****

# 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** | **03/19/2023** | **Christopher Richards** | **Initial documentation** |

## Client



## Instructions

Submit this completed vulnerability assessment report. Replace the bracketed text with the relevant information. In the report, identify your findings of security vulnerabilities and provide recommendations for 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 choose to include images or supporting materials. If you include them, make certain to insert them in all 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

Chris Richards

## Interpreting Client Needs

The client, Artemis Financial, wants to modernize their application with the latest in software security. Their application uses a RESTful API that should be protected against external threats and unauthorized access as well as any existing security vulnerabilities should be assessed. Since Artemis Financial provides financial services, there are certain government regulations that would need to be implemented within the RESTful application. Secure communication between Artemis Financial’s API and the end user should be encrypted using guidelines set forth by the governing body over said regulations. If there are international transactions, certain government regulations may set limits on individual and aggregate transactions for any individual entity.

## Areas of Security

* Cryptography – As a financial institution, Artemis Financial will have a responsibility from a regulatory and ethical perspective to prevent data from being transmitted via plain text, which could be intercepted by unauthorized third parties.
* Input Validation – If Artemis Financial’s RESTful API is to receive any sort of requests that perform CRUD operations, input validation needs to be performed to prevent any kind of malicious input causing injection attacks or input that could cause error states that could result in a denial of service.
* Code Error – Again, if Artemis Financial’s RESTful API receives requests that perform CRUD operations, secure error handling needs to be done to prevent inadvertently exposing details of the application architecture within the HTTP error responses or prevent unhandled error states that could cause a denial-of-service condition.
* Code Quality – Due to the sensitive nature of customer data that is being used, industry best practices, regulatory guidelines and trusted third-party and trusted opensource libraries should be utilized. Opensource libraries will allow for inspection of source code, as well as modification of source code to ensure it meets code standards for the project.
* Encapsulation – Because the data that Artemis Financial houses and serves within their RESTful API, it would be necessary to encapsulate some methods and fields of classes within the program to limit access to information and functionality that should not just be freely accessible.

## Manual Review

The following vulnerabilities were found within the RESTful application source code provided by Artemis Financial.

* Input validation – Within the “/greeting” mapping of the GreetingController class, it is defined that the request parameter “name” should be a string. There is not check to ensure that a string and not some other datatype is entered before using the name variable to create a greeting.
* Secure Error Handling – In the read\_document method of the DocData class, the server and instance name, as well as username and password of the SQL server are defined in plain text within the application code. While this section of code is contained within a try block and exception handling is implemented, the exception handling prints the stack trace, potentially leaking all necessary data to gain unauthorized access to the application SQL database. This really can be considered both a Secure Error handling issue as well as a Code Quality issue.
* Code Quality – In the CRUDController class, when a new CRUD object is created via the CRUDController, the toString() method is utilized by an instance of the DocData, but the toString() method is not overridden in the class. This will output the memory address of the reference to the object instance. This, at the very least, will produce undesired output, possibly of a vulnerable nature.
* Encapsulation – The customer class has no constructor. The account\_number field is a private int, but the account\_balance field does not specify whether it is public or private. This prevents two problems. Since there are no setter methods for account\_number it can never be set. Since the account\_balance field is not specified as private or public, it is essentially a public field and an be freely accessed. This presents a vulnerability where specially crafted requests could directly change the balance of an account.

## Static Testing

Graphical user interface, text, application, email

Description automatically generated

The following vulnerabilities were found within the following external libraries within the existing application. It should be noted that false positives relating to functionality not being used within external dependencies were suppressed.

