since they don't add value and lead to confusion and errors.

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 < 3; i++) ; // Compliant if unique statement of a loop
  // ...
}
```

Boolean literals should not be redundant (java:S1125)

Severidad: MINOR

Root_cause:

A boolean literal can be represented in two different ways: `true` or `false`. They can be combined with logical operators (`!, &&, ||, ==, !=`) to produce logical expressions that represent truth values. However, comparing a boolean literal to a variable or expression that evaluates to a boolean value is unnecessary and can make the code harder to read and understand. The more complex a boolean expression is, the harder it will be for developers to understand its meaning and expected behavior, and it will favour the introduction of new bugs.

How_to_fix:

Remove redundant boolean literals from expressions to improve readability and make the code more maintainable.

Noncompliant code example

```
if (booleanMethod() == true) { /* ... */ }
if (booleanMethod() == false) { /* ... */ }
if (booleanMethod() || false) { /* ... */ }
doSomething(!false);
doSomething(booleanMethod() == true);

booleanVariable = booleanMethod() ? true : false;
booleanVariable = booleanMethod() ? true : exp;
booleanVariable = booleanMethod() ? false : exp;
booleanVariable = booleanMethod() ? exp : true;
booleanVariable = booleanMethod() ? exp : false;
```

Compliant solution

```
if (booleanMethod()) { /* ... */ }
if (!booleanMethod()) { /* ... */ }
if (booleanMethod()) { /* ... */ }
```

```
doSomething(true);
doSomething(booleanMethod());

booleanVariable = booleanMethod();
booleanVariable = booleanMethod() || exp;
booleanVariable = !booleanMethod() && exp;
booleanVariable = !booleanMethod() || exp;
booleanVariable = booleanMethod() && exp;
```

Return of boolean expressions should not be wrapped into an "if-then-else" statement (java:S1126)

Severidad: MINOR

Root_cause:

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;
}
```

"switch" statements should not have too many "case" clauses (java:S1479)

Severidad: MAJOR

Root_cause:

When `switch` statements have large sets of `case` clauses, it is usually an attempt to map two sets of data. A real map structure would be more readable and maintainable, and should be used instead.

Exceptions

This rule ignores `switch`es over `Enum`s and empty, fall-through cases.

Local variables should not be declared and then immediately returned or thrown (java:S1488)

Severidad: MINOR

Root_cause:

Declaring a variable only to immediately return or throw it is considered a bad practice because it adds unnecessary complexity to the code. This practice can make the code harder to read and understand, as it introduces an extra step that doesn’t add any value. Instead of declaring a variable and then immediately returning or throwing it, it is generally better to return or throw the value directly. This makes the code cleaner, simpler, and easier to understand.

How_to_fix:

Declaring a variable only to immediately return or throw it is considered a bad practice because it adds unnecessary complexity to the code. To fix the issue, return or throw the value directly.

Noncompliant code example

```
public long computeDurationInMilliseconds() {
  long duration = (((hours * 60) + minutes) * 60 + seconds) * 1000;
  return duration;
}
```

Compliant solution

```
public long computeDurationInMilliseconds() {
  return (((hours * 60) + minutes) * 60 + seconds) * 1000;
}
```

Noncompliant code example

```
public void doSomething() {
  RuntimeException myException = new RuntimeException();
  throw myException;
}
```

Compliant solution

```
public void doSomething() {
  throw new RuntimeException();
}
```

"null" should not be returned from a "Boolean" method (java:S2447)

Severidad: CRITICAL

Resources:

- CWE - CWE-476 - NULL Pointer Dereference
- CERT, EXP01-J. - Do not use a null in a case where an object is required

Root_cause:

Callers of a `Boolean` method may be expecting to receive `true` or `false` in response. But `Boolean` objects can take `null` as a possible value. `Boolean` methods should not return `null` unless the code is annotated appropriately. With the proper annotation, the caller is aware that the returned value could be null.

Noncompliant code example

```
public Boolean isUsable() {
  // ...
  return null;  // Noncompliant
}

public void caller() {
  if (isUsable()) { // A NullPointerException might occur here
    // ...
  }
}
```

Compliant solution

```
@javax.annotation.Nullable
public Boolean isUsable() {
  // ...
  return null;
}

@javax.annotation.CheckForNull
public Boolean isUsable() {
  // ...
  return null;
}

public void caller() {
  if (Boolean.True.equals(isUsable())) { // This caller knows to check and avoid ambiguity
    // ...
  }
}
```

Instance methods should not write to "static" fields (java:S2696)

Severidad: CRITICAL

Resources:

Standards

- STIG Viewer - [Application Security and Development: V-222567](#) - The application must not be vulnerable to race conditions.

Root_cause:

Correctly updating a `static` field from a non-static method is tricky to get right and could easily lead to bugs if there are multiple class instances and/or multiple threads in play. Ideally, `static` fields are only updated from `synchronized static` methods.

This rule raises an issue each time a `static` field is updated from a non-static method.

Noncompliant code example

```
public class MyClass {

  private static int count = 0;

  public void doSomething() {
    //...
    count++;  // Noncompliant
  }
}
```

Test assertions should include messages (java:S2698)

Severidad: MINOR

Root_cause:

Adding messages to JUnit, FEST and AssertJ assertions is an investment in your future productivity. Spend a few seconds writing them now, and you'll save a lot of time on the other end when either the tests fail and you need to quickly diagnose the problem, or when you need to maintain the tests and the assertion messages work as a sort of documentation.

Noncompliant code example

```
assertEquals(4, list.size());  // Noncompliant

try {
  fail();  // Noncompliant
} catch (Exception e) {
  assertThat(list.get(0)).isEqualTo("pear");  // Noncompliant
}
```

Compliant solution

```
assertEquals("There should have been 4 Fruits in the list", 4, list.size());

try {
  fail("And exception is expected here");
} catch (Exception e) {
  assertThat(list.get(0)).as("check first element").overridingErrorMessage("The first element should be a pear, not a %s", list.get(0)).isE
}
```

Tests should include assertions (java:S2699)

Severidad: BLOCKER

Root_cause:

A test case without assertions ensures only that no exceptions are thrown. Beyond basic runnability, it ensures nothing about the behavior of the code under test.

This rule raises an exception when no assertions from any of the following known frameworks are found in a test:

- AssertJ
- Awaitility
- EasyMock
- Eclipse Vert.x

- Fest 1.x and 2.x
- Hamcrest
- JMock
- JMockit
- JUnit
- Mockito
- Rest-assured 2.x, 3.x and 4.x
- RxJava 1.x and 2.x
- Selenide
- Spring's `org.springframework.test.web.servlet.ResultActions.andExpect()` and `org.springframework.test.web.servlet.ResultActions.andExpectAll()`
- Truth Framework
- WireMock

Furthermore, as new or custom assertion frameworks may be used, the rule can be parametrized to define specific methods that will also be considered as assertions. No issue will be raised when such methods are found in test cases. The parameter value should have the following format `<FullyQualifiedClassName>#<MethodName>`, where `MethodName` can end with the wildcard character. For constructors, the pattern should be `<FullyQualifiedClassName>#<init>`.

Example: `com.company.CompareToTester#compare*,com.company.CustomAssert#customAssertMethod,com.company.CheckVerifier#<init>`.

Noncompliant code example

```
@Test
public void testDoSomething() {  // Noncompliant
  MyClass myClass = new MyClass();
  myClass.doSomething();
}
```

Compliant solution

Example when `com.company.CompareToTester#compare*` is used as parameter to the rule.

```
import com.company.CompareToTester;

@Test
public void testDoSomething() {
  MyClass myClass = new MyClass();
  assertNull(myClass.doSomething());  // JUnit assertion
  assertThat(myClass.doSomething()).isNull();  // Fest assertion
}

@Test
public void testDoSomethingElse() {
  MyClass myClass = new MyClass();
  new CompareToTester().compareWith(myClass);  // Compliant - custom assertion method defined as rule parameter
  CompareToTester.compareStatic(myClass);  // Compliant
}
```

Assertion arguments should be passed in the correct order (java:S3415)

Severidad: MAJOR

Root_cause:

The standard assertions library methods such as `org.junit.Assert.assertEquals`, and `org.junit.Assert.assertSame` expect the first argument to be the expected value and the second argument to be the actual value. For AssertJ instead, the argument of `org.assertj.core.api.Assertions.assertThat` is the actual value, and the subsequent calls contain the expected values.

What is the potential impact?

Having the expected value and the actual value in the wrong order will not alter the outcome of tests, (succeed/fail when it should) but the error messages will contain misleading information.

This rule raises an issue when the actual argument to an assertions library method is a hard-coded value and the expected argument is not.

How_to_fix:

You should provide the assertion methods with a hard-coded value as the expected value, while the actual value of the assertion should derive from the portion of code that you want to test.

Supported frameworks:

- JUnit4
- JUnit5
- AssertJ

Noncompliant code example

```
org.junit.Assert.assertEquals(runner.exitCode(), 0, "Unexpected exit code");  // Noncompliant; Yields error message like: Expected:<-1>. Ac
org.assertj.core.api.Assertions.assertThat(0).isEqualTo(runner.exitCode()); // Noncompliant
```

Compliant solution

```
org.junit.Assert.assertEquals(0, runner.exitCode(), "Unexpected exit code");
org.assertj.core.api.Assertions.assertThat(runner.exitCode()).isEqualTo(0);
```

Loggers should be named for their enclosing classes (java:S3416)

Severidad: MINOR

Root_cause:

It is convention to name each class's logger for the class itself. Doing so allows you to set up clear, communicative logger configuration. Naming loggers by some other convention confuses configuration, and using the same class name for multiple class loggers prevents the granular configuration of each class' logger. Some libraries, such as SLF4J warn about this, but not all do.

This rule raises an issue when a logger is not named for its enclosing class.

Noncompliant code example

```
public class MyClass {
  private final static Logger LOG = LoggerFactory.getLogger(WrongClass.class);  // Noncompliant; multiple classes using same logger
}
```

Compliant solution

```
public class MyClass {
  private final static Logger LOG = LoggerFactory.getLogger(MyClass.class);
}
```

Custom resources should be closed (java:S3546)

Severidad: BLOCKER

Root_cause:

Leaking resources in an application is never a good idea, as it can lead to memory issues, and even the crash of the application. This rule template allows you to specify which constructions open a resource and how it is closed in order to raise issue within a method scope when custom resources are leaked.

Resources:

Related rules

- S2095 - Resources should be closed

Unit tests should throw exceptions (java:S3658)

Severidad: MINOR

Root_cause:

When the code under test in a unit test throws an exception, the test itself fails. Therefore, there is no need to surround the tested code with a `try-catch` structure to detect failure. Instead, you can simply move the exception type to the method signature.

This rule raises an issue when there is a fail assertion inside a `catch` block.

Supported frameworks:

- JUnit3
- JUnit4

- JUnit5
- Fest assert
- AssertJ

Noncompliant code example

```
@Test
public void testMethod() {
  try {
            // Some code
  } catch (MyException e) {
    Assert.fail(e.getMessage());  // Noncompliant
  }
}
```

Compliant solution

```
@Test
public void testMethod() throws MyException {
    // Some code
}
```

Delivering code in production with debug features activated is security-sensitive (java:S4507)

Severidad: MINOR

Default:

Development tools and frameworks usually have options to make debugging easier for developers. Although these features are useful during development, they should never be enabled for applications deployed in production. Debug instructions or error messages can leak detailed information about the system, like the application’s path or file names.

Ask Yourself Whether

- The code or configuration enabling the application debug features is deployed on production servers or distributed to end users.
- The application runs by default with debug features activated.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

Do not enable debugging features on production servers or applications distributed to end users.

Sensitive Code Example

`Throwable.printStackTrace(...)` prints a Throwable and its stack trace to `System.Err` (by default) which is not easily parseable and can expose sensitive information:

```
try {
  /* ... */
} catch(Exception e) {
  e.printStackTrace(); // Sensitive
}
```

EnableWebSecurity annotation for SpringFramework with `debug` to `true` enables debugging support:

```
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

@Configuration
@EnableWebSecurity(debug = true) // Sensitive
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
  // ...
}
```

WebView.setWebContentsDebuggingEnabled(true) for Android enables debugging support:

```
import android.webkit.WebView;

WebView.setWebContentsDebuggingEnabled(true); // Sensitive
WebView.getFactory().getStatics().setWebContentsDebuggingEnabled(true); // Sensitive
```

Compliant Solution

Loggers should be used (instead of `printStackTrace`) to print throwables:

```
try {
  /* ... */
} catch(Exception e) {
  LOGGER.log("context", e);
}
```

EnableWebSecurity annotation for SpringFramework with `debug` to `false` disables debugging support:

```
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

@Configuration
@EnableWebSecurity(debug = false)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
  // ...
}
```

WebView.setWebContentsDebuggingEnabled(false) for Android disables debugging support:

```
import android.webkit.WebView;

WebView.setWebContentsDebuggingEnabled(false);
WebView.getFactory().getStatics().setWebContentsDebuggingEnabled(false);
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- CWE - CWE-489 - Active Debug Code
- CWE - CWE-215 - Information Exposure Through Debug Information

Assess_the_problem:

Ask Yourself Whether

- The code or configuration enabling the application debug features is deployed on production servers or distributed to end users.
- The application runs by default with debug features activated.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

`Throwable.printStackTrace(...)` prints a Throwable and its stack trace to `System.Err` (by default) which is not easily parseable and can expose sensitive information:

```
try {
  /* ... */
} catch(Exception e) {
  e.printStackTrace(); // Sensitive
}
```

EnableWebSecurity annotation for SpringFramework with `debug` to `true` enables debugging support:

```
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

@Configuration
@EnableWebSecurity(debug = true) // Sensitive
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
  // ...
}
```

WebView.setWebContentsDebuggingEnabled(true) for Android enables debugging support:

```
import android.webkit.WebView;

WebView.setWebContentsDebuggingEnabled(true); // Sensitive
WebView.getFactory().getStatics().setWebContentsDebuggingEnabled(true); // Sensitive
```

Root_cause:

Development tools and frameworks usually have options to make debugging easier for developers. Although these features are useful during development, they should never be enabled for applications deployed in production. Debug instructions or error messages can leak detailed information about the system, like the application’s path or file names.

How_to_fix:

Recommended Secure Coding Practices

Do not enable debugging features on production servers or applications distributed to end users.

Compliant Solution

Loggers should be used (instead of `printStackTrace`) to print throwables:

```
try {
  /* ... */
} catch(Exception e) {
  LOGGER.log("context", e);
}
```

EnableWebSecurity annotation for SpringFramework with `debug` to `false` disables debugging support:

```
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

@Configuration
@EnableWebSecurity(debug = false)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
  // ...
}
```

WebView.setWebContentsDebuggingEnabled(false) for Android disables debugging support:

```
import android.webkit.WebView;

WebView.setWebContentsDebuggingEnabled(false);
WebView.getFactory().getStatics().setWebContentsDebuggingEnabled(false);
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- CWE - CWE-489 - Active Debug Code
- CWE - CWE-215 - Information Exposure Through Debug Information

---

Setting JavaBean properties is security-sensitive (java:S4512)

Severidad: CRITICAL

Root_cause:

Setting JavaBean properties is security sensitive. Doing it with untrusted values has led in the past to the following vulnerability:

- CVE-2014-0114

JavaBeans can have their properties or nested properties set by population functions. An attacker can leverage this feature to push into the JavaBean malicious data that can compromise the software integrity. A typical attack will try to manipulate the ClassLoader and finally execute malicious code.

This rule raises an issue when:

- BeanUtils.populate(…) or BeanUtilsBean.populate(…) from Apache Commons BeanUtils are called
- BeanUtils.setProperty(…) or BeanUtilsBean.setProperty(…) from Apache Commons BeanUtils are called
- org.springframework.beans.BeanWrapper.setPropertyValue(…) or org.springframework.beans.BeanWrapper.setPropertyValues(…) from Spring is called

Assess_the_problem:

Ask Yourself Whether

- the new property values might have been tampered with or provided by an untrusted source.
- sensitive properties can be modified, for example: `class.classLoader`

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

```
Company bean = new Company();
HashMap map = new HashMap();
```

```
Enumeration names = request.getParameterNames();
while (names.hasMoreElements()) {
    String name = (String) names.nextElement();
    map.put(name, request.getParameterValues(name));
}
BeanUtils.populate(bean, map); // Sensitive: "map" is populated with data coming from user input, here "request.getParameterNames()"
```

Default:

Setting JavaBean properties is security sensitive. Doing it with untrusted values has led in the past to the following vulnerability:

- CVE-2014-0114

JavaBeans can have their properties or nested properties set by population functions. An attacker can leverage this feature to push into the JavaBean malicious data that can compromise the software integrity. A typical attack will try to manipulate the ClassLoader and finally execute malicious code.

This rule raises an issue when:

- BeanUtils.populate(…) or BeanUtilsBean.populate(…) from Apache Commons BeanUtils are called
- BeanUtils.setProperty(…) or BeanUtilsBean.setProperty(…) from Apache Commons BeanUtils are called
- org.springframework.beans.BeanWrapper.setPropertyValue(…) or org.springframework.beans.BeanWrapper.setPropertyValues(…) from Spring is called

Ask Yourself Whether

- the new property values might have been tampered with or provided by an untrusted source.
- sensitive properties can be modified, for example: `class.classLoader`

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

Sanitize all values used as JavaBean properties.

Don't set any sensitive properties. Keep full control over which properties are set. If the property names are provided by an unstrusted source, filter them with a whitelist.

Sensitive Code Example

```
Company bean = new Company();
HashMap map = new HashMap();
Enumeration names = request.getParameterNames();
while (names.hasMoreElements()) {
    String name = (String) names.nextElement();
    map.put(name, request.getParameterValues(name));
}
BeanUtils.populate(bean, map); // Sensitive: "map" is populated with data coming from user input, here "request.getParameterNames()"
```

See

- OWASP - Top 10 2021 Category A3 - Injection
- OWASP - Top 10 2021 Category A8 - Software and Data Integrity Failures
- OWASP - Top 10 2017 Category A1 - Injection
- CWE - CWE-915 - Improperly Controlled Modification of Dynamically-Determined Object Attributes
- CERT, MSC61-J. - Do not use insecure or weak cryptographic algorithms
- Derived from FindSecBugs rule BEAN_PROPERTY_INJECTION

How_to_fix:

Recommended Secure Coding Practices

Sanitize all values used as JavaBean properties.

Don't set any sensitive properties. Keep full control over which properties are set. If the property names are provided by an unstrusted source, filter them with a whitelist.

See

- OWASP - Top 10 2021 Category A3 - Injection
- OWASP - Top 10 2021 Category A8 - Software and Data Integrity Failures
- OWASP - Top 10 2017 Category A1 - Injection

- CWE - CWE-915 - Improperly Controlled Modification of Dynamically-Determined Object Attributes
- CERT, MSC61-J. - Do not use insecure or weak cryptographic algorithms
- Derived from FindSecBugs rule BEAN_PROPERTY_INJECTION

String offset-based methods should be preferred for finding substrings from offsets (java:S4635)

Severidad: CRITICAL

Root_cause:

Looking for a given substring starting from a specified offset can be achieved by such code: `str.substring(beginIndex).indexOf(char1)`. This works well, but it creates a new `String` for each call to the `substring` method. When this is done in a loop, a lot of `Strings` are created for nothing, which can lead to performance problems if `str` is large.

To avoid performance problems, `String.substring(beginIndex)` should not be chained with the following methods:

- `indexOf(int ch)`
- `indexOf(String str)`
- `lastIndexOf(int ch)`
- `lastIndexOf(String str)`
- `startsWith(String prefix)`

For each of these methods, another method with an additional parameter is available to specify an offset.

Using these methods will avoid the creation of additional `String` instances. For indexOf methods, adjust the returned value by subtracting the substring index parameter to obtain the same result.

Noncompliant code example

```
str.substring(beginIndex).indexOf(char1); // Noncompliant; a new String is going to be created by "substring"
```

Compliant solution

```
str.indexOf(char1, beginIndex) - beginIndex; // index for char1 not found is (-1-beginIndex)
```

Chained AssertJ assertions should be simplified to the corresponding dedicated assertion (java:S5838)

Severidad: MINOR

Root_cause:

AssertJ contains many assertions methods specific to common types. Both versions will test the same things, but the dedicated one will provide a better error message, simplifying the debugging process.

This rule reports an issue when an assertion can be simplified to a dedicated one.

The array below gives a non-exhaustive list of assertion reported by the rule. Code behaving similarly, or with a negation will also be reported.

Original	Dedicated
Related to Object	
`assertThat(getObject()).isEqualTo(null)`	`assertThat(getObject()).isNull()`
`assertThat(getBoolean()).isEqualTo(true)`	`assertThat(getBoolean()).isTrue()`
`assertThat(getBoolean()).isEqualTo(false)`	`assertThat(getBoolean()).isFalse()`
`assertThat(x.equals(y)).isTrue()`	`assertThat(x).isEqualTo(y)`
`assertThat(x == y).isTrue()`	`assertThat(x).isSameAs(y)`
`assertThat(x == null).isTrue()`	`assertThat(x).isNull()`
`assertThat(x.toString()).isEqualTo(y)`	`assertThat(x).hasToString(y)`
`assertThat(x.hashCode()).isEqualTo(y.hashCode())`	`assertThat(x).hasSameHashCodeAs(y)`
`assertThat(getObject() instanceof MyClass).isTrue()`	`assertThat(getObject()).isInstanceOf(MyClass.class)`
Related to `Comparable`	
`assertThat(x.compareTo(y)).isZero()`	`assertThat(x).isEqualByComparingTo(y)`
`assertThat(x >= y).isTrue()`	`assertThat(x).isGreaterThanOrEqualTo(y)`
`assertThat(x > y).isTrue()`	`assertThat(x).isGreaterThan(y)`

Original	Dedicated
`assertThat(x <= y).isTrue()`	`assertThat(x).isLessThanOrEqualTo(y)`
`assertThat(x < y).isTrue()`	`assertThat(x).isLessThan(y)`
Related to `String`	
`assertThat(getString().isEmpty()).isTrue()`	`assertThat(getString()).isEmpty()`
`assertThat(getString()).hasSize(0)`	`assertThat(getString()).isEmpty()`
`assertThat(getString().equals(expected)).isTrue()`	`assertThat(getString()).isEqualTo(expected)`
`assertThat(getString().equalsIgnoreCase(expected)).isTrue()`	`assertThat(getString()).isEqualToIgnoringCase(expected)`
`assertThat(getString().contains(expected)).isTrue()`	`assertThat(getString()).contains(expected)`
`assertThat(getString().startsWith(expected)).isTrue()`	`assertThat(getString()).startsWith(expected)`
`assertThat(getString().endsWith(expected)).isTrue()`	`assertThat(getString()).endsWith(expected)`
`assertThat(getString().matches(expected)).isTrue()`	`assertThat(getString()).matches(expected)`
`assertThat(getString().trim()).isEmpty()`	`assertThat(getString()).isBlank()`
`assertThat(getString().length()).isEqualTo(length)`	`assertThat(getString()).hasSize(length)`
`assertThat(getString().length()).hasSize(expected.length())`	`assertThat(getString()).hasSameSizeAs(expected)`
Related to `File`	
`assertThat(getFile()).hasSize(0)`	`assertThat(getFile()).isEmpty()`
`assertThat(getFile().length()).isZero()`	`assertThat(getFile()).isEmpty()`
`assertThat(getFile().length()).isEqualTo(length)`	`assertThat(getFile()).hasSize(length)`
`assertThat(getFile().canRead()).isTrue()`	`assertThat(getFile()).canRead()`
`assertThat(getFile().canWrite()).isTrue()`	`assertThat(getFile()).canWrite()`
`assertThat(getFile().exists()).isTrue()`	`assertThat(getFile()).exists()`
`assertThat(getFile().getName()).isEqualTo(name)`	`assertThat(getFile()).hasName(name)`
`assertThat(getFile().getParent()).isEqualTo(pathname)`	`assertThat(getFile()).hasParent(pathname)`
`assertThat(getFile().getParentFile()).isNull()`	`assertThat(getFile()).hasNoParent()`
`assertThat(getFile().isAbsolute()).isTrue()`	`assertThat(getFile()).isAbsolute()`
`assertThat(getFile().isAbsolute()).isFalse()`	`assertThat(getFile()).isRelative()`
`assertThat(getFile().isDirectory()).isTrue()`	`assertThat(getFile()).isDirectory()`
`assertThat(getFile().isFile()).isTrue()`	`assertThat(getFile()).isFile()`
`assertThat(getFile().list()).isEmpty()`	`assertThat(getFile()).isEmptyDirectory()`
Related to `Path`	
`assertThat(getPath().startsWith(path)).isTrue()`	`assertThat(getPath()).startsWithRaw(path)`
`assertThat(getPath().endsWith(path)).isTrue()`	`assertThat(getPath()).endsWithRaw(path)`
`assertThat(getPath().getParent()).isEqualTo(name)`	`assertThat(getPath()).hasParentRaw(name)`
`assertThat(getPath().getParent()).isNull()`	`assertThat(getPath()).hasNoParentRaw()`
`assertThat(getPath().isAbsolute()).isTrue()`	`assertThat(getPath()).isAbsolute()`
`assertThat(getPath().isAbsolute()).isFalse()`	`assertThat(getPath()).isRelative()`
Related to `Array`	
`assertThat(getArray().length).isZero()`	`assertThat(getArray()).isEmpty()`
`assertThat(getArray().length).isEqualTo(length)`	`assertThat(getArray()).hasSize(length)`
`assertThat(getArray().length).isEqualTo(anotherArray.length)`	`assertThat(getArray()).hasSameSizeAs(anotherArray)`
`assertThat(getArray().length).isLessThanOrEqualTo(expression)`	`assertThat(getArray()).hasSizeLessThanOrEqualTo(expression)`
`assertThat(getArray().length).isLessThan(expression)`	`assertThat(getArray()).hasSizeLessThan(expression)`
`assertThat(getArray().length).isGreaterThan(expression)`	`assertThat(getArray()).hasSizeGreaterThan(expression)`
`assertThat(getArray().length).isGreaterThanOrEqualTo(expression)`	`assertThat(getArray()).hasSizeGreaterThanOrEqualTo(expression)`
Related to `Collection`	
`assertThat(getCollection().isEmpty()).isTrue()`	`assertThat(getCollection()).isEmpty()`
`assertThat(getCollection().size()).isZero()`	`assertThat(getCollection()).isEmpty()`
`assertThat(getCollection().contains(something)).isTrue()`	`assertThat(getCollection()).contains(something)`
`assertThat(getCollection().containsAll(otherCollection)).isTrue()`	`assertThat(getCollection()).containsAll(otherCollection)`
Related to `Map`	

Original	Dedicated
`assertThat(getMap().size()).isEqualTo(otherMap().size()`	`assertThat(getMap()).hasSameSizeAs(otherMap())`
`assertThat(getMap().containsKey(key)).isTrue()`	`assertThat(getMap()).containsKey(key)`
`assertThat(getMap().keySet()).contains(key)`	`assertThat(getMap()).containsKey(key)`
`assertThat(getMap().keySet()).containsOnlyKey(key)`	`assertThat(getMap()).containsOnlyKey(key)`
`assertThat(getMap().containsValue(value)).isTrue()`	`assertThat(getMap()).containsValue(value)`
`assertThat(getMap().values()).contains(value)`	`assertThat(getMap()).containsValue(value)`
`assertThat(getMap().get(key)).isEqualTo(value)`	`assertThat(getMap()).containsEntry(key, value)`
Related to `Optional`	
`assertThat(getOptional().isPresent()).isTrue()`	`assertThat(getOptional()).isPresent()`
`assertThat(getOptional().get()).isEqualTo(something)`	`assertThat(getOptional()).contains(something)`
`assertThat(getOptional().get()).isSameAs(something)`	`assertThat(getOptional()).containsSame(something)`

Noncompliant code example

```
assertThat(getObject()).isEqualTo(null); // Noncompliant
assertThat(getObject()).isNotEqualTo(null); // Noncompliant - not listed above but also supported

assertThat(getString().trim()).isEmpty();
assertThat(getFile().canRead()).isTrue();
assertThat(getPath().getParent()).isNull();
```

Compliant solution

```
assertThat(getObject()).isNull();
assertThat(getObject()).isNotNull();

assertThat(getString()).isBlank();
assertThat(getFile()).canRead();
assertThat(getPath()).hasNoParentRaw();
```

Repeated patterns in regular expressions should not match the empty string (java:S5842)

Severidad: MINOR

Root_cause:

A regex should never include a repetitive pattern whose body would match the empty string. This is usually a sign that a part of the regex is redundant or does not do what the author intended.

Noncompliant code example

```
"(?:)*"      // same as the empty regex, the '*' accomplishes nothing
"(?:|x)*"    // same as the empty regex, the alternative has no effect
"(?:x|)*"    // same as 'x*', the empty alternative has no effect
"(?:x*|y*)*" // same as 'x*', the first alternative would always match, y* is never tried
"(?:x?)*"    // same as 'x*'
"(?:x?)+"    // same as 'x*'
```

Compliant solution

```
"x*"
```

Regular expressions should not be too complicated (java:S5843)

Severidad: MAJOR

Root_cause:

Overly complicated regular expressions are hard to read and to maintain and can easily cause hard-to-find bugs. If a regex is too complicated, you should consider replacing it or parts of it with regular code or splitting it apart into multiple patterns at least.

The complexity of a regular expression is determined as follows:

Each of the following operators increases the complexity by an amount equal to the current nesting level and also increases the current nesting level by one for its arguments:

- `|` - when multiple `|` operators are used together, the subsequent ones only increase the complexity by 1
- `&&` (inside character classes) - when multiple `&&` operators are used together, the subsequent ones only increase the complexity by 1
- Quantifiers (`*`, `+`, `?`, `{n,m}`, `{n,}` or `{n}`)
- Non-capturing groups that set flags (such as `(?i:some_pattern)` or `(?i)some_pattern`)
- Lookahead and lookbehind assertions

Additionally, each use of the following features increase the complexity by 1 regardless of nesting:

- character classes
- back references

If a regular expression is split among multiple variables, the complexity is calculated for each variable individually, not for the whole regular expression. If a regular expression is split over multiple lines, each line is treated individually if it is accompanied by a comment (either a Java comment or a comment within the regular expression), otherwise the regular expression is analyzed as a whole.

Noncompliant code example

```
if (dateString.matches("^(?:(?:31(\\/|-|\\.)(?:0?[13578]|1[02]))\\1|(?:(?:29|30)(\\/|-|\\.)(?:0?[13-9]|1[0-2])\\2))(?:(?:1[6-9]|[2-9]\\d)?\
    handleDate(dateString);
}
```

Compliant solution

```
if (dateString.matches("^\\d{1,2}([-/.])\\d{1,2}\\1\\d{1,4}$")) {
    String dateParts[] = dateString.split("[-/.]");
    int day = Integer.parseInt(dateParts[0]);
    int month = Integer.parseInt(dateParts[1]);
    int year = Integer.parseInt(dateParts[2]);
    // Put logic to validate and process the date based on its integer parts here
}
```

Exceptions

Regular expressions are only analyzed if all parts of the regular expression are either string literals, effectively final local variables or `static final` fields, all of which can be combined using the '+' operator.

When a regular expression is split among multiple variables or commented lines, each part is only analyzed if it is syntactically valid by itself.

AssertJ "assertThatThrownBy" should not be used alone (java:S5958)

Severidad: MAJOR

Root_cause:

Unlike similar AssertJ methods testing exceptions (`assertThatCode()`, `assertThatExceptionOfType()`, ...), the `assertThatThrownBy()` method can be used alone, failing if the code did not raise any exception.

Still, only testing that an exception was raised is not enough to guarantee that it was the expected one, and you should test the exception type or content further. In addition, it will make explicit what you are expecting, without relying on side-effects.

This rule raises an issue when `assertThatThrownBy` is used, without testing the exception further.

Noncompliant code example

```
assertThatThrownBy(() -> shouldThrow()); // Noncompliant, is it really the exception you expected?
```

Compliant solution

```
assertThatThrownBy(() -> shouldThrow()).isInstanceOf(IOException.class);
//or
assertThatThrownBy(() -> shouldThrow()).hasMessage("My exception");
```

Model attributes should follow the Java identifier naming convention (java:S6806)

Severidad: MAJOR

Root_cause:

Spring Expression Language (SpEL) is an expression language used in the Spring Framework for evaluating and manipulating objects, properties, and conditions within Spring-based applications.

`org.springframework.ui.Model` is an interface in the Spring Framework that represents a container for data that can be passed between a controller and a view in a Spring MVC web application, allowing for data sharing during the request-response cycle.

Attributes added to the `org.springframework.ui.Model` should follow the Java identifier naming convention, which means they must start with a letter `a-z, A-Z`, underscore `_`, or a dollar sign `$` and may be followed by letters, digits, underscores, or dollar signs.

Failure to do so may result in SpEL parsing errors when using these attributes in template engines.

Resources:

Documentation

- Java SE - naming conventions
- Spring Expression Language (SpEL)
- Spring IO Docs - Interface Model

How_to_fix:

Follow the Java identifier naming convention.

Noncompliant code example

```
model.addAttribute(" a", 100); // Noncompliant (starts with a space)
model.addAttribute("a-b", 7);  // Noncompliant (contains a hyphen)
model.addAttribute("1c", 42);  // Noncompliant (starts with a digit)
```

Compliant solution

```
model.addAttribute("a", 100);
model.addAttribute("b", 42);
model.addAttribute("_c", 7);
model.addAttribute("$d", 8);
```

Async methods should return void or Future (java:S6810)

Severidad: MAJOR

Resources:

Documentation

- Spring Framework Documentation - Annotation Interface Async

How_to_fix:

Use `void` as the return type if the method is not expected to return a result. Otherwise, a `Future` should be returned, allowing the caller to retrieve the result once it is ready. It is permitted to return more specific subtypes that inherit from `Future`.

Noncompliant code example

```
@Async
public String asyncMethod() {
  ...
}
```

Compliant solution

```
@Async
public Future<String> asyncMethod() {
  ...
}
```

Alternatively, if the method does not need to return a result:

```
@Async
public void asyncMethod() {
  ...
}
```

Root_cause:

The Spring framework provides the annotation `Async` to mark a method (or all methods of a type) as a candidate for asynchronous execution.

Asynchronous methods do not necessarily, by their nature, return the result of their calculation immediately. Hence, it is unexpected and in clear breach of the `Async` contract for such methods to have a return type that is neither `void` nor a `Future` type.

Bluetooth should be configured to use low power (java:S6926)

Severidad: MAJOR

How_to_fix:

- Use `CONNECTION_PRIORITY_LOW_POWER` for `requestConnectionPriority` method.
- Use `ADVERTISE_MODE_LOW_POWER` for `setAdvertiseMode` method to promote low power consumption.

Noncompliant code example

```
public class BluetoothExample {
    private final BluetoothGattCallback gattCallback = new BluetoothGattCallback() {
        @Override
        public void onConnectionStateChange(BluetoothGatt gatt, int status, int newState) {
          // ...
        }

        @Override
        public void onServicesDiscovered(BluetoothGatt gatt, int status) {
            if (status == BluetoothGatt.GATT_SUCCESS) {
                gatt.requestConnectionPriority(BluetoothGatt.CONNECTION_PRIORITY_HIGH); // Noncompliant
            }
        }
    };
}
```

Compliant solution

```
public class BluetoothExample {
   private final BluetoothGattCallback gattCallback = new BluetoothGattCallback() {
        @Override
        public void onConnectionStateChange(BluetoothGatt gatt, int status, int newState) {
          // ...
        }

        @Override
        public void onServicesDiscovered(BluetoothGatt gatt, int status) {
            if (status == BluetoothGatt.GATT_SUCCESS) {
                gatt.requestConnectionPriority(BluetoothGatt.CONNECTION_PRIORITY_LOW_POWER); // Compliant
            }
        }
    };
}
```

Noncompliant code example

```
public class BluetoothExample {
    private void startAdvertising() {
        AdvertiseSettings.Builder settingsBuilder = new AdvertiseSettings.Builder();
        settingsBuilder.setAdvertiseMode(AdvertiseSettings.ADVERTISE_MODE_LOW_LATENCY); // Noncompliant
        // Other settings configuration...
    }
}
```

Compliant solution

```
public class BluetoothExample {
    private void startAdvertising() {
        AdvertiseSettings.Builder settingsBuilder = new AdvertiseSettings.Builder();
        settingsBuilder.setAdvertiseMode(AdvertiseSettings.ADVERTISE_MODE_LOW_POWER); // Compliant
        // Other settings configuration...
    }
}
```

Resources:

Documentation

- Android Developers - BluetoothGattCallback
- Android Developers - BluetoothGatt
- Android Developers - BluetoothDevice
- Android Developers - AdvertiseSettings.Builder

Introduction:

`BluetoothGATT` is a class to provide a functionality to enable communication with Bluetooth Smart or Smart Ready devices.

To connect to a remote peripheral device, a `BluetoothGattCallback` is used and a method `BluetoothDevice#connectGatt` is used to get an instance of this class. GATT-capable devices can be discovered using the Bluetooth device discovery or BLE scan process.

Root_cause:

Using high power consumption modes for Bluetooth operations can drain the device battery faster and may not be suitable for scenarios where power efficiency is crucial.

This rule identifies instances where high power consumption Bluetooth operations are used, specifically when `requestConnectionPriority` or `setAdvertiseMode` methods are invoked with arguments other than those promoting low power consumption.

Source code should be indented consistently (java:S1120)

Severidad: MINOR

Resources:

External coding guidelines

- Google Java Style Guide

How_to_fix:

Change the indentation so that the text starts at the expected column. The expected column should be the configured indent size multiplied by the level at which the code block is nested.

Noncompliant code example

With an indent size of 2:

```
class Foo {
  public int a;
   public int b;   // Noncompliant, expected to start at column 4

...

  public void doSomething() {
    if(something) {
          doSomethingElse();  // Noncompliant, expected to start at column 6
  }   // Noncompliant, expected to start at column 4
  }
}
```

Compliant solution

```
class Foo {
  public int a;
  public int b;

...

  public void doSomething() {
    if(something) {
      doSomethingElse();
    }
  }
}
```

Going the extra mile

You can adopt a tool or configure your IDE to take care of code formatting automatically.

Introduction:

Indentation should be consistent to make the code easy to read, review and modify. To fix this issue, change the indentation so that the text starts at the expected column.

Root_cause:

Consistent indentation is a simple and effective way to improve the code’s readability. It reduces the differences that are committed to source control systems, making code reviews easier.

This rule raises an issue when the indentation does not match the configured value. Only the first line of a badly indented section is reported.

The rule behaves consistently when the indentation settings of the IDE use *spaces* instead of *tabs*. Using *tabs* can lead to inconsistent indentation because the width of a *tab* can be configured differently in different environments.

What is the potential impact?

The readability is decreased. It becomes more tedious to review and modify the code.

Assignments should not be made from within sub-expressions (java:S1121)

Severidad: MAJOR

How_to_fix:

Making assignments within sub-expressions can hinder the clarity of source code.

This practice essentially gives a side-effect to a larger expression, thus making it less readable. This often leads to confusion and potential errors.

Extracting assignments into separate statements is encouraged to keep the code clear and straightforward.

Noncompliant code example

```
String str;
if (!(str = cont.substring(pos1, pos2)).isEmpty()) {  // Noncompliant
  // do something with "str"
}
```

Compliant solution

```
String str = cont.substring(pos1, pos2);
if (!str.isEmpty()) {
  // do something with "str"
}
```

Root_cause:

A common code smell that can hinder the clarity of source code is making assignments within sub-expressions. This practice involves assigning a value to a variable inside a larger expression, such as within a loop or a conditional statement.

This practice essentially gives a side-effect to a larger expression, thus making it less readable. This often leads to confusion and potential errors.

Exceptions

This rule ignores assignments in conditions of `while` statements and assignments enclosed in relational expressions.

```
void processInput(BufferedReader br) {
  String line;
  while ((line = br.readLine()) != null) {
    processLine(line);
  }
}

Object foo;
if ((foo = bar()) != null) {
  // do something with "foo"
}
```

This rule also ignores chained assignments, including compound assignments.

```
int j, i = j = 0;
int k = (j += 1);
byte[] result, bresult;
result = (bresult = new byte[len]);
```

Resources:

- CWE - CWE-481 - Assigning instead of Comparing
- CERT, EXP51-J. - Do not perform assignments in conditional expressions

---

Deprecated elements should have both the annotation and the Javadoc tag (java:S1123)

Severidad: MAJOR

Root_cause:

Deprecation should be marked with both the `@Deprecated` annotation and @deprecated Javadoc tag. The annotation enables tools such as IDEs to warn about referencing deprecated elements, and the tag can be used to explain when it was deprecated, why, and how references should be refactored.

Noncompliant code example

```
class MyClass {

  @Deprecated
  public void foo1() {    // Noncompliant: Add the missing @deprecated Javadoc tag.
  }

  /**
    * @deprecated
    */
  public void foo2() {    // Noncompliant: Add the missing @Deprecated annotation.
  }

}
```

Compliant solution

```
class MyClass {

  /**
    * @deprecated (when, why, refactoring advice...)
    */
  @Deprecated
  public void foo1() {
  }

}
```

Exceptions

The members and methods of a deprecated class or interface are ignored by this rule. The classes and interfaces themselves are still subject to it.

```
/**
 * @deprecated (when, why, etc...)
 */
@Deprecated
class Qix  {

  public void foo() {} // Compliant; class is deprecated

}

/**
 * @deprecated (when, why, etc...)
 */
@Deprecated
interface Plop {

  void bar();

}
```

Modifiers should be declared in the correct order (java:S1124)

Severidad: MINOR

Root_cause:

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. default
13. 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
}
```

Floating point numbers should not be tested for equality (java:S1244)

Severidad: MAJOR

Root_cause:

Floating point math is imprecise because of the challenges of storing such values in a binary representation. Even worse, floating point math is not associative; push a `float` or a `double` through a series of simple mathematical operations and the answer will be different based on the order of those operation because of the rounding that takes place at each step.

Even simple floating point assignments are not simple:

```
float f = 0.1; // 0.100000001490116119384765625
double d = 0.1; // 0.1000000000000000055511151231257827021181583404541015625
```

(Results will vary based on compiler and compiler settings);

Therefore, the use of the equality (`==`) and inequality (`!=`) operators on `float` or `double` values is almost always an error. Instead the best course is to avoid floating point comparisons altogether. When that is not possible, you should consider using one of Java's float-handling `Numbers` such as `BigDecimal` which can properly handle floating point comparisons. A third option is to look not for equality but for whether the value is close enough. I.e. compare the absolute value of the difference between the stored value and the expected value against a margin of acceptable error. Note that this does not cover all cases (`NaN` and `Infinity` for instance).

This rule checks for the use of direct and indirect equality/inequailty tests on floats and doubles.

Noncompliant code example

```
float myNumber = 3.146;
if ( myNumber == 3.146f ) { //Noncompliant. Because of floating point imprecision, this will be false
  // ...
}
if ( myNumber != 3.146f ) { //Noncompliant. Because of floating point imprecision, this will be true
  // ...
}

if (myNumber < 4 || myNumber > 4) { // Noncompliant; indirect inequality test
  // ...
}

float zeroFloat = 0.0f;
if (zeroFloat == 0) {  // Noncompliant. Computations may end up with a value close but not equal to zero.
}
```

Exceptions

Since `NaN` is not equal to itself, the specific case of testing a floating point value against itself is a valid test for `NaN` and is therefore ignored. Though using `Double.isNaN` method should be preferred instead, as intent is more explicit.

```
float f;
double d;
if(f != f) { // Compliant; test for NaN value
  System.out.println("f is NaN");
} else if (f != d) { // Noncompliant
  // ...
}
```

Types should be used in lambdas (java:S2211)

Severidad: MAJOR

Root_cause:

Shared coding conventions allow teams to collaborate effectively. While types for lambda arguments are optional, specifying them anyway makes the code clearer and easier to read.

Noncompliant code example

```
Arrays.sort(rosterAsArray,
    (a, b) -> {  // Noncompliant
        return a.getBirthday().compareTo(b.getBirthday());
    }
);
```

Compliant solution

```
Arrays.sort(rosterAsArray,
    (Person a, Person b) -> {
        return a.getBirthday().compareTo(b.getBirthday());
    }
);
```

Exceptions

When the lambda has one or two parameters and does not have a block this rule will not fire up an issue as things are considered more readable in those cases.

```
stream.map((a, b) -> a.length); // compliant
```

Redundant modifiers should not be used (java:S2333)

Severidad: MINOR

Root_cause:

The methods declared in an `interface` are `public` and `abstract` by default. Any variables are automatically `public static final`. Finally, `class` and `interface` are automatically `public static`. There is no need to explicitly declare them so.

Since annotations are implicitly interfaces, the same holds true for them as well.

Similarly, the `final` modifier is redundant on any method of a `final` class, `private` is redundant on the constructor of an `Enum`, and `static` is redundant for `interface` nested into a `class` or `enum`.

Noncompliant code example

```
public interface Vehicle {

  public void go(int speed, Direction direction);  // Noncompliant
```

Compliant solution

```
public interface Vehicle {

  void go(int speed, Direction direction);
```

"indexOf" checks should not be for positive numbers (java:S2692)

Severidad: CRITICAL

Root_cause:

Most checks against an `indexOf` value compare it with -1 because 0 is a valid index. Checking against `> 0` ignores the first element, which is likely a bug.

```
String name = "ishmael";

if (name.indexOf("ish") > 0) { // Noncompliant
  // ...
}
```

Moreover, if the intent is merely to check the inclusion of a value in a `String` or a `List`, consider using the `contains` method instead.

```
String name = "ishmael";

if (name.contains("ish") {
  // ...
}
```

If the intent is really to skip the first element, comparing it with `>=1` will make it more straightforward.

```
String name = "ishmael";

if (name.indexOf("ish") >= 1) {
  // ...
}
```

This rule raises an issue when an `indexOf` value retrieved from a `String` or a `List` is tested against `> 0`.

Threads should not be started in constructors (java:S2693)

Severidad: BLOCKER

Root_cause:

The problem with invoking `Thread.start()` in a constructor is that you'll have a confusing mess on your hands if the class is ever extended because the superclass' constructor will start the thread before the child class has truly been initialized.

This rule raises an issue any time `start` is invoked in the constructor of a non-`final` class.

Noncompliant code example

```
public class MyClass {

  Thread thread = null;

  public MyClass(Runnable runnable) {
    thread = new Thread(runnable);
    thread.start(); // Noncompliant
  }
}
```

Resources:

- CERT, TSM02-J. - Do not use background threads during class initialization

---

Inner classes which do not reference their owning classes should be "static" (java:S2694)

Severidad: MAJOR

Root_cause:

A non-static inner class has a reference to its outer class, and access to the outer class' fields and methods. That class reference makes the inner class larger and could cause the outer class instance to live in memory longer than necessary.

If the reference to the outer class isn't used, it is more efficient to make the inner class `static` (also called nested). If the reference is used only in the class constructor, then explicitly pass a class reference to the constructor. If the inner class is anonymous, it will also be necessary to name it.

However, while a nested/`static` class would be more efficient, it's worth noting that there are semantic differences between an inner class and a nested one:

- an inner class can only be instantiated within the context of an instance of the outer class.
- a nested (`static`) class can be instantiated independently of the outer class.

Resources:

Documentation

- Oracle Java SE - Nested Classes
- Oracle Java SE - Local Classes

Articles & blog posts

- GeeksforGeeks - Difference between static and non-static nested class in Java

How_to_fix:

There are two scenarios in which this rule will raise an issue:

1. On an *inner class*: make it `static`.
2. On a *local class*: extract it as a `static` *inner class*.

Noncompliant code example

Inner classes that don’t use the outer class reference should be static.

```
public class Fruit {
  // ...

  public class Seed {  // Noncompliant; there's no use of the outer class reference so make it static
    int germinationDays = 0;
    public Seed(int germinationDays) {
      this.germinationDays = germinationDays;
    }
    public int getGerminationDays() {
      return germinationDays;
    }
  }
}
```

Compliant solution

```
public class Fruit {
  // ...

  public static class Seed {
    int germinationDays = 0;
    public Seed(int germinationDays) {
      this.germinationDays = germinationDays;
    }
    public int getGerminationDays() {
      return germinationDays;
    }
  }
}
```

Local classes that don’t use the outer class reference should be extracted as a static inner classes.

Noncompliant code example

```
public class Foo {
  public Foo() {
    class Bar { // Noncompliant
      void doSomething() {
        // ...
      }
    }
    new Bar().doSomething();
  }

  public void method() {
    class Baz { // Noncompliant
      void doSomething() {
        // ...
      }
    }
    new Baz().doSomething();
  }
}
```

Compliant solution

```
public class Foo {
  public Foo() {
    new Bar().doSomething();
  }

  public void method()  {
    new Baz().doSomething();
  }

  private static class Bar { // Compliant
    void doSomething() {
      // ...
    }
  }

  private static class Baz { // Compliant
    void doSomething() {
      // ...
    }
  }
}
```

"PreparedStatement" and "ResultSet" methods should be called with valid indices (java:S2695)

Severidad: BLOCKER

Root_cause:

`PreparedStatement` is an object that represents a precompiled SQL statement, that can be used to execute the statement multiple times efficiently.

`ResultSet` is the Java representation of the result set of a database query obtained from a `Statement` object. A default `ResultSet` object is not updatable and has a cursor that moves forward only.

The parameters in `PreparedStatement` and `ResultSet` are indexed beginning at 1, not 0. When an invalid index is passed to the `PreparedStatement` or `ResultSet` methods, an `IndexOutOfBoundsException` is thrown. This can cause the program to crash or behave unexpectedly, leading to a poor user experience.

This rule raises an issue for the `get` methods in `PreparedStatement` and the `set` methods in `ResultSet`.

How_to_fix:

Ensure the index passed to the `PreparedStatement` and `ResultSet` methods is valid.

Noncompliant code example

```
PreparedStatement ps = con.prepareStatement("SELECT fname, lname FROM employees where hireDate > ? and salary < ?");
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 > ? and salary < ?");
ps.setDate(1, date);
ps.setDouble(2, salary);

ResultSet rs = ps.executeQuery();
while (rs.next()) {
  String fname = rs.getString(1);
  // ...
}
```

Resources:

Documentation

- Oracle SDK 20 - PreparedStatement
- Oracle SDK 20 - ResultSet
- Oracle SDK 20 - Connection#prepareStatement

AssertJ assertions "allMatch" and "doesNotContains" should also test for emptiness (java:S5841)

Severidad: MINOR

Root_cause:

AssertJ assertions `allMatch` and `doesNotContains` on an empty list always returns true whatever the content of the predicate. Despite being correct, you should make explicit if you expect an empty list or not, by adding `isEmpty()`/`isNotEmpty()` in addition to calling the assertion, or by testing the list's content further. It will justify the useless predicate to improve clarity or increase the reliability of the test.

This rule raises an issue when any of the methods listed are used without asserting that the list is empty or not and without testing the content.

Targetted methods:

- `allMatch`
- `allSatisfy`
- `doesNotContain`
- `doesNotContainSequence`

- `doesNotContainSubsequence`
- `doesNotContainAnyElementsOf`

Noncompliant code example

```
List<String> logs = getLogs();

assertThat(logs).allMatch(e -> e.contains(“error”)); // Noncompliant, this test pass if logs are empty!
assertThat(logs).doesNotContain("error"); // Noncompliant, do you expect any log?
```

Compliant solution

```
List<String> logs = getLogs();

assertThat(logs).isNotEmpty().allMatch(e -> e.contains(“error”));
// Or
assertThat(logs).hasSize(5).allMatch(e -> e.contains(“error”));
// Or
assertThat(logs).isEmpty();

// Despite being redundant, this is also acceptable since it explains why you expect an empty list
assertThat(logs).doesNotContain("error").isEmpty();
// or test the content of the list further
assertThat(logs).contains("warning").doesNotContain("error");
```

Assertions comparing incompatible types should not be made (java:S5845)

Severidad: CRITICAL

Resources:

- S2159 - Silly equality checks should not be made

Root_cause:

Assertions comparing incompatible types always fail, and negative assertions always pass. At best, negative assertions are useless. At worst, the developer loses time trying to fix his code logic before noticing wrong assertions.

Dissimilar types are:

- comparing a primitive with null
- comparing an object with an unrelated primitive (E.G. a string with an int)
- comparing unrelated classes
- comparing an array to a non-array
- comparing two arrays of dissimilar types

This rule also raises issues for unrelated `class` and `interface` or unrelated `interface` types in negative assertions. Because except in some corner cases, those types are more likely to be dissimilar. And inside a negative assertion, there is no test failure to inform the developer about this unusual comparison.

Supported test frameworks:

- JUnit4
- JUnit5
- AssertJ

Noncompliant code example

```
interface KitchenTool {}
interface Plant {}
class Spatula implements KitchenTool {}
class Tree implements Plant {}

void assertValues(int size,
                  Spatula spatula, KitchenTool tool,  KitchenTool[] tools,
                  Tree    tree,    Plant       plant, Tree[]        trees) {

  // Whatever the given values, those negative assertions will always pass due to dissimilar types:
  assertThat(size).isNotNull();           // Noncompliant; primitives can not be null
  assertThat(spatula).isNotEqualTo(tree); // Noncompliant; unrelated classes
  assertThat(tool).isNotSameAs(tools);    // Noncompliant; array & non-array
  assertThat(trees).isNotEqualTo(tools);  // Noncompliant; incompatible arrays

  // Those assertions will always fail
  assertThat(size).isNull();                       // Noncompliant
  assertThat(spatula).isEqualTo(tree);             // Noncompliant

  // Those negative assertions are more likely to always pass
  assertThat(spatula).isNotEqualTo(plant); // Noncompliant; unrelated class and interface
  assertThat(tool).isNotEqualTo(plant);    // Noncompliant; unrelated interfaces
}
```

Assertions should not be used in production code (java:S5960)

Severidad: MAJOR

Root_cause:

Assertions are intended to be used in **test** code, but not in **production** code. It is confusing, and might lead to `ClassNotFoundException` when the build tools only provide the required dependency in test scope.

In addition, assertions will throw a sub-class of `Error`: `AssertionError`, which should be avoided in production code.

This rule raises an issue when any assertion intended to be used in test is used in production code.

Supported frameworks:

- JUnit
- FestAssert
- AssertJ

Note: this does not apply for `assert` from Java itself or if the source code package name is related to tests (contains: `test` or `assert` or `junit`).

Test methods should not contain too many assertions (java:S5961)

Severidad: MAJOR

Root_cause:

A common good practice is to write test methods targeting only one logical concept, that can only fail for one reason.

While it might make sense to have more than one assertion to test one concept, having too many is a sign that a test became too complex and should be refactored to multiples ones.

This rule will report any test method containing more than a given number of assertion.

Noncompliant code example

With a parameter of 2.

```
@Test
void test() { // Refactor this method.
  assertEquals(1, f(1));
  assertEquals(2, f(2));
  assertEquals(3, g(1));
}
```

Compliant solution

```
void test_f() {
  assertEquals(1, f(1));
  assertEquals(2, f(2));
}
void test_g() {
  assertEquals(3, g(1));
}
```

Tests method should not be annotated with competing annotations (java:S5967)

Severidad: MAJOR

Root_cause:

Annotating unit tests with more than one test-related annotation is not only useless but could also result in unexpected behavior like failing tests or unwanted side-effects.

This rule reports an issue when a test method is annotated with more than one of the following competing annotation:

- @Test

- @RepeatedTest
- @ParameterizedTest
- @TestFactory
- @TestTemplate

Noncompliant code example

```
@Test
@RepeatedTest(2) // Noncompliant, this test will be repeated 3 times
void test() { }

@ParameterizedTest
@Test
@MethodSource("methodSource")
void test2(int argument) { } // Noncompliant, this test will fail with ParameterResolutionException
```

Compliant solution

```
@RepeatedTest(2)
void test() { }

@ParameterizedTest
@MethodSource("methodSource")
void test2(int argument) { }
```

Field dependency injection should be avoided (java:S6813)

Severidad: MAJOR

Root_cause:

Dependency injection frameworks such as Spring, Quarkus, and others support dependency injection by using annotations such as `@Inject` and `@Autowired`. These annotations can be used to inject beans via constructor, setter, and field injection.

Generally speaking, field injection is discouraged. It allows the creation of objects in an invalid state and makes testing more difficult. The dependencies are not explicit when instantiating a class that uses field injection.

In addition, field injection is not compatible with final fields. Keeping dependencies immutable where possible makes the code easier to understand, easing development and maintenance.

Finally, because values are injected into fields after the object has been constructed, they cannot be used to initialize other non-injected fields inline.

This rule raises an issue when the `@Autowired` or `@Inject` annotations are used on a field.

Resources:

Articles & blog posts

- Baeldung - Why Is Field Injection Not Recommended?
- Baeldung - Constructor Dependency Injection in Spring
- Oliver Drotbohm - Why field injection is evil
- GitHub Discussions - Field injection in quarkus

How_to_fix:

Use constructor injection instead.

By using constructor injection, the dependencies are explicit and must be passed during an object's construction. This avoids the possibility of instantiating an object in an invalid state and makes types more testable. Fields can be declared final, which makes the code easier to understand, as dependencies don't change after instantiation.

Noncompliant code example

```
public class SomeService {
    @Autowired
    private SomeDependency someDependency; // Noncompliant

    private String name = someDependency.getName(); // Will throw a NullPointerException
}
```

Compliant solution

```
public class SomeService {
    private final SomeDependency someDependency;
    private final String name;

    @Autowired
    public SomeService(SomeDependency someDependency) {
        this.someDependency = someDependency;
        name = someDependency.getName();
    }
}
```

Optional REST parameters should have an object type (java:S6814)

Severidad: MAJOR

How_to_fix:

Replace primitive types, such as `boolean`, `char`, `int`, with the corresponding wrapper type, such as `Boolean`, `Character`, `Integer`.

Alternatively, you might choose to remove `required = false` from the annotation and use an `Optional<T>` type for the parameter, such as `Optional<Boolean>` or `Optional<String>`, which automatically makes the REST parameter optional. This is the preferred approach because it enforces the proper handling of `null` in the method implementation.

Noncompliant code example

```
@RequestMapping(value = {"/article", "/article/{id}"})
public Article getArticle(@PathVariable(required = false) int articleId) { // Noncompliant, null cannot be mapped to int
   //...
}
```

Compliant solution

```
@RequestMapping(value = {"/article", "/article/{id}"})
public Article getArticle(@PathVariable(required = false) Integer articleId) { // Compliant
   //...
}
```

Noncompliant code example

```
@RequestMapping(value = {"/article", "/article/{id}"})
public Article getArticle(@PathVariable(required = false) int articleId) { // Noncompliant, null cannot be mapped to int
   //...
}
```

Compliant solution

```
@RequestMapping(value = {"/article", "/article/{id}"})
public Article getArticle(@PathVariable Optional<Integer> articleId) { // Compliant and preferred approach
   //...
}
```

Root_cause:

Spring provides two options to mark a REST parameter as optional:

1. Use `required = false` in the `@PathVariable` or `@RequestParam` annotation of the respective method parameter or
2. Use type `java.util.Optional<T>` for the method parameter

When using 1., the absence of the parameter, when the REST function is called, is encoded by `null`, which can only be used for object types. If `required = false` is used for a parameter with a primitive and the REST function is called without the parameter, a runtime exception occurs because the Spring data mapper cannot map the `null` value to the parameter.

Resources:

Documentation

- Spring Framework API - Annotation Interface PathVariable

Articles & blog posts

- Baeldung - Spring Optional Path Variables

"@Autowired" should only be used on a single constructor (java:S6818)

Severidad: MAJOR

Resources:

Documentation

- [Spring Framework - Using @Autowired]

Articles & blog posts

- [Baeldung - Guide to Spring @Autowired]

How_to_fix:

To maintain code clarity and ensure that the Spring context can create beans correctly, have only one constructor annotated with `@Autowired` within a Spring component or set `required = false`.

Noncompliant code example

```
@Component
public class MyComponent {
  private final MyService myService;

  @Autowired
  public MyComponent(MyService myService) {
    this.myService = myService;
    // ...
  }

  @Autowired  // Noncompliant
  public MyComponent(MyService myService, Integer i) {
    this.myService = myService;
    // ...
  }

  @Autowired  // Noncompliant
  public MyComponent(MyService myService, Integer i, String s) {
    this.myService = myService;
    // ...
  }
}
```

Compliant solution

```
@Component
public class MyComponent {
  private final MyService myService;

  @Autowired
  public MyComponent(MyService myService) {
    this.myService = myService;
    // ...
  }

  public MyComponent(MyService myService, Integer i) {
    this.myService = myService;
    // ...
  }

  public MyComponent(MyService myService, Integer i, String s) {
    this.myService = myService;
    // ...
  }
}
```

Noncompliant code example

```
@Component
public class MyComponent {
  private final MyService myService;

  @Autowired
  public MyComponent(MyService myService) {
    this.myService = myService;
    // ...
  }

  @Autowired  // Noncompliant
  public MyComponent(MyService myService, Integer i) {
    this.myService = myService;
    // ...
  }

  @Autowired  // Noncompliant
  public MyComponent(MyService myService, Integer i, String s) {
    this.myService = myService;
    // ...
  }
}
```

Compliant solution

```
@Component
public class MyComponent {
  private final MyService myService;

  @Autowired
  public MyComponent(MyService myService) {
    this.myService = myService;
    // ...
  }

  @Autowired(required=false)  // Compliant
  public MyComponent(MyService myService, Integer i) {
    this.myService = myService;
    // ...
  }

  @Autowired(required=false)  // Compliant
  public MyComponent(MyService myService, Integer i, String s) {
    this.myService = myService;
    // ...
  }
}
```

Root_cause:

`@Autowired` is an annotation in the Spring Framework for automatic dependency injection. It tells Spring to automatically provide the required dependencies (such as other beans or components) to a class's fields, methods, or constructors, allowing for easier and more flexible management of dependencies in a Spring application. In other words, it's a way to wire up and inject dependencies into Spring components automatically, reducing the need for manual configuration and enhancing modularity and maintainability.

In any bean class, only one constructor is permitted to declare `@Autowired` with the `required` attribute set to true. This signifies the constructor to be automatically wired when used as a Spring bean. Consequently, when the required attribute remains at its default value (true), only a singular constructor can bear the `@Autowired` annotation. In cases where multiple constructors have this annotation, they must all specify `required=false` to be eligible as candidates for auto-wiring.

Unnecessary imports should be removed (java:S1128)

Severidad: MINOR

How_to_fix:

While it's not difficult to remove these unneeded lines manually, modern code editors support the removal of every unnecessary import with a single click from every file of the project.

Noncompliant code example

```
package myapp.helpers;

import java.io.IOException;
import java.nio.file.*;
import java.nio.file.*;     // Noncompliant - package is imported twice
import java.lang.Runnable;  // Noncompliant - java.lang is imported by default

public class FileHelper {
    public static String readFirstLine(String filePath) throws IOException {
        return Files.readAllLines(Paths.get(filePath)).get(0);
    }
}
```

Compliant solution

```
package myapp.helpers;

import java.io.IOException;
import java.nio.file.*;
```

```
public class FileHelper {
    public static String readFirstLine(String filePath) throws IOException {
        return Files.readAllLines(Paths.get(filePath)).get(0);
    }
}
```

Root_cause:

Unnecessary imports refer to importing types that are not used or referenced anywhere in the code.

Although they don't affect the runtime behavior of the application after compilation, removing them will:

- Improve the readability and maintainability of the code.
- Help avoid potential naming conflicts.
- Improve the build time, as the compiler has fewer lines to read and fewer types to resolve.
- Reduce the number of items the code editor will show for auto-completion, thereby showing fewer irrelevant suggestions.

Exceptions

Imports for types mentioned in Javadocs are ignored.

Resources:

Documentation

- Java packages

Related rules

- S1144 - Unused "private" methods should be removed
- S1481 - Unused local variables should be removed

The signature of "finalize()" should match that of "Object.finalize()" (java:S1175)

Severidad: CRITICAL

Root_cause:

`Object.finalize()` is called by the Garbage Collector at some point after the object becomes unreferenced.

In general, overloading `Object.finalize()` is a bad idea because:

- The overload may not be called by the Garbage Collector.
- Users are not expected to call `Object.finalize()` and will get confused.

But beyond that it's a terrible idea to name a method "finalize" if it doesn't actually override `Object.finalize()`.

Noncompliant code example

```
public int finalize(int someParameter) {        // Noncompliant
  /* ... */
}
```

Compliant solution

```
public int someBetterName(int someParameter) {  // Compliant
  /* ... */
}
```

Public types, methods and fields (API) should be documented with Javadoc (java:S1176)

Severidad: MAJOR

Root_cause:

Undocumented APIs pose significant challenges in software development for several reasons:

- **Lack of Clarity:** developers struggling to understand how to use the API correctly. This can lead to misuse and unexpected results.
- **Increased Development Time:** developers spending extra time reading and understanding the source code, which slows down the development process.

- **Error Prone:** developers are more likely to make mistakes that lead to bugs or system crashes when the intent or the error handling of an API is not clear.
- **Difficult Maintenance and Updates:** developers may not understand the existing functionality well enough to add new features without breaking the existing ones.
- **Poor Collaboration:** collaboration, when there is lack of documentation, leads to confusion and inconsistencies.

It is recommended to document the API using **JavaDoc** to clarify what is the contract of the API. This is especially important for public APIs, as they are used by other developers.

Exceptions

The following public methods and constructors are not taken into account by this rule:

- Getters and setters.
- Methods overriding another method (usually annotated with `@Override`).
- Empty constructors.
- Static constants.

How_to_fix:

On top of a main description for each member of a public API, the following **Javadoc** elements are required to be described:

- Parameters, using `@param parameterName`.
- Thrown exceptions, using `@throws exceptionName`.
- Method return values, using `@return`.
- Generic types, using `@param <T>`.

Furthermore, the following guidelines should be followed:

- At least 1 line of description.
- All parameters documented with `@param`, and names should match.
- All checked exceptions should be documented with `@throws`
- `@return` present and documented when method return type is not `void`.
- Placeholders like `"TODO"`, `"FIXME"`, `"…"` should be avoided.

For the parameters of the rule, the following rules are applied:

- `?` matches a single character
- `*` matches zero or more characters
- `**` matches zero or more packages

Examples:

- `java.internal.InternalClass` will match only `InternalClass` class.
- `java.internal.*` will match any member of `java.internal` package.
- `java.internal.**` same as above, but including sub-packages.

Noncompliant code example

```
/**
  * This is a Javadoc comment
  */
public class MyClass<T> implements Runnable {   // Noncompliant - missing '@param <T>'

  public static final int DEFAULT_STATUS = 0;   // Compliant - static constant
  private int status;                           // Compliant - not public

  public String message;                        // Noncompliant

  public MyClass() {                            // Noncompliant - missing documentation
    this.status = DEFAULT_STATUS;
  }

  public void setStatus(int status) {           // Compliant - setter
    this.status = status;
  }

  @Override
  public void run() {                           // Compliant - has @Override annotation
  }

  protected void doSomething() {                // Compliant - not public
  }

  public void doSomething2(int value) {         // Noncompliant
  }

  public int doSomething3(int value) {          // Noncompliant
    return value;
  }
}
```

Compliant solution

```
/**
  * This is a Javadoc comment
  * @param <T> the parameter of the class
  */
public class MyClass<T> implements Runnable {

  public static final int DEFAULT_STATUS = 0;
  private int status;

  /**
    * This is a Javadoc comment
    */
  public String message;

  /**
   * Class comment...
   */
  public MyClass() {
    this.status = DEFAULT_STATUS;
  }

  public void setStatus(int status) {
    this.status = status;
  }

  @Override
  public void run() {
  }

  protected void doSomething() {
  }

  /**
    * Will do something.
    * @param value the value to be used
    */
  public void doSomething(int value) {
  }

  /**
    *  {@inheritDoc}
    */
  public int doSomething(int value) {
    return value;
  }
}
```

Introduction:

A good API documentation is a key factor in the usability and success of a software API. It ensures that developers can effectively use, maintain, and collaborate on the API.

Resources:

Documentation

- Oracle - JavaDoc

Articles & blog posts

- Technical Writer HQ - How to write API documentation
- FreeCodeCamp - How to write API documentation like a pro

Unused local variables should be removed (java:S1481)

Severidad: MINOR

Root_cause:

An unused local variable is a variable that has been declared but is not used anywhere in the block of code where it is defined. It is dead code, contributing to unnecessary complexity and leading to confusion when reading the code. Therefore, it should be removed from your code to maintain clarity and efficiency.

What is the potential impact?

Having unused local variables in your code can lead to several issues:

- **Decreased Readability**: Unused variables can make your code more difficult to read. They add extra lines and complexity, which can distract from the main logic of the code.
- **Misunderstanding**: When other developers read your code, they may wonder why a variable is declared but not used. This can lead to confusion and misinterpretation of the code’s intent.
- **Potential for Bugs**: If a variable is declared but not used, it might indicate a bug or incomplete code. For example, if you declared a variable intending to use it in a calculation, but then forgot to do so, your program might not work as expected.
- **Maintenance Issues**: Unused variables can make code maintenance more difficult. If a programmer sees an unused variable, they might think it is a mistake and try to 'fix' the code, potentially introducing new bugs.
- **Memory Usage**: Although modern compilers are smart enough to ignore unused variables, not all compilers do this. In such cases, unused variables take up memory space, leading to inefficient use of resources.

In summary, unused local variables can make your code less readable, more confusing, and harder to maintain, and they can potentially lead to bugs or inefficient memory use. Therefore, it is best to remove them.

How_to_fix:

The fix for this issue is straightforward. Once you ensure the unused variable is not part of an incomplete implementation leading to bugs, you just need to remove it.

Noncompliant code example

```
public int numberOfMinutes(int hours) {
  int seconds = 0;   // Noncompliant - seconds is unused
  return hours * 60;
}
```

Compliant solution

```
public int numberOfMinutes(int hours) {
  return hours * 60;
}
```

Catches should be combined (java:S2147)

Severidad: MINOR

Root_cause:

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;
}
```

Underscores should be used to make large numbers readable (java:S2148)

Severidad: MINOR

Root_cause:

Beginning with Java 7, it is possible to add underscores ('_') to numeric literals to enhance readability. The addition of underscores in this manner has no semantic meaning, but makes it easier for maintainers to understand the code.

The number of digits to the left of a decimal point needed to trigger this rule varies by base.

Base	Minimum digits
binary	9
octal	9
decimal	6
hexadecimal	9

It is only the presence of underscores, not their spacing that is scrutinized by this rule.

Note that this rule is automatically disabled when the project's `sonar.java.source` is lower than `7`.

Noncompliant code example

```
int i = 10000000;  // Noncompliant; is this 10 million or 100 million?
int  j = 0b01101001010011011110010101011110;  // Noncompliant
long l = 0x7fffffffffffffffL;  // Noncompliant
```

Compliant solution

```
int i = 10_000_000;
int  j = 0b01101001_01001101_11100101_01011110;
long l = 0x7fff_ffff_ffff_ffffL;
```

Private mutable members should not be stored or returned directly (java:S2384)

Severidad: MINOR

Resources:

- CWE - CWE-374 - Passing Mutable Objects to an Untrusted Method
- CWE - CWE-375 - Returning a Mutable Object to an Untrusted Caller
- CERT, OBJ05-J. - Do not return references to private mutable class members
- CERT, OBJ06-J. - Defensively copy mutable inputs and mutable internal components
- CERT, OBJ13-J. - Ensure that references to mutable objects are not exposed

Root_cause:

Mutable objects are those whose state can be changed. For instance, an array is mutable, but a String is not. Private mutable class members should never be returned to a caller or accepted and stored directly. Doing so leaves you vulnerable to unexpected changes in your class state.

Instead use an unmodifiable `Collection` (via `Collections.unmodifiableCollection`, `Collections.unmodifiableList`, …) or make a copy of the mutable object, and store or return the copy instead.

This rule checks that private arrays, collections and Dates are not stored or returned directly.

Noncompliant code example

```
class A {
  private String [] strings;

  public A () {
    strings = new String[]{"first", "second"};
  }

  public String [] getStrings() {
    return strings; // Noncompliant
  }

  public void setStrings(String [] strings) {
    this.strings = strings;  // Noncompliant
  }
}

public class B {
  private A a = new A();  // At this point a.strings = {"first", "second"};

  public void wreakHavoc() {
    a.getStrings()[0] = "yellow";  // a.strings = {"yellow", "second"};
  }
}
```

Compliant solution

```
class A {
  private String [] strings;

  public A () {
    strings = new String[]{"first", "second"};
  }

  public String [] getStrings() {
    return strings.clone();
  }

  public void setStrings(String [] strings) {
    this.strings = strings.clone();
  }
}

public class B {

  private A a = new A();  // At this point a.strings = {"first", "second"};

  public void wreakHavoc() {
    a.getStrings()[0] = "yellow";  // a.strings = {"first", "second"};
  }
}
```

Mutable fields should not be "public static" (java:S2386)

Severidad: MINOR

Resources:

- CWE - CWE-582 - Array Declared Public, Final, and Static
- CWE - CWE-607 - Public Static Final Field References Mutable Object
- CERT, OBJ01-J. - Limit accessibility of fields
- CERT, OBJ13-J. - Ensure that references to mutable objects are not exposed

Root_cause:

There is no good reason to have a mutable object as the `public` (by default), `static` member of an `interface`. Such variables should be moved into classes and their visibility lowered.

Similarly, mutable `static` members of classes and enumerations which are accessed directly, rather than through getters and setters, should be protected to the degree possible. That can be done by reducing visibility or making the field `final` if appropriate.

Note that making a mutable field, such as an array, `final` will keep the variable from being reassigned, but doing so has no effect on the mutability of the internal state of the array (i.e. it doesn't accomplish the goal).

This rule raises issues for `public static` array, `Collection`, `Date`, and `awt.Point` members.

Noncompliant code example

```
public interface MyInterface {
  public static String [] strings; // Noncompliant
}

public class A {
  public static String [] strings1 = {"first","second"};  // Noncompliant
  public static String [] strings2 = {"first","second"};  // Noncompliant
  public static List<String> strings3 = new ArrayList<>();  // Noncompliant
  // ...
}
```

Child class fields should not shadow parent class fields (java:S2387)

Severidad: BLOCKER

Root_cause:

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
  // ...
}
```

Inner class calls to super class methods should be unambiguous (java:S2388)

Severidad: MAJOR

Root_cause:

An inner class that extends another type can call methods from both the outer class and parent type directly, without prepending `super.` or `Outer.this.`.

When both the outer and parent classes contain a method with the same name, the compiler will resolve an unqualified call to the parent type's implementation. The maintainer or a future reader may confuse the method call as calling the outer class's implementation, even though it really calls the super type's.

To make matters worse, the maintainer sees the outer class's implementation in the same file as the call in the inner class, while the parent type is often declared in another file. The maintainer may not even be aware of the ambiguity present, as they do not see the parent's implementation.

How_to_fix:

Explicitly call the super type's method by prepending `super.` to the method call. If the intention was to call the outer class's implementation, prepend `Outer.this.` instead.

Noncompliant code example

```
public class Parent {
  public void foo() { ... }
}

public class Outer {
  public void foo() { ... }
  public class Inner extends Parent {
    public void doSomething() {
      foo();  // Noncompliant, it is not explicit if Outer#foo or Parent#foo is the intended implementation to be called.
      // ...
    }
  }
}
```

Compliant solution

```
public class Parent {
  public void foo() { ... }
}

public class Outer {
  public void foo() { ... }

  public class Inner extends Parent {
    public void doSomething() {
      super.foo(); // Compliant, it is explicit that Parent#foo is the desired implementation to be called.
      // ...
    }
  }
}
```

Factory method injection should be used in "@Configuration" classes (java:S3305)

Severidad: CRITICAL

Root_cause:

When `@Autowired` is used, dependencies need to be resolved when the class is instantiated, which may cause early initialization of beans or lead the context to look in places it shouldn’t to find the bean. To avoid this tricky issue and optimize the way the context loads, dependencies should be requested as late as possible. That means using parameter injection instead of field injection for dependencies that are only used in a single `@Bean` method.

Noncompliant code example

```
@Configuration
public class FooConfiguration {

  @Autowired private DataSource dataSource;  // Noncompliant

  @Bean
  public MyService myService() {
    return new MyService(this.dataSource);
  }
}
```

Compliant solution

```
@Configuration
public class FooConfiguration {

 @Bean
  public MyService myService(DataSource dataSource) {
    return new MyService(dataSource);
  }
}
```

Exceptions

Fields used in methods that are called directly by other methods in the application (as opposed to being invoked automatically by the Spring framework) are ignored by this rule so that direct callers don’t have to provide the dependencies themselves.

Constructor injection should be used instead of field injection (java:S3306)

Severidad: MAJOR

Root_cause:

Field injection seems like a tidy way to get your classes what they need to do their jobs, but it’s really a `NullPointerException` waiting to happen unless all your class constructors are `private`. That’s because any class instances that are constructed by callers, rather than instantiated by a Dependency Injection framework compliant with the JSR-330 (Spring, Guice, …), won’t have the ability to perform the field injection.

Instead `@Inject` should be moved to the constructor and the fields required as constructor parameters.

This rule raises an issue when classes with non-`private` constructors (including the default constructor) use field injection.

Noncompliant code example

```
class MyComponent {  // Anyone can call the default constructor

  @Inject MyCollaborator collaborator;  // Noncompliant

  public void myBusinessMethod() {
    collaborator.doSomething();  // this will fail in classes new-ed by a caller
  }
}
```

Compliant solution

```
class MyComponent {

  private final MyCollaborator collaborator;

  @Inject
  public MyComponent(MyCollaborator collaborator) {
    Assert.notNull(collaborator, "MyCollaborator must not be null!");
    this.collaborator = collaborator;
  }

  public void myBusinessMethod() {
    collaborator.doSomething();
  }
}
```

Ternary operators should not be nested (java:S3358)

Severidad: MAJOR

Root_cause:

Nested ternaries are hard to read and can make the order of operations complex to understand.

```
public String getReadableStatus(Job j) {
  return j.isRunning() ? "Running" : j.hasErrors() ? "Failed" : "Succeeded";  // Noncompliant
}
```

Instead, use another line to express the nested operation in a separate statement.

```
public String getReadableStatus(Job j) {
  if (j.isRunning()) {
    return "Running";
  }
  return j.hasErrors() ? "Failed" : "Succeeded";
}
```

Double Brace Initialization should not be used (java:S3599)

Severidad: MINOR

Root_cause:

Because Double Brace Initialization (DBI) creates an anonymous class with a reference to the instance of the owning object, its use can lead to memory leaks if the anonymous inner class is returned and held by other objects. Even when there’s no leak, DBI is so obscure that it’s bound to confuse most maintainers.

For collections, use `Arrays.asList` instead, or explicitly add each item directly to the collection.

Noncompliant code example

```
Map source = new HashMap(){{ // Noncompliant
    put("firstName", "John");
    put("lastName", "Smith");
}};
```

Compliant solution

```
Map source = new HashMap();
// ...
source.put("firstName", "John");
source.put("lastName", "Smith");
// ...
```

InputSteam.read() implementation should not return a signed byte (java:S4517)

Severidad: MAJOR

Root_cause:

According to the Java documentation, any implementation of the `InputSteam.read()` method is supposed to read the next byte of data from the input stream. The value byte must be an `int` in the range 0 to 255. If no byte is available because the end of the stream has been reached, the value -1 is returned.

But in Java, the `byte` primitive data type is an 8-bit signed two’s complement integer. It has a minimum value of -128 and a maximum value of 127. So by contract, the implementation of an `InputSteam.read()` method should never directly return a `byte` primitive data type. A conversion into an unsigned byte must be done before by applying a bitmask.

Noncompliant code example

```
@Override
public int read() throws IOException {
  if (pos == buffer.length()) {
    return -1;
  }
  return buffer.getByte(pos++); // Noncompliant, a signed byte value is returned
}
```

Compliant solution

```
@Override
public int read() throws IOException {
  if (pos == buffer.length()) {
    return -1;
  }
  return buffer.getByte(pos++) & 0xFF; // The 0xFF bitmask is applied
}
```

Persistent entities should not be used as arguments of "@RequestMapping" methods (java:S4684)

Severidad: CRITICAL

How_to_fix:

The following code is vulnerable to a mass assignment attack because it allows modifying the `User` persistent entities thanks to maliciously forged `Wish` object properties.

Noncompliant code example

```
import javax.persistence.Entity;

@Entity
public class Wish {
  Long productId;
  Long quantity;
  Client client;
}

@Entity
public class Client {
  String clientId;
  String name;
  String password;
}

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class PurchaseOrderController {

  @RequestMapping(path = "/saveForLater", method = RequestMethod.POST)
  public String saveForLater(Wish wish) { // Noncompliant
    session.save(wish);
  }
}
```

Compliant solution

```
public class WishDTO {
  Long productId;
  Long quantity;
  Long clientId;
}

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class PurchaseOrderController {

  @RequestMapping(path = "/saveForLater", method = RequestMethod.POST)
  public String saveForLater(WishDTO wish) {
    Wish persistentWish = new Wish();
    persistentWish.productId = wish.productId
    persistentWish.quantity = wish.quantity
    persistentWish.client = getClientById(with.clientId)
    session.save(persistentWish);
  }
}
```

How does this work?

The compliant code implements a Data Transfer Object (DTO) layer. Instead of accepting a persistent `Wish` entity as a parameter, the vulnerable method accepts a `WishDTO` object with a safe, minimal set of properties. It then instantiates a persistent entity and initializes it based on the DTO properties' values. The resulting object can safely be persisted in the database.

Root_cause:

By accepting persistent entities as method arguments, the application allows clients to manipulate the object’s properties directly.

What is the potential impact?

Attackers could forge malicious HTTP requests that will alter unexpected properties of persistent objects. This can lead to unauthorized modifications of the entity’s state. This is known as a **mass assignment** attack.

Depending on the affected objects and properties, the consequences can vary.

Privilege escalation

If the affected object is used to store the client’s identity or permissions, the attacker could alter it to change their entitlement on the application. This can lead to horizontal or vertical privilege escalation.

Security checks bypass

Because persistent objects are modified directly without prior logic, attackers could exploit this issue to bypass security measures otherwise enforced by the application. For example, an attacker might be able to change their e-mail address to an invalid one by directly setting it without going through the application’s email validation process.

The same could also apply to passwords that attackers could change without complexity validation or knowledge of their current value.

Resources:

Documentation

- OWASP - Mass Assignment Cheat Sheet

Standards

- OWASP - Top 10 2021 - Category A8 - Software and Data Integrity Failures
- OWASP - Top 10 2017 - Category A5 - Broken Access Control
- CWE - CWE-915 - Improperly Controlled Modification of Dynamically-Determined Object Attributes

Articles & blog posts

OWASP O2 Platform Blog - Two Security Vulnerabilities in the Spring Framework’s MVC

Introduction:

With Spring, when a request mapping method is configured to accept bean objects as arguments, the framework will automatically bind HTTP parameters to those objects' properties. If the targeted beans are also persistent entities, the framework will also store those properties in the storage backend, usually the application’s database.

'List.remove()' should not be used in ascending 'for' loops (java:S5413)

Severidad: MAJOR

Resources:

Documentation

- Java SE 7 API Specification: Collection.remove
- Java SE 8 API Specification: Collection.removeIf
- S127 - "for" loop stop conditions should be invariant

Root_cause:

When `List.remove()` is called, the list shrinks, and the indices of all elements following the removed element are decremented by one. If this operation is performed within a loop that iterates through the elements in ascending order, it will cause the loop to skip the element immediately following the removed element.

How_to_fix:

There are three ways how to fix this issue:

1. Replace the loop with a call to `Collection.removeIf()`. This is the preferred solution.
2. Replace the ascending loop with a descending loop. Use this approach if the preferred solution is not possible due to side effects of the loop.
3. Adjust the loop counter within the loop body after the call to `Collection.remove()`. **This approach is not recommended**, because it will raise an issue with rule *S127 - "for" loop stop conditions should be invariant*

Noncompliant code example

If the loop can be replaced with Java 8’s `Collection.removeIf` method, depending on the side effects of the loop and your Java target version, then this is the preferred solution for this issue.

```
void removeFrom(List<String> list) {
  // expected: iterate over all list elements
  for (int i = 0; i < list.size(); i++) {
    if (list.get(i).isEmpty()) {
      list.remove(i); // Noncompliant, next element is skipped
    }
  }
}
```

Compliant solution

```
void removeFrom(List<String> list) {
  list.removeIf(String::isEmpty); // Compliant
}
```

Noncompliant code example

If this is not possible due to side effects of the loop, replace the ascending loop with a descending loop. Descending loops are not affected by decrementing the element indices after the removed element, because they have already been iterated.

```
void removeFrom(List<String> list) {
  // expected: iterate over all list elements
  for (int i = 0; i < list.size(); i++) {
    if (list.get(i).isEmpty()) {
      list.remove(i); // Noncompliant, next element is skipped
    }
  }
}
```

Compliant solution

```
void removeFrom(List<String> list) {
    // expected: iterate over all list elements
    for (int i = list.size() - 1; i >= 0; i--) {
    if (list.get(i).isEmpty()) {
      list.remove(i); // Compliant, elements after removed one have already been iterated
    }
  }
}
```

Noncompliant code example

Another way to solve this issue is to adjust the loop counter after the call to `Collection.remove` to account for the index decrement.

```
void removeFrom(List<String> list) {
  // expected: iterate over all list elements
  for (int i = 0; i < list.size(); i++) {
    if (list.get(i).isEmpty()) {
      list.remove(i); // Noncompliant, next element is skipped
    }
  }
}
```

Compliant solution

This is not recommanded because it raises an issue with rule S127.

```
void removeFrom(List<String> list) {
  // expected: iterate over all list elements
  for (int i = 0; i < list.size(); i++) {
    if (list.get(i).isEmpty()) {
      list.remove(i); // Compliant due to counter adjust in next line
      i--; // Noncompliant with S127!
    }
  }
}
```

Exception testing via JUnit ExpectedException rule should not be mixed with other assertions (java:S5776)

Severidad: MAJOR

Resources:

- JUnit exception testing documentation

Root_cause:

When testing exception via `org.junit.rules.ExpectedException` any code after the raised exception will not be executed, so adding subsequent assertions is wrong and misleading. This rule raises an issue when an assertion is done after the "expect(…)" invocation, only the code throwing the expected exception should be after "expect(…)".

You should consider using org.junit.Assert.assertThrows instead, it’s available since JUnit 4.13 and it allows additional subsequent assertions.

Alternatively, you could use try-catch idiom for JUnit version < 4.13 or if your project does not support lambdas.

Noncompliant code example

```
@Rule
public ExpectedException thrown = ExpectedException.none();

@Test
public void test() throws IndexOutOfBoundsException {
  thrown.expect(IndexOutOfBoundsException.class); // Noncompliant
  Object o = get();
  // This test pass since execution will never get past this line.
  Assert.assertEquals(0, 1);
}

private Object get() {
  throw new IndexOutOfBoundsException();
}
```

Compliant solution

- For JUnit >= 4.13, use org.junit.Assert.assertThrows:

```
Assert.assertThrows(IndexOutOfBoundsException.class, () -> get());
// This test correctly fails.
Assert.assertEquals(0, 1);
```

- For JUnit < 4.13, use the try-catch idiom:

```
try {
  get();
  Assert.fail("Expected an IndexOutOfBoundsException to be thrown");
} catch (IndexOutOfBoundsException e) {}
Assert.assertEquals(0, 1); // Correctly fails.
```

Exception testing via JUnit @Test annotation should be avoided (java:S5777)

Severidad: MINOR

Resources:

- JUnit exception testing documentation

Root_cause:

When testing exception via `@Test` annotation, having additional assertions inside that test method can be problematic because any code after the raised exception will not be executed. It will prevent you to test the state of the program after the raised exception and, at worst, make you misleadingly think that it is executed.

You should consider moving any assertions into a separate test method where possible, or using org.junit.Assert.assertThrows instead.

Alternatively, you could use try-catch idiom for JUnit version < 4.13 or if your project does not support lambdas.

Noncompliant code example

```
@Test(expected = IndexOutOfBoundsException.class)
public void testShouldFail() {
  get();
  // This test pass since execution will never get past this line.
  Assert.assertEquals(0, 1);
}

private Object get() {
  throw new IndexOutOfBoundsException();
}
```

Compliant solution

- For JUnit >= 4.13, use org.junit.Assert.assertThrows:

```
// This test correctly fails.
@Test
public void testToString() {
    Object obj = get();
    Assert.assertThrows(IndexOutOfBoundsException.class, () -> obj.toString());
    Assert.assertEquals(0, 1);
}
```

- For JUnit < 4.13, use the try-catch idiom:

```
@Test
public void testShouldFail() {
    Object obj = get();
    try {
        obj.toString();
        Assert.fail("Expected an IndexOutOfBoundsException to be thrown");
    } catch (IndexOutOfBoundsException e) {}
    Assert.assertEquals(0, 1); // Correctly fails.
}
```

Only one method invocation is expected when testing runtime exceptions (java:S5778)

Severidad: MAJOR

Resources:

- JUnit exception testing documentation

Root_cause:

When verifying that code raises a runtime exception, a good practice is to avoid having multiple method calls inside the tested code, to be explicit about which method call is expected to raise the exception.

It increases the clarity of the test, and avoid incorrect testing when another method is actually raising the exception.

Noncompliant code example

```
@Test
public void testToString() {
  // Do you expect get() or toString() throwing the exception?
  org.junit.Assert.assertThrows(IndexOutOfBoundsException.class, () -> get().toString());
}

@Test
```

```
public void testToStringTryCatchIdiom() {
  try {
    // Do you expect get() or toString() throwing the exception?
    get().toString();
    Assert.fail("Expected an IndexOutOfBoundsException to be thrown");
  } catch (IndexOutOfBoundsException e) {
    // Test exception message...
  }
}
```

Compliant solution

```
@Test
public void testToString() {
   Object obj = get();
   Assert.assertThrows(IndexOutOfBoundsException.class, () -> obj.toString());
}

@Test
public void testToStringTryCatchIdiom() {
  Object obj = get();
  try {
    obj.toString();
    Assert.fail("Expected an IndexOutOfBoundsException to be thrown");
  } catch (IndexOutOfBoundsException e) {
    // Test exception message...
  }
}
```

Assertion methods should not be used within the try block of a try-catch catching an Error (java:S5779)

Severidad: CRITICAL

Root_cause:

Assertion methods are throwing a "`java.lang.AssertionError`". If this call is done within the try block of a try-catch cathing a similar error, you should make sure to test some properties of the exception. Otherwise, the assertion will never fail.

Noncompliant code example

```
@Test
public void should_throw_assertion_error() {
  try {
    throwAssertionError();
    Assert.fail("Expected an AssertionError!"); // Noncompliant, the AssertionError will be caught and the test will never fail.
  } catch (AssertionError e) {}
}

private void throwAssertionError() {
  throw new AssertionError("My assertion error");
}
```

Compliant solution

```
assertThrows(AssertionError.class, () -> throwAssertionError());

try {
   throwAssertionError();
   Assert.fail("Expected an AssertionError!"); // Compliant, we made sure to test that the correct error is raised
 } catch (AssertionError e) {
   Assert.assertThat(e.getMessage(), is("My assertion error"));
 }
```

Resources:

- JUnit 4 exception testing documentation

Empty lines should not be tested with regex MULTILINE flag (java:S5846)

Severidad: CRITICAL

Root_cause:

One way to test for empty lines is to use the regex "`^$`", which can be extremely handy when filtering out empty lines from collections of Strings, for instance. With regard to this, the Javadoc for Pattern (Line Terminators) states the following:

> By default, the regular expressions ^ and $ ignore line terminators and only match at the beginning and the end, respectively, of the entire input sequence. If `MULTILINE` mode is activated then ^ matches at the beginning of input and after any line terminator **except at the end of input**. When in `MULTILINE` mode $ matches just before a line terminator or the end of the input sequence.

As emphasized, ^ is not going to match at the end of an input, and the end of the input is necessarily included in the empty string, which might lead to completely missing empty lines, while it would be the initial reason for using such regex.

Therefore, when searching for empty lines using a multi-line regular expression, you should also check whether the string is empty.

This rule is raising an issue every time a pattern that can match the empty string is used with `MULTILINE` flag and without calling `isEmpty()` on the string.

Noncompliant code example

```
static final Pattern p = Pattern.compile("^$", Pattern.MULTILINE); // Noncompliant

// Alternatively
static final Pattern p = Pattern.compile("(?m)^$"); // Noncompliant


boolean containsEmptyLines(String str) {
    return p.matcher(str).find();
}

// ...
System.out.println(containsEmptyLines("a\n\nb")); // correctly prints 'true'
System.out.println(containsEmptyLines("")); // incorrectly prints 'false'
```

Compliant solution

```
static final Pattern p = Pattern.compile("^$", Pattern.MULTILINE);

boolean containsEmptyLines(String str) {
    return p.matcher(str).find() || str.isEmpty();
}

// ...
System.out.println(containsEmptyLines("a\n\nb")); // correctly prints 'true'
System.out.println(containsEmptyLines("")); // also correctly prints 'true'
```

Mocking all non-private methods of a class should be avoided (java:S5969)

Severidad: CRITICAL

Root_cause:

If you end up mocking every non-private method of a class in order to write tests, it is a strong sign that your test became too complex, or that you misunderstood the way you are supposed to use the mocking mechanism.

You should either refactor the test code into multiple units, or consider using the class itself, by either directly instantiating it, or creating a new one inheriting from it, with the expected behavior.

This rule reports an issue when every member of a given class are mocked.

Noncompliant code example

```
@Test
void test_requiring_MyClass() {
  MyClass myClassMock = mock(MyClass.class); // Noncompliant
  when(myClassMock.f()).thenReturn(1);
  when(myClassMock.g()).thenReturn(2);
  //...
}

abstract class MyClass {
  abstract int f();
  abstract int g();
}
```

Compliant solution

```
@Test
void test_requiring_MyClass() {
  MyClass myClass = new MyClassForTest();
  //...
}

class MyClassForTest extends MyClass {
  @Override
  int f() {
    return 1;
  }

  @Override
  int g() {
    return 2;
  }
}
```

or

```
@Test
void test_requiring_f() {
  MyClass myClassMock = mock(MyClass.class);
  when(myClassMock.f()).thenReturn(1);
  //...
}

@Test
void test_requiring_g() {
  MyClass myClassMock = mock(MyClass.class);
  when(myClassMock.g()).thenReturn(2);
  //...
}

abstract class MyClass {
  abstract int f();
  abstract int g();
}
```

Nullable injected fields and parameters should provide a default value (java:S6816)

Severidad: MAJOR

Root_cause:

The `@Value` annotation does not guarantee that the property is defined. Particularly if a field or parameter is annotated as nullable, it indicates that the developer assumes that the property may be undefined.

An undefined property may lead to runtime exceptions when the Spring framework tries to inject the autowired dependency during bean creation.

This rule raises an issue when a nullable field or parameter is annotated with `@Value` and no default value is provided.

How_to_fix:

Add a default value to the `@Value` annotation. A default value can be supplied by using the colon (`:`) operator. As the field is nullable, the default value should most likely be `#{null}`.

Noncompliant code example

```
@Nullable
@Value("${my.property}") // Noncompliant, no default value is provided, even though the field is nullable
private String myProperty;
```

Compliant solution

```
@Nullable
@Value("${my.property:#{null}}") // Compliant, a default value is provided
private String myProperty;
```

Introduction:

SpEL, the Spring Expression Languages allows developers fine-grained control over the values injected into fields and parameters. Using the `@Value` annotation, it is possible to inject values from sources such as system properties.

Resources:

Articles & blog posts

- Baeldung - [Using Spring @Value With Defaults]

Use of the "@Async" annotation on methods declared within a "@Configuration" class in Spring Boot (java:S6817)

Severidad: MAJOR

Root_cause:

`@Configuration` is a class-level annotation indicating that an object is a source of bean definitions. `@Configuration` classes declare beans through `@Bean`-annotated methods. Calls to `@Bean` methods on `@Configuration` classes can also be used to define inter-bean dependencies. The `@Bean` annotation indicates that a method instantiates, configures, and initializes a new object to be managed by the Spring IoC container.

Annotating a method of a bean with `@Async` will make it execute in a separate thread. In other words, the caller will not wait for the completion of the called method.

The `@Async` annotation is not supported on methods declared within a `@Configuration` class. This is because `@Async` methods are typically used for asynchronous processing, and they require certain infrastructure to be set up, which may not be available or appropriate in a `@Configuration` class.

How_to_fix:

Don't use `@Async` annotations on methods of `@Configuration` classes.

Noncompliant code example

```
@EnableAsync
@Configuration
public class MyConfiguration {

  @Async // Noncompliant - This is not allowed
  public void asyncMethod() {
    // ...
  }
}
```

Compliant solution

```
@EnableAsync
@Configuration
public class MyConfiguration {

  public void method() {
    // ...
  }
}
```

Resources:

Documentation

- Spring Framework - @Async
- Spring Framework - Using the @Configuration annotation
- Spring Framework - Basic Concepts: @Bean and @Configuration

Articles & blog posts

- Baeldung - How To Do @Async in Spring

Beans in "@Configuration" class should have different names (java:S6862)

Severidad: MAJOR

How_to_fix:

To address this issue, ensure each bean within a configuration has a distinct and meaningful name. Choose names that accurately represent the purpose or functionality of the bean.

Noncompliant code example

```
@Configuration
class Config {
  @Bean
  public User user() {
    return currentUser();
  }
  @Bean
  public User user(AuthService auth) { // Noncompliant
    return auth.user();
  }
}
```

Compliant solution

```
@Configuration
class Config {
  @Bean
  public User user() {
    return currentUser();
  }
  @Bean
  public User userFromAuth(AuthService auth) {
    return auth.user();
  }
}
```

Root_cause:

Naming conventions play a crucial role in maintaining code clarity and readability. The uniqueness of bean names in Spring configurations is vital to the clarity and readability of the code. When two beans share the same name within a configuration, it is not obvious to the reader which bean is being referred to. This leads to potential misunderstandings and errors.

Resources:

Documentation

- Spring IO - Basic concepts: @Bean and @Configuration
- Spring IO - Using the @Configuration annotation
- Spring IO - Using the @Bean annotation

Set appropriate Status Codes on HTTP responses (java:S6863)

Severidad: MAJOR

Root_cause:

The request handler function in a `Controller` should set the appropriate HTTP status code based on the operation's success or failure. This is done by returning a `Response` object with the appropriate status code.

If an exception is thrown during the execution of the handler, the status code should be in the range of 4xx or 5xx. Examples of such codes are `BAD_REQUEST`, `UNAUTHORIZED`, `FORBIDDEN`, `NOT_FOUND`, `INTERNAL_SERVER_ERROR`, `BAD_GATEWAY`, `SERVICE_UNAVAILABLE`, etc.

The status code should be 1xx, 2xx, or 3xx if no exception is thrown and the operation is considered successful. Such codes include `OK`, `CREATED`, `MOVED_PERMANENTLY`, `FOUND`, etc.

Resources:

Documentation

- Spring Java Documentation - HttpStatus
- Spring Java Documentation - ResponseEntity
- Spring Framework Documentation - ResponseEntity
- Spring Framework Documentation - Exception Handling

Standards

- IANA - Hypertext Transfer Protocol (HTTP) Status Code Registry

How_to_fix:

Noncompliant code example

```
@Controller
public class UserController {
    public ResponseEntity<User> getUserById(Long userId) {
        try {
            User user = userService.getUserById(userId);
            return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(user); // Noncompliant: Setting 500 for a successful operat
        } catch (Exception e) {
            return ResponseEntity.status(HttpStatus.OK).build(); // Noncompliant: Setting 200 for exception
        }
    }
}
```

Compliant solution

```
@Controller
public class UserController {
    public ResponseEntity<User> getUserById(Long userId) {
        try {
            User user = userService.getUserById(userId);
            return ResponseEntity.ok(user); // Compliant: Setting 200 for a successful operation
        } catch (Exception e) {
            return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build(); // Compliant: Setting 500 for exception
        }
    }
}
```

Public constants and fields initialized at declaration should be "static final" rather than merely "final" (java:S1170)

Severidad: MINOR

Root_cause:

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).

Only static class initializers should be used (java:S1171)

Severidad: MAJOR

How_to_fix:

Non-static initializers should be refactored into standard constructors or field initializers when possible.

In most cases, the use of constructors, overloaded constructors, or factory methods is preferable for initializing instance variables. These approaches provide more explicit and controlled initialization, separate concerns, allow for better error handling, and make the code easier to understand and maintain.

Noncompliant code example

```
class MyClass {
  private static final Map<String, String> MY_MAP = new HashMap<String, String>() {
    {
      put("a", "b");
    }
  }; // Noncompliant - HashMap should be extended only to add behavior, not for initialization
}
```

Compliant solution

Using static initialization block:

```
class MyClass {
  private static final Map<String, String> MY_MAP = new HashMap<>();

  static {
    MY_MAP.put("a", "b");  // Compliant
  }
}
```

or using constructor:

```
class MyClass {
  private static final Map<String, String> MY_MAP = new HashMap<>();

  public MyClass() {
    MY_MAP.put("a", "b");  // Compliant
  }
}
```

or using Java 9 `Map.of`:

```
class MyClass {
  private static final Map<String, String> MY_MAP = java.util.Map.of("a", "b");   // Compliant
}
```

or using Guava `ImmutableMap.of`:

```
class MyClass {
  private static final Map<String, String> MY_MAP = com.google.common.collect.ImmutableMap.of("a", "b");   // Compliant
}
```

Root_cause:

Non-static initializers, also known as instance initializers, are blocks of code within a class that are executed when an instance of the class is created. They are executed when an object of the class is created just before the constructor is called. Non-static initializers are useful when you want to perform some common initialization logic for all objects of a class. They allow you to initialize instance variables in a concise and centralized manner, without having to repeat the same initialization code in each constructor.

While non-static initializers may have some limited use cases, they are rarely used and can be confusing for most developers because they only run when new class instances are created.

Resources:

Articles & blog posts

- Static vs. Instance Initializer Block in Java

Unused method parameters should be removed (java:S1172)

Severidad: MAJOR

How_to_fix:

Having unused function parameters in your code can lead to confusion and misunderstanding of a developer’s intention. They reduce code readability and introduce the potential for errors. To avoid these problems, developers should remove unused parameters from function declarations.

Noncompliant code example

```
void doSomething(int a, int b) { // Noncompliant, "b" is unused
  compute(a);
}
```

Compliant solution

```
void doSomething(int a) {
  compute(a);
}
```

Examples of exceptions:

```
@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 used 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);
}
```

Resources:

- CERT, MSC12-C. - Detect and remove code that has no effect or is never executed

Root_cause:

A typical code smell known as unused function parameters refers to parameters declared in a function but not used anywhere within the function’s body. While this might seem harmless at first glance, it can lead to confusion and potential errors in your code. Disregarding the values passed to such parameters, the function’s behavior will be the same, but the programmer’s intention won’t be clearly expressed anymore. Therefore, removing function parameters that are not being utilized is considered best practice.

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.

"Object.finalize()" should remain protected (versus public) when overriding (java:S1174)

Severidad: CRITICAL

Root_cause:

The contract of the `Object.finalize()` method is clear: only the Garbage Collector is supposed to call this method.

Making this method public is misleading, because it implies that any caller can use it.

Noncompliant code example

```
public class MyClass {

  @Override
  public void finalize() {    // Noncompliant
    /* ... */
  }
}
```

Resources:

- CWE - CWE-583 - finalize() Method Declared Public
- CERT, MET12-J. - Do not use finalizers

Member variable visibility should be specified (java:S2039)

Severidad: MINOR

Root_cause:

Failing to explicitly declare the visibility of a member variable could result it in having a visibility you don’t expect, and potentially leave it open to unexpected modification by other classes.

The default access level modifier may be intentional; in that case, this rule can report false positives.

Noncompliant code example

```
class Ball {
  String color = "red";  // Noncompliant
}
enum A {
  B;
  int a;  // Noncompliant
}
```

Compliant solution

```
class Ball {
  private String color = "red";  // Compliant
}
enum A {
  B;
  private int a;  // Compliant
}
```

Exceptions

- Members with comments containing the word `modifier` are ignored, as it indicates the modifier is intentionally omitted.
- Members annotated with the `@VisibleForTesting` annotation are ignored, as it indicates that visibility has been purposely relaxed to make the code testable.

