# CS 305 Project One Template

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **05/25/2025** | **Nathan Sakhichand** |  |

## 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

Nathan Sakhichand

**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?

In conducting a vulnerability assessment for Artemis Financial, it's crucial to interpret their needs accurately. Secure communications are of utmost importance for Artemis Financial as they deal with highly sensitive client information, including Social Security Numbers, tax details, and financial plans. This necessitates robust security measures to ensure confidentiality and maintain client trust. Considering that Artemis Financial likely engages in international transactions, they must ensure secure communication channels that comply with international privacy standards to protect data during cross-border transfers. Although no specific governmental restrictions are mentioned, the company must remain compliant with laws concerning trade secrets and data protection to avoid legal repercussions. The primary external threats include data breaches, phishing attacks, and ransomware, all of which target client information, requiring strong encryption and regular security updates to mitigate these risks. In terms of modernization, Artemis Financial should focus on keeping their open-source libraries current to incorporate the latest security patches and bug fixes, reducing vulnerabilities. Additionally, staying updated with evolving web application technologies is essential, particularly in adopting modern encryption standards, secure coding practices, and new security protocols. This comprehensive approach to security will ensure that Artemis Financial's web-based software application remains secure against both current and future threats.

**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.

Based on the functionality of Artemis Financial's web application and the vulnerability assessment process flow diagram, several areas of security are particularly relevant:

Input validation is critical for Artemis Financial's application as it deals with user input, which could potentially be exploited for attacks like SQL injection if not properly sanitized. By validating input strings, the application ensures that only expected and safe data formats are processed, protecting both the system and user data from malicious inputs. This is especially important given that financial data and personal identifiers are involved.

APIs are essential since Artemis Financial's application interacts both internally within the company and externally with end-users via web browsers. A well-developed and secure API defines how users interact with the program, specifying which data access methods are permissible. This is crucial for maintaining secure interactions with third-party software, ensuring that only authorized data exchanges occur.

Cryptography is necessary due to the international nature of Artemis Financial's transactions, which involve proprietary customer information. Encrypting data during transfer ensures compliance with both North American and international regulations, safeguarding sensitive information from unauthorized access during transmission across borders.

Code error handling should work in conjunction with input validation and API security. Proper error handling mechanisms are vital to prevent unauthorized access or escalation of privileges, which could occur if errors expose system details or allow for manipulation of the application's behavior.

Code quality is imperative to prevent unintentional data exposure and to maintain the integrity of the application. High-quality code ensures that access controls are correctly implemented, allowing users to access only their own information and not others' or sensitive system areas. This also helps in managing user permissions effectively, ensuring that different user levels have appropriate access to methods and data.

By focusing on these areas, Artemis Financial can ensure their web application is secure, protecting against common vulnerabilities while maintaining the functionality required for their operations. Not all areas from the vulnerability assessment process flow diagram are relevant here, as some might pertain more to different types of applications or environments. However, for Artemis Financial, these selected areas directly address the security needs associated with their specific use case involving financial data, user interaction, and international transactions.

**3. Manual Review**

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

Upon conducting a manual review of the Project One Code Base for Artemis Financial, applying the knowledge from the vulnerability assessment process and the seven security areas, I have identified several vulnerabilities:

1. Input Validation (GreetingController.java): There is a noticeable absence of input validation within the GreetingController class. The code accepts user input without any checks, which could lead to potential SQL injection or cross-site scripting (XSS) attacks. The input is directly used in the greeting message without sanitization.

2. Input Validation (POM.XML): The POM.XML file was reviewed to check for any validation libraries like Apache Commons Validator, but none were found. This indicates that the project might not be utilizing any external validation tools, increasing the risk associated with improper input handling.

3. API Security (GreetingController.java): The API interaction in the GreetingController does not use a POST method for data submission, instead, it uses URL parameters, which can expose user input in browser history and logs. This method of data transfer is insecure as it can be easily intercepted or manipulated.

4. API Design (General): There is no clear, well-defined API structure. Without a properly designed RESTful API, end-users lack a structured way to interact with the application, which could lead to misuse or misinterpretation of how to securely engage with the system.

