JDS

Software Dependability Report

LEONARDO PORCELLI and GUIDO IMMEDIATA

Research Article

CONTEXT OF THE PROJECT

The project involves utilizing and applying CI/CD tools and techniques. In our case, the source code analysed is Apache Commons Imaging (https://commons.apache.org/proper/commons-imaging/), a library that enables reading and writing various image formats. This library offers efficient parsing of image information such as size, color space, ICC profile, and metadata.

The library is written entirely in Java and, as stated in the Apache documentation, it offers greater portability compared to native code image I/O libraries. It also offers enhanced reliability and security against corrupt or malicious images, while maintaining satisfactory performance. Additionally, it supports a wide range of formats and ensures accurate handling of metadata.

GOALS OF THE PROJECT

The main goal of the project is to evaluate the dependability and security measures of the library. To achieve this objective, a combination of code analysis, testing, and security tools were employed.

By conducting code analysis, potential vulnerabilities and weaknesses within the library were identified and addressed. Testing procedures were implemented to ensure the reliability and functionality of the library under various scenarios. Additionally, security tools were used to detect and mitigate any potential security risks.

METHODOLOGY

SonarCloud

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

Software Quality	Number of Issues	Severity	Type
Security	0	/	/
Maintainability	1.9k	High: 221 Medium: 580 Low: 1.1k	Code Smell
Reliability	120	High: 54 Medium: 3 Low: 63	Bug

We focused mainly on Reliability issues. Particularly:

- A total of 54 bugs have been identified and classified as high severity bugs. These bugs primarily involve errors in divisions where the divisor is not properly checked for the possibility of being zero. Additionally, there are other errors that arise from referencing a static member of a subclass from its parent during class initialization.

Authors' address: Leonardo Porcelli; Guido Immediata.

- A total of 3 bugs have been identified and classified as medium severity bugs. These bugs involve errors where an attempt is made to retrieve the length of a palette that has the potential to be null.
- A total of 63 bugs have been identified and classified as low severity bugs. Most of these issues are related to the suggestion to remove shifts. Additionally, there are some issues related to number casting.

All the issues regarding the elimination of unnecessary shifts have been successfully resolved.

A decision has been made to not address high severity bugs that occur when referencing a static member of a subclass from its parent during class initialization. Although this coding practice is considered bad and has the potential to lead to a deadlock or initialization cycle, it is important to note that our environment is not concurrent. Therefore, a deadlock cannot occur. Furthermore, since the child constructor does not invoke the parent constructor, the occurrence of an initialization cycle is also ruled out.

JaCoCo

JaCoCo was used for calculating the code coverage. By employing this tool, we are able to generate a comprehensive report that shows the extent to which the code has been covered. The picture below provides a clear and concise overview of the code coverage achieved:

