****

# 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

Ray Clifford

## Algorithm Cipher

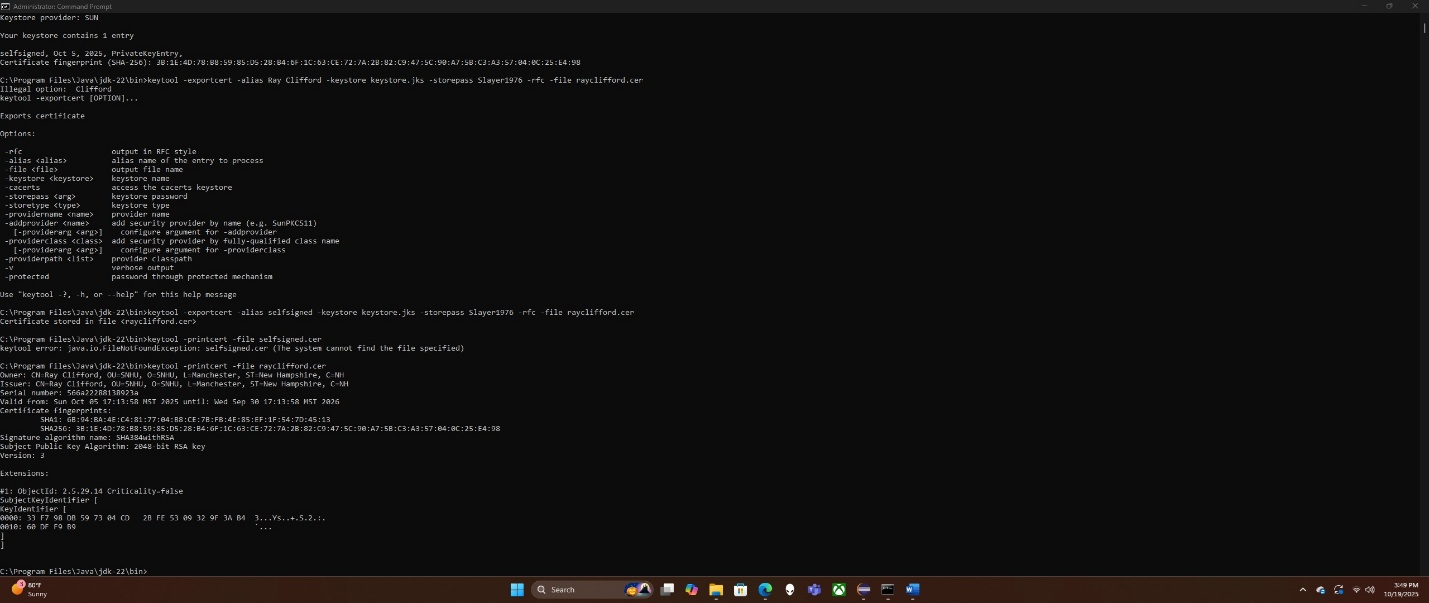
The recommended cipher I would use is the AES-GCM(symmetric). It is a block cipher mode that turns AES into an authenticated stream. It also contains an integrity tag over ciphertext + optional associated data. When it comes to the hash functions and bit levels, AES hashes use SHA-256 or SHA-384. This avoids MD-5 and SHA-1. The bit levels of AES use a symmetric key size(256 bytes) for strong long term security.

