

Software Dependability Report

Gianluca Ronga

January 5, 2025

1 Context of the Project

The major disadvantage of the Java platform is that still today the only portable way to start a Java application relies on a single point of entry: the public static void main(String[]) method.

Having a single-point of entry is a valid solution for client applications, where interactively a user can command to the application to quit, but in those cases where the application is not interactive (server applications) there is currently no portable way to notify the Virtual Machine of its imminent shutdown.

The Apache Commons Daemon project is part of the Apache Commons library and provides utilities for creating, managing, and running Java applications as daemons or services. A "daemon" (on Unix-based systems) or "service" (on Windows) is a background process that runs independently of any interactive user session, which makes it suitable for applications that need to perform tasks without requiring user intervention, like servers, network applications, or scheduled background jobs.

2 Goal of the Project

The goal of the Apache Commons Daemon project is to provide a standardized framework for creating and managing Java applications that run as background services. In particular, Apache Commons Daemon simplifies the process of running Java applications as persistent low-level background services that start and stop with the operating system. The project includes tools and utilities to ensure that Java applications can be controlled using typical system administration commands, such as starting, stopping, and restarting.

3 Methodology

3.1 SonarCloud

SonarCloud was used to perform an in-depth analysis of the software's quality. After analyzing the code, the following issues were found:

▼ Type	▼ Software Quality
✖ Bug 19	Security 0
🔒 Vulnerability 0	Reliability 19
⊕ Code Smell 552	Maintainability 552

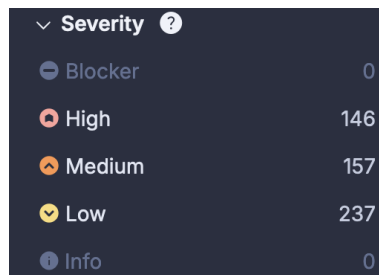
▼ Severity ?	
🛑 Blocker 0	
🔴 High 160	
🟡 Medium 169	
🟢 Low 242	
ℹ Info 0	

3.1.1 SonarCloud Report

The focus was set on Maintainability issues, in particular:

- A total of 150 issues were classified as high-severity code smells, most of them regarding the Cognitive Complexity of functions being too high. Cognitive complexity is a measure of how hard it is to understand the control flow of a unit of code. Code with high cognitive complexity is hard to read, understand, test, and modify.
- A total of 153 issues were classified as medium severity code smells, they were of several types: replacing the `system.err` or `system.out` with a logger, catching the `Exception` instead of a `Throwable` and some "unused" and "redundancy" errors.
- A total of 242 issues were classified as low severity, they were all code smells and include: deprecated functions, redundant casts, resolve the `TODO` comments, etc.

Some of these issues were solved from all of the three severity of errors, lowering the number of issues as follows:



Severity ?	
Blocker	0
High	146
Medium	157
Low	237
Info	0

3.1.2 Issues Severity after Corrections

The decision not to address many Cognitive Complexity issues was primarily due to two factors: the risk of introducing new bugs or code smells during refactoring, as these are inherently complex functions, and the challenge of isolating them. These functions often have numerous dependencies, so refactoring would require extensive modifications to other parts of the code.

3.2 Docker

Docker is an open-source platform designed to automate the deployment, scaling, and management of applications in lightweight, portable containers. In order to achieve containerization, a `Dockerfile` was needed, a file that contains collections of commands that will be automatically executed, in sequence, in the Docker environment to build a new Docker image.

In the `Dockerfile`, the following code was written:

```
FROM openjdk:8
ADD target/simple-docker.jar simple-docker.jar
ENTRYPOINT ["java", "-jar", "simple-docker.jar"]
EXPOSE 8080
```

- **FROM:** The `FROM` keyword tells the Docker the base image for the build.
- **ADD and ENTRYPOINT:** those keywords find and manage the new `jar` file created in the `/target` directory.
- **EXPOSE:** this keyword makes the 8080 port available outside the Docker container.

Additionally, several Java classes were created to simulate a Daemon process and test whether the project was runnable. The goal was to generate two files: `output.txt` and `log.txt`. One class writes to the `output.txt` file, whereas a thread monitors changes to this file and logs the infos in `log.txt`. The Daemon process includes a 5-second cooldown to prevent the `log.txt` file from overflowing due to excessive inputs.

