Report Software Dependability

APACHE COMMONS-EMAIL LIBRARY

Baggiano Antonio | 23 Giugno 2023

Supervised By:
Prof. Dario Di Nucci
A.A. 2022/2023

1. Introduction

The project I have chosen for this report is "Commons Email" from the "Apache Foundation" that is a public Repository on GitHub and reachable by this Link: https://github.com/apache/commons-email . The repository is well-maintained by the authors and various contributors, accumulating 1150 commits over the course of a decade. It is also interesting to note the presence of 82 public forks of the project, which attests to its importance within the community.

The Apache Commons Email library is a Java library that simplifies sending emails programmatically. It provides a range of features for creating, sending, and managing emails, such as adding attachments, setting senders and recipients, formatting text, and utilizing SMTP servers. In short, the Apache Commons Email library streamlines the integration of email sending in Java applications, reducing code complexity and offering a straightforward interface to use.

Throughout the duration of the analysis and experiments I conducted, I used Java 11 for ARM64 architecture on the MacOS operating system with an M1 Pro processor. Inside Docker, I virtualized containers based on the AMD64 architecture.

The first step I took was to fork the project into my GitHub. My personal repository is accessible at the following link: https://github.com/AntonioKill98/commons-email-mod . Once the fork was done, I focused on preliminary analyses.

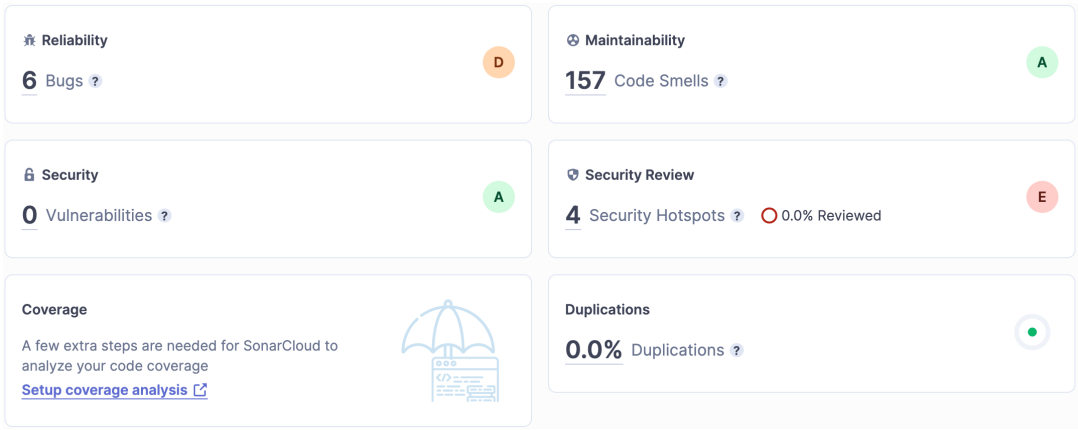
2. Build the Project

After downloading the Project locally using the "GitHub Desktop" application, I started by executing the initial packaging using the command “mvn clean package”, and the project compiled successfully right away.
Immediately after that, I had to deactivate the Maven R.A.T. (Release Audit Tool) plugin, as it would have caused errors with any further code modifications I would make. To do this, I made changes to the "pom.xml" file and in the section of the R.A.T. plugin, I added the "<skip>true</skip>" tag, which causes the plugin to skip the verification process. After that, I proceeded to perform an initial surface analysis using SonarCloud.

