****

# 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** | **1/28/2024** | **Mikaela Spence** |  |

## 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

Mikaela Spence

## Interpreting Client Needs

Global Rain is a software design and development company that works with entrepreneurs, businesses, and government agencies around the world. The current project they are working on is with Artemis Financial that creates financial plans for customers including savings, retirement, and insurance plans. Global Rain has been hired by Artemis Financial to protect them from external threats. Secure communications are extremely important to this company as they handle sensitive personal and financial information for their customers. It is an international company, so different national and governmental considerations need to be considered. Their company is at significant risk from external threats since they are handling money, insurance information, and people’s personal information. They may be infiltrated to steal customer information and banking information such as routing numbers. Modernization for this company should include the use of open-source libraries.

## Areas of Security

1. **Input Validation**

Since input is being obtained from the user, input validation is extremely important. Input validation is a means of limiting input that an application will accept and can be done in a variety of ways (Manica & Detlefson, 2015). Input validation can include things like white or blacklisting, use of regular expressions, validating minimum and maximum input lengths, validating numerical or open text input, and URL validation (Manica & Detlefson, 2015). Sanitizing potentially hazardous input from users is also an important part of this step (OWASP, 2010).

1. **APIs**

Secure API interactions are also important in this scenario. APIs can assist with security authentication and authorization mechanisms. This is a way to ensure that users are only able to access parts of the application that they are authorized to use and requires authentication of the user. However, using built in APIs alone may not be enough to prevent a security attack (Manica & Detlefson, 2016). REST APIs can be built in the Spring Framework and help to build web-scale systems (Spring.io, 2024). REST APIs can build evolvable, backward compatible APIs and allow for scalable and securable services (Spring.io, 2024).

1. **Cryptography**

Cryptography is an important consideration for this scenario as well. Cryptography can help to protect sensitive information as it moves from one place to another. Cryptography is important for keeping information confidential and user verification (Manica & Detlefson, 2016). Using cryptography can also help with tracking information transfers, and is often important for governmental regulatory standards, especially when finances are involved, as they are with this case. Cryptography is important for the security and trust of the customers.

1. **Client/Server**

Client/server interactions are also a very important security concern for Artemis Financial. Ensuring the client/server interactions are secure are a means of protecting against DoS attacks (Manica & Detlefson, 2016). These considerations are also often important for regulatory compliance of governmental and financial agencies. Since this application will be dealing with sensitive and private information, ensuring the connections between the client and server are secure is vital.

1. **Code Error**

Secure error handling is another important aspect to consider. Secure error handling can help mitigate the risks from injection attacks (Manica & Detlefson, 2015). If errors are not handled correctly or defined correctly, hackers may be able to access sensitive data within the application (Manica & Detlefson, 2015).

1. **Code Quality**

Code quality should also be considered in this scenario. Ensuring that code is written with security in mind from the start of a project can prevent unnecessary delays later on, and can also create code that is less vulnerable to attacks (Manica & Detlefson, 2015). Using code that has been previously tested and approved is also a helpful way to ensure quality code, especially for common tasks (OWASP, 2010).

1. **Encapsulation**

Encapsulation is important to reduce the risk of SQL injection attacks. It is also useful in in the authorization and authentication aspect of the application, ensuring that users have the lowest level of privilege when accessing a database (OWASP, 2015). Instead of allowing any user access to all of the information, encapsulation is a means of keeping things in separate categories so that users are not able to access things they should not have access to. Modules are a means of hiding internal packages in an application, and strongly encapsulates classes and interfaces (ORACLE, 2023).

## Manual Review

**Input Validation**

1. In GreetingController, there is no input validation checking for the information provided by the client. Without validation checking for the “name” input, the application is at risk for injection attacks, and hackers could gain access to sensitive data. There are also no checks for the “name” field being only letters, and if a user were to input non-alphanumeric information it could cause data corruption or unexpected results. Finally, it is vulnerable to a DoS attack, since users can enter very large inputs into the field, which can cause unexpected results.
2. Similarly, “business\_name” in the CRUD controller has the same problems.
3. In the DocData file, values for “key” and “value” are accepted without input validation, leaving it open to injection attacks as described above
4. In the customer file, “deposit” accepts an integer value with no validation. This could lead to unexpected behaviors and security risks.

**APIs**

1. In the application.properties file, database information and API keys are hardcoded in which increases risks for exposure.
2. No HTTPS communication is in place.
3. There is no mechanism in place in any of the files for authentication and authorization of users. This leaves the API vulnerable to unauthorized access.

**Cryptography**

1. In the application.properties file, the server.port does not use HTTPS, which helps to encrypt data between the clients and server, which is very important when handling financial and personal information.
2. In the pom.xml file, it is shown that BouncyCastle version 1.46 is used. This is a cryptography library and is meant to provide updated security features. However, using an old version leaves the application at risk for security leaks.

