****

# Practices for Secure Software Report

Table of Contents

[Document Revision History 3](#_Toc102040754)

[Client 3](#_Toc102040755)

[Instructions 3](#_Toc102040756)

[Developer 4](#_Toc102040757)

[1. Algorithm Cipher 4](#_Toc102040758)

[2. Certificate Generation 4](#_Toc102040759)

[3. Deploy Cipher 4](#_Toc102040760)

[4. Secure Communications 4](#_Toc102040761)

[5. Secondary Testing 4](#_Toc102040762)

[6. Functional Testing 4](#_Toc102040763)

[7. Summary 4](#_Toc102040764)

[8. Industry Standard Best Practices 4](#_Toc102040765)

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **10/18/2025** | **Michael Foster** |  |

## Client



## Developer

Michael Foster

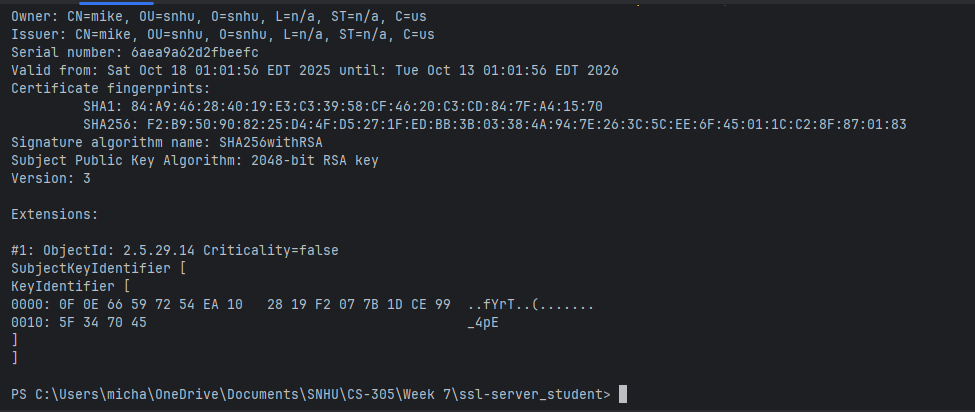
## Algorithm Cipher

Artemis Financial will likely face many threats from malicious actors attempting to access customers’ sensitive financial data. To protect against this data must be secure during transport, and at rest. This will require the use of multiple tools. For data in transit encryption and hashes’ will be necessary to ensure that data is not only secure but also that integrity is maintained. To best achieve this, I would recommend AES-256 encryption paired with the TLSv1.2 protocol. This will ensure that the best practices are followed while using a 256-bit encryption which should assist in maintaining compliance with regulations like the Gramm-Leach-Bliley Act as new threats appear and regulations require stronger security measures. This partnership of using AES and TLS is so vital because AES uses symmetric keys. An encryption system that has symmetric keys uses the same key to encrypt and decrypt the data which means that keys must be exchanged between the user and client. This is clearly a security vulnerability, but TLS provides a solution as it is asymmetric. This allows the client and server to create a secure handshake and establish a secure connection to pass keys and then implement the higher performance symmetric key usage. To ensure the integrity of what has been sent securely a hashing algorithm should be used so that data checksums can be put in place. The best choice for this is the SHA-256 hashing algorithm. This algorithm is best because the chance of collision is 1 in 2^256 greatly reducing the chance of a collision and therefore creating a valid system to check integrity of data. While the data is at rest it should also be encrypted to protect data in the case a malicious actor gains access to the system. The AES-256 encryption is still the best choice for this purpose as it is a robust encryption that is extremely resistant to brute force attacks.

Algorithms utilize random numbers as the starting point to prevent malicious actors from being able to quickly break the algorithm. To capitalize on how randomness helps prevent malicious actors from determining how the algorithm is working new random numbers are injected as the cipher is used to prevent a recognizable pattern from occurring and assisting an actor in their efforts to circumvent the encryption.

As new threats are constantly emerging and the computing power available to malicious actors increase the requirements of encryption algorithms will continue to evolve. The best example of this can be seen in bit size of the algorithms which are directly to contend with the increased computer power of at home computers. 128-bit encryption was considered to be extremely secure but in recent years a push to 256-bit encryption has become common place to protect from moder threats. Another evolution is that TLSv1.2 now recommended s to not use RSA, which was a standard usage in the past but now using DHE or ECDHE is the preferred method.

