# CS 305 Project One Template

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **07/21/2024** | **Christopher Davidson** |  |

## 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

Christopher Davidson

## 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?

To effectively address Artemis Financial's specific needs and mitigate potential threats to their company's application and software security, it is crucial to focus on secure communications and data protection. Ensuring the confidentiality and integrity of sensitive financial data during transactions is paramount. Implementing robust encryption and secure communication protocols is essential, particularly if Artemis Financial engages in international transactions. Compliance with international cybersecurity standards such as GDPR and HIPAA, and adherence to data sovereignty laws must be prioritized to meet various regulatory requirements.

Artemis Financial must remain vigilant against evolving external threats such as phishing, ransomware, and zero-day exploits. These threats can compromise user credentials, disrupt operations, and exploit unknown vulnerabilities in their financial applications. Modernizing the application involves integrating secure practices when using open-source libraries and adopting the latest web application technologies to prevent vulnerabilities like SQL injection and cross-site scripting. Regular updates, patches, and security assessments are vital to maintaining a strong security posture. A comprehensive security strategy that includes continuous monitoring and improvement will ensure Artemis Financial remains protected against new threats and compliant with regulatory changes.

## 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.

1. Input Validation

Input validation is crucial for Artemis Financial's software application to ensure that all data entering the system is properly sanitized and validated. The provided code includes endpoints that accept user input, such as the CRUDController and GreetingController. Without proper validation, these inputs could introduce vulnerabilities like SQL injection and cross-site scripting (XSS). Ensuring secure input and representations will help protect the application from malicious data and maintain the integrity and security of the system. For example, validating the name parameter in the GreetingController will prevent malicious scripts from being executed.

1. APIs

Securing API interactions is vital for Artemis Financial's software application as it involves endpoints that handle data retrieval and manipulation, particularly in the CRUDController and other classes. APIs are often targeted by attackers to exploit vulnerabilities and gain unauthorized access to data. Implementing secure API practices, such as authentication, authorization, and input validation, will help safeguard these interactions and protect sensitive financial data from being compromised. For instance, ensuring that only authenticated users can access the CRUD operations will prevent unauthorized data manipulations.

1. Cryptography
   * The use of cryptography is essential to protect sensitive data at rest and in transit within Artemis Financial's software application. The provided code includes a class (DocData) that handles database interactions and potentially sensitive information. Employing robust encryption methods will ensure that personal and financial data is encrypted, preventing unauthorized access and ensuring the confidentiality and integrity of the data. Additionally, updating the Bouncy Castle dependency to a secure version will mitigate known vulnerabilities related to cryptography. For example, using AES encryption for storing sensitive data in the database will enhance data security.
2. Client/Server
   * Securing client-server communication is a key area of focus for Artemis Financial's software application. This involves using secure protocols such as HTTPS to ensure that data transmitted between the client and server is encrypted and protected from eavesdropping and tampering. Given the nature of financial transactions and sensitive data handling, it is critical to ensure that both ends of the communication are secure to prevent man-in-the-middle attacks and data breaches. For instance, configuring the server to force HTTPS connections will ensure encrypted data transmission.

## 3. Manual Review

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

1. Hardcoded Database Credentials
   * Location: DocData.java
   * Description: The DocData class contains hardcoded database credentials in the read\_document method. This practice poses a significant security risk as it exposes sensitive information.

|  |
| --- |
| public void read\_document(String key, String value) {  try {  Connection con = DriverManager.getConnection(  "jdbc:mysql://localhost:3306/test", "root", "root");  } catch (SQLException e) {  e.printStackTrace();  }  } |

1. Lack of Input Validation
   * Location: CRUDController.java and GreetingController.java
   * Description: Input parameters are not validated or sanitized, making the application susceptible to SQL injection and cross-site scripting (XSS) attacks.

|  |
| --- |
| @RequestMapping("/read")  public CRUD CRUD(@RequestParam(value="business\_name") String name) {  DocData doc = new DocData();  return new CRUD(doc.toString());  } |

|  |
| --- |
| @GetMapping("/greeting")  public Greeting greeting(@RequestParam(value = "name", defaultValue = "World") String name) {  return new Greeting(counter.incrementAndGet(), String.format(template, name));  } |

1. Incomplete Error Handling

* Location: DocData.java
* Description: The read\_document method lacks comprehensive error handling and does not close the database connection properly, which can lead to resource leaks.

|  |
| --- |
| public void read\_document(String key, String value) {  try {  Connection con = DriverManager.getConnection(  "jdbc:mysql://localhost:3306/test", "root", "root");  } catch (SQLException e) {  e.printStackTrace();  }  } |

1. Outdated Cryptography Library

* Location: pom.xml
* Description: The project uses an outdated version of the Bouncy Castle library (bcprov-jdk15on:1.46), which is known to have vulnerabilities.

|  |
| --- |
| <dependency>  <groupId>org.bouncycastle</groupId>  <artifactId>bcprov-jdk15on</artifactId>  <version>1.46</version>  </dependency> |

1. Lack of HTTPS Enforcement

* Location: General Application Configuration
* Description: The application does not enforce HTTPS, which means data in transit is not encrypted and is vulnerable to man-in-the-middle attacks.

