sonar

RULES

Secrets

ABAP

Apex

C

C++

CloudFormation

COBOL

C#

CSS

Flex

Go

HTML

**Java**

JavaScript

Kotlin

Objective C

PHP

PL/I

PL/SQL

Python

RPG

Ruby

Scala

Swift

Terraform

Text


TypeScript

T-SQL

VB.NET

VB6

XML

Java

# Java static code analysis

Unique rules to find Bugs, Vulnerabilities, Security Hotspots, and Code Smells in your JAVA code

All rules632

Vulnerability53

Bug154

Security Hotspot36

Code Smell389

Quick Fix42

Tags

Search by name...

Weak SSL/TLS protocols should not be used

Vulnerability

"SecureRandom" seeds should not be predictable

Vulnerability

Cipher Block Chaining IVs should be unpredictable

Vulnerability

Basic authentication should not be used

Vulnerability

Regular expressions should not be vulnerable to Denial of Service attacks

Vulnerability

"HttpServletRequest.getRequestSession" should not be used

Vulnerability

Hashes should include an unpredictable salt

Vulnerability

Calls to methods should not trigger an IllegalArgumentException

Bug

Unsupported methods should not be called on some collection implementations

Bug

Cast operations should not trigger a ClassCastException

Bug

Members ignored during record serialization should not be used

Bug

"clone" should not be overridden

Analyze your code

Code Smell

Blocker

suspicious

Many consider `clone` and `Cloneable` broken in Java, largely because the rules for overriding `clone` are tricky and difficult to get right, according to Joshua Bloch:

Object's `clone` method is very tricky. It's based on field copies, and it's "extra-linguistic." It creates an object without calling a constructor. There are no guarantees that it preserves the invariants established by the constructors. There have been lots of bugs over the years, both in and outside Sun, stemming from the fact that if you just call `super.clone` repeatedly up the chain until you have cloned an object, you have a shallow copy of the object. The clone generally shares state with the object being cloned. If that state is mutable, you don't have two independent objects. If you modify one, the other changes as well. And all of a sudden, you get random behavior.

A copy constructor or copy factory should be used instead.

This rule raises an issue when `clone` is overridden, whether or not `Cloneable` is implemented.

Noncompliant Code Example

```
public class MyClass {
    // ...

    public Object clone() { // Noncompliant
        //...
    }
}
```

Compliant Solution

```
public class MyClass {
    // ...

    MyClass (MyClass source) {
        //...
    }
}
```

See

- [Copy Constructor versus Cloning](#)







See Also

- `{rule:java:S2157}` - "Cloneables" should implement "clone"
- `{rule:java:S1182}` - Classes that override "clone" should be "Cloneable" and call "super.clone()"

Available In:

https://rules.sonarsource.com/java/RSPEC-2975

1/2

<p>Map "computeIfAbsent()" and "computeIfPresent()" should not be used to add "null" values.</p> <p> Bug</p>	<div><div>sonarlint    sonarcloud    sonarqube </div><div>© 2008-2022 SonarSource S.A., Switzerland. All content is copyright protected. SONAR, SONARSOURCE, SONARLINT, SONARQUBE and SONARCLOUD are trademarks of SonarSource S.A. All other trademarks and copyrights are the property of their respective owners. All rights are expressly reserved. <a href="#">Privacy Policy</a></div></div>
<p>Regex lookahead assertions should not be contradictory</p> <p> Bug</p>	
<p>Back references in regular expressions should only refer to capturing groups that are matched before the reference</p> <p> Bug</p>	
<p>Regex boundaries should not be used in a way that can never be matched</p>	