Geophys. J. Int. (0000) **000**, 000–000

Apache Commons Imaging

Francesco Pagano

4 January 2025

1 INTRODUCTION

Apache Commons Imaging is a Java library designed for working with image files and is a part of the Apache Commons project. The library provides tools to read and write various image formats, including JPEG, PNG, BMP, GIF, TIFF, and more. It is available on GitHub at the following link: https://github.com/apache/commons-imaging Apache Commons imaging key features are:

- **Pure Java implementation**: the library is written entirely in Java, making it more portable compared to typical image libraries, while still performing reasonably well.
- **Basic Image Manipulation**: the library offers some image manipulation capabilities (e.g., resizing, cropping, color adjustments).
- **Image Metadata Handling**: provides easy access to images metadata.

The Commons Imaging project size is 31k LOC (lines of code) with 448 classes. In the project are also provided configuration files to build, compile, test and package the project with GitHub Actions. The GitHub repository that I created and that includes my modifications to the Apache Commons Imaging library is available at the following link: https://github.com/Francesco-Pagano/commons-imaging.

2 SOFTWARE QUALITY ANALYSIS

A software quality analysis was performed by scanning the project's Github repository using the SonarCloud tool. To make it work, I set the CI/CD with GitHub actions so that when particular action in GitHub is executed (like push, pull requests and so on), GitHub performs an automatic building of the project. The analysis uncovered **139 bugs** and **2.2k code smells** in the project. These bugs were classified as 62 critical and 77 minor problems and they all referred to the reliability of the code, instead all the code smells referred to the maintainability.

Sonarcloud identified 9 critical bugs instances of ArrayIndexOutOfBoundsException. To address these bugs, I used a conditional operation to make sure that the array was not empty before trying to get elements from it. The others 52 critical bugs were references to FieldType. The issues reported were that a parent class was referencing a static member of a subclass during initialization. This could lead to a different behavior from the one expected from the program if the order of the static class members was changed. I analyzed the code responsible for these issues and determined that a change in the order of the class members would not cause a different behavior because there was no dependency between them.

Sonarcloud identified 71 minor bugs instances of usage of a shift operator that was shifting by 0 positions. I considered these as *false positives* because even though shifting by 0 does not do anything it improves readability of the code when other shifting operations are performed on the same line of code. The remaining 6 minor bugs were related to the casting of operands and they were fixed by adding the appropriate cast.

Many medium code smells referred to commented-out code left in different classes. Commented-out code distracts the focus from the actual executed code. It creates a noise that increases maintenance code. And because it is never executed, it quickly becomes out of date and invalid. I removed this code also because it could be retrieved from source control history if required.

Some minor code smells referred to superfluous exceptions within throws clauses that could have negative effects on the readability and maintainability of the code. In these cases the exceptions were superfluous because they were subclasses of another listed exception. I solved these code smells by removing the superfluous exceptions. Others minor code smells were about the use of the "public" modifier for test classes and methods. I solved these issues by removing the public modifier.

At the end of the software quality analysis process, I solved all of the 77 minor bugs, 12 critical bugs and about 1200 code smells.

3 DOCKER

Docker is used to develop, ship, and run applications. Docker enables you to separate your applications from your infrastructure so that you can deliver software quickly. To Dockerize my maven project, I created a *Dockerfile*:

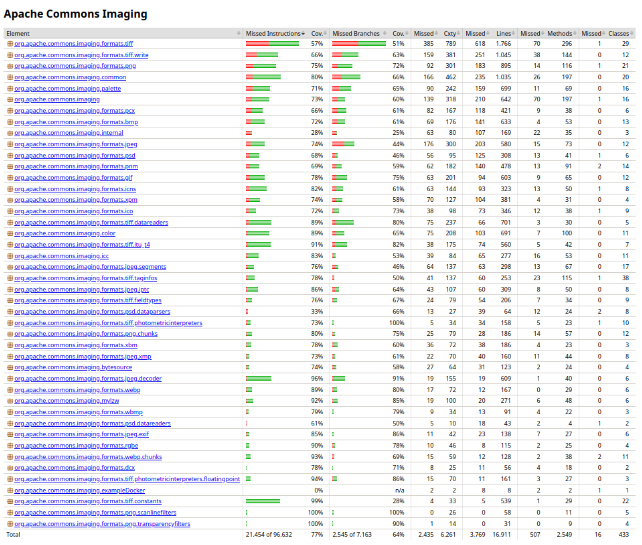
$$FROM\ openjdk:8$$
$$ADD\ target/example-docker.jarexample-docker.jar$$
$$ENTRYPOINT\ [''java'','' -jar'',''example-docker.jar'']$$
$$EXPOSE\ 8080$$

