# CS 305 Project One

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **July 15, 2024** | **Damean Murphy-Short** | **Origin of the Document** |

## Client



## 

## Developer

Damean Murphy-Short

**1. Interpreting Client Needs**

As a consulting company that must protect the sensitive financial information of its clients, security is an utmost concern to Artemis Financial. Encryption and secure transfer of this information is critical, as broadcasting it in plain-text would be an invitation for immediate theft. In fact, at least in the US, this information must be protected against unauthorized access and anticapated threats to follow legal regulations such as the Graham-Leach-Bliley Act (FTC, 2023). These anticipated threats manifest as: man-in-the-middle attacks to access sensitive data without authorization, injection attacks that seek to expose vulnerabilities in database queries to the same end, denial of service attacks that aim to cause damage to the company by overwhelming its infrastructure, among others.

To create a modernized web application, we must take all of the above into consideration, while utilizing the array of tools available to us by open source projects that have the united goal of providing a secure and stable web. Because the web is always changing in a security arms-race against attackers, it is important to always keep up to date with these tools to eliminate the most-recently discovered vulnerabilities and remain in compliance with the expectations of regulators and consumers. To attempt to do so without any already-available libraries would be an enormous undertaking for the company.

**2. Areas of Security**

Provided with this document is a vulnerability assessment process flow diagram, which will be referenced to identify a few areas of security that will be of the greatest impact to Artemis Financial. Input validation will be of great importance because, as mentioned above, attackers will be attempting injections against our services, which could potentially expose data without authorization. A secure API to transport our data and validate input will be essential in standardizing the level of security across the entire application, given that the API is adhered to throughout. Cryptography, previously mentioned with the need of encrypted communications, will also provide a layer of protection for client data to prevent an attacker who has intercepted data from being able to read it without being the intended recipient. It is also essential that every transaction is verified on the server rather than being subject to requests from the client as part of secure Client/Server architecture. Finally, it will be important for data at rest (not currently being requested or transferred) to be protected in Secure Data Structures since the company will undoubtedly be handling massive databases full of sensitive client data that may not be in use at every moment, but still vulnerable to attackers.

**3. Manual Review**

In this section, we will identify potential security vulnerabilities in the provided source code and explain why they pose a threat to Artemis financial. In the ‘Mitigation Plan’ section, solutions to these vulnerabilities will be discussed.

- The DocData.java class connects to the mysql database as the root(highest privilege) user. This presents a threat because this connection will have unhindered access to all data on the database as well as potentially being an entry-point to running arbitrary commands on the server.

- The DocData.java class uses a hardcoded password. This means anyone with access to the source code is aware of the password, and worse, anyone who can intercept the messages between the web application and the database (JDBC does not require encryption) can potentially read the password in plain text.

- The CrudController.java class does not require any input validation for the business\_name parameter, which could allow for malformed input to reach commands in the server (at this stage, the business\_name is not used, but should still be validated with more care for whatever future use is planned).

- The CrudController.java class accepts this parameter in the url, rather than somewhere that would be more difficult to intercept for a man-in-the-middle attacker, as in an HTTPS POST request using an encrypted connection.

