CS 305 Project One

Section 1.01 Document Revision History

Version	Date	Author	Comments
1.0	01/24/2026	Joshua Sevy	

Section 1.02 Client

ARTEMIS FINANCIAL

Section 1.03 Developer
Joshua Sevy

1. Interpreting Client Needs

Artemis Financial offers individual consulting services across savings, retirement, investments, and insurance. Considering the sensitive nature of this information, it is important to maintain confidentiality through secure communication means to avoid data breaches and build user trust, as any mishandling of customer information will cause financial risks and legal liabilities.

Despite the scenario not indicating that Artemis Financial conducts international transactions, the use of a web-based REST API implies some form of remote access, thereby increasing vulnerability to outside threats from any location. Financial applications also must consider regulatory needs regarding information assurance, especially when communicating, as well as data security and auditing.

Some of the most relevant threats from outside include unauthorized access to APIs, injection threats, information interception, denial-of-service attacks, and the use of libraries, especially third-party libraries. Financial applications are often targets due to the highly sensitive information that is processed. It seems that there will be a persistence or emergence of similar threats even as technologies improve.

Further, modernization requirements will also affect the application's security profile. This is because the firm uses various open-source tools and libraries, as mentioned previously, while still incurring risks when vulnerable versions are used and when it lacks continuous, appropriate servicing and updates during the application's development and evolution cycles.

2. Areas of Security

Based on the process flow diagram of the process of vulnerability assessment, there are various areas with regards to the system's security that can be considered on the basis of the Artemis Financial system's functions as a web application, particularly with regards to its exposure to REST endpoints and data provided by a client through web system, where an architectural evaluation process is significant.

Input validation is a key area to focus on since inputs are coming from outside the application's trust domain. This could allow an attacker to manipulate input data to produce unintended behavior or inject malicious data, or to compromise application/service availability. This relates to controllers that manipulate customer or business-related data.

API security is also a concern, given that the application provides access to financial information via the API. Protecting secure API interactions will prevent unauthorized access and exposure of internal features and functionality. Because the application uses a client-server model, all incoming requests must be treated as untrusted, underscoring the importance of client-server security dynamics within the application. Code quality and error handling, including safe application failure, need to be reviewed. Poor exception handling or logging might leak internal implementation details and help attackers. Encapsulation issues are also relevant: poor data modeling or unrestricted access to internal objects can increase the impact of vulnerabilities.

Finally, dependency and supply chain security need to be considered, given the application's reliance on open-source libraries. Vulnerabilities in third-party dependencies can undermine otherwise secure code; for this reason, testing and reviewing dependencies are essential parts of the assessment.

3. Manual Review

1. Hardcoded database login details (exposed secret in code).
    a. In DocData.java, database access username and password credentials are hardcoded (e.g., "root/root"). This creates an immediate confidentiality risk if source code is shared, pushed to a public repository, or accessed by unauthorized parties.
2. Risk of SQL injection due to missing parameterized queries.
    a. DocData.java does not demonstrate the use of prepared statements or parameter binding. If query logic is added later using string concatenation, it can introduce SQL injection risk, especially since user-controlled fields are already being accepted upstream.
3. Database connections are not safely closed.
    a. In DocData.java, connections are not consistently closed, and error handling prints stack traces. Poor resource management may cause connection exhaustion and stability issues, which can also become an availability concern under load or abuse.
4. Verbose exception handling may expose internal details.
    a. DocData.java includes exception behavior that prints stack traces. If exception propagation to API responses through default handlers, the application may leak implementation details useful to attackers.
5. Unvalidated request input in the controller.
    a. In CRUDController.java, the business_name request parameter is neither validated nor sanitized. Because the value originates outside the trust boundary, a lack of validation increases the risk of injection attacks and input-based denial-of-service attacks.
6. Direct data-layer instantiation in controller
    a. CRUDController.java instantiates DocData directly rather than using dependency injection and a service layer. This reduces testability and increases the risk that unsafe data-access patterns are reused across endpoints.
7. Incorrect return behavior may leak internal object details.
    a. CRUDController.java returns doc.toString() rather than returning the intended data from DocData.read_document. Default toString() output can reveal internal class names and object IDs, suggesting the endpoint is not returning a controlled response format.
8. The data model lacks encapsulation and validation.
    a. In customer.java, fields are not private, and there are no accessors or input validation. This makes it easier to set invalid values and increases the risk of insecure direct object reference attacks when the model is used across controllers or persistence layers.
9. Financial data validity risk due to currency type and update safety.
    a. In customer.java, account_balance uses an int, which is not appropriate for currency representation and can lead to rounding and integrity issues as calculations become more complex.
10. Reflection input in the API response creates an XSS risk depending on the client rendering.
    a. GreetingController.java echoes user-supplied input into the response. If a consumer of this API renders the returned value into HTML without encoding, it may enable cross-site scripting through reflected content.