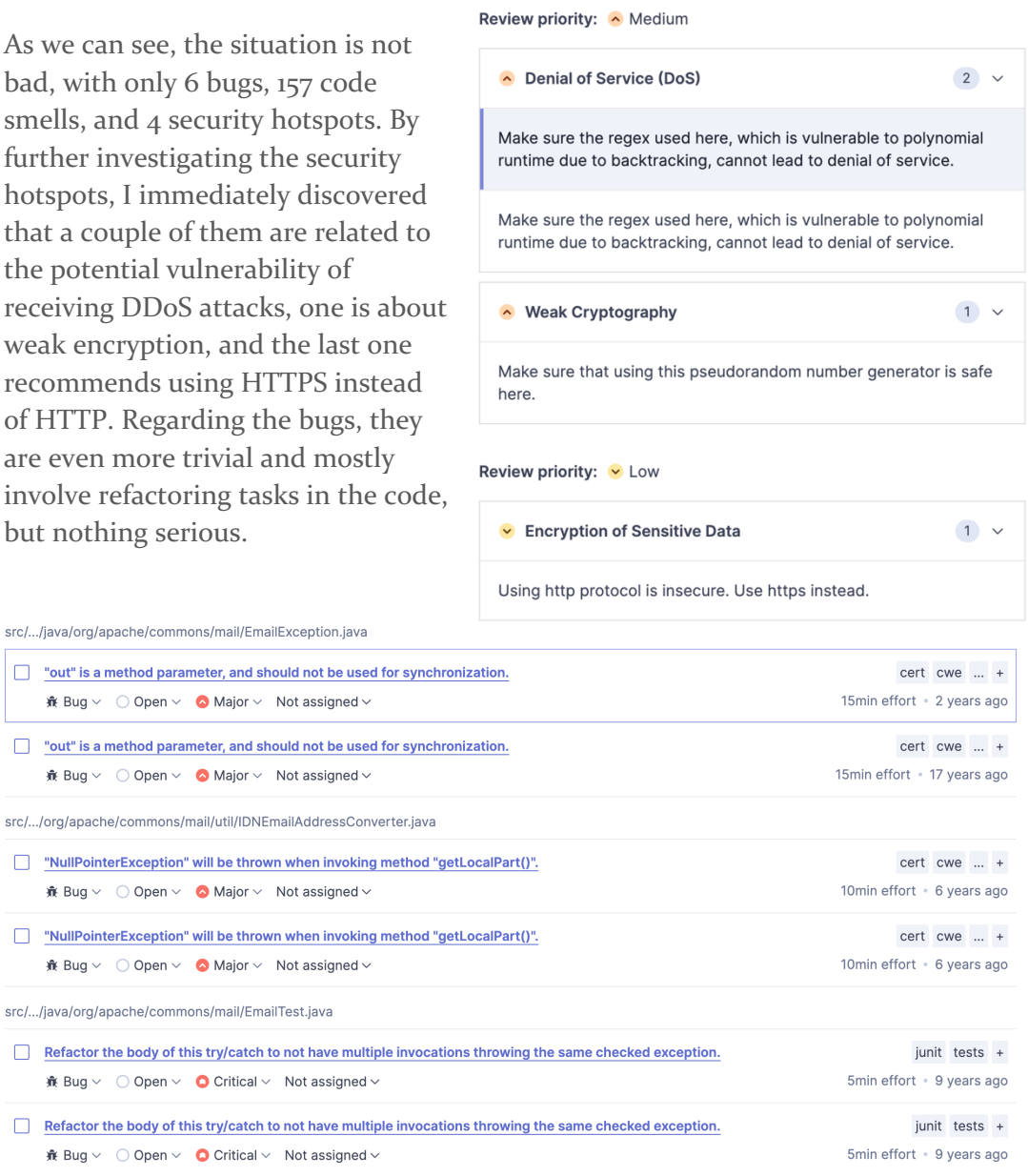
3. Sonar Cloud

I imported the project into SonarCloud and performed an initial analysis. The initial analysis revealed the following situation Out of a total of 3.3K lines of code, 2.7K lines are written in Java, and 511 lines belong to XML files. The analysis reports the following

situation:



As we can see, the situation is not bad, with only 6 bugs, 157 code smells, and 4 security hotspots. By further investigating the security hotspots, I immediately discovered that a couple of them are related to the potential vulnerability of receiving DDoS attacks, one is about weak encryption, and the last one recommends using HTTPS instead of HTTP. Regarding the bugs, they are even more trivial and mostly involve refactoring tasks in the code, but nothing serious.



