# Practices for Secure Software Report

Zachary N. Locke

Southern New Hampshire University

CS-305 Software Security

Bing Victorio

March 2, 2025

****

# 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

Zachary Locke

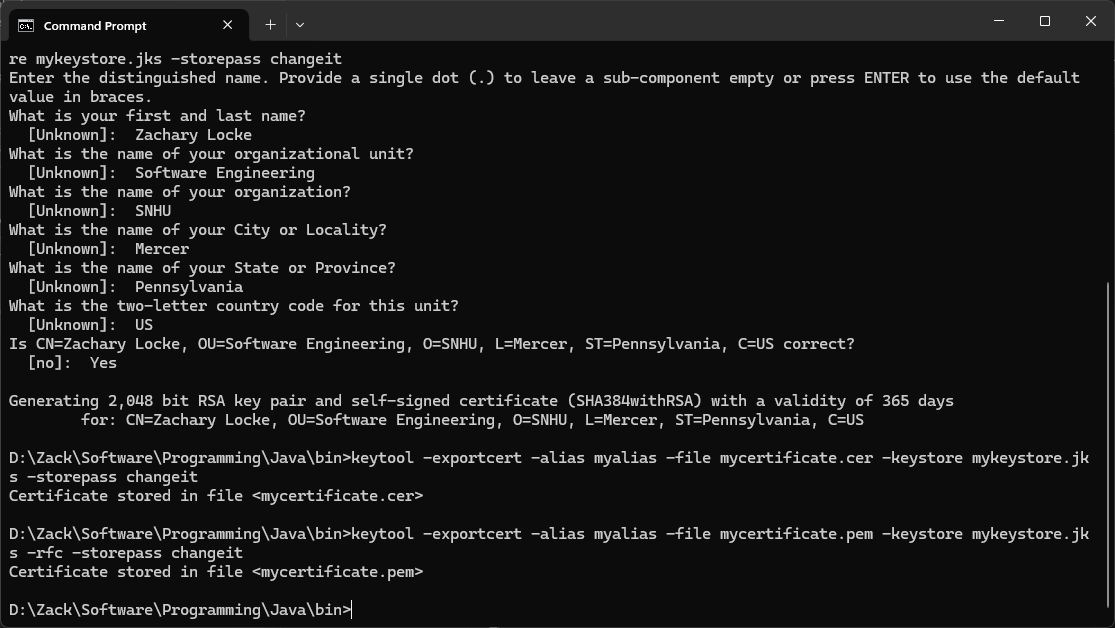
## Algorithm Cipher

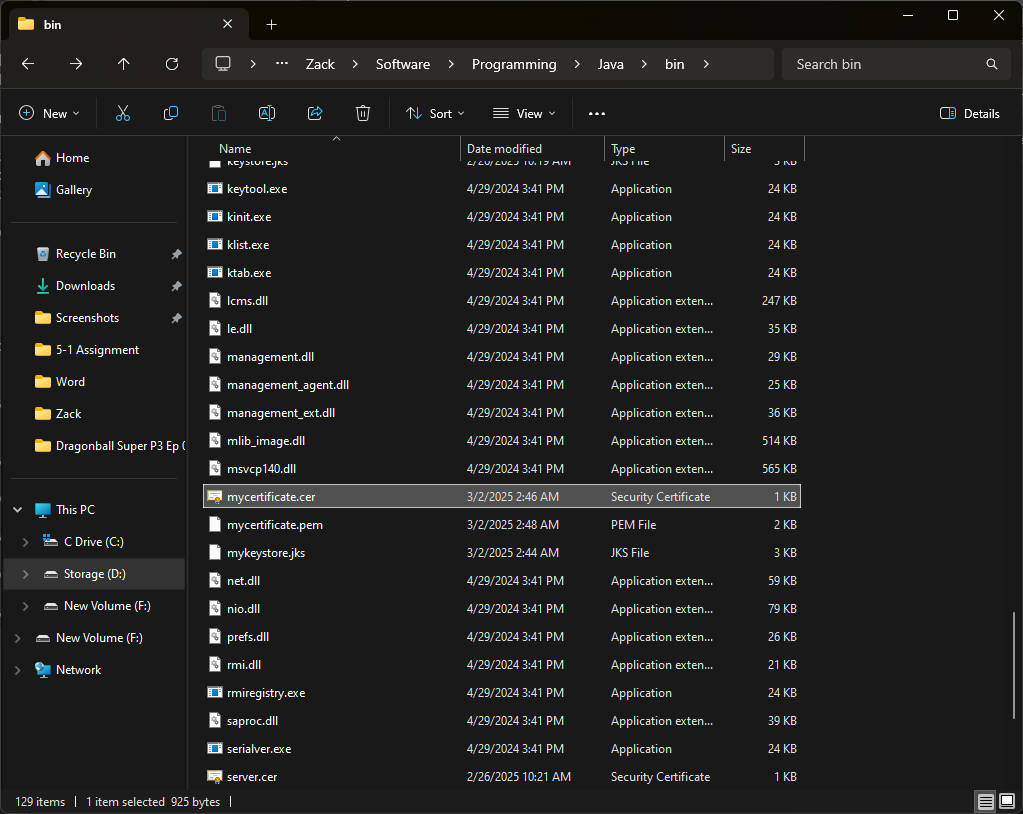
Given the security strategies, the weaknesses of the previous encryption strategies, and the security weaknesses related to the weaknesses of cruel-form, hypersensitive key control, and the weaknesses of insufficient cryptographic strength, I recommend implementing advanced encryption standards (AES). The AES was established in 2001 as FIPS 197 by the National Institute of Standards and Technology (NIST) and has ended as a globally recognized and recommended encryption standard to secure sensitive records (NIST, 2001). It encrypts data into 128-bit blocks with significant sizes of 128, 192, or 256 bits, ensuring strong security against cryptographic attacks. The encryption system consists of a couple of transformation rounds: 10 rounds for 128-bit keys, 12 for 192-bit keys, and 14 for 256-bit keys, using operations to use replacement, order, and operations (Daemen & Rijmen, 2002).

