****

# CS 305 Project Two

**Practices for Secure Software Report**

Table of Contents

[Document Revision History 3](#_Toc33111302)

[Client 3](#_Toc33111303)

[Instructions 3](#_Toc33111304)

[Developer 4](#_Toc33111305)

[1. Algorithm Cipher 4](#_Toc33111306)

[2. Certificate Generation 4](#_Toc33111307)

[3. Deploy Cipher 4](#_Toc33111308)

[4. Secure Communications 4](#_Toc33111309)

[5. Secondary Testing 4](#_Toc33111310)

[6. Functional Testing 5](#_Toc33111311)

[7. Summary 5](#_Toc33111312)

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **2/21/2021** | **Noah Sherry** |  |

## Client



## Instructions

Deliver this completed Practices for Secure Software Report documenting your process for writing secure communications and refactoring code that complies with software security testing protocols.

Respond to the steps outlined below and replace the bracketed text with your findings in your own words. If you choose to include images or supporting materials, be sure to insert them throughout.

## Developer

Noah Sherry

## 1. Algorithm Cipher

Determine an appropriate encryption algorithm cipher to deploy given the security vulnerabilities, justifying your reasoning. Be sure to address the following:

* Provide a brief, high-level overview of the encryption algorithm cipher.
* Discuss the hash functions and bit levels of the cipher.
* Explain the use of random numbers, symmetric vs non-symmetric keys, and so on.
* Describe the history and current state of encryption algorithms.

Artemis Financial is requesting an encryption algorithm recommendation that will be used to encrypt long-term archive files. We should assume that the most likely attack vector for these files will be bad actors somehow acquiring access to these files, so they should be encrypted such that they would be useless even if they were stolen. The files will not be transported anywhere, so there is no need for Asymmetric\* keys to be used. Additionally, there is no need for these files to be encrypted quickly, as they will be archived long-term. Therefore, I recommend using the SHA-256 cipher algorithm with 256-bit keys to encrypt these files. SHA-256 encryption is the most secure default option available within all standard installations of Java, since it provides the highest level of bitwise encryption (256-bit refers to the number of bits in the length of the key. More bits mean more possible key combinations, thus making the key harder to brute-force and less likely to have collisions). SHA-256 also uses Symmetrical\* encryption keys. This will be fine, as Artemis Financial will be the only party accessing these encrypted files. The SHA-256 algorithm also makes efficient use of Java’s random number generation to ensure that each encrypted file is as secure as possible. Using random numbers allows for the cipher to securely create a non-reversible checksum that still verifies the authenticity of the file/message.

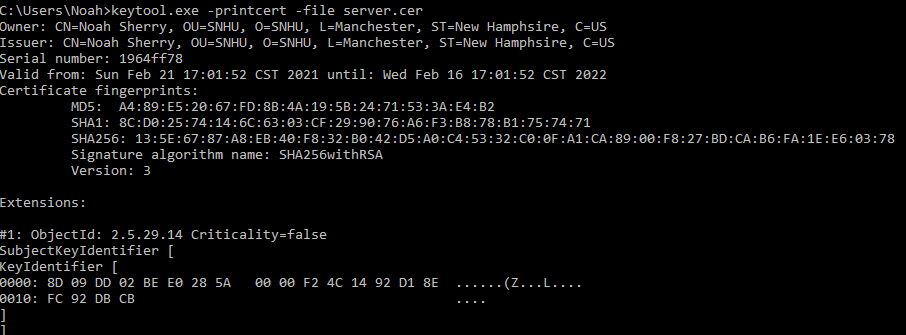
The hash function to verify files will use the SHA-256 cipher to create a checksum signature of the provided message.

\*Symmetrical encryption keys means that the algorithm uses the same key to encode/decode the data. Asymmetrical keys use a combination of a public key to encode data and a private key to decode the data.

## 2. Certificate Generation

Generate appropriate self-signed certificates using the Java Keytool, which is used through the command line.

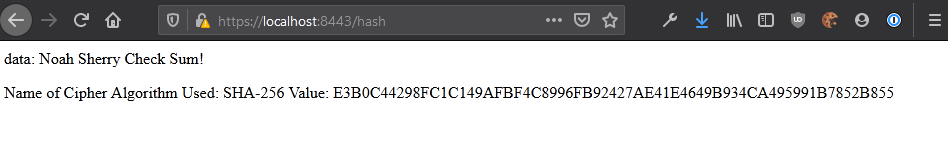
* To demonstrate that the keys were effectively generated, export your certificates (CER file) and submit a screenshot of the CER file below.



