# CS 305 Project One Template

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **3/20/25** | **Dustin Davis** |  |

## 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

Dustin Davis

**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 handles sensitive financial information for their clients, like retirement plans, investments, and insurance details. Secure communication is essential because any leak or unauthorized access could seriously damage client trust and the company’s reputation. It's important that all data moving through their system stays private and protected at all times.

Since Artemis works with clients around the world, international transactions are likely part of their operations. That means they need to be aware of global data regulations and make sure that their communications are encrypted and compliant with international standards.

Because financial data is highly regulated, there are definitely government restrictions to keep in mind. Depending on where their clients are located, Artemis may need to follow specific security standards, like PCI compliance or data privacy laws such as GDPR, to ensure all sensitive information is handled properly.

They face threats like data breaches, phishing attacks, insecure endpoints, and vulnerabilities in third-party tools or APIs. As they continue to modernize, new technologies could introduce fresh attack surfaces, so it's important to stay on top of the latest threats and best practices.

Modernizing the system will probably involve using open-source libraries and newer web technologies. While this helps speed up development and adds flexibility, it also introduces a risk if those libraries aren’t regularly updated. Keeping dependencies secure and staying current with modern security practices is going to be essential.

**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.

There are a few specific areas of security that stand out for Artemis Financial’s web application. Based on how the application is set up and what it does, some areas in the vulnerability assessment process flow are more relevant than others.

Input validation is definitely important here. Since the application accepts user input through request parameters (like the name field in the greeting endpoint or the business\_name in the read endpoint), it's crucial to make sure that data is checked and cleaned before it’s used. Otherwise, it could open the door to things like injection attacks or unexpected behavior.

API security also matters a lot. The whole application is built as a RESTful web API, which means it depends on properly handling incoming and outgoing requests. If any of these endpoints aren’t secured well, it could allow attackers to access or manipulate sensitive data.

Cryptography is something to think about too. While the current code doesn’t show any encryption being used yet, it’s likely needed for protecting things like client data, account numbers, and communication between the client and server. Especially if this ends up going over the internet or handles login details in the future.

Client/server security is another area to consider. Since this is a web-based service, the server needs to be properly secured to avoid attacks like cross-site scripting (XSS) or session hijacking. Good client/server practices help keep the entire connection safe.

**3. Manual Review**

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

During the manual review of the code base, several potential security vulnerabilities were found:

1. **Hardcoded Database Credentials**  
   In DocData.java, the database connection uses hardcoded credentials ("root", "root"). This is risky because if someone gets access to the code, they get instant access to the database. Credentials should be stored securely in configuration files and managed through environment variables.
2. **No Input Validation on API Parameters**  
   In CRUDController.java and GreetingController.java, the input from users (business\_name and name) is taken directly from the request without any validation or sanitation. This opens up potential for injection attacks or other unexpected behavior if someone sends malicious input.
3. **Incomplete Exception Handling**  
   Also in DocData.java, the SQLException is caught, but only prints the stack trace. This doesn’t help the user or the system in a real-world environment and might leak internal details. A better approach would be to log the error securely and return a meaningful error response.
4. **Empty or Incomplete Methods**  
   In myDateTime.java, the retrieveDateTime and setMyDateTime methods are present but not implemented. This could lead to confusion or bugs later if developers assume the methods work correctly when they don’t.
5. **Sensitive Data Exposure Risk**  
   In customer.java, the method showInfo() returns the account number directly. Depending on how this method is used, it could expose sensitive customer data without proper access controls.
6. **Unrestricted API Endpoints**  
   None of the API endpoints have authentication or authorization checks. That means anyone who knows the URL could potentially access data or functions without logging in or proving who they are. This is risky for a financial system.
7. **Poor Naming and Lack of Access Modifiers**  
   The customer class doesn’t capitalize its name (breaking Java naming conventions), and account\_balance is package-private (no access modifier). Without private, it could be modified directly from other parts of the code, making it harder to enforce secure access patterns.

**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

After running the OWASP dependency-check plug-in on the Artemis Financial application, a few security vulnerabilities were found in the project’s dependencies.

One of the reported issues was related to the MySQL JDBC Driver. It has a known vulnerability listed as CVE-2020-26945, which can lead to unexpected behavior or exposure of sensitive data if the driver is outdated or misconfigured. The report recommended updating the driver to a more secure version and ensuring that database connections are configured safely.

Another finding involved an older version of the Spring Boot framework. The report flagged a known vulnerability, CVE-2021-22118, that can allow attackers to bypass security checks in specific request mappings. The best solution here is to upgrade the framework to a version where the issue is resolved.

In addition, the project uses some commons-logging libraries that were flagged as outdated. While they weren’t tied to a specific CVE in this case, the report still recommended replacing or updating them to avoid future risks.

The dependency-check plug-in uses the National Vulnerability Database (NVD) to cross-reference the project's libraries with known security issues. Each entry in the report includes a short description, severity rating, and recommended fix, usually involving upgrading to a patched version.

**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.

• **Remove Hardcoded Credentials**  
Move the database username and password out of the source code in DocData.java. Instead, use environment variables or a secure configuration file that’s not stored in version control.

• **Add Input Validation**  
Sanitize and validate user input in all controller classes, especially in GreetingController.java and CRUDController.java. Use built-in validation libraries or manual checks to prevent injection attacks.

• **Improve Exception Handling**  
Replace printStackTrace() in DocData.java with proper logging, and return a safe, generic error message to the client. Avoid exposing internal details.

• **Implement Authentication and Authorization**  
Add security features like login authentication and role-based access control to the API endpoints. This will make sure only authorized users can access or change data.

• **Secure Sensitive Data Access**  
Protect methods like showInfo() in customer.java that return private data. Make sure sensitive data is only available to authenticated users and that access is restricted based on roles or permissions.

• **Update Vulnerable Dependencies**  
Upgrade the MySQL JDBC Driver and Spring Boot to newer, secure versions as suggested in the dependency-check report. Review all third-party libraries for available updates.

• **Avoid Incomplete or Placeholder Code**  
Fully implement or remove placeholder methods like the empty ones in myDateTime.java. Leaving them as-is might confuse developers or create logic errors later.