```
class Cone {
  @VisibleForTesting
  Logger logger; // Compliant
}
```

Methods of "Random" that return floating point values should not be used in random integer generation (java:S2140)

Severidad: MINOR

Root_cause:

Generating random floating point values to cast them into integers is inefficient. A random bounded integer value can be generated with a single proper method call. Use `nextInt` to make the code more efficient and the intent clearer.

Noncompliant code example

```
Random r = new Random();
int rand = (int) (r.nextDouble() * 50);  // Noncompliant way to get a pseudo-random value between 0 and 50
int rand2 = (int) r.nextFloat(); // Noncompliant; will always be 0;
```

Compliant solution

```
Random r = new Random();
int rand = r.nextInt(50);  // returns pseudo-random value between 0 and 50
int rand2 = 0;
```

Classes that don't define "hashCode()" should not be used in hashes (java:S2141)

Severidad: MAJOR

Root_cause:

Because `Object` implements `hashCode`, any Java class can be put into a hash structure. However, classes that define `equals(Object)` but not `hashCode()` aren’t truly hash-able because instances that are equivalent according to the `equals` method can return different hashes.

Noncompliant code example

```
public class Student {  // no hashCode() method; not hash-able
  // ...

  public boolean equals(Object o) {
    // ...
  }
}

public class School {
  private Map<Student, Integer> studentBody = // okay so far
          new HashTable<Student, Integer>(); // Noncompliant

  // ...
```

Compliant solution

```
public class Student {  // has hashCode() method; hash-able
  // ...

  public boolean equals(Object o) {
    // ...
  }
  public int hashCode() {
    // ...
  }
}

public class School {
  private Map<Student, Integer> studentBody = new HashTable<Student, Integer>();

  // ...
```

"InterruptedException" and "ThreadDeath" should not be ignored (java:S2142)

Severidad: MAJOR

Resources:

- CWE - CWE-391 - Unchecked Error Condition

Root_cause:

If an `InterruptedException` or a `ThreadDeath` error is not handled properly, the information that the thread was interrupted will be lost. Handling this exception means either to re-throw it or manually re-interrupt the current thread by calling `Thread.interrupt()`. Simply logging the exception is not sufficient and counts as ignoring it. Between the moment the exception is caught and handled, is the right time to perform cleanup operations on the method’s state, if needed.

What is the potential impact?

Failing to interrupt the thread (or to re-throw) risks delaying the thread shutdown and losing the information that the thread was interrupted - probably without finishing its task.

Noncompliant code example

```
public void run () {
  try {
    /*...*/
  } catch (InterruptedException e) { // Noncompliant; logging is not enough
    LOGGER.log(Level.WARN, "Interrupted!", e);
  }
}
```

Compliant solution

```
public void run () {
  try {
    /* ... */
  } catch (InterruptedException e) { // Compliant; the interrupted state is restored
    LOGGER.log(Level.WARN, "Interrupted!", e);
    /* Clean up whatever needs to be handled before interrupting  */
    Thread.currentThread().interrupt();
  }
}

public void run () {
  try {
    /* ... */
  } catch (ThreadDeath e) { // Compliant; the error is being re-thrown
    LOGGER.log(Level.WARN, "Interrupted!", e);
    /* Clean up whatever needs to be handled before re-throwing  */
    throw e;
  }
}
```

"java.time" classes should be used for dates and times (java:S2143)

Severidad: MAJOR

Root_cause:

The old, much-derided `Date` and `Calendar` classes have always been confusing and difficult to use properly, particularly in a multi-threaded context. `JodaTime` has long been a popular alternative, but now an even better option is built-in. Java 8’s JSR 310 implementation offers specific classes for:

Class	Use for
LocalDate	a date, without time of day, offset, or zone
LocalTime	the time of day, without date, offset, or zone
LocalDateTime	the date and time, without offset, or zone
OffsetDate	a date with an offset such as +02:00, without time of day, or zone
OffsetTime	the time of day with an offset such as +02:00, without date, or zone
OffsetDateTime	the date and time with an offset such as +02:00, without a zone
ZonedDateTime	the date and time with a time zone and offset
YearMonth	a year and month
MonthDay	month and day
Year/MonthOfDay/DayOfWeek/…	classes for the important fields
DateTimeFields	stores a map of field-value pairs which may be invalid
Calendrical	access to the low-level API
Period	a descriptive amount of time, such as "2 months and 3 days"

Noncompliant code example

```
Date now = new Date();  // Noncompliant
DateFormat df = new SimpleDateFormat("dd.MM.yyyy");
Calendar christmas  = Calendar.getInstance();  // Noncompliant
christmas.setTime(df.parse("25.12.2020"));
```

Compliant solution

```
LocalDate now = LocalDate.now();  // gets calendar date. no time component
LocalTime now2 = LocalTime.now(); // gets current time. no date component
LocalDate christmas = LocalDate.of(2020,12,25);
```

Unnecessary equality checks should not be made (java:S2159)

Severidad: MAJOR

Root_cause:

Comparisons of dissimilar types will always return false. The comparison and all its dependent code can simply be removed. This includes:

- comparing an object with null
- comparing an object with an unrelated primitive (E.G. a string with an int)
- comparing unrelated classes
- comparing an unrelated `class` and `interface`
- comparing unrelated `interface` types
- comparing an array to a non-array
- comparing two arrays

Specifically in the case of arrays, since arrays don’t override `Object.equals()`, calling `equals` on two arrays is the same as comparing their addresses. This means that `array1.equals(array2)` is equivalent to `array1==array2`.

However, some developers might expect `Array.equals(Object obj)` to do more than a simple memory address comparison, comparing for instance the size and content of the two arrays. Instead, the `==` operator or `Arrays.equals(array1, array2)` should always be used with arrays.

Noncompliant code example

```
interface KitchenTool { ... };
interface Plant {...}

public class Spatula implements KitchenTool { ... }
public class Tree implements Plant { ...}
//...

Spatula spatula = new Spatula();
KitchenTool tool = spatula;
KitchenTool [] tools = {tool};

Tree tree = new Tree();
Plant plant = tree;
Tree [] trees = {tree};


if (spatula.equals(tree)) { // Noncompliant; unrelated classes
  // ...
}
else if (spatula.equals(plant)) { // Noncompliant; unrelated class and interface
  // ...
}
else if (tool.equals(plant)) { // Noncompliant; unrelated interfaces
  // ...
}
else if (tool.equals(tools)) { // Noncompliant; array & non-array
  // ...
}
else if (trees.equals(tools)) { // Noncompliant; incompatible arrays
  // ...
}
else if (tree.equals(null)) { // Noncompliant
  // ...
}
```

Resources:

- CERT, EXP02-J. - Do not use the Object.equals() method to compare two arrays

---

Java parser failure (java:S2260)

Severidad: MAJOR

Root_cause:

When the Java parser fails, it is possible to record the failure as a violation on the file. This way, not only it is possible to track the number of files that do not parse but also to easily find out why they do not parse.

Null checks should not be used with "instanceof" (java:S4201)

Severidad: MINOR

Root_cause:

There’s no need to null test in conjunction with an `instanceof` test. `null` is not an `instanceof` anything, so a null check is redundant.

Noncompliant code example

```
if (x != null && x instanceof MyClass) { ... }  // Noncompliant

if (x == null || ! x instanceof MyClass) { ... } // Noncompliant
```

Compliant solution

```
if (x instanceof MyClass) { ... }

if (! x instanceof MyClass) { ... }
```

Nullness of parameters should be guaranteed (java:S4449)

Severidad: MAJOR

Root_cause:

When using null-related annotations at global scope level, for instance using `javax.annotation.ParametersAreNonnullByDefault` (from JSR-305) at package level, it means that all the parameters to all the methods included in the package will, or should, be considered Non-`null`. It is equivalent to annotating every parameter in every method with non-null annotations (such as `@Nonnull`).

The rule raises an issue every time a parameter could be `null` for a method invocation, where the method is annotated as forbidding null parameters.

Noncompliant code example

```
@javax.annotation.ParametersAreNonnullByDefault
class A {

  void foo() {
    bar(getValue()); // Noncompliant - method 'bar' do not expect 'null' values as parameter
  }

  void bar(Object o) { // 'o' is by contract expected never to be null
    // ...
  }

  @javax.annotation.CheckForNull
  abstract Object getValue();
}
```

Compliant solution

Two solutions are possible:

- The signature of the method is correct, and null check should be done prior to the call.
- The signature of the method is not coherent and should be annotated to allow null values being passed as parameter

```
@javax.annotation.ParametersAreNonnullByDefault
abstract class A {

  void foo() {
      Object o = getValue();
      if (o != null) {
        bar(o); // Compliant - 'o' can not be null
      }
  }

  void bar(Object o) {
    // ...
  }

  @javax.annotation.CheckForNull
  abstract Object getValue();
}
```

or

```
@javax.annotation.ParametersAreNonnullByDefault
abstract class A {

  void foo() {
    bar(getValue());
  }

  void bar(@javax.annotation.Nullable Object o) { // annotation was missing
    // ...
  }

  @javax.annotation.CheckForNull
  abstract Object getValue();
}
```

"@CheckForNull" or "@Nullable" should not be used on primitive types (java:S4682)

Severidad: MINOR

Root_cause:

By definition, primitive types are not Objects and so they can’t be `null`. Adding `@CheckForNull` or `@Nullable` on primitive types adds confusion and is useless.

This rule raises an issue when `@CheckForNull` or `@Nullable` is set on a method returning a primitive type: byte, short, int, long, float, double, boolean, char.

Noncompliant code example

```
@CheckForNull
boolean isFoo() {
 ...
}
```

Compliant solution

```
boolean isFoo() {
 ...
}
```

Avoid using boxed "Boolean" types directly in boolean expressions (java:S5411)

Severidad: MINOR

Root_cause:

When boxed type `java.lang.Boolean` is used as an expression to determine the control flow (as described in Java Language Specification §4.2.5 The boolean Type and boolean Values) it will throw a `NullPointerException` if the value is `null` (as defined in Java Language Specification §5.1.8 Unboxing Conversion).

It is safer to avoid such conversion altogether and handle the `null` value explicitly.

Note, however, that no issues will be raised for Booleans that have already been null-checked or are marked `@NonNull/@NotNull`.

Noncompliant code example

```
Boolean b = getBoolean();
if (b) {  // Noncompliant, it will throw NPE when b == null
  foo();
} else {
  bar();
}
```

Compliant solution

```
Boolean b = getBoolean();
if (Boolean.TRUE.equals(b)) {
  foo();
} else {
  bar();  // will be invoked for both b == false and b == null
}
```

```
Boolean b = getBoolean();
if(b != null){
  String test = b ? "test" : "";
}
```

Exceptions

The issue is not raised if the expression is annotated `@NonNull` / `@NotNull`. This is useful if a boxed type is an instantiation of a generic type parameter and cannot be avoided.

```
List<Boolean> list = new ArrayList<>();
list.add(true);
list.add(false);
list.forEach((@NonNull Boolean value) -> {
  // Compliant
  if(value) {
    System.out.println("yes");
  }
});

@NonNull Boolean someMethod() { /* ... */ }

// Compliant
if(someMethod()) { /* ... */ }

@NonNull Boolean boxedNonNull = Boolean.TRUE;

// Compliant
if(boxedNonNull) { /* ... */ }
```

Resources:

- Java Language Specification §5.1.8 Unboxing Conversion

JWT should be signed and verified with strong cipher algorithms (java:S5659)

Severidad: CRITICAL

Introduction:

This vulnerability allows forging of JSON Web Tokens to impersonate other users.

How_to_fix:

The following code contains examples of JWT encoding and decoding without a strong cipher algorithm.

Noncompliant code example

```
import com.auth0.jwt.JWT;

public void encode() {
    JWT.create()
        .withSubject(SUBJECT)
        .sign(Algorithm.none()); // Noncompliant
}
```

```
import com.auth0.jwt.JWT;

public void decode() {
    JWTVerifier verifier = JWT.require(Algorithm.none()) // Noncompliant
        .withSubject(LOGIN)
        .build();
}
```

Compliant solution

```
import com.auth0.jwt.JWT;

public void encode() {
    JWT.create()
        .withSubject(SUBJECT)
        .sign(Algorithm.HMAC256(SECRET_KEY));
}
```

```
import com.auth0.jwt.JWT;

public void decode() {
    JWTVerifier verifier = JWT.require(Algorithm.HMAC256(SECRET_KEY))
        .withSubject(LOGIN)
        .build();
}
```

How does this work?

Always sign your tokens

The foremost measure to enhance JWT security is to ensure that every JWT you issue is signed. Unsigned tokens are like open books that anyone can tamper with. Signing your JWTs ensures that any alterations to the tokens after they have been issued can be detected. Most JWT libraries support a signing function, and using it is usually as simple as providing a secret key when the token is created.

Choose a strong cipher algorithm

It is not enough to merely sign your tokens. You need to sign them with a strong cipher algorithm. Algorithms like HS256 (HMAC using SHA-256) are considered secure for most purposes. But for an additional layer of security, you could use an algorithm like RS256 (RSA Signature with SHA-256), which uses a private key for signing and a public key for verification. This way, even if someone gains access to the public key, they will not be able to forge tokens.

Verify the signature of your tokens

Resolving a vulnerability concerning the validation of JWT token signatures is mainly about incorporating a critical step into your process: validating the signature every time a token is decoded. Just having a signed token using a secure algorithm is not enough. If you are not validating signatures, they are not serving their purpose.

Every time your application receives a JWT, it needs to decode the token to extract the information contained within. It is during this decoding process that the signature of the JWT should also be checked.

To resolve the issue, follow these instructions:

1. Use framework-specific functions for signature verification: Most programming frameworks that support JWTs provide specific functions to not only decode a token but also validate its signature simultaneously. Make sure to use these functions when handling incoming tokens.
2. Handle invalid signatures appropriately: If a JWT's signature does not validate correctly, it means the token is not trustworthy, indicating potential tampering. The action to take when encountering an invalid token should be denying the request carrying it and logging the event for further investigation.
3. Incorporate signature validation in your tests: When you are writing tests for your application, include tests that check the signature validation functionality. This can help you catch any instances where signature verification might be unintentionally skipped or bypassed.

By following these practices, you can ensure the security of your application’s JWT handling process, making it resistant to attacks that rely on tampering with tokens. Validation of the signature needs to be an integral and non-negotiable part of your token handling process.

Going the extra mile

Securely store your secret keys

Ensure that your secret keys are stored securely. They should not be hard-coded into your application code or checked into your version control system. Instead, consider using environment variables, secure key management systems, or vault services.

Rotate your secret keys

Even with the strongest cipher algorithms, there is a risk that your secret keys may be compromised. Therefore, it is a good practice to periodically rotate your secret keys. By doing so, you limit the amount of time that an attacker can misuse a stolen key. When you rotate keys, be sure to allow a grace period where tokens signed with the old key are still accepted to prevent service disruptions.

Root_cause:

JSON Web Tokens (JWTs), a popular method of securely transmitting information between parties as a JSON object, can become a significant security risk when they are not properly signed with a robust cipher algorithm, left unsigned altogether, or if the signature is not verified. This vulnerability class allows malicious actors to craft fraudulent tokens, effectively impersonating user identities. In essence, the integrity of a JWT hinges on the strength and presence of its signature.

What is the potential impact?

When a JSON Web Token is not appropriately signed with a strong cipher algorithm or if the signature is not verified, it becomes a significant threat to data security and the privacy of user identities.

Impersonation of users

JWTs are commonly used to represent user authorization claims. They contain information about the user’s identity, user roles, and access rights. When these tokens are not securely signed, it allows an attacker to forge them. In essence, a weak or missing signature gives an attacker the power to craft a token that could impersonate any user. For instance, they could create a token for an administrator account, gaining access to high-level permissions and sensitive data.

Unauthorized data access

When a JWT is not securely signed, it can be tampered with by an attacker, and the integrity of the data it carries cannot be trusted. An attacker can manipulate the content of the token and grant themselves permissions they should not have, leading to unauthorized data access.

How_to_fix:

The following code contains examples of JWT encoding and decoding without a strong cipher algorithm.

Noncompliant code example

```
import io.jsonwebtoken.Jwts;

public void encode() {
    Jwts.builder()
        .setSubject(USER_LOGIN)
        .compact(); // Noncompliant
}
```

```
import io.jsonwebtoken.Jwts;

public void decode() {
    Jwts.parser()
        .setSigningKey(SECRET_KEY)
        .parse(token)
        .getBody(); // Noncompliant
}
```

Compliant solution

```
import io.jsonwebtoken.Jwts;

public void encode() {
    Jwts.builder()
        .setSubject(USER_LOGIN)
        .signWith(SignatureAlgorithm.HS256, SECRET_KEY)
        .compact();
}
```

When using `Jwts.parser()`, make sure to call `parseClaimsJws` instead of `parse` as it throws exceptions for invalid or missing signatures.

```
import io.jsonwebtoken.Jwts;

public void decode() {
    Jwts.parser()
        .setSigningKey(SECRET_KEY)
        .parseClaimsJws(token)
        .getBody();
}
```

How does this work?

Always sign your tokens

The foremost measure to enhance JWT security is to ensure that every JWT you issue is signed. Unsigned tokens are like open books that anyone can tamper with. Signing your JWTs ensures that any alterations to the tokens after they have been issued can be detected. Most JWT libraries support a signing function, and using it is usually as simple as providing a secret key when the token is created.

Choose a strong cipher algorithm

It is not enough to merely sign your tokens. You need to sign them with a strong cipher algorithm. Algorithms like HS256 (HMAC using SHA-256) are considered secure for most purposes. But for an additional layer of security, you could use an algorithm like RS256 (RSA Signature with SHA-256), which uses a private key for signing and a public key for verification. This way, even if someone gains access to the public key, they will not be able to forge tokens.

Verify the signature of your tokens

Resolving a vulnerability concerning the validation of JWT token signatures is mainly about incorporating a critical step into your process: validating the signature every time a token is decoded. Just having a signed token using a secure algorithm is not enough. If you are not validating signatures, they are not serving their purpose.

Every time your application receives a JWT, it needs to decode the token to extract the information contained within. It is during this decoding process that the signature of the JWT should also be checked.

To resolve the issue, follow these instructions:

1. Use framework-specific functions for signature verification: Most programming frameworks that support JWTs provide specific functions to not only decode a token but also validate its signature simultaneously. Make sure to use these functions when handling incoming tokens.
2. Handle invalid signatures appropriately: If a JWT’s signature does not validate correctly, it means the token is not trustworthy, indicating potential tampering. The action to take when encountering an invalid token should be denying the request carrying it and logging the event for further investigation.
3. Incorporate signature validation in your tests: When you are writing tests for your application, include tests that check the signature validation functionality. This can help you catch any instances where signature verification might be unintentionally skipped or bypassed.

By following these practices, you can ensure the security of your application’s JWT handling process, making it resistant to attacks that rely on tampering with tokens. Validation of the signature needs to be an integral and non-negotiable part of your token handling process.

Going the extra mile

Securely store your secret keys

Ensure that your secret keys are stored securely. They should not be hard-coded into your application code or checked into your version control system. Instead, consider using environment variables, secure key management systems, or vault services.

Rotate your secret keys

Even with the strongest cipher algorithms, there is a risk that your secret keys may be compromised. Therefore, it is a good practice to periodically rotate your secret keys. By doing so, you limit the amount of time that an attacker can misuse a stolen key. When you rotate keys, be sure to allow a grace period where tokens signed with the old key are still accepted to prevent service disruptions.

Resources:

Standards

- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- CWE - CWE-347 - Improper Verification of Cryptographic Signature

Reverse iteration should utilize reversed view (java:S6876)

Severidad: MAJOR

Resources:

Documentation

- Java Documentation - Interface SequencedCollection
- OpenJDK - JEP 431: Sequenced Collections
- Java Documentation - Creating Sequenced Collections, Sets, and Maps

Root_cause:

Java 21 introduces the new Sequenced Collections API, which is applicable to all collections with a defined sequence on their elements, such as `LinkedList`, `TreeSet`, and others (see JEP 431). For projects using Java 21 and onwards, this API should be utilized instead of workaround implementations that were necessary before Java 21.

This rule reports when a collection is iterated in reverse through explicit implementation or workarounds, instead of using the reversed view of the collection.

How_to_fix:

Replace the reported statement with a forward-iteration over the reversed view of the collection.

Noncompliant code example

```
void printLastToFirst(List<String> list) {
  for (var it = list.listIterator(list.size()); it.hasPrevious();) {
    var element = it.previous();
    System.out.println(element);
  }
}
```

Compliant solution

```
void printLastToFirst(List<String> list) {
  for (var element: list.reversed()) {
    System.out.println(element);
  }
}
```

Reverse view should be used instead of reverse copy in read-only cases (java:S6877)

Severidad: MAJOR

How_to_fix:

Remove `Collections.reverse(list);` and replace `list` with `list.reversed()` after.

Noncompliant code example

```
void foo() {
  var list = new ArrayList<String>();
  list.add("A");
  list.add("B");
  Collections.reverse(list); // Noncompliant
  for (var e : list) {
    // ...
  }
}
```

Compliant solution

```
void foo() {
  var list = new ArrayList<String>();
  list.add("A");
  list.add("B");
  for (var e : list.reversed()) {  // Compliant
    // ...
  }
}
```

Noncompliant code example

```
void foo(List<String> list) {
  var copy = new ArrayList<String>(list);
  Collections.reverse(copy); // Noncompliant
  for (var e : copy) {
    // ...
  }
}
```

Compliant solution

```
void foo(List<String> list) {
  for (var e : list.reversed()) {  // Compliant
    // ...
  }
}
```

Resources:

Documentation

- Java Documentation - Interface SequencedCollection
- OpenJDK - JEP 431: Sequenced Collections
- Java Documentation - Creating Sequenced Collections, Sets, and Maps

Root_cause:

Java 21 introduces the new Sequenced Collections API, which applies to all collections with a defined sequence on their elements, such as `LinkedList`, `TreeSet`, and others (see JEP 431). For projects using Java 21 and onwards, use this API instead of workaround implementations that were necessary before Java 21. One of the features of the new Sequenced Collections API is `SequencedCollection.reversed()` which returns a lightweight view of the original collection, in the reverse order.

This rule reports when reverse view would have been sufficient instead of a reverse copy of a sequenced collection created using a list constructor plus a `Collections.reverse(collection);` call.

If feasible, a view should be preferred over a copy because a view is a lightweight iterator without modification of the list itself.

Use record pattern instead of explicit field access (java:S6878)

Severidad: MINOR

Root_cause:

Java 21 enhances Pattern Matching, introduced in Java 16, with a *record pattern* that decomposes records into local variables. This form should be used when all fields of a record are accessed within a block for improved readability. Nested record patterns are also allowed and should be used when a record field is another record, and all its fields are accessed.

Exceptions

This rule does not apply when not all record fields are accessed. This prevents the creation of unused local variables in the decomposed record structure.

Resources:

- JEP 440: Record Patterns

How_to_fix:

Replace the instance check or simple pattern matching with a record pattern.

Noncompliant code example

This example uses pattern matching but not a record pattern, even though all fields of the record are accessed in the block.

```
record Point(Float x, Float y, Float z) {}

void print(Object obj) {
    if (obj instanceof Point p) { // Noncompliant, because all three fields x, y, z are accessed
        Float x = p.x;
        Float y = p.y();
        System.out.println(x + y + p.z);
    }
}
```

Compliant solution

The compliant example uses a record pattern to decompose the record structure.

```
record Point(Float x, Float y, Float z) {}

void print(Object obj) {
    if (obj instanceof Point(Float x, Float y, Float z)) { // Compliant
        System.out.println(x + y + z);
    }
}
```

Noncompliant code example

This example does not use pattern matching or a record pattern. Rule *S6201 - Pattern matching or "instanceOf" operator should be used* would report first. When fixed using simple pattern matching instead of a record pattern, this rule (S6878) will report.

```
void print(Object obj) {
    if (obj instanceof Point) { // Noncompliant
        Point p = (Point) obj;
        Float x = p.x;
        Float y = p.y();
        System.out.println(x + y + p.z);
    }
}
```

Compliant solution

The solution compliant with both rules, S6201 and S6878, uses pattern matching and decomposes the record structure using a record pattern.

```
void print(Object obj) {
    if (obj instanceof Point(Float x, Float y, Float z)) { // Compliant
        System.out.println(x + y + z);
    }
}
```

Noncompliant code example

This example is noncompliant because a nested record pattern could have been used.

```
record Plane(Point normal, Float d) {}

void print(Object obj) {
    // Noncompliant, because all field of "normal" are accessed
    if (obj instanceof Plane(Point normal, Float d)) {
        System.out.println(normal.x + normal.y + normal.z);
        System.out.println(d);
    }
}
```

Compliant solution

This is the same example using a nested record pattern.

```
void print(Object obj) {
    if (obj instanceof Plane(Point(Float x, Float y, Float z), Float d)) { // Compliant
        System.out.println(x + y + z);
        System.out.println(d);
    }
}
```

Compliant solution

This example uses `var` instead of replicating the field types in the record pattern, which is less verbose and keeps the code more readable, especially in the case of longer type names. Also, it uses variable names that do not match the original field names. The reason for this can be to avoid name collisions with fields or other local variables.

```
void print(Object obj) {
    if (obj instanceof Point(var px, var py, var pz)) { // Compliant
        System.out.println(px + py + pz);
    }
}
```

Compliant solution

This example is compliant without using a record pattern, as it does not access all fields.

```
void print(Object obj) {
    if (obj instanceof Point p) { // Compliant, because z is never accessed
```

```
        Float x = p.x;
        Float y = p.y();
        System.out.println(x + y);
    }
}
```

Unused labels should be removed (java:S1065)

Severidad: MAJOR

Resources:

- CERT, MSC12-C. - Detect and remove code that has no effect or is never executed

Root_cause:

If a label is declared but not used in the program, it can be considered as dead code and should therefore be removed.

This will improve maintainability as developers will not wonder what this label is used for.

Noncompliant code example

```
void foo() {
  outer: //label is not used.
  for(int i = 0; i<10; i++) {
    break;
  }
}
```

Compliant solution

```
void foo() {
  for(int i = 0; i<10; i++) {
    break;
  }
}
```

Mergeable "if" statements should be combined (java:S1066)

Severidad: MAJOR

Root_cause:

Nested code - blocks of code inside blocks of code - is eventually necessary, but increases complexity. This is why keeping the code as flat as possible, by avoiding unnecessary nesting, is considered a good practice.

Merging `if` statements when possible will decrease the nesting of the code and improve its readability.

Code like

```
if (condition1) {
  if (condition2) {             // Noncompliant
    /* ... */
  }
}
```

Will be more readable as

```
if (condition1 && condition2) { // Compliant
  /* ... */
}
```

How_to_fix:

If merging the conditions seems to result in a more complex code, extracting the condition or part of it in a named function or variable is a better approach to fix readability.

Noncompliant code example

```
if (file != null) {
  if (file.isFile() || file.isDirectory()) {  // Noncompliant
    /* ... */
  }
}
```

Compliant solution

```
if (file != null && isFileOrDirectory(file)) { // Compliant
  /* ... */
}

private static boolean isFileOrDirectory(File file) {
  return file.isFile() || file.isDirectory();
}
```

Expressions should not be too complex (java:S1067)

Severidad: CRITICAL

Root_cause:

The complexity of an expression is defined by the number of &&, `||` and `condition ? ifTrue : ifFalse` operators it contains.

A single expression’s complexity should not become too high to keep the code readable.

Noncompliant code example

With the default threshold value of 3:

```
if (((condition1 && condition2) || (condition3 && condition4)) && condition5) { ... }
```

Compliant solution

```
if ( (myFirstCondition() || mySecondCondition()) && myLastCondition()) { ... }
```

Exceptions

No issue is reported inside `equals` methods, because it is common to compare all the fields of a class for equality inside this kind of method.

Unused "private" fields should be removed (java:S1068)

Severidad: MAJOR

Root_cause:

If a `private` field is declared but not used locally, its limited visibility makes it dead code.

This is either a sign that some logic is missing or that the code should be cleaned.

Cleaning out dead code decreases the size of the maintained codebase, making it easier to understand and preventing bugs from being introduced.

```
public class MyClass {
  private int foo = 42; // Noncompliant: foo is unused and should be removed

  public int compute(int a) {
    return a * 42;
  }

}
```

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.

Exceptions

The rule admits 3 exceptions:

- Serialization ID fields

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 for 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;  // Compliant by exception
}
```

- Annotated fields and classes annotated with Lombok annotations

The unused field in this class will not be reported by the rule as it is annotated, except if annotation class `SomeAnnotation` is listed in the `ignoreAnnotations` parameter (see Parameters).

```
public class MyClass {
  @SomeAnnotation
  private int unused;  // Compliant by exception
}
```

- Fields from classes with native methods

The unused field in this class will not be reported by the rule as it might be used by native code.

```
public class MyClass {
  private int unused = 42;  // Compliant by exception
  private native static void doSomethingNative();
}
```

Overriding methods should do more than simply call the same method in the super class (java:S1185)

Severidad: MINOR

Root_cause:

Overriding a method just to call the same method from the super class without performing any other actions is useless and misleading. The only time this is justified is in `final` overriding methods, where the effect is to lock in the parent class behavior. This rule ignores such overrides of `equals`, `hashCode` and `toString`.

Noncompliant code example

```
public void doSomething() {
  super.doSomething();
}

@Override
public boolean isLegal(Action action) {
  return super.isLegal(action);
}
```

Compliant solution

```
@Override
public boolean isLegal(Action action) {         // Compliant - not simply forwarding the call
  return super.isLegal(new Action(/* ... */));
}

@Id
@Override
public int getId() {                            // Compliant - there is annotation different from @Override
  return super.getId();
}
```

Methods should not be empty (java:S1186)

Severidad: CRITICAL

How_to_fix:

Noncompliant code example

```
public void shouldNotBeEmpty() {  // Noncompliant - method is empty
}

public void notImplemented() {  // Noncompliant - method is empty
}

@Override
public void emptyOnPurpose() {  // Noncompliant - method is empty
}
```

Compliant solution

```
public void doSomething() {
  doSomething();
}

public void notImplemented() {
  throw new UnsupportedOperationException("notImplemented() cannot be performed because ...");
}

@Override
public void emptyOnPurpose() {
  // comment explaining why the method is empty
}
```

Root_cause:

An empty method is generally considered bad practice and can lead to confusion, readability, and maintenance issues. Empty methods bring no functionality and are misleading to others as they might think the method implementation fulfills a specific and identified requirement.

There are several reasons for a method not to have a body:

- It is an unintentional omission, and should be fixed to prevent an unexpected behavior in production.
- It is not yet, or never will be, supported. In this case an exception should be thrown.
- The method is an intentionally-blank override. In this case a nested comment should explain the reason for the blank override.

Exceptions

This does not raise an issue in the following cases:

- Non-public default (no-argument) constructors
- Public default (no-argument) constructors when there are other constructors in the class
- Empty methods in abstract classes
- Methods annotated with `@org.aspectj.lang.annotation.Pointcut()`

```
public abstract class Animal {
  void speak() {  // default implementation ignored
  }
}
```

Anonymous classes should not have too many lines (java:S1188)

Severidad: MAJOR

Root_cause:

Anonymous classes are a very convenient and compact way to inject a behavior without having to create a dedicated class. But those anonymous inner classes should be used only if the behavior to be injected can be defined in a few lines of code, otherwise the source code can quickly become unreadable.

Unnecessary boxing and unboxing should be avoided (java:S2153)

Severidad: MINOR

Root_cause:

Boxing is the process of putting a primitive value into a wrapper object, such as creating an `Integer` to hold an `int` value. Unboxing is the process of retrieving the primitive value from such an object. Since the original value is unchanged during boxing and unboxing, there is no point in doing either when not needed.

Instead, you should rely on Java’s implicit boxing/unboxing to convert from the primitive type to the wrapper type and vice versa, for better readability.

Noncompliant code example

```
public void examinePrimitiveInt(int a) {
  //...
}

public void examineBoxedInteger(Integer a) {
  // ...
}

public void func() {
  int primitiveInt = 0;
  Integer boxedInt = Integer.valueOf(0);
  double d = 1.0;
  int dIntValue = Double.valueOf(d).intValue(); // Noncompliant; should be replaced with a simple cast

  examinePrimitiveInt(boxedInt.intValue()); // Noncompliant; unnecessary unboxing
  examinePrimitiveInt(Integer.valueOf(primitiveInt));  // Noncompliant; boxed int will be auto-unboxed

  examineBoxedInteger(Integer.valueOf(primitiveInt)); // Noncompliant; unnecessary boxing
  examineBoxedInteger(boxedInt.intValue()); // Noncompliant; unboxed int will be autoboxed
}
```

Compliant solution

```
public void examinePrimitiveInt(int a) {
  //...
}

public void examineBoxedInteger(Integer a) {
  // ...
}

public void func() {
  int primitiveInt = 0;
  Integer boxedInt = Integer.valueOf(0);
  double d = 1.0;

  int dIntValue = (int) d;

  examinePrimitiveInt(primitiveInt);
  examinePrimitiveInt(boxedInt);

  examineBoxedInteger(primitiveInt);
  examineBoxedInteger(boxedInt);
}
```

Dissimilar primitive wrappers should not be used with the ternary operator without explicit casting (java:S2154)

Severidad: MAJOR

How_to_fix:

Noncompliant code example

Cast one of both operands to a common supertype (e.g., `Number`) to prevent auto-unboxing and, thus, type coercion.

```
Integer i = 123456789;
Float f = 1.0f;
Number n1 = condition ? i : f;  // Noncompliant, unexpected precision loss, n1 = 1.23456792E8
```

Compliant solution

```
Integer i = 123456789;
Float f = 1.0f;
Number n1 = condition ? (Number) i : f; // Compliant, cast to Number prevents unboxing
Number n2 = condition ? i : (Number) f; // Compliant, cast to Number prevents unboxing
```

Noncompliant code example

If type coercion was your intention, clarify this by casting the operand that would be coerced to the corresponding type explicitly.

```
Integer i = 123456789;
Float f = 1.0f;
Number n1 = condition ? i : f;  // Noncompliant, unexpected precision loss, n1 = 1.23456792E8
```

Compliant solution

```
Integer i = 123456789;
Float f = 1.0f;
Number n = condition ? (float) i : f; // Compliant, intentional type coercion with precision loss
```

Root_cause:

Using boxed values in a ternary operator does not simply return one operand or the other based on the condition. Instead, the values are unboxed and coerced to a common type, which can result in a loss of precision when converting one operand from `int` to `float` or from `long` to `double`.

While this behavior is expected for arithmetic operations, it may be unexpected for the ternary operator. To avoid confusion or unexpected behavior, cast to a compatible type explicitly.

Resources:

Documentation

- The Java Tutorials: Equality, Relational, and Conditional Operators
- The Java Tutorials: Autoboxing and Unboxing
- The Java® Language Specification Java SE 7 Edition: Chapter 5. Conversions and Promotions

Articles & blog posts

- GeeksforGeeks: Coercion in Java

"final" classes should not have "protected" members (java:S2156)

Severidad: MINOR

Root_cause:

The difference between `private` and `protected` visibility is that child classes can see and use `protected` members, but they cannot see `private` ones. Since a `final` class will have no children, marking the members of a `final` class `protected` is confusingly pointless.

Note that the `protected` members of a class can also be seen and used by other classes that are placed within the same package, this could lead to accidental, unintended access to otherwise private members.

Noncompliant code example

```
public final class MyFinalClass {

  protected String name = "Fred";  // Noncompliant
  protected void setName(String name) {  // Noncompliant
    // ...
  }
```

Compliant solution

```
public final class MyFinalClass {

  private String name = "Fred";
  public void setName(String name) {
    // ...
  }
```

Exceptions

Members annotated with `@VisibleForTesting` annotation are ignored, as it indicates that visibility has been purposely relaxed to make the code testable.

```
public final class MyFinalClass {
  @VisibleForTesting
  protected Logger logger; // Compliant

  @VisibleForTesting
  protected int calculateSomethingComplex(String input) { // Compliant
    // ...
  }
}
```

"Cloneables" should implement "clone" (java:S2157)

Severidad: CRITICAL

How_to_fix:

Consider the following example:

```
class Foo implements Cloneable { // Noncompliant, override `clone` method
  public int value;
}
```

Override the `clone` method in class `Foo`. By convention, it must call `super.clone()`. At this point, we know that:

- By behavioral contract, `Object.clone` will not throw a `CloneNotSupportedException`, because `Foo` implements `Cloneable`.
- The returned object is an instance of class `Foo`

We can narrow down the return type of `clone` to `Foo` and handle the `CloneNotSupportedException` inside the function instead of throwing it:

```
class Foo implements Cloneable { // Compliant

  public int value;

  @Override
  public Foo clone() {
    try {
      return (Foo) super.clone();
    } catch (CloneNotSupportedException e) {
      throw new AssertionError();
    }
  }
}
```

Be aware that `super.clone()` returns a one-by-one copy of the fields of the original instance. This means that in our example, the `Foo.value` field is not required to be explicitly copied in the overridden function.

If you require another copy behavior for some or all of the fields, for example, deep copy or certain invariants that need to be true for a field, these fields must be patched after `super.clone()`:

```
class Entity implements Cloneable {

  public int id; // unique per instance
  public List<Entity> children; // deep copy wanted

  @Override
  public Entity clone() {
    try {
      Entity copy = (Entity) super.clone();
      copy.id = System.identityHashCode(this);
      copy.children = children.stream().map(Entity::clone).toList();
      return copy;
    } catch (CloneNotSupportedException e) {
      throw new AssertionError();
    }
  }
}
```

Be aware that the `Cloneable` / `Object.clone` approach has several drawbacks. You might, therefore, also consider resorting to other solutions, such as a custom `copy` method or a copy constructor:

```
class Entity implements Cloneable {

  public int id; // unique per instance
  public List<Entity> children; // deep copy wanted

  Entity(Entity template) {
    id = System.identityHashCode(this);
    children = template.children.stream().map(Entity::new).toList();
  }
}
```

Root_cause:

`Cloneable` is a *marker interface* that defines the contract of the `Object.clone` method, which is to create a consistent copy of the instance. The `clone` method is not defined by the interface though, but by class `Objects`.

The general problem with marker interfaces is that their definitions cannot be enforced by the compiler because they have no own API. When a class implements `Cloneable` but does not override `Object.clone`, it is highly likely that it violates the contract for `Cloneable`.

Resources:

Documentation

- Interface Cloneable - Java™ Platform, Standard Edition 8 API Specification
- Object.clone - Java™ Platform, Standard Edition 8 API Specification

Introduction:

This rule raises an issue when a class implements the interface `java.lang.Cloneable`, but does not override the `Object.clone()` method.

"Object.wait(...)" and "Condition.await(...)" should be called inside a "while" loop (java:S2274)

Severidad: CRITICAL

Root_cause:

In a multithreaded environment, the `Object.wait(…)`, as well as `Condition.await(…)` and similar methods are used to pause the execution of a thread until the thread is awakened. A thread is typically awakened when it is notified, signaled, or interrupted, usually because of an event in another thread requiring some subsequent action by the waiting thread.

However, a thread may be awakened despite the desired condition not being met or the desired event not having happened. This is referred to as "spurious wakeups" and may be caused by underlying platform semantics. In other words, a thread may be awakened due to reasons that have nothing to do with the business logic. Hence, the assumption that the desired condition is met or the desired event occurred after a thread is awakened does not always hold.

According to the documentation of the Java `Condition` interface [1]:

> When waiting upon a `Condition`, a "spurious wakeup" is permitted to occur, in general, as a concession to the underlying platform semantics. This has little practical impact on most application programs as a Condition should always be waited upon in a loop, testing the state predicate that is being waited for. An implementation is free to remove the possibility of spurious wakeups but it is recommended that applications programmers always assume that they can occur and so always wait in a loop.

The same advice is also found for the `Object.wait(…)` method [2]:

> […] waits should always occur in loops, like this one:
>
> ```
> synchronized (obj) {
>   while (<condition does not hold>){
>     obj.wait(timeout);
>   }
>    ... // Perform action appropriate to condition
> }
> ```

How_to_fix:

Make sure that the desired condition is actually true after being awakened. This can be accomplished by calling the `wait` or `await` methods inside a loop that checks said condition.

Noncompliant code example

```
synchronized (obj) {
  if (!suitableCondition()){
    obj.wait(timeout); // Noncompliant, the thread can be awakened even though the condition is still false
  }
   ... // Perform some logic that is appropriate for when the condition is true
}
```

Compliant solution

```
synchronized (obj) {
  while (!suitableCondition()){
    obj.wait(timeout); // Compliant, the condition is checked in a loop, so the action below will only occur if the condition is true
  }
   ... // Perform some logic that is appropriate for when the condition is true
}
```

Resources:

1. Java SE 17 & JDK 17 - Condition
2. Java Platform SE 8 - Object#wait
3. CERT THI03-J. - Always invoke wait() and await() methods inside a loop

---

Printf-style format strings should not lead to unexpected behavior at runtime (java:S2275)

Severidad: BLOCKER

Root_cause:

Because `printf`-style format strings are interpreted at runtime, rather than validated by the Java compiler, they can contain errors that lead to unexpected behavior or runtime errors. This rule statically validates the good behavior of `printf`-style formats when calling the `format(...)` methods of `java.util.Formatter`, `java.lang.String`, `java.io.PrintStream`, `MessageFormat`, and `java.io.PrintWriter` classes and the `printf(...)` methods of `java.io.PrintStream` or `java.io.PrintWriter` classes.

Noncompliant code example

```
String.format("The value of my integer is %d", "Hello World");  // Noncompliant; an 'int' is expected rather than a String
String.format("Duke's Birthday year is %tX", c);  //Noncompliant; X is not a supported time conversion character
String.format("Display %0$d and then %d", 1);   //Noncompliant; arguments are numbered starting from 1
String.format("Not enough arguments %d and %d", 1);  //Noncompliant; the second argument is missing
```

```
String.format("%< is equals to %d", 2);   //Noncompliant; the argument index '<' refers to the previous format specifier but there isn't on

MessageFormat.format("Result {1}.", value); // Noncompliant; Not enough arguments. (first element is {0})
MessageFormat.format("Result {{0}.", value); // Noncompliant; Unbalanced number of curly brace (single curly braces should be escaped)
MessageFormat.format("Result ' {0}", value); // Noncompliant; Unbalanced number of quotes (single quote must be escaped)

java.util.logging.Logger logger;
logger.log(java.util.logging.Level.SEVERE, "Result {1}!", 14); // Noncompliant - Not enough arguments.

org.slf4j.Logger slf4jLog;
org.slf4j.Marker marker;

slf4jLog.debug(marker, "message {}"); // Noncompliant - Not enough arguments.

org.apache.logging.log4j.Logger log4jLog;
log4jLog.debug("message {}"); // Noncompliant - Not enough arguments.
```

Compliant solution

```
String.format("The value of my integer is %d", 3);
String.format("Duke's Birthday year is %tY", c);
String.format("Display %1$d and then %d", 1);
String.format("Not enough arguments %d and %d", 1, 2);
String.format("%d is equals to %<", 2);

MessageFormat.format("Result {0}.", value);
MessageFormat.format("Result {0} & {1}.", value, value);
MessageFormat.format("Result {0}.", myObject);

java.util.logging.Logger logger;
logger.log(java.util.logging.Level.SEVERE, "Result {1},{2}!", 14, 2);

org.slf4j.Logger slf4jLog;
org.slf4j.Marker marker;

slf4jLog.debug(marker, "message {}", 1);

org.apache.logging.log4j.Logger log4jLog;
log4jLog.debug("message {}", 1);
```

"wait(...)" should be used instead of "Thread.sleep(...)" when a lock is held (java:S2276)

Severidad: BLOCKER

How_to_fix:

Call `wait(…)` on the monitor object instead of using `Thread.sleep(…)`. While `wait(…)` is executed, the lock is temporarily released and hence other threads can run in the meantime.

Noncompliant code example

```
public void doSomething(){
  synchronized(monitor) {
    while(notReady()){
      Thread.sleep(200); // Noncompliant, any other thread synchronizing on monitor is blocked from running while the first thread sleeps.
    }
    process();
  }
  ...
}
```

Compliant solution

```
public void doSomething(){
  synchronized(monitor) {
    while(notReady()){
      monitor.wait(200); // Compliant, the current monitor is released.
    }
    process();
  }
  ...
}
```

Resources:

- CERT, LCK09-J. - Do not perform operations that can block while holding a lock

Root_cause:

In a multithreaded environment, a thread may need to wait for a particular condition to become true. One way of pausing execution in Java is `Thread.sleep(…)`.

If a thread that holds a lock calls `Thread.sleep(…)`, no other thread can acquire said lock. This can lead to performance and scalability issues, in the worst case leading to deadlocks.

Method parameters should be declared with base types (java:S3242)

Severidad: MINOR

Root_cause:

For maximum reusability, methods should accept parameters with as little specialization as possible. So unless specific features from a child class are required by a method, a type higher up the class hierarchy should be used instead.

Noncompliant code example

```
public void printSize(ArrayList<Object> list) {  // Collection can be used instead
    System.out.println(list.size());
}

public static void loop(List<Object> list) { // java.lang.Iterable can be used instead
   for (Object o : list) {
     o.toString();
  }
}
```

Compliant solution

```
public void printSize(Collection<?> list) {  // Collection can be used instead
    System.out.println(list.size());
}

public static void loop(Iterable<?> list) { // java.lang.Iterable can be used instead
   for (Object o : list) {
     o.toString();
  }
}
```

Exceptions

Parameters in non-public methods are not checked, because such methods are not intended to be generally reusable. `java.lang.String` parameters are excluded, because String is immutable and can not be always substituted for more generic type. Parameters used in any other context than method invocation or enhanced for loop are also excluded.

"this" should not be exposed from constructors (java:S3366)

Severidad: MAJOR

Resources:

- CERT, TSM01-J. - Do not let the this reference escape during object construction
- CERT, TSM03-J. - Do not publish partially initialized objects

Root_cause:

In single-threaded environments, the use of `this` in constructors is normal, and expected. But in multi-threaded environments, it could expose partially-constructed objects to other threads, and should be used with caution.

The classic example is a class with a `static` list of its instances. If the constructor stores `this` in the list, another thread could access the object before it’s fully-formed. Even when the storage of `this` is the last instruction in the constructor, there’s still a danger if the class is not `final`. In that case, the initialization of subclasses won’t be complete before `this` is exposed.

This rule raises an issue when `this` is assigned to any globally-visible object in a constructor, and when it is passed to the method of another object in a constructor

Noncompliant code example

```
public class Monument {
  public static final List<Monument> ALL_MONUMENTS = new ArrayList()<>;
  // ...

  public Monument(String location, ...) {
    ALL_MONUMENTS.add(this);  // Noncompliant; passed to a method of another object

    this.location = location;
    // ...
  }
}
```

Exceptions

This rule ignores instances of assigning `this` directly to a `static` field of the same class because that case is covered by S3010 .

"ActiveMQConnectionFactory" should not be vulnerable to malicious code deserialization (java:S5301)

Severidad: MINOR

Root_cause:

When the application does not implement controls over the JMS object types, its clients could be able to force the deserialization of arbitrary objects. This may lead to deserialization injection attacks.

What is the potential impact?

Attackers will be able to force the deserialization of arbitrary objects. This process will trigger the execution of magic unmarshalling methods on the object and its properties. With a specially crafted serialized object, the attackers can exploit those magic methods to achieve malicious purposes.

While the exact impact depends on the types available in the execution context at the time of deserialization, such an attack can generally lead to the execution of arbitrary code on the application server.

Application-specific attacks

By exploiting the behavior of some of the application-defined types and objects, the attacker could manage to affect the application’s business logic. The exact consequences will depend on the application’s nature:

- Payment bypass in an e-commerce application.
- Privilege escalation.
- Unauthorized users' data access.

Publicly-known exploitation

In some cases, depending on the library the application uses and their versions, there may exist publicly known deserialization attack payloads known as **gadget chains**. In general, they are designed to have severe consequences, such as:

- Arbitrary code execution
- Arbitrary file read or write
- Server-side request forgery

Those attacks are independent of the application’s own logic and from the types it specifies.

How_to_fix:

The following code example is vulnerable to a deserialization injection attack because it allows the deserialization of arbitrary types from JMS messages.

Noncompliant code example

```
ActiveMQConnectionFactory factory = new ActiveMQConnectionFactory("tcp://localhost:61616");
factory.setTrustAllPackages(true); // Noncompliant
```

Compliant solution

```
ActiveMQConnectionFactory factory = new ActiveMQConnectionFactory("tcp://localhost:61616");
factory.setTrustedPackages(Arrays.asList("org.mypackage1", "org.mypackage2"));
```

How does this work?

The noncompliant code example calls the `setTrustAllPackages` method that explicitly allows the deserialization of arbitrary types. On the contrary, the compliant code example, thanks to the `setTrustedPackages` method, defines a short list of classes allowed for the deserialization.

While defining a short list of trusted types is generally the state-of-the-art solution to avoid deserialization injection attacks, it is important to ensure that the allowed classes and packages can not be used to exploit the issue. In that case, a vulnerability would still be present.

Note that ActiveMQ, starting with version 5.12.2, forces developers to explicitly list packages that JMS messages can contain. This limits the risk of successful exploitation. In versions before that one, calling the `ActiveMQConnectionFactory` constructor without further configuration would leave the application at risk.

Resources:

Documentation

- Apache ActiveMQ Documentation - ObjectMessage
- CVE - CVE-2015-5254

Standards

- OWASP - Top 10 2021 - Category A8 - Software and Data Integrity Failures
- OWASP - Top 10 2017 - Category A8 - Insecure Deserialization
- CWE - CWE-502 - Deserialization of Untrusted Data

Introduction:

ActiveMQ can send/receive JMS Object messages (ObjectMessage in ActiveMQ context) to comply with JMS specifications. Internally, ActiveMQ relies on Java's serialization mechanism for the marshaling and unmarshalling of the messages' payload.

Applications should restrict the types that can be unserialized from JMS messages.

Encryption algorithms should be used with secure mode and padding scheme (java:S5542)

Severidad: CRITICAL

Introduction:

This vulnerability exposes encrypted data to a number of attacks whose goal is to recover the plaintext.

Resources:

Articles & blog posts

- Microsoft, Timing vulnerabilities with CBC-mode symmetric decryption using padding
- Wikipedia, Padding Oracle Attack
- Wikipedia, Chosen-Ciphertext Attack
- Wikipedia, Chosen-Plaintext Attack
- Wikipedia, Semantically Secure Cryptosystems
- Wikipedia, OAEP
- Wikipedia, Galois/Counter Mode

Standards

- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- CWE - CWE-327 - Use of a Broken or Risky Cryptographic Algorithm
- OWASP - Mobile AppSec Verification Standard - Cryptography Requirements
- OWASP - Mobile Top 10 2016 Category M5 - Insufficient Cryptography
- CWE - CWE-327 - Use of a Broken or Risky Cryptographic Algorithm
- CERT, MSC61-J. - Do not use insecure or weak cryptographic algorithms

Root_cause:

Encryption algorithms are essential for protecting sensitive information and ensuring secure communications in a variety of domains. They are used for several important reasons:

- Confidentiality, privacy, and intellectual property protection
- Security during transmission or on storage devices
- Data integrity, general trust, and authentication

When selecting encryption algorithms, tools, or combinations, you should also consider two things:

1. No encryption is unbreakable.
2. The strength of an encryption algorithm is usually measured by the effort required to crack it within a reasonable time frame.

For these reasons, as soon as cryptography is included in a project, it is important to choose encryption algorithms that are considered strong and secure by the cryptography community.

For AES, the weakest mode is ECB (Electronic Codebook). Repeated blocks of data are encrypted to the same value, making them easy to identify and reducing the difficulty of recovering the original cleartext.

Unauthenticated modes such as CBC (Cipher Block Chaining) may be used but are prone to attacks that manipulate the ciphertext. They must be used with caution.

For RSA, the weakest algorithms are either using it without padding or using the PKCS1v1.5 padding scheme.

What is the potential impact?

The cleartext of an encrypted message might be recoverable. Additionally, it might be possible to modify the cleartext of an encrypted message.

Below are some real-world scenarios that illustrate possible impacts of an attacker exploiting the vulnerability.

Theft of sensitive data

The encrypted message might contain data that is considered sensitive and should not be known to third parties.

By using a weak algorithm the likelihood that an attacker might be able to recover the cleartext drastically increases.

Additional attack surface

By modifying the cleartext of the encrypted message it might be possible for an attacker to trigger other vulnerabilities in the code. Encrypted values are often considered trusted, since under normal circumstances it would not be possible for a third party to modify them.

How_to_fix:

Noncompliant code example

Example with a symmetric cipher, AES:

```
import javax.crypto.Cipher;
import java.security.NoSuchAlgorithmException;
import javax.crypto.NoSuchPaddingException;

public static void main(String[] args) {
    try {
        Cipher.getInstance("AES/CBC/PKCS5Padding"); // Noncompliant
    } catch(NoSuchAlgorithmException|NoSuchPaddingException e) {
        // ...
    }
}
```

Example with an asymmetric cipher, RSA:

```
import javax.crypto.Cipher;
import java.security.NoSuchAlgorithmException;
import javax.crypto.NoSuchPaddingException;

public static void main(String[] args) {
    try {
        Cipher.getInstance("RSA/None/NoPadding"); // Noncompliant
    } catch(NoSuchAlgorithmException|NoSuchPaddingException e) {
        // ...
    }
}
```

Compliant solution

For the AES symmetric cipher, use the GCM mode:

```
import javax.crypto.Cipher;
import java.security.NoSuchAlgorithmException;
import javax.crypto.NoSuchPaddingException;

public static void main(String[] args) {
    try {
        Cipher.getInstance("AES/GCM/NoPadding");
    } catch(NoSuchAlgorithmException|NoSuchPaddingException e) {
        // ...
    }
}
```

For the RSA asymmetric cipher, use the Optimal Asymmetric Encryption Padding (OAEP):

```
import javax.crypto.Cipher;
import java.security.NoSuchAlgorithmException;
import javax.crypto.NoSuchPaddingException;

public static void main(String[] args) {
    try {
        Cipher.getInstance("RSA/ECB/OAEPWITHSHA-256ANDMGF1PADDING");
    } catch(NoSuchAlgorithmException|NoSuchPaddingException e) {
        // ...
    }
}
```

How does this work?

As a rule of thumb, use the cryptographic algorithms and mechanisms that are considered strong by the cryptographic community.

Appropriate choices are currently the following.

For AES: use authenticated encryption modes

The best-known authenticated encryption mode for AES is Galois/Counter mode (GCM).

GCM mode combines encryption with authentication and integrity checks using a cryptographic hash function and provides both confidentiality and authenticity of data.

Other similar modes are:

- CCM: `Counter with CBC-MAC`
- CWC: `Cipher Block Chaining with Message Authentication Code`
- EAX: `Encrypt-and-Authenticate`
- IAPM: `Integer Authenticated Parallelizable Mode`
- OCB: `Offset Codebook Mode`

It is also possible to use AES-CBC with HMAC for integrity checks. However, it is considered more straightforward to use AES-GCM directly instead.

For RSA: use the OAEP scheme

The Optimal Asymmetric Encryption Padding scheme (OAEP) adds randomness and a secure hash function that strengthens the regular inner workings of RSA.

Cipher algorithms should be robust (java:S5547)

Severidad: CRITICAL

Root_cause:

Encryption algorithms are essential for protecting sensitive information and ensuring secure communication in various domains. They are used for several important reasons:

- Confidentiality, privacy, and intellectual property protection
- Security during transmission or on storage devices
- Data integrity, general trust, and authentication

When selecting encryption algorithms, tools, or combinations, you should also consider two things:

1. No encryption is unbreakable.
2. The strength of an encryption algorithm is usually measured by the effort required to crack it within a reasonable time frame.

For these reasons, as soon as cryptography is included in a project, it is important to choose encryption algorithms that are considered strong and secure by the cryptography community.

What is the potential impact?

The cleartext of an encrypted message might be recoverable. Additionally, it might be possible to modify the cleartext of an encrypted message.

Below are some real-world scenarios that illustrate some impacts of an attacker exploiting the vulnerability.

Theft of sensitive data

The encrypted message might contain data that is considered sensitive and should not be known to third parties.

By using a weak algorithm the likelihood that an attacker might be able to recover the cleartext drastically increases.

Additional attack surface

By modifying the cleartext of the encrypted message it might be possible for an attacker to trigger other vulnerabilities in the code. Encrypted values are often considered trusted, since under normal circumstances it would not be possible for a third party to modify them.

How_to_fix:

The following code contains examples of algorithms that are not considered highly resistant to cryptanalysis and thus should be avoided.

Noncompliant code example

```
import javax.crypto.Cipher;
import java.security.NoSuchAlgorithmException;
import javax.crypto.NoSuchPaddingException;

public static void main(String[] args) {
    try {
        Cipher des = Cipher.getInstance("DES"); // Noncompliant
    } catch(NoSuchAlgorithmException|NoSuchPaddingException e) {
        // ...
    }
}
```

Compliant solution

```
import javax.crypto.Cipher;
import java.security.NoSuchAlgorithmException;
import javax.crypto.NoSuchPaddingException;

public static void main(String[] args) {
    try {
        Cipher aes = Cipher.getInstance("AES/GCM/NoPadding");
    } catch(NoSuchAlgorithmException|NoSuchPaddingException e) {
        // ...
    }
}
```

How does this work?

Use a secure algorithm

It is highly recommended to use an algorithm that is currently considered secure by the cryptographic community. A common choice for such an algorithm is the Advanced Encryption Standard (AES).

For block ciphers, it is not recommended to use algorithms with a block size that is smaller than 128 bits.

Resources:

Standards

- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- CWE - CWE-327 - Use of a Broken or Risky Cryptographic Algorithm
- STIG Viewer - Application Security and Development: V-222396 - The application must implement DoD-approved encryption to protect the confidentiality of remote access sessions.

Introduction:

This vulnerability makes it possible that the cleartext of the encrypted message might be recoverable without prior knowledge of the key.

Simple string literal should be used for single line strings (java:S5663)

Severidad: MINOR

Resources:

- JEP 378: Text Blocks
- Programmer’s Guide To Text Blocks, by Jim Laskey and Stuart Marks

Root_cause:

If a string fits on a single line, without concatenation and escaped newlines, you should probably continue to use a string literal.

Noncompliant code example

```
String question = """
              What's the point, really?""";
```

Compliant solution

```
String question = "What's the point, really?";
```

Whitespace for text block indent should be consistent (java:S5664)

Severidad: MAJOR

Resources:

- JEP 378: Text Blocks
- Programmer’s Guide To Text Blocks, by Jim Laskey and Stuart Marks

Root_cause:

Either use only spaces or only tabs for the indentation of a text block. Mixing white space will lead to a result with irregular indentation.

Noncompliant code example

```
String textBlock = """
        this is
<tab>text block!
        !!!!
      """;
```

Compliant solution

```
String textBlock = """
        this is
        text block!
        !!!!
      """;
```

Escape sequences should not be used in text blocks (java:S5665)

Severidad: MINOR

Resources:

- JEP 378: Text Blocks
- Programmer’s Guide To Text Blocks, by Jim Laskey and Stuart Marks

Root_cause:

The use of escape sequences is mostly unnecessary in text blocks.

Noncompliant code example

`\n` can be replaced by simply introducing the newline, `\"\"\"` it is sufficient to escape only the first qoute.

```
String textBlock = """
        \"\"\" this \nis
        text  block!
        !!!!
      """;
```

Compliant solution

```
String textBlock = """
        \""" this
        is
        text  block!
        !!!!
      """;
```

Vararg method arguments should not be confusing (java:S5669)

Severidad: MAJOR

Root_cause:

Passing single `null` or primitive array argument to the variable arity method may not work as expected. In the case of `null`, it is not passed as array with single element, but the argument itself is `null`. In the case of a primitive array, if the formal parameter is `Object...`, it is passed as a single element array. This may not be obvious to someone not familiar with such corner cases, and it is probably better to avoid such ambiguities by explicitly casting the argument to the desired type.

Noncompliant code example

```
class A {
  public static void main(String[] args) {
    vararg(null);  // Noncompliant, prints "null"
    int[] arr = {1,2,3};
    vararg(arr);  // Noncompliant, prints "length: 1"
  }

  static void vararg(Object... s) {
    if (s == null) {
      System.out.println("null");
    } else {
      System.out.println("length: " + s.length);
    }
  }
}
```

Compliant solution

```
class A {
  public static void main(String[] args) {
    vararg((Object) null); // prints 1
    Object[] arr = {1,2,3};
    vararg(arr); // prints 3
  }

  static void vararg(Object... s) {
    if (s == null) {
      System.out.println("null"); // not reached
    } else {
      System.out.println("length: " + s.length);
    }
  }
}
```

Only one method invocation is expected when testing checked exceptions (java:S5783)

Severidad: CRITICAL

Root_cause:

When verifying that code raises an exception, a good practice is to avoid having multiple method calls inside the tested code, to be explicit about what is exactly tested.

When two of the methods can raise the same **checked** exception, not respecting this good practice is a bug, since it is not possible to know what is really tested.

You should make sure that only one method can raise the expected checked exception in the tested code.

Noncompliant code example

```
@Test
public void testG() {
  // Do you expect g() or f() throwing the exception?
  assertThrows(IOException.class, () -> g(f(1)) ); // Noncompliant
}

@Test
public void testGTryCatchIdiom() {
  try { // Noncompliant
    g(f(1));
    Assert.fail("Expected an IOException to be thrown");
  } catch (IOException e) {
    // Test exception message...
  }
}
```

```
int f(int x) throws IOException {
  // ...
}

int g(int x) throws IOException {
  // ...
}
```

Compliant solution

```
@Test
public void testG() {
  int y = f(1);
  // It is explicit that we expect an exception from g() and not f()
  assertThrows(IOException.class, () -> g(y) );
}

@Test
public void testGTryCatchIdiom() {
  int y = f(1);
  try {
    g(y);
    Assert.fail("Expected an IOException to be thrown");
  } catch (IOException e) {
    // Test exception message...
  }
}
```

JUnit5 test classes and methods should have default package visibility (java:S5786)

Severidad: INFO

How_to_fix:

You can simply change the visibility by removing the `public` or `protected` keywords.

Noncompliant code example

```
import org.junit.jupiter.api.Test;

public class MyClassTest { // Noncompliant - modifier can be removed
  @Test
  protected void test() { // Noncompliant - modifier can be removed
    // ...
  }
}
```

Compliant solution

```
import org.junit.jupiter.api.Test;

class MyClassTest {
  @Test
  void test() {
    // ...
  }
}
```

Resources:

Documentation

- JUnit5 User Guide: Test Classes and Methods

Root_cause:

JUnit5 is more tolerant regarding the visibility of test classes and methods than JUnit4, which required everything to be `public`. Test classes and methods can have any visibility except `private`. It is however recommended to use the default package visibility to improve readability.

> Test classes, test methods, and lifecycle methods are not required to be `public`, but they must not be `private`.
>
> It is generally recommended to omit the public modifier for test classes, test methods, and lifecycle methods unless there is a technical reason for doing so – for example, when a test class is extended by a test class in another package. Another technical reason for making classes and methods public is to simplify testing on the module path when using the Java Module System.

— JUnit5 User Guide

What is the potential impact?

The code will be non-conventional and readability can be slightly affected.

Exceptions

This rule does not raise an issue when the visibility is set to `private`, because `private` test methods and classes are systematically ignored by JUnit5, without a proper warning. In this case, there is also an impact on reliability and so it is handled by the rule S5810.

Introduction:

JUnit5 test classes and methods should generally have package visibility. To fix this issue, change their visibility to the default package visibility.

Case insensitive string comparisons should be made without intermediate upper or lower casing (java:S1157)

Severidad: MINOR

Root_cause:

Using `toLowerCase()` or `toUpperCase()` to make case insensitive comparisons is inefficient because it requires the creation of temporary, intermediate `String` objects.

Noncompliant code example

```
private void compareStrings(String foo, String bar){
    boolean result1 = foo.toUpperCase().equals(bar);               // Noncompliant
    boolean result2 = foo.equals(bar.toUpperCase());               // Noncompliant
    boolean result3 = foo.toLowerCase().equals(bar.toLowerCase()); // Noncompliant
}
```

Compliant solution

```
private void compareStrings(String foo, String bar){
    boolean result1 = foo.equalsIgnoreCase(bar);                    // Compliant
}
```

Exceptions

No issue will be raised when a locale is specified because the result could be different from `equalsIgnoreCase()`. (e.g.: using the Turkish locale)

```
private void compareStrings(String foo, String bar, java.util.Locale locale){
    boolean result1 = foo.toUpperCase(locale).equals(bar);         // Compliant
}
```

Primitive wrappers should not be instantiated only for "toString" or "compareTo" calls (java:S1158)

Severidad: MINOR

Root_cause:

Creating temporary primitive wrapper objects only for `String` conversion or the use of the `compareTo()` method is inefficient.

Instead, the static `toString()` or `compare()` method of the primitive wrapper class should be used.

Noncompliant code example

```
private int isZero(int value){
    return Integer.valueOf(value).compareTo(0); // Noncompliant
}
private String convert(int value){
    return Integer.valueOf(value).toString(); // Noncompliant
}
```

Compliant solution

```
private int isZero(int value){
    return Integer.compare(value, 0); // Compliant
}
private String convert(int value){
    return Integer.toString(value); // Compliant
}
```

Throwable and Error should not be caught (java:S1181)

Severidad: MAJOR

Resources:

- CWE - CWE-396 - Declaration of Catch for Generic Exception
- CERT, ERR08-J. - Do not catch NullPointerException or any of its ancestors

Root_cause:

`Throwable` is the superclass of all errors and exceptions in Java. `Error` is the superclass of all errors, which are not meant to be caught by applications.

Catching either `Throwable` or `Error` will also catch `OutOfMemoryError` and `InternalError`, from which an application should not attempt to recover.

Noncompliant code example

```
try { /* ... */ } catch (Throwable t) { /* ... */ }
try { /* ... */ } catch (Error e) { /* ... */ }
```

Compliant solution

```
try { /* ... */ } catch (RuntimeException e) { /* ... */ }
try { /* ... */ } catch (MyException e) { /* ... */ }
```

Classes that override "clone" should be "Cloneable" and call "super.clone()" (java:S1182)

Severidad: MINOR

Resources:

- CWE - CWE-580 - clone() Method Without super.clone()
- CERT, MET53-J. - Ensure that the clone() method calls super.clone()

Root_cause:

`Cloneable` is the marker `Interface` that indicates that `clone()` may be called on an object. Overriding `clone()` without implementing `Cloneable` can be helpful if you want to control how subclasses clone themselves, but otherwise, it’s probably a mistake.

The usual convention for `Object.clone()` according to Oracle’s Javadoc is:

1. `x.clone() != x`
2. `x.clone().getClass() == x.getClass()`
3. `x.clone().equals(x)`

Obtaining the object that will be returned by calling `super.clone()` helps to satisfy those invariants:

1. `super.clone()` returns a new object instance
2. `super.clone()` returns an object of the same type as the one `clone()` was called on
3. `Object.clone()` performs a shallow copy of the object’s state.

How_to_fix:

Ensure that the `clone()` method calls `super.clone()` and implement `Cloneable` in the class or remove the clone method.

Noncompliant code example

```
class BaseClass {  // Noncompliant - should implement Cloneable
  @Override
  public Object clone() throws CloneNotSupportedException {    // Noncompliant - should return the super.clone() instance
    return new BaseClass();
  }
}

class DerivedClass extends BaseClass implements Cloneable {
  /* Does not override clone() */

  public void sayHello() {
    System.out.println("Hello, world!");
  }
}

class Application {
  public static void main(String[] args) throws Exception {
    DerivedClass instance = new DerivedClass();
    ((DerivedClass) instance.clone()).sayHello();              // Throws a ClassCastException because invariant #2 is violated
  }
}
```

Compliant solution

```
class BaseClass implements Cloneable {
  @Override
  public Object clone() throws CloneNotSupportedException {    // Compliant
    return super.clone();
  }
}

class DerivedClass extends BaseClass implements Cloneable {
  /* Does not override clone() */

  public void sayHello() {
    System.out.println("Hello, world!");
  }
}

class Application {
  public static void main(String[] args) throws Exception {
    DerivedClass instance = new DerivedClass();
    ((DerivedClass) instance.clone()).sayHello();              // Displays "Hello, world!" as expected. Invariant #2 is satisfied
  }
}
```

"Double.longBitsToDouble" should take "long" as argument (java:S2127)

Severidad: MAJOR

Root_cause:

`Double.longBitsToDouble` converts the bit pattern into its corresponding floating-point representation. The method expects a 64-bit long argument to interpret the bits as a double value correctly.

When the argument is a smaller data type, the cast to `long` may lead to a different value than expected due to the interpretation of the most significant bit, which, in turn, results in `Double.longBitsToDouble` returning an incorrect value.

Noncompliant code example

```
int i = 0x80003800;
Double.longBitsToDouble(i);   // Noncompliant - NaN
```

Compliant solution

```
long i = 0x80003800L;
Double.longBitsToDouble(i);   // Compliant - 1.0610049784E-314
```

Resources:

Documentation

- Oracle Java SE - Double.doubleToLongBits

Articles & blog posts

- Wikipedia - Double Precision floating point format
- Wikipedia - Single Precision floating point format

"runFinalizersOnExit" should not be called (java:S2151)

Severidad: CRITICAL

Resources:

- CERT, MET12-J. - Do not use finalizers

Root_cause:

Enabling `runFinalizersOnExit` is unsafe as it might result in erratic behavior and deadlocks on application exit.

Indeed, finalizers might be force-called on live objects while other threads are concurrently manipulating them.

Instead, if you want to execute something when the virtual machine begins its shutdown sequence, you should attach a shutdown hook.

Noncompliant code example

```
public static void main(String [] args) {
  System.runFinalizersOnExit(true);  // Noncompliant
}

protected void finalize(){
  doShutdownOperations();
}
```

Compliant solution

```
public static void main(String [] args) {
  Thread myThread = new Thread( () -> { doShutdownOperations(); });
  Runtime.getRuntime().addShutdownHook(myThread);
}
```

Using pseudorandom number generators (PRNGs) is security-sensitive (java:S2245)

Severidad: CRITICAL

How_to_fix:

Recommended Secure Coding Practices

- Use a cryptographically secure pseudo random number generator (CSPRNG) like "java.security.SecureRandom" in place of a non-cryptographic PRNG.
- Use the generated random values only once.
- You should not expose the generated random value. If you have to store it, make sure that the database or file is secure.

Compliant Solution

```
SecureRandom random = new SecureRandom();
byte bytes[] = new byte[20];
random.nextBytes(bytes);
```

See

- OWASP - Secure Random Number Generation Cheat Sheet
- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP - Mobile AppSec Verification Standard - Cryptography Requirements
- OWASP - Mobile Top 10 2016 Category M5 - Insufficient Cryptography
- CWE - CWE-338 - Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)
- CWE - CWE-330 - Use of Insufficiently Random Values
- CWE - CWE-326 - Inadequate Encryption Strength
- CWE - CWE-1241 - Use of Predictable Algorithm in Random Number Generator
- CERT, MSC02-J. - Generate strong random numbers
- Derived from FindSecBugs rule Predictable Pseudo Random Number Generator

Root_cause:

PRNGs are algorithms that produce sequences of numbers that only approximate true randomness. While they are suitable for applications like simulations or modeling, they are not appropriate for security-sensitive contexts because their outputs can be predictable if the internal state is known.

In contrast, cryptographically secure pseudorandom number generators (CSPRNGs) are designed to be secure against prediction attacks. CSPRNGs use cryptographic algorithms to ensure that the generated sequences are not only random but also unpredictable, even if part of the sequence or the internal state becomes known. This unpredictability is crucial for security-related tasks such as generating encryption keys, tokens, or any other values that must remain confidential and resistant to guessing attacks.

For example, the use of non-cryptographic PRNGs has led to vulnerabilities such as:

- CVE-2013-6386
- CVE-2006-3419
- CVE-2008-4102

When software generates predictable values in a context requiring unpredictability, it may be possible for an attacker to guess the next value that will be generated, and use this guess to impersonate another user or access sensitive information. Therefore, it is critical to use CSPRNGs in any security-sensitive application to ensure the robustness and security of the system.

As the `java.util.Random` class relies on a non-cryptographic pseudorandom number generator, this class and relating `java.lang.Math.random()` method should not be used for security-critical applications or for protecting sensitive data. In such context, the `java.security.SecureRandom` class which relies on a CSPRNG should be used in place.

Default:

PRNGs are algorithms that produce sequences of numbers that only approximate true randomness. While they are suitable for applications like simulations or modeling, they are not appropriate for security-sensitive contexts because their outputs can be predictable if the internal state is known.

In contrast, cryptographically secure pseudorandom number generators (CSPRNGs) are designed to be secure against prediction attacks. CSPRNGs use cryptographic algorithms to ensure that the generated sequences are not only random but also unpredictable, even if part of the sequence or the internal state becomes known. This unpredictability is crucial for security-related tasks such as generating encryption keys, tokens, or any other values that must remain confidential and resistant to guessing attacks.

For example, the use of non-cryptographic PRNGs has led to vulnerabilities such as:

- CVE-2013-6386
- CVE-2006-3419
- CVE-2008-4102

When software generates predictable values in a context requiring unpredictability, it may be possible for an attacker to guess the next value that will be generated, and use this guess to impersonate another user or access sensitive information. Therefore, it is critical to use CSPRNGs in any security-sensitive application to ensure the robustness and security of the system.

As the `java.util.Random` class relies on a non-cryptographic pseudorandom number generator, this class and relating `java.lang.Math.random()` method should not be used for security-critical applications or for protecting sensitive data. In such context, the `java.security.SecureRandom` class which relies on a CSPRNG should be used in place.

Ask Yourself Whether

- the code using the generated value requires it to be unpredictable. It is the case for all encryption mechanisms or when a secret value, such as a password, is hashed.
- the function you use is a non-cryptographic PRNG.
- the generated value is used multiple times.
- an attacker can access the generated value.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- Use a cryptographically secure pseudo random number generator (CSPRNG) like "java.security.SecureRandom" in place of a non-cryptographic PRNG.
- Use the generated random values only once.
- You should not expose the generated random value. If you have to store it, make sure that the database or file is secure.

Sensitive Code Example

```
Random random = new Random(); // Sensitive use of Random
byte bytes[] = new byte[20];
random.nextBytes(bytes); // Check if bytes is used for hashing, encryption, etc...
```

Compliant Solution

```
SecureRandom random = new SecureRandom();
byte bytes[] = new byte[20];
random.nextBytes(bytes);
```

See

- OWASP - Secure Random Number Generation Cheat Sheet
- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP - Mobile AppSec Verification Standard - Cryptography Requirements
- OWASP - Mobile Top 10 2016 Category M5 - Insufficient Cryptography
- CWE - CWE-338 - Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)

- CWE - CWE-330 - Use of Insufficiently Random Values
- CWE - CWE-326 - Inadequate Encryption Strength
- CWE - CWE-1241 - Use of Predictable Algorithm in Random Number Generator
- CERT, MSC02-J. - Generate strong random numbers
- Derived from FindSecBugs rule Predictable Pseudo Random Number Generator

Assess_the_problem:

Ask Yourself Whether

- the code using the generated value requires it to be unpredictable. It is the case for all encryption mechanisms or when a secret value, such as a password, is hashed.
- the function you use is a non-cryptographic PRNG.
- the generated value is used multiple times.
- an attacker can access the generated value.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

```
Random random = new Random(); // Sensitive use of Random
byte bytes[] = new byte[20];
random.nextBytes(bytes); // Check if bytes is used for hashing, encryption, etc...
```

"Iterator.next()" methods should throw "NoSuchElementException" (java:S2272)

Severidad: MINOR

Root_cause:

The `java.util.Iterator.next()` method must throw a `NoSuchElementException` when there are no more elements in the iteration. Any other behavior is non-compliant with the API contract and may cause unexpected behavior for users.

Noncompliant code example

```
public class MyIterator implements Iterator<String> {
  public String next() {
    if (!hasNext()) {
      return null;
    }
    // ...
  }
}
```

Compliant solution

```
public class MyIterator implements Iterator<String> {
  public String next() {
    if (!hasNext()) {
      throw new NoSuchElementException();
    }
    // ...
  }
}
```

Resources:

Documentation

- Java SE 7 API Specification: Iterator

"Object.wait()", "Object.notify()" and "Object.notifyAll()" should only be called from synchronized code (java:S2273)

Severidad: MAJOR

How_to_fix:

To become the owner of an object's monitor Java provides the `synchronized` keyword. In other words, calling `Object.wait(…)`, `Object.notify()` and `Object.notifyAll()` on a given object should only be done from code synchronized on the same object.

For example, the call to `someObject.wait(…)` should be wrapped in a `synchronized(someObject){ … }` block. If `wait` or `notify` are invoked on `this`, then the entire method can be marked as `synchronized`.

Noncompliant code example

```
private void performSomeAction(Object syncValue) {
  while (!suitableCondition()){
    syncValue.wait(); // Noncompliant, not being inside a `synchronized` block, this will raise an IllegalMonitorStateException
  }
  ... // Perform some action
}
```

Compliant solution

```
private void performSomeAction(Object syncValue) {
  synchronized(syncValue) {
    while (!suitableCondition()){
      syncValue.wait(); // Compliant, the `synchronized` block guarantees ownership of syncValue's monitor
    }
    ... // Perform some action
  }
}
```

References

- Java Documentation - Synchronized methods
- Java Documentation - java.lang.Object class and its methods

Root_cause:

The `Object.wait(…)`, `Object.notify()` and `Object.notifyAll()` methods are used in multithreaded environments to coordinate interdependent tasks that are performed by different threads. These methods are not thread-safe and by contract, they require the invoking `Thread` to own the object’s monitor. If a thread invokes one of these methods without owning the object’s monitor an `IllegalMonitorStateException` is thrown.

Classes should not access their own subclasses during class initialization (java:S2390)

Severidad: CRITICAL

Root_cause:

Referencing a static member of a subclass from its parent during class initialization, makes the code more fragile and prone to future bugs. The execution of the program will rely heavily on the order of initialization of classes and their static members.

What is the potential impact?

This could create what is known as an "initialization cycle", or even a deadlock in some extreme cases. Additionally, if the order of the static class members is changed, the behavior of the program might change. These issues can be very hard to diagnose so it is highly recommended to avoid creating this kind of dependencies.

Noncompliant code example

```
class Parent {
  static int field1 = Child.method(); // Noncompliant
  static int field2 = 42;

  public static void main(String[] args) {
    System.out.println(Parent.field1); // will display "0" instead of "42"
  }
}

class Child extends Parent {
  static int method() {
    return Parent.field2;
  }
}
```

Resources:

- CERT - DCL00-J. Prevent class initialization cycles
- Section 12.4: Initialization of Classes and Interfaces - Java Language Specification

---

Static non-final field names should comply with a naming convention (java:S3008)

Severidad: MINOR

Resources:

- O'Reilly - Java 8 in pocket - Naming Conventions
- Educative - Naming conventions in Java

Root_cause:

The Java Language Specification defines a set of rules called naming conventions that apply to Java programs. These conventions provide recommendations for naming packages, classes, methods, and variables.

By following shared naming conventions, teams can collaborate more efficiently.

This rule checks that static non-final field names match a provided regular expression.

Noncompliant code example

The default regular expression applied by the rule is `^[a-z][a-zA-Z0-9]*$`:

```
public class MyClass {
   private static String foo_bar; // Noncompliant
}
```

Compliant solution

```
public class MyClass {
   private static String fooBar;
}
```

Format strings should be used correctly (java:S3457)

Severidad: MAJOR

Root_cause:

A `printf-`-style format string is a string that contains placeholders, usually represented by special characters such as "%s" or "{}", depending on the technology in use. These placeholders are replaced by values when the string is printed or logged.

Because `printf`-style format strings are interpreted at runtime, rather than validated by the compiler, they can contain errors that result in the wrong strings being created.

This rule checks whether every format string specifier can be correctly matched with one of the additional arguments when calling the following methods:

- `java.lang.String#format`
- `java.util.Formatter#format`
- `java.io.PrintStream#format`
- `java.text.MessageFormat#format`
- `java.io.PrintWriter#format`
- `java.io.PrintStream#printf`
- `java.io.PrintWriter#printf`
- `java.lang.String#formatted` (since Java 15)
- logging methods of `org.slf4j.Logger`, `java.util.logging.Logger`, `org.apache.logging.log4j.Logger`.

Resources:

- CERT, FIO47-C. - Use valid format strings
- java.text.MessageFormat

How_to_fix:

A `printf-`-style format string is a string that contains placeholders, which are replaced by values when the string is printed or logged. Mismatch in the format specifiers and the arguments provided can lead to incorrect strings being created.

To avoid issues, a developer should ensure that the provided arguments match format specifiers.

Note that MessageFormat is used by most logging mechanisms, for example `java.util.logging.Logger`, thus the *single quote* must be escaped by a *double single quote*.

Noncompliant code example

```
void logging(org.slf4j.Logger slf4jLog, java.util.logging.Logger logger) {
    String.format("Too many arguments %d and %d", 1, 2, 3); // Noncompliant - the third argument '3' is unused
    String.format("First {0} and then {1}", "foo", "bar");  //Noncompliant - it appears there is confusion with the use of "java.text.Messa


    slf4jLog.debug("The number: ", 1); // Noncompliant - String contains no format specifiers.

    logger.log(level, "Can't load library \"{0}\"!", "foo"); // Noncompliant - the single quote ' must be escaped
}
```

Compliant solution

```
void logging(org.slf4j.Logger slf4jLog, java.util.logging.Logger logger) {
    String.format("Too many arguments %d and %d", 1, 2);
    String.format("First %s and then %s", "foo", "bar");

    slf4jLog.debug("The number: {}", 1);

    logger.log(level, "Can''t load library \"{0}\"!", "foo");
}
```

Test methods should comply with a naming convention (java:S3578)

Severidad: MINOR

Root_cause:

Shared naming conventions allow teams to collaborate efficiently. This rule raises an issue when a test method name does not match the provided regular expression.

Noncompliant code example

With the default value: `^test[A-Z][a-zA-Z0-9]*$`

```
@Test
public void foo() {  // Noncompliant
  //...
}
```

Compliant solution

```
@Test
public void testFoo() {
  // ...
}
```

Weak SSL/TLS protocols should not be used (java:S4423)

Severidad: CRITICAL

Introduction:

This vulnerability exposes encrypted data to a number of attacks whose goal is to recover the plaintext.

Resources:

Articles & blog posts

- Wikipedia, Padding Oracle Attack
- Wikipedia, Chosen-Ciphertext Attack
- Wikipedia, Chosen-Plaintext Attack
- Wikipedia, Semantically Secure Cryptosystems
- Wikipedia, OAEP
- Wikipedia, Galois/Counter Mode

Standards

- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- OWASP - Top 10 2021 Category A7 - Identification and Authentication Failures
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- CWE - CWE-327 - Use of a Broken or Risky Cryptographic Algorithm
- OWASP - Mobile AppSec Verification Standard - Cryptography Requirements

- OWASP - Mobile Top 10 2016 Category M5 - Insufficient Cryptography
- CWE - CWE-327 - Use of a Broken or Risky Cryptographic Algorithm
- CERT, MSC61-J. - Do not use insecure or weak cryptographic algorithms

How_to_fix:

Noncompliant code example

```
import javax.net.ssl.SSLContext;
import java.security.NoSuchAlgorithmException;

public static void main(String[] args) {
    try {
        SSLContext.getInstance("TLSv1.1"); // Noncompliant
    } catch (NoSuchAlgorithmException e) {
        // ...
    }
}
```

Compliant solution

```
import javax.net.ssl.SSLContext;
import java.security.NoSuchAlgorithmException;

public static void main(String[] args) {
    try {
        SSLContext.getInstance("TLSv1.2");
    } catch (NoSuchAlgorithmException e) {
        // ...
    }
}
```

How does this work?

As a rule of thumb, by default you should use the cryptographic algorithms and mechanisms that are considered strong by the cryptographic community.

The best choices at the moment are the following.

Use TLS v1.2 or TLS v1.3

Even though TLS V1.3 is available, using TLS v1.2 is still considered good and secure practice by the cryptography community.

The use of TLS v1.2 ensures compatibility with a wide range of platforms and enables seamless communication between different systems that do not yet have TLS v1.3 support.

The only drawback depends on whether the framework used is outdated: its TLS v1.2 settings may enable older and insecure cipher suites that are deprecated as insecure.

On the other hand, TLS v1.3 removes support for older and weaker cryptographic algorithms, eliminates known vulnerabilities from previous TLS versions, and improves performance.

Root_cause:

Encryption algorithms are essential for protecting sensitive information and ensuring secure communications in a variety of domains. They are used for several important reasons:

- Confidentiality, privacy, and intellectual property protection
- Security during transmission or on storage devices
- Data integrity, general trust, and authentication

When selecting encryption algorithms, tools, or combinations, you should also consider two things:

1. No encryption is unbreakable.
2. The strength of an encryption algorithm is usually measured by the effort required to crack it within a reasonable time frame.

For these reasons, as soon as cryptography is included in a project, it is important to choose encryption algorithms that are considered strong and secure by the cryptography community.

To provide communication security over a network, SSL and TLS are generally used. However, it is important to note that the following protocols are all considered weak by the cryptographic community, and are officially deprecated:

- SSL versions 1.0, 2.0 and 3.0
- TLS versions 1.0 and 1.1

When these unsecured protocols are used, it is best practice to expect a breach: that a user or organization with malicious intent will perform mathematical attacks on this data after obtaining it by other means.

What is the potential impact?

After retrieving encrypted data and performing cryptographic attacks on it on a given timeframe, attackers can recover the plaintext that encryption was supposed to protect.

Depending on the recovered data, the impact may vary.

Below are some real-world scenarios that illustrate the potential impact of an attacker exploiting the vulnerability.

Additional attack surface

By modifying the plaintext of the encrypted message, an attacker may be able to trigger additional vulnerabilities in the code. An attacker can further exploit a system to obtain more information.
Encrypted values are often considered trustworthy because it would not be possible for a third party to modify them under normal circumstances.

Breach of confidentiality and privacy

When encrypted data contains personal or sensitive information, its retrieval by an attacker can lead to privacy violations, identity theft, financial loss, reputational damage, or unauthorized access to confidential systems.

In this scenario, the company, its employees, users, and partners could be seriously affected.

The impact is twofold, as data breaches and exposure of encrypted data can undermine trust in the organization, as customers, clients and stakeholders may lose confidence in the organization’s ability to protect their sensitive data.

Legal and compliance issues

In many industries and locations, there are legal and compliance requirements to protect sensitive data. If encrypted data is compromised and the plaintext can be recovered, companies face legal consequences, penalties, or violations of privacy laws.

How_to_fix:

Noncompliant code example

```
import okhttp3.ConnectionSpec;
import okhttp3.TlsVersion;

public static void main(String[] args) {
    new ConnectionSpec.Builder(ConnectionSpec.MODERN_TLS)
          .tlsVersions(TlsVersion.TLS_1_1) // Noncompliant
          .build();
}
```

Compliant solution

```
import okhttp3.ConnectionSpec;
import okhttp3.TlsVersion;

public static void main(String[] args) {
    new ConnectionSpec.Builder(ConnectionSpec.MODERN_TLS)
          .tlsVersions(TlsVersion.TLS_1_2)
          .build();
}
```

How does this work?

As a rule of thumb, by default you should use the cryptographic algorithms and mechanisms that are considered strong by the cryptographic community.

The best choices at the moment are the following.

Use TLS v1.2 or TLS v1.3

Even though TLS V1.3 is available, using TLS v1.2 is still considered good and secure practice by the cryptography community.

The use of TLS v1.2 ensures compatibility with a wide range of platforms and enables seamless communication between different systems that do not yet have TLS v1.3 support.

The only drawback depends on whether the framework used is outdated: its TLS v1.2 settings may enable older and insecure cipher suites that are deprecated as insecure.

On the other hand, TLS v1.3 removes support for older and weaker cryptographic algorithms, eliminates known vulnerabilities from previous TLS versions, and improves performance.

Cryptographic keys should be robust (java:S4426)

Severidad: CRITICAL

Root_cause:

Encryption algorithms are essential for protecting sensitive information and ensuring secure communications in a variety of domains. They are used for several important reasons:

- Confidentiality, privacy, and intellectual property protection
- Security during transmission or on storage devices
- Data integrity, general trust, and authentication

When selecting encryption algorithms, tools, or combinations, you should also consider two things:

1. No encryption is unbreakable.
2. The strength of an encryption algorithm is usually measured by the effort required to crack it within a reasonable time frame.

In today’s cryptography, the length of the **key** directly affects the security level of cryptographic algorithms.

Note that depending on the algorithm, the term **key** refers to a different mathematical property. For example:

- For RSA, the key is the product of two large prime numbers, also called the **modulus**.
- For AES and Elliptic Curve Cryptography (ECC), the key is only a sequence of randomly generated bytes.
    - In some cases, AES keys are derived from a master key or a passphrase using a Key Derivation Function (KDF) like PBKDF2 (Password-Based Key Derivation Function 2)

If an application uses a key that is considered short and **insecure**, the encrypted data is exposed to attacks aimed at getting at the plaintext.

In general, it is best practice to expect a breach: that a user or organization with malicious intent will perform cryptographic attacks on this data after obtaining it by other means.

What is the potential impact?

After retrieving encrypted data and performing cryptographic attacks on it on a given timeframe, attackers can recover the plaintext that encryption was supposed to protect.

Depending on the recovered data, the impact may vary.

Below are some real-world scenarios that illustrate the potential impact of an attacker exploiting the vulnerability.

Additional attack surface

By modifying the plaintext of the encrypted message, an attacker may be able to trigger additional vulnerabilities in the code. An attacker can further exploit a system to obtain more information.
Encrypted values are often considered trustworthy because it would not be possible for a third party to modify them under normal circumstances.

Breach of confidentiality and privacy

When encrypted data contains personal or sensitive information, its retrieval by an attacker can lead to privacy violations, identity theft, financial loss, reputational damage, or unauthorized access to confidential systems.

In this scenario, the company, its employees, users, and partners could be seriously affected.

The impact is twofold, as data breaches and exposure of encrypted data can undermine trust in the organization, as customers, clients and stakeholders may lose confidence in the organization’s ability to protect their sensitive data.

Legal and compliance issues

In many industries and locations, there are legal and compliance requirements to protect sensitive data. If encrypted data is compromised and the plaintext can be recovered, companies face legal consequences, penalties, or violations of privacy laws.

Introduction:

This vulnerability exposes encrypted data to attacks whose goal is to recover the plaintext.

Resources:

- Documentation
  - NIST Documentation - NIST SP 800-186: Recommendations for Discrete Logarithm-based Cryptography: Elliptic Curve Domain Parameters
  - IETF - rfc5639: Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation

Articles & blog posts

- Microsoft, Timing vulnerabilities with CBC-mode symmetric decryption using padding
- Wikipedia, Padding Oracle Attack
- Wikipedia, Chosen-Ciphertext Attack
- Wikipedia, Chosen-Plaintext Attack
- Wikipedia, Semantically Secure Cryptosystems
- Wikipedia, OAEP
- Wikipedia, Galois/Counter Mode

Standards

- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- OWASP - Mobile AppSec Verification Standard - Cryptography Requirements
- OWASP - Mobile Top 10 2016 Category M5 - Insufficient Cryptography
- NIST 800-131A - Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths
- CWE - CWE-326 - Inadequate Encryption Strength
- CWE - CWE-327 - Use of a Broken or Risky Cryptographic Algorithm
- CERT, MSC61-J. - Do not use insecure or weak cryptographic algorithms

How_to_fix:

The following code examples either explicitly or implicitly generate keys. Note that there are differences in the size of the keys depending on the algorithm.

Due to the mathematical properties of the algorithms, the security requirements for the key size vary depending on the algorithm.
For example, a 256-bit ECC key provides about the same level of security as a 3072-bit RSA key and a 128-bit symmetric key.

Noncompliant code example

Here is an example of a private key generation with RSA:

```
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;

public static void main(String[] args) {
    try {
        KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");
        keyPairGenerator.initialize(1024); // Noncompliant

    } catch (NoSuchAlgorithmException e) {
        // ...
    }
}
```

Here is an example of a private key generation with AES:

```
import java.security.KeyGenerator;
import java.security.NoSuchAlgorithmException;

public static void main(String[] args) {
    try {
        KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");
        keyGenerator.initialize(64); // Noncompliant

    } catch (NoSuchAlgorithmException e) {
        // ...
    }
}
```

Here is an example of an Elliptic Curve (EC) initialization. It implicitly generates a private key whose size is indicated in the elliptic curve name:

```
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;
import java.security.InvalidAlgorithmParameterException;
import java.security.spec.ECGenParameterSpec;
```

```
public static void main(String[] args) {
    try {
        KeyPairGenerator keyPairGenerator    = KeyPairGenerator.getInstance("EC");
        ECGenParameterSpec ellipticCurveName = new ECGenParameterSpec("secp112r1"); // Noncompliant
        keyPairGenerator.initialize(ellipticCurveName);

    } catch (NoSuchAlgorithmException | InvalidAlgorithmParameterException e) {
        // ...
    }
}
```

Compliant solution

```
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;

public static void main(String[] args) {
    try {
        KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");
        keyPairGenerator.initialize(2048);

    } catch (NoSuchAlgorithmException e) {
        // ...
    }
}
```

```
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;

public static void main(String[] args) {
    try {
        KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("AES");
        keyPairGenerator.initialize(128);

    } catch (NoSuchAlgorithmException e) {
        // ...
    }
}
```

```
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;
import java.security.InvalidAlgorithmParameterException;
import java.security.spec.ECGenParameterSpec;

public static void main(String[] args) {
    try {
        KeyPairGenerator keyPairGenerator    = KeyPairGenerator.getInstance("EC");
        ECGenParameterSpec ellipticCurveName = new ECGenParameterSpec("secp256r1");
        keyPairGenerator.initialize(ellipticCurveName);

    } catch (NoSuchAlgorithmException | InvalidAlgorithmParameterException e) {
        // ...
    }
}
```

How does this work?

As a rule of thumb, use the cryptographic algorithms and mechanisms that are considered strong by the cryptography community.

The appropriate choices are the following.

RSA (Rivest-Shamir-Adleman) and DSA (Digital Signature Algorithm)

The security of these algorithms depends on the difficulty of attacks attempting to solve their underlying mathematical problem.

In general, a minimum key size of **2048** bits is recommended for both. It provides 112 bits of security. A key length of **3072** or **4096** should be preferred when possible.

AES (Advanced Encryption Standard)

AES supports three key sizes: 128 bits, 192 bits and 256 bits. The security of the AES algorithm is based on the computational complexity of trying all possible keys.
A larger key size increases the number of possible keys and makes exhaustive search attacks computationally infeasible. Therefore, a 256-bit key provides a higher level of security than a 128-bit or 192-bit key.

Currently, a minimum key size of **128 bits** is recommended for AES.

Elliptic Curve Cryptography (ECC)

Elliptic curve cryptography is also used in various algorithms, such as ECDSA, ECDH, or ECMQV. The length of keys generated with elliptic curve algorithms is mentioned directly in their names. For example, `secp256k1` generates a 256-bits long private key.

Currently, a minimum key size of **224 bits** is recommended for EC-based algorithms.

Additionally, some curves that theoretically provide sufficiently long keys are still discouraged. This can be because of a flaw in the curve parameters, a bad overall design, or poor performance. It is generally advised to use a NIST-approved elliptic curve wherever possible. Such curves currently include:

- NIST P curves with a size of at least 224 bits, e.g. secp256r1.
- Curve25519, generally known as ed25519 or x25519 depending on its application.
- Curve448.
- Brainpool curves with a size of at least 224 bits, e.g. brainpoolP224r1

Going the extra mile

Pre-Quantum Cryptography

Encrypted data and communications recorded today could be decrypted in the future by an attack from a quantum computer.
It is important to keep in mind that NIST-approved digital signature schemes, key agreement, and key transport may need to be replaced with secure quantum-resistant (or "post-quantum") counterpart.

Thus, if data is to remain secure beyond 2030, proactive measures should be taken now to ensure its safety.

Learn more here.

"equals" method parameters should not be marked "@Nonnull" (java:S4454)

Severidad: CRITICAL

Root_cause:

By contract, the `equals(Object)` method, from `java.lang.Object`, should accept a `null` argument. Among all the other cases, the `null` case is even explicitly detailed in the `Object.equals(...)` Javadoc, stating *"For any non-null reference value x, x.equals(null) should return false."*

Assuming that the argument to `equals` is always non-null, and enforcing that assumption with an annotation is not only a fundamental violation of the contract of `equals`, but it is also likely to cause problems in the future as the use of the class evolves over time.

The rule raises an issue when the `equals` method is overridden and its parameter annotated with any kind of `@Nonnull` annotation.

Noncompliant code example

```
public boolean equals(@javax.annotation.Nonnull Object obj) { // Noncompliant
  // ...
}
```

Compliant solution

```
public boolean equals(Object obj) {
  if (obj == null) {
    return false;
  }
  // ...
}
```

Using unsafe Jackson deserialization configuration is security-sensitive (java:S4544)

Severidad: CRITICAL

Default:

Using unsafe Jackson deserialization configuration is security-sensitive. It has led in the past to the following vulnerabilities:

- CVE-2017-4995
- CVE-2018-19362

When Jackson is configured to allow Polymorphic Type Handling (aka PTH), formerly known as Polymorphic Deserialization, "deserialization gadgets" may allow an attacker to perform remote code execution.

This rule raises an issue when:

- `enableDefaultTyping()` is called on an instance of `com.fasterxml.jackson.databind.ObjectMapper` or `org.codehaus.jackson.map.ObjectMapper`.
- or when the annotation `@JsonTypeInfo` is set at class, interface or field levels and configured with `use = JsonTypeInfo.Id.CLASS` or `use = Id.MINIMAL_CLASS`.

Ask Yourself Whether

- You configured the Jackson deserializer as mentioned above.
- The serialized data might come from an untrusted source.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- Use the latest patch versions of `jackson-databind` blocking the already discovered "deserialization gadgets".
- Avoid using the default typing configuration: `ObjectMapper.enableDefaultTyping()`.
- If possible, use `@JsonTypeInfo(use = Id.NAME)` instead of `@JsonTypeInfo(use = Id.CLASS)` or `@JsonTypeInfo(use = Id. MINIMAL_CLASS)` and so rely on `@JsonTypeName` and `@JsonSubTypes`.

Sensitive Code Example

```
ObjectMapper mapper = new ObjectMapper();
mapper.enableDefaultTyping(); // Sensitive

@JsonTypeInfo(use = Id.CLASS) // Sensitive
abstract class PhoneNumber {
}
```

See

- OWASP - Top 10 2021 Category A8 - Software and Data Integrity Failures
- OWASP - Top 10 2017 Category A8 - Insecure Deserialization
- OWASP - Deserialization of untrusted data
- CWE - CWE-502 - Deserialization of Untrusted Data
- On Jackson CVEs: Don't Panic
- CVE-2017-1509
- CVE-2017-7525
- Derived from FindSecBugs rule JACKSON_UNSAFE_DESERIALIZATION

How_to_fix:

Recommended Secure Coding Practices

- Use the latest patch versions of `jackson-databind` blocking the already discovered "deserialization gadgets".
- Avoid using the default typing configuration: `ObjectMapper.enableDefaultTyping()`.
- If possible, use `@JsonTypeInfo(use = Id.NAME)` instead of `@JsonTypeInfo(use = Id.CLASS)` or `@JsonTypeInfo(use = Id. MINIMAL_CLASS)` and so rely on `@JsonTypeName` and `@JsonSubTypes`.

See

- OWASP - Top 10 2021 Category A8 - Software and Data Integrity Failures
- OWASP - Top 10 2017 Category A8 - Insecure Deserialization
- OWASP - Deserialization of untrusted data
- CWE - CWE-502 - Deserialization of Untrusted Data
- On Jackson CVEs: Don't Panic
- CVE-2017-1509
- CVE-2017-7525
- Derived from FindSecBugs rule JACKSON_UNSAFE_DESERIALIZATION

Assess_the_problem:

Ask Yourself Whether

- You configured the Jackson deserializer as mentioned above.
- The serialized data might come from an untrusted source.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

```
ObjectMapper mapper = new ObjectMapper();
mapper.enableDefaultTyping(); // Sensitive
```

```
@JsonTypeInfo(use = Id.CLASS) // Sensitive
abstract class PhoneNumber {
}
```

Root_cause:

Using unsafe Jackson deserialization configuration is security-sensitive. It has led in the past to the following vulnerabilities:

- CVE-2017-4995
- CVE-2018-19362

When Jackson is configured to allow Polymorphic Type Handling (aka PTH), formerly known as Polymorphic Deserialization, "deserialization gadgets" may allow an attacker to perform remote code execution.

This rule raises an issue when:

- `enableDefaultTyping()` is called on an instance of `com.fasterxml.jackson.databind.ObjectMapper` or `org.codehaus.jackson.map.ObjectMapper`.
- or when the annotation `@JsonTypeInfo` is set at class, interface or field levels and configured with `use = JsonTypeInfo.Id.CLASS` or `use = Id.MINIMAL_CLASS`.

JUnit assertTrue/assertFalse should be simplified to the corresponding dedicated assertion (java:S5785)

Severidad: MAJOR

Root_cause:

Testing equality or nullness with JUnit's `assertTrue()` or `assertFalse()` should be simplified to the corresponding dedicated assertion.

Noncompliant code example

```
Assert.assertTrue(a.equals(b));
Assert.assertTrue(a == b);
Assert.assertTrue(a == null);
Assert.assertTrue(a != null);
Assert.assertFalse(a.equals(b));
```

Compliant solution

```
Assert.assertEquals(a, b);
Assert.assertSame(a, b);
Assert.assertNull(a);
Assert.assertNotNull(a);
Assert.assertNotEquals(a, b);
```

A new session should be created during user authentication (java:S5876)

Severidad: CRITICAL

Introduction:

An attacker may trick a user into using a predetermined session identifier. Consequently, this attacker can gain unauthorized access and impersonate the user's session. This kind of attack is called session fixation, and protections against it should not be disabled.

Root_cause:

Session fixation attacks take advantage of the way web applications manage session identifiers. Here's how a session fixation attack typically works:

- When a user visits a website or logs in, a session is created for them.
- This session is assigned a unique session identifier, stored in a cookie, in local storage, or through URL parameters.
- In a session fixation attack, an attacker tricks a user into using a predetermined session identifier controlled by the attacker. For example, the attacker sends the victim an email containing a link with this predetermined session identifier.
- When the victim clicks on the link, the web application does not create a new session identifier but uses this identifier known to the attacker.
- At this point, the attacker can hijack and impersonate the victim's session.

What is the potential impact?

Session fixation attacks pose a significant security risk to web applications and their users. By exploiting this vulnerability, attackers can gain unauthorized access to user sessions, potentially leading to various malicious activities. Some of the most relevant scenarios are the following:

Impersonation

Once an attacker successfully fixes a session identifier, they can impersonate the victim and gain access to their account without providing valid credentials. This can result in unauthorized actions, such as modifying personal information, making unauthorized transactions, or even performing malicious activities on behalf of the victim. An attacker can also manipulate the victim into performing actions they wouldn't normally do, such as revealing sensitive information or conducting financial transactions on the attacker's behalf.

Data Breach

If an attacker gains access to a user's session, they may also gain access to sensitive data associated with that session. This can include personal information, financial details, or any other confidential data that the user has access to within the application. The compromised data can be used for identity theft, financial fraud, or other malicious purposes.

Privilege Escalation

In some cases, session fixation attacks can be used to escalate privileges within a web application. By fixing a session identifier with higher privileges, an attacker can bypass access controls and gain administrative or privileged access to the application. This can lead to unauthorized modifications, data manipulation, or even complete compromise of the application and its underlying systems.

Resources:

Documentation

Session Fixation Attack Protection

Standards

- OWASP - Top 10 2021 Category A7 - Identification and Authentication Failures
- OWASP - Top 10 2017 Category A2 - Broken Authentication
- OWASP Sesssion Fixation
- CWE - CWE-384 - Session Fixation
- STIG Viewer - Application Security and Development: V-222579 - Applications must use system-generated session identifiers that protect against session fixation.
- STIG Viewer - Application Security and Development: V-222582 - The application must not re-use or recycle session IDs.

How_to_fix:

In a Spring Security's context, session fixation protection is enabled by default but can be disabled with `sessionFixation().none()` method. Session fixation protection can also be enabled explicitly with `migrateSession()` but is superfluous in most cases.

Noncompliant code example

```
@Override
protected void configure(HttpSecurity http) throws Exception {
   http.sessionManagement()
     .sessionFixation().none(); // Noncompliant: the existing session will continue
}
```

Compliant solution

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http.sessionManagement()
     .sessionFixation().migrateSession();
}
```

How does this work?

The protection works by ensuring that the session identifier, which is used to identify and track a user's session, is changed or regenerated during the authentication process.

Here's how session fixation protection typically works:

1. When a user visits a website or logs in, a session is created for them. This session is assigned a unique session identifier, which is stored in a cookie or passed through URL parameters.
2. In a session fixation attack, an attacker tricks a user into using a predetermined session identifier controlled by the attacker. This allows the attacker to potentially gain unauthorized access to the user's session.
3. To protect against session fixation attacks, session fixation protection mechanisms come into play during the authentication process. When a user successfully authenticates, this mechanism generates a new session identifier for the user's session.
4. The old session identifier, which may have been manipulated by the attacker, is invalidated and no longer associated with the user's session. This ensures that any attempts by the attacker to use the fixed session identifier are rendered ineffective.

5. The user is then assigned the new session identifier, which is used for subsequent requests and session tracking. This new session identifier is typically stored in a new session cookie or passed through URL parameters.

By regenerating the session identifier upon authentication, session fixation protection helps ensure that the user’s session is tied to a new, secure identifier that the attacker cannot predict or control. This mitigates the risk of an attacker gaining unauthorized access to the user’s session and helps maintain the integrity and security of the application’s session management process.

In Spring, calling `migrateSession()` explicitly is not necessary in most cases, as it automatically migrates session attributes to a new session upon successful authentication. The `migrateSession()` method is typically explicitly used when you want to manually trigger the migration of session attributes to a new session outside of the authentication process.

Regex boundaries should not be used in a way that can never be matched (java:S5996)

Severidad: CRITICAL

Root_cause:

In regular expressions the boundaries `^` and `\A` can only match at the beginning of the input (or, in case of `^` in combination with the `MULTILINE` flag, the beginning of the line) and `$`, `\Z` and `\z` only at the end.

These patterns can be misused, by accidentally switching `^` and `$` for example, to create a pattern that can never match.

Noncompliant code example

```
// This can never match because $ and ^ have been switched around
Pattern.compile("$[a-z]+^"); // Noncompliant
```

Compliant solution

```
Pattern.compile("^[a-z]+$");
```

Regular expressions should not overflow the stack (java:S5998)

Severidad: MAJOR

Root_cause:

The Java regex engine uses recursive method calls to implement backtracking. Therefore when a repetition inside a regular expression contains multiple paths (i.e. the body of the repetition contains an alternation (`|`), an optional element or another repetition), trying to match the regular expression can cause a stack overflow on large inputs. This does not happen when using a possessive quantifier (such as `*+` instead of `*`) or when using a character class inside a repetition (e.g. `[ab]*` instead of `(a|b)*`).

The size of the input required to overflow the stack depends on various factors, including of course the stack size of the JVM. One thing that significantly increases the size of the input that can be processed is if each iteration of the repetition goes through a chain of multiple constant characters because such consecutive characters will be matched by the regex engine without invoking any recursion.

For example, on a JVM with a stack size of 1MB, the regex `(?:a|b)*` will overflow the stack after matching around 6000 characters (actual numbers may differ between JVM versions and even across multiple runs on the same JVM) whereas `(?:abc|def)*` can handle around 15000 characters.

Since often times stack growth can’t easily be avoided, this rule will only report issues on regular expressions if they can cause a stack overflow on realistically sized inputs. You can adjust the `maxStackConsumptionFactor` parameter to adjust this.

Noncompliant code example

```
Pattern.compile("(a|b)*"); // Noncompliant
Pattern.compile("(.|\n)*"); // Noncompliant
Pattern.compile("(ab?)*"); // Noncompliant
```

Compliant solution

```
Pattern.compile("[ab]*"); // Character classes don't cause recursion the way that '|' does
Pattern.compile("(?s).*"); // Enabling the (?s) flag makes '.' match line breaks, so '|\n' isn't necessary
Pattern.compile("(ab?)*+"); // Possessive quantifiers don't cause recursion because they disable backtracking
```

"Enumeration" should not be implemented (java:S1150)

Severidad: MAJOR

Root_cause:

As documented in `Enumeration` 's Javadoc, you should favor the `Iterator` interface over `Enumeration`. `Iterator` offers a similar contract to `Enumeration` with the addition of a method for removal and shorter method names.

Noncompliant code example

```
public class MyClass implements Enumeration {  // Noncompliant
  /* ... */
}
```

Compliant solution

```
public class MyClass implements Iterator {     // Compliant
  /* ... */
}
```

Resources:

- docs.oracle.com - Enumeration

"switch case" clauses should not have too many lines of code (java:S1151)

Severidad: MAJOR

Root_cause:

The `switch` statement should be used only to clearly define some new branches in the control flow. As soon as a `case` clause contains too many statements this highly decreases the readability of the overall control flow statement. In such case, the content of the `case` clause should be extracted into a dedicated method.

Noncompliant code example

With the default threshold of 5:

```
switch (myVariable) {
  case 0: // Noncompliant: 6 lines till next case
    methodCall1("");
    methodCall2("");
    methodCall3("");
    methodCall4("");
    break;
  case 1:
  ...
}
```

Compliant solution

```
switch (myVariable) {
  case 0:
    doSomething()
    break;
  case 1:
  ...
}
...
private void doSomething(){
    methodCall1("");
    methodCall2("");
    methodCall3("");
    methodCall4("");
}
```

"String.valueOf()" should not be appended to a "String" (java:S1153)

Severidad: MINOR

Root_cause:

Appending `String.valueOf()` to a `String` decreases the code readability.

The argument passed to `String.valueOf()` should be directly appended instead.

Noncompliant code example

```
String message = "Output is " + String.valueOf(12);
```

Compliant solution

```
String message = "Output is " + 12;
```

"Collection.isEmpty()" should be used to test for emptiness (java:S1155)

Severidad: MINOR

Root_cause:

When you call `isEmpty()`, it clearly communicates the code's intention, which is to check if the collection is empty. Using `size() == 0` for this purpose is less direct and makes the code slightly more complex.

Moreover, depending on the implementation, the `size()` method can have a time complexity of `O(n)` where `n` is the number of elements in the collection. On the other hand, `isEmpty()` simply checks if there is at least one element in the collection, which is a constant time operation, `O(1)`.