In this case, I used the OpenJDK 8 official docker image as the base image on which I built my image.
In the new package *exampleDocker*, I created 4 java classes: *ExampleDocker*, *ExampleDockerHandler*, *ExampleImageHandler*, *ExampleDockerTagHandler*.
In *ExampleDockerHandler* an image is taken from the website https://thispersondoesnotexist.com and saved in jpg format, initially without metadata (*ExampleDockerImagewithoutMetadata*).
In *ExampleDockerTagHandler* with the method *setExifGPSTag*, metadata relating to the position are added in a new image (*ExampleDockerImagewithMetadata*), such as latitude, longitude, and so on.
In *ExampleDocker* I created an HTML page exposed on port 8080, where with *metadataExample* method in *ExampleDockerHandler*,

every time the get-image button is pressed the latest image taken from the website is shown with the related metadata add thanks to the *setExifGPSTag* method.
This docker image is available on https://hub.docker.com/r/francescopagano45/example-docker.jar

4 CODE COVERAGE ANALYSIS

JaCoCo is a free code coverage library for Java. In my project JaCoCo was already in the pom.xml, so I simply generated the reports with the *mvn verify* command.
There are 44 packages, and the instructions coverage is 77% while the branches coverage is 64% (Figure 1).

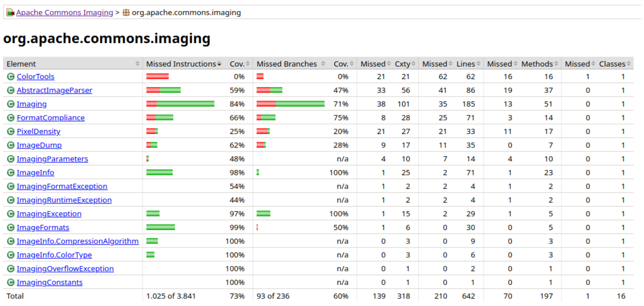


Apache Commons Imaging

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.apache.commons.imaging.formats.tiff		57%		51%	385	789	618	1.766	70	296	1	29
org.apache.commons.imaging.formats.tiff.write		66%		63%	159	381	251	1.045	38	144	0	12
org.apache.commons.imaging.formats.png		75%		72%	92	301	183	895	14	116	1	21
org.apache.commons.imaging.common		80%		66%	166	462	235	1.035	26	197	0	20
org.apache.commons.imaging.palette		71%		65%	90	242	159	699	11	69	0	16
org.apache.commons.imaging		73%		60%	139	318	210	642	70	197	1	16
org.apache.commons.imaging.formats.pcx		66%		61%	82	167	118	421	9	38	0	6
org.apache.commons.imaging.formats.bmp		72%		61%	69	176	141	633	4	53	0	13
org.apache.commons.imaging.internal		28%		25%	63	80	107	169	22	35	0	3
org.apache.commons.imaging.formats.jpeg		74%		44%	176	300	203	580	15	73	0	12
org.apache.commons.imaging.formats.psd		68%		46%	56	95	125	308	13	41	1	6
org.apache.commons.imaging.formats.pnm		69%		59%	62	182	140	478	13	91	2	14
org.apache.commons.imaging.formats.gif		78%		75%	63	201	94	603	9	65	0	12