## Certificate Generation



This screen shot shows a self-generated certificate that was created using Java Keytool. Self-signed certificates are utilized by developers in a development environment to allow the use of HTTPs. In a production environment a server certificate registered with a certificate authority would be used to ensure users are able to trust that they are on the right web domain. In the development environment we self-generate certificates to allow verification that the system can connect utilizing the proper protocols and the connection is secure.

## Deploy Cipher

A computer screen shot of a program code

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

The first screen shot shows the code generated to create the checksum verification. This is done using three different methods. In order of calls it starts with the myHash method. This method is called when the “/hash” path is requested from the server. It takes the string of data that will be passed and calls the getHash method to generate a SHA-256 hash. In this method the Java.Security library is used and the function MessageDirect.diget is used to hash the string into a byte format. The last method is called byteToString. In this method I convert the byte representation of the hash into string format. This is then returned through all the calls and passed with the original string to the client’s side.

## Secure Communications

A screenshot of a computer code

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

To secure the communication of the application the application.properties file must be modified. In this file we set the port to 8443 which is typically used in development environments in place of 443 which is HTTPS. The other details are to provide the details for the self-signed cert so that a secure connection can be made. In the second screenshot we can see that https is active, although because we are using a self-signed certificate the browser is providing a warning that the certificate in use is not registered with a CA and could be the indication of a malicious site.

## Secondary Testing

A black screen with text

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

Running the OWASP dependency check revealed 15 software library dependencies with known CVE’s. All the software dependencies are dependencies that are introduced to the system through the use of spring-boot-2.2.4. These libraries create vulnerabilities for code injection attacks, man in the middle attacks, DoS attacks, remote code execution, security bypass, hijacking, redirection attacks, improper privilege management, log neutralization, and resource allocation attacks. All the dependency vulnerabilities identified in the check are real vulnerabilities, but with how the application is currently configured and the functionality it provides the vulnerabilities do not currently provide an applicable exploit. Most of the vulnerabilities are related to specific protocols like SMTP which the application does not use or introduced to the system through the deserialization of malicious data which the program does not do. This provides two possible solutions either to suppress the vulnerabilities as they are not a current threat or refactor the application to utilize a more recent and currently more secure version of spring-boot.

## Functional Testing

A screen shot of a computer

AI-generated content may be incorrect.

A black background with text

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.  
  
 Following the dependency check and keeping in mind the likely additional features that will be added to this application the vulnerabilities identified will likely provide an attack surface with future feature addition. With the application in an early stage of development I decided that the version of spring-boot used should be upgraded. I chose to refactor to spring-boot 4.0.0-M3 due to the only identified vulnerability with the spring-boot-starter-parent version 4.0.0-M3 is that there is a possibility of injection attacks against LDAP authentication checks. This vulnerability can be corrected by upgrading to Java 21 and Derby 10.17.1.0. Although Derby is not in use by the system, I needed to refactor the version of Java to support this library change, so I chose to upgrade to version 21 to prevent future risk.

## Summary

To conduct this software security test a standard process was carried out. The first step was to identify the risks to the system. Thinking of who the possible malicious actors would be, and what their motivation is helped identify the areas within the application that pose the greatest risk. When working through the areas that I identified as most important and addressed in this application were Cryptography through the implementation of encrypted communication between the client and server, Client/Server implemented though the use of correct protocols and use of certificates to ensure secure connections. Code Error by implementing error handling where required, and code quality by using updated library dependencies and proper design patterns. Each one of these choices and implementations is vital to secure one area but are used in conjunction to create defense-in-depth strategy. By using certificates and HTTPS to secure communication between client and server we have a trusted environment. By adding checksums within the secure environment, we ensure that even if our cryptography fails, we can identify data integrity violations within the system. Lastly by identifying dependency vulnerabilities and refactoring to correct them we reduce the attack surface which assists in hardening the overall system.

## Industry Standard Best Practices

Industry standard best practices are identified throughout the recommendations and corrective actions applied. The first was to move to the use of HTTPS/TLSv1.2 ensuring that data is transmitted is utilizing the standard security protocols. This implementation was also tested with self-signed certificates to ensure functionality, but the recommendation to use a CA in the future is in line with standard security practices. The implementation of AES-256 and SHA-256 are both in line with the recommendation of the National Institute of Standards and Technology. Lastly keeping software up to date by identifying and updating dependencies to ensure a reduced attack surface and applying a defense-in-depth strategy are security principles found in all technological domains. These practices are important not only because they are the best practices to reduce attacks, but they also increase the ability for applications to integrate or interact with other solutions expanding capabilities.