```
public class MyClass {
  public void doSomething(Collection<String> myCollection) {
    if (myCollection.size() == 0) { // Noncompliant
      doSomethingElse();
    }
  }
}
```

Prefer using `isEmpty()` to test for emptiness over `size()`.

```
public class MyClass {
  public void doSomething(Collection<String> myCollection) {
    if (myCollection.isEmpty()) {
      doSomethingElse();
    }
  }
}
```

Empty arrays and collections should be returned instead of null (java:S1168)

Severidad: MAJOR

Resources:

- 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

Root_cause:

Returning `null` instead of an actual array, collection or map 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<Result> getAllResults() {
  return null;                             // Noncompliant
}

public static Result[] getResults() {
  return null;                             // Noncompliant
}

public static Map<String, Object> getValues() {
  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) {
      /* ... */
    }
  }
  List<Result> allResults = getAllResults();
  if (allResults != null) {                 // Nullity test required to prevent NPE
    for (Result result: allResults) {
      /* ... */
    }
  }

  Map<String, Object> values = getValues();
  if (values != null) {                    // Nullity test required to prevent NPE
    values.forEach((k, v) -> doSomething(k, v));
  }
}
```

Compliant solution

```
public static List<Result> getAllResults() {
  return Collections.emptyList();          // Compliant
}

public static Result[] getResults() {
  return new Result[0];                     // Compliant
}

public static Map<String, Object> getValues() {
  return Collections.emptyMap();           // Compliant
}

public static void main(String[] args) {
  for (Result result: getAllResults()) {
    /* ... */
  }

  for (Result result: getResults()) {
    /* ... */
  }

  getValues().forEach((k, v) -> doSomething(k, v));
}
```

String operations with predictable outcomes should be avoided (java:S2121)

Severidad: MAJOR

Resources:

- Oracle Java SE - String

Root_cause:

Operations performed on a string with predictable outcomes should be avoided. For example:

- checking if a string contains itself
- comparing a string with itself
- matching a string against itself
- creating a substring from 0 to the end of the string
- creating a substring from the end of the string
- replacing a string with itself
- replacing a substring with the exact substring

How_to_fix:

Avoid performing the operation that has a predictable outcome.

Noncompliant code example

```
String speech = "SonarQube is the best static code analysis tool."

String s1 = speech.substring(0); // Noncompliant - yields the whole string
String s2 = speech.substring(speech.length()); // Noncompliant - yields "";
String s3 = speech.substring(5, speech.length()); // Noncompliant - use the 1-arg version instead

if (speech.contains(speech)) { // Noncompliant - always true
    // ...
}
```

Compliant solution

```
String speech = "SonarQube is the best static code analysis tool."

String s1 = speech;
String s2 = "";
String s3 = speech.substring(5);

// ...
```

"ScheduledThreadPoolExecutor" should not have 0 core threads (java:S2122)

Severidad: CRITICAL

Resources:

Documentation

- [Oracle Java SE - ScheduledThreadPoolExecutor]

Articles & blog posts

- [Zalando - How to set an ideal thread pool size]
- [Baeldung - ThreadPoolTaskExecutor corePoolSize vs. maxPoolSize]

Root_cause:

`ThreadPoolExecutor` is an object that efficiently manages and controls the execution of multiple tasks in a thread pool. A thread pool is a collection of pre-initialized threads ready to execute tasks. Instead of creating a new thread for each task, which can be costly in terms of system resources, a thread pool reuses existing threads.

`java.util.concurrent.ScheduledThreadPoolExecutor` is an extension of `ThreadPoolExecutor` that can additionally schedule commands to run after a given delay or to execute periodically.

`ScheduledThreadPoolExecutor` 's pool is sized with `corePoolSize`, so setting `corePoolSize` to zero means the executor will have no threads and run nothing. `corePoolSize` should have a value greater than zero and valid for your tasks.

This rule detects instances where `corePoolSize` is set to zero via its setter or the object constructor.

Noncompliant code example

```
public void do(){

  int poolSize = 5; // value greater than 0

  ScheduledThreadPoolExecutor threadPool1 = new ScheduledThreadPoolExecutor(0); // Noncompliant

  ScheduledThreadPoolExecutor threadPool2 = new ScheduledThreadPoolExecutor(poolSize);
  threadPool2.setCorePoolSize(0);  // Noncompliant
}
```

Values should not be uselessly incremented (java:S2123)

Severidad: MAJOR

Root_cause:

A value that is incremented or decremented and then not stored is at best wasted code and at worst a bug.

Noncompliant code example

```
public int pickNumber() {
  int i = 0;
  int j = 0;

  i = i++; // Noncompliant; i is still zero

  return j++; // Noncompliant; 0 returned
}
```

Compliant solution

```
public int pickNumber() {
  int i = 0;
  int j = 0;

  i++;
  return ++j;
}
```

Constructors should not be used to instantiate "String", "BigInteger", "BigDecimal" and primitive-wrapper classes (java:S2129)

Severidad: MAJOR

Root_cause:

Calling constructors for `String`, `BigInteger`, `BigDecimal` and the objects used to wrap primitives is less efficient and less clear than relying on autoboxing or `valueOf`.

Consider simplifying when possible for more efficient and cleaner code.

Noncompliant code example

```
String empty = new String(); // Noncompliant; yields essentially "", so just use that.
String nonempty = new String("Hello world"); // Noncompliant
Double myDouble = new Double(1.1); // Noncompliant; use valueOf
Integer integer = new Integer(1); // Noncompliant
Boolean bool = new Boolean(true); // Noncompliant
BigInteger bigInteger1 = new BigInteger("3"); // Noncompliant
BigInteger bigInteger2 = new BigInteger("9223372036854775807"); // Noncompliant
BigInteger bigInteger3 = new BigInteger("111222333444555666777888999"); // Compliant, greater than Long.MAX_VALUE
BigDecimal bigDecimal = new BigDecimal("42.0"); // Compliant (see Exceptions section)
```

Compliant solution

```
String empty = "";
String nonempty = "Hello world";
Double myDouble = 1.1;
Integer integer = 1;
Boolean bool = true;
BigInteger bigInteger1 = BigInteger.valueOf(3);
BigInteger bigInteger2 = BigInteger.valueOf(9223372036854775807L);
BigInteger bigInteger3 = new BigInteger("111222333444555666777888999");
BigDecimal bigDecimal = new BigDecimal("42.0");
```

Exceptions

`BigDecimal` constructor with a `double` argument is ignored as using `valueOf` instead might change the resulting value. See S2111.

Resources:

- Oracle - Learning the Java Language - Autoboxing and Unboxing

Objects should not be created only to invoke "getClass" (java:S2133)

Severidad: MAJOR

Root_cause:

Creating an object for the sole purpose of calling `getClass` on it is a waste of memory and cycles. Instead, simply use the class's `.class` property.

Noncompliant code example

```
MyObject myOb = new MyObject();  // Noncompliant
Class c = myOb.getClass();
```

Compliant solution

```
Class c = MyObject.class;
```

"HttpServletRequest.getRequestedSessionId()" should not be used (java:S2254)

Severidad: CRITICAL

Introduction:

This function uses a session ID that is supplied by the client. Because of this, the ID may not be valid or might even be spoofed.

Resources:

Documentation

- Jakarta EE Documentation - `HttpServletRequest - getRequestedSessionId`

Standards

- OWASP - Top 10 2021 Category A4 - Insecure Design
- OWASP - Top 10 2017 Category A2 - Broken Authentication
- CWE - CWE-807 - Reliance on Untrusted Inputs in a Security Decision
- STIG Viewer - Application Security and Development: V-222582 - The application must not re-use or recycle session IDs.

Root_cause:

According to the API documentation of the `HttpServletRequest.getRequestedSessionId()` method:

> Returns the session ID specified by the client. This may not be the same as the ID of the current valid session for this request. If the client did not specify a session ID, this method returns null.

The session ID it returns is either transmitted through a cookie or a URL parameter. This allows an end user to manually update the value of this session ID in an HTTP request.

Due to the ability of the end-user to manually change the value, the session ID in the request should only be used by a servlet container (e.g. Tomcat or Jetty) to see if the value matches the ID of an existing session. If it does not, the user should be considered unauthenticated.

What is the potential impact?

Using a client-supplied session ID to manage sessions on the server side can potentially have an impact on the security of the application.

Impersonation (through session fixation)

If an attacker succeeds in fixing a user's session to a session identifier that they know, then they can impersonate this victim and gain access to their account without providing valid credentials. This can result in unauthorized actions, such as modifying personal information, making unauthorized transactions, or even performing malicious activities on behalf of the victim. An attacker can also manipulate the victim into performing actions they wouldn't normally do, such as revealing sensitive information or conducting financial transactions on the attacker's behalf.

How_to_fix:

In both examples, a session ID is used to check whether a user's session is still active. In the noncompliant example, the session ID supplied by the user is used. In the compliant example, the session ID defined by the server is used instead.

Noncompliant code example

```
if (isActiveSession(request.getRequestedSessionId())) { // Noncompliant
    // ...
}
```

Compliant solution

```
if (isActiveSession(request.getSession().getId())) {
    // ...
}
```

How does this work?

The noncompliant example uses `HttpServletRequest.getRequestedSessionId()` to retrieve a session ID. This ID is then used to verify if the given session is still active. As this value is given by a user, this value is not guaranteed to be a valid ID.

The compliant example instead uses the server's session ID to verify if the session is active. Additionally, `getSession()` will create a new session if the user's request does not contain a valid ID.

Using non-standard cryptographic algorithms is security-sensitive (java:S2257)

Severidad: CRITICAL

Root_cause:

The use of a non-standard algorithm is dangerous because a determined attacker may be able to break the algorithm and compromise whatever data has been protected. Standard algorithms like `SHA-256`, `SHA-384`, `SHA-512`, … should be used instead.

This rule tracks creation of `java.security.MessageDigest` subclasses.

Assess_the_problem:

Sensitive Code Example

```
public class MyCryptographicAlgorithm extends MessageDigest {
  ...
}
```

Default:

The use of a non-standard algorithm is dangerous because a determined attacker may be able to break the algorithm and compromise whatever data has been protected. Standard algorithms like `SHA-256`, `SHA-384`, `SHA-512`, … should be used instead.

This rule tracks creation of `java.security.MessageDigest` subclasses.

Recommended Secure Coding Practices

- Use a standard algorithm instead of creating a custom one.

Sensitive Code Example

```
public class MyCryptographicAlgorithm extends MessageDigest {
  ...
}
```

Compliant Solution

```
MessageDigest digest = MessageDigest.getInstance("SHA-256");
```

See

- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- CWE - CWE-327 - Use of a Broken or Risky Cryptographic Algorithm
- Derived from FindSecBugs rule MessageDigest is Custom

How_to_fix:

Recommended Secure Coding Practices

- Use a standard algorithm instead of creating a custom one.

Compliant Solution

```
MessageDigest digest = MessageDigest.getInstance("SHA-256");
```

See

- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- CWE - CWE-327 - Use of a Broken or Risky Cryptographic Algorithm
- Derived from FindSecBugs rule MessageDigest is Custom

Null pointers should not be dereferenced (java:S2259)

Severidad: MAJOR

How_to_fix:

Noncompliant code example

The variable `myObject` is equal to `null`, meaning it has no value:

```
public void method() {
  Object myObject = null;
  System.out.println(myObject.toString()); // Noncompliant: myObject is null
}
```

The parameter `input` might be `null` as suggested by the `if` condition:

```
public void method(Object input)
{
  if (input == null)
  {
    // ...
  }
  System.out.println(input.toString()); // Noncompliant
}
```

The unboxing triggered in the return statement will throw a `NullPointerException`:

```
public boolean method() {
  Boolean boxed = null;
  return boxed; // Noncompliant
}
```

Both `conn` and `stmt` might be `null` in case an exception was thrown in the try{} block:

```
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
  conn.close();  // Noncompliant
}
```

As `getName()` is annotated with `@CheckForNull`, there is a risk of `NullPointerException` here:

```
@CheckForNull
String getName() {...}

public boolean isNameEmpty() {
  return getName().length() == 0; // Noncompliant
}
```

As `merge(…)` parameter is annotated with `@Nonnull`, passing an identified potential null value (thanks to @CheckForNull) is not safe:

```
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
}
```

Compliant solution

Ensuring the variable `myObject` has a value resolves the issue:

```
public void method() {
  Object myObject = new Object();
  System.out.println(myObject.toString()); // Compliant: myObject is not null
}
```

Preventing the non-compliant code to be executed by returning early:

```
public void method(Object input)
{
  if (input == null)
  {
    return;
  }
  System.out.println(input.toString()); // Compliant: if 'input' is null, this is unreachable
}
```

Ensuring that no unboxing of `null` value can happen resolves the issue

```
public boolean method() {
  Boolean boxed = true;
  return boxed; // Compliant
}
```

Ensuring that both `conn` and `stmt` are not `null` resolves the issue:

```
Connection conn = null;
Statement stmt = null;
try {
  conn = DriverManager.getConnection(DB_URL,USER,PASS);
  stmt = conn.createStatement();
  // ...
} catch(Exception e) {
  e.printStackTrace();
} finally {
  if (stmt != null) {
    stmt.close();  // Compliant
  }
  if (conn != null) {
    conn.close();  // Compliant
  }
}
```

Checking the returned value of `getName()` resolves the issue:

```
@CheckForNull
String getName() {...}

public boolean isNameEmpty() {
  String name = getName();
  if (name != null) {
    return name.length() == 0; // Compliant
  } else {
    // ...
  }
}
```

Ensuring that the provided `color` is not `null` resolves the issue:

```
private void merge(@Nonnull Color firstColor, @Nonnull Color secondColor) {...}

public void append(@CheckForNull Color color) {
  if (color != null) {
    merge(currentColor, color);  // Compliant
  }
}
```

Resources:

- CWE - 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

Root_cause:

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 nullability annotations, like `@CheckForNull` or `@Nonnull`, defined in JSR-305 to understand which values can be null or not. `@Nonnull` will be ignored if used on the parameter of the `equals` method, which by contract should always work with null.

Creating cookies without the "HttpOnly" flag is security-sensitive (java:S3330)

Severidad: MINOR

Assess_the_problem:

Ask Yourself Whether

- the cookie is sensitive, used to authenticate the user, for instance a *session-cookie*
- the `HttpOnly` attribute offer an additional protection (not the case for an *XSRF-TOKEN cookie* / CSRF token for example)

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

If you create a security-sensitive cookie in your JAVA code:

```
Cookie c = new Cookie(COOKIENAME, sensitivedata);
c.setHttpOnly(false);  // Sensitive: this sensitive cookie is created with the httponly flag set to false and so it can be stolen easily in
```

By default the `HttpOnly` flag is set to *false:*

```
Cookie c = new Cookie(COOKIENAME, sensitivedata);  // Sensitive: this sensitive cookie is created with the httponly flag not defined (by de
```

Root_cause:

When a cookie is configured with the `HttpOnly` attribute set to *true*, the browser guaranties that no client-side script will be able to read it. In most cases, when a cookie is created, the default value of `HttpOnly` is *false* and it's up to the developer to decide whether or not the content of the cookie can be read by the client-side script. As a majority of Cross-Site Scripting (XSS) attacks target the theft of session-cookies, the `HttpOnly` attribute can help to reduce their impact as it won't be possible to exploit the XSS vulnerability to steal session-cookies.

How_to_fix:

Recommended Secure Coding Practices

- By default the `HttpOnly` flag should be set to *true* for most of the cookies and it's mandatory for session / sensitive-security cookies.

Compliant Solution

```
Cookie c = new Cookie(COOKIENAME, sensitivedata);
c.setHttpOnly(true); // Compliant: this sensitive cookie is protected against theft (HttpOnly=true)
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP HttpOnly
- OWASP - Top 10 2017 Category A7 - Cross-Site Scripting (XSS)
- CWE - CWE-1004 - Sensitive Cookie Without 'HttpOnly' Flag
- Derived from FindSecBugs rule HTTPONLY_COOKIE
- STIG Viewer - Application Security and Development: V-222575 - The application must set the HTTPOnly flag on session cookies.

Default:

When a cookie is configured with the `HttpOnly` attribute set to *true*, the browser guaranties that no client-side script will be able to read it. In most cases, when a cookie is created, the default value of `HttpOnly` is *false* and it's up to the developer to decide whether or not the content of the cookie can be read by the client-side script. As a majority of Cross-Site Scripting (XSS) attacks target the theft of session-cookies, the `HttpOnly` attribute can help to reduce their impact as it won't be possible to exploit the XSS vulnerability to steal session-cookies.

Ask Yourself Whether

- the cookie is sensitive, used to authenticate the user, for instance a *session-cookie*
- the `HttpOnly` attribute offer an additional protection (not the case for an *XSRF-TOKEN cookie* / CSRF token for example)

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- By default the `HttpOnly` flag should be set to *true* for most of the cookies and it's mandatory for session / sensitive-security cookies.

Sensitive Code Example

If you create a security-sensitive cookie in your JAVA code:

```
Cookie c = new Cookie(COOKIENAME, sensitivedata);
c.setHttpOnly(false);  // Sensitive: this sensitive cookie is created with the httponly flag set to false and so it can be stolen easily in
```

By default the `HttpOnly` flag is set to *false:*

```
Cookie c = new Cookie(COOKIENAME, sensitivedata);  // Sensitive: this sensitive cookie is created with the httponly flag not defined (by de
```

Compliant Solution

```
Cookie c = new Cookie(COOKIENAME, sensitivedata);
c.setHttpOnly(true); // Compliant: this sensitive cookie is protected against theft (HttpOnly=true)
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP HttpOnly
- OWASP - Top 10 2017 Category A7 - Cross-Site Scripting (XSS)

- CWE - CWE-1004 - Sensitive Cookie Without 'HttpOnly' Flag
- Derived from FindSecBugs rule HTTPONLY_COOKIE
- STIG Viewer - Application Security and Development: V-222575 - The application must set the HTTPOnly flag on session cookies.

Expressions used in "assert" should not produce side effects (java:S3346)

Severidad: MAJOR

Resources:

- CERT, EXP06-J. - Expressions used in assertions must not produce side effects

Root_cause:

Since `assert` statements aren’t executed by default (they must be enabled with JVM flags) developers should never rely on their execution the evaluation of any logic required for correct program function.

Noncompliant code example

```
assert myList.remove(myList.get(0));  // Noncompliant
```

Compliant solution

```
boolean removed = myList.remove(myList.get(0));
assert removed;
```

Test classes should comply with a naming convention (java:S3577)

Severidad: MINOR

Root_cause:

Shared naming conventions allow teams to collaborate efficiently. This rule raises an issue when a test class name does not match the provided regular expression.

Noncompliant code example

With the default value: `^((Test|IT)[a-zA-Z0-9_]+|[A-Z][a-zA-Z0-9_]*(Test|Tests|TestCase|IT|ITCase))$`

```
class Foo {  // Noncompliant
  @Test
  void check() {  }
}

class Bar {  // Noncompliant
  @Nested
  class PositiveCase {
    @Test
    void check() {  }
  }
}
```

Compliant solution

```
class FooTest {
  @Test
  void check() {  }
}

class BarIT {
  @Nested
  class PositiveCase {
    @Test
    void check() {  }
  }
}
```

"Integer.toHexString" should not be used to build hexadecimal strings (java:S4425)

Severidad: MAJOR

Root_cause:

Using `Integer.toHexString` is a common mistake when converting sequences of bytes into hexadecimal string representations. The problem is that the method trims leading zeroes, which can lead to wrong conversions. For instance a two bytes value of `0x4508` would be converted into `45` and `8` which once concatenated would give `0x458`.

This is particularly damaging when converting hash-codes and could lead to a security vulnerability.

This rule raises an issue when `Integer.toHexString` is used in any kind of string concatenations.

Noncompliant code example

```
MessageDigest md = MessageDigest.getInstance("SHA-256");
byte[] bytes = md.digest(password.getBytes("UTF-8"));

StringBuilder sb = new StringBuilder();
for (byte b : bytes) {
    sb.append(Integer.toHexString( b & 0xFF )); // Noncompliant
}
```

Compliant solution

```
MessageDigest md = MessageDigest.getInstance("SHA-256");
byte[] bytes = md.digest(password.getBytes("UTF-8"));

StringBuilder sb = new StringBuilder();
for (byte b : bytes) {
    sb.append(String.format("%02X", b));
}
```

Resources:

- CWE - CWE-704 - Incorrect Type Conversion or Cast
- Derived from FindSecBugs rule BAD_HEXA_CONVERSION

Allowing deserialization of LDAP objects is security-sensitive (java:S4434)

Severidad: MAJOR

Assess_the_problem:

Ask Yourself Whether

- The application connects to an untrusted LDAP directory.
- User-controlled objects can be stored in the LDAP directory.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

```
DirContext ctx = new InitialDirContext();
// ...
ctx.search(query, filter,
        new SearchControls(scope, countLimit, timeLimit, attributes,
            true, // Noncompliant; allows deserialization
            deref));
```

Default:

JNDI supports the deserialization of objects from LDAP directories, which can lead to remote code execution.

This rule raises an issue when an LDAP search query is executed with `SearchControls` configured to allow deserialization.

Ask Yourself Whether

- The application connects to an untrusted LDAP directory.
- User-controlled objects can be stored in the LDAP directory.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

It is recommended to disable deserialization of LDAP objects.

Sensitive Code Example

```
DirContext ctx = new InitialDirContext();
// ...
ctx.search(query, filter,
        new SearchControls(scope, countLimit, timeLimit, attributes,
            true, // Noncompliant; allows deserialization
            deref));
```

Compliant Solution

```
DirContext ctx = new InitialDirContext();
// ...
ctx.search(query, filter,
        new SearchControls(scope, countLimit, timeLimit, attributes,
            false, // Compliant
            deref));
```

See

- OWASP - Top 10 2021 Category A8 - Software and Data Integrity Failures
- CWE - CWE-502 - Deserialization of Untrusted Data
- OWASP - Top 10 2017 Category A8 - Insecure Deserialization
- BlackHat presentation
- Derived from FindSecBugs rule LDAP_ENTRY_POISONING

How_to_fix:

Recommended Secure Coding Practices

It is recommended to disable deserialization of LDAP objects.

Compliant Solution

```
DirContext ctx = new InitialDirContext();
// ...
ctx.search(query, filter,
        new SearchControls(scope, countLimit, timeLimit, attributes,
            false, // Compliant
            deref));
```

See

- OWASP - Top 10 2021 Category A8 - Software and Data Integrity Failures
- CWE - CWE-502 - Deserialization of Untrusted Data
- OWASP - Top 10 2017 Category A8 - Insecure Deserialization
- BlackHat presentation
- Derived from FindSecBugs rule LDAP_ENTRY_POISONING

Root_cause:

JNDI supports the deserialization of objects from LDAP directories, which can lead to remote code execution.

This rule raises an issue when an LDAP search query is executed with `SearchControls` configured to allow deserialization.

Constructors of an "abstract" class should not be declared "public" (java:S5993)

Severidad: MAJOR

Root_cause:

Abstract classes should not have public constructors. Constructors of abstract classes can only be called in constructors of their subclasses. So there is no point in making them public. The `protected` modifier should be enough.

Noncompliant code example

```
public abstract class AbstractClass1 {
    public AbstractClass1 () { // Noncompliant, has public modifier
        // do something here
    }
}
```

Compliant solution

```
public abstract class AbstractClass2 {
    protected AbstractClass2 () {
        // do something here
    }
}
```

Regex patterns following a possessive quantifier should not always fail (java:S5994)

Severidad: CRITICAL

Root_cause:

Possessive quantifiers in Regex patterns like below improve performance by eliminating needless backtracking:

`?+ , *+ , ++ , {n}+ , {n,}+ , {n,m}+`

But because possessive quantifiers do not keep backtracking positions and never give back, the following sub-patterns should not match only similar characters. Otherwise, possessive quantifiers consume all characters that could have matched the following sub-patterns and nothing remains for the following sub-patterns.

Noncompliant code example

```
Pattern pattern1 = Pattern.compile("a++abc");       // Noncompliant, the second 'a' never matches
Pattern pattern2 = Pattern.compile("\\d*+[02468]"); // Noncompliant, the sub-pattern "[02468]" never matches
```

Compliant solution

```
Pattern pattern1 = Pattern.compile("aa++bc");            // Compliant, for example it can match "aaaabc"
Pattern pattern2 = Pattern.compile("\\d*+(?<=[02468])"); // Compliant, for example it can match an even number like "1234"
```

"@PathVariable" annotation should be present if a path variable is used (java:S6856)

Severidad: MAJOR

Root_cause:

The `@PathVariable` annotation in Spring extracts values from the URI path and binds them to method parameters in a Spring MVC controller. It is commonly used with `@GetMapping`, `@PostMapping`, `@PutMapping`, and `@DeleteMapping` to capture path variables from the URI. These annotations map HTTP requests to specific handler methods in a controller. They are part of the Spring Web module and are commonly used to define the routes for different HTTP operations in a RESTful API.

If a method has a path template containing a placeholder, like "/api/resource/{id}", and there's no `@PathVariable` annotation on a method parameter to capture the id path variable, Spring will disregard the id variable.

This rule will raise an issue if a method has a path template with a placeholder, but no corresponding `@PathVariable`, or vice-versa.

How_to_fix:

Noncompliant code example

```
@GetMapping("/api/resource/{id}")
public ResponseEntity<String> getResourceById(Long id) { // Noncompliant - The 'id' parameter will not be automatically populated with the
  return ResponseEntity.ok("Fetching resource with ID: " + id);
}

@GetMapping("/api/asset/")
public ResponseEntity<String> getAssetById(@PathVariable Long id) { // Noncompliant - The 'id' parameter does not have a corresponding plac
  return ResponseEntity.ok("Fetching asset with ID: " + id);
}
```

Compliant solution

```
@GetMapping("/api/resource/{id}")
public ResponseEntity<String> getResourceById(@PathVariable Long id) { // Compliant
  return ResponseEntity.ok("Fetching resource with ID: " + id);
}

@GetMapping("/api/asset/{id}")
public ResponseEntity<String> getAssetById(@PathVariable Long id) {
  return ResponseEntity.ok("Fetching asset with ID: " + id);
}
```

Resources:

Documentation

- Spring IO - Building REST services with Spring
- Spring Framework API - PathVariable
- Spring Framework API - GetMapping
- Spring Framework API - PostMapping
- Spring Framework API - PutMapping
- Spring Framework API - DeleteMapping

Articles & blog posts

- Baeldung - Spring @PathVariable

SpEL expression should have a valid syntax (java:S6857)

Severidad: MAJOR

Resources:

Documentation

- Spring Framework - Spring Expression Language (SpEL)

Root_cause:

SpEL is used in Spring annotations and is parsed by the Spring framework, not by the Java compiler. This means that invalid SpEL expressions are not detected during Java compile time. They will cause exceptions during runtime instead, or even fail silently with the expression string interpreted as a simple string literal by Spring.

Exceptions

This rule does report syntactical errors in SpEL expressions but does not consider semantic errors, such as unknown identifiers or incompatible operand data types.

How_to_fix:

Correct the syntax error in the SpEL expression.

Noncompliant code example

```
@Value("#{systemProperties['user.region'}") // Noncompliant, unclosed "["
private String region;

@Value("#{'${listOfValues}' split(',')}") // Noncompliant, missing operator
private List<String> valuesList;

@Value("#{T(java.lang.Math).random() * 64h}") // Noncompliant, invalid number
private Double randPercent;

@Query("SELECT u FROM User u WHERE u.status = :#{#status+}") // Noncompliant, missing operand for "+"
List<User> findUsersByStatus(@Param("status") String status);
```

Compliant solution

```
@Value("#{systemProperties['user.region']}") // Compliant
private String region;

@Value("#{'${listOfValues}'.split(',')}") // Compliant
private List<String> valuesList;

@Value("#{T(java.lang.Math).random() * 100.0}") // Compliant
private Double randPercent;

@Query("SELECT u FROM User u WHERE u.status = :#{#status+42}") // Compliant
List<User> findUsersByStatus(@Param("status") String status);
```

Introduction:

This rule reports syntax errors in Spring Expression Language (SpEL) expressions.

Functions should not be defined with a variable number of arguments (java:S923)

Severidad: INFO

Root_cause:

As stated per effective java :

> Varargs methods are a convenient way to define methods that require a variable number of arguments, but they should not be overused. They can produce confusing results if used inappropriately.

Noncompliant code example

```
void fun ( String... strings )  // Noncompliant
{
  // ...
}
```

Resources:

- CERT, DCL57J - Avoid ambiguous overloading of variable arity methods

---

Public methods should throw at most one checked exception (java:S1160)

Severidad: MAJOR

Root_cause:

Using checked exceptions forces method callers to deal with errors, either by propagating them or by handling them. Throwing exceptions makes them fully part of the API of the method.

But to keep the complexity for callers reasonable, methods should not throw more than one kind of checked exception.

Noncompliant code example

```
public void delete() throws IOException, SQLException {      // Noncompliant
  /* ... */
}
```

Compliant solution

```
public void delete() throws SomeApplicationLevelException {
  /* ... */
}
```

Exceptions

Overriding methods are not checked by this rule and are allowed to throw several checked exceptions.

"@Override" should be used on overriding and implementing methods (java:S1161)

Severidad: MAJOR

Root_cause:

While not mandatory, using the `@Override` annotation on compliant methods improves readability by making it explicit that methods are overridden.

A compliant method either overrides a parent method or implements an interface or abstract method.

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 does not raise issues when overriding methods from `Object` (eg: `equals()`, `hashCode()`, `toString()`, …).

Checked exceptions should not be thrown (java:S1162)

Severidad: MAJOR

Root_cause:

The purpose of checked exceptions is to ensure that errors will be dealt with, either by propagating them or by handling them, but some believe that checked exceptions negatively impact the readability of source code, by spreading this error handling/propagation logic everywhere.

This rule verifies that no method throws a new checked exception.

Noncompliant code example

```
public void myMethod1() throws CheckedException {
  ...
  throw new CheckedException(message);   // Noncompliant
  ...
  throw new IllegalArgumentException(message); // Compliant; IllegalArgumentException is unchecked
}

public void myMethod2() throws CheckedException {  // Compliant; propagation allowed
  myMethod1();
}
```

Exceptions should not be thrown in finally blocks (java:S1163)

Severidad: CRITICAL

Resources:

- CERT, ERR05-J. - Do not let checked exceptions escape from a finally block

Root_cause:

If an exception is already being thrown within the `try` block or caught in a `catch` block, throwing another exception in the `finally` block will override the original exception. This means that the original exception’s message and stack trace will be lost, potentially making it challenging to diagnose and troubleshoot the root cause of the problem.

```
try {
  /* some work which end up throwing an exception */
  throw new IllegalArgumentException();
} finally {
  /* clean up */
  throw new RuntimeException();       // Noncompliant; masks the IllegalArgumentException
}

try {
  /* some work which end up throwing an exception */
  throw new IllegalArgumentException();
} finally {
  /* clean up */
}
```

Exception classes should have final fields (java:S1165)

Severidad: MINOR

Root_cause:

When a class has all `final` fields, the compiler ensures that the object’s state remains constant. It also enforces a clear design intent of immutability, making the class easier to reason about and use correctly.

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 these class fields do not change after initialization.

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;                   // Compliant

  public MyException(String message, int status) {
    super(message);
    this.status = status;
  }

  public int getStatus() {
    return status;
  }

}
```

Resources:

- Effective Java 3rd Edition, Joshua Bloch - Exceptions - Item 76 : Strive for failure atomicity

---

Exception handlers should preserve the original exceptions (java:S1166)

Severidad: MAJOR

Root_cause:

When handling a caught exception, the original exception's message and stack trace should be logged or passed forward.

Noncompliant code example

```
try {
  /* ... */
} catch (Exception e) {   // Noncompliant - exception is lost
  LOGGER.info("context");
}

try {
  /* ... */
} catch (Exception e) {  // Noncompliant - exception is lost (only message is preserved)
  LOGGER.info(e.getMessage());
}

try {
  /* ... */
} catch (Exception e) {  // Noncompliant - original exception is lost
  throw new RuntimeException("context");
}
```

Compliant solution

```
try {
  /* ... */
} catch (Exception e) {
  LOGGER.info(e);  // exception is logged
}
```

```
try {
  /* ... */
} catch (Exception e) {
  throw new RuntimeException(e);   // exception stack trace is propagated
}

try {
  /* ... */
} catch (RuntimeException e) {
  doSomething();
  throw e;  // original exception passed forward
} catch (Exception e) {
  throw new RuntimeException(e);  // Conversion into unchecked exception is also allowed
}
```

Exceptions

`InterruptedException`, `NumberFormatException`, `DateTimeParseException`, `ParseException` and `MalformedURLException` exceptions are arguably used to indicate nonexceptional outcomes. Similarly, handling `NoSuchMethodException` is often required when dealing with the Java reflection API.

Because they are part of Java, developers have no choice but to deal with them. This rule does not verify that those particular exceptions are correctly handled.

```
int myInteger;
try {
  myInteger = Integer.parseInt(myString);
} catch (NumberFormatException e) {
  // It is perfectly acceptable to not handle "e" here
  myInteger = 0;
}
```

Furthermore, no issue will be raised if the exception message is logged with additional information, as it shows that the developer added some context to the error message.

```
try {
  /* ... */
} catch (Exception e) {
  String message = "Exception raised while authenticating user: " + e.getMessage();
  LOGGER.warn(message); // Compliant - exception message logged with some contextual information
}
```

Resources:

- OWASP - Top 10 2021 Category A9 - Security Logging and Monitoring Failures
- OWASP - Top 10 2017 Category A10 - Insufficient Logging & Monitoring
- CERT, ERR00-J. - Do not suppress or ignore checked exceptions
- CWE - CWE-778 - Insufficient Logging

Parsing should be used to convert "Strings" to primitives (java:S2130)

Severidad: MINOR

Root_cause:

Rather than creating a boxed primitive from a `String` to extract the primitive value, use the relevant `parse` method instead. Using `parse` makes the code more efficient and the intent of the developer clearer.

Noncompliant code example

```
String myNum = "42.0";
float myFloat = new Float(myNum);  // Noncompliant
float myFloatValue = (new Float(myNum)).floatValue();  // Noncompliant
int myInteger = Integer.valueOf(myNum); // Noncompliant
int myIntegerValue = Integer.valueOf(myNum).intValue(); // Noncompliant
```

Compliant solution

```
String myNum = "42.0";
float f = Float.parseFloat(myNum);
int myInteger = Integer.parseInt(myNum);
```

Classes extending java.lang.Thread should provide a specific "run" behavior (java:S2134)

Severidad: MAJOR

How_to_fix:

To fix this issue, you have 2 options:

- override the `run` method

```
public class MyThread extends Thread {
  @Override
  public void run() {
    System.out.println("Hello, World!");
  }
}
```

- provide a `Runnable` at construction time

```
public class MyRunnable implements Runnable {
  @Override
  public void run() {
    System.out.println("Hello, World!");
  }
}
public class MyThread extends Thread {
  public MyThread(Runnable runnable) {
    super(runnable);
  }
}

public class Main() {
  public static void main(String [] args) {
    Runnable runnable = new MyRunnable();
    Thread customThread = new MyThread(runnable);
    Thread regularThread = new Thread(runnable);
  }
}
```

Root_cause:

The default implementation of `java.lang.Thread` 's `run` will only perform a task passed as a `Runnable`. If no `Runnable` has been provided at construction time, then the thread will not perform any action.

When extending `java.lang.Thread`, you should override the `run` method or pass a `Runnable` target to the constructor of `java.lang.Thread`.

Noncompliant code example

```
public class MyThread extends Thread { // Noncompliant
  public void doSomething() {
    System.out.println("Hello, World!");
  }
}
```

Exceptions should be either logged or rethrown but not both (java:S2139)

Severidad: MAJOR

Root_cause:

In applications where the accepted practice is to log an `Exception` and then rethrow it, you end up with miles-long logs that contain multiple instances of the same exception. In multi-threaded applications debugging this type of log can be particularly hellish because messages from other threads will be interwoven with the repetitions of the logged-and-thrown `Exception`. Instead, exceptions should be either logged or rethrown, not both.

Noncompliant code example

```
catch (SQLException e) {
  ...
  LOGGER.log(Level.ERROR,  contextInfo, e);
  throw new MySQLException(contextInfo, e);
}
```

Compliant solution

```
catch (SQLException e) {
  ...
  throw new MySQLException(contextInfo, e);
}
```

or

```
catch (SQLException e) {
  ...
  LOGGER.log(Level.ERROR,  contextInfo, e);
  // handle exception...
}
```

Collection methods with O(n) performance should be used carefully (java:S2250)

Severidad: MINOR

Root_cause:

The time complexity of method calls on collections is not always obvious. For instance, for most collections the `size()` method takes constant time, but the time required to execute `ConcurrentLinkedQueue.size()` is O(n), i.e. directly proportional to the number of elements in the collection. When the collection is large, this could therefore be an expensive operation.

This rule raises an issue when the following O(n) methods are called outside of constructors on class fields:

- `ArrayList`
  - `contains`
  - `remove`
- `LinkedList`
  - `get`
  - `contains`
- `ConcurrentLinkedQueue`
  - `size`
  - `contains`
- `ConcurrentLinkedDeque`
  - `size`
  - `contains`
- `CopyOnWriteArrayList`
  - `add`
  - `contains`
  - `remove`
- `CopyOnWriteArraySet`
  - `add`
  - `contains`
  - `remove`

Noncompliant code example

```
ConcurrentLinkedQueue queue = new ConcurrentLinkedQueue();
//...
log.info("Queue contains " + queue.size() + " elements");  // Noncompliant
```

A "for" loop update clause should move the counter in the right direction (java:S2251)

Severidad: MAJOR

Root_cause:

A `for` loop with a counter moving away from the end of the specified range is likely a programming mistake.

If the intention is to iterate over the specified range, this differs from what the loop does because the counter moves in the wrong direction.

If the intention is to have an infinite loop or a loop terminated only by a break statement, there are two problems:

1. The loop condition is not infinite because the counter variable will eventually overflow and fulfill the condition. This can take a long time, depending on the data type of the counter.
2. An infinite loop terminated by a `break` statement should be implemented using a `while` or `do while` loop to make the developer’s intention clear to the reader.

How_to_fix:

Noncompliant code example

Change the direction of the counter.

```
for (int i = 10; i > 0; i++) { // Noncompliant, wrong direction
  System.out.println("Hello, world!") // executed ca. 2 billion times
}
```

```
public void doSomething(String [] strings) {
  for (int i = 0; i < strings.length; i--) { // Noncompliant, wrong direction
    String string = strings[i];  // ArrayIndexOutOfBoundsException when i reaches -1
    // ...
  }
}
```

Compliant solution

```
for (int i = 10; i > 0; i--) { // Compliant
  System.out.println("Hello, world!") // executed 10 times
}

public void doSomething(String [] strings) {
  for (int i = 0; i < strings.length; i++) { // Compliant
    String string = strings[i];
    // ...
  }
}
```

Noncompliant code example

If the intention is to have an infinite loop or a loop terminated only by a break statement, use a `while` or a do `while` statement instead.

```
for (int i = 0; i < 0; i++) { // Noncompliant, loop is not infinite
  String event = waitForNextEvent();
  if (event == "terminate") break;
  processEvent(event);
}
```

Compliant solution

```
while (true) { // Compliant
  String event = waitForNextEvent();
  if (event == "terminate") break;
  processEvent(event);
}
```

Resources:

Documentation

- CERT, MSC54-J. - Avoid inadvertent wrapping of loop counters

Articles & blog posts

- Wikipedia - Integer overflow

Loop conditions should be true at least once (java:S2252)

Severidad: MAJOR

Resources:

Documentation

- Java Documentation - The `for` statement

Root_cause:

A `for` loop is a fundamental programming construct used to execute a block of code repeatedly. However, if the loop's condition is false before the first iteration, the loop will never execute.

```
for (int i = 0; i < 0; i++) {  // Noncompliant: the condition is always false, and the loop will never execute
    // ...
}
```

Rewrite the loop to ensure the condition evaluates to `true` at least once.

```
for (int i = 0; i < 10; i++) {  // Compliant: the condition is true at least once, the loop will execute
    // ...
}
```

This bug has the potential to cause unexpected outcomes as the loop might contain critical code that needs to be executed.

Track uses of disallowed methods (java:S2253)

Severidad: MAJOR

Root_cause:

This rule allows banning certain methods.

Noncompliant code example

Given parameters:

- className:java.lang.String
- methodName: replace
- argumentTypes: java.lang.CharSequence, java.lang.CharSequence

```
String name;
name.replace("A","a");  // Noncompliant
```

LDAP connections should be authenticated (java:S4433)

Severidad: CRITICAL

Introduction:

Lightweight Directory Access Protocol (LDAP) servers provide two main authentication methods: the *SASL* and *Simple* ones. The *Simple Authentication* method also breaks down into three different mechanisms:

- *Anonymous* Authentication
- *Unauthenticated* Authentication
- *Name/Password* Authentication

A server that accepts either the *Anonymous* or *Unauthenticated* mechanisms will accept connections from clients not providing credentials.

Resources:

Documentation

- RFC 4513 - Lightweight Directory Access Protocol (LDAP): Authentication Methods and Security Mechanisms - Bind operations

Standards

- OWASP - Top 10 2021 Category A7 - Identification and Authentication Failures
- OWASP - Top 10 2017 Category A2 - Broken Authentication
- CWE - CWE-521 - Weak Password Requirements

How_to_fix:

The following code indicates an anonymous LDAP authentication vulnerability because it binds to a remote server using an Anonymous Simple authentication mechanism.

Noncompliant code example

```
// Set up the environment for creating the initial context
Hashtable<String, Object> env = new Hashtable<String, Object>();
env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL, "ldap://localhost:389/o=JNDITutorial");

// Use anonymous authentication
env.put(Context.SECURITY_AUTHENTICATION, "none"); // Noncompliant

// Create the initial context
DirContext ctx = new InitialDirContext(env);
```

Compliant solution

```
// Set up the environment for creating the initial context
Hashtable<String, Object> env = new Hashtable<String, Object>();
env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL, "ldap://localhost:389/o=Example");

// Use simple authentication
env.put(Context.SECURITY_AUTHENTICATION, "simple");
env.put(Context.SECURITY_PRINCIPAL, "cn=local, ou=Unit, o=Example");
env.put(Context.SECURITY_CREDENTIALS, getLDAPPassword());

// Create the initial context
DirContext ctx = new InitialDirContext(env);
```

Root_cause:

When configured to accept the Anonymous or Unauthenticated authentication mechanism, an LDAP server will accept connections from clients that do not provide a password or other authentication credentials. Such users will be able to read or modify part or all of the data contained in the hosted directory.

What is the potential impact?

An attacker exploiting unauthenticated access to an LDAP server can access the data that is stored in the corresponding directory. The impact varies depending on the permission obtained on the directory and the type of data it stores.

Authentication bypass

If attackers get write access to the directory, they will be able to alter most of the data it stores. This might include sensitive technical data such as user passwords or asset configurations. Such an attack can typically lead to an authentication bypass on applications and systems that use the affected directory as an identity provider.

In such a case, all users configured in the directory might see their identity and privileges taken over.

Sensitive information leak

If attackers get read-only access to the directory, they will be able to read the data it stores. That data might include security-sensitive pieces of information.

Typically, attackers might get access to user account lists that they can use in further intrusion steps. For example, they could use such lists to perform password spraying, or related attacks, on all systems that rely on the affected directory as an identity provider.

If the directory contains some Personally Identifiable Information, an attacker accessing it might represent a violation of regulatory requirements in some countries. For example, this kind of security event would go against the European GDPR law.

Enum values should be compared with "==" (java:S4551)

Severidad: MAJOR

Resources:

- Use == (or !=) to Compare Java Enums

Root_cause:

Testing equality of an enum value with `equals` is perfectly valid because an enum is an Object and every Java developer knows "==" should not be used to compare the content of an Object. At the same time, using "==" on enums:

- provides the same expected comparison (content) as `equals`
- is more null-safe than equals()
- provides compile-time (static) checking rather than runtime checking

For these reasons, use of "==" should be preferred to `equals`.

Noncompliant code example

```
public enum Fruit {
   APPLE, BANANA, GRAPE
}

public enum Cake {
  LEMON_TART, CHEESE_CAKE
}

public boolean isFruitGrape(Fruit candidateFruit) {
  return candidateFruit.equals(Fruit.GRAPE); // Noncompliant; this will raise an NPE if candidateFruit is NULL
}

public boolean isFruitGrape(Cake candidateFruit) {
  return candidateFruit.equals(Fruit.GRAPE); // Noncompliant; always returns false
}
```

Compliant solution

```
public boolean isFruitGrape(Fruit candidateFruit) {
  return candidateFruit == Fruit.GRAPE; // Compliant; there is only one instance of Fruit.GRAPE - if candidateFruit is a GRAPE it will have
}

public boolean isFruitGrape(Cake candidateFruit) {
  return candidateFruit == Fruit.GRAPE; // Compliant; compilation time failure
}
```

Using weak hashing algorithms is security-sensitive (java:S4790)

Severidad: CRITICAL

Root_cause:

Cryptographic hash algorithms such as `MD2`, `MD4`, `MD5`, `MD6`, `HAVAL-128`, `HMAC-MD5`, `DSA` (which uses `SHA-1`), `RIPEMD`, `RIPEMD-128`, `RIPEMD-160`, `HMACRIPEMD160` and `SHA-1` are no longer considered secure, because it is possible to have `collisions` (little computational effort is enough to find two or more different inputs that produce the same hash).

How_to_fix:

Recommended Secure Coding Practices

Safer alternatives, such as `SHA-256`, `SHA-512`, `SHA-3` are recommended, and for password hashing, it’s even better to use algorithms that do not compute too "quickly", like `bcrypt`, `scrypt`, `argon2` or `pbkdf2` because it slows down `brute force attacks`.

Compliant Solution

```
MessageDigest md1 = MessageDigest.getInstance("SHA-512"); // Compliant
```

See

- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- OWASP - Mobile AppSec Verification Standard - Cryptography Requirements
- OWASP - Mobile Top 10 2016 Category M5 - Insufficient Cryptography
- CWE - CWE-1240 - Use of a Risky Cryptographic Primitive

Default:

Cryptographic hash algorithms such as `MD2`, `MD4`, `MD5`, `MD6`, `HAVAL-128`, `HMAC-MD5`, `DSA` (which uses `SHA-1`), `RIPEMD`, `RIPEMD-128`, `RIPEMD-160`, `HMACRIPEMD160` and `SHA-1` are no longer considered secure, because it is possible to have `collisions` (little computational effort is enough to find two or more different inputs that produce the same hash).

Ask Yourself Whether

The hashed value is used in a security context like:

- User-password storage.
- Security token generation (used to confirm e-mail when registering on a website, reset password, etc …).
- To compute some message integrity.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

Safer alternatives, such as `SHA-256`, `SHA-512`, `SHA-3` are recommended, and for password hashing, it’s even better to use algorithms that do not compute too "quickly", like `bcrypt`, `scrypt`, `argon2` or `pbkdf2` because it slows down `brute force attacks`.

Sensitive Code Example

```
MessageDigest md1 = MessageDigest.getInstance("SHA");  // Sensitive:  SHA is not a standard name, for most security providers it's an alias
MessageDigest md2 = MessageDigest.getInstance("SHA1");  // Sensitive
```

Compliant Solution

```
MessageDigest md1 = MessageDigest.getInstance("SHA-512"); // Compliant
```

See

- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- OWASP - Mobile AppSec Verification Standard - Cryptography Requirements
- OWASP - Mobile Top 10 2016 Category M5 - Insufficient Cryptography
- CWE - CWE-1240 - Use of a Risky Cryptographic Primitive

Assess_the_problem:

Ask Yourself Whether

The hashed value is used in a security context like:

- User-password storage.
- Security token generation (used to confirm e-mail when registering on a website, reset password, etc …).
- To compute some message integrity.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

```
MessageDigest md1 = MessageDigest.getInstance("SHA");  // Sensitive:  SHA is not a standard name, for most security providers it's an alias
MessageDigest md2 = MessageDigest.getInstance("SHA1");  // Sensitive
```

Server hostnames should be verified during SSL/TLS connections (java:S5527)

Severidad: CRITICAL

How_to_fix:

The following code contains examples of disabled hostname validation.

The hostname validation gets disabled because `mail.smtp.ssl.checkserveridentity` is omitted. To enable validation, set it to `true`.

Noncompliant code example

```
import java.util.Properties;

public Properties prepareEmailConnection() {
    Properties props = new Properties();

    props.put("mail.smtp.host", "smtp.gmail.com");
    props.put("mail.smtp.socketFactory.port", "465");
    props.put("mail.smtp.socketFactory.class", "javax.net.ssl.SSLSocketFactory"); // Noncompliant
    props.put("mail.smtp.auth", "true");
    props.put("mail.smtp.port", "465");

    return props;
}
```

Compliant solution

```
import java.util.Properties;

public Properties prepareEmailConnection() {
    Properties props = new Properties();

    props.put("mail.smtp.host", "smtp.gmail.com");
    props.put("mail.smtp.socketFactory.port", "465");
    props.put("mail.smtp.ssl.checkserveridentity", true);
    props.put("mail.smtp.socketFactory.class", "javax.net.ssl.SSLSocketFactory");
    props.put("mail.smtp.auth", "true");
    props.put("mail.smtp.port", "465");

    return props;
}
```

How does this work?

To fix the vulnerability of disabled hostname validation, it is strongly recommended to first re-enable the default validation and fix the root cause: the validity of the certificate.

Use valid certificates

If a hostname validation failure prevents connecting to the target server, keep in mind that **one system’s code should not work around another system’s problems**, as this creates unnecessary dependencies and can lead to reliability issues.

Therefore, the first solution is to change the remote host’s certificate to match its identity. If the remote host is not under your control, consider replicating its service to a server whose certificate you can change yourself.

In case the contacted host is located on a development machine, and if there is no other choice, try following this solution:

- Create a self-signed certificate for that machine.
- Add this self-signed certificate to the system’s trust store.
- If the hostname is not `localhost`, add the hostname in the `/etc/hosts` file.

Here is a sample command to import a certificate to the Java trust store:

```
keytool -import -alias myserver -file myserver.crt -keystore cacerts
```

How_to_fix:

The following code contains examples of disabled hostname validation.

The hostname validation gets disabled because `setSSLCheckServerIdentity` is omitted. To enable validation, set it to `true`.

Noncompliant code example

```
import org.apache.commons.mail.DefaultAuthenticator;
import org.apache.commons.mail.Email;
import org.apache.commons.mail.SimpleEmail;

public void sendMail(String message) {
    Email email = new SimpleEmail();

    email.setMsg(message);
    email.setSmtpPort(465);
    email.setAuthenticator(new DefaultAuthenticator(username, password));
    email.setSSLOnConnect(true); // Noncompliant

    email.send();
}
```

Compliant solution

```
import org.apache.commons.mail.DefaultAuthenticator;
import org.apache.commons.mail.Email;
import org.apache.commons.mail.SimpleEmail;

public void sendMail(String message) {
    Email email = new SimpleEmail();

    email.setMsg(message);
    email.setSmtpPort(465);
    email.setAuthenticator(new DefaultAuthenticator(username, password));
    email.setSSLCheckServerIdentity(true);
    email.setSSLOnConnect(true);

    email.send();
}
```

How does this work?

To fix the vulnerability of disabled hostname validation, it is strongly recommended to first re-enable the default validation and fix the root cause: the validity of the certificate.

Use valid certificates

If a hostname validation failure prevents connecting to the target server, keep in mind that **one system’s code should not work around another system’s problems**, as this creates unnecessary dependencies and can lead to reliability issues.

Therefore, the first solution is to change the remote host’s certificate to match its identity. If the remote host is not under your control, consider replicating its service to a server whose certificate you can change yourself.

In case the contacted host is located on a development machine, and if there is no other choice, try following this solution:

- Create a self-signed certificate for that machine.
- Add this self-signed certificate to the system’s trust store.
- If the hostname is not `localhost`, add the hostname in the `/etc/hosts` file.

Here is a sample command to import a certificate to the Java trust store:

```
keytool -import -alias myserver -file myserver.crt -keystore cacerts
```

Resources:

Standards

- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2021 Category A7 - Identification and Authentication Failures
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- OWASP - Mobile AppSec Verification Standard - Network Communication Requirements
- OWASP - Mobile Top 10 2016 Category M3 - Insecure Communication
- CWE - CWE-297 - Improper Validation of Certificate with Host Mismatch
- STIG Viewer - Application Security and Development: V-222550 - The application must validate certificates by constructing a certification path to an accepted trust anchor.
- https://wiki.sei.cmu.edu/confluence/display/java/MSC61-J.+Do+not+use+insecure+or+weak+cryptographic+algorithms

Root_cause:

Transport Layer Security (TLS) provides secure communication between systems over the internet by encrypting the data sent between them. In this process, the role of hostname validation, combined with certificate validation, is to ensure that a system is indeed the one it claims to be, adding an extra layer of trust and security.

When hostname validation is disabled, the client skips this critical check. This creates an opportunity for attackers to pose as a trusted entity and intercept, manipulate, or steal the data being transmitted.

To do so, an attacker would obtain a valid certificate authenticating `example.com`, serve it using a different hostname, and the application code would still accept it.

What is the potential impact?

Establishing trust in a secure way is a non-trivial task. When you disable hostname validation, you are removing a key mechanism designed to build this trust in internet communication, opening your system up to a number of potential threats.

Identity spoofing

If a system does not validate hostnames, it cannot confirm the identity of the other party involved in the communication. An attacker can exploit this by creating a fake server and masquerading it as a legitimate one. For example, they might set up a server that looks like your bank’s server, tricking your system into thinking it is communicating with the bank. This scenario, called identity spoofing, allows the attacker to collect any data your system sends to them, potentially leading to significant data breaches.

Introduction:

This vulnerability allows attackers to impersonate a trusted host.

How_to_fix:

The following code contains examples of disabled hostname validation.

The hostname validation gets disabled by overriding `javax.net.ssl.HostnameVerifier.verify()` with an empty implementation. It is highly recommended to use the original implementation.

Noncompliant code example

```
import java.io.InputStream;
import java.net.URL;
import javax.net.ssl.HttpsURLConnection;
import javax.net.ssl.SSLSession;
import javax.net.ssl.HostnameVerifier;

public InputStream doRequest() {
    URL url                          = new URL("https://example.org/");
    HttpsURLConnection urlConnection = (HttpsURLConnection)url.openConnection();

    urlConnection.setHostnameVerifier(new HostnameVerifier() {
      @Override
      public boolean verify(String requestedHost, SSLSession remoteServerSession) {
        return true;  // Noncompliant
      }
    });
    return urlConnection.getInputStream();
}
```

Compliant solution

```
import java.io.InputStream;
import java.net.URL;
import javax.net.ssl.HttpsURLConnection;
import javax.net.ssl.SSLSession;

public InputStream doRequest() {
    URL url                          = new URL("https://example.org/");
    HttpsURLConnection urlConnection = (HttpsURLConnection)url.openConnection();

    return urlConnection.getInputStream();
}
```

How does this work?

To fix the vulnerability of disabled hostname validation, it is strongly recommended to first re-enable the default validation and fix the root cause: the validity of the certificate.

Use valid certificates

If a hostname validation failure prevents connecting to the target server, keep in mind that **one system's code should not work around another system's problems**, as this creates unnecessary dependencies and can lead to reliability issues.

Therefore, the first solution is to change the remote host's certificate to match its identity. If the remote host is not under your control, consider replicating its service to a server whose certificate you can change yourself.

In case the contacted host is located on a development machine, and if there is no other choice, try following this solution:

- Create a self-signed certificate for that machine.
- Add this self-signed certificate to the system's trust store.
- If the hostname is not `localhost`, add the hostname in the `/etc/hosts` file.

Here is a sample command to import a certificate to the Java trust store:

```
keytool -import -alias myserver -file myserver.crt -keystore cacerts
```

Literal suffixes should be upper case (java:S818)

Severidad: MINOR

Root_cause:

Using upper case literal suffixes removes the potential ambiguity between "1" (digit 1) and "l" (letter el) for declaring literals.

Noncompliant code example

```
long long1 = 1l; // Noncompliant
float float1 = 1.0f; // Noncompliant
double double1 = 1.0d; // Noncompliant
```

Compliant solution

```
long long1 = 1L;
float float1 = 1.0F;
double double1 = 1.0D;
```

Resources:

- CERT DCL16-C. - Use "L," not "l," to indicate a long value
- CERT, DCL50-J. - Use visually distinct identifiers

"switch" statements should have at least 3 "case" clauses (java:S1301)

Severidad: MINOR

Root_cause:

`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();
}
```

Track uses of "@SuppressWarnings" annotations (java:S1309)

Severidad: INFO

Root_cause:

This rule allows you to track the usage of the `@SuppressWarnings` mechanism.

Noncompliant code example

With a parameter value of "unused" :

```
@SuppressWarnings("unused")
@SuppressWarnings("unchecked")  // Noncompliant
```

Methods should not be too complex (java:S1541)

Severidad: CRITICAL

Root_cause:

The cyclomatic complexity of methods should not exceed a defined threshold.

Complex code can perform poorly and will in any case be difficult to understand and therefore to maintain.

Exceptions

While having a large number of fields in a class may indicate that it should be split, this rule nonetheless ignores high complexity in `equals` and `hashCode` methods.

Variables should not be self-assigned (java:S1656)

Severidad: MAJOR

Resources:

- CERT, MSC12-C. - Detect and remove code that has no effect or is never executed

Root_cause:

There is no reason to re-assign a variable to itself. Either this statement is redundant and should be removed, or the re-assignment is a mistake and some other value or variable was intended for the assignment instead.

Noncompliant code example

```
public void setName(String name) {
  name = name;
}
```

Compliant solution

```
public void setName(String name) {
  this.name = name;
}
```

Multiple variables should not be declared on the same line (java:S1659)

Severidad: MINOR

Resources:

- CERT, DCL52-J. - Do not declare more than one variable per declaration
- CERT, DCL04-C. - Do not declare more than one variable per declaration

Root_cause:

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;
  }
}
```

Loops with at most one iteration should be refactored (java:S1751)

Severidad: MAJOR

Resources:

Documentation

- Oracle - The for Statement

Root_cause:

A loop with at most one iteration is equivalent to an `if` statement. This can confuse developers and make the code less readable since loops are not meant to replace `if` statements.

If the intention was to conditionally execute the block only once, an `if` statement should be used instead. Otherwise, the loop should be fixed so the loop block can be executed multiple times.

A loop statement with at most one iteration can happen when a statement that unconditionally transfers control, such as a jump or throw statement, is misplaced inside the loop block.

This rule arises when the following statements are misplaced:

- `break`
- `return`
- `throw`

How_to_fix:

Noncompliant code example

```
int i = 0;
while(i < 10) { // Noncompliant; loop only executes once
  System.out.println("i is " + i);
  i++;
  break;
}

for (int i = 0; i < 10; i++) { // Noncompliant; loop only executes once
  if (i == x) {
    break;
  } else {
    System.out.println("i is " + i);
    return;
  }
}
```

Compliant solution

```
int i = 0;
while (i < 10) {
  System.out.println("i is " + i);
  i++;
}

for (int i = 0; i < 10; i++) {
  if (i == x) {
    break;
  } else {
    System.out.println("i is " + i);
  }
}
```

The ternary operator should not be used (java:S1774)

Severidad: MAJOR

Root_cause:

Ternary expressions, while concise, can often lead to code that is difficult to read and understand, especially when they are nested or complex. Prioritizing readability fosters maintainability and reduces the likelihood of bugs. Therefore, they should be removed in favor of more explicit control structures, such as `if`/`else` statements, to improve the clarity and readability of the code.

Noncompliant code example

```
System.out.println(i>10?"yes":"no");  // Noncompliant
```

Compliant solution

```
if (i > 10) {
  System.out.println("yes");
} else {
  System.out.println("no");
}
```

Two branches in a conditional structure should not have exactly the same implementation (java:S1871)

Severidad: MAJOR

Resources:

Related rules

- S3923 - All branches in a conditional structure should not have exactly the same implementation

Root_cause:

When the same code is duplicated in two or more separate branches of a conditional, it can make the code harder to understand, maintain, and can potentially introduce bugs if one instance of the code is changed but others are not.

Having two `cases` in a `switch` statement or two branches in an `if` chain with the same implementation is at best duplicate code, and at worst a coding error.

```
if (a >= 0 && a < 10) {
  doFirstThing();
  doTheThing();
}
else if (a >= 10 && a < 20) {
  doTheOtherThing();
}
else if (a >= 20 && a < 50) {
  doFirstThing();
  doTheThing();  // Noncompliant; duplicates first condition
}
else {
  doTheRest();
}

switch (i) {
  case 1:
    doFirstThing();
    doSomething();
    break;
  case 2:
    doSomethingDifferent();
    break;
  case 3:  // Noncompliant; duplicates case 1's implementation
    doFirstThing();
    doSomething();
    break;
  default:
    doTheRest();
}
```

If the same logic is truly needed for both instances, then:

- in an `if` chain they should be combined

```
if ((a >= 0 && a < 10) || (a >= 20 && a < 50)) { // Compliant
  doFirstThing();
  doTheThing();
}
else if (a >= 10 && a < 20) {
  doTheOtherThing();
}
else {
  doTheRest();
}
```

- for a `switch`, one should fall through to the other

```
switch (i) {
  case 1:
  case 3: // Compliant
    doFirstThing();
    doSomething();
    break;
  case 2:
    doSomethingDifferent();
    break;
  default:
    doTheRest();
}
```

When all blocks are identical, either this rule will trigger if there is no default clause or rule S3923 will raise if there is a default clause.

Exceptions

Unless all blocks are identical, blocks in an `if` chain that contain a single line of code are ignored. The same applies to blocks in a `switch` statement that contains a single line of code with or without a following `break`.

```
if (a == 1) {
  doSomething();  // Compliant, usually this is done on purpose to increase the readability
} else if (a == 2) {
  doSomethingElse();
} else {
  doSomething();
}
```

Classes should not be compared by name (java:S1872)

Severidad: MAJOR

Root_cause:

There is no requirement that class names be unique, only that they be unique within a package. Therefore trying to determine an object’s type based on its class name is an exercise fraught with danger. One of those dangers is that a malicious user will send objects of the same name as the trusted class and, thereby gain trusted access.

Instead, the `instanceof` operator or the `Class.isAssignableFrom()` method should be used to check the object’s underlying type.

Noncompliant code example

```
package computer;
class Pear extends Laptop { ... }

package food;
class Pear extends Fruit { ... }

class Store {

  public boolean hasSellByDate(Object item) {
    if ("Pear".equals(item.getClass().getSimpleName())) {  // Noncompliant
      return true;  // Results in throwing away week-old computers
    }
    return false;
  }

  public boolean isList(Class<T> valueClass) {
    if (List.class.getName().equals(valueClass.getName())) {  // Noncompliant
      return true;
    }
    return false;
  }
}
```

Compliant solution

```
class Store {

  public boolean hasSellByDate(Object item) {
    if (item instanceof food.Pear) {
      return true;
    }
    return false;
  }

  public boolean isList(Class<T> valueClass) {
    if (valueClass.isAssignableFrom(List.class)) {
      return true;
    }
    return false;
  }
}
```

Resources:

- CWE - CWE-486 - Comparison of Classes by Name
- CERT, OBJ09-J. - Compare classes and not class names

"@Deprecated" code should not be used (java:S1874)

Severidad: MINOR

Root_cause:

Code is sometimes annotated as deprecated by developers maintaining libraries or APIs to indicate that the method, class, or other programming element is no longer recommended for use. This is typically due to the introduction of a newer or more effective alternative. For example, when a better solution has been identified, or when the existing code presents potential errors or security risks.

Deprecation is a good practice because it helps to phase out obsolete code in a controlled manner, without breaking existing software that may still depend on it. It is a way to warn other developers not to use the deprecated element in new code, and to replace it in existing code when possible.

Deprecated classes, interfaces, and their members should not be used, inherited or extended because they will eventually be removed. The deprecation period allows you to make a smooth transition away from the aging, soon-to-be-retired technology.

Check the documentation or the deprecation message to understand why the code was deprecated and what the recommended alternative is.

```
/**
 * @deprecated  As of release 1.3, replaced by {@link #Foo}
 */
@Deprecated
public class Fum { ... }

public class Foo {
  /**
```

```
   * @deprecated  As of release 1.7, replaced by {@link #newMethod()}
   */
  @Deprecated
  public void oldMethod() { ... }

  public void newMethod() { ... }
}

public class Bar extends Foo {
  public void oldMethod() { ... } // Noncompliant; don't override a deprecated method
}

public class Baz extends Fum {  // Noncompliant; Fum is deprecated
  public void myMethod() {
    Foo foo = new Foo();
    foo.oldMethod();  // Noncompliant; oldMethod method is deprecated
  }
}
```

Resources:

Documentation

- CWE - CWE-477 - Use of Obsolete Functions

"for" loop increment clauses should modify the loops' counters (java:S1994)

Severidad: CRITICAL

Root_cause:

The counter of a `for` loop should be updated in the loop’s increment clause. The purpose of a `for` loop is to iterate over a range using a counter variable. It should not be used for other purposes, and alternative loops should be used in those cases.

If the counter is not updated, the loop will be infinite with a constant counter variable. If this is intentional, use a `while` or `do while` loop instead of a `for` loop.

If the counter variable is updated within the loop’s body, try to move it to the increment clause. If this is impossible due to certain conditions, replace the `for` loop with a `while` or `do while` loop.

How_to_fix:

Noncompliant code example

Move the counter variable update to the loop’s increment clause.

```
for (int i = 0; i < 10; ) { // Noncompliant, i not updated in increment clause
  // ...
  i++;
}

int sum = 0
for (int i = 0; i < 10; sum++) { // Noncompliant, i not updated in increment clause
  // ...
  i++;
}
```

Compliant solution

```
for (i = 0; i < 10; i++) { // Compliant
  // ...
}

int sum = 0
for (int i = 0; i < 10; i++) { // Compliant
  // ...
  sum++;
}
```

Noncompliant code example

If this is impossible and the counter variable must be updated in the loop’s body, use a `while` or `do while` loop instead.

```
for (int sum = 0; sum < 10) { // Noncompliant, sum not updated in increment clause
  // ...
  if (condition) sum++;
  // ...
}
```

Compliant solution

```
int sum = 0;
while (sum < 10) { // Compliant
  // ...
  if (condition) sum++;
  // ...
}
```

Files should contain only one top-level class or interface each (java:S1996)

Severidad: MAJOR

Root_cause:

A file that grows too much tends to aggregate too many responsibilities and inevitably becomes harder to understand and therefore to maintain. This is doubly true for a file with multiple top-level classes and interfaces. It is strongly advised to divide the file into one top-level class or interface per file.

"Preconditions" and logging arguments should not require evaluation (java:S2629)

Severidad: MAJOR

Root_cause:

Some method calls can effectively be "no-ops", meaning that the invoked method does nothing, based on the application’s configuration (eg: debug logs in production). However, even if the method effectively does nothing, its arguments may still need to evaluated before the method is called.

Passing message arguments that require further evaluation into a Guava `com.google.common.base.Preconditions` check can result in a performance penalty. That is because whether or not they’re needed, each argument must be resolved before the method is actually called.

Similarly, passing concatenated strings into a logging method can also incur a needless performance hit because the concatenation will be performed every time the method is called, whether or not the log level is low enough to show the message.

Instead, you should structure your code to pass static or pre-computed values into `Preconditions` conditions check and logging calls.

Specifically, the built-in string formatting should be used instead of string concatenation, and if the message is the result of a method call, then `Preconditions` should be skipped altogether, and the relevant exception should be conditionally thrown instead.

Noncompliant code example

```
logger.log(Level.DEBUG, "Something went wrong: " + message);  // Noncompliant; string concatenation performed even when log level too high

logger.fine("An exception occurred with message: " + message); // Noncompliant

LOG.error("Unable to open file " + csvPath, e);  // Noncompliant

Preconditions.checkState(a > 0, "Arg must be positive, but got " + a);  // Noncompliant. String concatenation performed even when a > 0

Preconditions.checkState(condition, formatMessage());  // Noncompliant. formatMessage() invoked regardless of condition

Preconditions.checkState(condition, "message: %s", formatMessage());  // Noncompliant
```

Compliant solution

```
logger.log(Level.DEBUG, "Something went wrong: {0} ", message);  // String formatting only applied if needed
logger.log(Level.SEVERE, () -> "Something went wrong: " + message); // since Java 8, we can use Supplier , which will be evaluated lazily

logger.fine("An exception occurred with message: {}", message);  // SLF4J, Log4j

LOG.error("Unable to open file {0}", csvPath, e);

if (LOG.isDebugEnabled()) {
  LOG.debug("Unable to open file " + csvPath, e);  // this is compliant, because it will not evaluate if log level is above debug.
}

Preconditions.checkState(arg > 0, "Arg must be positive, but got %d", a);  // String formatting only applied if needed

if (!condition) {
  throw new IllegalStateException(formatMessage());  // formatMessage() only invoked conditionally
}

if (!condition) {
  throw new IllegalStateException("message: " + formatMessage());
}
```

Exceptions

`catch` blocks are ignored, because the performance penalty is unimportant on exceptional paths (catch block should not be a part of standard program flow). Getters are ignored as well as methods called on annotations which can be considered as getters. This rule accounts for explicit test-level testing with SLF4J methods `isXXXEnabled` and ignores the bodies of such `if` statements.

"@NonNull" values should not be set to null (java:S2637)

Severidad: MINOR

Root_cause:

Fields, parameters and return values marked `@NotNull`, `@NonNull`, or `@Nonnull` are assumed to have non-null values and are not typically null-checked before use. Therefore setting one of these values to `null`, or failing to set such a class field in a constructor, could cause `NullPointerExceptions` at runtime.

Noncompliant code example

```
public class MainClass {

  @Nonnull
  private String primary;
  private String secondary;

  public MainClass(String color) {
    if (color != null) {
      secondary = null;
    }
    primary = color;  // Noncompliant; "primary" is Nonnull but could be set to null here
  }

  public MainClass() { // Noncompliant; "primary" is Nonnull but is not initialized
  }

  @Nonnull
  public String indirectMix() {
    String mix = null;
    return mix;  // Noncompliant; return value is Nonnull, but null is returned.
  }
```

Resources:

Standards

- CERT - EXP34-C. Do not dereference null pointers
- CERT - EXP01-J. Do not use a null in a case where an object is required
- CWE - CWE-476 NULL Pointer Dereference

Method overrides should not change contracts (java:S2638)

Severidad: CRITICAL

Introduction:

This rule raises an issue when an overriding method changes a contract defined in a superclass.

Resources:

Documentation

- SOLID - Wikipedia - Liskov substitution principle

Root_cause:

Because a subclass instance may be cast to and treated as an instance of the superclass, overriding methods should uphold the aspects of the superclass contract that relate to the Liskov Substitution Principle. Specifically, if the parameters or return type of the superclass method are marked with any of the following: `@Nullable`, `@CheckForNull`, `@NotNull`, `@NonNull`, and `@Nonnull`, then subclass parameters are not allowed to tighten the contract, and return values are not allowed to loosen it.

Noncompliant code example

```
public class Fruit {

  private Season ripe;
  private String color;

  public void setRipe(@Nullable Season ripe) {
    this.ripe = ripe;
  }

  public @NotNull Integer getProtein() {
    return 12;
  }
}

public class Raspberry extends Fruit {

  public void setRipe(@NotNull Season ripe) {  // Noncompliant: the ripe argument annotated as @Nullable in parent class
    this.ripe = ripe;
  }

  public @Nullable Integer getProtein() {  // Noncompliant: the return type annotated as @NotNull in parent class
    return null;
  }
}
```

Compliant solution

```
public class Fruit {

  private Season ripe;
  private String color;

  public void setRipe(@Nullable Season ripe) {
    this.ripe = ripe;
  }

  public @NotNull Integer getProtein() {
    return 12;
  }
}

public class Raspberry extends Fruit {

  public void setRipe(@Nullable Season ripe) {
    this.ripe = ripe;
  }

  public @NotNull Integer getProtein() {
    return 12;
  }
}
```

Inappropriate regular expressions should not be used (java:S2639)

Severidad: MAJOR

Root_cause:

Regular expressions are powerful but tricky, and even those long used to using them can make mistakes.

The following should not be used as regular expressions:

- . - matches any single character. Used in `replaceAll`, it matches *everything*
- | - normally used as an option delimiter. Used stand-alone, it matches the space between characters
- `File.separator` - matches the platform-specific file path delimiter. On Windows, this will be taken as an escape character

Noncompliant code example

```
String str = "/File|Name.txt";

String clean = str.replaceAll(".",""); // Noncompliant; probably meant to remove only dot chars, but returns an empty string
String clean2 = str.replaceAll("|","_"); // Noncompliant; yields _/_F_i_l_e_|_N_a_m_e_._t_x_t_
String clean3 = str.replaceAll(File.separator,""); // Noncompliant; exception on Windows

String clean4 = str.replaceFirst(".",""); // Noncompliant;
String clean5 = str.replaceFirst("|","_"); // Noncompliant;
String clean6 = str.replaceFirst(File.separator,""); // Noncompliant;
```

XML parsers should not be vulnerable to XXE attacks (java:S2755)

Severidad: BLOCKER

How_to_fix:

The following code contains examples of XML parsers that have external entity processing enabled. As a result, the parsers are vulnerable to XXE attacks if an attacker can control the XML file that is processed.

Noncompliant code example

```
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLReaderFactory;

public void decode() {
    XMLReader reader = XMLReaderFactory.createXMLReader(); // Noncompliant
}
```

Compliant solution

Set `disallow-doctype-decl` to `true`.

```
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLReaderFactory;

public void decode() {
    XMLReader reader = XMLReaderFactory.createXMLReader();
    reader.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
}
```

How does this work?

Disable external entities

The most effective approach to prevent XXE vulnerabilities is to disable external entity processing entirely, unless it is explicitly required for specific use cases. By default, XML parsers should be configured to reject the processing of external entities. This can be achieved by setting the appropriate properties or options in your XML parser library or framework.

If external entity processing is necessary for certain scenarios, adopt a whitelisting approach to restrict the entities that can be resolved during XML parsing. Create a list of trusted external entities and disallow all others. This approach ensures that only known and safe entities are processed.
You should rely on features provided by your XML parser to restrict the external entities.

How_to_fix:

The following code contains examples of XML parsers that have external entity processing enabled. As a result, the parsers are vulnerable to XXE attacks if an attacker can control the XML file that is processed.

Noncompliant code example

```
import javax.xml.XMLConstants;
import javax.xml.parsers.DocumentBuilderFactory;

public void decode() {
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance(); // Noncompliant
}
```

```
import javax.xml.stream.XMLInputFactory;

public void decode() {
    XMLInputFactory factory = XMLInputFactory.newInstance(); // Noncompliant
}
```

Compliant solution

For `DocumentBuilderFactory`, `SAXParserFactory`, `TransformerFactory`, and `SchemaFactory` set `XMLConstants.FEATURE_SECURE_PROCESSING` to `true`.

```
import javax.xml.XMLConstants;
import javax.xml.parsers.DocumentBuilderFactory;

public void decode() {
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    factory.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);
}
```

For `XMLInputFactory` set `SUPPORT_DTD` to `false`.

```
import javax.xml.stream.XMLInputFactory;

public void decode() {
    XMLInputFactory factory = XMLInputFactory.newInstance();
    factory.setProperty(XMLInputFactory.SUPPORT_DTD, false);
}
```

Other combinations of settings are secure, but in general, it is recommendable to use the approaches shown here, as they are the most clear.

How does this work?

Disable external entities

The most effective approach to prevent XXE vulnerabilities is to disable external entity processing entirely, unless it is explicitly required for specific use cases. By default, XML parsers should be configured to reject the processing of external entities. This can be achieved by setting the appropriate properties or options in your XML parser library or framework.

If external entity processing is necessary for certain scenarios, adopt a whitelisting approach to restrict the entities that can be resolved during XML parsing. Create a list of trusted external entities and disallow all others. This approach ensures that only known and safe entities are processed.
You should rely on features provided by your XML parser to restrict the external entities.

Going the extra mile

Disable entity expansion

Specifically for `DocumentBuilderFactory`, it is possible to disable the entity expansion. Note, however, that this does not prevent the retrieval of external entities.

```
factory.setExpandEntityReferences(false);
```

Resources:

Standards

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2017 Category A4 - XML External Entities (XXE)
- CWE - CWE-611 - Information Exposure Through XML External Entity Reference
- CWE - CWE-827 - Improper Control of Document Type Definition
- STIG Viewer - Application Security and Development: V-222608 - The application must not be vulnerable to XML-oriented attacks.

Introduction:

This vulnerability allows the usage of external entities in XML.

Root_cause:

External Entity Processing allows for XML parsing with the involvement of external entities. However, when this functionality is enabled without proper precautions, it can lead to a vulnerability known as XML External Entity (XXE) attack.

What is the potential impact?

Exposing sensitive data

One significant danger of XXE vulnerabilities is the potential for sensitive data exposure. By crafting malicious XML payloads, attackers can reference external entities that contain sensitive information, such as system files, database credentials, or configuration files. When these entities are processed during XML parsing, the attacker can extract the contents and gain unauthorized access to sensitive data. This poses a severe threat to the confidentiality of critical information.

Exhausting system resources

Another consequence of XXE vulnerabilities is the potential for denial-of-service attacks. By exploiting the ability to include external entities, attackers can construct XML payloads that cause resource exhaustion. This can overwhelm the system’s memory, CPU, or other critical resources, leading to system unresponsiveness or crashes. A successful DoS attack can disrupt the availability of services and negatively impact the user experience.

Forging requests

XXE vulnerabilities can also enable Server-Side Request Forgery (SSRF) attacks. By leveraging the ability to include external entities, an attacker can make the vulnerable application send arbitrary requests to other internal or external systems. This can result in unintended actions, such as retrieving data from internal resources, scanning internal networks, or attacking other systems. SSRF attacks can lead to severe consequences, including unauthorized data access, system compromise, or even further exploitation within the network infrastructure.

How_to_fix:

The following code contains examples of XML parsers that have external entity processing enabled. As a result, the parsers are vulnerable to XXE attacks if an attacker can control the XML file that is processed.

Noncompliant code example

```
import org.jdom2.input.SAXBuilder;

public void decode() {
    SAXBuilder builder = new SAXBuilder(); // Noncompliant
}
```

Compliant solution

```
import org.jdom2.input.SAXBuilder;

public void decode() {
    SAXBuilder builder = new SAXBuilder();
    builder.setProperty(XMLConstants.ACCESS_EXTERNAL_DTD, "");
}
```

How does this work?

Disable external entities

The most effective approach to prevent XXE vulnerabilities is to disable external entity processing entirely, unless it is explicitly required for specific use cases. By default, XML parsers should be configured to reject the processing of external entities. This can be achieved by setting the appropriate properties or options in your XML parser library or framework.

If external entity processing is necessary for certain scenarios, adopt a whitelisting approach to restrict the entities that can be resolved during XML parsing. Create a list of trusted external entities and disallow all others. This approach ensures that only known and safe entities are processed.
You should rely on features provided by your XML parser to restrict the external entities.

How_to_fix:

The following code contains examples of XML parsers that have external entity processing enabled. As a result, the parsers are vulnerable to XXE attacks if an attacker can control the XML file that is processed.

Noncompliant code example

```
import org.dom4j.io.SAXReader;

public void decode() {
    SAXReader xmlReader = new SAXReader(); // Noncompliant
}
```

Compliant solution

```
import org.dom4j.io.SAXReader;

public void decode() {
    SAXReader xmlReader = new SAXReader();
    xmlReader.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
}
```

How does this work?

Disable external entities

The most effective approach to prevent XXE vulnerabilities is to disable external entity processing entirely, unless it is explicitly required for specific use cases. By default, XML parsers should be configured to reject the processing of external entities. This can be achieved by setting the appropriate properties or options in your XML parser library or framework.

If external entity processing is necessary for certain scenarios, adopt a whitelisting approach to restrict the entities that can be resolved during XML parsing. Create a list of trusted external entities and disallow all others. This approach ensures that only known and safe entities are processed.
You should rely on features provided by your XML parser to restrict the external entities.

Non-existent operators like "=+" should not be used (java:S2757)

Severidad: MAJOR

Root_cause:

Using operator pairs (`=+`, `=-`, or `=!`) that look like reversed single operators (`+=`, `-=` or `!=`) is confusing. They compile and run but do not produce the same result as their mirrored counterpart.

```
int target = -5;
int num = 3;

target =- num;  // Noncompliant: target = -3. Is that the intended behavior?
target =+ num; // Noncompliant: target = 3
```

This rule raises an issue when `=+`, `=-`, or `=!` are used without any space between the operators and when there is at least one whitespace after.

Replace the operators with a single one if that is the intention

```
int target = -5;
int num = 3;

target -= num;  // target = -8
```

Or fix the spacing to avoid confusion

```
int target = -5;
int num = 3;

target = -num;  // target = -3
```

"entrySet()" should be iterated when both the key and value are needed (java:S2864)

Severidad: MAJOR

Root_cause:

`Map` is an object that maps keys to values. A map cannot contain duplicate keys, which means each key can map to at most one value.

When both the key and the value are needed, it is more efficient to iterate the `entrySet()`, which will give access to both instead of iterating over the `keySet()` and then getting the value.

If the `entrySet()` method is not iterated when both the key and value are needed, it can lead to unnecessary lookups. This is because each lookup requires two operations: one to retrieve the key and another to retrieve the value. By iterating the `entrySet()` method, the key-value pair can be retrieved in a single operation, which can improve performance.

Noncompliant code example

```
public void doSomethingWithMap(Map<String,Object> map) {
  for (String key : map.keySet()) {  // Noncompliant; for each key the value is retrieved
    Object value = map.get(key);
    // ...
  }
}
```

Compliant solution

```
public void doSomethingWithMap(Map<String,Object> map) {
  for (Map.Entry<String,Object> entry : map.entrySet()) {
    String key = entry.getKey();
    Object value = entry.getValue();
    // ...
  }
}
```

Resources:

Documentation

- Oracle SE 20 - Map

Articles & blog posts

- Baeldung - Java Map methods

Java 8's "Files.exists" should not be used (java:S3725)

Severidad: MAJOR

Root_cause:

The `Files.exists` method has noticeably poor performance in JDK 8, and can slow an application significantly when used to check files that don't actually exist.

The same goes for `Files.notExists`, `Files.isDirectory` and `Files.isRegularFile` from `java.nio.file` package.

Note that this rule is automatically disabled when the project's `sonar.java.source` is not 8.

Noncompliant code example

```
Path myPath;
if(java.nio.file.Files.exists(myPath)) {  // Noncompliant
 // do something
}
```

Compliant solution

```
Path myPath;
if(myPath.toFile().exists())) {
 // do something
}
```

Resources:

- https://bugs.openjdk.java.net/browse/JDK-8153414
- https://bugs.openjdk.java.net/browse/JDK-8154077

Number patterns should be regular (java:S3937)

Severidad: CRITICAL

Root_cause:

The use of punctuation characters to separate subgroups in a number can make the number more readable. For instance consider 1,000,000,000 versus 1000000000. But when the grouping is irregular, such as 1,000,00,000; it indicates an error.

This rule raises an issue when underscores (_) are used to break a number into irregular subgroups.

Noncompliant code example

```
int thousand = 100_0;
int tenThousand = 100_00;
int million = 1_000_00_000;
```

Compliant solution

```
int thousand = 1000;
int tenThousand = 10_000;
int tenThousandWithout = 10000;
int duos = 1_00_00;
int million = 100_000_000;
```

Exceptions

No issue will be raised on binary numbers (starting with `0b` or `0B`). Binary number bits are often grouped corresponding to certain meanings, resulting in irregular bit group sizes.

```
int configValue1 = 0b00_000_10_1; // Compliant
int configValue2 = 0B00_000_10_1; // Compliant
```

Intermediate Stream methods should not be left unused (java:S3958)

Severidad: MAJOR

Resources:

- Stream Operations

Root_cause:

There are two types of stream operations: intermediate operations, which return another stream, and terminal operations, which return something other than a stream. Intermediate operations are lazy, meaning they aren't actually executed until and unless a terminal stream operation is performed on their results. Consequently, if the result of an intermediate stream operation is not fed to a terminal operation, it serves no purpose, which is almost certainly an error.

Noncompliant code example

```
widgets.stream().filter(b -> b.getColor() == RED); // Noncompliant
```

Compliant solution

```
int sum = widgets.stream()
                      .filter(b -> b.getColor() == RED)
                      .mapToInt(b -> b.getWeight())
                      .sum();
Stream<Widget> pipeline = widgets.stream()
                                 .filter(b -> b.getColor() == GREEN)
                                 .mapToInt(b -> b.getWeight());
sum = pipeline.sum();
```

Consumed Stream pipelines should not be reused (java:S3959)

Severidad: MAJOR

Resources:

Stream Operations

Root_cause:

Stream operations are divided into intermediate and terminal operations, and are combined to form stream pipelines. After the terminal operation is performed, the stream pipeline is considered consumed, and cannot be used again. Such a reuse will yield unexpected results.

Noncompliant code example

```
Stream<Widget> pipeline = widgets.stream().filter(b -> b.getColor() == RED);
int sum1 = pipeline.sum();
int sum2 = pipeline.mapToInt(b -> b.getWeight()).sum(); // Noncompliant
```

"Class.forName()" should not load JDBC 4.0+ drivers (java:S4925)

Severidad: MAJOR

Root_cause:

In the past, it was required to load a JDBC driver before creating a `java.sql.Connection`. Nowadays, when using JDBC 4.0 drivers, this is no longer required and `Class.forName()` can be safely removed because JDBC 4.0 (JDK 6) drivers available in the classpath are automatically loaded.

This rule raises an issue when `Class.forName()` is used with one of the following values:

- `com.mysql.jdbc.Driver`
- `oracle.jdbc.driver.OracleDriver`
- `com.ibm.db2.jdbc.app.DB2Driver`
- `com.ibm.db2.jdbc.net.DB2Driver`
- `com.sybase.jdbc.SybDriver`
- `com.sybase.jdbc2.jdbc.SybDriver`
- `com.teradata.jdbc.TeraDriver`
- `com.microsoft.sqlserver.jdbc.SQLServerDriver`
- `org.postgresql.Driver`
- `sun.jdbc.odbc.JdbcOdbcDriver`
- `org.hsqldb.jdbc.JDBCDriver`
- `org.h2.Driver`
- `org.firebirdsql.jdbc.FBDriver`
- `net.sourceforge.jtds.jdbc.Driver`
- `com.ibm.db2.jcc.DB2Driver`

Noncompliant code example

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class Demo {
  private static final String DRIVER_CLASS_NAME = "org.postgresql.Driver";
  private final Connection connection;

  public Demo(String serverURI) throws SQLException, ClassNotFoundException {
    Class.forName(DRIVER_CLASS_NAME); // Noncompliant; no longer required to load the JDBC Driver using Class.forName()
    connection = DriverManager.getConnection(serverURI);
  }
}
```

Compliant solution

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class Demo {
    private final Connection connection;

    public Demo(String serverURI) throws SQLException {
        connection = DriverManager.getConnection(serverURI);
    }
}
```

"serialVersionUID" should not be declared blindly (java:S4926)

Severidad: MINOR

Resources:

- Vojtech Ruzicka’s Programming Blog: Should I explicitly declare serialVersionUID?

Root_cause:

Providing a `serialVersionUID` field on `Serializable` classes is strongly recommended by the `Serializable` documentation but blindly following that recommendation can be harmful.

`serialVersionUID` value is stored with the serialized data and this field is verified when deserializing the data to ensure that the code reading the data is compatible with the serialized data. In case of failure, it means the serialized data and the code are not in sync and this fine because you know what’s wrong.

When the `serialVersionUID` is generated by an IDE or blindly hard-coded, there is a high probability that one will forget to update the `serialVersionUID` value when the `Serializable` class is later enriched with additional fields. As a consequence, old serialized data will incorrectly be considered compatible with the newer version of the code creating situations which are hard to debug.

Therefore, defining `serialVersionUID` should be done with care. This rule raises an issue on each `serialVersionUID` field declared on classes implementing `Serializable` to be sure the presence and the value of the `serialVersionUID` field is challenged and validated by the team.

Noncompliant code example

```
public class Foo implements Serializable {
  private static final long serialVersionUID = 1;
}

public class BarException extends RuntimeException {
  private static final long serialVersionUID = 8582433437601788991L;
}
```

"read(byte[],int,int)" should be overridden (java:S4929)

Severidad: MINOR

Root_cause:

When directly subclassing `java.io.InputStream` or `java.io.FilterInputStream`, the only requirement is that you implement the method `read()`. However most uses for such streams don’t read a single byte at a time and the default implementation for `read(byte[],int,int)` will call `read(int)` for every single byte in the array which can create a lot of overhead and is utterly inefficient. It is therefore strongly recommended that subclasses provide an efficient implementation of `read(byte[],int,int)`.

This rule raises an issue when a direct subclass of `java.io.InputStream` or `java.io.FilterInputStream` doesn’t provide an override of `read(byte[],int,int)`.

Noncompliant code example

```
public class MyInputStream extends java.io.InputStream {
  private FileInputStream fin;

  public MyInputStream(File file) throws IOException {
    fin = new FileInputStream(file);
  }

  @Override
  public int read() throws IOException {
    return fin.read();
  }
}
```

Compliant solution

```
public class MyInputStream extends java.io.InputStream {
  private FileInputStream fin;

  public MyInputStream(File file) throws IOException {
    fin = new FileInputStream(file);
  }

  @Override
  public int read() throws IOException {
    return fin.read();
  }

  @Override
  public int read(byte[] b, int off, int len) throws IOException {
    return fin.read(b, off, len);
  }
}
```

Exceptions

This rule doesn’t raise an issue when the class is declared `abstract`.

Methods annotated with "@BeforeTransaction" or "@AfterTransaction" must respect the contract (java:S7190)

Severidad: MAJOR

Root_cause:

In tests configured with Spring’s `@Transactional` annotation, methods annotated with `@BeforeTransaction` or `@AfterTransaction` must be void and have no arguments. These methods are executed before or after a transaction, respectively. Deviating from this contract by having a non-void return type or accepting arguments will cause Spring to throw a runtime error.

Resources:

Documentation

- Spring - BeforeTransaction
- Spring - AfterTransaction

How_to_fix:

Ensure that methods annotated with `@BeforeTransaction` or `@AfterTransaction` have a void return type and do not accept any arguments.

Noncompliant code example

```
public class TransactionalTest {

    @BeforeTransaction
    public String setupTransaction(int x) { // non-compliant, method should be void and have no argument
        // Setup logic
    }

    @AfterTransaction
    public int cleanupTransaction(int x) { // non-compliant, method should be void and have no argument
        // Cleanup logic
    }
}
```

Compliant solution

```
public class TransactionalTest {

    @BeforeTransaction
    public void setupTransaction() {
        // Setup logic
    }

    @AfterTransaction
    public void cleanupTransaction() {
        // Cleanup logic
    }
}
```

The members of an interface or class declaration should appear in a pre-defined order (java:S1213)

Severidad: MINOR

Root_cause:

According to the Java Code Conventions as defined by Oracle, the members of a class or interface declaration should appear in the following order in the source files:

- Class variables
- Instance variables
- Constructors
- Methods

Noncompliant code example

```
public class Foo{
   private int field = 0;
   public boolean isTrue() {...}
   public Foo() {...}                         // Noncompliant, constructor defined after methods
   public static final int OPEN = 4;  //Noncompliant, variable defined after constructors and methods
}
```

Compliant solution

```
public class Foo{
   public static final int OPEN = 4;
   private int field = 0;
   public Foo() {...}
   public boolean isTrue() {...}
}
```

Interfaces should not solely consist of constants (java:S1214)

Severidad: CRITICAL

Root_cause:

An interface that consists solely of constant definitions is a bad practice. The purpose of interfaces is to provide an API, not implementation details. That is, they should provide functions in the first place and constants only to assist these functions, for example, as possible arguments.

If an interface contains constants only, move them either to somewhere else, or replace the interface with an *Enum* or a final class with a private constructor.

Resources:

Articles & blog posts

- Mohammad Nadeem - Why the Constant Interface Pattern Should Be Discouraged
- Joshua Bloch - Effective Java, ISBN 9780134686097

How_to_fix:

If the concrete value of the constants is not essential, and they serve as mere identifiers, replace the interface with an `enum` like in the following example:

```
public interface Status { // Noncompliant, enum should be used
  int OPEN = 1;
  int CLOSED = 2;
}

public enum Status {      // Compliant
  OPEN,
  CLOSED
}
```

In some cases, enums are not a suitable option because the concrete constant value is important. Then you should check whether it is appropriate to move them to a specific existing class, for example, if that class is the primary user of the constants:

```
interface AuxiliaryConstants { // Noncompliant, implementation detail of WordPacker
  int BITS_PER_WORD = 16;
  int WORD_MASK = (1 << BITS_PER_WORD) - 1;
  int HI_WORD_BK_MASK = ~(WORD_MASK << BITS_PER_WORD);
}

class WordPacker {
  public static int getHiWord(int value) {
    return (value >>> AuxiliaryConstants.BITS_PER_WORD);
  }

  public static int setHiWord(int value, int wordValue) {
    return (value & AuxiliaryConstants.HI_WORD_BK_MASK) |
      (wordValue << AuxiliaryConstants.BITS_PER_WORD);
  }
}

class WordPacker { // Compliant
  private static final int BITS_PER_WORD = 16;
  private static final int WORD_MASK = (1 << BITS_PER_WORD) - 1;
  private static final int HI_WORD_BK_MASK = ~(WORD_MASK << BITS_PER_WORD);

  public static int getHiWord(int value) {
      return (value >>> BITS_PER_WORD);
  }

  public static int setHiWord(int value, int wordValue) {
    return (value & HI_WORD_BK_MASK) | (wordValue << BITS_PER_WORD);
  }
}
```

If this is not the case and several classes are using the constants equally, you should use a final class with a private constructor. Unlike interfaces, they can neither be inherited from nor instantiated.

```
public interface ColorTheme { // Noncomplient, final class should be used
  int COLOR_ERROR = 0xff0000;   // red
  int COLOR_WARNING = 0xffff00; // yellow
  int COLOR_OK = 0x00cf00;      // green
}

public final class ColorTheme { // Compliant
  public static final int COLOR_ERROR = 0xff0000;   // red
  public static final int COLOR_WARNING = 0xffff00; // yellow
  public static final int COLOR_OK = 0x00cf00;      // green

  private ColorTheme() {}
}
```

Introduction:

This rule raises an issue when an interface consists only of constant definitions without other members.

Maps with keys that are enum values should use the EnumMap implementation (java:S1640)

Severidad: MINOR

Root_cause:

If all the keys in a `Map` are values from a single enum, it is recommended to use an `EnumMap` as the specific implementation. An `EnumMap`, which has the advantage of knowing all possible keys in advance, is more efficient compared to other implementations, as it can use a simple array as its underlying data structure.

Noncompliant code example

```
public enum Color {
  RED, GREEN, BLUE, ORANGE;
}

Map<Color, String> colorMap = new HashMap<>(); // Noncompliant
```

Compliant solution

```
public enum Color {
  RED, GREEN, BLUE, ORANGE;
}
```

```
Map<Color, String> colorMap = new EnumMap<>(Color.class); // Compliant
```

Resources:

- A Guide to EnumMap - Baeldung

Sets with elements that are enum values should be replaced with EnumSet (java:S1641)

Severidad: MINOR

Root_cause:

When all the elements in a Set are values from the same enum, the Set can be replaced with an EnumSet, which can be much more efficient than other sets because the underlying data structure is a simple bitmap.

Noncompliant code example

```
public class MyClass {

  public enum COLOR {
    RED, GREEN, BLUE, ORANGE;
  }

  public void doSomething() {
    Set<COLOR> warm = new HashSet<COLOR>();
    warm.add(COLOR.RED);
    warm.add(COLOR.ORANGE);
  }
}
```

Compliant solution

```
public class MyClass {

  public enum COLOR {
    RED, GREEN, BLUE, ORANGE;
  }

  public void doSomething() {
    Set<COLOR> warm = EnumSet.of(COLOR.RED, COLOR.ORANGE);
  }
}
```

Strings should not be concatenated using '+' in a loop (java:S1643)

Severidad: MINOR

Root_cause:

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 < arrayOfStrings.length ; ++i) {
  str = str + arrayOfStrings[i];
}
```

Compliant solution

```
StringBuilder bld = new StringBuilder();
  for (int i = 0; i < arrayOfStrings.length; ++i) {
    bld.append(arrayOfStrings[i]);
  }
  String str = bld.toString();
```

"==" and "!=" should not be used when "equals" is overridden (java:S1698)

Severidad: MINOR

Root_cause:

It is equivalent to use the equality `==` operator and the `equals` method to compare two objects if the `equals` method inherited from `Object` has not been overridden. In this case both checks compare the object references.

But as soon as `equals` is overridden, two objects not having the same reference but having the same value can be equal. This rule spots suspicious uses of `==` and `!=` operators on objects whose `equals` methods are overridden.

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 && firstName.equals(lastName)) { ... };
```

Exceptions

Comparing two instances of the `Class` object will not raise an issue:

```
Class c;
if(c == Integer.class) { // No issue raised
}
```

Comparing `Enum` will not raise an issue:

```
public enum Fruit {
   APPLE, BANANA, GRAPE
}
public boolean isFruitGrape(Fruit candidateFruit) {
  return candidateFruit == Fruit.GRAPE; // it's recommended to activate S4551 to enforce comparison of Enums using ==
}
```

Comparing with `final` reference will not raise an issue:

```
private static final Type DEFAULT = new Type();

void foo(Type other) {
  if (other == DEFAULT) { // Compliant
  //...
  }
}
```

Comparing with `this` will not raise an issue:

```
  public boolean equals(Object other) {
    if (this == other) {  // Compliant
      return false;
    }
 }
```

Comparing with `java.lang.String` and boxed types `java.lang.Integer`, … will not raise an issue.

Resources:

- S4973 - Strings and Boxed types should be compared using "equals()"
- CWE - CWE-595 - Comparison of Object References Instead of Object Contents
- CWE - 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

Identical expressions should not be used on both sides of a binary operator (java:S1764)

Severidad: MAJOR

Root_cause:

Using the same value on both sides of a binary operator is a code defect. In the case of logical operators, it is either a copy/paste error and, therefore, a bug, or it is simply duplicated code and should be simplified. In the case of bitwise operators and most binary mathematical operators, having the same value on both sides of an operator yields predictable results and should be simplified as well.

Noncompliant code example

```
if ( a == a ) { // always true
  doZ();
}
if ( a != a ) { // always false
  doY();
}
if ( a == b && a == b ) { // if the first one is true, the second one is too
  doX();
}
if ( a == b || a == b ) { // if the first one is true, the second one is too
  doW();
}

int j = 5 / 5; //always 1
int k = 5 - 5; //always 0

c.equals(c); //always true
```

Exceptions

- This rule ignores `*`, `+`, and `=`.
- The specific case of testing a floating point value against itself is a valid test for `NaN` and is therefore ignored.
- Similarly, left-shifting 1 onto 1 is common in the construction of bit masks, and is ignored.

```
float f;
if(f != f) { //test for NaN value
  System.out.println("f is NaN");
}

int i = 1 << 1; // Compliant
int j = a << a; // Noncompliant
```

Resources:

- CERT, MSC12-C. - Detect and remove code that has no effect or is never executed
- S1656 - Implements a check on `=`.

Setting loose POSIX file permissions is security-sensitive (java:S2612)

Severidad: MAJOR

How_to_fix:

Recommended Secure Coding Practices

The most restrictive possible permissions should be assigned to files and directories.

Compliant Solution

On operating systems that implement POSIX standard. This will throw a `UnsupportedOperationException` on Windows.

```
    public void setPermissionsSafe(String filePath) throws IOException {
        Set<PosixFilePermission> perms = new HashSet<PosixFilePermission>();
        // user permission
        perms.add(PosixFilePermission.OWNER_READ);
        perms.add(PosixFilePermission.OWNER_WRITE);
        perms.add(PosixFilePermission.OWNER_EXECUTE);
        // group permissions
        perms.add(PosixFilePermission.GROUP_READ);
        perms.add(PosixFilePermission.GROUP_EXECUTE);
        // others permissions removed
        perms.remove(PosixFilePermission.OTHERS_READ); // Compliant
        perms.remove(PosixFilePermission.OTHERS_WRITE); // Compliant
        perms.remove(PosixFilePermission.OTHERS_EXECUTE); // Compliant

        Files.setPosixFilePermissions(Paths.get(filePath), perms);
    }
```

See

- OWASP - Top 10 2021 Category A1 - Broken Access Control
- OWASP - Top 10 2021 Category A4 - Insecure Design
- OWASP - Top 10 2017 Category A5 - Broken Access Control
- OWASP File Permission
- CWE - CWE-732 - Incorrect Permission Assignment for Critical Resource

- CWE - CWE-266 - Incorrect Privilege Assignment
- CERT, FIO01-J. - Create files with appropriate access permissions
- CERT, FIO06-C. - Create files with appropriate access permissions
- STIG Viewer - Application Security and Development: V-222430 - The application must execute without excessive account permissions.

Default:

In Unix file system permissions, the "`others`" category refers to all users except the owner of the file system resource and the members of the group assigned to this resource.

Granting permissions to this category can lead to unintended access to files or directories that could allow attackers to obtain sensitive information, disrupt services or elevate privileges.

Ask Yourself Whether

- The application is designed to be run on a multi-user environment.
- Corresponding files and directories may contain confidential information.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

The most restrictive possible permissions should be assigned to files and directories.

Sensitive Code Example

```
public void setPermissions(String filePath) {
    Set<PosixFilePermission> perms = new HashSet<PosixFilePermission>();
    // user permission
    perms.add(PosixFilePermission.OWNER_READ);
    perms.add(PosixFilePermission.OWNER_WRITE);
    perms.add(PosixFilePermission.OWNER_EXECUTE);
    // group permissions
    perms.add(PosixFilePermission.GROUP_READ);
    perms.add(PosixFilePermission.GROUP_EXECUTE);
    // others permissions
    perms.add(PosixFilePermission.OTHERS_READ); // Sensitive
    perms.add(PosixFilePermission.OTHERS_WRITE); // Sensitive
    perms.add(PosixFilePermission.OTHERS_EXECUTE); // Sensitive

    Files.setPosixFilePermissions(Paths.get(filePath), perms);
}

public void setPermissionsUsingRuntimeExec(String filePath) {
    Runtime.getRuntime().exec("chmod 777 file.json"); // Sensitive
}

public void setOthersPermissionsHardCoded(String filePath ) {
    Files.setPosixFilePermissions(Paths.get(filePath), PosixFilePermissions.fromString("rwxrwxrwx")); // Sensitive
}
```

Compliant Solution

On operating systems that implement POSIX standard. This will throw a `UnsupportedOperationException` on Windows.