org.apache.commons.imaging.formats.icns		82%		61%	63	144	93	323	13	50	1	8
org.apache.commons.imaging.formats.xpm		74%		58%	70	127	104	381	4	31	0	4
org.apache.commons.imaging.formats.ico		72%		73%	38	98	73	346	12	38	1	9
org.apache.commons.imaging.formats.tiff.datareaders		89%		80%	75	237	66	701	3	30	0	5
org.apache.commons.imaging.color		89%		65%	75	208	103	691	7	100	0	11
org.apache.commons.imaging.formats.tiff.itu_t4		91%		82%	38	175	74	560	5	42	0	7
org.apache.commons.imaging.icc		83%		53%	39	84	65	277	16	53	0	11
org.apache.commons.imaging.formats.jpeg.segments		76%		46%	64	137	63	298	13	67	0	17
org.apache.commons.imaging.formats.tiff.taginfos		78%		50%	41	137	60	253	23	115	1	38
org.apache.commons.imaging.formats.jpeg.iptc		86%		64%	43	107	60	309	8	50	0	8
org.apache.commons.imaging.formats.tiff.fieldtypes		76%		67%	24	79	54	206	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		73%		100%	5	34	34	158	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	186	4	23	0	3
org.apache.commons.imaging.formats.jpeg.xmp		73%		61%	22	70	40	160	11	44	0	8
org.apache.commons.imaging.bytesource		74%		58%	27	64	31	123	2	24	0	4
org.apache.commons.imaging.formats.jpeg.decoder		96%		91%	19	155	19	609	1	40	0	6
org.apache.commons.imaging.formats.webp		89%		80%	17	72	12	167	0	29	0	6
org.apache.commons.imaging.mylzw		92%		85%	19	100	20	271	6	48	0	6
org.apache.commons.imaging.formats.wbmp		79%		79%	9	34	13	91	4	22	0	3
org.apache.commons.imaging.formats.psd.datareaders		61%		50%	5	10	18	43	2	4	1	2
org.apache.commons.imaging.formats.jpeg.exif		85%		86%	11	42	23	138	7	27	0	6
org.apache.commons.imaging.formats.rgbe		90%		78%	10	46	8	115	2	25	0	4
org.apache.commons.imaging.formats.webp.chunks		93%		69%	15	59	12	128	2	38	2	11
org.apache.commons.imaging.formats.dcx		78%		71%	8	25	11	56	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.exampleDocker		0%		n/a	2	2	8	8	2	2	1	1
org.apache.commons.imaging.formats.tiff.constants		99%		28%	4	33	5	539	1	29	0	22
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	21.454 of 96.632	77%	2.545 of 7.163	64%	2.435	6.261	3.769	16.911	507	2.549	16	433

Figure 1: JaCoCo - package coverage

Taking the *org.apache.commons.imaging* package as an example you can see the code coverage of the individual classes, here the instructions coverage is 73% while the branches coverage is 60% (Figure 2).



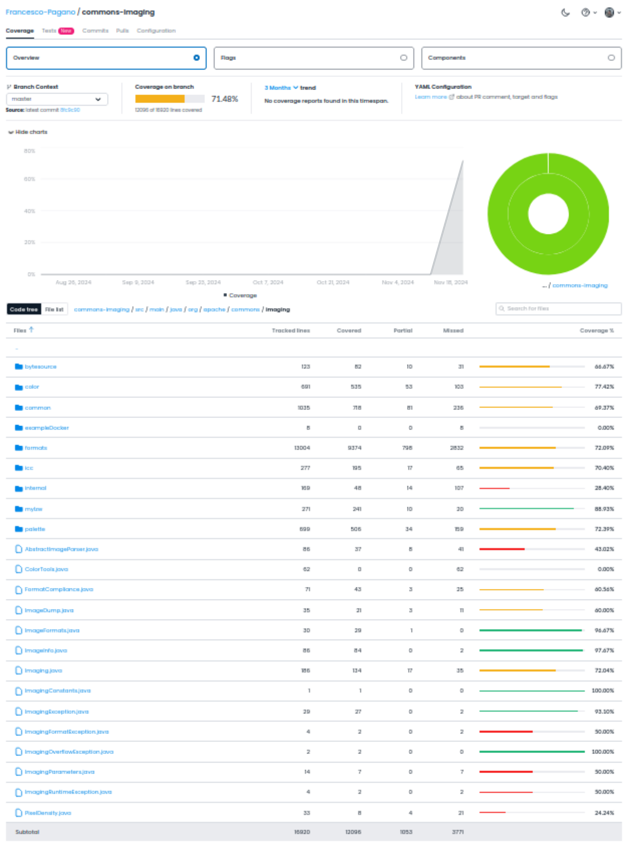
Apache Commons Imaging > org.apache.commons.imaging