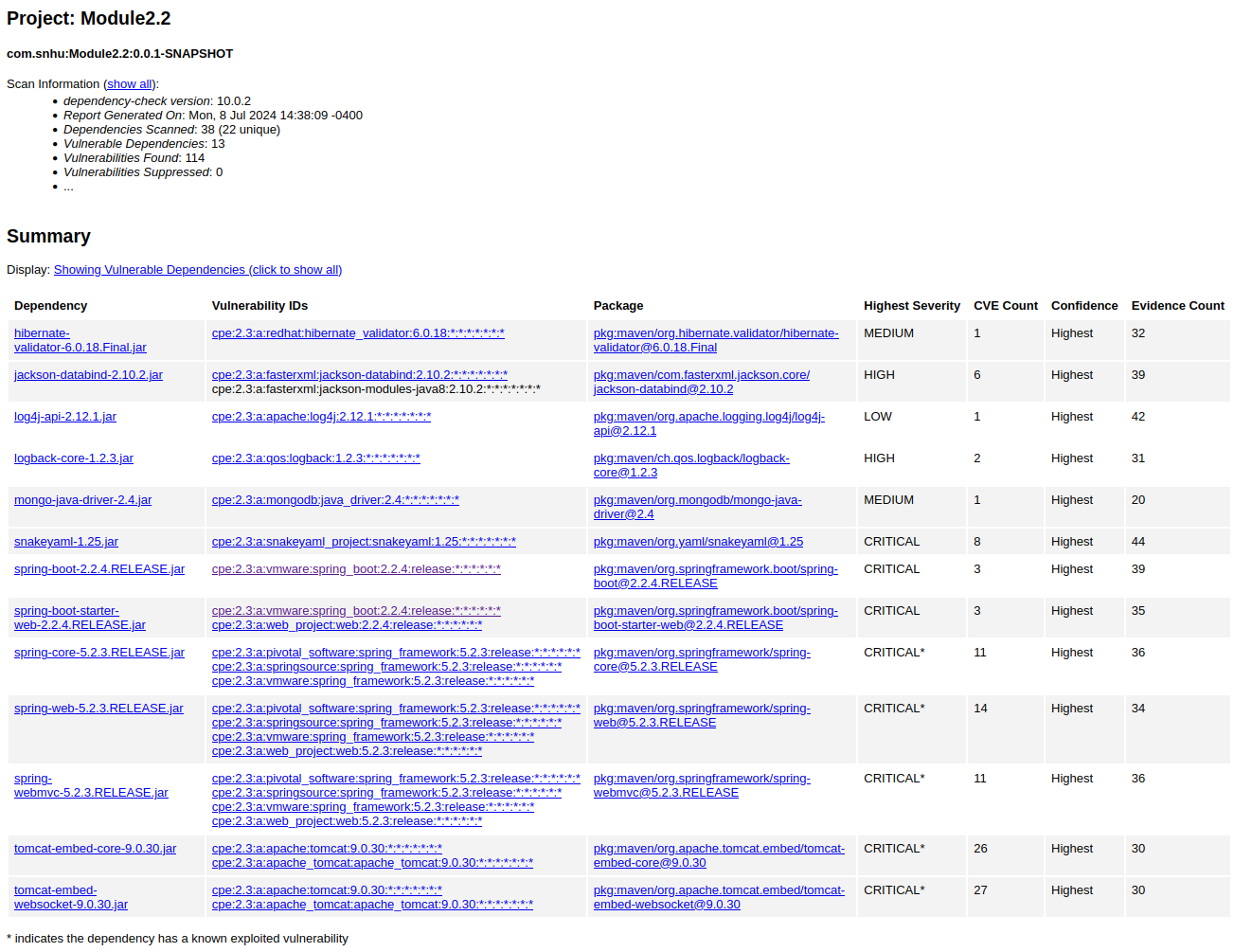
- The GreetingController.java class shares the same two vulnerabilities with the ‘name’ parameter.

- The public method ‘deposit’ in the customer.java class does not validate the amount ‘a’ that is entered.

