****

# 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** | **02/25/2022** | **Alice Norris** |  |

## 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

Alice Norris

## 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.

Two encryption algorithm ciphers were chosen, namely, TLS\_ECDHE\_ECDSA\_WITH\_AES\_256\_CBC\_SHA and TLS\_ECDHE\_RSA\_WITH\_AES\_256\_CBC\_SHA. These were chosen because they use ECDHE and DHE for key exchanges, as they provide forward secrecy. Even if the keys were compromised, previously sent messages could not be decrypted in the case of an attack recording messages in the hopes of decrypting them later. The AES cipher uses a key size of 256 bits, and in the CBC mode each block processed is modified by the block that comes before it, mutating each, preventing identical blocks of repeated data from having the same cipher. ECDHE is a mechanism by which keys can be exchanged over an insecure channel. The CBC mode specified also uses a random number for the data blocks of 128 bits each, padding as necessary, further obfuscating the data. ECDSA is used for authentication in the first, and RSA is used in the second. AES-256 is used with a CBC mode to encrypt the data, and SHA is used as the hashing algorithm. AES is symmetrical, meaning that the same key is used on both sides of the transaction: both encrypting and decrypting.

The first widely adopted block cipher akin to the ones used today, the Data Encryption Standard (DES), was developed in the 1970s at IBM. However, as computational power grew due to Moore’s Law, it became easier to discover patterns and weaknesses in early block ciphers. However, it is easier to encrypt data with known keys than it is to try to discover and break patterns in encrypted data. This resulted in a sort of “arms race”, involving increasingly complex and clever patterns being used in ciphers, followed by attackers attempting to break them. At a certain point, however, it became feasible to create encryption methods that were nearly immune to brute-force attacks, requiring attacks on stored keys, discovering weaknesses in implementation, and intercepting encrypted communications before they were well-established. For example, AES 256 is still considered secure as brute-force would require attempting every key possible. Since each bit has two possible values, being binary, it would require 2256 attempts to perform an exhaustive search, requiring a prohibitive number of attacks. It is far easier to attempt to compromise keys or perform another sort of attack entirely to break them.

## 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.

A picture containing text, plaque

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.

Shape

Description automatically generated with medium confidence

## 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.

A picture containing text

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.

* Include the following below:
  + A screenshot of the refactored code executed without errors
  + A screenshot of the dependency check report

The code, executed without error using the Maven test goal can be seen successfully built here:

Text

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.

The dependency vulnerability chart returned after all refactoring was complete is as follows:

Graphical user interface, text, application, email

Description automatically generated

The dependency vulnerability report generated by running the codebase in its unaltered state has the same dependencies reported with the same vulnerability count:

Graphical user interface, text, application, email

Description automatically generated

No new vulnerabilities have been introduced by my code.

## 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.

Referring specifically to the vulnerability assessment diagram, I addressed cryptography issues by selecting TLS version 1.2, and choosing an appropriate cipher suite, implementing suites capable of appropriately strong encryption, including forward secrecy. Secure distributed composing was also implemented by securing individual aspects of the system, as the overall security of a system is only as strong as its weakest component. Any file and cryptographical functions that can throw errors related to the environment or input/output issues are handled gracefully. Encapsulation was achieved by segmenting functionalities into appropriate components. The hash endpoint handles security internally, application-wide security configuration is done in its own class by extending automatically configured properties, and options are set as needed in configuration files, including application.properties and in the pom.xml file. These changes can be seen in the code files attached along with the report.

Security layers were added incrementally, first by implementing commonly used best practices, such as using properly configured self-signed certificates, securing the certificates used to authenticate the server, and selecting an appropriate cryptographic suite for TLS to allow encrypted transfer of data. Data that is transferred by the TLS-secured channel are protected from alteration at rest or en-route by providing checksums of data, using the SHA-256 algorithm, which is highly unlikely to be compromised by furnishing manipulated data with a key that collides with the checksum of the original, unaltered data. Common HTTP/2 security protocols are implemented using the Spring framework’s security dependency, which protects against XZZ, CORS, and other spoofing or MITM type attacks.

To achieve security, several dependencies required specifying version numbers in order to maintain compatibility with existing dependencies while also minimizing vulnerabilities. Dependency checks should be run regularly in order to provide fixes to new vulnerabilities as they appear, and all versions should be kept as up-to-date as possible. If proper versioning and code maintenance are done at regular intervals, including automated checks for new vulnerabilities, the code should remain secure.