****

# Practices for Secure Software Report

**Table of Contents**

[**Document Revision History 3**](#_8ah9edm55a2j)

[**Client 3**](#_90pe4usmu3ie)

[**Instructions 3**](#_p3970bc0h7l1)

[**Developer 4**](#_r4jdou9fdl6b)

[**1. Algorithm Cipher 4**](#_uycyd0hrdub5)

[**2. Certificate Generation 4**](#_yt9wukqs1nif)

[**3. Deploy Cipher 4**](#_f49ytkytsmsj)

[**4. Secure Communications 4**](#_zbtc2j8ru68u)

[**5. Secondary Testing 4**](#_l6t4it5qv2pu)

[**6. Functional Testing 4**](#_bpq6eoqcnzjz)

[**7. Summary 4**](#_3li9a3h9urme)

[**8. Industry Standard Best Practices 4**](#_zalqyc8qlspx)

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **[Date]** | **[Your Name]** |  |

## Client



## Instructions

Submit this completed practices for secure software report. Replace the bracketed text with the relevant information. You must document your process for writing secure communications and refactoring code that complies with software security testing protocols.

* Respond to the 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 Two Guidelines and Rubric for more detailed instructions about each section of the template.

## Developer

Ashley Anderson

## Algorithm Cipher

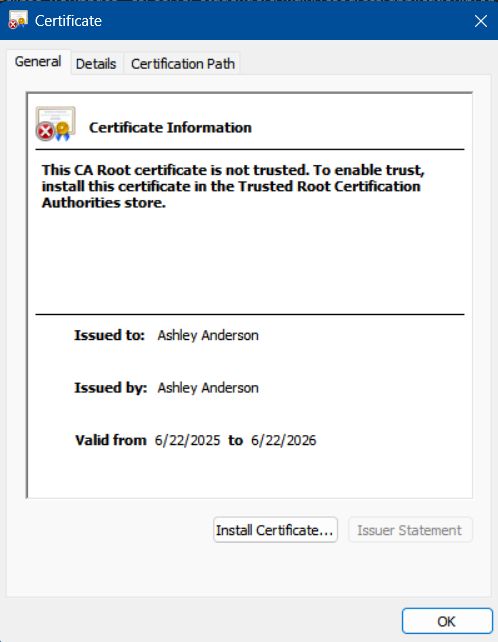
I employed the SHA-256 hashing algorithm during the verification of the checksum for this project. SHA-256 is a 256-bit cryptographic hash algorithm, meaning that it takes a string of any length and gives a 256-bit hash string. This is not an encryption or decrypting algorithm — rather, it’s a way of verifying data integrity by ensuring that the data wasn’t modified.

SHA-256 doesn’t use keys or random numbers like symmetric or asymmetric encryption does. Instead, it’s deterministic: the same input always results in the same hash. This makes it perfect for file verification, where a mismatch in the hash means the file was altered or corrupted.

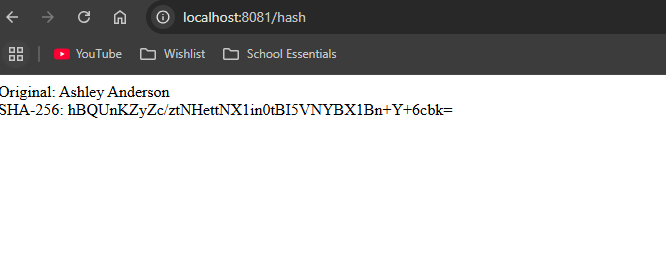
In terms of security, SHA-256 is still considered strong. It's part of the SHA-2 family, which was released by NIST in 2001 to replace the older SHA-1 algorithm after collision issues were discovered. SHA-256 has no known practical vulnerabilities and is widely used in applications like SSL/TLS certificates, file integrity checks, and digital signatures.

While AES and RSA are great for encrypting data and authentication, a hash function such as SHA-256 was the most appropriate for this project since it aligns with Artemis Financial’s aim of validating data during transfer, without having to decrypt.