5. Cryptography (General): No evidence of cryptographic practices was found in the codebase. Given the need for secure international transactions and storage of sensitive financial data, the absence of encryption mechanisms is a significant vulnerability, risking data breaches and non-compliance with international data protection standards.

6. Error Handling (DocData.java): The DocData class lacks proper error handling. While it includes try-catch blocks, there is no comprehensive error management strategy in place. This can lead to information leakage through stack traces or unexpected application behavior when errors occur.

7. Code Quality (General): Although the overall code quality appears good in terms of structure, the absence of an API makes the application less user-friendly and less secure. Moreover, the lack of input validation and the use of URL parameters for data input compromise the security posture.

8. Code Error Handling (GreetingController.java): Similar to DocData.java, the GreetingController does not implement any error handling. This could result in application crashes or sensitive information exposure if exceptions are not caught and handled appropriately.

9. Data Exposure (GreetingController.java): Since the application accepts input via URL parameters without any encryption or secure method, there's a risk of sensitive data being exposed in logs, browser history, or through network sniffing.

10. User Access Control (General): There's no indication of user access control or authentication mechanisms within the provided code, which means any user could potentially access any part of the application if they know the URL structure, posing a significant security risk.

These findings highlight critical areas where Artemis Financial needs to focus on improving security to protect against potential threats and ensure compliance with data protection standards, especially considering the sensitive nature of the financial information they handle.

**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

|  |  |  |
| --- | --- | --- |
| **Dependency Name** | **Brief Description** | **Severity** |
| bcprov-jdk15on-1.46.jar | Bouncy Castle is a cryptography API for Java. It has 18 CVEs, with concerns including potential cryptographic weaknesses or vulnerabilities that could compromise data security | **HIGH** |
| hibernate-validator-6.0.18.Final.jar | Hibernate Validator is used for bean validation in Java applications. It has 3 CVEs, with issues potentially related to validation bypasses that might allow invalid data to pass through. | **MEDIUM** |
| jackson-core-2.10.2.jar | Jackson Core is part of the Jackson library, handling JSON parsing and generation. It has 1 CVE, with low confidence, possibly related to parsing vulnerabilities. | **MEDIUM** |
| jackson-databind-2.10.2.jar | Jackson Databind provides data-binding functionality for JSON in Java. It has 6 CVEs, with high severity issues like deserialization vulnerabilities that could lead to remote code execution. | **HIGH** |
| log4j-api-2.12.1.jar | Log4j API is part of the Log4j logging framework. It has 1 CVE, with low severity, possibly related to minor configuration issues. | **LOW** |
| logback-classic-1.2.3.jar | Logback Classic is a logging framework for Java, extending Log4j. It has 2 CVEs, with potential vulnerabilities like information disclosure. | **HIGH** |
| logback-core-1.2.3.jar | Logback Core provides the core functionality for the Logback logging framework. It has 4 CVEs, with similar security risks as Logback Classic. | **HIGH** |
| snakeyaml-1.25.jar | SnakeYAML is a YAML parser and emitter for Java. It has 8 CVEs, the most critical of which could allow for arbitrary code execution due to unsafe deserialization of YAML content. | **CRITICAL** |
| spring-boot-2.2.4.RELEASE.jar | Spring Boot simplifies Spring application development. It has 3 CVEs, with risks including security bypass or data exposure vulnerabilities. | **CRITICAL** |
| spring-boot-starter-web-2.2.4.RELEASE.jar | This starter pack is for building web applications with Spring Boot, including Spring MVC. It has 3 CVEs, similar to Spring Boot, involving potential security bypasses. | **CRITICAL** |
| spring-context-5.2.3.RELEASE.jar | Spring Context manages beans and their lifecycle in the Spring Framework. It has 13 CVEs, with potential for severe issues like remote code execution or data tampering. | **CRITICAL** |
| spring-core-5.2.3.RELEASE.jar | Spring Core contains the fundamental parts of the Spring Framework, including IoC container. It has 12 CVEs, with concerns about data integrity and security bypasses. | **CRITICAL** |
| spring-expression-5.2.3.RELEASE.jar | Spring Expression Language (SpEL) is used for querying and manipulating an object graph. It has 13 CVEs, where vulnerabilities might allow for expression injection leading to code execution. | **CRITICAL** |
| spring-web-5.2.3.RELEASE.jar | Spring Web provides HTTP abstractions for web applications. It has 18 CVEs, with risks including path traversal or security bypass. | **CRITICAL** |
| spring-webmvc-5.2.3.RELEASE.jar | Spring Web MVC is the original web framework built on the Servlet API. It has 13 CVEs, similar to Spring Web, with potential for severe security issues. | **CRITICAL** |
| tomcat-embed-core-9.0.30.jar | Tomcat Embed Core allows embedding Tomcat within applications. It has 35 CVEs, with critical vulnerabilities like potential remote code execution or information disclosure. | **CRITICAL** |
| tomcat-embed-websocket-9.0.30.jar | Tomcat Embed Websocket provides WebSocket support for embedded Tomcat. It has 36 CVEs, with critical issues similar to Tomcat Core, including potential for remote attacks. | **CRITICAL** |

