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 static code analysis

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

All rules 632 | Vulnerability 53 | Bug 154 | Security Hotspot 36 | Code Smell 389 | Quick Fix 42

Tags

Search by name...

Bug

Printf-style format strings should not lead to unexpected behavior at runtime
Bug

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

Methods should not call same-class methods with incompatible "@Transactional" values
Bug

Recursion should not be infinite
Bug

Loops should not be infinite
Bug

Double-checked locking should not be used
Bug

Resources should be closed
Bug

Hard-coded credentials are security-sensitive
Security Hotspot

Methods returns should not be invariant
Code Smell

"ThreadGroup" should not be used
Code Smell

"clone" should not be overridden
Code Smell

I/O function calls should not be vulnerable to path injection attacks

Analyze your code

Vulnerability | Blocker | injection cwe owasp sans-top25

User-provided data, such as URL parameters, POST data payloads, or cookies, should always be considered untrusted and tainted. Constructing file system paths directly from tainted data could enable an attacker to inject specially crafted values, such as `'../'`, that change the initial path and, when accessed, resolve to a path on the filesystem where the user should normally not have access.

A successful attack might give an attacker the ability to read, modify, or delete sensitive information from the file system and sometimes even execute arbitrary operating system commands. This is often referred to as a "path traversal" or "directory traversal" attack.

The mitigation strategy should be based on the whitelisting of allowed paths or characters.

Noncompliant Code Example

```
protected void doGet(HttpServletRequest req, HttpServletResp
    String file = request.getParameter("file");

    File fileUnsafe = new File(file);
    try {
      FileUtils.forceDelete(fileUnsafe); // Noncompliant
    }
    catch(IOException ex){
      System.out.println (ex.toString());
    }
}
```

Compliant Solution

```
protected void doGet(HttpServletRequest req, HttpServletResp
    String file = request.getParameter("file");

    File fileUnsafe = new File(file);
    File directory = new File("/tmp/");

    try {
      if(FileUtils.directoryContains(directory, fileUnsafe))
        FileUtils.forceDelete(fileUnsafe); // Compliant
      }
    }
    catch(IOException ex){
      System.out.println (ex.toString());
    }
}
```

See

Assertions should be complete

Code Smell

Tests should include assertions

Code Smell

Silly bit operations should not be performed

Code Smell

Child class fields should not shadow parent class fields

Code Smell

JUnit test cases should call super methods

- OWASP Top 10 2021 Category A1 - Broken Access Control
- OWASP Top 10 2021 Category A3 - Injection
- OWASP Top 10 2017 Category A1 - Injection
- OWASP Top 10 2017 Category A5 - Broken Access Control
- MITRE, CWE-20 - Improper Input Validation
- MITRE, CWE-22 - Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
- MITRE, CWE-99 - Improper Control of Resource Identifiers ('Resource Injection')
- MITRE, CWE-641 - Improper Restriction of Names for Files and Other Resources
- SANS Top 25 - Risky Resource Management

Available In:

sonarcloud | sonarqube Developer Edition