****

# 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** | **4/20/2025** | **Collin Ogren** |  |

## 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

Collin Ogren

## Algorithm Cipher

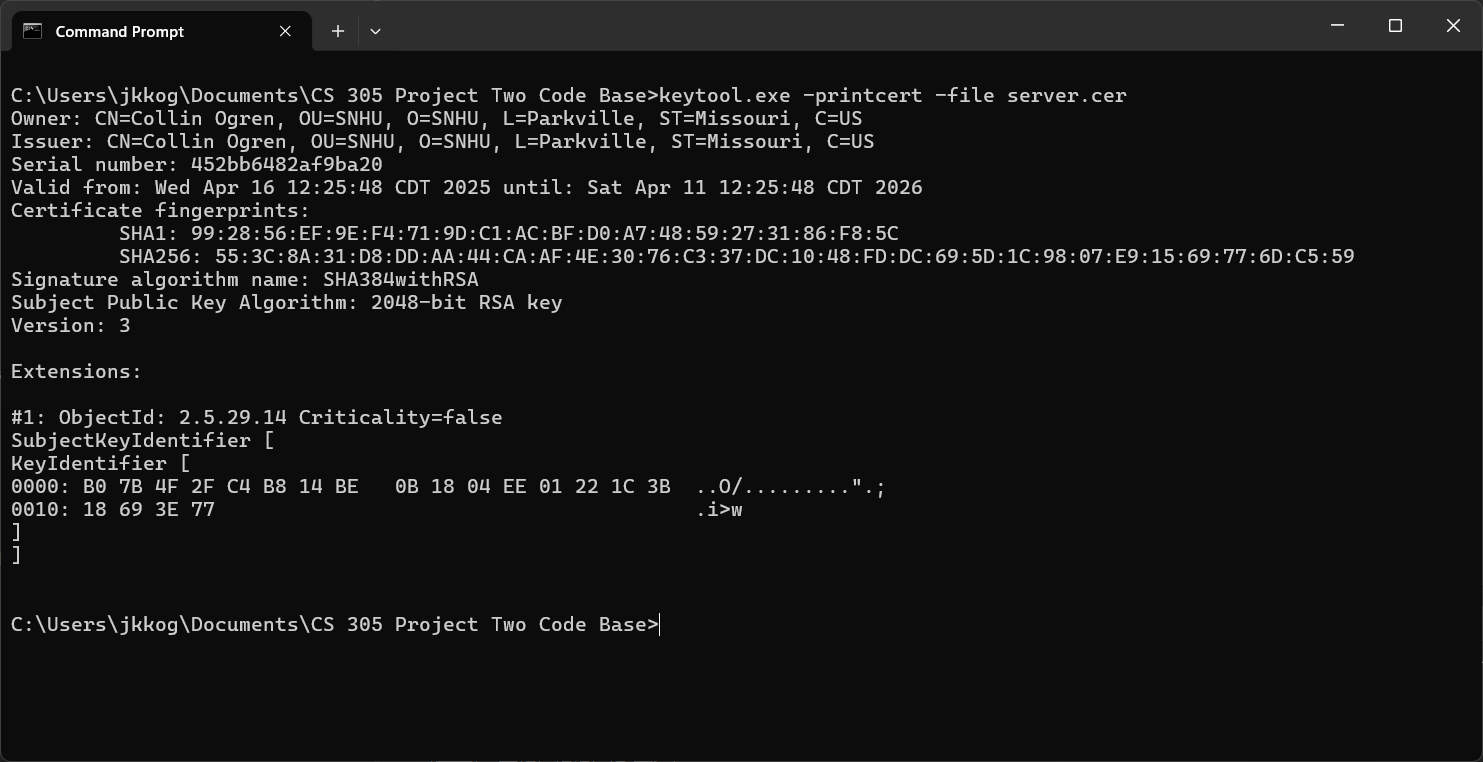
The SHA-256 cipher algorithm was chosen for this project. SHA-256, a SHA-2 algorithm, is a hashing algorithm that was created by the National Institute of Standards and Technology and the National Security Agency as a replacement for older SHA-1 algorithms which were vulnerable to attacks (“What is SHA-256?,” n.d.). Hashing is used to verify the authenticity and integrity of data during transmission. It involves converting a specific input string into a unique, fixed-length hash value through a one-way function, making it computationally infeasible to reverse. This process ensures that the data has not been modified from the outside and remains authentic. The goal of a secure hashing algorithm like SHA-256 is to create a unique output for every input (Cafiero, 2025). However, because of the complexity of this problem, collisions may occur. Some hashing algorithms may be more or less secure than others due to the length of the produced hashes and because of differing chances for collisions to occur. SHA-256 has a hash length of 256 bits and has a chance for collision of 1 in 2^256. This means it is statistically impossible for two inputs to result in the same output (“Sha256 collision,” 2025) and it would be nearly impossible to brute force a matching value due to the length. This makes SHA-256 a solid choice when creating a secure checksum verification system.