**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.

Based on the findings from the manual review and static testing, here's a detailed mitigation plan for the identified security vulnerabilities in Artemis Financial's software application:

1. Input Validation (GreetingController.java):

- Action: Implement input validation using a library like Hibernate Validator or Apache Commons Validator. Add annotations or validation methods to check user inputs before processing them to prevent SQL injection and XSS attacks.

- Justification: This will ensure that only sanitized and expected data formats are accepted, reducing the risk of malicious input.

2. Input Validation (POM.XML):

- Action: Update the POM.XML to include dependencies for validation libraries such as org.hibernate.validator:hibernate-validator or commons-validator:commons-validator.

- Justification: Incorporating these libraries will provide robust validation mechanisms out of the box, enhancing input security.

3. API Security (GreetingController.java):

- Action: Modify the API to use HTTP POST methods instead of GET for data submission. Update the endpoints to accept JSON payloads securely.

- Justification: Using POST methods will prevent data from being exposed in URLs, logs, and browser history, enhancing data privacy.

4. API Design (General):

- Action: Design and implement a RESTful API with clear endpoints, methods, and data formats. Use tools like Swagger or OpenAPI to document the API.

- Justification: A well-defined API will provide a structured and secure way for users to interact with the application, reducing the risk of misuse.

5. Cryptography (General):

- Action: Integrate a cryptographic library like BouncyCastle for data encryption during storage and transmission. Implement encryption for sensitive data fields.

- Justification: Encryption will secure data against unauthorized access, ensuring compliance with international data protection laws.

6. Error Handling (DocData.java):

- Action: Enhance error handling by implementing comprehensive try-catch blocks with specific exception handling. Log errors securely without exposing sensitive information.

- Justification: Proper error handling prevents information leakage and ensures the application remains stable under error conditions.

7. Code Quality (General):

- Action: Conduct code reviews focusing on security practices, especially around API design and input handling. Consider implementing automated security testing tools.

- Justification: Improving code quality will reduce vulnerabilities and make the application more robust and user-friendly.

8. Code Error Handling (GreetingController.java):

- Action: Add error handling to the GreetingController, ensuring exceptions are caught and managed appropriately, possibly redirecting to a generic error page or logging securely.

- Justification: This will prevent application crashes and protect against information disclosure through error messages.

9. Data Exposure (GreetingController.java):

- Action: Shift from URL parameter input to form data or JSON payload submission through POST requests. Ensure all data transmission is encrypted.

- Justification: This reduces the risk of data exposure in logs and browser history, enhancing data confidentiality.

10. User Access Control (General):

- Action: Implement user authentication and authorization mechanisms. Use Spring Security or a similar framework to manage user roles and permissions.

- Justification: This will restrict access to sensitive parts of the application based on user roles, preventing unauthorized access.

Additionally, upgrading to current versions of libraries like SnakeyAML, Hibernate Validator, Apache Tomcat, and BouncyCastle will address known vulnerabilities in these components, ensuring the application uses the latest security patches and features. This comprehensive approach will significantly enhance the security posture of Artemis Financial's web application.