**JAVA 17(LTS)**  
  
Java 17 was a significant milestone, but Java 21 has now taken 17's place as the next long-term support release (LTS).

Java 21 is infinitely better than Java 8. It’s technically superior on all fronts. It’s faster, more secure, more operations-friendly, more performant and more memory efficient.

Each new Java version builds on the previous ones, adding new features, improving performance, and enhancing the developer experience. Upgrading to a newer LTS version like Java 17 or Java 21 can provide significant benefits in terms of features, performance, and support.

Why should we move from Java 11?

* Although Java 11 is also an LTS version and is used by many application, there are some major reasons why we might want to shift to Java 17.
* Ending Support for Java 11: Java 11 will be supported till September 2023 and extended support will be provided till September 2026. This means that after the support ends, we would have no patches(not even the security ones).
* Spring 6: The latest version of Spring, Spring 6 will require Java 17 to work, and as there are many libraries which work along with them, they would also be moving to Java 17. If your applications rely on the Spring Framework, you should definitely consider moving to Java 17.
* Free Oracle JDK present for Java 17: Java 17 is issued under the new NFTC (Oracle No-Fee Terms and Conditions) license. It is therefore again allowed to use the Oracle JDK version for free for production and commercial use.(was not allowed for Java 11).

**Process**

Starting with Java update first is a good idea since it often has broader implications for the rest of the project, particularly if you’re using frameworks or libraries that are closely tied to Java. Once Java is updated and verified, you can move on to Python, Node.js, and npm updates, followed by SonarQube.

**1. Preparation**

* **Backup**: Ensure you have a complete backup of your current project and configurations.
* **Document Current State**: Record the versions of Java, Python, Node.js, npm, and dependencies.

**2. Update Java**

1. **Download and Install Java 17**: Install the latest stable Java version.
2. **Update pom.xml**:
   * **Set Java Version**: Modify maven.compiler.plugin in pom.xml.(It requires JDK version)
   * **In this project case change only <jdk.min.version>**
   * **Update Plugins and Dependencies**: Ensure the maven-compiler-plugin and other dependencies are compatible.
3. **Build and Test**:
   * **Clean Build**: Execute mvn clean install.
   * **Run Tests**: Ensure all unit and integration tests pass.
   * **Refactor Code**: Address any deprecation or compatibility issues

**4. Update Node.js and npm**

1. **Download and Install Latest Node.js and npm**: Install the latest stable versions.
2. **NodeJS version** : v20.16.0 (Latest LTS)
3. **NPM is bundled with node: npm version : 10.8.1**
4. **Update Dependencies**:
   * **Update package.json**: Adjust versions of dependencies and scripts.
   * In this project we have used the Caret (^) symbol for versions.
   * npm outdated – to find outdated packages
   * npm update – to update packages
   * **Run npm install**: Update installed packages.
5. **Test Compatibility**:
   * **Run Tests**: Ensure all functionality works with the new Node.js and npm versions.

**SASS error**

* npm list sass
* npm install sass@latest sass-loader@latest
* npm update(to verify)

old version   
+-- [sass-loader@13.3.3](mailto:sass-loader@13.3.3)

| `-- sass@1.32.12 deduped

New version(not throwing out any error but affecting UI)  
+-- sass-loader@13.2.1

| `-- sass@1.59.3

**5. Update SonarQube**

1. **Download and Install Latest SonarQube**: Version 9.9(LTA)
2. **Update Plugins**:
   * **Check Compatibility**: Update plugins and custom rules to be compatible with the new SonarQube version.
3. **Test SonarQube**:
   * **Run Analysis**: Verify that SonarQube analysis works correctly.

**6. Systematic Testing and Validation**

1. **Automated Tests**: Run all automated tests across the updated environment.
2. **Manual Testing**: Conduct manual tests for critical functionality.
3. **OWASP Checks**: Ensure compliance with OWASP security guidelines.