org.apache.commons.imaging

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
ColorTools		0%		0%	21	21	62	62	16	16	1	1
AbstractImageParser		59%		47%	33	56	41	86	19	37	0	1
Imaging		84%		71%	38	101	35	185	13	51	0	1
FormatCompliance		66%		75%	8	28	25	71	3	14	0	1
PixelDensity		25%		20%	21	27	21	33	11	17	0	1
ImageDump		62%		28%	9	17	11	35	0	7	0	1
ImagingParameters		48%		n/a	4	10	7	14	4	10	0	1
ImageInfo		98%		100%	1	25	2	71	1	23	0	1
ImagingFormatException		54%		n/a	1	2	2	4	1	2	0	1
ImagingRuntimeException		44%		n/a	1	2	2	4	1	2	0	1
ImagingException		97%		100%	1	15	2	29	1	5	0	1
ImageFormats		99%		50%	1	6	0	30	0	5	0	1
ImageInfo.CompressionAlgorithm		100%		n/a	0	3	0	9	0	3	0	1
ImageInfo.ColorType		100%		n/a	0	3	0	6	0	3	0	1
ImagingOverflowException		100%		n/a	0	1	0	2	0	1	0	1
ImagingConstants		100%		n/a	0	1	0	1	0	1	0	1
Total	1.025 of 3.841	73%	93 of 236	60%	139	318	210	642	70	197	1	16

Figure 2: JaCoCo - classes coverage

Another tool to measure the coverage is Codecov.
When a push is made to GitHub, Codecov automatically calculates the project's code coverage.
The coverage on branch is 71.48%, 12096 of 16920 lines covered (Figure 3).



Francesco-Pagano / commons-imaging

Branch Context: master
Source: latest commit 8fc9c90

Coverage on branch 71.48%
12096 of 16920 lines covered

3 Months trend
No coverage reports found in this timespan.

YAML Configuration
Learn more about PR comment, target and flags

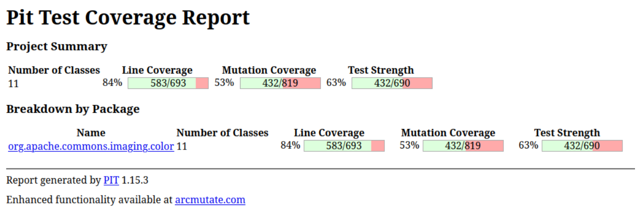
commons-imaging / src / main / java / org / apache / commons / imaging

Files	Tracked lines	Covered	Partial	Missed	Coverage %
..					
bytesource	123	82	10	31	66.67%
color	691	535	53	103	77.42%
common	1035	718	81	236	69.37%
exampleDocker	8	0	0	8	0.00%
formats	13004	9374	798	2832	72.09%
icc	277	195	17	65	70.40%
internal	169	48	14	107	28.40%
mylzw	271	241	10	20	88.93%
palette	699	506	34	159	72.39%
AbstractImageParser.java	86	37	8	41	43.02%
ColorTools.java	62	0	0	62	0.00%
FormatCompliance.java	71	43	3	25	60.56%
ImageDump.java	35	21	3	11	60.00%
ImageFormats.java	30	29	1	0	96.67%
ImageInfo.java	86	84	0	2	97.67%
Imaging.java	186	134	17	35	72.04%
ImagingConstants.java	1	1	0	0	100.00%
ImagingException.java	29	27	0	2	93.10%
ImagingFormatException.java	4	2	0	2	50.00%
ImagingOverflowException.java	2	2	0	0	100.00%
ImagingParameters.java	14	7	0	7	50.00%
ImagingRuntimeException.java	4	2	0	2	50.00%
PixelDensity.java	33	8	4	21	24.24%
Subtotal	16920	12096	1053	3771	

Figure 3: Codecov coverage

5 MUTATION TESTING

I used PiTest for the mutation testing.
During my test I used the default configuration.
For simplicity I selected only one package, *org.apache.commons.imaging.color*, as *targetTest* (Figure 4).



