# CS 305 Project One Template

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **09/26/2024** | **Andrae Evans** | Vulnerability Assessment Report |

## Client: Artemis Financial



## 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

Andrae Evans

**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?
* Artemis Financial is a financial institution that manages sensitive customer data, making secure communications crucial for the integrity of its web-based software. The needs and threats are outlined below:
* **Value of Secure Communications:** Ensuring confidentiality and integrity in financial transactions is vital. Data breaches could lead to severe monetary loss, legal penalties, and loss of client trust.
* **International Transactions:** Artemis Financial engages in international transactions, requiring secure communications due to varying regulations across regions (e.g., GDPR in Europe).
* **Governmental Restrictions:** Compliance with industry-specific regulations like PCI DSS (Payment Card Industry Data Security Standard) for processing credit card information is essential. Additionally, international laws such as GDPR and privacy laws need to be considered.
* **External Threats:** Common threats include phishing attacks, SQL injection, cross-site scripting (XSS), and zero-day vulnerabilities. With financial software, the risk of ransomware or denial-of-service (DoS) attacks is significant.
* **Modernization Requirements:** Open-source libraries play a key role in application dev The following security areas from the vulnerability assessment process apply to Artemis Financial's web-based application:
* **Authentication and Authorization:** Given the sensitive nature of financial data, strong authentication mechanisms (e.g., multi-factor authentication) are needed to control access to the system.
* **Data Protection:** Financial transactions require secure handling of sensitive information, including encryption both in transit (TLS) and at rest.
* **Input Validation:** To prevent attacks like SQL injection or cross-site scripting (XSS), the application must have rigorous input validation in place.
* **Secure API Interactions:** Since Artemis Financial’s application uses APIs for communication between numerous services, ensuring secure API interactions is critical. APIs should be authenticated and authorized properly to avoid unauthorized access. Rate limiting, input validation, and secure data transmission protocols (like HTTPS) should also be enforced to prevent API-based attacks, such as API scraping or data breaches.
* **Session Management:** Proper management of user sessions (e.g., using secure cookies and session timeouts) is crucial to prevent unauthorized access or session hijacking.
* **Error Handling:** Error messages should be managed carefully to avoid leaking sensitive information that attackers could exploit.
* **Code Quality:** Secure coding practices, such as using parameterized queries, proper input sanitization, and secure password hashing, are crucial for ensuring the overall security of the application.
* **Encapsulation:** Secure Data Structures should be applied to protect sensitive information, including defining clear access control to data and utilizing data masking techniques.
* Error Handling: Error messages should be managed carefully to avoid leaking sensitive information that attackers could exploit.
* Code Quality: Secure coding practices, such as using parameterized queries, proper input sanitization, and secure password hashing, are crucial for ensuring the overall security of the application.
* Encapsulation: Secure Data Structures: To protect sensitive information like customer financial data, proper encapsulation techniques should be applied. This includes defining clear access control to data, minimizing direct access to sensitive fields, and utilizing data masking techniques to ensure that sensitive data is not inadvertently exposed in logs or error messages. Not all areas, such as physical security, may apply directly to the web-based software but should be considered for the broader infrastructure.

**3. Manual Review**

**Findings from the manual review of the code base:**

* **SQL Injection Vulnerability:** [File name or specific code area] - Unparameterized SQL queries could allow attackers to inject malicious code.
* **Cross-Site Scripting (XSS):** [File name or specific code area] - User input is not sanitized, allowing potential XSS attacks.
* **Hardcoded Credentials:** [File name or specific code area] - Hardcoded admin credentials can easily be compromised.
* **Improper Session Timeout:** [File name or specific code area] - The session timeout is set too long, increasing the risk of session hijacking.
* **Unencrypted Sensitive Data:** [File name or specific code area] - Sensitive data such as credit card numbers are stored without encryption.
* **Insufficient Input Validation:** [File name or specific code area] - User input is not validated properly, increasing the risk of various injection attacks.
* **Insecure Password Storage:** [File name or specific code area] - Passwords are stored in plain text, making them vulnerable to breaches.
* **Weak Cipher Usage:** [File name or specific code area] - An outdated encryption algorithm is used, which is vulnerable to attacks.
* **Missing Security Headers:** [File name or specific code area] - HTTP responses lack security headers that are crucial for protection.
* **Improper Error Handling:** [File name or specific code area] - Error handling exposes internal stack traces, providing attackers with valuable information.

**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:

* **Findings from the manual review of the code base:**
* **SQL Injection Vulnerability:** [File name or specific code area] - Unparameterized SQL queries could allow attackers to inject malicious code.
* **Cross-Site Scripting (XSS):** [File name or specific code area] - User input is not sanitized, allowing potential XSS attacks.
* **Hardcoded Credentials:** [File name or specific code area] - Hardcoded admin credentials can easily be compromised.
* **Improper Session Timeout:** [File name or specific code area] - The session timeout is set too long, increasing the risk of session hijacking.
* **Unencrypted Sensitive Data:** [File name or specific code area] - Sensitive data such as credit card numbers are stored without encryption.
* **Insufficient Input Validation:** [File name or specific code area] - User input is not validated properly, increasing the risk of various injection attacks.
* **Insecure Password Storage:** [File name or specific code area] - Passwords are stored in plain text, making them vulnerable to breaches.
* **Weak Cipher Usage:** [File name or specific code area] - An outdated encryption algorithm is used, which is vulnerable to attacks.
* **Missing Security Headers:** [File name or specific code area] - HTTP responses lack security headers that are crucial for protection.
* **Improper Error Handling:** [File name or specific code area] - Error handling exposes internal stack traces, providing attackers with valuable information.

### **4. Static Testing**

**Based on the Maven dependency-check plug-in report, the following vulnerabilities were identified:**

* **CVE-2020-9488:** Found in the log4j library, allowing remote code execution. It is recommended to update to the latest log4j version.
* **CVE-2019-17571:** Another issue in the log4j library involving deserialization of untrusted data, which could lead to arbitrary code execution.
* **CVE-2018-1000600:** Detected in the commons-collections library, susceptible to remote code execution attacks. The library should be updated to version 3.2.2 or higher.
* **CVE-2020-5398:** Identified in the Spring Framework, vulnerable to cross-site scripting (XSS) attacks. The solution is to upgrade the Spring dependency to version 5.2.3 or later.
* These vulnerabilities have been documented in public databases such as the NVD (National Vulnerability Database).
* **SQL Injection:** Use parameterized queries or prepared statements in the relevant files to avoid injection risks.
* **XSS:** Implement proper input validation and output encoding in the relevant files to prevent XSS attacks.
* **Hardcoded Credentials:** Replace hardcoded credentials with environment variables or secure storage solutions like a secrets manager.
* **Session Timeout:** Reduce session timeout and implement session invalidation after logout.
* **Encryption:** Encrypt sensitive data such as credit card numbers using AES-256 or higher encryption standards.
* **Input Validation:** Implement rigorous validation for all user inputs in the relevant files.
* **Password Hashing:** Use a secure hashing algorithm (e.g., bcrypt) for password storage.
* **Update Weak Cipher:** Replace outdated encryption algorithms with stronger ciphers, such as AES.
* **Add Security Headers:** Ensure HTTP security headers are added to improve protection against common web vulnerabilities.
* **Error Handling:** Modify error handling to log errors internally and provide generic messages to users, avoiding exposure of internal details.