****

# Artemis Financial Vulnerability Assessment Report

Table of Contents

[Document Revision History 3](#_Toc32574607)

[Client 3](#_Toc32574608)

[Instructions 3](#_Toc32574609)

[Developer 4](#_Toc32574610)

[1. Interpreting Client Needs 4](#_Toc32574611)

[2. Areas of Security 4](#_Toc32574612)

[3. Manual Review 4](#_Toc32574613)

[4. Static Testing 4](#_Toc32574614)

[5. Mitigation Plan 4](#_Toc32574615)

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **01/28/2024** | **Pierrot Ngimbidi** |  |

## Client



## 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

Pierrot Ngimbidi

## Interpreting Client Needs

The value of secure communications to Artemis Financial cannot be overstated. In the finance sector, where sensitive client data and large monetary transactions are commonplace, secure communication ensures the confidentiality and integrity of data. It's not just a matter of protecting client information; it's also about maintaining the company's reputation. Trust is a cornerstone in finance, and secure communications are crucial to building and maintaining this trust. Any breach or lapse in communication security can lead to significant financial losses and erosion of client confidence, which can be devastating for a business in this sector.

Artemis Financial's engagement in international transactions adds another layer of complexity to its operations. Such transactions increase the exposure to cyber threats and bring various international financial regulations and compliance requirements into play. The company must navigate different financial laws and data protection regulations, like the General Data Protection Regulation (GDPR) in Europe. Secure and reliable handling of international transactions is crucial to avoid legal penalties and maintain smooth cross-border financial operations.

Governmental restrictions on secure communications are another critical area for Artemis Financial to consider. Different countries have varied regulations regarding data protection, encryption, and privacy. This becomes particularly pertinent if the company handles international transactions or operates in multiple jurisdictions. Compliance with these regulations is not optional; it’s a necessity. Non-compliance can lead to hefty fines, legal issues, and potential damage to the company's reputation.

External threats, such as cyber-attacks, are an ever-present and evolving challenge. Threats like phishing, ransomware, and advanced persistent threats (APTs) are becoming more sophisticated. Additionally, as the financial sector increasingly relies on digital technologies, the risk of data breaches and unauthorized access to sensitive financial information grows. Artemis Financial must be prepared for today's threats and emerging threats in the immediate future, adapting its security strategies to stay ahead of malicious actors.

Modernization requirements for Artemis Financial include integrating and utilizing open-source libraries and evolving web application technologies. Open-source libraries can offer powerful tools and frameworks for developing robust web applications. However, they also require careful management to avoid introducing vulnerabilities. Staying abreast of evolving web technologies is equally important. Technologies such as cloud computing, API integrations, and advanced encryption methods are becoming staples in modern web applications. Implementing these technologies securely is vital for Artemis Financial to enhance efficiency, improve client service, and maintain a competitive edge in the financial industry.

## Areas of Security

Based on the Vulnerability Assessment Process Flow Diagram and considering the functionality of Artemis Financial’s web application, let's document the relevant areas of security:

**Architecture Review**

Artemis Financial’s application architecture forms the backbone of its security posture. A review will identify how components like servers, databases, and applications interact and where vulnerabilities may exist. Given the sensitive nature of financial data, ensuring that the architecture supports strong security measures is crucial.

**Input Validation**

Secure input validation is essential to prevent common web vulnerabilities like SQL injection, which can be particularly detrimental in a financial context where data integrity is paramount. Ensuring that inputs are properly validated and sanitized protects the application from malicious data that could compromise the system.

**APIs**

Artemis Financial likely uses APIs for various functionalities, including transaction processing and data retrieval. Secure API interactions are vital to prevent unauthorized access and ensure that data exchanged through these APIs is protected, considering the sensitive financial information involved.

**Cryptography**

Cryptography is critical in protecting data in transit and at rest, especially in a financial application where sensitive information like account details and personal client data is handled. Understanding and addressing potential vulnerabilities in cryptographic implementations is essential to maintain data confidentiality and integrity.

**Client/Server**

In a client/server architecture, securing the communication between client applications (like web or mobile interfaces) and servers is crucial. This includes ensuring secure session management and protecting against threats like man-in-the-middle attacks, which are significant risks in financial applications.

**Code Error**

Secure code handling is important to prevent bugs that could lead to security vulnerabilities. Since financial applications are often targets for cybercriminals, minimizing coding errors can significantly reduce the risk of data breaches and other security incidents.

**Code Quality**

Adhering to secure coding practices and patterns is vital in a financial application to prevent vulnerabilities that attackers could exploit. High code quality is fundamental to ensure the application's reliability and security.

**Encapsulation**

Secure data structures and encapsulation are essential to protect sensitive data within the application. Proper encapsulation ensures that internal structures are not exposed and are only accessible through defined interfaces, which is crucial in a financial application handling confidential data.

## Manual Review

Based on the initial review of the code files, here are the vulnerabilities identified, categorized according to the seven security areas from the Vulnerability Assessment Process Flow Diagram:

**Architecture Review**

* **DocData.java**: The **DocData** class handles database connections directly, suggesting a possible lack of separation between data access and business logic layers. This can lead to security issues and make the code less maintainable.

**Input Validation**

* The method **CRUD** in **CRUDController** takes a parameter **name** without validating it. This could lead to security vulnerabilities like SQL injection if the input is not properly sanitized before being used in database queries.
* There is no visible input validation in the constructors or getter methods. This might lead to security issues if the **CRUD** class handles user input directly. Without proper validation, it could be vulnerable to injection attacks.
* Like **CRUDController**, input validation is missing for the **greeting** method's parameter.