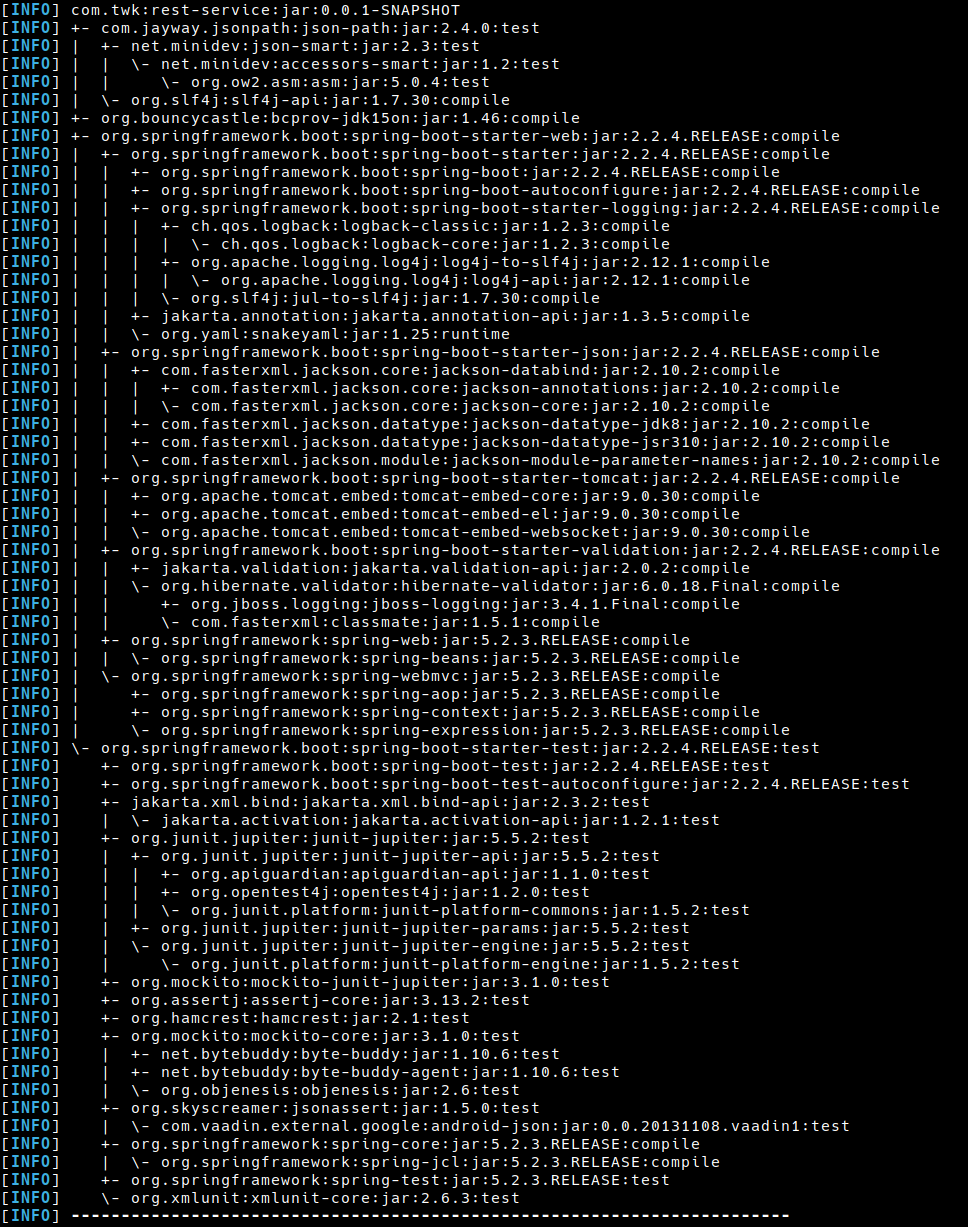
- The above method also does not reference any authorization method to make sure the caller has proper rights to make a deposit, nor does it verify the transaction by any other means.

**4. Static Testing**

Pictured below is a summary of the vulnerabilities found in the dependencies brought into the project.



Also pictured, the tree structure of dependencies to help understand which primary dependencies are being used/are required by the project.



Detailed in the images, the project is utilizing 3 main dependencies, which are introducing the others as depenencies of their own: BouncyCastle, JayWay.json-path, and SpringBoot. In our case, the analyzer did not find any vulnerabilities in JayWay json-path, so we will be interested only in the vulnerabilites introduced by BouncyCastle, SpringBoot, and their dependencies.

As can be seen in the below CPE (Common Platform Enumeration), all vulnerabilities in BouncyCastle are identified only in versions 1.73 and earlier, including denial of service and cryptographic failure vulnerabilities:

[cpe:2.3:a:bouncycastle:bouncy\_castle\_for\_java:1.46:\*:\*:\*:\*:\*:\*:\*](https://nvd.nist.gov/vuln/search/results?form_type=Advanced&results_type=overview&search_type=all&cpe_vendor=cpe%3A%2F%3Abouncycastle&cpe_product=cpe%3A%2F%3Abouncycastle%3Abouncy_castle_for_java&cpe_version=cpe%3A%2F%3Abouncycastle%3Abouncy_castle_for_java%3A1.46)

[cpe:2.3:a:bouncycastle:legion-of-the-bouncy-castle-java-crytography-api:1.46:\*:\*:\*:\*:\*:\*:\*](https://nvd.nist.gov/vuln/search/results?form_type=Advanced&results_type=overview&search_type=all&cpe_vendor=cpe%3A%2F%3Abouncycastle&cpe_product=cpe%3A%2F%3Abouncycastle%3Alegion-of-the-bouncy-castle-java-crytography-api&cpe_version=cpe%3A%2F%3Abouncycastle%3Alegion-of-the-bouncy-castle-java-crytography-api%3A1.46)

A wide number of springboot and related dependency vulnerabilites exist only in springboot 2.2.4 or older, or for use cases that are outside of our application’s current configuration or needs.