There is the full `fileDaemon.java` class (excluding imports):

```
public class fileDaemon {

    private static final Logger logger = Logger.getLogger(example.class.getName());
```

```

static class FileMonitorDaemon implements Runnable {
    private final String monitoredFile;
    private final String logFile;
    private long lastModified = 0;

    public FileMonitorDaemon(String monitoredFile, String logFile) {
        this.monitoredFile = monitoredFile;
        this.logFile = logFile;
    }

    @Override
    public void run() {
        System.out.println("File monitor daemon avviato.");
        while (true) {
            try {
                Thread.sleep(5000);
                checkForNewEntries();
            } catch (InterruptedException | IOException e) {
                logger.log(Level.SEVERE, "An error occurred", e);

                if (e instanceof InterruptedException) {
                    Thread.currentThread().interrupt();
                }
            }
        }
    }

    private void checkForNewEntries() throws IOException {
        File file = new File(monitoredFile);
        if (file.exists() && file.lastModified() > lastModified) {
            logEntry(logFile, "File " + monitoredFile +
                " modificato alle " + new Date(file.lastModified()));
            lastModified = file.lastModified();
        }
    }

    private void logEntry(String logFileName, String message) throws IOException {
        try (OutputStream logFile = new FileOutputStream(logFileName, true);
            PrintStream logOut = new PrintStream(logFile)) {
            SimpleDateFormat fmt = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
            String logEntry = fmt.format(new Date()) + " - " + message;
            logOut.println(logEntry);
        }
    }
}

```

3.3 JaCoCo

JaCoCo was used in order to measure code coverage. By employing this tool it was possible to generate a comprehensive report that shows the extent to which the code has been covered. These pictures below provides a clear and concise overview of the code coverage achieved:

Apache Commons Daemon

[Sessions](#)

Apache Commons Daemon





Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Cxty	Missed Lines	Missed Methods	Missed Classes
org.apache.commons.daemon.support	<div><div></div></div>	0%	<div><div></div></div>	0%	98	98	317	317
org.apache.commons.daemon	<div><div></div></div>	1%	<div><div></div></div>	0%	41	42	92	94
org.apache.commons.daemon.docker	<div><div></div></div>	0%	<div><div></div></div>	0%	18	18	57	57
Total	1.782 of 1.787	0%	171 of 171	0%	157	158	466	468

Created with JaCoCo 0.8.12.202403310830

3.3.1 JaCoCo Project Report

As shown in the image 3.3.1, the Instructions and Branches coverages are basically 0%, in the following image we can see the report of the only class that had some coverage, that is DaemonInitException:

DaemonInitException

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
getMessageWithCause()		0%		0%	2	2	2	2	1	1
DaemonInitException(String)		0%		n/a	1	1	2	2	1	1
DaemonInitException(String, Throwable)		100%		n/a	0	1	0	2	0	1
Total	29 of 34	14%	2 of 2	0%	3	4	4	6	2	3

3.3.2 JaCoCo DaemonException Report

The probable reason on why this class was the only one tested may be that Commons Daemon is a project designed to facilitate the integration of Java applications with system daemons. A significant portion of its core functionality relies heavily on native code (C/C++) and interaction with the operating environment. This strong dependency on external systems may have rendered Java unit tests less effective in thoroughly verifying the project's behavior. As a result, the developers might have prioritized manual testing or system-level integration tests, leaving the implementation of a comprehensive Java unit test suite as a lower priority.

3.4 Pitest

Pitest is an open-source tool designed for mutation testing in Java applications. This testing approach evaluates the quality of a test suite by intentionally injecting defects, known as mutations, into the code and checking whether the test suite can identify them. Pitest works by generating altered versions of the source code using different mutation operators, such as eliminating method calls or modifying mathematical operators. Each mutation simulates a possible defect, helping to gauge the effectiveness of the test suite.

Line Coverage

Line coverage measures the percentage of code lines executed during testing. It identifies which lines of the source code are reached by the test suite, ensuring that critical paths in the application are exercised. However, high line coverage does not guarantee comprehensive testing, as it doesn't account for untested logic or edge cases.