SHA-256 is a one-way cipher. It cannot be used to encrypt a message and decrypt it later. This makes it useful for such purposes and verifying a secure checksum, but it does not work for storing data securely at rest or for securing data in transit since it cannot be decrypted. Instead, symmetric-key and asymmetric-key encryption algorithms are used. For data at rest, an algorithm that uses a single key, a symmetric-key algorithm, such as AES can be used to store data at rest and comes in 128-, 192-, and 256-bit versions (“Advanced encryption standard,” 2023). Different modes of operation exist for AES as well such as AES/GCM which encrypts data and also ensures integrity by using authentication tags. AES-256/GCM is nearly impossible to break without the key and would be an optimal choice for data storage at rest. To facilitate encryption of data in transit, an algorithm that uses a private and a public key, asymmetric encryption, is needed. RSA is a very popular example of an asymmetric encryption algorithm that allows for secure communications between client and server. RSA works by having a public key that is known by everyone and is used to encrypt data. The encrypted data is then sent and decrypted by a secret, private key held by the receiver. Only the private key may decrypt the message once it has been encrypted. Therefore, just as with symmetric encryption, it is important that the private key be kept secret or security will be compromised (“RSA algorithm,” 2025). When using TLS, such as in this application, RSA is used to for authentication and passing of the AES private key for a robust encrypted connection.

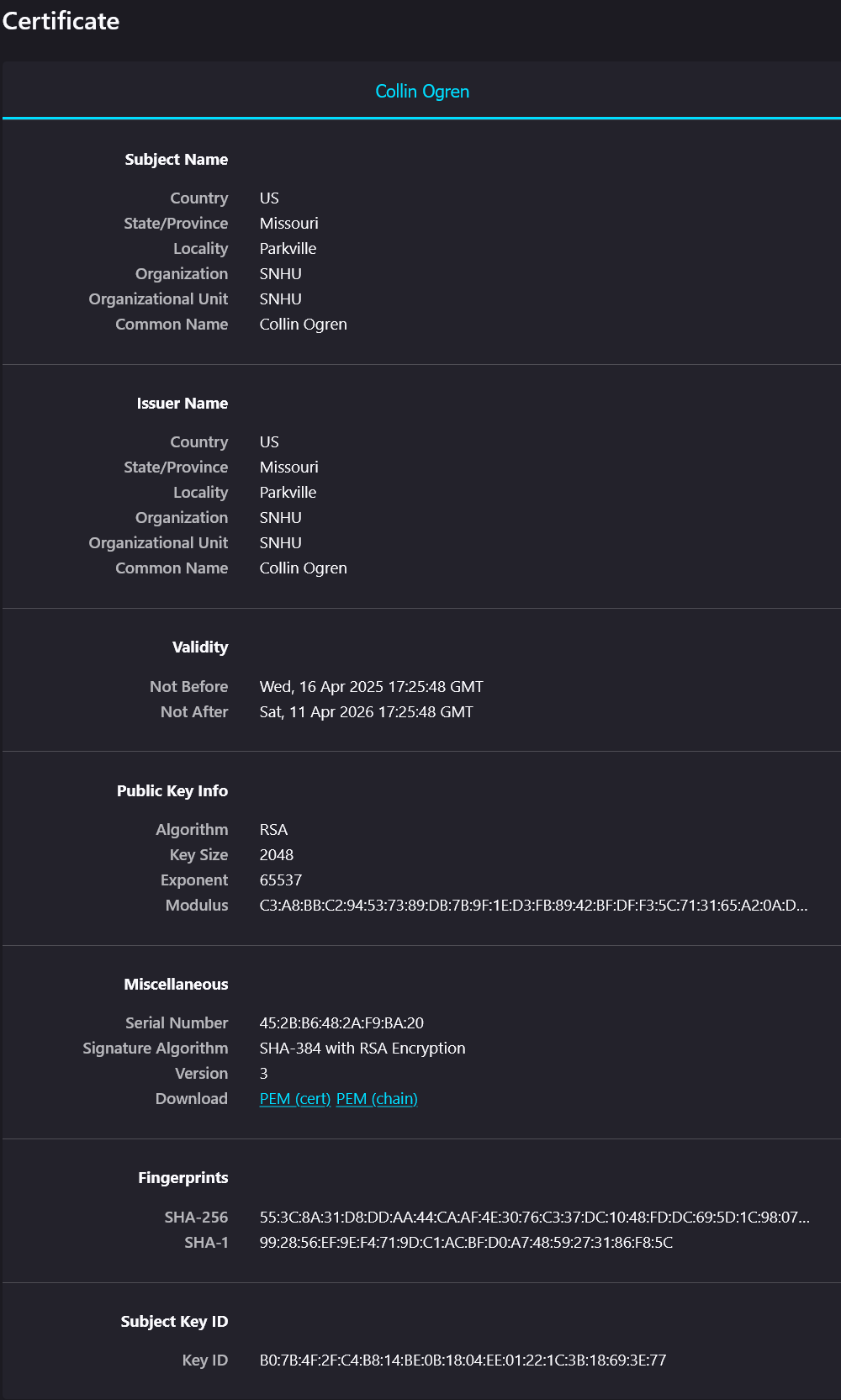
Encryption has evolved from primitive ciphers that relied on constant shifts of letters by a certain number of characters, the Caesar Cipher is one such example, to extremely advanced methods, like AES, that are essentially impossible to break by brute force (Schneider, 2024). Throughout history there have always been code makers and code breakers, this is why it is important to always stay on top of the latest encryption developments to ensure that data always stays secure in the fast-pasted, ever-changing environment of encryption. As of right now, symmetric algorithms such as AES are thought to be resistant to the new threats posed by the advent of quantum computing (“Post-quantum cryptography,” 2025), but asymmetric algorithms like RSA are thought to be vulnerable. However, there are new, post-quantum encryption algorithms are being developed to address these issues (“Quantum computing*,*” 2024). These new algorithms may need to be used if quantum computing ever becomes a large-scale threat. However, classical computers currently cannot crack either RSA or AES in a reasonable amount of time, given a large enough key size.