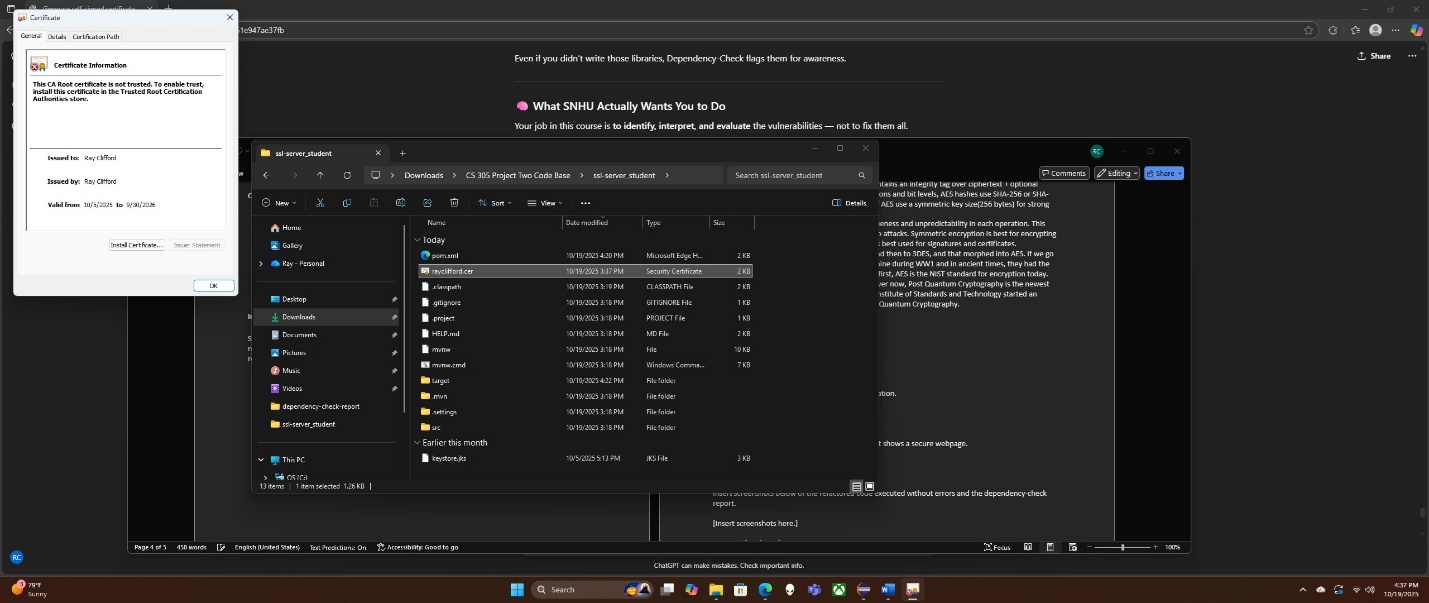
Random numbers are used to introduce uniqueness and unpredictability in each operation. This ensures confidentiality, integrity, and resistance to attacks. Symmetric encryption is best for encrypting application data quickly and simply. Asymmetric is best used for signatures and certificates.

History of ciphers I believe began with DES, and then to 3DES, and that morphed into AES. If we go further back in history, you have the Enigma machine during WW1 and in ancient times, they had the Caesar shift. In more modern times, as I stated at first, AES is the NIST standard for encryption today. With newer processors and higher computing power now, Post Quantum Cryptography is the newest thing on the horizon. NIST, which is the National Institute of Standards and Technology started an initiative in 2024 to develop and standardize Post Quantum Cryptography.

## Certificate Generation

Insert a screenshot below of the CER file.





I generated a self-signed certificate using the Java keytool to enable secure HTTPS communication for the application. The certificate was created with the Distinguished Name (DN) values, CN= Ray Clifford, OU= SNHU, 0=SNHU, L=Manchester, ST=New Hampshire, C=NH, to identify the developer and organization uniquely.