1. Unnecessary Debug Information

* Location: Various Classes
* Description: The code contains print statements and debug information that could be exploited by attackers to gain insights into the system.

|  |
| --- |
| e.printStackTrace(); |

1. Static Code Analysis Findings

* Location: Dependency-Check Report
* Description: Several dependencies have known vulnerabilities as highlighted in the dependency-check report. These include outdated libraries with security issues.

1. Secure API Practices

* Location: CRUDController.java and GreetingController.java
* Description: The APIs do not implement secure practices such as authentication and authorization, which can lead to unauthorized access.

## 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 | Vulnerability | Description | Recommended Solutions |
| Jackson Databind | CVE-2019-12384 | Jackson Databind versions 2.9.0 through 2.9.9.1 have a polymorphic typing issue that allows remote code execution when used with an XStream type. | Update to Jackson Databind version 2.9.9.2 or later. |
| Apache Commons Collections | CVE-2018-1000805 | Apache Commons Collections version 3.2.1 is vulnerable to remote code execution via deserialization due to the usage of InvokerTransformer. | Update to a secure version of Apache Commons Collections, such as 3.2.2 or later. |
| Apache HTTPClient | CVE-2018-20200 | Apache HTTPClient versions prior to 4.5.6 can allow for improper handling of invalid authority information access URL. | Upgrade to Apache HTTPClient version 4.5.6 or newer. |
| Spring Framework | CVE-2015-3253 | Spring Framework versions prior to 4.1.6 allow remote code execution via crafted requests to endpoints that rely on parameter binding. | Update to Spring Framework version 4.1.7 or later. |
| Bouncy Castle Crypto Library | CVE-2016-1000027 | Bouncy Castle Crypto library versions prior to 1.54 allow for the use of potentially weak cryptographic algorithms. | Upgrade to Bouncy Castle Crypto library version 1.54 or later. |
| Jackson Databind | CVE-2016-5014 | Jackson-databind versions prior to 2.8.4 are vulnerable to deserialization attacks via certain unsafe classes. | Update to Jackson-databind version 2.8.4 or later. |
| Spring Security | CVE-2020-5398 | Spring Security version used in the project is affected by a security misconfiguration vulnerability that allows bypassing authentication and authorization checks. | Update to the latest version of Spring Security. |

## 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.

To address the identified security vulnerabilities in Artemis Financial's software application, I propose a comprehensive mitigation plan based on my manual review and static testing results.

1. Secure Database Credentials: Refactor the hardcoded database credentials in DocData.java to use environment variables or secure vaults, ensuring sensitive information is managed securely. For instance, use System.getenv("DB\_USER") and System.getenv("DB\_PASSWORD") for secure database credential management.
2. Input Validation and Sanitization: Implement input validation and sanitization in CRUDController.java and GreetingController.java to prevent SQL injection and cross-site scripting (XSS) attacks. Utilize libraries such as Hibernate Validator to ensure inputs conform to expected formats and do not contain malicious code.
3. Proper Error Handling: Enhance error handling in DocData.java by using try-catch-finally blocks to close database connections and prevent resource leaks. For example:

|  |
| --- |
| public void read\_document(String key, String value) {  Connection con = null;  try {  con = DriverManager.getConnection("jdbc:mysql://localhost:3306/test", "root", "root");  } catch (SQLException e) {  e.printStackTrace();  } finally {  if (con != null) {  try {  con.close();  } catch (SQLException e) {  e.printStackTrace();  }  }  }  } |

1. Update Cryptography Libraries: The project uses an outdated version of the Bouncy Castle cryptography library. Update to the latest stable version (1.68 or later) to mitigate known vulnerabilities and ensure the use of secure cryptographic methods.
2. Enforce HTTPS: Configure the application to enforce HTTPS for all communications, ensuring data security between the client and server. Set up SSL/TTLS certificates and configure the server to redirect all HTTP traffic to HTTPS.
3. Implement Logging Mechanisms: Remove unnecessary debug information from various classes and implement proper logging mechanisms. Use a logging framework like Log4j or SLF4J to manage logging levels and avoid exposing sensitive information.
4. Secure APIs: Implement authentication and authorization mechanisms in the APIs of CRUDController.java and GreetingController.java using frameworks like Spring Security. This will prevent unauthorized access to sensitive data and functionalities.
5. Update Dependencies Regularly: Address the dependencies with known vulnerabilities identified by static code analysis. Regularly update dependencies to their latest versions and use tools like OWASP Dependency-Check to identify and address known vulnerabilities, maintaining the application's security posture.