## Certificate Generation

Insert a screenshot below of the CER file.



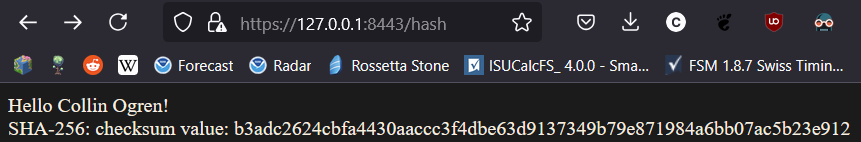
Screenshot of the TLS certificate generated by keytool and viewed with keytool.



Screenshot of the TLS certificate being served through the Java Spring application and viewed in Firefox.

## Deploy Cipher

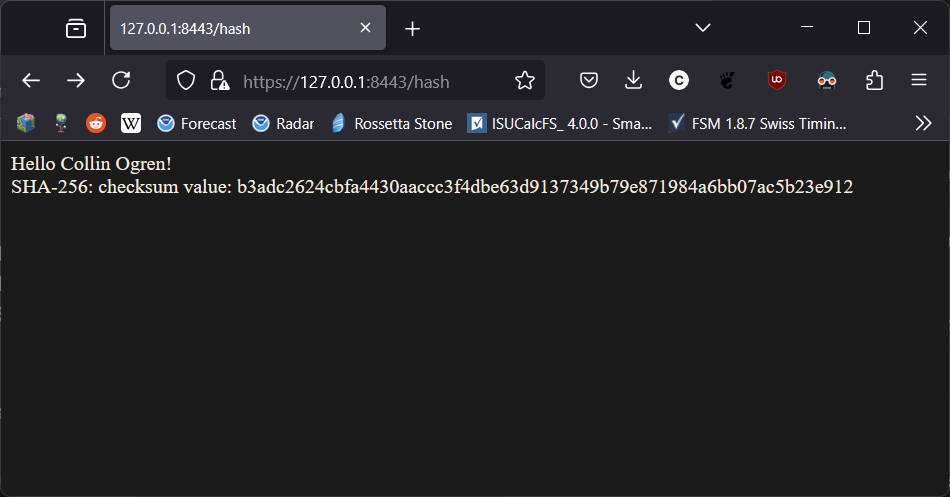
Insert a screenshot below of the checksum verification.