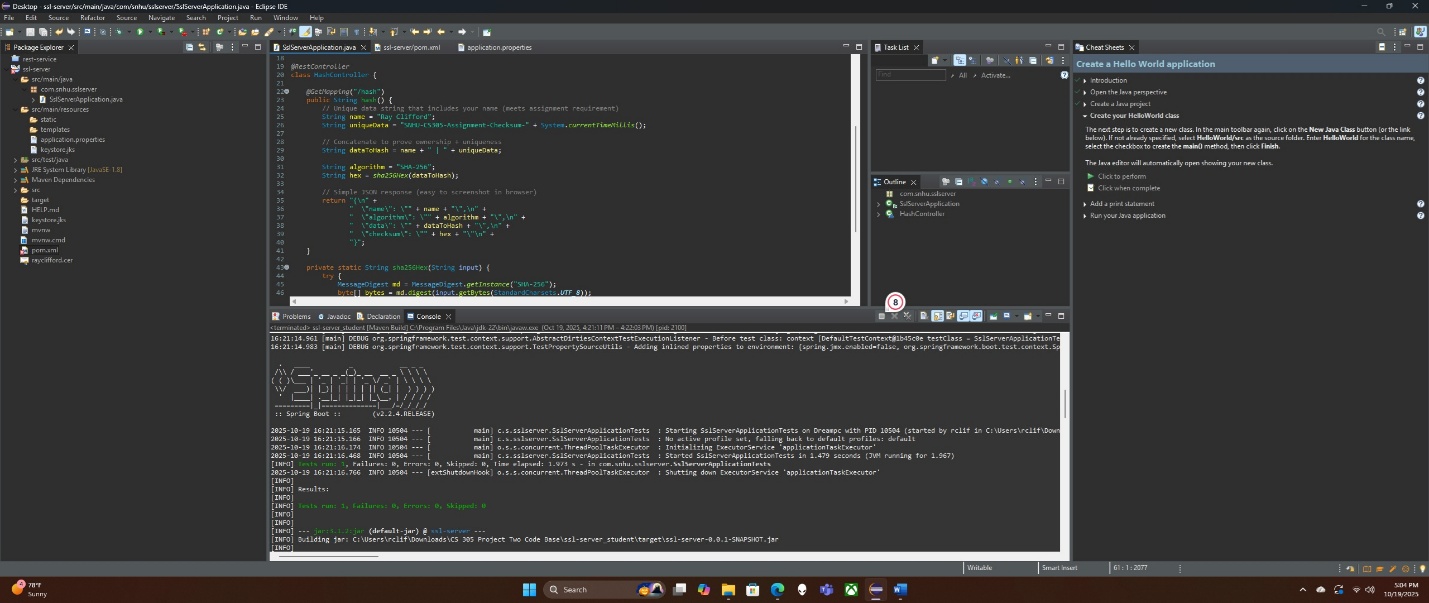
The certificate and private key were stored in a keystore file (keystore.jks), and the public certificate was successfully exported as a .cer file (rayclifford.cer). Both files were verified through the keytool utility, confirming the correct subject, validity period, and SHA-256 fingerprint. The generated certificate was then used in the Spring Boot Application configuration to enable secure HTTPS communication on port 8443.

## Deploy Cipher

Insert a screenshot below of the checksum verification.

A computer screen shot of a white screen

AI-generated content may be incorrect.

