****

# 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** | **2/19/21** | **Patrick Hensley** |  |

## 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

[insert name here]

## 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.

An encryption algorithm cypher utilizes keys for validation. They come in a variety of different sized encryptions, for this project we used a 256-bit key. A 256-bit key is exponentially harder to attack and decode, it is not just double the effort of a 128-bit key.

When using keys, there are 2 options. Symmetric and non-symmetric. A symmetric key requires the host and the client to have a key. This mean, the client will need to receive a key and use it to gain access to the server or data. If that key is lost or stolen, the data will be at risk.

Non-symmetric keys use a public key and private key. The most secure way to do this is to have the private key residing on a known and trusted third party server. The client will have a public key and when it wants to access the data, it will send that public key for the third party to verify it against the private key. The private key does not need to reside on a third-party server. The public and private keys can reside on the same server, it is just slightly less secure.

Encryption algorithms have changed over time due to the complexity needed. Now that computing power is a lot stronger and quicker, the keys can be larger and not take up too much time. Before processing speeds were as fast as they are today, 128-bit keys were the best security to processing time and power needed ratio. As attackers get better and computers get better, the upgrade to 256-bit keys makes them more secure.

## 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, application, email

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, application

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, text, application

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.

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.

In this refactored code, I used cryptography, client/server, and code quality from the Vulnerability Assessment Process Flow Diagram. By creating a certificate from the keytool, I was able to encrypt the data sent. I also used a checksum to validate that the program was secure.

By adding layers of security, you add more security to the program. An attacker may know how to get through one layer but adding multiple layers will protect your data further. Also, by staying current with known vulnerabilities and modifying your code or tool versions, you can secure your code even more.

Best practice for maintaining current security is to check for updated known vulnerabilities. This can be done with different tools. We utilized the dependency check. This can lead to false positives and that needs to be addressed as well. They can be parsed out if you know what they are. Another good practice is to always have security focused on by every developer. No matter what they are doing, they should be thinking about what they can do to secure their code and what vulnerabilities they may have and how they can mitigate them.