# CS 305 Project One Template

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **05/26/2024** | **Hannah Rose Morgenstein** | **Initial version** |

## 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

Hannah Rose Morgenstein

**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 vital for Artemis Financial as they handle sensitive financial data, including savings, retirement, investments, and insurance plans for their clients. Secure communications ensure that data confidentiality and integrity are maintained, which is crucial for preserving client trust, preventing financial fraud, and complying with regulatory requirements. Without secure communications, the company risks data breaches that could lead to significant financial and reputational damage. Implementing specific security standards such as TLS 1.3 ensures the highest level of secure communications.

*International Transactions*

Yes, Artemis Financial likely produces international transactions as part of its consulting services. These transactions necessitate robust security measures to protect data during cross-border transfers and to comply with international data protection regulations. Secure transactions ensure that client data remains protected regardless of geographical boundaries.

*Governmental Restrictions*

Artemis Financial must adhere to several governmental restrictions on secure communications. These include data protection laws such as the General Data Protection Regulation (GDPR) in the European Union, the California Consumer Privacy Act (CCPA) in the United States, and the Sarbanes-Oxley Act (SOX) for financial reporting. These regulations mandate the use of strong encryption, secure data handling practices, and regular security audits to ensure compliance.

*External Threats*

Current external threats include phishing attacks aimed at stealing sensitive financial information, SQL injection attacks to manipulate database queries, cross-site scripting (XSS) attacks to hijack user sessions, and man-in-the-middle (MITM) attacks during data transmission. In the immediate future, Artemis Financial must also be vigilant against advanced persistent threats (APTs) and zero-day vulnerabilities, which can be exploited by sophisticated attackers to gain unauthorized access to their systems.

*Modernization Requirements*

To stay ahead in the rapidly evolving technological landscape, Artemis Financial must consider the following modernization requirements:

* Open-Source Libraries: Regularly update and patch open-source libraries to protect against known vulnerabilities. Use tools such as Dependabot or Snyk to monitor and manage these dependencies.
* Evolving Web Application Technologies: Adopt secure coding practices for new web technologies such as Progressive Web Apps (PWAs), microservices, and containerization. Implement automated security testing in the development pipeline to detect vulnerabilities early.
* Security Best Practices: Implement multi-factor authentication (MFA), secure APIs, and robust encryption methods for both data in transit and at rest. Ensure secure session management and regular security training for employees.

**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.

*Architecture Review*

* Relevance: Analyzing the application architecture is crucial to understanding how different components interact and identifying potential structural weaknesses. For Artemis Financial, this involves assessing the overall design to ensure that sensitive financial data is adequately protected at every layer of the application.

*Input Validation*

* Relevance: Securing input is essential to prevent common web application attacks such as SQL injection, cross-site scripting (XSS), and other injection flaws. For Artemis Financial, input validation ensures that user-submitted data is properly sanitized and validated, protecting the application from malicious inputs that could compromise client data.

*APIs*

* Relevance: APIs are the backbone of data exchange between different services and applications. Ensuring secure API interactions prevents unauthorized access and data breaches. For Artemis Financial, securing APIs is critical as they likely use APIs to integrate various financial services and third-party tools. Specific security measures such as OAuth for authentication and rate limiting to prevent abuse should be implemented.

*Cryptography*

* Relevance: Using strong encryption for data at rest and in transit protects sensitive financial information from being intercepted or accessed by unauthorized parties. For Artemis Financial, robust cryptographic practices are necessary to secure communications and data storage, meeting both client expectations and regulatory requirements. Recommended encryption standards include AES-256 for data at rest and TLS 1.3 for data in transit.

*Client/Server*

* Relevance: Secure distributed computing ensures that the communication between the client (user) and server (backend) is protected. For Artemis Financial, this involves implementing HTTPS to secure data in transit and ensuring that the server infrastructure is hardened against attacks.

*Code Error*

* Relevance: Secure code handling involves proper error management to prevent leakage of sensitive information through error messages. For Artemis Financial, handling errors securely means ensuring that error messages do not reveal technical details that could be exploited by attackers.