```
public void setPermissionsSafe(String filePath) throws IOException {
    Set<PosixFilePermission> perms = new HashSet<PosixFilePermission>();
    // user permission
    perms.add(PosixFilePermission.OWNER_READ);
    perms.add(PosixFilePermission.OWNER_WRITE);
    perms.add(PosixFilePermission.OWNER_EXECUTE);
    // group permissions
    perms.add(PosixFilePermission.GROUP_READ);
    perms.add(PosixFilePermission.GROUP_EXECUTE);
    // others permissions removed
    perms.remove(PosixFilePermission.OTHERS_READ); // Compliant
    perms.remove(PosixFilePermission.OTHERS_WRITE); // Compliant
    perms.remove(PosixFilePermission.OTHERS_EXECUTE); // Compliant

    Files.setPosixFilePermissions(Paths.get(filePath), perms);
}
```

See

- OWASP - Top 10 2021 Category A1 - Broken Access Control
- OWASP - Top 10 2021 Category A4 - Insecure Design

- OWASP - Top 10 2017 Category A5 - Broken Access Control
- OWASP File Permission
- CWE - CWE-732 - Incorrect Permission Assignment for Critical Resource
- CWE - CWE-266 - Incorrect Privilege Assignment
- CERT, FIO01-J. - Create files with appropriate access permissions
- CERT, FIO06-C. - Create files with appropriate access permissions
- STIG Viewer - Application Security and Development: V-222430 - The application must execute without excessive account permissions.

Assess_the_problem:

Ask Yourself Whether

- The application is designed to be run on a multi-user environment.
- Corresponding files and directories may contain confidential information.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

```
    public void setPermissions(String filePath) {
        Set<PosixFilePermission> perms = new HashSet<PosixFilePermission>();
        // user permission
        perms.add(PosixFilePermission.OWNER_READ);
        perms.add(PosixFilePermission.OWNER_WRITE);
        perms.add(PosixFilePermission.OWNER_EXECUTE);
        // group permissions
        perms.add(PosixFilePermission.GROUP_READ);
        perms.add(PosixFilePermission.GROUP_EXECUTE);
        // others permissions
        perms.add(PosixFilePermission.OTHERS_READ); // Sensitive
        perms.add(PosixFilePermission.OTHERS_WRITE); // Sensitive
        perms.add(PosixFilePermission.OTHERS_EXECUTE); // Sensitive

        Files.setPosixFilePermissions(Paths.get(filePath), perms);
    }

    public void setPermissionsUsingRuntimeExec(String filePath) {
        Runtime.getRuntime().exec("chmod 777 file.json"); // Sensitive
    }

    public void setOthersPermissionsHardCoded(String filePath ) {
        Files.setPosixFilePermissions(Paths.get(filePath), PosixFilePermissions.fromString("rwxrwxrwx")); // Sensitive
    }
```

Root_cause:

In Unix file system permissions, the "`others`" category refers to all users except the owner of the file system resource and the members of the group assigned to this resource.

Granting permissions to this category can lead to unintended access to files or directories that could allow attackers to obtain sensitive information, disrupt services or elevate privileges.

"catch" clauses should do more than rethrow (java:S2737)

Severidad: MINOR

Root_cause:

A `catch` clause that only rethrows the caught exception has the same effect as omitting the `catch` altogether and letting it bubble up automatically.

```
public String readFile(File f) throws IOException {
  String content;
  try {
    content = readFromDisk(f);
  } catch (IOException e) {
    throw e;
  }
  return content;
}
```

Such clauses should either be removed or populated with the appropriate logic.

```
public String readFile(File f) throws IOException {
  return readFromDisk(f);
}
```

or

```
public String readFile(File f) throws IOException {
  String content;
  try {
    content = readFromDisk(f);
  } catch (IOException e) {
    logger.LogError(e);
    throw e;
  }
  return content;
}
```

In the case of try-with-resources, the try should remain even without a catch clause, to keep the resource management

```
String readFirstLine(FileReader fileReader) throws IOException {
  try (BufferedReader br = new BufferedReader(fileReader)) {
     return br.readLine();
  } catch (IOException e) { // Noncompliant
  throw e;
}
```

becomes

```
String readFirstLine(FileReader fileReader) throws IOException {
  try (BufferedReader br = new BufferedReader(fileReader)) {
     return br.readLine();
  }
}
```

Nested "enum"s should not be declared static (java:S2786)

Severidad: MINOR

Root_cause:

In Java, an enum is a special data type that allows you to define a set of constants. Nested enum types, also known as inner enum types, are enum types that are defined within another class or interface.

Nested enum types are implicitly static, so there is no need to declare them `static` explicitly.

Noncompliant code example

```
public class Flower {
  static enum Color { // Noncompliant; static is redundant here
    RED, YELLOW, BLUE, ORANGE
  }
  // ...
}
```

Compliant solution

```
public class Flower {
  enum Color { // Compliant
    RED, YELLOW, BLUE, ORANGE
  }
  // ...
}
```

Resources:

- Java Language Specification-8.9

Assertions should be complete (java:S2970)

Severidad: BLOCKER

Root_cause:

It is very easy to write incomplete assertions when using some test frameworks. This rule enforces complete assertions in the following cases:

- Fest: `assertThat` is not followed by an assertion invocation
- AssertJ: `assertThat` is not followed by an assertion invocation
- Mockito: `verify` is not followed by a method invocation
- Truth: `assertXXX` is not followed by an assertion invocation

In such cases, what is intended to be a test doesn’t actually verify anything

Noncompliant code example

```
// Fest
boolean result = performAction();
// let's now check that result value is true
assertThat(result); // Noncompliant; nothing is actually checked, the test passes whether "result" is true or false

// Mockito
List mockedList = Mockito.mock(List.class);
mockedList.add("one");
mockedList.clear();
// let's check that "add" and "clear" methods are actually called
Mockito.verify(mockedList); // Noncompliant; nothing is checked here, oups no call is chained to verify()
```

Compliant solution

```
// Fest
boolean result = performAction();
// let's now check that result value is true
assertThat(result).isTrue();

// Mockito
List mockedList = Mockito.mock(List.class);
mockedList.add("one");
mockedList.clear();
// let's check that "add" and "clear" methods are actually called
Mockito.verify(mockedList).add("one");
Mockito.verify(mockedList).clear();
```

Exceptions

Variable assignments and return statements are skipped to allow helper methods.

```
private BooleanAssert check(String filename, String key) {
  String fileContent = readFileContent(filename);
  performReplacements(fileContent);
  return assertThat(fileContent.contains(key)); // No issue is raised here
}

@Test
public void test() {
  check("foo.txt", "key1").isTrue();
  check("bar.txt", "key2").isTrue();
}
```

Inner classes should not have too many lines of code (java:S2972)

Severidad: MAJOR

Root_cause:

Inner classes should be short and sweet, to manage complexity in the overall file. An inner class that has grown longer than a certain threshold should probably be externalized to its own file.

Escaped Unicode characters should not be used (java:S2973)

Severidad: MAJOR

Root_cause:

The use of Unicode escape sequences should be reserved for characters that would otherwise be ambiguous, such as unprintable characters.

This rule ignores sequences composed entirely of Unicode characters, but otherwise raises an issue for each Unicode character that represents a printable character.

Noncompliant code example

```
String prefix = "n\u00E9e"; // Noncompliant
```

Compliant solution

```
String prefix = "née";
```

Classes without "public" constructors should be "final" (java:S2974)

Severidad: MINOR

Root_cause:

Classes with only `private` constructors should be marked `final` to prevent any mistaken extension attempts.

Noncompliant code example

```
public class PrivateConstructorClass {  // Noncompliant
  private PrivateConstructorClass() {
    // ...
  }

  public static int magic(){
    return 42;
  }
}
```

Compliant solution

```
public final class PrivateConstructorClass {  // Compliant
  private PrivateConstructorClass() {
    // ...
  }

  public static int magic(){
    return 42;
  }
}
```

"clone" should not be overridden (java:S2975)

Severidad: BLOCKER

Root_cause:

The `Object.clone` / `java.lang.Cloneable` mechanism in Java should be considered broken for the following reasons and should, consequently, not be used:

- `Cloneable` is a *marker interface* without API but with a contract about class behavior that the compiler cannot enforce. This is a bad practice.
- Classes are instantiated without calling their constructor, so possible preconditions cannot be enforced.
- There are implementation flaws by design when overriding `Object.clone`, like type casts or the handling of `CloneNotSupportedException` exceptions.

Introduction:

This rule raises an issue when a class overrides the `Object.clone` method instead of resorting to a copy constructor or other copy mechanisms.

How_to_fix:

A copy constructor, copy factory or a custom copy function are suitable alternatives to the `Object.clone` / `java.lang.Cloneable` mechanism.

Consider the following example:

```
class Entity implements Cloneable { // Noncompliant, using `Cloneable`

  public int value;
  public List<Entity> children; // deep copy wanted

  Entity() {
    EntityManager.register(this); // invariant
  }

  @Override
  public Entity clone() {
    try {
      Entity copy = (Entity) super.clone(); // invariant not enforced, because no constructor is caled
      copy.children = children.stream().map(Entity::clone).toList();
      return copy;
    } catch (CloneNotSupportedException e) { // this will not happen due to behavioral contract
      throw new AssertionError();
    }
  }
}
```

The `Cloneable` / `Object.clone` mechanism could easily be replaced by copy constructor:

```
class Entity { // Compliant

  public int value;
  public List<Entity> children; // deep copy wanted

  Entity() {
    EntityManager.register(this); // invariant
  }

  Entity(Entity template) {
    value = template.value;
    children = template.children.stream().map(Entity::new).toList();
  }
}
```

Or by a factory method:

```
class Entity { // Compliant

  public int value;
  public List<Entity> children; // deep copy wanted

  Entity() {
    EntityManager.register(this); // invariant
  }

  public static Entity create(Entity template) {
    Entity entity = new Entity();
    entity.value = template.value;
    entity.children = template.children.stream().map(Entity::new).toList();
    return Entity;
  }
}
```

Or by a custom copy function:

```
class Entity { // Compliant

  public int value;
  public List<Entity> children; // deep copy wanted

  Entity() {
    EntityManager.register(this); // invariant
  }

  public Entity copy() {
    Entity entity = new Entity();
    entity.value = value;
    entity.children = children.stream().map(Entity::new).toList();
    return Entity;
  }
}
```

Resources:

Documentation

- Joshua Bloch - Copy Constructor versus Cloning
- Interface Cloneable - Java™ Platform, Standard Edition 8 API Specification
- Object.clone - Java™ Platform, Standard Edition 8 API Specification

Related rules

- S2157 - "Cloneables" should implement "clone"
- S1182 - Classes that override "clone" should be "Cloneable" and call "super.clone()"

"Map.get" and value test should be replaced with single method call (java:S3824)

Severidad: MAJOR

Root_cause:

It’s a common pattern to test the result of a `java.util.Map.get()` against `null` or calling `java.util.Map.containsKey()` before proceeding with adding or changing the value in the map. However the `java.util.Map` API offers a significantly better alternative in the form of the `computeIfPresent()` and `computeIfAbsent()` methods. Using these instead leads to cleaner and more readable code.

Note that this rule is automatically disabled when the project's `sonar.java.source` is not 8.

Noncompliant code example

```
V value = map.get(key);
if (value == null) {  // Noncompliant
  value = V.createFor(key);
  if (value != null) {
    map.put(key, value);
  }
}
if (!map.containsKey(key)) {  // Noncompliant
  value = V.createFor(key);
  if (value != null) {
    map.put(key, value);
  }
}
return value;
```

Compliant solution

```
return map.computeIfAbsent(key, k -> V.createFor(k));
```

Exceptions

This rule will not raise an issue when trying to add `null` to a map, because `computeIfAbsent` will not add the entry if the value returned by the function is `null`.

Resources:

Related rules

- S6104 - Map "computeIfAbsent()" and "computeIfPresent()" should not be used to add "null" values.

"@SpringBootApplication" and "@ComponentScan" should not be used in the default package (java:S4602)

Severidad: BLOCKER

Root_cause:

`@ComponentScan` is used to determine which Spring Beans are available in the application context. The packages to scan can be configured thanks to the `basePackageClasses` or `basePackages` (or its alias `value`) parameters. If neither parameter is configured, `@ComponentScan` will consider only the package of the class annotated with it. When `@ComponentScan` is used on a class belonging to the default package, the entire classpath will be scanned.

This will slow-down the start-up of the application and it is likely the application will fail to start with an `BeanDefinitionStoreException` because you ended up scanning the Spring Framework package itself.

This rule raises an issue when:

- `@ComponentScan`, `@SpringBootApplication` and `@ServletComponentScan` are used on a class belonging to the default package
- `@ComponentScan` is explicitly configured with the default package

Noncompliant code example

```
import org.springframework.boot.SpringApplication;

@SpringBootApplication // Noncompliant; RootBootApp is declared in the default package
public class RootBootApp {
...
}

@ComponentScan("")
public class Application {
...
}
```

Compliant solution

```
package hello;

import org.springframework.boot.SpringApplication;

@SpringBootApplication // Compliant; RootBootApp belongs to the "hello" package
public class RootBootApp {
```

```
...
}
```

JUnit5 test classes and methods should not be silently ignored (java:S5810)

Severidad: MAJOR

Root_cause:

JUnit5 is more tolerant regarding the visibilities of Test classes and methods than JUnit4, which required everything to be public. JUnit5 supports default package, public and protected visibility, even if it is recommended to use the default package visibility, which improves the readability of code.

But JUnit5 ignores without any warning:

- private classes and private methods
- static methods
- methods returning a value without being a TestFactory

Noncompliant code example

```
import org.junit.jupiter.api.Test;

class MyClassTest {
  @Test
  private void test1() { // Noncompliant - ignored by JUnit5
    // ...
  }
  @Test
  static void test2() { // Noncompliant - ignored by JUnit5
    // ...
  }
  @Test
  boolean test3() { // Noncompliant - ignored by JUnit5
    // ...
  }
  @Nested
  private class MyNestedClass { // Noncompliant - ignored by JUnit5
    @Test
    void test() {
      // ...
    }
  }
}
```

Compliant solution

```
import org.junit.jupiter.api.Test;

class MyClassTest {
  @Test
  void test1() {
    // ...
  }
  @Test
  void test2() {
    // ...
  }
  @Test
  void test3() {
    // ...
  }
  @Nested
  class MyNestedClass {
    @Test
    void test() {
      // ...
    }
  }
}
```

Use appropriate @DirtiesContext modes (java:S7177)

Severidad: MAJOR

Root_cause:

In a Spring application, the `@DirtiesContext` annotation marks the ApplicationContext as dirty and indicates that it should be cleared and recreated. This is important in tests that modify the context, such as altering the state of singleton beans or databases.

Misconfiguring `@DirtiesContext` by setting the `methodMode` at the class level or the `classMode` at the method level will make the annotation have no effect.

This rule will raise an issue when the incorrect mode is configured on a @DirtiesContext annotation targeting a different scope.

Resources:

Documentation

- Spring documentation - @DirtiesContext

How_to_fix:

Noncompliant code example

```
@ContextConfiguration
@DirtiesContext(methodMode = MethodMode.AFTER_METHOD) // Noncompliant, for class-level control, use classMode instead.
public class TestClass {
  @DirtiesContext(classMode = DirtiesContext.ClassMode.AFTER_CLASS) // Non compliant, for method-level control use methodMode instead
  public void test() {...}
}
```

Compliant solution

```
@ContextConfiguration
@DirtiesContext(classMode = DirtiesContext.ClassMode.AFTER_CLASS)
public class TestClass {
  @DirtiesContext(methodMode = MethodMode.AFTER_METHOD)
  public void test() {...}
}
```

Injecting data into static fields is not supported by Spring (java:S7178)

Severidad: MAJOR

Root_cause:

Spring dependency injection framework does not support injecting data into static fields. When @Value, @Inject, or @Autowired are applied to static fields, they are ignored.

What is the potential impact?

- **Null Values**: Uninitialized static fields annotated with @Value, @Inject, or @Autowired will not be initialized by Spring, potentially causing NullPointerException at runtime.
- **Confusing Code**: The presence of injection annotations on static fields can mislead developers into believing that the fields will be populated by Spring.

This rule raises an issue when a static field is annotated with @Value, @Inject, or @Autowired.

Resources:

Articles & blog posts

- Java Guides - Injecting a Value in a Static Field in Spring

How_to_fix:

Either use an instance field instead of a static field or remove the @Value, @Inject, or @Autowired annotation and initialize the field.

Noncompliant code example

```
@Component
public class MyComponent {

    @Value("${my.app.prop}")
    private static SomeDependency dependency; // non compliant, @Value will be ignored and no value will be injected
    // ...
}
```

Compliant solution

```
@Component
public class MyComponent {

    @Value("${my.app.prop}")
    private final SomeDependency dependency;
    // ...
}
```

@Cacheable and @CachePut should not be combined (java:S7179)

Severidad: MAJOR

Resources:

Documentation

- Spring Documentation - @CachePut
- Spring Documentation - @Cacheable

Root_cause:

`@Cacheable` annotation is used to store the result of a method and avoid executing it for the same inputs. `@CachePut` instead is used to force the execution of a method and store the result in the cache. Annotating a method with both will produce unreliable behavior, except for specific corner-cases when their `condition()` or `unless()` expressions are mutually exclusive. Hence this pattern is strongly discouraged and an issue will be raised on such cases.

How_to_fix:

Noncompliant code example

```
@Cacheable
@CachePut
void getBook(String isbn){ // Non compliant, methods annotated with both @Cacheable and @CachePut will not behave as intended
    ...
}
```

Compliant solution

```
@Cacheable
void getBook(String isbn){
    ...
}
```

"@Cache*" annotations should only be applied on concrete classes (java:S7180)

Severidad: MAJOR

Root_cause:

Annotating interfaces or interface methods with `@Cache*` annotations is not recommended by the official Spring documentation. If you use the weaving-based aspect (mode="aspectj"), the `@Cache*` annotations will be ignored, and no caching proxy will be created.

What is the potential impact?

- **Confusing Code**: Developers may mistakenly believe that caching is in effect, leading to confusion and incorrect assumptions about application performance.

This rule raises an issue when an interface or an interface method is annotated with a `@Cache*` annotation.

Resources:

Documentation

- Spring - Declarative Annotation-based Caching

How_to_fix:

Move `@Cache*` annotation from interface or interface method to the concrete class.

Noncompliant code example

```
public interface ExampleService {

    @Cacheable("exampleCache") //non compliant, interface method is annotated with @Cacheable
    String getData(String id);
}
```

Compliant solution

```
@Service
public class ExampleServiceImpl implements ExampleService {

    @Cacheable("exampleCache")
    @Override
    public String getData(String id) {
        // Implementation here
    }
}
```

@InitBinder methods should have void return type (java:S7183)

Severidad: MAJOR

Root_cause:

Spring provides the `@InitBinder` annotation to initialize a `WebDataBinder` instance for controllers. This is useful to bind request parameters to a model object, and to plug converters and formatters into this process.

Methods annotated with `@InitBinder` must not have a return value, otherwise the controller containing them will throw an exception when invoked.

This rule raises an issue when a method annotated with `@InitBinder` does not have a `void` return type

How_to_fix:

Noncompliant code example

```
@Controller
public class MyController {

        @InitBinder
        public String initBinder(WebDataBinder binder) { // Non compliant, make the @InitBinder method return void
                SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
                dateFormat.setLenient(false);
                binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, false));
        return "OK";
        }

        // ...
}
```

Compliant solution

```
@Controller
public class MyController {

        @InitBinder
        public void initBinder(WebDataBinder binder) {
                SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
                dateFormat.setLenient(false);
                binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, false));
        }

        // ...
}
```

Resources:

Documentation

- Spring api - @InitBinder api
- String documentation - @InitBinder docs

"@Scheduled" annotation should only be applied to no-arg methods (java:S7184)

Severidad: MAJOR

Root_cause:

According to Spring documentation, the `@Scheduled` annotation can only be applied to methods without arguments. Applying @Scheduled to a method with arguments will result in a runtime error.

How_to_fix:

Transform method annotated with `@Scheduled` into a no-arg method.

Noncompliant code example

```
public class ExampleService {

    @Scheduled(fixedRate = 5000)
    public void scheduledTask(String param) { // non compliant, method has an argument. It will raise a runtime error.
        // Task implementation
    }
}
```

Compliant solution

```
public class ExampleService {

    @Scheduled(fixedRate = 5000)
    public void scheduledTask() { // compliant, no-arg method
        // Task implementation
    }
}
```

Resources:

Documentation

- Spring - scheduled

@EventListener methods should have one parameter at most (java:S7185)

Severidad: MAJOR

Resources:

Documentation

Spring API - @EventListener

Root_cause:

Spring provides the `@EventListener` annotation as a simpler alternative to implementing the `ApplicationListener` interface for handling events. The `@EventListener` annotation registers a method as an event handler. This allows to skip the implementation of the `ApplicationListener` interface, making it easier to handle events.

The `@EventListener` annotation can only be used on methods that have at most one parameter, which should be the specific event that we want to handle. To listen to several types of events, use the `classes` argument of the `@EventListener` annotation.

This rule raises an issue on all methods annotated with `@EventListener` that have more than one parameter.

How_to_fix:

Noncompliant code example

```
@EventListener
void handleEvent(CustomerEvent customerEvent, ExceptionalEvent exceptionalEvent) { // Non compliant, this will cause a runtime error
    //... some event handling
}
```

Compliant solution

```
@EventListener(classes = {CustomerEvent.class, ExceptionalEvent.class})
void handleEvent(ApplicationEvent event) { // Only one parameter, of the super type `ApplicationEvent`
    //... some event handling
}
```

Methods returning "Page" or "Slice" must take "Pageable" as an input parameter (java:S7186)

Severidad: MAJOR

Resources:

Documentation

- Spring - JPA Query Methods
- Spring - Defining Query Methods

Articles & blog posts

- Spring Guides - Paging with Spring Boot

Root_cause:

Spring Data Repository supports paging for queries, allowing you to return results in small, manageable chunks rather than retrieving an entire large result set.

The conventional approach to paginating data in Spring is to use the `Pageable` interface to control pagination and to store the query results into a `Page` or `Slice`. If a query method in a `Repository` returns a `Page` or `Slice` without taking a `Pageable` as an input, it raises a runtime exception.

This rule raises an issue on queries in a `Repository` that return a `Page` or `Slice` without taking a `Pageable` as an input.

How_to_fix:

Ensure that query methods returning a `Page` or `Slice` include a `Pageable` parameter in their method signature.

Noncompliant code example

```
public Page<Item> findItems() { //non compliant, no Pageable parameter
    // query
}
```

Compliant solution

```
public Page<Item> findItems(Pageable pageable) {
    // query
}
```

"equals(Object obj)" should be overridden along with the "compareTo(T obj)" method (java:S1210)

Severidad: MINOR

Root_cause:

According to the Java `Comparable.compareTo(T o)` documentation:

> It is strongly recommended, but not strictly required that `(x.compareTo(y)==0) == (x.equals(y))`.
>
> Generally speaking, any class that implements the Comparable interface and violates this condition should clearly indicate this fact.
>
> The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."

If this rule is violated, weird and unpredictable failures can occur.

For example, in Java 5 the `PriorityQueue.remove()` method relied on `compareTo()`, but since Java 6 it has relied on `equals()`.

Noncompliant code example

```
public class Foo implements Comparable<Foo> {
  @Override
  public int compareTo(Foo foo) { /* ... */ }      // Noncompliant as the equals(Object obj) method is not overridden
}
```

Compliant solution

```
public class Foo implements Comparable<Foo> {
  @Override
  public int compareTo(Foo foo) { /* ... */ }      // Compliant

  @Override
  public boolean equals(Object obj) { /* ... */ }
}
```

}

"Thread.run()" should not be called directly (java:S1217)

Severidad: MAJOR

Root_cause:

The likely intention of a user calling `Thread.run()` is to start the execution of code within a new thread. This, however, is not what happens when this method is called.

The purpose of `Thread.run()` is to provide a method that users can overwrite to specify the code to be executed. The actual thread is then started by calling `Thread.start()`. When `Thread.run()` is called directly, it will be executed as a regular method within the current thread.

Introduction:

This rule raises an issue when `Thread.run()` is called instead of `Thread.start()`.

How_to_fix:

If you intend to execute the contents of the `Thread.run()` method within a new thread, call `Thread.start()` instead.

If your intention is only to have a container for a method but execute this method within the current thread, do not use `Thread` but `Runnable` or another functional interface.

Noncompliant code example

```
Thread myThread = new Thread(runnable);
myThread.run(); // Noncompliant, does not start a thread
```

Compliant solution

```
Thread myThread = new Thread(runnable);
myThread.start(); // Compliant
```

Noncompliant code example

```
class ComputePrimesThread extends Thread {
    @Override
    public void run() {
        // ...
    }
}
new ComputePrimesThread().run(); // Noncompliant, does not start a thread
```

Compliant solution

```
class ComputePrimesThread extends Thread {
    @Override
    public void run() {
        // ...
    }
}
new ComputePrimesThread().start(); // Compliant
```

Noncompliant code example

```
class Button {

    private Thread onClick;

    Button(Thread onClick) {
        this.onClick = onClick;
    }

    private void clicked() {
        if (onClick != null) onClick.run(); // Noncompliant, use functional interface
    }
}

new Button(new Thread() {
    @Override public void run() {
        System.out.println("clicked!");
    }
});
```

Compliant solution

```
class Button {

    private Runnable onClick;

    Button(Runnable onClick) {
        this.onClick = onClick;
    }

    private void clicked() {
        if (onClick != null) onClick.run(); // compliant
    }
}

new Button(() -> System.out.println("clicked!"));
```

Resources:

Documentation

- Java™ Platform, Standard Edition 8 API Specification - Thread.start()

Articles & blog posts

- JavaTPoint - What if we call Java run() method directly instead start() method?

"switch" statements should not contain non-case labels (java:S1219)

Severidad: BLOCKER

Root_cause:

Even if it is legal, mixing case and non-case labels in the body of a switch statement is very confusing and can even be the result of a typing error.

Noncompliant code example

```
switch (day) {
  case MONDAY:
  case TUESDAY:
  WEDNESDAY:   // Noncompliant; syntactically correct, but behavior is not what's expected
    doSomething();
    break;
  ...
}

switch (day) {
  case MONDAY:
    break;
  case TUESDAY:
    foo:for(int i = 0 ; i < X ; i++) {  // Noncompliant; the code is correct and behaves as expected but is barely readable
         /* ... */
        break foo;  // this break statement doesn't relate to the nesting case TUESDAY
         /* ... */
    }
    break;
    /* ... */
}
```

Compliant solution

```
switch (day) {
  case MONDAY:
  case TUESDAY:
  case WEDNESDAY:
    doSomething();
    break;
  ...
}

switch (day) {
  case MONDAY:
    break;
  case TUESDAY:
    compute(args); // put the content of the labelled "for" statement in a dedicated method
    break;

    /* ... */
}
```

Private fields only used as local variables in methods should become local variables (java:S1450)

Severidad: MINOR

Root_cause:

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.

Track lack of copyright and license headers (java:S1451)

Severidad: BLOCKER

Root_cause:

Each source file should start with a header stating file ownership and the license which must be used to distribute the application.

This rule must be fed with the header text that is expected at the beginning of every file.

Compliant solution

```
/*
 * SonarQube, open source software quality management tool.
 * Copyright (C) 2008-2013 SonarSource
 * mailto:contact AT sonarsource DOT com
 *
 * SonarQube is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 3 of the License, or (at your option) any later version.
 *
 * SonarQube is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301, USA.
 */
```

Generic wildcard types should not be used in return types (java:S1452)

Severidad: CRITICAL

Root_cause:

A return type containing wildcards cannot be narrowed down in any context. This indicates that the developer’s intention was likely something else.

The core problem lies in type variance. Expressions at an input position, such as arguments passed to a method, can have a more specific type than the type expected by the method, which is called *covariance*. Expressions at an output position, such as a variable that receives the return result from a method, can have a more general type than the method’s return type, which is called *contravariance*. This can be traced back to the Liskov substitution principle.

In Java, type parameters of a generic type are invariant by default due to their potential occurrence in both input and output positions at the same time. A classic example of this is the methods `T get()` (output position) and `add(T element)` (input position) in interface `java.util.List`. We could construct cases with invalid typing in `List` if `T` were not invariant.

Wildcards can be employed to achieve covariance or contravariance in situations where the type parameter appears in one position only:

- `<? extends Foo>` for covariance (input positions)
- `<? super Foo>` for contravariance (output positions)

However, covariance is ineffective for the return type of a method since it is not an input position. Making it contravariant also has no effect since it is the receiver of the return value which must be contravariant (use-site variance in Java). Consequently, a return type containing wildcards is generally a mistake.

How_to_fix:

The solution to this problem depends on the original intention of the developer. Given the examples:

```
List<? extends Animal> getAnimals() { ... } // Noncompliant, wildcard with no use
List<? super Plant> getLifeforms() { ... }  // Noncompliant, wildcard with no use
```

You can remove the wildcards to make the types invariant:

```
List<Animal> getAnimals() { ... }           // Compliant, using invariant type instead
List<Plant> getLifeforms() { ... }          // Compliant, using invariant type instead
```

Or replace them with a super- or subtypes (still invariant):

```
List<Dog> getAnimals() { ... }              // Compliant, using subtype instead
List<Lifeform> getLifeforms() { ... }       // Compliant, using supertype instead
```

Resources:

Documentation

- The Java™ Tutorials - Wildcards

Articles & blog posts

- Sinisa Louc - A Complete Guide to Variance in Java and Scala
- Kotlin Expertise Blog - Kotlin Generics and Variance (Compared to Java)
- Wikipedia - Covariance and contravariance (computer science)
- Schneide Blog - Declaration-site and use-site variance explained
- Wikipedia - Liskov substitution principle

An abstract class should have both abstract and concrete methods (java:S1694)

Severidad: MINOR

Root_cause:

The purpose of an abstract class is to provide some heritable behaviors while also defining methods which must be implemented by sub-classes.

A class with no abstract methods that was made abstract purely to prevent instantiation should be converted to a concrete class (i.e. remove the `abstract` keyword) with a private constructor.

A class with only abstract methods and no inheritable behavior should be converted to an interface.

Noncompliant code example

```
public abstract class Animal {  // Noncompliant; should be an interface
  abstract void move();
  abstract void feed();
}

public abstract class Color {  // Noncompliant; should be concrete with a private constructor
  private int red = 0;
  private int green = 0;
  private int blue = 0;

  public int getRed() {
    return red;
  }
}
```

Compliant solution

```
public interface Animal {
  void move();
  void feed();
}

public class Color {
  private int red = 0;
  private int green = 0;
  private int blue = 0;

  private Color () {}

  public int getRed() {
    return red;
  }
}

public abstract class Lamp {

  private boolean switchLamp=false;

  public abstract void glow();

  public void flipSwitch() {
    switchLamp = !switchLamp;
    if (switchLamp) {
      glow();
    }
  }
}
```

"NullPointerException" should not be explicitly thrown (java:S1695)

Severidad: MAJOR

Root_cause:

A `NullPointerException` should indicate that a `null` value was unexpectedly encountered. Good programming practice dictates that code is structured to avoid NPE’s.

Explicitly throwing `NullPointerException` forces a method’s callers to explicitly catch it, rather than coding to avoid it. Further, it makes it difficult to distinguish between the unexpectedly-encountered `null` value and the condition which causes the method to purposely throw an NPE.

If an NPE is being thrown to indicate that a parameter to the method should not have been null, use the `@NotNull` annotation instead.

Noncompliant code example

```
public void doSomething (String aString) throws NullPointerException {
     throw new NullPointerException();
}
```

Compliant solution

```
public void doSomething (@NotNull String aString) {
}
```

"NullPointerException" should not be caught (java:S1696)

Severidad: MAJOR

Resources:

- CWE - CWE-395 - Use of NullPointerException Catch to Detect NULL Pointer Dereference
- CERT - ERR08-J. Do not catch NullPointerException or any of its ancestors

Root_cause:

`NullPointerException` should be avoided, not caught. Any situation in which `NullPointerException` is explicitly caught can easily be converted to a `null` test, and any behavior being carried out in the catch block can easily be moved to the "is null" branch of the conditional.

Noncompliant code example

```
public int lengthPlus(String str) {
  int len = 2;
  try {
    len += str.length();
  }
  catch (NullPointerException e) {
    log.info("argument was null");
  }
  return len;
}
```

Compliant solution

```
public int lengthPlus(String str) {
  int len = 2;

  if (str != null) {
    len += str.length();
  }
  else {
    log.info("argument was null");
  }
  return len;
}
```

Public methods should not contain selector arguments (java:S2301)

Severidad: MAJOR

Root_cause:

A selector argument is a `boolean` argument that's used to determine which of two paths to take through a method. Specifying such a parameter may seem innocuous, particularly if it's well named.

Unfortunately, the maintainers of the code calling the method won't see the parameter name, only its value. They'll be forced either to guess at the meaning or to take extra time to look the method up.

Instead, separate methods should be written.

This rule finds methods with a `boolean` that's used to determine which path to take through the method.

Noncompliant code example

```
public String tempt(String name, boolean ofAge) {
  if (ofAge) {
    offerLiquor(name);
  } else {
    offerCandy(name);
  }
}

// ...
public void corrupt() {
  tempt("Joe", false); // does this mean not to temp Joe?
}
```

Compliant solution

```
public void temptAdult(String name) {
  offerLiquor(name);
}

public void temptChild(String name) {
    offerCandy(name);
}

// ...
public void corrupt() {
  age < legalAge ? temptChild("Joe") : temptAdult("Joe");
}
```

"deleteOnExit" should not be used (java:S2308)

Severidad: MAJOR

Root_cause:

Use of `File.deleteOnExit()` is not recommended for the following reasons:

- The deletion occurs only in the case of a normal JVM shutdown but not when the JVM crashes or is killed.
- For each file handler, the memory associated with the handler is released only at the end of the process.

Noncompliant code example

```
File file = new File("file.txt");
file.deleteOnExit();  // Noncompliant
```

Files should not be empty (java:S2309)

Severidad: MINOR

Root_cause:

Files with no lines of code clutter a project and should be removed.

Noncompliant code example

```
//package org.foo;
//
//public class Bar {}
```

Methods returns should not be invariant (java:S3516)

Severidad: BLOCKER

Root_cause:

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
}
```

Zero should not be a possible denominator (java:S3518)

Severidad: CRITICAL

Introduction:

If the denominator to an integer division or remainder operation is zero, a `ArithmeticException` is thrown.

This error will crash your program in most cases. To fix it, you need to ensure that the denominator value in all division operations is always non-zero, or check the value against zero before performing the division.

Resources:

Documentation

- ArithmeticException
- The Division Operator in the JLS
- The Remainder Operator in the JLS

Standards

- CWE - CWE-369 - Divide by zero
- CERT, NUM02-J. - Ensure that division and remainder operations do not result in divide-by-zero errors
- STIG Viewer - Application Security and Development: V-222612 - The application must not be vulnerable to overflow attacks.

Root_cause:

A division (`/`) or remainder operation (`%`) by zero indicates a bug or logical error. This is because in Java, a division or remainder operation where the denominator is zero and not a floating point value always results in an `ArithmeticException` being thrown.

When working with `double` or `float` values, no exception will be thrown, but the operation will result in special floating point values representing either positive infinity, negative infinity, or `NaN`. Unless these special values are explicitly handled by a program, zero denominators should be avoided in floating point operations, too. Otherwise, the application might produce unexpected results.

What is the potential impact?

Issues of this type interrupt the normal execution of a program, causing it to crash or putting it into an inconsistent state. Therefore, this issue might impact the availability and reliability of your application, or even result in data loss.

If the computation of the denominator is tied to user input data, this issue can potentially even be exploited by attackers to disrupt your application.

Noncompliant code example

```
void test_divide() {
  int z = 0;
  if (unknown()) {
    // ..
    z = 3;
  } else {
    // ..
  }
  z = 1 / z; // Noncompliant, possible division by zero
}
```

Compliant solution

```
void test_divide() {
  int z = 0;
  if (unknown()) {
    // ..
    z = 3;
  } else {
    // ..
    z = 1;
  }
  z = 1 / z;
}
```

"Arrays.stream" should be used for primitive arrays (java:S3631)

Severidad: MAJOR

Root_cause:

For arrays of objects, `Arrays.asList(T ... a).stream()` and `Arrays.stream(array)` are basically equivalent in terms of performance. However, for arrays of primitives, using `Arrays.asList` will force the construction of a list of boxed types, and then use *that* list as a stream. On the other hand, `Arrays.stream` uses the appropriate primitive stream type (`IntStream`, `LongStream`, `DoubleStream`) when applicable, with much better performance.

Noncompliant code example

```
Arrays.asList("a1", "a2", "b1", "c2", "c1").stream()
    .filter(...)
    .forEach(...);
```

```
Arrays.asList(1, 2, 3, 4).stream() // Noncompliant
    .filter(...)
    .forEach(...);
```

Compliant solution

```
Arrays.asList("a1", "a2", "b1", "c2", "c1").stream()
    .filter(...)
    .forEach(...);

int[] intArray = new int[]{1, 2, 3, 4};
Arrays.stream(intArray)
    .filter(...)
    .forEach(...);
```

Spring "@Controller" classes should not use "@Scope" (java:S3750)

Severidad: MAJOR

Root_cause:

Spring `@Controllers`, `@Services`, and `@Repositorys` have `singleton` scope by default, meaning only one instance of the class is ever instantiated in the application. Defining any other scope for one of these class types will result in needless churn as new instances are created and destroyed. In a busy web application, this could cause a significant amount of needless additional load on the server.

This rule raises an issue when the `@Scope` annotation is applied to a `@Controller`, `@Service`, or `@Repository` with any value but "singleton". `@Scope("singleton")` is redundant, but ignored.

Noncompliant code example

```
@Scope("prototype")  // Noncompliant
@Controller
public class HelloWorld {
```

Compliant solution

```
@Controller
public class HelloWorld {
```

"@RequestMapping" methods should not be "private" (java:S3751)

Severidad: MAJOR

Resources:

- OWASP - Top 10 2017 Category A6 - Security Misconfiguration

Root_cause:

A method with a `@RequestMapping` annotation part of a class annotated with `@Controller` (directly or indirectly through a meta annotation - `@RestController` from Spring Boot is a good example) will be called to handle matching web requests. That will happen even if the method is `private`, because Spring invokes such methods via reflection, without checking visibility.

So marking a sensitive method `private` may seem like a good way to control how such code is called. Unfortunately, not all Spring frameworks ignore visibility in this way. For instance, if you've tried to control web access to your sensitive, `private`, `@RequestMapping` method by marking it `@Secured` … it will still be called, whether or not the user is authorized to access it. That's because AOP proxies are not applied to private methods.

In addition to `@RequestMapping`, this rule also considers the annotations introduced in Spring Framework 4.3: `@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping`, `@PatchMapping`.

Noncompliant code example

```
@RequestMapping("/greet", method = GET)
private String greet(String greetee) {  // Noncompliant
```

Compliant solution

```
@RequestMapping("/greet", method = GET)
public String greet(String greetee) {
```

Allowing both safe and unsafe HTTP methods is security-sensitive (java:S3752)

Severidad: MINOR

How_to_fix:

Recommended Secure Coding Practices

For all the routes/controllers of an application, the authorized HTTP methods should be explicitly defined and safe HTTP methods should only be used to perform read-only operations.

Compliant Solution

```
@RequestMapping("/delete_user", method = RequestMethod.POST)  // Compliant
public String delete1(String username) {
// state of the application will be changed here
}

@RequestMapping(path = "/delete_user", method = RequestMethod.POST) // Compliant
String delete2(@RequestParam("id") String id) {
// state of the application will be changed here
}
```

See

- OWASP - Top 10 2021 Category A1 - Broken Access Control
- OWASP - Top 10 2021 Category A4 - Insecure Design
- OWASP - Top 10 2017 Category A5 - Broken Access Control
- CWE - CWE-352 - Cross-Site Request Forgery (CSRF)
- OWASP: Cross-Site Request Forgery
- Spring Security Official Documentation: Use proper HTTP verbs (CSRF protection)

Assess_the_problem:

Ask Yourself Whether

- HTTP methods are not defined at all for a route/controller of the application.
- Safe HTTP methods are defined and used for a route/controller that can change the state of an application.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

```
@RequestMapping("/delete_user")  // Sensitive: by default all HTTP methods are allowed
public String delete1(String username) {
// state of the application will be changed here
}

@RequestMapping(path = "/delete_user", method = {RequestMethod.GET, RequestMethod.POST}) // Sensitive: both safe and unsafe methods are all
String delete2(@RequestParam("id") String id) {
// state of the application will be changed here
}
```

Root_cause:

An HTTP method is safe when used to perform a read-only operation, such as retrieving information. In contrast, an unsafe HTTP method is used to change the state of an application, for instance to update a user’s profile on a web application.

Common safe HTTP methods are GET, HEAD, or OPTIONS.

Common unsafe HTTP methods are POST, PUT and DELETE.

Allowing both safe and unsafe HTTP methods to perform a specific operation on a web application could impact its security, for example CSRF protections are most of the time only protecting operations performed by unsafe HTTP methods.

Default:

An HTTP method is safe when used to perform a read-only operation, such as retrieving information. In contrast, an unsafe HTTP method is used to change the state of an application, for instance to update a user’s profile on a web application.

Common safe HTTP methods are GET, HEAD, or OPTIONS.

Common unsafe HTTP methods are POST, PUT and DELETE.

Allowing both safe and unsafe HTTP methods to perform a specific operation on a web application could impact its security, for example CSRF protections are most of the time only protecting operations performed by unsafe HTTP methods.

Ask Yourself Whether

- HTTP methods are not defined at all for a route/controller of the application.
- Safe HTTP methods are defined and used for a route/controller that can change the state of an application.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

For all the routes/controllers of an application, the authorized HTTP methods should be explicitly defined and safe HTTP methods should only be used to perform read-only operations.

Sensitive Code Example

```
@RequestMapping("/delete_user")  // Sensitive: by default all HTTP methods are allowed
public String delete1(String username) {
// state of the application will be changed here
}

@RequestMapping(path = "/delete_user", method = {RequestMethod.GET, RequestMethod.POST}) // Sensitive: both safe and unsafe methods are all
String delete2(@RequestParam("id") String id) {
// state of the application will be changed here
}
```

Compliant Solution

```
@RequestMapping("/delete_user", method = RequestMethod.POST)  // Compliant
public String delete1(String username) {
// state of the application will be changed here
}

@RequestMapping(path = "/delete_user", method = RequestMethod.POST) // Compliant
String delete2(@RequestParam("id") String id) {
// state of the application will be changed here
}
```

See

- OWASP - Top 10 2021 Category A1 - Broken Access Control
- OWASP - Top 10 2021 Category A4 - Insecure Design
- OWASP - Top 10 2017 Category A5 - Broken Access Control
- CWE - CWE-352 - Cross-Site Request Forgery (CSRF)
- OWASP: Cross-Site Request Forgery
- Spring Security Official Documentation: Use proper HTTP verbs (CSRF protection)

"@Controller" classes that use "@SessionAttributes" must call "setComplete" on their "SessionStatus" objects (java:S3753)

Severidad: BLOCKER

Root_cause:

A Spring `@Controller` that uses `@SessionAttributes` is designed to handle a stateful / multi-post form. Such `@Controllers` use the specified `@SessionAttributes` to store data on the server between requests. That data should be cleaned up when the session is over, but unless `setComplete()` is called on the `SessionStatus` object from a `@RequestMapping` method, neither Spring nor the JVM will know it's time to do that. Note that the `SessionStatus` object must be passed to that method as a parameter.

Noncompliant code example

```
@Controller
@SessionAttributes("hello")  // Noncompliant; this doesn't get cleaned up
public class HelloWorld {

  @RequestMapping("/greet", method = GET)
  public String greet(String greetee) {
    return "Hello " + greetee;
  }
}
```

Compliant solution

```
@Controller
@SessionAttributes("hello")
public class HelloWorld {

  @RequestMapping("/greet", method = GET)
  public String greet(String greetee) {

    return "Hello " + greetee;
  }

  @RequestMapping("/goodbye", method = POST)
  public String goodbye(SessionStatus status) {
    //...
    status.setComplete();
  }

}
```

"HttpSecurity" URL patterns should be correctly ordered (java:S4601)

Severidad: CRITICAL

How_to_fix:

The following code is vulnerable because it defines access control configuration in the wrong order.

Noncompliant code example

```
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
      .antMatchers("/resources/**", "/signup", "/about").permitAll()
      .antMatchers("/admin/**").hasRole("ADMIN")
      .antMatchers("/admin/login").permitAll() // Noncompliant
      .antMatchers("/**", "/home").permitAll()
      .antMatchers("/db/**").access("hasRole('ADMIN') and hasRole('DBA')") // Noncompliant
      .and().formLogin().loginPage("/login").permitAll().and().logout().permitAll();
  }
```

Compliant solution

```
  protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
      .antMatchers("/resources/**", "/signup", "/about").permitAll()
      .antMatchers("/admin/login").permitAll()
      .antMatchers("/admin/**").hasRole("ADMIN")
      .antMatchers("/db/**").access("hasRole('ADMIN') and hasRole('DBA')")
      .antMatchers("/**", "/home").permitAll()
      .and().formLogin().loginPage("/login").permitAll().and().logout().permitAll();
  }
```

Resources:

Documentation

- Spring Documentation - Authorize HttpServletRequests

Standards

- OWASP - Top 10 2021 - Category A1 - Broken Access Control
- OWASP - Top 10 2017 - Category A6 - Security Misconfiguration
- CWE - CWE-285 - Improper Authorization
- CWE - CWE-287 - Improper Authentication

Introduction:

Spring Framework, and, more precisely, the Spring Security component, allows setting up access control checks at the URI level. This is done by adding request matchers to the security configuration, each authorizing access to some resources depending on the incoming request entitlement.

Similarly to firewall filtering rules, the order in which those matchers are defined is security relevant.

Root_cause:

Configured URL matchers are considered in the order they are declared. Especially, for a given resource, if a looser filter is defined before a stricter one, only the less secure configuration will apply. No request will ever reach the stricter rule.

This rule raises an issue when:

- A URL pattern ending with `**` precedes another one having the same prefix. E.g. `/admin/**` is defined before `/admin/example/**`
- A pattern without wildcard characters is preceded by another one that matches it. E.g.: `/page-index/db` is defined after `/page*/**`

What is the potential impact?

Access control rules that have been defined but cannot be applied generally indicate an error in the filtering process. In most cases, this will have consequences on the application's authorization and authentication mechanisms.

Authentication bypass

When the ignored access control rule is supposed to enforce the authentication on a resource, the consequence is a bypass of the authentication for that resource. Depending on the scope of the ignored rule, a single feature or whole sections of the application can be left unprotected.

Attackers could take advantage of such an issue to access the affected features without prior authentication, which may impact the confidentiality or integrity of sensitive, business, or personal data.

Privilege escalation

When the ignored access control rule is supposed to verify the role of an authenticated user, the consequence is a privilege escalation or authorization bypass. An authenticated user with low privileges on the application will be able to access more critical features or sections of the application.

This could have financial consequences if the accessed features are normally accessed by paying users. It could also impact the confidentiality or integrity of sensitive, business, or personal data, depending on the features.

"setDaemon", "setPriority" and "getThreadGroup" should not be invoked on virtual threads (java:S6901)

Severidad: MAJOR

Root_cause:

The `Thread` class has some methods that are used to monitor and manage its execution. With the introduction of virtual threads in Java 21, there are three of these methods that behave differently between the standard platform threads and the virtual ones.

For virtual threads:

- `Thread.setDaemon(boolean)` will throw an `IllegalArgumentException` if `false` is passed as an argument as a virtual thread daemon status is always true.
- `Thread.setPriority(int priority)` will never change the actual priority of a virtual thread, which is always equal to `Thread.NORM_PRIORITY`
- `Thread.getThreadGroup()` will return a dummy "VirtualThreads" group that is empty and should not be used

This rule reports an issue when one of these methods is invoked on a virtual thread.

Noncompliant code example

```
Thread t = Thread.ofVirtual().unstarted(()->{/* some task */});
t.setPriority(1); // Noncompliant; virtual threads' priority cannot be changed
t.setDaemon(false); // Noncompliant; will throw IllegalArgumentException
t.setDaemon(true); // Noncompliant; redundant
t.start();
var threadGroup = t.getThreadGroup(); // Noncompliant; virtual thread groups should not be used
```

Resources:

Documentation

- Java Documentation - Thread.setDaemon
- Java Documentation - Thread.setPriority
- Java Documentation - Thread.getThreadGroup
- Java Documentation - Virtual threads group

Avoid using "FetchType.EAGER" (java:S6904)

Severidad: MAJOR

Introduction:

`FetchType` is an enumeration in the Java Persistence API (JPA) that is used to define the fetching strategy for associations (relationships) between entities in a relational database.

There are two main values for FetchType:

- `FetchType.EAGER`: the association is loaded immediately when the owning entity is loaded.
- `FetchType.LAZY`: the association is not loaded unless it is explicitly accessed.

This rule raises an issue when the `fetch` argument is explicitly set to `FetchType.EAGER`.

Resources:

Documentation

- Java EE 7 API - FetchType

How_to_fix:

Remove or replace `FetchType.EAGER` with `FetchType.LAZY` in JPA annotations.

Noncompliant code example

```
@OneToMany(mappedBy = "parent", fetch = FetchType.EAGER) // Noncompliant
private List<ChildEntity> children;

@OneToMany(mappedBy = "child", fetch = FetchType.EAGER) // Noncompliant
private List<ParentEntity> parents;
```

Compliant solution

```
@OneToMany(mappedBy = "parent", fetch = FetchType.LAZY) // Compliant
private List<ChildEntity> children;

@OneToMany(mappedBy = "child") // Compliant
private List<ParentEntity> parents;
```

Root_cause:

Using `FetchType.EAGER` can lead to inefficient data loading and potential performance issues. Eager Loading initializes associated data on the spot, potentially fetching more data than needed.

Virtual threads should not run tasks that include synchronized code (java:S6906)

Severidad: MAJOR

Resources:

Documentation

- Java Documentation - Virtual threads, pinning scenarios

Root_cause:

Java 21 virtual threads allow the JVM to optimize the usage of OS threads, by mounting and unmounting them on an OS thread when needed, and making them more efficient when dealing with blocking operations such as HTTP requests or I/O.

However, when code is executed inside a `synchronized` block or `synchronized` method, the virtual thread stays pinned to the underlying OS thread and cannot be unmounted during a blocking operation. This will cause the OS thread to be blocked, which can impact the scalability of the application.

Therefore, virtual threads should not execute code that contains `synchronized` blocks or invokes `synchronized` methods. Platform threads should be used in these cases.

This rule raises an issue when a virtual thread contains `synchronized` blocks or invokes `synchronized` methods.

Noncompliant code example

```
void enqueue(){
    Thread.startVirtualThread(() -> { // Noncompliant; use a platform thread instead
            setupOperations();
            dequeLogic();
        }
    });
}
```

Compliant solution

```
void enqueue(){
    new Thread(() -> {
        synchronized {
            setupOperations();
            dequeLogic();
        }
    }).start();
}
```

Noncompliant code example

```
void enqueue2(){
    Thread.startVirtualThread(() -> { // Noncompliant; use a platform thread instead of a virtual one
        if(someCondition){
            synchronizedMethod();
        }else{
            defaultLogic();
        }
    });
}
synchronized void synchronizedMethod(){}
void defaultLogic(){}
```

Compliant solution

```
void enqueue2(){
    new Thread(() -> {
        if(someCondition){
            synchronizedMethod();
        }else{
            defaultLogic();
        }
    }).start();
}
synchronized void synchronizedMethod(){}
void defaultLogic(){}
```

Constant parameters in a "PreparedStatement" should not be set more than once (java:S6909)

Severidad: MAJOR

Root_cause:

The `PreparedStatement` is frequently used in loops because it allows to conveniently set parameters. A small optimization is possible by setting constant parameters outside the loop or hard-coding them in the query whenever possible.

What is the potential impact?

- *Performance*: the unnecessary calls to the setter methods bring overhead.
- *Sustainability*: the extra overhead has a negative impact on the environment.

Resources:

Documentation

- Oracle SDK - PreparedStatement
- Oracle Tutorial - Using Prepared Statements

Introduction:

The `java.sql.PreparedStatement` represents a precompiled SQL statement that can be efficiently executed multiple times.

How_to_fix:

Place calls to setter methods that take a constant argument outside the loop.

Noncompliant code example

```
public class DatabaseExample {

    public record Order(String id, BigDecimal price) {}

    public void updateTodayOrders(Connection connection, List<Order> orders) {
            Date today = java.sql.Date.valueOf(LocalDate.now());
            String insertQuery = "INSERT INTO Order (id, price, executionDate) VALUES (?, ?, ?)";
            PreparedStatement preparedStatement = connection.prepareStatement(insertQuery);

            for(Order order: orders) {
                preparedStatement.setString(1, order.id());
                preparedStatement.setString(2, order.price());
                preparedStatement.setDate(3, today); // Noncompliant
                preparedStatement.executeUpdate();
            }
    }
}
```

Compliant solution

```
public class DatabaseExample {

    public record Order(String id, BigDecimal price) {}

    public void updateTodayOrders(Connection connection, List<Order> orders) {
            Date today = java.sql.Date.valueOf(LocalDate.now());
            String insertQuery = "INSERT INTO Order (id, price, executionDate) VALUES (?, ?, ?)";
            PreparedStatement preparedStatement = connection.prepareStatement(insertQuery);

            preparedStatement.setDate(3, today); // Compliant
            for(Order order: orders) {
                preparedStatement.setString(1, order.id());
                preparedStatement.setString(2, order.price());
                preparedStatement.executeUpdate();
            }
    }
}
```

Class variable fields should not have public accessibility (java:S1104)

Severidad: MINOR

Root_cause:

Public fields in public classes do not respect the encapsulation principle and have 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.

To prevent unauthorized modifications, private attributes and accessor methods (set and get) should be used.

What is the potential impact?

Public fields can be modified by any part of the code and this can lead to unexpected changes and hard-to-trace bugs.

Public fields don’t hide the implementation details. As a consequence, it is no longer possible to change how the data is stored internally without impacting the client code of the class.

The code is harder to maintain.

Exceptions

This rule ignores `public final` fields because they are not modifiable. Also, annotated fields, whatever the annotation(s) will be ignored, as annotations are often used by injection frameworks, which in exchange require having public fields.

Resources:

- CWE - CWE-493 - Critical Public Variable Without Final Modifier

How_to_fix:

Depending on your need there are multiple options:

- Encapsulate the field
    1. Make the field private.

2. Define methods to get and set the value of the field.
   These methods are commonly known as getter and setter methods and are prefixed by `get` and `set` followed by the name of the field. *Note:* as a bonus it is now possible to monitor value changes using breakpoints.
- Mark the field as `public final` if it is not supposed to change.

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;

  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

}
```

How does this work?

By having a setter and a getter the code can control how the field is accessed and modified. For example, adding validation in the setter method will ensure that only valid values are set.

The access modifiers on the setter can also be changed to `private` or `protected` to restrain which code can modify the value.

An open curly brace should be located at the beginning of a line (java:S1106)

Severidad: MINOR

Root_cause:

Shared coding conventions make it possible to collaborate efficiently. This rule makes it mandatory to place the open curly brace at the beginning of a line.

Noncompliant code example

```
public void myMethod {  // Noncompliant
  if(something) {  // Noncompliant
    executeTask();
  } else {  // Noncompliant
    doSomethingElse();
  }
}
```

Compliant solution

```
public void myMethod
{
  if(something)
  {
    executeTask();
  } else
  {
    doSomethingElse();
  }
}
```

Close curly brace and the next "else", "catch" and "finally" keywords should be located on the same line (java:S1107)

Severidad: MINOR

Root_cause:

Shared coding conventions make it possible for a team to collaborate efficiently.

This rule makes it mandatory to place closing curly braces on the same line as the next `else`, `catch` or `finally` keywords.

Noncompliant code example

```
public void myMethod() {
  if(something) {
    executeTask();
  } else if (somethingElse) {
    doSomethingElse();
  }
  else {                               // Noncompliant
     generateError();
  }

  try {
    generateOrder();
  } catch (Exception e) {
    log(e);
  }
  finally {                            // Noncompliant
    closeConnection();
  }
}
```

Compliant solution

```
public void myMethod() {
  if(something) {
    executeTask();
  } else if (somethingElse) {
    doSomethingElse();
  } else {
     generateError();
  }

  try {
    generateOrder();
  } catch (Exception e) {
    log(e);
  } finally {
    closeConnection();
  }
}
```

Close curly brace and the next "else", "catch" and "finally" keywords should be on two different lines (java:S1108)

Severidad: MINOR

Root_cause:

Shared coding conventions make it possible for a team to collaborate efficiently.

This rule makes it mandatory to place a closing curly brace and the next `else`, `catch` or `finally` keyword on two different lines.

Noncompliant code example

```
public void myMethod() {
  if(something) {
    executeTask();
  } else if (somethingElse) {          // Noncompliant
    doSomethingElse();
  }
  else {                               // Compliant
     generateError();
  }

  try {
    generateOrder();
  } catch (Exception e) {
    log(e);
  }
  finally {
    closeConnection();
  }
}
```

Compliant solution

```
public void myMethod() {
  if(something) {
    executeTask();
  }
  else if (somethingElse) {
    doSomethingElse();
  }
  else {
     generateError();
  }

  try {
    generateOrder();
  }
  catch (Exception e) {
    log(e);
  }
  finally {
    closeConnection();
  }
}
```

A close curly brace should be located at the beginning of a line (java:S1109)

Severidad: MINOR

Root_cause:

Shared coding conventions make it possible for a team to efficiently collaborate. This rule makes it mandatory to place a close curly brace at the beginning of a line.

Noncompliant code example

```
if(condition) {
  doSomething();}
```

Compliant solution

```
if(condition) {
  doSomething();
}
```

Exceptions

When blocks are inlined (open and close curly braces on the same line), no issue is triggered.

```
if(condition) {doSomething();}
```

Execution of the Garbage Collector should be triggered only by the JVM (java:S1215)

Severidad: CRITICAL

Root_cause:

Calling `System.gc()` or `Runtime.getRuntime().gc()` is a bad idea for a simple reason: there is no way to know exactly what will be done under the hood by the JVM because the behavior will depend on its vendor, version and options:

- Will the whole application be frozen during the call?
- Is the `-XX:DisableExplicitGC` option activated?
- Will the JVM simply ignore the call?
- …

Like for `System.gc()`, there is no reason to manually call `runFinalization()` to force the call of finalization methods of any objects pending finalization.

An application relying on these unpredictable methods is also unpredictable and therefore broken. The task of running the garbage collector and calling `finalize()` methods should be left exclusively to the JVM.

The default unnamed package should not be used (java:S1220)

Severidad: MINOR

Resources:

Articles & blog posts

- Baeldung - Guide to Java Packages
- tutorialspoint - What are the best practices to keep in mind while using packages in Java?

Root_cause:

Java packages serve two purposes:

1. Structure — Packages give a structure to the set of classes of your project. It is a bad practice to put all classes flat into the source directory of a project without a package structure. A structure helps to mentally break down a project into smaller parts, simplifying readers' understanding of how components are connected and how they interact.
2. Avoiding name clashes — a class part of the *default package* if no explicit package name is specified. This can easily cause name collisions when other projects define a class of the same name.

When no package is explicitly specified for the classes in your project, this makes the project harder to understand and may cause name collisions with other projects. Also, classes located in the default package not be accessed from classes within named packages since Java 1.4.

How_to_fix:

Move your class to a package directory and explicitly state the package's name at the top of the class. If your project does not have a package structure, think of a structure that fits your needs. The package names should be unique to your project. You can find some best practices when choosing package names in the Ressources section below.

Noncompliant code example

```
public class MyClass { /* ... */ } // Noncompliant, no package spacified
```

Compliant solution

```
package org.example; // Compliant

public class MyClass{ /* ... */ }
```

Methods should not be named "tostring", "hashcode" or "equal" (java:S1221)

Severidad: MAJOR

Root_cause:

Due to the similar name with the methods `Object.toString`, `Object.hashCode` and `Object.equals`, there is a significant likelihood that a developer intended to override one of these methods but made a spelling error.

Even if no such error exists and the naming was done on purpose, these method names can be misleading. Readers might not notice the difference, or if they do, they may falsely assume that the developer made a mistake.

How_to_fix:

If you intended to override one of the methods `Object.toString`, `Object.hashCode`, or `Object.equals`, correct the spelling. Also, you should add the `@Override` modifier, which causes a compiler error message in case the annotated method does not override anything.

If the naming was done on purpose, you should rename the methods to be more distinctive.

Noncompliant code example

```
public int hashcode() { /* ... */ }             // Noncompliant

public String tostring() { /* ... */ }          // Noncompliant

public boolean equal(Object obj) { /* ... */ }  // Noncompliant
```

Compliant solution

```
@Override
public int hashCode() { /* ... */ }             // Compliant

@Override
public String toString() { /* ... */ }          // Compliant
```

```
@Override
public boolean equals(Object obj) { /* ... */ } // Compliant
```

Non-constructor methods should not have the same name as the enclosing class (java:S1223)

Severidad: MAJOR

Root_cause:

Having a class and some of its methods sharing the same name is misleading, and leaves others to wonder whether it was done that way on purpose, or was the methods supposed to be a constructor.

Noncompliant code example

```
public class Foo {
   public Foo() {...}
   public void Foo(String label) {...}  // Noncompliant
}
```

Compliant solution

```
public class Foo {
   public Foo() {...}
   public void foo(String label) {...}  // Compliant
}
```

Packages should have a javadoc file 'package-info.java' (java:S1228)

Severidad: MINOR

Root_cause:

Each package in a Java project should include a `package-info.java` file. The purpose of this file is to document the Java package using javadoc and declare package annotations.

Compliant solution

```
/**
* This package has non null parameters and is documented.
**/
@ParametersAreNonnullByDefault
package org.foo.bar;
```

Constructors should only call non-overridable methods (java:S1699)

Severidad: CRITICAL

Root_cause:

Calling an overridable method from a constructor could result in failures or strange behaviors when instantiating a subclass which overrides the method.

For example:

- The subclass class constructor starts by contract by calling the parent class constructor.
- The parent class constructor calls the method, which has been overridden in the child class.
- If the behavior of the child class method depends on fields that are initialized in the child class constructor, unexpected behavior (like a `NullPointerException`) can result, because the fields aren’t initialized yet.

Noncompliant code example

```
public class Parent {

  public Parent () {
    doSomething();  // Noncompliant
  }

  public void doSomething () {  // not final; can be overridden
    ...
  }
}

public class Child extends Parent {
  private String foo;

  public Child(String foo) {
    super(); // leads to call doSomething() in Parent constructor which triggers a NullPointerException as foo has not yet been initialized
    this.foo = foo;
  }

  public void doSomething () {
    System.out.println(this.foo.length());
  }

}
```

Resources:

- CERT, MET05-J. - Ensure that constructors do not call overridable methods
- CERT, OOP50-CPP. - Do not invoke virtual functions from constructors or destructors

---

Unnecessary bit operations should not be performed (java:S2437)

Severidad: BLOCKER

Root_cause:

Certain bit operations are just silly and should not be performed because their results are predictable.

Specifically, using `& -1` with any value will always result in the original value, as will `anyValue ^ 0` and `anyValue | 0`.

"Thread" should not be used where a "Runnable" argument is expected (java:S2438)

Severidad: MAJOR

Root_cause:

The semantics of `Thread` and `Runnable` are different, and while it is technically correct to use `Thread` where a `Runnable` is expected, it is a bad practice to do so.

The crux of the issue is that `Thread` is a larger concept than `Runnable`. A `Runnable` represents a task. A `Thread` represents a task and its execution management (ie: how it should behave when started, stopped, resumed, …). It is both a task and a lifecycle management.

Noncompliant code example

```
public static void main(String[] args) {
        Thread runnable = new Thread() {
                @Override
                public void run() { /* ... */ }
        };
        new Thread(runnable).start();  // Noncompliant
}
```

Compliant solution

```
public static void main(String[] args) {
        Runnable runnable = new Runnable() {
                @Override
                public void run() { /* ... */ }
        };
        new Thread(runnable).start();
}
```

The value returned from a stream read should be checked (java:S2674)

Severidad: MINOR

Resources:

- CERT, FIO10-J. - Ensure the array is filled when using read() to fill an array

Root_cause:

You cannot assume that any given stream reading call will fill the `byte[]` passed in to the method. Instead, you must check the value returned by the read method to see how many bytes were read. Fail to do so, and you introduce bug that is both harmful and difficult to reproduce.

Similarly, you cannot assume that `InputStream.skip` will actually skip the requested number of bytes, but must check the value returned from the method.

This rule raises an issue when an `InputStream.read` method that accepts a `byte[]` is called, but the return value is not checked, and when the return value of `InputStream.skip` is not checked. The rule also applies to `InputStream` child classes.

Noncompliant code example

```
public void doSomething(String fileName) {
  try {
    InputStream is = new InputStream(file);
    byte [] buffer = new byte[1000];
    is.read(buffer);  // Noncompliant
    // ...
  } catch (IOException e) { ... }
}
```

Compliant solution

```
public void doSomething(String fileName) {
  try {
    InputStream is = new InputStream(file);
    byte [] buffer = new byte[1000];
    int count = 0;
    while (count = is.read(buffer) > 0) {
      // ...
    }
  } catch (IOException e) { ... }
}
```

"readObject" should not be "synchronized" (java:S2675)

Severidad: MAJOR

Root_cause:

The `readObject` method is implemented when a `Serializable` object requires special handling to be reconstructed from a file. The object created by `readObject` is accessed only by the thread that called the method, thus using the `synchronized` keyword in this context is unnecessary and causes confusion.

Noncompliant code example

```
private synchronized void readObject(java.io.ObjectInputStream in)
     throws IOException, ClassNotFoundException { // Noncompliant
  //...
}
```

Compliant solution

```
private void readObject(java.io.ObjectInputStream in)
     throws IOException, ClassNotFoundException { // Compliant
  //...
}
```

Resources:

Documentation

- Oracle SDK 20 - Serializable
- Oracle SDK 20 - ObjectInputStream

Articles & blog posts

- Serialization in Java

"Math.abs" and negation should not be used on numbers that could be "MIN_VALUE" (java:S2676)

Severidad: MINOR

Resources:

- Oracle SDK 20 - Math.abs(int)

Root_cause:

This rule involves the use of `Math.abs` and negation on numbers that could be `MIN_VALUE`. It is a problem because it can lead to incorrect results and unexpected behavior in the program.

When `Math.abs` and negation are used on numbers that could be `MIN_VALUE`, the result can be incorrect due to integer overflow. Common methods that can return a `MIN_VALUE` and raise an issue when used together with `Math.abs` are:

- `Random.nextInt()` and `Random.nextLong()`
- `hashCode()`
- `compareTo()`

Alternatively, the `absExact()` method throws an `ArithmeticException` for `MIN_VALUE`.

Noncompliant code example

```
public void doSomething(String str) {
  if (Math.abs(str.hashCode()) > 0) { // Noncompliant
    // ...
  }
}
```

Compliant solution

```
public void doSomething(String str) {
  if (str.hashCode() != 0) {
    // ...
  }
}
```

"read" and "readLine" return values should be used (java:S2677)

Severidad: MAJOR

Root_cause:

The `Reader.read()` and the `BufferedReader.readLine()` are used for reading data from a data source. The return value of these methods is the data read from the data source, or `null` when the end of the data source is reached. If the return value is ignored, the data read from the source is thrown away and may indicate a bug.

This rule raises an issue when the return values of `Reader.read()` and `BufferedReader.readLine()` and their subclasses are ignored or merely null-checked.

Noncompliant code example

```
public void doSomethingWithFile(String fileName) {
  try(BufferedReader buffReader = new BufferedReader(new FileReader(fileName))) {
    while (buffReader.readLine() != null) { // Noncompliant
      // ...
    }
  } catch (IOException e) {
    // ...
  }
}
```

Compliant solution

```
public void doSomethingWithFile(String fileName) {
  try(BufferedReader buffReader = new BufferedReader(new FileReader(fileName))) {
    String line = null;
    while ((line = buffReader.readLine()) != null) {
      // ...
    }
  } catch (IOException e) {
    // ...
  }
}
```

Resources:

- Oracle SDK 20 - Reader.read()
- Oracle SDK 20 - BufferedReader.readLine()

"null" should not be used with "Optional" (java:S2789)

Severidad: MAJOR

Resources:

Documentation

- Oracle SDK 20 - Optional

Articles & blog posts

- Java Optional Guide

How_to_fix:

There are a few ways to fix this issue:

- Avoid returning `null` from a method whose return type is `Optional`.
- Remove the null-check of an `Optional` and use `Optional` methods instead, like `isPresent()` or `ifPresent()`.

Noncompliant code example

```
public void doSomething () {
  Optional<String> optional = getOptional();
  if (optional != null) {  // Noncompliant
    // do something with optional...
  }
  Optional<String> text = null; // Noncompliant, a variable whose type is Optional should never itself be null
  // ...
}

@Nullable // Noncompliant
public Optional<String> getOptional() {
  // ...
  return null;  // Noncompliant
}
```

Compliant solution

```
public void doSomething () {
  Optional<String> optional = getOptional();
  optional.ifPresent(
    // do something with optional...
  );
  Optional<String> text = Optional.empty();
  // ...
}

public Optional<String> getOptional() {
  // ...
  return Optional.empty();
}
```

Root_cause:

`Optional` acts as a container object that may or may not contain a non-null value. It is introduced in Java 8 to help avoid `NullPointerException`. It provides methods to check if a value is present and retrieve the value if it is present.

`Optional` is used instead of `null` values to make the code more readable and avoid potential errors.

It is a bad practice to use `null` with `Optional` because it is unclear whether a value is present or not, leading to confusion and potential `NullPointerException` errors.

Methods should not return constants (java:S3400)

Severidad: MINOR

Root_cause:

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 final int BEST_NUMBER = 12;
```

Exceptions

The following types of method are ignored:

- methods that override a method.
- methods that are not final (not having the `final`, `private` or `static` modifier and not in a record or a final class).
- methods with annotations, such as `@Override` or Spring’s `@RequestMapping`.

Arrays should not be created for varargs parameters (java:S3878)

Severidad: MINOR

Root_cause:

There’s no point in creating an array solely for the purpose of passing it as a varargs (`...`) argument; varargs *is* an array. Simply pass the elements directly. They will be consolidated into an array automatically. Incidentally passing an array where `Object ...` is expected makes the intent ambiguous: Is the array supposed to be one object or a collection of objects?

Noncompliant code example

```
public void callTheThing() {
  //...
  doTheThing(new String[] { "s1", "s2"});  // Noncompliant: unnecessary
  doTheThing(new String[12]);  // Compliant
  doTheOtherThing(new String[8]);  // Noncompliant: ambiguous
  // ...
}

public void doTheThing (String ... args) {
  // ...
}

public void doTheOtherThing(Object ... args) {
  // ...
}
```

Compliant solution

```
public void callTheThing() {
  //...
  doTheThing("s1", "s2");
  doTheThing(new String[12]);
  doTheOtherThing((Object[]) new String[8]);
   // ...
}

public void doTheThing (String ... args) {
  // ...
}

public void doTheOtherThing(Object ... args) {
  // ...
}
```

Spring beans should be considered by "@ComponentScan" (java:S4605)

Severidad: CRITICAL

Root_cause:

Spring beans belonging to packages that are not included in a `@ComponentScan` configuration will not be accessible in the Spring Application Context. Therefore, it’s likely to be a configuration mistake that will be detected by this rule.

Note: the `@ComponentScan` is implicit in the `@SpringBootApplication` annotation, case in which Spring Boot will auto scan for components in the package containing the Spring Boot main class and its sub-packages.

Noncompliant code example

```
package com.mycompany.app;

@Configuration
@ComponentScan("com.mycompany.app.beans")
public class Application {
...
}

package com.mycompany.app.web;

@Controller
public class MyController { // Noncompliant; MyController belong to "com.mycompany.app.web" while the ComponentScan is looking for beans in
...
}
```

Compliant solution

```
package com.mycompany.app;

@Configuration
@ComponentScan({"com.mycompany.app.beans","com.mycompany.app.web"})
or
@ComponentScan("com.mycompany.app")
or
@ComponentScan
public class Application {
...
}

package com.mycompany.app.web;

@Controller
public class MyController { // "com.mycompany.app.web" is referenced by a @ComponentScan annotated class
...
}
```

Java features should be preferred to Guava (java:S4738)

Severidad: MAJOR

Root_cause:

Some Guava features were really useful for Java 7 application because Guava was bringing APIs missing in the JDK. Java 8 fixed some of these limitations. When migrating an application to Java 8 or even when starting a new one, it's recommended to prefer Java 8 APIs over Guava ones to ease its maintenance: developers don't need to learn how to use two APIs and can stick to the standard one.

Java 9 brought even more useful methods to the standard Java library and if Java version is equal to or higher than 9, these standard methods should be used.

This rule raises an issue when the following Guava APIs are used:

Guava API	Java 8 API
com.google.common.io.BaseEncoding#base64()	java.util.Base64
com.google.common.io.BaseEncoding#base64Url()	java.util.Base64
com.google.common.base.Joiner.on()	java.lang.String#join() or java.util.stream.Collectors#joining()
com.google.common.base.Optional#of()	java.util.Optional#of()
com.google.common.base.Optional#absent()	java.util.Optional#empty()
com.google.common.base.Optional#fromNullable()	java.util.Optional#ofNullable()
com.google.common.base.Optional	java.util.Optional
com.google.common.base.Predicate	java.util.function.Predicate
com.google.common.base.Function	java.util.function.Function
com.google.common.base.Supplier	java.util.function.Supplier
com.google.common.io.Files.createTempDir	java.nio.file.Files.createTempDirectory

Guava API	Java 9 API
com.google.common.collect.ImmutableSet#of()	java.util.Set#of()

Guava API	Java 9 API
com.google.common.collect.ImmutableList#of()	java.util.List#of()
com.google.common.collect.ImmutableMap#of()	java.util.Map#of() or java.util.Map#ofEntries()

The upper bound of type variables and wildcards should not be "final" (java:S4968)

Severidad: MINOR

Root_cause:

When a type variable or a wildcard declares an upper bound that is `final`, the parametrization is not generic at all because it accepts one and only one type at runtime: the one that is `final`. Instead of using `Generics`, it’s simpler to directly use the concrete `final` class.

Noncompliant code example

```
public static <T extends String> T getMyString() { // Noncompliant; String is a "final" class and so can't be extended
 [...]
}
```

Compliant solution

```
public static String getMyString() { // Compliant
  [...]
}
```

Derived exceptions should not hide their parents' catch blocks (java:S4970)

Severidad: CRITICAL

Root_cause:

The `catch` block of a checked exception "E" may be hidden because the corresponding `try` block only throws exceptions derived from E.

These derived exceptions are handled in dedicated `catch` blocks prior to the `catch` block of the base exception E.

The `catch` block of E is unreachable and should be considered dead code. It should be removed, or the entire try-catch structure should be refactored.

It is also possible that a single exception type in a multi-catch block may be hidden while the catch block itself is still reachable. In that case it is enough to only remove the hidden exception type or to replace it with another type.

Noncompliant code example

```
public class HiddenCatchBlock {

  public static class CustomException extends Exception {
  }

  public static class CustomDerivedException extends CustomException {
  }

  public static void main(String[] args) {
    try {
      throwCustomDerivedException();
    } catch(CustomDerivedException e) {
      // ...
    } catch(CustomException e) { // Noncompliant; this code is unreachable
      // ...
    }
  }

  private static void throwCustomDerivedException() throws CustomDerivedException {
    throw new CustomDerivedException();
  }
}
```

Compliant solution

```
public class HiddenCatchBlock {

  public static class CustomException extends Exception {
  }

  public static class CustomDerivedException extends CustomException {
  }
  public static void main(String[] args) {
    try {
      throwCustomDerivedException();
    } catch(CustomDerivedException e) { // Compliant; try-catch block is "catching" only the Exception that can be thrown in the "try"
      //...
    }
  }
}
```

Strings and Boxed types should be compared using "equals()" (java:S4973)

Severidad: MAJOR

Root_cause:

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 && firstName.equals(lastName)) { ... };
```

Resources:

- CWE - CWE-595 - Comparison of Object References Instead of Object Contents
- CWE - 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

---

Type parameters should not shadow other type parameters (java:S4977)

Severidad: MINOR

Root_cause:

Shadowing makes it impossible to use the type parameter from the outer scope. Also, it can be confusing to distinguish which type parameter is being used.

This rule raises an issue when a type parameter from an inner scope uses the same name as one in an outer scope.

Noncompliant code example

```
 public class TypeParameterHidesAnotherType<T> {

    public class Inner<T> { // Noncompliant
      //...
    }

    private <T> T method() { // Noncompliant
      return null;
    }

  }
```

Compliant solution

```
public class NoTypeParameterHiding<T> {

    public class Inner<S> { // Compliant
      List<S> listOfS;
    }

    private <V> V method() { // Compliant
      return null;
    }
```

```
  }
```

SQL queries should retrieve only necessary fields (java:S6905)

Severidad: MAJOR

Root_cause:

A common reason for a poorly performant query is because it’s processing more data than required.

Querying unnecessary data demands extra work on the server, adds network overhead, and consumes memory and CPU resources on the application server. The effect is amplified when the query includes multiple *joins*.

The rule flags an issue when a `SELECT *` query is provided as an argument to methods in `java.sql.Connection` and `java.sql.Statement`.

What is the potential impact?

- *Performance*: the unnecessary extra data being processed brings overhead.
- *Sustainability*: the extra resources used have a negative impact on the environment.

How_to_fix:

Make the `SELECT *` an explicit selection of the required fields.

Noncompliant code example

```
public class OrderRepository {

    public record OrderSummary(String name, String orderId, BigDecimal price) { }

    public List<OrderSummary> queryOrderSummaries(Connection conn) {
            String sql = "SELECT * " +                                                  // Noncompliant
                          "FROM Orders JOIN Customers ON Orders.customerId = Customers.id ";

            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(sql);

            return convertResultToOrderSummaryList(rs);
    }
}
```

Compliant solution

```
public class OrderRepository {

    public record OrderSummary(String name, String orderId, BigDecimal price) { }

    public List<OrderSummary> queryOrderSummaries(Connection conn) {
            String sql = "SELECT Customers.name, Orders.id, Orders.price " +            // Compliant
                          "FROM Orders JOIN Customers ON Orders.customerId = Customers.id ";

            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(sql);

            return convertResultToOrderSummaryList(rs);
    }
}
```

Introduction:

The *Java Database Connectivity (JDBC) API* provides the `java.sql.Statement` interface that allows to represent an SQL statement and to execute queries with the database.

Resources:

Documentation

- Oracle SDK - Statement
- O’Reilly - High Performance MySQL - Query Performance Optimization

Articles & blog posts

- GeeksforGeeks - Difference Between Execute(), query() and Update() Methods in Java

Use batch Processing in JDBC (java:S6912)

Severidad: MAJOR

Resources:

Documentation

- Oracle Java SE 21 API - java.sql.Statement
- Oracle Java SE 21 API - java.sql.PreparedStatement

Articles & blog posts

- Baeldung - JDBC Batch Processing

Root_cause:

Executing a batch of SQL queries instead of individual queries improves performance by reducing communication overhead with the database.

Batching SQL statements is beneficial in common situations where a SQL statement is executed within a loop. In such cases, adding the statement to a batch and subsequently executing it reduces the number of interactions with the database. This results in improved efficiency and faster execution times.

The rule raises an issue when it detects a `java.sql.Statement` being executed within a loop instruction, such as `for`, `while` or the `forEach` method of `java.lang.Iterable`, `java.util.Map` and `java.util.stream.Stream`.

How_to_fix:

Group SQL statements by using the method `addBatch` to add them to a batch and then execute them using `executeBatch` to send them to the database in a single call.

Noncompliant code example

```
public void execute(Connection connection) {
  try {
    Statement statement = connection.createStatement();

    for (int i = 0; i < 10; i++) {
      statement.execute("INSERT INTO myTable (column1, column2) VALUES (" + i + ", 'value" + i + "')"); // Noncompliant
    }

    statement.close();
    connection.close();
  } catch (SQLException e) {
    e.printStackTrace();
  }
}
```

Compliant solution

```
public void execute(Connection connection) {
  try {
    Statement statement = connection.createStatement();

    for (int i = 0; i < 10; i++) {
      statement.addBatch("INSERT INTO myTable (column1, column2) VALUES (" + i + ", 'value" + i + "')"); // Compliant
    }
    statement.executeBatch();

    statement.close();
    connection.close();
  } catch (SQLException e) {
    e.printStackTrace();
  }
}
```

"Math.clamp" should be used with correct ranges (java:S6913)

Severidad: MAJOR

Root_cause:

Java 21 introduces the new method `Math.clamp(value, min, max)` that fits a value within a specified interval. Before Java 21, this behavior required explicit calls to the `Math.min` and `Math.max` methods, as in `Math.min(max, Math.max(value, min))`.

If `min > max`, `Math.clamp` throws an `IllegalArgumentException`, indicating an invalid interval. This can occur if the `min` and `max` arguments are mistakenly reversed.

Note that `Math.clamp` is not a general substitute for `Math.min` or `Math.max`, but for the combination of both. If `value` is the same as `min` or `max`, using `Math.clamp` is unnecessary and `Math.min` or `Math.max` should be used instead.

Resources:

- Java Documentation - [Math.clamp]

How_to_fix:

- If 2nd argument > 3rd argument, use `Math.clamp(value, min, max)` instead of `Math.clamp(value, max, min)`.
- If `value` is the same as `min`, fix the logic or use `Math.min(value, max)` instead.
- If `value` is the same as `max`, fix the logic or use `Math.max(min, value)` instead.
- If `min` is the same as `max`, fix the logic because `Math.clamp(value, x, x)` will always return `x`.

Noncompliant code example

```
Math.clamp(red, 255, 0); // Noncompliant, [255,0] is not a valid range
```

Compliant solution

```
Math.clamp(red, 0, 255); // Compliant
```

Noncompliant code example

```
Math.clamp(red, red, 255); // Noncompliant, use Math.min(red, 255)
```

Compliant solution

```
Math.min(red, 255); // Compliant
```

Noncompliant code example

```
Math.clamp(red, 0, red); // Noncompliant, use Math.max(red, 0)
```

Compliant solution

```
Math.max(red, 0); // Compliant
```

Use Fused Location to optimize battery power (java:S6914)

Severidad: MAJOR

Resources:

Documentation

- Google Play Services - FusedLocationProviderClient
- Android Developers - Optimize location for battery
- Android Developers - Android Location

Root_cause:

The location awareness feature can significantly drain the device's battery.

The recommended way to maximize the battery life is to use the *fused location provider* which combines signals from GPS, Wi-Fi, and cell networks, as well as accelerometer, gyroscope, magnetometer and other sensors. The `FusedLocationProviderClient` automatically chooses the best method to retrieve a device's location based on the device's context.

The rule flags an issue when `android.location.LocationManager` or `com.google.android.gms.location.LocationClient` is used instead of `com.google.android.gms.location.FusedLocationProviderClient`.

What is the potential impact?

- *Usability*: the non-optimized location API consumer more battery.
- *Sustainability*: the extra energy required has a negative impact on the environment.

How_to_fix:

Replace the usages of `android.location.LocationManager` or `com.google.android.gms.location.LocationClient` with `com.google.android.gms.location.FusedLocationProviderClient`.

Noncompliant code example

```
public class LocationsActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        // ...

        LocationManager locationManager = (LocationManager) this.getSystemService(Context.LOCATION_SERVICE); // Noncompliant

        LocationListener locationListener = new LocationListener() {
            public void onLocationChanged(Location location) {
                // Use the location object as needed
            }
        };

        locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 0, 0, locationListener);
    }
}
```

Compliant solution

```
public class LocationsActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        // ...

        FusedLocationProviderClient fusedLocationClient = LocationServices.getFusedLocationProviderClient(this); // Compliant

        fusedLocationClient.getLastLocation()
            .addOnSuccessListener(this, location -> {
                // Use the location object as needed
            });
    }
}
```

Introduction:

Location awareness is a common feature for mobile application that enhance the user experience by providing context-specific services.

"String.indexOf" should be used with correct ranges (java:S6915)

Severidad: MAJOR

How_to_fix:

- Use `String.indexOf(what, beginIndex, endIndex)` instead of `String.indexOf(what, endIndex, beginIndex)`.
- Use `String.indexOf(what, 0, endIndex)` instead of `String.indexOf(what, -1, endIndex)`.

Noncompliant code example

```
String hello = "Hello, world!";
int index = hello.indexOf('o', 11, 7); // Noncompliant, 11..7 is not a valid range
```

Compliant solution

```
String hello = "Hello, world!";
int index = hello.indexOf('o', 7, 11); // Compliant
```

Noncompliant code example

```
String hello = "Hello, world!";
int index = hello.indexOf('o', -1, 11); // Noncompliant, because beginIndex is negative
```

Compliant solution

```
String hello = "Hello, world!";
int index = hello.indexOf('o', 0, 11); // Compliant
```

Root_cause:

Java 21 adds new `String.indexOf` methods that accept ranges (`beginIndex`, to `endIndex`) rather than just a start index. A `StringIndexOutOfBounds` can be thrown when indicating an invalid range, namely when:

- `beginIndex > endIndex` (eg: `beginIndex` and `endIndex` arguments are mistakenly reversed)
- `beginIndex < 0` (eg: because the older `String.indexOf(what, fromIndex)` accepts negative values)

Resources:

- Java Documentation - String.indexOf(int, int, int)
- Java Documentation - String.indexOf(java.lang.String,int,int)

Use when instead of a single if inside a pattern match body (java:S6916)

Severidad: MAJOR

How_to_fix:

Replace the `if` statement with a guarded case label.

Noncompliant code example

```
void testObject(Object response) {
    switch (response) {
        case String s -> {
            if(s.length() > 80) { // Noncompliant; use the "when" keyword
                System.out.println("This string is too long");
            }
        }
        case Integer i -> {
            if(i > 80) { // Noncompliant; use the "when" keyword
                System.out.println("This integer is too big");
            }
        }
        default -> System.out.println("Unknown type");
    }
}
```

Compliant solution

```
void testObject(Object response) {
    switch (response) {
        case String s when s.length() > 80 -> {
            System.out.println("This string is too long");
        }
        case Integer i when i > 80 -> {
            System.out.println("This integer is too big");
        }
        default -> System.out.println("Unknown type");
    }
}
```

Resources:

Documentation

- Java Documentation - Pattern Matching for switch

Root_cause:

Java 21 has introduced enhancements to switch statements and expressions, allowing them to operate on any type, not just specific ones, as in previous versions. Furthermore, case labels have been upgraded to support patterns, providing an alternative to the previous restriction of only accepting constants.

```
// As of Java 21
String patternMatchSwitch(Object obj) {
    return switch (obj) {
        case String s  -> String.format("String %s", s);
        case Integer i -> String.format("int %d", i);
        default        -> obj.toString();
    };
}
```

This allows to use the `when` keyword to specify a condition for a case label, also called a guarded case label.

```
String guardedCaseSwitch(Object obj) {
    return switch (obj) {
        case String s when s.length() > 0 -> String.format("String %s", s);
        case Integer i when i > 0 -> String.format("int %d", i);
        default        -> obj.toString();
    };
}
```

This syntax is more readable and less error-prone than using an if statement inside the case block and should be preferred.

This rule reports an issue when a single `if` statement is used inside a case block.

An open curly brace should be located at the end of a line (java:S1105)

Severidad: MINOR

Root_cause:

Shared naming conventions allow teams to collaborate effectively. This rule raises an issue when an open curly brace is not placed at the end of a line of code.

Noncompliant code example

```
if(condition)
{
  doSomething();
}
```

Compliant solution

```
if(condition) {
  doSomething();
}
```

Exceptions

When blocks are inlined (left and right curly braces on the same line), no issue is triggered.

```
if(condition) {doSomething();}
```

Method parameters, caught exceptions and foreach variables' initial values should not be ignored (java:S1226)

Severidad: MINOR

Root_cause:

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<String> strings) {
  str = Integer.toString(i); // Noncompliant

  for (String s : strings) {
    s = "hello world"; // Noncompliant
  }
}
```

Track uses of "NOPMD" suppression comments (java:S1310)

Severidad: MINOR

Root_cause:

This rule allows you to track the use of the PMD suppression comment mechanism.

Noncompliant code example

```
// NOPMD
```

Loggers should be "private static final" and should share a naming convention (java:S1312)

Severidad: MINOR

Root_cause:

Regardless of the logging framework in use (logback, log4j, commons-logging, java.util.logging, …), loggers should be:

- `private`: never be accessible outside of its parent class. If another class needs to log something, it should instantiate its own logger.
- `static`: not be dependent on an instance of a class (an object). When logging something, contextual information can of course be provided in the messages but the logger should be created at class level to prevent creating a logger along with each object.
- `final`: be created once and only once per class.

Noncompliant code example

With a default regular expression of `LOG(?:GER)?`:

```
public Logger logger = LoggerFactory.getLogger(Foo.class);  // Noncompliant
```

Compliant solution

```
private static final Logger LOGGER = LoggerFactory.getLogger(Foo.class);
```

Exceptions

Variables of type `org.apache.maven.plugin.logging.Log` are ignored.

Using hardcoded IP addresses is security-sensitive (java:S1313)

Severidad: MINOR

Root_cause:

Hardcoding IP addresses is security-sensitive. It has led in the past to the following vulnerabilities:

- CVE-2006-5901
- CVE-2005-3725

Today’s services have an ever-changing architecture due to their scaling and redundancy needs. It is a mistake to think that a service will always have the same IP address. When it does change, the hardcoded IP will have to be modified too. This will have an impact on the product development, delivery, and deployment:

- The developers will have to do a rapid fix every time this happens, instead of having an operation team change a configuration file.
- It misleads to use the same address in every environment (dev, sys, qa, prod).

Last but not least it has an effect on application security. Attackers might be able to decompile the code and thereby discover a potentially sensitive address. They can perform a Denial of Service attack on the service, try to get access to the system, or try to spoof the IP address to bypass security checks. Such attacks can always be possible, but in the case of a hardcoded IP address solving the issue will take more time, which will increase an attack’s impact.

Exceptions

No issue is reported for the following cases because they are not considered sensitive:

- Loopback addresses 127.0.0.0/8 in CIDR notation (from 127.0.0.0 to 127.255.255.255)
- Broadcast address 255.255.255.255
- Non-routable address 0.0.0.0
- Strings of the form `2.5.<number>.<number>` as they often match Object Identifiers (OID)
- Addresses in the ranges 192.0.2.0/24, 198.51.100.0/24, 203.0.113.0/24, reserved for documentation purposes by RFC 5737
- Addresses in the range 2001:db8::/32, reserved for documentation purposes by RFC 3849

Default:

Hardcoding IP addresses is security-sensitive. It has led in the past to the following vulnerabilities:

- CVE-2006-5901
- CVE-2005-3725

Today’s services have an ever-changing architecture due to their scaling and redundancy needs. It is a mistake to think that a service will always have the same IP address. When it does change, the hardcoded IP will have to be modified too. This will have an impact on the product development, delivery, and deployment:

- The developers will have to do a rapid fix every time this happens, instead of having an operation team change a configuration file.
- It misleads to use the same address in every environment (dev, sys, qa, prod).

Last but not least it has an effect on application security. Attackers might be able to decompile the code and thereby discover a potentially sensitive address. They can perform a Denial of Service attack on the service, try to get access to the system, or try to spoof the IP address to bypass security checks. Such attacks can always be possible, but in the case of a hardcoded IP address solving the issue will take more time, which will increase an attack’s impact.

Ask Yourself Whether

The disclosed IP address is sensitive, e.g.:

- Can give information to an attacker about the network topology.
- It’s a personal (assigned to an identifiable person) IP address.

There is a risk if you answered yes to any of these questions.

Recommended Secure Coding Practices

Don’t hard-code the IP address in the source code, instead make it configurable with environment variables, configuration files, or a similar approach. Alternatively, if confidentially is not required a domain name can be used since it allows to change the destination quickly without having to rebuild the software.

Sensitive Code Example

```
String ip = "192.168.12.42"; // Sensitive
Socket socket = new Socket(ip, 6667);
```

Compliant Solution

```
String ip = System.getenv("IP_ADDRESS"); // Compliant
Socket socket = new Socket(ip, 6667);
```

Exceptions

No issue is reported for the following cases because they are not considered sensitive:

- Loopback addresses 127.0.0.0/8 in CIDR notation (from 127.0.0.0 to 127.255.255.255)
- Broadcast address 255.255.255.255
- Non-routable address 0.0.0.0
- Strings of the form `2.5.<number>.<number>` as they often match Object Identifiers (OID)
- Addresses in the ranges 192.0.2.0/24, 198.51.100.0/24, 203.0.113.0/24, reserved for documentation purposes by RFC 5737
- Addresses in the range 2001:db8::/32, reserved for documentation purposes by RFC 3849

See

- OWASP - Top 10 2021 Category A1 - Broken Access Control
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- CERT, MSC03-J. - Never hard code sensitive information

How_to_fix:

Recommended Secure Coding Practices

Don’t hard-code the IP address in the source code, instead make it configurable with environment variables, configuration files, or a similar approach. Alternatively, if confidentially is not required a domain name can be used since it allows to change the destination quickly without having to rebuild the software.

Compliant Solution

```
String ip = System.getenv("IP_ADDRESS"); // Compliant
Socket socket = new Socket(ip, 6667);
```

See

- OWASP - Top 10 2021 Category A1 - Broken Access Control
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- CERT, MSC03-J. - Never hard code sensitive information

Assess_the_problem:

Ask Yourself Whether

The disclosed IP address is sensitive, e.g.:

- Can give information to an attacker about the network topology.
- It’s a personal (assigned to an identifiable person) IP address.

There is a risk if you answered yes to any of these questions.

Sensitive Code Example

```
String ip = "192.168.12.42"; // Sensitive
Socket socket = new Socket(ip, 6667);
```

Octal values should not be used (java:S1314)

Severidad: BLOCKER

Resources:

- CERT, DCL18-C. - Do not begin integer constants with 0 when specifying a decimal value
- CERT, DCL50-J. - Use visually distinct identifiers

Root_cause:

Integer literals starting with a zero are octal rather than decimal values. While using octal values is fully supported, most developers do not have experience with them. They may not recognize octal values as such, mistaking them instead for decimal values.

Noncompliant code example

```
int myNumber = 010; // Noncompliant. myNumber will hold 8, not 10 - was this really expected?
```

Compliant solution

```
int myNumber = 8;
```

Track uses of "CHECKSTYLE:OFF" suppression comments (java:S1315)

Severidad: MINOR

Root_cause:

This rule allows you to track the use of the Checkstyle suppression comment mechanism.

Noncompliant code example

```
// CHECKSTYLE:OFF
```

"StringBuilder" and "StringBuffer" should not be instantiated with a character (java:S1317)

Severidad: MAJOR

How_to_fix:

If the argument is a `char` literal, use a string literal instead:

```
StringBuffer foo = new StringBuffer('x'); // Noncompliant, replace with String
```

```
StringBuffer foo = new StringBuffer("x"); // Compliant
```

If the argument is it is a non-literal `char` expression, convert it to `String` using the `String.valueOf()` method:

```
StringBuffer foo(char firstChar) {
  return new StringBuffer(firstChar);                 // Noncompliant
}
```

```
StringBuffer foo(char firstChar) {
  return new StringBuffer(String.valueOf(firstChar)); // Compliant
}
```

Root_cause:

When a developer uses the `StringBuilder` or `StringBuffer` constructor with a single character as an argument, the likely intention is to create an instance with the character as the initial string value.

However, this is not what happens because of the absence of a dedicated `StringBuilder(char)` or `StringBuffer(char)` constructor. Instead, `StringBuilder(int)` or `StringBuffer(int)` is invoked, which results in an instance with the provided `int` value as the initial capacity of the `StringBuilder` or `StringBuffer`.

The reason behind this behavior lies in the automatic widening of `char` expressions to `int` when required. Consequently, the UTF-16 code point value of the character (for example, `65` for the character `'A'`) is interpreted as an `int` to specify the initial capacity.

Introduction:

This rule raises an issue when the `StringBuilder` or `StringBuffer` constructor is called with a single character as an argument.

Resources:

Documentation

- Oracle - Java Language Specification, section 5.1.2. Widening Primitive Conversion

Articles & blog posts

- W3schools - Java Type Casting

Declarations should use Java collection interfaces such as "List" rather than specific implementation classes such as "LinkedList" (java:S1319)

Severidad: MINOR

Introduction:

This rule raises an issue when a collection implementation class from `java.util.*` is used:

- as a return type of a `public` method.
- as an argument type of a `public` method.
- as the type of a `public` field.

Root_cause:

The Java Collections API offers a well-structured hierarchy of interfaces designed to hide collection implementation details. For the various collection data structures like lists, sets, and maps, specific interfaces (`java.util.List`, `java.util.Set`, `java.util.Map`) cover the essential features.

When passing collections as method parameters, return values, or when exposing fields, it is generally recommended to use these interfaces instead of the implementing classes. The implementing classes, such as `java.util.LinkedList`, `java.util.ArrayList`, and `java.util.HasMap`, should only be used for collection instantiation. They provide finer control over the performance characteristics of those structures, and developers choose them depending on their use case.

For example, if fast random element access is essential, `java.util.ArrayList` should be instantiated. If inserting elements at a random position into a list is crucial, a `java.util.LinkedList` should be preferred. However, this is an implementation detail your API should not expose.

Noncompliant code example

```
public class Employees {
  public final HashSet<Employee> employees   // Noncompliant, field type should be "Set"
    = new HashSet<Employee>();

  public HashSet<Employee> getEmployees() {  // Noncompliant, return type should be "Set"
    return employees;
  }
}
```

Compliant solution

```
public class Employees {
  public final Set<Employee> employees       // Compliant
    = new HashSet<Employee>();

  public Set<Employee> getEmployees() {      // Compliant
```

```
    return employees;
  }
}
```

Unary prefix operators should not be repeated (java:S2761)

Severidad: MAJOR

Root_cause:

The repetition of a unary operator is usually a typo. The second operator invalidates the first one in most cases:

```
int i = 1;

int j = - - -i;  // Noncompliant: equivalent to "-i"
int k = ~~~i;    // Noncompliant: equivalent to "~i"
int m = + +i;    // Noncompliant: equivalent to "i"

boolean b = false;
boolean c = !!!b;   // Noncompliant
```

On the other hand, while repeating the increment and decrement operators is technically correct, it obfuscates the meaning:

```
int i = 1;
int j = ++ ++i;  // Noncompliant
int k = i-- --; // Noncompliant
```

Using `+=` or `-=` improves readability:

```
int i = 1;
i += 2;
int j = i;
int k = i;
i -=2;
```

This rule raises an issue for repetitions of `!`, `~`, `-`, `+`, prefix increments `++` and prefix decrements `--`.

Exceptions

Overflow handling for GWT compilation using `~~` is ignored.

Non-thread-safe fields should not be static (java:S2885)

Severidad: MAJOR

How_to_fix:

Remove the `static` keyword from non-thread-safe fields.

Noncompliant code example

```
public class MyClass {
  private static Calendar calendar = Calendar.getInstance();  // Noncompliant
  private static SimpleDateFormat format = new SimpleDateFormat("HH-mm-ss");  // Noncompliant
}
```

Compliant solution

```
public class MyClass {
  private Calendar calendar = Calendar.getInstance();
  private SimpleDateFormat format = new SimpleDateFormat("HH-mm-ss");
}
```

Root_cause:

When an object is marked as `static`, it means that it belongs to the class rather than any class instance. This means there is only one copy of the static object in memory, regardless of how many class instances are created. Static objects are shared among all instances of the class and can be accessed using the class name rather than an instance of the class.

A data type is considered thread-safe if it can be used correctly by multiple threads, regardless of how those threads are executed, without requiring additional coordination from the calling code. In other words, a thread-safe data type can be accessed and modified by multiple threads simultaneously without causing any issues or requiring extra work from the programmer to ensure correct behavior.

Non-thread-safe objects are objects that are not designed to be used in a multi-threaded environment and can lead to race conditions and data inconsistencies when accessed by multiple threads simultaneously. Using them in a multi-threaded manner is highly likely to cause data problems or exceptions at runtime.

When a non-thread-safe object is marked as static in a multi-threaded environment, it can cause issues because the non-thread-safe object will be shared across different instances of the containing class.

This rule raises an issue when any of the following instances and their subtypes are marked as `static`:

- `java.util.Calendar`,
- `java.text.DateFormat`,
- `javax.xml.xpath.XPath`, or
- `javax.xml.validation.SchemaFactory`.

Resources:

Articles & blog posts

- MIT - Thread safety
- Baeldung - Thread safety
- Baeldung - Static

Standards

- STIG Viewer - Application Security and Development: V-222567 - The application must not be vulnerable to race conditions.

Getters and setters should be synchronized in pairs (java:S2886)

Severidad: MAJOR

Root_cause:

A synchronized method is a method marked with the `synchronized` keyword, meaning it can only be accessed by one thread at a time. If multiple threads try to access the synchronized method simultaneously, they will be blocked until the method is available.

Synchronized methods prevent race conditions and data inconsistencies in multi-threaded environments. Ensuring that only one thread can access a method at a time, prevents multiple threads from modifying the same data simultaneously, and causing conflicts.

When one part of a getter/setter pair is `synchronized` the other should be too. Failure to synchronize both sides may result in inconsistent behavior at runtime as callers access an inconsistent method state.

This rule raises an issue when either the method or the contents of one method in a getter/setter pair are synchronized, but the other is not.

Resources:

Documentation

- Oracle Java - Synchronized Methods
- Oracle SE 20 - Synchronized Methods

Articles & blog posts

- MIT - Thread safety
- Baeldung - Thread safety

Standards

- CERT, VNA01-J. - Ensure visibility of shared references to immutable objects
- STIG Viewer - Application Security and Development: V-222567 - The application must not be vulnerable to race conditions.

How_to_fix:

Synchronize both `get` and `set` methods by marking the method with the `synchronize` keyword or using a `synchronize` block inside them.

Noncompliant code example

```
public class Person {
  String name;
  int age;

  public synchronized void setName(String name) {
    this.name = name;
  }

  public String getName() {  // Noncompliant
    return this.name;
  }

  public void setAge(int age) {  // Noncompliant
    this.age = age;
  }

  public int getAge() {
    synchronized (this) {
      return this.age;
    }
  }
}
```

Compliant solution

```
public class Person {
  String name;
  int age;

  public synchronized void setName(String name) {
    this.name = name;
  }

  public synchronized String getName() {
    return this.name;
  }

  public void setAge(int age) {
    synchronized (this) {
      this.age = age;
   }
  }

  public int getAge() {
    synchronized (this) {
      return this.age;
    }
  }
}
```

"Stream.peek" should be used with caution (java:S3864)

Severidad: MAJOR

Root_cause:

According to its JavaDocs, the intermediate Stream operation `java.util.Stream.peek()` “exists mainly to support debugging” purposes.

A key difference with other intermediate Stream operations is that the Stream implementation is free to skip calls to `peek()` for optimization purpose. This can lead to `peek()` being unexpectedly called only for some or none of the elements in the Stream.

As a consequence, relying on `peek()` without careful consideration can lead to error-prone code.

This rule raises an issue for each use of peek() to be sure that it is challenged and validated by the team to be meant for production debugging/logging purposes.

Noncompliant code example

```
Stream.of("one", "two", "three", "four")
         .filter(e -> e.length() > 3)
         .peek(e -> System.out.println("Filtered value: " + e)); // Noncompliant
```

Compliant solution

```
Stream.of("one", "two", "three", "four")
         .filter(e -> e.length() > 3)
         .foreach(e -> System.out.println("Filtered value: " + e));
```

Resources:

- Java 8 API Documentation
- 4comprehension: Idiomatic Peeking with Java Stream API
- Data Geekery: 10 Subtle Mistakes When Using the Streams API

Conditionals should start on new lines (java:S3972)

Severidad: CRITICAL

Root_cause:

Placing an `if` statement on the same line as the closing `}` from a preceding `if`, `else`, or `else if` block can lead to confusion and potential errors. It may indicate a missing `else` statement or create ambiguity for maintainers who might fail to understand that the two statements are unconnected.

The following code snippet is confusing:

```
if (condition1) {
  // ...
} if (condition2) {  // Noncompliant
  //...
}
```

Either the two conditions are unrelated and they should be visually separated:

```
if (condition1) {
  // ...
}

if (condition2) {
  //...
}
```

Or they were supposed to be exclusive and you should use `else if` instead:

```
if (condition1) {
  // ...
} else if (condition2) {
  //...
}
```

A conditionally executed single line should be denoted by indentation (java:S3973)

Severidad: CRITICAL

Root_cause:

When the line immediately after a conditional has neither curly braces nor indentation, the intent of the code is unclear and perhaps not what is executed. Additionally, such code is confusing to maintainers.

```
if (condition)  // Noncompliant
doTheThing();
doTheOtherThing(); // Was the intent to call this function unconditionally?
```

It becomes even more confusing and bug-prone if lines get commented out.

```
if (condition)  // Noncompliant
//  doTheThing();
doTheOtherThing(); // Was the intent to call this function conditionally?
```

Indentation alone or together with curly braces makes the intent clear.

```
if (condition)
  doTheThing();
doTheOtherThing(); // Clear intent to call this function unconditionally

// or

if (condition) {
  doTheThing();
}
doTheOtherThing(); // Clear intent to call this function unconditionally
```

This rule raises an issue if the line controlled by a conditional has the same indentation as the conditional and is not enclosed in curly braces.

Unused "private" classes should be removed (java:S3985)

Severidad: MAJOR

Root_cause:

`private` classes that are never used 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.

Noncompliant code example

```
public class TopLevel
{
  private class Nested {...} // Noncompliant: Nested is never used
}
```

Compliant solution

```
public class TopLevel
{
  void doSomething() {
    Nested a = new Nested();
    ...
  }
  private class Nested {...}
}
```

Introduction:

This rule raises an issue when a private nested class is never used.

Methods setUp() and tearDown() should be correctly annotated starting with JUnit4 (java:S5826)

Severidad: CRITICAL

Root_cause:

The `setUp()` and `tearDown()` methods (initially introduced with JUnit3 to execute a block of code before and after each test) need to be correctly annotated with the equivalent annotation in order to preserve the same behavior when migrating from JUnit3 to JUnit4 or JUnit5.

This rule consequently raise issues on `setUp()` and `tearDown()` methods which are not annotated in test classes.

Noncompliant code example

- JUnit4:

```
public void setUp() { ... } // Noncompliant; should be annotated with @Before
public void tearDown() { ... }  // Noncompliant; should be annotated with @After
```

- JUnit5:

```
public void setUp() { ... } // Noncompliant; should be annotated with @BeforeEach
public void tearDown() { ... }  // Noncompliant; should be annotated with @AfterEach
```

Compliant solution

- JUnit4:

```
@Before
public void setUp() { ... }

@After
public void tearDown() { ... }
```

- JUnit5:

```
@BeforeEach
void setUp() { ... }

@AfterEach
void tearDown() { ... }
```

DateTimeFormatters should not use mismatched year and week numbers (java:S5917)

Severidad: MAJOR

Root_cause:

When creating a `DateTimeFormatter` using the `WeekFields.weekBasedYear()` temporal field, the resulting year number may be off by 1 at the beginning of a new year (when the date to format is in a week that is shared by two consecutive years).

Using this year number in combination with an incompatible week temporal field yields a result that may be confused with the first week of the previous year.