Apache Commons Imaging

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.apache.commons.imaging.formats.tiff		56%		50%	390	782	672	1.812	71	296	1	29
org.apache.commons.imaging.formats.tiff.write		65%		63%	162	381	269	1.060	41	144	0	12
org.apache.commons.imaging.formats.png		74%		71%	95	301	196	912	15	116	1	21
org.apache.commons.imaging.palette		69%		59%	106	242	179	726	12	69	0	16
org.apache.commons.imaging.common		79%		67%	153	453	240	1.037	28	200	0	20
org.apache.commons.imaging		71%		57%	158	337	217	675	70	197	1	16
org.apache.commons.imaging.formats.pcx		66%		61%	82	167	122	440	9	38	0	6
org.apache.commons.imaging.formats.bmp		71%		61%	69	175	143	637	4	53	0	13
org.apache.commons.imaging.internal		28%		25%	63	80	110	179	22	35	0	3
org.apache.commons.imaging.formats.jpeg		74%		44%	178	302	207	595	15	73	0	12
org.apache.commons.imaging.formats.pnm		68%		58%	69	183	148	498	13	86	2	14
org.apache.commons.imaging.formats.psd		68%		43%	60	99	131	319	13	41	1	6
org.apache.commons.imaging.formats.gif		78%		75%	60	197	93	609	7	62	0	12
org.apache.commons.imaging.formats.icns		81%		61%	63	144	100	358	13	50	1	8
org.apache.commons.imaging.formats.xpm		74%		59%	70	128	101	380	4	31	0	4
org.apache.commons.imaging.formats.ico		72%		73%	38	98	73	354	12	38	1	9
org.apache.commons.imaging.formats.tiff.datareaders		89%		80%	75	237	66	703	3	30	0	5
org.apache.commons.imaging.color		89%		65%	75	208	104	701	7	100	0	11
org.apache.commons.imaging.formats.tiff.itu_t4		91%		82%	38	175	75	562	5	42	0	7
org.apache.commons.imaging.icc		82%		53%	39	84	65	286	16	53	0	11
org.apache.commons.imaging.formats.jpeg.segments		76%		46%	64	137	68	304	13	67	0	17
org.apache.commons.imaging.formats.tiff.taginfos		78%		50%	41	137	61	256	23	115	1	38
org.apache.commons.imaging.formats.jpeg.iptc		84%		64%	43	107	62	356	8	50	0	8
org.apache.commons.imaging.formats.tiff.fieldtypes		75%		64%	26	81	56	219	7	34	0	9
org.apache.commons.imaging.formats.psd.dataparsers		33%		66%	13	27	39	64	12	24	2	8
org.apache.commons.imaging.formats.tiff.photometricinterpreters		74%		100%	5	34	34	159	5	23	1	10
org.apache.commons.imaging.formats.png.chunks		80%		75%	25	79	28	186	14	57	0	12
org.apache.commons.imaging.formats.xbm		78%		60%	36	72	38	188	4	23	0	3
org.apache.commons.imaging.formats.jpeg.xmp		73%		61%	22	70	42	164	11	44	0	8
org.apache.commons.imaging.bytesource		74%		58%	27	64	31	125	2	24	0	4
org.apache.commons.imaging.formats.jpeg.decoder		96%		90%	21	155	22	624	1	40	0	6
org.apache.commons.imaging.formats.webp		89%		80%	17	72	12	172	0	29	0	6
org.apache.commons.imaging.mylzw		92%		85%	19	100	20	273	6	48	0	6
org.apache.commons.imaging.formats.wbmp		79%		79%	9	34	15	94	4	22	0	3
org.apache.commons.imaging.formats.psd.datareaders		61%		50%	5	10	19	45	2	4	1	2
org.apache.commons.imaging.formats.jpeg.exif		85%		86%	11	42	23	140	7	27	0	6
org.apache.commons.imaging.formats.rgbe		90%		78%	10	46	8	122	2	25	0	4
org.apache.commons.imaging.formats.webp.chunks		93%		69%	15	59	12	131	2	38	2	11
org.apache.commons.imaging.formats.dcx		78%		71%	8	25	11	58	4	18	0	2
org.apache.commons.imaging.formats.tiff.photometricinterpreters.floatingpoint		94%		86%	15	70	11	161	3	27	0	3
org.apache.commons.imaging.formats.tiff.constants		99%		28%	4	33	5	606	1	29	0	22
benchmark		0%		n/a	2	2	2	2	2	2	1	1
org.apache.commons.imaging.formats.png.scanlinefilters		100%		100%	0	26	0	58	0	11	0	5
org.apache.commons.imaging.formats.png.transparencyfilters		100%		90%	1	14	0	31	0	9	0	4
Total	21.706 of 95.444	77%	2.608 of 7.204	63%	2.482	6.269	3.930	17.381	513	2.544	16	433

Fig. 1. JaCoCo Report

The analysis conducted using JaCoCo has disclosed a baseline level of 77% Instructions Coverage and 63% Branches Coverage. These data provide us valuable insights into the codebase, highlighting areas that require attention and improvement.

