****

# 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** | **10/15/2021** | **Jorge Sierra** | **Final Version** |

## 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

Jorge Sierra Alzaga

## 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 will be dealing with privileged information regarding its customers, such as Personal Information (name, address, social security number, etc.), Financial Information (bank accounts, investments, assets), and handling financial transactions in general. When dealing with this kind of data, security is a top priority, thus I recommend using AES as the encryption algorithm cipher to use.

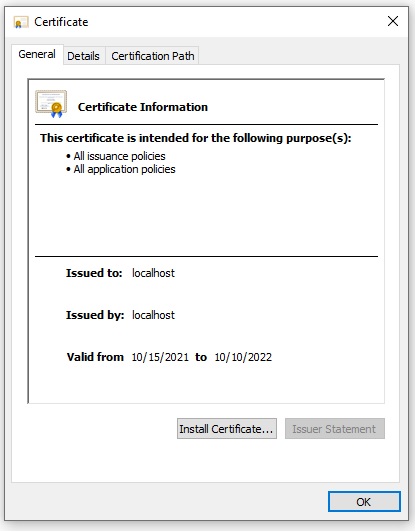
Advanced Encryption Standard (AES) is the strongest encryption algorithm currently in use; it was published in 1998, it uses a block size of 128 bits, and it has key sizes of 128, 192 and 256 bits. The block size means that it will take a 128 bits input, modify it, and produce a 128 bits output. The key of varying size just gives us options, the larger the key, the longer time it would take to “break” it and the more transformation rounds would be applied on the original input. The key is also symmetrical, which means the sender and the receiver share the same private key to decipher the encryption. From all the available algorithms, AES is the best one, as it is fast and secure, working so well that it is the only publicly accessible cipher approved by the NSA for use in top secret information.

The hash algorithm SHA – 256 is one of the most secures there is, it has little chance of collision, with it requiring evaluations. The algorithm is part of the SHA-2 cryptographic hash functions designed by the NSA.

## 2. Certificate Generation

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

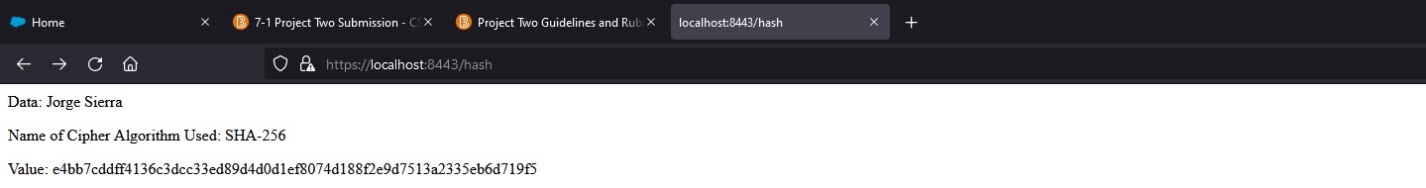
* To demonstrate that the keys were effectively generated, export your certificates (CER file) and submit a screenshot of the CER file below.