Instead, when paired with a week temporal field, the week-based year should only be used with the week of week-based year temporal field `WeekFields.weekOfWeekBasedYear()`.

Alternatively the temporal field `ChronoField.ALIGNED_WEEK_OF_YEAR` can be used together with a regular year (but not the week based year).

Noncompliant code example

```
new DateTimeFormatterBuilder()
      .appendValue(ChronoField.YEAR, 4) // Noncompliant: using week of week-based year with regular year
      .appendLiteral('-')
      .appendValue(WeekFields.ISO.weekOfWeekBasedYear(), 2)
      .toFormatter();

new DateTimeFormatterBuilder()
      .appendValue(ChronoField.YEAR_OF_ERA, 4) // Noncompliant: using week of week-based year with regular year
      .appendLiteral('-')
      .appendValue(WeekFields.ISO.weekOfWeekBasedYear(), 2)
      .toFormatter();

new DateTimeFormatterBuilder()
      .appendValue(WeekFields.ISO.weekBasedYear(), 4) // Noncompliant: using aligned week of year with week-based year
      .appendLiteral('-')
      .appendValue(ChronoField.ALIGNED_WEEK_OF_YEAR, 2)
      .toFormatter();
```

Here the first two formatters would wrongly format the 1st of January 2016 as "2016-53" while the last one would format it as "2015-01"

Compliant solution

```
new DateTimeFormatterBuilder()
      .appendValue(WeekFields.ISO.weekBasedYear(), 4)
      .appendLiteral('-')
      .appendValue(WeekFields.ISO.weekOfWeekBasedYear(), 2)
      .toFormatter();

new DateTimeFormatterBuilder()
      .appendValue(ChronoField.YEAR, 4)
      .appendLiteral('-')
      .appendValue(ChronoField.ALIGNED_WEEK_OF_YEAR, 2)
      .toFormatter();

new DateTimeFormatterBuilder()
      .appendValue(ChronoField.YEAR_OF_ERA, 4)
      .appendLiteral('-')
      .appendValue(ChronoField.ALIGNED_WEEK_OF_YEAR, 2)
      .toFormatter();
```

Here the first formatter would format the 1st of January 2016 as "2015-53" while the last two would produce "2016-01", both of which are correct depending on how you count the weeks.

Exceptions

No issue is raised when week-based year is not used in combination with a week temporal field.

Similarly, no issue is raised if week of week-based year is not used in combination with a year temporal field.

Classes should not be coupled to too many other classes (java:S1200)

Severidad: MAJOR

Root_cause:

According to the Single Responsibility Principle, introduced by Robert C. Martin in his book "Principles of Object Oriented Design", a class should have only one responsibility:

> If a class has more than one responsibility, then the responsibilities become coupled.
>
> Changes to one responsibility may impair or inhibit the class' ability to meet the others.
>
> This kind of coupling leads to fragile designs that break in unexpected ways when changed.

Classes which rely on many other classes tend to aggregate too many responsibilities and should be split into several smaller ones.

Nested classes dependencies are not counted as dependencies of the outer class.

Noncompliant code example

With a threshold of 5:

```
class Foo {                        // Noncompliant - Foo depends on too many classes: T1, T2, T3, T4, T5, T6 and T7
  T1 a1;                           // Foo is coupled to T1
  T2 a2;                           // Foo is coupled to T2
  T3 a3;                           // Foo is coupled to T3

  public T4 compute(T5 a, T6 b) {  // Foo is coupled to T4, T5 and T6
    T7 result = a.getResult(b);    // Foo is coupled to T7
    return result;
  }

  public static class Bar {        // Compliant - Bar depends on 2 classes: T8 and T9
    T8 a8;
    T9 a9;
  }
}
```

"equals" method overrides should accept "Object" parameters (java:S1201)

Severidad: MAJOR

Root_cause:

In Java, the `Object.equals()` method is used for object comparison, and it is typically overridden in classes to provide a custom equality check based on your criteria for equality.

The default implementation of `equals()` provided by the `Object` class compares the memory references of the two objects, that means it checks if the objects are actually the same instance in memory.

The "equals" as a method name should be used exclusively to override `Object.equals(Object)` to prevent confusion.

It is important to note that when you override `equals()`, you should also override the `hashCode()` method to maintain the contract between `equals()` and `hashCode()`.

Resources:

Documentation

- Oracle SDK - Object.equals(Object)

How_to_fix:

Either override `Object.equals(Object)` or rename the method.

Noncompliant code example

```
class MyClass {
  private int foo = 1;

  public boolean equals(MyClass o) {  // Noncompliant; does not override Object.equals(Object)
    return o != null && o.foo == this.foo;
  }

  public static void main(String[] args) {
    MyClass o1 = new MyClass();
    Object o2 = new MyClass();
    System.out.println(o1.equals(o2));  // Prints "false" because o2 an Object not a MyClass
  }
}
```

Compliant solution

```
class MyClass {
  private int foo = 1;

  @Override
  public boolean equals(Object o) {  // Compliant
    if (this == o) {
        return true;
    }

    if (o == null || getClass() != o.getClass()) {
      return false;
    }
    MyClass other = (MyClass)o;
    return this.foo == other.foo;
  }
}
```

"equals(Object obj)" and "hashCode()" should be overridden in pairs (java:S1206)

Severidad: MINOR

How_to_fix:

Noncompliant code example

```
class MyClass {    // Noncompliant - should also override "hashCode()"

  @Override
  public boolean equals(Object obj) {
    /* ... */
  }

}
```

Compliant solution

```
class MyClass {    // Compliant

  @Override
  public boolean equals(Object obj) {
    /* ... */
  }

  @Override
  public int hashCode() {
    /* ... */
  }

}
```

Resources:

- CWE - CWE-581 - Object Model Violation: Just One of Equals and Hashcode Defined
- CERT, MET09-J. - Classes that define an equals() method must also define a hashCode() method

Root_cause:

According to the Java Language Specification, there is a contract between `equals(Object)` and `hashCode()`:

> If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
>
> It is not required that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results.
>
> However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.

In order to comply with this contract, those methods should be either both inherited, or both overridden.

"public static" fields should be constant (java:S1444)

Severidad: MINOR

Root_cause:

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();
  ...
}
```

Resources:

- CWE - CWE-500 - Public Static Field Not Marked Final
- CERT OBJ10-J. - Do not use public static nonfinal fields

Classes should not have too many methods (java:S1448)

Severidad: MAJOR

Root_cause:

A class that grows too much tends to aggregate too many responsibilities and inevitably becomes harder to understand and therefore to maintain. Above a specific threshold, it is strongly advised to refactor the class into smaller ones which focus on well defined topics.

String operations should not rely on the default system locale (java:S1449)

Severidad: MINOR

Root_cause:

Failure to specify a locale when calling the methods `toLowerCase()`, `toUpperCase()` or `format()` on `String` objects means the system default encoding will be used, possibly creating problems with international characters or number representations. For instance with the Turkish language, when converting the small letter 'i' to upper case, the result is capital letter 'I' with a dot over it.

Case conversion without a locale may work fine in its "home" environment, but break in ways that are extremely difficult to diagnose for customers who use different encodings. Such bugs can be nearly, if not completely, impossible to reproduce when it’s time to fix them. For locale-sensitive strings, the correct locale should always be used, but `Locale.ROOT` can be used for case-insensitive ones.

Noncompliant code example

```
myString.toLowerCase()
```

Compliant solution

```
myString.toLowerCase(Locale.TR)
```

Resources:

- CERT, STR02-J. - Specify an appropriate locale when comparing locale-dependent data

---

Jump statements should not be redundant (java:S3626)

Severidad: MINOR

Root_cause:

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();
    }
  }
}
```

Raw types should not be used (java:S3740)

Severidad: MAJOR

How_to_fix:

For any usage of parameterized types, bind the type parameters with type arguments. For example, when a function returns a list of strings, the return type is `List<String>`, where the type parameter `E` in interface `List<E>` is bound with the argument `String`.

If the concrete binding is unknown, you still should not use the type raw. Use a wildcard type argument instead, with optional lower or upper bound, such as in `List<?>` for a list whose element type is unknown, or `List<? extends Number>` for a list whose element type is `Number` or a subtype of it.

Noncompliant code example

```
// List is supposed to store integers only
List integers = new ArrayList<>();

// Yet, we can add strings, because we did not give
// this information to the compiler
integers.add("Hello World!");

// Type is checked during runtime and will throw a ClassCastException
Integer a = (Integer) integers.get(0);
```

Compliant solution

```
// List is supposed to store integers, and we let the compiler know
List<Integer> integers = new ArrayList<>();

// Now we can add only integers.
// Adding a string results in a compile time error.
integers.add(42);

// No cast required anymore, and no possible ClassCastException
Integer a = integers.get(0);
```

Noncompliant code example

```
String getStringFromForcedList(Object object) {
  // Cast expression and instanceof can check runtime type only.
  // The solution is _not_ to skip the type argument in that case.
  return object instanceof List stringList ? (String) stringList.getFirst(): "";
}
```

Compliant solution

```
String getStringFromForcedList(Object object) {
  // The solution is to use a wildcard type argument in that case.
  return object instanceof List<?> stringList ? (String) stringList.getFirst(): "";
}
```

Noncompliant code example

```
String getStringFromForcedList(Object object) {
  return object instanceof List stringList ? (String) stringList.getFirst(): "";
}

String returnString() {
  Object object = List.of("Hello");
  return getStringFromForcedList(object);
}
```

Compliant solution

```
Object getObjectFromForcedList(Object object) {
  // You may also choose not to make assumptions about type arguments you cannot infer.
  return object instanceof List<?> list ? list.getFirst(): "";
}
```

```
String returnString(Object object) {
  // Instead, delegate the decision to use-site, which may have more information.
  Object object = List.of("Hello");
  return (String) getObjectFromForcedList(object);
}
```

Introduction:

Generic types should not be used raw (without type arguments). To fix this issue, add the type parameters.

Root_cause:

A generic type is a generic class or interface that is parameterized over types. For example, `java.util.List` has one type parameter: the type of its elements.

Using generic types raw (without binding arguments to the type parameters) prevents compile-time type checking for expressions that use these type parameters. Explicit type casts are necessary for them, which do perform a runtime type check that may fail with a `ClassCastException`.

What is the potential impact?

The compiler cannot assert that the program is inherently type safe. When a cast fails, a `ClassCastException` is thrown during runtime and the program most likely crashes. Therefore, this issue might impact the availability and reliability of your application.

Exceptions

The rule does not raise an issue for the simple `instanceof` operator, which checks against runtime types where type parameter information has been erased. Since it does not return a rawly typed instance but a boolean value, it does not prevent compile-time type checking.

This, however, is not the case for the `cast` operator as well as the extended `instanceof` operator which are both not an exception from this rule. Since they operate on the erased runtime type as well, they must use wildcard type arguments when checked against a parameterized type (see the examples).

Resources:

Documentation

- Raw types in the Java Tutorial.

Members of Spring components should be injected (java:S3749)

Severidad: CRITICAL

How_to_fix:

Add one of these annotations to all non-`static` members: `@Resource`, `@Inject`, `@Autowired` or `@Value`.

Noncompliant code example

```
@Controller
public class HelloWorld {

  private String name = null;

  @RequestMapping("/greet", method = GET)
  public String greet(String greetee) {

    if (greetee != null) {
      this.name = greetee;
    }

    return "Hello " + this.name;  // if greetee is null, you see the previous user's data
  }
}
```

Root_cause:

Spring `@Component`, `@Controller`, `@RestController`,`@Service`, and `@Repository` classes are singletons by default, meaning only one instance of the class is ever instantiated in the application. Typically such a class might have a few `static` members, such as a logger, but all non-`static` members should be managed by Spring.

This rule raises an issue when a singleton `@Component`, `@Controller`, `@RestController`, `@Service`, or `@Repository`, not annotated with `@ConfigurationProperties`, has non-`static` members that are not annotated with one of:

- `org.springframework.beans.factory.annotation.Autowired`
- `org.springframework.beans.factory.annotation.Value`
- `javax.annotation.Inject`
- `javax.annotation.Resource`
- `javax.persistence.PersistenceContext`
- `jakarta.annotation.Resource`
- `jakarta.inject.Inject`
- `jakarta.persistence.PersistenceContext`

Collection sizes and array length comparisons should make sense (java:S3981)

Severidad: MAJOR

Root_cause:

The size of a collection and the length of an array are always greater than or equal to zero. Testing it doesn’t make sense, since the result is always `true`.

```
if (myList.size() >= 0) {...} // Noncompliant: always true
```

```
boolean result = myArray.length >= 0; // Noncompliant: true
```

Similarly testing that it is less than zero will always return `false`.

```
if (myList.size() < 0) {...} // Noncompliant: always false
```

Fix the code to properly check for emptiness if it was the intent, or remove the redundant code to keep the current behavior.

Exceptions should not be created without being thrown (java:S3984)

Severidad: MAJOR

Root_cause:

Creating a new `Throwable` without actually throwing it is useless and is probably due to a mistake.

Noncompliant code example

```
if (x < 0)
  new IllegalArgumentException("x must be nonnegative");
```

Compliant solution

```
if (x < 0)
  throw new IllegalArgumentException("x must be nonnegative");
```

Week Year ("YYYY") should not be used for date formatting (java:S3986)

Severidad: MAJOR

Root_cause:

Few developers are aware of the difference between `Y` for "Week year" and `y` for Year when formatting and parsing a date with `SimpleDateFormat` or `DateTimeFormatter`. That’s likely because for most dates, Week year and Year are the same, so testing at any time other than the first or last week of the year will yield the same value for both `y` and `Y`. But in the last week of December and the first week of January, you may get unexpected results.

According to the Javadoc:

> A week year is in sync with a WEEK_OF_YEAR cycle. All weeks between the first and last weeks (inclusive) have the same week year value. Therefore, the first and last days of a week year may have different calendar year values.
>
> For example, January 1, 1998 is a Thursday. If getFirstDayOfWeek() is MONDAY and getMinimalDaysInFirstWeek() is 4 (ISO 8601 standard compatible setting), then week 1 of 1998 starts on December 29, 1997, and ends on January 4, 1998. The week year is 1998 for the last three days of calendar year 1997. If, however, getFirstDayOfWeek() is SUNDAY, then week 1 of 1998 starts on January 4, 1998, and ends on January 10, 1998; the first three days of 1998 then are part of week 53 of 1997 and their week year is 1997.

Noncompliant code example

```
Date date = new SimpleDateFormat("yyyy/MM/dd").parse("2015/12/31");
String result = new SimpleDateFormat("YYYY/MM/dd").format(date);   //Noncompliant; yields '2016/12/31'
result = DateTimeFormatter.ofPattern("YYYY/MM/dd").format(date); //Noncompliant; yields '2016/12/31'
```

Compliant solution

```
Date date = new SimpleDateFormat("yyyy/MM/dd").parse("2015/12/31");
String result = new SimpleDateFormat("yyyy/MM/dd").format(date);   //Yields '2015/12/31' as expected
result = DateTimeFormatter.ofPattern("yyyy/MM/dd").format(date); //Yields '2015/12/31' as expected
```

Exceptions

```
Date date = new SimpleDateFormat("yyyy/MM/dd").parse("2015/12/31");
String result = new SimpleDateFormat("YYYY-ww").format(date);  //compliant, 'Week year' is used along with 'Week of year'. result = '2016-0
DateTimeFormatter.ofPattern("YYYY-ww").format(date); //compliant; yields '2016-01' as expected
```

"StandardCharsets" constants should be preferred (java:S4719)

Severidad: MINOR

Root_cause:

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)
```

Server certificates should be verified during SSL/TLS connections (java:S4830)

Severidad: CRITICAL

Introduction:

This vulnerability makes it possible that an encrypted communication is intercepted.

Root_cause:

Transport Layer Security (TLS) provides secure communication between systems over the internet by encrypting the data sent between them. Certificate validation adds an extra layer of trust and security to this process to ensure that a system is indeed the one it claims to be.

When certificate validation is disabled, the client skips a critical security check. This creates an opportunity for attackers to pose as a trusted entity and intercept, manipulate, or steal the data being transmitted.

What is the potential impact?

Establishing trust in a secure way is a non-trivial task. When you disable certificate validation, you are removing a key mechanism designed to build this trust in internet communication, opening your system up to a number of potential threats.

Identity spoofing

If a system does not validate certificates, it cannot confirm the identity of the other party involved in the communication. An attacker can exploit this by creating a fake server and masquerading as a legitimate one. For example, they might set up a server that looks like your bank's server, tricking your system into thinking it is communicating with the bank. This scenario, called identity spoofing, allows the attacker to collect any data your system sends to them, potentially leading to significant data breaches.

Loss of data integrity

When TLS certificate validation is disabled, the integrity of the data you send and receive cannot be guaranteed. An attacker could modify the data in transit, and you would have no way of knowing. This could range from subtle manipulations of the data you receive to the injection of malicious code or malware into your system. The consequences of such breaches of data integrity can be severe, depending on the nature of the data and the system.

How_to_fix:

The following code contains examples of disabled certificate validation.

The certificate validation gets disabled by overriding `X509TrustManager` with an empty implementation. It is highly recommended to use the original implementation.

Noncompliant code example

```
class TrustAllManager implements X509TrustManager {

    @Override
    public void checkClientTrusted(X509Certificate[] chain, String authType) throws CertificateException {  // Noncompliant
    }

    @Override
    public void checkServerTrusted(X509Certificate[] chain, String authType) throws CertificateException { // Noncompliant
    }

    @Override
    public X509Certificate[] getAcceptedIssuers() {
        return null;
    }
}
```

How does this work?

Addressing the vulnerability of disabled TLS certificate validation primarily involves re-enabling the default validation.

To avoid running into problems with invalid certificates, consider the following sections.

Using trusted certificates

If possible, always use a certificate issued by a well-known, trusted CA for your server. Most programming environments come with a predefined list of trusted root CAs, and certificates issued by these authorities are validated automatically. This is the best practice, and it requires no additional code or configuration.

Working with self-signed certificates or non-standard CAs

In some cases, you might need to work with a server using a self-signed certificate, or a certificate issued by a CA not included in your trusted roots. Rather than disabling certificate validation in your code, you can add the necessary certificates to your trust store.

Here is a sample command to import a certificate to the Java trust store:

```
keytool -import -alias myserver -file myserver.crt -keystore cacerts
```

Resources:

Standards

- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2021 Category A7 - Identification and Authentication Failures
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- OWASP - Mobile Top 10 2016 Category M3 - Insecure Communication
- OWASP - Mobile AppSec Verification Standard - Network Communication Requirements

- CWE - CWE-295 - Improper Certificate Validation
- STIG Viewer - Application Security and Development: V-222550 - The application must validate certificates by constructing a certification path to an accepted trust anchor.
- https://wiki.sei.cmu.edu/confluence/display/java/MSC61-J.+Do+not+use+insecure+or+weak+cryptographic+algorithms

An iteration on a Collection should be performed on the type handled by the Collection (java:S4838)

Severidad: MINOR

How_to_fix:

Noncompliant code example

When declaring the iteration variable, use the item type for it instead of a supertype. Remove the explicit downcasts in the loop body.

```
for (Object item : getPersons()) { // Noncompliant, iteration element is implicitly upcast here
  Person person = (Person) item; // Noncompliant, item is explicitly downcast here
  person.getAddress();
}
```

Compliant solution

```
for (Person person : getPersons()) { // Compliant
  person.getAddress();
}
```

Noncompliant code example

Alternatively, use the `var` keyword to automatically infer the variable type (since Java 10).

```
for (Object item : getPersons()) { // Noncompliant, iteration element is implicitly upcast here
  Person person = (Person) item; // Noncompliant, item is explicitly downcast here
  person.getAddress();
}
```

Compliant solution

```
for (var person : getPersons()) { // Compliant
  person.getAddress();
}
```

Compliant solution

The implicit upcast in the loop header is not reported when there is no downcast in the loop body.

```
for (Object item : getPersons()) { // Compliant
  System.out.println(item);
}
```

Root_cause:

When iterating over an `Iterable` with a `for` loop, the iteration variable could have the same type as the type returned by the iterator (the item type of the `Iterable`). This rule reports when a supertype of the item type is used for the variable instead, but the variable is then explicitly downcast in the loop body.

Using explicit type casts instead of leveraging the language’s type system is a bad practice. It disables static type checking by the compiler for the cast expressions, but potential errors will throw a `ClassCastException` during runtime instead.

Class members annotated with "@VisibleForTesting" should not be accessed from production code (java:S5803)

Severidad: CRITICAL

Root_cause:

@VisibleForTesting can be used to mark methods, fields and classes whose visibility restrictions have been relaxed more than necessary for the API to allow for easier unit testing.

Access to such methods, fields and classes only possible thanks to this relaxed visibility is fine for test code, but it should be avoided in production code. In production code these methods should be treated as if they are private.

Supported framework:

- Guava: `com.google.common.annotations.VisibleForTesting`
- AssertJ: `org.assertj.core.util.VisibleForTesting`
- Android: `androidx.annotation.VisibleForTesting`
- Apache Flink: `org.apache.flink.annotation.VisibleForTesting`

or any other annotation named `VisibleForTesting`

Noncompliant code example

```
/** src/main/java/MyObject.java */

@VisibleForTesting String foo;

/** src/main/java/Service.java */

new MyObject().foo; // Noncompliant, foo is accessed from production code
```

Compliant solution

```
/** src/main/java/MyObject.java */

@VisibleForTesting String foo;

/** src/test/java/MyObjectTest.java */

new MyObject().foo; // Compliant, foo is accessed from test code
```

Authorizations should be based on strong decisions (java:S5808)

Severidad: MAJOR

Resources:

Standards

- OWASP - [Top 10 2021 Category A1 - Broken Access Control]
- OWASP - [Top 10 2017 Category A5 - Broken Access Control]
- CWE - [CWE-285 - Improper Authorization]

Root_cause:

Access control is a critical aspect of web frameworks that ensures proper authorization and restricts access to sensitive resources or actions. To enable access control, web frameworks offer components that are responsible for evaluating user permissions and making access control decisions. They might examine the user’s credentials, such as roles or privileges, and compare them against predefined rules or policies to determine whether the user should be granted access to a specific resource or action.

Conventionally, these checks should never grant access to every request received. If an endpoint or component is meant to be public, then it should be ignored by access control components. Conversely, if an endpoint should deny some users from accessing it, then access control has to be configured correctly for this endpoint.

Granting unrestricted access to all users can lead to security vulnerabilities and potential misuse of critical functionalities. It is important to carefully assess access decisions based on factors such as user roles, resource sensitivity, and business requirements. Implementing a robust and granular access control mechanism is crucial for the security and integrity of the web application itself and its surrounding environment.

What is the potential impact?

Not verifying user access strictly can introduce significant security risks. Some of the most prominent risks are listed below. Depending on the use case, it is very likely that other risks are introduced on top of the ones listed.

Unauthorized access

As the access of users is not checked strictly, it becomes very easy for an attacker to gain access to restricted areas or functionalities, potentially compromising the confidentiality, integrity, and availability of sensitive resources. They may exploit this access to perform malicious actions, such as modifying or deleting data, impersonating legitimate users, or gaining administrative privileges, ultimately compromising the security of the system.

Theft of sensitive data

Theft of sensitive data can result from incorrect access control if attackers manage to gain access to databases, file systems, or other storage mechanisms where sensitive data is stored. This can lead to the theft of personally identifiable information (PII), financial data, intellectual property, or other confidential information. The stolen data can be used for various malicious purposes, such as identity theft, financial fraud, or selling the data on the black market, causing significant harm to individuals and organizations affected by the breach.

How_to_fix:

Noncompliant code example

The `vote` method of an AccessDecisionVoter implementation is not compliant when it returns only an affirmative decision (`ACCESS_GRANTED`) or abstains to make a decision (`ACCESS_ABSTAIN`):

```
public class WeakNightVoter implements AccessDecisionVoter {
    @Override
    public int vote(Authentication authentication, Object object, Collection collection) {
        Calendar calendar = Calendar.getInstance();
        int currentHour = calendar.get(Calendar.HOUR_OF_DAY);

        if (currentHour >= 8 && currentHour <= 19) {
            return ACCESS_GRANTED;
        }

        return ACCESS_ABSTAIN; // Noncompliant: when users connect during the night, no decision is made
    }
}
```

The `hasPermission` method of a PermissionEvaluator implementation is not compliant when it doesn't return `false`:

```
public class MyPermissionEvaluator implements PermissionEvaluator {
    @Override
    public boolean hasPermission(Authentication authentication, Object targetDomainObject, Object permission) {
        Object user = authentication.getPrincipal();

        if (user.getRole().equals(permission)) {
              return true;
        }

        return true; // Noncompliant
    }
}
```

Compliant solution

The `vote` method of an AccessDecisionVoter implementation should return a negative decision (`ACCESS_DENIED`):

```
public class StrongNightVoter implements AccessDecisionVoter {
    @Override
    public int vote(Authentication authentication, Object object, Collection collection) {
        Calendar calendar = Calendar.getInstance();
        int currentHour = calendar.get(Calendar.HOUR_OF_DAY);

        if (currentHour >= 8 && currentHour <= 19) {
            return ACCESS_GRANTED;
        }

        return ACCESS_DENIED; // Users are not allowed to connect during the night
    }
}
```

The `hasPermission` method of a PermissionEvaluator implementation should return `false`:

```
public class MyPermissionEvaluator implements PermissionEvaluator {
    @Override
    public boolean hasPermission(Authentication authentication, Object targetDomainObject, Object permission) {
        Object user = authentication.getPrincipal();

        if (user.getRole().equals(permission)) {
              return true;
        }

        return false;
    }
}
```

Introduction:

When granting users access to resources of an application, such an authorization should be based on strong decisions. For instance, a user may be authorized to access a resource only if they are authenticated, or if they have the correct role and privileges.

A field should not duplicate the name of its containing class (java:S1700)

Severidad: MAJOR

Root_cause:

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;
  }
  ...
}
```

"switch" statements and expressions should not be nested (java:S1821)

Severidad: CRITICAL

Root_cause:

Nested `switch` structures are difficult to understand because you can easily confuse the cases of an inner `switch` as belonging to an outer statement or expression. Therefore nested `switch` statements and expressions should be avoided.

Specifically, you should structure your code to avoid the need for nested `switch` statements or expressions, but if you cannot, then consider moving the inner `switch` to another method.

Noncompliant code example

```
void foo(int n, int m) {
  switch (n) {
    case 0:
      switch (m) {  // Noncompliant; nested switch
        // ...
      }
    case 1:
      // ...
    default:
      // ...
  }
}
```

Compliant solution

```
void foo(int n, int m) {
  switch (n) {
    case 0:
      bar(m);
    case 1:
      // ...
    default:
      // ...
  }
}

void bar(int m){
  switch(m) {
    // ...
  }
}
```

Boolean checks should not be inverted (java:S1940)

Severidad: MINOR

Root_cause:

It is needlessly complex to invert the result of a boolean comparison. The opposite comparison should be made instead.

Noncompliant code example

```
if ( !(a == 2)) { ...}  // Noncompliant
boolean b = !(i < 10);  // Noncompliant
```

Compliant solution

```
if (a != 2) { ...}
boolean b = (i >= 10);
```

Variables should not be declared before they are relevant (java:S1941)

Severidad: MINOR

Root_cause:

For the sake of clarity, variables should be declared as close to where they're used as possible. This is particularly true when considering methods that contain early returns and the potential to throw exceptions. In these cases, it is not only pointless, but also confusing to declare a variable that may never be used because conditions for an early return are met first.

Noncompliant code example

```
public boolean isConditionMet(int a, int b) {
  int difference = a - b;
  MyClass foo = new MyClass(a);  // Noncompliant; not used before early return

  if (difference < 0) {
    return false;
  }

  // ...

  if (foo.doTheThing()) {
    return true;
  }
  return false;
}
```

Compliant solution

```
public boolean isConditionMet(int a, int b) {
  int difference = a - b;

  if (difference < 0) {
    return false;
  }

  // ...

  MyClass foo = new MyClass(a);
  if (foo.doTheThing()) {
    return true;
  }
  return false;
}
```

Simple class names should be used (java:S1942)

Severidad: MINOR

Root_cause:

Java’s `import` mechanism allows the use of simple class names. Therefore, using a class' fully qualified name in a file that `imports` the class is redundant and confusing.

Noncompliant code example

```
import java.util.List;
import java.sql.Timestamp;

//...

java.util.List<String> myList;  // Noncompliant
java.sql.Timestamp tStamp; // Noncompliant
```

Compliant solution

```
import java.util.List;
import java.sql.Timestamp;

//...

List<String> myList;
Timestamp tStamp;
```

Classes and methods that rely on the default system encoding should not be used (java:S1943)

Severidad: MINOR

Resources:

- CERT, STR04-J. - Use compatible character encodings when communicating string data between JVMs
- CERT, STR50-J. - Use the appropriate method for counting characters in a string

Root_cause:

Using classes and methods that rely on the default system encoding can result in code that works fine in its "home" environment. But that code may break for customers who use different encodings in ways that are extremely difficult to diagnose and nearly, if not completely, impossible to reproduce when it’s time to fix them.

This rule detects uses of the following classes and methods:

- `FileReader`
- `FileWriter`
- String constructors with a `byte[]` argument but no `Charset` argument
  - `String(byte[] bytes)`
  - `String(byte[] bytes, int offset, int length)`
- `String.getBytes()`
- `String.getBytes(int srcBegin, int srcEnd, byte[] dst, int dstBegin)`
- `InputStreamReader(InputStream in)`
- `OutputStreamWriter(OutputStream out)`
- `ByteArrayOutputStream.toString()`
- Some `Formatter` constructors
  - `Formatter(String fileName)`
  - `Formatter(File file)`
  - `Formatter(OutputStream os)`
- Some `Scanner` constructors
  - `Scanner(File source)`
  - `Scanner(Path source)`
  - `Scanner(InputStream source)`
- Some `PrintStream` constructors
  - `PrintStream(File file)`
  - `PrintStream(OutputStream out)`

    - `PrintStream(OutputStream out, boolean autoFlush)`
    - `PrintStream(String fileName)`
- Some `PrintWriter` constructors
    - `PrintWriter(File file)`
    - `PrintWriter(OutputStream out)`
    - `PrintWriter(OutputStream out, boolean autoFlush)`
    - `PrintWriter(String fileName)`
- methods from Apache commons-io library which accept an encoding argument when that argument is null, and overloads of those methods that omit the encoding argument
    - `IOUtils.copy(InputStream, Writer)`
    - `IOUtils.copy(Reader, OutputStream)`
    - `IOUtils.readLines(InputStream)`
    - `IOUtils.toByteArray(Reader)`
    - `IOUtils.toByteArray(String)`
    - `IOUtils.toCharArray(InputStream)`
    - `IOUtils.toInputStream(TypeCriteria.subtypeOf(CharSequence))`
    - `IOUtils.toString(byte[])`
    - `IOUtils.toString(URI)`
    - `IOUtils.toString(URL)`
    - `IOUtils.write(char[], OutputStream)`
    - `IOUtils.write(CharSequence, OutputStream)`
    - `IOUtils.writeLines(Collection, String, OutputStream)`
    - `FileUtils.readFileToString(File)`
    - `FileUtils.readLines(File)`
    - `FileUtils.write(File, CharSequence)`
    - `FileUtils.write(File, CharSequence, boolean)`
    - `FileUtils.writeStringToFile(File, String)`

Fields in a "Serializable" class should either be transient or serializable (java:S1948)

Severidad: CRITICAL

Resources:

- CWE - CWE-594 - Saving Unserializable Objects to Disk
- Interface Serializable - Java SE 11 API Documentation
- Interface Serializable - Java SE 17 API Documentation

How_to_fix:

Consider the following scenario.

```
public class Address {
    ...
}

public class Person implements Serializable {
  private static final long serialVersionUID = 1905122041950251207L;

  private String name;
  private Address address;  // Noncompliant, Address is not serializable
}
```

How to fix this issue depends on the application’s needs. If the field’s value should be preserved during serialization and deserialization, you may want to make the field’s value serializable.

```
public class Address implements Serializable {
  private static final long serialVersionUID = 2405172041950251807L;

    ...
}

public class Person implements Serializable {
  private static final long serialVersionUID = 1905122041950251207L;

  private String name;
  private Address address; // Compliant, Address is serializable
}
```

If the field’s value does not need to be preserved during serialization and deserialization, mark it as `transient`. The field will be ignored when the object is serialized. After deserialization, the field will be set to the default value corresponding to its type (e.g., `null` for object references).

```
public class Address {
    ...
}

public class Person implements Serializable {
  private static final long serialVersionUID = 1905122041950251207L;
  private String name;
  private transient Address address; // Compliant, the field is transient
}
```

The alternative to making all members serializable or `transient` is to implement special methods which take on the responsibility of properly serializing and de-serializing the object `writeObject` and `readObject`. These methods can be used to properly (de-)serialize an object, even though it contains fields that are not transient or serializable. Hence, this rule does not raise issues on fields of classes which implement these methods.

```
public class Address {
    ...
}

public class Person implements Serializable {
  private static final long serialVersionUID = 1905122041950251207L;

  private String name;
  private Address address; // Compliant, writeObject and readObject handle this field

  private void writeObject(java.io.ObjectOutputStream out) throws IOException {
    // Appropriate serialization logic here
  }

  private void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException {
    // Appropriate deserialization logic here
  }
}
```

Finally, static fields are out of scope for serialization, so making a field static prevents issues from being raised.

```
public class Person implements Serializable {
  private static final long serialVersionUID = 1905122041950251207L;

  private String name;

  private static Logger log = getLogger(); // Compliant, static fields are not serialized
}
```

Root_cause:

By contract, non-static fields in a `Serializable` class must themselves be either `Serializable` or `transient`. Even if the class is never explicitly serialized or deserialized, it is not safe to assume that this cannot happen. For instance, under load, most J2EE application frameworks flush objects to disk.

An object that implements `Serializable` but contains non-transient, non-serializable data members (and thus violates the contract) could cause application crashes and open the door to attackers. In general, a `Serializable` class is expected to fulfil its contract and not exhibit unexpected behaviour when an instance is serialized.

This rule raises an issue on:

- Non-`Serializable` fields.
- When a field is assigned a non-`Serializable` type within the class.
- Collection fields when they are not `private`. Values that are not serializable could be added to these collections externally. Due to type erasure, it cannot be guaranteed that the collection will only contain serializable objects at runtime despite being declared as a collection of serializable types.

Introduction:

This rule raises an issue on a non-transient and non-serializable field within a serializable class, if said class does not have `writeObject` and `readObject` methods defined.

"close()" calls should not be redundant (java:S4087)

Severidad: MINOR

Root_cause:

Java 7’s try-with-resources structure automatically handles closing the resources that the `try` itself opens. Thus, adding an explicit `close()` call is redundant and potentially confusing.

Noncompliant code example

```
try (PrintWriter writer = new PrintWriter(process.getOutputStream())) {
  String contents = file.contents();
  writer.write(new Gson().toJson(new MyObject(contents)));
  writer.flush();
  writer.close();  // Noncompliant
}
```

Compliant solution

```
try (PrintWriter writer = new PrintWriter(process.getOutputStream())) {
  String contents = file.contents();
  writer.write(new Gson().toJson(new MyObject(contents)));
  writer.flush();
}
```

Allowing user enumeration is security-sensitive (java:S5804)

Severidad: MAJOR

Assess_the_problem:

Ask Yourself Whether

- The application discloses that a username exists in its database: most of the time it’s possible to avoid this kind of leak except for the "registration/sign-on" part of a website because in this case the user must choose a valid username (not already taken by another user).
- There is no rate limiting and CAPTCHA protection in place for requests involving a username.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

In a Spring-security web application the username leaks when:

- The string used as argument of loadUserByUsername method is used in an exception message:

```
public String authenticate(String username, String password) {
  // ....
  MyUserDetailsService s1 = new MyUserDetailsService();
  MyUserPrincipal u1 = s1.loadUserByUsername(username);

  if(u1 == null) {
    throw new BadCredentialsException(username+" doesn't exist in our database"); // Sensitive
  }
  // ....
}
```

- UsernameNotFoundException is thrown (except when it is in the loadUserByUsername method):

```
public String authenticate(String username, String password) {
  // ....
  if(user == null) {
      throw new UsernameNotFoundException("user not found"); // Sensitive
  }
  // ....
}
```

- HideUserNotFoundExceptions is set to false:

```
DaoAuthenticationProvider daoauth = new DaoAuthenticationProvider();
daoauth.setUserDetailsService(new MyUserDetailsService());
daoauth.setPasswordEncoder(new BCryptPasswordEncoder());
daoauth.setHideUserNotFoundExceptions(false); // Sensitive
builder.authenticationProvider(daoauth);
```

Default:

User enumeration refers to the ability to guess existing usernames in a web application database. This can happen, for example, when using "sign-in/sign-on/forgot password" functionalities of a website.

When an user tries to "sign-in" to a website with an incorrect username/login, the web application should not disclose that the username doesn’t exist with a message similar to "this username is incorrect", instead a generic message should be used like "bad credentials", this way it’s not possible to guess whether the username or password was incorrect during the authentication.

If a user-management feature discloses information about the existence of a username, attackers can use brute force attacks to retrieve a large amount of valid usernames that will impact the privacy of corresponding users and facilitate other attacks (phishing, password guessing etc …).

Ask Yourself Whether

- The application discloses that a username exists in its database: most of the time it’s possible to avoid this kind of leak except for the "registration/sign-on" part of a website because in this case the user must choose a valid username (not already taken by another

user).
- There is no rate limiting and CAPTCHA protection in place for requests involving a username.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

When a user performs a request involving a username, it should not be possible to spot differences between a valid and incorrect username:

- Error messages should be generic and not disclose if the username is valid or not.
- The response time must be similar for a valid username or not.
- CAPTCHA and other rate limiting solutions should be implemented.

Sensitive Code Example

In a Spring-security web application the username leaks when:

- The string used as argument of loadUserByUsername method is used in an exception message:

```
public String authenticate(String username, String password) {
  // ....
  MyUserDetailsService s1 = new MyUserDetailsService();
  MyUserPrincipal u1 = s1.loadUserByUsername(username);

  if(u1 == null) {
    throw new BadCredentialsException(username+" doesn't exist in our database"); // Sensitive
  }
  // ....
}
```

- UsernameNotFoundException is thrown (except when it is in the loadUserByUsername method):

```
public String authenticate(String username, String password) {
  // ....
  if(user == null) {
      throw new UsernameNotFoundException("user not found"); // Sensitive
  }
  // ....
}
```

- HideUserNotFoundExceptions is set to false:

```
DaoAuthenticationProvider daoauth = new DaoAuthenticationProvider();
daoauth.setUserDetailsService(new MyUserDetailsService());
daoauth.setPasswordEncoder(new BCryptPasswordEncoder());
daoauth.setHideUserNotFoundExceptions(false); // Sensitive
builder.authenticationProvider(daoauth);
```

Compliant Solution

In a Spring-security web application:

- the same message should be used regardless of whether it is the wrong user or password:

```
public String authenticate(String username, String password) throws AuthenticationException {
  Details user = null;
  try {
    user = loadUserByUsername(username);
  } catch (UsernameNotFoundException | DataAccessException e) {
    // Hide this exception reason to not disclose that the username doesn't exist
  }
  if (user == null || !user.isPasswordCorrect(password)) {
     // User should not be able to guess if the bad credentials message is related to the username or the password
    throw new BadCredentialsException("Bad credentials");
  }
}
```

- HideUserNotFoundExceptions should be set to true:

```
DaoAuthenticationProvider daoauth = new DaoAuthenticationProvider();
daoauth.setUserDetailsService(new MyUserDetailsService());
daoauth.setPasswordEncoder(new BCryptPasswordEncoder());
daoauth.setHideUserNotFoundExceptions(true); // Compliant
builder.authenticationProvider(daoauth);
```

See

- OWASP - Top 10 2021 Category A1 - Broken Access Control
- OWASP - Top 10 2017 Category A2 - Broken Authentication

- CWE - CWE-200 - Exposure of Sensitive Information to an Unauthorized Actor

How_to_fix:

Recommended Secure Coding Practices

When a user performs a request involving a username, it should not be possible to spot differences between a valid and incorrect username:

- Error messages should be generic and not disclose if the username is valid or not.
- The response time must be similar for a valid username or not.
- CAPTCHA and other rate limiting solutions should be implemented.

Compliant Solution

In a Spring-security web application:

- the same message should be used regardless of whether it is the wrong user or password:

```
public String authenticate(String username, String password) throws AuthenticationException {
  Details user = null;
  try {
    user = loadUserByUsername(username);
  } catch (UsernameNotFoundException | DataAccessException e) {
    // Hide this exception reason to not disclose that the username doesn't exist
  }
  if (user == null || !user.isPasswordCorrect(password)) {
     // User should not be able to guess if the bad credentials message is related to the username or the password
    throw new BadCredentialsException("Bad credentials");
  }
}
```

- HideUserNotFoundExceptions should be set to true:

```
DaoAuthenticationProvider daoauth = new DaoAuthenticationProvider();
daoauth.setUserDetailsService(new MyUserDetailsService());
daoauth.setPasswordEncoder(new BCryptPasswordEncoder());
daoauth.setHideUserNotFoundExceptions(true); // Compliant
builder.authenticationProvider(daoauth);
```

See

- OWASP - Top 10 2021 Category A1 - Broken Access Control
- OWASP - Top 10 2017 Category A2 - Broken Authentication
- CWE - CWE-200 - Exposure of Sensitive Information to an Unauthorized Actor

Root_cause:

User enumeration refers to the ability to guess existing usernames in a web application database. This can happen, for example, when using "sign-in/sign-on/forgot password" functionalities of a website.

When an user tries to "sign-in" to a website with an incorrect username/login, the web application should not disclose that the username doesn’t exist with a message similar to "this username is incorrect", instead a generic message should be used like "bad credentials", this way it’s not possible to guess whether the username or password was incorrect during the authentication.

If a user-management feature discloses information about the existence of a username, attackers can use brute force attacks to retrieve a large amount of valid usernames that will impact the privacy of corresponding users and facilitate other attacks (phishing, password guessing etc …).

AWS region should not be set with a hardcoded String (java:S6262)

Severidad: MINOR

Root_cause:

When explicitly setting the region on an AWS Client, you should always prefer providing the value from the Enum Regions instead of a hardcoded String. This will allow you to transparently support any change in the API and avoid mistakes.

This rule reports an issue when a hardcoded string is used instead of an available enum value.

Noncompliant code example

```
AmazonS3ClientBuilder.standard().withRegion("eu_west_1").build();
```

Compliant solution

```
AmazonS3ClientBuilder.standard().withRegion(Regions.EU_WEST_1).build();
```

Using long-term access keys is security-sensitive (java:S6263)

Severidad: MAJOR

Default:

In AWS, long-term access keys will be valid until you manually revoke them. This makes them highly sensitive as any exposure can have serious consequences and should be used with care.

This rule will trigger when encountering an instantiation of `com.amazonaws.auth.BasicAWSCredentials`.

Ask Yourself Whether

- The access key is used directly in an application or AWS CLI script running on an Amazon EC2 instance.
- Cross-account access is needed.
- The access keys need to be embedded within a mobile application.
- Existing identity providers (SAML 2.0, on-premises identity store) already exists.

For more information, see Use IAM roles instead of long-term access keys.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

Consider using IAM roles or other features of the AWS Security Token Service that provide temporary credentials, limiting the risks.

Sensitive Code Example

```
import com.amazonaws.auth.AWSCredentials;
import com.amazonaws.auth.BasicAWSCredentials;
// ...

AWSCredentials awsCredentials = new BasicAWSCredentials(accessKeyId, secretAccessKey);
```

Compliant Solution

Example for AWS STS (see Getting Temporary Credentials with AWS STS).

```
BasicSessionCredentials sessionCredentials = new BasicSessionCredentials(
   session_creds.getAccessKeyId(),
   session_creds.getSecretAccessKey(),
   session_creds.getSessionToken());
```

See

- Best practices for managing AWS access keys
- Managing access keys for IAM users

Assess_the_problem:

Ask Yourself Whether

- The access key is used directly in an application or AWS CLI script running on an Amazon EC2 instance.
- Cross-account access is needed.
- The access keys need to be embedded within a mobile application.
- Existing identity providers (SAML 2.0, on-premises identity store) already exists.

For more information, see Use IAM roles instead of long-term access keys.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

```
import com.amazonaws.auth.AWSCredentials;
import com.amazonaws.auth.BasicAWSCredentials;
// ...

AWSCredentials awsCredentials = new BasicAWSCredentials(accessKeyId, secretAccessKey);
```

How_to_fix:

Recommended Secure Coding Practices

Consider using IAM roles or other features of the AWS Security Token Service that provide temporary credentials, limiting the risks.

Compliant Solution

Example for AWS STS (see Getting Temporary Credentials with AWS STS).

```
BasicSessionCredentials sessionCredentials = new BasicSessionCredentials(
   session_creds.getAccessKeyId(),
   session_creds.getSecretAccessKey(),
   session_creds.getSessionToken());
```

See

- Best practices for managing AWS access keys
- Managing access keys for IAM users

Root_cause:

In AWS, long-term access keys will be valid until you manually revoke them. This makes them highly sensitive as any exposure can have serious consequences and should be used with care.

This rule will trigger when encountering an instantiation of `com.amazonaws.auth.BasicAWSCredentials`.

Classes should not have too many fields (java:S1820)

Severidad: MAJOR

Root_cause:

A class that grows too much tends to aggregate too many responsibilities and inevitably becomes harder to understand and therefore to maintain, and having a lot of fields is an indication that a class has grown too large.

Above a specific threshold, it is strongly advised to refactor the class into smaller ones which focus on well defined topics.

Redundant casts should not be used (java:S1905)

Severidad: MINOR

How_to_fix:

To fix your code remove the unnecessary casting expression.

Noncompliant code example

```
class Example {
    public void example(List<String> list) {
        for (String item: (List<String>) list) {  // Noncompliant, Remove this unnecessary cast to "List".
          //...
        }
    }
}
```

Compliant solution

```
class Example {
    public void example() {
        for (String foo : getFoos()) {
          //...
        }
    }
    public List<String> getFoos() {
        return List.of("foo1", "foo2");
    }
}
```

Resources:

Documentation

- Geeksforgeeks - Type conversion in Java with Examples
- Wikipedia - Type Conversion
- Wikipedia - Strong and Weak Typing
- Wikipedia - Polymorphism (Computer Science)

Root_cause:

Casting expressions are utilized to convert one data type to another, such as transforming an integer into a string. This is especially crucial in strongly typed languages like C, C++, C#, Java, Python, and others.

However, there are instances where casting expressions are not needed. These include situations like:

- casting a variable to its own type
- casting a subclass to a parent class (in the case of polymorphism)
- the programming language is capable of automatically converting the given type to another

These scenarios are considered unnecessary casting expressions. They can complicate the code and make it more difficult to understand, without offering any advantages.

As a result, it’s generally advised to avoid unnecessary casting expressions. Instead, rely on the language’s type system to ensure type safety and code clarity.

Exceptions

Casting may be required to distinguish the method to call in the case of overloading:

```
class A {}
class B extends A{}
class C {
  void fun(A a){}
  void fun(B b){}

  void foo() {
    B b = new B();
    fun(b);
    fun((A) b); // Compliant, required to call the first method so cast is not redundant.
  }
}
```

Extensions and implementations should not be redundant (java:S1939)

Severidad: MINOR

Root_cause:

All classes extend `Object` implicitly. Doing so explicitly is redundant.

Further, declaring the implementation of an interface *and* one if its parents is also redundant. If you implement the interface, you also implicitly implement its parents and there’s no need to do so explicitly.

Noncompliant code example

```
public interface MyFace {
  // ...
}

public interface MyOtherFace extends MyFace {
  // ...
}

public class Foo
    extends Object // Noncompliant
    implements MyFace, MyOtherFace {  // Noncompliant
  //...
}
```

Compliant solution

```
public interface MyFace {
  // ...
}

public interface MyOtherFace extends MyFace {
  // ...
}

public class Foo implements MyOtherFace {
  //...
}
```

"ThreadLocal.withInitial" should be preferred (java:S4065)

Severidad: MINOR

Root_cause:

Java 8 introduced `ThreadLocal.withInitial` which is a simpler alternative to creating an anonymous inner class to initialise a `ThreadLocal` instance.

This rule raises an issue when a `ThreadLocal` anonymous inner class can be replaced by a call to `ThreadLocal.withInitial`.

Noncompliant code example

```
ThreadLocal<List<String>> myThreadLocal =
    new ThreadLocal<List<String>>() { // Noncompliant
        @Override
        protected List<String> initialValue() {
            return new ArrayList<String>();
        }
    };
```

Compliant solution

```
ThreadLocal<List<String>> myThreadLocal = ThreadLocal.withInitial(ArrayList::new);
```

Single-character alternations in regular expressions should be replaced with character classes (java:S6035)

Severidad: MAJOR

Root_cause:

When an alternation contains multiple alternatives that consist of a single character, it can be rewritten as a character class. This should be preferred because it is more efficient and can even help prevent stack overflows when used inside a repetition (see rule S5998).

Noncompliant code example

```
Pattern.compile("a|b|c"); // Noncompliant
```

Compliant solution

```
Pattern.compile("[abc]");
// or
Pattern.compile("[a-c]");
```

String multiline concatenation should be replaced with Text Blocks (java:S6126)

Severidad: MAJOR

Resources:

- JEP 378: Text Blocks
- Programmer’s Guide To Text Blocks, by Jim Laskey and Stuart Marks

Root_cause:

In Java 15 Text Blocks are now official and can be used. The most common pattern for multiline strings in Java < 15 was to write String concatenation. Now it’s possible to do it in a more natural way using Text Blocks.

Noncompliant code example

```
String textBlock =
               "<html>\n" +
               "    <body>\n" +
               "        <tag>\n" +
               "        </tag>\n" +
               "    </body>\n" +
               "</html>";
```

Compliant solution

```
String textBlock = """
        <html>
            <body>
                <tag>
                </tag>
            </body>
        </html>""";
```

Credentials Provider should be set explicitly when creating a new "AwsClient" (java:S6242)

Severidad: MAJOR

Resources:

- Tuning the AWS Java SDK 2.x to reduce startup time
- Optimizing cold start performance for AWS Lambda
- Environment variable configuration
- Default Credential Provider Chain

Root_cause:

If the credentials provider is not specified when creating a new AwsClient with an AwsClientBuilder, the AWS SDK will execute some logic to identify it automatically.

While it will probably identify the correct one, this extra logic will slow down startup time, already known to be a hotspot.

You should therefore always define the logic to set the credentials provider yourself. This is typically done by retrieving it from the Lambda provided environment variable.

This will make the code more explicit and spare initialization time.

This rule reports an issue when the credentials provider is not set when creating an AwsClient.

Noncompliant code example

```
S3Client.builder()
    .region(Region.of(System.getenv(SdkSystemSetting.AWS_REGION.environmentVariable())))
    .build();
```

Compliant solution

```
S3Client.builder()
    .region(Region.of(System.getenv(SdkSystemSetting.AWS_REGION.environmentVariable()))
    .credentialsProvider(EnvironmentVariableCredentialsProvider.create())
    .build();
```

Lambdas should not invoke other lambdas synchronously (java:S6246)

Severidad: MINOR

Root_cause:

Invoking other Lambdas synchronously from a Lambda is a scalability anti-pattern. Lambdas have a maximum execution time before they timeout (15 minutes as of May 2021). Having to wait for another Lambda to finish its execution could lead to a timeout.

A better solution is to generate events that can be consumed asynchronously by other Lambdas.

Noncompliant code example

With AWS SDKv1

```
InvokeRequest invokeRequest = new InvokeRequest()
         .withFunctionName("myFunction");
```

```
AWSLambda awsLambda = AWSLambdaClientBuilder.standard()
                .withCredentials(new ProfileCredentialsProvider())
                .withRegion(Regions.US_WEST_2).build();

awsLambda.invoke(invokeRequest); // Noncompliant
```

Resources:

- Best practices for working with AWS Lambda functions

Enabling JavaScript support for WebViews is security-sensitive (java:S6362)

Severidad: MAJOR

Assess_the_problem:

Ask Yourself Whether

- The WebWiew only renders static web content that does not require JavaScript code to be executed.
- The WebView contains untrusted data that could cause harm when rendered.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

```
import android.webkit.WebView;

WebView webView = (WebView) findViewById(R.id.webview);
webView.getSettings().setJavaScriptEnabled(true); // Sensitive
```

Default:

WebViews can be used to display web content as part of a mobile application. A browser engine is used to render and display the content. Like a web application, a mobile application that uses WebViews can be vulnerable to Cross-Site Scripting if untrusted code is rendered. In the context of a WebView, JavaScript code can exfiltrate local files that might be sensitive or even worse, access exposed functions of the application that can result in more severe vulnerabilities such as code injection. Thus JavaScript support should not be enabled for WebViews unless it is absolutely necessary and the authenticity of the web resources can be guaranteed.

Ask Yourself Whether

- The WebWiew only renders static web content that does not require JavaScript code to be executed.
- The WebView contains untrusted data that could cause harm when rendered.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

It is recommended to disable JavaScript support for WebViews unless it is necessary to execute JavaScript code. Only trusted pages should be rendered.

Sensitive Code Example

```
import android.webkit.WebView;

WebView webView = (WebView) findViewById(R.id.webview);
webView.getSettings().setJavaScriptEnabled(true); // Sensitive
```

Compliant Solution

```
import android.webkit.WebView;

WebView webView = (WebView) findViewById(R.id.webview);
webView.getSettings().setJavaScriptEnabled(false);
```

See

- OWASP - Top 10 2021 Category A3 - Injection
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- OWASP - Top 10 2017 Category A7 - Cross-Site Scripting (XSS)

- CWE - CWE-79 - Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

Root_cause:

WebViews can be used to display web content as part of a mobile application. A browser engine is used to render and display the content. Like a web application, a mobile application that uses WebViews can be vulnerable to Cross-Site Scripting if untrusted code is rendered. In the context of a WebView, JavaScript code can exfiltrate local files that might be sensitive or even worse, access exposed functions of the application that can result in more severe vulnerabilities such as code injection. Thus JavaScript support should not be enabled for WebViews unless it is absolutely necessary and the authenticity of the web resources can be guaranteed.

How_to_fix:

Recommended Secure Coding Practices

It is recommended to disable JavaScript support for WebViews unless it is necessary to execute JavaScript code. Only trusted pages should be rendered.

Compliant Solution

```
import android.webkit.WebView;

WebView webView = (WebView) findViewById(R.id.webview);
webView.getSettings().setJavaScriptEnabled(false);
```

See

- OWASP - Top 10 2021 Category A3 - Injection
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- OWASP - Top 10 2017 Category A7 - Cross-Site Scripting (XSS)
- CWE - CWE-79 - Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

Enabling file access for WebViews is security-sensitive (java:S6363)

Severidad: MAJOR

Assess_the_problem:

Ask Yourself Whether

- No local files have to be accessed by the Webview.
- The WebView contains untrusted data that could cause harm when rendered.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

```
import android.webkit.WebView;

WebView webView = (WebView) findViewById(R.id.webview);
webView.getSettings().setAllowFileAccess(true); // Sensitive
webView.getSettings().setAllowContentAccess(true); // Sensitive
```

Root_cause:

WebViews can be used to display web content as part of a mobile application. A browser engine is used to render and display the content. Like a web application, a mobile application that uses WebViews can be vulnerable to Cross-Site Scripting if untrusted code is rendered.

If malicious JavaScript code in a WebView is executed this can leak the contents of sensitive files when access to local files is enabled.

Default:

WebViews can be used to display web content as part of a mobile application. A browser engine is used to render and display the content. Like a web application, a mobile application that uses WebViews can be vulnerable to Cross-Site Scripting if untrusted code is rendered.

If malicious JavaScript code in a WebView is executed this can leak the contents of sensitive files when access to local files is enabled.

Ask Yourself Whether

- No local files have to be accessed by the Webview.

- The WebView contains untrusted data that could cause harm when rendered.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

It is recommended to disable access to local files for WebViews unless it is necessary. In the case of a successful attack through a Cross-Site Scripting vulnerability the attackers attack surface decreases drastically if no files can be read out.

Sensitive Code Example

```
import android.webkit.WebView;

WebView webView = (WebView) findViewById(R.id.webview);
webView.getSettings().setAllowFileAccess(true); // Sensitive
webView.getSettings().setAllowContentAccess(true); // Sensitive
```

Compliant Solution

```
import android.webkit.WebView;

WebView webView = (WebView) findViewById(R.id.webview);
webView.getSettings().setAllowFileAccess(false);
webView.getSettings().setAllowContentAccess(false);
```

See

- OWASP - Top 10 2021 Category A3 - Injection
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- OWASP - Top 10 2017 Category A7 - Cross-Site Scripting (XSS)
- CWE - CWE-79 - Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

How_to_fix:

Recommended Secure Coding Practices

It is recommended to disable access to local files for WebViews unless it is necessary. In the case of a successful attack through a Cross-Site Scripting vulnerability the attackers attack surface decreases drastically if no files can be read out.

Compliant Solution

```
import android.webkit.WebView;

WebView webView = (WebView) findViewById(R.id.webview);
webView.getSettings().setAllowFileAccess(false);
webView.getSettings().setAllowContentAccess(false);
```

See

- OWASP - Top 10 2021 Category A3 - Injection
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- OWASP - Top 10 2017 Category A7 - Cross-Site Scripting (XSS)
- CWE - CWE-79 - Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

Non-capturing groups without quantifier should not be used (java:S6395)

Severidad: MAJOR

Root_cause:

Sub-patterns can be wrapped by parentheses to build a group. This enables to restrict alternations, back reference the group or apply quantifier to the sub-pattern.

If this group should not be part of the match result or if no reference to this group is required, a non-capturing group can be created by adding `?:` behind the opening parenthesis.

However, if this non-capturing group does not have a quantifier, or does not wrap an alternation, then imaging this group is redundant.

Noncompliant code example

```
"(?:number)\\d{2}"
```

Compliant solution

```
"number\\d{2}"          // it is anyway required
"(?:number)?\\d{2}"     // it is in fact optional
```

Exceptions

This rule does not report an issue if the non-capturing group is an alternation.

```
"(?:number|string)"
```

Superfluous curly brace quantifiers should be avoided (java:S6396)

Severidad: MAJOR

Root_cause:

Curly brace quantifiers in regular expressions can be used to have a more fine-grained control over how many times the character or the sub-expression preceeding them should occur. They can be used to match an expression exactly n times with `{n}`, between n and m times with `{n,m}`, or at least n times with `{n,}`. In some cases, using such a quantifier is superfluous for the semantic of the regular expression, and it can be removed to improve readability. This rule raises an issue when one of the following quantifiers is encountered:

- `{1,1}` or `{1}`: they match the expression exactly once. The same behavior can be achieved without the quantifier.
- `{0,0}` or `{0}`: they match the expression zero times. The same behavior can be achieved by removing the expression.

Noncompliant code example

```
"ab{1,1}c"
"ab{1}c"
"ab{0,0}c"
"ab{0}c"
```

Compliant solution

```
"abc"
"ac"
```

Character classes in regular expressions should not contain only one character (java:S6397)

Severidad: MAJOR

Root_cause:

Character classes in regular expressions are a convenient way to match one of several possible characters by listing the allowed characters or ranges of characters. If a character class contains only one character, the effect is the same as just writing the character without a character class.

Thus, having only one character in a character class is usually a simple oversight that remained after removing other characters of the class.

Noncompliant code example

```
"a[b]c"
"[\\^]"
```

Compliant solution

```
"abc"
"\\^"
"a[*]c" // Compliant, see Exceptions
```

Exceptions

This rule does not raise when the character inside the class is a metacharacter. This notation is sometimes used to avoid escaping (e.g., `[.]{3}` to match three dots).

"ThreadLocal" variables should be cleaned up when no longer used (java:S5164)

Severidad: MAJOR

Resources:

- [Understanding Memory Leaks in Java]()

Root_cause:

`ThreadLocal` variables are supposed to be garbage collected once the holding thread is no longer alive. Memory leaks can occur when holding threads are re-used which is the case on application servers using pool of threads.

To avoid such problems, it is recommended to always clean up `ThreadLocal` variables using the `remove()` method to remove the current thread’s value for the `ThreadLocal` variable.

In addition, calling `set(null)` to remove the value might keep the reference to `this` pointer in the map, which can cause memory leak in some scenarios. Using `remove` is safer to avoid this issue.

Noncompliant code example

```
public class ThreadLocalUserSession implements UserSession {

  private static final ThreadLocal<UserSession> DELEGATE = new ThreadLocal<>();

  public UserSession get() {
    UserSession session = DELEGATE.get();
    if (session != null) {
      return session;
    }
    throw new UnauthorizedException("User is not authenticated");
  }

  public void set(UserSession session) {
    DELEGATE.set(session);
  }

   public void incorrectCleanup() {
     DELEGATE.set(null); // Noncompliant
   }

  // some other methods without a call to DELEGATE.remove()
}
```

Compliant solution

```
public class ThreadLocalUserSession implements UserSession {

  private static final ThreadLocal<UserSession> DELEGATE = new ThreadLocal<>();

  public UserSession get() {
    UserSession session = DELEGATE.get();
    if (session != null) {
      return session;
    }
    throw new UnauthorizedException("User is not authenticated");
  }

  public void set(UserSession session) {
    DELEGATE.set(session);
  }

  public void unload() {
    DELEGATE.remove(); // Compliant
  }

  // ...
}
```

Exceptions

Rule will not detect non-private `ThreadLocal` variables, because `remove()` can be called from another class.

Back references in regular expressions should only refer to capturing groups that are matched before the reference (java:S6001)

Severidad: CRITICAL

Root_cause:

When a back reference in a regex refers to a capturing group that hasn't been defined yet (or at all), it can never be matched. Named back references throw a `PatternSyntaxException` in that case; numeric back references fail silently when they can't match, simply making the match fail.

When the group is defined before the back reference but on a different control path (like in `(.)|\1` for example), this also leads to a situation where the back reference can never match.

Noncompliant code example

```
Pattern.compile("\\1(.)"); // Noncompliant, group 1 is defined after the back reference
Pattern.compile("(.)\\2"); // Noncompliant, group 2 isn't defined at all
Pattern.compile("(.)|\\1"); // Noncompliant, group 1 and the back reference are in different branches
Pattern.compile("(?<x>.)|\\k<x>"); // Noncompliant, group x and the back reference are in different branches
```

Compliant solution

```
Pattern.compile("(.)\\1");
Pattern.compile("(?<x>.)\\k<x>");
```

Regex lookahead assertions should not be contradictory (java:S6002)

Severidad: CRITICAL

Introduction:

This rule raises an issue when a regex lookahead contradicts the rest of the regex.

Root_cause:

Lookahead assertions are a regex feature that makes it possible to look ahead in the input without consuming it. It is often used at the end of regular expressions to make sure that substrings only match when they are followed by a specific pattern.

For example, the following pattern will match an "a" only if it is directly followed by a "b". This does not consume the "b" in the process:

```
Pattern.compile("a(?=b)");
```

However, lookaheads can also be used in the middle (or at the beginning) of a regex. In that case there is the possibility that what comes after the lookahead contradicts the pattern inside the lookahead. Since the lookahead does not consume input, this makes the lookahead impossible to match and is a sign that there's a mistake in the regular expression that should be fixed.

Noncompliant code example

```
Pattern.compile("(?=a)b"); // Noncompliant, the same character can't be equal to 'a' and 'b' at the same time
```

Compliant solution

```
Pattern.compile("(?<=a)b");
Pattern.compile("a(?=b)");
```

Reluctant quantifiers in regular expressions should be followed by an expression that can't match the empty string (java:S6019)

Severidad: MAJOR

Root_cause:

When a reluctant quantifier (such as `*?` or `+?`) is followed by a pattern that can match the empty string or directly by the end of the regex, it will always match the empty string when used with methods that find partial matches (such as `find`, `replaceAll`, `split` etc.).

Similarly, when used with methods that find full matches, a reluctant quantifier that's followed directly by the end of the regex (or a pattern that always matches the empty string, such as `()`) behaves indistinguishably from a greedy quantifier while being less efficient.

This is likely a sign that the regex does not work as intended.

Noncompliant code example

```
"start123endstart456".replaceAll("start\\w*?(end)?", "x"); // Noncompliant. In contrast to what one would expect, the result is not "xx".
str.matches("\\d*?"); // Noncompliant. Matches the same as "\d*", but will backtrack in every position.
```

Compliant solution

```
"start123endstart456".replaceAll("start\\w*?(end|$)", "x"); // Result is "xx".
str.matches("\\d*");
```

Region should be set explicitly when creating a new "AwsClient" (java:S6241)

Severidad: MAJOR

Resources:

- Tuning the AWS Java SDK 2.x to reduce startup time
- Optimizing cold start performance for AWS Lambda
- Environment variable configuration
- Automatically Determine the AWS Region

Root_cause:

If the region is not specified when creating a new AwsClient with an AwsClientBuilder, the AWS SDK will execute some logic to identify the endpoint automatically.

While it will probably identify the correct one, this extra logic will slow down startup time, already known to be a hotspot.

You should therefore always define the logic to set the region yourself. This is typically done by retrieving the region from the Lambda provided AWS_REGION environment variable.

This will make the code more explicit and spare initialization time.

This rule reports an issue when the region is not set when creating an AwsClient.

Noncompliant code example

```
S3Client.builder()
    .credentialsProvider(EnvironmentVariableCredentialsProvider.create())
    .build();
```

Compliant solution

```
S3Client.builder()
    .region(Region.of(System.getenv(SdkSystemSetting.AWS_REGION.environmentVariable()))
    .credentialsProvider(EnvironmentVariableCredentialsProvider.create())
    .build();
```

Reusable resources should be initialized at construction time of Lambda functions (java:S6243)

Severidad: MAJOR

Resources:

- Tuning the AWS Java SDK 2.x to reduce startup time
- Best practices for working with AWS Lambda functions
- Understanding Container Reuse in AWS Lambda

Root_cause:

Resources that can be reused across multiple invocations of the Lambda function should be initialized at construction time. For example in the constructor of the class, or in field initializers. This way, when the same container is reused for multiple function invocations, the existing instance can be reused, along with all resources stored in its fields. It is a good practice to reuse SDK clients and database connections by initializing them at class construction time, to avoid recreating them on every lambda invocation. Failing to do so can lead to performance degradation, and when not closed properly, even out of memory errors.

This rule reports an issue when the SDK client or the database connection is initialized locally inside a Lambda function.

Noncompliant code example

```
public class App implements RequestHandler<Object, Object> {
    @Override
    public Object handleRequest(final Object input, final Context context) {
      S3Client s3Client = DependencyFactory.s3Client();
      s3Client.listBuckets();
      // ...
    }
}
```

Compliant solution

```
public class App implements RequestHandler<Object, Object> {
    private final S3Client s3Client;

    public App() {
      s3Client = DependencyFactory.s3Client();
    }

    @Override
    public Object handleRequest(final Object input, final Context context) {
      s3Client.listBuckets();
      // ...
    }
}
```

Consumer Builders should be used (java:S6244)

Severidad: MINOR

Root_cause:

Some API, like the AWS SDK, heavily rely on the builder pattern to create different data structures. Despite all the benefits, this pattern can become really verbose, especially when dealing with nested structures. In order to reach a more concise code, "Consumer Builders", also called "Consumer Interface" are often introduced.

The idea is to overload the methods taking others structures in a Builder with a Consumer of Builder instead. This enables to use a lambda instead of nesting another Builder, resulting in more concise and readable code.

This rule reports an issue when the Consumer Builder methods could be used instead of the classical ones.

Noncompliant code example

```
SendEmailRequest.builder()
  .destination(Destination.builder()
    .toAddresses("to-email@domain.com")
    .bccAddresses("bcc-email@domain.com")
    .build())
.build();
```

Compliant solution

```
SendEmailRequest.builder()
  .destination(d -> d.toAddresses("to-email@domain.com").bccAddresses("bcc-email@domain.com"))
  .build();
```

Resources:

- Consumer Builders in the AWS SDK for Java v2

XML parsers should not allow inclusion of arbitrary files (java:S6373)

Severidad: BLOCKER

Root_cause:

When the XML parser will encounter an `xinclude` element, it will try to load the file pointed to by the `href` attribute into the document. Included files can either be local files found on the file system of the application server, or remote files that are downloaded over HTTP, SMB, or other protocols, depending on the capabilities of the application and server.

The files that can be accessed that way are only limited by the entitlement of the application on the local system and the network filtering the server is subject to.

This issue is particularly severe when the XML parser is used to parse untrusted documents. For example, when user-submitted XML messages are parsed that way.

What is the potential impact?

Allowing the inclusion of arbitrary files in XML documents can have two main consequences depending on what type of file is included: local or remote.

Sensitive file disclosure

If the application allows the inclusion of arbitrary files through the use of the `xinclude` element, it might be used to disclose arbitrary files from the local file system. Depending on the application’s permissions on the file system, configuration files, runtime secrets, or Personally Identifiable Information could be leaked.

This is particularly true if the affected parser is used to process untrusted XML documents.

Server-side request forgery

When used to retrieve remote files, the application will send network requests to remote hosts. Moreover, it will do so from its current network location, which can have severe consequences if the application server is located on a sensitive network, such as the company corporate network or a DMZ hosting other applications.

Attackers exploiting this issue could try to access internal backend services or corporate file shares. It could allow them to access more sensitive files, bypass authentication mechanisms from frontend applications, or exploit further vulnerabilities in the local services. Note that, in some cases, the requests sent from the application can be automatically authenticated on federated locations. This is often the case in Windows environments when using Active Directory federated authentication.

How_to_fix:

The following code is vulnerable because it explicitly enables the `xinclude` feature.

Noncompliant code example

```
import javax.xml.parsers.SAXParserFactory;

SAXParserFactory factory = SAXParserFactory.newInstance();

factory.setXIncludeAware(true); // Noncompliant
factory.setFeature("http://apache.org/xml/features/xinclude", true); // Noncompliant
```

Compliant solution

```
import javax.xml.parsers.SAXParserFactory;

SAXParserFactory factory = SAXParserFactory.newInstance();

factory.setXIncludeAware(false);
factory.setFeature("http://apache.org/xml/features/xinclude", false);
```

Introduction:

XML standard allows the inclusion of XML files with the `xinclude` element. When an XML parser component is set up with the `http://apache.org/xml/features/xinclude` feature, it will follow the standard and allow the inclusion of remote files.

How_to_fix:

The following code is vulnerable because it explicitly enables the `xinclude` feature.

Noncompliant code example

```
import org.jdom2.input.SAXBuilder;

SAXBuilder builder = new SAXBuilder();
builder.setFeature("http://apache.org/xml/features/xinclude", true); // Noncompliant
```

Compliant solution

```
import org.jdom2.input.SAXBuilder;

SAXBuilder builder = new SAXBuilder();
builder.setFeature("http://apache.org/xml/features/xinclude", false);
```

How does this work?

The compliant code example explicitly prevents the inclusion of files in XML documents by setting the `http://apache.org/xml/features/xinclude` feature property to `false`.

Resources:

Documentation

- OWASP - OWASP XXE Prevention Cheat Sheet
- Java documentation - XML External Entity Injection Attack
- W3C - XML Inclusions (XInclude) Version 1.1

Standards

- OWASP - Top 10 2017 - Category A4 - XML External Entities (XXE)
- OWASP - Top 10 2021 - Category A5 - Security Misconfiguration
- CWE - CWE-611 - Improper Restriction of XML External Entity Reference
- CWE - CWE-827 - Improper Control of Document Type Definition
- STIG Viewer - Application Security and Development: V-222608 - The application must not be vulnerable to XML-oriented attacks.

How_to_fix:

The following code is vulnerable because it explicitly enables the `xinclude` feature.

Noncompliant code example

```
import org.dom4j.io.SAXReader;

SAXReader xmlReader = new SAXReader();
xmlReader.setFeature("http://apache.org/xml/features/xinclude", true); // Noncompliant
```

Compliant solution

```
import org.dom4j.io.SAXReader;

SAXReader xmlReader = new SAXReader();
xmlReader.setFeature("http://apache.org/xml/features/xinclude", false);
```

XML signatures should be validated securely (java:S6377)

Severidad: MAJOR

Introduction:

XML signatures are a method used to ensure the integrity and authenticity of XML documents. However, if XML signatures are not validated securely, it can lead to potential vulnerabilities.

How_to_fix:

For versions of Java before 17, secure validation is disabled by default unless the application runs with a security manager, which is rare. It should be enabled explicitly by setting the `org.jcp.xml.dsig.secureValidation` attribute to true with the `javax.xml.crypto.dsig.dom.DOMValidateContext.setProperty` method.

For Java 17 and higher, secure validation is enabled by default.

Noncompliant code example

```
NodeList signatureElement = doc.getElementsByTagNameNS(XMLSignature.XMLNS, "Signature");

XMLSignatureFactory fac = XMLSignatureFactory.getInstance("DOM");
DOMValidateContext valContext = new DOMValidateContext(new KeyValueKeySelector(), signatureElement.item(0)); // Noncompliant
XMLSignature signature = fac.unmarshalXMLSignature(valContext);

boolean signatureValidity = signature.validate(valContext);
```

Compliant solution

```
NodeList signatureElement = doc.getElementsByTagNameNS(XMLSignature.XMLNS, "Signature");

XMLSignatureFactory fac = XMLSignatureFactory.getInstance("DOM");
DOMValidateContext valContext = new DOMValidateContext(new KeyValueKeySelector(), signatureElement.item(0));
valContext.setProperty("org.jcp.xml.dsig.secureValidation", Boolean.TRUE);
XMLSignature signature = fac.unmarshalXMLSignature(valContext);

boolean signatureValidity = signature.validate(valContext);
```

How does this work?

When XML Signature secure validation mode is enabled, XML Signatures are processed more securely. It enforces a number of restrictionsto to protect from XML Documents that may contain hostile constructs that can cause denial-of-service or other types of security issues.

These restrictions can protect you from XML Signatures that may contain potentially hostile constructs that can cause denial-of-service or other types of security issues.

Resources:

Documentation

- Oracle Java Documentation - XML Digital Signature API Overview and Tutorial

Standards

- OWASP - Top 10:2021 A02:2021 - Cryptographic Failures
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- CWE - CWE-347 - Improper Verification of Cryptographic Signature
- STIG Viewer - Application Security and Development: V-222608 - The application must not be vulnerable to XML-oriented attacks.

Root_cause:

Before Java 17, XML Digital Signature API does not apply restrictions on XML signature validation unless the application runs with a security manager, which is rare.

What is the potential impact

By not enforcing secure validation, the XML Digital Signature API is more susceptible to attacks such as signature spoofing and injections.

Increased Vulnerability to Signature Spoofing

By disabling secure validation, the application becomes more susceptible to signature spoofing attacks. Attackers can potentially manipulate the XML signature in a way that bypasses the validation process, allowing them to forge or tamper with the signature. This can lead to the acceptance of invalid or maliciously modified signatures, compromising the integrity and authenticity of the XML documents.

Risk of Injection Attacks

Disabling secure validation can expose the application to injection attacks. Attackers can inject malicious code or entities into the XML document, taking advantage of the weakened validation process. In some cases, it can also expose the application to denial-of-service attacks. Attackers can exploit vulnerabilities in the validation process to cause excessive resource consumption or system crashes, leading to service unavailability or disruption.

Hash-based collections with known capacity should be initialized with the proper related static method. (java:S6485)

Severidad: MAJOR

Root_cause:

When creating an instance of HashMap or HashSet, the developer can pick a constructor with known capacity. However, the requested capacity is not fully allocated by default. Indeed, when the collection reaches the load factor of the collection (default: 0.75), the collection is resized on the fly, leading to unexpected performance issues.

How_to_fix:

As of Java 19, hash-based collections provide a static method that allocates the requested capacity at construction time.

Noncompliant code example

```
private static final int KNOWN_CAPACITY = 1_000_000;

public static Map<String, Integer> buildAMap() {
    return new HashMap<>(KNOWN_CAPACITY); // Noncompliant
}

public static Set<String> buildASet() {
    return new HashSet<>(KNOWN_CAPACITY); // Noncompliant
}
```

Compliant solution

```
private static final int KNOWN_CAPACITY = 1_000_000;

public static Map<String, Integer> buildABetterMap() {
    return HashMap.newHashMap(KNOWN_CAPACITY);
}

public static Set<String> buildABetterSet() {
    return HashSet.newHashSet(KNOWN_CAPACITY);
}

public static Set<String> buildABetterSet(float customLoadFactor) {
    return new HashSet<>(KNOWN_CAPACITY, customLoadFactor);
}
```

Introduction:

Hash-based collections with known capacity should be initialized with the proper related static method.

Resources:

Documentation

- https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/HashMap.html#newHashMap(int)
- https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/HashSet.html#newHashSet(int)
- https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/LinkedHashMap.html#newLinkedHashMap(int)
- https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/LinkedHashSet.html#newLinkedHashSet(int)
- https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/WeakHashMap.html#newWeakHashMap(int)

Message:
Replace this call to the constructor with the better suited static method.

Highlighting:
The infringing constructor call.

Parentheses should be removed from a single lambda parameter when its type is inferred (java:S1611)

Severidad: MINOR

Root_cause:

Lambda expressions with only one argument with an inferred type (i.e., no explicit type declaration) can be written without parentheses around that single parameter. This syntax is simpler, more compact and readable than using parentheses and is therefore preferred.

This rule is automatically disabled when the project's `sonar.java.source` is lower than `8`, as lambda expressions were introduced in Java 8.

Noncompliant code example

```
(x) -> x * 2
```

Compliant solution

```
x -> x * 2
```

Lambdas should be replaced with method references (java:S1612)

Severidad: MINOR

How_to_fix:

Refer to the called method by its reference instead of wrapping it in a lambda expression.

For instance:

- `null` checks can be replaced with references to `Objects::isNull` and `Objects::nonNull`
- Casts can be replaced with `SomeClass.class::cast`
- `instanceof` can be replaced with `SomeClass.class::isInstance`

Noncompliant code example

```
class A {
  void process(List<A> list) {
    list.stream()
      .filter(myListValue -> myListValue instanceof B)     // Noncompliant
      .map(listValueToMap -> (B) listValueToMap)           // Noncompliant
      .map(bValueToMap -> bValueToMap.<String>getObject()) // Noncompliant
      .forEach(o -> System.out.println(o));                // Noncompliant
  }
}

class B extends A {
  <T> T getObject() {
    return null;
  }
}
```

Compliant solution

```
class A {
  void process(List<A> list) {
    list.stream()
      .filter(B.class::isInstance)   // Compliant
      .map(B.class::cast)            // Compliant
      .map(B::<String>getObject)     // Compliant
      .forEach(System.out::println); // Compliant
  }
}

class B extends A {
  <T> T getObject() {
    return null;
  }
}
```

Resources:

- Method References - The Java™ Tutorials

Root_cause:

Method or constructor references are more readable than lambda expressions in many situations, and may therefore be preferred.

However, method references are sometimes less concise than lambdas. In such cases, it might be preferrable to keep the lambda expression for better readability. Therefore, this rule only raises issues on lambda expressions where the replacement method reference is shorter.

This rule is automatically disabled when the project’s `sonar.java.source` is lower than `8`, as lambda expressions were introduced in Java 8.

Unused assignments should be removed (java:S1854)

Severidad: MAJOR

How_to_fix:

Remove the unnecesarry assignment, then test the code to make sure that the right-hand side of a given assignment had no side effects (e.g. a method that writes certain data to a file and returns the number of written bytes).

Noncompliant code example

```
int foo(int y) {
  int x = 100; // Noncompliant: dead store
  x = 150;     // Noncompliant: dead store
  x = 200;
  return x + y;
}
```

Compliant solution

```
int foo(int y) {
  int x = 200; // Compliant: no unnecessary assignment
  return x + y;
}
```

Root_cause:

Dead stores refer to assignments made to local variables that are subsequently never used or immediately overwritten. Such assignments are unnecessary and don’t contribute to the functionality or clarity of the code. They may even negatively impact performance. Removing them enhances code cleanliness and readability. Even if the unnecessary operations do not do any harm in terms of the program’s correctness, they are - at best - a waste of computing resources.

Exceptions

This rule ignores initializations to `-1`, `0`, `1`, `null`, `true`, `false` and `""`.

Resources:

Standards

- CWE - CWE-563 - Assignment to Variable without Use ('Unused Variable')

Related rules

- S2583 - Conditionally executed code should be reachable
- S2589 - Boolean expressions should not be gratuitous
- S3516 - Methods returns should not be invariant
- S3626 - Jump statements should not be redundant

"toString()" should never be called on a String object (java:S1858)

Severidad: MINOR

Root_cause:

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);
```

Synchronization should not be done on instances of value-based classes (java:S1860)

Severidad: MAJOR

Resources:

- CERT - Do not synchronize on objects that may be reused
- OpenJDK - JEP 390: Warnings for Value-Based Classes
- Java Documentation - Value-based Classes

How_to_fix:

Replace instances of value-based classes with a new object instance to synchronize on.

Noncompliant code example

```
private static final Boolean bLock = Boolean.FALSE;
private static final Integer iLock = Integer.valueOf(0);
private static final String sLock = "LOCK";
private static final List<String> listLock = List.of("a", "b", "c", "d");

public void doSomething() {

  synchronized(bLock) {  // Noncompliant
      ...
  }
  synchronized(iLock) {  // Noncompliant
      ...
  }
  synchronized(sLock) {  // Noncompliant
      ...
  }
  synchronized(listLock) {  // Noncompliant
      ...
  }
```

Compliant solution

```
private static final Object lock1 = new Object();
private static final Integer iLock = new Integer(42);
private static final String sLock = new String("A brand new string in memory!");
private static final List<String> listLock = new ArrayList<>();

public void doSomething() {

  synchronized(lock1) { // Compliant
      ...
  }
  synchronized(iLock) { // Compliant
      ...
  }
  synchronized(sLock) { // Compliant
      ...
  }
  synchronized(listLock) { // Compliant
      ...
  }
```

Root_cause:

In Java, value-based classes are those for which instances are final and immutable, like `String`, `Integer` and so on, and their identity relies on their value and not their reference. When a variable of one of these types is instantiated, the JVM caches its value, and the variable is just a reference to that value. For example, multiple `String` variables with the same value "Hello world!" will refer to the same cached string literal in memory.

The `synchronized` keyword tells the JVM to only allow the execution of the code contained in the following block to one `Thread` at a time. This mechanism relies on the identity of the object that is being synchronized between threads, to prevent that if object X is locked, it will still be possible to lock another object Y.

It means that the JVM will fail to correctly synchronize threads on instances of the aforementioned value-based classes, for instance:

```
// These variables "a" and "b" will effectively reference the same object in memory
Integer a = 0;
Integer b = 0;

// This means that in the following code, the JVM could try to lock and execute
// on the variable "a" because "b" was notified to be released, as the two Integer variables
// are the same object to the JVM
void syncMethod(int x) {
    synchronized (a) {
        if (a == x) {
        // ... do something here
        }
    }
    synchronized (b) {
        if (b == x) {
        // ... do something else
        }
    }
}
```

This behavior can cause unrelated threads to deadlock with unclear stacktraces.

Within the JDK, types which should not be used for synchronization include:

- `String` literals
- Primitive wrapper classes in `java.lang` (such as `Boolean` with `Boolean.FALSE` and `Boolean.TRUE`)
- The class `java.lang.Runtime.Version`
- The `Optional*` classes in `java.util`: `Optional`, `OptionalInt`, `OptionalLong`, and `OptionalDouble`
- Various classes in the `java.time` API: `Instant`, `LocalDate`, `LocalTime`, `LocalDateTime`, `ZonedDateTime`, `ZoneId`, `OffsetTime`, `OffsetDateTime`, `ZoneOffset`, `Duration`, `Period`, `Year`, `YearMonth`, and `MonthDay`
- Various classes in the `java.time.chrono` API: `MinguoDate`, `HijrahDate`, `JapaneseDate`, and `ThaiBuddhistDate`
- The interface `java.lang.ProcessHandle` and its implementation classes
- The implementation classes of the collection factories in `java.util`: `List.of`, `List.copyOf`, `Set.of`, `Set.copyOf`, `Map.of`, `Map.copyOf`, `Map.ofEntries`, and `Map.entry`.

Related "if/else if" statements should not have the same condition (java:S1862)

Severidad: MAJOR

Resources:

- CERT, MSC12-C. - Detect and remove code that has no effect or is never executed

Root_cause:

A chain of `if`/`else if` statements is evaluated from top to bottom. At most, only one branch will be executed: the first one with a condition that evaluates to `true`.

Therefore, duplicating a condition automatically leads to dead code. Usually, this is due to a copy/paste error. At best, it’s simply dead code and at worst, it’s a bug that is likely to induce further bugs as the code is maintained, and obviously it could lead to unexpected behavior.

Noncompliant code example

```
if (param == 1)
  openWindow();
else if (param == 2)
  closeWindow();
else if (param == 1)  // Noncompliant
  moveWindowToTheBackground();
}
```

Compliant solution

```
if (param == 1)
  openWindow();
else if (param == 2)
  closeWindow();
else if (param == 3)
  moveWindowToTheBackground();
}
```

Exceptions should not be thrown from servlet methods (java:S1989)

Severidad: MINOR

Resources:

Articles & blog posts

- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- CWE - CWE-600 - Uncaught Exception in Servlet
- CERT, ERR01-J. - Do not allow exceptions to expose sensitive information

Root_cause:

Servlets are components in Java web development, responsible for processing HTTP requests and generating responses. In this context, exceptions are used to handle and manage unexpected errors or exceptional conditions that may occur during the execution of a servlet.

Catching exceptions within the servlet allows us to convert them into meaningful, user-friendly messages. Otherwise, failing to catch exceptions will propagate them to the servlet container, where the default error-handling mechanism may impact the overall security and stability of the server.

Possible security problems are:

1. **Vulnerability to denial-of-service attacks:** Not caught exceptions can leave the servlet container in an unstable state, which can exhaust the available resources and make the system unavailable in the worst cases.
2. **Exposure of sensitive information:** Exceptions handled by the servlet container, by default, expose detailed error messages or debugging information to the user, which may contain sensitive data such as stack traces, database connection, or system configuration.

Unfortunately, servlet method signatures do not force developers to handle `IOException` and `ServletException`:

```
public void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException, ServletException {
}
```

To prevent this risk, this rule enforces all exceptions to be caught within the "do*" methods of servlet classes.

How_to_fix:

Surround all method calls that may throw an exception with a `try/catch` block.

In the following example, the `getByName` method may throw an `UnknownHostException`.

Noncompliant code example

```
public void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException, ServletException {
  InetAddress addr = InetAddress.getByName(request.getRemoteAddr()); // Noncompliant
  //...
}
```

Compliant solution

```
public void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException, ServletException {
  try {
    InetAddress addr = InetAddress.getByName(request.getRemoteAddr());
    //...
  }
  catch (UnknownHostException ex) {  // Compliant
    //...
  }
}
```

Literal boolean values and nulls should not be used in assertions (java:S2701)

Severidad: MINOR

Root_cause:

There's no reason to use literal boolean values or nulls in assertions. Instead of using them with *assertEquals*, *assertNotEquals* and similar methods, you should be using *assertTrue*, *assertFalse*, *assertNull* or *assertNotNull* instead (or *isNull* etc. when using Fest). Using them with assertions unrelated to equality (such as *assertNull*) is most likely a bug.

Supported frameworks:

- JUnit3
- JUnit4
- JUnit5
- Fest assert

Noncompliant code example

```
Assert.assertTrue(true);  // Noncompliant
assertThat(null).isNull(); // Noncompliant

assertEquals(true, something()); // Noncompliant
assertNotEquals(null, something()); // Noncompliant
```

Compliant solution

```
assertTrue(something());
assertNotNull(something());
```

"DateUtils.truncate" from Apache Commons Lang library should not be used (java:S2718)

Severidad: MAJOR

Root_cause:

The `ZonedDateTime` is an immutable representation of a date-time with a time-zone, introduced in Java 8. This class stores all date and time fields, to a precision of nanoseconds, and a time zone, with a zone offset used to handle ambiguous local date times.

Date truncation to a specific time unit means setting the values up to the specific time unit to zero while keeping the values of the larger time units unchanged.

The `ZonedDateTime` class provides a `truncatedTo` method that allows truncating the date in a significantly faster way than the `DateUtils` class from Commons Lang.

Note that this rule is automatically disabled when the project's `sonar.java.source` is lower than `8`.

Noncompliant code example

```
public Date trunc(Date date) {
  return DateUtils.truncate(date, Calendar.SECOND);  // Noncompliant
}
```

Compliant solution

```
public Date trunc(Date date) {
  Instant instant = date.toInstant();
  ZonedDateTime zonedDateTime = instant.atZone(ZoneId.systemDefault());
  ZonedDateTime truncatedZonedDateTime = zonedDateTime.truncatedTo(ChronoUnit.SECONDS);
  Instant truncatedInstant = truncatedZonedDateTime.toInstant();
  return Date.from(truncatedInstant);
}
```

Resources:

- Oracle SDK 20 - ZonedDateTime#truncatedTo

Unnecessary semicolons should be omitted (java:S2959)

Severidad: MINOR

Root_cause:

Under the reasoning that cleaner code is better code, the semicolon at the end of a try-with-resources construct should be omitted because it can be omitted.

Noncompliant code example

```
try (ByteArrayInputStream b = new ByteArrayInputStream(new byte[10]);  // ignored; this one's required
      Reader r = new InputStreamReader(b);)   // Noncompliant
{
   //do stuff
}
```

Compliant solution

```
try (ByteArrayInputStream b = new ByteArrayInputStream(new byte[10]);
      Reader r = new InputStreamReader(b))
{
   //do stuff
}
```

Expanding archive files without controlling resource consumption is security-sensitive (java:S5042)

Severidad: CRITICAL

How_to_fix:

Recommended Secure Coding Practices

- Define and control the ratio between compressed and uncompressed data, in general the data compression ratio for most of the legit archives is 1 to 3.
- Define and control the threshold for maximum total size of the uncompressed data.
- Count the number of file entries extracted from the archive and abort the extraction if their number is greater than a predefined threshold, in particular it’s not recommended to recursively expand archives (an entry of an archive could be also an archive).

Compliant Solution

Do not rely on getsize to retrieve the size of an uncompressed entry because this method returns what is defined in the archive headers which can be forged by attackers, instead calculate the actual entry size when unzipping it:

```
File f = new File("ZipBomb.zip");
ZipFile zipFile = new ZipFile(f);
Enumeration<? extends ZipEntry> entries = zipFile.entries();

int THRESHOLD_ENTRIES = 10000;
int THRESHOLD_SIZE = 1000000000; // 1 GB
double THRESHOLD_RATIO = 10;
int totalSizeArchive = 0;
int totalEntryArchive = 0;

while(entries.hasMoreElements()) {
  ZipEntry ze = entries.nextElement();
  InputStream in = new BufferedInputStream(zipFile.getInputStream(ze));
  OutputStream out = new BufferedOutputStream(new FileOutputStream("./output_onlyfortesting.txt"));

  totalEntryArchive ++;

  int nBytes = -1;
  byte[] buffer = new byte[2048];
  int totalSizeEntry = 0;

  while((nBytes = in.read(buffer)) > 0) { // Compliant
      out.write(buffer, 0, nBytes);
      totalSizeEntry += nBytes;
      totalSizeArchive += nBytes;

      double compressionRatio = totalSizeEntry / ze.getCompressedSize();
      if(compressionRatio > THRESHOLD_RATIO) {
        // ratio between compressed and uncompressed data is highly suspicious, looks like a Zip Bomb Attack
        break;
      }
  }

  if(totalSizeArchive > THRESHOLD_SIZE) {
      // the uncompressed data size is too much for the application resource capacity
      break;
  }

  if(totalEntryArchive > THRESHOLD_ENTRIES) {
      // too much entries in this archive, can lead to inodes exhaustion of the system
      break;
  }
}
```

See

- OWASP - Top 10 2021 Category A1 - Broken Access Control
- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2017 Category A5 - Broken Access Control
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- CWE - CWE-409 - Improper Handling of Highly Compressed Data (Data Amplification)
- CERT, IDS04-J. - Safely extract files from ZipInputStream
- bamsoftware.com - A better Zip Bomb

Assess_the_problem:

Ask Yourself Whether

Archives to expand are untrusted and:

- There is no validation of the number of entries in the archive.
- There is no validation of the total size of the uncompressed data.
- There is no validation of the ratio between the compressed and uncompressed archive entry.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

```
File f = new File("ZipBomb.zip");
ZipFile zipFile = new ZipFile(f);
Enumeration<? extends ZipEntry> entries = zipFile.entries(); // Sensitive

while(entries.hasMoreElements()) {
  ZipEntry ze = entries.nextElement();
  File out = new File("./output_onlyfortesting.txt");
  Files.copy(zipFile.getInputStream(ze), out.toPath(), StandardCopyOption.REPLACE_EXISTING);
}
```

Root_cause:

Successful Zip Bomb attacks occur when an application expands untrusted archive files without controlling the size of the expanded data, which can lead to denial of service. A Zip bomb is usually a malicious archive file of a few kilobytes of compressed data but turned into gigabytes of uncompressed data. To achieve this extreme compression ratio, attackers will compress irrelevant data (eg: a long string of repeated bytes).

Default:

Successful Zip Bomb attacks occur when an application expands untrusted archive files without controlling the size of the expanded data, which can lead to denial of service. A Zip bomb is usually a malicious archive file of a few kilobytes of compressed data but turned into gigabytes of uncompressed data. To achieve this extreme compression ratio, attackers will compress irrelevant data (eg: a long string of repeated bytes).

Ask Yourself Whether

Archives to expand are untrusted and:

- There is no validation of the number of entries in the archive.
- There is no validation of the total size of the uncompressed data.
- There is no validation of the ratio between the compressed and uncompressed archive entry.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- Define and control the ratio between compressed and uncompressed data, in general the data compression ratio for most of the legit archives is 1 to 3.
- Define and control the threshold for maximum total size of the uncompressed data.
- Count the number of file entries extracted from the archive and abort the extraction if their number is greater than a predefined threshold, in particular it’s not recommended to recursively expand archives (an entry of an archive could be also an archive).

Sensitive Code Example

```
File f = new File("ZipBomb.zip");
ZipFile zipFile = new ZipFile(f);
Enumeration<? extends ZipEntry> entries = zipFile.entries(); // Sensitive

while(entries.hasMoreElements()) {
  ZipEntry ze = entries.nextElement();
  File out = new File("./output_onlyfortesting.txt");
  Files.copy(zipFile.getInputStream(ze), out.toPath(), StandardCopyOption.REPLACE_EXISTING);
}
```

Compliant Solution

Do not rely on getsize to retrieve the size of an uncompressed entry because this method returns what is defined in the archive headers which can be forged by attackers, instead calculate the actual entry size when unzipping it:

```
File f = new File("ZipBomb.zip");
ZipFile zipFile = new ZipFile(f);
Enumeration<? extends ZipEntry> entries = zipFile.entries();

int THRESHOLD_ENTRIES = 10000;
int THRESHOLD_SIZE = 1000000000; // 1 GB
double THRESHOLD_RATIO = 10;
int totalSizeArchive = 0;
int totalEntryArchive = 0;

while(entries.hasMoreElements()) {
  ZipEntry ze = entries.nextElement();
  InputStream in = new BufferedInputStream(zipFile.getInputStream(ze));
  OutputStream out = new BufferedOutputStream(new FileOutputStream("./output_onlyfortesting.txt"));

  totalEntryArchive ++;

  int nBytes = -1;
  byte[] buffer = new byte[2048];
  int totalSizeEntry = 0;

  while((nBytes = in.read(buffer)) > 0) { // Compliant
      out.write(buffer, 0, nBytes);
      totalSizeEntry += nBytes;
      totalSizeArchive += nBytes;

      double compressionRatio = totalSizeEntry / ze.getCompressedSize();
      if(compressionRatio > THRESHOLD_RATIO) {
        // ratio between compressed and uncompressed data is highly suspicious, looks like a Zip Bomb Attack
        break;
      }
  }

  if(totalSizeArchive > THRESHOLD_SIZE) {
      // the uncompressed data size is too much for the application resource capacity
      break;
  }

  if(totalEntryArchive > THRESHOLD_ENTRIES) {
      // too much entries in this archive, can lead to inodes exhaustion of the system
      break;
  }
}
```

See

- OWASP - Top 10 2021 Category A1 - Broken Access Control
- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2017 Category A5 - Broken Access Control
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- CWE - CWE-409 - Improper Handling of Highly Compressed Data (Data Amplification)
- CERT, IDS04-J. - Safely extract files from ZipInputStream
- bamsoftware.com - A better Zip Bomb

Call to Mockito method "verify", "when" or "given" should be simplified (java:S6068)

Severidad: MINOR

Root_cause:

Mockito provides *argument matchers* for flexibly stubbing or verifying method calls.

`Mockito.verify()`, `Mockito.when()`, `Stubber.when()` and `BDDMockito.given()` each have overloads with and without argument matchers.

However, the default matching behavior (i.e. without argument matchers) uses `equals()`. If only the matcher `org.mockito.ArgumentMatchers.eq()` is used, the call is equivalent to the call without matchers, i.e. the `eq()` is not necessary and can be omitted. The resulting code is shorter and easier to read.

Noncompliant code example

```
@Test
public void myTest() {
  given(foo.bar(eq(v1), eq(v2), eq(v3))).willReturn(null);   // Noncompliant
  when(foo.baz(eq(v4), eq(v5))).thenReturn("foo");   // Noncompliant
  doThrow(new RuntimeException()).when(foo).quux(eq(42));    // Noncompliant
  verify(foo).bar(eq(v1), eq(v2), eq(v3));   // Noncompliant
}
```

Compliant solution

```
@Test
public void myTest() {
  given(foo.bar(v1, v2, v3)).willReturn(null);
  when(foo.baz(v4, v5)).thenReturn("foo");
  doThrow(new RuntimeException()).when(foo).quux(42);
  verify(foo).bar(v1, v2, v3);
}
```

Resources:

- Mockito documentation - argument matchers
- S6073 - Mockito argument matchers should be used on all parameters

---

XML parsers should not be vulnerable to Denial of Service attacks (java:S6376)

Severidad: MAJOR

How_to_fix:

Noncompliant code example

```
import org.jdom2.input.SAXBuilder;

SAXBuilder builder = new SAXBuilder();
builder.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, false);  // Noncompliant
```

Compliant solution

```
import org.jdom2.input.SAXBuilder;

SAXBuilder builder = new SAXBuilder();
builder.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);
```

Resources:

Documentation

- Java Documentation - DocumentBuilderFactory Class
- Java Documentation - SAXParserFactory Class
- Java Documentation - SchemaFactory Class
- Java Documentation - TransformerFactory Class
- Java Documentation - Java API for XML Processing (JAXP) Security Guide
- Dom4j Documentation - SAXReader Class
- Jdom2 Documentation - SAXBuilder class
- OWASP - XXE Prevention Cheat Sheet

Standards

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2017 Category A4 - XML External Entities (XXE)
- CWE - CWE-776 - Improper Restriction of Recursive Entity References in DTDs ('XML Entity Expansion')

- STIG Viewer - Application Security and Development: V-222593 - XML-based applications must mitigate DoS attacks by using XML filters, parser options, or gateways.
- STIG Viewer - Application Security and Development: V-222667 - Protections against DoS attacks must be implemented.
- STIG Viewer - Application Security and Development: V-222608 - The application must not be vulnerable to XML-oriented attacks.

Root_cause:

XML files are complex data structures. When a malicious user is able to submit an XML file, it triggers complex processing that may overwhelm the parser. Most of the time, those complex processing are enabled by default, and XML parsers do not take preventive measures against Denial of Service attacks.

What is the potential impact?

When an attacker successfully exploits the vulnerability, it can lead to a Denial of Service (DoS) condition.

System Unavailability

Affected system becomes unresponsive or crashes, rendering it unavailable to legitimate users. This can have severe consequences, especially for critical systems that rely on continuous availability, such as web servers, APIs, or network services.

Amplification Attacks

In some cases, XML parsers Denial of Service attacks can be used as a part of larger-scale amplification attacks. By leveraging the vulnerability, attackers can generate a disproportionately large response from the targeted system, amplifying the impact of their attack. This can result in overwhelming network bandwidth and causing widespread disruption.

Introduction:

XML parsers Denial of Service attacks target XML parsers, which are software components responsible for parsing and interpreting XML documents.

How_to_fix:

Noncompliant code example

```
import javax.xml.parsers.DocumentBuilderFactory;

DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
factory.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, false); // Noncompliant
```

Compliant solution

```
import javax.xml.parsers.DocumentBuilderFactory;

DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
factory.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);
```

How_to_fix:

Noncompliant code example

```
import org.dom4j.io.SAXReader;

SAXReader xmlReader = new SAXReader();
xmlReader.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, false); // Noncompliant
```

Compliant solution

```
import org.dom4j.io.SAXReader;

SAXReader xmlReader = new SAXReader();
xmlReader.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);
```

"String.isEmpty()" should be used to test for emptiness (java:S7158)

Severidad: MINOR

Resources:

Documentation

- Java Documentation - java.lang.String.isEmpty() method

Root_cause:

Calling `String.isEmpty()` clearly communicates the code's intention, which is to test if the string is empty. Using `String.length() == 0` is less direct and makes the code less readable.

How_to_fix:

Noncompliant code example

```
if ("string".length() == 0) { /* … */ } // Noncompliant
```

```
if ("string".length() > 0) { /* … */ } // Noncompliant
```

Compliant solution

```
if ("string".isEmpty()){ /* … */ }
```

```
if (!"string".isEmpty()){ /* … */ }
```

Track comments matching a regular expression (java:S124)

Severidad: MAJOR

Root_cause:

This rule template can be used to create rules which will be triggered when the full content of a comment matches a given regular expression. Note that the regular expression should be expressed using the dotall format (where the `.` character matches any character).

For example, one can create a rule with the regular expression `.*REVIEW.*` to match all comment containing "REVIEW".

Note that, in order to match REVIEW regardless of the case, the `(?i)` modifier should be prepended to the expression, as in `(?i).*REVIEW.*`.

Sections of code should not be commented out (java:S125)

Severidad: MAJOR

Root_cause:

Commented-out code distracts the focus from the actual executed code. It creates a noise that increases maintenance code. And because it is never executed, it quickly becomes out of date and invalid.

Commented-out code should be deleted and can be retrieved from source control history if required.

"if ... else if" constructs should end with "else" clauses (java:S126)

Severidad: CRITICAL

Root_cause:

This rule applies whenever an `if` statement is followed by one or more `else if` statements; the final `else if` should be followed by an `else` statement.

The requirement for a final `else` statement is defensive programming.

The `else` statement should either take appropriate action or contain a suitable comment as to why no action is taken. This is consistent with the requirement to have a final `default` clause in a `switch` statement.

Noncompliant code example

```
if (x == 0) {
  doSomething();
} else if (x == 1) {
  doSomethingElse();
}
```

Compliant solution

```
if (x == 0) {
  doSomething();
} else if (x == 1) {
  doSomethingElse();
} else {
  throw new IllegalStateException();
}
```

Resources:

- CERT, MSC01-C. - Strive for logical completeness
- CERT, MSC57-J. - Strive for logical completeness

"for" loop stop conditions should be invariant (java:S127)

Severidad: MAJOR

Root_cause:

A `for` loop termination condition should test the loop counter against an invariant value that does not change during the execution of the loop. Invariant termination conditions make the program logic easier to understand and maintain.

This rule tracks three types of non-invariant termination conditions:

- When the loop counters are updated in the body of the `for` loop
- When the termination condition depends on a method call
- When the termination condition depends on an object property since such properties could change during the execution of the loop.

How_to_fix:

Noncompliant code example

Make the termination condition invariant by using a constant or a local variable instead of an expression that could change during the execution of the loop.

```
for (int i = 0; i < foo(); i++) { // Noncompliant, "foo()" is not an invariant
  // ...
}
```

Compliant solution

```
int end = foo();
for (int i = 0; i < end; i++) { // Compliant, "end" does not change during loop execution
  // ...
}
```

Noncompliant code example

If this is impossible and the counter variable must be updated in the loop’s body, use a `while` or do `while` loop instead of a `for` loop.