Mutation Coverage

Mutation coverage evaluates the quality of the test suite by introducing small changes (mutations) to the code, such as altering operators or variables, and checking if the tests fail as expected. If tests do not catch these mutations, it indicates insufficient coverage. This approach highlights gaps in testing logic that might otherwise be missed by traditional metrics like line coverage.

Test Strength

Test strength refers to the ability of a test suite to detect bugs or subtle errors in the code. It assesses whether the tests not only execute the code but also validate its correctness against expected behavior. A strong test suite ensures robust validations, covering edge cases, boundary conditions, and error scenarios effectively.

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
8	0% 2/467	0% 0/225	100% 0/0

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage	Test Strength
org.apache.commons.daemon	2	2% 2/94	0% 0/54	100% 0/0
org.apache.commons.daemon.docker	3	0% 0/56	0% 0/22	100% 0/0
org.apache.commons.daemon.support	3	0% 0/317	0% 0/149	100% 0/0

Report generated by [PIT](#) 1.17.1

Enhanced functionality available at [arcmutate.com](#)

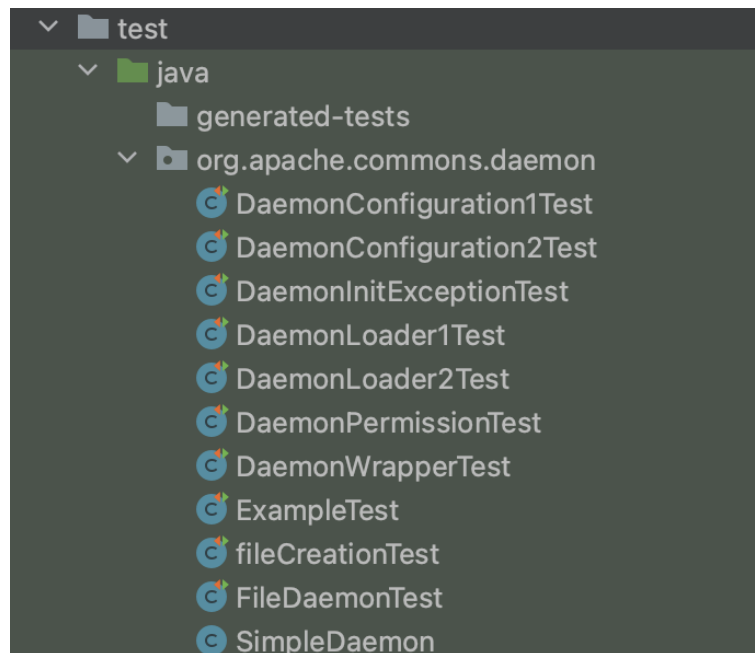
3.4.1 Pitest Project Report

As shown in the image, the mutation coverage value is nearly identical to the line coverage. Given the current state of the project, it is meaningless to delve deeper into the coverage reports without first creating additional test cases.

3.5 Randoop

Randoop is an automated test generation tool for Java programs. It creates unit tests by randomly exploring the program's behavior, constructing sequences of method calls, and validating the program's output against its contracts, such as avoiding exceptions or returning consistent results.

This tool was used in order to drastically increase the amount of code coverage of the project, and it's output was limited to 1000 in order to reduce the generation time.



3.5.1 Test Classes Generated by Randoop

The tests generated by Randoop encountered compatibility issues with the JUnit version being used, specifically due to the use of the generic "assert" method. This issue was resolved by replacing occurrences of "assert" with the appropriate specific assertions, such as "assertNull" and "assertNotNull" throughout the test classes.

