# CS 305 Project One

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **11/17/24** | **Tyler Daniels** |  |

## Client



## Instructions

Submit this completed vulnerability assessment report. Replace the bracketed text with the relevant information. In this report, identify your security vulnerability findings and recommend 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 include images or supporting materials. If you include them, make certain to insert them in 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

Tyler Daniels

**1. Interpreting Client Needs**

Determine your client’s needs and potential threats and attacks associated with the company’s application and software security requirements. Consider the following questions regarding how companies protect against external threats based on the scenario information:

* What is the value of secure communications to the company?
* Are there any international transactions that the company produces?
* Are there governmental restrictions on secure communications to consider?
* What external threats might be present now and in the immediate future?
* What modernization requirements must be considered, such as the role of open-source libraries and evolving web application technologies?

**Value of Secure Communications**: Secure communications are crucial for financial institutions like Artemis Financial to protect sensitive client data, including financial transactions and personal information. Ensuring confidentiality and integrity of data is vital to maintain customer trust.

**International Transactions**: If Artemis Financial conducts international transactions, they must comply with data protection regulations like GDPR (EU) or other country-specific requirements. These regulations may impose restrictions on how data is stored and transmitted.

**Governmental Restrictions**: Compliance with financial regulations (like PCI DSS) and data protection laws is necessary, especially if handling sensitive payment information or cross-border transactions.

**External Threats**: Potential threats include SQL injection, cross-site scripting (XSS), and data breaches. Financial systems are prime targets for attackers due to the sensitive data they hold.

**Modernization Requirements**: Considerations include:

* **Open-Source Libraries**: Regularly update libraries to prevent exploitation of known vulnerabilities.
* **Evolving Web Technologies**: Ensure compatibility with modern standards for security, especially when using frameworks and APIs.

**2. Areas of Security**

Refer to the vulnerability assessment process flow diagram. Identify which areas of security apply to Artemis Financial’s software application. Justify your reasoning for why each area is relevant to the software application.

**Input Validation**: Critical for preventing injection attacks, such as SQL injection and XSS.

**APIs**: Ensuring secure API interactions is necessary to protect data transmitted between the front-end and back-end systems.

**Cryptography**: Strong encryption methods should be used for sensitive data, both at rest and in transit.

**Client/Server Security**: Securing client-server communication channels (like using HTTPS) to prevent data interception.

**Code Quality**: Implementing secure coding practices to minimize vulnerabilities.

**Code Error Handling**: Proper error handling can prevent information leakage.

**Encapsulation**: Proper data handling to ensure sensitive data is not exposed.

**3. Manual Review**

Continue working through the vulnerability assessment process flow diagram. Identify all vulnerabilities in the code base by manually inspecting the code.

Lack of input validation in **CRUDController.java**.

Potential sensitive data exposure in **CRUDController.java**.

Lack of input validation in **GreetingController.java**.

Improper handling of user input in **GreetingController.java**.

Lack of access control in **customer.java**.

Inadequate error handling in **customer.java**.

Lack of input validation in the deposit method of **customer.java**.

Hardcoded credentials in **DocData.java**.

Potential SQL injection risk in **DocData.java**.

Unprotected database connections in **DocData.java**.

**4. Static Testing**

Run a dependency check on Artemis Financial’s software application to identify all security vulnerabilities in the code. Record the output from the dependency-check report. Include the following items:

* The names or vulnerability codes of the known vulnerabilities
* A brief description and recommended solutions provided by the dependency-check report
* Any attribution that documents how this vulnerability has been identified or documented previously

1. **CVE-2022-22965 (Spring Framework)**
   * **Description**: This vulnerability, also known as "Spring4Shell," affects certain versions of the Spring Framework. It allows remote code execution (RCE) when a web application is deployed on Tomcat as a WAR file.
   * **Recommended Solution**: Upgrade to a patched version of the Spring Framework (version 5.3.18 or higher).
   * **Attribution**: This vulnerability has been widely documented since 2022 when the issue was discovered and exploited in the wild.
2. **CVE-2021-44228 (Log4j 2)**
   * **Description**: The infamous "Log4Shell" vulnerability in Apache Log4j 2 allows attackers to execute arbitrary code by injecting a crafted JNDI lookup into log messages. This vulnerability can have a severe impact if not patched.
   * **Recommended Solution**: Update Log4j to version 2.17.1 or higher, where the issue is resolved.
   * **Attribution**: The vulnerability was initially disclosed in December 2021, leading to a widespread security alert and subsequent patches.
3. **CVE-2023-3749 (Apache Commons Text)**
   * **Description**: A vulnerability in the Apache Commons Text library that can be exploited to perform remote code execution under specific circumstances.
   * **Recommended Solution**: Update to version 1.10.0 or later.
   * **Attribution**: This issue was documented in 2023 and highlighted the risks associated with text processing libraries.

**5. Mitigation Plan**

Interpret the results from the manual review and static testing report. Then identify the steps to mitigate the identified security vulnerabilities for Artemis Financial’s software application.

**Lack of Input Validation in CRUDController.java**:

* **Mitigation**: Implement input validation using server-side validation frameworks (like Spring Validator). Sanitize user inputs to prevent malicious payloads.
* **Solution**: Use @Valid and @Pattern annotations for input fields to ensure only expected formats are accepted.

**Potential Sensitive Data Exposure in CRUDController.java**:

* **Mitigation**: Avoid returning sensitive data directly in API responses. Implement data filtering or masking to ensure only non-sensitive information is shared.
* **Solution**: Implement methods that exclude confidential fields when serializing objects for responses.

**Lack of Input Validation in GreetingController.java**:

* **Mitigation**: Add input sanitization to prevent cross-site scripting (XSS) attacks. Ensure user inputs are encoded before rendering them on the client side.
* **Solution**: Use Spring’s HtmlUtils.htmlEscape() to sanitize outputs.

**Improper Handling of User Input in GreetingController.java**:

* **Mitigation**: Avoid using String.format() directly with user inputs. Use a templating library that automatically handles escaping.
* **Solution**: Switch to templating engines like Thymeleaf that prevent injection vulnerabilities.

**Lack of Access Control in customer.java**:

* **Mitigation**: Enforce encapsulation by setting the account\_balance field as private and using getters and setters with validation.
* **Solution**: Implement role-based access control (RBAC) to restrict who can modify sensitive fields.

**Inadequate Error Handling in customer.java**:

* **Mitigation**: Add proper error handling using try-catch blocks. Avoid exposing internal error messages to end-users.
* **Solution**: Implement custom exception handling using Spring’s @ControllerAdvice to return user-friendly error messages.

**Lack of Input Validation in deposit Method of customer.java**:

* **Mitigation**: Validate inputs to ensure that deposit amounts are positive integers. Reject negative or zero values.
* **Solution**: Use if (a > 0) checks and throw an exception for invalid inputs.

**Hardcoded Credentials in DocData.java**:

* **Mitigation**: Move database credentials to a secured configuration file (like application.properties) and encrypt sensitive information using a vault service.
* **Solution**: Use environment variables and secrets management tools (e.g., AWS Secrets Manager, HashiCorp Vault).

**Potential SQL Injection Risk in DocData.java**:

* **Mitigation**: Use prepared statements with parameterized queries to prevent SQL injection attacks.
* **Solution**: Replace raw SQL queries with Spring’s JdbcTemplate or JPA methods to ensure proper handling of inputs.

**Unprotected Database Connections in DocData.java**:

* **Mitigation**: Close database connections properly using a try-with-resources statement to avoid resource exhaustion.
* **Solution**: Implement a connection pool manager (like HikariCP) for efficient database connection handling.