****

# 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** | **06/13/2021** | **Kyle Gaskill** |  |

## Client



## Instructions

## Developer

Kyle Gaskill

## 1. Algorithm Cipher

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

I utilized a Sha-256 message digest cipher to create a hash from the certificate key. By utilizing random number data, they can be harder to decrypt without the proper keys. In addition, SHA-256 is one of the strongest hash encryptions with its 32-byte signature. Random numbers is safer to use than symmetric keys. Symmetric keys use only a single key for encrypting and decrypting. Which means once this key has been compromised, so is your data.

Non-symmetric keys are using a public and private key for the encryption and decryption. Since the public and private keys are not identical, cracking one does not compromise your data as you still need the second key.

Keys being generated from Random Numbers are more difficult to crack as there is not a pattern to their encryption. Cryptography via encryption algorithms have paved the way for more secure lines of communication based on the level of encryption being more difficult to break. It started with a single key being used to encrypt and decrypt messages being sent but has further progressed by the computing power of machine-learning algorithms or stronger bit level hash being generated. With the introduction to asymmetric algorithms requiring two keys further complicates the forced decryption. This ultimately keeps data and messages more private. The turning point has been the improved of Sha hash functions and the unquantifiable approach with random numbers giving further defensive in cipher algorithms.

## 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. *(****please note, keystore.jks has been changed to pleasework.jks)***

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

## 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.

Text

Description automatically generated

## 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.

Graphical user interface, text, application, email

Description automatically generated

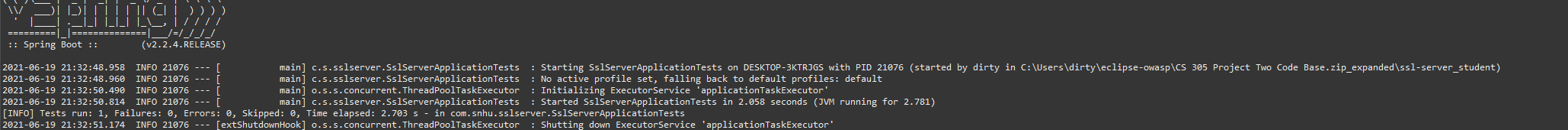
## 5. Secondary Testing

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. A screenshot of the refactored code executed without errors.

Text

Description automatically generated Graphical user interface, text, application, email

Description automatically generated



Text

Description automatically generated

Text

Description automatically generated

* + A screenshot of the dependency check report

Graphical user interface, text, application, email

Description automatically generated

## 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.

Text

Description automatically generated

## 7. Summary

I initially started looking at vulnerability dependencies that had nothing to do with the core function of my program to start eliminating the false positives. Some vulnerabilities I immediately noticed were the ones pertaining to SMTP mail vulnerabilities. From there I went on to find vulnerabilities that were patched out by versions unsupported by the program as there is no workaround besides changing versions. I first created a keystore that would be utilized to give an extra layer of protection and change the http protocol to https. I imported the self-signed certificate that was generated and infused that with a SHA-256 hash function in order to create a two-step checksum. To continue to improve security of the software, it is best to handle all handshake interactions view security certificates in order to prevent unwanted access. It is also best to utilize the principle of least privilege by preventing certificates from being generated from unwanted users that could be bundled into the software. You should continue to monitor the dependency vulnerability report after each code change for possible changes in current known vulnerabilities so you may patch them out or find a proper work around until a fix can be created.