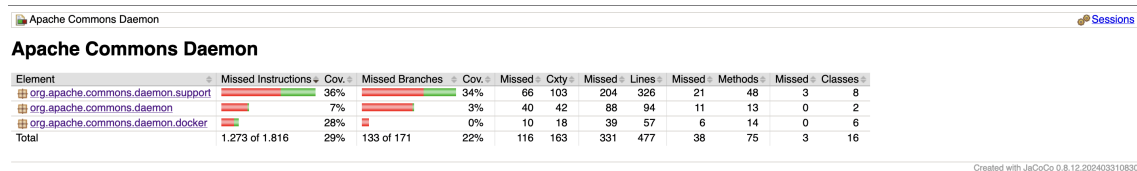
Randoop encountered challenges in generating effective tests for the DaemonWrapper class, as the tests produced were inconclusive. Additionally, compatibility issues with the JUnit version used led to numerous errors and, as a result, GitHub Copilot was utilized to generate test suite specifically for that class.

3.6 Github Copilot

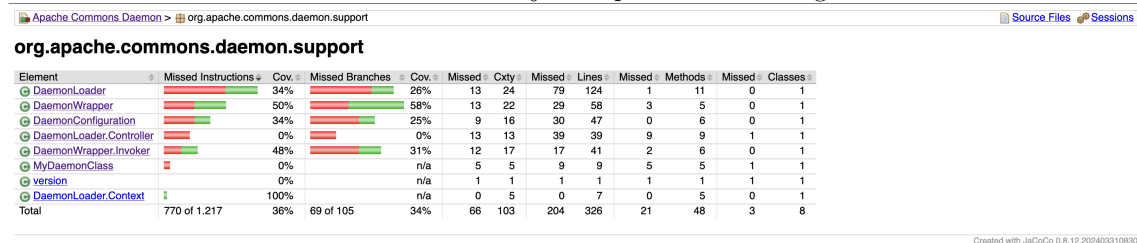
GitHub Copilot is an AI-powered code assistant developed by GitHub in collaboration with OpenAI. It integrates directly into popular code editors, like IntelliJ IDEA, to provide real-time code suggestions, auto-completions, and solutions based on the context of the code being written.

GitHub Copilot was utilized to generate the test suite for the DaemonWrapper class due to multiple issues encountered with Randoop. The generated tests were specifically tailored for JUnit 5 to ensure compatibility. Additionally, GitHub Copilot was employed to create a test case for the load() method of the DaemonLoader class, as Randoop did not adequately test that method.

The following images display the updated JaCoCo reports for both the entire project and the "support" package, which contains the highest number of Java classes in the project.



3.6.1 JaCoCo Project Report After Testing



3.6.2 JaCoCo Support Package Report After Testing

The PIT Test Coverage Report shows a significant improvement compared to the previous analysis, where the project had almost no test coverage. The current report indicates 20% line coverage, 14% mutation coverage, and 71% test strength across the entire project. Within the "support" package the mutation coverage increased to 18%, with a notable test strength of 70%. While the mutation coverage is still relatively low, these metrics highlight the progress made in testing the project, demonstrating the effectiveness of the test cases introduced since the initial evaluation.

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
9	20% 94/476	14% 32/229	71% 32/45

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage	Test Strength
org.apache.commons.daemon	2	7% 7/94	2% 1/54	50% 1/2
org.apache.commons.daemon.docker	3	16% 9/56	14% 3/22	100% 3/3
org.apache.commons.daemon.support	4	24% 78/326	18% 28/153	70% 28/40

Report generated by PIT 1.15.2

Enhanced functionality available at [arcmutate.com](https://arc42.org/arcmutate/)

3.6.3 Pitest Project Report After Testing

3.7 JMH

JMH (Java Microbenchmark Harness) is a Java library developed by the same team that created the Java Virtual Machine (JVM). It is designed for building reliable and accurate benchmarks to measure the performance of Java code. JMH handles complexities such as JVM optimizations, warm-up phases, and multi-threading, ensuring precise and reproducible benchmarking results.