4. Static Testing

To identify known vulnerabilities introduced by third-party libraries, the OWASP Dependency-Check Maven plug-in was integrated into the Artemis Financial codebase and executed. The generated HTML report identifies vulnerable dependencies and maps each finding to established vulnerability databases such as the Common Vulnerabilities and Exposures (CVE) list and the National Vulnerability Database (NVD).

The dependency-check report identified vulnerabilities in SnakeYAML (org.yaml:snakeyaml@1.25):

- **CVE-2022-1471:** 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 parsinug untrusted content to restrict deserialization. We recommend upgrading to version 2.0 and beyond.
- **CVE-2022-25857:** The package org.yaml:snakeyaml from 0 and before 1.31 are vulnerable to Denial of Service (DoS) due missing to nested depth limitation for collections.

The report also identified vulnerabilities in Spring Web (org.springframework:spring-web@5.2.3.RELEASE) and Spring Core (org.springframework:spring-core@5.2.3.RELEASE).

- **CVE-2022-22965:** A Spring MVC or Spring WebFlux application running on JDK 9+ may be vulnerable to remote code execution (RCE) via data binding. The specific exploit requires the application to run on Tomcat as a WAR deployment. If the application is deployed as a Spring Boot executable jar, i.e. the default, it is not vulnerable to the exploit. However, the nature of the vulnerability is more general, and there sermay be other ways to exploit it.
    - **CISA Known Exploited Vulnerability:**
        - Product: VMware Spring Framework
        - Name: Spring Framework JDK 9+ Remote Code Execution Vulnerability
        - Date Added: 2022-04-04
        - Description: Spring MVC or Spring WebFlux application running on JDK 9+ may be vulnerable to remote code execution (RCE) via data binding.
        - Required Action: Apply updates per vendor instructions.
        - Due Date: 2022-04-25
        - Notes: https://nvd.nist.gov/vuln/detail/CVE-2022-22965

The report also identified vulnerabilities in Spring Boot (spring-boot@2.2.4.RELEASE).

- **CVE-2023-20873:** In Spring Boot versions 3.0.0 - 3.0.5, 2.7.0 - 2.7.10, and older unsupported versions, an application that is deployed to Cloud Foundry could be susceptible to a security bypass. Users of affected versions should apply the following mitigation: 3.0.x users should upgrade to 3.0.6+. 2.7.x users should upgrade to 2.7.11+. Users of older, unsupported versions should upgrade to 3.0.6+ or 2.7.11+.

5. Mitigation Plan

In order to mitigate the issues found in the application and the vulnerabilities found during the manual code review and static dependency analysis, the following mitigation actions are recommended for Artemis Financial to reduce possible security risks in the web-based application.

Some of the solutions presented to solve the problem of eliminating hard-coded credentials with regard to the database are the removal of sensitive data from the source code. Details such as the username and password are eliminated. As an alternative, the suggestion is to incorporate environmental variables and the concept of least privilege.

