# CS 305 Project One

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **12/1/2024** | **Cole Flournoy** | **-** |

## Client



## Developer

Cole Flournoy

**1. Interpreting Client Needs**

As a financial company, secure communications are non-negotiable for Artemis Financial. Although secure communication is important in every software application, in this industry extra care needs to be taken, because companies in the financial sector are targeted more frequently than those in other industries (Pryimenko, 2024). One way to measure the value of this security is in dollars. A 2024 IBM report showed that the average cost of a data breach in the financial services sector is $4.45 million (Sobers, 2024). Customer trust is harder to measure but also very important for a financial services company.

Because Artemis Financial deals with client’s investment accounts, there are specific regulations from the Securities and Exchange Commission (SEC) and Financial Industry Regulatory Authority (FINRA) that must be followed to protect the customers’ information from unauthorized access (Federal Cybersecurity and Data Privacy Laws Directory, 2024). There may also be international transactions for clients who use Artemis Financial to invest in markets outside of the United States. The company will need to follow the regulations of all countries involved in the transaction.

One of the top security threats for financial services platforms is phishing, or manipulating users or employees into providing their login credentials for the application (Kost, 2024). Phishing for employee credentials can be especially dangerous, because a 2021 report showed that for the majority of financial services companies (64%), there were over one thousand sensitive files that every employee at the company had access to (Sobers, 2024). The attacks that are most relevant to this report are ones that can be prevented by managing dependencies and maintaining secure coding practices. Those are attacks like SQL injection, which allows users to write code that directly queries the database, and Denial of Service, in which an attacker will exploit a vulnerability in the code or system configuration to overwhelm the system and cause it to stop functioning, often with the goal of collecting a ransom to stop the attack (Kost, 2024).

**2. Areas of Security**

I think the most important area of security for Artemis Financial is cryptography. Cryptography is the obfuscation of sensitive information through encryption, and in any software application it is important because it secures the transmission and storage of that information. It can also reduce the damage done by an infiltration of the system. If a bad actor gains access to my database and all of my user’s emails and passwords are stored in plaintext, that infiltration has catastrophic cascading security implications for all of my users. That type of exploit is obviously never good news, but if those emails and passwords are encrypted, the damage is at least much more contained. For a company like Artemis Financial that deals with customers’ bank accounts, investments, and likely highly sensitive personal information such as social security numbers, cryptography is absolutely essential.

Other areas of security that are also very important are code error handling and input validation. Secure error handling also helps avoid exposing sensitive information, particularly information that might help bad actors exploit other parts of the application in the future. An example would be errors that show stack traces, code snippets, or database structures. Input validation is important because unfiltered input, even if it is not intentionally malicious, can break the program or unintentionally expose data. For example, when asking a user to input their age, it is not enough to assume that the user will enter an integer. If a user enters 65.3 or ‘sixty-five’ instead, input validations should catch that and prompt the user for the expected type of input rather than continue on blindly executing code that expects an integer, which would lead the program to crash. We also need to consider actively malicious user behavior, such as modifying database queries in SQL injection attacks or adding code to inputs that would be executed and alter the behavior of the application.

The stakes are higher for a financial institution, and responding to an attack after the fact is too late. A 2021 Report showed that companies in the financial services sector “take an average of 233 days to detect and contain a data breach” (Sobers, 2024). Extra time and care should be taken with the above security measures so that vulnerabilities are found as early as possible and exploits are proactively prevented.

**3. Manual Review**

The first vulnerability that jumped out in this code review was that Tomcat, the service being used to set up the server for the project, is using HTTP by default, which is not encrypted and should not be used. Custom configurations for Tomcat should be stated in the *application.properties* file.

Another set of vulnerabilities comes from the combination of error handling and routing. Navigating to the URL endpoint ‘*/read*’ brings up an error message that says, “*Required String parameter 'business\_name' is not present*”. This gives unnecessary insight to users into the internal structure of the application (i.e. that the URL should be structured ‘*/read?business\_name=<name\_of\_business>*’). A malicious user could then take that information and start trying business names that they think may be customers of Artemis Financial. Guessing correctly would expose that business’ information, and because of the lack of authentication for users making requests, there is nothing verifying that a user has permission to view that business’ information. Another related issue is that navigating to ‘*/read?business\_name*’ and omitting a particular business name still returns data from the database, which shouldn’t be possible because a company wasn’t specified.