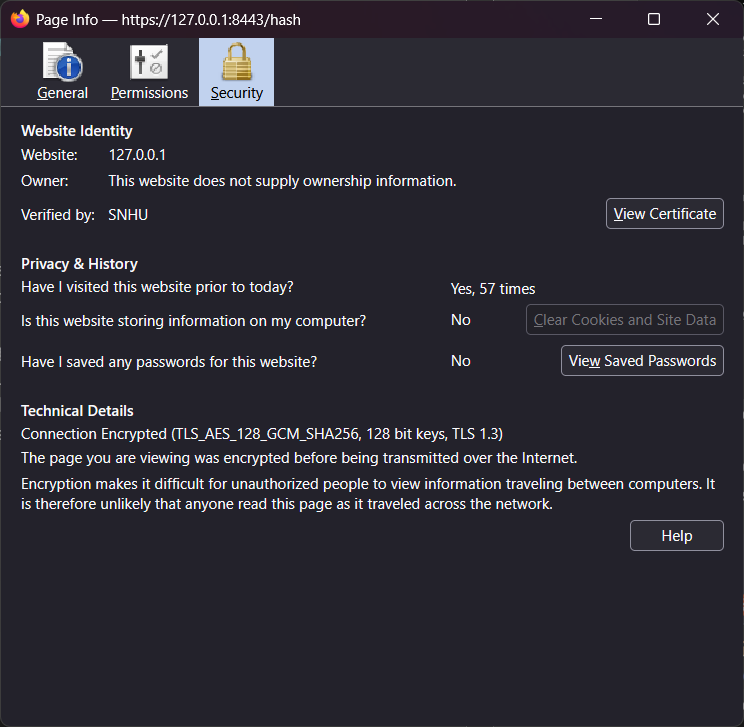
Screenshot of the checksum verification on the /hash mapping.

## Secure Communications

Insert a screenshot below of the web browser that shows a secure webpage.



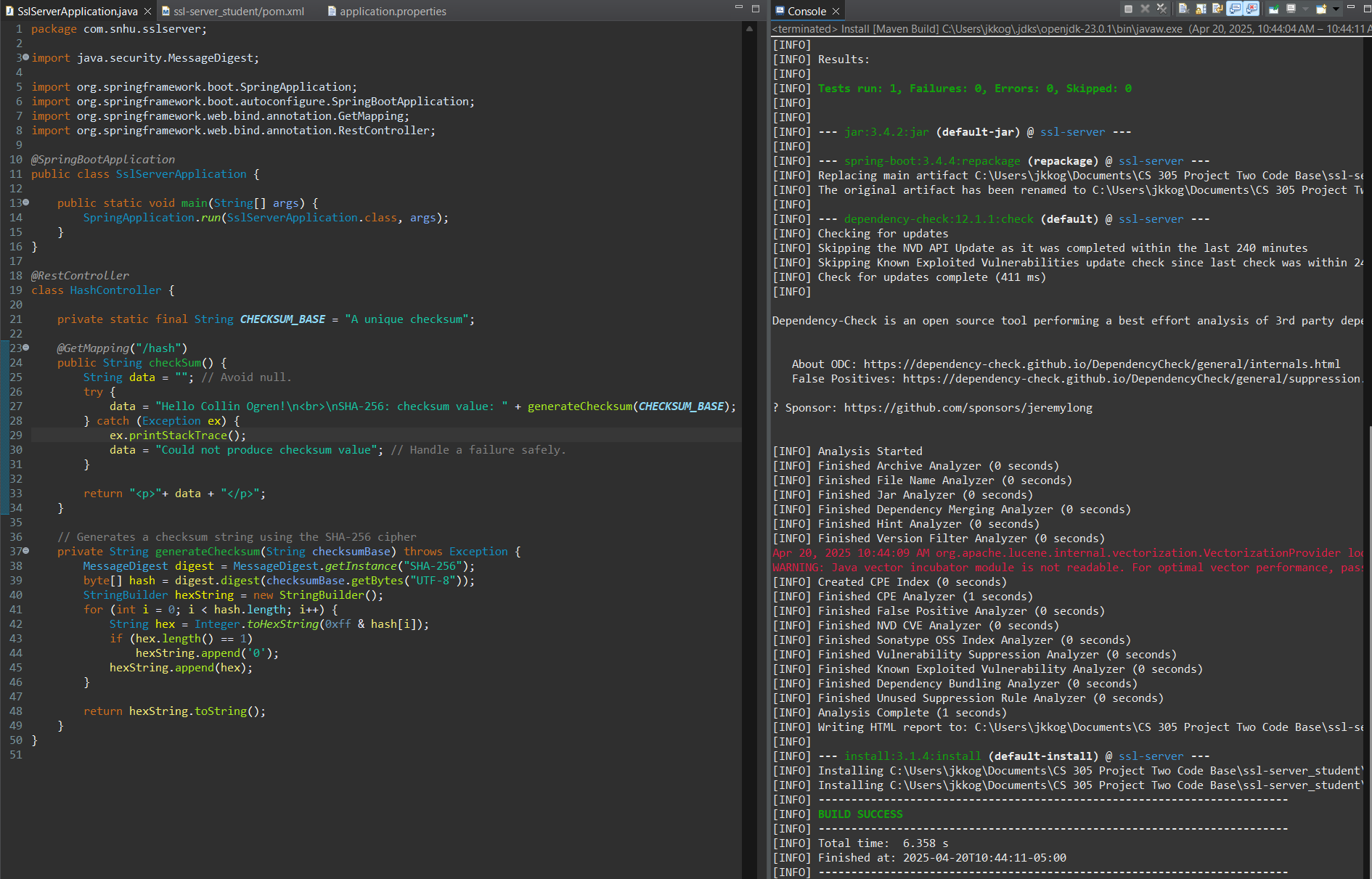
Screenshot of web page showing HTTPS connection. Note that because the certificate is self-signed, most browsers, including Firefox as used here, will still show a warning for the website since the certificate is not signed by a trusted certificate authority.