**Dependencies(versions)**

**<properties>**

**<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>**

**<gson.version>2.9.1</gson.version>**

**<jdk.min.version>11</jdk.min.version>**

**<sonar.sources>src/main/java,src/main/js</sonar.sources>**

**<sonarjava.version>7.2.0.26923</sonarjava.version>**

**<sonarphp.version>3.23.1.8766</sonarphp.version>**

**<sonarpython.version>3.4.1.8066</sonarpython.version>**

**<sonar.apiVersion>8.9.9.56886</sonar.apiVersion>**

**</properties>**

**sonarqube-community-branch-plugin**

(https://github.com/mc1arke/sonarqube-community-branch-plugin/releases?page=1)

[**1.8.2**](https://github.com/mc1arke/sonarqube-community-branch-plugin/releases/tag/1.8.2)

**Note: This version only supports Sonarqube 8.9. Sonarqube 8.8 and below or 9.0 and above are not supported in this release**

**New Features:**

* **Hide outdated summary comments in Github Pull Request decoration (**[**#435**](https://github.com/mc1arke/sonarqube-community-branch-plugin/pull/435)**)**
* **Support mono-repo decoration for Bitbucket repositories (**[**#554**](https://github.com/mc1arke/sonarqube-community-branch-plugin/issues/554)**,**[**#578**](https://github.com/mc1arke/sonarqube-community-branch-plugin/pull/578)**)**

**Bug Fixes:**

* **Azure Devops decoration fails where project or repository names contain spaces (**[**#461**](https://github.com/mc1arke/sonarqube-community-branch-plugin/issues/461)**,**[**#466**](https://github.com/mc1arke/sonarqube-community-branch-plugin/pull/466)**)**
* **Azure Devops server requires preview flag to be set on status API requests (**[**#442**](https://github.com/mc1arke/sonarqube-community-branch-plugin/issues/442)**,**[**#467**](https://github.com/mc1arke/sonarqube-community-branch-plugin/pull/467)**)**
* **Azure Devops decoration fails when findings are set to certain statuses (**[**#441**](https://github.com/mc1arke/sonarqube-community-branch-plugin/pull/441)**,**[**#439**](https://github.com/mc1arke/sonarqube-community-branch-plugin/issues/439)**)**
* **Content with accented or cyrillic characters causes errors during Gitlab Merge request decoration (**[**#445**](https://github.com/mc1arke/sonarqube-community-branch-plugin/issues/445)**,**[**#472**](https://github.com/mc1arke/sonarqube-community-branch-plugin/pull/472)**)**
* **Decoration of Gitlab repositories fails for forked repositories (**[**#554**](https://github.com/mc1arke/sonarqube-community-branch-plugin/issues/554)**,**[**#555**](https://github.com/mc1arke/sonarqube-community-branch-plugin/pull/555)**)**

**Benefits of upgrading java version**

Upgrading the Java version in your SonarQube project can offer several benefits and some considerations. Here’s a comparison of what you might gain by moving from Java 11 to Java 17

**1. Performance Improvements**

* **Enhanced Performance**: Newer Java versions often include optimizations and improvements in the JVM, garbage collection, and runtime performance.
* **Better Startup Time**: Modern versions of Java may have faster startup times and reduced memory usage.

**2. New Language Features**

* **Language Enhancements**: New Java versions come with updated language features that can make code more concise and expressive. For instance:
  + **Java 12**: Switch expressions (preview).
  + **Java 14**: Records and pattern matching (preview).
  + **Java 16**: Sealed classes.
  + **Java 17**: Enhanced pattern matching for switch, new sealed types.
  + **Java 21**: New features like virtual threads (preview), enhanced pattern matching, and more.
* **Improved Code Quality**: Use modern features to write more robust and readable code.

**3. Long-Term Support (LTS)**

* **LTS Versions**: Java 17 is an LTS version, meaning it will receive long-term support and updates, making it a stable choice for production environments. Java 21 is also an LTS version with extended support.
* **Security Updates**: LTS versions get extended security updates and bug fixes.

**4. Improved Libraries and APIs**

* **Enhanced Standard Library**: New Java versions come with updated standard libraries and APIs that may offer new functionality or improve existing features.

**5. Tooling and Ecosystem**

* **Updated Tools**: Build tools (like Maven, Gradle), IDEs, and other tools in the Java ecosystem are often optimized for newer Java versions.
* **SonarQube Plugins**: Ensure that SonarQube plugins for Java are compatible with the new Java version to leverage improvements and bug fixes.

**6. Security Enhancements**

* **Better Security**: Newer Java versions include security improvements and fixes for vulnerabilities discovered in older versions.

**Considerations**

* **Compatibility**: Ensure that all your project’s dependencies, libraries, and custom rules are compatible with the new Java version.
* **Testing**: Thoroughly test your application to catch any issues introduced by the new Java version.

