GLOBALRAIN

Artemis Financial Vulnerability Assessment Report

Table of Contents

Document Revision History

Version	Date	Author	Comments
1.0	11/11/2023	Eduardo Rodrigues	CS-305-H7011

Client

ARTEMIS FINANCIAL

Instructions

Submit this completed vulnerability assessment report. Replace the bracketed text with the relevant information. In the report, identify your findings of security vulnerabilities and provide recommendations for the next steps to remedy the issues you have found.

- Respond to the five steps outlined below and include your findings.
- Respond using your own words. You may also choose to include images or supporting materials. If you include them, make certain to insert them in all the relevant locations in the document.
- Refer to the Project One Guidelines and Rubric for more detailed instructions about each section of the template.

Developer: Eduardo Rodrigues

1. Interpreting Client Needs

Artemis Financial wants to provide their RESTful web API with state-of-the-art security features.

As web facing API in the Finance Sector, it was to meet various security requirements and industry standards in security and laws governing it, foreign and domestic.

As a Java API using Spring Boot and its Various dependencies; it must be under some form of developer support and maintenance to be as secure as possible to minimize the attack service and optimize the means for mitigation. Threats include theft, infiltration, and terrorism.

As a modern system, it needs to be maintainable, up to date, able to be periodically checked for vulnerability, constantly monitored, and observe the upmost level of visibility and traceability.

2. Areas of Security

The assessment of the web app will focus on input validation, APIs used within Spring Boot and dependencies, the embedded server being used (Apache Tomcat) via Spring and error handling there in, and the Java code used thus far. We will also take note of threats that may be introduced via cryptography employed with out-of-date software libraries and remedies for bad coding practices and patterns using Java.

3. Manual Review

The current web API gives users access to the endpoints "/greeting?name=SomeUser" and "/read?business_name=SomeBussiness" There is no form of authentication or authorization that governs them, and their content is not validated (Note that the current code does implement any queries yet that need parameterizing. Any queries will need to do so.). The code that governs these endpoints is also outdated and has substantially more documented vulnerabilities than the most up to date distributions available. (as per mvnrepository.com) These problems can be addressed with the use of exceptions that set a harness for access control mechanisms. The use of the long data type in the Greeting class can also be used to monitor for viral resource utilization that could signal malicious activity.

The code on its face shows adequate decoupling and encapsulation and access control being used in the class entities (POJOs) implemented in Java. The proper Data Object Access is in place, making it easier to work up positive patterns that aim to prevent runtime errors and tentative SQL injections and attacks at large. We can use built-in validation and query parameterization to gain more control and visibility of the potential attack surfaces. Cases in test should also use a non-root level account, even in the test environment, in order to coordinate development with the actual database implementation, permission settings being taken into consideration. The current DocData Class uses a connection that logs in as root. This will not reflect the most secure implementation of database moving into the deployment (Root access is only used to configure the system, not access it). With respect to the UI, access to the API can benefit from proper authentication and authorization implementations to restrict usage to a particular userbase. Additional consideration should also be given to the verification of the number of query results and the database configuration at large.

Supplemental to code and database concerns, the current Apache Tomcat configuration reveals itself to users should we encounter a runtime error. This is remedied in the server configuration via the Spring YAML configuration. A custom Whitelabel error page is a simple solution that can keep can conceal the software to the unauthorized and unauthenticated. Further control of the endpoints and messages they display needs to be under scrutiny moving forward. (see https://www.baeldung.com/spring-boot-custom-error-page)

Other vulnerabilities can be easily avoided by upgrading the project dependencies while leaning into a robust security plan on deployment of the system and use of monitoring tools. The code base itself leaves ample room to implement industry standard security practices. The remaining undeveloped code should also note TODO style comments in order for the team to start assimilating positive security patterns. The only incorrigible vulnerability lies with our current host environment and use of the X509LDAPCertStoreSpi.java class should it be implemented.

4. Static Testing

The OWASP dependency check shows us close to 100 CVE given the current fitting. These include packages for the web server itself, Apache Tomcat, Spring itself, the JSON parser, Jackson, and two packages that have to do with logging, Maven, and databases. Updating to the latest Spring Boor distribution only leaves us with one vulnerability to consider, 2023 CVE 33201 with regards to X509LDAPCertStoreSpi.java class in Bouncy Castel's bcprov-jdk15on-1.46.jar.

The Bouncy Castle Vulnerabilities are remedied by upgrading to the most current, stable, and maintained implementation. The Java version used to compile binaries and various host environments and tools must also be considered when accessing the proper workaround or remedies. The latest bcprov-jdk15, 1.70 still maintains that its' use with the LDAP certificate store is at risk, patching or workarounds are needed as per the specific implementation. Alternative and/or custom solutions should be considered when for this use case.

Spring itself advises using a supported version and upgrading according to their instructions. As of 2023, they advise to upgrade to 3.0.7+ or 2.7.12+. Upgrading to the latest Spring dependencies should update the following libraries notwithstanding the persistent vulnerability found with Bouncy Castle bcprov-jdk15 that is also a sub dependency of Spring, regardless of its explicit declaration in the Maven POM. (see CVE-2023-20883: Spring Boot Welcome Page DoS Vulnerability, spring.io/security/cve-2023-20883)

The various logging libraries, logback and log4j require upgrades before implementation given that patches could impact how code needs to be worked into the solution. Later respective versions, 1.4.x and 2.21.x don't have any reported vulnerabilities. The most recent version of SnakeYAML has no reported vulnerabilities. Updates and patches may be vendor and software specific (as in use of NetApp products). Jackson Databind also shows no vulnerabilities from 2.13.4.2 and on. Tomcat-embedded-core has been patched as of 11.0.0-M11 and on. There are numerous version dependent vulnerabilities between 9 and 11, an upgrade to the latest version of the server is paramount to operational security. Hibernate Validator Engine has been stable as of version 8.0.x. (see mvnrepository.com)

5. Mitigation Plan

Development moving forward should leave comments with respect to implementation of unused code and especially directed at the database harness and respective implementation, making clear distinctions between production integration and test harnesses or temporary code.

Particular attention needs to be directed towards validation of all forms of untrusted user input. The userbase can be isolated with access control; and user input will be filtered using the best practices for parameterization and validation to avoid the threat of injection-based vulnerabilities.

Additional points of hardening that are an issue is the database configuration, in particular database permissions. The Spring configuration that dictates the embedded server also needs to be wired for security. Developing in this fashion will make sure that the code is prepared for change and ready for deployment.