For instance, snake-yaml [cpe:2.3:a:snakeyaml\_project:snakeyaml:1.25:\*:\*:\*:\*:\*:\*:\*](https://nvd.nist.gov/vuln/search/results?form_type=Advanced&results_type=overview&search_type=all&cpe_vendor=cpe%3A%2F%3Asnakeyaml_project&cpe_product=cpe%3A%2F%3Asnakeyaml_project%3Asnakeyaml&cpe_version=cpe%3A%2F%3Asnakeyaml_project%3Asnakeyaml%3A1.25) is only of concern if being used for deserialization of java objects, which does not occur in our project. Despite this, it may be a feature that is utilized in older versions of springboot, and so care must be taken in considering its inclusion.

Similarly, [cpe:2.3:a:apache:tomcat:9.0.30:\*:\*:\*:\*:\*:\*:\*](https://nvd.nist.gov/vuln/search/results?form_type=Advanced&results_type=overview&search_type=all&cpe_vendor=cpe%3A%2F%3Aapache&cpe_product=cpe%3A%2F%3Aapache%3Atomcat&cpe_version=cpe%3A%2F%3Aapache%3Atomcat%3A9.0.30) lists vulnerabilities in Apache Tomcat that only exist outside of our use case or for non-default configurations. For this reason, it is important to always cross-reference these vulnerabilites whenever we make a change to the tomcat configuration, but may not be an issue in newer versions of tomcat if a newer version of springboot is introduced.

[cpe:2.3:a:vmware:spring\_boot:2.2.4:release:\*:\*:\*:\*:\*:\*](https://nvd.nist.gov/vuln/search/results?form_type=Advanced&results_type=overview&search_type=all&cpe_vendor=cpe%3A%2F%3Avmware&cpe_product=cpe%3A%2F%3Avmware%3Aspring_boot&cpe_version=cpe%3A%2F%3Avmware%3Aspring_boot%3A2.2.4), [cpe:2.3:a:vmware:spring\_framework:5.2.3:release:\*:\*:\*:\*:\*:\*](https://nvd.nist.gov/vuln/search/results?form_type=Advanced&results_type=overview&search_type=all&cpe_vendor=cpe%3A%2F%3Avmware&cpe_product=cpe%3A%2F%3Avmware%3Aspring_framework&cpe_version=cpe%3A%2F%3Avmware%3Aspring_framework%3A5.2.3), all detail vulnerabilities that are shared by only older versions of springboot. The older versions of springboot also bring with them older dependencies, such as: [cpe:2.3:a:fasterxml:jackson-databind:2.10.2:\*:\*:\*:\*:\*:\*:\*](https://nvd.nist.gov/vuln/search/results?form_type=Advanced&results_type=overview&search_type=all&cpe_vendor=cpe%3A%2F%3Afasterxml&cpe_product=cpe%3A%2F%3Afasterxml%3Ajackson-databind&cpe_version=cpe%3A%2F%3Afasterxml%3Ajackson-databind%3A2.10.2), [cpe:2.3:a:redhat:hibernate\_validator:6.0.18:\*:\*:\*:\*:\*:\*:\*](https://nvd.nist.gov/vuln/search/results?form_type=Advanced&results_type=overview&search_type=all&cpe_vendor=cpe%3A%2F%3Aredhat&cpe_product=cpe%3A%2F%3Aredhat%3Ahibernate_validator&cpe_version=cpe%3A%2F%3Aredhat%3Ahibernate_validator%3A6.0.18), [cpe:2.3:a:apache:log4j:2.12.1:\*:\*:\*:\*:\*:\*:\*](https://nvd.nist.gov/vuln/search/results?form_type=Advanced&results_type=overview&search_type=all&cpe_vendor=cpe%3A%2F%3Aapache&cpe_product=cpe%3A%2F%3Aapache%3Alog4j&cpe_version=cpe%3A%2F%3Aapache%3Alog4j%3A2.12.1), which further complicate the security of our system by relying on several outdated libraries.

