Java static code analysis

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

Security Hotspot

Allowing deserialization of LDAP objects is security-sensitive
Security Hotspot

Setting loose POSIX file permissions is security-sensitive
Security Hotspot

Formatting SQL queries is security-sensitive
Security Hotspot

Deprecated annotations should include explanations
Code Smell

Restricted Identifiers should not be used as Identifiers
Code Smell

Redundant constructors/methods should be avoided in records
Code Smell

Records should be used instead of ordinary classes when representing immutable data structure
Code Smell

"Stream.toList()" method should be used instead of "collectors" when unmodifiable list needed
Code Smell

Operator "instanceof" should be used instead of "A.class.isInstance()"
Code Smell

String multiline concatenation should be replaced with Text Blocks
Code Smell

Single-character alternations in regular expressions should be

XML operations should not be vulnerable to injection attacks

Analyze your code

Vulnerability Major injection cwe owasp

User-provided data, such as URL parameters, POST data payloads, or cookies, are tainted with user-controlled inputs. Therefore, they should always be considered untrusted.

XML injection occurs when user-controlled input is embedded into XML syntax. In such a scenario, an attacker can abuse the XML syntax and inject arbitrary XML elements. This can lead to cross-site scripting issues or denial-of-service attacks.

In addition, the application may parse the XML document with a weakly configured XML parser (namely, a parser with external entity support enabled). In this case, an XML External Entity (XXE) injection would expose the application to sensitive file disclosure or server-side request forgeries (SSRF).

An attacker can also elevate privileges if the XML data is used for authentication.

Noncompliant Code Example

```
protected void doGet(HttpServletRequest req, HttpServletResp
  String xml = "<node id=\""+req.getParameter("id")+\"></nod
  FileOutputStream fos = new FileOutputStream("output.xml");
  fos.write(xml.getBytes(Charset.forName("UTF-8")));  // Non
}
```

javax.xml.parsers.DocumentBuilder

```
protected void doGet(HttpServletRequest req, HttpServletResp
  String xml = "<node id=\""+req.getParameter("id")+\"></nod

  DocumentBuilderFactory factory = DocumentBuilderFactory.ne
  DocumentBuilder builder = factory.newDocumentBuilder();
  Document doc = builder.parse(new InputSource(new StringRea
}
```

Compliant Solution

```
protected void doGet(HttpServletRequest req, HttpServletResp
  String id = req.getParameter("id");
  if(id.matches("^[A-Za-z0-9_]+$")) {
    String xml = "<node id=\""+id+\"></node>";
    FileOutputStream fos = new FileOutputStream("output.xml"
    fos.write(xml.getBytes(Charset.forName("UTF-8")));
  }
}
```

javax.xml.parsers.DocumentBuilder

```
protected void doGet(HttpServletRequest req, HttpServletResp
    String xml = "<node></node>;
```

replaced with character classes

Code Smell

Reluctant quantifiers in regular expressions should be followed by an expression that can't match the empty string

Code Smell

Constructors of an "abstract" class should not be declared "public"

Code Smell

Similar tests should be grouped in a single Parameterized test

Code Smell

```
    DocumentBuilderFactory factory = DocumentBuilderFactory.
    DocumentBuilder builder = factory.newDocumentBuilder();
    Document doc = builder.parse(new InputSource(new StringR
    Element element = (Element) doc.getElementsByTagName("so
    element.setAttribute("id", req.getParameter("id"));
}
```

See

- OWASP Top 10 2021 Category A3 - Injection
- OWASP Top 10 2017 Category A1 - Injection
- MITRE, CWE-20 - Improper Input Validation
- MITRE, CWE-76 - Improper Neutralization of Equivalent Special Elements

Available In:
sonarcloud sonarqube Developer Edition