## 3. Deploy Cipher

Refactor the code and use security libraries to deploy and implement the encryption algorithm cipher to the software application. Verify this additional functionality with a checksum.

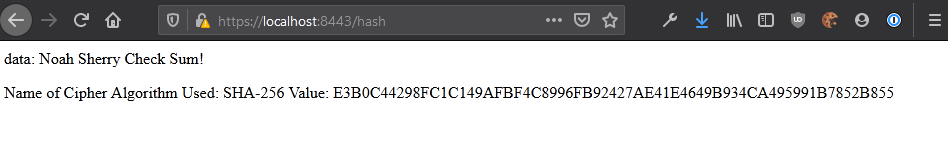
* Insert a screenshot below of the checksum verification. The screenshot must show your name and a unique data string that has been created.



## 4. Secure Communications

Refactor the code to convert HTTP to the HTTPS protocol. Compile and run the refactored code to verify secure communication by typing **https://localhost:8443/hash** in a new browser window to demonstrate that the secure communication works successfully.

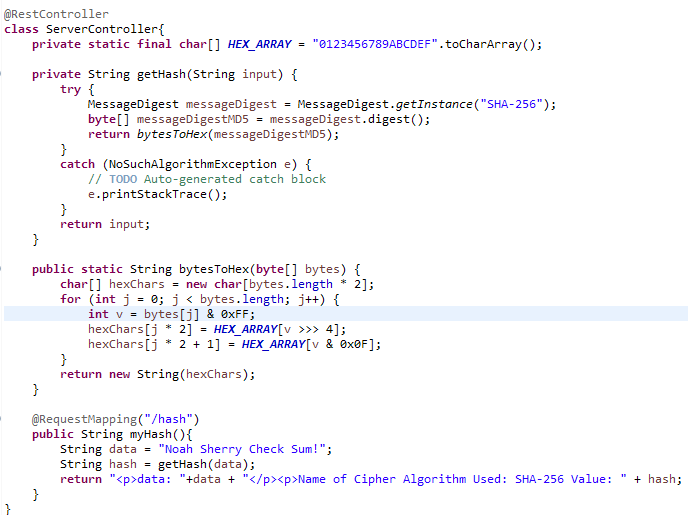
* Insert a screenshot below of the web browser that shows a secure webpage.