In the project, JMH was utilized to evaluate the performance of various functionalities, as shown in the image 3.7.1.; this analysis identified the class with the most time-consuming test execution, which turned out to be DaemonLoader1Test with 0.411 seconds.

```
[INFO] Running org.apache.commons.daemon.DaemonWrapperTest
[INFO] Tests run: 9, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.036 s - in org.apache.commons.daemon.DaemonWrapperTest
[INFO] Running org.apache.commons.daemon.fileCreationTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.002 s - in org.apache.commons.daemon.fileCreationTest
[INFO] Running org.apache.commons.daemon.DaemonConfiguration2Test
[INFO] Tests run: 43, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.027 s - in org.apache.commons.daemon.DaemonConfiguration2Test
[INFO] Running org.apache.commons.daemon.ExampleTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.035 s - in org.apache.commons.daemon.ExampleTest
[INFO] Running org.apache.commons.daemon.DaemonInitExceptionTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.002 s - in org.apache.commons.daemon.DaemonInitExceptionTest
[INFO] Running org.apache.commons.daemon.DaemonLoader1Test
[INFO] Tests run: 500, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.411 s - in org.apache.commons.daemon.DaemonLoader1Test
[INFO] Running org.apache.commons.daemon.DaemonPermissionTest
[INFO] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.003 s - in org.apache.commons.daemon.DaemonPermissionTest
[INFO] Running org.apache.commons.daemon.DaemonLoader2Test
[INFO] Tests run: 16, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.014 s - in org.apache.commons.daemon.DaemonLoader2Test
[INFO] Running org.apache.commons.daemon.FileDaemonTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.001 s - in org.apache.commons.daemon.FileDaemonTest
[INFO] Running org.apache.commons.daemon.DaemonConfiguration1Test
[INFO] Tests run: 500, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.092 s - in org.apache.commons.daemon.DaemonConfiguration1Test
[INFO]
```

3.7.1 Time Spent to Run Tests

The benchmark compares two DaemonLoader constructors with varying inputs (1, 10, 100 actions). The "With Actions" constructor demonstrates stable performance (546k ops/ms) with minimal variation as the number of actions increases. In contrast, the "Without Actions" constructor shows lower and more variable performance, with a temporary improvement at 10 actions (450k ops/ms) followed by a decline at 100. Overall, the "With Actions" constructor is more consistent and efficient across all number of Actions.

Benchmark	Actions	Score (ops/ms)	Error (\pm ops/ms)
benchmarkConstructorWithActions	1	546,617.720	5,533.537
benchmarkConstructorWithActions	10	546,257.316	3,405.375
benchmarkConstructorWithActions	100	544,888.425	3,949.784
benchmarkConstructorWithoutActions	1	379,669.034	48,684.822
benchmarkConstructorWithoutActions	10	450,315.957	13,725.568
benchmarkConstructorWithoutActions	100	381,581.043	47,497.208

Table 1: JMH Benchmark Results for DaemonLoader Constructors

After analyzing the initial benchmarks, the DaemonLoader class was modified by optimizing logging through the introduction of conditional checks to prevent unnecessary string concatenations, thereby reducing logging overhead. Exception handling was also improved by refactoring the logic to avoid costly try-catch blocks when unnecessary. Following these optimizations, the benchmarks were re-run, demonstrating consistent improvements in performance and reduced variability, as shown in the table 2.

Benchmark	Actions	Score (ops/ms)	Error (\pm ops/ms)
benchmarkConstructorWithActions	1	558,626.778	7,038.149
benchmarkConstructorWithActions	10	557,390.637	7,024.314
benchmarkConstructorWithActions	100	554,514.871	8,886.876
benchmarkConstructorWithoutActions	1	407,609.975	46,338.702
benchmarkConstructorWithoutActions	10	403,775.187	48,640.758
benchmarkConstructorWithoutActions	100	378,158.506	45,936.472