Finally, there are several dependencies in the codebase that have known vulnerabilities. The version of *Spring Boot Starter Parent*, which is the foundation of this project, has over 140 known vulnerabilities (mostly due to dependencies that it uses) and is more than four years old. Another dependency, *Bouncy Castle Provider* (which is used for implementing cryptographic algorithms) has eight direct vulnerabilities and is more than a decade out of date. Cryptography evolves very quickly, and it is essential to keep all dependencies, especially dependencies that deal with encryption, up to date.

**4. Static Testing**

Below are highlighted the top vulnerabilities found through the static dependency check testing. They are ordered first by severity level and then by the number of Common Vulnerabilities and Exposures (CVE) found. Although the number of CVEs alone is not necessarily the most important metric for determining risk, the difference between a dependency with 27 CVEs and one with 3 still gives a reasonable indication of general priority. These dependencies were selected because they represent either a critical or high risk to the security of our application, and all should be addressed, regardless of their CVE count. It is important to also note that these CVE counts are not necessarily unique, meaning that many dependencies can, and do in this case, share the same CVEs. This list has also not been filtered for potential false positives.

The full list of dependencies with specific details about the CVEs for each one is available in the Dependency Check Report attached as a separate document.

| **Dependency** | **Description** | **Severity** |
| --- | --- | --- |
| tomcat-embed-core-9.0.30.jar | This dependency is the core component of Apache Tomcat, which is designed for embedding Tomcat servers into Java applications. It makes the application vulnerable to Denial of Service attacks and leaks of sensitive information between requests.  CVE count: 27 | **CRITICAL** |
| snakeyaml-1.25.jar | This dependency is used to read and write YAML files, which Spring Boot uses for configuration. Vulnerabilities include Denial of Service attacks and remote code execution.  CVE count: 10 | **CRITICAL** |
| log4j-api-2.12.1.jar | This dependency provides a standardized set of interfaces that can be used to manage logging in Java applications. It makes the application vulnerable to man-in-the-middle attacks and remote code execution.  CVE count: 5 | **CRITICAL** |
| spring-boot-2.2.4.RELEASE.jar | This dependency provides the basic functionality needed to bootstrap and run a Spring Boot application. Vulnerabilities include Denial of Service and temporary directory hijacking.  CVE count: 3 | **CRITICAL** |
| bcprov-jdk15on-1.46.jar | This dependency is a cryptography API, which enables various standard cryptographic implementations, such as encryption, decryption, signing, key management, etc. It is vulnerable to timing attacks, which expose information about the encryption process through observation and analysis of timing data.  CVE count: 20 | **HIGH** |
| spring-web-5.2.3.RELEASE.jar | This dependency provides support for core web functionalities, including HTTP client support, REST APIs, and data-binding in Spring Framework. It introduces a wide number of vulnerabilities that can be seen in more detail in the attached full report.  CVE count: 8 | **HIGH** |
| jackson-databind-2.10.2.jar | This dependency is used for JSON processing in Java applications when converting Java objects to JSON and vice versa. Vulnerabilities include XML external entity and Denial of Service attacks.  CVE count: 6 | **HIGH** |
| logback-core-1.2.3.jar | This dependency provides comprehensive logging for Java applications. Two known vulnerabilities Denial of Service attacks and code execution.  CVE count: 2 | **HIGH** |
| spring-webmvc-5.2.3.RELEASE.jar | This dependency is part of the Spring Framework and supports the development of web applications using the Model-View-Controller (MVC) architecture. Vulnerabilities include the insertion of log entries and the ability to obtain file information through malicious HTTP requests.  CVE count: 2 | **HIGH** |
| spring-beans-5.2.3.RELEASE.jar | This dependency manages the lifecycle of ‘beans’ (objects managed by the Inversion of Control container) in the Spring Framework. This dependency makes the application vulnerable to remote code execution via data binding.  CVE count: 1 | **HIGH** |

**5. Mitigation Plan**

Tomcat can easily be configured to use HTTPS instead of the default HTTP by adding these preferences to the *application.properties* file. This would ensure that all requests to our server are encrypted, which is significantly preferred.