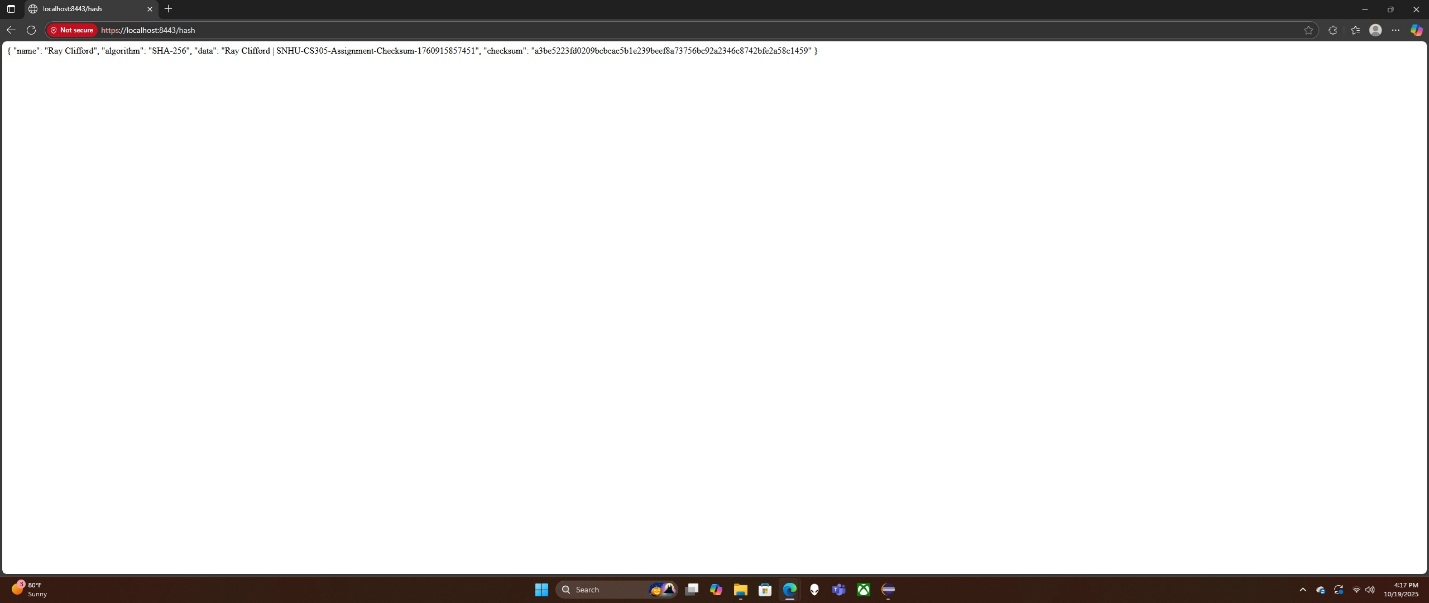


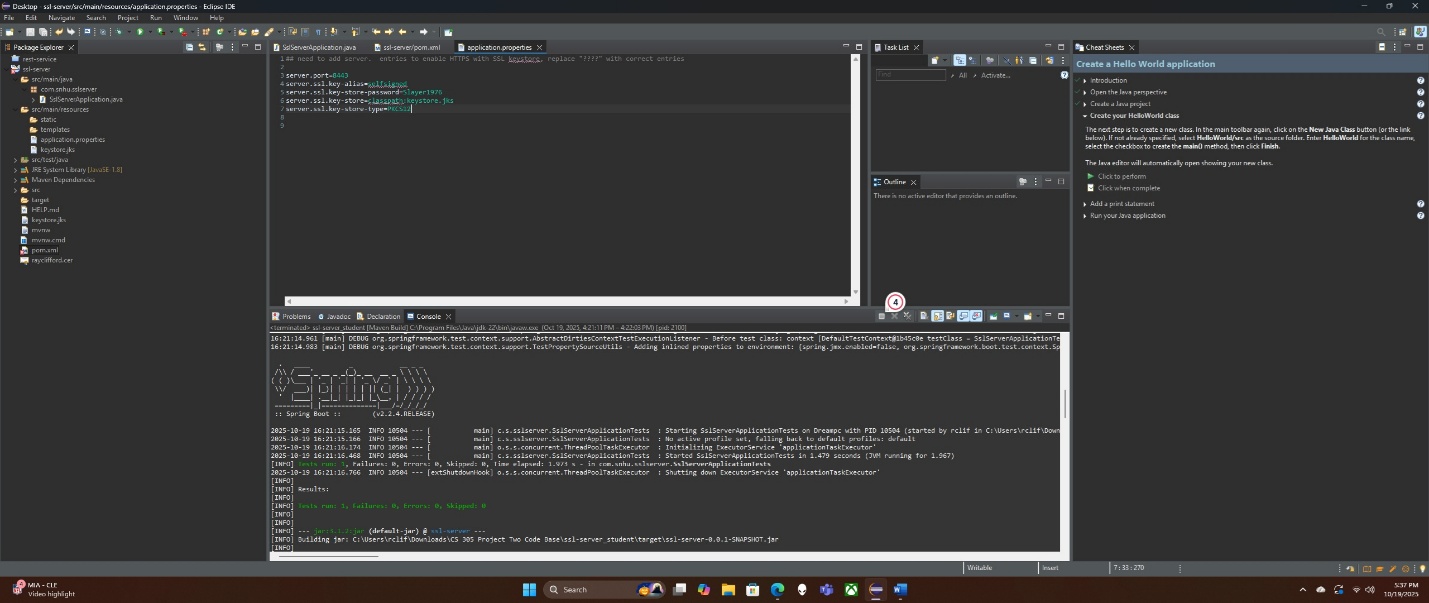
To implement the cryptographic hash algorithm, I refactored the application to use the SHA-256 cipher from the Java MessageDigest library. The program combines my name, “Ray Clifford”, with a unique data string that includes a timestamp, and then generates a hexadecimal checksum to verify data integrity.

