sonar RULES

Products

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...

"Collections.EMPTY_LIST", "EMPTY_MAP", and "EMPTY_SET" should not be used
Code Smell

Local variables should not be declared and then immediately returned or thrown
Code Smell

Unused local variables should be removed
Code Smell

Private fields only used as local variables in methods should become local variables
Code Smell

"public static" fields should be constant
Code Smell

Loops should not contain more than a single "break" or "continue" statement
Code Smell

Declarations should use Java collection interfaces such as "List" rather than specific implementation classes such as "LinkedList"
Code Smell

"switch" statements should have at least 3 "case" clauses
Code Smell

A "while" loop should be used instead of a "for" loop
Code Smell

The default unnamed package should not be used
Code Smell

"equals(Object obj)" should be

Formatting SQL queries is security-sensitive

Analyze your code

Security Hotspot Major

cwe spring owasp sans-top25 bad-practice cert hibernate sql

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 {rule:javasecurity: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 SQLE

  Statement stmt1 = null;
  Statement stmt2 = null;
  PreparedStatement pstmt;
  try {
    stmt1 = con.createStatement();
    ResultSet rs1 = stmt1.executeQuery("GETDATE()"); // No i

    stmt2 = con.createStatement();
    ResultSet rs2 = stmt2.executeQuery("select FNAME, LNAME,
                 "from USERS where UNAME=" + user);  // Sens

    pstmt = con.prepareStatement("select FNAME, LNAME, SSN "
                 "from USERS where UNAME=" + user);  // Sens
    ResultSet rs3 = pstmt.executeQuery();

    //...
}

public User getUserHibernate(org.hibernate.Session session,

  org.hibernate.Query query = session.createQuery(
            "FROM students where fname = " + data);  // Sens
  // ...
}
```

"equals(Object obj)" should be overridden along with the "compareTo(T obj)" method

Code Smell

Package names should comply with a naming convention

Code Smell

Nested code blocks should not be used

Code Smell

Array designators "[]" should be on the type, not the variable

Code Smell

Compliant Solution

```
public User getUser(Connection con, String user) throws SQLE

  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 e
    ResultSet rs2 = pstmt.executeQuery();

    //...
  }
}

public User getUserHibernate(org.hibernate.Session session,

  org.hibernate.Query query =  session.createQuery("FROM stu
  query = query.setParameter(0,data);  // Good; Parameter bi

  org.hibernate.Query query2 =  session.createQuery("FROM st
  // ...
```

See

- OWASP Top 10 2021 Category A3 - Injection
- OWASP Top 10 2017 Category A1 - Injection
- MITRE, CWE-89 - Improper Neutralization of Special Elements used in an SQL Command
- MITRE, CWE-564 - SQL Injection: Hibernate
- MITRE, CWE-20 - Improper Input Validation
- MITRE, CWE-943 - Improper Neutralization of Special Elements in Data Query Logic
- CERT, IDS00-J. - Prevent SQL injection
- SANS Top 25 - Insecure Interaction Between Components
- Derived from FindSecBugs rules Potential SQL/JPQL Injection (JPA), Potential SQL/JDOQL Injection (JDO), Potential SQL/HQL Injection (Hibernate)

Available In:

sonarcloud sonarqube