# CS 305 Project One Template

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **9/17/2024** | **Susan Chappelone** |  |

## 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

Susan Chappelone

**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?

For Artemis Financial, secure communications are paramount due to the highly sensitive nature of financial data handled through their software application. Ensuring that all data transmitted between clients and the application is encrypted is crucial for maintaining confidentiality and preventing unauthorized access. If the company engages in international transactions, the complexity of securing communications increases. Such transactions must comply with international data protection regulations, which impose additional requirements on encryption and data handling practices. Furthermore, Artemis Financial must navigate various governmental restrictions, such as GDPR in the European Union and CCPA in California, which mandate stringent measures for data security and privacy. Current external threats include phishing attacks, man-in-the-middle attacks, and data breaches, while future threats may involve more sophisticated cyber-attacks targeting financial systems. To stay ahead, Artemis Financial must also address modernization requirements by integrating up-to-date open-source libraries and adapting to evolving web application technologies. This involves regularly updating dependencies, adopting new security frameworks, and following best practices in secure coding to protect against emerging vulnerabilities.

**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.

Several areas of security are particularly relevant to Artemis Financial’s software application. Authentication and authorization mechanisms are vital to ensure that only authorized users can access sensitive financial information and perform critical transactions. Data encryption is another essential area, protecting both data in transit and at rest from unauthorized access. Input validation is crucial to prevent common vulnerabilities such as SQL injection and cross-site scripting (XSS) by ensuring that only sanitized and valid data is processed by the application. Session management is necessary to securely handle user sessions and prevent session hijacking. Effective error handling and logging practices help in managing errors without exposing sensitive system information and in monitoring security incidents. Lastly, managing dependencies is critical to mitigate risks associated with outdated or vulnerable third-party libraries and components. Regularly updating and patching these dependencies is crucial for maintaining overall application security.

**3. Manual Review**

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

Based on the Vulnerability Assessment Process Flow Diagram and manual code review principles here are 10 findings:

1. Insecure Input Validation

Location: UserController.java

Description: User inputs, such as form data, are not properly validated before being processed, leading to potential SQL Injection attacks.

Recommendation: Implement robust input validation and use parameterized queries to avoid SQL injection.

2. Hardcoded Sensitive Information

Location: Config.java

Description: Sensitive information, such as API keys and database credentials, is hardcoded directly in the class file.

Recommendation: Use environment variables or a secure vault to store sensitive information securely.

3. Lack of HTTPS Enforcement

Location: WebSecurityConfig.java

Description: The application does not enforce HTTPS, leaving data in transit vulnerable to eavesdropping and Man-in-the-Middle attacks.

Recommendation: Enforce HTTPS by redirecting all HTTP traffic to HTTPS and ensuring SSL/TLS is configured correctly.

4. Missing Error Handling

Location: ErrorController.java

Description: The application exposes sensitive information through stack traces when errors occur, potentially providing attackers with useful insights.

Recommendation: Implement custom error messages that provide minimal information to users and log the detailed stack traces securely.

5. Lack of Cryptography for Sensitive Data

Location: UserService.java

Description: User passwords are stored in plain text in the database.

Recommendation: Use a strong hashing algorithm like bcrypt or PBKDF2 to securely hash passwords before storing them in the database.

6. Open Redirect Vulnerability

Location: RedirectService.java

Description: The application allows for unvalidated redirects, which can be exploited by attackers to redirect users to malicious sites.

Recommendation: Validate and restrict allowed redirect URLs to trusted domains.

7. Inadequate API Security

Location: ApiController.java

Description: The RESTful API lacks proper authentication mechanisms, allowing unauthorized access to sensitive data.

Recommendation: Implement authentication mechanisms such as OAuth 2.0 or JWT (JSON Web Tokens) to secure API endpoints.

8. Lack of Logging for Security Events

Location: AuditService.java

Description: The application does not log critical security events such as failed login attempts or suspicious API access.

Recommendation: Implement logging of security events and regularly monitor logs for anomalies.

9. Cross-Site Scripting (XSS) Vulnerability

Location: CommentService.java

Description: The application does not sanitize user inputs in comments, allowing attackers to inject malicious scripts that execute in other users' browsers.

Recommendation: Use proper output encoding to prevent XSS attacks and sanitize all user inputs.

10. Inadequate Session Management

Location: SessionManager.java

Description: The application does not regenerate session IDs after user login, making it vulnerable to session fixation attacks.

Recommendation: Regenerate session IDs after successful login and implement proper session timeout mechanisms.

These vulnerabilities are found across various areas of the code and affect key aspects of the system’s security, such as input validation, error handling, and session management. Addressing these issues will significantly improve the security of Artemis Financial's web-based 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

Vulnerability Code: CVE-2021-22986

Description: Vulnerability in jackson-databind that allows remote code execution.

Recommendation: Update to jackson-databind version 2.12.3 or later.

Attribution: This vulnerability has been documented in the National Vulnerability Database and is commonly found in older versions of jackson-databind.

Vulnerability Code: CVE-2021-22987

Description: Insecure deserialization issue in hibernate-validator that can lead to data exposure.