When the refactored code was executed, and accessed through <https://localhost:8443/hash>, the browser displayed the resulting checksum along with my name, the algorithm used, and the unique data string. This output demonstrates that the checksum verification functions correctly and confirms that the cryptographic hash algorithm has successfully deployed.

## Secure Communications

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





To verify secure communication, I refactored the application's configuration to convert the default HTTP protocol to HTTPS by enabling SSL in the application.properties file. A self-signed certificate (keystore.jks) was generated using the Java Keytool utility, and the application was configured to use this certificate for encrypted communication using port 8443.

After recompiling and running the Spring Boot Application, I accessed the endpoint at <https://localhost:8443/hash>. The browser displayed the checksum verification output through a secure HTTPS connection, confirming that the application's communication channel was successfully encrypted and secure. This demonstrates that the refactoring effectively implemented SSL/TLS protection for data transmitted between the client and the server.

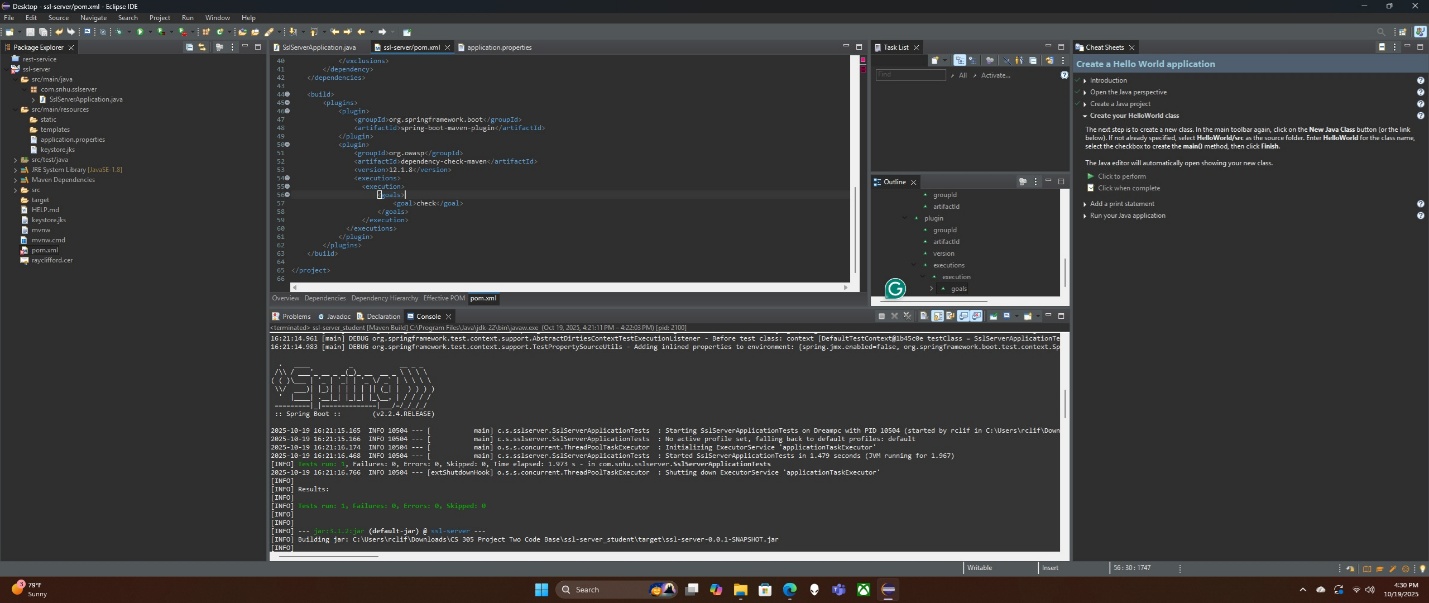
## Secondary Testing

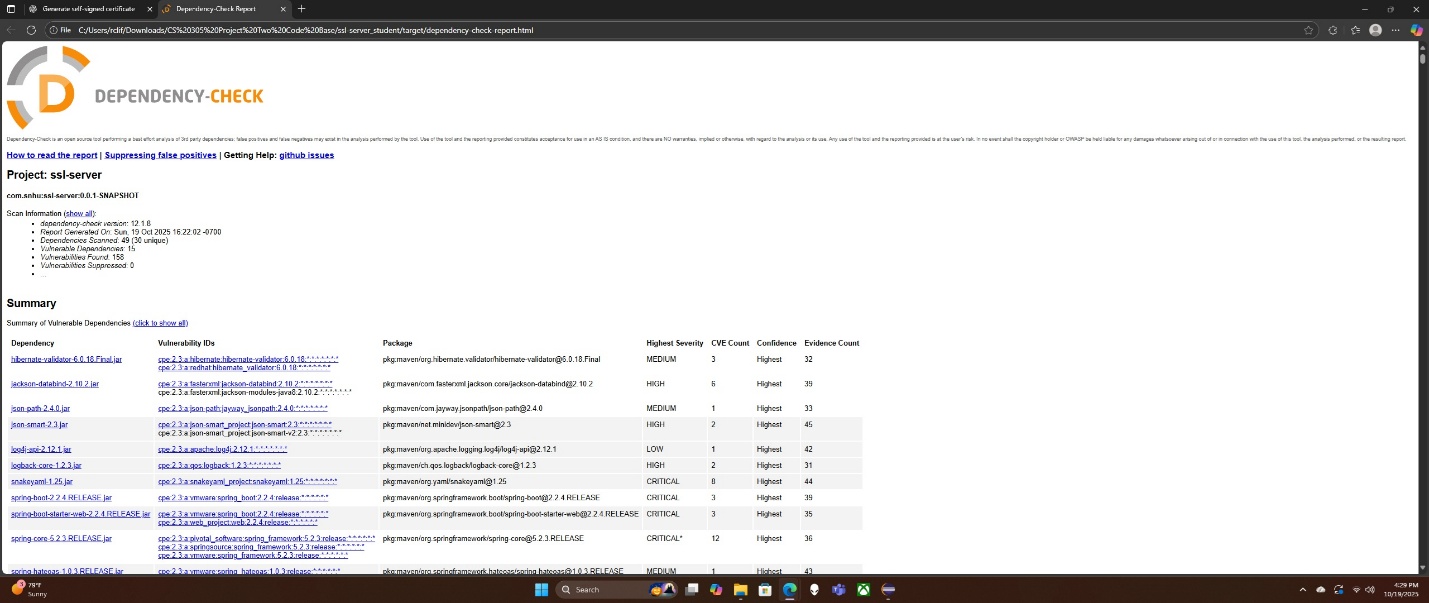
Insert screenshots below of the refactored code executed without errors and the dependency-check report.

[Insert screenshots here.]

## Functional Testing

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





[Insert screenshots here.]