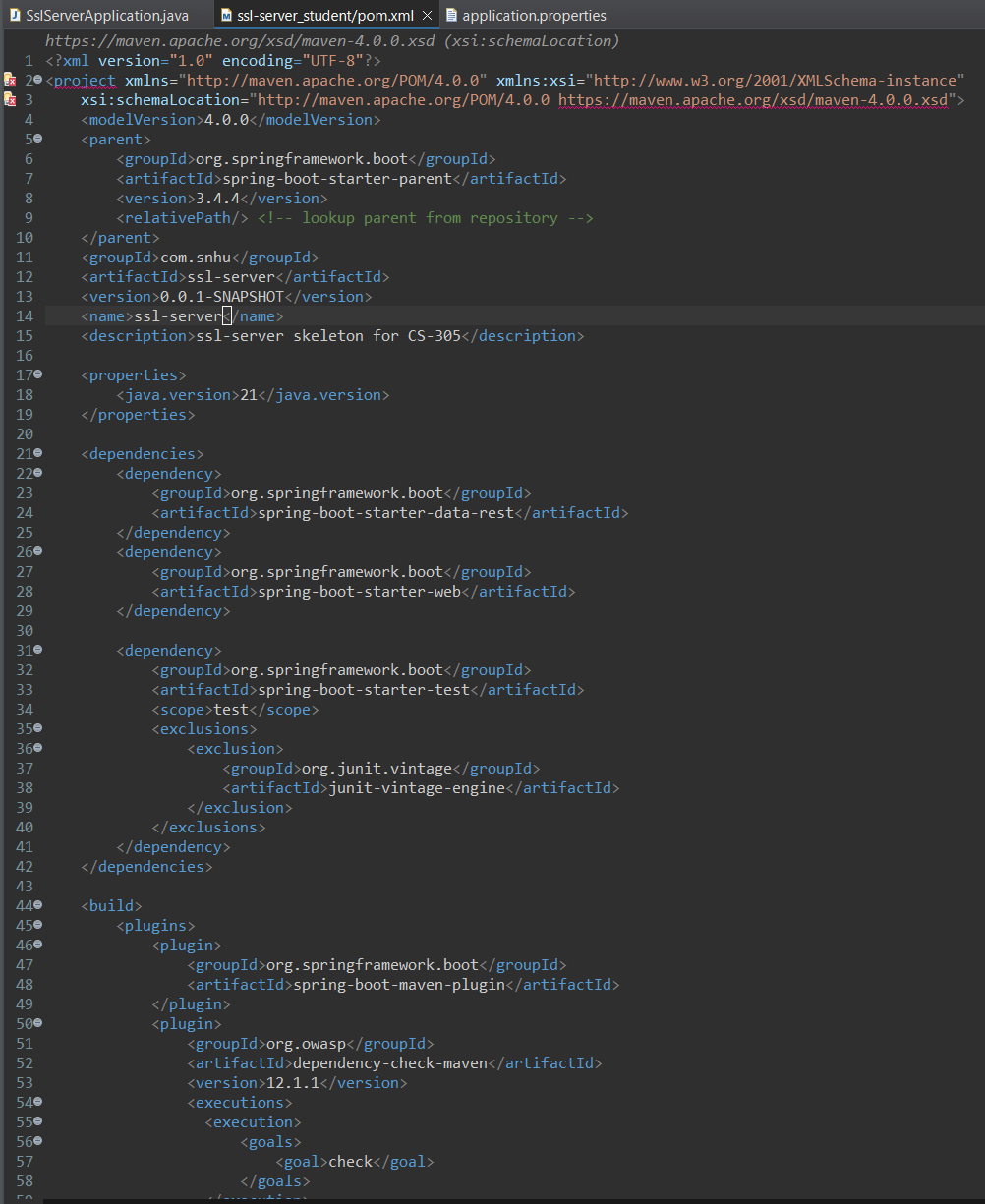
Screenshot of page info showing an encrypted connection. Even though the certificate is self-signed, the connection is still encrypted.