Further examination of the metrics reveals specific details, such as 2482 out of 6269 missed cyclomatic complexities, 3930 out of 17381 missed lines, 513 out of 2544 missed methods, and 16 out of 433 missed classes. These metrics serve as indicators of potential issues within the code, allowing for targeted efforts to enhance the overall quality and efficiency of the software.

Pitest

Pitest (Pit Mutation Testing) is an open-source mutation testing tool for Java applications. Mutation testing is a technique used to assess the effectiveness of a test suite by introducing intentional defects, or mutations, into the source code and then observing whether the test suite can detect these mutations. Pitest analyzes the source code and generates mutated versions of it by applying various mutation operators (e.g., changing mathematical operators, removing method calls, etc.). Each mutation represents a potential defect in the code.

Explanation of Software Testing Metrics.

Line Coverage:

- **Definition:** Line coverage, also known as code coverage, measures the percentage of lines of code in a software program that are executed during a set of tests.
- **Calculation:** It is calculated by dividing the number of lines of code executed by the total number of lines of code in the program.
- **Purpose:** Line coverage helps assess the comprehensiveness of test suites by indicating which parts of the code are exercised during testing.

Mutation Coverage:

- **Definition:** Mutation coverage is a measure of how well a test suite can detect mutations (intentional changes) made to the source code.
- **Calculation:** It is calculated as the percentage of mutations that are detected by the test suite out of the total number of mutations introduced.
- **Purpose:** Mutation testing aims to evaluate the effectiveness of a test suite by introducing small changes (mutations) to the code and checking if the tests can identify and fail due to these changes.

Test Strength:

- **Definition:** Test strength is a qualitative measure of how effective a test suite is in revealing potential issues in the software.
- **Calculation:** It is often expressed as a percentage, indicating the proportion of identified issues or covered scenarios compared to the total potential issues or scenarios.
- **Purpose:** Test strength provides an overall assessment of the quality and thoroughness of testing efforts. It considers factors such as line coverage, mutation coverage, and other testing metrics to evaluate the reliability of the software.

We decided to use Pitest on two different package to better understand its results:

(1) **palette package.**
(2) **png package.**

1. palette package

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
ColorSpaceSubset		33%		33%	10	19	43	81	4	10	0	1
PaletteFactory		64%		63%	33	84	73	235	3	17	0	1
ColorGroup		65%		26%	15	20	23	76	2	5	0	1
LongestAxisMedianCut		67%		33%	18	23	16	58	1	5	0	1
MostPopulatedBoxesMedianCut		89%		59%	15	22	12	76	0	2	0	1
MedianCutQuantizer		91%		73%	8	19	5	63	0	4	0	1
ColorCount		61%		0%	4	5	5	12	2	3	0	1
ColorCountComparator		80%		60%	2	6	2	9	0	2	0	1
QuantizedPalette		98%		90%	1	10	0	19	0	5	0	1
Dithering		100%		100%	0	16	0	56	0	2	0	1
ColorComponent		100%		n/a	0	3	0	9	0	3	0	1
SimplePalette		100%		100%	0	6	0	9	0	4	0	1
ColorGroupCut		100%		100%	0	3	0	10	0	2	0	1
MedianCutPalette		100%		100%	0	3	0	7	0	2	0	1
ColorSpaceSubset.RgbComparator		100%		n/a	0	2	0	2	0	2	0	1
PaletteFactory.DivisionCandidate		100%		n/a	0	1	0	4	0	1	0	1
Total	1.165 of 3.873	69%	135 of 337	59%	106	242	179	726	12	69	0	16

Fig. 2. Palette package Coverage

As we can see the most important class in this package is PaletteFactory, The **PaletteFactory** class serves as a software component functioning as a factory for creating color palettes. Its primary utility lies in image processing, providing methods for analyzing and generating various types of color palettes. The class manages the creation of exact, inexact, translucent, and opaque palettes, using algorithms such as Median Cut. It also includes functions to count transparent colors, check for transparency in an image, and determine if an image is in grayscale. The modularity of the class allows it to adapt to different image processing contexts.