4. Docker and Docker-Hub

The next step was to containerize my project using Docker. First, I attempted local containerization by creating a "Dockerfile" that used a Maven-based image with Java 11 to execute the command "mvn clean package", than I launched the process by the command line with “docker build -t commons-email-mod .” :

```
#Scelta dell'immagine openjdk-11 e Maven
FROM maven:3.8.4-openjdk-11

# Copia il codice sorgente del tuo progetto nella directory /usr/src/app all'interno dell'immagine
COPY . /usr/src/app

# Imposta la directory di lavoro
WORKDIR /usr/src/app

# Esegui il comando Maven per compilare e confezionare il tuo progetto
RUN mvn clean package
```

Once I ensured that it was working locally, I moved to GitHub, where I configured an action that would update an image of my project on DockerHub with every commit to the master branch. The workflow in question is saved in a file named "docker-publish.yml" located in the ".github/workflows" folder.

```
name: DockerHub - Aggiornamento

on:
  push:
    branches: [ master ]

jobs:
  publish:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout code
      uses: actions/checkout@v2

    - name: Login to DockerHub
      uses: docker/login-action@v1
      with:
        username: ${{ secrets.DOCKERHUB_USERNAME }}
        password: ${{ secrets.DOCKERHUB_TOKEN }}

    - name: Build and Publish Docker image
      uses: docker/build-push-action@v2
      with:
        context: .
        push: true
        tags: antoniob98/commons-email-mod:latest
```

The publicly available image on DockerHub can be accessed at the following link: https://hub.docker.com/r/antoniob98/commons-email-mod

3. Code Coverage: JaCoCo

I computed the code coverage of the project using JaCoCo, which generated a report folder in the project's "target" directory. I also copied this folder to the project's main directory to publish it on GitHub under the name "JaCoCo_Report_V1". The situation is as follows:

Apache Commons Email

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.apache.commons.mail		69%		65%	135	347	244	824	39	170	1	11
org.apache.commons.mail.resolver		81%		74%	18	57	20	104	5	30	0	5
org.apache.commons.mail.util		86%		75%	17	73	16	126	2	38	0	3
Total	1.160 of 4.268	72%	154 of 478	67%	170	477	280	1.054	46	238	1	19

As we can observe, the overall code coverage is not bad at all, but it can be improved. In particular, we can see that "org.apache.commons.mail.HtmlEmail" has a coverage of 53%, and "org.apache.commons.mail.MultiPartEmail" has a coverage of 54%. These areas can certainly be improved:

org.apache.commons.mail

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
HtmlEmail		53%		60%	22	41	56	140	3	12	0	1
Email		84%		74%	47	169	58	339	13	83	0	1
MultiPartEmail		54%		62%	18	39	48	126	6	23	0	1
ByteArrayDataSource		0%		0%	16	16	48	48	9	9	1	1
EmailUtils		83%		63%	18	39	8	67	0	9	0	1
EmailException		24%		n/a	4	7	14	20	4	7	0	1
HtmlEmail.InlineImage		34%		0%	6	8	9	15	4	6	0	1
ImageHtmlEmail		91%		66%	4	12	3	40	0	6	0	1
EmailAttachment		100%		n/a	0	11	0	20	0	11	0	1
SimpleEmail		100%		100%	0	3	0	5	0	2	0	1
DefaultAuthenticator		100%		n/a	0	2	0	4	0	2	0	1
Total	1.007 of 3.270	69%	123 of 354	65%	135	347	244	824	39	170	1	11

I avoided running another coverage test with Cobertura because it proved to be incompatible with Java 11. To save time and avoid constantly switching between Java versions 8 and 11, I directly moved on to Mutation Testing.

4. PiTest

With PiTest, I performed Mutation Testing to uncover how the test suite of the project handles mutations in the source code.

