****

# Practices for Secure Software Report

Table of Contents

[Document Revision History 3](#_Toc102040754)

[Client 3](#_Toc102040755)

[Instructions 3](#_Toc102040756)

[Developer 4](#_Toc102040757)

[1. Algorithm Cipher 4](#_Toc102040758)

[2. Certificate Generation 4](#_Toc102040759)

[3. Deploy Cipher 4](#_Toc102040760)

[4. Secure Communications 4](#_Toc102040761)

[5. Secondary Testing 4](#_Toc102040762)

[6. Functional Testing 4](#_Toc102040763)

[7. Summary 4](#_Toc102040764)

[8. Industry Standard Best Practices 4](#_Toc102040765)

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **[Date]** | **[Your Name]** |  |

## Client



## Instructions

Submit this completed practices for secure software report. Replace the bracketed text with the relevant information. You must document your process for writing secure communications and refactoring code that complies with software security testing protocols.

* Respond to the steps outlined below and include your findings.
* Respond using your own words. You may also choose to include images or supporting materials. If you include them, make certain to insert them in all the relevant locations in the document.
* Refer to the Project Two Guidelines and Rubric for more detailed instructions about each section of the template.

## Developer

William Maguire

## Algorithm Cipher

The best algorithm cipher in this context is AES-256 encryption. A study has shown that this cipher out-performs other encryption algorithms, such as RSA (Fatima et al., 2022). It is a widely accepted and adopted industry standard for data encryption.

AES-256 uses a 13-round block cipher to encrypt data. While the algorithm supports keys at several bit levels (128, 192, and 256), encryption is performed at the 128-bit level. Selecting the highest bit level for our keys is best for future-proofing our data security. This is because higher bit level keys are exponentially harder to brute force.

AES employs a cryptographically secure pseudo-random number generator (CSPRNG) to generate a symmetric key. This symmetric key is then used for both encryption and decryption of the target data (in this case, archive files). Having a single key is appropriate for this context because archive files are not going to be routinely accessed. Managing two different keys (as needed in an asymmetric encryption algorithm) is an unnecessary extra layer of difficulty for seldom-accessed files.

Data encryption outdates modern computing by a considerable margin. The most well-known of these antiquated encryption methods is the Caesar cipher, a basic transposition method, used by Julius Caesar over 2,000 years ago to send messages to his generals. The next method a layperson may be familiar with is the Enigma machine used to encrypt messages during WWII. While breaking this cipher took years of effort, a modern computer could crack it in hours.

Modern encryption methods began with the development of the Data Encryption Standard (DES) by IBM in the 1970’s. DES was widely adopted by government agencies and businesses, but it was later found to have vulnerabilities and was replaced by the AES in the early 2000s. AES has remained industry standard ever since, for good reason, which is why I selected it for this use case.

## Certificate Generation

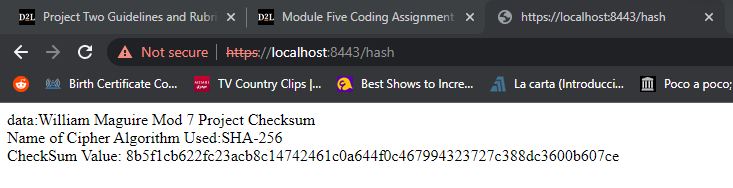
Insert a screenshot below of the CER file.

Text

Description automatically generated

## Deploy Cipher

Insert a screenshot below of the checksum verification.



## Secure Communications

Insert a screenshot below of the web browser that shows a secure webpage.

The project was reconfigured and rerun on port 443. Despite the “not secure” warning on Google Chrome, this page has been served over SSL. The warning arises with any self-signed cert (any cert that does not have a chain leading back to a certificate authority)

Graphical user interface, text, application

Description automatically generated

## Secondary Testing

Insert screenshots below of the refactored code executed without errors and the dependency-check report.

Screenshot of error-free refactored code execution:

Text

Description automatically generated

Dependency Check Report:

Text

Description automatically generated

No additional vulnerabilities were introduced due to code refactoring.

## Functional Testing

Insert a screenshot below of the refactored code executed without errors.

The only discernible security concerns in this codebase were access modifiers in the /hash route. Wherever possible, private access modifiers should be used to prevent runtime errors that may lead to vulnerabilities. The access modifiers were changed from public to private, and the code ran error-free.

Text

Description automatically generated

## Summary

This code has been refactored to comply with security testing protocols. The primary areas of security that were addressed are the following:

**Cryptography**: Checksum validation using SHA-256 hashing algorithm.

**Client/Server**: Implementation of HTTPS using a self-signed certificate.

These specific areas have been highlighted based on the security priorities within the scope of this application. An HTTPS connection is basically a necessity in the current web landscape, so that had to be implemented immediately. Next was addressing the primary purpose of the application (generating a checksum verification), for which there was no reason *not* to use the best technology to date (SHA-256).

## Industry Standard Best Practices

First, a self-signed SSL certificate was generated to run the application over a secure HTTPS connection. While a self-signed certificate doesn’t offer the broad-spectrum trust of a certificate subsidiary to a CA, it is enough to verify the identity of the server to known clients.

Next, an industry standard cryptographic hash algorithm (SHA-256) was implemented for checksum verification. This specific algorithm was chosen for its resistance to hash collisions. As computers continue to get faster and more efficient, it is important to use the most secure algorithms to delay the possibility of brute-force or algorithmically generated hash collisions.

The company will benefit from the implementation of these best practices by circumventing common, well-known attack vectors. This will prevent the spillage of sensitive information and protect against the possibility of damage to the company’s information system.

**References**

Fatima, S., Rehman, T., Fatima, M., Khan, S., & Ali, M. A. (2022, July 29). Comparative analysis of AES and RSA algorithms for Data Security in cloud computing. MDPI. Retrieved March 24, 2023, from https://www.mdpi.com/2673-4591/20/1/14