The picture below shows the results of the mutuation test campaign on this package:

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
14	71%	517/724	35%	226/640	52%	226/431

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
org.apache.commons.imaging.palette	14	71%	517/724	35%	226/640	52%	226/431

Fig. 3. palette package pitest report

Analyzing in deep a couple of classes we have the following resulting table:

Name	Line Coverage	Mutation Coverage	Test Strength
PaletteFactory.java	58% (136/235)	32% (54/171)	63% (54/86)
ColorSpaceSubset.java	48% (40/83)	6% (6/106)	14% (6/44)
MostPopulatedBoxesMedianCut.java	84% (64/76)	36% (14/39)	40% (14/35)

Table 1. Line Coverage, Mutation Coverage, and Test Strength for the most important classes in the package

As we can see the test strenght results show a poor robustness to defects. The test suite doesn't provide a good Line Coverage, and also has a low mutuation coverage percentage.

2. png package

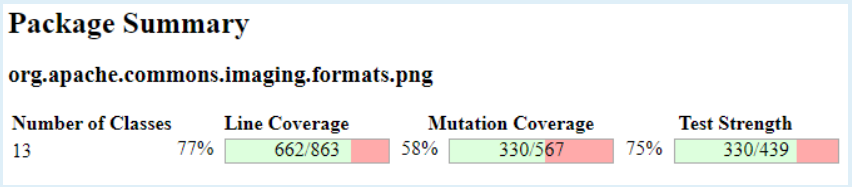

Fig. 4. PNG package coverage

Name	Line Coverage	Mutation Coverage	Test Strength
AbstractScanExpediter.java	88% (68/77)	88% (38/43)	88% (38/43)
BitParser.java	73% (19/26)	29% (11/38)	44% (11/25)
PngWriter.java	83% (224/270)	69% (154/222)	83% (154/185)

Table 2. Line Coverage, Mutation Coverage, and Test Strength for specified Java classes.

In this case we have better result for test strenght, the main difference with the previous package is the better mututation coverage result, which tell us that the package is more robust against mutuations.

JMH

JMH is a Java harness for building, running, and analysing nano/micro/milli/macro benchmarks written in Java and other languages targeting the JVM.

Due to strictly specifications, it was difficult to find a suitable class in our project for this testing. Our choice ended up with a method from Format Compliance class. This class is a utility for checking and reporting format compliance of a data source to an image format, with capabilities for logging, debugging, and throwing exceptions based on compliance checks.

```
@State(Scope.Benchmark)
@BenchmarkMode(Mode.Throughput)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
@Fork(3)
public class FormatCompliance {
```

Fig. 5. Format Compliance class

```java
@Benchmark
@Warmup(iterations = 5, time = 10, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 10, timeUnit = TimeUnit.SECONDS)
public void dump() {
    try (StringWriter sw = new StringWriter(); PrintWriter pw = new PrintWriter(sw)) {
        dump(pw);
        pw.flush();
        sw.flush();
        LOGGER.fine(sw.toString());
    } catch (final IOException e) {
        LOGGER.log(Level.SEVERE, e.getMessage(), e);
    }
}
```

Fig. 6. Tested method

The **dump()** method is responsible for generating a formatted representation of the format compliance information, flushing the writers, and logging the result.

Annotations. Here some description of the annotations used for the test:

- **@State** indicates that instances of this class will be used as benchmark state, and it will be managed by JMH.
- **Scope.Benchmark** specifies that the state should be shared among all threads in a benchmark
- **@BenchmarkMode(Mode.Throughput)** indicates that the benchmark should measure throughput, which is the number of operations per unit of time
- **@Fork(3)** indicates that the benchmark will be executed three times in separate JVM instances
- **@Warmup(iterations = 5, time = 10, timeUnit = TimeUnit.SECONDS)** This JMH annotation specifies the warm-up phase for the benchmark
- **@Measurement(iterations = 5, time = 10, timeUnit = TimeUnit.SECONDS)** This JMH annotation specifies the measurement phase for the benchmark
- **@Benchmark** indicates that the annotated method is a benchmark to be executed and measured

```
# Run progress: 33,33% complete, ETA 00:03:21
# Fork: 2 of 3
# Warmup Iteration   1: 4841,318 ops/ms
# Warmup Iteration   2: 4880,469 ops/ms
# Warmup Iteration   3: 4898,943 ops/ms
# Warmup Iteration   4: 4881,367 ops/ms
# Warmup Iteration   5: 4897,674 ops/ms
Iteration   1: 4892,063 ops/ms
Iteration   2:
```