**Client/Server**

1. The DocData file opens a new connection with the database but does not close it appropriately. This can lead to data leaks.
2. In the customer file, account balance is being increased, but there is no check for integer overflow. This could lead to a DoS attack.

**Code Error**

1. In the DocData file, the “read\_document” method prints to console, which may leave it vulnerable to data leaks.
2. Using improper validation techniques for “deposit” in the customer class could lead to unexpected errors that the code does not account for.

**Code Quality**

1. In the DocData file, the source code shows the database credentials (“jdbc:mysql://localhost:3306/test”, “root”, “root”) This poses a risk for unauthorized access to information and does not follow coding best practices.
2. In the pom.xml file, Java 1.8 is used, which is older and not in best practice. It is more susceptible to vulnerabilities.
3. Using outdated dependencies (Ex BouncyCastle ver 1.46) is not proper code quality.

**Encapsulation**

1. In myDateTime, encapsulation is not used since the class is not private.
2. In the customer file, there is direct access to “account\_balance”, which can be a security risk. It should be private with public getter method access.

## Static Testing

|  |  |
| --- | --- |
| [**bcprov-jdk15on-1.46.jar**](#RANGE!l2_991c96a4e31e6c19e2b9136c8955bd) | **HIGH** |
| This is the Bouncy Castle Crypto package for Java implementation of cryptographic algorithms. Versions for Java prior to 1.48 do not properly consider time side-channel attacks which allows remote accessors to conduct distinguishing attacks and plain-text recovery attacks. Versions before 1.51 do not validate a point is within elliptic curve, making it easy for attackers to gain remote keys. Versions before 1.55 have improper verification of cryptographic signatures. There were also key management errors. |  |
| [**spring-boot-2.2.4.RELEASE.jar**](#RANGE!l3_225a4fd31156c254e3bb92adb42ee8)  Spring Boot. Versions prior to 2.2.11 vulnerable to temporary directory hijacking. Exposure of Resource to Wrong Sphere. In versions 3.0.0 and older, applications deployed to Cloud Foundry vulnerable to security bypass. Also vulnerable to uncontrolled resource consumption. Recommended upgrade to versions 3.0.6+ | **CRITICAL** |
| [**logback-core-1.2.3.jar**](#RANGE!l4_864344400c3d4d92dfeb0a305dc87d)  logback-core module. In versions 1.2.7 and older, an attacker with privileges to edit configurations could craft malicious code. Deserialization of untrusted data. A vulnerability in 1.4.11 allows a DoS attack. | **HIGH** |
| [**log4j-api-2.12.1.jar**](#RANGE!l5_a55e6d987f50a515c9260b0451b4fa)  The Apache Log4j API. Had an improper certificate validation in versions before 2.12.3 and later. This led to a man-in-the middle attack that could leak message logs. Versions up to 2.15.0 (excluding 2.12.2 and 2.12.3) were vulnerable to uncontrolled resource consumption, deserialization of untrusted data, and improper input validation. Versions through 2.16.0 (excluding 2.12.3 and 2.3.1) were vulnerable to uncontrolled recursion. | **CRITICAL** |
| [**snakeyaml-1.25.jar**](#RANGE!l8_8b6e01ef661d8378ae6dd7b511a7f2)  YAML 1.1 parser and emitter for Java. Versions before 1.26 allow entity expansion during load operations, which is improper restriction of recursive entity references in DTDs. Unbound alias chasing can result in a DoS attack. The constructor class does not restrict types, which can result in deserializing of untrusted data. Recommendation to use SnakeYami’s SafeConstructor when parsing. Recommended to use version 2.0 and later. Parsing can also lead to uncontrolled resource consumption and an out-of-bounds write. | **CRITICAL** |
|  |
| [**jackson-databind-2.10.2.jar**](#RANGE!l9_0528de95f198afafbcfb0c09d2e43b)  General data-binding functionality for Jackson: works on core streaming API. Has a vulnerability due to improper restriction of XML External Entity Reference (XXE). Versions before 2.13.0 allow Java StackOverflow exception and DoS due to out of bounds write. Also DoS vulnerability due to allocation of resources without limits or throttling. | **HIGH** |  |
| [**tomcat-embed-core-9.0.30.jar**](#RANGE!l13_ad32909314fe2ba02cec036434c0a)  Core Tomcat implementation. Vulnerable to Inconsistent Interpretation of HTTP requests (HTTP smuggling). Vulnerable to a non-responsive server due to allowing too many requests. Several versions vulnerable to improper release of memory before removing last reference. Several versions vulnerable to Loop with unreachable exit condition, Inconsistent Interpretation of HTTP Requests, Incorrect Default Permissions, Deserialization of Untrusted Data, Use of Incorrectly Resolved Name or Reference, Improper Encoding or Escaping of Output, Improper Neutralization of Input During Web Page Generation, Unprotected Transport of Credentials, Incomplete cleanup and Information Exposure. | **CRITICAL** |  |
|  |
| [**hibernate-validator-6.0.18.Final.jar**](#RANGE!l16_7fd00bcd87e14b6ba66279282ef15)  Hibernate’s Bean Validation reference implementation. Vulnerable to improper input validation. A flaw in version 6.1.2 allows invalid EL expressions to be handled as valid, allowing attackers to bypass input sanitation. | **MEDIUM** |  |
| [**spring-web-5.2.3.RELEASE.jar**](#RANGE!l19_dd386a02e40b915ab400a3bf9f586)  Spring Web. 5.3.16 and older vulnerable to potential remote code execution – deserialization of untrusted data. Versions 5.3.0 and older vulnerable to improper output neutralization for logs and exposure of resources to wrong spheres. | **HIGH** |  |
|  |
| [**spring-beans-5.2.3.RELEASE.jar**](#RANGE!l20_0250c8c641433dc06b1b44e4563fa)  Spring Beans. Vulnerable to Improper Control of generation of Code (Code injection) when Spring MVC or Spring WebFlux were running on JDK9+. | **HIGH** |  |
|  |
| [**spring-webmvc-5.2.3.RELEASE.jar**](#RANGE!l21_745a62502023d2496b565b7fe102b)  Spring Web MVC. Vulnerable to Improper Output Neutralization for Logs in 5.3.13 and older where users could provide malicious input to cause insertion of additional log entries. | **MEDIUM** |  |
|  |
| [**spring-context-5.2.3.RELEASE.jar**](#RANGE!l22_7750c95c96c7a1885c8b1b503ba91)  Spring Context. In 5.3.18 and older Improper Handling of Case Sensitivity, was not properly protected unless a field was listed with both upper and lower case for the first character. | **MEDIUM** |  |
|  |
| [**spring-expression-5.2.3.RELEASE.jar**](#RANGE!l23_d0c6bb10758805b2153c589686b80)  Spring Expression Language (SpEL). In 5.3.16 and older it is possible for users to craft special SpEL expression that caused a DoS. Allocation of Resources Without Limits or Throttling. Versions prior to 5.2.24, 5.3.27, and 6.0.8+ possible to cause a DoS due to Improper Neutralization of Special Elements used in an Expression Language Statement (Expression Language Injection). | **MEDIUM** |  |
|  |
|  |  |  |