AES is typically desired over RSA (Rivest-Shamir-Adleman) for encrypting massive facts due to its drastically higher performance and speed. RSA, an uneven encryption algorithm, is predicated on the computationally high-priced prime factorization problem, making it impractical for bulk encryption. Instead, RSA is generally used for secure key exchanges, encrypting a small quantity of data to set up a session key for AES encryption (Menezes et al., 1996). Secure key storage is also necessary to prevent unauthorized rights of entry and potential key compromise. Furthermore, regular key rotation is usually recommended to mitigate dangers associated with prolonged key usage, ensuring encryption stays secure through the years (NIST, 2012). Historically, encryption has evolved notably—from historic substitution ciphers to modern cryptographic requirements. The Data Encryption Standard (DES) added in the 1970s became widely used but obsolete due to its quick 56-bit key length, making it susceptible to brute-pressure assaults (Diffie & Hellman, 1976). Recognizing the requirement for an adequate encryption standard, the NIST started the AES development process in 1997, selecting AES after a global competition to establish a more secure algorithm (Schneier, 2015). However, ongoing security promotion, including multi-layer encryption, safe primary exchange mechanisms, and hybrid cryptographic approaches, should be considered to strengthen security further.

## Certificate Generation

Insert a screenshot below of the CER file.





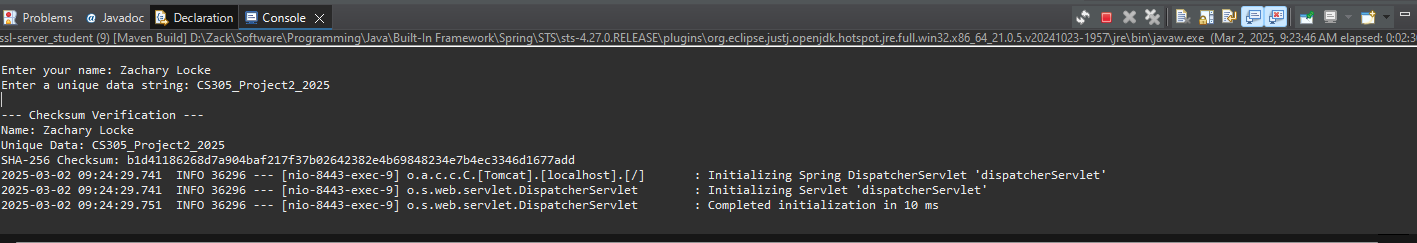
## Deploy Cipher

Insert a screenshot below of the checksum verification.A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

