****

# 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** | **[Date]** | **[Your name]** |  |

## Client



## Developer

Jeffrey Flores

## Interpreting Client Needs

Artemis Financial is a consulting company specializing in financial planning that seeks to modernize its operations by leveraging custom software solutions. As part of this modernization effort, Artemis Financial recognizes the critical importance of software security to protect sensitive financial data and ensure the integrity and confidentiality of its services. I am a developer in Global Rain that has been tasked with providing expertise on securing Artemis Financials’ RESTful web application programming interface (API) against external threats.

Secure communications are invaluable to Artemis Financial as they handle sensitive financial information, including savings, retirement plans, investments, and insurance details. The secure communications protect the confidentiality and integrity of this data, fostering trust with clients and regulatory compliance.

Artemis Financial may also engage in international transactions, needing a robust security measure to safeguard against cross-border threats and comply with international data protection regulations. This is very important to safeguard their transactions that happen overseas.

Governmental restrictions on secure communications may vary depending on the jurisdictions Artemis Financial operates. Compliance with regulations such as GDPR, HIPAA, or PCI DSS may impose requirements on data security and communication protocols. External threats such as data breaches, DDoS attacks, SQL injection, and phishing attempts pose immediate risks to Artemis Financials’ web application. As cyber threats evolve, Artemis Financial must remain aware of emerging attacks and vulnerabilities.

Artemis Financial must carefully vet and manage the use of open-source libraries in its software stack to mitigate the risk of incorporating vulnerable or outdated dependencies. By getting the most updated and secure open-source libraries that are regularly updated Artemis can ensure its secure use of these libraries. Regular security audits and dependency scanning are essential to identify and remediate vulnerabilities in third-party components.

Artemis Financials’ modernization efforts will need to consider the latest web application technologies and best practices for security to stay within the standards of the market. This includes implementing robust authentication and authorization mechanisms, enforcing secure coding practices, and adopting frameworks with built-in security features. Artemis should use new technologies to combat any security vulnerabilities like tokenization and multi-factor authentication for secure access.

## Areas of Security

**Input Validation:** Given the sensitive nature of financial data handled by Artemis Financial, ensuring secure input validation is crucial to prevent injection attacks and data tampering. For this web application we have to make sure correct information is inputted to make sure the correct data is saved to each user’s profile.

**APIs**: Artemis Financial relies on APIs to interact with its systems and third-party services. Securing these API interactions is essential to protect sensitive data transmitted between different systems. Any open vulnerabilities in the API software used can be a potential risk to the entire web application.

**Client/Server**: Artemis Financials’ web application involves communication between clients and servers. Secure distributed composing is necessary to safeguard data transmission and ensure secure authentication and session management. All the sessions started between client and server need to be secured and managed by secure tokenization and encryption.

**Encapsulation**: Secure data structures help protect sensitive financial data from unauthorized access and manipulation. Proper encapsulation ensures that data is accessed and modified only through authorized means, reducing the risk of data breaches.

## Manual Review

Here are the identified vulnerabilities found in the code base:

**SQL Injection**: In the **DocData.java file**

The **read\_document** method directly uses user-supplied input (**key** and **value**) to construct a SQL query without proper validation or sanitization. This leaves the application vulnerable to SQL injection attacks.

**Unused Import Statement**: In the **GreetingController.java file**

The import statement **import java.util.concurrent.atomic.AtomicLong;** is present but not used in the **GreetingController** class, which can clutter the code and may confuse developers.

**Insecure Database Connection**: In the **DocData.java file**

The database connection is established using hard-coded credentials (**"root", "root"**) without any encryption or secure storage mechanisms. Storing credentials directly in the source code poses a security risk, especially if the code is shared or exposed.

**Potential Information Disclosure**: In the **CRUDController.java file**

The **CRUD** controller exposes sensitive information by returning an instance of **DocData** as a string in response to the **/read** endpoint. This may inadvertently disclose internal details or error messages, providing attackers with potentially useful information for attacking and gaining unauthorized access.