## Secondary Testing

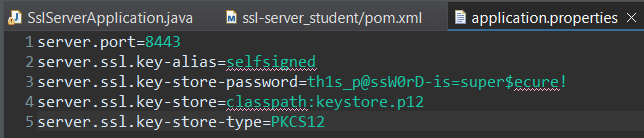
Insert screenshots below of the refactored code executed without errors and the dependency-check report.



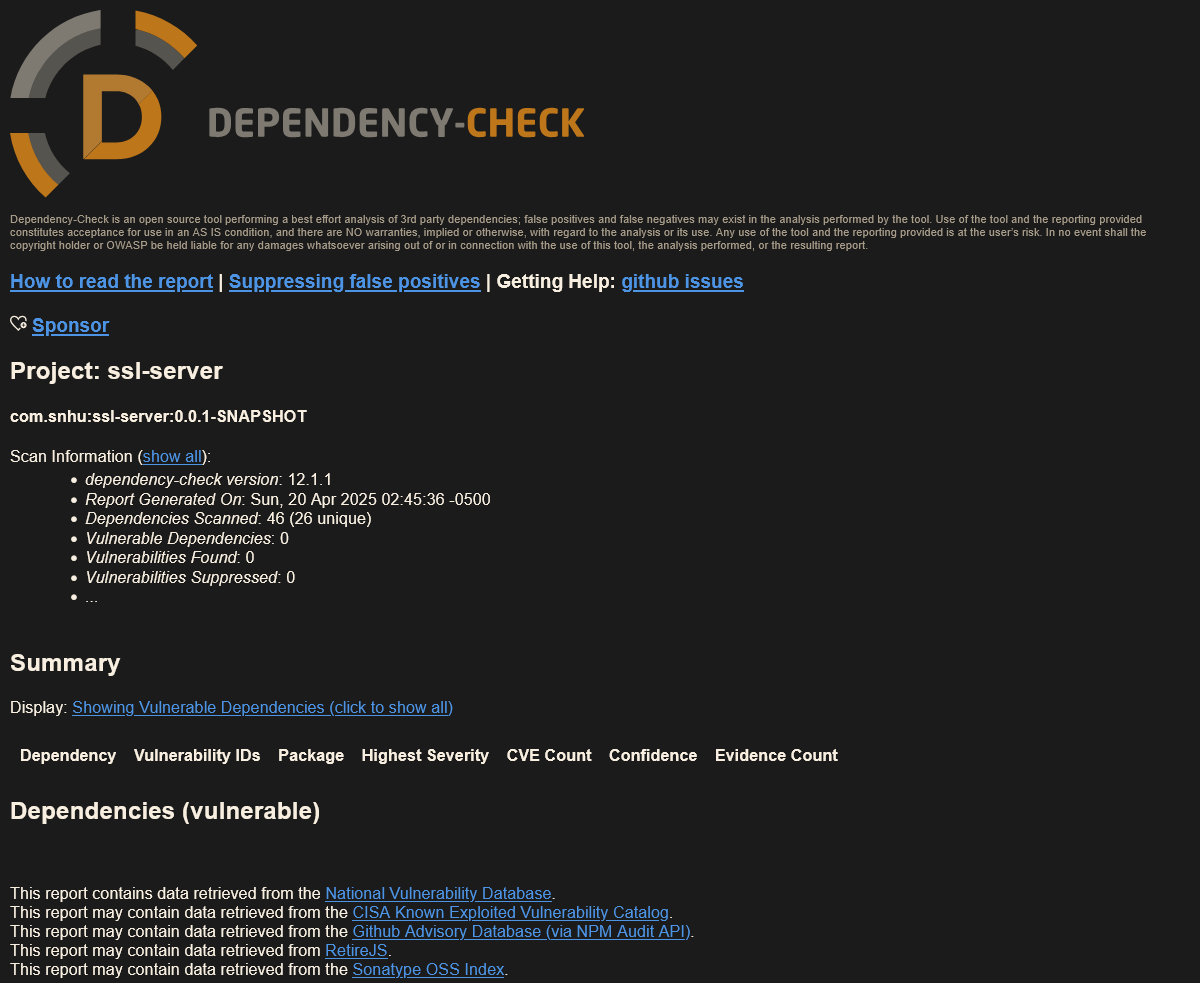
Screenshot of the refactored SslServerApplication.java file with an errorless “mvn install” output.



Refactored pom.xml with updated dependencies.



Refactored application.properties.