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
18	79%	846/1070	62%	344/553	74%	344/466

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
org.apache.commons.mail	10	78%	648/832	55%	230/415	68%	230/339
org.apache.commons.mail.resolver	5	81%	84/104	90%	44/49	94%	44/47
org.apache.commons.mail.util	3	85%	114/134	79%	70/89	88%	70/80

Using PiTest has once again confirmed that in some classes, there are few or no test cases, and I could significantly improve the quantity and quality of the tests. We will see later how to do that.

org.apache.commons.mail

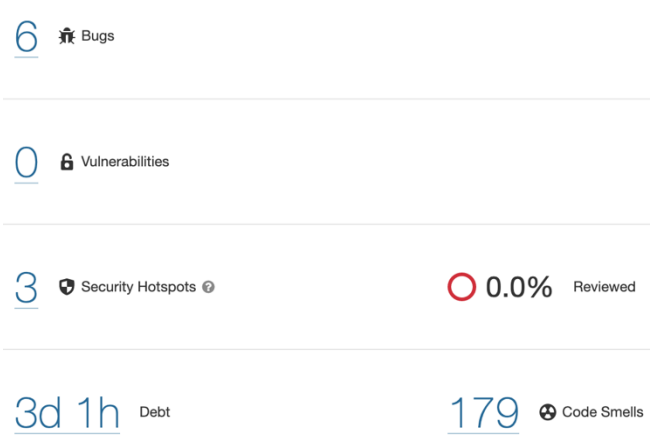
Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
10	78%	648/832	55%	230/415	68%	230/339

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
ByteArrayDataSource.java	0%	0/51	0%	0/23	0%	0/0
DefaultAuthenticator.java	100%	4/4	100%	1/1	100%	1/1
Email.java	83%	282/339	48%	96/199	57%	96/167
EmailAttachment.java	100%	20/20	100%	5/5	100%	5/5
EmailException.java	30%	6/20	0%	0/4	0%	0/0
EmailUtils.java	86%	59/69	71%	32/45	84%	32/38
HtmlEmail.java	82%	128/157	63%	44/70	71%	44/62
ImageHtmlEmail.java	93%	37/40	73%	8/11	73%	8/11
MultiPartEmail.java	84%	107/127	78%	42/54	81%	42/52
SimpleEmail.java	100%	5/5	67%	2/3	67%	2/3

5. SonarQube and EcoCode

I started scanning the project with SonarQube, which immediately detected the same 3 Security Hotspot and the same 6 bugs detected by SonarCloud, but it found additional code smells (22 more) for a total of 179. Now it's time for some code refactoring to fix some of the bugs and perform a new scan with SonarQube.



a. The first and second bugs were in "src/main/java/org/apache/commons/mail/EmailException.java" and they were related to using a parameter as the lock object in `synchronized()` which should not happen because one thread could modify the parameter while another thread is using it as a lock object, resulting in two different values for the same parameter. These two bugs are then fixed by removing the `synchronized()` block from the method and making the entire method `synchronized`.
b. The second and third bugs where located in the class "src/.../java/org/apache/commons/mail/util/IDNEmailAddressConverter.java" and involve the possibility of a NullPointerException occurring within the code. By analyzing the code, I noticed that there was simply a possibility, in case the input String value `eMail` was `null`, that the `getLocalPart()` function could be called with a `null` input string, resulting in an exception. The fix was to add an if statement that, in the case where the "email" parameter is null, skips the function execution and returns null, just as the original code would have done.
c. The last two bugs were reported in the class "src/test/java/org/apache/commons/mail/EmailTest.java," a test class. The bug involves the possibility of calling the same exception multiple times within a try-catch block, and they should be "separated." I then attempted to divide those instructions into individual try-catch blocks, effectively fixing the bug.