**Steps to Upgrade**

1. **Update pom.xml**:
   * Change <jdk.min.version> to the new version (e.g., <jdk.min.version>17</jdk.min.version>).
   * Update maven-compiler-plugin and other relevant plugins to versions compatible with the new Java version.
2. **Update Dependencies**:
   * Check for compatibility of your dependencies with the new Java version and update them if necessary.
3. **Rebuild and Test**:
   * Perform a clean build and run all tests to ensure everything works correctly with the new Java version.
4. **Update SonarQube**:
   * Make sure SonarQube and its plugins are compatible with the new Java version and update them if needed.

Upgrading to a newer Java version can provide performance, security, and feature enhancements, but it’s essential to carefully manage compatibility and testing to ensure a smooth transition.

🡪**Java 21 and Java 17**

**Execution Speed and Startup Time**

* **Java 21**:
  + **Virtual Threads**: Can significantly improve concurrent applications' performance by reducing thread management overhead.
  + **Garbage Collection Improvements**: Includes continued enhancements to garbage collectors like ZGC for better performance and reduced latency.
* **Java 17**:
  + **Garbage Collection**: Includes improvements to ZGC and G1GC, but not as advanced as those in Java 21.
  + **No Virtual Threads**: Lacks the virtual threads feature, which could impact performance in highly concurrent applications.

**Tooling Integration**

* **Maven**:
  + **Java 21**: Compatible with the latest Maven versions and plugins, benefiting from ongoing updates and improvements.
  + **Java 17**: Fully compatible with Maven; however, Java 21 may provide newer Maven plugin features or optimizations.
* **Spring**:
  + **Java 21**: Compatible with the latest Spring Framework versions, which may leverage new Java features for better performance and developer experience.
  + **Java 17**: Also well-supported by Spring; however, Java 21’s new features might offer additional performance and functionality benefits.
* **Hibernate**:
  + **Java 21**: Works with the latest Hibernate versions, potentially utilizing new Java features to enhance ORM performance and capabilities.
  + **Java 17**: Fully supported by Hibernate; Java 21 may provide incremental performance improvements or new features.
* **SonarQube**:
  + **Java 21**: Compatible with the latest SonarQube versions and plugins, which might benefit from new Java features for code analysis.
  + **Java 17**: Fully supported; Java 21 could offer additional benefits based on updates in SonarQube tooling.

**Summary**

* **Java 21**: Offers enhancements in execution speed and startup time through virtual threads and garbage collection improvements. It integrates with the latest versions of Maven, Spring, Hibernate, and SonarQube, providing updated features and optimizations.
* **Java 17**: Provides solid performance improvements and full support for key tools, but lacks some of the advanced features in Java 21, such as virtual threads.

Java 21 is designed to build on the improvements of Java 17 with additional features that can lead to better performance and a more modern development experience.

## **OWASP**

## What Is OWASP Dependency-Check and How It Can Improve Application Security

**OWASP Dependency-Check is a Software Composition Analysis (SCA) tool that actively scans through a project’s dependencies to detect and report on publicly disclosed vulnerabilities, thereby improving application security.**

Open source projects often suffer from security vulnerabilities. If left unchecked, these vulnerabilities can compromise entire systems that rely on these open source tools. According to a recent security [report](https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html) by Open Source Security and Risk Analysis (OSSRA), an average of 528 open source components were found per application in 2020.

It is critical for organizations to [focus on the software supply chain](https://www.aquasec.com/cloud-native-academy/supply-chain-security/software-supply-chain-attacks/), including third-party components. Developers need a solution integrated with the application code and used to track vulnerabilities in external dependencies. OWASP Dependency-Check is an open-source solution created by the OWASP project, famous for its OWASP Top 10 list of vulnerabilities, designed to help developers mitigate open-source security threats, thereby securing the application.

OWASP Dependency-Check is a [Software Composition Analysis](https://www.aquasec.com/cloud-native-academy/supply-chain-security/software-composition-analysis-sca/) (SCA) tool that actively scans through a project’s dependencies, detects and reports publicly disclosed vulnerabilities, ensuring application security.