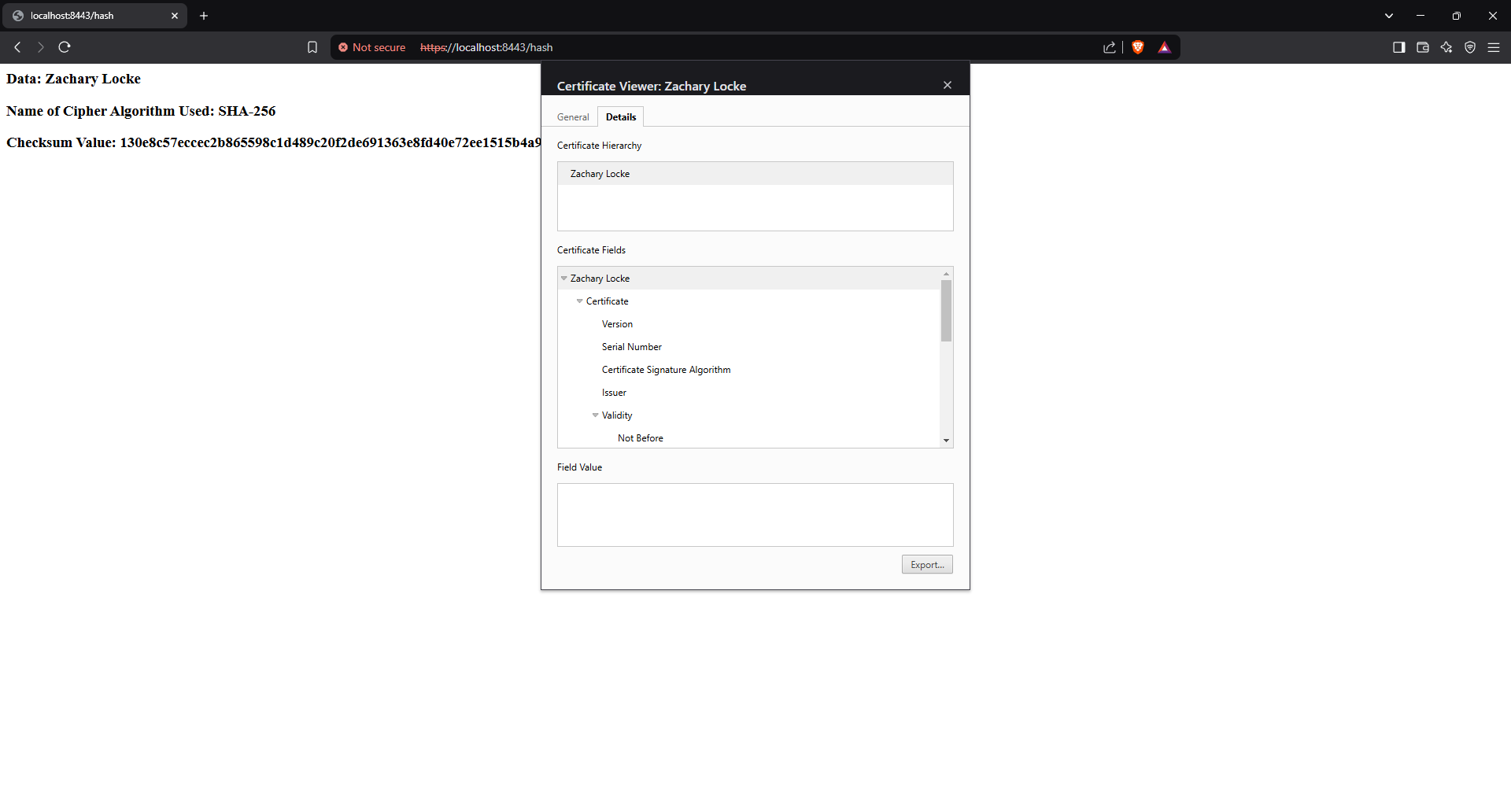


## Secure Communications

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

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.



## Secondary Testing

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

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A computer screen shot of a computer

AI-generated content may be incorrect.

## Functional Testing

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

A screenshot of a computer

AI-generated content may be incorrect.

A computer screen shot of a program

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

## Summary

The reflected code is updated to improve security and comply with the software safety test protocol. This correction of the vulgarity assessment process is achieved through the vandal evaluation process flight form, which focuses on data integrity, secure communication, and input confirmation. One of the most vital corrections is the checksum verification using SHA-256 hashing, ensuring record integrity by stopping unauthorized changes during transmission. The challenge now uses a Java KeyStore (JKS) for storing SSL certificates, strengthening the security of HTTPS communications over port 8443. To enhance input security, the code now validates input data before processing. It checks for empty inputs and enforces a maximum length of 256 characters, mitigating risks such as buffer overflow and injection attacks. An exception is thrown if invalid input is detected, preventing further execution with potentially harmful data. A massive safety-targeted alternate in the refactored code becomes the transfer from @GetMapping to @PostMapping for the /hash endpoint. POST requests ensure that touchy data is dispatched inside the request body instead of being uncovered within the URL, making it more challenging for attackers to intercept or log sensitive data. This follows best practices in secure API design, reducing the risks of data leakage and caching vulnerabilities.