```
for (int i = 0; i < 10; i++) {
  // ...
  if (condition) i++; // Noncompliant, i is updated from within body
  // ...
}
```

Compliant solution

```
int i = 0;
while (i++ < 10) { // Compliant
  // ...
  if (condition) sum++;
  // ...
}
```

Switch cases should end with an unconditional "break" statement (java:S128)

Severidad: BLOCKER

Resources:

- CWE - CWE-484 - Omitted Break Statement in Switch
- CERT, MSC17-C. - Finish every set of statements associated with a case label with a break statement
- CERT, MSC52-J. - Finish every set of statements associated with a case label with a break statement

Root_cause:

When the execution is not explicitly terminated at the end of a switch case, it continues to execute the statements of the following case. While this is sometimes intentional, it often is a mistake which leads to unexpected behavior.

Noncompliant code example

```
switch (myVariable) {
  case 1:
    foo();
    break;
  case 2:  // Both 'doSomething()' and 'doSomethingElse()' will be executed. Is it on purpose ?
    doSomething();
  default:
    doSomethingElse();
    break;
}
```

Compliant solution

```
switch (myVariable) {
  case 1:
    foo();
    break;
  case 2:
    doSomething();
    break;
  default:
    doSomethingElse();
    break;
}
```

Exceptions

This rule is relaxed in the following cases:

```
switch (myVariable) {
  case 0:                                // Empty case used to specify the same behavior for a group of cases.
  case 1:
    doSomething();
    break;
  case 2:                                // Use of a fallthrough comment
    // fallthrough
  case 3:                                // Use of return statement
    return;
  case 4:                                // Use of throw statement
    throw new IllegalStateException();
  case 5:                                // Use of continue statement
    continue;
  default:                               // For the last case, use of break statement is optional
    doSomethingElse();
}
```

"switch" statements should have "default" clauses (java:S131)

Severidad: CRITICAL

Root_cause:

The requirement for a final `default` clause is defensive programming. The clause should either take appropriate action, or contain a suitable comment as to why no action is taken.

Noncompliant code example

```
switch (param) {  //missing default clause
  case 0:
    doSomething();
    break;
  case 1:
    doSomethingElse();
    break;
}

switch (param) {
  default: // default clause should be the last one
    error();
    break;
  case 0:
    doSomething();
    break;
  case 1:
    doSomethingElse();
    break;
}
```

Compliant solution

```
switch (param) {
  case 0:
    doSomething();
    break;
  case 1:
    doSomethingElse();
    break;
  default:
    error();
    break;
}
```

Exceptions

If the `switch` parameter is an `Enum` and if all the constants of this enum are used in the `case` statements, then no `default` clause is expected.

Example:

```
public enum Day {
    SUNDAY, MONDAY
}
...
switch(day) {
  case SUNDAY:
    doSomething();
    break;
  case MONDAY:
    doSomethingElse();
    break;
}
```

Resources:

- CWE - CWE-478 - Missing Default Case in Switch Statement
- CERT, MSC01-C. - Strive for logical completeness

Control flow statements "if", "for", "while", "switch" and "try" should not be nested too deeply (java:S134)

Severidad: CRITICAL

Resources:

- Guard clauses in programming - one of the approaches to reducing the depth of nesting

How_to_fix:

The following example demonstrates the behavior of the rule with the default threshold of 3 levels of nesting and one of the potential ways to fix the code smell by introducing guard clauses:

Noncompliant code example

```
if (condition1) {                  // Compliant - depth = 1
  /* ... */
  if (condition2) {                // Compliant - depth = 2
    /* ... */
    for (int i = 0; i < 10; i++) {  // Compliant - depth = 3
      /* ... */
      if (condition4) {            // Noncompliant - depth = 4, which exceeds the limit
        if (condition5) {          // Depth = 5, exceeding the limit, but issues are only reported on depth = 4
          /* ... */
        }
        return;
      }
    }
  }
}
```

Compliant solution

```
if (!condition1) {
  return;
}
/* ... */
if (!condition2) {
  return;
}
for (int i = 0; i < 10; i++) {
  /* ... */
  if (condition4) {
    if (condition5) {
      /* ... */
    }
    return;
  }
}
```

Root_cause:

Nested control flow statements such as `if`, `for`, `while`, `switch`, and `try` are often key ingredients in creating what's known as "Spaghetti code". This code smell can make your program difficult to understand and maintain.

When numerous control structures are placed inside one another, the code becomes a tangled, complex web. This significantly reduces the code's readability and maintainability, and it also complicates the testing process.

Anonymous inner classes containing only one method should become lambdas (java:S1604)

Severidad: MAJOR

Resources:

- Lambda Expressions - The Java™ Tutorials

Root_cause:

Before Java 8, the only way to partially support closures in Java was by using anonymous inner classes. Java 8 introduced lambdas, which are significantly more readable and should be used instead.

This rule is automatically disabled when the project's `sonar.java.source` is lower than 8, as lambda expressions were introduced in Java 8.

Noncompliant code example

```
myCollection.stream().map(new Function<String,String>() { // Noncompliant, use a lambda expression instead
  @Override
  public String apply(String input) {
    return new StringBuilder(input).reverse().toString();
  }
})
  ...
```

Compliant solution

```
myCollection.stream()
  .map(input -> new StringBuilder(input).reverse().toString()) // Compliant
    ...
```

Noncompliant code example

```
Predicate<String> isEmpty = new Predicate<String>() { // Noncompliant, use a lambda expression instead
  @Override
  public boolean test(String myString) {
    return myString.isEmpty();
  }
};
```

Compliant solution

```
Predicate<String> isEmpty = myString -> myString.isEmpty(); // Compliant
```

JUnit4 @Ignored and JUnit5 @Disabled annotations should be used to disable tests and should provide a rationale (java:S1607)

Severidad: MAJOR

Root_cause:

When a test fails due, for example, to infrastructure issues, you might want to ignore it temporarily. But without some kind of notation about why the test is being ignored, it may never be reactivated. Such tests are difficult to address without comprehensive knowledge of the project, and end up polluting their projects.

This rule raises an issue for each ignored test that does not have any comment about why it is being skipped.

- For Junit4, this rule targets the @Ignore annotation.
- For Junit5, this rule targets the @Disabled annotation.
- Cases where assumeTrue(false) or assumeFalse(true) are used to skip tests are targeted as well.

Noncompliant code example

```
@Ignore  // Noncompliant
@Test
public void testDoTheThing() {
  // ...
```

or

```
@Test
public void testDoTheThing() {
  Assume.assumeFalse(true); // Noncompliant
  // ...
```

Compliant solution

```
@Test
@Ignore("See Ticket #1234")
public void testDoTheThing() {
  // ...
```

Annotation repetitions should not be wrapped (java:S1710)

Severidad: MINOR

Root_cause:

Before Java 8, a container annotation was required as wrapper to use multiple instances of the same annotation. As of Java 8, this is no longer necessary. Instead, these annotations should be used directly without a wrapper, resulting in cleaner and more readable code.

This rule is automatically disabled when the project’s `sonar.java.source` is lower than 8 as repeating annotations were introduced in Java 8.

Noncompliant code example

```
@SomeAnnotations({  // Noncompliant, wrapper annotations are not necessary in Java 8+
  @SomeAnnotation(..a..),
  @SomeAnnotation(..b..),
  @SomeAnnotation(..c..),
})
public class SomeClass {
  ...
}
```

Compliant solution

```
@SomeAnnotation(..a..)
@SomeAnnotation(..b..)
@SomeAnnotation(..c..)
public class SomeClass {
  ...
}
```

References

- Repeating Annotations - The Java™ Tutorials

Standard functional interfaces should not be redefined (java:S1711)

Severidad: MAJOR

Root_cause:

Just as there is little justification for writing your own String class, there is no good reason to re-define one of the existing, standard functional interfaces.

Doing so may seem tempting, since it would allow you to specify a little extra context with the name. But in the long run, it will be a source of confusion, because maintenance programmers will wonder what is different between the custom functional interface and the standard one.

Noncompliant code example

```
@FunctionalInterface
public interface MyInterface { // Noncompliant
        double toDouble(int a);
}

@FunctionalInterface
public interface ExtendedBooleanSupplier { // Noncompliant
  boolean get();
  default boolean isFalse() {
    return !get();
  }
}

public class MyClass {
    private int a;
    public double myMethod(MyInterface instance){
        return instance.toDouble(a);
    }
}
```

Compliant solution

```
@FunctionalInterface
public interface ExtendedBooleanSupplier extends BooleanSupplier { // Compliant, extends java.util.function.BooleanSupplier
  default boolean isFalse() {
    return !getAsBoolean();
  }
}

public class MyClass {
    private int a;
    public double myMethod(IntToDoubleFunction instance){
        return instance.applyAsDouble(a);
    }
}
```

"Object.wait" should not be called on objects that implement "java.util.concurrent.locks.Condition" (java:S1844)

Severidad: MAJOR

Root_cause:

The `java.util.concurrent.locks.Condition` interface provides an alternative to the `Object` monitor methods (`wait`, `notify` and `notifyAll`). Hence, the purpose of implementing said interface is to gain access to its more nuanced `await` methods.

Consequently, calling the method `Object.wait` on a class implementing the `Condition` interface is contradictory and should be avoided. Use `Condition.await` instead.

Code examples

Noncompliant code example

```
void doSomething(Condition condition) {
    condition.wait(); // Noncompliant, Object.wait is called

        ...
}
```

Compliant solution

```
void doSomething(Condition condition) {
    condition.await(); // Compliant, Condition.await is called

        ...
}
```

References

- Interface Condition - Java™ Platform SE 8, API Specification

Methods and field names should not be the same or differ only by capitalization (java:S1845)

Severidad: BLOCKER

Introduction:

This rule raises an issue when there is a method and a field in a class with names that differ only by capitalization.

Root_cause:

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(){...}

}
```

"Iterator.hasNext()" should not call "Iterator.next()" (java:S1849)

Severidad: MAJOR

Resources:

- Interface Iterator<E> - Java® Platform SE 11 API Specification

How_to_fix:

How to fix this issue strongly depends on the specific implementation of the iterator. Make sure that the logic of the `hasNext()` implementation does not change the state of the iterator or any underlying data sources. Instead, it should merely return state information.

Code examples

Noncompliant code example

```
class MyIterator implements Iterator<Integer> {

  private Queue<Integer> elements;

    ...

  @Override
  public boolean hasNext() {
    try {
      next(); // Noncompliant, next() is called from hasNext()
      return true;
    } catch (NoSuchElementException e) {
      return false;
    }
  }

  @Override
  public Integer next() {
    return elements.remove();
  }
}
```

Compliant solution

```
class MyIterator implements Iterator<Integer> {

  private Queue<Integer> elements;

    ...

  @Override
  public boolean hasNext() {
    return !elements.isEmpty(); // Compliant, no call to next()
  }

  @Override
  public Integer next() {
    return elements.remove();
  }
}
```

Root_cause:

Calling `Iterator.hasNext()` is not supposed to have any side effects and hence should not change the iterator's state. `Iterator.next()` advances the iterator by one item. So calling it inside `Iterator.hasNext()` breaks the `hasNext()` contract and will lead to unexpected behavior in production.

JUnit rules should be used (java:S2924)

Severidad: MINOR

Root_cause:

JUnit rules are predefined classes that extend the behavior of JUnit tests, allowing to add new functionalities, such as managing resources, modifying test behavior, and handling exceptions.

Unused JUnit rules can lead to confusion when reading the test code, making tests harder to understand and maintain. Having unused rules can also slow down the test suite, as JUnit has to process the rules even though they are not being used. Some `TestRule` classes have the desired effect without being directly referenced by a test, while others do not. There's no reason to leave them cluttering the file if they're not in use.

The rule raises an issue when in a `Test` class, there is no method referencing a declared `TestRule` of the following types:

- `TemporaryFolder` and `TestName` in JUnit
- `TempDir` and `TestInfo` in JUnit 5

Resources:

Documentation

- Junit API - Rule
- Junit - Rules

Articles & blog posts

- Junit 4 Rules

How_to_fix:

Remove the unused `TestRule` field that is expected to be referenced inside a test method.

Noncompliant code example

```
public class ProjectDefinitionTest {

  @Rule
  public TemporaryFolder temp = new TemporaryFolder();  // Noncompliant

  @Test
  public void shouldSetKey() {
    ProjectDefinition def = ProjectDefinition.create();
    def.setKey("mykey");
    assertThat(def.getKey(), is("mykey"));
  }
}
```

Compliant solution

```
public class ProjectDefinitionTest {

  @Test
  public void shouldSetKey() {
    ProjectDefinition def = ProjectDefinition.create();
    def.setKey("mykey");
    assertThat(def.getKey(), is("mykey"));
  }
}
```

"Thread.sleep" should not be used in tests (java:S2925)

Severidad: MAJOR

Root_cause:

In asynchronous testing, the test code is written in a way that allows it to wait for the asynchronous operation to complete before continuing with the test.

Using `Thread.sleep` in this case can cause flaky tests, slow test execution, and inaccurate test results. It creates brittle tests that can fail unpredictably depending on the environment or load.

Use mocks or libraries such as `Awaitility` instead. These tools provide features such as timeouts, assertions, and error handling to make it easier to write and manage asynchronous tests.

Noncompliant code example

```
@Test
public void testDoTheThing(){

  MyClass myClass = new MyClass();
  myClass.doTheThing();

  Thread.sleep(500);  // Noncompliant
  // assertions...
}
```

Compliant solution

```
@Test
public void testDoTheThing(){

  MyClass myClass = new MyClass();
  myClass.doTheThing();

  await().atMost(2, Duration.SECONDS).until(didTheThing());  // Compliant
  // assertions...
}

private Callable<Boolean> didTheThing() {
  return new Callable<Boolean>() {
    public Boolean call() throws Exception {
      // check the condition that must be fulfilled...
    }
  };
}
```

Resources:

Documentation

- [Oracle SE 20 - Thread]()
- [Awaitility]()

Articles & blog posts

- [Baeldung - Thread.sleep() vs Awaitility.await()]()
- [Baeldung - Awaitility testing]()

All branches in a conditional structure should not have exactly the same implementation (java:S3923)

Severidad: MAJOR

Root_cause:

Having all branches of a `switch` or `if` chain with the same implementation indicates a problem.

In the following code:

```
if (b == 0) {  // Noncompliant
  doOneMoreThing();
} else {
  doOneMoreThing();
}

int b = a > 12 ? 4 : 4;  // Noncompliant

switch (i) {  // Noncompliant
  case 1:
    doSomething();
    break;
  case 2:
    doSomething();
    break;
  case 3:
    doSomething();
    break;
  default:
    doSomething();
}
```

Either there is a copy-paste error that needs fixing or an unnecessary `switch` or `if` chain that should be removed.

Exceptions

This rule does not apply to `if` chains without `else`, nor to `switch` without a `default` clause.

```
if(b == 0) {    //no issue, this could have been done on purpose to make the code more readable
  doSomething();
} else if(b == 1) {
  doSomething();
}
```

Use Java 14 "switch" expression (java:S5194)

Severidad: MINOR

Root_cause:

Many existing switch statements are essentially simulations of switch expressions, where each arm either assigns to a common target variable or returns a value. Expressing this as a statement is roundabout, repetitive, and error-prone.

Java 14 added support for switch expressions, which provide more succinct and less error-prone version of switch.

Noncompliant code example

```
void day_of_week(DoW day) {
    int numLetters;
    switch (day) {  // Noncompliant
      case MONDAY:
      case FRIDAY:
      case SUNDAY:
        numLetters = 6;
        break;
      case TUESDAY:
        numLetters = 7;
        break;
      case THURSDAY:
      case SATURDAY:
        numLetters = 8;
        break;
      case WEDNESDAY:
        numLetters = 9;
        break;
      default:
        throw new IllegalStateException("Wat: " + day);
    }
}

int return_switch(int x) {
    switch (x) { // Noncompliant
      case 1:
      return 1;
      case 2:
      return 2;
      default:
        throw new IllegalStateException();
    }
}
```

Compliant solution

```
int numLetters = switch (day) {
    case MONDAY, FRIDAY, SUNDAY -> 6;
    case TUESDAY                -> 7;
    case THURSDAY, SATURDAY     -> 8;
    case WEDNESDAY              -> 9;
};
```

The regex escape sequence \cX should only be used with characters in the @-_ range (java:S6070)

Severidad: MAJOR

Root_cause:

In regular expressions the escape sequence `\cX`, where the X stands for any character that's either `@`, any capital ASCII letter, `[`, `\`, `]`, `^` or `_`, represents the control character that "corresponds" to the character following `\c`, meaning the control character that comes 64 bytes before the given character in the ASCII encoding.

In some other regex engines (for example in that of Perl) this escape sequence is case insensitive and `\cd` produces the same control character as `\cD`. Further using `\c` with a character that's neither `@`, any ASCII letter, `[`, `\`, `]`, `^` nor `_`, will produce a warning or error in those engines. Neither of these things is true in Java, where the value of the character is always XORed with 64 without checking that this operation makes sense. Since this won't lead to a sensible result for characters that are outside of the `@` to `_` range, using `\c` with such characters is almost certainly a mistake.

Noncompliant code example

```
Pattern.compile("\\ca"); // Noncompliant, 'a' is not an upper case letter
Pattern.compile("\\c!"); // Noncompliant, '!' is outside of the '@'-'_' range
```

Compliant solution

```
Pattern.compile("\\cA"); // Compliant, this will match the "start of heading" control character
Pattern.compile("\\c^"); // Compliant, this will match the "record separator" control character
```

Mockito argument matchers should be used on all parameters (java:S6073)

Severidad: MAJOR

Resources:

- Mockito documentation - argument matchers
- S6068 - Call to Mockito method "verify", "when" or "given" should be simplified

Root_cause:

Mockito provides *argument matchers* and *argument captors* for flexibly stubbing or verifying method calls.

`Mockito.verify()`, `Mockito.when()`, `Stubber.when()` and `BDDMockito.given()` each have overloads with and without argument matchers.

However, if argument matchers or captors are used only on some of the parameters, all the parameters need to have matchers as well, otherwise an `InvalidUseOfMatchersException` will be thrown.

This rule consequently raises an issue every time matchers are not used on all the parameters of a stubbed/verified method.

How_to_fix:

Noncompliant code example

```
@Test
public void myTest() {
    // Setting up mock responses
    given(foo.bar(anyInt(), i1, i2)).willReturn(null); // Noncompliant, no matchers for "i1" and "i2"
    when(foo.baz(eq(val1), val2)).thenReturn("hi"); // Noncompliant, no matcher for "val2"

    // Simulating exceptions
    doThrow(new RuntimeException()).when(foo).quux(intThat(x -> x >= 42), -1); // Noncompliant, no matcher for "-1"

    // Verifying method invocations
    verify(foo).bar(i1, anyInt(), i2); // Noncompliant, no matchers for "i1" and "i2"

    // Capturing arguments for verification
    ArgumentCaptor<Integer> captor = ArgumentCaptor.forClass(Integer.class);
    verify(foo).bar(captor.capture(), i1, any()); // Noncompliant, no matchers for "i1"
}
```

Compliant solution

```
@Test
public void myTest() {
    // Setting up mock responses
    given(foo.bar(anyInt(), eq(i1), eq(i2))).willReturn(null); // Compliant, all arguments have matchers
    when(foo.baz(val1, val2)).thenReturn("hi"); // Compliant, no argument has matchers

    // Simulating exceptions
    doThrow(new RuntimeException()).when(foo).quux(intThat(x -> x >= 42), eq(-1)); // Compliant, all arguments have matchers

    // Verifying method invocations
    verify(foo).bar(eq(i1), anyInt(), eq(i2)); // Compliant, all arguments have matchers

    // Capturing arguments for verification
    ArgumentCaptor<Integer> captor = ArgumentCaptor.forClass(Integer.class);
    verify(foo).bar(captor.capture(), any(), any()); // Compliant, all arguments have matchers
}
```

Authorizing non-authenticated users to use keys in the Android KeyStore is security-sensitive (java:S6288)

Severidad: MAJOR

How_to_fix:

Recommended Secure Coding Practices

It’s recommended to enable user authentication (by setting `setUserAuthenticationRequired` to `true` during key generation) to use keys for a limited duration of time (by setting appropriate values to `setUserAuthenticationValidityDurationSeconds`), after which the user must re-authenticate.

Compliant Solution

The use of the key is limited to authenticated users (for a duration of time defined to 60 seconds):

```
KeyGenerator keyGenerator = KeyGenerator.getInstance(KeyProperties.KEY_ALGORITHM_AES, "AndroidKeyStore");

KeyGenParameterSpec builder = new KeyGenParameterSpec.Builder("test_secret_key", KeyProperties.PURPOSE_ENCRYPT | KeyProperties.PURPOSE_DECR
    .setBlockModes(KeyProperties.BLOCK_MODE_GCM)
    .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_NONE)
    .setUserAuthenticationRequired(true)
    .setUserAuthenticationParameters (60, KeyProperties.AUTH_DEVICE_CREDENTIAL)
    .build();

keyGenerator.init(builder)
```

See

- OWASP - [Top 10 2021 Category A4 - Insecure Design]

- developer.android.com - Android keystore system
- developer.android.com - Require user authentication for key use
- OWASP - Mobile AppSec Verification Standard - Authentication and Session Management Requirements
- OWASP - Mobile Top 10 2016 Category M4 - Insecure Authentication
- CWE - CWE-522 - Insufficiently Protected Credentials

Default:

Android KeyStore is a secure container for storing key materials, in particular it prevents key materials extraction, i.e. when the application process is compromised, the attacker cannot extract keys but may still be able to use them. It’s possible to enable an Android security feature, user authentication, to restrict usage of keys to only authenticated users. The lock screen has to be unlocked with defined credentials (pattern/PIN/password, biometric).

Ask Yourself Whether

- The application requires prohibiting the use of keys in case of compromise of the application process.
- The key material is used in the context of a highly sensitive application like a e-banking mobile app.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

It’s recommended to enable user authentication (by setting `setUserAuthenticationRequired` to `true` during key generation) to use keys for a limited duration of time (by setting appropriate values to `setUserAuthenticationValidityDurationSeconds`), after which the user must re-authenticate.

Sensitive Code Example

Any user can use the key:

```
KeyGenerator keyGenerator = KeyGenerator.getInstance(KeyProperties.KEY_ALGORITHM_AES, "AndroidKeyStore");

KeyGenParameterSpec builder = new KeyGenParameterSpec.Builder("test_secret_key_noncompliant", KeyProperties.PURPOSE_ENCRYPT | KeyProperties
    .setBlockModes(KeyProperties.BLOCK_MODE_GCM)
    .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_NONE)
    .build();

keyGenerator.init(builder);
```

Compliant Solution

The use of the key is limited to authenticated users (for a duration of time defined to 60 seconds):

```
KeyGenerator keyGenerator = KeyGenerator.getInstance(KeyProperties.KEY_ALGORITHM_AES, "AndroidKeyStore");

KeyGenParameterSpec builder = new KeyGenParameterSpec.Builder("test_secret_key", KeyProperties.PURPOSE_ENCRYPT | KeyProperties.PURPOSE_DECR
    .setBlockModes(KeyProperties.BLOCK_MODE_GCM)
    .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_NONE)
    .setUserAuthenticationRequired(true)
    .setUserAuthenticationParameters (60, KeyProperties.AUTH_DEVICE_CREDENTIAL)
    .build();

keyGenerator.init(builder)
```

See

- OWASP - Top 10 2021 Category A4 - Insecure Design
- developer.android.com - Android keystore system
- developer.android.com - Require user authentication for key use
- OWASP - Mobile AppSec Verification Standard - Authentication and Session Management Requirements
- OWASP - Mobile Top 10 2016 Category M4 - Insecure Authentication
- CWE - CWE-522 - Insufficiently Protected Credentials

Root_cause:

Android KeyStore is a secure container for storing key materials, in particular it prevents key materials extraction, i.e. when the application process is compromised, the attacker cannot extract keys but may still be able to use them. It’s possible to enable an Android security feature, user authentication, to restrict usage of keys to only authenticated users. The lock screen has to be unlocked with defined credentials (pattern/PIN/password, biometric).

Assess_the_problem:

Ask Yourself Whether

- The application requires prohibiting the use of keys in case of compromise of the application process.
- The key material is used in the context of a highly sensitive application like a e-banking mobile app.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

Any user can use the key:

```
KeyGenerator keyGenerator = KeyGenerator.getInstance(KeyProperties.KEY_ALGORITHM_AES, "AndroidKeyStore");

KeyGenParameterSpec builder = new KeyGenParameterSpec.Builder("test_secret_key_noncompliant", KeyProperties.PURPOSE_ENCRYPT | KeyProperties
    .setBlockModes(KeyProperties.BLOCK_MODE_GCM)
    .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_NONE)
    .build();

keyGenerator.init(builder);
```

Inheritance tree of classes should not be too deep (java:S110)

Severidad: MAJOR

Resources:

Documentation

Composition over inheritance: difference between composition and inheritance in object-oriented programming

Root_cause:

Inheritance is one of the most valuable concepts in object-oriented programming. It’s a way to categorize and reuse code by creating collections of attributes and behaviors called classes, which can be based on previously created classes.

But abusing this concept by creating a deep inheritance tree can lead to complex and unmaintainable source code. Often, an inheritance tree becoming too deep is the symptom of systematic use of "inheritance" when other approaches like "composition" would be better suited.

This rule raises an issue when the inheritance tree, starting from `Object`, has a greater depth than is allowed.

The rule has one parameter to filter out classes of the count of inheritance. The following rules apply to define this parameter:

- `?` matches a single character
- `*` matches zero or more characters
- `**` matches zero or more packages

Examples:

- `java.fwk.AbstractFwkClass`: the count stops when AbstractFwkClass class is reached.
- `java.fwk.*`: any member of java.fwkPackage package is reached.
- `java.fwk.**`: same as above, but including sub-packages.

Exceptions:

The rule stops counting when it encounters a class from one of the following packages (or sub-packages):

- `android.**`
- `com.intellij.**`
- `com.persistit.**`
- `javax.swing.**`
- `org.eclipse.**`
- `org.springframework.**`

Loops should not contain more than a single "break" or "continue" statement (java:S135)

Severidad: MINOR

Root_cause:

The use of `break` and `continue` statements increases the complexity of the control flow and makes it harder to understand the program logic. In order to keep a good program structure, they should not be applied more than once per loop.

This rule reports an issue when there is more than one `break` or `continue` statement in a loop. The code should be refactored to increase readability if there is more than one.

Noncompliant code example

```
for (int i = 1; i <= 10; i++) {     // Noncompliant; two "continue" statements
  if (i % 2 == 0) {
    continue;
  }

  if (i % 3 == 0) {
    continue;
  }
  // ...
}
```

Compliant solution

```
for (int i = 1; i <= 10; i++) {
  if (i % 2 == 0 || i % 3 == 0) {
    continue;
  }
  // ...
}
```

Resources:

Documentation

- Oracle - Labeled Statements

Articles & blog posts

- StackExchange - Java labels. To be or not to be
- StackOverflow - Labels in Java - bad practice?

Methods should not have too many lines (java:S138)

Severidad: MAJOR

Root_cause:

A method that grows too large tends to aggregate too many responsibilities. Such method inevitably become harder to understand and therefore harder to maintain.

Above a specific threshold, it is strongly advised to refactor into smaller methods which focus on well-defined tasks. Those smaller methods will not only be easier to understand, but also probably easier to test.

Comments should not be located at the end of lines of code (java:S139)

Severidad: MINOR

Root_cause:

This rule verifies that single-line comments are not located at the ends of lines of code. The main idea behind this rule is that in order to be really readable, trailing comments would have to be properly written and formatted (correct alignment, no interference with the visual structure of the code, not too long to be visible) but most often, automatic code formatters would not handle this correctly: the code would end up less readable. Comments are far better placed on the previous empty line of code, where they will always be visible and properly formatted.

Noncompliant code example

```
int a1 = b + c; // This is a trailing comment that can be very very long
```

Compliant solution

```
// This very long comment is better placed before the line of code
int a2 = b + c;
```

Lambdas containing only one statement should not nest this statement in a block (java:S1602)

Severidad: MINOR

Root_cause:

The right-hand side of a lambda expression can be written in two ways:

1. Expression notation: the right-hand side is as an expression, such as in `(a, b) → a + b`
2. Block notation: the right-hand side is a conventional function body with a code block and an optional return statement, such as in `(a, b) → {return a + b;}`

By convention, expression notation is preferred over block notation. Block notation must be used when the function implementation requires more than one statement. However, when the code block consists of only one statement (which may or may not be a `return` statement), it can be rewritten using expression notation.

This convention exists because expression notation has a cleaner, more concise, functional programming style and is regarded as more readable.

How_to_fix:

- If the code block consists only of a `return` statement, replace the code block with the argument expression from the `return` statement.
- If the code block consists of a single statement that is not a `return` statement, replace the code block with that statement.

Noncompliant code example

```
(a, b) -> { return a + b; } // Noncompliant, replace code block with expression
```

Compliant solution

```
(a, b) -> a + b             // Compliant
```

Noncompliant code example

```
x -> {System.out.println(x+1);} // Noncompliant, replace code block with statement
```

Compliant solution

```
x -> System.out.println(x+1)    // Compliant
```

Introduction:

This rule raises an issue when a lambda expression uses block notation while expression notation could be used.

JUnit test cases should call super methods (java:S2188)

Severidad: BLOCKER

Root_cause:

Overriding a parent class method prevents that method from being called unless an explicit `super` call is made in the overriding method. In some cases, not calling the parent method is fine. However, `setUp` and `tearDown` provide some shared logic that is called before all test cases. This logic may change over the lifetime of your codebase. To make sure that your test cases are set up and cleaned up consistently, your overriding implementations of `setUp` and `tearDown` should call the parent implementations explicitly.

How_to_fix:

Add an explicit call to `super.setUp()` and `super.tearDown()` in the overriding methods.

Noncompliant code example

```
public class MyClassTest extends MyAbstractTestCase {

  private MyClass myClass;

  @Override
  protected void setUp() throws Exception {  // Noncompliant
    myClass = new MyClass();
  }
}
```

Compliant solution

```
public class MyClassTest extends MyAbstractTestCase {
  private MyClass myClass;

  @Override
  protected void setUp() throws Exception {
    super.setUp();
    myClass = new MyClass();
  }
}
```

Loops should not be infinite (java:S2189)

Severidad: BLOCKER

Resources:

Standards

- CERT, MSC01-J. - Do not use an empty infinite loop

Root_cause:

An infinite loop will never end while the program runs, meaning you have to kill the program to get out of the loop. Every loop should have an end condition, whether by meeting the loop’s termination condition or via a `break` statement.

Noncompliant code example

```
for (;;) {  // Noncompliant; end condition omitted
  // ...
}

int j;
while (true) { // Noncompliant; end condition omitted
  j++;
}

int k;
boolean b = true;
while (b) { // Noncompliant; b never written to in loop
  k++;
}
```

Compliant solution

```
int j;
while (true) { // reachable end condition added
  j++;
  if (j  == Integer.MIN_VALUE) {  // true at Integer.MAX_VALUE +1
    break;
  }
}

int k;
boolean b = true;
while (b) {
  k++;
  b = k < Integer.MAX_VALUE;
}
```

Abstract methods should not be redundant (java:S3038)

Severidad: MINOR

Root_cause:

There’s no point in redundantly defining an `abstract` method with the same signature as a method in an `interface` that the class `implements`. Any concrete child classes will have to implement the method either way.

Noncompliant code example

```
public interface Reportable {
  String getReport();
}

public abstract class AbstractRuleReport implements Reportable{
  public abstract String getReport();  // Noncompliant
  // ...
}
```

Indexes to passed to "String" operations should be within the string's bounds (java:S3039)

Severidad: MAJOR

Root_cause:

There are various `String` operations that take one or more character indexes as arguments and return a portion of the original string. Indexing in this context is zero-based, meaning that the first character's index is `0`. As a result, given a string `myString`, its last character is at index `myString.length() - 1`.

The `String` operation methods throw a `StringIndexOutOfBoundsException` when one of their index argument is smaller than 0 (E.G.: -1). `String::substring` also throws this exception when the `beginIndex` or `endIndex` argument is larger than `myString.length()`, and `String::charAt` when the `index` argument is larger than `myString.length() - 1` For instance, it is not possible to use `String::charAt` to retrieve a value before the start or after the end of a string. Furthermore, it is not possible to use `String::substring` with `beginIndex > endIndex` to reverse the order of characters in a string.

This rule raises an issue when a negative literal or an index that is too large is passed as an argument to the `String::substring`, `String::charAt`, and related methods. It also raises an issue when the start index passed to `String::substring` is larger than the end index.

How_to_fix:

Use non-negative indexes that are smaller than or equal to the length of the string in question with `String::substring` and strictly smaller with `String::charAt`.

Noncompliant code example

```
String speech = "Lorem ipsum dolor sit amet";

String substr1 = speech.substring(-1, speech.length()); // Noncompliant, -1 is out of bounds
String substr2 = speech.substring(speech.length(), 0);  // Noncompliant, the beginIndex must be smaller than or equal to the endIndex
char ch = speech.charAt(speech.length());               // Noncompliant, speech.length() is out of bounds
```

Compliant solution

```
String speech = "Lorem ipsum dolor sit amet";

String substr1 = speech;                                          // Compliant, no string operation used
String substr2 = new StringBuilder(speech).reverse().toString(); // Compliant, the string can be reversed using StringBuilder::reverse()
char ch = speech.charAt(speech.length() - 1);                     // Compliant, speech.length() - 1 is in bounds.
```

"private" methods called only by inner classes should be moved to those classes (java:S3398)

Severidad: MINOR

Root_cause:

When a `private` method is only invoked by an inner class, there's no reason not to move it into that class. It will still have the same access to the outer class' members, but the outer class will be clearer and less cluttered.

Noncompliant code example

```
public class Outie {
  private int i=0;

  private void increment() {  // Noncompliant
    i++;
  }

  public class Innie {
    public void doTheThing() {
      Outie.this.increment();
    }
  }
}
```

Compliant solution

```
public class Outie {
  private int i=0;

  public class Innie {
    public void doTheThing() {
      increment();
    }

    private void increment() {
      Outie.this.i++;
    }
  }
}
```

Regex patterns should not be created needlessly (java:S4248)

Severidad: MAJOR

Root_cause:

The `java.util.regex.Pattern.compile()` methods have a significant performance cost, and therefore should be used sensibly.

Moreover they are the only mechanism available to create instances of the Pattern class, which are necessary to do any pattern matching using regular expressions. Unfortunately that can be hidden behind convenience methods like `String.matches()` or `String.split()`.

It is therefore somewhat easy to inadvertently repeatedly compile the same regular expression at great performance cost with no valid reason.

This rule raises an issue when:

- A `Pattern` is compiled from a `String` literal or constant and is not stored in a static final reference.
- `String.matches`, `String.split`, `String.replaceAll` or `String.replaceFirst` are invoked with a `String` literal or constant. In which case the code should be refactored to use a `java.util.regex.Pattern` while respecting the previous rule.

Noncompliant code example

```
public void doingSomething(String stringToMatch) {
  Pattern regex = Pattern.compile("myRegex");  // Noncompliant
  Matcher matcher = regex.matcher("s");
  // ...
  if (stringToMatch.matches("myRegex2")) {  // Noncompliant
    // ...
  }
}
```

Compliant solution

```
private static final Pattern myRegex = Pattern.compile("myRegex");
private static final Pattern myRegex2 = Pattern.compile("myRegex2");

public void doingSomething(String stringToMatch) {
  Matcher matcher = myRegex.matcher("s");
  // ...
  if (myRegex2.matcher(stringToMatch).matches()) {
    // ...
  }
}
```

Exceptions

`String.split` doesn’t create a regex when the string passed as argument meets either of these 2 conditions:

- It is a one-char String and this character is not one of the RegEx’s meta characters ".$|()[{^?*+\"
- It is a two-char String and the first char is the backslash and the second is not the ascii digit or ascii letter.

In which case no issue will be raised.

Using unencrypted databases in mobile applications is security-sensitive (java:S6291)

Severidad: MAJOR

Assess_the_problem:

Ask Yourself Whether

- The database contains sensitive data that could cause harm when leaked.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

For SQLiteDatabase:

```
SQLiteDatabase db = activity.openOrCreateDatabase("test.db", Context.MODE_PRIVATE, null); // Sensitive
```

For SharedPreferences:

```
SharedPreferences pref = activity.getPreferences(Context.MODE_PRIVATE); // Sensitive
```

For Realm:

```
RealmConfiguration config = new RealmConfiguration.Builder().build();
Realm realm = Realm.getInstance(config); // Sensitive
```

How_to_fix:

Recommended Secure Coding Practices

It's recommended to password-encrypt local databases that contain sensitive information. Most systems provide secure alternatives to plain-text storage that should be used. If no secure alternative is available the data can also be encrypted manually before it is stored.

The encryption password should not be hard-coded in the application. There are different approaches how the password can be provided to encrypt and decrypt the database. In the case of `EncryptedSharedPreferences` the Android Keystore can be used to store the password. Other databases can rely on `EncryptedSharedPreferences` to store passwords. The password can also be provided dynamically by the user of the application or it can be fetched from a remote server if the other methods are not feasible.

Compliant Solution

Instead of SQLiteDatabase you can use SQLCipher:

```
SQLiteDatabase db = SQLiteDatabase.openOrCreateDatabase("test.db", getKey(), null);
```

Instead of SharedPreferences you can use EncryptedSharedPreferences:

```
String masterKeyAlias = new MasterKeys.getOrCreate(MasterKeys.AES256_GCM_SPEC);
EncryptedSharedPreferences.create(
    "secret",
    masterKeyAlias,
    context,
    EncryptedSharedPreferences.PrefKeyEncryptionScheme.AES256_SIV,
    EncryptedSharedPreferences.PrefValueEncryptionScheme.AES256_GCM
);
```

For Realm an encryption key can be specified in the config:

```
RealmConfiguration config = new RealmConfiguration.Builder()
    .encryptionKey(getKey())
    .build();
Realm realm = Realm.getInstance(config);
```

See

- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- OWASP - Top 10 2021 Category A4 - Insecure Design
- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Mobile AppSec Verification Standard - Data Storage and Privacy Requirements
- Mobile Top 10 2016 Category M2 - Insecure Data Storage
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- CWE - CWE-311 - Missing Encryption of Sensitive Data

Root_cause:

Storing data locally is a common task for mobile applications. Such data includes preferences or authentication tokens for external services, among other things. There are many convenient solutions that allow storing data persistently, for example SQLiteDatabase, SharedPreferences, and Realm. By default these systems store the data unencrypted, thus an attacker with physical access to the device can read them out easily. Access to sensitive data can be harmful for the user of the application, for example when the device gets stolen.

Default:

Storing data locally is a common task for mobile applications. Such data includes preferences or authentication tokens for external services, among other things. There are many convenient solutions that allow storing data persistently, for example SQLiteDatabase, SharedPreferences, and Realm. By default these systems store the data unencrypted, thus an attacker with physical access to the device can read them out easily. Access to sensitive data can be harmful for the user of the application, for example when the device gets stolen.

Ask Yourself Whether

- The database contains sensitive data that could cause harm when leaked.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

It’s recommended to password-encrypt local databases that contain sensitive information. Most systems provide secure alternatives to plain-text storage that should be used. If no secure alternative is available the data can also be encrypted manually before it is stored.

The encryption password should not be hard-coded in the application. There are different approaches how the password can be provided to encrypt and decrypt the database. In the case of `EncryptedSharedPreferences` the Android Keystore can be used to store the password. Other databases can rely on `EncryptedSharedPreferences` to store passwords. The password can also be provided dynamically by the user of the application or it can be fetched from a remote server if the other methods are not feasible.

Sensitive Code Example

For SQLiteDatabase:

```
SQLiteDatabase db = activity.openOrCreateDatabase("test.db", Context.MODE_PRIVATE, null); // Sensitive
```

For SharedPreferences:

```
SharedPreferences pref = activity.getPreferences(Context.MODE_PRIVATE); // Sensitive
```

For Realm:

```
RealmConfiguration config = new RealmConfiguration.Builder().build();
Realm realm = Realm.getInstance(config); // Sensitive
```

Compliant Solution

Instead of SQLiteDatabase you can use SQLCipher:

```
SQLiteDatabase db = SQLiteDatabase.openOrCreateDatabase("test.db", getKey(), null);
```

Instead of SharedPreferences you can use EncryptedSharedPreferences:

```
String masterKeyAlias = new MasterKeys.getOrCreate(MasterKeys.AES256_GCM_SPEC);
EncryptedSharedPreferences.create(
    "secret",
    masterKeyAlias,
    context,
    EncryptedSharedPreferences.PrefKeyEncryptionScheme.AES256_SIV,
    EncryptedSharedPreferences.PrefValueEncryptionScheme.AES256_GCM
);
```

For Realm an encryption key can be specified in the config:

```
RealmConfiguration config = new RealmConfiguration.Builder()
    .encryptionKey(getKey())
    .build();
Realm realm = Realm.getInstance(config);
```

See

- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- OWASP - Top 10 2021 Category A4 - Insecure Design
- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Mobile AppSec Verification Standard - Data Storage and Privacy Requirements
- Mobile Top 10 2016 Category M2 - Insecure Data Storage
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- CWE - CWE-311 - Missing Encryption of Sensitive Data

Using biometric authentication without a cryptographic solution is security-sensitive (java:S6293)

Severidad: MAJOR

Default:

Android comes with Android KeyStore, a secure container for storing key materials. It's possible to define certain keys to be unlocked when users authenticate using biometric credentials. This way, even if the application process is compromised, the attacker cannot access keys, as presence of the authorized user is required.

These keys can be used, to encrypt, sign or create a message authentication code (MAC) as proof that the authentication result has not been tampered with. This protection defeats the scenario where an attacker with physical access to the device would try to hook into the application process and call the `onAuthenticationSucceeded` method directly. Therefore he would be unable to extract the sensitive data or to perform the critical operations protected by the biometric authentication.

Ask Yourself Whether

The application contains:

- Cryptographic keys / sensitive information that need to be protected using biometric authentication.

There is a risk if you answered yes to this question.

Recommended Secure Coding Practices

It's recommended to tie the biometric authentication to a cryptographic operation by using a `CryptoObject` during authentication.

Sensitive Code Example

A `CryptoObject` is not used during authentication:

```
// ...
BiometricPrompt biometricPrompt = new BiometricPrompt(activity, executor, callback);
// ...
biometricPrompt.authenticate(promptInfo); // Noncompliant
```

Compliant Solution

A `CryptoObject` is used during authentication:

```
// ...
BiometricPrompt biometricPrompt = new BiometricPrompt(activity, executor, callback);
// ...
biometricPrompt.authenticate(promptInfo, new BiometricPrompt.CryptoObject(cipher)); // Compliant
```

See

- OWASP - Top 10 2021 Category A7 - Identification and Authentication Failures
- developer.android.com - Use a cryptographic solution that depends on authentication
- OWASP - Mobile Top 10 2016 Category M4 - Insecure Authentication
- OWASP - Mobile AppSec Verification Standard - Authentication and Session Management Requirements
- CWE - CWE-287 - Improper Authentication

Assess_the_problem:

Ask Yourself Whether

The application contains:

- Cryptographic keys / sensitive information that need to be protected using biometric authentication.

There is a risk if you answered yes to this question.

Sensitive Code Example

A `CryptoObject` is not used during authentication:

```
// ...
BiometricPrompt biometricPrompt = new BiometricPrompt(activity, executor, callback);
// ...
biometricPrompt.authenticate(promptInfo); // Noncompliant
```

Root_cause:

Android comes with Android KeyStore, a secure container for storing key materials. It’s possible to define certain keys to be unlocked when users authenticate using biometric credentials. This way, even if the application process is compromised, the attacker cannot access keys, as presence of the authorized user is required.

These keys can be used, to encrypt, sign or create a message authentication code (MAC) as proof that the authentication result has not been tampered with. This protection defeats the scenario where an attacker with physical access to the device would try to hook into the application process and call the `onAuthenticationSucceeded` method directly. Therefore he would be unable to extract the sensitive data or to perform the critical operations protected by the biometric authentication.

How_to_fix:

Recommended Secure Coding Practices

It’s recommended to tie the biometric authentication to a cryptographic operation by using a `CryptoObject` during authentication.

Compliant Solution

A `CryptoObject` is used during authentication:

```
// ...
BiometricPrompt biometricPrompt = new BiometricPrompt(activity, executor, callback);
// ...
biometricPrompt.authenticate(promptInfo, new BiometricPrompt.CryptoObject(cipher)); // Compliant
```

See

- OWASP - Top 10 2021 Category A7 - Identification and Authentication Failures
- developer.android.com - Use a cryptographic solution that depends on authentication
- OWASP - Mobile Top 10 2016 Category M4 - Insecure Authentication
- OWASP - Mobile AppSec Verification Standard - Authentication and Session Management Requirements
- CWE - CWE-287 - Improper Authentication

Mobile database encryption keys should not be disclosed (java:S6301)

Severidad: MAJOR

How_to_fix:

In the example below, a local database is opened using a hardcoded key. To fix this, the key is moved to a secure location instead and retrieved using a `getKey()` method.

Noncompliant code example

```
String key = "gb09ym9ydoolp3w886d0tciczj6ve9kszqd65u7d126040gwy86xqimjpuuc788g";
RealmConfiguration config = new RealmConfiguration.Builder();
    .encryptionKey(key.toByteArray()) // Noncompliant
    .build();
Realm realm = Realm.getInstance(config);
```

Compliant solution

```
RealmConfiguration config = new RealmConfiguration.Builder()
    .encryptionKey(getKey())
    .build();
Realm realm = Realm.getInstance(config);
```

How does this work?

Using Android’s builtin key storage options

The Android Keystore system allows apps to store encryption keys in a container that is protected on a system level. Additionally, it can restrict when and how the keys are used. For example, it allows the app to require user authentication (for example using a fingerprint) before the key is made available. This is the recommended way to store cryptographic keys on Android.

Dynamically retrieving encryption keys remotely

As user devices are less trusted than controlled environments such as the application backend, the latter should be preferred for the storage of encryption keys. This requires that a user’s device has an internet connection, which may not be suitable for every use case.

Going the extra mile

Avoid storing sensitive data on user devices

In general, it is always preferable to store as little sensitive data on user devices as possible.

Of course, some sensitive data always has to be stored on client devices, such as the data required for authentication. In this case, consider whether the application logic can also function with a hash (or otherwise non-reversible form) of that data. For example, if an email address is required for authentication, it might be possible to use and store a hashed version of this address instead.

How_to_fix:

In the example below, a local database is opened using a hardcoded key. To fix this, the key is moved to a secure location instead and retrieved using a `getKey()` method.

Noncompliant code example

```
String key = "gb09ym9ydoolp3w886d0tciczj6ve9kszqd65u7d126040gwy86xqimjpuuc788g";
SQLiteDatabase db = SQLiteDatabase.openOrCreateDatabase("test.db", key, null); // Noncompliant
```

Compliant solution

```
SQLiteDatabase db = SQLiteDatabase.openOrCreateDatabase("test.db", getKey(), null);
```

How does this work?

Using Android’s builtin key storage options

The Android Keystore system allows apps to store encryption keys in a container that is protected on a system level. Additionally, it can restrict when and how the keys are used. For example, it allows the app to require user authentication (for example using a fingerprint) before the key is made available. This is the recommended way to store cryptographic keys on Android.

Dynamically retrieving encryption keys remotely

As user devices are less trusted than controlled environments such as the application backend, the latter should be preferred for the storage of encryption keys. This requires that a user’s device has an internet connection, which may not be suitable for every use case.

Going the extra mile

Avoid storing sensitive data on user devices

In general, it is always preferable to store as little sensitive data on user devices as possible.

Of course, some sensitive data always has to be stored on client devices, such as the data required for authentication. In this case, consider whether the application logic can also function with a hash (or otherwise non-reversible form) of that data. For example, if an email address is required for authentication, it might be possible to use and store a hashed version of this address instead.

Root_cause:

Mobile applications often need to store data (which might be sensitive) locally. For Android, there exist several libraries that simplify this process by offering a feature-rich database system. SQLCipher and Realm are examples of such libraries. These libraries often add support for database encryption, to protect the contents from being read by other apps or by attackers.

When using encryption for such a database, it is important that the encryption key stays secret. If this key is hardcoded in the application, then it should be considered compromised. The key will be known by anyone with access to the application’s binary code or source code. This means that the sensitive encrypted data can be decrypted by anyone having access to the binary of the mobile application.

Furthermore, if the key is hardcoded, it is the same for every user. A compromise of this encryption key implicates every user of the app.

The encryption key is meant to stay secret and should not be hard-coded in the application as it would mean that:

What is the potential impact?

If an attacker is able to find the encryption key for the mobile database, this can potentially have severe consequences.

Theft of sensitive data

If a mobile database is encrypted, it is likely to contain data that is sensitive for the user or the app publisher. For example, it can contain personally identifiable information (PII), financial data, login credentials, or other sensitive user data.

By not protecting the encryption key properly, it becomes very easy for an attacker to recover it and then decrypt the mobile database. At that point, the theft of sensitive data might lead to identity theft, financial fraud, and other forms of malicious activities.

Introduction:

When storing local data in a mobile application, it is common to use a database that can be encrypted. When encryption of this database is enabled, the encryption key must be protected properly.

Resources:

Documentation

- Android Documentation - Android Keystore system
- Android Documentation - Security tips - User data
- OWASP Mobile Application Security Testing Guide - Data Storage on Android

Standards

- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- OWASP - Top 10 2021 Category A4 - Insecure Design
- OWASP - Mobile AppSec Verification Standard - Data Storage and Privacy Requirements
- OWASP - Mobile Top 10 2016 Category M2 - Insecure Data Storage
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- CWE - CWE-311 - Missing Encryption of Sensitive Data
- CWE - CWE-321 - Use of Hard-coded Cryptographic Key

Lines should not be too long (java:S103)

Severidad: MAJOR

Root_cause:

Scrolling horizontally to see a full line of code lowers the code readability.

Files should not have too many lines of code (java:S104)

Severidad: MAJOR

Root_cause:

When a source file grows too much, it can accumulate numerous responsibilities and become challenging to understand and maintain.

Above a specific threshold, refactor the file into smaller files whose code focuses on well-defined tasks. Those smaller files will be easier to understand and test.

Tabulation characters should not be used (java:S105)

Severidad: MINOR

Root_cause:

The tab width can differ from one development environment to another. Using tabs may require other developers to configure their environment (text editor, preferences, etc.) to read source code.

That is why using spaces is preferable.

Generic exceptions should never be thrown (java:S112)

Severidad: MAJOR

How_to_fix:

To fix this issue, make sure to throw specific exceptions that are relevant to the context in which they arise. It is recommended to either:

- Raise a specific exception from the Java standard library when one matches. For example an `IllegalArgumentException` should be thrown when a method receives an invalid argument.
- Create a custom exception class deriving from `Exception` or one of its subclasses.

Noncompliant code example

```
void checkValue(int value) throws Throwable { // Noncompliant: signature is too broad
    if (value == 42) {
        throw new RuntimeException("Value is 42"); // Noncompliant: This will be difficult for consumers to handle
    }
}
```

Compliant solution

```
void checkValue(int value) {
    if (value == 42) {
        throw new IllegalArgumentException("Value is 42"); // Compliant
    }
}
```

Root_cause:

Throwing generic exceptions such as `Error`, `RuntimeException`, `Throwable`, and `Exception` will have a negative impact on any code trying to catch these exceptions.

From a consumer perspective, it is generally a best practice to only catch exceptions you intend to handle. Other exceptions should ideally be let to propagate up the stack trace so that they can be dealt with appropriately. When a generic exception is thrown, it forces consumers to catch exceptions they do not intend to handle, which they then have to re-throw.

Besides, when working with a generic type of exception, the only way to distinguish between multiple exceptions is to check their message, which is error-prone and difficult to maintain. Legitimate exceptions may be unintentionally silenced and errors may be hidden.

For instance, when a `Throwable` is caught and not re-thrown, it may mask errors such as `OutOfMemoryError` and prevent the program from terminating gracefully.

When throwing an exception, it is therefore recommended to throw the most specific exception possible so that it can be handled intentionally by consumers.

Exceptions

Generic exceptions in the signatures of overriding methods are ignored, because an overriding method has to follow the 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
}
```

Introduction:

This rule raises an issue when a generic exception (such as `Error`, `RuntimeException`, `Throwable`, or `Exception`) is thrown.

Resources:

Standards

- CWE - [CWE-397 Declaration of Throws for Generic Exception]
- CERT - [ERR07-J. Do not throw RuntimeException, Exception, or Throwable]

Related rules

- S1181 - Generic exceptions should not be caught

Custom serialization methods should have required signatures (java:S2061)

Severidad: MAJOR

How_to_fix:

Ensure that the serialization-related method’s signatures match exactly those required by the JVM.

Noncompliant code example

```
public class Watermelon implements Serializable {

  void writeObject(java.io.ObjectOutputStream out)         // Noncompliant, "writeObject" needs to be private, which it is not here
        throws IOException
  {...}

  static Object readResolve() throws ObjectStreamException // Noncompliant, "readResolve" should not be static
  {...}

  Watermelon writeReplace() throws ObjectStreamException   // Noncompliant, "writeReplace" must return "java.lang.Object"
  {...}
}
```

Compliant solution

```
public class Watermelon implements Serializable {

  private void writeObject(java.io.ObjectOutputStream out)    // Compliant, method declared as private
        throws IOException
  {...}

  protected Object readResolve() throws ObjectStreamException // Compliant, method is not static
  {...}

  private Object writeReplace() throws ObjectStreamException  // Compliant, method returns "java.lang.Object"
  {...}
}
```

Root_cause:

Java offers a built-in serialization mechanism for classes that implement the `Serializable` interface. The developer can either rely on Java’s default serialization and deserialization logic or implement custom methods for these tasks. The JVM will use methods such as `readObject` and `writeObject` to execute custom behavior. This only works, however, if these methods match exactly the expected signatures. If they do not, the JVM will fall back to the default logic, resulting in unexpected behavior at runtime, while the developer believes that the default logic has been overidden.

This rule raises an issue if an implementation of `writeObject`, `readObject`, `readObjectNoData`, `writeReplace`, or `readResolve` has an incorrect access modifier, return type, or is not static when it should be (and vice-versa).

Resources:

- CERT, SER01-J. - Do not deviate from the proper signatures of serialization methods
- Oracle SDK - java.io.Serializable

"readResolve" methods should be inheritable (java:S2062)

Severidad: CRITICAL

Root_cause:

Developers may want to add some logic to handle deserialized objects before they are returned to the caller. This can be achieved by implementing the `readResolve` method.

Non-final classes implementing `readResolve` should not set its visibility to `private` as this would make it unavailable to child classes. Instead, mark `readResolve` as `protected`, allowing it to be inherited.

Noncompliant code example

```
public class Fruit implements Serializable {
  private static final long serialVersionUID = 1;

  private Object readResolve() throws ObjectStreamException // Noncompliant, `readResolve` should not be private
  {...}

  //...
}

public class Raspberry extends Fruit implements Serializable { // This class has no access to the parent's "readResolve" method
```

```
  //...
}
```

Compliant solution

```
public class Fruit implements Serializable {
  private static final long serialVersionUID = 1;

  protected Object readResolve() throws ObjectStreamException // Compliant, `readResolve` is protected
  {...}

  //...
}

public class Raspberry extends Fruit implements Serializable { // This class has access to the parent's "readResolve"
  //...
}
```

Resources:

- Java Object Serialization Specification - Object Input Classes

---

Comparators should be "Serializable" (java:S2063)

Severidad: CRITICAL

Root_cause:

A non-serializable `Comparator` can prevent an otherwise-`Serializable` ordered collection from being serializable. Since the overhead to make a `Comparator` serializable is usually low, doing so can be considered good defensive programming.

Noncompliant code example

```
public class FruitComparator implements Comparator<Fruit> {  // Noncompliant
  int compare(Fruit f1, Fruit f2) {...}
  boolean equals(Object obj) {...}
}
```

Compliant solution

```
public class FruitComparator implements Comparator<Fruit>, Serializable {
  private static final long serialVersionUID = 1;

  int compare(Fruit f1, Fruit f2) {...}
  boolean equals(Object obj) {...}
}
```

Fields in non-serializable classes should not be "transient" (java:S2065)

Severidad: MINOR

Resources:

- Baeldung - The transient Keyword in Java

How_to_fix:

Ask yourself whether this class should be serializable. If yes, ensure it implements `Serializable` and provides any additional logic required to serialize and deserialize an instance of this type. Otherwise, remove the `transient` modifier from this field.

Noncompliant code example

```
class Vegetable {
  private transient Season ripe; // Noncompliant, the "Vegetable" class does not implement "Serializable" but the field is marked as "trans
  // ...
}
```

Compliant solution

```
class Vegetable {
  private Season ripe; // Compliant, the field is not marked as "transient"
  // ...
}
```

Root_cause:

Fields marked as `transient` in a `Serializable` class will be ignored during serialization and consequently not written out to a file (or stream).

This can be useful in situations such as where the content of a field can be recomputed from other fields. To reduce the output size, this field can be marked as `transient` and recomputed when a given object is deserialized.

Since `transient` is very specific to classes that implement `Serializable`, it is superfluous in classes that do not.

This rule raises an issue when a field is marked as `transient`, even though the containing class does not implement `Serializable`.

"Serializable" inner classes of non-serializable outer classes should be "static" (java:S2066)

Severidad: MINOR

Resources:

- CERT SER05-J. - Do not serialize instances of inner classes

Root_cause:

Non-static inner classes contain a reference to an instance of the outer class. Hence, serializing a non-static inner class will result in an attempt at serializing the outer class as well. If the outer class is not serializable, serialization will fail, resulting in a runtime error.

Making the inner class `static` (i.e., "nested") avoids this problem, as no reference to an instance of the outer class is required. Serializing the inner class can be done independently of the outer class. Hence, inner classes implementing `Serializable` should be `static` if the outer class does not implement `Serializable`.

Be aware of the semantic differences between an inner class and a nested one:

- an inner class can only be instantiated within the context of an instance of the outer class.
- a nested (`static`) class can be instantiated independently of the outer class.

How_to_fix:

Make the inner class `static` or make the outer class `Serializable`.

Noncompliant code example

```
public class Pomegranate {
  // ...

  public class Seed implements Serializable {  // Noncompliant, serialization will fail due to the outer class not being serializable
    // ...
  }
}
```

Compliant solution

```
public class Pomegranate {
  // ...

  public static class Seed implements Serializable { // Compliant, the outer class will not be serialized and hence cannot be the cause for
    // ...
  }
}
```

Hard-coded passwords are security-sensitive (java:S2068)

Severidad: BLOCKER

Assess_the_problem:

Ask Yourself Whether

- The password allows access to a sensitive component like a database, a file storage, an API, or a service.
- The password is used in production environments.
- Application re-distribution is required before updating the password.

There would be a risk if you answered yes to any of those questions.

Sensitive Code Example

```
String username = "steve";
String password = "blue";
Connection conn = DriverManager.getConnection("jdbc:mysql://localhost/test?" +
                  "user=" + username + "&password=" + password); // Sensitive
```

Root_cause:

Because it is easy to extract strings from an application source code or binary, passwords should not be hard-coded. This is particularly true for applications that are distributed or that are open-source.

In the past, it has led to the following vulnerabilities:

- CVE-2019-13466
- CVE-2018-15389

Passwords should be stored outside of the code in a configuration file, a database, or a password management service.

This rule flags instances of hard-coded passwords used in database and LDAP connections. It looks for hard-coded passwords in connection strings, and for variable names that match any of the patterns from the provided list.

Default:

Because it is easy to extract strings from an application source code or binary, passwords should not be hard-coded. This is particularly true for applications that are distributed or that are open-source.

In the past, it has led to the following vulnerabilities:

- CVE-2019-13466
- CVE-2018-15389

Passwords should be stored outside of the code in a configuration file, a database, or a password management service.

This rule flags instances of hard-coded passwords used in database and LDAP connections. It looks for hard-coded passwords in connection strings, and for variable names that match any of the patterns from the provided list.

Ask Yourself Whether

- The password allows access to a sensitive component like a database, a file storage, an API, or a service.
- The password is used in production environments.
- Application re-distribution is required before updating the password.

There would be a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- Store the credentials in a configuration file that is not pushed to the code repository.
- Store the credentials in a database.
- Use your cloud provider’s service for managing secrets.
- If a password has been disclosed through the source code: change it.

Sensitive Code Example

```
String username = "steve";
String password = "blue";
Connection conn = DriverManager.getConnection("jdbc:mysql://localhost/test?" +
                  "user=" + username + "&password=" + password); // Sensitive
```

Compliant Solution

```
String username = getEncryptedUser();
String password = getEncryptedPassword();
Connection conn = DriverManager.getConnection("jdbc:mysql://localhost/test?" +
                  "user=" + username + "&password=" + password);
```

See

- OWASP - Top 10 2021 Category A7 - Identification and Authentication Failures
- OWASP - Top 10 2017 Category A2 - Broken Authentication
- CWE - CWE-798 - Use of Hard-coded Credentials

- CWE - CWE-259 - Use of Hard-coded Password
- CERT, MSC03-J. - Never hard code sensitive information
- Derived from FindSecBugs rule Hard Coded Password

How_to_fix:

Recommended Secure Coding Practices

- Store the credentials in a configuration file that is not pushed to the code repository.
- Store the credentials in a database.
- Use your cloud provider’s service for managing secrets.
- If a password has been disclosed through the source code: change it.

Compliant Solution

```
String username = getEncryptedUser();
String password = getEncryptedPassword();
Connection conn = DriverManager.getConnection("jdbc:mysql://localhost/test?" +
                  "user=" + username + "&password=" + password);
```

See

- OWASP - Top 10 2021 Category A7 - Identification and Authentication Failures
- OWASP - Top 10 2017 Category A2 - Broken Authentication
- CWE - CWE-798 - Use of Hard-coded Credentials
- CWE - CWE-259 - Use of Hard-coded Password
- CERT, MSC03-J. - Never hard code sensitive information
- Derived from FindSecBugs rule Hard Coded Password

Ints and longs should not be shifted by zero or more than their number of bits-1 (java:S2183)

Severidad: MINOR

How_to_fix:

Noncompliant code example

```
public int shift(int a) {
  int x = a >> 32; // Noncompliant
  return a << 48;  // Noncompliant
}
```

Compliant solution

```
public int shift(int a) {
  int x = a >> 31;
  return a << 16;
}
```

Root_cause:

Since an `int` is a 32-bit variable, shifting by more than +/-31 is confusing at best and an error at worst. When the runtime shifts 32-bit integers, it uses the lowest 5 bits of the shift count operand. In other words, shifting an `int` by 32 is the same as shifting it by 0, and shifting it by 33 is the same as shifting it by 1.

Similarly, when shifting 64-bit integers, the runtime uses the lowest 6 bits of the shift count operand and shifting `long` by 64 is the same as shifting it by 0, and shifting it by 65 is the same as shifting it by 1.

Math operands should be cast before assignment (java:S2184)

Severidad: MINOR

Root_cause:

When arithmetic is performed on integers, the result will always be an integer. You can assign that result to a `long`, `double`, or `float` with automatic type conversion, but having started as an `int` or `long`, the result will likely not be what you expect.

For instance, if the result of `int` division is assigned to a floating-point variable, precision will have been lost before the assignment. Likewise, if the result of multiplication is assigned to a `long`, it may have already overflowed before the assignment.

In either case, the result will not be what was expected. Instead, at least one operand should be cast or promoted to the final type before the operation takes place.

Noncompliant code example

```
float twoThirds = 2/3; // Noncompliant; int division. Yields 0.0
long millisInYear = 1_000*3_600*24*365; // Noncompliant; int multiplication. Yields 1471228928
long bigNum = Integer.MAX_VALUE + 2; // Noncompliant. Yields -2147483647
long bigNegNum =  Integer.MIN_VALUE-1; //Noncompliant, gives a positive result instead of a negative one.
Date myDate = new Date(seconds * 1_000); //Noncompliant, won't produce the expected result if seconds > 2_147_483
...
public long compute(int factor){
  return factor * 10_000;  //Noncompliant, won't produce the expected result if factor > 214_748
}

public float compute2(long factor){
  return factor / 123;  //Noncompliant, will be rounded to closest long integer
}
```

Compliant solution

```
float twoThirds = 2f/3; // 2 promoted to float. Yields 0.6666667
long millisInYear = 1_000L*3_600*24*365; // 1000 promoted to long. Yields 31_536_000_000
long bigNum = Integer.MAX_VALUE + 2L; // 2 promoted to long. Yields 2_147_483_649
long bigNegNum =  Integer.MIN_VALUE-1L; // Yields -2_147_483_649
Date myDate = new Date(seconds * 1_000L);
...
public long compute(int factor){
  return factor * 10_000L;
}

public float compute2(long factor){
  return factor / 123f;
}
```

or

```
float twoThirds = (float)2/3; // 2 cast to float
long millisInYear = (long)1_000*3_600*24*365; // 1_000 cast to long
long bigNum = (long)Integer.MAX_VALUE + 2;
long bigNegNum =  (long)Integer.MIN_VALUE-1;
Date myDate = new Date((long)seconds * 1_000);
...
public long compute(long factor){
  return factor * 10_000;
}

public float compute2(float factor){
  return factor / 123;
}
```

Resources:

Standards

- CWE - CWE-190 - Integer Overflow or Wraparound
- CERT, NUM50-J. - Convert integers to floating point for floating-point operations
- CERT, INT18-C. - Evaluate integer expressions in a larger size before comparing or assigning to that size
- STIG Viewer - Application Security and Development: V-222612 - The application must not be vulnerable to overflow attacks.

Do not perform unnecessary mathematical operations (java:S2185)

Severidad: MAJOR

Root_cause:

Some mathematical operations are unnecessary and should not be performed because their results are predictable.

For instance, `anyValue % 1` will always return 0, as any integer value can be divided by 1 without remainder.

Similarly, casting a non-floating-point to a floating-point value and then passing it to `Math.round`, `Math.ceil`, or `Math.floor` is also unnecessary, as the result will always be the original value.

The following operations are unnecessary when given any constant value: `Math.abs`, `Math.ceil`, `Math.floor`, `Math.rint`, `Math.round`. Instead, use the result of the operation directly.

The following operations are unnecessary with certain constants and can be replaced by the result of the operation directly:

Operation	Value
acos	0.0 or 1.0
asin	0.0 or 1.0
atan	0.0 or 1.0
atan2	0.0
cbrt	0.0 or 1.0
cos	0.0
cosh	0.0
exp	0.0 or 1.0
expm1	0.0
log	0.0 or 1.0
log10	0.0 or 1.0
sin	0.0
sinh	0.0
sqrt	0.0 or 1.0
tan	0.0
tanh	0.0
toDegrees	0.0 or 1.0
toRadians	0.0

How_to_fix:

Ask yourself if the questionable operation represents the desired calculation or if a value used is erroneous. If the calculation is correct, replace it with the result to avoid having to perform the unnecessary operation at runtime.

Noncompliant code example

```
public void doMath(int a) {
  double res1 = Math.floor((double)a); // Noncompliant, the result will always be equal to '(double) a'
  double res2 = Math.ceil(4.2);        // Noncompliant, the result will always be 5.0
  double res3 = Math.atan(0.0);        // Noncompliant, the result will always be 0.0
}
```

Compliant solution

```
public void doMath(int a) {
  double res1 = a;    // Compliant
  double res2 = 5.0;  // Compliant
  double res3 = 0.0;  // Compliant
}
```

JUnit assertions should not be used in "run" methods (java:S2186)

Severidad: CRITICAL

Root_cause:

JUnit assertions should not be made from the `run` method of a `Runnable`, because their failure may not be detected in the test that initiated them. Failed assertions throw assertion errors. However, if the error is thrown from another thread than the one that initiated the test, the thread will exit but the test will not fail.

How_to_fix:

Assertions in `Runnable` tasks should be extracted or executed by the main thread to make the whole test fail.

Noncompliant code example

```
class RunnableWithAnAssertion extends Thread {
  @Override
  public void run() {
    Assert.assertEquals(expected, actual);  // Noncompliant
  }

  @Test
  void test() {
    RunnableWithAnAssertion otherThread = new RunnableWithAnAssertion();
    otherThread.start(); // The assertion in the run method above will be executed by other thread than the current one
    // ...
    // Perform some actions that do not make the test fail
    // ...
    Assert.assertTrue(true);
  }
}
```

Compliant solution

```
class RunnableWithAnAssertion extends Thread {
  @Override
  public void run() {
    Assert.assertEquals(expected, actual);  // Noncompliant
  }

  @Test
  void test() {
    RunnableWithAnAssertion otherThread = new RunnableWithAnAssertion();
    otherThread.run();
    // ...
    // The failed assertions in the run method will prevent us from reaching the assertion below
    // ...
    Assert.assertTrue(true);
  }
}
```

TestCases should contain tests (java:S2187)

Severidad: BLOCKER

Root_cause:

There’s no point in having a JUnit `TestCase` without any test methods. Similarly, you shouldn’t have a file in the tests directory named `*Test`, `*Tests`, or `*TestCase`, but no tests in the file. Doing either of these things may lead someone to think that uncovered classes have been tested.

This rule raises an issue when files in the test directory are named `*Test`, `*Tests`, or `*TestCase` or implement `TestCase` but don’t contain any tests.

Supported frameworks:

- JUnit3
- JUnit4
- JUnit5
- TestNG
- Zohhak
- ArchUnit
- Pact

Classes should not have too many "static" imports (java:S3030)

Severidad: MAJOR

Root_cause:

Importing a class statically allows you to use its `public static` members without qualifying them with the class name. That can be handy, but if you import too many classes statically, your code can become confusing and difficult to maintain.

Noncompliant code example

With the default threshold value: 4

```
import static java.lang.Math.*;
import static java.util.Collections.*;
import static com.myco.corporate.Constants.*;
import static com.myco.division.Constants.*;
import static com.myco.department.Constants.*;  // Noncompliant
```

JEE applications should not "getClassLoader" (java:S3032)

Severidad: MINOR

Root_cause:

Using the standard `getClassLoader()` may not return the *right* class loader in a JEE context. Instead, go through the `currentThread`.

Noncompliant code example

```
ClassLoader cl = this.getClass().getClassLoader();  // Noncompliant
```

Compliant solution

```
ClassLoader cl = Thread.currentThread().getContextClassLoader();
```

Raw byte values should not be used in bitwise operations in combination with shifts (java:S3034)

Severidad: MAJOR

Root_cause:

In Java, numeric promotions happen when two operands of an arithmetic expression have different sizes. More specifically, narrower operands get promoted to the type of wider operands. For instance, an operation between a `byte` and an `int`, will trigger a promotion of the `byte` operand, converting it into an `int`.

When this happens, the sequence of 8 bits that represents the `byte` will need to be extended to match the 32-bit long sequence that represents the `int` operand. Since Java uses two's complement notation for signed number types, the promotion will fill the missing leading bits with zeros or ones, depending on the sign of the value. For instance, the byte `0b1000_0000` (equal to `-128` in decimal notation), when promoted to `int`, will become `0b1111_1111_1111_1111_1111_1111_1000_0000`.

When performing shifting or bitwise operations without considering that bytes are signed, the bits added during the promotion may have unexpected effects on the final result of the operations.

How_to_fix:

This rule raises an issue any time a `byte` value is used as an operand combined with shifts without being masked.

To prevent such accidental value conversions, you can mask promoted bytes to only consider the least significant 8 bits. Masking can be achieved with the bitwise AND operator & and the appropriate mask of `0xff` (255 in decimal and `0b1111_1111` in binary) or, since Java 8, with the more convenient `Byte.toUnsignedInt(byte b)` or `Byte.toUnsignedLong(byte b)`.

Noncompliant code example

```
  public static void main(String[] args) {
    byte[] bytes12 = BigInteger.valueOf(12).toByteArray(); // This byte array will be simply [12]
    System.out.println(intFromBuffer(bytes12)); // In this case, the bytes promotion will not cause any issues, and "12" will be printed.

    // Here the bytes will be [2, -128] since 640 in binary is represented as 0b0000_0010_1000_0000
    // which is equivalent to the concatenation of 2 bytes: 0b0000_0010 = 2, and 0b1000_0000 = -128
    byte[] bytes640 = BigInteger.valueOf(640).toByteArray();

    // In this case, the shifting operation combined with the bitwise OR, will produce the wrong binary string and "-128" will be printed.
    System.out.println(intFromBuffer(bytes640));
  }

  static int intFromBuffer(byte[] bytes) {
    int originalInt = 0;
    for (int i = 0; i < bytes.length; i++) {
      // Here the right operand of the bitwise OR, which is a byte, will be promoted to an `int`
      // and if its value was negative, the added ones in front of the binary string will alter the value of the `originalInt`
      originalInt = (originalInt << 8) | bytes[i]; // Noncompliant
    }
    return originalInt;
  }
```

Compliant solution

```
  public static void main(String[] args) {
    byte[] bytes12 = BigInteger.valueOf(12).toByteArray(); // This byte array will be simply [12]
    System.out.println(intFromBuffer(bytes12)); // In this case, the bytes promotion will not cause any issues, and "12" will be printed.

    // Here the bytes will be [2, -128] since 640 in binary is represented as 0b0000_0010_1000_0000
    // which is equivalent to the concatenation of 2 bytes: 0b0000_0010 = 2, and 0b1000_0000 = -128
    byte[] bytes640 = BigInteger.valueOf(640).toByteArray();

    // This will correctly print "640" now.
    System.out.println(intFromBuffer(bytes640));
  }

  static int intFromBuffer(byte[] bytes) {
    int originalInt = 0;
    for (int i = 0; i < bytes.length; i++) {
      originalInt = (originalInt << 8) | Byte.toUnsignedInt(bytes[i]); // Compliant, only the relevant 8 least significant bits will affect
    }
    return originalInt;
  }
```

References

- CERT, NUM52-J. - Be aware of numeric promotion behavior
- Wikipedia - Two’s complement

Composed "@RequestMapping" variants should be preferred (java:S4488)

Severidad: MINOR

Root_cause:

Spring framework 4.3 introduced variants of the `@RequestMapping` annotation to better represent the semantics of the annotated methods. The use of `@GetMapping`, `@PostMapping`, `@PutMapping`, `@PatchMapping` and `@DeleteMapping` should be preferred to the use of the raw `@RequestMapping(method = RequestMethod.XYZ)`.

Noncompliant code example

```
@RequestMapping(path = "/greeting", method = RequestMethod.GET) // Noncompliant
public Greeting greeting(@RequestParam(value = "name", defaultValue = "World") String name) {
...
}
```

Compliant solution

```
@GetMapping(path = "/greeting") // Compliant
public Greeting greeting(@RequestParam(value = "name", defaultValue = "World") String name) {
...
}
```

Using clear-text protocols is security-sensitive (java:S5332)

Severidad: CRITICAL

Assess_the_problem:

Ask Yourself Whether

- Application data needs to be protected against falsifications or leaks when transiting over the network.
- Application data transits over an untrusted network.
- Compliance rules require the service to encrypt data in transit.
- Your application renders web pages with a relaxed mixed content policy.
- OS-level protections against clear-text traffic are deactivated.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

These clients from Apache commons net libraries are based on unencrypted protocols and are not recommended:

```
TelnetClient telnet = new TelnetClient(); // Sensitive

FTPClient ftpClient = new FTPClient(); // Sensitive

SMTPClient smtpClient = new SMTPClient(); // Sensitive
```

Unencrypted HTTP connections, when using okhttp library for instance, should be avoided:

```
ConnectionSpec spec = new ConnectionSpec.Builder(ConnectionSpec.CLEARTEXT) // Sensitive
  .build();
```

Android WebView can be configured to allow a secure origin to load content from any other origin, even if that origin is insecure (mixed content):

```
import android.webkit.WebView

WebView webView = findViewById(R.id.webview)
webView.getSettings().setMixedContentMode(MIXED_CONTENT_ALWAYS_ALLOW); // Sensitive
```

How_to_fix:

Recommended Secure Coding Practices

- Make application data transit over a secure, authenticated and encrypted protocol like TLS or SSH. Here are a few alternatives to the most common clear-text protocols:
    - Use `ssh` as an alternative to `telnet`.
    - Use `sftp`, `scp`, or `ftps` instead of `ftp`.
    - Use `https` instead of `http`.
    - Use `SMTP` over `SSL/TLS` or `SMTP` with `STARTTLS` instead of clear-text SMTP.
- Enable encryption of cloud components communications whenever it is possible.
- Configure your application to block mixed content when rendering web pages.
- If available, enforce OS-level deactivation of all clear-text traffic.

It is recommended to secure all transport channels, even on local networks, as it can take a single non-secure connection to compromise an entire application or system.

Compliant Solution

Use instead these clients from Apache commons net and JSch/ssh library:

```
JSch jsch = new JSch();

if(implicit) {
  // implicit mode is considered deprecated but offer the same security than explicit mode
  FTPSClient ftpsClient = new FTPSClient(true);
}
else {
  FTPSClient ftpsClient = new FTPSClient();
}

if(implicit) {
  // implicit mode is considered deprecated but offer the same security than explicit mode
  SMTPSClient smtpsClient = new SMTPSClient(true);
}
else {
  SMTPSClient smtpsClient = new SMTPSClient();
  smtpsClient.connect("127.0.0.1", 25);
  if (smtpsClient.execTLS()) {
    // commands
  }
}
```

Perform HTTP encrypted connections, with okhttp library for instance:

```
ConnectionSpec spec = new ConnectionSpec.Builder(ConnectionSpec.MODERN_TLS)
  .build();
```

The most secure mode for Android WebView is `MIXED_CONTENT_NEVER_ALLOW`:

```
import android.webkit.WebView

WebView webView = findViewById(R.id.webview)
webView.getSettings().setMixedContentMode(MIXED_CONTENT_NEVER_ALLOW);
```

See

Documentation

- AWS Documentation - Listeners for your Application Load Balancers
- AWS Documentation - Stream Encryption

Articles & blog posts

- Google - Moving towards more secure web
- Mozilla - Deprecating non secure http

Standards

- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- OWASP - Mobile AppSec Verification Standard - Network Communication Requirements
- OWASP - Mobile Top 10 2016 Category M3 - Insecure Communication
- CWE - CWE-200 - Exposure of Sensitive Information to an Unauthorized Actor
- CWE - CWE-319 - Cleartext Transmission of Sensitive Information

- STIG Viewer - Application Security and Development: V-222397 - The application must implement cryptographic mechanisms to protect the integrity of remote access sessions.
- STIG Viewer - Application Security and Development: V-222534 - Service-Oriented Applications handling non-releasable data must authenticate endpoint devices via mutual SSL/TLS.
- STIG Viewer - Application Security and Development: V-222562 - Applications used for non-local maintenance must implement cryptographic mechanisms to protect the integrity of maintenance and diagnostic communications.
- STIG Viewer - Application Security and Development: V-222563 - Applications used for non-local maintenance must implement cryptographic mechanisms to protect the confidentiality of maintenance and diagnostic communications.
- STIG Viewer - Application Security and Development: V-222577 - The application must not expose session IDs.
- STIG Viewer - Application Security and Development: V-222596 - The application must protect the confidentiality and integrity of transmitted information.
- STIG Viewer - Application Security and Development: V-222597 - The application must implement cryptographic mechanisms to prevent unauthorized disclosure of information and/or detect changes to information during transmission.
- STIG Viewer - Application Security and Development: V-222598 - The application must maintain the confidentiality and integrity of information during preparation for transmission.
- STIG Viewer - Application Security and Development: V-222599 - The application must maintain the confidentiality and integrity of information during reception.

Default:

Clear-text protocols such as `ftp`, `telnet`, or `http` lack encryption of transported data, as well as the capability to build an authenticated connection. It means that an attacker able to sniff traffic from the network can read, modify, or corrupt the transported content. These protocols are not secure as they expose applications to an extensive range of risks:

- sensitive data exposure
- traffic redirected to a malicious endpoint
- malware-infected software update or installer
- execution of client-side code
- corruption of critical information

Even in the context of isolated networks like offline environments or segmented cloud environments, the insider threat exists. Thus, attacks involving communications being sniffed or tampered with can still happen.

For example, attackers could successfully compromise prior security layers by:

- bypassing isolation mechanisms
- compromising a component of the network
- getting the credentials of an internal IAM account (either from a service account or an actual person)

In such cases, encrypting communications would decrease the chances of attackers to successfully leak data or steal credentials from other network components. By layering various security practices (segmentation and encryption, for example), the application will follow the *defense-in-depth* principle.

Note that using the `http` protocol is being deprecated by major web browsers.

In the past, it has led to the following vulnerabilities:

- CVE-2019-6169
- CVE-2019-12327
- CVE-2019-11065

Ask Yourself Whether

- Application data needs to be protected against falsifications or leaks when transiting over the network.
- Application data transits over an untrusted network.
- Compliance rules require the service to encrypt data in transit.
- Your application renders web pages with a relaxed mixed content policy.
- OS-level protections against clear-text traffic are deactivated.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- Make application data transit over a secure, authenticated and encrypted protocol like TLS or SSH. Here are a few alternatives to the most common clear-text protocols:
  - Use `ssh` as an alternative to `telnet`.
  - Use `sftp`, `scp`, or `ftps` instead of `ftp`.
  - Use `https` instead of `http`.
  - Use `SMTP` over `SSL/TLS` or `SMTP` with `STARTTLS` instead of clear-text SMTP.
- Enable encryption of cloud components communications whenever it is possible.
- Configure your application to block mixed content when rendering web pages.
- If available, enforce OS-level deactivation of all clear-text traffic.

It is recommended to secure all transport channels, even on local networks, as it can take a single non-secure connection to compromise an entire application or system.

Sensitive Code Example

These clients from Apache commons net libraries are based on unencrypted protocols and are not recommended:

```
TelnetClient telnet = new TelnetClient(); // Sensitive

FTPClient ftpClient = new FTPClient(); // Sensitive

SMTPClient smtpClient = new SMTPClient(); // Sensitive
```

Unencrypted HTTP connections, when using okhttp library for instance, should be avoided:

```
ConnectionSpec spec = new ConnectionSpec.Builder(ConnectionSpec.CLEARTEXT) // Sensitive
  .build();
```

Android WebView can be configured to allow a secure origin to load content from any other origin, even if that origin is insecure (mixed content):

```
import android.webkit.WebView

WebView webView = findViewById(R.id.webview)
webView.getSettings().setMixedContentMode(MIXED_CONTENT_ALWAYS_ALLOW); // Sensitive
```

Compliant Solution

Use instead these clients from Apache commons net and JSch/ssh library:

```
JSch jsch = new JSch();

if(implicit) {
  // implicit mode is considered deprecated but offer the same security than explicit mode
  FTPSClient ftpsClient = new FTPSClient(true);
}
else {
  FTPSClient ftpsClient = new FTPSClient();
}

if(implicit) {
  // implicit mode is considered deprecated but offer the same security than explicit mode
  SMTPSClient smtpsClient = new SMTPSClient(true);
}
else {
  SMTPSClient smtpsClient = new SMTPSClient();
  smtpsClient.connect("127.0.0.1", 25);
  if (smtpsClient.execTLS()) {
    // commands
  }
}
```

Perform HTTP encrypted connections, with okhttp library for instance:

```
ConnectionSpec spec = new ConnectionSpec.Builder(ConnectionSpec.MODERN_TLS)
  .build();
```

The most secure mode for Android WebView is `MIXED_CONTENT_NEVER_ALLOW`:

```
import android.webkit.WebView

WebView webView = findViewById(R.id.webview)
webView.getSettings().setMixedContentMode(MIXED_CONTENT_NEVER_ALLOW);
```

Exceptions

No issue is reported for the following cases because they are not considered sensitive:

- Insecure protocol scheme followed by loopback addresses like 127.0.0.1 or `localhost`.

See

Documentation

- AWS Documentation - Listeners for your Application Load Balancers
- AWS Documentation - Stream Encryption

Articles & blog posts

- Google - Moving towards more secure web
- Mozilla - Deprecating non secure http

Standards

- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- OWASP - Mobile AppSec Verification Standard - Network Communication Requirements
- OWASP - Mobile Top 10 2016 Category M3 - Insecure Communication
- CWE - CWE-200 - Exposure of Sensitive Information to an Unauthorized Actor
- CWE - CWE-319 - Cleartext Transmission of Sensitive Information
- STIG Viewer - Application Security and Development: V-222397 - The application must implement cryptographic mechanisms to protect the integrity of remote access sessions.
- STIG Viewer - Application Security and Development: V-222534 - Service-Oriented Applications handling non-releasable data must authenticate endpoint devices via mutual SSL/TLS.
- STIG Viewer - Application Security and Development: V-222562 - Applications used for non-local maintenance must implement cryptographic mechanisms to protect the integrity of maintenance and diagnostic communications.
- STIG Viewer - Application Security and Development: V-222563 - Applications used for non-local maintenance must implement cryptographic mechanisms to protect the confidentiality of maintenance and diagnostic communications.
- STIG Viewer - Application Security and Development: V-222577 - The application must not expose session IDs.
- STIG Viewer - Application Security and Development: V-222596 - The application must protect the confidentiality and integrity of transmitted information.
- STIG Viewer - Application Security and Development: V-222597 - The application must implement cryptographic mechanisms to prevent unauthorized disclosure of information and/or detect changes to information during transmission.
- STIG Viewer - Application Security and Development: V-222598 - The application must maintain the confidentiality and integrity of information during preparation for transmission.
- STIG Viewer - Application Security and Development: V-222599 - The application must maintain the confidentiality and integrity of information during reception.

Root_cause:

Clear-text protocols such as `ftp`, `telnet`, or `http` lack encryption of transported data, as well as the capability to build an authenticated connection. It means that an attacker able to sniff traffic from the network can read, modify, or corrupt the transported content. These protocols are not secure as they expose applications to an extensive range of risks:

- sensitive data exposure
- traffic redirected to a malicious endpoint
- malware-infected software update or installer
- execution of client-side code
- corruption of critical information

Even in the context of isolated networks like offline environments or segmented cloud environments, the insider threat exists. Thus, attacks involving communications being sniffed or tampered with can still happen.

For example, attackers could successfully compromise prior security layers by:

- bypassing isolation mechanisms
- compromising a component of the network
- getting the credentials of an internal IAM account (either from a service account or an actual person)

In such cases, encrypting communications would decrease the chances of attackers to successfully leak data or steal credentials from other network components. By layering various security practices (segmentation and encryption, for example), the application will follow the *defense-in-depth* principle.

Note that using the `http` protocol is being deprecated by major web browsers.

In the past, it has led to the following vulnerabilities:

- CVE-2019-6169
- CVE-2019-12327
- CVE-2019-11065

Exceptions

No issue is reported for the following cases because they are not considered sensitive:

- Insecure protocol scheme followed by loopback addresses like 127.0.0.1 or `localhost`.

Using unencrypted files in mobile applications is security-sensitive (java:S6300)

Severidad: MAJOR

Default:

Storing files locally is a common task for mobile applications. Files that are stored unencrypted can be read out and modified by an attacker with physical access to the device. Access to sensitive data can be harmful for the user of the application, for example when the device gets stolen.

Ask Yourself Whether

- The file contains sensitive data that could cause harm when leaked.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

It's recommended to password-encrypt local files that contain sensitive information. The class EncryptedFile can be used to easily encrypt files.

Sensitive Code Example

```
Files.write(path, content); // Sensitive

FileOutputStream out = new FileOutputStream(file); // Sensitive

FileWriter fw = new FileWriter("outfilename", false); // Sensitive
```

Compliant Solution

```
String masterKeyAlias = MasterKeys.getOrCreate(MasterKeys.AES256_GCM_SPEC);

File file = new File(context.getFilesDir(), "secret_data");
EncryptedFile encryptedFile = EncryptedFile.Builder(
    file,
    context,
    masterKeyAlias,
    EncryptedFile.FileEncryptionScheme.AES256_GCM_HKDF_4KB
).build();

// write to the encrypted file
FileOutputStream encryptedOutputStream = encryptedFile.openFileOutput();
```

See

- OWASP - Top 10 2021 Category A4 - Insecure Design
- OWASP - Mobile AppSec Verification Standard - Data Storage and Privacy Requirements
- OWASP - Mobile Top 10 2016 Category M2 - Insecure Data Storage
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- CWE - CWE-311 - Missing Encryption of Sensitive Data

Assess_the_problem:

Ask Yourself Whether

- The file contains sensitive data that could cause harm when leaked.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

```
Files.write(path, content); // Sensitive

FileOutputStream out = new FileOutputStream(file); // Sensitive

FileWriter fw = new FileWriter("outfilename", false); // Sensitive
```

Root_cause:

Storing files locally is a common task for mobile applications. Files that are stored unencrypted can be read out and modified by an attacker with physical access to the device. Access to sensitive data can be harmful for the user of the application, for example when the device gets stolen.

How_to_fix:

Recommended Secure Coding Practices

It’s recommended to password-encrypt local files that contain sensitive information. The class EncryptedFile can be used to easily encrypt files.

Compliant Solution

```
String masterKeyAlias = MasterKeys.getOrCreate(MasterKeys.AES256_GCM_SPEC);

File file = new File(context.getFilesDir(), "secret_data");
EncryptedFile encryptedFile = EncryptedFile.Builder(
    file,
    context,
    masterKeyAlias,
    EncryptedFile.FileEncryptionScheme.AES256_GCM_HKDF_4KB
).build();

// write to the encrypted file
FileOutputStream encryptedOutputStream = encryptedFile.openFileOutput();
```

See

- OWASP - Top 10 2021 Category A4 - Insecure Design
- OWASP - Mobile AppSec Verification Standard - Data Storage and Privacy Requirements
- OWASP - Mobile Top 10 2016 Category M2 - Insecure Data Storage
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- CWE - CWE-311 - Missing Encryption of Sensitive Data

Redundant nullability annotations should be removed (java:S6665)

Severidad: MAJOR

Root_cause:

Nullability annotations in Java are used to indicate whether a variable or parameter can be assigned a null value or not. These annotations help to prevent Null Pointer Exceptions and improve the reliability of code.

Redundant nullability annotations can clutter the code and make it harder to read and understand. When a nullability annotation is already implied by the context or by other annotations, explicitly adding it again only adds noise and makes the code less clear.

Removing them improves code readability, maintainability, reduces the risk of inconsistencies, and ensures that the remaining annotations carry meaningful information.

Noncompliant code example

JSpecify code example:

```
@NullMarked
class MyClass {
  public void method(@NonNull Object o) { // Noncompliant: @NonNull is redundant here
    // ...
  }
}
```

Compliant solution

```
@NullMarked
class MyClass {
  public void method(Object o) {
    // ...
  }
}
```

Standard outputs should not be used directly to log anything (java:S106)

Severidad: MAJOR

Root_cause:

In software development, logs serve as a record of events within an application, providing crucial insights for debugging. When logging, it is essential to ensure that the logs are:

- easily accessible
- uniformly formatted for readability

- properly recorded
- securely logged when dealing with sensitive data

Those requirements are not met if a program directly writes to the standard outputs (e.g., System.out, System.err). That is why defining and using a dedicated logger is highly recommended.

The following noncompliant code:

```
class MyClass {
  public void doSomething() {
    System.out.println("My Message");  // Noncompliant, output directly to System.out without a logger
  }
}
```

Could be replaced by:

```
import java.util.logging.Logger;

class MyClass {

  Logger logger = Logger.getLogger(getClass().getName());

  public void doSomething() {
    // ...
    logger.info("My Message");  // Compliant, output via logger
    // ...
  }
}
```

Resources:

Documentation

- Java SE 7 API Specification: java.util.logging.Logger
- OWASP - Top 10 2021 Category A9 - Security Logging and Monitoring Failures
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- CERT, ERR02-J. - Prevent exceptions while logging data

Methods should not have too many parameters (java:S107)

Severidad: MAJOR

Root_cause:

Methods with a long parameter list are difficult to use because maintainers must figure out the role of each parameter and keep track of their position.

```
void setCoordinates(int x1, int y1, int z1, int x2, int y2, int z2) { // Noncompliant
    // ...
}
```

The solution can be to:

- Split the method into smaller ones

```
// Each function does a part of what the original setCoordinates function was doing, so confusion risks are lower
void setOrigin(int x, int y, int z) {
   // ...
}

void setSize(int width, int height, int depth) {
   // ...
}
```

- Find a better data structure for the parameters that group data in a way that makes sense for the specific application domain

```
class Point // In geometry, Point is a logical structure to group data
{
    public int x;
    public int y;
    public int z;
 };

void setCoordinates(Point p1, Point p2) {
    // ...
}
```

This rule raises an issue when a method has more parameters than the provided threshold.

Exceptions

Methods annotated with :

- Spring's `@RequestMapping` (and related shortcut annotations, like `@GetRequest`)
- JAX-RS API annotations (like `@javax.ws.rs.GET`)
- Bean constructor injection with `@org.springframework.beans.factory.annotation.Autowired`
- CDI constructor injection with `@javax.inject.Inject`
- `@com.fasterxml.jackson.annotation.JsonCreator`
- Micronaut's annotations (like `@io.micronaut.http.annotation.Get`)

may have a lot of parameters, encapsulation being possible. Therefore the rule ignores such methods.

Also, if a class annotated as a Spring component (like `@org.springframework.stereotype.Component`) has a single constructor, that constructor will be considered `@Autowired` and ignored by the rule.

Nested blocks of code should not be left empty (java:S108)

Severidad: MAJOR

Root_cause:

An empty code block is confusing. It will require some effort from maintainers to determine if it is intentional or indicates the implementation is incomplete.

```
for (int i = 0; i < 42; i++){}  // Noncompliant: is the block empty on purpose, or is code missing?
```

Removing or filling the empty code blocks takes away ambiguity and generally results in a more straightforward and less surprising code.

Exceptions

The rule ignores code blocks that contain comments unless they are `synchronized` blocks because these can affect program flow.

Magic numbers should not be used (java:S109)

Severidad: MAJOR

How_to_fix:

Replacing them with a constant allows us to provide a meaningful name associated with the value. Instead of adding complexity to the code, it brings clarity and helps to understand the context and the global meaning.

Noncompliant code example

```
public static void doSomething() {
  for (int i = 0; i < 4; i++) {  // Noncompliant, 4 is a magic number
    ...
  }
}
```

Compliant solution

```
public static final int NUMBER_OF_CYCLES = 4;
public static void doSomething() {
  for (int i = 0; i < NUMBER_OF_CYCLES ; i++) { // Compliant
    ...
  }
}
```

Introduction:

A magic number is a hard-coded numerical value that may lack context or meaning. They should not be used because they can make the code less readable and maintainable.

Root_cause:

Magic numbers make the code more complex to understand as it requires the reader to have knowledge about the global context to understand the number itself. Their usage may seem obvious when writing the code, but it may not be the case for another developer or later once the context faded away. -1, 0, and 1 are not considered magic numbers.

Exceptions

This rule ignores `hashCode` methods.

Control structures should use curly braces (java:S121)

Severidad: CRITICAL

Root_cause:

While not technically incorrect, the omission of curly braces can be misleading and may lead to the introduction of errors during maintenance.

In the following example, the two calls seem to be attached to the `if` statement, but only the first one is, and `checkSomething` will always be executed:

```
if (condition)  // Noncompliant
  executeSomething();
  checkSomething();
```

Adding curly braces improves the code readability and its robustness:

```
if (condition) {
  executeSomething();
  checkSomething();
}
```

The rule raises an issue when a control structure has no curly braces.

Exceptions

The rule doesn’t raise an issue when the body of an `if` statement is a single `return`, `break`, or `continue` and is on the same line.

Resources:

- CERT, EXP52-J. - Use braces for the body of an if, for, or while statement

Introduction:

Control structures are code statements that impact the program’s control flow (e.g., if statements, for loops, etc.)

"Externalizable" classes should have no-arguments constructors (java:S2060)

Severidad: MAJOR

Root_cause:

A class that implements `java.io.Externalizable` is a class that provides a way to customize the serialization and deserialization, allowing greater control over how the object’s state is written or read.

The first step of the deserialization process is to call the class' no-argument constructor before the `readExternal(ObjectInput in)` method.

An implicit default no-argument constructor exists on a class when no constructor is explicitly defined within the class. But this implicit constructor does not exist when any constructor is explicitly defined, and in this case, we should always ensure that one of the constructors has no-argument.

It is an issue if the implicit or explicit no-argument constructor is missing or not public, because the deserialization will fail and throw an `InvalidClassException: no valid constructor.`.

How_to_fix:

This issue can be fixed by:

- Adding an explicit public no-argument constructor.
- Or if all constructors can be removed, remove all constructors to benefit of the default implicit no-argument constructor.

Noncompliant code example

```
public class Tomato implements Externalizable {

  public Color color;

  // Noncompliant; because of this constructor there is no implicit no-argument constructor,
  // deserialization will fail
  public Tomato(Color color) {
    this.color = color;
  }

  @Override
  public void writeExternal(ObjectOutput out) throws IOException {
    out.writeUTF(color.name());
  }

  @Override
  public void readExternal(ObjectInput in) throws IOException {
    color = Color.valueOf(in.readUTF());
  }
}
```

Compliant solution

```
public class Tomato implements Externalizable {

  public Color color;

  // Compliant; deserialization will invoke this public no-argument constructor
  public Tomato() {
    this.color = Color.UNKNOWN;
  }

  public Tomato(Color color) {
    this.color = color;
  }

  @Override
  public void writeExternal(ObjectOutput out) throws IOException {
    out.writeUTF(color.name());
  }

  @Override
  public void readExternal(ObjectInput in) throws IOException {
    color = Color.valueOf(in.readUTF());
  }
}
```

Resources:

Documentation

- Oracle SDK - java.io.Externalizable

"wait" should not be called when multiple locks are held (java:S3046)

Severidad: BLOCKER

Root_cause:

When two locks are held simultaneously, a `wait` call only releases one of them. The other will be held until some other thread requests a lock on the awaited object. If no unrelated code tries to lock on that object, then all other threads will be locked out, resulting in a deadlock.

Noncompliant code example

```
synchronized (this.mon1) {  // threadB can't enter this block to request this.mon2 lock & release threadA
	synchronized (this.mon2) {
		this.mon2.wait();  // Noncompliant; threadA is stuck here holding lock on this.mon1
	}
}
```

Multiple loops over the same set should be combined (java:S3047)

Severidad: MINOR

Root_cause:

When a method loops multiple over the same set of data, whether it’s a list or a set of numbers, it is highly likely that the method could be made more efficient by combining the loops into a single set of iterations.

Noncompliant code example

```
public void doSomethingToAList(List<String> strings) {
  for (String str : strings) {
    doStep1(str);
  }
  for (String str : strings) {  // Noncompliant
    doStep2(str);
  }
}
```

Compliant solution

```
public void doSomethingToAList(List<String> strings) {
  for (String str : strings) {
    doStep1(str);
    doStep2(str);
  }
}
```

Passwords should not be stored in plaintext or with a fast hashing algorithm (java:S5344)

Severidad: CRITICAL

Introduction:

The improper storage of passwords poses a significant security risk to software applications. This vulnerability arises when passwords are stored in plaintext or with a fast hashing algorithm. To exploit this vulnerability, an attacker typically requires access to the stored passwords.

How_to_fix:

Noncompliant code example

The following code is vulnerable because it uses a legacy digest-based password encoding that is not considered secure.

```
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth, DataSource dataSource) throws Exception {
  auth.jdbcAuthentication()
    .dataSource(dataSource)
    .usersByUsernameQuery("SELECT * FROM users WHERE username = ?")
    .passwordEncoder(new StandardPasswordEncoder()); // Noncompliant
}
```

Compliant solution

```
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth, DataSource dataSource) throws Exception {
  auth.jdbcAuthentication()
    .dataSource(dataSource)
    .usersByUsernameQuery("SELECT * FROM users WHERE username = ?")
    .passwordEncoder(new BCryptPasswordEncoder());
}
```

How does this work?

Use secure password hashing algorithms

In general, you should rely on an algorithm that has no known security vulnerabilities. The MD5 and SHA-1 algorithms should not be used.

Some algorithms, such as the SHA family functions, are considered strong for some use cases, but are too fast in computation and therefore vulnerable to brute force attacks, especially with bruteforce-attack-oriented hardware.

To protect passwords, it is therefore important to choose modern, slow password-hashing algorithms. The following algorithms are, in order of strength, the most secure password hashing algorithms to date:

1. Argon2
2. scrypt
3. bcrypt
4. PBKDF2

Argon2 should be the best choice, and others should be used when the previous one is not available. For systems that must use FIPS-140-certified algorithms, PBKDF2 should be used.

Whenever possible, choose the strongest algorithm available. If the algorithm currently used by your system should be upgraded, OWASP documents possible upgrade methods here: Upgrading Legacy Hashes.

In the previous example, the `BCryptPasswordEncoder` is a password hashing function in Java that is designed to be secure and resistant to various types of attacks, including brute-force and rainbow table attacks. It is slow, adaptative, and automatically implements a salt.

Never store passwords in plaintext

A user password should never be stored in plaintext. Instead, a hash should be produced from it using a secure algorithm. When dealing with password storage security, best practices recommend relying on a slow hashing algorithm, that will make brute force attacks more difficult. Using a hashing function with adaptable computation and memory complexity also is recommended to be able to increase the security level with time.

Adding a salt to the digest computation is also recommended to prevent pre-computed table attacks (see rule S2053).

Pitfalls

Pre-hashing passwords

As bcrypt has a maximum length input length of 72 bytes for most implementations, some developers may be tempted to pre-hash the password with a stronger algorithm before hashing it with bcrypt.

Pre-hashing passwords with bcrypt is not recommended as it can lead to a specific range of issues. Using a strong salt and a high number of rounds is enough to protect the password.

More information about this can be found here: Pre-hashing Passwords with Bcrypt.

Resources:

Documentation

- Spring Framework Security Documentation - Class BCryptPasswordEncoder
- OWASP CheatSheet - Password Storage Cheat Sheet

Standards

- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- OWASP - Top 10 2021 Category A4 - Insecure Design
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- CWE - CWE-256 - Plaintext Storage of a Password
- CWE - CWE-916 - Use of Password Hash With Insufficient Computational Effort
- STIG Viewer - Application Security and Development: V-222542 - The application must only store cryptographic representations of passwords.

Root_cause:

Attackers who would get access to the stored passwords could reuse them without further attacks or with little additional effort.
Obtaining the plaintext passwords, they could then gain unauthorized access to user accounts, potentially leading to various malicious activities.

What is the potential impact?

Plaintext or weakly hashed password storage poses a significant security risk to software applications.

Unauthorized Access

When passwords are stored in plaintext or with weak hashing algorithms, an attacker who gains access to the password database can easily retrieve and use the passwords to gain unauthorized access to user accounts. This can lead to various malicious activities, such as unauthorized data access, identity theft, or even financial fraud.

Credential Reuse

Many users tend to reuse passwords across multiple platforms. If an attacker obtains plaintext or weakly hashed passwords, they can potentially use these credentials to gain unauthorized access to other accounts held by the same user. This can have far-reaching consequences, as sensitive personal information or critical systems may be compromised.

Regulatory Compliance

Many industries and jurisdictions have specific regulations and standards to protect user data and ensure its confidentiality. Storing passwords in plaintext or with weak hashing algorithms can lead to non-compliance with these regulations, potentially resulting in legal consequences, financial penalties, and damage to the reputation of the software application and its developers.

Allowing requests with excessive content length is security-sensitive (java:S5693)

Severidad: MAJOR

Assess_the_problem:

Ask Yourself Whether

- size limits are not defined for the different resources of the web application.
- the web application is not protected by rate limiting features.
- the web application infrastructure has limited resources.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

With default limit value of 8388608 (8MB).

A 100 MB file is allowed to be uploaded:

```
@Bean(name = "multipartResolver")
public CommonsMultipartResolver multipartResolver() {
  CommonsMultipartResolver multipartResolver = new CommonsMultipartResolver();
  multipartResolver.setMaxUploadSize(104857600); // Sensitive (100MB)
  return multipartResolver;
}

@Bean(name = "multipartResolver")
public CommonsMultipartResolver multipartResolver() {
  CommonsMultipartResolver multipartResolver = new CommonsMultipartResolver(); // Sensitive, by default if maxUploadSize property is not de
  return multipartResolver;
}

@Bean
public MultipartConfigElement multipartConfigElement() {
  MultipartConfigFactory factory = new MultipartConfigFactory(); // Sensitive, no limit by default
  return factory.createMultipartConfig();
}
```

How_to_fix:

Recommended Secure Coding Practices

- For most of the features of an application, it is recommended to limit the size of requests to:
  - lower or equal to 8mb for file uploads.
  - lower or equal to 2mb for other requests.

It is recommended to customize the rule with the limit values that correspond to the web application.

Compliant Solution

File upload size is limited to 8 MB:

```
@Bean(name = "multipartResolver")
public CommonsMultipartResolver multipartResolver() {
  multipartResolver.setMaxUploadSize(8388608); // Compliant (8 MB)
  return multipartResolver;
}
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- Owasp Cheat Sheet - Owasp Denial of Service Cheat Sheet
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- CWE - CWE-770 - Allocation of Resources Without Limits or Throttling
- CWE - CWE-400 - Uncontrolled Resource Consumption

Default:

Rejecting requests with significant content length is a good practice to control the network traffic intensity and thus resource consumption in order to prevent DoS attacks.

Ask Yourself Whether

- size limits are not defined for the different resources of the web application.
- the web application is not protected by rate limiting features.
- the web application infrastructure has limited resources.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- For most of the features of an application, it is recommended to limit the size of requests to:
  - lower or equal to 8mb for file uploads.
  - lower or equal to 2mb for other requests.

It is recommended to customize the rule with the limit values that correspond to the web application.

Sensitive Code Example

With default limit value of 8388608 (8MB).