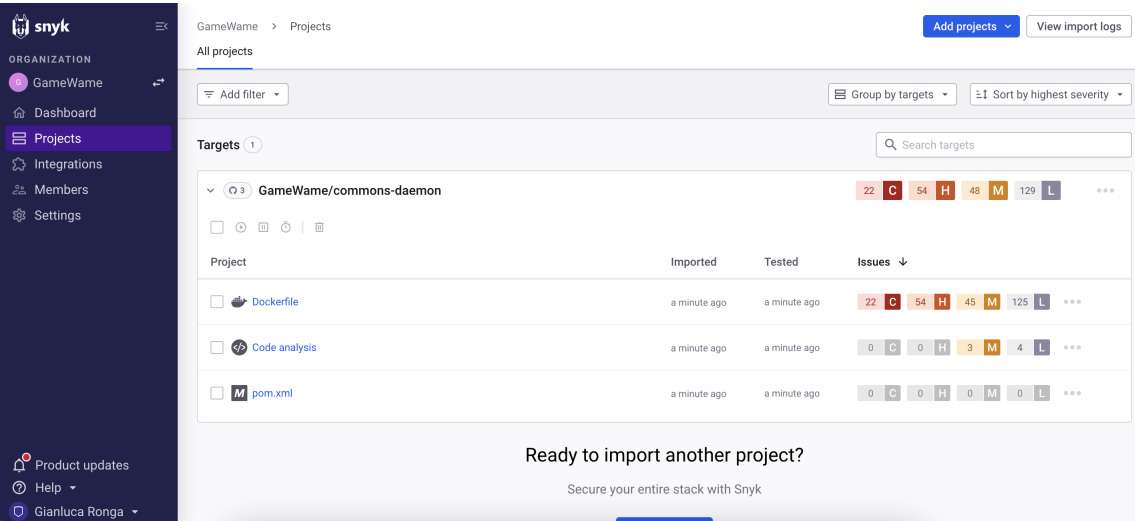
Table 2: JMH Benchmark Results for DaemonLoader after Optimization

From the comparison of these two tables we can see that the first report consistently shows higher throughput and generally similar error margins than the second one. The second benchmark report has better results because the Score (throughput) is higher in all cases. This indicates the code performs more operations per millisecond in the second run, which is the primary metric for evaluating performance.

3.8 Snyp

Snyk is a developer-first security platform designed to help identify and remediate vulnerabilities in code, open-source dependencies, container images, and infrastructure as code. By providing actionable insights and automated fixes, Snyk helps ensure secure application development without disrupting productivity.


When Snyk analyzed the project, the results (shown in image 3.8.1) revealed that nearly all security vulnerabilities were located in the Dockerfile, with no available fixes (as shown in image 3.8.2 and 3.8.3). As a result, the focus shifted towards addressing vulnerabilities in the codebase.



3.8.1 Security Bugs Summary

COMPUTED FIXABILITY	
<input type="checkbox"/> Fixable	0
<input type="checkbox"/> Partially fixable	0
<input type="checkbox"/> No supported fix	246

3.8.2 Security Bugs Fixability (Dockerfile)

 **nghttp2/libnghttp2-14** · CVE-2023-44487

VULNERABILITY

CVE-2023-44487

CVSS 7.5

HIGH

SNYK-DEBIAN11-NGHTTP2-5953384

SCORE
829

Introduced through

nghttp2/libnghttp2-14@1.43.0-1

Exploit maturity

MATURE

Fixed in

nghttp2/libnghttp2-14@1.43.0-1+deb11u1, @1.43.0-1+deb11u1

Show more detail

Ignore

3.8.3 Example of Security Bug (Dockerfile)

The code analysis report uncovered several issues, all of medium or low severity, with no high-severity bugs identified (as shown in image 3.8.4). A total of seven bugs were found, including four instances of "Missing Release of File Descriptor or Handle after Effective Lifetime," two cases of "Path Traversal," and one "Buffer Overflow." Notably, only the Path Traversal issues were related to the Java classes, while the remaining bugs originated from the C code in the project.

Project:

Scan Information ([show less](#)):

- *dependency-check version*: 8.2.1
- *Report Generated On*: Sat, 14 Dec 2024 18:59:03 +0100
- *Dependencies Scanned*: 80 (72 unique)
- *Vulnerable Dependencies*: 12
- *Vulnerabilities Found*: 20
- *Vulnerabilities Suppressed*: 0
- *NVD CVE Checked*: 2024-12-14T18:52:52
- *NVD CVE Modified*: 2024-12-14T18:00:01
- *VersionCheckOn*: 2024-12-14T18:53:00
- *kev.checked*: 1734198782

3.9.1 OWASP DC Report Summary

After cross-checking with Snyk, it was determined that some of the identified vulnerabilities were false positives, while others were linked to older releases of the dependencies that contained known security issues. To avoid disrupting the project's functionality or introducing unintended issues by upgrading these dependencies, they were left unchanged.