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
11	84%	583/693	53%	432/819	63%	432/690

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
org.apache.commons.imaging.color	11	84%	583/693	53%	432/819	63%	432/690

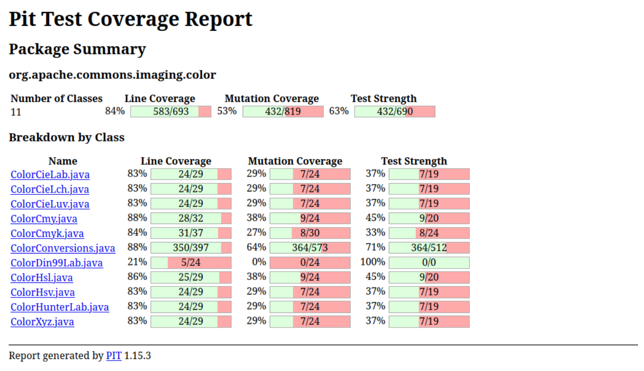
Report generated by PIT 1.15.3

Enhanced functionality available at arcmutate.com

Figure 4: PITest - package coverage

- Line Coverage = 84% (583/693)
- Mutation Coverage = 53% (432/819)
- Test Strength = 63% (432/690)

In this package, there are 11 classes.
In Figure 5 you can see the results for the individual classes in more detail.



Pit Test Coverage Report

Package Summary

org.apache.commons.imaging.color

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
11	84%	583/693	53%	432/819	63%	432/690

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
ColorCieLab.java	83%	24/29	29%	7/24	37%	7/19
ColorCieLch.java	83%	24/29	29%	7/24	37%	7/19
ColorCieLuv.java	83%	24/29	29%	7/24	37%	7/19
ColorCmy.java	88%	28/32	38%	9/24	45%	9/20
ColorCmyk.java	84%	31/37	27%	8/30	33%	8/24
ColorConversions.java	88%	350/397	64%	364/573	71%	364/512
ColorDin99Lab.java	21%	5/24	0%	0/24	100%	0/0
ColorHsl.java	86%	25/29	38%	9/24	45%	9/20
ColorHsv.java	83%	24/29	29%	7/24	37%	7/19
ColorHunterLab.java	83%	24/29	29%	7/24	37%	7/19
ColorXyz.java	83%	24/29	29%	7/24	37%	7/19

Report generated by PIT 1.15.3

Figure 5: PITest - classes coverage

6 AUTOMATIC TEST CASE GENERATION

6.1 Randoop

Randoop is a tool, written in Java, that automatically generates *unit tests*.
I followed the instructions from this GitHub repository: https://github.com/emaiannone/tools-tutorial/tree/master/randoop.
So, I downloaded the jar file "*randoop-all-4.3.2.jar*" and then I used the command:

- *java -cp randoop-all-4.3.2.jar:commons-imaging/target/classes randoop.main.Main gentests –testclass=org.apache.commons.imaging.color.ColorConversions –time-limit=20 –junit-output-dir=randoop-tests*

4 classes named *RegressionTest.java*, *RegressionTest0.java*, *RegressionTest1.java*, *RegressionTest2.java* were generated with default configurations and 1389 test cases for ColorConversions.java.

Then to compile them, I used the command:

- *javac $(find randoop-tests -name "*.java") -cp commons-imaging/target/classes/:randoop-all-4.3.2.jar:commons-imaging/target/dependency/junit-jupiter-5.9.1.jar:commons-imaging/target/dependency/hamcrest-2.2.jar*

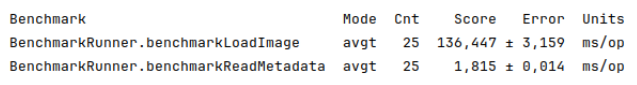
Finally I used the test suite with this command:

- *java -cp randoop-tests:commons-imaging/target/classes:randoop-all-4.3.2.jar:commons-imaging/target/dependency/junit-platform-console-standalone-1.9.2.jar org.junit.platform.console.ConsoleLauncher –scan-class-path*

All the 1389 tests passed successfully.