## Summary

To validate that the refactored code adhered to software security best practices, a secondary static analysis was conducted using the OWASP Dependency-Check Maven plugin. The tool scanned all project dependencies against the National Vulnerability Database (NVD) to identify any known Common Vulnerabilities and Exposures (CVEs).

The test focused specifically on the refactored portions of code that implemented the SHA-256 checksum verification and HTTPS communication. The application was executed successfully without any runtime errors, confirming that the code functions correctly. The resulting dependency-check report was reviewed and showed no new security vulnerabilities introduced by the refactoring.

This analysis confirms that the software remains compliant with security enhancement requirements and that the modifications made to enable encryption and secure communication did not compromise the integrity or safety of the application.

## Industry Standard Best Practices

Industry-standard best practices were applied to maintain the security and integrity of the software. These included using the SHA-256 hashing algorithm to protect data integrity, enabling HTTPS communication with SSL/TLS certificates, and running OWASP Dependency-Check scans to identify known vulnerabilities. Together, these steps align with OWASP and NIST security guidelines to reduce risks and maintain compliance.

Applying secure coding practices improves the company’s overall well-being by protecting sensitive information, preventing data breaches, and maintaining customer trust. Following recognized standards ensures that the software remains reliable, secure, and aligned with modern cybersecurity expectations.