## Mitigation Plan

There are several mitigation techniques that can be used to minimize the security risks that exist in the current application. Validating input, securing APIs, proper cryptography, client/server considerations, minimizing code errors, using up to date secure quality coding practices, and encapsulation are some examples that relate to this project. Below are more specific examples of mitigation techniques for security risks based off each of these principles with examples from the application code.

**Input Validation**

1. Checking user input in the GreetingController, DocData and CRUD controller for alphanumerical characters only, checking for input length, checking for prohibited characters.
2. Also using parameters to prevent SQL injection attacks since there will be interactions with databases for this client.
3. Same as above
4. In the customer file, input validation should be done to make sure inputs fall within parameters. There should be limits on the amounts that are checked, checked for non-negative numbers. Also, int may not be the most appropriate as financial transactions are often done to two decimal places.

**APIs**

1. API information should not be hardcoded into the application.properties file.
2. The API will be interacting with servers over a network. HTTPS should be used to protect information.
3. API should enforce authentication and authorization of users. Checks should be in place to verify proper access.

**Cryptography**

1. Use HTTPS with a secure port.
2. Use BouncyCastle version 1.69 or newer.

**Client/Server**

1. In the DocData file, database connections should be securely closed after they are used.
2. Limit checks should not suffer from integer overflow. Use “long” or “BigInteger” for larger data types. Secondly, a check for overflow can prevent the risk for a DoS attack.

**Code Error**

1. In the DocData file, exceptions should be logged securely and error messages should not reveal any information to the user.
2. Incorporate proper validation techniques.

**Code Quality**

1. Do not store credentials in plain text in the source code. Database credentials should be securely managed.
2. Using Java 11 or later would enhance security.
3. Use BouncyCastle version 1.6 or higher

**Encapsulation**

1. Make classes private so that users do not have access to information they should not see or need.
2. In the customer file, “account\_balance” should be private with public getter methods. Validation checks should be in place for access.

**References**

Manico, J. & Detlefson, A. (2015).  *Iron-Clad Java: Building Secure Web Applications.* McGraw Hill.

ORACLE. (2023). *Secure Coding Guidelines for Java SE.* Oracle.com. Accessed via

<https://www.oracle.com/java/technologies/javase/seccodeguide.html>

OWASP. (2010). *OWASP SCP Quick Reference Guide.*  OWASP.org. Accessed via <https://owasp.org/www->

pdf-archive/OWASP\_SCP\_Quick\_Reference\_Guide\_v1.pdf

Spring.io. (2024). *Building REST Services with Spring.* Spring.io. Accessed via

<https://spring.io/guides/tutorials/rest/>