* Bouncy Castle (bcprov-jdk15on-1.46.jar). Bouncy Castle is a package containing cryptographic algorithms. While the current implementation of the application for Artemis Financial does not specifically utilize Bouncy Castle’s cryptographic features, the nature of data being transmitted to and from the Artemis application will require encryption. The following CVEs were found for Bouncy Castle:
  + CVE-2016-1000338 – In this vulnerability, the DSA (digital signature algorithm) does not properly validate signatures encoded in ASN.1. This makes it possible to inject data into a signature and still validate it as if it were authentic.
  + CVE2016-1000342 – In this vulnerability, the ECDSA (elliptic curve digital signature algorithm) does not properly validate signatures encoded in ASN.1. This makes it possible to inject data into a signature and still validate it as if it were authentic.
  + CVE-2016-1000343 – The DSA in Bouncy Castle in version 1.55 generates a weak private key if the key pair generator is used with default values.
  + CVE-2016-1000341 – In this vulnerability, the DSA has a vulnerability dealing with the signature generation. Bad actors could closely monitor the timing of signatures, which could lead to the ascertaining of the k value of the signature, due to a lack of blinding.
  + CVE-2016-1000345 – This vulnerability made it possible to identify when decryption fails due to padding when signature timings are able to be monitored.
  + CVE-2016-13098 – In this vulnerability, RSA key negotiation could allow an attacker to retrieve the private key of an application due to a weak Bleichenbacher oracle when using TLS cipher suites using RSA key exchanges.
  + CVE-2015-7940 – Bouncy Castle is not validating whether a point is within an elliptic curve, which can allow attackers to retrieve private keys using specific elliptic curve Diffie-Hellman key exchanges.
  + CVE-2016-1000316 – In this vulnerability Diffie Hellman public keys are not fully validated, possibly allowing invalid public keys to reveal private key information where a static Diffie-Hellman cipher is used.
* Hibernate Bean Validation (hibernate-validator-6.0.18.Final.jar). Hibernate is used to validate Java beans. This also relates to Expressive Language expressions. While this was not specifically in the client scenario, Expressive Language is used to validate Java beans.
  + CVE-2020-10693 – In this vulnerability, a bug allowed invalid Expressive Language expressions to be treated as valid expressions. This could allow an attacker to bypass input validation of EL expression inputs.
* Spring AOP (spring-aop-5.2.3.RELEASE.jar). This is part of the Spring framework.
  + CVE-2020-5421 – Protections against reflective file download (RFD) attacks can be bypassed using jsessionid path parameters.
  + CVE-2022-22950 – Specially designed SpEL expressions can trigger a denial of service.
* Spring Core (spring-core-5.2.3.RELEASE.jar)
  + CVE-2022-22950 – This is the same vulnerability previously found in the Spring AOP library. Specially designed SpEL expressions can trigger a denial of service.
* Apache Tomcat (tomcat-embed-core-9.0.30.jar). This is the core of the Tomcat webserver implementation.
  + CVE-2020-11996 – Specific sequences of HTTP/2 requests can cause a spike in CPU usage and cause the server to become unresponsive, effectively causing a denial of service.
  + CVE-2020-17527 – In some cases, Tomcat may reuse HTTP request headers from previous HTTP/2 streams, causing an information leak between web requests.
  + CVE-2021-41079 – Improperly validated TLS packets could occur during very specific TLS configurations, causing a denial of service.

## Mitigation Plan

Regarding mitigation dealing with the CVEs listed in the Static Testing portion of this report, suggested remediation for these CVEs should include an upgrade to the latest version of the Hibernate, Spring Framework and Tomcat dependencies. The client scenario does not specifically mention a need to retain a specific version of any dependency, leading to the recommendation that these dependencies should be upgraded to the latest release versions.

The following remediations should be made regarding the mentioned vulnerabilities listed in the Manual Code Review section of the report:

* Within the input validation issue regarding the greeting mapping within the GreetingController class, the name variable should be checked to ensure that it does in fact contain a string data type, or explicitly cast to a string, though the fact that the parameter is defined as a string should cause a runtime error if any other data type is passed within the parameter.
* To remediate the secure error handling / secure coding issue regarding the read\_document method of the DocData class, the stack trace should not be output upon reaching any exception, except if debugging parameters are placed in a nonproduction environment. Instead, an error message pertinent to the exception should be included in the output. The server and instance, as well as the user credentials for the SQL driver should be stored independently from the source code. This could be done using an encrypted credential file, a java key store if private key authentication is being used, or if the application is being hosted on a Windows system, using SQL Server, the database driver can specify the trusted connection option as true. This would allow the application to be run under a service account with proper credentials, ideally using principle of least privilege, to connect to the SQL server using its existing active directory session as authentication.
* For the code quality issue regarding the CRUDController class invoking the toString method of the DocData class, this can be remediated by overriding the toString method, returning some string representation of the DocData instance instead of the memory address of the object instance.
* To remediate the encapsulation issue regarding the customer class, the account\_number and account\_balance should both be private fields of the customer class. At the very least, a default constructor should be defined that initializes these fields to some default value, though multiple constructors should be defined, in accordance with the principles of polymorphism. These class fields should not be accessible directly, and should only be accessed using “getter” methods. These fields should also only be modified using “setter” methods, or methods that perform some modification of the data, as in the deposit method of the customer class. Two more items to note regarding the customer class: To maintain a proper code quality, the customer class should be named Customer, as to keep with the standard of using a capital letter for the first character in the definition of a class. It can also be noted that while the class is named customer, it seems that this class houses information about the customer account and not the customer themselves. A more appropriate name could be Account, which would serve a self-documenting purpose of the class.