****

# 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** | **02/20/2022** | **Jon Curtis** | **Secure Comms, and Code Refactor** |

## 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

Jon Curtis

## 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.

I have gone with SHA-256 encryption as the algorithm, SHA stands for “Secure Hash Algorithm 2” and was developed by the National Security Agency and is currently required by US government for is agencies. It is currently the most secure hashing algorithm on the market. The input to the hash function is unimportant in concerns of length but its output is always of fixed length. Hashing is irreversible and a cipher like SHA-256 does not make duplicate hashes so it is perfect for verification. Bit levels is the number of bits, in AES-256 it means it would take 2^256 maximum number of attempts to break, but SHA-256 is the length it outputs not a bit level. Hashes are one way so the are not measured in bit levels since there is no undoing them. Keys usually use random number, and character generators for the fact that if a simple word is used, the password is susceptible to brute force attacks that use a simple dictionary program. Random generation insures a more secure key. Symmetric use the same key to encrypt and unencrypt, non-symmetric or asymmetric keys create a “public key” produced from the multiplication of two prime numbers.

Encryption is an ever-evolving field since if no loopholes are found in an algorithm, breaking the encryption becomes based on processing speed, and luck. Brute force attacks take time based on the amount of processing power available. So, it is how quick you can iterate through the guesses. DES was 56 or 80 bit compared to AES minimum 128, as processing power went up DES became obsolete because it became able to break easily. As advancement happen so must cipher algorithms evolve. With quantum computing on the verge of becoming more common place ciphers like RSA may become obsolete as well.

## 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.

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.

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

Graphical user interface, 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

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.

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

Graphical user interface, application

Description automatically generated

Pre-Refractor

Graphical user interface, text, application, email

Description automatically generated

Mid-Suppression

Graphical user interface, text, application, email

Description automatically generated

Post-Refractor

Graphical user interface, text, application, email

Description automatically generated

Links:

<https://missingcurtis.github.io/Software-Security/Project_2/P2-Pre-dependency-check-report.html>

<https://missingcurtis.github.io/Software-Security/Project_2/P2-Mid-dependency-check-report.html>

<https://missingcurtis.github.io/Software-Security/Project_2/P2-Post-dependency-check-report.html>

## 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, text, application

Description automatically generated

## 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.

I first started out by integrating the most up to date Owasp dependency check and running a static check. Since I am only worried about the could I am refactoring I suppressed the current “Vulnerabilities” and reran the check to make sure I was starting at zero. After analyzing the “Vulnerability Assessment Process Flow” I then refactored the code to use SHA-256 to hash data to satisfy the “Cryptography” section and return a “checksum” value making the data verifiable. I also encapsulated the data for more security to satisfy “Encapsulation”. For “Code Eror” I implemented a “try/catch” for the refactored code. I would suggest revisiting the suppressed “Vulnerabilities” and updating or refactoring where needed for a mor secure application. Then running weekly dependency checks to verify the app is still secure.