Error handling for the ‘*/read*’ endpoint can be fixed by adding a custom error message that does not include information about which parameters are required for the request. Users don’t need this information because they should never be navigating the application by typing parameters directly into the URL, it is our job as developers to give the users simple navigation options *inside* of the application and to manage which requests and parameters should be sent internally. An even better solution for the user experience would be for an error to never appear at all in this case and for the user to be simply redirected to another page. This gives a better experience for users who are trying to use the application genuinely, because they don’t have to manually navigate back to a functioning page. It also obscures information about the application from users trying to exploit it.

Regarding the personal information exposed when using the ‘*/read*’ endpoint normally, there are a few possible steps to take. First, the business names shown in the URL can be encrypted, so that companies are not semantically connected to their information in the request. For example, instead of a URL that shows ‘*/read?business\_name=TJMaxx*’, a more secure option would be a URL that reads ‘*/read?business\_name=1kj4slkk914z*’, where *1kj4slkk914z* represents the hashed version of the company’s name, which is decrypted in the backend of the application and used to determine which resources from the database to return. This prevents business name guessing and obscures as much information as possible transmitted in the request. The second step is to remove the possibility that a user can navigate to ‘*/read?business\_name*’ without specifying a value for the *business\_name* parameter. This is already done successfully for the ‘*/greeting*’ endpoint on line 16 of the *GreetingController.java* file by adding a default value for the parameter. Implementing the same strategy on line 13 of the *CRUDController.java* file would address the problem, but there is another option. We could also perform a check inside of the *check\_document()* function (defined on line 21 of the *DocData.java* file) that throws an error if the *business\_name* is null. Both of these solutions would prevent user data from being exposed due to incorrectly formed requests.

Another concern is the lack of user authentication, which allows all users to have the same permissions to access records from the database. Users should not be able to access data from companies that they don’t belong to, and having information be available to anyone who visits the site makes exploits by outsiders much easier. My recommendation is to implement a user login flow so that only users who are authenticated can reach any of the application’s endpoints. Using that baseline, we can then verify the user, as well as their specific permissions, making a request to a particular endpoint. This verification logic could go in each of the files that retrieves data based on a request. Some examples of the functions that would use this logic include: *getContent()* inside of *Greeting.java* and *getContent()* and *getContent2()* inside of *CRUD.java*.

Finally, there are the out of date dependencies. Versions for *Spring Boot Starter Parent* and *Bouncy Castle Provider* are defined in the *pom.xml* file (on lines 8 and 30, respectively). If a particular version is being set because of a compatibility issue, the most recent version that is still compatible should be used, and the compatibility issue should be prioritized. The most secure option is to have both of these dependencies running on the latest version to avoid the 150+ known vulnerabilities that the older versions introduce combined. To do so automatically, which is recommended for longevity, lines 8 and 30 can be omitted completely, and the dependencies will always search for and use the latest version when the application is run.

The dependencies listed in the static testing section should also be prioritized for update, because they represent several critical vulnerabilities in the application. I recommend addressing these dependencies in the order given in the sorted table above. The simplest solution is often just to update to the latest version. For example, our most critical dependency, tomcat-embed-core, has 27 CVEs on the version used in this project (9.0.30) but none on the latest version (11.0.1 at time of writing). This is not a perfect indication that there are no security vulnerabilities with this version (because vulnerabilities are found and exploited the longer a version is in use), but dealing with known vulnerabilities is a good first step. That said, we may not always be able to update to the latest version of a dependency for compatibility reasons or be able to guarantee that there will be a version available that has addressed the vulnerabilities. For that reason, an additional step should always be to assess the code of the application itself based on specific implementation and usage. Some vulnerabilities, like SQL injection, can be mitigated in code through proper input validation and parameterization.

**References**

*Federal Cybersecurity and Data Privacy Laws Directory*. IT Governance. (n.d.). https://www.itgovernanceusa.com/federal-cybersecurity-and-privacy-laws

Kost, E. (2024, November 18). The 6 Biggest Cyber Threats for Financial Services in 2024. *UpGuard*. December 1, 2024, https://www.upguard.com/blog/biggest-cyber-threats-for-financial-services

Pryimenko, L. (2024, March 25). The 7 Industries Most Vulnerable to Cyberattacks . *Systeca*. November 29, 2024, https://www.syteca.com/en/blog/5-industries-most-risk-of-data-breaches

Sobers, R. (2024, September 13). 157 Cybersecurity Statistics and Trends. *Varonis*. November 29, 2024, https://www.varonis.com/blog/cybersecurity-statistics