Fig. 7. JMH Execution

```
Benchmark                Mode  Cnt     Score     Error   Units
FormatCompliance.dump   thrpt   15  4799,590 ± 126,973  ops/ms
```

Fig. 8. JMH Result

class modifications

```
3 usages
private /*final*/ boolean failOnError = false;
3 usages
private /*final*/ String description = "";
new *
public FormatCompliance(){

}
```

Fig. 9. Class modification to make the class suitable

Randoop

Randoop is a unit test generator for Java. It automatically creates unit tests for classes, in JUnit format. Randoop generates unit tests using feedback-directed random test generation. This technique pseudo-randomly, generates sequences of method/constructor invocations for the classes under test. Randoop executes the sequences it creates, using the results of the execution to create assertions that capture the behavior of your program. It creates tests from the code sequences and assertions.

Some definitions.

Container:

Definition: A container refers to a test element that contains other tests or containers.
Usage: Containers can be classes, packages, or other logical groupings of tests. They serve as organizational units for grouping and executing multiple tests.

Test:

Definition: A test refers to a specific method within a class that is generated by Randoop to exercise and explore the behavior of the corresponding class.
Usage: Randoop generates tests by invoking methods on the classes under test, exploring different code paths, and capturing the behavior of the classes. Each generated test is aimed at achieving high code coverage and revealing potential corner cases and bugs.

The test campaign

The campaign involves:

- PaletteFactory class
- ColorSpaceSubset class

- MostPopulatedBoxesMedianCut class
- several classes from formats.png package

Palette Factory

As mentioned earlier, the **PaletteFactory** class serves as a crucial computational consumer within the **palette** package and is notably one of the most resource-intensive classes across the entire library. For this reason, Randoop generated a significant number of tests.

```
Test run finished after 256 ms
[         7 containers found      ]
[         0 containers skipped    ]
[         7 containers started    ]
[         0 containers aborted    ]
[         7 containers successful ]
[         0 containers failed     ]
[      1241 tests found           ]
[         0 tests skipped         ]
[      1241 tests started         ]
[         0 tests aborted         ]
[      1241 tests successful      ]
[         0 tests failed          ]
```

Fig. 10. PaletteFactory generated tests result

ColorSpaceSubset

For the ColorSpaceSubset class a new behavior has been found. The tool will not create the tests because of the nature of the class:

Randoop for Java version "4.3.2, local changes, branch master, commit df17bc8, 2023-01-08". Cannot instantiate non-accessible class

The message above indicates that Randoop is encountering an issue while trying to generate tests for the class.
The reason of the message is the **accessibility of the class**
The ColorSpaceSubset class is a package-private (default access) class within the org.apache.commons.imaging.palette package.