To effectively prevent SQL Injection attacks and ensure database security, all database queries should be parameterized or handled by an object-relational mapping tool. This makes sure that user data is never concatenated or embedded into a query string. The database connection should be handled appropriately to avoid leaks or unavailability.

Exception handling and logging can be improved by following best practices, which can help prevent internal implementation details from leaking. Thus, it is advised not to include any direct logging of stack trace, and as a general statement, error messages can be simple, leaving everything else to be included within structured logs.

In addition, validation of inputs must occur at every API boundary, meaning parameters passed as part of requests (e.g., business_name) must still be validated to ensure they meet specific character length and set requirements to prevent injection and denial-of-service attacks from user input.

To better segregate architectural boundaries, instead of having data access classes instantiated directly by the controllers, it is important to introduce a service layer with dependency injection to avoid dangerous data access patterns.

The data models must also be updated to support data encapsulation principles and validation. Fields should be designed as private with controlled accessors. The data models should also include support for validation to prevent malicious data from being added. Financial values should use appropriate data types for currency handling, and support for concurrency should also be included.

In addition, to reduce reflected input and potential cross-site scripting attacks, user input shouldn’t be returned in an HTTP response in raw form or concatenated with other content. Responses from the API should instead use structured data objects.

For dependency-related vulnerabilities detected by static analysis, the issues may be resolved by simply upgrading to more recent, supported library versions. For the SnakeYAML library, an updated version that follows safe parsing practices may be obtained to fix the associated issues. For the Spring Framework dependencies, the dependencies may be updated to contain patches to fix known related remote code execution and order-violation vulnerabilities.

Lastly, dependency scanning and manual code review should be part of the standard development process. Verifying the findings from dependency scans and removing known false positives would enable developers to focus on areas where remediation of vulnerabilities is necessary, thereby minimizing development efficiency losses.

Migration Action List:

1. **Removal of hardcoded credentials.**
    a. All usernames and passwords used in database connections should be replaced and stored securely using configuration mechanisms such as environment variables. Access permissions should adhere to the principle of least privilege.
2. **Implement parameterized database queries.**
    a. All database access should be done using prepared statements or an object-relational mapping framework to stop SQL injection attacks. Never concatenate user input with queries.
3. **Ensure Safe Database Resource Management.**
    a. The connections, statements, and result sets used in databases must be handled to eliminate linkage leaks and unavailability issues.
4. **Improve exception handling and logging practices.**
    a. Stack traces must not be presented to the users. Similarly, it also must not be printed. Error responses to the user should be kept generic, while detailed error responses should be logged.
5. **Validate and Sanitize All Data inputs.**
    a. All parameters and values passed in the requests should be validated for length and format. Input validation should occur at the API boundaries.
6. **Introduce a service layer and dependency injection.**
    a. Controllers must not instantiate data access directly; instead, they should add a service level that helps avoid unsafe data access pattern reuse.
7. **Return controlled response objects from APIs.**
    a. API endpoint responses need to return data transfer objects rather than string representations of the object. Otherwise, there is an exposure to internal class implementation.
8. **Enforce encapsulation and validation on data models.**
    a. The model fields should be private properties with controlled accessors, and validation rules should be enforced to ensure that data is not compromised during persistence or processing.
9. **Choose appropriate data types for financial data, including concurrency-sensitive data.**
    a. Data types for representing financial values shall be used appropriately, while concurrency controls shall be implemented to handle inconsistent updates arising from multiple transactions.
10. **Prevent reflected input from being rendered without encoding.**

   a. User input data must not be echoed back without proper encoding if it is incorporated into a response. API response data returned to clients must be structured safely without direct echoing.

11. **Upgrade Vulnerable Dependencies.**
    a. All dependencies listed in the static testing report should be updated to a currently supported version with vendor-supplied patches. SnakeYAML, Spring Framework, and Spring Boot should all be updated in accordance with the information provided by CVE.
12. **Integrate security checks into the development lifecycle**
    a. Manual code reviews and dependency scans should take place and be conducted regularly. Verify scan results and remove any false positives to ensure actions taken address real risks only.