****

# 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** | **12/11/2021** | **Russell Pallas** |  |

## 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

Russell Pallas

## 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.

We will be using the SHA-256 ciphering method for this program. The SHA-256 method is an industry standard of encryption in many fields including the banking industry. This will create a unique 256bit 32byte hash value for the given string. The hash appears to be random and asymmetric and in a sense it is just that. However, there is an implicit conversion that can be created given the cipher. This cipher has a very low collision rate <0.1% currently. This means that there is very little chance that any outside attackers will be able to intercept and decode these messages. Especially considering the fact that a conversion from http to https was made given our self signed certificates created using the keytool. Mistakes have been made in the past when it comes to thinking that encryption is perfect, however, lessons were learned from those mistakes. Encryption has come a long way in a relatively short amount of time. Most encryption nowadays is pretty solid. There is very little chance that given the layers of security that we have in the program that there will be unauthorized access here. This program also doesn’t even give the chance for a SQL injection attack. The completed codebase is included as an attachment in zipped form.

## 2. Certificate Generation

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.

“ab2aca08da294c82c67ae581bb5d309004220bece2ee07a84e13902029daa2cb”

## 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.

Text

Description automatically generated

## 5. Secondary Testing

A screenshot of a computer

Description automatically generated

No new vulnerabilities were introduced. Refactoring the codebase did not produce any new errors, there are no error icons or notifications present.

## 6. Functional Testing

No syntactical, logical, or security vulnerabilities present. All tests passed with 0 failures, 0 errors, and nothing skipped.

A screenshot of a computer

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:

All areas of the Vulnerability Assessment Process Flow Diagram were considered, however, the main area that we focused on was client/server communications. We ensured that proper conversion was made from http to https using the self signed certificate that we generated through the keytool. This was just one tool that we used to ensure safe communications.

Another way that we ensured safe communications was through the analysis of dependencies within the project. We used the OWASP plugin to examine the dependencies that the program used and determined which of the dependencies were flagged that needed attention. We determined which of the dependencies were false positives, that is they appear as though they are vulnerable, however, they are not. We have the option to suppress those false positives should we deem it necessary. This is standard practice.

All of these elements by themselves seem as though they are not making much of a difference. However, when combined together and layered, they make for a more secure program. That is not to say that it is impenetrable, however, it creates more resistance for outside attackers. More security means that customers will feel more confident using the company’s program for their banking needs. We recommend that the company continues to run the OWASP checks to ensure that the current dependencies haven’t become vulnerable. Should they determine that they have become vulnerable, they should update the dependencies to the most current version as dependency vulnerabilities tend to be patched up through regular dependency updates.