MostPopulatedBoxesMedianCut

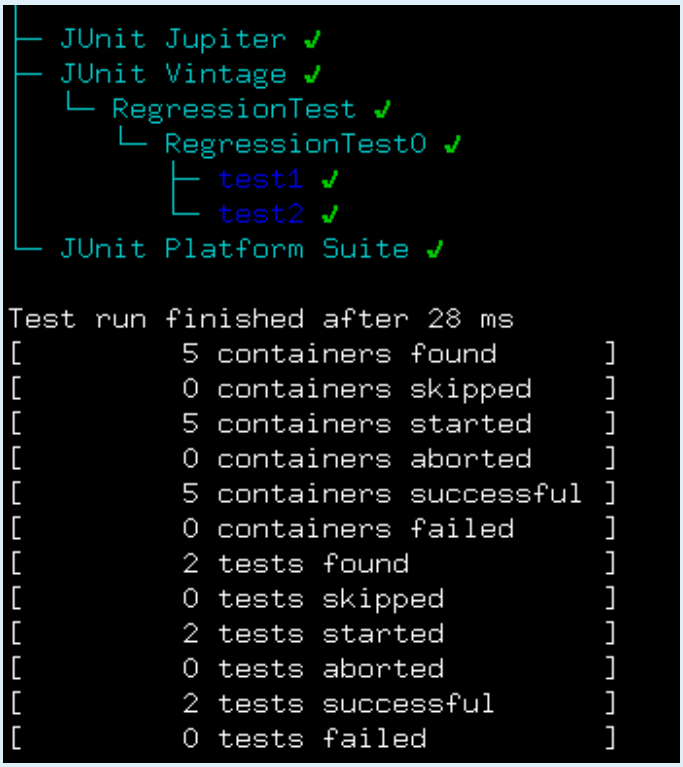

Fig. 11. MPBMC generated tests result

As we can see from the image above, there are few tests for the *MostPopulatedBoxesMedianCut* class, due to fact that implements two simple methods.

formats.png package

In this final case we decided to try a different approach in the generation of test cases, in particular we replaced –testclass="mypackage.myclass" with –classlist="classes.txt". In this way we can write in a text file all the classes for which we want the tests to be generated and test them all in once.

```
org.apache.commons.imaging.formats.png.AbstractPngText
org.apache.commons.imaging.formats.png.GammaCorrection
org.apache.commons.imaging.formats.png.PhysicalScale
org.apache.commons.imaging.formats.png.PngImagingParameters
org.apache.commons.imaging.formats.png.PngWriter
org.apache.commons.imaging.formats.png.PngColorType
org.apache.commons.imaging.formats.png.BitParser
org.apache.commons.imaging.formats.png.ChunkType
```

Fig. 12. png package list of classes

```
Test run finished after 153 ms
[         7 containers found      ]
[         0 containers skipped    ]
[         7 containers started    ]
[         0 containers aborted    ]
[         7 containers successful ]
[         0 containers failed     ]
[      1130 tests found           ]
[         0 tests skipped         ]
[      1130 tests started         ]
[         0 tests aborted         ]
[      1130 tests successful      ]
[         0 tests failed          ]
```

Fig. 13. PNG package result

The primary advantage of utilizing the –classlist option is its efficiency as a time-saving specification. However, it comes with the limitation that we lack specific information about individual classes within the provided list. Consequently, it is most effective when our objective is to conduct general tests to uncover errors. In scenarios where issues arise with a specific set of classes, such as a package, it is recommended to transition to using –testclass on those specific classes for more targeted testing. As we can see in the figure below, also "classlist" specifies classes that cannot be instantiated.

```
Randoop for Java version "4.3.2, local changes, branch master, commit df17bc8, 2
023-01-08".
Cannot instantiate non-accessible class org.apache.commons.imaging.formats.png.B
itParser specified via --testclass or --classlist.

Will try to generate tests for 7 out of 8 classes.
```

Fig. 14. private class warning

Find Security Bugs

FindSecBugs is a java tool, that automatically find security bugs in Java code.

Summary

Warning Type	Number
Bad practice Warnings	1
Correctness Warnings	20
Malicious code vulnerability Warnings	25
Total	**46**

Fig. 15. FindSecBugs Report

Below, the details of the security bugs found by the plugin.

EI_EXPOSE_REP: May expose internal representation by returning reference to mutable object

Returning a reference to a mutable object value stored in one of the object's fields exposes the internal representation of the object. If instances are accessed by untrusted code, and unchecked changes to the mutable object would compromise security or other important properties, it will need to do something different. Returning a new copy of the object is a better approach in many situations.