**APIs**

* **CRUDController.java** & **GreetingController.java**: Both controllers expose methods as API endpoints without evident security measures (like authentication and input sanitization), potentially leading to unauthorized access or manipulation.

**Cryptography**

* There is no direct reference to using cryptography in the provided code snippets. However, given the nature of the application, it’s crucial to ensure that sensitive data, especially customer-related data, is encrypted at rest and in transit.

**Client/Server**

* The direct database connection in **DocData.java** could expose the server to risks if not properly managed. Ensuring secure communication between client and server, and between server and database, is critical.
* The method **read\_document** in the **DocData** class suggests a database connection, but it is commented out. If used, this approach to database connection is highly vulnerable, especially with hardcoded credentials (**"root","root"**). This could lead to SQL Injection attacks if user input (**key**, **value**) is not correctly sanitized before being used in a SQL query.

**Code Error**

* The method **read\_document** in **DocData** contains commented-out code and an empty **catch** block. This indicates potentially incomplete error handling, which could lead to unhandled exceptions and security vulnerabilities.

**Code Quality**

* The class name **customer** does not follow Java naming conventions, which should start with an uppercase letter. This can lead to confusion and potential errors in a larger codebase.
* The class **myDateTime** and its methods do not follow standard naming conventions and lack implementation, which could lead to confusion and errors in maintenance.

**Encapsulation**

* **customer.java**: The field **account\_balance** is not private, which violates the encapsulation principle. This could lead to unauthorized access or modification of the account balance.

## Static Testing

1. **bcprov-jdk15on-1.46.jar**
   * **Vulnerabilities**: 19 High Severity Vulnerabilities.
   * **Description**: Issues in Bouncy Castle Crypto Package.
   * **Solution**: Update to a later version of Bouncy Castle.
   * **Attribution**: Common vulnerabilities documented under various CVE IDs.
2. **hibernate-validator-6.0.18.Final.jar**
   * **Vulnerabilities**: 1 Medium Severity Vulnerability.
   * **Description**: A specific issue in Hibernate Validator.
   * **Solution**: Consider updating to a newer version of Hibernate Validator.
   * **Attribution**: Documented under specific CVE ID.
3. **jackson-databind-2.10.2.jar**
   * **Vulnerabilities**: 6 High Severity Vulnerabilities.
   * **Description**: Vulnerabilities in Jackson Databind, often related to deserialization issues.
   * **Solution**: Update Jackson Databind to a more secure version.
   * **Attribution**: Multiple CVE entries.
4. **log4j-api-2.12.1.jar**
   * **Vulnerabilities**: 1 Low Severity Vulnerability.
   * **Description**: Possible vulnerability in Apache Log4j.
   * **Solution**: Update to a newer version of Log4j.
   * **Attribution**: CVE entry for this specific version.
5. **logback-core-1.2.3.jar**
   * **Vulnerabilities**: 2 High Severity Vulnerabilities.
   * **Description**: Vulnerabilities in Logback Core library.
   * **Solution**: Update to a more recent and secure version.
   * **Attribution**: Documented in CVEs.
6. **snakeyaml-1.25.jar**
   * **Vulnerabilities**: 8 Critical Severity Vulnerabilities.
   * **Description**: Issues in the SnakeYAML library, potentially in YAML processing.
   * **Solution**: Update to a newer, secure version of SnakeYAML.
   * **Attribution**: Multiple CVEs detailing these vulnerabilities.
7. **spring-boot-2.2.4.RELEASE.jar** and related Spring jars (spring-boot-starter-web, spring-core, spring-web, spring-webmvc)
   * **Vulnerabilities**: Several Critical Severity Vulnerabilities.
   * **Description**: Vulnerabilities in various Spring components, often related to the framework's core functionalities.
   * **Solution**: Upgrade to a later, more secure version of Spring Boot and its related components.
   * **Attribution**: Documented across multiple CVEs.
8. **tomcat-embed-core-9.0.30.jar** and **tomcat-embed-websocket-9.0.30.jar**
   * **Vulnerabilities**: 26 and 27 Critical Severity Vulnerabilities, respectively.
   * **Description**: Multiple vulnerabilities in Apache Tomcat embedded versions.
   * **Solution**: Update to a newer version of Apache Tomcat.
   * **Attribution**: Extensively documented under various CVE IDs.

## Mitigation Plan

**Code Quality and Standards Compliance**

Objective: Improve code readability, maintainability, and adherence to Java best practices.

Actions: Refactor Class and Method Names: Update customer.java and myDateTime.java to follow Java naming conventions.

Implement Encapsulation: Modify customer.java to ensure all fields are private with public getters and setters.

**Secure Input Handling**

Objective: Prevent injection attacks and ensure the integrity of data inputs.

Actions: Input Validation in Controllers: In CRUDController.java and GreetingController.java, implement robust input validation using Java validation frameworks or regex.

**Database Access and Security**

Objective: Secure and optimize database interactions.

Actions: Implement DAO Pattern: Refactor DocData.java to use a Data Access Object (DAO) for database interactions.

Exception Handling for Database Operations: Improve error handling in DocData.java by adding meaningful catch blocks and logging exceptions.

**Dependency Management and Vulnerability Patching**

Objective: Address known vulnerabilities in third-party libraries and frameworks.

Actions: Update Vulnerable Dependencies: Upgrade all dependencies identified in the Maven Dependency-Check report to the latest, secure versions. Key updates include:

**Bouncy Castle (bcprov-jdk15on)**

**Hibernate Validator**

**Jackson Databind**

**Log4j API**

**Logback Core**

**SnakeYAML**

**Spring Boot and related components (core, web, webmvc)**

**Apache Tomcat (embed-core, embed-websocket)**