# CS 305 Project One Template

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **[Date]** | **[Your name]** |  |

## 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

Kain Semonis

**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’s needs would include a secure RESTful web application, the protection of customer financial data and defense against external threats. These should comply with any legal regulations, utilize secure communication protocols, and should, ideally, do so in as cost-effective a manner as possible.

The value in security in this scenario is fairly simple; financial information is extremely sensitive, so preventing unauthorized access to and use of financial information is crucial. From a business standpoint, it’s important because higher security leads to higher client trust, which tends to lead to higher number of and quality of clients, which, in turn, leads to more money.

Nothing in the scenario seems to explicitly indicate that the company produces any international transactions, though, there is also nothing indicating that it is a purely local financial institution, either. However, considering the importance of growth in business, it is not out of the realm of possibility that the company could grow to handle international transactions, so it seems worthwhile to design around the idea that there will be international transactions.

The main governmental restrictions in play seem to be things like PCI-DSS, assuming the likely scenario that the financial company handles any credit card transactions, and things like the Gramm-Leach-Bliley Act, and the Sarbanes-Oxley Act, for specifically financial data security in the US. There is also the California Consumer Privacy Act, if the company has clients in California, and the General Data Protection Regulation for clients in the EU.

The most common external threats are likely to be things like phishing and social-engineering attacks, which are extremely difficult to design around, and can mostly only be prevented by educating potential targets, as by their nature the target gives the attacker access to the system. There is also the potential for SQL injection through the RESTful API, if user inputs are not properly handled. Cross-site request forgeries, in which an attacker performs unauthorized actions under the authority of someone else, are also potential risks. DDoS and DoS attacks also are likely to be among the most common attacks, and while they are unlikely to lead directly to any data breaches or unauthorized activity, they do slow or entirely remove access to the system if not properly handled. By their nature, APIs also present a potential vulnerability, often without much ability for the system using that API to really handle these vulnerabilities without fully removing the reliance on that API.

For modernization, a good authentication system is needed, like OAuth or JWTs would be useful. Any data that’s sensitive at all should, pretty obviously, be encrypted, so enforcing HTTPS for client-server communications would be a good idea. Any dependencies should be updated regularly, especially open-source ones, to reduce the likelihood of that dependency having any known vulnerability that hasn’t been patched. DDoS and DoS attacks could also be mitigated somewhat by ensuring that APIs have their inputs validated, and have proper firewalls and rate limiting measures in place to reduce their ability to be used for such attacks. It’s also important to remember that modernization shouldn’t be a one-time process, and should be done pretty much continually, so having security audits and penetration testing would also be a good measure to have in place.

**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 is important to review because it tells how each part of the overall system will interact with each other, and so it can identify potential vulnerabilities in the overall design of the system.  
 Input validation is important because it is one of the main vectors for injection attacks, so properly validating and sanitizing the inputs should prevent these attacks from taking place.

The nature of APIs pretty much requiring that they are fairly widely accessible leads to them being fairly common vectors for attacks. In this scenario, the usage of an API seems fairly central, so reviewing the API’s security is, in this case, important.

The data in use in this scenario should also be encrypted whenever that data may be sensitive; which, considering it is financial information, is probably going to be most of the time. Keeping data encrypted can also reduce the impact of data breaches, which are nearly inevitable. And, even if encryption wasn’t a good security measure, most data protection regulations require at least some degree of encryption.

Because the application can essentially be boiled down to users interfacing with the system through a web application, and the server processing whatever requests are made and managing data, the client/server area is necessary to prevent any data tampering or other unauthorized access.

Code errors need to be managed to prevent things like the application just crashing, data being corrupted, or those errors being used as attack vectors or simply to gain more information about the system and its structure to facilitate further attacks.

Code Quality is important because it can easily introduce vulnerabilities through the presence of insecure dependencies, flaws in the logic, and simple resource mismanagement. It also helps in the previously mentioned continual modernization; the code should be as easy to maintain as possible.

Encapsulation would assist in protecting any of the sensitive data structures of the system, similarly to the code error section, where the vulnerability isn’t likely to be prevented by encapsulating the data, so much as it is likely to keep that vulnerability from being discovered.

**3. Manual Review**

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

In CRUDController.java, there is no input validation on the business\_name parameter, which could be used for an injection attack. CRUDController.java also exposes the /read endpoint in a way that leaves it accessible with no authentication, and so is vulnerable.

Customer.java stores the account\_number integer in plain text, rather than being encrypted.

DocData.java creates the connection object, but never closes it, which has the potential to be abused by DDoS and DoS attacks.

GreetingController.java has a similar issue as the other controller, but the exposed endpoint is, instead, /greeting.

RestServiceApplication.java doesn’t have any of the common HTTP security headers, like Content-Security-Policy, which leaves it vulnerable to multiple common attacks. Overall, it lacks any of the configuration for enforcing HTTPS.

**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

The critical severity, known-exploited vulnerabilities are as follows:

CVE-2020-1938 is a vulnerability in which Apache JServ Protocol has a higher degree of trust than similar HTTP connections. Mitigation methods include not allowing untrusted users access to AJP ports, and, again upgrading to more recent versions.

CVE-2022-22965 is a vulnerability in Spring MVC or WebFlux on certain versions of the Java dev kit potentially allowing RCE through data binding. The main mitigation method seems to be to run the application as an executable jar, rather than as a WAR deployment, if possible.

CVE-2016-1000027 is a vulnerability that allows RCE through the deserialization of certain data.

CVE-2023-20873 is a vulnerability in a range of Spring Boot versions in which applications deployed to Cloud Foundry can potentially be susceptible to security bypass, and can be mitigated by upgrading versions.

**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.

I will probably focus a little bit more on the manual review side of this than the static testing side than is probably totally necessary, but I feel somewhat more confident on that front. The hard, plaintext username and password in DocData.java expose information that can potentially allow unauthorized access to the system, and, depending on the username and password that are compromised, the effects could be as ‘small’ as a data breach or as large as a full system compromise, where the attacker gains full control over the entire system. This should be mitigated through the use of things like secure config files, or other variables in their place.

The un-encrypted account\_number and account\_balance data in customer.java should be encrypted.

The exposed API endpoints should have security measures like OAuth or JWT implemented, in order to keep them secure.

The un-closed connection object in DocData.java should, pretty obviously, be closed.

RestServiceApplication.java’s lack of HTTP security configuration is also very easy to point out that it should, in fact, have configured HTTP security.