Regarding the 3 Security Hotspot:

a. The first two are related to the potential for receiving DDoS attacks through the class "src/main/java/org/apache/commons/mail/ImageHtmlEmail.java", which, according to SonarQube, uses a method from the "Pattern" object (Pattern.compile()) that can be susceptible to such attacks due to backtracking. After performing some tests and unsuccessful attempts to refactor the code, I have decided to leave these two Security Hotspots unchanged for now.
b. The last Security Hotspot was related to the usage of an object from the `java.util.Random` class within the `EmailUtils.java` class located at

`src/main/java/org/apache/commons/mail/`. It has been promptly replaced with an object from the `java.security.SecureRandom` class.

The refactor, as it has been done, resolved 6 bugs and 1 Security Hotspot. However, there are 4 new Code Smells, but we don't need to worry about them since they only concern the commented code in the classes that I left to track the changes I was making.

After that, I started analyzing the code with the EcoCode plugin, and the number of Code Smells increased to 371. I have identified mainly four types of CodeSmells:

a. Incrementing variables, using "++i" instead of "i++".
b. Attempting to use Switch Case instead of sequential if statements.
c. Replacing forEach loops with regular for loops using a counter variable.
d. Usage of global variables.

I have corrected several of these errors in the classes "src/main/java/org/apache/commons/mail/EmailUtils.java" and "src/main/java/org/apache/commons/mail/ByteArrayDataSource.java". However, I had to leave some of them as they were because the optimal solution to comply with EcoCode rules would have introduced additional CodeSmells or, even worse, worsened the code optimization. For example, removing certain global variables would have required copying their values to all the places where they were referenced, which wouldn't have made sense. Regarding the if statements, I received several notifications that couldn't be corrected using switch cases, so I left them as they were.

6. Performance Testing – Java Microbenchmark Harness

Regarding the use of JMH for conducting small benchmarks on classes and methods of the library, I started by adding the JMH dependencies to version 1.35 in the pom.xml file. Then, I created a class named jmhBenchmark in the Test section of the project. Each benchmark is annotated with @Benchmark to indicate that it should be executed by JMH. Additionally, appropriate annotations like @BenchmarkMode, @Fork, @Warmup, and @Measurement are used to specify the desired benchmark mode, the number of the forks, warmup iterations, and measurement iterations. These benchmarks provide valuable insights into the performance characteristics of the library's classes and methods, helping to identify areas for optimization or potential bottlenecks. I have configured five benchmarks:

a. benchGetSetHTMLMsg(): This benchmark, using the HtmlEmailTest test class, performs the composition and retrieval of a Mockup HTML eMail. The measurement is conducted in Mode.Throughput, which measures the number of iterations per unit of time.
b. benchGetMsg(): Through the SimpleEmailTest class, we conduct a benchmark for the writing and retrieval of a plain text test email. The measurement is conducted in Mode.Throughput, which measures the number of iterations per unit of time.

c. benchSendHtmlMail(): Using the HtmlEmailTest class, we benchmark the process of composing and sending an HTML email (via a fake SMTP server). The measurement is performed in Mode.AverageTime, which calculates the average time in seconds for a single iteration.
d. benchSendSimpleMail(): Using the `SimpleEmailTest` class, we benchmark the process of composing and sending a simple email (via a fake SMTP server). The measurement is performed in Mode.AverageTime, which calculates the average time in seconds for a single iteration.
e. benchSendMultiImageMail(): Using the `ImageHtmlEmailTest` class, we benchmark the process of composing and sending an HTML email that includes images. The measurement is performed in Mode.AverageTime, which calculates the average time in seconds for a single iteration.