## 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.

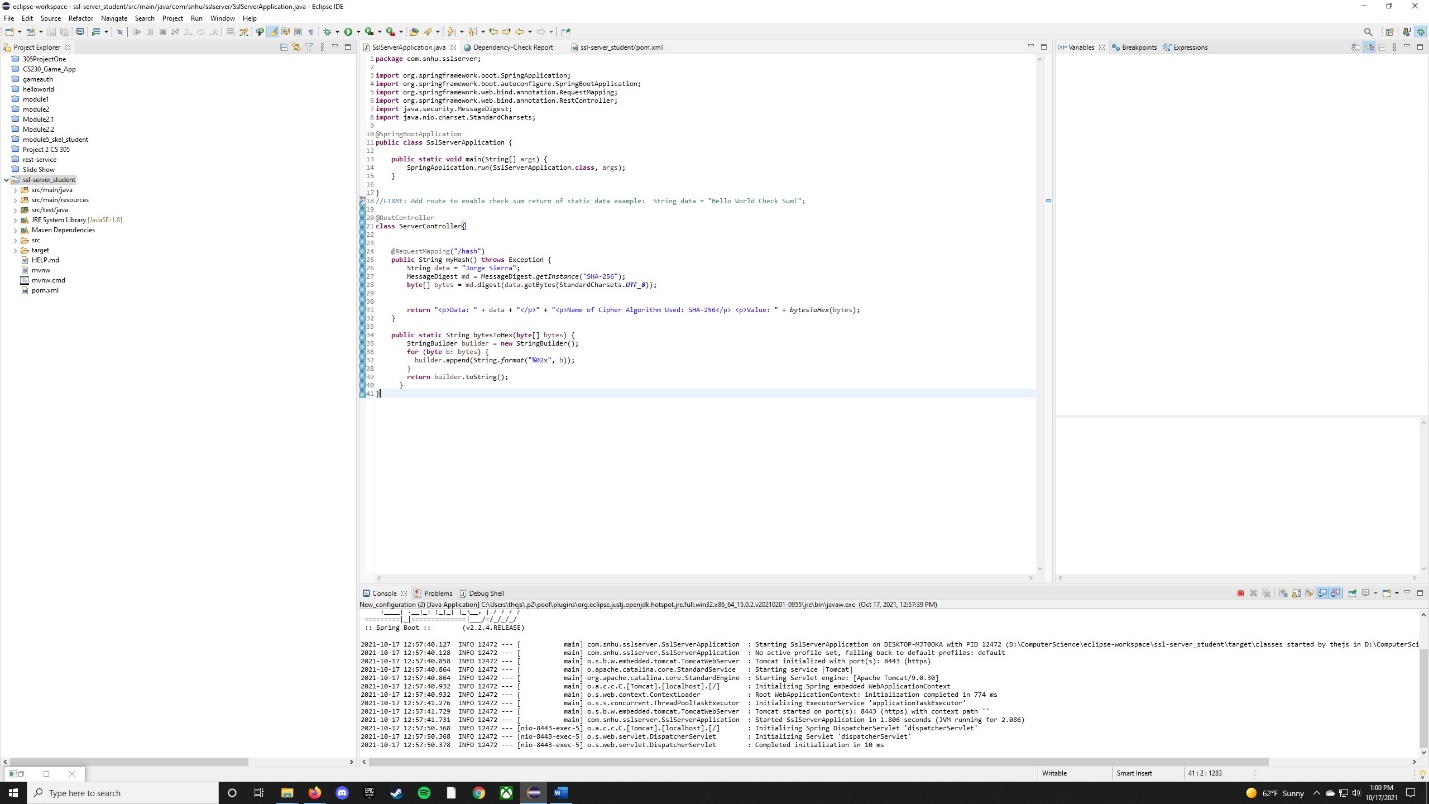
A screenshot of a computer

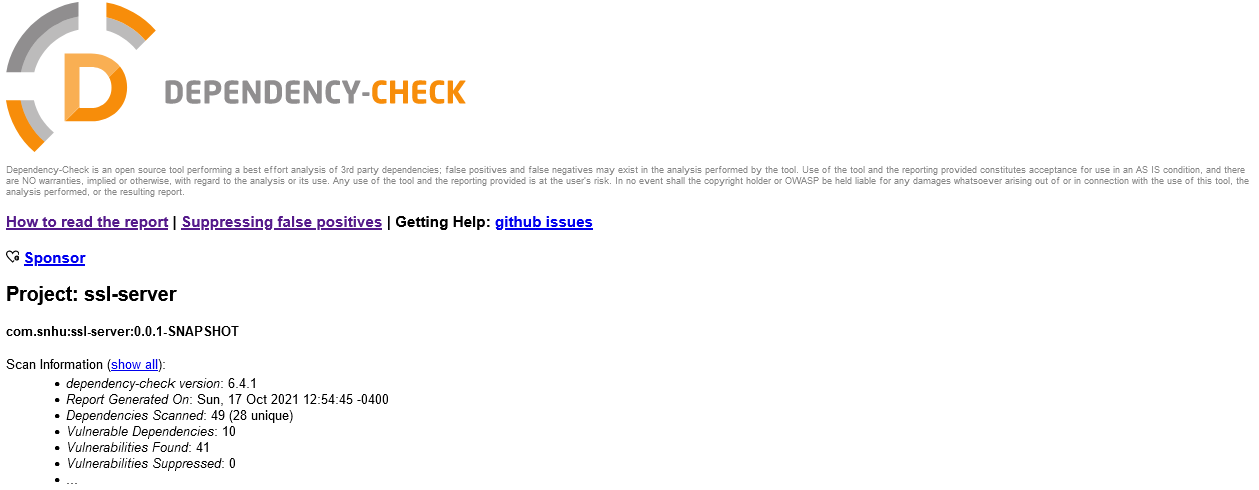
Description automatically generated with medium confidence