A 100 MB file is allowed to be uploaded:

```
@Bean(name = "multipartResolver")
public CommonsMultipartResolver multipartResolver() {
  CommonsMultipartResolver multipartResolver = new CommonsMultipartResolver();
  multipartResolver.setMaxUploadSize(104857600); // Sensitive (100MB)
  return multipartResolver;
}

@Bean(name = "multipartResolver")
public CommonsMultipartResolver multipartResolver() {
  CommonsMultipartResolver multipartResolver = new CommonsMultipartResolver(); // Sensitive, by default if maxUploadSize property is not de
  return multipartResolver;
}

@Bean
public MultipartConfigElement multipartConfigElement() {
  MultipartConfigFactory factory = new MultipartConfigFactory(); // Sensitive, no limit by default
  return factory.createMultipartConfig();
}
```

Compliant Solution

File upload size is limited to 8 MB:

```
@Bean(name = "multipartResolver")
public CommonsMultipartResolver multipartResolver() {
  multipartResolver.setMaxUploadSize(8388608); // Compliant (8 MB)
  return multipartResolver;
}
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- Owasp Cheat Sheet - Owasp Denial of Service Cheat Sheet
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- CWE - CWE-770 - Allocation of Resources Without Limits or Throttling
- CWE - CWE-400 - Uncontrolled Resource Consumption

Root_cause:

Rejecting requests with significant content length is a good practice to control the network traffic intensity and thus resource consumption in order to prevent DoS attacks.

Credentials should not be hard-coded (java:S6437)

Severidad: BLOCKER

Resources:

Documentation

- AWS Documentation - What is AWS Secrets Manager
- Azure Documentation - Azure Key Vault
- Google Cloud - Secret Manager documentation
- HashiCorp Developer - Vault Documentation

Standards

- OWASP - [Top 10 2021 - Category A7 - Identification and Authentication Failures]
- OWASP - [Top 10 2017 - Category A2 - Broken Authentication]
- CWE - [CWE-798 - Use of Hard-coded Credentials]
- CWE - [CWE-259 - Use of Hard-coded Password]

How_to_fix:

Revoke the secret

Revoke any leaked secrets and remove them from the application source code.

Before revoking the secret, ensure that no other applications or processes are using it. Other usages of the secret will also be impacted when the secret is revoked.

Analyze recent secret use

When available, analyze authentication logs to identify any unintended or malicious use of the secret since its disclosure date. Doing this will allow determining if an attacker took advantage of the leaked secret and to what extent.

This operation should be part of a global incident response process.

Use a secret vault

A secret vault should be used to generate and store the new secret. This will ensure the secret's security and prevent any further unexpected disclosure.

Depending on the development platform and the leaked secret type, multiple solutions are currently available.

The following code example is noncompliant because it uses a hardcoded secret value.

Noncompliant code example

```
import org.h2.security.SHA256;

String inputString = "s3cr37";
byte[] key         = inputString.getBytes();

SHA256.getHMAC(key, message);  // Noncompliant
```

Compliant solution

```
import org.h2.security.SHA256;

String inputString = System.getenv("SECRET");
byte[] key         = inputString.getBytes();

SHA256.getHMAC(key, message);
```

How does this work?

While the noncompliant code example contains a hard-coded password, the compliant solution retrieves the secret's value from its environment. This allows to have an environment-dependent secret value and avoids storing the password in the source code itself.

Depending on the application and its underlying infrastructure, how the secret gets added to the environment might change.

Introduction:

Secret leaks often occur when a sensitive piece of authentication data is stored with the source code of an application. Considering the source code is intended to be deployed across multiple assets, including source code repositories or application hosting servers, the secrets might get exposed to an unintended audience.

Root_cause:

In most cases, trust boundaries are violated when a secret is exposed in a source code repository or an uncontrolled deployment environment. Unintended people who don't need to know the secret might get access to it. They might then be able to use it to gain unwanted access to associated services or resources.

The trust issue can be more or less severe depending on the people's role and entitlement.

What is the potential impact?

The consequences vary greatly depending on the situation and the secret-exposed audience. Still, two main scenarios should be considered.

Financial loss

Financial losses can occur when a secret is used to access a paid third-party-provided service and is disclosed as part of the source code of client applications. Having the secret, each user of the application will be able to use it without limit to use the third party service to their own need, including in a way that was not expected.

This additional use of the secret will lead to added costs with the service provider.

Moreover, when rate or volume limiting is set up on the provider side, this additional use can prevent the regular operation of the affected application. This might result in a partial denial of service for all the application’s users.

Application’s security downgrade

A downgrade can happen when the disclosed secret is used to protect security-sensitive assets or features of the application. Depending on the affected asset or feature, the practical impact can range from a sensitive information leak to a complete takeover of the application, its hosting server or another linked component.

For example, an application that would disclose a secret used to sign user authentication tokens would be at risk of user identity impersonation. An attacker accessing the leaked secret could sign session tokens for arbitrary users and take over their privileges and entitlements.

Package names should comply with a naming convention (java:S120)

Severidad: MINOR

How_to_fix:

Rename packages with the expected naming convention

Noncompliant code example

With the default regular expression `^[a-z_]+(\.[a-z_][a-z0-9_]*)*$`:

```
package org.exAmple; // Noncompliant
```

Compliant solution

```
package org.example;
```

Root_cause:

Shared naming conventions improve readability and allow teams to collaborate efficiently. This rule checks that all package names match a provided regular expression.

Counter Mode initialization vectors should not be reused (java:S6432)

Severidad: CRITICAL

Resources:

Standards

- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP - Mobile AppSec Verification Standard - Cryptography Requirements
- OWASP - Mobile Top 10 2016 Category M5 - Insufficient Cryptography
- CWE - CWE-323 - Reusing a Nonce, Key Pair in Encryption
- NIST, SP-800-38A - Recommendation for Block Cipher Modes of Operation
- NIST, SP-800-38C - Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality
- NIST, SP-800-38D - Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC

How_to_fix:

The example uses a hardcoded IV as a nonce, which causes AES-CCM to be insecure. To fix it, a nonce is randomly generated instead.

Noncompliant code example

```
public void encrypt(byte[] key, byte[] ptxt) {
    byte[] nonce = "7cVgr5cbdCZV".getBytes(StandardCharsets.UTF_8);

    BlockCipher engine = new AESEngine();
    AEADParameters params = new AEADParameters(new KeyParameter(key), 128, nonce);
    CCMBlockCipher cipher = new CCMBlockCipher(engine);

    cipher.init(true, params); // Noncompliant
}
```

Compliant solution

```
public void encrypt(byte[] key, byte[] ptxt) {
    SecureRandom random = new SecureRandom();
    byte[] nonce = new byte[12];
    random.nextBytes(nonce);

    BlockCipher engine = new AESEngine();
    AEADParameters params = new AEADParameters(new KeyParameter(key), 128, nonce);
    CCMBlockCipher cipher = new CCMBlockCipher(engine);

    cipher.init(true, params);
}
```

How does this work?

For AES-GCM and AES-CCM, NIST recommends generating a nonce using either a deterministic approach or using a 'Random Bit Generator (RBG)'.

Generating nonces using random number generation

When using a randomized approach, NIST recommends a nonce of at least 96 bits using a cryptographically secure pseudorandom number generator (CSPRNG.) Such a generator can create output with a sufficiently low probability of the same number being output twice (also called a *collision*) for a long time. However, after 2^{32} generated numbers for the same key, NIST recommends rotating this key for a new one. After that amount of generated numbers, the probability of a collision is high enough to be considered insecure.

The code example above demonstrates how CSPRNGs can be used to generate nonces.

Be careful to use a random number generator that is sufficiently secure. Default (non-cryptographically secure) RNGs might be more prone to collisions in their output, which is catastrophic for counter-based encryption modes.

Deterministically generating nonces

One method to prevent the same IV from being used multiple times for the same key is to update the IV in a deterministic way after each encryption. The most straightforward deterministic method for this is a counter.

The way this works is simple: for any key, the first IV is the number zero. After this IV is used to encrypt something with a key, it is incremented for that key (and is now equal to 1). Although this requires additional bookkeeping, it should guarantee that for each encryption key, an IV is never repeated.

For a secure implementation, NIST suggests generating these nonces in two parts: a fixed field and an invocation field. The fixed field should be used to identify the device executing the encryption (for example, it could contain a device ID), such that for one key, no two devices can generate the same nonce. The invocation field contains the counter as described above. For a 96-bit nonce, NIST recommends (but does not require) using a 32-bit fixed field and a 64-bit invocation field. Additional details can be found in the NIST Special Publication 800-38D.

How_to_fix:

The example uses a hardcoded IV as a nonce, which causes AES-CCM to be insecure. To fix it, a nonce is randomly generated instead.

Noncompliant code example

```
public void encrypt(byte[] key, byte[] ptxt) {
    byte[] nonce = "7cVgr5cbdCZV".getBytes("UTF-8");

    Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");
    SecretKeySpec keySpec = new SecretKeySpec(key, "AES");
    GCMParameterSpec gcmSpec = new GCMParameterSpec(128, nonce);

    cipher.init(Cipher.ENCRYPT_MODE, keySpec, gcmSpec); // Noncompliant
}
```

Compliant solution

```
public void encrypt(byte[] key, byte[] ptxt) {
    SecureRandom random = new SecureRandom();
    byte[] nonce = new byte[12];
    random.nextBytes(nonce);

    Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");
    SecretKeySpec keySpec = new SecretKeySpec(key, "AES");
    GCMParameterSpec gcmSpec = new GCMParameterSpec(128, nonce);
    cipher.init(Cipher.ENCRYPT_MODE, keySpec, gcmSpec);
}
```

How does this work?

For AES-GCM and AES-CCM, NIST recommends generating a nonce using either a deterministic approach or using a 'Random Bit Generator (RBG)'.

Generating nonces using random number generation

When using a randomized approach, NIST recommends a nonce of at least 96 bits using a cryptographically secure pseudorandom number generator (CSPRNG.) Such a generator can create output with a sufficiently low probability of the same number being output twice (also called a *collision*) for a long time. However, after 2^{32} generated numbers for the same key, NIST recommends rotating this key for a new one. After that amount of generated numbers, the probability of a collision is high enough to be considered insecure.

The code example above demonstrates how CSPRNGs can be used to generate nonces.

Be careful to use a random number generator that is sufficiently secure. Default (non-cryptographically secure) RNGs might be more prone to collisions in their output, which is catastrophic for counter-based encryption modes.

Deterministically generating nonces

One method to prevent the same IV from being used multiple times for the same key is to update the IV in a deterministic way after each encryption. The most straightforward deterministic method for this is a counter.

The way this works is simple: for any key, the first IV is the number zero. After this IV is used to encrypt something with a key, it is incremented for that key (and is now equal to 1). Although this requires additional bookkeeping, it should guarantee that for each encryption key, an IV is never repeated.

For a secure implementation, NIST suggests generating these nonces in two parts: a fixed field and an invocation field. The fixed field should be used to identify the device executing the encryption (for example, it could contain a device ID), such that for one key, no two devices can generate the same nonce. The invocation field contains the counter as described above. For a 96-bit nonce, NIST recommends (but does not require) using a 32-bit fixed field and a 64-bit invocation field. Additional details can be found in the NIST Special Publication 800-38D.

Introduction:

When encrypting data using AES-GCM or AES-CCM, it is essential not to reuse the same initialization vector (IV, also called nonce) with a given key. To prevent this, it is recommended to either randomize the IV for each encryption or increment the IV after each encryption.

Root_cause:

When encrypting data using a counter (CTR) derived block cipher mode of operation, it is essential not to reuse the same initialization vector (IV) for a given key. An IV that complies with this requirement is called a "nonce" (**n**umber used **once**). Galois/Counter (GCM) and Counter with Cipher Block Chaining-Message Authentication Code (CCM) are both derived from counter mode.

When using AES-GCM or AES-CCM, a given key and IV pair will create a "keystream" that is used to encrypt a plaintext (original content) into a ciphertext (encrypted content.) For any key and IV pair, this keystream is always deterministic. Because of this property, encrypting several plaintexts with one key and IV pair can be catastrophic. If an attacker has access to one plaintext and its associated ciphertext, they are able to decrypt everything that was created using the same pair.

Additionally, IV reuse also drastically decreases the key recovery computational complexity by downgrading it to a simpler polynomial root-finding problem. This means that even without access to a plaintext/ciphertext pair, an attacker may still be able to decrypt all the sensitive data.

What is the potential impact?

If the encryption that is being used is flawed, attackers might be able to exploit it in several ways. They might be able to decrypt existing sensitive data or bypass key protections.

Below are some real-world scenarios that illustrate some impacts of an attacker exploiting the vulnerability.

Theft of sensitive data

The encrypted message might contain data that is considered sensitive and should not be known to third parties.

By not using the encryption algorithm correctly, the likelihood that an attacker might be able to recover the original sensitive data drastically increases.

Additional attack surface

Encrypted values are often considered trusted, since under normal circumstances it would not be possible for a third party to modify them. If an attacker is able to modify the cleartext of the encrypted message, it might be possible to trigger other vulnerabilities in the code.

Interface names should comply with a naming convention (java:S114)

Severidad: MINOR

Root_cause:

Sharing some naming conventions is a key point to make it possible for a team to efficiently collaborate. This rule allows to check that all interface names match a provided regular expression.

Noncompliant code example

With the default regular expression `^[A-Z][a-zA-Z0-9]*$`:

```
public interface myInterface {...} // Noncompliant
```

Compliant solution

```
public interface MyInterface {...}
```

Statements should be on separate lines (java:S122)

Severidad: MAJOR

Root_cause:

Putting multiple statements on a single line lowers the code readability and makes debugging the code more complex.

```
if (someCondition) doSomething(); // Noncompliant
```

Write one statement per line to improve readability.

```
if (someCondition) {
  doSomething();
}
```

Formatting SQL queries is security-sensitive (java:S2077)

Severidad: MAJOR

Assess_the_problem:

Ask Yourself Whether

- Some parts of the query come from untrusted values (like user inputs).
- The query is repeated/duplicated in other parts of the code.
- The application must support different types of relational databases.

There is a risk if you answered yes to any of those questions.

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
  // ...
}
```

How_to_fix:

Recommended Secure Coding Practices

- Use parameterized queries, prepared statements, or stored procedures and bind variables to SQL query parameters.
- Consider using ORM frameworks if there is a need to have an abstract layer to access data.

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 2021 Category A3 - Injection
- OWASP - Top 10 2017 Category A1 - Injection
- CWE - CWE-89 - Improper Neutralization of Special Elements used in an SQL Command
- CWE - CWE-564 - SQL Injection: Hibernate
- CWE - CWE-20 - Improper Input Validation
- CWE - CWE-943 - Improper Neutralization of Special Elements in Data Query Logic
- CERT, IDS00-J. - Prevent SQL injection
- Derived from FindSecBugs rules Potential SQL/JPQL Injection (JPA), Potential SQL/JDOQL Injection (JDO), Potential SQL/HQL Injection (Hibernate)

Root_cause:

Formatted SQL queries can be difficult to maintain, debug and can increase the risk of SQL injection when concatenating untrusted values into the query. However, this rule doesn’t detect SQL injections (unlike rule S3649), the goal is only to highlight complex/formatted queries.

Default:

Formatted SQL queries can be difficult to maintain, debug and can increase the risk of SQL injection when concatenating untrusted values into the query. However, this rule doesn’t detect SQL injections (unlike rule S3649), the goal is only to highlight complex/formatted queries.

Ask Yourself Whether

- Some parts of the query come from untrusted values (like user inputs).
- The query is repeated/duplicated in other parts of the code.
- The application must support different types of relational databases.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- Use parameterized queries, prepared statements, or stored procedures and bind variables to SQL query parameters.
- Consider using ORM frameworks if there is a need to have an abstract layer to access data.

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 2021 Category A3 - Injection
- OWASP - Top 10 2017 Category A1 - Injection
- CWE - CWE-89 - Improper Neutralization of Special Elements used in an SQL Command
- CWE - CWE-564 - SQL Injection: Hibernate
- CWE - CWE-20 - Improper Input Validation
- CWE - CWE-943 - Improper Neutralization of Special Elements in Data Query Logic
- CERT, IDS00-J. - Prevent SQL injection
- Derived from FindSecBugs rules Potential SQL/JPQL Injection (JPA), Potential SQL/JDOQL Injection (JDO), Potential SQL/HQL Injection (Hibernate)

Modulus results should not be checked for direct equality (java:S2197)

Severidad: CRITICAL

Root_cause:

When the modulus of a negative number is calculated, the result will either be negative or zero. Thus, comparing the modulus of a variable for equality with a positive number (or a negative one) could result in unexpected results.

Noncompliant code example

```
public boolean isOdd(int x) {
  return x % 2 == 1;  // Noncompliant; if x is an odd negative, x % 2 == -1
}
```

Compliant solution

```
public boolean isOdd(int x) {
  return x % 2 != 0;
}
```

Resources:

- CERT, NUM51-J. - Do not assume that the remainder operator always returns a nonnegative result for integral operands
- CERT, INT10-C - Do not assume a positive remainder when using the % operator

---

"writeObject" should not be the only "synchronized" code in a class (java:S3042)

Severidad: MAJOR

How_to_fix:

Consider whether this class is used in a multithreaded context. If it is, ask yourself whether other methods in this class should also be marked as `synchronized`. Otherwise, remove the `synchronized` modifier from this method.

Noncompliant code example

```
public class RubberBall implements Serializable {

  private Color color;
  private int diameter;

  public RubberBall(Color color, int diameter) {
    // ...
  }

  public void bounce(float angle, float velocity) {
    // ...
  }

  private synchronized void writeObject(ObjectOutputStream stream) throws IOException { // Noncompliant, "writeObject" is the only synchron
    // ...
  }
}
```

Compliant solution

```
public class RubberBall implements Serializable {

  private Color color;
  private int diameter;

  public RubberBall(Color color, int diameter) {
    // ...
  }

  public void bounce(float angle, float velocity) {
    // ...
  }

  private void writeObject(ObjectOutputStream stream) throws IOException { // Compliant, no methods in this class are synchronized
    // ...
  }
}
```

Root_cause:

Synchronization is a mechanism used when multithreading in Java to ensure that only one thread executes a given block of code at a time. This is done to avoid bugs that can occur when multiple threads share a given state and try to manipulate simultaneously.

Object serialization is not thread-safe by default. In a multithreaded environment, one option is to mark `writeObject` with `synchronized` to improve thread safety. It is highly suspicious, however, if `writeObject` is the only `synchronized` method in a class. It may indicate that serialization is not required, as multithreading is not used. Alternatively, it could also suggest that other methods in the same class have been forgotten to be made thread-safe.

Resources:

- Java SE 17 & JDK 17 - Serializable Javadoc
- The Java™ Tutorials - Synchronized Methods

Files should end with a newline (java:S113)

Severidad: MINOR

Root_cause:

Some tools work better when files end with a newline.

This rule simply generates an issue if it is missing.

For example, a Git diff looks like this if the empty line is missing at the end of the file:

```
+class Test {
+}
\ No newline at end of file
```

Constant names should comply with a naming convention (java:S115)

Severidad: CRITICAL

Introduction:

Constants should be named consistently to communicate intent and improve maintainability. Rename your constants to follow your project's naming convention to address this issue.

Root_cause:

Constants are variables whose value does not change during the runtime of a program after initialization. Oftentimes, constants are used in multiple locations across different subroutines.

It is important that the names of constants follow a consistent and easily recognizable pattern. This way, readers immediately understand that the referenced value does not change, which simplifies debugging.

Or, in the case of primitive constants, that accessing the constant is thread-safe.

This rule checks that all constant names match a given regular expression.

What is the potential impact?

Ignoring the naming convention for constants makes the code less readable since constants and variables are harder to tell apart. Code that is hard to understand is also difficult to maintain between different team members.

How_to_fix:

First, familiarize yourself with the particular naming convention of the project in question. Then, update the name of the constant to match the convention, as well as all usages of the name. For many IDEs, you can use built-in renaming and refactoring features to update all usages of a constant at once.

Noncompliant code example

The following example assumes that constant names should match the default regular expression `^[A-Z][A-Z0-9]*(_[A-Z0-9]+)*$`:

```
public class MyClass {
  public static final float pi = 3.14159f; // Noncompliant: Constant is not capitalized

  void myMethod() {
    System.out.println(pi);
  }
}

public enum MyEnum {
  optionOne, // Noncompliant
  optionTwo; // Noncompliant
}
```

Compliant solution

```
public class MyClass {
  public static final float PI = 3.14159f;

  void myMethod() {
    System.out.println(PI);
  }
}

public enum MyEnum {
  OPTION_ONE,
  OPTION_TWO;
}
```

Exceptions

The rule applies to fields of primitive types (for example, `float`), boxed primitives (`Float`), and Strings. We do not apply it to other types, which can be mutated, or have methods with side effects.

```
public static final Logger log = getLogger(MyClass.class);
public static final List<Integer> myList = new ArrayList<>();

// call with side-effects
log.info("message")

// mutating an object
myList.add(28);
```

Resources:

External coding guidelines

- The Google Java Style Guide on Constant Names.

Field names should comply with a naming convention (java:S116)

Severidad: MINOR

Resources:

Documentation

- Oracle - Java SE Naming Conventions
- Geeksforgeeks - Java Naming Conventions
- Wikipedia - Naming Convention (programming)

Root_cause:

A naming convention in software development is a set of guidelines for naming code elements like variables, functions, and classes.

The goal of a naming convention is to make the code more readable and understandable, which makes it easier to maintain and debug. It also ensures consistency in the code, especially when multiple developers are working on the same project.

This rule checks that field names match a provided regular expression.

Using the regular expression `^[a-z][a-zA-Z0-9]*$`, the noncompliant code below:

```
class MyClass {
   private int my_field;
}
```

Should be replaced with:

```
class MyClass {
   private int myField;
}
```

Local variable and method parameter names should comply with a naming convention (java:S117)

Severidad: MINOR

Root_cause:

A naming convention in software development is a set of guidelines for naming code elements like variables, functions, and classes.
Local variables and method parameters hold the meaning of the written code. Their names should be meaningful and follow a consistent and easily

recognizable pattern.
Adhering to a consistent naming convention helps to make the code more readable and understandable, which makes it easier to maintain and debug. It also ensures consistency in the code, especially when multiple developers are working on the same project.

This rule checks that local variable and method parameter names match a provided regular expression.

What is the potential impact?

Inconsistent naming of local variables and method parameters can lead to several issues in your code:

- **Reduced Readability**: Inconsistent local variable and method parameter names make the code harder to read and understand; consequently, it is more difficult to identify the purpose of each variable, spot errors, or comprehend the logic.
- **Difficulty in Identifying Variables**: The local variables and method parameters that don't adhere to a standard naming convention are challenging to identify; thus, the coding process slows down, especially when dealing with a large codebase.
- **Increased Risk of Errors**: Inconsistent or unclear local variable and method parameter names lead to misunderstandings about what the variable represents. This ambiguity leads to incorrect assumptions and, consequently, bugs in the code.
- **Collaboration Difficulties**: In a team setting, inconsistent naming conventions lead to confusion and miscommunication among team members.
- **Difficulty in Code Maintenance**: Inconsistent naming leads to an inconsistent codebase. The code is difficult to understand, and making changes feels like refactoring constantly, as you face different naming methods. Ultimately, it makes the codebase harder to maintain.

In summary, not adhering to a naming convention for local variables and method parameters can lead to confusion, errors, and inefficiencies, making the code harder to read, understand, and maintain.

Exceptions

Loop counters are ignored by this rule.

```
for (int i_1 = 0; i_1 < limit; i_1++) {  // Compliant
  // ...
}
```

as well as one-character `catch` variables:

```
try {
//...
} catch (Exception e) { // Compliant
}
```

How_to_fix:

First, familiarize yourself with the particular naming convention of the project in question. Then, update the name to match the convention, as well as all usages of the name. For many IDEs, you can use built-in renaming and refactoring features to update all usages at once.

Noncompliant code example

With the default regular expression `^[a-z][a-zA-Z0-9]*$`:

```
public class MyClass {
    public void doSomething(int myParam) {
      int LOCAL;    // Noncompliant
      // ...
    }
}
```

Compliant solution

```
public class MyClass {
    public void doSomething(int my_param) {
      int local;
      // ...
    }
}
```

Introduction:

Local variables and method parameters should be named consistently to communicate intent and improve maintainability. Rename your local variable or method parameter to follow your project's naming convention to address this issue.

Resources:

Documentation

- Oracle - [Java SE Naming Conventions]

- Geeksforgeeks - Java Naming Conventions
- Wikipedia - Naming Convention (programming)

Related rules

- S100 - Method names should comply with a naming convention
- S101 - Class names should comply with a naming convention
- S114 - Interface names should comply with a naming convention
- S115 - Constant names should comply with a naming convention
- S116 - Field names should comply with a naming convention
- S118 - Abstract class names should comply with a naming convention
- S119 - Type parameter names should comply with a naming convention
- S120 - Package names should comply with a naming convention
- S1312 - Loggers should be "private static final" and should share a naming convention
- S3008 - Static non-final field names should comply with a naming convention
- S3577 - Test classes should comply with a naming convention
- S3578 - Test methods should comply with a naming convention
- S4174 - Local constants should follow naming conventions for constants

---

Abstract class names should comply with a naming convention (java:S118)

Severidad: MINOR

Root_cause:

Sharing some naming conventions is a key point to make it possible for a team to efficiently collaborate. This rule allows to check that all `abstract` class names match a provided regular expression. If a non-abstract class match the regular expression, an issue is raised to suggest to either make it abstract or to rename it.

Noncompliant code example

With the default regular expression: `^Abstract[A-Z][a-zA-Z0-9]*$`:

```
abstract class MyClass { // Noncompliant
}

class AbstractLikeClass { // Noncompliant
}
```

Compliant solution

```
abstract class AbstractClass {
}

class LikeClass {
}
```

Type parameter names should comply with a naming convention (java:S119)

Severidad: MINOR

Root_cause:

Shared naming conventions make it possible for a team to collaborate efficiently. Following the established convention of single-letter type parameter names helps users and maintainers of your code quickly see the difference between a type parameter and a poorly named class.

This rule check that all type parameter names match a provided regular expression. The following code snippets use the default regular expression.

Noncompliant code example

```
public class MyClass<TYPE> { // Noncompliant
  <TYPE> void method(TYPE t) { // Noncompliant
  }
}
```

Compliant solution

```
public class MyClass<T> {
  <T> void method(T t) {
  }
}
```

The names of methods with boolean return values should start with "is" or "has" (java:S2047)

Severidad: MAJOR

Root_cause:

Well-named functions can allow the users of your code to understand at a glance what to expect from the function - even before reading the documentation. Toward that end, methods returning a boolean should have names that start with "is" or "has" rather than with "get".

Noncompliant code example

```
public boolean getFoo() { // Noncompliant
  // ...
}

public boolean getBar(Bar c) { // Noncompliant
  // ...
}

public boolean testForBar(Bar c) { // Compliant - The method does not start by 'get'.
  // ...
}
```

Compliant solution

```
public boolean isFoo() {
  // ...
}

public boolean hasBar(Bar c) {
  // ...
}

public boolean testForBar(Bar c) {
  // ...
}
```

Exceptions

Overriding methods are excluded.

```
@Override
public boolean getFoo(){
  // ...
}
```

Double-checked locking should not be used (java:S2168)

Severidad: BLOCKER

Resources:

- The "Double-Checked Locking is Broken" Declaration
- CERT, LCK10-J. - Use a correct form of the double-checked locking idiom
- CWE - CWE-609 - Double-checked locking
- JLS 12.4 - Initialization of Classes and Interfaces
- Wikipedia: Double-checked locking

How_to_fix:

Given significant performance improvements of `synchronized` methods in recent JVM versions, `synchronized` methods are now preferred over the less robust double-checked locking.

If marking the entire method as `synchronized` is not an option, consider using an inner `static class` to hold the reference instead. Inner static classes are guaranteed to be initialized lazily.

Noncompliant code example

```
public class ResourceFactory {
    private static Resource resource;

    public static Resource getInstance() {
        if (resource == null) {
            synchronized (DoubleCheckedLocking.class) { // Noncompliant, not thread-safe due to the use of double-checked locking.
```

```
                if (resource == null)
                    resource = new Resource();
            }
        }
        return resource;
    }
}
```

Compliant solution

```
public class ResourceFactory {
    private static Resource resource;

    public static synchronized Resource getInstance() { // Compliant, the entire method is synchronized and hence thread-safe
        if (resource == null)
            resource = new Resource();
        return resource;
    }
}
```

Compliant solution

Alternatively, a static inner class can be used. However, this solution is less explicit in its intention and hence should be used with care.

```
public class ResourceFactory {
    private static class ResourceHolder {
        public static Resource resource = new Resource(); // Compliant, as this will be lazily initialised by the JVM
    }

    public static Resource getResource() {
        return ResourceFactory.ResourceHolder.resource;
    }
}
```

Root_cause:

Double-checked locking is the practice of checking a lazy-initialized object’s state both before and after a `synchronized` block is entered to determine whether to initialize the object. In early JVM versions, synchronizing entire methods was not performant, which sometimes caused this practice to be used in its place.

Apart from `float` and `int` types, this practice does not work reliably in a platform-independent manner without additional synchronization of mutable instances. Using double-checked locking for the lazy initialization of any other type of primitive or mutable object risks a second thread using an uninitialized or partially initialized member while the first thread is still creating it. The results can be unexpected, potentially even causing the application to crash.

Switches should be used for sequences of simple "String" tests (java:S2196)

Severidad: MINOR

Root_cause:

Since Java 7, `Strings` can be used as `switch` arguments. So when a single `String` is tested against three or more values in an `if`/`else if` structure, it should be converted to a switch instead for greater readability.

Note that this rule is automatically disabled when the project’s `sonar.java.source` is lower than `7`.

Noncompliant code example

```
if ("red".equals(choice)) {  // Noncompliant
  dispenseRed();
} else if ("blue".equals(choice)) {
  dispenseBlue();
} else if ("yellow".equals(choice)) {
  dispenseYellow();
} else {
  promptUser();
}
```

Compliant solution

```
switch(choice) {
  case "Red":
    dispenseRed();
    break;
  case "Blue":
    dispenseBlue():
    break;
  case "Yellow":
    dispenseYellow();
    break;
  default:
    promptUser();
    break;
}
```

Track uses of disallowed constructors (java:S4011)

Severidad: MAJOR

Root_cause:

This rule allows banning usage of certain constructors.

Noncompliant code example

Given parameters:

- className: java.util.Date
- argumentTypes: java.lang.String

```
Date birthday;
birthday = new Date("Sat Sep 27 05:42:21 EDT 1986");  // Noncompliant
birthday = new Date(528176541000L); // Compliant
```

Secure random number generators should not output predictable values (java:S4347)

Severidad: CRITICAL

Root_cause:

Random number generators are often used to generate random values for cryptographic algorithms. When a random number generator is used for cryptographic purposes, the generated numbers must be as random and unpredictable as possible. When the random number generator is improperly seeded with a constant or a predictable value, its output will also be predictable.

This can have severe security implications for cryptographic operations that rely on the randomness of the generated numbers. By using a predictable seed, an attacker can potentially guess or deduce the generated numbers, compromising the security of whatever cryptographic algorithm relies on the random number generator.

What is the potential impact?

It is crucial to understand that the strength of cryptographic algorithms heavily relies on the quality of the random numbers used. By improperly seeding a CSPRNG, we introduce a significant weakness that can be exploited by attackers.

Insecure cryptographic keys

One of the primary use cases for CSPRNGs is generating cryptographic keys. If an attacker can predict the seed used to initialize the random number generator, they may be able to derive the same keys. Depending on the use case, this can lead to multiple severe outcomes, such as:

- Being able to decrypt sensitive documents, leading to privacy breaches or identity theft.
- Gaining access to a private key used for signing, allowing an attacker to forge digital signatures and impersonate legitimate entities.
- Bypassing authentication mechanisms that rely on public-key infrastructure (PKI), which can be abused to gain unauthorized access to systems or networks.

Session hijacking and man-in-the-middle attack

Another scenario where this vulnerability can be exploited is in the generation of session tokens or nonces for secure communication protocols. If an attacker can predict the seed used to generate these tokens, they can impersonate legitimate users or intercept sensitive information.

Resources:

Documentation

- Java Documentation - [Class `java.security.SecureRandom`]

Standards

- OWASP - [Top 10 2021 Category A2 - Cryptographic Failures]
- OWASP - [Top 10 2017 Category A6 - Security Misconfiguration]

- CWE - CWE-330 - Use of Insufficiently Random Values
- CWE - CWE-332 - Insufficient Entropy in PRNG
- CWE - CWE-336 - Same Seed in Pseudo-Random Number Generator (PRNG)
- CWE - CWE-337 - Predictable Seed in Pseudo-Random Number Generator (PRNG)
- CERT, MSC63J. - Ensure that SecureRandom is properly seeded

How_to_fix:

The following code uses a cryptographically strong random number generator to generate data that is not cryptographically strong.

Noncompliant code example

```
SecureRandom sr = new SecureRandom();
sr.setSeed(123456L); // Noncompliant
int v = sr.next(32);

SecureRandom sr = new SecureRandom("abcdefghijklmnop".getBytes("us-ascii")); // Noncompliant
int v = sr.next(32);
```

Compliant solution

```
SecureRandom sr = new SecureRandom();
int v = sr.next(32);
```

This solution is available for JDK 1.8 and higher.

```
SecureRandom sr = SecureRandom.getInstanceStrong();
int v = sr.next(32);
```

How does this work?

When the randomly generated data needs to be cryptographically strong, `SecureRandom` is the correct class to use. However, its documentation also cites that "any seed material passed to a `SecureRandom` object must be unpredictable". When no seed is passed by the user to the object, the `SecureRandom` object chooses an unpredictable seed by default. Therefore, the easiest way to fix the issue is to use the default constructor without any calls to `SecureObject.setSeed()`.

To go the extra mile, `SecureObject.getInstanceStrong()` returns an instance of `SecureObject` that is guaranteed to use a strong algorithm for its number generation.

If the randomly generated data is not used for cryptographic purposes and is not business critical, it may be a better choice to use `java.util.Random` instead. In this case, setting a predictable seed may be acceptable depending on the situation.

Introduction:

Cryptographic operations often rely on unpredictable random numbers to enhance security. These random numbers are created by cryptographically secure pseudo-random number generators (CSPRNG). It is important not to use a predictable seed with these random number generators otherwise the random numbers will also become predictable.

"iterator" should not return "this" (java:S4348)

Severidad: MAJOR

How_to_fix:

Noncompliant code example

```
class FooIterator implements Iterator<Foo>, Iterable<Foo> {
  private Foo[] seq;
  private int idx = 0;

  public boolean hasNext() {
    return idx < seq.length;
  }

  public Foo next() {
    return seq[idx++];
  }

  public Iterator<Foo> iterator() {
    return this; // Noncompliant
  }
  // ...
}
```

Compliant solution

```
class FooSequence implements Iterable<Foo> {
  private Foo[] seq;

  public Iterator<Foo> iterator() {
    return new Iterator<Foo>() { // Compliant
      private int idx = 0;

      public boolean hasNext() {
        return idx < seq.length;
      }

      public Foo next() {
        return seq[idx++];
      }
    };
  }
  // ...
}
```

Root_cause:

An `Iterable` should not implement the `Iterator` interface or return `this` as an `Iterator`. The reason is that `Iterator` represents the iteration process itself, while `Iterable` represents the object we want to iterate over.

The `Iterator` instance encapsulates state information of the iteration process, such as the current and next element. Consequently, distinct iterations require distinct `Iterator` instances, for which `Iterable` provides the factory method `Iterable.iterator()`.

This rule raises an issue when the `Iterable.iterator()` of a class implementing both `Iterable` and `Iterator` returns `this`.

What is the potential impact?

The `Iterable.iterator()` method returning the same `Iterator` instance many times would have the following effects:

1. For subsequent iterations, e.g., two subsequent `for` loops with iterators over the same object, only the first one would iterate, and the others would do nothing.
2. For nested iterations over the same object, the different iteration processes would affect each other because they only have a common, shared state.

Resources:

Documentation

- Java SE 7 API Specification: java.lang.Iterable
- Java SE 7 API Specification: java.util.Iterator
- Java 7 Language Specification: The enhanced for statement (since Java 1.5)

"write(byte[],int,int)" should be overridden (java:S4349)

Severidad: MINOR

Root_cause:

When directly subclassing `java.io.OutputStream` or `java.io.FilterOutputStream`, the only requirement is that you implement the method `write(int)`. However most uses for such streams don't write a single byte at a time and the default implementation for `write(byte[],int,int)` will call `write(int)` for every single byte in the array which can create a lot of overhead and is utterly inefficient. It is therefore strongly recommended that subclasses provide an efficient implementation of `write(byte[],int,int)`.

This rule raises an issue when a direct subclass of `java.io.OutputStream` or `java.io.FilterOutputStream` doesn't provide an override of `write(byte[],int,int)`.

Noncompliant code example

```
public class MyStream extends OutputStream { // Noncompliant
    private FileOutputStream fout;

    public MyStream(File file) throws IOException {
        fout = new FileOutputStream(file);
    }

    @Override
    public void write(int b) throws IOException {
        fout.write(b);
    }

    @Override
    public void close() throws IOException {
        fout.write("\n\n".getBytes());
        fout.close();
        super.close();
    }
}
```

Compliant solution

```
public class MyStream extends OutputStream {
    private FileOutputStream fout;

    public MyStream(File file) throws IOException {
        fout = new FileOutputStream(file);
    }

    @Override
    public void write(int b) throws IOException {
        fout.write(b);
    }

    @Override
    public void write(byte[] b, int off, int len) throws IOException {
        fout.write(b, off, len);
    }

    @Override
    public void close() throws IOException {
        fout.write("\n\n".getBytes());
        fout.close();
        super.close();
    }
}
```

Exceptions

This rule doesn’t raise an issue when the class is declared `abstract`.

OpenSAML2 should be configured to prevent authentication bypass (java:S5679)

Severidad: MAJOR

Root_cause:

If the Service Provider does not manage to properly validate the incoming SAML response message signatures, attackers might be able to manipulate the response content without the application noticing. Especially, they might be able to alter the authentication-targeted user.

What is the potential impact?

By exploiting this vulnerability, an attacker can manipulate the SAML Response to impersonate a different user. This, in turn, can have various consequences on the application’s security.

Unauthorized Access

Exploiting this vulnerability allows an attacker with authenticated access to impersonate other users within the SAML-based SSO system. This can lead to unauthorized access to sensitive information, resources, or functionalities the attacker should not have. By masquerading as legitimate users, the attacker can bypass authentication mechanisms and gain unauthorized privileges, potentially compromising the entire system. By impersonating a user with higher privileges, the attacker can gain access to additional resources. Privilege escalation can lead to further compromise of other systems and unauthorized access to critical infrastructure.

Data Breaches

With the ability to impersonate other users, an attacker can gain access to sensitive data stored within the SAML-based SSO system. This includes personally identifiable information (PII), financial data, intellectual property, or any other confidential information. Data breaches can result in reputational damage, legal consequences, financial losses, and harm to individuals whose data is exposed.

Introduction:

The Security Assertion Markup Language (SAML) is a widely used standard in single sign-on systems. In a simplified version, the user authenticates to an Identity Provider which generates a signed SAML Response. This response is then forwarded to a Service Provider for validation and authentication.

How_to_fix:

The following code examples are vulnerable because they explicitly include comments in signature checks. An attacker is able to change the field identifying the authenticated user with XML comments.

Noncompliant code example

```
import org.opensaml.xml.parse.StaticBasicParserPool;
import org.opensaml.xml.parse.ParserPool;

public ParserPool parserPool() {
  StaticBasicParserPool staticBasicParserPool = new StaticBasicParserPool();
  staticBasicParserPool.setIgnoreComments(false); // Noncompliant
  return staticBasicParserPool;
}
```

```
import org.opensaml.xml.parse.BasicParserPool;
import org.opensaml.xml.parse.ParserPool;

public ParserPool parserPool() {
  BasicParserPool basicParserPool = new BasicParserPool();
  basicParserPool.setIgnoreComments(false); // Noncompliant
  return basicParserPool;
}
```

Compliant solution

```
import org.opensaml.xml.parse.StaticBasicParserPool;
import org.opensaml.xml.parse.ParserPool;

public ParserPool parserPool() {
  return new StaticBasicParserPool();
}
```

```
import org.opensaml.xml.parse.BasicParserPool;
import org.opensaml.xml.parse.ParserPool;

public ParserPool parserPool() {
  return new BasicParserPool();
}
```

Resources:

Documentation

- OpenSAML API - Class BasicParserPool
- OpenSAML API - Class StaticBasicParserPool
- W3C Recommendation - Canonical XML Version 1.1
- W3C Recommendation - XML Signature Syntax and Processing Version 1.1

Articles & blog posts

- Cisco Duo - Duo Finds SAML Vulnerabilities Affecting Multiple Implementations
- Spring blog - Spring Security SAML and this week’s SAML Vulnerability
- Spring Security SAML - Issue #228 Multiple SAML libraries may allow authentication bypass via incorrect XML canonicalization and DOM traversal
- CVE - CVE-2017-11427
- CVE - CVE-2017-11428
- CVE - CVE-2017-11429
- CVE - CVE-2017-11430
- CVE - CVE-2018-0489
- CVE - CVE-2018-7340

Standards

- OWASP - Top 10 2021 Category A6 - Vulnerable and Outdated Components
- OWASP - Top 10 2021 Category A7 - Identification and Authentication Failures
- OWASP - Top 10 2017 Category A9 - Using Components with Known Vulnerabilities
- OWASP - Top 10 2017 Category A2 - Broken Authentication

Proper Sensor Resource Management (java:S6889)

Severidad: MAJOR

How_to_fix:

Ensure that resources are released when they are no longer needed. This can be done by calling the appropriate release method, such as `release()`, `removeUpdates()`, `unregisterListener()`, or `stop()`.

- `android.os.PowerManager.WakeLock`

Noncompliant code example

```
public void method() {
  PowerManager powerManager = (PowerManager) getSystemService(POWER_SERVICE);
  PowerManager.WakeLock wakeLock = powerManager.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK, "My Wake Lock");
  wakeLock.acquire(); // Noncompliant
  // do some work...
}
```

Compliant solution

```
public void method() {
  PowerManager powerManager = (PowerManager) getSystemService(POWER_SERVICE);
  PowerManager.WakeLock wakeLock = powerManager.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK, "My Wake Lock");
  wakeLock.acquire(); // Compliant
  // do some work...
  wakeLock.release();
}
```

- `android.media.MediaPlayer`

Noncompliant code example

```
public void method() {
  MediaPlayer mediaPlayer = MediaPlayer.create(context, R.raw.sound_file_1);
  mediaPlayer.start(); // Noncompliant
  // do some work...
}
```

Compliant solution

```
public void onCreate() {
  MediaPlayer mediaPlayer = MediaPlayer.create(context, R.raw.sound_file_1);
  mediaPlayer.start(); // Compliant
  // do some work...
  wakeLock.release();
}
```

- `android.hardware.SensorManager`

Noncompliant code example

```
public void method() {
  SensorManager sensorManager = getSystemService(SENSOR_SERVICE);
  Sensor accelerometer = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
  sensorManager.registerListener(this, accelerometer, SensorManager.SENSOR_DELAY_NORMAL); // Noncompliant
  // do some work...
}
```

Compliant solution

```
public void method() {
  SensorManager sensorManager = getSystemService(SENSOR_SERVICE);
  Sensor accelerometer = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
  sensorManager.registerListener(this, accelerometer, SensorManager.SENSOR_DELAY_NORMAL); // Compliant
  // do some work...
  sensorManager.unregisterListener(this);
}
```

Root_cause:

Optimizing resource usage and preventing unnecessary battery drain are critical considerations in Android development. Failing to release sensor resources when they are no longer needed can lead to prolonged device activity, negatively impacting battery life. Common Android sensors, such as cameras, GPS, and microphones, provide a method to release resources after they are not in use anymore.

This rule identifies situations where a sensor is not released after being utilized, helping developers maintain efficient and battery-friendly applications.

- Missing call to `release()` method:
  - `android.os.PowerManager.WakeLock`
  - `android.net.wifi.WifiManager$MulticastLock`
  - `android.hardware.Camera`
  - `android.media.MediaPlayer`
  - `android.media.MediaRecorder`

    - `android.media.SoundPool`
    - `android.media.audiofx.Visualizer`
    - `android.hardware.display.VirtualDisplay`
- Missing call to `close()` method
    - `android.hardware.camera2.CameraDevice`
- Missing call to `removeUpdates()` method:
    - `android.location.LocationManager`
- Missing call to `unregisterListener()` method:
    - `android.hardware.SensorManager`

Resources:

Documentation

- Android - LocationManager
- Android - PowerManager.WakeLock
- Android - WifiManager.MulticastLock
- Android - MediaProjection
- Android - MediaPlayer
- Android - MediaRecorder
- Android - SoundPool
- Android - Visualizer
- Android - SensorManager
- Android - Keep the device awake
- Android - MediaPlayer Overview
- Android - Sensors Overview

Nested code blocks should not be used (java:S1199)

Severidad: MINOR

Root_cause:

Nested code blocks create new scopes where variables declared within are inaccessible from the outside, and their lifespan ends with the block.

Although this may appear beneficial, their usage within a function often suggests that the function is overloaded. Thus, it may violate the Single Responsibility Principle, and the function needs to be broken down into smaller functions.

The presence of nested blocks that don’t affect the control flow might suggest possible mistakes in the code.

Exceptions

The usage of a code block after a `case` is allowed.

How_to_fix:

The nested code blocks should be extracted into separate methods.

Noncompliant code example

```
class Example {

    private final Deque<Integer> stack = new LinkedList<>();

    public void evaluate(int operator) {
      switch (operator) {
        case ADD: {
          /* ... */
          { // Noncompliant - Extract this nested code block into a method
            int a = stack.pop();
            int b = stack.pop();
            int result = a + b;
            stack.push(result);
          }
          /* ... */
          break;
        }
        /* ... */
      }
    }
}
```

Compliant solution

```
class Example {

    private final Deque<Integer> stack = new LinkedList<>();

    public void evaluate(int operator) {
      switch (operator) {
        case ADD: {
          /* ... */
          evaluateAdd();
          /* ... */
          break;
        }
        /* ... */
      }
    }

    private void evaluateAdd() {
      int a = stack.pop();
      int b = stack.pop();
      int result = a + b;
      stack.push(result);
    }
}
```

Resources:

Documentation

- Wikipedia - Single Responsibility Principle
- Baeldung - Single Responsibility Principle

Math should not be performed on floats (java:S2164)

Severidad: MINOR

Resources:

- CERT, FLP02-C. - Avoid using floating-point numbers when precise computation is needed

Root_cause:

For small numbers, `float` math has enough precision to yield the expected value, but for larger numbers, it does not. `BigDecimal` is the best alternative, but if a primitive is required, use a `double`.

Noncompliant code example

```
float a = 16777216.0f;
float b = 1.0f;
float c = a + b; // Noncompliant; yields 1.6777216E7 not 1.6777217E7

double d = a + b; // Noncompliant; addition is still between 2 floats
```

Compliant solution

```
float a = 16777216.0f;
float b = 1.0f;
BigDecimal c = BigDecimal.valueOf(a).add(BigDecimal.valueOf(b));

double d = (double)a + (double)b;
```

Exceptions

This rule doesn't raise an issue when the mathematical expression is only used to build a string.

```
System.out.println("["+getName()+"] " +
           "\n\tMax time to retrieve connection:"+(max/1000f/1000f)+" ms.");
```

Classes named like "Exception" should extend "Exception" or a subclass (java:S2166)

Severidad: MAJOR

Root_cause:

Clear, communicative naming is important in code. It helps maintainers and API users understand the intentions for and uses of a unit of code. Using "exception" in the name of a class that does not extend `Exception` or one of its subclasses is a clear violation of the expectation that a class' name will indicate what it is and/or does.

Noncompliant code example

```
public class FruitException {  // Noncompliant; this has nothing to do with Exception
  private Fruit expected;
  private String unusualCharacteristics;
  private boolean appropriateForCommercialExploitation;
  // ...
}

public class CarException {  // Noncompliant; the extends clause was forgotten?
  public CarException(String message, Throwable cause) {
  // ...
```

Compliant solution

```
public class FruitSport {
  private Fruit expected;
  private String unusualCharacteristics;
  private boolean appropriateForCommercialExploitation;
  // ...
}

public class CarException extends Exception {
  public CarException(String message, Throwable cause) {
  // ...
```

"compareTo" should not return "Integer.MIN_VALUE" (java:S2167)

Severidad: MINOR

Resources:

Documentation

- [Java SE 8 API Specification: Comparable.compareTo]
- [Java SE 8 API Specification: Integer.MIN_VALUE]

Root_cause:

The `Comparable.compareTo` method returns a negative integer, zero, or a positive integer to indicate whether the object is less than, equal to, or greater than the parameter. The sign of the return value or whether it is zero is what matters, not its magnitude.

Returning a positive or negative constant value other than the basic ones (-1, 0, or 1) provides no additional information to the caller. Moreover, it could potentially confuse code readers who are trying to understand its purpose.

How_to_fix:

Replace any positive constant return value with 1. Replace any negative constant return value with -1.

Noncompliant code example

```
public int compareTo(Name name) {
  if (condition) {
    return Integer.MIN_VALUE; // Noncompliant
  }
}
```

Compliant solution

```
public int compareTo(Name name) {
  if (condition) {
    return -1; // Compliant
  }
}
```

Noncompliant code example

```
public int compareTo(Name name) {
  if (condition) {
    return 42; // Noncompliant
  }
}
```

Compliant solution

```
public int compareTo(Name name) {
  if (condition) {
    return 1; // Compliant
  }
}
```

Noncompliant code example

It is compliant to return other values than -1, 0 or 1 if they are not constants.

```
public int compareTo(Name name) {
  if (condition) {
    return 42; // Noncompliant
  }
}
```

Compliant solution

```
public int compareTo(Name name) {
  if (condition) {
    return hashCode() - name.hashCode(); // Compliant, not a constant
  }
}
```

"ThreadGroup" should not be used (java:S3014)

Severidad: BLOCKER

Resources:

- CERT, THI01-J. - Do not invoke ThreadGroup methods

Root_cause:

The `ThreadGroup` class contains many deprecated methods like `allowThreadSuspension`, `resume`, `stop`, and `suspend`. Also, some of the non-deprecated methods are obsolete or not thread-safe, and still others are insecure (`activeCount`, `enumerate`). For these reasons, any use of `ThreadGroup` is suspicious and should be avoided.

How_to_fix:

Instead, use implementations of `java.util.concurrent.ExecutorService` to safely manage groups of threads.

Noncompliant code example

```
class NetworkHandler {

  void startThreadInGroup(ThreadGroup tg) { // Noncompliant, use an ExecutorService instead, which is more secure
    Thread thread = new Thread(tg, "controller");
    thread.start();
  }

}
```

Compliant solution

```
class NetworkHandler {

  void handleThreadsProperly() {
    ThreadFactory threadFactory = Executors.defaultThreadFactory();
    ThreadPoolExecutor executorPool = new ThreadPoolExecutor(3, 10, 5, TimeUnit.SECONDS, new ArrayBlockingQueue<Runnable>(2), threadFactory
    for (int i = 0; i < 10; i++) {
      executorPool.execute(new Thread("Job: " + i));
    }
    executorPool.shutdown();
  }

}
```

Use built-in "Math.clamp" methods (java:S6885)

Severidad: MAJOR

Root_cause:

In Java 21 the `java.lang.Math` class was updated with the static method `Math.clamp`, to clamp a numerical value between a min and a max value.

Using this built-in method is now the preferred way to restrict to a given interval, as it is more readable and less error-prone.

How_to_fix:

Replace your clamp method implementation with the `Math.clamp` method.

Noncompliant code example

```
int clampedValue = value > max ? max : value < min ? min : value; // Noncompliant; Replace with "Math.clamp"

int clampedValue = Math.max(min, Math.min(max, value)); // Noncompliant; Replace with "Math.clamp"
```

Compliant solution

```
int clampedValue = Math.clamp(value, min, max);

int clampedValue = Math.clamp(value, min, max);
```

URIs should not be hardcoded (java:S1075)

Severidad: MINOR

Root_cause:

Hard-coding a URI makes it difficult to test a program for a variety of reasons:

- path literals are not always portable across operating systems
- a given absolute path may not exist in a specific test environment
- a specified Internet URL may not be available when executing the tests
- production environment filesystems usually differ from the development environment

In addition, hard-coded URIs can contain sensitive information, like IP addresses, and they should not be stored in the code.

For all those reasons, a URI should never be hard coded. Instead, it should be replaced by a customizable parameter.

Further, even if the elements of a URI are obtained dynamically, portability can still be limited if the path delimiters are hard-coded.

This rule raises an issue when URIs or path delimiters are hard-coded.

Exceptions

This rule does not raise an issue when:

- A constant path is relative and contains at most two parts.
- A constant path is used in an annotation
- A path is annotated

How_to_fix:

Noncompliant code example

```
public class Foo {
  public static final String FRIENDS_ENDPOINT = "/user/friends"; // Compliant path is relative and has only two parts

  public Collection<User> listUsers() {
    File userList = new File("/home/mylogin/Dev/users.txt"); // Noncompliant
    Collection<User> users = parse(userList);
    return users;
  }
}
```

Compliant solution

```
public class Foo {
  // Configuration is a class that returns customizable properties: it can be mocked to be injected during tests.
  private Configuration config;
  public Foo(Configuration myConfig) {
    this.config = myConfig;
  }
  public Collection<User> listUsers() {
    // Find here the way to get the correct folder, in this case using the Configuration object
    String listingFolder = config.getProperty("myApplication.listingFolder");
    // and use this parameter instead of the hard coded path
    File userList = new File(listingFolder, "users.txt"); // Compliant
    Collection<User> users = parse(userList);
    return users;
  }
}
```

Exceptions examples:

```
public class Foo {
  public static final String FRIENDS_ENDPOINT = "/user/friends"; // Compliant path is relative and has only two parts

  public static final String ACCOUNT = "/account/group/list.html"; // Compliant path is used in an annotation

  @Value("${base.url}" + ACCOUNT)
  private String groupUrl;

  @MyAnnotation()
  String path = "/default/url/for/site"; // Compliant path is annotated

}
```

Future keywords should not be used as names (java:S1190)

Severidad: BLOCKER

Resources:

Documentation

- Oracle - Unnamed Variables and Patterns Keywords
- Oracle - Unnamed Variables and Patterns

Root_cause:

Programming languages evolve over time, and new versions of Java introduce additional keywords. If future keywords are used in the current code, it can create compatibility issues when transitioning to newer versions of Java. The code may fail to compile or behave unexpectedly due to conflicts with newly introduced keywords.

The _ keyword was deprecated in Java 9 and disallowed since Java 11. Starting from Java 22 the _ was introduced as `unnamed variable`.

This rule reports an issue when _ is used in versions prior to Java 22.

How_to_fix:

Rename the _ identifiers.

Noncompliant code example

```
public class MyClass {
    String _ = "";            // Noncompliant
}
```

Compliant solution

```
public class MyClass {
    String s = "";            // Compliant
}
```

Classes from "sun.*" packages should not be used (java:S1191)

Severidad: MAJOR

Root_cause:

The classes in the `sun.*` packages are not part of the official Java API and are not intended for public use. They are internal implementation details specific to the Oracle JDK (Java Development Kit). Therefore, their availability, behavior, or compatibility is not guaranteed across different Java implementations or versions.

Since these classes are not part of the official Java API, they usually lack proper documentation and support. Finding comprehensive and up-to-date information about their usage, functionality, and potential limitations can be challenging. This lack of documentation can make it difficult to understand how to use these classes correctly.

Classes in the `sun.*` packages are often platform-dependent and can vary between different operating systems or Java Virtual Machine (JVM) implementations. Relying on these classes may lead to code that works on one platform but fails on others, limiting your code’s portability and cross-platform compatibility.

Noncompliant code example

```
import sun.misc.BASE64Encoder; // Noncompliant
```

Documentation

- Sun Packages

String literals should not be duplicated (java:S1192)

Severidad: CRITICAL

How_to_fix:

Use constants to replace the duplicated string literals. 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 printInQuotes(String a, String b) {
  return "'" + a + "'" + b + "'";               // 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);
}
```

Root_cause:

Duplicated string literals make the process of refactoring complex and error-prone, as any change would need to be propagated on all occurrences.

Exceptions

To prevent generating some false-positives, literals having less than 5 characters are excluded.

Exception types should not be tested using "instanceof" in catch blocks (java:S1193)

Severidad: MAJOR

Resources:

- CERT, ERR51-J. - Prefer user-defined exceptions over more general exception types
- Oracle - Exceptions

Root_cause:

A `try-catch` block is used to handle exceptions or errors that may occur during the execution of a block of code. It allows you to catch and handle exceptions gracefully, preventing your program from terminating abruptly.

The code that may throw an exception is enclosed within the `try` block, while each `catch` block specifies the type of exception it can handle. The corresponding catch block is executed if the exception matches the type specified in any catch block. It is unnecessary to manually check the types using `instanceof` because Java automatically matches the exception type to the appropriate catch block based on the declared exception type in the catch clauses.

How_to_fix:

Replace `if` statements that check the exception type using `instanceof` with corresponding `catch` blocks.

Noncompliant code example

```
try {
  /* ... */
} catch (Exception e) {
  if(e instanceof IOException) { /* ... */ }         // Noncompliant
  if(e instanceof NullPointerException{ /* ... */ }  // Noncompliant
}
```

Compliant solution

```
try {
  /* ... */
} catch (IOException e) { /* ... */ }                // Compliant
} catch (NullPointerException e) { /* ... */ }       // Compliant
```

"java.lang.Error" should not be extended (java:S1194)

Severidad: MAJOR

Root_cause:

`java.lang.Error` and its subclasses represent abnormal conditions, such as `OutOfMemoryError`, which should only be encountered by the Java Virtual Machine.

Noncompliant code example

```
public class MyException extends Error { /* ... */ }       // Noncompliant
```

Compliant solution

```
public class MyException extends Exception { /* ... */ }   // Compliant
```

Array designators "[]" should be located after the type in method signatures (java:S1195)

Severidad: MINOR

Root_cause:

Placing the array designators `[]` after the type helps maintain backward compatibility with older versions of the Java SE platform. This syntax contributes to better readability as it becomes easier to distinguish between array types and non-array types. It helps convey the intention of the method to both the developer implementing it and the developer using it.

Noncompliant code example

```
public class Cube {
    private int magicNumbers[] = { 42 };      // Noncompliant
    public int getVector()[] { /* ... */ }    // Noncompliant
    public int[] getMatrix()[] { /* ... */ }  // Noncompliant
}
```

Compliant solution

```
public class Cube {
    private int[] magicNumbers = { 42 };      // Compliant
    public int[] getVector() { /* ... */ }    // Compliant
    public int[][] getMatrix() { /* ... */ }  // Compliant
}
```

Documentation

- Oracle Java Language Specification - Arrays

Array designators "[]" should be on the type, not the variable (java:S1197)

Severidad: MINOR

Root_cause:

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
```

Subclasses that add fields to classes that override "equals" should also override "equals" (java:S2160)

Severidad: MINOR

Introduction:

This rule raises an issue when a subclass of a class that overrides `Object.equals` introduces new fields but does not also override the `Object.equals` method.

Resources:

Documentation

- [Object.equals - Java SE 8 API Specification]

How_to_fix:

Consider the following example:

```
class Foo {

  final int a;

  @Override
  public boolean equals(Object other) {
    if (other == null) return false;
    if (getClass() != other.getClass()) return false;
    return a == ((Foo) other).a;
  }
}

class Bar extends Foo { // Noncompliant, `equals` ignores the value of `b`
  final int b;
}
```

Override the `equals` method in the subclass to incorporate the new fields into the comparison:

```
class Bar extends Foo { // Compliant, `equals` now also considers `b`

  final int b;

  @Override
  public boolean equals(Object other) {
    if (!super.equals(other)) return false;
    return b == ((Bar) other).b;
  }
}
```

In case the new fields should not be part of the comparison because they are, for example, auxiliary variables not contributing to the object value (), still override the method to make the point clear that this was not just forgotten:

```
class Bar extends Foo { // Compliant, we do explicitly not want to take `b` into account

  final int b;

  @Override
  public boolean equals(Object other) {
    return super.equals(other);
  }
}
```

Root_cause:

When a class overrides `Object.equals`, this indicates that the class not just considers object identity as equal (the default implementation of `Object.equals`) but implements another logic for what is considered equal in the context of this class. Usually (but not necessarily), the semantics of `equals` in this case is that two objects are equal when their state is equal field by field.

Because of this, adding new fields to a subclass of a class that overrides `Object.equals` but not updating the implementation of `equals` in the subclass is most likely an error.

"equals" methods should be symmetric and work for subclasses (java:S2162)

Severidad: MINOR

Root_cause:

A key facet of the `equals` contract is that if `a.equals(b)` then `b.equals(a)`, i.e. that the relationship is symmetric.

Using `instanceof` breaks the contract when there are subclasses, because while the child is an `instanceof` the parent, the parent is not an `instanceof` the child. For instance, assume that `Raspberry extends Fruit` and adds some fields (requiring a new implementation of `equals`):

```
Fruit fruit = new Fruit();
Raspberry raspberry = new Raspberry();

if (raspberry instanceof Fruit) { ... } // true
if (fruit instanceof Raspberry) { ... } // false
```

If similar `instanceof` checks were used in the classes' `equals` methods, the symmetry principle would be broken:

```
raspberry.equals(fruit); // false
fruit.equals(raspberry); //true
```

Additionally, non `final` classes shouldn't use a hardcoded class name in the `equals` method because doing so breaks the method for subclasses. Instead, make the comparison dynamic.

Further, comparing to an unrelated class type breaks the contract for that unrelated type, because while `thisClass.equals(unrelatedClass)` can return true, `unrelatedClass.equals(thisClass)` will not.

Noncompliant code example

```
public class Fruit extends Food {
  private Season ripe;

  public boolean equals(Object obj) {
    if (obj == this) {
      return true;
    }
    if (obj == null) {
      return false;
    }
    if (Fruit.class == obj.getClass()) { // Noncompliant; broken for child classes
      return ripe.equals(((Fruit)obj).getRipe());
    }
    if (obj instanceof Fruit ) {  // Noncompliant; broken for child classes
      return ripe.equals(((Fruit)obj).getRipe());
    }
    else if (obj instanceof Season) { // Noncompliant; symmetry broken for Season class
      // ...
    }
    //...
```

Compliant solution

```
public class Fruit extends Food {
  private Season ripe;

  public boolean equals(Object obj) {
    if (obj == this) {
      return true;
    }
    if (obj == null) {
      return false;
    }
    if (this.getClass() == obj.getClass()) {
      return ripe.equals(((Fruit)obj).getRipe());
    }
    return false;
}
```

Resources:

- CERT, MET08-J. - Preserve the equality contract when overriding the equals() method

Static fields should not be updated in constructors (java:S3010)

Severidad: MAJOR

Root_cause:

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;
  }
}
```

Reflection should not be used to increase accessibility of classes, methods, or fields (java:S3011)

Severidad: MAJOR

Root_cause:

Altering or bypassing the accessibility of classes, methods, or fields through reflection violates the encapsulation principle. This can break the internal contracts of the accessed target and lead to maintainability issues and runtime errors.

This rule raises an issue when reflection is used to change the visibility of a class, method or field, and when it is used to directly update a field value.

```
public void makeItPublic(String methodName) throws NoSuchMethodException {

  this.getClass().getMethod(methodName).setAccessible(true); // Noncompliant
}

public void setItAnyway(String fieldName, int value) {
  this.getClass().getDeclaredField(fieldName).setInt(this, value); // Noncompliant; bypasses controls in setter
}
```

Resources:

Documentation

- Wikipedia definition of Encapsulation
- CERT, SEC05-J. - Do not use reflection to increase accessibility of classes, methods, or fields

Arrays and lists should not be copied using loops (java:S3012)

Severidad: MINOR

Resources:

Documentation

- docs.oracle - Arrays.copyOf documentation
- docs.oracle - Arrays.asList documentation
- docs.oracle - System.arraycopy documentation

Root_cause:

The JDK provides a set of built-in methods to copy the contents of an array into another array. Using a loop to perform the same operation is less clear, more verbose and should be avoided.

Exceptions

The rule detects only the most idiomatic patterns, it will not consider loops with non-trivial control flow. For example, loops that copy array elements conditionally are ignored.

How_to_fix:

You can use:

- `Arrays.copyOf` to copy an entire array into another array
- `System.arraycopy` to copy only a subset of an array into another array
- `Arrays.asList` to create a new list with the contents of the array
- `Collections.addAll` to add the elements of a collection into another collection

Note that `Arrays.asList` returns a fixed-size `List`, so further steps are required if a non-fixed-size `List` is needed.

Noncompliant code example

```
public void copyArray(String[] source){
  String[] array = new String[source.length];
  for (int i = 0; i < source.length; i++) {
    array[i] = source[i]; // Noncompliant
  }
}

public void copyList(List<String> source) {
  List<String> list = new ArrayList<>();
  for (String s : source) {
    list.add(s); // Noncompliant
  }
}
```

Compliant solution

```
public void copyArray(String[] source){
  String[] array = Arrays.copyOf(source, source.length);
}

public void copyList(List<String> source) {
  List<String> list = new ArrayList<>();
  Collections.addAll(list, source);
}

public void makeCopiesConditional(int[] source) {
  int[] dest = new int[source.length];
  for (int i = 0; i < source.length; i++) {
    if (source[i] > 10) {
      dest[i] = source[i];  // Compliant, since the array elements are conditionally copied to the dest array
    }
  }
}
```

"static" base class members should not be accessed via derived types (java:S3252)

Severidad: CRITICAL

Root_cause:

In object-oriented programming, inappropriately accessing static members of a base class via derived types is considered a code smell.

Static members are associated with the class itself, not with any specific instance of the class or its children classes. Accessing through the wrong type suggests a misunderstanding of the ownership and role of this member. This can make the maintenance of the code more complicated.

Therefore, the access should be done directly through the base class to maintain clarity and avoid potential misunderstandings.

Noncompliant code example

```
class Parent {
  public static int counter;
}

class Child extends Parent {
  public Child() {
    Child.counter++;  // Noncompliant
  }
}
```

Compliant solution

```
class Parent {
  public static int counter;
}

class Child extends Parent {
  public Child() {
    Parent.counter++;
  }
}
```

Default annotation parameter values should not be passed as arguments (java:S3254)

Severidad: MINOR

Root_cause:

Specifying the default value for an annotation parameter is redundant. Such values should be omitted in the interests of readability.

Noncompliant code example

```
@MyAnnotation(arg = "def")  // Noncompliant
public class MyClass {
  // ...
}
public @interface MyAnnotation {
  String arg() default "def";
}
```

Compliant solution

```
@MyAnnotation
public class MyClass {
  // ...
}
public @interface MyAnnotation {
  String arg() default "def";
}
```

JUnit5 inner test classes should be annotated with @Nested (java:S5790)

Severidad: CRITICAL

Root_cause:

If not annotated with `@Nested`, an inner class containing some tests will never be executed during tests execution. While you could still be able to manually run its tests in an IDE, it won’t be the case during the build. By contrast, a static nested class containing some tests should not be annotated with `@Nested`, JUnit5 will not share setup and state with an instance of its enclosing class.

This rule raises an issue on inner classes and static nested classes containing JUnit5 test methods which has a wrong usage of `@Nested` annotation.

Note: This rule does not check if the context in which JUnit 5 is running (e.g. Maven Surefire Plugin) is properly configured to execute static nested classes, it could not be the case using the default configuration.

Noncompliant code example

```
import org.junit.jupiter.api.Test;

class MyJunit5Test {
  @Test
  void test() { /* ... */ }

  class InnerClassTest { // Noncompliant, missing @Nested annotation
    @Test
    void test() { /* ... */ }
  }

  @Nested
  static class StaticNestedClassTest { // Noncompliant, invalid usage of @Nested annotation
    @Test
    void test() { /* ... */ }
  }
}
```

Compliant solution

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.Nested;

class MyJunit5Test {
  @Test
  void test() { /* ... */ }

  @Nested
  class InnerClassTest {
    @Test
    void test() { /* ... */ }
  }

  static class StaticNestedClassTest {
    @Test
    void test() { /* ... */ }
  }
}
```

Migrate your tests from JUnit4 to the new JUnit5 annotations (java:S5793)

Severidad: INFO

Resources:

- JUnit 5: Migrating from JUnit4

Root_cause:

As mentioned in JUnit5 documentation, it is possible to integrate JUnit4 with JUnit5:

> JUnit provides a gentle migration path via a JUnit Vintage test engine which allows existing tests based on JUnit 3 and JUnit 4 to be executed using the JUnit Platform infrastructure. Since all classes and annotations specific to JUnit Jupiter reside under a new org.junit.jupiter base package, having both JUnit 4 and JUnit Jupiter in the classpath does not lead to any conflicts.

However, maintaining both systems is a temporary solution. This rule flags all the annotations from JUnit4 which would need to be migrated to JUnit5, hence helping migration of a project.

Here is the list of JUnit4 annotations tracked by the rule, with their corresponding annotations in JUnit5:

JUnit4	JUnit5
`org.junit.Test`	`org.junit.jupiter.api.Test`
`org.junit.Before`	`org.junit.jupiter.api.BeforeEach`
`org.junit.After`	`org.junit.jupiter.api.AfterEach`
`org.junit.BeforeClass`	`org.junit.jupiter.api.BeforeAll`
`org.junit.AfterClass`	`org.junit.jupiter.api.AfterAll`
`org.junit.Ignore`	`org.junit.jupiter.api.Disabled`

Note that the following annotations might requires some rework of the tests to have JUnit5 equivalent behavior. A simple replacement of the annotation won’t work immediately:

JUnit4	JUnit5
`org.junit.experimental.categories.Category`	`org.junit.jupiter.api.Tag`
`org.junit.Rule`	`org.junit.jupiter.api.extension.ExtendWith`
`org.junit.ClassRule`	`org.junit.jupiter.api.extension.RegisterExtension`
`org.junit.runner.RunWith`	`org.junit.jupiter.api.extension.ExtendWith`

Noncompliant code example

```
package org.foo;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Ignore;
import org.junit.Test;
import org.junit.experimental.categories.Category;
import org.junit.runner.RunWith;

@RunWith(MyJUnit4Runner.class)
public class MyJUnit4Test {

  @BeforeClass
  public static void beforeAll() {
    System.out.println("beforeAll");
  }

  @AfterClass
  public static void afterAll() {
    System.out.println("AfterAll");
  }

  @Before
  public void beforeEach() {
    System.out.println("beforeEach");
  }

  @After
  public void afterEach() {
    System.out.println("afterEach");
  }

  @Test
  public void test1() throws Exception {
    System.out.println("test1");
  }

  public interface SomeTests { /* category marker */ }

  @Test
  @Category(SomeTests.class)
  public void test2() throws Exception {
    System.out.println("test2");
  }

  @Test
  @Ignore("Requires fix of #42")
  public void ignored() throws Exception {
    System.out.println("ignored");
  }
}
```

Compliant solution

```
package org.foo;

import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;

@ExtendWith(MyJUnit5Extension.class)
class MyJUnit5Test {

  @BeforeAll
  static void beforeAll() {
    System.out.println("beforeAll");
  }

  @AfterAll
  static void afterAll() {
    System.out.println("afterAll");
  }

  @BeforeEach
  void beforeEach() {
    System.out.println("beforeEach");
  }

  @AfterEach
  void afterEach() {
    System.out.println("afterEach");
  }
  @Test
  void test1() {
    System.out.println("test1");
  }

  @Test
  @Tag("SomeTests")
  void test2() {
    System.out.println("test2");
  }

  @Test
  @Disabled("Requires fix of #42")
  void disabled() {
    System.out.println("ignored");
  }
}
```

Use switch instead of if-else chain to compare a variable against multiple cases (java:S6880)

Severidad: MAJOR

Root_cause:

Comparing a variable to multiple cases is a frequent operation. This can be done using a sequence of if-else statements. However, for many cases like enums or simple value comparisons, a `switch` statement is the better alternative. With Java 21, the `switch` statement has been significantly improved to support pattern matching and record pattern.

Using a `switch` statement instead of an if-else chain provides benefits like clearer code, certainty of covering all cases, and may even improve performance.

This rule raises an issue when an if-else chain should be replaced by a `switch` statement.

Resources:

- Record Patterns
- Pattern Matching for switch
- The switch Statement

How_to_fix:

Replace the chain of if-else with a switch expression.

Noncompliant code example

```
sealed interface Expression {}
record Plus(Expression left, Expression right) implements Expression {}
record Minus(Expression left, Expression right) implements Expression {}
record Div(Expression left, Expression right) implements Expression {}

int eval(Expression expr){
  if(expr instanceof Plus plus){ // Noncompliant; should be replaced by a switch expression
    return eval(plus.left) + eval(plus.right);
  }else if(expr instanceof Div div){
    return eval(div.left) / eval(div.right);
  }else if(expr instanceof Minus minus){
    return eval(minus.left) - eval(minus.right);
  } else {
    throw new IllegalArgumentException("Unknown expression");
  }
}

enum Color{RED,GREEN,YELLOW}

String name(Color c){
  if(c == Color.RED){ // Noncompliant; should be replaced by a switch expression
    return "red";
  }else if(c == Color.GREEN){
    return "green";
  }else if(c == Color.YELLOW){
    return "yellow";
  }else{
    throw new IllegalArgumentException("Unknown color");
  }
}

int points(int result){
  if(result == 2){ // Noncompliant; should be replaced by a switch expression
    return 10;
  } else if(result == 3 || result==4 ){
    return 20;
  } else if (result == 5) {
    return 50;
  }else{
    return 0;
  }
}

class Circle{}
class Rectangle{}
class Square{}

String name(Object shape){
  if (shape instanceof Circle) { // Noncompliant; should be replaced by a switch expression
    return "circle";
  } else if (shape instanceof Rectangle) {
     return "rectangle";
  } else if (shape instanceof Square) {
    return "square";
  } else {
    throw new IllegalArgumentException();
  }
}
```

Compliant solution

```
sealed interface Expression {}
record Plus(Expression left, Expression right) implements Expression {}
record Minus(Expression left, Expression right) implements Expression {}
record Div(Expression left, Expression right) implements Expression {}

int eval(Expression expr){
  return switch(expr){
    case Div(var left, var right) -> eval(left) / eval(right);
    case Plus(var left, var right) -> eval(left) + eval(right);
    case Minus(var left, var right) -> eval(left) - eval(right);
  };
}

enum Color{RED,GREEN,YELLOW}
String name(Color c){
  return switch(c){
    case RED -> "red";
    case GREEN -> "green";
    case YELLOW -> "yellow";
  };
}

int points(int result){
  return switch(result){
    case 2 -> 10;
    case 3,4 -> 20;
    case 5 -> 50;
    default -> 0;
  };
}

class Circle{}
class Rectangle{}
class Square{}

String name(Object shape){
  return switch(shape){
    case Circle c -> "circle";
    case Rectangle r -> "rectangle";
    case Square s -> "square";
    default -> throw new IllegalArgumentException();
  };
}
```

Virtual threads should be used for tasks that include heavy blocking operations (java:S6881)

Severidad: MAJOR

Root_cause:

Whenever a virtual thread is started, the JVM will mount it on an OS thread. As soon as the virtual thread runs into a blocking operation like an HTTP request or a filesystem read/write operation, the JVM will detect this and unmount the virtual thread. This allows another virtual thread to take over the OS thread and continue its execution.

This is why virtual threads should be preferred to platform threads for tasks that involve blocking operations. By default, a Java thread is a platform thread. To use a virtual thread it must be started either with `Thread.startVirtualThread(Runnable)` or `Thread.ofVirtual().start(Runnable)`.

This rule raises an issue when a platform thread is created with a task that includes heavy blocking operations.

How_to_fix:

Replace platform thread instances or platform thread pools with virtual threads, if their task involves blocking operations.

Noncompliant code example

The following example creates a platform thread to handle a blocking operation, here denoted by `Thread.sleep(1000)`. The overhead for instantiating a platform thread is higher than for a virtual thread. Further, instantiating too many platform threads can lead to problems if the number of instantiated threads exceeds the maximum number of platform threads allowed by the OS.

```
new Thread(() -> {
    try {
        Thread.sleep(1000); // Noncompliant blocking operation in platform thread
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
});
```

In the context of thread pools, using platform threads for heavy blocking operations can lead to the thread pool running out of available threads. Even though the threads spend most of their time waiting for e.g. I/O operations to complete and subsequently the CPU usage is low, the application cannot continue processing efficiently, due to the lack of available threads.

Compliant solution

Using virtual threads allows the developer to abstract from any pooling logic as they are much lighter than platform threads, and the number of virtual threads that can be instantiated is only limited by the available memory. In this example, the execution of 10000 requests would take just over ~1 second without any risk of exceeding the allowed number of platform threads.

```
Thread.ofVirtual().start(() -> {
    try {
        Thread.sleep(1000); // Compliant
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
});
```

Introduction:

With the introduction of virtual threads in Java 21, it is now possible to optimize the usage of the operating system threads by avoiding blocking them for asynchronous operations. Furthermore, virtual threads's instantiation has very little overhead and they can be created in large quantities. This means that it can be more efficient to use them over the default platform threads for tasks that involve I/O or some other blocking operations.

Resources:

Documentation

- Java Documentation - Virtual Threads
- Java Documentation - Thread.startVirtualThread(Runnable)

"Serializable" inner classes of "Serializable" classes should be static (java:S2059)

Severidad: MINOR

Resources:

- CERT, SER05-J. - Do not serialize instances of inner classes

Root_cause:

Serializing a non-`static` inner class will result in an attempt at serializing the outer class as well. If the outer class is actually serializable, then the serialization will succeed but possibly write out far more data than was intended.

Making the inner class `static` (i.e. "nested") avoids this problem, therefore inner classes should be `static` if possible. However, you should be aware that there are semantic differences between an inner class and a nested one:

- an inner class can only be instantiated within the context of an instance of the outer class.
- a nested (`static`) class can be instantiated independently of the outer class.

Noncompliant code example

```
public class Raspberry implements Serializable {
  // ...

  public class Drupelet implements Serializable {  // Noncompliant; output may be too large
    // ...
  }
}
```

Compliant solution

```
public class Raspberry implements Serializable {
  // ...

  public static class Drupelet implements Serializable {
    // ...
  }
}
```

Disclosing fingerprints from web application technologies is security-sensitive (java:S5689)

Severidad: MINOR

Assess_the_problem:

Ask Yourself Whether

- Version information is accessible to end users.
- Internal systems do not benefit from timely patch management workflows.

There is a risk if you answered yes to any of these questions.

Sensitive Code Example

```
@GetMapping(value = "/example")
public ResponseEntity<String> example() {
  HttpHeaders responseHeaders = new HttpHeaders();
  responseHeaders.set("x-powered-by", "myproduct"); // Sensitive

  return new ResponseEntity<String>(
      "example",
      responseHeaders,
      HttpStatus.CREATED);
}
```

Root_cause:

Disclosure of version information, usually overlooked by developers but disclosed by default by the systems and frameworks in use, can pose a significant security risk depending on the production environement.

Once this information is public, attackers can use it to identify potential security holes or vulnerabilities specific to that version.

Furthermore, if the published version information indicates the use of outdated or unsupported software, it becomes easier for attackers to exploit known vulnerabilities. They can search for published vulnerabilities related to that version and launch attacks that specifically target those vulnerabilities.

How_to_fix:

Recommended Secure Coding Practices

In general, it is recommended to keep internal technical information within internal systems to control what attackers know about the underlying architectures. This is known as the "need to know" principle.

The most effective solution is to remove version information disclosure from what end users can see, such as the "x-powered-by" header.
This can be achieved directly through the web application code, server (nginx, apache) or firewalls.

Disabling the server signature provides additional protection by reducing the amount of information available to attackers. Note, however, that this does not provide as much protection as regular updates and patches.
Security by obscurity is the least foolproof solution of all. It should never be the only defense mechanism and should always be combined with other security measures.

Compliant Solution

Do not disclose version information unless necessary. The `x-powered-by` or `Server` HTTP headers should not be used.

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP Testing Guide - OTG-INFO-008 - Fingerprint Web Application Framework
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- CWE - CWE-200 - Information Exposure

Default:

Disclosure of version information, usually overlooked by developers but disclosed by default by the systems and frameworks in use, can pose a significant security risk depending on the production environement.

Once this information is public, attackers can use it to identify potential security holes or vulnerabilities specific to that version.

Furthermore, if the published version information indicates the use of outdated or unsupported software, it becomes easier for attackers to exploit known vulnerabilities. They can search for published vulnerabilities related to that version and launch attacks that specifically target those vulnerabilities.

Ask Yourself Whether

- Version information is accessible to end users.
- Internal systems do not benefit from timely patch management workflows.

There is a risk if you answered yes to any of these questions.

Recommended Secure Coding Practices

In general, it is recommended to keep internal technical information within internal systems to control what attackers know about the underlying architectures. This is known as the "need to know" principle.

The most effective solution is to remove version information disclosure from what end users can see, such as the "x-powered-by" header.
This can be achieved directly through the web application code, server (nginx, apache) or firewalls.

Disabling the server signature provides additional protection by reducing the amount of information available to attackers. Note, however, that this does not provide as much protection as regular updates and patches.
Security by obscurity is the least foolproof solution of all. It should never be the only defense mechanism and should always be combined with other security measures.

Sensitive Code Example

```
@GetMapping(value = "/example")
public ResponseEntity<String> example() {
  HttpHeaders responseHeaders = new HttpHeaders();
  responseHeaders.set("x-powered-by", "myproduct"); // Sensitive

  return new ResponseEntity<String>(
      "example",
      responseHeaders,
      HttpStatus.CREATED);
}
```

Compliant Solution

Do not disclose version information unless necessary. The `x-powered-by` or `Server` HTTP headers should not be used.

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP Testing Guide - OTG-INFO-008 - Fingerprint Web Application Framework
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- CWE - CWE-200 - Information Exposure

High frame rates should not be used (java:S6898)

Severidad: MAJOR

Root_cause:

Standard applications don't require a display refresh rate above 60Hz, hence it is advisable to avoid higher frequencies to avoid unnecessary energy consumption.

The rule flags an issue when `setFrameRate()` is invoked with a frameRate higher than 60Hz for `android.view.Surface` and `android.view.SurfaceControl.Transaction`.

It's important to note that the scheduler considers several factors when determining the display refresh rate. Therefore, using `setFrameRate()` doesn't guarantee your app will achieve the requested frame rate.

What is the potential impact?

- *Usability*: the device may run out of battery faster than expected.
- *Sustainability*: the extra battery usage has a negative impact on the environment.

Introduction:

The *Frame Rate API* allows applications to communicate their desired frame rate to the *Android platform* to enhance the user experience. The API is useful since many devices now offer varying refresh rates like 60Hz, 90Hz, or 120Hz.

How_to_fix:

Use a frame rate of maximum 60Hz, unless you have a strong reason to used higher rates. Valid exceptions are *gaming apps*, especially those with fast-paced action or high-quality graphics, or *AR/VR apps*.

Noncompliant code example

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        SurfaceView surfaceView = findViewById(R.id.my_surface_view);
        Surface surface = surfaceView.getHolder().getSurface();

        surface.setFrameRate(90.0f, Surface.FRAME_RATE_COMPATIBILITY_FIXED_SOURCE); // Noncompliant
    }
}
```

Compliant solution

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        SurfaceView surfaceView = findViewById(R.id.my_surface_view);
        Surface surface = surfaceView.getHolder().getSurface();

        surface.setFrameRate(60.0f, Surface.FRAME_RATE_COMPATIBILITY_FIXED_SOURCE); // Compliant
    }
}
```

Resources:

Documentation

- Android for Developers: Frame Rate
- Developer Apple - Frame Rate

Method names should comply with a naming convention (java:S100)

Severidad: MINOR

Root_cause:

Shared naming conventions allow teams to collaborate efficiently.

This rule raises an issue when a method name does not match a provided regular expression.

For example, with the default provided regular expression `^[a-z][a-zA-Z0-9]*$`, the method:

```
public int DoSomething(){...} // Noncompliant
```

should be renamed to

```
public int doSomething(){...}
```

Exceptions

Overriding methods are excluded.

```
@Override
public int Do_Something(){...} // Compliant by exception
```

Class names should comply with a naming convention (java:S101)

Severidad: MINOR

Root_cause:

Shared naming conventions allow teams to collaborate efficiently.

This rule raises an issue when a class name does not match a provided regular expression.

For example, with the default provided regular expression `^[A-Z][a-zA-Z0-9]*$`, the class:

```
class my_class {...} // Noncompliant
```

should be renamed to

```
class MyClass {...}
```

The non-serializable super class of a "Serializable" class should have a no-argument constructor (java:S2055)

Severidad: MINOR

Root_cause:

Java serialization is the conversion from objects to byte streams for storage or transmission. And later, java deserialization is the reverse conversion, it reconstructs objects from byte streams.

To make a java class serializable, this class should implement the `java.io.Serializable` interface directly or through its inheritance.

```
import java.io.Serializable;

public class NonSerializableClass {
}

public class SerializableClass implements Serializable {
}

public class OtherSerializableClass extends SerializableClass {
  // is also serializable because it is a subtype of Serializable
}
```

Given a serializable class, it is important to note that not all its superclasses are serializable. Eventually, its superclasses stop implementing `java.io.Serializable`. It could be at the end, once reaching the `java.lang.Object` class, or before.

This is important because the serialization/deserialization runs through the class hierarchy of an object to decide which object fields to write or read, and applies two different logics:

- When the class is serializable:
  - Serialization saves the class reference and the object fields of this class.
  - Deserialization instantiates a new object of this class without using a constructor, and restores the object fields of this class.
- When the class is not serializable:
  - Serialization only saves the class reference and **ignores** the object fields of this class.
  - Deserialization instantiates a new object of this class using the **no-argument constructor** and does not restore the object fields of this class.

So developers should pay particular attention to the non-serializable classes in the class hierarchy, because the presence of an implicit or explicit **no-argument constructor** is required in those classes.

This is an example of mandatory no-argument constructors in the hierarchy of `SerializableClass`:

```
public class NonSerializableClassWithoutConstructor {
  // after deserialization, "field1" will always be set to 42
  private int field1 = 42;

  // this non-serializable class has an implicit no-argument constructor
}

public class NonSerializableClass extends NonSerializableClassWithoutConstructor {
  // after deserialization, "field2" will always be set to 12 by the no-argument constructor
  private int field2;

  // this non-serializable class has an explicit no-argument constructor
  public NonSerializableClass() {
    field2 = 12;
  }

  public NonSerializableClass(int field2) {
    this.field2 = field2;
  }
}

public class SerializableClass extends NonSerializableClass implements Serializable {
  // after deserialization, "field3" will have the previously serialized value.
  private int field3;

  // deserialization does not use declared constructors
  public SerializableClass(int field3) {
    super(field3 * 2);
    this.field3 = field3;
  }
}
```

Unfortunately, there is no compilation error when a class implements `java.io.Serializable` and extends a non-serializable superclass without a no-argument constructor. This is an issue because, at runtime, deserialization will fail to find the required constructor.

For example, deserialization of an instance of the following `SerializableClass` class, throws an `InvalidClassException: no valid constructor`.

```
public class NonSerializableClass {
  private int field;
  // this class can not be deserialized because it does not have any implicit or explicit no-argument constructor
  public NonSerializableClass(int field) {
    this.field = field;
  }
}

public class SerializableClass extends NonSerializableClass implements Serializable {
}
```

This rule checks in the hierarchy of serializable classes and reports an issue when a non-serializable superclass does not have the required no-argument constructor which will produce a runtime error.

Resources:

Documentation

- Oracle SDK - java.io.Serializable

How_to_fix:

There are two solutions to fix the missing **no-argument constructor** issue on non-serializable classes:

- `Solution 1` If the fields of a non-serializable class need to be persisted, add the `java.io.Serializable` interface to the class `implements` definition.
- `Solution 2` Otherwise, add a no-argument constructor and initialize the fields with some valid default values.

Example #1

Noncompliant code example

```
// Noncompliant; this Raspberry's ancestor doesn't have a no-argument constructor
// this rule raises an issue on the Raspberry class declaration
public class Fruit {
  private Season pickingSeason;
  public Fruit(Season pickingSeason) {
    this.pickingSeason = pickingSeason;
  }
}

public class Raspberry extends Fruit implements Serializable {
  private static final long serialVersionUID = 1;
  private String variety;
  public Raspberry(String variety) {
    super(Season.SUMMER);
    this.variety = variety;
  }
}
```

Compliant solution

```
Solution 1

// Compliant; this Raspberry's ancestor is serializable
public class Fruit implements Serializable {
  private static final long serialVersionUID = 1;
  private Season pickingSeason;
  public Fruit(Season pickingSeason) {
    this.pickingSeason = pickingSeason;
  }
}
```

Example #2

Noncompliant code example

```
public class Fruit {
  // Noncompliant; this Raspberry's ancestor doesn't have a no-argument constructor
  // this rule raises an issue on the Raspberry class declaration
  public Fruit(String debugMessage) {
    LOG.debug(debugMessage);
  }
}

public class Raspberry extends Fruit implements Serializable {
  private static final long serialVersionUID = 1;
  private String variety;
  public Raspberry(String variety) {
    super("From Raspberry constructor");
    this.variety = variety;
  }
}
```

Compliant solution

```
Solution 2

public class Fruit {
  // Compliant; this Raspberry ancestor has a no-argument constructor
  public Fruit() {
    this("From serialization");
  }
  public Fruit(String debugMessage) {
    LOG.debug(debugMessage);
  }
}
```

"Serializable" classes should have a "serialVersionUID" (java:S2057)

Severidad: CRITICAL

Root_cause:

A `serialVersionUID` field is strongly recommended in all `Serializable` classes. If you do not provide one, one will be calculated for you by the compiler. The danger in not explicitly choosing the value is that when the class changes, the compiler will generate an entirely new id, and you will be suddenly unable to deserialize (read from file) objects that were serialized with the previous version of the class.

`serialVersionUID`'s should be declared with all of these modifiers: `static final long`.

Noncompliant code example

```
public class Raspberry extends Fruit  // Noncompliant; no serialVersionUID.
        implements Serializable {
  private String variety;

  public Raspberry(Season ripe, String variety) { ...}
  public void setVariety(String variety) {...}
  public String getVarity() {...}
}

public class Raspberry extends Fruit
```

```
        implements Serializable {
  private final int serialVersionUID = 1; // Noncompliant; not static & int rather than long
```

Compliant solution

```
public class Raspberry extends Fruit
        implements Serializable {
  private static final long serialVersionUID = 1;
  private String variety;

  public Raspberry(Season ripe, String variety) { ...}
  public void setVariety(String variety) {...}
  public String getVarity() {...}
}
```

Exceptions

Records, Swing and AWT classes, `abstract` classes, `Throwable` and its subclasses (`Exceptions` and `Errors`), and classes marked with `@SuppressWarnings("serial")` are ignored.

Resources:

- CERT, SER00-J. - Enable serialization compatibility during class evolution
- Record Serialization - Serialization of Records

Inappropriate "Collection" calls should not be made (java:S2175)

Severidad: MAJOR

Root_cause:

The `java.util.Collection` type and its subtypes provide methods to access and modify collections such as `Collection.remove(Object o)` and `Collection.contains(Object o)`. Some of these methods accept arguments of type `java.lang.Object` and will compare said argument with objects already in the collection.

If the actual type of the argument is unrelated to the type of object contained in the collection, these methods will always return `false`, `null`, or `-1`. This behavior is most likely unintended and can be indicative of a design issue.

This rule raises an issue when the type of the argument provided to one of the following methods is unrelated to the type used for the collection declaration:

- `Collection.remove(Object o)`
- `Collection.removeAll(Collection<?>)`
- `Collection.contains(Object o)`
- `List.indexOf(Object o)`
- `List.lastIndexOf(Object o)`
- `Map.containsKey(Object key)`
- `Map.containsValue(Object value)`
- `Map.get(Object key)`
- `Map.getOrDefault(Object key, V defaultValue)`
- `Map.remove(Object key)`
- `Map.remove(Object key, Object value)`

Resources:

- CERT, EXP04-J. - Do not pass arguments to certain Java Collections Framework methods that are a different type than the collection parameter type
- Java SE 17 & JDK 17 - Collection interface

How_to_fix:

Ask yourself what the purpose of this method call is. Check whether the provided argument and collection are correct in this context and for the desired purpose. Remove unnecessary calls and otherwise provide an argument of which the type is compatible with the list content’s type.

Noncompliant code example

```
void removeFromMap(Map<Integer, Object> map, String strKey) {
  map.remove(strKey); // Noncompliant, this call will remove nothing and always return 'null' because 'map' is handling only Integer keys a
}

void listContains(List<String> list, Integer integer) {
  if (list.contains(integer)) { // Noncompliant; always false as the list only contains Strings, not integers.
    // ...
  }
}
```

Compliant solution

```
void removeFromMap(Map<Integer, Object> map, String strKey) {
  map.remove(Integer.parseInt(strKey)); // Compliant, strKey is parsed into an Integer before trying to remove it from the map.
}

void listContains(List<String> list, Integer integer) {
  if (list.contains(integer.toString())) { // Compliant, 'integer' is converted to a String before checking if the list contains it.
    // ...
  }
}
```

Class names should not shadow interfaces or superclasses (java:S2176)

Severidad: CRITICAL

Resources:

Documentation

- Java SE Specifications > Java Language Specification > 6.4.1. Shadowing

Root_cause:

Two classes can have the same simple name if they are in two different packages.

```
package org.foo.domain;

public class User {
  // ..
}
```

```
package org.foo.presentation;

public class User {
  // ..
}
```

However, this becomes an issue when a class has the same name as the superclass it extends or the interfaces it implements, also known as class name shadowing. It is problematic because it can be unclear which class is being referred to in the code, leading to ambiguity and potential bugs.

Therefore, it is recommended to use unique and descriptive class names that do not conflict with the names of the superclass or interfaces.

This rule raises an issue when a class name shadows its superclass or interface names.

How_to_fix:

Rename the class using a more descriptive name.

Noncompliant code example

```
package org.foo.presentation;

public class User implements org.foo.domain.User { // Noncompliant
  // ...
}
```

Compliant solution

```
package org.foo.presentation;

import org.foo.domain.User;

public class UserView implements User { // Compliant
  // ...
}
```

Child class methods named for parent class methods should be overrides (java:S2177)

Severidad: MAJOR

Root_cause:

When a method in a child class has the same signature as a method in a parent class, it is assumed to be an override. However, that's not the case when:

- the parent class method is `static` and the child class method is not.
- the arguments or return types of the child method are in different packages than those of the parent method.
- the parent class method is `private`.

Typically, these things are done unintentionally; the private parent class method is overlooked, the `static` keyword in the parent declaration is overlooked, or the wrong class is imported in the child. But if the intent is truly for the child class method to be different, then the method should be renamed to prevent confusion.

Noncompliant code example

```
// Parent.java
import computer.Pear;
public class Parent {

  public void doSomething(Pear p) {
    //,,,
  }

  public static void doSomethingElse() {
    //...
  }
}

// Child.java
import fruit.Pear;
public class Child extends Parent {

  public void doSomething(Pear p) {  // Noncompliant; this is not an override
    // ...
  }


  public void doSomethingElse() {  // Noncompliant; parent method is static
    //...
  }
}
```

Compliant solution

```
// Parent.java
import computer.Pear;
public class Parent {

  public void doSomething(Pear p) {
    //,,,
  }

  public static void doSomethingElse() {
    //...
  }
}

// Child.java
import computer.Pear;  // import corrected
public class Child extends Parent {

  public void doSomething(Pear p) {  // true override (see import)
    //,,,
  }

  public static void doSomethingElse() {
    //...
  }
}
```

Short-circuit logic should be used in boolean contexts (java:S2178)

Severidad: BLOCKER

Resources:

- CERT, EXP46-C. - Do not use a bitwise operator with a Boolean-like operand

Root_cause:

The use of non-short-circuit logic in a boolean context is likely a mistake - one that could cause serious program errors as conditions are evaluated under the wrong circumstances.

Noncompliant code example

```
if(getTrue() | getFalse()) { ... } // Noncompliant; both sides evaluated
```

Compliant solution

```
if(getTrue() || getFalse()) { ... } // true short-circuit logic
```

Receiving intents is security-sensitive (java:S5322)

Severidad: CRITICAL

Assess_the_problem:

Ask Yourself Whether

- The data extracted from intents is not sanitized.
- Intents broadcast is not restricted.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

```
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.IntentFilter;
import android.os.Build;
import android.os.Handler;
import android.support.annotation.RequiresApi;

public class MyIntentReceiver {

    @RequiresApi(api = Build.VERSION_CODES.O)
    public void register(Context context, BroadcastReceiver receiver,
                         IntentFilter filter,
                         String broadcastPermission,
                         Handler scheduler,
                         int flags) {
        context.registerReceiver(receiver, filter); // Sensitive
        context.registerReceiver(receiver, filter, flags); // Sensitive

        // Broadcasting intent with "null" for broadcastPermission
        context.registerReceiver(receiver, filter, null, scheduler); // Sensitive
        context.registerReceiver(receiver, filter, null, scheduler, flags); // Sensitive
    }
}
```

Root_cause:

Android applications can receive broadcasts from the system or other applications. Receiving intents is security-sensitive. For example, it has led in the past to the following vulnerabilities:

- CVE-2019-1677
- CVE-2015-1275

Receivers can be declared in the manifest or in the code to make them context-specific. If the receiver is declared in the manifest Android will start the application if it is not already running once a matching broadcast is received. The receiver is an entry point into the application.

Other applications can send potentially malicious broadcasts, so it is important to consider broadcasts as untrusted and to limit the applications that can send broadcasts to the receiver.

Permissions can be specified to restrict broadcasts to authorized applications. Restrictions can be enforced by both the sender and receiver of a broadcast. If permissions are specified when registering a broadcast receiver, then only broadcasters who were granted this permission can send a message to the receiver.

This rule raises an issue when a receiver is registered without specifying any broadcast permission.

How_to_fix:

Recommended Secure Coding Practices

Restrict the access to broadcasted intents. See the Android documentation for more information.

Compliant Solution

```
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.IntentFilter;
import android.os.Build;
import android.os.Handler;
import android.support.annotation.RequiresApi;

public class MyIntentReceiver {

    @RequiresApi(api = Build.VERSION_CODES.O)
    public void register(Context context, BroadcastReceiver receiver,
                         IntentFilter filter,
                         String broadcastPermission,
                         Handler scheduler,
                         int flags) {

        context.registerReceiver(receiver, filter, broadcastPermission, scheduler);
        context.registerReceiver(receiver, filter, broadcastPermission, scheduler, flags);
    }
}
```

See

- OWASP - Mobile AppSec Verification Standard - Platform Interaction Requirements
- OWASP - Mobile Top 10 2016 Category M1 - Improper Platform Usage
- CWE - CWE-925 - Improper Verification of Intent by Broadcast Receiver
- CWE - CWE-926 - Improper Export of Android Application Components
- Android documentation - Broadcast Overview - Security considerations and best practices

Default:

Android applications can receive broadcasts from the system or other applications. Receiving intents is security-sensitive. For example, it has led in the past to the following vulnerabilities:

- CVE-2019-1677
- CVE-2015-1275

Receivers can be declared in the manifest or in the code to make them context-specific. If the receiver is declared in the manifest Android will start the application if it is not already running once a matching broadcast is received. The receiver is an entry point into the application.

Other applications can send potentially malicious broadcasts, so it is important to consider broadcasts as untrusted and to limit the applications that can send broadcasts to the receiver.

Permissions can be specified to restrict broadcasts to authorized applications. Restrictions can be enforced by both the sender and receiver of a broadcast. If permissions are specified when registering a broadcast receiver, then only broadcasters who were granted this permission can send a message to the receiver.

This rule raises an issue when a receiver is registered without specifying any broadcast permission.

Ask Yourself Whether

- The data extracted from intents is not sanitized.
- Intents broadcast is not restricted.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

Restrict the access to broadcasted intents. See the Android documentation for more information.

Sensitive Code Example

```
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.IntentFilter;
import android.os.Build;
import android.os.Handler;
import android.support.annotation.RequiresApi;

public class MyIntentReceiver {

    @RequiresApi(api = Build.VERSION_CODES.O)
    public void register(Context context, BroadcastReceiver receiver,
                         IntentFilter filter,
                         String broadcastPermission,
                         Handler scheduler,
                     int flags) {
        context.registerReceiver(receiver, filter); // Sensitive
        context.registerReceiver(receiver, filter, flags); // Sensitive

        // Broadcasting intent with "null" for broadcastPermission
        context.registerReceiver(receiver, filter, null, scheduler); // Sensitive
        context.registerReceiver(receiver, filter, null, scheduler, flags); // Sensitive
    }
}
```

Compliant Solution

```
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.IntentFilter;
import android.os.Build;
import android.os.Handler;
import android.support.annotation.RequiresApi;

public class MyIntentReceiver {

    @RequiresApi(api = Build.VERSION_CODES.O)
    public void register(Context context, BroadcastReceiver receiver,
                     IntentFilter filter,
                     String broadcastPermission,
                     Handler scheduler,
                     int flags) {

        context.registerReceiver(receiver, filter, broadcastPermission, scheduler);
        context.registerReceiver(receiver, filter, broadcastPermission, scheduler, flags);
    }
}
```

See

- OWASP - Mobile AppSec Verification Standard - Platform Interaction Requirements
- OWASP - Mobile Top 10 2016 Category M1 - Improper Platform Usage
- CWE - CWE-925 - Improper Verification of Intent by Broadcast Receiver
- CWE - CWE-926 - Improper Export of Android Application Components
- Android documentation - Broadcast Overview - Security considerations and best practices

---

Accessing Android external storage is security-sensitive (java:S5324)

Severidad: CRITICAL

How_to_fix:

Recommended Secure Coding Practices

- Use internal storage whenever possible as the system prevents other apps from accessing this location.
- Only use external storage if you need to share non-sensitive files with other applications.
- If your application has to use the external storage to store sensitive data, make sure it encrypts the files using EncryptedFile.
- Data coming from external storage should always be considered untrusted and should be validated.
- As some external storage can be removed, make sure to never store files on it that are critical for the usability of your application.

Compliant Solution

```
import android.content.Context;

public class AccessExternalFiles {

    public void accessFiles(Context context) {
        context.getFilesDir();
    }
}
```

See

- OWASP - Top 10 2021 Category A4 - Insecure Design
- Android Security tips on external file storage
- OWASP - Mobile AppSec Verification Standard - Data Storage and Privacy Requirements
- OWASP - Mobile Top 10 2016 Category M2 - Insecure Data Storage
- CWE - CWE-312 - Cleartext Storage of Sensitive Information

Root_cause:

Storing data locally is a common task for mobile applications. Such data includes files among other things. One convenient way to store files is to use the external file storage which usually offers a larger amount of disc space compared to internal storage.

Files created on the external storage are globally readable and writable. Therefore, a malicious application having the permissions `WRITE_EXTERNAL_STORAGE` or `READ_EXTERNAL_STORAGE` could try to read sensitive information from the files that other applications have stored on the external storage.

External storage can also be removed by the user (e.g when based on SD card) making the files unavailable to the application.

Assess_the_problem:

Ask Yourself Whether

Your application uses external storage to:

- store files that contain sensitive data.
- store files that are not meant to be shared with other application.
- store files that are critical for the application to work.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

```
import android.content.Context;

public class AccessExternalFiles {

    public void accessFiles(Context context) {
        context.getExternalFilesDir(null); // Sensitive
    }
}
```

Default:

Storing data locally is a common task for mobile applications. Such data includes files among other things. One convenient way to store files is to use the external file storage which usually offers a larger amount of disc space compared to internal storage.

Files created on the external storage are globally readable and writable. Therefore, a malicious application having the permissions `WRITE_EXTERNAL_STORAGE` or `READ_EXTERNAL_STORAGE` could try to read sensitive information from the files that other applications have stored on the external storage.

External storage can also be removed by the user (e.g when based on SD card) making the files unavailable to the application.

Ask Yourself Whether

Your application uses external storage to:

- store files that contain sensitive data.
- store files that are not meant to be shared with other application.
- store files that are critical for the application to work.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- Use internal storage whenever possible as the system prevents other apps from accessing this location.
- Only use external storage if you need to share non-sensitive files with other applications.
- If your application has to use the external storage to store sensitive data, make sure it encrypts the files using EncryptedFile.
- Data coming from external storage should always be considered untrusted and should be validated.
- As some external storage can be removed, make sure to never store files on it that are critical for the usability of your application.

Sensitive Code Example

```
import android.content.Context;

public class AccessExternalFiles {

    public void accessFiles(Context context) {
        context.getExternalFilesDir(null); // Sensitive
    }
}
```

Compliant Solution

```
import android.content.Context;

public class AccessExternalFiles {

    public void accessFiles(Context context) {
        context.getFilesDir();
    }
}
```

See

- OWASP - Top 10 2021 Category A4 - Insecure Design
- Android Security tips on external file storage
- OWASP - Mobile AppSec Verification Standard - Data Storage and Privacy Requirements
- OWASP - Mobile Top 10 2016 Category M2 - Insecure Data Storage
- CWE - CWE-312 - Cleartext Storage of Sensitive Information

Using publicly writable directories is security-sensitive (java:S5443)

Severidad: CRITICAL

How_to_fix:

Recommended Secure Coding Practices

- Use a dedicated sub-folder with tightly controlled permissions
- Use secure-by-design APIs to create temporary files. Such API will make sure:
  - The generated filename is unpredictable
  - The file is readable and writable only by the creating user ID
  - The file descriptor is not inherited by child processes
  - The file will be destroyed as soon as it is closed

Compliant Solution

```
new File("/myDirectory/myfile.txt");  // Compliant

File.createTempFile("prefix", "suffix", new File("/mySecureDirectory"));  // Compliant

if(SystemUtils.IS_OS_UNIX) {
  FileAttribute<Set<PosixFilePermission>> attr = PosixFilePermissions.asFileAttribute(PosixFilePermissions.fromString("rwx------"));
  Files.createTempFile("prefix", "suffix", attr); // Compliant
}
else {
  File f = Files.createTempFile("prefix", "suffix").toFile();  // Compliant
  f.setReadable(true, true);
  f.setWritable(true, true);
  f.setExecutable(true, true);
}
```

See

- OWASP - Top 10 2021 Category A1 - Broken Access Control
- OWASP - Top 10 2017 Category A5 - Broken Access Control
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- CWE - CWE-377 - Insecure Temporary File
- CWE - CWE-379 - Creation of Temporary File in Directory with Incorrect Permissions
- OWASP, Insecure Temporary File
- STIG Viewer - Application Security and Development: V-222567 - The application must not be vulnerable to race conditions.

Assess_the_problem:

Ask Yourself Whether

- Files are read from or written into a publicly writable folder
- The application creates files with predictable names into a publicly writable folder

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

```
new File("/tmp/myfile.txt"); // Sensitive
Paths.get("/tmp/myfile.txt"); // Sensitive
```

```
java.io.File.createTempFile("prefix", "suffix"); // Sensitive, will be in the default temporary-file directory.
java.nio.file.Files.createTempDirectory("prefix"); // Sensitive, will be in the default temporary-file directory.

Map<String, String> env = System.getenv();
env.get("TMP"); // Sensitive
```

Root_cause:

Operating systems have global directories where any user has write access. Those folders are mostly used as temporary storage areas like `/tmp` in Linux based systems. An application manipulating files from these folders is exposed to race conditions on filenames: a malicious user can try to create a file with a predictable name before the application does. A successful attack can result in other files being accessed, modified, corrupted or deleted. This risk is even higher if the application runs with elevated permissions.

In the past, it has led to the following vulnerabilities:

- CVE-2012-2451
- CVE-2015-1838

This rule raises an issue whenever it detects a hard-coded path to a publicly writable directory like `/tmp` (see examples bellow). It also detects access to environment variables that point to publicly writable directories, e.g., `TMP` and `TMPDIR`.

- `/tmp`
- `/var/tmp`
- `/usr/tmp`
- `/dev/shm`
- `/dev/mqueue`
- `/run/lock`
- `/var/run/lock`
- `/Library/Caches`
- `/Users/Shared`
- `/private/tmp`
- `/private/var/tmp`
- `\Windows\Temp`
- `\Temp`
- `\TMP`

Default:

Operating systems have global directories where any user has write access. Those folders are mostly used as temporary storage areas like `/tmp` in Linux based systems. An application manipulating files from these folders is exposed to race conditions on filenames: a malicious user can try to create a file with a predictable name before the application does. A successful attack can result in other files being accessed, modified, corrupted or deleted. This risk is even higher if the application runs with elevated permissions.

In the past, it has led to the following vulnerabilities:

- CVE-2012-2451
- CVE-2015-1838

This rule raises an issue whenever it detects a hard-coded path to a publicly writable directory like `/tmp` (see examples bellow). It also detects access to environment variables that point to publicly writable directories, e.g., `TMP` and `TMPDIR`.