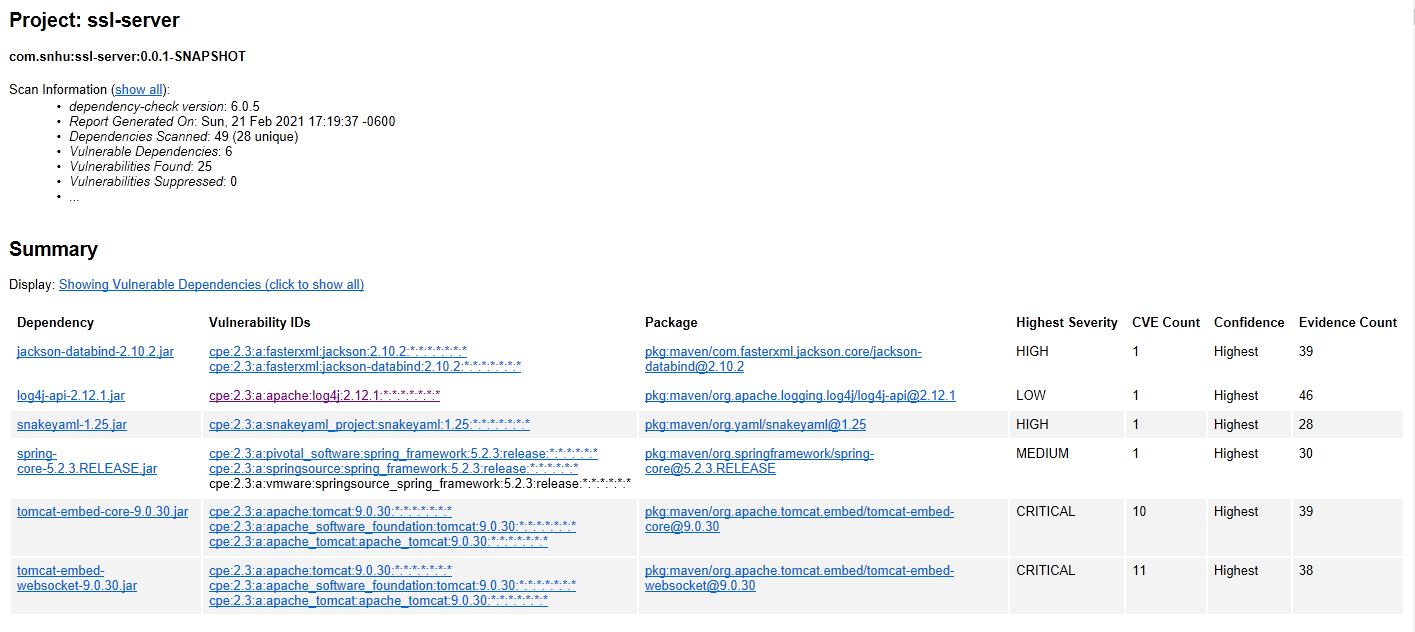


## 5. Secondary Testing

Complete a secondary static testing of the refactored code using the dependency check tool to ensure code complies with software security enhancements. You only need to focus on the code you have added as part of the refactoring. Complete the dependency check and review the output to ensure you did not introduce additional security vulnerabilities.

* Include the following below:
  + A screenshot of the refactored code executed without errors
  + A screenshot of the dependency check report

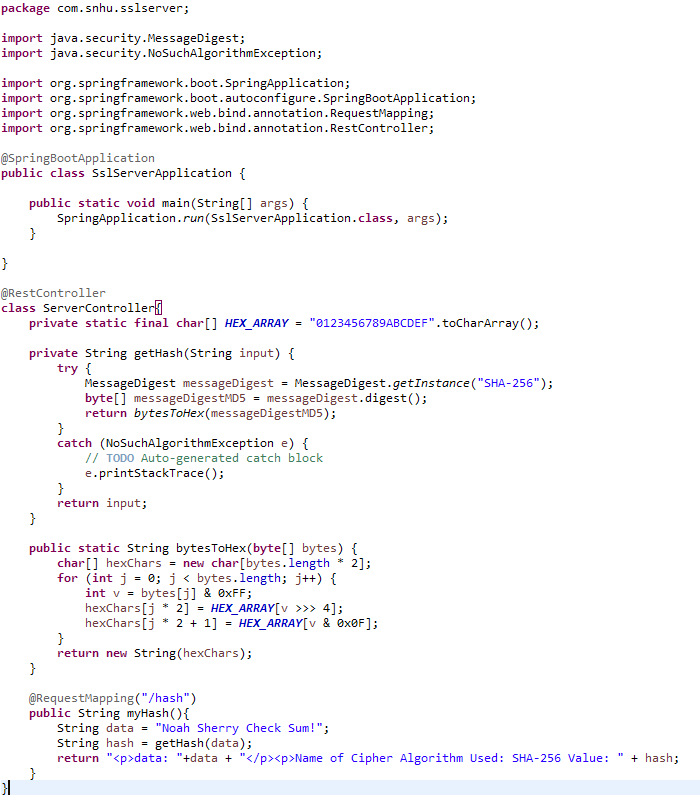




## 6. Functional Testing

Identify syntactical, logical, and security vulnerabilities for the software application by manually reviewing code.

* Complete this functional testing and include a screenshot below of the refactored code executed without errors.



## 7. Summary

Discuss how the code has been refactored and how it complies with security testing protocols. Be sure to address the following:

* Refer to the Vulnerability Assessment Process Flow Diagram and highlight the areas of security that you addressed by refactoring the code.
* Discuss your process for adding layers of security to the software application and the value that security adds to the company’s overall wellbeing.
* Point out best practices for maintaining the current security of the software application to your customer.

In my code refactoring, I’ve added a secure RestController to the application to serve as the secure controller for the hash RESTful endpoint. This ServerController class addresses the secure coding concern in the Vulnerability Assessment Diagram and fulfills that concern. I’ve chosen to use SHA-256 as the hashing cipher for this function, and the code is very minimal so as to reduce the potential attack surface.

I’ve also updated the version of the Maven Dependency check version from 5.3.0 to 6.0.5, so that the static dependency checking is as accurate and up to date as possible.

To maintain the current security of the application, I’d recommend that the dependency checker is run at least once or twice per month in order to check for new vulnerabilities that have been discovered so that they may be fixed. Additionally, updating the plugins in the pom.xml configuration file is necessary every so often to ensure that the plugins remain up to date.