****

# 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** | **June 19, 2022** | **Steven Cruickshank** | **Completed Secure Software Report** |

## 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

Steven Cruickshank

## 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.

There are some good rules of thumb, for instance avoiding cipher suites which contain ANON, NULL, and EXPORT. We also want to stick with bulk ciphers with large key sizes (ideally > 128 bits). In deciding the best cipher for this job, I chose the SHA 256 algorithm cipher. The ability of a human to crack a key using SHA 256 encryption is near impossible. Even using SHA 128 would provide quite a bit of security. It would require the advancement, and mass adoption of quantum computers to be able to brute force a key run through SHA-256. There are some minor limitations when using SHA-256. For instance: the message and digest length needs to be under 264 bits in length. 512 bits will get shoved through a compression function, which applies 64 rounds of data scrambling, and bit shifting operations. Once these rounds of encryption are complete, the digest will output at a length of 256 bits.

Random number generation is an integral part of many aspects of programming and computer science. Random number generation in regard to cryptography is called Cryptographic Pseudorandom Number Generation (CPRNG). When dealing with cryptography, most encryption algorithms get seeded with a random number via CPRNG. Oftentimes this algorithm is constantly reseeded as to increase the level of security. I would suggest using symmetric cryptography for Global Rain due to its relative ease in rolling out.

Symmetric cryptography utilizes a single private key to cipher and decipher data accordingly. Asymmetric cryptography utilizes two separate keys: a public key, and a private key. These two keys do not match, and the private key cannot be deciphered using the public key. Asymmetric cryptography also utilizes hash functions as a means of verification. These hash functions will make note if any data has been tampered with. Because of this, it has boosted security compared to a single-key solution.

## 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.

Graphical user interface

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, application

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, website

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

A screenshot of a computer

Description automatically generated with medium confidence

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.

A screenshot of a computer

Description automatically generated with medium confidence

## 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 will outline below the areas of the Vulnerability Assessment Process flowchart I have improved by refactoring the base code.

**Client/Server**: This application now displays a valid checksum using secure HTTPS web protocol

**Cryptography**: This application now uses a SHA-256 encryption algorithm to secure data between the client and server.

**APIs**: In running dependency checks, we can ensure that there are no dependency disparities between API communications.

I touched on it above, but the security that SHA-256 encryption algorithms bring is near unmatched. It is used everywhere from large scale corporations to the US government to secure data. In running dependency checks, we can bring this application up the par. Many exploits happen due to mismatched dependencies. Luckily, they are easily identifiable, and mostly easy to fix. Using an HTTPS protocol is a given for this day and age. The implementation of a secure HTTPS connection is of immense value to the client.

I would recommend to the client that they routinely run dependency checks. New cracks and exploits are being introduced every day. Many of those exploits rely on out-of-date APIs that become a backdoor into an application. By keeping up on these dependency fixes, the client can take a proactive approach to security before there is a problem.