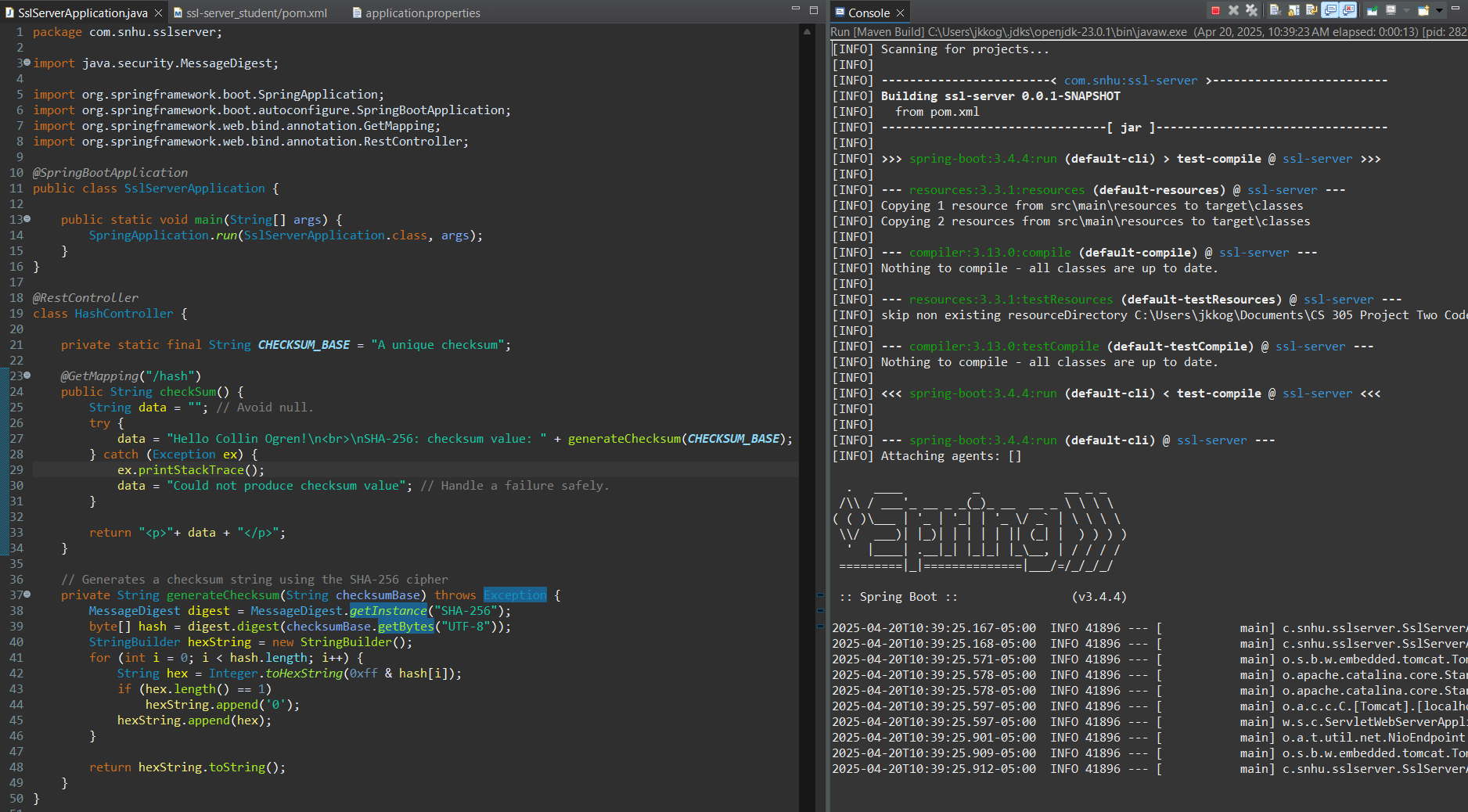


Screenshot of the dependency report showing zero vulnerabilities after updating the dependencies in pom.xml.

## Functional Testing

Insert a screenshot below of the refactored code executed without errors.

The secure checksum functionality was added, adhering to secure software development protocols.



Screenshot of the refactored code running successfully.

## Summary

The codebase has been refactored to address all known vulnerabilities and implement the required software security functionalities. Below is a summary of each change made and its purpose. Major areas of focus during the refactoring were APIs, cryptography, client / server, code error, code quality, and encapsulation. Input validation would be considered if there was user input to validate.