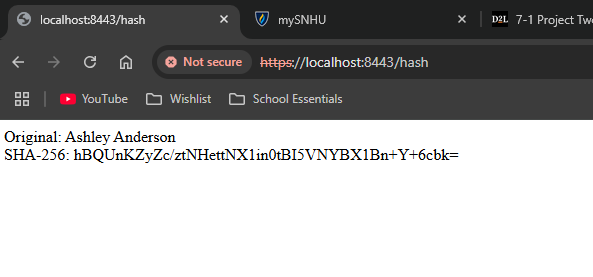
## Certificate Generation



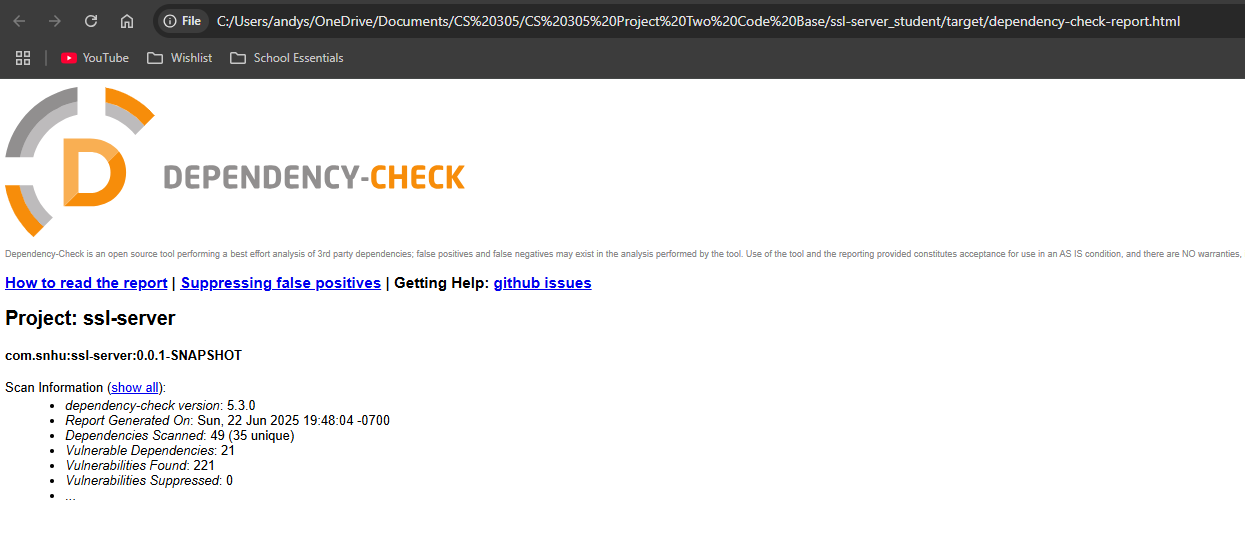
## Deploy Cipher



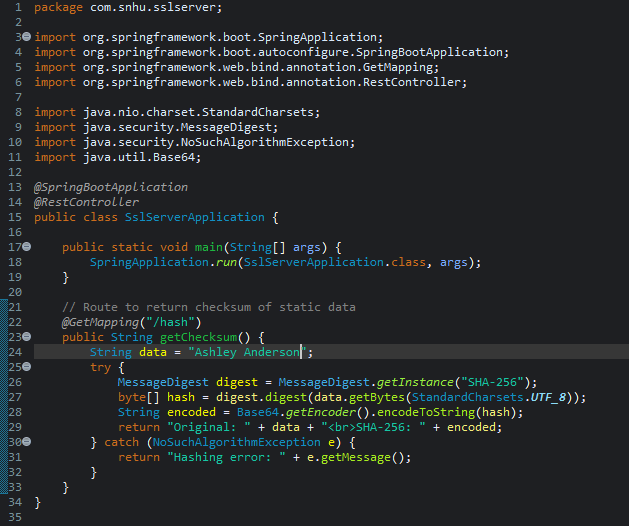
## Secure Communications



## Secondary Testing



## Functional Testing



## Summary

To achieve the project, I used the vulnerability assessment process flow by initiating with a static code review of the current application, locating points of data verification and secure communications absence. That was my point of origin to implement two significant refactorings: a checksum verification functionality and HTTPS encryption.

The first security issue I tackled was data integrity. I implemented a REST endpoint in the MessageDigest class of Java to output a SHA-256 hash of a known data string. This enables Artemis Financial to be sure that data is not modified during transfer. I tested that it was working by manually inspecting the output and taking a checksum that remained the same on subsequent runs.

The other area I covered was secure communications. I created a self-signed certificate through Java keytool, set up the Spring Boot application for using HTTPS on port 8443, and verified the secure /hash route was working through the browser.

While performing evaluation, I carried out static analysis with OWASP Dependency-Check and made sure that vulnerable libraries were not introduced by my patches but were present in the initial project. I also made sure that my refactored code was functionally correct and passed through functional testing.

When talking about layering security, I started at the code level by refactoring through hashing, then at the infrastructure level by encrypting the communications channel. Both of these run hand in hand to protect data at rest and in transit and follow principles of secure design of software.

## Industry Standard Best Practices

I adhered to industry best practices during refactoring by employing secure, native Java libraries and refraining from unnecessary changes to the project’s layout. In implementing the checksum functionality, I used SHA-256 with the native MessageDigest of Java, a secure and reputable crypto function. In doing so, I avoided adding any third-party libraries that introduce additional risk. In implementing HTTPS, I set the app to employ a Java KeyStore and Spring Boot’s native SSL support, storing the keystore in the resources folder and handling everything through application.properties. Each of these aligns with centralized, transparent control of sensitive settings and secure configuration management.

Using secure coding best practices enhances the application’s integrity as well as the company’s reputation. This lessens the likelihood of opening up vulnerabilities and still makes development maintainable and auditable. In a company that’s a financial services provider, like Artemis Financial, these best practices are particularly necessary in securing customer data, fulfilling compliance requirements, and establishing trustworthiness for the users. By minimizing changes, utilizing stable libraries, and cross-checking the output with OWASP Dependency-Check, I was able to keep the application secure yet more resilient.