*Code Quality*

* Relevance: Adhering to secure coding practices and patterns reduces the risk of introducing vulnerabilities into the codebase. For Artemis Financial, maintaining high code quality involves regular code reviews, automated testing, and following best practices for secure coding to prevent security flaws.

*Encapsulation*

* Relevance: Using secure data structures and encapsulation techniques ensures that data is protected from unauthorized access and manipulation. For Artemis Financial, encapsulation helps safeguard sensitive client information by enforcing strict access controls and data handling policies.

*Justification for Relevance:*

* *Architecture Review* ensures a robust design that addresses security from the ground up.
* *Input Validation* prevents a wide range of injection attacks that could compromise data integrity.
* *APIs* are secured to prevent unauthorized data access and ensure secure interactions between services.
* *Cryptography* protects sensitive financial data through strong encryption.
* *Client/Server security* ensures that data in transit is protected, and server infrastructure is hardened.
* *Code Error* management prevents the leakage of sensitive information through error messages.
* *Code Quality* ensures that the application is free from common vulnerabilities through secure coding practices.
* *Encapsulation* protects data from unauthorized access and manipulation through strict access controls.

By focusing on these areas of security, Artemis Financial can ensure that its web application is resilient against current and emerging threats, thereby protecting sensitive client data, and maintaining compliance with regulatory standards.

**3. Manual Review**

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

*Insecure Database Connection Handling:*

* Description: The database connection is established using hardcoded credentials and is not closed properly, leading to resource leakage and potential exposure of sensitive information.
* Recommendation: Utilize environment variables or a secure vault to store database credentials securely. Ensure database connections are properly closed using a try-with-resources statement or in a finally block.

*Potential SQL Injection:*

* Description: Although not fully implemented, any SQL operations should use prepared statements to prevent SQL injection.
* Recommendation: Always use prepared statements for SQL queries to avoid SQL injection attacks.

*Insufficient Input Validation:*

* Description: The name parameter in the read endpoint is not validated, which could lead to injection attacks or other security issues.
* Recommendation: Validate and sanitize user inputs in the CRUDController to ensure they meet expected formats and constraints.

*Hardcoded Strings:*

* Description: Strings such as database credentials and greeting templates are hardcoded, which is insecure.
* Recommendation: Externalize sensitive information such as database credentials and greeting templates to configuration files or environment variables.

*Missing Access Modifiers:*

* Description: The instance variables in customer.java and myDateTime.java lack access modifiers, making them package-private by default.
* Recommendation: Use appropriate access modifiers (e.g., private) in customer.java and myDateTime.java to encapsulate data and ensure proper access control.

*Improper Naming Conventions:*

* Description: Class names should follow Java naming conventions, starting with an uppercase letter.
* Recommendation: Rename classes to follow Java naming conventions (e.g., Customer and MyDateTime).

*Lack of Concurrency Handling:*

* Description: The AtomicLong counter is thread-safe, but any future stateful operations could introduce concurrency issues.
* Recommendation: Ensure thread safety for shared mutable state by using appropriate synchronization mechanisms in GreetingController. Consider using Java concurrency utilities such as ‘synchronized’ blocks or the ‘java.util.concurrent’ package.

*Unimplemented Methods:*

* Description: The methods retrieveDateTime and setMyDateTime in myDateTime.java are stubs and lack implementation.
* Recommendation: Implement these methods or remove them if they are not needed to avoid confusion and maintain clean code.

*Lack of Error Handling:*

* Description: The catch block in DocData.java only prints the stack trace without proper error handling or logging.
* Recommendation: Implement proper error handling and logging mechanisms in DocData.java to prevent the exposure of sensitive information.

*Insufficient Logging:*

* Description: There is a lack of logging for operations, especially those involving significant actions like handling requests.
* Recommendation: Implement logging throughout the codebase to record significant events, errors, and security incidents, using a secure logging framework.