**Insufficient Input Validation**: In the **CRUDController.java file**

The **/read** endpoint accepts a **business\_name** parameter without validating or sanitizing it. Lack of input validation can lead to various security issues, including parameter tampering and injection attacks.

**Unused Class and Unused Variable**: In the **myDateTime.java** and **customer.java files**

Both **myDateTime** and **customer** classes appear to be unused in the provided code base, indicating potential dead code that should be removed to improve code maintainability and reduce attack surface.

**Potential Redundancy and Unused Constructor**: In the **CRUD.java file**

The **CRUD** class has two constructors, but the second constructor (**public CRUD(String content1, String content2)**) appears to be unused. This redundancy may lead to confusion and should be reviewed for possible elimination.

## Static Testing

[bcprov-jdk15on-1.46.jar](file:///Users/jeffrey/eclipse-workspace/rest-service/target/dependency-check-report.html#l2_991c96a4e31e6c19e2b9136c8955bd423f2dc4c7)

In the Bouncy Castle JCE Provider version 1.55 and earlier the ECIES implementation allowed the use of ECB mode. This mode is regarded as unsafe and support for it has been removed from the provider. In order to use this correctly the version would need to be updated.

[spring-boot-2.2.4.RELEASE.jar](file:///Users/jeffrey/eclipse-workspace/rest-service/target/dependency-check-report.html#l3_225a4fd31156c254e3bb92adb42ee8c6de812714)

In Spring Boot versions 3.0.0 - 3.0.6, 2.7.0 - 2.7.11, 2.6.0 - 2.6.14, 2.5.0 - 2.5.14 and older unsupported versions, there is potential for a denial-of-service (DoS) attack if Spring MVC is used together with a reverse proxy cache.

[logback-core-1.2.3.jar](file:///Users/jeffrey/eclipse-workspace/rest-service/target/dependency-check-report.html#l4_864344400c3d4d92dfeb0a305dc87d953677c03c)

A serialization vulnerability in logback receiver component part of logback version 1.4.11 allows an attacker to mount a Denial-Of-Service attack by sending poisoned data.

[log4j-api-2.12.1.jar](file:///Users/jeffrey/eclipse-workspace/rest-service/target/dependency-check-report.html#l5_a55e6d987f50a515c9260b0451b4fa217dc539cb)

Apache Log4j2 versions 2.0-beta7 through 2.17.0 (excluding security fix releases 2.3.2 and 2.12.4) are vulnerable to a remote code execution (RCE) attack when a configuration uses a JDBC Appender with a JNDI LDAP data source URI when an attacker has control of the target LDAP server. This issue is fixed by limiting JNDI data source names to the java protocol in Log4j2 versions 2.17.1, 2.12.4, and 2.3.2.

[snakeyaml-1.25.jar](file:///Users/jeffrey/eclipse-workspace/rest-service/target/dependency-check-report.html#l8_8b6e01ef661d8378ae6dd7b511a7f2a33fae1421)

SnakeYaml's Constructor() class does not restrict types which can be instantiated during deserialization. Deserializing yaml content provided by an attacker can lead to remote code execution. We recommend using SnakeYaml's SafeConsturctor when parsing untrusted content to restrict deserialization. recommend upgrading to version 2.0 and beyond.

[jackson-databind-2.10.2.jar](file:///Users/jeffrey/eclipse-workspace/rest-service/target/dependency-check-report.html#l9_0528de95f198afafbcfb0c09d2e43b6e0ea663ec)

jackson-databind through 2.15.2 allows attackers to cause a denial of service or other unspecified impact via a crafted object that uses cyclic dependencies. NOTE: the vendor's perspective is that this is not a valid vulnerability report, because the steps of constructing a cyclic data structure and trying to serialize it cannot be achieved by an external attacker.

