****

# CS 305 Project Two

**Practices for Secure Software Report**

Table of Contents

[\_Toc33111301](#_Toc33111301)

[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** | **6/17/2022** | **Mathew Denison** |  |

## 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

Mathew Denison

## 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.

The cipher that I have chosen for the organization is SHA-256. The reason for this is that there is no chance of collisions and that the cipher has never been broken. It is the most secure encryption that I was able to find. Since the larger the bit hashes the less chance of a collision this provides a high level of security. As the number of hashes increases the less the likelihood of 2 values parsing the same number. We will also be using random numbers to generate the keys along with symmetric keys as there will be a single cert that will be generated to decrypt. The key will be generated in the step below. With most of the other encryption methods being hacked, the SHA-256 cipher should provide lasting security for now and through the future.

## 2. Certificate Generation

Generate appropriate self-signed certificates using the Java Key tool, 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.

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

Text

Description automatically generated

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

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 after being refactored incorporated a few different fixes that enabled some security vulnerability testing that made it much more secure. After running the dependency check and determining some changes that needed to be implemented, I did the following. Signed our own certificates with the java tool to validate the certs of the site. While this did result in an insecure connection as we are not “public cert guarantors” we did enable a symmetric key that was not changed as per the checksum message on the site. We also utilized SHA-256-bit encryption to encrypt the key which does not at this time have any way to be broken. There were some vulnerabilities that were shown in the report; however, it was determined that they were likely false positives. The only thing that could have made this more secure is if we purchased our own validated domain that does certificate generation/validation on its side.

The process of adding layers to security by unit testing, manual review, and static testing are that they provide further guarantees that the code is safe. If a hacker can break into the server and take the generated key, then this would be extremely bad. We could lose lots of data, however not all of it as we did implement one way validation. One way to guarantee that the code is not vulnerable is to run it through dependency tests regularly. The public government site that tests for all public instances of vulnerability is a powerful tool that we should implement nearly every day. Also, this means that any code added to the codebase should also be run through static testing to confirm that it is also not vulnerable as well. Before being published and merged in, the code should be tested, and changes implemented. It is also important that when false positives are found that they are suppressed from the test so as to not set off false alarms. This can be done by manual review of the code to confirm that it is valid, and no security threats are active.