EI_EXPOSE_REP2: May expose internal representation by incorporating reference to mutable object

This code stores a reference to an externally mutable object into the internal representation of the object. If instances are accessed by untrusted code, and unchecked changes to the mutable object would compromise security or other important properties, it will need to do something different. Storing a copy of the object is a better approach in many situations.

RCN_REDUNDANT_NULLCHECK_WOULD_HAVE_BEEN_A_NPE: Nullcheck of value previously dereferenced

A value is checked here to see whether it is null, but this value can't be null because it was previously dereferenced, and if it were null, a null pointer exception would have occurred at the earlier dereference. Essentially, this code and the previous dereference disagree as to whether this value is allowed to be null. Either the check is redundant or the previous dereference is erroneous.

SE_COMPARATOR_SHOULD_BE_SERIALIZABLE: Comparator doesn't implement Serializable

This class implements the Comparator interface. Should consider whether or not it should also implement the Serializable interface. If a comparator is used to construct an ordered collection such as a TreeMap, then the TreeMap will be serializable only if the comparator is also serializable. As most comparators have little or no state, making them serializable is generally easy and good defensive programming.

Warning type	Number
Bad Practice Warnings	1
Correctness Warnings	20
Malicious code vulnerability Warnings	1

As we can see we managed only malicious code vulnerability (24/25) that corresponded to the following warnings:

- EI_EXPOSE_REP
- EI_EXPOSE_REP2

This warnings were solved returning or incorporating, the copy of the requested object. Since a large part were arrays, we used the built-in function *Arrays.copyof(array, lenght)*.

Not solved warnings. **RCN_REDUNDANT_NULLCHECK_WOULD_HAVE_BEEN_A_NPE** and **SE_COMPARATOR_SHOULD_BE_SERIALIZABLE** were not been solved because:

- RCN_REDUNDANT_NULLCHECK_WOULD_HAVE_BEEN_A_NPE: The second check comes from the built-in function used in the methods.
- SE_COMPARATOR_SHOULD_BE_SERIALIZABLE: Its a good practice to make a class that implements comparator, serializable but in this case it would have no advantages.

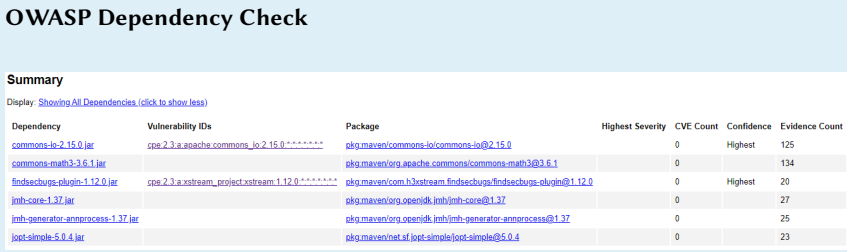

Fig. 16. OWASP Dependency Check

If we look at either Vulnerability IDs that we got from the dependency check, we get the result below:

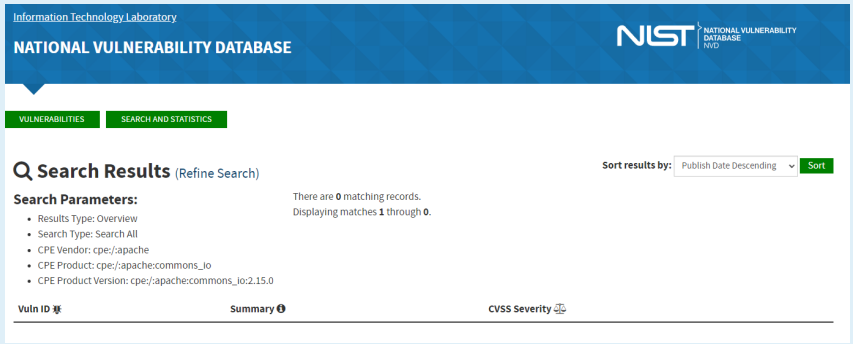

Fig. 17. apache:commons_io:2.15.0 details