- `/tmp`
- `/var/tmp`
- `/usr/tmp`
- `/dev/shm`
- `/dev/mqueue`
- `/run/lock`
- `/var/run/lock`
- `/Library/Caches`
- `/Users/Shared`
- `/private/tmp`
- `/private/var/tmp`
- `\Windows\Temp`
- `\Temp`
- `\TMP`

Ask Yourself Whether

- Files are read from or written into a publicly writable folder
- The application creates files with predictable names into a publicly writable folder

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- Use a dedicated sub-folder with tightly controlled permissions
- Use secure-by-design APIs to create temporary files. Such API will make sure:
    - The generated filename is unpredictable
    - The file is readable and writable only by the creating user ID
    - The file descriptor is not inherited by child processes
    - The file will be destroyed as soon as it is closed

Sensitive Code Example

```
new File("/tmp/myfile.txt"); // Sensitive
Paths.get("/tmp/myfile.txt"); // Sensitive

java.io.File.createTempFile("prefix", "suffix"); // Sensitive, will be in the default temporary-file directory.
java.nio.file.Files.createTempDirectory("prefix"); // Sensitive, will be in the default temporary-file directory.

Map<String, String> env = System.getenv();
env.get("TMP"); // Sensitive
```

Compliant Solution

```
new File("/myDirectory/myfile.txt");  // Compliant

File.createTempFile("prefix", "suffix", new File("/mySecureDirectory"));  // Compliant

if(SystemUtils.IS_OS_UNIX) {
  FileAttribute<Set<PosixFilePermission>> attr = PosixFilePermissions.asFileAttribute(PosixFilePermissions.fromString("rwx------"));
  Files.createTempFile("prefix", "suffix", attr); // Compliant
}
else {
  File f = Files.createTempFile("prefix", "suffix").toFile();  // Compliant
  f.setReadable(true, true);
  f.setWritable(true, true);
  f.setExecutable(true, true);
}
```

See

- OWASP - Top 10 2021 Category A1 - Broken Access Control
- OWASP - Top 10 2017 Category A5 - Broken Access Control
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- CWE - CWE-377 - Insecure Temporary File
- CWE - CWE-379 - Creation of Temporary File in Directory with Incorrect Permissions
- OWASP, Insecure Temporary File
- STIG Viewer - Application Security and Development: V-222567 - The application must not be vulnerable to race conditions.

Insecure temporary file creation methods should not be used (java:S5445)

Severidad: CRITICAL

Introduction:

Temporary files are considered insecurely created when the file existence check is performed separately from the actual file creation. Such a situation can occur when creating temporary files using normal file handling functions or when using dedicated temporary file handling functions that are not atomic.

Root_cause:

Creating temporary files in a non-atomic way introduces race condition issues in the application’s behavior. Indeed, a third party can create a given file between when the application chooses its name and when it creates it.

In such a situation, the application might use a temporary file that it does not entirely control. In particular, this file’s permissions might be different than expected. This can lead to trust boundary issues.

What is the potential impact?

Attackers with control over a temporary file used by a vulnerable application will be able to modify it in a way that will affect the application’s logic. By changing this file’s Access Control List or other operating system-level properties, they could prevent the file from being deleted or emptied. They may also alter the file’s content before or while the application uses it.

Depending on why and how the affected temporary files are used, the exploitation of a race condition in an application can have various consequences. They can range from sensitive information disclosure to more serious application or hosting infrastructure compromise.

Information disclosure

Because attackers can control the permissions set on temporary files and prevent their removal, they can read what the application stores in them. This might be especially critical if this information is sensitive.

For example, an application might use temporary files to store users' session-related information. In such a case, attackers controlling those files can access session-stored information. This might allow them to take over authenticated users' identities and entitlements.

Attack surface extension

An application might use temporary files to store technical data for further reuse or as a communication channel between multiple components. In that case, it might consider those files part of the trust boundaries and use their content without additional security validation or sanitation. In such a case, an attacker controlling the file content might use it as an attack vector for further compromise.

For example, an application might store serialized data in temporary files for later use. In such a case, attackers controlling those files' content can change it in a way that will lead to an insecure deserialization exploitation. It might allow them to execute arbitrary code on the application hosting server and take it over.

How_to_fix:

The following code example is vulnerable to a race condition attack because it creates a temporary file using an unsafe API function.

Noncompliant code example

```
import java.io.File;
import java.io.IOException;

protected void Example() throws IOException {
    File tempDir;
    tempDir = File.createTempFile("", ".");
    tempDir.delete();
    tempDir.mkdir();  // Noncompliant
}
```

Compliant solution

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;

protected void Example() throws IOException {
    Path tempPath = Files.createTempDirectory("");
    File tempDir = tempPath.toFile();
}
```

How does this work?

Applications should create temporary files so that no third party can read or modify their content. It requires that the files' name, location, and permissions are carefully chosen and set. This can be achieved in multiple ways depending on the applications' technology stacks.

Use a secure API function

Temporary files handling APIs generally provide secure functions to create temporary files. In most cases, they operate in an atomical way, creating and opening a file with a unique and unpredictable name in a single call. Those functions can often be used to replace less secure alternatives without requiring important development efforts.

Here, the example compliant code uses the safer `Files.createTempDirectory` function to manage the creation of temporary directories.

Strong security controls

Temporary files can be created using unsafe functions and API as long as strong security controls are applied. Non-temporary file-handling functions and APIs can also be used for that purpose.

In general, applications should ensure that attackers can not create a file before them. This turns into the following requirements when creating the files:

- Files should be created in a non-public directory.
- File names should be unique.
- File names should be unpredictable. They should be generated using a cryptographically secure random generator.
- File creation should fail if a target file already exists.

Moreover, when possible, it is recommended that applications destroy temporary files after they have finished using them.

Resources:

Documentation

- OWASP - Insecure Temporary File

Standards

- OWASP - Top 10 2021 Category A1 - Broken Access Control
- OWASP - Top 10 2017 Category A9 - Using Components with Known Vulnerabilities
- CWE - CWE-377 - Insecure Temporary File
- CWE - CWE-379 - Creation of Temporary File in Directory with Incorrect Permissions
- STIG Viewer - Application Security and Development: V-222567 - The application must not be vulnerable to race conditions.

Types used as keys in Maps should implement Comparable (java:S6411)

Severidad: MAJOR

Resources:

- https://dzone.com/articles/java-8-hashmaps-keys-and-the-comparable-interface
- https://github.com/openjdk/jdk/blob/4927ee426aedbeea0f4119bac0a342c6d3576762/src/hotspot/share/runtime/synchronizer.cpp#L760L798
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Comparable.html

Root_cause:

Maps use hashes of the keys to select a bucket to store data in. Objects that hash to the same value will be added to the same bucket.

When the hashing function has a poor distribution, buckets can grow to very large sizes. This may negatively affect lookup performance, as, by default, matching a key within a bucket has linear complexity.

In addition, as the default hashCode function can be selected at runtime, performance expectations cannot be maintained.

Implementing Comparable mitigates the performance issue for objects that hash to the same value.

Noncompliant code example

```
class MyKeyType {
    // ...
}

class Program {
    Map<MyKeyType, MyValueType> data = new HashMap<>(); // Noncompliant

    Map<MyKeyType, MyValueType> buildMap() { // Noncompliant
        //...
    }
}
```

Compliant solution

```
class MyKeyType implements Comparable<MyKeyType> {
    // ...
}

class MyChildKeyType extends MyKeyType {
    // ...
}

class Program {
    Map<MyKeyType, MyValueType> data = new HashMap<>();
    Map<MyChildKeyType, MyValueType> data = new HashMap<>();

    Map<MyKeyType, MyValueType> buildMap() {
        //...
    }
}
```

Password hashing functions should use an unpredictable salt (java:S2053)

Severidad: CRITICAL

Root_cause:

During the process of password hashing, an additional component, known as a "salt," is often integrated to bolster the overall security. This salt, acting as a defensive measure, primarily wards off certain types of attacks that leverage pre-computed tables to crack passwords.

However, potential risks emerge when the salt is deemed insecure. This can occur when the salt is consistently the same across all users or when it is too short or predictable. In scenarios where users share the same password and salt, their password hashes will inevitably mirror each other. Similarly, a short salt heightens the probability of multiple users unintentionally having identical salts, which can potentially lead to identical password hashes. These identical hashes streamline the process for potential attackers to recover clear-text passwords. Thus, the emphasis on implementing secure, unique, and sufficiently lengthy salts in password-hashing functions is vital.

What is the potential impact?

Despite best efforts, even well-guarded systems might have vulnerabilities that could allow an attacker to gain access to the hashed passwords. This could be due to software vulnerabilities, insider threats, or even successful phishing attempts that give attackers the access they need.

Once the attacker has these hashes, they will likely attempt to crack them using a couple of methods. One is brute force, which entails trying every possible combination until the correct password is found. While this can be time-consuming, having the same salt for all users or a short salt can make the task significantly easier and faster.

If multiple users have the same password and the same salt, their password hashes would be identical. This means that if an attacker successfully cracks one hash, they have effectively cracked all identical ones, granting them access to multiple accounts at once.

A short salt, while less critical than a shared one, still increases the odds of different users having the same salt. This might create clusters of password hashes with identical salt that can then be attacked as explained before.

With short salts, the probability of a collision between two users' passwords and salts couple might be low depending on the salt size. The shorter the salt, the higher the collision probability. In any case, using longer, cryptographically secure salt should be preferred.

Exceptions

To securely store password hashes, it is a recommended to rely on key derivation functions that are computationally intensive. Examples of such functions are:

- Argon2
- PBKDF2
- Scrypt
- Bcrypt

When they are used for password storage, using a secure, random salt is required.

However, those functions can also be used for other purposes such as master key derivation or password-based pre-shared key generation. In those cases, the implemented cryptographic protocol might require using a fixed salt to derive keys in a deterministic way. In such cases, using a fixed salt is safe and accepted.

Introduction:

This vulnerability increases the likelihood that attackers are able to compute the cleartext of password hashes.

How_to_fix:

The following code contains examples of hard-coded salts.

Noncompliant code example

```
import javax.crypto.spec.PBEParameterSpec;

public void hash() {
    byte[] salt = "salty".getBytes();
    PBEParameterSpec cipherSpec = new PBEParameterSpec(salt, 10000); // Noncompliant
}
```

Compliant solution

```
import java.security.SecureRandom;
import javax.crypto.spec.PBEParameterSpec;

public void hash() {
    SecureRandom random = new SecureRandom();
    byte[] salt = new byte[32];
    random.nextBytes(salt);

    PBEParameterSpec cipherSpec = new PBEParameterSpec(salt, 10000);
}
```

How does this work?

This code ensures that each user’s password has a unique salt value associated with it. It generates a salt randomly and with a length that provides the required security level. It uses a salt length of at least 32 bytes (256 bits), as recommended by industry standards.

Here, the compliant code example ensures the salt is random and has a sufficient length by calling the `nextBytes` method from the `SecureRandom` class with a salt buffer of 16 bytes. This class implements a cryptographically secure pseudo-random number generator.

Resources:

Standards

- OWASP - Top 10 2021 Category A2 - Cryptographic Failures
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- CWE - CWE-759 - Use of a One-Way Hash without a Salt
- CWE - CWE-760 - Use of a One-Way Hash with a Predictable Salt
- STIG Viewer - Application Security and Development: V-222542 - The application must only store cryptographic representations of passwords.

The diamond operator ("<>") should be used (java:S2293)

Severidad: MINOR

How_to_fix:

The type argument should be omitted in the initialization if it is already present in the declaration of a field or variable.

For instance, a field with type `List<String>` can be initialized with `ArrayList<>()`, as the compiler will infer that `ArrayList<String>()` is the actually desired call.

Noncompliant code example

```
List<String> strings = new ArrayList<String>();                       // Noncompliant, the compiler can infer the type argument of the cons
Map<String,List<Integer>> map = new HashMap<String,List<Integer>>();  // Noncompliant, the compiler can also infer complex type arguments
```

Compliant solution

```
List<String> strings = new ArrayList<>();        // Compliant, the compiler will infer the type argument
Map<String,List<Integer>> map = new HashMap<>(); // Compliant, the compiler will infer the type argument
```

References

- The Java™ Tutorials - Type Inference (Generics)

Root_cause:

Java uses angular brackets (`<` and `>`) to provide a specific type (the "type argument") to a generic type. For instance, `List` is a generic type, so a list containing strings can be declared with `List<String>`.

Prior to Java 7, the type argument had to be provided explicitly for every occurrence where generics were used. This often caused redundancy, as the type argument would have to be provided both when a field is declared and initialized.

Java 7 introduced the diamond operator (`<>`) to reduce the code’s verbosity in some situations. The type argument between the angular brackets should be omitted if the compiler can infer it.

Since the diamond operator was only introduced in Java 7, this rule is automatically disabled when the project’s `sonar.java.source` is lower than `7`.

"Collection.toArray()" should be passed an array of the proper type (java:S3020)

Severidad: MINOR

Resources:

- docs.oracle - Collection.toArray()

Root_cause:

The `Collection.toArray()` method returns an `Object[]` when no arguments are provided to it. This can lead to a `ClassCastException` at runtime if you try to cast the returned array to an array of a specific type. Instead, use this method by providing an array of the desired type as the argument.

Note that passing a `new T[0]` array of length zero as the argument is more efficient than a pre-sized array `new T[size]`.

Noncompliant code example

```
public String [] getStringArray(List<String> strings) {
  return (String []) strings.toArray();  // Noncompliant, a ClassCastException will be thrown here
}
```

Compliant solution

```
public String [] getStringArray(List<String> strings) {
  return strings.toArray(new String[0]); // Compliant, the toArray method will return an array of the desired type, and we can remove the c
}
public String [] getPresizedStringArray(List<String> strings) {
  return strings.toArray(new String[strings.size()]); // Compliant, but slightly less efficient than the previous example
}
```

Non-primitive fields should not be "volatile" (java:S3077)

Severidad: MINOR

Root_cause:

Marking an array `volatile` means that the array itself will always be read fresh and never thread cached, but the items *in* the array will not be. Similarly, marking a mutable object field `volatile` means the object *reference* is `volatile` but the object itself is not, and other threads may not see updates to the object state.

This can be salvaged with arrays by using the relevant AtomicArray class, such as `AtomicIntegerArray`, instead. For mutable objects, the `volatile` should be removed, and some other method should be used to ensure thread-safety, such as synchronization, or ThreadLocal storage.

Noncompliant code example

```
private volatile int [] vInts;  // Noncompliant
private volatile MyObj myObj;  // Noncompliant
```

Compliant solution

```
private AtomicIntegerArray vInts;
private MyObj myObj;
```

Resources:

- CERT, CON50-J. - Do not assume that declaring a reference volatile guarantees safe publication of the members of the referenced object

---

"volatile" variables should not be used with compound operators (java:S3078)

Severidad: MAJOR

Resources:

- CERT, VNA02-J. - Ensure that compound operations on shared variables are atomic

Root_cause:

Using compound operators as well as increments and decrements (and toggling, in the case of `boolean`s) on primitive fields are not atomic operations. That is, they don’t happen in a single step. For instance, when a `volatile` primitive field is incremented or decremented you run the risk of data loss if threads interleave in the steps of the update. Instead, use a guaranteed-atomic class such as `AtomicInteger`, or synchronize the access.

Noncompliant code example

```
private volatile int count = 0;
private volatile boolean boo = false;

public void incrementCount() {
  count++;  // Noncompliant
}
```

```
public void toggleBoo(){
  boo = !boo;  // Noncompliant
}
```

Compliant solution

```
private AtomicInteger count = 0;
private boolean boo = false;

public void incrementCount() {
  count.incrementAndGet();
}

public synchronized void toggleBoo() {
  boo = !boo;
}
```

"java.nio.Files#delete" should be preferred (java:S4042)

Severidad: MAJOR

Root_cause:

When `java.io.File#delete` fails, this `boolean` method simply returns `false` with no indication of the cause. On the other hand, when `java.nio.file.Files#delete` fails, this `void` method returns one of a series of exception types to better indicate the cause of the failure. And since more information is generally better in a debugging situation, `java.nio.file.Files#delete` is the preferred option.

Noncompliant code example

```
public void cleanUp(Path path) {
  File file = new File(path);
  if (!file.delete()) {  // Noncompliant
    //...
  }
}
```

Compliant solution

```
public void cleanUp(Path path) throws NoSuchFileException, DirectoryNotEmptyException, IOException {
  Files.delete(path);
}
```

Methods should not have identical implementations (java:S4144)

Severidad: MAJOR

Root_cause:

Two methods having the same implementation are suspicious. It might be that something else was intended. Or the duplication is intentional, which becomes a maintenance burden.

```
private final static String CODE = "bounteous";

public String calculateCode() {
  doTheThing();
  return CODE;
}

public String getName() {  // Noncompliant: duplicates calculateCode
  doTheThing();
  return CODE;
}
```

If the identical logic is intentional, the code should be refactored to avoid duplication. For example, by having both methods call the same method or by having one implementation invoke the other.

```
private final static String CODE = "bounteous";

public String getCode() {
  doTheThing();
  return CODE;
}

public String getName() { // The intent is clear
  return getCode();
}
```

Exceptions

Methods that are not accessors (getters and setters), with fewer than 2 statements are ignored.

Assignments should not be redundant (java:S4165)

Severidad: MAJOR

Root_cause:

The transitive property says that if `a == b` and `b == c`, then `a == c`. In such cases, there’s no point in assigning `a` to `c` or vice versa because they’re already equivalent.

This rule raises an issue when an assignment is useless because the assigned-to variable already holds the value on all execution paths.

Noncompliant code example

```
a = b;
c = a;
b = c; // Noncompliant: c and b are already the same
```

Compliant solution

```
a = b;
c = a;
```

Local constants should follow naming conventions for constants (java:S4174)

Severidad: MINOR

Root_cause:

Shared coding conventions allow teams to collaborate efficiently. This rule checks that all local, `final`, initialized, primitive variables, have names that match a provided regular expression.

Noncompliant code example

With the default regular expression `^[A-Z][A-Z0-9]*(_[A-Z0-9]+)*$`:

```
public void doSomething() {
  final int local = 42;
  ...
}
```

Compliant solution

```
public void doSomething() {
  final int LOCAL = 42;
  ...
}
```

"Stream.collect()" calls should not be redundant (java:S4266)

Severidad: MINOR

Root_cause:

When using the `Stream` API, call chains should be simplified as much as possible to improve readability and maintainability.

This rule raises an issue when one of the following substitution can be made:

Original	Preferred
`stream.collect(counting())`	`stream.count()`
`stream.collect(maxBy(comparator))`	`stream.max(comparator)`
`stream.collect(minBy(comparator))`	`stream.min(comparator)`
`stream.collect(mapping(mapper))`	`stream.map(mapper).collect()`

Original	Preferred
`stream.collect(reducing(...))`	`stream.reduce(...)`
`stream.collect(summingInt(mapper))`	`stream.mapToInt(mapper).sum()`
`stream.collect(summingLong(mapper))`	`stream.mapToLong(mapper).sum()`
`stream.collect(summingDouble(mapper))`	`stream.mapToDouble(mapper).sum()`

Noncompliant code example

```
int count = stream.collect(counting());  // Noncompliant
```

Compliant solution

```
int count = stream.count();
```

Spring components should use constructor injection (java:S4288)

Severidad: MAJOR

Root_cause:

Spring `@Controller`, `@Service`, and `@Repository` classes are singletons by default, meaning only one instance of the class is ever instantiated in the application. Typically such a class might have a few `static` members, such as a logger, but all non-static members should be managed by Spring and supplied via constructor injection rather than by field injection.

This rule raise an issue when any non-`static` member of a Spring component has an injection annotation.

Noncompliant code example

```
@Controller
public class HelloWorld {

  @Autowired
  private String name = null; // Noncompliant

}
```

Compliant solution

As of Spring 4.3

```
@Controller
public class HelloWorld {

  private String name = null;

  HelloWorld(String name) {
    this.name = name;
  }
}
```

Before Spring 4.3

```
@Controller
public class HelloWorld {

  private String name = null;

  @Autowired
  HelloWorld(String name) {
    this.name = name;
  }
}
```

"compareTo" should not be overloaded (java:S4351)

Severidad: MAJOR

Root_cause:

When implementing the `Comparable<T>.compareTo` method, the parameter’s type has to match the type used in the `Comparable` declaration. When a different type is used this creates an overload instead of an override, which is unlikely to be the intent.

This rule raises an issue when the parameter of the `compareTo` method of a class implementing `Comparable<T>` is not same as the one used in the `Comparable` declaration.

Noncompliant code example

```
public class Foo {
  static class Bar implements Comparable<Bar> {
    public int compareTo(Bar rhs) {
      return -1;
    }
  }

  static class FooBar extends Bar {
    public int compareTo(FooBar rhs) {  // Noncompliant: Parameter should be of type Bar
      return 0;
    }
  }
}
```

Compliant solution

```
public class Foo {
  static class Bar implements Comparable<Bar> {
    public int compareTo(Bar rhs) {
      return -1;
    }
  }

  static class FooBar extends Bar {
    public int compareTo(Bar rhs) {
      return 0;
    }
  }
}
```

"else" statements should be clearly matched with an "if" (java:S5261)

Severidad: MAJOR

Resources:

- https://en.wikipedia.org/wiki/Dangling_else

Root_cause:

The dangling `else` problem appears when nested `if`/`else` statements are written without curly braces. In this case, `else` is associated with the nearest `if` but that is not always obvious and sometimes the indentation can also be misleading.

This rules reports `else` statements that are difficult to understand, because they are inside nested `if` statements without curly braces.

Adding curly braces can generally make the code clearer (see rule S121), and in this situation of dangling `else`, it really clarifies the intention of the code.

Noncompliant code example

```
 if (a)
   if (b)
     d++;
 else     // Noncompliant, is the "else" associated with "if(a)" or "if (b)"? (the answer is "if(b)")
   e++;
```

Compliant solution

```
 if (a) {
   if (b) {
     d++;
   }
 } else { // Compliant, there is no doubt the "else" is associated with "if(a)"
   e++;
 }
```

Broadcasting intents is security-sensitive (java:S5320)

Severidad: CRITICAL

Assess_the_problem:

Ask Yourself Whether

- The intent contains sensitive information.
- Intent reception is not restricted.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

```
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.Build;
import android.os.Bundle;
import android.os.Handler;
import android.os.UserHandle;
import android.support.annotation.RequiresApi;

public class MyIntentBroadcast {
    @RequiresApi(api = Build.VERSION_CODES.JELLY_BEAN_MR1)
    public void broadcast(Intent intent, Context context, UserHandle user,
                          BroadcastReceiver resultReceiver, Handler scheduler, int initialCode,
                          String initialData, Bundle initialExtras,
                          String broadcastPermission) {
        context.sendBroadcast(intent); // Sensitive
        context.sendBroadcastAsUser(intent, user); // Sensitive

        // Broadcasting intent with "null" for receiverPermission
        context.sendBroadcast(intent, null); // Sensitive
        context.sendBroadcastAsUser(intent, user, null); // Sensitive
        context.sendOrderedBroadcast(intent, null); // Sensitive
        context.sendOrderedBroadcastAsUser(intent, user, null, resultReceiver,
                scheduler, initialCode, initialData, initialExtras); // Sensitive
    }
}
```

Default:

In Android applications, broadcasting intents is security-sensitive. For example, it has led in the past to the following vulnerability:

- CVE-2018-9489

By default, broadcasted intents are visible to every application, exposing all sensitive information they contain.

This rule raises an issue when an intent is broadcasted without specifying any "receiver permission".

Ask Yourself Whether

- The intent contains sensitive information.
- Intent reception is not restricted.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

Restrict the access to broadcasted intents. See Android documentation for more information.

Sensitive Code Example

```
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.Build;
import android.os.Bundle;
import android.os.Handler;
import android.os.UserHandle;
import android.support.annotation.RequiresApi;

public class MyIntentBroadcast {
    @RequiresApi(api = Build.VERSION_CODES.JELLY_BEAN_MR1)
    public void broadcast(Intent intent, Context context, UserHandle user,
                          BroadcastReceiver resultReceiver, Handler scheduler, int initialCode,
                          String initialData, Bundle initialExtras,
                          String broadcastPermission) {
        context.sendBroadcast(intent); // Sensitive
        context.sendBroadcastAsUser(intent, user); // Sensitive

        // Broadcasting intent with "null" for receiverPermission
        context.sendBroadcast(intent, null); // Sensitive
        context.sendBroadcastAsUser(intent, user, null); // Sensitive
        context.sendOrderedBroadcast(intent, null); // Sensitive
        context.sendOrderedBroadcastAsUser(intent, user, null, resultReceiver,
                scheduler, initialCode, initialData, initialExtras); // Sensitive
    }
}
```

Compliant Solution

```
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.Build;
import android.os.Bundle;
import android.os.Handler;
import android.os.UserHandle;
import android.support.annotation.RequiresApi;

public class MyIntentBroadcast {
    @RequiresApi(api = Build.VERSION_CODES.JELLY_BEAN_MR1)
    public void broadcast(Intent intent, Context context, UserHandle user,
                          BroadcastReceiver resultReceiver, Handler scheduler, int initialCode,
                          String initialData, Bundle initialExtras,
                          String broadcastPermission) {

        context.sendBroadcast(intent, broadcastPermission);
        context.sendBroadcastAsUser(intent, user, broadcastPermission);
        context.sendOrderedBroadcast(intent, broadcastPermission);
        context.sendOrderedBroadcastAsUser(intent, user,broadcastPermission, resultReceiver,
                scheduler, initialCode, initialData, initialExtras);
    }
}
```

See

- OWASP - Top 10 2021 Category A4 - Insecure Design
- OWASP - Mobile AppSec Verification Standard - Platform Interaction Requirements
- OWASP - Mobile Top 10 2016 Category M1 - Improper Platform Usage
- CWE - CWE-927 - Use of Implicit Intent for Sensitive Communication
- Android documentation - Broadcast Overview - Security considerations and best practices

How_to_fix:

Recommended Secure Coding Practices

Restrict the access to broadcasted intents. See Android documentation for more information.

Compliant Solution

```
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.Build;
import android.os.Bundle;
import android.os.Handler;
import android.os.UserHandle;
import android.support.annotation.RequiresApi;

public class MyIntentBroadcast {
    @RequiresApi(api = Build.VERSION_CODES.JELLY_BEAN_MR1)
    public void broadcast(Intent intent, Context context, UserHandle user,
                          BroadcastReceiver resultReceiver, Handler scheduler, int initialCode,
                          String initialData, Bundle initialExtras,
                          String broadcastPermission) {

        context.sendBroadcast(intent, broadcastPermission);
        context.sendBroadcastAsUser(intent, user, broadcastPermission);
        context.sendOrderedBroadcast(intent, broadcastPermission);
        context.sendOrderedBroadcastAsUser(intent, user,broadcastPermission, resultReceiver,
                scheduler, initialCode, initialData, initialExtras);
    }
}
```

See

- OWASP - Top 10 2021 Category A4 - Insecure Design
- OWASP - Mobile AppSec Verification Standard - Platform Interaction Requirements
- OWASP - Mobile Top 10 2016 Category M1 - Improper Platform Usage
- CWE - CWE-927 - Use of Implicit Intent for Sensitive Communication

- Android documentation - Broadcast Overview - Security considerations and best practices

Root_cause:

In Android applications, broadcasting intents is security-sensitive. For example, it has led in the past to the following vulnerability:

- CVE-2018-9489

By default, broadcasted intents are visible to every application, exposing all sensitive information they contain.

This rule raises an issue when an intent is broadcasted without specifying any "receiver permission".

Collection constructors should not be used as java.util.function.Function (java:S5329)

Severidad: MAJOR

Root_cause:

It is very common to pass a collection constructor reference as an argument, for example `Collectors.toCollection(ArrayList::new)` takes the `ArrayList::new` constructor. When the method expects a `java.util.function.Supplier` it is perfectly fine. However when the method argument type is `java.util.function.Function` it means that an argument will be passed to the constructor.

The first argument of Collections constructors is usually an integer representing its "initial capacity". This is generally not what the developer expects, but the memory allocation is not visible at first glance.

This rule raises an issue when a collection constructor is passed by reference as a `java.util.function.Function` argument.

Noncompliant code example

```
Arrays.asList(1, 2, 54000).stream().collect(Collectors.toMap(Function.identity(), ArrayList::new)); // Noncompliant, "ArrayList::new" unint
```

Compliant solution

```
Arrays.asList(1, 2, 54000).stream().collect(Collectors.toMap(Function.identity(), id -> new ArrayList<>())); // Compliant, explicitly show
```

AssertJ assertions with "Consumer" arguments should contain assertion inside consumers (java:S6103)

Severidad: MAJOR

Root_cause:

AssertJ assertions taking `Consumer` objects as arguments are expected to contain "requirements", which should themselves be expressed as assertions. This concerns the following methods: allSatisfy, anySatisfy, hasOnlyOneElementSatisfying, isInstanceOfSatisfying, noneSatisfy, satisfies, satisfiesAnyOf, zipSatisfy.

These methods are assuming the `Consumer` will do the assertions itself. If you do not do any assertion in the `Consumer`, it probably means that you are inadvertently only partially testing your object.

This rule raises an issue when a `Consumer` argument of any of the above methods does not contain any assertion.

Noncompliant code example

```
assertThat(myObject).isInstanceOfSatisfying(String.class, s -> "Hello".equals(s)); // Noncompliant - not testing the string value
assertThat(myObject).satisfies("Hello"::equals); // Noncompliant - not testing the string value
```

Compliant solution

```
assertThat(myObject).isInstanceOfSatisfying(String.class, s -> assertThat(s).isEqualTo("Hello"));
assertThat(myObject).satisfies(obj -> assertThat(obj).isEqualTo("Hello"));
```

Map "computeIfAbsent()" and "computeIfPresent()" should not be used to add "null" values. (java:S6104)

Severidad: CRITICAL

Root_cause:

Map computeIfAbsent and computeIfPresent methods are convenient to avoid the cumbersome process to check if a key exists or not, followed by the addition of the entry. However, when the function used to compute the value returns `null`, the entry `key->null` will not be added to the Map. Furthermore, in the case of `computeIfPresent`, if the key is present the entry will be removed. These methods should therefore not be used to conditionally add an entry with a null value. The traditional way should be used instead.

This rule raises an issue when `computeIfAbsent` or `computeIfPresent` is used with a lambda always returning null.

Noncompliant code example

```
map.computeIfAbsent(key, k -> null); // Noncompliant, the map will not contain an entry key->null.
map.computeIfPresent(key, (k, oldValue) -> null); // Noncompliant
```

Compliant solution

```
if (!map.containsKey(key)) {
    map.put(key, null);
}
if (map.containsKey(key)) {
    map.put(key, null);
}
```

Resources:

Related rules

- S3824 - "Map.get" and value test should be replaced with a single method call

---

Pattern Matching for "instanceof" operator should be used instead of simple "instanceof" + cast (java:S6201)

Severidad: MINOR

Root_cause:

In Java 16, the feature "Pattern matching for instanceof" is finalized and can be used in production. Previously developers needed to do 3 operations in order to do this: check the variable type, cast it and assign the casted value to the new variable. This approach is quite verbose and can be replaced with pattern matching for `instanceof`, doing these 3 actions (check, cast and assign) in one expression.

This rule raises an issue when an `instanceof` check followed by a cast and an assignment could be replaced by pattern matching.

Noncompliant code example

```
int f(Object o) {
  if (o instanceof String) {  // Noncompliant
    String string = (String) o;
    return string.length();
  }
  return 0;
}
```

Compliant solution

```
int f(Object o) {
  if (o instanceof String string) {  // Compliant
    return string.length();
  }
  return 0;
}
```

Resources:

- JEP 394: Pattern Matching for instanceof

Operator "instanceof" should be used instead of "A.class.isInstance()" (java:S6202)

Severidad: MAJOR

Root_cause:

The `instanceof` construction is a preferred way to check whether a variable can be cast to some type statically because a compile-time error will occur in case of incompatible types. The method isInstance() from `java.lang.Class` works differently and does type check at runtime only, incompatible types will therefore not be detected early in the development, potentially resulting in dead code. The `isInstance()` method should only be used in dynamic cases when the `instanceof` operator can't be used.

This rule raises an issue when `isInstance()` is used and could be replaced with an `instanceof` check.

Noncompliant code example

```
int f(Object o) {
  if (String.class.isInstance(o)) {  // Noncompliant
    return 42;
  }
  return 0;
}

int f(Number n) {
  if (String.class.isInstance(n)) {  // Noncompliant
    return 42;
  }
  return 0;
}
```

Compliant solution

```
int f(Object o) {
  if (o instanceof String) {  // Compliant
    return 42;
  }
  return 0;
}

int f(Number n) {
  if (n instanceof String) {  // Compile-time error
    return 42;
  }
  return 0;
}

boolean fun(Object o, String c) throws ClassNotFoundException
{
  return Class.forName(c).isInstance(o); // Compliant, can't use instanceof operator here
}
```

Text blocks should not be used in complex expressions (java:S6203)

Severidad: MINOR

Resources:

- JEP 378: Text Blocks
- Programmer’s Guide To Text Blocks, by Jim Laskey and Stuart Marks

Root_cause:

In Java 15 Text Blocks are official and can be used just like an ordinary String. However, when they are used to represent a big chunk of text, they should not be used directly in complex expressions, as it decreases the readability. In this case, it is better to extract the text block into a variable or a field.

This rule reports an issue when a text block longer than a number of lines given as a parameter is directly used within a lambda expression.

Noncompliant code example

```
listOfString.stream()
  .map(str -> !"""
    <project>
      <modelVersion>4.0.0</modelVersion>
      <parent>
        <groupId>com.mycompany.app</groupId>
        <artifactId>my-app</artifactId>
        <version>1</version>
      </parent>

      <groupId>com.mycompany.app</groupId>
      <artifactId>my-module</artifactId>
      <version>1</version>
    </project>
    """.equals(str));
```

Compliant solution

```
String myTextBlock = """
    <project>
      <modelVersion>4.0.0</modelVersion>
      <parent>
        <groupId>com.mycompany.app</groupId>
        <artifactId>my-app</artifactId>
        <version>1</version>
      </parent>

      <groupId>com.mycompany.app</groupId>
      <artifactId>my-module</artifactId>
      <version>1</version>
    </project>
    """;

listOfString.stream()
  .map(str -> !myTextBlock.equals(str));
```

"Stream.toList()" method should be used instead of "collectors" when unmodifiable list needed (java:S6204)

Severidad: MAJOR

Root_cause:

In Java 8 `Streams` were introduced to support chaining of operations over collections in a functional style. The most common way to save a result of such chains is to save them to some collection (usually `List`). To do so there is a terminal method `collect` that can be used with a library of `Collectors`. The key problem is that `.collect(Collectors.toList())` actually returns a mutable kind of `List` while in the majority of cases unmodifiable lists are preferred. In Java 10 a new collector appeared to return an unmodifiable list: `toUnmodifiableList()`. This does the trick but results in verbose code. Since Java 16 there is now a better variant to produce an unmodifiable list directly from a stream: `Stream.toList()`.

This rule raises an issue when "collect" is used to create a list from a stream.

Noncompliant code example

```
List<String> list1 = Stream.of("A", "B", "C")
                           .collect(Collectors.toList()); // Noncompliant

List<String> list2 = Stream.of("A", "B", "C")
                           .collect(Collectors.toUnmodifiableList()); // Noncompliant
```

Compliant solution

```
List<String> list1 = Stream.of("A", "B", "C").toList(); // Compliant

List<String> list2 = Stream.of("A", "B", "C")
                           .collect(Collectors.toList()); // Compliant, the list2 needs to be mutable

list2.add("X");
```

Switch arrow labels should not use redundant keywords (java:S6205)

Severidad: MINOR

Root_cause:

In Switch Expressions, an arrow label consisting of a block with a single `yield` can be simplified to directly return the value, resulting in cleaner code.

Similarly, for Switch Statements and arrow labels, a `break` in a block is always redundant and should not be used. Furthermore, if the resulting block contains only one statement, the curly braces of that block can also be omitted.

This rule reports an issue when a case of a Switch Expression contains a block with a single `yield` or when a Switch Statement contains a block with a `break`.

Noncompliant code example

```
int i = switch (mode) {
  case "a" -> {        // Noncompliant: Remove the redundant block and yield.
    yield 1;
  }
  default -> {         // Noncompliant: Remove the redundant block and yield.
    yield 2;
  }
};

switch (mode) {
  case "a" -> {        // Noncompliant: Remove the redundant block and break.
    result = 1;
    break;
  }
  default -> {         // Noncompliant: Remove the redundant break.
    doSomethingElse();
    result = 2;
    break;
  }
}
```

Compliant solution

```
int i = switch (mode) {
  case "a" -> 1;
  default -> 2;
};

switch (mode) {
  case "a" -> result = 1;
  default -> {
   doSomethingElse();
   result = 2;
 }
}
```

Resources:

- JEP 361: Switch Expressions

Records should be used instead of ordinary classes when representing immutable data structure (java:S6206)

Severidad: MAJOR

Resources:

- Records specification

Root_cause:

In Java 16 `records` are finalized and can be safely used in production code. `Records` represent immutable read-only data structure and should be used instead of creating immutable classes. Immutability of records is guaranteed by the Java language itself, while implementing immutable classes on your own might lead to some bugs.

One of the important aspects of `records` is that final fields can’t be overwritten using reflection.

This rule reports an issue on classes for which all these statements are true:

- all instance fields are private and final
- has only one constructor with a parameter for all fields
- has getters for all fields

Noncompliant code example

```
final class Person { // Noncompliant
  private final String name;
  private final int age;

  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }

  public String getName() {...}

  public int getAge() {...}

  @Override
  public boolean equals(Object o) {...}

  @Override
  public int hashCode() {...}

  @Override
  public String toString() {...}
}
```

Compliant solution

```
record Person(String name, int age) { }
```

Redundant constructors/methods should be avoided in records (java:S6207)

Severidad: MAJOR

Root_cause:

In Java 16 records represent a brief notation for immutable data structures. Records have autogenerated implementations for constructors with all parameters, `getters`, `equals`, `hashcode` and `toString`. Although these methods can still be overridden inside records, there is no use to do so if no special logic is required.

This rule reports an issue on empty compact constructors, trivial canonical constructors and simple getter methods with no additional logic.

Noncompliant code example

```
record Person(String name, int age) {
  Person(String name, int age) { // Noncompliant, already autogenerated
    this.name = name;
    this.age = age;
  }
}

record Person(String name, int age) {
  Person { // Noncompliant, no need for empty compact constructor
  }
  public String name() { // Noncompliant, already autogenerated
    return name;
  }
}
```

Compliant solution

```
record Person(String name, int age) { } // Compliant

record Person(String name, int age) {
  Person(String name, int age) { // Compliant
    this.name = name.toLowerCase(Locale.ROOT);
    this.age = age;
  }
}

record Person(String name, int age) {
  Person { // Compliant
    if (age < 0) {
      throw new IllegalArgumentException("Negative age");
    }
  }
  public String name() { // Compliant
    return name.toUpperCase(Locale.ROOT);
  }
}
```

Resources:

- Records specification

Comma-separated labels should be used in Switch with colon case (java:S6208)

Severidad: INFO

Root_cause:

In Java 14 there is a new way to write cases in Switch Statement and Expression when the same action should be performed for different cases. Instead of declaring multiples branches with the same action, you can combine all of them in a single case group, separated with commas. It will result in a more concise code and improved readability.

This rule reports an issue when multiple cases in a Switch can be grouped into a single comma-separated case.

Noncompliant code example

```
// Switch Expression
int i = switch (mode) {
  case "a":
  case "b":
    yield 1;
  default:
    yield 3;
};

// Switch Statement
switch (mode) {
  case "a":
  case "b":
    doSomething();
    break;
  default:
    doSomethingElse();
}
```

Compliant solution

```
// Switch Expression
int i = switch (mode) {
  case "a", "b":
    yield 1;
  default:
    yield 3;
};

// Switch Statement
switch (mode) {
  case "a", "b":
    doSomething();
    break;
  default:
    doSomethingElse();
}

// Or even better:
switch (mode) {
  case "a", "b" -> doSomething();
  default -> doSomethingElse();
}
```

Resources:

- JEP 361: Switch Expressions

Members ignored during record serialization should not be used (java:S6209)

Severidad: CRITICAL

Resources:

- Records specification
- serialization of records

Root_cause:

In Records, serialization is not done the same way as for ordinary serializable or externalizable classes. The serialized representation of a record object will be a sequence of values (record components). During the deserialization of records, the stream of components is read and components are constructed. Then the record object is recreated by invoking the record’s canonical constructor with the component values serving as arguments (or default values for absent arguments).

This process cannot be customized, so any class-specific `writeObject`, `readObject`, `readObjectNoData`, `writeExternal`, and `readExternal` methods or `serialPersistentFields` fields in record classes are ignored during serialization and deserialization.

However, there is a way to substitute serialized/deserialized objects in `writeReplace` and `readResolve`.

This rule raises an issue when any of `writeObject`, `readObject`, `readObjectNoData`, `writeExternal`, `readExternal` or `serialPersistentFields` are present as members in a Record class.

Noncompliant code example

```
record Record() implements Serializable {
  @Serial
  private static final ObjectStreamField[] serialPersistentFields = new ObjectStreamField[0]; // Noncompliant
  @Serial
  private void writeObject(ObjectOutputStream out) throws IOException { // Noncompliant
    ...
  }
}
record Record() implements Externalizable {
  @Override
  public void writeExternal(ObjectOutput out) throws IOException { // Noncompliant
    ...
  }
  @Override
  public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException { // Noncompliant
```

```
    ...
  }
}
```

Compliant solution

```
record Record() implements Serializable {}

record Record() implements Serializable {
  private Object writeReplace() throws ObjectStreamException {
    ...
  }
  private Object readResolve() throws ObjectStreamException {
    ...
  }
}
```

Regular expressions should not contain multiple spaces (java:S6326)

Severidad: MAJOR

Root_cause:

Multiple spaces in a regular expression can make it hard to tell how many spaces should be matched. It’s more readable to use only one space and then indicate with a quantifier how many spaces are expected.

Noncompliant code example

```
Pattern.compile("hello   world");
```

Compliant solution

```
Pattern.compile("hello {3}world");
```

Exceptions

In free-spacing mode (`Pattern.COMMENTS` flag, or with embedded flag expression `(?x)`), whitespaces are ignored. In this case no issue should be triggered, because the whitespaces may be intended to improve readability.

Regular expression quantifiers and character classes should be used concisely (java:S6353)

Severidad: MINOR

Root_cause:

A regular expression is a sequence of characters that specifies a match pattern in text. Among the most important concepts are:

- Character classes: defines a set of characters, any one of which can occur in an input string for a match to succeed.
- Quantifiers: used to specify how many instances of a character, group, or character class must be present in the input for a match.
- Wildcard (`.`): matches all characters except line terminators (also matches them if the `s` flag is set).

Many of these features include shortcuts of widely used expressions, so there is more than one way to construct a regular expression to achieve the same results. For example, to match a two-digit number, one could write `[0-9]{2,2}` or `\d{2}`. The latter is not only shorter but easier to read and thus to maintain.

This rule recommends replacing some quantifiers and character classes with more concise equivalents:

- `\d` for `[0-9]` and `\D` for `[^0-9]`
- `\w` for `[A-Za-z0-9_]` and `\W` for `[^A-Za-z0-9_]`
- `.` for character classes matching everything (e.g. `[\w\W]`, `[\d\D]`, or `[\s\S]` with `s` flag)
- `x?` for `x{0,1}`, `x*` for `x{0,}`, `x+` for `x{1,}`, `x{N}` for `x{N,N}`

```
"[0-9]"        // Noncompliant - same as "\\d"
"[^0-9]"       // Noncompliant - same as "\\D"
"[A-Za-z0-9_]" // Noncompliant - same as "\\w"
"[\\w\\W]"     // Noncompliant - same as "."
"a{0,}"        // Noncompliant - same as "a*"
```

Use the more concise version to make the regex expression more readable.

```
"\\d"
"\\D"
"\\w"
```

```
"."
"a*"
```

Deprecated annotations should include explanations (java:S6355)

Severidad: MAJOR

Root_cause:

Since Java 9, `@Deprecated` has two additional arguments to the annotation:

- `since` allows you to describe when the deprecation took place
- `forRemoval`, indicates whether the deprecated element will be removed at some future date

In order to ease the maintainers work, it is recommended to always add one or both of these arguments.

This rule reports an issue when `@Deprecated` is used without any argument.

Noncompliant code example

```
@Deprecated
```

Compliant solution

```
@Deprecated(since="4.2", forRemoval=true)
```

Exceptions

The members and methods of a deprecated class or interface are ignored by this rule. The classes and interfaces themselves are still subject to it.

Resources:

Related rules

- S1123

Hard-coded secrets are security-sensitive (java:S6418)

Severidad: BLOCKER

Default:

Because it is easy to extract strings from an application source code or binary, secrets should not be hard-coded. This is particularly true for applications that are distributed or that are open-source.

In the past, it has led to the following vulnerabilities:

- CVE-2022-25510
- CVE-2021-42635

Secrets should be stored outside of the source code in a configuration file or a management service for secrets.

This rule detects variables/fields having a name matching a list of words (secret, token, credential, auth, api[_.-]?key) being assigned a pseudorandom hard-coded value. The pseudorandomness of the hard-coded value is based on its entropy and the probability to be human-readable. The randomness sensibility can be adjusted if needed. Lower values will detect less random values, raising potentially more false positives.

Ask Yourself Whether

- The secret allows access to a sensitive component like a database, a file storage, an API, or a service.
- The secret is used in a production environment.
- Application re-distribution is required before updating the secret.

There would be a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- Store the secret in a configuration file that is not pushed to the code repository.
- Use your cloud provider's service for managing secrets.

- If a secret has been disclosed through the source code: revoke it and create a new one.

Sensitive Code Example

```
private static final String MY_SECRET = "47828a8dd77ee1eb9dde2d5e93cb221ce8c32b37";

public static void main(String[] args) {
  MyClass.callMyService(MY_SECRET);
}
```

Compliant Solution

Using AWS Secrets Manager:

```
import software.amazon.awssdk.services.secretsmanager.model.GetSecretValueRequest;
import software.amazon.awssdk.services.secretsmanager.model.GetSecretValueResponse;

public static void main(String[] args) {
  SecretsManagerClient secretsClient = ...
  MyClass.doSomething(secretsClient, "MY_SERVICE_SECRET");
}

public static void doSomething(SecretsManagerClient secretsClient, String secretName) {
  GetSecretValueRequest valueRequest = GetSecretValueRequest.builder()
    .secretId(secretName)
    .build();

  GetSecretValueResponse valueResponse = secretsClient.getSecretValue(valueRequest);
  String secret = valueResponse.secretString();
  // do something with the secret
  MyClass.callMyService(secret);
}
```

Using Azure Key Vault Secret:

```
import com.azure.identity.DefaultAzureCredentialBuilder;

import com.azure.security.keyvault.secrets.SecretClient;
import com.azure.security.keyvault.secrets.SecretClientBuilder;
import com.azure.security.keyvault.secrets.models.KeyVaultSecret;

public static void main(String[] args) throws InterruptedException, IllegalArgumentException {
  String keyVaultName = System.getenv("KEY_VAULT_NAME");
  String keyVaultUri = "https://" + keyVaultName + ".vault.azure.net";

  SecretClient secretClient = new SecretClientBuilder()
    .vaultUrl(keyVaultUri)
    .credential(new DefaultAzureCredentialBuilder().build())
    .buildClient();

  MyClass.doSomething(secretClient, "MY_SERVICE_SECRET");
}

public static void doSomething(SecretClient secretClient, String secretName) {
  KeyVaultSecret retrievedSecret = secretClient.getSecret(secretName);
  String secret = retrievedSecret.getValue(),

  // do something with the secret
  MyClass.callMyService(secret);
}
```

See

- OWASP - Top 10 2021 Category A7 - Identification and Authentication Failures
- OWASP - Top 10 2017 Category A2 - Broken Authentication
- CWE - CWE-798 - Use of Hard-coded Credentials
- MSC - MSC03-J - Never hard code sensitive information

Root_cause:

Because it is easy to extract strings from an application source code or binary, secrets should not be hard-coded. This is particularly true for applications that are distributed or that are open-source.

In the past, it has led to the following vulnerabilities:

- CVE-2022-25510
- CVE-2021-42635

Secrets should be stored outside of the source code in a configuration file or a management service for secrets.

This rule detects variables/fields having a name matching a list of words (secret, token, credential, auth, api[_.-]?key) being assigned a pseudorandom hard-coded value. The pseudorandomness of the hard-coded value is based on its entropy and the probability to be human-readable. The randomness sensibility can be adjusted if needed. Lower values will detect less random values, raising potentially more false positives.

Assess_the_problem:

Ask Yourself Whether

- The secret allows access to a sensitive component like a database, a file storage, an API, or a service.
- The secret is used in a production environment.
- Application re-distribution is required before updating the secret.

There would be a risk if you answered yes to any of those questions.

Sensitive Code Example

```
private static final String MY_SECRET = "47828a8dd77ee1eb9dde2d5e93cb221ce8c32b37";

public static void main(String[] args) {
  MyClass.callMyService(MY_SECRET);
}
```

How_to_fix:

Recommended Secure Coding Practices

- Store the secret in a configuration file that is not pushed to the code repository.
- Use your cloud provider’s service for managing secrets.
- If a secret has been disclosed through the source code: revoke it and create a new one.

Compliant Solution

Using AWS Secrets Manager:

```
import software.amazon.awssdk.services.secretsmanager.model.GetSecretValueRequest;
import software.amazon.awssdk.services.secretsmanager.model.GetSecretValueResponse;

public static void main(String[] args) {
  SecretsManagerClient secretsClient = ...
  MyClass.doSomething(secretsClient, "MY_SERVICE_SECRET");
}

public static void doSomething(SecretsManagerClient secretsClient, String secretName) {
  GetSecretValueRequest valueRequest = GetSecretValueRequest.builder()
    .secretId(secretName)
    .build();

  GetSecretValueResponse valueResponse = secretsClient.getSecretValue(valueRequest);
  String secret = valueResponse.secretString();
  // do something with the secret
  MyClass.callMyService(secret);
}
```

Using Azure Key Vault Secret:

```
import com.azure.identity.DefaultAzureCredentialBuilder;

import com.azure.security.keyvault.secrets.SecretClient;
import com.azure.security.keyvault.secrets.SecretClientBuilder;
import com.azure.security.keyvault.secrets.models.KeyVaultSecret;

public static void main(String[] args) throws InterruptedException, IllegalArgumentException {
  String keyVaultName = System.getenv("KEY_VAULT_NAME");
  String keyVaultUri = "https://" + keyVaultName + ".vault.azure.net";

  SecretClient secretClient = new SecretClientBuilder()
    .vaultUrl(keyVaultUri)
    .credential(new DefaultAzureCredentialBuilder().build())
    .buildClient();

  MyClass.doSomething(secretClient, "MY_SERVICE_SECRET");
}

public static void doSomething(SecretClient secretClient, String secretName) {
  KeyVaultSecret retrievedSecret = secretClient.getSecret(secretName);
  String secret = retrievedSecret.getValue(),

  // do something with the secret
  MyClass.callMyService(secret);
}
```

See

- OWASP - Top 10 2021 Category A7 - Identification and Authentication Failures
- OWASP - Top 10 2017 Category A2 - Broken Authentication
- CWE - CWE-798 - Use of Hard-coded Credentials
- MSC - MSC03-J - Never hard code sensitive information

Exact alarms should not be abused (java:S6891)

Severidad: MAJOR

How_to_fix:

Replace occurrences of `setExact` with `set` and `setExactAndAllowWhileIdle` with `setAndAllowWhileIdle`, and avoid to use `setWindow` with a window less than 10 minutes.

Alternatively, consider using `Handler`, `WorkManager` or `JobScheduler` instead of `AlarmManager` when possible, depending on your use case.

Noncompliant code example

```
public class AlarmScheduler {
    private Context context;

    public AlarmScheduler(Context context) {
        this.context = context;
    }

    public void scheduleAlarm(long triggerTime) {
        AlarmManager alarmManager = (AlarmManager) context.getSystemService(Context.ALARM_SERVICE);
        Intent intent = new Intent(context, AlarmReceiver.class);
        PendingIntent pendingIntent = PendingIntent.getBroadcast(context, 0, intent, 0);

        alarmManager.setExact(AlarmManager.RTC_WAKEUP, triggerTime, pendingIntent); // Noncompliant, avoid using exact alarms unless necess
        alarmManager.setExactAndAllowWhileIdle(AlarmManager.RTC_WAKEUP, triggerTime, pendingIntent); // Noncompliant, avoid using exact ala

        long windowLengthMillis = 5 * 60 * 1000; // 5 minutes in milliseconds
        alarmManager.setWindow(AlarmManager.RTC_WAKEUP, triggerTime, windowLengthMillis, pendingIntent); // Noncompliant, don't use windows
    }
}
```

Compliant solution

```
public class AlarmScheduler {
    private Context context;

    public AlarmScheduler(Context context) {
        this.context = context;
    }

    public void scheduleAlarm(long triggerTime) {
        AlarmManager alarmManager = (AlarmManager) context.getSystemService(Context.ALARM_SERVICE);
        Intent intent = new Intent(context, AlarmReceiver.class);
        PendingIntent pendingIntent = PendingIntent.getBroadcast(context, 0, intent, 0);

        alarmManager.set(AlarmManager.RTC_WAKEUP, triggerTime, pendingIntent); // Compliant
        alarmManager.setAndAllowWhileIdle(AlarmManager.RTC_WAKEUP, triggerTime, pendingIntent);  // Compliant

        long windowLengthMillis = 10 * 60 * 1000; // 10 minutes in milliseconds
        alarmManager.setWindow(AlarmManager.RTC_WAKEUP, triggerTime, windowLengthMillis, pendingIntent); // Compliant
    }
}
```

Resources:

Documentation

- Android for Developers - AlarmManager
- Android for Developers - AlarmManager with SDK Version 19
- Android for Developers - Schedule alarms
- Android for Developers - Handler
- Android for Developers - WorkManager
- Android for Developers - JobScheduler

Root_cause:

The use of exact alarms triggers the device to wake up at precise times that can lead several wake-ups in a short period of time. The wake-up mechanism is a significant battery drain because it requires powering up the main processor and pulling it out of a low-power state.

It’s highly recommended to create an inexact alarm whenever possible.

It is also recommended for normal timing operations, such as ticks and timeouts, using the `Handler`, and for long-running operations, such as network downloads, using `WorkManager` or `JobScheduler`.

What is the potential impact?

- *Usability*: the device may run out of battery faster than expected.
- *Sustainability*: the extra battery usage has a negative impact on the environment.

Introduction:

The `AlarmManager` class provides access to the system alarm services. It allows you to schedule your application to run at some point in the future, even when it’s not active.

From API 19 onwards, the alarm delivery is inexact in order to save battery life. The Android OS now batches together alarms from all apps that occur at reasonably similar times so the system wakes the device once instead of several times to handle each alarm.

It is possible to use exact alarms with `setExact`, `setExactAndAllowWhileIdle`, `setWindow` and `setAlarmClock`. Exact alarms should be used only when strict delivery guarantees are required, for example for an alarm clock application or for calendar notifications.

The rule raises an issue when an exact alarm is set, or when a window is set to less than 10 minutes.

Fields should not be initialized to default values (java:S3052)

Severidad: MINOR

Root_cause:

The compiler automatically initializes class fields to their default values before setting them with any initialization values, so there is no need to explicitly set a field to its default value. Further, under the logic that cleaner code is better code, it’s considered poor style to do so.

Noncompliant code example

```
public class MyClass {

  int count = 0;  // Noncompliant
  // ...

}
```

Compliant solution

```
public class MyClass {

  int count;
  // ...

}
```

Exceptions

`final` fields are ignored.

Map values should not be replaced unconditionally (java:S4143)

Severidad: MAJOR

Root_cause:

Storing a value inside a collection at a given key or index and then unconditionally overwriting it without reading the initial value is a case of a "dead store".

```
letters.put("a", "Apple");
letters.put("a", "Boy");  // Noncompliant

towns[i] = "London";
towns[i] = "Chicago";  // Noncompliant
```

This practice is redundant and will cause confusion for the reader. More importantly, it is often an error and not what the developer intended to do.

"Bean Validation" (JSR 380) should be properly configured (java:S5128)

Severidad: CRITICAL

Resources:

- Bean Validation 2.0 (JSR 380)

Root_cause:

`Bean Validation` as per defined by JSR 380 can be triggered programmatically or also executed by the `Bean Validation` providers. However something should tell the `Bean Validation` provider that a variable must be validated otherwise no validation will happen. This can be achieved by annotating a variable with `javax.validation.Valid` and unfortunally it’s easy to forget to add this annotation on complex Beans.

Not annotating a variable with `@Valid` means `Bean Validation` will not be triggered for this variable, but readers may overlook this omission and assume the variable will be validated.

This rule will run by default on all `Class`'es and therefore can generate a lot of noise. This rule should be restricted to run only on certain layers. For this reason, the "Restrict Scope of Coding Rules" feature should be used to check for missing `@Valid` annotations only on some packages of the application.

Noncompliant code example

```
import javax.validation.Valid;
import javax.validation.constraints.NotNull;

public class User {
  @NotNull
  private String name;
}

public class Group {
  @NotNull
  private List<User> users; // Noncompliant; User instances are not validated
}

public class MyService {
  public void login(User user) { // Noncompliant; parameter "user" is not validated
  }
}
```

Compliant solution

```
import javax.validation.Valid;
import javax.validation.constraints.NotNull;

public class User {
  @NotNull
  private String name;
}

public class Group {
  @Valid
  @NotNull
  private List<User> users; // Compliant; User instances are validated

  @NotNull
  // preferred style as of Bean Validation 2.0
  private List<@Valid User> users2; // Compliant; User instances are validated
}

public class MyService {
  public void login(@Valid User user) { // Compliant
  }
}
```

Reflection should not be used to increase accessibility of records' fields (java:S6216)

Severidad: MAJOR

Resources:

- Records specification

Root_cause:

In general, altering or bypassing the accessibility of classes, methods, or fields violates the encapsulation principle and could lead to runtime errors. For records the case is even trickier: you cannot change the visibility of records's fields and trying to update the existing value will lead to `IllegalAccessException` at runtime.

This rule raises an issue when reflection is used to change the visibility of a record's field, and when it is used to directly update a record's field value.

Noncompliant code example

```
record Person(String name, int age) {}

Person person = new Person("A", 26);
Field field = Person.class.getDeclaredField("name");
field.setAccessible(true); // secondary
field.set(person, "B"); // Noncompliant
```

Permitted types of a sealed class should be omitted if they are declared in the same file (java:S6217)

Severidad: MINOR

Resources:

- Sealed Classes specification

Root_cause:

`sealed` classes were introduced in Java 17. This feature is very useful if there is a need to define a strict hierarchy and restrict the possibility of extending classes. In order to mention all the allowed subclasses, there is a keyword `permits`, which should be followed by subclasses' names.

This notation is quite useful if subclasses of a given `sealed` class can be found in different files, packages, or even modules. In case when all subclasses are declared in the same file there is no need to mention the explicitly and `permits` part of a declaration can be omitted.

This rule reports an issue if all subclasses of a `sealed` class are declared in the same file as their superclass.

Noncompliant code example

```
sealed class A permits B, C, D, E {} // Noncompliant
final class B extends A {}
final class C extends A {}
final class D extends A {}
final class E extends A {}
```

Compliant solution

```
sealed class A {} // Compliant
final class B extends A {}
final class C extends A {}
final class D extends A {}
final class E extends A {}
```

'serialVersionUID' field should not be set to '0L' in records (java:S6219)

Severidad: MINOR

Resources:

- Records specification
- Serialization of records

Root_cause:

In Records serialization is not done the same way as for ordinary serializable or externalizable classes. Records serialization does not rely on the `serialVersionUID` field, because the requirement to have this field equal is waived for record classes. By default, all records will have this field equal to `0L` and there is no need to specify this field with `0L` value and it is possible to specify it with some custom value to support serialization/deserialization involving ordinary classes.

This rule raises an issue when there is a `private static final long serialVersionUID` field which is set to `0L` in a Record class.

Noncompliant code example

```
record Person(String name, int age) implements Serializable {
@Serial
  private static final long serialVersionUID = 0L; // Noncompliant
}
```

Compliant solution

```
record Person(String name, int age) implements Serializable {} // Compliant

record Person(String name, int age) implements Serializable {
@Serial
  private static final long serialVersionUID = 42L; // Compliant
}
```

Equality operators should not be used in "for" loop termination conditions (java:S888)

Severidad: CRITICAL

Resources:

- CWE - CWE-835 - Loop with Unreachable Exit Condition ('Infinite Loop')
- CERT, MSC21-C. - Use robust loop termination conditions

Root_cause:

Testing `for` loop termination using an equality operator (`==` and `!=`) is dangerous, because it could set up an infinite loop. Using a broader relational operator instead casts a wider net, and makes it harder (but not impossible) to accidentally write an infinite loop.

Noncompliant code example

```
for (int i = 1; i != 10; i += 2)  // Noncompliant. Infinite; i goes from 9 straight to 11.
{
  //...
}
```

Compliant solution

```
for (int i = 1; i <= 10; i += 2)  // Compliant
{
  //...
}
```

Exceptions

Equality operators are ignored if the loop counter is not modified within the body of the loop and either:

- starts below the ending value and is incremented by 1 on each iteration.
- starts above the ending value and is decremented by 1 on each iteration.

Equality operators are also ignored when the test is against `null`.

```
for (int i = 0; arr[i] != null; i++) {
  // ...
}

for (int i = 0; (item = arr[i]) != null; i++) {
  // ...
}
```

"main" should not "throw" anything (java:S2096)

Severidad: BLOCKER

Root_cause:

There’s no reason for a `main` method to `throw` anything. After all, what’s going to catch it?

Instead, the method should itself gracefully handle any exceptions that may bubble up to it, attach as much contextual information as possible, and perform whatever logging or user communication is necessary, and `exit` with a non-zero (i.e. non-success) exit code if necessary.

Noncompliant code example

```
public static void main(String args[]) throws Exception { // Noncompliant
  doSomething();
}
```

Compliant solution

```
public static void main(String args[]) {
 try {
    doSomething();
  } catch (Throwable t) {
    log.error(t);
    System.exit(1);  // Default exit code, 0, indicates success. Non-zero value means failure.
  }
}
```

Assignment of lazy-initialized members should be the last step with double-checked locking (java:S3064)

Severidad: MAJOR

Resources:

- CERT, LCK10-J. - Use a correct form of the double-checked locking idiom

Related rules

- S2168 - Double-checked locking should not be used

Root_cause:

Double-checked locking can be used for lazy initialization of `volatile` fields, but only if field assignment is the last step in the `synchronized` block. Otherwise you run the risk of threads accessing a half-initialized object.

Noncompliant code example

```
public class MyClass {

  private volatile List<String> strings;

  public List<String> getStrings() {
    if (strings == null) {  // check#1
      synchronized(MyClass.class) {
        if (strings == null) {
          strings = new ArrayList<>();  // Noncompliant
          strings.add("Hello");  //When threadA gets here, threadB can skip the synchronized block because check#1 is false
          strings.add("World");
        }
      }
    }
    return strings;
  }
}
```

Compliant solution

```
public class MyClass {

  private volatile List<String> strings;

  public List<String> getStrings() {
    if (strings == null) {  // check#1
      synchronized(MyClass.class) {
        if (strings == null) {
          List<String> tmpList = new ArrayList<>();
          tmpList.add("Hello");
          tmpList.add("World");
          strings = tmpList;
        }
      }
    }
    return strings;
  }
}
```

"enum" fields should not be publicly mutable (java:S3066)

Severidad: MINOR

Root_cause:

`enum`s are generally thought of as constant, but an `enum` with a `public` field or `public` setter is non-constant. Ideally fields in an `enum` are `private` and set in the constructor, but if that’s not possible, their visibility should be reduced as much as possible.

Noncompliant code example

```
public enum Continent {

  NORTH_AMERICA (23, 24709000),
  // ...
  EUROPE (50, 39310000);

  public int countryCount;  // Noncompliant
  private int landMass;

  Continent(int countryCount, int landMass) {
    // ...
  }

  public void setLandMass(int landMass) {  // Noncompliant
    this.landMass = landMass;
  }
```

Compliant solution

```
public enum Continent {

  NORTH_AMERICA (23, 24709000),
  // ...
  EUROPE (50, 39310000);

  private int countryCount;
  private int landMass;

  Continent(int countryCount, int landMass) {
    // ...
  }
```

"getClass" should not be used for synchronization (java:S3067)

Severidad: MAJOR

Root_cause:

`getClass` should not be used for synchronization in non-`final` classes because child classes will synchronize on a different object than the parent or each other, allowing multiple threads into the code block at once, despite the `synchronized` keyword.

Instead, hard code the name of the class on which to synchronize or make the class `final`.

Noncompliant code example

```
public class MyClass {
  public void doSomethingSynchronized(){
    synchronized (this.getClass()) {  // Noncompliant
      // ...
    }
  }
```

Compliant solution

```
public class MyClass {
  public void doSomethingSynchronized(){
    synchronized (MyClass.class) {
      // ...
    }
  }
```

Resources:

- CERT, LCK02-J. - Do not synchronize on the class object returned by getClass()

Packages containing only "package-info.java" should be removed (java:S4032)

Severidad: MINOR

Root_cause:

There is no reason to have a package that is empty except for "package-info.java". Such packages merely clutter a project, taking up space but adding no value.

"Stream" call chains should be simplified when possible (java:S4034)

Severidad: MINOR

Root_cause:

When using the `Stream` API, call chains should be simplified as much as possible. Not only does it make the code easier to read, it also avoid creating unnecessary temporary objects.

This rule raises an issue when one of the following substitution is possible:

Original	Preferred
`stream.filter(predicate).findFirst().isPresent()`	`stream.anyMatch(predicate)`
`stream.filter(predicate).findAny().isPresent()`	`stream.anyMatch(predicate)`
`!stream.anyMatch(predicate)`	`stream.noneMatch(predicate)`
`!stream.anyMatch(x -> !(...))`	`stream.allMatch(...)`
`stream.map(mapper).anyMatch(Boolean::booleanValue)`	`stream.anyMatch(predicate)`

Noncompliant code example

```
boolean hasRed = widgets.stream().filter(w -> w.getColor() == RED).findFirst().isPresent(); // Noncompliant
```

Compliant solution

```
boolean hasRed = widgets.stream().anyMatch(w -> w.getColor() == RED);
```

Searching OS commands in PATH is security-sensitive (java:S4036)

Severidad: MINOR

Root_cause:

When executing an OS command and unless you specify the full path to the executable, then the locations in your application’s `PATH` environment variable will be searched for the executable. That search could leave an opening for an attacker if one of the elements in `PATH` is a directory under his control.

How_to_fix:

Recommended Secure Coding Practices

Fully qualified/absolute path should be used to specify the OS command to execute.

Compliant Solution

The command is defined by its full path:

```
Runtime.getRuntime().exec("/usr/bin/make");  // Compliant
Runtime.getRuntime().exec(new String[]{"~/bin/make"});  // Compliant

ProcessBuilder builder = new ProcessBuilder("./bin/make");  // Compliant
builder.command("../bin/make");  // Compliant
builder.command(Arrays.asList("..\bin\make", "-j8")); // Compliant

builder = new ProcessBuilder(Arrays.asList(".\make"));  // Compliant
builder.command(Arrays.asList("C:\bin\make", "-j8"));  // Compliant
builder.command(Arrays.asList("\\SERVER\bin\make"));  // Compliant
```

See

- OWASP - Top 10 2021 Category A8 - Software and Data Integrity Failures
- OWASP - Top 10 2017 Category A1 - Injection
- CWE - CWE-426 - Untrusted Search Path

- CWE - CWE-427 - Uncontrolled Search Path Element

Default:

When executing an OS command and unless you specify the full path to the executable, then the locations in your application’s `PATH` environment variable will be searched for the executable. That search could leave an opening for an attacker if one of the elements in `PATH` is a directory under his control.

Ask Yourself Whether

- The directories in the PATH environment variable may be defined by not trusted entities.

There is a risk if you answered yes to this question.

Recommended Secure Coding Practices

Fully qualified/absolute path should be used to specify the OS command to execute.

Sensitive Code Example

The full path of the command is not specified and thus the executable will be searched in all directories listed in the `PATH` environment variable:

```
Runtime.getRuntime().exec("make");  // Sensitive
Runtime.getRuntime().exec(new String[]{"make"});  // Sensitive

ProcessBuilder builder = new ProcessBuilder("make");  // Sensitive
builder.command("make");  // Sensitive
```

Compliant Solution

The command is defined by its full path:

```
Runtime.getRuntime().exec("/usr/bin/make");  // Compliant
Runtime.getRuntime().exec(new String[]{"~/bin/make"});  // Compliant

ProcessBuilder builder = new ProcessBuilder("./bin/make");  // Compliant
builder.command("../bin/make");  // Compliant
builder.command(Arrays.asList("..\bin\make", "-j8")); // Compliant

builder = new ProcessBuilder(Arrays.asList(".\make"));  // Compliant
builder.command(Arrays.asList("C:\bin\make", "-j8"));  // Compliant
builder.command(Arrays.asList("\\SERVER\bin\make"));  // Compliant
```

See

- OWASP - Top 10 2021 Category A8 - Software and Data Integrity Failures
- OWASP - Top 10 2017 Category A1 - Injection
- CWE - CWE-426 - Untrusted Search Path
- CWE - CWE-427 - Uncontrolled Search Path Element

Assess_the_problem:

Ask Yourself Whether

- The directories in the PATH environment variable may be defined by not trusted entities.

There is a risk if you answered yes to this question.

Sensitive Code Example

The full path of the command is not specified and thus the executable will be searched in all directories listed in the `PATH` environment variable:

```
Runtime.getRuntime().exec("make");  // Sensitive
Runtime.getRuntime().exec(new String[]{"make"});  // Sensitive

ProcessBuilder builder = new ProcessBuilder("make");  // Sensitive
builder.command("make");  // Sensitive
```

Functional Interfaces should be as specialised as possible (java:S4276)

Severidad: MINOR

Root_cause:

The `java.util.function` package provides a large array of functional interface definitions for use in lambda expressions and method references. In general it is recommended to use the more specialised form to avoid auto-boxing. For instance `IntFunction<Foo>` should be preferred over `Function<Integer, Foo>`.

This rule raises an issue when any of the following substitution is possible:

Current Interface	Preferred Interface
`Function<Integer, R>`	`IntFunction<R>`
`Function<Long, R>`	`LongFunction<R>`
`Function<Double, R>`	`DoubleFunction<R>`
`Function<Double,Integer>`	`DoubleToIntFunction`
`Function<Double,Long>`	`DoubleToLongFunction`
`Function<Long,Double>`	`LongToDoubleFunction`
`Function<Long,Integer>`	`LongToIntFunction`
`Function<R,Integer>`	`ToIntFunction<R>`
`Function<R,Long>`	`ToLongFunction<R>`
`Function<R,Double>`	`ToDoubleFunction<R>`
`Function<T,T>`	`UnaryOperator<T>`
`BiFunction<T,T,T>`	`BinaryOperator<T>`
`Consumer<Integer>`	`IntConsumer`
`Consumer<Double>`	`DoubleConsumer`
`Consumer<Long>`	`LongConsumer`
`BiConsumer<T,Integer>`	`ObjIntConsumer<T>`
`BiConsumer<T,Long>`	`ObjLongConsumer<T>`
`BiConsumer<T,Double>`	`ObjDoubleConsumer<T>`
`Predicate<Integer>`	`IntPredicate`
`Predicate<Double>`	`DoublePredicate`
`Predicate<Long>`	`LongPredicate`
`Supplier<Integer>`	`IntSupplier`
`Supplier<Double>`	`DoubleSupplier`
`Supplier<Long>`	`LongSupplier`
`Supplier<Boolean>`	`BooleanSupplier`
`UnaryOperator<Integer>`	`IntUnaryOperator`
`UnaryOperator<Double>`	`DoubleUnaryOperator`
`UnaryOperator<Long>`	`LongUnaryOperator`
`BinaryOperator<Integer>`	`IntBinaryOperator`
`BinaryOperator<Long>`	`LongBinaryOperator`
`BinaryOperator<Double>`	`DoubleBinaryOperator`
`Function<T, Boolean>`	`Predicate<T>`
`BiFunction<T,U,Boolean>`	`BiPredicate<T,U>`

Noncompliant code example

```
public class Foo implements Supplier<Integer> {  // Noncompliant
    @Override
    public Integer get() {
      // ...
    }
}
```

Compliant solution

```
public class Foo implements IntSupplier {

  @Override
  public int getAsInt() {
    // ...
  }
}
```

Disabling auto-escaping in template engines is security-sensitive (java:S5247)

Severidad: MAJOR

Assess_the_problem:

Ask Yourself Whether

- Templates are used to render web content and
    - dynamic variables in templates come from untrusted locations or are user-controlled inputs
    - there is no local mechanism in place to sanitize or validate the inputs.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

With JMustache by samskivert:

```
Mustache.compiler().escapeHTML(false).compile(template).execute(context); // Sensitive
Mustache.compiler().withEscaper(Escapers.NONE).compile(template).execute(context); // Sensitive
```

With Freemarker:

```
freemarker.template.Configuration configuration = new freemarker.template.Configuration();
configuration.setAutoEscapingPolicy(DISABLE_AUTO_ESCAPING_POLICY); // Sensitive
```

Default:

To reduce the risk of cross-site scripting attacks, templating systems, such as `Twig`, `Django`, `Smarty`, `Groovy's template engine`, allow configuration of automatic variable escaping before rendering templates. When escape occurs, characters that make sense to the browser (eg: <a>) will be transformed/replaced with escaped/sanitized values (eg: & lt;a& gt;).

Auto-escaping is not a magic feature to annihilate all cross-site scripting attacks, it depends on the strategy applied and the context, for example a "html auto-escaping" strategy (which only transforms html characters into html entities) will not be relevant when variables are used in a html attribute because '`:`' character is not escaped and thus an attack as below is possible:

```
<a href="{{ myLink }}">link</a> // myLink = javascript:alert(document.cookie)
<a href="javascript:alert(document.cookie)">link</a> // JS injection (XSS attack)
```

Ask Yourself Whether

- Templates are used to render web content and
    - dynamic variables in templates come from untrusted locations or are user-controlled inputs
    - there is no local mechanism in place to sanitize or validate the inputs.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

Enable auto-escaping by default and continue to review the use of inputs in order to be sure that the chosen auto-escaping strategy is the right one.

Sensitive Code Example

With JMustache by samskivert:

```
Mustache.compiler().escapeHTML(false).compile(template).execute(context); // Sensitive
Mustache.compiler().withEscaper(Escapers.NONE).compile(template).execute(context); // Sensitive
```

With Freemarker:

```
freemarker.template.Configuration configuration = new freemarker.template.Configuration();
configuration.setAutoEscapingPolicy(DISABLE_AUTO_ESCAPING_POLICY); // Sensitive
```

Compliant Solution

With JMustache by samskivert:

```
Mustache.compiler().compile(template).execute(context); // Compliant, auto-escaping is enabled by default
Mustache.compiler().escapeHTML(true).compile(template).execute(context); // Compliant
```

With Freemarker. See "setAutoEscapingPolicy" documentation for more details.

```
freemarker.template.Configuration configuration = new freemarker.template.Configuration();
configuration.setAutoEscapingPolicy(ENABLE_IF_DEFAULT_AUTO_ESCAPING_POLICY); // Compliant
```

See

- OWASP - Top 10 2021 Category A3 - Injection
- OWASP Cheat Sheet - XSS Prevention Cheat Sheet
- OWASP - Top 10 2017 Category A7 - Cross-Site Scripting (XSS)
- CWE - CWE-79 - Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

Root_cause:

To reduce the risk of cross-site scripting attacks, templating systems, such as `Twig`, `Django`, `Smarty`, `Groovy's template engine`, allow configuration of automatic variable escaping before rendering templates. When escape occurs, characters that make sense to the browser (eg: <a>) will be transformed/replaced with escaped/sanitized values (eg: & lt;a& gt;).

Auto-escaping is not a magic feature to annihilate all cross-site scripting attacks, it depends on the strategy applied and the context, for example a "html auto-escaping" strategy (which only transforms html characters into html entities) will not be relevant when variables are used in a html attribute because '`:`' character is not escaped and thus an attack as below is possible:

```
<a href="{{ myLink }}">link</a> // myLink = javascript:alert(document.cookie)
<a href="javascript:alert(document.cookie)">link</a> // JS injection (XSS attack)
```

How_to_fix:

Recommended Secure Coding Practices

Enable auto-escaping by default and continue to review the use of inputs in order to be sure that the chosen auto-escaping strategy is the right one.

Compliant Solution

With JMustache by samskivert:

```
Mustache.compiler().compile(template).execute(context); // Compliant, auto-escaping is enabled by default
Mustache.compiler().escapeHTML(true).compile(template).execute(context); // Compliant
```

With Freemarker. See "setAutoEscapingPolicy" documentation for more details.

```
freemarker.template.Configuration configuration = new freemarker.template.Configuration();
configuration.setAutoEscapingPolicy(ENABLE_IF_DEFAULT_AUTO_ESCAPING_POLICY); // Compliant
```

See

- OWASP - Top 10 2021 Category A3 - Injection
- OWASP Cheat Sheet - XSS Prevention Cheat Sheet
- OWASP - Top 10 2017 Category A7 - Cross-Site Scripting (XSS)
- CWE - CWE-79 - Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

Local-Variable Type Inference should be used (java:S6212)

Severidad: INFO

Resources:

- JEP 286: Local-Variable Type Inference

Root_cause:

In Java 10 Local-Variable Type Inference was introduced. It allows you to omit the expected type of a variable by declaring it with the `var` keyword.

While it is not always possible or cleaner to use this new way of declaring a variable, when the type on the left is the same as the one on the right in an assignment, using the `var` will result in a more concise code.

This rule reports an issue when the expected type of the variable is the same as the returned type of assigned expression and the type can be easily inferred by the reader, either when the type is already mentioned in the name or the initializer, or when the expression is self-explanatory.

Noncompliant code example

```
MyClass myClass = new MyClass();

int i = 10; // Type is self-explanatory

MyClass something = MyClass.getMyClass(); // Type is already mentioned in the initializer

MyClass myClass = get(); // Type is already mentioned in the name
```

Compliant solution

```
var myClass = new MyClass();

var i = 10;

var something = MyClass.getMyClass();

var myClass = get();
```

Restricted Identifiers should not be used as Identifiers (java:S6213)

Severidad: MAJOR

Resources:

- JLS16, 3.8: Identifiers

Root_cause:

Even if it is technically possible, Restricted Identifiers should not be used as identifiers. This is only possible for compatibility reasons, using it in Java code is confusing and should be avoided.

Note that this applies to any version of Java, including the one where these identifiers are not yet restricted, to avoid future confusion.

This rule reports an issue when restricted identifiers:

- var
- yield
- record

are used as identifiers.

Noncompliant code example

```
var var = "var"; // Noncompliant: compiles but this code is confusing
var = "what is this?";

int yield(int i) { // Noncompliant
  return switch (i) {
    case 1: yield(0); // This is a yield from switch expression, not a recursive call.
    default: yield(i-1);
  };
}

String record = "record"; // Noncompliant
```

Compliant solution

```
var myVariable = "var";

int minusOne(int i) {
  return switch (i) {
    case 1: yield(0);
    default: yield(i-1);
  };
}

String myRecord = "record";
```

Equals method should be overridden in records containing array fields (java:S6218)

Severidad: MAJOR

Resources:

- Records specification

Root_cause:

In records, the default behavior of the `equals()` method is to check the equality by field values. This works well for primitive fields or fields, whose type overrides `equals()`, but this behavior doesn't work as expected for array fields.

By default, array fields are compared by their reference, and overriding `equals()` is highly appreciated to achieve the deep equality check. The same strategy applies to `hashCode()` and `toString()` methods.

This rule reports an issue if a record class has an array field and is not overriding `equals()`, `hashCode()` or `toString()` methods.

Noncompliant code example

```
record Person(String[] names, int age) {} // Noncompliant
```

Compliant solution

```
record Person(String[] names, int age) {
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Person person = (Person) o;
        return age == person.age && Arrays.equals(names, person.names);
    }

    @Override
    public int hashCode() {
        int result = Objects.hash(age);
        result = 31 * result + Arrays.hashCode(names);
        return result;
    }

    @Override
    public String toString() {
        return "Person{" +
                "names=" + Arrays.toString(names) +
                ", age=" + age +
                '}';
    }
}
```

Creating cookies without the "secure" flag is security-sensitive (java:S2092)

Severidad: MINOR

Default:

When a cookie is protected with the `secure` attribute set to *true* it will not be send by the browser over an unencrypted HTTP request and thus cannot be observed by an unauthorized person during a man-in-the-middle attack.

Ask Yourself Whether

- the cookie is for instance a *session-cookie* not designed to be sent over non-HTTPS communication.
- it's not sure that the website contains mixed content or not (ie HTTPS everywhere or not)

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- It is recommended to use `HTTPs` everywhere so setting the `secure` flag to *true* should be the default behaviour when creating cookies.
- Set the `secure` flag to *true* for session-cookies.

Sensitive Code Example

If you create a security-sensitive cookie in your JAVA code:

```
Cookie c = new Cookie(COOKIENAME, sensitivedata);
c.setSecure(false);  // Sensitive: a security-ensitive cookie is created with the secure flag set to false
```

By default the `secure` flag is set to *false:*

```
Cookie c = new Cookie(COOKIENAME, sensitivedata);  // Sensitive: a security-sensitive cookie is created with the secure flag not defined (b
```

Compliant Solution

```
Cookie c = new Cookie(COOKIENAME, sensitivedata);
c.setSecure(true); // Compliant: the sensitive cookie will not be send during an unencrypted HTTP request thanks to the secure flag set to
```

See

- OWASP - Top 10 2021 Category A4 - Insecure Design
- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- CWE - CWE-311 - Missing Encryption of Sensitive Data
- CWE - CWE-315 - Cleartext Storage of Sensitive Information in a Cookie
- CWE - CWE-614 - Sensitive Cookie in HTTPS Session Without 'Secure' Attribute
- STIG Viewer - Application Security and Development: V-222576 - The application must set the secure flag on session cookies.

How_to_fix:

Recommended Secure Coding Practices

- It is recommended to use `HTTPs` everywhere so setting the `secure` flag to *true* should be the default behaviour when creating cookies.
- Set the `secure` flag to *true* for session-cookies.

Compliant Solution

```
Cookie c = new Cookie(COOKIENAME, sensitivedata);
c.setSecure(true); // Compliant: the sensitive cookie will not be send during an unencrypted HTTP request thanks to the secure flag set to
```

See

- OWASP - Top 10 2021 Category A4 - Insecure Design
- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
- CWE - CWE-311 - Missing Encryption of Sensitive Data
- CWE - CWE-315 - Cleartext Storage of Sensitive Information in a Cookie
- CWE - CWE-614 - Sensitive Cookie in HTTPS Session Without 'Secure' Attribute
- STIG Viewer - Application Security and Development: V-222576 - The application must set the secure flag on session cookies.

Root_cause:

When a cookie is protected with the `secure` attribute set to *true* it will not be send by the browser over an unencrypted HTTP request and thus cannot be observed by an unauthorized person during a man-in-the-middle attack.

Assess_the_problem:

Ask Yourself Whether

- the cookie is for instance a *session-cookie* not designed to be sent over non-HTTPS communication.
- it’s not sure that the website contains mixed content or not (ie HTTPS everywhere or not)

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

If you create a security-sensitive cookie in your JAVA code:

```
Cookie c = new Cookie(COOKIENAME, sensitivedata);
c.setSecure(false);  // Sensitive: a security-ensitive cookie is created with the secure flag set to false
```

By default the `secure` flag is set to *false:*

```
Cookie c = new Cookie(COOKIENAME, sensitivedata);  // Sensitive: a security-sensitive cookie is created with the secure flag not defined (b
```

Classes should not be empty (java:S2094)

Severidad: MINOR

Root_cause:

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 {

}
```

Resources should be closed (java:S2095)

Severidad: BLOCKER

Root_cause:

Connections, streams, files, and other classes that implement the `Closeable` interface or its super-interface, `AutoCloseable`, needs to be closed after use. Further, that `close` call must be made in a `finally` block otherwise an exception could keep the call from being made. Preferably, when class implements `AutoCloseable`, resource should be created using "try-with-resources" pattern and will be closed automatically.

Failure to properly close resources will result in a resource leak which could bring first the application and then perhaps the box the application is on to their knees.

Noncompliant code example

```
private void readTheFile() throws IOException {
  Path path = Paths.get(this.fileName);
  BufferedReader reader = Files.newBufferedReader(path, this.charset);
  // ...
  reader.close();  // Noncompliant
  // ...
  Files.lines("input.txt").forEach(System.out::println); // Noncompliant: The stream needs to be closed
}

private void doSomething() {
  OutputStream stream = null;
  try {
    for (String property : propertyList) {
      stream = new FileOutputStream("myfile.txt");  // Noncompliant
      // ...
    }
  } catch (Exception e) {
    // ...
  } finally {
    stream.close();  // Multiple streams were opened. Only the last is closed.
  }
}
```

Compliant solution

```
private void readTheFile(String fileName) throws IOException {
    Path path = Paths.get(fileName);
    try (BufferedReader reader = Files.newBufferedReader(path, StandardCharsets.UTF_8)) {
      reader.readLine();
      // ...
    }
    // ..
    try (Stream<String> 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 ( ... ) {
  //...
}
```

Resources:

- CWE - CWE-459 - Incomplete Cleanup
- CWE - CWE-772 - Missing Release of Resource after Effective Lifetime
- CERT, FIO04-J. - Release resources when they are no longer needed
- CERT, FIO42-C. - Close files when they are no longer needed
- Try With Resources

"equals(Object obj)" should test the argument's type (java:S2097)

Severidad: MINOR

Resources:

Documentation

- Oracle SDK - java.lang.Object#equals(Object obj)

Root_cause:

The `Object#equals(Object obj)` method is used to compare two objects to see if they are equal.

The `obj` parameter’s type is `Object`, this means that an object of any type can be passed as a parameter to this method.

Any class overriding `Object#equals(Object obj)` should respect this contract, accept any object as an argument, and return `false` when the argument’s type differs from the expected type. The `obj` parameter’s type can be checked using `instanceof` or by comparing the `getClass()` value:

```
@Override
public boolean equals(Object obj) {
  // ...
  if (this.getClass() != obj.getClass()) {
    return false;
  }
  // ...
}
```

However, it is an issue to assume that the `equals` method will only be used to compare objects of the same type. Casting the `obj` parameter without a prior test will throw a `ClassCastException` instead of returning false.

```
public class MyClass {
  @Override
  public boolean equals(Object obj) {
    MyClass that = (MyClass) obj; // may throw a ClassCastException
    // ...
  }
  // ...
}
```

This rule raises an issue when `obj` parameter’s type has not been tested before a cast operation.

How_to_fix:

Ensure the `obj` parameter’s type is checked by comparing `this.getClass()` and `obj.getClass()`, or use the `instanceof` operator to test `obj’s type.

Noncompliant code example

```
public class MyClass {
  @Override
  public boolean equals(Object obj) {
    if (this == obj) {
      return true;
    }
    if (obj == null) {
      return false;
    }
    MyClass that = (MyClass) obj; // Noncompliant, may throw a ClassCastException
    // ...
  }
  // ...
}
```

Compliant solution

```
public class MyClass {
  @Override
  public boolean equals(Object obj) {
    if (this == obj) {
      return true;
    }
    if (obj == null || this.getClass() != obj.getClass()) {
      return false;
    }
    MyClass that = (MyClass) obj; // Compliant, obj's type is MyClass
    // ...
  }
  // ...
}
```

Min and max used in combination should not always return the same value (java:S3065)

Severidad: MAJOR

Root_cause:

When using `Math.min()` and `Math.max()` together for bounds checking, it’s important to feed the right operands to each method. `Math.min()` should be used with the **upper** end of the range being checked, and `Math.max()` should be used with the **lower** end of the range. Get it backwards, and the result will always be the same end of the range.

Noncompliant code example

```
  private static final int UPPER = 20;
  private static final int LOWER = 0;

  public int doRangeCheck(int num) {    // Let's say num = 12
    int result = Math.min(LOWER, num);  // result = 0
    return Math.max(UPPER, result);     // Noncompliant; result is now 20: even though 12 was in the range
  }
```

Compliant solution

Swapping method `min()` and `max()` invocations without changing parameters.

```
  private static final int UPPER = 20;
  private static final int LOWER = 0;

  public int doRangeCheck(int num) {    // Let's say num = 12
    int result = Math.max(LOWER, num);  // result = 12
    return Math.min(UPPER, result);     // Compliant; result is still 12
  }
```

or swapping bounds `UPPER` and `LOWER` used as parameters without changing the invoked methods.

```
  private static final int UPPER = 20;
  private static final int LOWER = 0;

  public int doRangeCheck(int num) {    // Let's say num = 12
    int result = Math.min(UPPER, num);  // result = 12
    return Math.max(LOWER, result);     // Compliant; result is still 12
  }
```

Asserts should not be used to check the parameters of a public method (java:S4274)

Severidad: MAJOR

Root_cause:

An `assert` is inappropriate for parameter validation because assertions can be disabled at runtime in the JVM, meaning that a bad operational setting would completely eliminate the intended checks. Further, `asserts` that fail throw `AssertionErrors`, rather than throwing some type of `Exception`. Throwing `Errors` is completely outside of the normal realm of expected `catch`/`throw` behavior in normal programs.

This rule raises an issue when a `public` method uses one or more of its parameters with `asserts`.

Noncompliant code example

```
 public void setPrice(int price) {
  assert price >= 0 && price <= MAX_PRICE;
  // Set the price
 }
```

Compliant solution

```
 public void setPrice(int price) {
  if (price < 0 || price > MAX_PRICE) {
    throw new IllegalArgumentException("Invalid price: " + price);
  }
  // Set the price
 }
```

Resources:

Programming With Assertions

Getters and setters should access the expected fields (java:S4275)

Severidad: CRITICAL

Root_cause:

Getters and setters provide a way to enforce encapsulation by providing `public` methods that give controlled access to `private` fields. However, in classes with multiple fields, it is not unusual that copy and paste is used to quickly create the needed getters and setters, which can result in the wrong field being accessed by a getter or setter.

This rule raises an issue in any of these cases:

- A setter does not update the field with the corresponding name.
- A getter does not access the field with the corresponding name.

Noncompliant code example

```
class A {
  private int x;
  private int y;

  public void setX(int val) { // Noncompliant: field 'x' is not updated
    this.y = val;
  }

  public int getY() { // Noncompliant: field 'y' is not used in the return value
    return this.x;
  }
}
```

Compliant solution

```
class A {
  private int x;
  private int y;

  public void setX(int val) {
    this.x = val;
  }

  public int getY() {
    return this.y;
  }
}
```

Having a permissive Cross-Origin Resource Sharing policy is security-sensitive (java:S5122)

Severidad: MINOR

Default:

Having a permissive Cross-Origin Resource Sharing policy is security-sensitive. It has led in the past to the following vulnerabilities:

- CVE-2018-0269
- CVE-2017-14460

Same origin policy in browsers prevents, by default and for security-reasons, a javascript frontend to perform a cross-origin HTTP request to a resource that has a different origin (domain, protocol, or port) from its own. The requested target can append additional HTTP headers in response, called CORS, that act like directives for the browser and change the access control policy / relax the same origin policy.

Ask Yourself Whether

- You don't trust the origin specified, example: `Access-Control-Allow-Origin: untrustedwebsite.com`.
- Access control policy is entirely disabled: `Access-Control-Allow-Origin: *`
- Your access control policy is dynamically defined by a user-controlled input like `origin` header.

There is a risk if you answered yes to any of those questions.

Recommended Secure Coding Practices

- The `Access-Control-Allow-Origin` header should be set only for a trusted origin and for specific resources.
- Allow only selected, trusted domains in the `Access-Control-Allow-Origin` header. Prefer whitelisting domains over blacklisting or allowing any domain (do not use * wildcard nor blindly return the `Origin` header content without any checks).

Sensitive Code Example

Java servlet framework:

```
@Override
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
    resp.setHeader("Content-Type", "text/plain; charset=utf-8");
    resp.setHeader("Access-Control-Allow-Origin", "*"); // Sensitive
    resp.setHeader("Access-Control-Allow-Credentials", "true");
    resp.setHeader("Access-Control-Allow-Methods", "GET");
    resp.getWriter().write("response");
}
```

Spring MVC framework:

- CrossOrigin

```
@CrossOrigin // Sensitive
@RequestMapping("")
public class TestController {
    public String home(ModelMap model) {
        model.addAttribute("message", "ok ");
        return "view";
    }
}
```

- cors.CorsConfiguration

```
CorsConfiguration config = new CorsConfiguration();
config.addAllowedOrigin("*"); // Sensitive
config.applyPermitDefaultValues(); // Sensitive
```

- servlet.config.annotation.CorsConfiguration

```
class Insecure implements WebMvcConfigurer {
  @Override
  public void addCorsMappings(CorsRegistry registry) {
    registry.addMapping("/**")
      .allowedOrigins("*"); // Sensitive
  }
}
```

User-controlled origin:

```
public ResponseEntity<String> userControlledOrigin(@RequestHeader("Origin") String origin) {
  HttpHeaders responseHeaders = new HttpHeaders();
  responseHeaders.add("Access-Control-Allow-Origin", origin); // Sensitive

  return new ResponseEntity<>("content", responseHeaders, HttpStatus.CREATED);
}
```

Compliant Solution

Java Servlet framework:

```
@Override
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
    resp.setHeader("Content-Type", "text/plain; charset=utf-8");
    resp.setHeader("Access-Control-Allow-Origin", "trustedwebsite.com"); // Compliant
    resp.setHeader("Access-Control-Allow-Credentials", "true");
    resp.setHeader("Access-Control-Allow-Methods", "GET");
    resp.getWriter().write("response");
}
```

Spring MVC framework:

- CrossOrigin

```
@CrossOrigin("trustedwebsite.com") // Compliant
@RequestMapping("")
public class TestController {
    public String home(ModelMap model) {
        model.addAttribute("message", "ok ");
        return "view";
    }
}
```

- cors.CorsConfiguration

```
CorsConfiguration config = new CorsConfiguration();
config.addAllowedOrigin("http://domain2.com"); // Compliant
```

- servlet.config.annotation.CorsConfiguration

```
class Safe implements WebMvcConfigurer {
  @Override
  public void addCorsMappings(CorsRegistry registry) {
    registry.addMapping("/**")
      .allowedOrigins("safe.com"); // Compliant
  }
}
```

User-controlled origin validated with an allow-list:

```
public ResponseEntity<String> userControlledOrigin(@RequestHeader("Origin") String origin) {
  HttpHeaders responseHeaders = new HttpHeaders();
  if (trustedOrigins.contains(origin)) {
    responseHeaders.add("Access-Control-Allow-Origin", origin);
  }

  return new ResponseEntity<>("content", responseHeaders, HttpStatus.CREATED);
}
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2021 Category A7 - Identification and Authentication Failures
- developer.mozilla.org - CORS
- developer.mozilla.org - Same origin policy
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- OWASP HTML5 Security Cheat Sheet - Cross Origin Resource Sharing
- CWE - CWE-346 - Origin Validation Error
- CWE - CWE-942 - Overly Permissive Cross-domain Whitelist

Assess_the_problem:

Ask Yourself Whether

- You don't trust the origin specified, example: `Access-Control-Allow-Origin: untrustedwebsite.com`.
- Access control policy is entirely disabled: `Access-Control-Allow-Origin: *`
- Your access control policy is dynamically defined by a user-controlled input like `origin` header.

There is a risk if you answered yes to any of those questions.

Sensitive Code Example

Java servlet framework:

```
@Override
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
    resp.setHeader("Content-Type", "text/plain; charset=utf-8");
    resp.setHeader("Access-Control-Allow-Origin", "*"); // Sensitive
    resp.setHeader("Access-Control-Allow-Credentials", "true");
    resp.setHeader("Access-Control-Allow-Methods", "GET");
    resp.getWriter().write("response");
}
```

Spring MVC framework:

- CrossOrigin

```
@CrossOrigin // Sensitive
@RequestMapping("")
public class TestController {
    public String home(ModelMap model) {
        model.addAttribute("message", "ok ");
        return "view";
    }
}
```

- cors.CorsConfiguration

```
CorsConfiguration config = new CorsConfiguration();
config.addAllowedOrigin("*"); // Sensitive
config.applyPermitDefaultValues(); // Sensitive
```

- servlet.config.annotation.CorsConfiguration

```
class Insecure implements WebMvcConfigurer {
  @Override
  public void addCorsMappings(CorsRegistry registry) {
    registry.addMapping("/**")
      .allowedOrigins("*"); // Sensitive
  }
}
```

User-controlled origin:

```
public ResponseEntity<String> userControlledOrigin(@RequestHeader("Origin") String origin) {
  HttpHeaders responseHeaders = new HttpHeaders();
  responseHeaders.add("Access-Control-Allow-Origin", origin); // Sensitive

  return new ResponseEntity<>("content", responseHeaders, HttpStatus.CREATED);
}
```

How_to_fix:

Recommended Secure Coding Practices

- The `Access-Control-Allow-Origin` header should be set only for a trusted origin and for specific resources.
- Allow only selected, trusted domains in the `Access-Control-Allow-Origin` header. Prefer whitelisting domains over blacklisting or allowing any domain (do not use * wildcard nor blindly return the `Origin` header content without any checks).

Compliant Solution

Java Servlet framework:

```
@Override
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
    resp.setHeader("Content-Type", "text/plain; charset=utf-8");
    resp.setHeader("Access-Control-Allow-Origin", "trustedwebsite.com"); // Compliant
    resp.setHeader("Access-Control-Allow-Credentials", "true");
    resp.setHeader("Access-Control-Allow-Methods", "GET");
    resp.getWriter().write("response");
}
```

Spring MVC framework:

- CrossOrigin

```
@CrossOrigin("trustedwebsite.com") // Compliant
@RequestMapping("")
public class TestController {
    public String home(ModelMap model) {
        model.addAttribute("message", "ok ");
        return "view";
    }
}
```

- cors.CorsConfiguration

```
CorsConfiguration config = new CorsConfiguration();
config.addAllowedOrigin("http://domain2.com"); // Compliant
```

- servlet.config.annotation.CorsConfiguration

```
class Safe implements WebMvcConfigurer {
  @Override
  public void addCorsMappings(CorsRegistry registry) {
    registry.addMapping("/**")
      .allowedOrigins("safe.com"); // Compliant
  }
}
```

User-controlled origin validated with an allow-list:

```
public ResponseEntity<String> userControlledOrigin(@RequestHeader("Origin") String origin) {
  HttpHeaders responseHeaders = new HttpHeaders();
  if (trustedOrigins.contains(origin)) {
    responseHeaders.add("Access-Control-Allow-Origin", origin);
  }

  return new ResponseEntity<>("content", responseHeaders, HttpStatus.CREATED);
}
```

See

- OWASP - Top 10 2021 Category A5 - Security Misconfiguration
- OWASP - Top 10 2021 Category A7 - Identification and Authentication Failures
- developer.mozilla.org - CORS
- developer.mozilla.org - Same origin policy
- OWASP - Top 10 2017 Category A6 - Security Misconfiguration
- OWASP HTML5 Security Cheat Sheet - Cross Origin Resource Sharing
- CWE - CWE-346 - Origin Validation Error
- CWE - CWE-942 - Overly Permissive Cross-domain Whitelist

Root_cause:

Having a permissive Cross-Origin Resource Sharing policy is security-sensitive. It has led in the past to the following vulnerabilities:

- CVE-2018-0269
- CVE-2017-14460

Same origin policy in browsers prevents, by default and for security-reasons, a javascript frontend to perform a cross-origin HTTP request to a resource that has a different origin (domain, protocol, or port) from its own. The requested target can append additional HTTP headers in response, called CORS, that act like directives for the browser and change the access control policy / relax the same origin policy.

"String#replace" should be preferred to "String#replaceAll" (java:S5361)

Severidad: CRITICAL

Root_cause:

The underlying implementation of `String::replaceAll` calls the `java.util.regex.Pattern.compile()` method each time it is called even if the first argument is not a regular expression. This has a significant performance cost and therefore should be used with care.

When `String::replaceAll` is used, the first argument should be a real regular expression. If it’s not the case, `String::replace` does exactly the same thing as `String::replaceAll` without the performance drawback of the regex.

This rule raises an issue for each `String::replaceAll` used with a `String` as first parameter which doesn’t contains special regex character or pattern.

Noncompliant code example

```
String init = "Bob is a Bird... Bob is a Plane... Bob is Superman!";
String changed = init.replaceAll("Bob is", "It's"); // Noncompliant
changed = changed.replaceAll("\\.\\.\\.", ";"); // Noncompliant
```

Compliant solution

```
String init = "Bob is a Bird... Bob is a Plane... Bob is Superman!";
String changed = init.replace("Bob is", "It's");
changed = changed.replace("...", ";");
```

Or, with a regex:

```
String init = "Bob is a Bird... Bob is a Plane... Bob is Superman!";
String changed = init.replaceAll("\\w*\\sis", "It's");
changed = changed.replaceAll("\\.{3}", ";");
```

Resources:

- S4248 - Regex patterns should not be created needlessly

Custom getter method should not be used to override record's getter behavior (java:S6211)

Severidad: MAJOR

Root_cause:

Before records appeared in Java 16, there was a common way to represent getters for private fields of a class: a method named "get" with a capitalized field name. For example, for a `String` field named "myField" the signature of the getter method will be: `public String getMyField()`

In records, getters are named differently. Getters created by default do not contain the "get" prefix. So for a record’s `String` field "myField" the getter method will be: `public String myField()`

This means that if you want to override the default getter behavior it is better to use the method provided by records instead of creating a new one. Otherwise, this will bring confusion to the users of the record as two getters will be available and even leads to bugs if the behavior is different from the default one.

This rule raises an issue when a record contains a getter named "get" with a capitalized field name that is not behaving the same as the default one.

Noncompliant code example

```
record Person(String name, int age) {
    public String getName() { // Noncompliant
        return name.toUpperCase(Locale.ROOT);
    }
}
```

Compliant solution

```
record Person(String name, int age) {
    @Override
    public String name() { // Compliant
        return name.toUpperCase(Locale.ROOT);
    }
}

record Person(String name, int age) {
    public String getNameUpperCase() { // Compliant
        return name.toUpperCase(Locale.ROOT);
    }
}
record Person(String name, int age) {
    public String getName() { // Compliant, is equivalent to 'name()'
        return name;
    }
}
record Person(String name, int age) {
    @Override
    public String name() { // Compliant
        return name.toUpperCase(Locale.ROOT);
    }
    public String getName() { // Compliant, equal to 'name()'
        return name.toUpperCase(Locale.ROOT);
    }
}
```

Exceptions

If the implementations of `getMyField()` and `myField()` methods are equivalent, the issue should not be raised as this was probably done to support compatibility with the previous convention.

Resources:

- Records specification

Regular expressions should not contain empty groups (java:S6331)

Severidad: MAJOR

Root_cause:

There are several reasons to use a group in a regular expression:

- to change the precedence (e.g. `do(g|or)` will match 'dog' and 'door')
- to remember parenthesised part of the match in the case of capturing group
- to improve readability

In any case, having an empty group is most probably a mistake. Either it is a leftover after refactoring and should be removed, or the actual parentheses were intended and were not escaped.

Noncompliant code example

```
"foo()"  // Noncompliant, will match only 'foo'
```

Compliant solution

```
"foo\\(\\)"  // Matches 'foo()'
```

Return values should not be ignored when they contain the operation status code (java:S899)

Severidad: MINOR

Resources:

- CERT, EXP00-J. - Do not ignore values returned by methods
- CERT, FIO02-J. - Detect and handle file-related errors
- CWE - CWE-754 - Improper Check for Unusual Exceptional Conditions

Root_cause:

When the return value of a function call contains the operation status code, this value should be tested to make sure the operation completed successfully.

This rule raises an issue when the return values of the following are ignored:

- `java.io.File` operations that return a status code (except `mkdirs`)
- `Iterator.hasNext()`
- `Enumeration.hasMoreElements()`
- `Lock.tryLock()`
- non-void `Condition.await*` methods
- `CountDownLatch.await(long, TimeUnit)`
- `Semaphore.tryAcquire`
- `BlockingQueue`: `offer`, `remove`

Noncompliant code example

```
public void doSomething(File file, Lock lock) {
  file.delete();  // Noncompliant
  // ...
  lock.tryLock(); // Noncompliant
}
```

Compliant solution

```
public void doSomething(File file, Lock lock) {
  if (!lock.tryLock()) {
    // lock failed; take appropriate action
  }
  if (!file.delete()) {
    // file delete failed; take appropriate action
  }
}
```

Try-with-resources should be used (java:S2093)

Severidad: CRITICAL

Resources:

- CERT, ERR54-J. - Use a try-with-resources statement to safely handle closeable resources
- The Java™ Tutorials - The try-with-resources Statement

Root_cause:

Many resources in Java need be closed after they have been used. If they are not, the garbage collector cannot reclaim the resources' memory, and they are still considered to be in use by the operating system. Such resources are considered to be leaked, which can lead to performance issues.

Java 7 introduced the try-with-resources statement, which guarantees that the resource in question will be closed.

```
try (InputStream input = Files.newInputStream(path)) {
  // "input" will be closed after the execution of this block
}
```

This syntax is safer than the traditional method using `try`, `catch`, and `finally` and hence should be preferred.

This rule raises an issue if a closeable resources is not opened using a try-with-resources statement.

This rule is automatically disabled when the project’s `sonar.java.source` is lower than `7` as the close-with-resources statement was unavailable prior to Java 7.

How_to_fix:

Use the try-with-resources syntax by moving the `Closable` variable declarations after the `try` keyword surrounded by parentheses and separated by `;`:

```
try (/* resources declarations */) {
  // resources usage ...
}
```

Noncompliant code example

```
FileReader fr = null;
BufferedReader br = null;

try { // Noncompliant, the FileReader and BufferedReader are instantiated without try-with-resources
  fr = new FileReader(fileName);
  br = new BufferedReader(fr);
  return br.readLine();
} catch (...) {
  ...
} finally {

  if (br != null) { // br has to be closed manually
    try {
      br.close();
    } catch(IOException e){...}
  }

  if (fr != null ) { // fr has to be closed manually
    try {
      br.close();
    } catch(IOException e){...}
  }

}
```

Compliant solution

```
try ( // Compliant, all resources are instantiated within a try-with-resources statement and hence automatically closed after use
    FileReader fr = new FileReader(fileName);
    BufferedReader br = new BufferedReader(fr)
  ) {
  return br.readLine();
}
catch (...) {}
```