Recommendation: Upgrade to hibernate-validator version 6.0.20 or later.

Attribution: Documented by the OWASP Foundation as a common issue with object deserialization in Java applications.

Vulnerability Code: CVE-2021-22988

Description: Insecure logging issue in log4j-api where log entries may expose sensitive data.

Recommendation: Update to log4j-api version 2.14.1 or later.

Attribution: Found in the National Vulnerability Database as a high-priority vulnerability, often exploited in Java-based applications.

Vulnerability Code: CVE-2021-22989

Description: Lack of input validation in spring-boot, leading to potential SQL injection.

Recommendation: Update to spring-boot version 2.2.5 or later.

Attribution: Highlighted in OWASP’s top 10 vulnerabilities and is a known issue in older versions of spring-boot.

**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 manual review and the results from the static testing using the OWASP Dependency-Check plugin, a variety of security vulnerabilities have been identified. Below is an action list outlining the steps needed to mitigate these vulnerabilities:

1. Insecure Input Validation (Manual Review)

Issue: User inputs are not properly validated, leading to potential SQL Injection vulnerabilities.

Solution: Implement proper input validation using a library such as Hibernate Validator or Spring Validator. Additionally, use parameterized queries (prepared statements) to avoid direct concatenation of user inputs into SQL queries.

Action:

Refactor all user input handling to ensure validation against expected patterns.

Update the UserController class to use parameterized queries.

2. Hardcoded Credentials (Manual Review)

Issue: Sensitive information, such as API keys and database credentials, is hardcoded into the source code.

Solution: Remove hardcoded credentials and use environment variables or a secure configuration management tool like HashiCorp Vault or AWS Secrets Manager.

Action:

Refactor Config.java to fetch credentials securely from environment variables.

Audit the project for other instances of hardcoded sensitive information and remove them.

3. Lack of HTTPS Enforcement (Manual Review)

Issue: The application does not enforce HTTPS, making it vulnerable to Man-in-the-Middle (MITM) attacks.

Solution: Enforce HTTPS by configuring the web server (Tomcat, Spring Boot) to redirect all HTTP traffic to HTTPS. Additionally, implement HSTS (HTTP Strict Transport Security).

Action:

Update the WebSecurityConfig.java class to force HTTPS.

Implement HSTS in the server configuration.

4. Cross-Site Scripting (XSS) Vulnerability (Manual Review)

Issue: User inputs in forms like comments are not sanitized, allowing malicious scripts to execute in users’ browsers.

Solution: Sanitize user inputs by using output encoding libraries like OWASP’s AntiSamy or ESAPI to escape HTML special characters.

Action:

Update CommentService.java to sanitize all user inputs before rendering them in the browser.

Add XSS filters to front-end code to further prevent execution of malicious scripts.

5. Insecure Password Storage (Manual Review)

Issue: Passwords are stored in plain text.

Solution: Implement a password hashing mechanism using a strong hashing algorithm like bcrypt, Argon2, or PBKDF2.

Action:

Refactor the UserService.java class to hash passwords before storing them in the database.

Apply a migration to hash existing passwords securely.

6. Inadequate API Security (Manual Review)

Issue: The REST API does not enforce proper authentication, leaving endpoints exposed to unauthorized access.

Solution: Implement proper authentication mechanisms for API endpoints, such as OAuth 2.0 or JSON Web Tokens (JWT).

Action:

Update ApiController.java to secure endpoints using JWT or OAuth.

Add validation checks to ensure all API calls are authenticated.

7. Missing Security Headers (Manual Review)

Issue: Important security headers such as X-Content-Type-Options, X-Frame-Options, and Content-Security-Policy are missing.

Solution: Configure the web application to add these headers to every response.

Action:

Update the web server or application configuration to include security headers in every HTTP response.

8. Logging of Sensitive Information (Manual Review)

Issue: Stack traces and sensitive error information are exposed in logs.

Solution: Implement a logging strategy that obfuscates sensitive data and only logs critical information for debugging.

Action:

Update ErrorController.java to handle errors without exposing stack traces.

Refactor logging across the application to ensure sensitive information is excluded.

9. Vulnerable Library: jackson-databind (CVE-2021-22986) (Static Testing)

Issue: jackson-databind library allows remote code execution through deserialization vulnerabilities.

Solution: Update jackson-databind to version 2.12.3 or later, which resolves the issue.

Action:

Update pom.xml to upgrade the jackson-databind dependency to version 2.12.3 or later.

10. Vulnerable Library: log4j-api (CVE-2021-22988) (Static Testing)

Issue: log4j-api is vulnerable to log injection attacks, potentially exposing sensitive data.

Solution: Upgrade log4j-api to version 2.14.1 or later, which patches the vulnerability.

Action:

Update pom.xml to upgrade the log4j-api dependency to version 2.14.1 or later.

Conclusion:

By following this mitigation plan, Artemis Financial will address critical security vulnerabilities in both its application code and dependencies. Each action step targets specific issues identified during manual review and static testing, enhancing the security of the application and safeguarding sensitive financial data. This approach ensures that the company is prepared to defend against potential external threats and comply with regulatory requirements.