****

# CS 305 Project One

**Artemis Financial Vulnerability Assessment Report**

Table of Contents

[Document Revision History 3](#_Toc32574607)

[Client 3](#_Toc32574608)

[Instructions 3](#_Toc32574609)

[Developer 4](#_Toc32574610)

[1. Interpreting Client Needs 4](#_Toc32574611)

[2. Areas of Security 4](#_Toc32574612)

[3. Manual Review 4](#_Toc32574613)

[4. Static Testing 4](#_Toc32574614)

[5. Mitigation Plan 4](#_Toc32574615)

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **1/24/21** | **Keri Todd** |  |

## Client



## Instructions

Deliver this completed vulnerability assessment report, identifying your findings of security vulnerabilities and articulating recommendations for next steps to remedy the issues you have found.

Respond to the five steps outlined below and include your findings. Replace the bracketed text on all pages with your own words. If you choose to include images or supporting materials, be sure to insert them throughout.

## Developer

Keri Todd

## 1. Interpreting Client Needs

The value of secure communications to Artemis Financial is very great. Due to the fact that the company is using a RESTful web application, secure communication is extremely important. There are many threats that exist for web applications, particularly when a company handles sensitive information for their clients. There are many international transactions that the company produces. Since this is a finance company, there are likely to be international clients that will be using this web application. There are some governmental restrictions that need to be considered when developing this web application. Every country has their own regulations in regards to finances. This will need to be taken into consideration when revamping the company’s web application. While there are many threats that exist for RESTful web applications, the main ones I would recommend focusing on are injection attacks and Denial of Service (DoS) attacks. Web application technologies are constantly evolving. We will need to make sure that this web application is up-to-date in regards to the libraries that are being used and has adequate preventative measures in place to protect the application from attacks. For example, we will want to make sure that there is adequate input validation as this tends to be the first line of defense against threats.

## 2. Areas of Security

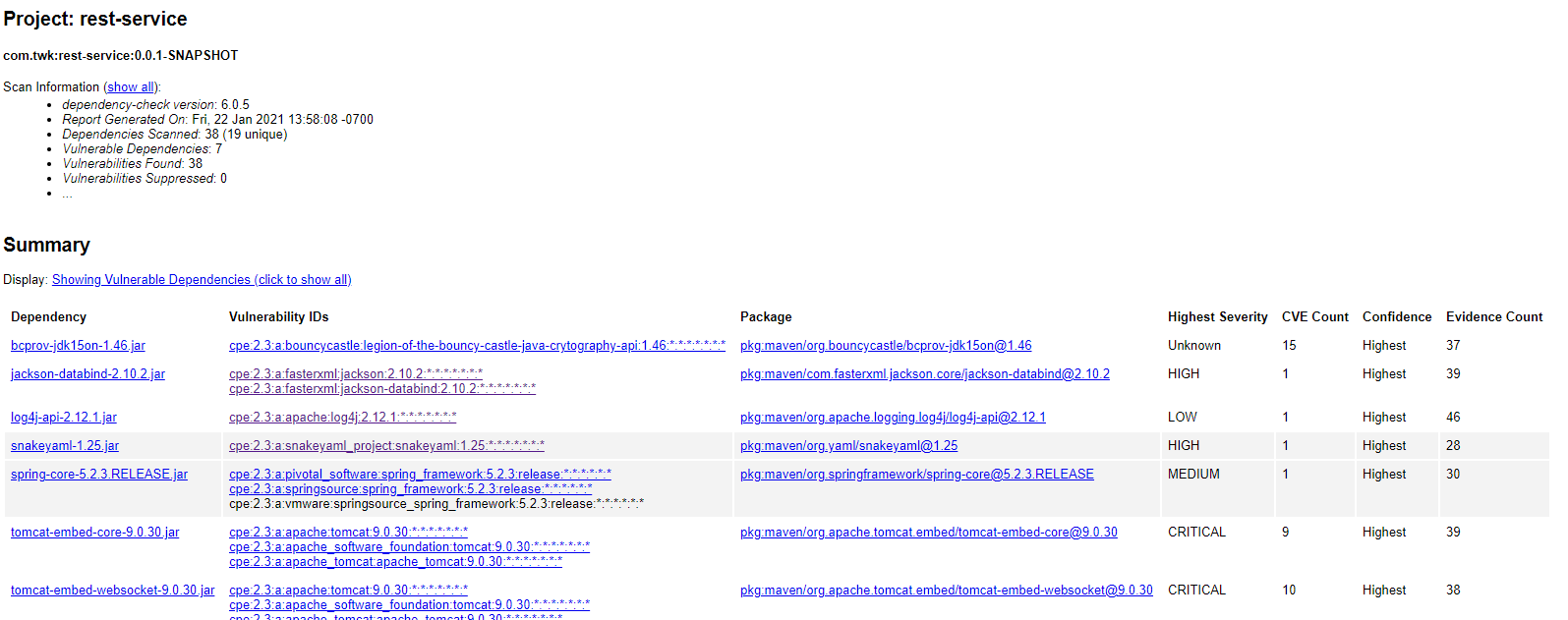
Based on the VAPFD, the security areas that should be checked are input validation, APIs, cryptography, client/server, and code quality. Input validation is one of the basics of software security. It ensures that any input from a user is secure and valid. When reviewing software security, input validation should always be at the top of the list because it can help prevent malicious data from entering the system. APIs should be evaluated because the company uses a RESTful web application. This means that the company will be using an API to ensure their application functions properly. By evaluating the API they’re using, we’ll be able to determine how secure users’ data is. APIs are like the middleman between the servers and the application itself. This means that user data will be transferred to the servers via an API. We need to ensure that the API is secure so there aren’t any data leaks. Due to the fact that Artemis Financial will be storing client data, it’s important that this data is encrypted so that external attackers cannot access the data. Client/server will need to be checked for security faults because this application will be interacting with clients and servers. It’s important that the communication between the clients and servers is secure because clients will be inputting their personal information into the system. The stream of communication needs to be secure so this data is not leaked or stolen. Finally, code quality should be reviewed because it is each developer’s responsibility to ensure that their code is secure by using different strategies and practices. This will allow me to ensure that each developer is following security protocols and identify any security flaws in the code.