* We have to first register and get an API key from national vulnerability database.
* The below link contains commands to validate the API key and various other information related to NVD API key.

https://jeremylong.github.io/DependencyCheck/dependency-check-maven/configuration.html

* if error 403 or 404 occurs  
  <https://github.com/jeremylong/Open-Vulnerability-Project/tree/main/vulnz>  
    
    
  **For certificates issue**
* you need to import the server's SSL certificate into your Java keystore (trust store). You can find java certificates in C:\Program Files\jdk-17.0.9\lib\security\ cacerts.

Use the keytool command to import the certificate:

keytool -import -alias <alias> -keystore <path\_to\_truststore> -file <certificate\_file>

You can list certificates in your TrustStore with the following command:

keytool -list -v -keystore <path\_to\_truststore>

* If adding in java keystore does not work try adding in local system

**Open the Microsoft Management Console (MMC)**:

* Press Win + R, type mmc, and press Enter.

**Add the Certificates Snap-in**:

* In the MMC, click **File** > **Add/Remove Snap-in**.
* Select **Certificates** and click **Add**.
* Choose **Computer account** and click **Next**, then select **Local computer** and click **Finish**.
* Click **OK** to close the Add/Remove Snap-in dialog.

**Import the Certificate**:

* In the left pane, expand **Certificates (Local Computer)** > **Trusted Root Certification Authorities**.
* Right-click on **Certificates**, select **All Tasks** > **Import**.
* Follow the wizard to import your certificate file (.crt or .pem) into the store.

**Complete the Wizard**:

* Browse to the certificate file and select it.
* Ensure that you choose **Place all certificates in the following store** and select **Trusted Root Certification Authorities**.
* Finish the import process and close the MMC.
* You do not need to configure the NVD API key in settings.xml if you are setting it up through the environment variables or directly in your Maven build configuration. However, if you'd like to ensure the key is available to Maven during the build process, you can add it to the settings.xml file under a server or property configuration.

Here’s an example of adding it as a property in the settings.xml:

<settings>

<profiles>

<profile>

<id>owasp-profile</id>

<properties>

<nvd.api.key>Your-API-Key-Here</nvd.api.key>

</properties>

</profile>

</profiles>

<activeProfiles>

<activeProfile>owasp-profile</activeProfile>

</activeProfiles>

</settings>

**On Windows:**

1. Open File Explorer.
2. Navigate to C:\Users\<YourUsername>\.m2.
3. Look for settings.xml. If it doesn’t exist, you can create it.

**Benefits of OWASP Dependency-Check**

### 1. Free tool

As OWASP Foundation is a non-profit organization, the Dependency-Check tool is free. Developers can download the tool and start using it as part of their security stack.

### 2. Ease of use

Dependency-Check is easy to get started and does not require reviewing documentation, training, or certification.

### 3. Lightweight

Dependency-Check is relatively small in size, using a simple process to scan the code. The only maintenance it requires is a regular update of a local copy of its vulnerability data.

### 4. Extensive Reporting

Dependency-Check offers multiple reporting options for developers to check and rectify open source vulnerabilities. The tool’s export features allow teams to create vulnerability reports that focus on their key metrics.

### 5. Compatibility

Dependency-check is compatible with many popular languages, technologies, and platforms, including:

* Complete support for Java and .NET-based products
* Experimental support for Node.js, Ruby, and Python projects
* Partially compatible with C and C++
* Intergrates with Maven, Jenkins, and Gradle via plugins and can run through the CLI as an Ant task

## How Dependency-Check Detects Vulnerabilities

OWASP Dependency-Check identifies vulnerabilities using Analyzers. These are dedicated open source projects that execute the entire dependency scanning process. Essentially a sniffing tool, analyzers are responsible for examining every data packet to verify its relevancy and record its information. They can be used for data retrieval and [error scanning](https://spectralops.io/blog/top-9-git-secret-scanning-tools/).

### How Analyzers Work

Dependency-Check scans through files and collects information about project dependencies through a series of analyzers. This information collected – known as evidence – is bucketed into three categories: vendor, platform, and version information. It is then tagged with a confidence level (low, medium, high, and highest). The confidence rating is a method of flagging the potential vulnerability that must be verified.

For example, the Jar analyzer accumulates data from the Manifest, pom.xml, and the package names within the JAR files scanned. After gathering the necessary information, it employs a process to place this information into one or more buckets of evidence.

The Common Platform Enumeration (CPE) of the dependency is determined, and the result is assigned a level of confidence. This level of confidence is based on the lowest confidence rating of the evidence used. The identified CPE is recorded in the Lucene Index and subsequently cross-checked against the Common Vulnerabilities and Exposures (CVE) entries in the National Vulnerability Database (NVD), a free-to-use database of known information-security vulnerabilities.