* The third-party dependencies have been updated.
  + All used Spring Boot libraries have been updated to the latest version, version 3.4.4 which was released in March of 2025, from version 2.2.4-RELEASE which was released in January of 2020. Version 2.2.4-RELEASE of Spring Boot Starter Parent, the package that provides all other necessary Java Spring libraries, contains 159 vulnerabilities according to the Maven Repository and Dependency Check showed a similarly large number. For version 3.4.4, however, the Maven Repository and Dependency Check both indicate that there are no known vulnerabilities as of the time of writing.
  + Dependency Check has been updated to version 12.1.1 which was released in April of 2025 from version 5.3.0 which was released in January of 2020. While the tool itself is not a runtime security concern, using an old version of Dependency Check may result in poor quality results, such as missing new vulnerabilities. This could, in turn, lead to security vulnerabilities. In order to avoid potential false negatives and subsequent security vulnerabilities, dependency check has been updated to the latest version.
* Implemented HTTPS using TLS.
  + A self-signed TLS certificate was created. The public key encryption algorithm is RSA 2048 and the signature algorithm is SHA-384 with RSA. Data travelling between client and server is encrypted using TLS\_AES\_128\_GCM\_SHA256.
  + The “application.properties” file was updated to use the key-store file called “keystore.p12” which contains both public and private keys. This change allows for HTTPS to be used, greatly enhancing the security of the web service.
* Implemented the checksum verification system at the “/hash” path.
  + This system uses the SHA-256 cipher algorithm to create a unique checksum string that can be used to verify the integrity and authenticity of data with statistically complete certainty.
  + The Java Spring @GetMapping annotation was used. Because @RequestMapping will map any valid HTTP request, @GetMapping was used to limit the possible ways that the web server can be exploited at that entry point.
* Updated to Java 21 which is the latest long term support Java version.
  + Java 21 fixes security vulnerabilities.
  + Java 21 contains modern features that may be used by the development team in the future to enhance the application.
  + Updating to a new Java version also helps to ensure that dependencies can be updated to the latest version easily, thus enhancing security.

## Industry Standard Best Practices

Industry standard best practices were followed throughout this program. The most significant improvement made to adhere to best practices was the updating of all dependencies. As of the time of writing, no known vulnerabilities remain in the list of dependencies. It is critical that the software team continues to keep third-party dependencies updated to avoid further security risks from third-party libraries. Because no known vulnerabilities exist, no workarounds were needed to ensure the security of the program at this time. Usage of third-party library code adheres to the documentation to remain secure. Java was also updated to the latest long term support version (Java 21) in order to keep the runtime environment modern and secure.

The hashing algorithm used for the checksum generation used to confirm the authenticity and integrity of data sent through the web service is SHA-256. SHA-256 is the industry standard for secure hashing algorithms with a chance for collision, and therefore insecurity, of 1 in 2^256. This is a statistically impossible chance.

The server connection is secured with HTTPS with TLS using RSA 2048 and AES-128 GCM encryption. The certificate signature algorithm is SHA-384 with RSA encryption. Data in encrypted using AES-128 GCM in transit after establishing the secure connection. This means that any data sent between the server and client will retain its authenticity, confidentiality, integrity, and non-repudiation.

Secure coding practices were used such as avoiding null, handling exceptions and possible errors safely without causing the program to fail, and avoiding uses of null. Data is encapsulated properly without unnecessarily permissive visibility modifiers.

These changes provide Artemis Financial with a secure and modern web service which they should continue to maintain to keep the application secure. Artemis Financial may wish to adopt a DevSecOps development pipeline to make security an integral part of their development cycle in the future. By maintaining strong software security principles, Artemis Financial can avoid security incidents that can incur major expenses through the loss consumer trust, lawsuits, and expense to fix damages. An investment in security is a self-repaying investment and should be a focus for all future development.

**References**

Advanced encryption standard (AES). *National Institute of Standards and Technology.* (2023, May 9). <https://doi.org/10.6028/NIST.FIPS.197-upd1>

Cafiero, C. (2025, January 5). Hash collisions. *University of Vermont.* [https://www.uvm.edu/~cbcafier/cs2240/content/09\_hashing/03\_collisions.html](https://www.uvm.edu/~cbcafier/cs2240/content/09_hashing/03_collisions.html%20)

Sha256 collision probability explained. *Restack.* (2025, April 6). [https://www.restack.io/p/secure-hashing-techniques-knowledge-answer-sha256-collision-probability-cat-ai](https://www.restack.io/p/secure-hashing-techniques-knowledge-answer-sha256-collision-probability-cat-ai%20)

Post-quantum cryptography. *National Institute of Standards and Technology.* (2025, March 27). <https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/faqs>

Quantum computing - how it changes encryption as we know it. *Division of Information Technology.* (2024, October 18). <https://it.umd.edu/security-privacy-audit-risk-and-compliance-services-sparcs/topic-week/quantum-computing-how-it-changes-encryption-we-know-it>

RSA algorithm in cryptography. *GeeksforGeeks.* (2025, January 6). <https://www.geeksforgeeks.org/rsa-algorithm-cryptography/>

Schneider, J. (2024, January 5). The history of Cryptography. *IBM.* <https://www.ibm.com/think/topics/cryptography-history>

What is SHA- 256?. *Encryption Consulting.* (n.d.). [https://www.encryptionconsulting.com/education-center/sha-256/](https://www.encryptionconsulting.com/education-center/sha-256/%20)