*Prioritization:*

Prioritizing vulnerabilities based on their severity and potential impact is crucial to effectively allocate resources for remediation efforts. Here's a prioritized list of vulnerabilities identified in the codebase:

1. Potential SQL Injection: This vulnerability poses a high risk as it could lead to unauthorized access to the database and manipulation of data, potentially compromising the integrity and confidentiality of sensitive information.
2. Insecure Database Connection Handling: While not immediately exploitable, this vulnerability presents a significant risk due to the potential exposure of sensitive information, such as database credentials, if exploited.
3. Insufficient Input Validation: Although this vulnerability may not directly lead to a security breach, it could be exploited in combination with other vulnerabilities to launch attacks such as SQL injection or cross-site scripting (XSS).
4. Hardcoded Strings: While insecure, this vulnerability may have a lower impact compared to others on the list as it primarily poses a risk of exposing sensitive information if the codebase is accessed by unauthorized parties.

*Risk Assessment:*

Providing a brief risk assessment for each vulnerability helps understand its potential impact on the application and its users. Here's a summary of the risk assessment for the identified vulnerabilities:

* Potential SQL Injection: High risk; could lead to unauthorized access to sensitive data and compromise data integrity.
* Insecure Database Connection Handling: Moderate risk; potential exposure of sensitive information such as database credentials.
* Insufficient Input Validation: Moderate risk; could lead to injection attacks or other security issues if exploited.
* Hardcoded Strings: Low risk; primarily poses a risk of exposing sensitive information but may not directly lead to a security breach.

*False Positives*

Identifying false positives is crucial to focus on actual vulnerabilities. While reviewing the code, it's essential to differentiate between true vulnerabilities and instances that may appear risky but do not pose a real threat. Here are some examples of identified false positives and their justifications:

* Unused Code: Certain pieces of code may appear vulnerable but are not used in the application, posing no real risk.
* Overly Strict Validation: Some validations may seem insufficient but are adequate given the context and actual usage patterns.
* Deprecated Libraries: While flagged as risky, deprecated libraries might not pose an immediate threat if not used in a vulnerable context.

*Summary of Findings*

In summary, the key vulnerabilities identified in the code base include insecure database handling, potential SQL injection, insufficient input validation, hardcoded strings, missing access modifiers, improper naming conventions, lack of concurrency handling, unimplemented methods, lack of error handling, and insufficient logging. Addressing these issues will enhance the security, maintainability, and robustness of Artemis Financials’ web application.

**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 Check Findings:*

*bcprov-jdk15on-1.46.jar*

* Vulnerability IDs: [High Severity] 22 vulnerabilities
* Description: The Bouncy Castle Crypto Package contains multiple high-severity vulnerabilities.
* Recommendation: Update to a patched version of the library to mitigate these vulnerabilities.

*hibernate-validator-6.0.18.Final.jar*

* Vulnerability IDs: [Medium Severity] 1 vulnerability
* Description: The Hibernate Validator library contains a medium-severity vulnerability.
* Recommendation: Update to a patched version of the library to mitigate this vulnerability.

*snakeyaml-1.25.jar*

* Vulnerability IDs: [Critical Severity] 8 vulnerabilities
* Description: The SnakeYAML library contains multiple critical-severity vulnerabilities.
* Recommendation: Update to a patched version of the library to mitigate these vulnerabilities.

*spring-core-5.2.3.RELEASE.jar*

* Vulnerability IDs: [Critical Severity] 11 vulnerabilities
* Description: The Spring Framework contains multiple critical-severity vulnerabilities.
* Recommendation: Update to a patched version of the library to mitigate these vulnerabilities.

*spring-web-5.2.3.RELEASE.jar*

* Vulnerability IDs: [Critical Severity] 14 vulnerabilities
* Description: The Spring Framework contains multiple critical-severity vulnerabilities.
* Recommendation: Update to a patched version of the library to mitigate these vulnerabilities.

*tomcat-embed-core-9.0.30.jar*

* Vulnerability IDs: [Critical Severity] 26 vulnerabilities
* Description: The Apache Tomcat library contains multiple critical-severity vulnerabilities.
* Recommendation: Update to a patched version of the library to mitigate these vulnerabilities.

*tomcat-embed-websocket-9.0.30.jar*

* Vulnerability IDs: [Critical Severity] 27 vulnerabilities
* Description: The Apache Tomcat library contains multiple critical-severity vulnerabilities.
* Recommendation: Update to a patched version of the library to mitigate these vulnerabilities.