Dependency-Check automatically updates itself with the NVD data as soon as it is run. These updates ensure that the reports show only the most recent data.

The results are compared and made available in various formats like HTML, XML, CSV, and JSON for developers to take appropriate action. However, the Dependency-Check tool doesn’t take the context of your dependencies when reporting the vulnerability scores. So, developers must verify if the vulnerability exposes their code.

Here is a list of the Analyzers currently available as part of the Dependency-Check tool.

### ****11 Different File Type Analyzers****

The tool contains the following analyzers, each of which is able to scan a specific type of file:

* **Archive** – extracts and scans archive content
* **Assembly** – needs .NET framework or Mono runtime
* **Jar** – scans archive manifest metadata, and Maven Project Object Model files
* **RetireJS** – examines JavaScript files
* **Node.js** – gathers a [bill-of-materials](https://www.aquasec.com/cloud-native-academy/supply-chain-security/sbom/) after parsing a [package.json](http://tabnine.com/blog/node-modules-package-json/)
* **Node Audit** – requires internet access and uses APIs to expose vulnerable node.js libraries
* **NugetConf** – parses specification XML using XPath
* **Nuspec** – parses specification XML using XPath
* **OpenSSL** – parses OPENSSL\_VERSION\_NUMBER macro definition
* **OSS Index** – similar to Node Audit, it uses internet to use APIs and report vulnerabilities not listed in the NVD
* **Ruby bundler-audit** – responsible of executing bundle-audit reports and adding the results in final report

To achieve OWASP Dependency Check in your SonarQube Maven project, you need to integrate OWASP Dependency-Check into your Maven build process. This tool helps identify vulnerabilities in project dependencies. Here’s a step-by-step guide to set it up:

**1. Add OWASP Dependency-Check Plugin to Your Maven pom.xml**

Add the OWASP Dependency-Check plugin to your pom.xml to configure it for your project.

<build>

<plugins>

<plugin>

<groupId>org.owasp</groupId>

<artifactId>dependency-check-maven</artifactId>

<version>8.2.2</version> <!-- Check for the latest version -->

<configuration>

<!-- Configuration options here -->

<format>HTML</format>

<failBuildOnCVSS>7</failBuildOnCVSS> <!-- Set the CVSS threshold to fail the build if vulnerabilities exceed this score -->

</configuration>

</plugin>

</plugins>

</build>

**2. Run OWASP Dependency-Check**

Execute the OWASP Dependency-Check plugin as part of your Maven build process:

bash

Copy code

mvn org.owasp:dependency-check-maven:check

This will generate a report with details of any vulnerabilities found in your dependencies.

**3. Review the Reports**

After running the check, review the generated report in the target/dependency-check-report.html file (or whichever format you configured). This will list any vulnerabilities detected in your dependencies.

**4. Integrate with SonarQube**

To integrate OWASP Dependency-Check results with SonarQube, follow these steps:

1. **Use the SonarQube Plugin for Security Reports**: If you’re using SonarQube Enterprise Edition or above, you can use the SonarQube Security Reports Plugin to import security issues, including those from OWASP Dependency-Check.
2. **Upload Reports Manually**: For Community Edition, you may need to manually review the OWASP Dependency-Check reports as SonarQube Community Edition does not natively support integrating security reports. However, you can still view and act on the vulnerabilities listed in the reports.

**5. Automate in CI/CD Pipeline**

Integrate the OWASP Dependency-Check command into your CI/CD pipeline to ensure that dependency checks are performed automatically with every build:

# Example for a CI pipeline

steps:

- name: Run OWASP Dependency-Check

run: mvn org.owasp:dependency-check-maven:check

**Summary**

* **Add** the OWASP Dependency-Check plugin to your Maven pom.xml.
* **Run** the dependency check command to generate reports.
* **Review** the reports for vulnerabilities.
* **Integrate** results with SonarQube (if using Enterprise Edition) or manually review vulnerabilities.
* **Automate** the process in your CI/CD pipeline.

This will help you maintain the security of your project's dependencies by regularly checking for vulnerabilities.

To check javatrust certificates

keytool -list -keystore "C:\Program Files\Java\jdk-17.0.9\lib\security\cacerts"