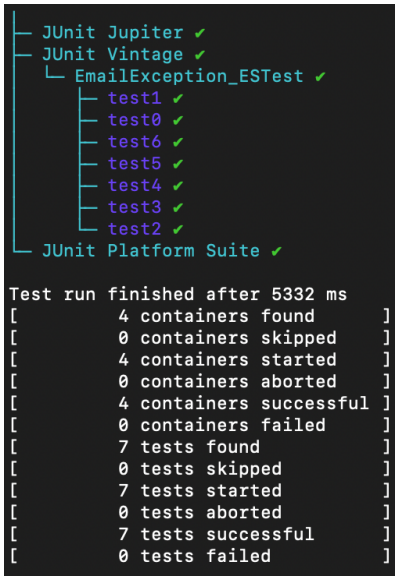
After creating the class containing the benchmarks, I proceeded to create another class, located in `test/util`, called `BenchmarkRunner`, which does exactly what its name suggests. It is an executable class that initiates the execution of all the benchmarks.
After saving everything and running `mvn clean package`, I found the executable class directly in the `target` folder. After the execution, which lasted 11 minutes, I obtained the following results:

```
Benchmark                                  Mode  Cnt       Score       Error  Units
jmhBenchmark.benchGetMsg                  thrpt   15  677724,910 ± 13890,582  ops/s
jmhBenchmark.benchGetSetHTMLMsg           thrpt   15  725663,482 ±  6579,832  ops/s
jmhBenchmark.benchSendHtmlMail             avgt    5       0,236 ±     0,019   s/op
jmhBenchmark.benchSendMultiImageMail       avgt    5       0,285 ±     0,019   s/op
jmhBenchmark.benchSendSimpleMail           avgt    5       0,217 ±     1,398   s/op

Process finished with exit code 0
```

7. TestCase Generation: EvoSuite and Randoop

I used Evosuite to attempt to achieve full code coverage of the library. I started by constructing test cases for the class org.apache.commons.mail.EmailException. I generated two test classes, which I then compiled and executed. The tests I have generated are saved in the 'evosuite-tests_V1' directory published on GitHub. With these new tests, I was able to achieve a coverage of 93% for the class.



Then I tried to generate test cases for the 'HtmlEmail' and 'MultiPartEmail' classes, but I was unsuccessful and kept encountering an error regarding the missing 'javax.EmailException' class. So, I switched to automatic generation using the

EvoSuite plugin for Maven. After declaring it in the pom.xml file and running 'mvn clean install', I was able to invoke it with 'mvn evosuite:generate'. It produced a '.evosuite' directory, which I renamed to 'EvoSuitePlugin_Output' to publish it on the project's GitHub. However, I faced issues executing these new tests, as the 'mvn test' phase always resulted in an error. Upon investigation, I concluded that the error is due to a conflicting dependency. The 'com.atlassian.jira:jira-rest-java-client-plugin' plugin requires Hamcrest 1.3, while the 'com.atlassian.jira:jira-rest-java-client-api' plugin requires Hamcrest 2.2. This conflict causes the tests to crash. Since I don't have advanced knowledge of Maven, I postponed resolving this issue in case of future updates.

8. FindSecBugs

With FindSecBugs, I performed static code analysis in order to identify possible security vulnerabilities in the source code. After downloading and compiling the source code of FindSecBugs, I ran it and here are the results: In total, 638,104 lines of code were analyzed in 15,065 classes and 877 packages. A total of 651 warnings were found, including 68 high-priority warnings and the rest being low-priority warnings.

Specifically, the higher-priority errors relate to the use of outdated encryption algorithms (such as MD5) within the library and the deserialization of certain objects in class methods.

A file containing the complete report named "reportFindSecBugs.html" has been added to the GitHub repository.

9. OWASP DependencyCheck

I performed a code analysis using OWASP DependencyCheck to ensure that the dependencies in my project were also secure. After cloning it from GitHub and compiling it, I executed the tool, and the result was an HTML report that I uploaded to the GitHub repository with the name "report_DependencyCheck.html".
As can be seen in the report, the only potential security issues highlighted are related to the executables of EvoSuite and Randoop, which I have left in the project directory.

10. The End

In this semester, during the Software Dependability course, we have learned, both theoretically and practically, how to use various tools that are necessary to obtain not only functional programs but also well-written and well-structured ones. We have learned how to use methods for team collaboration and individual contributions to a project. Additionally, we have explored how to analyze the security aspects of a project and test its quality. This project is specifically designed to assess our understanding of these concepts and our ability to apply them.