*logback-core-1.2.3.jar*

* Vulnerability IDs: [High Severity] 2 vulnerabilities
* Description: Logback Core version 1.2.3 contains two high-severity vulnerabilities.
* Recommendation: Update to Logback Core version 1.2.4 or later to mitigate these vulnerabilities.

*jackson-databind-2.10.2.jar*

* Vulnerability IDs: [High Severity] 6 vulnerabilities
* Description: The Jackson Databind library contains multiple high-severity vulnerabilities.
* Recommendation: Update to a patched version of the library to mitigate these vulnerabilities.

*log4j-api-2.12.1.jar*

* Vulnerability IDs: [Low Severity] 1 vulnerability
* Description: Log4j API version 2.12.1 contains one low-severity vulnerability.
* Recommendation: Update to a patched version of the library to mitigate this vulnerability.

*spring-boot-starter-web-2.2.4.RELEASE.jar*

* Vulnerability IDs: [Critical Severity] 3 vulnerabilities
* Description: The Spring Boot Starter Web library contains multiple critical-severity vulnerabilities.
* Recommendation: Update to a patched version of the library to mitigate these vulnerabilities.

**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.

*Insecure Database Connection Handling*

* Action: Utilize environment variables or a secure vault to store database credentials securely.
* Action: Ensure database connections are properly closed using a try-with-resources statement or in a finally block.

*Potential SQL Injection*

* Action: Implement prepared statements for SQL queries to prevent SQL injection attacks.

*Insufficient Input Validation*

* Action: Validate and sanitize user inputs in the CRUDController to ensure they meet expected formats and constraints.

*Hardcoded Strings*

* Action: Externalize sensitive information such as database credentials and greeting templates to configuration files or environment variables.

*Missing Access Modifiers*

* Action: Use appropriate access modifiers (e.g., private) in customer.java and myDateTime.java to encapsulate data and ensure proper access control.

*Improper Naming Conventions*

* Action: Rename classes to follow Java naming conventions (e.g., Customer and MyDateTime).

*Lack of Concurrency Handling*

* Action: Ensure thread safety for shared mutable state by using appropriate synchronization mechanisms in GreetingController. Consider using Java concurrency utilities such as ‘synchronized’ blocks or the ‘java.util.concurrent’ package.

*Unimplemented Methods*

* Action: Implement the retrieveDateTime and setMyDateTime methods in myDateTime.java or

remove them if unnecessary.

*Lack of Error Handling*

* Action: Implement proper error handling and logging mechanisms in DocData.java to prevent the exposure of sensitive information.

*Insufficient Logging*

* Action: Implement logging throughout the codebase to record significant events, errors, and security incidents, using a secure logging framework.

*Dependency Vulnerabilities*

* Action: Update vulnerable dependencies such as Bouncy Castle Crypto Package, Hibernate Validator, SnakeYAML, Spring Framework, and Apache Tomcat to patched versions to mitigate known vulnerabilities.

*Prioritized mitigation actions to help focus on the most critical vulnerabilities first:*

1. Critical vulnerabilities in dependencies (e.g., Bouncy Castle Crypto Package, SnakeYAML, Spring Framework, and Apache Tomcat).
2. Insecure database connection handling and potential SQL injection.
3. Insufficient input validation and hardcoded strings.
4. Missing access modifiers and improper naming conventions.
5. Concurrency handling, unimplemented methods, lack of error handling, and insufficient logging.

*Timeline for implementing these actions to provide a clear roadmap for addressing the issues:*

* Critical dependencies update: Within 1 week.
* Database connection handling and SQL injection prevention: Within 2 weeks.
* Input validation and hardcoded strings externalization: Within 3 weeks.
* Access modifiers and naming conventions: Within 1 month.
* Concurrency handling, method implementation, error handling, and logging: Within 1.5 months.

By addressing these vulnerabilities and implementing the recommended actions, Artemis Financial can significantly enhance the security, maintainability, and robustness of its web application, ensuring the protection of sensitive client data and compliance with security best practices and regulations.