7 PERFORMANCE TESTING

I used *Java MicroBenchmark Harness* (**JMH**) which is an annotation-based framework that enables control over the execution of test cases. I created a new class *BenchmarkRunner*, in which I added two benchmark tests: one to load an image (*benchmarkLoadImage*) and one to read the metadata of an image (*benchmarkReadMetadata*). I tested the average time of execution of each test case, with the annotation @BenchmarkMode(Mode.AverageTime).
Five forks were performed, each of five iterations, for a total of 25.
In *benchmarkLoadImage* the score is 136,447, and the error is 3,159.
The times are: *min* = 133,097 - *avg* = 136,447 - *max* = 151,574 ms/op.
In *benchmarkReadMetadata* the score is 1,815, and the error is 0,014.
The times are: *min* = 1,778 - *avg* = 1,815 - *max* = 1,845 ms/op.



```
Benchmark                            Mode  Cnt    Score    Error  Units
BenchmarkRunner.benchmarkLoadImage   avgt   25  136,447 ± 3,159  ms/op
BenchmarkRunner.benchmarkReadMetadata avgt   25    1,815 ± 0,014  ms/op
```

Figure 6: JMH - Benchmark results

8 SOFTWARE VULNERABILITIES

To conduct the static security analysis of the project, I used the tools SpotBugs and OWASP DC.

8.1 SpotBugs - Find Security Bugs

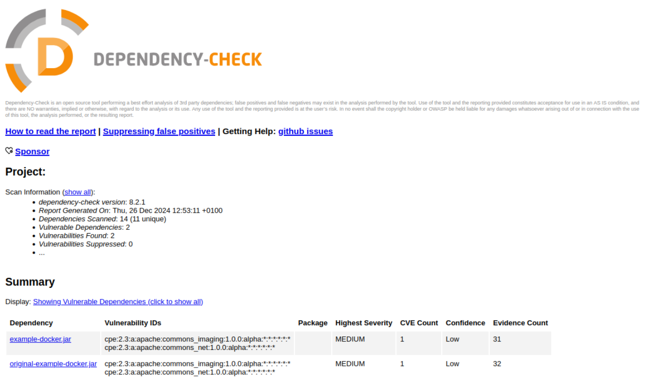
I followed the instructions from this Github repository: https://github.com/emaiannone/tools-tutorial/tree/master/findsecbugs.
There are 128129 lines of code analyzed, in 2762 classes, in 159 packages.

- *High Priority Warnings* are 19, Density = 0.15
- *Medium Priority Warnings* are 95, Density = 0.74
- There are in total 114 Security Warnings, Density = 0.89

*Density means defects per Thousand lines of non-commenting source statements.

8.2 OWASP DC

I followed the instructions from this Github repository: https://github.com/emaiannone/tools-tutorial/tree/master/owaspdc.



Dependency-Check is an open source tool performing a best effort analysis of 3rd party dependencies; false positives and false negatives may exist in the analysis performed by the tool. Use of the tool and the reporting provided constitutes acceptance for use in an AS IS condition, and there are NO warranties, implied or otherwise, with regard to the analysis or its use. Any use of the tool and the reporting provided is at the user's risk. In no event shall the copyright holder or OWASP be held liable for any damages whatsoever arising out of or in connection with the use of this tool, the analysis performed, or the resulting report.

How to read the report | Suppressing false positives | Getting Help: github issues

Sponsor

Project:

Scan Information (show all):

- *dependency-check version*: 8.2.1
- *Report Generated On*: Thu, 26 Dec 2024 12:53:11 +0100
- *Dependencies Scanned*: 14 (11 unique)
- *Vulnerable Dependencies*: 2
- *Vulnerabilities Found*: 2
- *Vulnerabilities Suppressed*: 0
- ...

Summary

Display: Showing Vulnerable Dependencies (click to show all)

Dependency	Vulnerability IDs	Package	Highest Severity	CVE Count	Confidence	Evidence Count
example-docker.jar	cpe:2.3:a:apache:commons_imaging:1.0.0:alpha:*:*:*:*:* cpe:2.3:a:apache:commons_net:1.0.0:alpha:*:*:*:*:*		MEDIUM	1	Low	31
original-example-docker.jar	cpe:2.3:a:apache:commons_imaging:1.0.0:alpha:*:*:*:*:* cpe:2.3:a:apache:commons_net:1.0.0:alpha:*:*:*:*:*		MEDIUM	1	Low	32

Figure 7: Dependency-Check results

With Dependency-Check I found two vulnerabilities, both medium severity, low confidence and both refer to the .jar I created for docker: *example-docker.jar* and *original-example-docker.jar*