The first step involved dependency analysis, where a dependency-check tool was utilized to scan for potential security vulnerabilities. This analysis identified 88 security risks, emphasizing the importance of keeping libraries up to date to mitigate potential exploits. Regular updates and monitoring third-party dependencies are crucial to maintaining a secure software environment. Following the dependency analysis, the refactoring and implementation phase focused on strengthening the application's security. Key improvements included the integration of secure hashing using SHA-256, using a Java KeyStore (JKS) for SSL certificate management, and input validation mechanisms to prevent malicious data manipulation. These updates not only enhanced security but also ensured the continued functional correctness of the application.

The final step in the process was testing and verification. The updated system underwent multiple levels of security validation, including checksum verification, dependency-check analysis, and functional execution testing. These tests confirmed that the application executed without errors, demonstrating compliance with established security standards and best practices. By incorporating @PostMapping for secure data transmission, input validation to prevent injection attacks, and enhanced communication security through HTTPS and certificate management, the refactored code adheres to industry best practices for secure software development. Potential security enhancements could include implementing authentication mechanisms, API tokens, or OAuth to restrict unauthorized access further and strengthen system security.

## Industry Standard Best Practices

To ensure the security and integrity of the software utility, industry-preferred excellent practices were applied during the development and refactoring manner. These practices have been vital in mitigating recognized security vulnerabilities, even while maintaining the existing protection of the utility. Several steady coding concepts have been followed to keep and enhance the application’s security. First, enter validation and exception dealing were applied to prevent malformed or malicious information from being processed. This step significantly reduced the risk of injection attacks, while proper exception handling was introduced to catch invalid inputs and prevent system crashes or undefined behavior. Secure cryptographic practices were also applied using SHA-256 hashing for checksum verification, ensuring data integrity. A secure keystore (mykeystore.jks) was integrated to prevent unauthorized access to cryptographic materials. This approach strengthened key management and protected sensitive information from potential security threats. Another essential improvement was securing the API design. The transition from GET to POST mapping for API requests enhanced security by preventing sensitive data from being exposed in URLs, reducing the risk of logging and caching vulnerabilities. Parameter validation was also implemented to restrict excessively long inputs, mitigating the risk of denial-of-service (DoS) attacks. Furthermore, a dependency security analysis was conducted using a dependency-check tool, identifying 88 vulnerabilities. This assessment emphasized the importance of maintaining updated libraries to address known exploits. Future improvements could involve removing or updating vulnerable dependencies to mitigate potential security threats further.

Applying industry-standard secure coding practices significantly enhances the company’s security posture and operational stability. One of the main benefits is data integrity and security. By implementing strong cryptographic hashing and input confirmation, the system ensures that the data remains unloaded and tamper-proof, which reduces the risk of corruption and fractures. In addition, it helps to follow secure coding principles, safety rules, and industry standards. Compliance reduces the risk of financial punishment associated with legal obligations and security rules. Using this practice improves the company's reputation and creates customer trust by preventing security phenomena and ensuring privacy and reliability of user data. Furthermore, secure coding practices help reduce the software's attack surface. By eliminating common vulnerabilities such as input validation failures, insecure key storage, and weak cryptographic implementations, the system becomes more resilient against potential cyber threats. It is less likely that the company utilizes a well-security application and contributes to the general security of the company's digital infrastructure. Finally, it rapidly integrates security measures into the cost-effectiveness of the development process. Addressing security weaknesses in the initial stages of software development reduces the expenses associated with post-deploy patches and reaction efforts for the incident. This active approach strengthens the company's security infrastructure and ensures that the software app is strong, reliable, and by modern safety standards. Implementing these best practices strengthens the company's commitment to secure software development and protects prestige from data, users, and potential security threats.

**References**

Daemen, J., & Rijmen, V. (2002). The design of Rijndael: AES—the Advanced Encryption Standard. Springer Science & Business Media.

Diffie, W., & Hellman, M. (1976). New directions in cryptography. IEEE Transactions on Information Theory, 22(6), 644–654.

Ferguson, N., Schneier, B., & Kohno, T. (2010). Cryptography Engineering: Design principles and practical applications. Wiley.

Menezes, A. J., Van Oorschot, P. C., & Vanstone, S. A. (1996). Handbook of applied cryptography. CRC press.

National Institute of Standards and Technology (NIST). (2001). FIPS PUB 197: Advanced Encryption Standard (AES). <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197-upd1.pdf>

National Institute of Standards and Technology (NIST). (2012). Recommendation for key management—Part 1: General (NIST Special Publication 800-57 Part 1 Rev. 3). <https://csrc.nist.gov/publications/detail/sp/800-57-part-1/rev-3/final>

Schneier, B. (2015). Applied cryptography: Protocols, algorithms, and source code in C. Wiley.