[tomcat-embed-core-9.0.30.jar](file:///Users/jeffrey/eclipse-workspace/rest-service/target/dependency-check-report.html#l13_ad32909314fe2ba02cec036434c0addd19bcc580)

Generation of Error Message Containing Sensitive Information vulnerability in Apache Tomcat.This issue affects Apache Tomcat: from 8.5.7 through 8.5.63, from 9.0.0-M11 through 9.0.43. Users are recommended to upgrade to version 8.5.64 onwards or 9.0.44 onwards, which contain a fix for the issue.

[hibernate-validator-6.0.18.Final.jar](file:///Users/jeffrey/eclipse-workspace/rest-service/target/dependency-check-report.html#l16_7fd00bcd87e14b6ba66279282ef15efa30dd2492)

A flaw was found in Hibernate Validator version 6.1.2.Final. A bug in the message interpolation processor enables invalid EL expressions to be evaluated as if they were valid. This flaw allows attackers to bypass input sanitation (escaping, stripping) controls that developers may have put in place when handling user-controlled data in error messages

[spring-web-5.2.3.RELEASE.jar](file:///Users/jeffrey/eclipse-workspace/rest-service/target/dependency-check-report.html#l19_dd386a02e40b915ab400a3bf9f586d2dc4c0852c)

[spring-beans-5.2.3.RELEASE.jar](file:///Users/jeffrey/eclipse-workspace/rest-service/target/dependency-check-report.html#l20_0250c8c641433dc06b1b44e4563fa08a2fbf8954)

[spring-webmvc-5.2.3.RELEASE.jar](file:///Users/jeffrey/eclipse-workspace/rest-service/target/dependency-check-report.html#l21_745a62502023d2496b565b7fe102bb1ee229d6b7)

[spring-context-5.2.3.RELEASE.jar](file:///Users/jeffrey/eclipse-workspace/rest-service/target/dependency-check-report.html#l22_7750c95c96c7a1885c8b1b503ba915bc33ca579a)

[spring-expression-5.2.3.RELEASE.jar](file:///Users/jeffrey/eclipse-workspace/rest-service/target/dependency-check-report.html#l23_d0c6bb10758805b2153c589686b8045554bfac2d)

## Mitigation Plan

This would be my checklist for mitigation on each of the vulnerabilities above:

1. Update to version 1.56 or later to remove support for ECB mode, which is considered unsafe. Ensure that the application code explicitly specifies a secure mode of operation for encryption algorithms.
2. Upgrade to a supported version (3.0.7 or later, 2.7.12 or later, 2.6.15 or later, 2.5.15 or later) to address the vulnerability. Implement proper configuration and security measures in the reverse proxy cache to mitigate the risk of DoS attacks.
3. Upgrade to version 1.4.12 or later, which includes a fix for the serialization vulnerability. Ensure that deserialized data is properly validated and sanitized to prevent potential exploitation.
4. Upgrade to version 2.17.1, 2.12.4, or 2.3.2 to address the RCE vulnerability. Additionally, limit JNDI data source names to the java protocol to prevent exploitation of the vulnerability.
5. Upgrade to version 2.0 or later, which implements safe deserialization by restricting the types that can be instantiated during parsing untrusted content.
6. Upgrade to a later version (2.15.3 or later) or apply vendor-provided patches to address the DoS vulnerability.
7. Upgrade to version 8.5.64 or later for Apache Tomcat 8.5.x, or version 9.0.44 or later for Apache Tomcat 9.x, which contain a fix for the vulnerability.
8. Upgrade to a later version of Hibernate Validator that includes a fix for the vulnerability. Additionally, sanitize and validate user-controlled data used in error messages to prevent bypassing input sanitation controls.
9. For all the Spring Frameworks we can regularly update to the latest versions of Spring Framework dependencies to address any potential security vulnerabilities and follow best practices for secure development and deployment.