**5. Mitigation Plan**

First, we will address what can be done to mitigate issues brought about by our dependencies, since the majority of these dependencies can be improved by updating to their newest release versions. Bcprov-jdk18on-1.78.1, the latest bouncy castle release(Bouncy Castle, 2024), is free from the identified issues present in the NVD database, and while this doesn’t guarantee complete security, at least puts to rest the vulnerabilties we already know about and is an easy step that can be taken to avoid known attacks.

Updating to the latest version of SpringBoot, 3.3.1, would eliminate the vast majority of vulnerabilities in our project (Halbritter, M., 2024), while also introducing newer versions of all its dependencies, further eliminating the amount of known insecure code that is being brought into the project. This is a similarly easy way to prevent attackers from being able to utilize readily-available and replicable vulnerabilities in our system.

Now, we will discuss steps that will improve the security of our own code in the project. Firstly, the application itself and the database user the application authenticates as should not be the root user of the system. Each user should have the least amount of privilege necessary to perform its function, in the event that malicious access is successful against these user accounts (since no system can ever be 100% secure). The application should have only the ability to a.) access specifically the data it needs from the mysql server, b.) make internal connections only on the database port, and c.) make outgoing connections only on the https port. The database should similarly only be able to a.) access and manage the files related to the database and b.) accept incoming connections from the local network on the database port.

It may also be better practice to store the authentication credentials for the mysql server in a local file that is accessed at runtime rather than compiled in and hardcoded for several reasons. This allows for easily changing the credentials should this ever need to happen (this should occur regularly, as should also be done for any user credentials) by modifying the file, while also meaning that the source code does not contain these details in plain text, should it ever be published or accessed by developers who need not know this information. Furthermore, the JDBC connection used between the application and database should be over SSL to ensure encryption of these credentials is in place.

Every single string of input from the client must be validated before being accepted into the program. This includes input that is taken as part of the CrudControlller and GreetingController classes. When unvalidated input is taken into the application, it is completely open to injection attacks, malformed requests that could expose unknown vulnerabilities, and undefined behaviour that could lead to denial of service. When reduced to a set of known inputs and known outputs, with some variation in the input as required by specific use cases, a much more predictable set of behaviours can be defined. Each input should be inspected for its length, undesirable characters (such as ‘;’, SQL commands, etc), and other attributes that may affect the runtime of the program in malicious ways (null values where a string is expected, characters where numerical input is expected, etc).

The program should also include some method of authenticating users and using this authentication to authorize different levels of access. For instance, the CrudController class accepts ‘business\_name’ as a parameter, where it really ought to be an attribute of an authenticated user, wherein the user is only able to access data for their particular business. This removes the possibility of an attack the client may try to execute to access this data by making the decision to send the data wholly performed on the server with locally stored information.

Finally, the deposit method of the customer.java class, while not a threat on its own, could be improperly utilized elsewhere in the code to allow for malicious manipulation of the data stored in the customer class. For instance, imagine an attacker accessing the application with a request like so “<http://server/account?deposit_amount=100000000000>.” If our program blindly accepts this parameter and passes it directly to the deposit method (as we currently do for the Greeting and DocData contsructors), then an attacker could arbitrarily modify the amounts stored in accounts without proper authorization or verification that the transaction is legitimate. To prevent this, it would be best, instead of relying on other areas of the program to responsibly use this method, to make the method private and introduce a try\_deposit() method that *includes* checks that the user making this transaction is authenticated, that the user is authorized to make this deposit, and verifies with other parts of the financial system that this deposit has *indeed* taken place and is a legitimate transaction before finally calling deposit() and passing the correct amount *as gathered from the verification process and not from user input*.

With the above mitigations in place, there should be far less room for attackers to manipulate access to the web server.

Resources.

Bouncy Castle. (2024, May 16). *Documentation Bouncy Castle Java - Bouncycastle*. Bouncycastle. https://www.bouncycastle.org/documentation/documentation-java/#release-notes

*FTC safeguards rule: What your business needs to know*. (2023, October 6). Federal Trade Commission. <https://www.ftc.gov/business-guidance/resources/ftc-safeguards-rule-what->your-business-needs-know

Halbritter, M. (2024, June 20). *Spring Boot 3.3.1 available now*. Spring Boot 3.3.1 Available Now. https://spring.io/blog/2024/06/20/spring-boot-3-3-1-available-now