****

# 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** | **12/7/2021** | **Shane Griffith** |  |

## 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

Shane Griffith

## 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 SHA – 256 encryption algorithm will be used to secure this project. This provides a 256-bit encryption against collisions to prevent attacks that can lead to data breaches form those who would be using the service. Until quantum computing because realistic in reality and not just probable theoretically, the time to crack a single 256-bit cipher is 3.31 x 1056 years even for today’s most powerful supercomputers (ClickSSL, 2012). Readable data getting turned into non readable data through encryption is known as “hashing” this is a one way process and can only be undone with the key. (*Hash Function in Cryptography: How Does It Work?*, 2020). The hash function that is being used in this program takes the input and essentially outputs a string of random characters 64-bits long.

Symmetric cryptography uses single keys to encrypt and decrypt data. Whereas non-symmetric cryptography uses two keys. One to encrypt the data and another to decrypt the data. In the symmetric cryptography model, the key must be kept safe. If it is lost, so it the data since the decryption of the data is not possible without the single key that is generated. In the non-symmetric encryption model, the key that is used to encrypt the data is in the public view while the second key is kept private and used to de-crypt the data. Both of these encryption types use randomly generated numbers for an immensely unlikely case for the keys to overlap with each other.

Encryption is a technique that has been utilized for encryption data transmission going back as far as data transmissions have been possible. An early, very effective example of this would be “Enigma”. Enigma is an encryption type that the Axis forces used in World War II to encrypt messages being sent over radio waves. IBM also created another encryption type called “Lucifer” that was designed for civilian use (*The Story of Cryptography : 20th Century Cryptography*, n.d.).

## 

## 2. Certificate Generation

Generate appropriate self-signed certificates using the Java Keytool, which is used through the command line.

.

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

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.

The webpage here is showing that the connection is not secure. However, I believe this is because I’m using a certificate that is “self-signed”. After looking into the details of the certificate itself, you can see that it is in fact using the SHA256 encryption type as shown below.

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, 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.

This screenshot shows the code executes and builds successfully.

Text

Description automatically generated

Here is the dependency check from the program that was ran before the code was refactored.

Graphical user interface, text, application, email

Description automatically generated

This is the result of the dependency check after the code has been refactored.

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 computer screen capture

  Description automatically generated with low confidence

Above is a screenshot showing that the code executed without having any errors. The application.properties file also is valid and without error as shown below.

A screenshot of a computer

Description automatically generated

The last check that was made is in the pom.xml file. I updated the version to 6.1.1 from 5.3.0. As you can see it also executes without error.

Graphical user interface, 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.

Cryptography, code quality, and server/client relationships were all addressed and managed throughout this project. Diving in a bit deeper here, a 256-bit hash algorithm was specifically implemented in the certificate that the client used to connect to the server using a secure TLS connection in the presented scenario. Refactoring the code also help tremendously with suppressing the original 72 vulnerabilities down to 0. The final point here being the quality of the code which was reviewed carefully and corrected of all errors.

One of the layers of security that was implemented in the program was the SSL certificate. This helps the client device ensure that the current connection is secure and encrypted. This ensures that the client is connected to the server that it believes it is and visa versa. Another layer of security that was added to this program was the type of encryption that was used. The SHA256 encryption type is very secure and is an industry standard due to how robust the algorithm is. Knowing what we know about encryption, secure the connection alone is not enough to assure security. This is why the SHA-256 algorithm was chosen to encrypt the data as well. Forcing the program to utilize HTTPS is another layer of security that was added to the program. This forces the user to connect to the server through a HTTPS connection opposed to HTTP which help mitigate attacks such as Man-In-The-Middle.

Everything previously in this section is what I would call in the area of “Best-Practice”. Implementing things like frequent dependency checks, input validation, and frequent code reviews would also fall under this category.

**References**

*ClickSSL. (2012, January 30). 128-Bit SSL Encryption Vs 256-Bit SSL Encryption. ClickSSL Blog - Information about SSL Certificates & Infosec. https://www.clickssl.net/blog/128-bit-ssl-encryption-vs-256-bit-ssl-encryption*

*Hash Function in Cryptography: How Does It Work? (2020, December 4). InfoSec Insights. https://sectigostore.com/blog/hash-function-in-cryptography-how-does-it-work/*

*The Story of Cryptography : 20th Century Cryptography. (n.d.). Ghostvolt.com. Retrieved December 13, 2021, from https://ghostvolt.com/articles/cryptography\_20th\_centuary.html#:~:text=The%20RSA%20algorithm%20for%20asy*

*‌*

*‌*

‌

‌