****

# 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** | **4/17/2021** | **John Elbogen** |  |

## 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

John Elbogen

## 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.

We are using an RSA SHA-256 encryption is because, there is a near impossible chance of collision. Also, our goal is to create a file verification step when transferring data. This will allow access for users with the private key and prevent verification of those who do not have the private key.

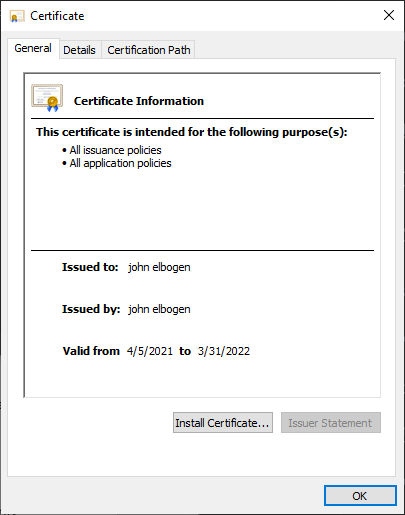
The SHA-256 hashing function will output a value that is 256 bits long, regardless of input. We will use an asymmetric key because we will have a public key for hashing and private key for access to use the web application decryption and to secure information transfer.

Historically, no collision has been found with SHA-256. Consider a commonly used example, bitcoin. The rate bitcoin was computing was 300 quadrillion hashes per second. At this rate it would still take 3.6x10^13 years to find a collision. The amount of time is so absurdly large that the chance of a collision through brute-force would be nearly impossible with current technology.

## 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.

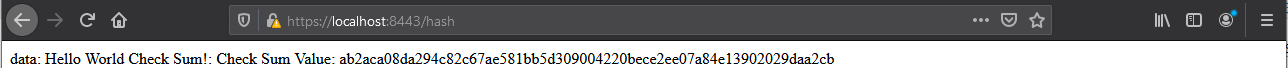
* Insert a screenshot below of the checksum verification. The screenshot must show your name and a unique data string that has been created.



## 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.

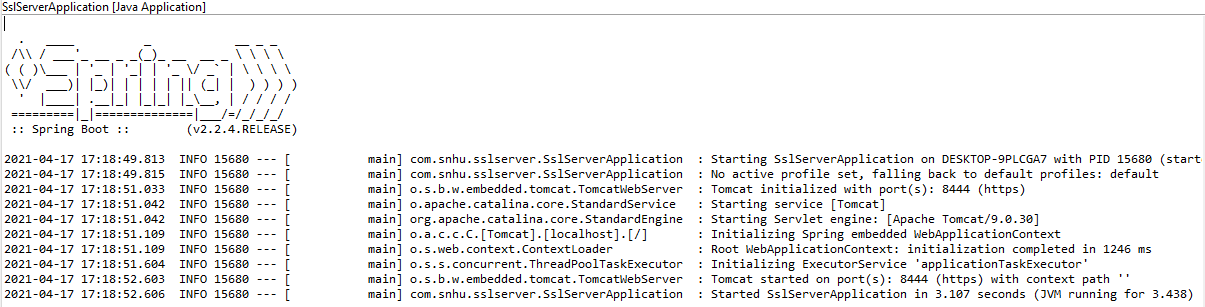


## 5. Secondary Testing

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

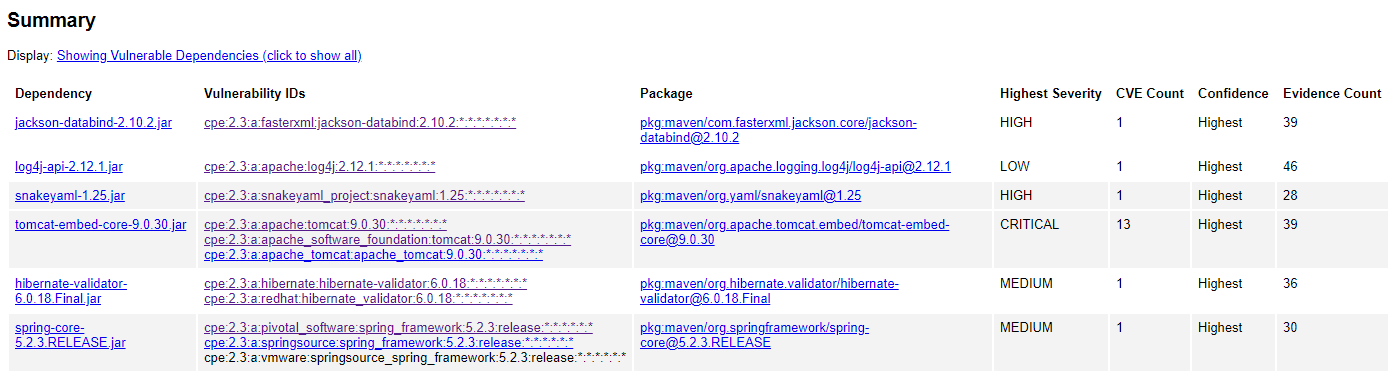
**Running no errors:**



**Dependency report, pre-refactoring:**



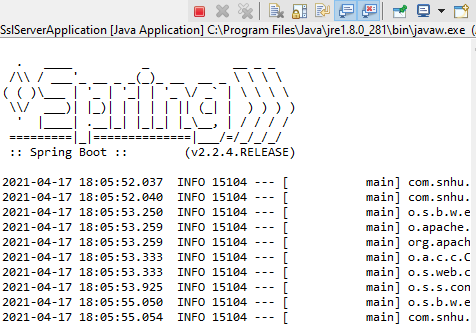
**Dependency report, post-refactoring:**



## 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.



The code meets security standards.

## 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.

According to the vulnerability assessment process flow diagram, I have added *cryptography, code error handling*, *code quality*, and *encapsulation* during my refactoring of the code. Let us examine how each layer has been added. Cryptography was added using, an RSA SHA-256 algorithm. I first generated a self-signed certification, to use for web application access. This utilized the keystore.exe function to generate a private and public key. This private key is kept secured and can be distributed to trusted sources to authenticate their end ensuring data is being sent to a trusted source, by Artemis Financial. SHA-256 avoids collisions, preventing a hacker from using brute force to find an input which produces a matching hash value, comprising the algorithm.

To implement code error handling, I utilized exceptions to prevent errors from occurring, like lacking the correct file locations, as runtime errors can lead to security vulnerabilities. Code quality is assured, by using maven dependency check tool, which looks at the refactored code, and can identify dependency issues. Often hacking can occur through outside dependencies. In addition to the maven static check, a manual revision of code for security vulnerabilities was executed to assure secure coding practices. Finally, encapsulation was used by using a server controller class, which allows the code to specifically be written to determine whether operations should be permitted or denied to accessors.

Finally, to maintain security of the software application, dependencies should be regularly updated, and checked, to stay up to date with security fixes. In addition, the private key must be kept safe and secure. Attached with this document are the code files including, the refactored code.