## 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





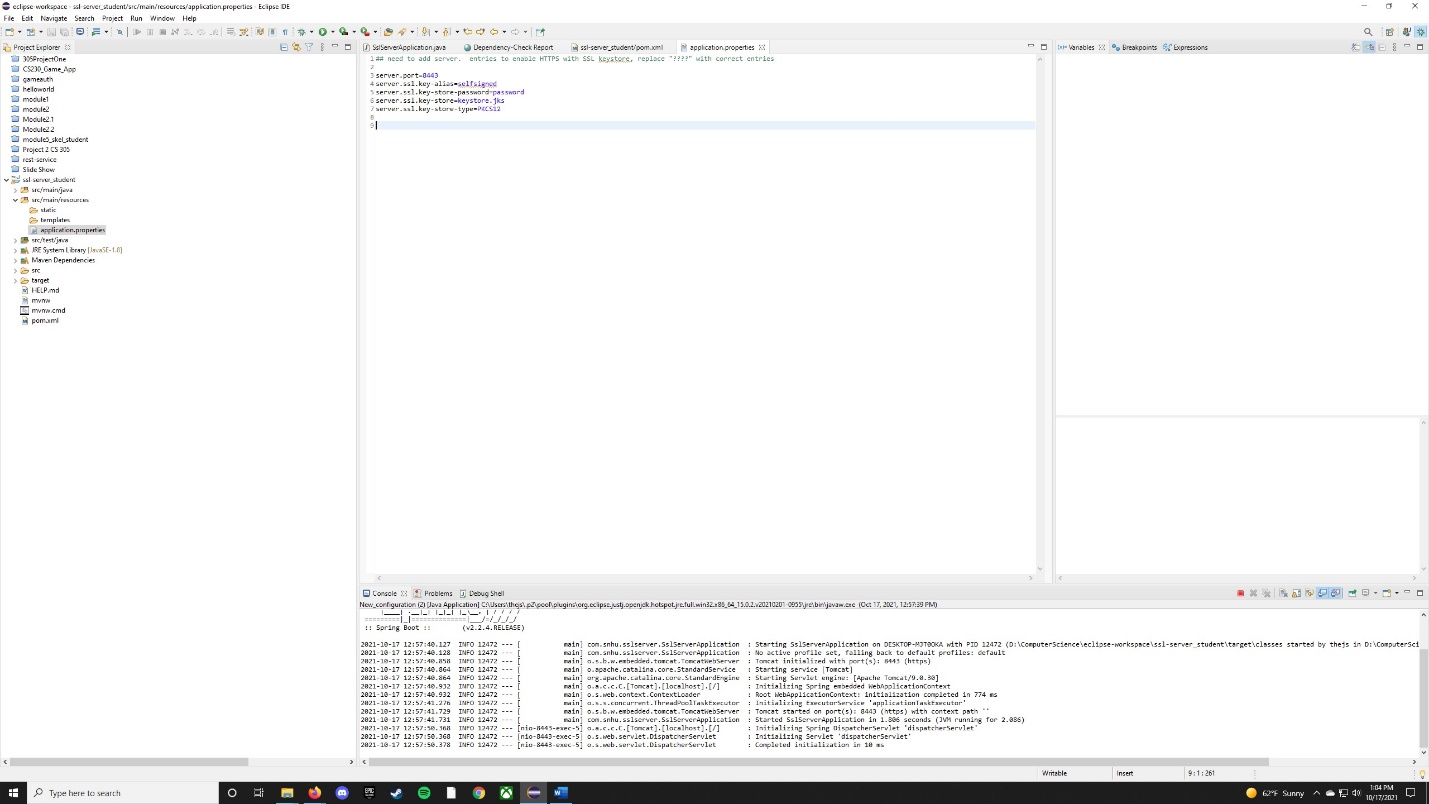
## 6. Functional Testing

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.

Graphical user interface, application

Description automatically generated



The main vulnerability found is having the password for our Security Certificate showing in plain text in our code, which could be solved by encrypting the file or not making it accessible to unauthorized personnel.

## 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.

The code implemented in this application targets different areas of security: Client/Server and APIs were addressed by implementing HTTPS security and transmitting the information between the server and the client, even though in this case there was no response from the user. Cryptography was addressed by implementing a cipher algorithm for the checksum verification. Code Error was addressed by handling exceptions inside the myHash function.

In order to add different layers of security to the application, several steps were taken, the first of which consisted of generating a Certificate with Java Keytool and adding it to the trusted root certificates. This certificate was used to implement a HTTPS connection through the use of SSL. The next layer of security was implementing a hashing algorithm using a cipher. This algorithm takes information in and returns it encrypted, which allows to secure critical information by not displaying it in plain text. The next step was to ensure no vulnerabilities were added to the application when implementing our code. The last step would be to solve any vulnerabilities that are found in the Maven Dependency Check.

Some of the best practices to maintain the security in the application is to restrict the access and encrypt critical files such as application.properties which sets up the HTTPS connection for the application. Another great practice is to keep all the dependencies up to date as it would reduce the chance of having vulnerabilities.