## 3. Manual Review

After manually inspecting the code, below are the vulnerabilities I found:

* In *customer.java* on line 5, the variable *account\_balance* isn’t listed as private.
* In *DocData.java* on line 26, the variable *con* is never used

## 4. Static Testing



* Bcprov-jdk15on-1.46.jar: CVE-2013-1624. This vulnerability focuses on the Bouncy Castle Crypto package, which is a Java implementation of cryptographic algorithms. This vulnerability can be fixed by using version 1.60 or later of Bouncy Castle Legion of the Bouncy Castle Java Cryptography APIs.
* Jackson-databind-2.10.2.jar: CVE-2020-25649. This vulnerability can be fixed with a variety of strategies. Some strategies include using less complete data formats, disable XML external entity and DTD processing in all XML parsers in the application, and using whitelisting server-side input validation.
* Log4j-api-2.12.1.jar: CVE-2020-9488. This vulnerability focuses on improper validation of certificate with host mismatch in Apache Log4j SMTP appender. This could allow a man-in-the-middle attack. You can prevent this type of attack by having strong encryption mechanisms on wireless access points, use VPNs, force HTTPS, and using public key pair-based authentication.
* Snakeyaml-1.25.jar: CVE-2017-18640. This vulnerability allows entity expansion during a load operation when using the Alias feature SnakeYAML 1.18. This can be fixed by using a different version of SnakeYAML and using whitelisting server-side input validation.
* Spring-core-5.2.3.RELEASE.jar: CVE-2020-5421. This vulnerability has weak protections against RFD attacks and these protections can be bypassed depending on the browser being used by using a jsessionid path parameter. This only occurs in older versions of the Spring Framework. This can be solved by using a newer version of the Spring Framework.
* Tomcat-embed-core-9.0.30.jar: CVE-2019-17569. This vulnerability focuses on the refactoring that is present in certain versions of Apache. This refactoring introduced a regression. The result of this regression was that invalid Transfer-Encoding headers were incorrectly processed which lead to a possibility of HTTP Request Smuggling. This can be fixed by using a different version of Apache. In addition, we should disable reuse of back-end connections, so that each back-end request is sent over a separate network connection. We can also use the HTTP/2 protocol for back-end connections because this protocol prevents ambiguity about the boundaries between HTTP requests.
* Tomcat-embed-websocket-9.0.30.jar: CVE-2019-17569. This vulnerability focuses on the refactoring that is present in certain versions of Apache. The refactoring introduced a regression which could cause invalid Transfer-Encoding headers to be incorrectly processed, which can lead to a possibility of HTTP Request Smuggling. This can be prevented by using a different version of Apache Tomcat and using the same web server software for the front-end and back-end servers, so that they agree about the boundaries between HTTP requests.

## 5. Mitigation Plan

After having completed the manual review of the code and the static testing, below are my recommendations as to how to ensure Artemis Financial’s web application is secure:

* Use version 1.60 or later of Bouncy Castle Legion. This will help prevent remote attackers to conduct distinguishing attacks and plaintext-recovery attacks.
* Implement whitelisting server-side input validation. This means that we will focus on only accepting input that is known to be good.
* Improve encryption mechanisms on wireless access points. I would recommend using WPA2 as it is the strongest form of encryption.
* Use a newer version of SnakeYAML.
* Use a newer version of the Spring Framework.
* Use the HTTP/2/ protocol for back-end server connections as this protocol prevents ambiguity about the boundaries between HTTP requests.
* Use a different version of Apache Tomcat.
* Make *account\_balance* private in *customer.java*. This will ensure that other classes won’t have free access to the variable.
* Ensure that the variable *con* is used in *DocData.java*.