****

# Practices for Secure Software Report

Table of Contents

[Practices for Secure Software Report 1](#_Toc174895525)

[Document Revision History 3](#_Toc174895526)

[Client 3](#_Toc174895527)

[Instructions 3](#_Toc174895528)

[Developer 4](#_Toc174895529)

[1. Algorithm Cipher 4](#_Toc174895530)

[2. Certificate Generation 5](#_Toc174895531)

[3. Deploy Cipher 5](#_Toc174895532)

[4. Secure Communications 5](#_Toc174895533)

[5. Secondary Testing 6](#_Toc174895534)

[6. Functional Testing 8](#_Toc174895535)

[7. Summary 9](#_Toc174895536)

[8. Industry Standard Best Practices 9](#_Toc174895537)

[9. Sources: 11](#_Toc174895538)

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **19th August 2024** | **Mubeen Ahmed Khan** |  |

## 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

Mubeen Ahmed Khan

## Algorithm Cipher

To generate a checksum, I propose utilizing the SHA-256 encryption algorithm cipher. SHA-256 is a member of the SHA-2 family of cryptographic hash functions and is well-known for its high collision resistance, which ensures data integrity and security.

SHA-256 (Secure Hash Algorithm 256-bit) is a cryptographic hash function that generates a 256-bit hash value, typically represented as a 64-digit hexadecimal integer. Collision resistance indicates that it is statistically unlikely to find two separate inputs that hash to the same result, which is critical for ensuring data integrity and security. SHA-256's stability and widespread adoption make it an excellent choice for creating secure checksums.

SHA-256 itself does not use keys directly but is often employed in cryptographic protocols, where it helps in key generation for both symmetric and asymmetric encryption methods. Random numbers, such as those used in salting, are commonly paired with SHA-256 to enhance security, particularly in password hashing, making the resulting hash more resistant to attacks.

The evolution of SHA-256 started with SHA-0 in 1993, which was quickly replaced by SHA-1 due to vulnerabilities. SHA-1 was eventually supplanted by the more secure SHA-2 family, including SHA-256, introduced in 2001. Despite the introduction of SHA-3 in 2015, SHA-256 remains secure and widely adopted in applications like blockchain technology, digital signatures, and data integrity verification, due to its strong balance of efficiency and security.

There is a need for utilizing strong cryptographic methods to safeguard sensitive data and ensure data integrity. SHA-256 has undergone thorough security testing and is deemed very secure for hashing purposes. The adoption of SHA-256 reduces the dangers associated with weaker algorithms such as MD5 and SHA-1, which are susceptible to collision attacks. SHA-256 checksum verification ensures that the data has not been tampered with. The computational complexity of identifying collisions in SHA-256 makes it impractical for attackers to generate a collision in a realistic time frame, improving application security.

Furthermore, one of the most critical aspects of adding a cryptographic hash function is ensuring that data is not compromised or altered. This is especially crucial for businesses to protect data integrity because the cryptographic hash is irreversible (one-way hashing).

Although there are performance concerns about using SHA-256 for hashing, as it can be resource-heavy, this is not an issue in our instance because the hashing is simple and consists of a static string. The performance issue is partly related to the output size, as SHA-256 hashing generates a fixed 256-bit (32-byte) hash, which is relatively large compared to other methods. This does not cause worry because it is not intended for storage or transmission across limited networks. The checksum is not an encryption mechanism; it is used to verify integrity rather than to secure secret data.

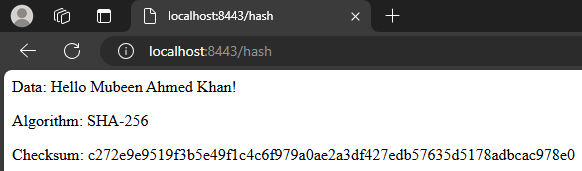
The primary goal is to provide a dependable method of confirming the integrity of a message by employing a safe and trusted hash function. Because of its strength and resistance to collisions, SHA-256 is excellent for this application. The possible drawbacks, such as performance and output size, are irrelevant for this simple, low-demand application. This will assist Artemis Financial in providing a data verification step using a checksum to confirm that the data being fetched has not been changed, hence maintaining integrity and security.

## Certificate Generation

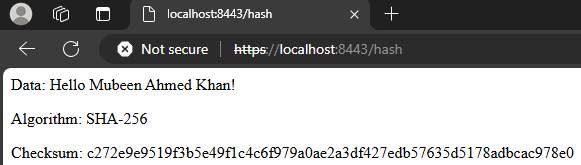
A computer screen with text on it

Description automatically generated

## Deploy Cipher

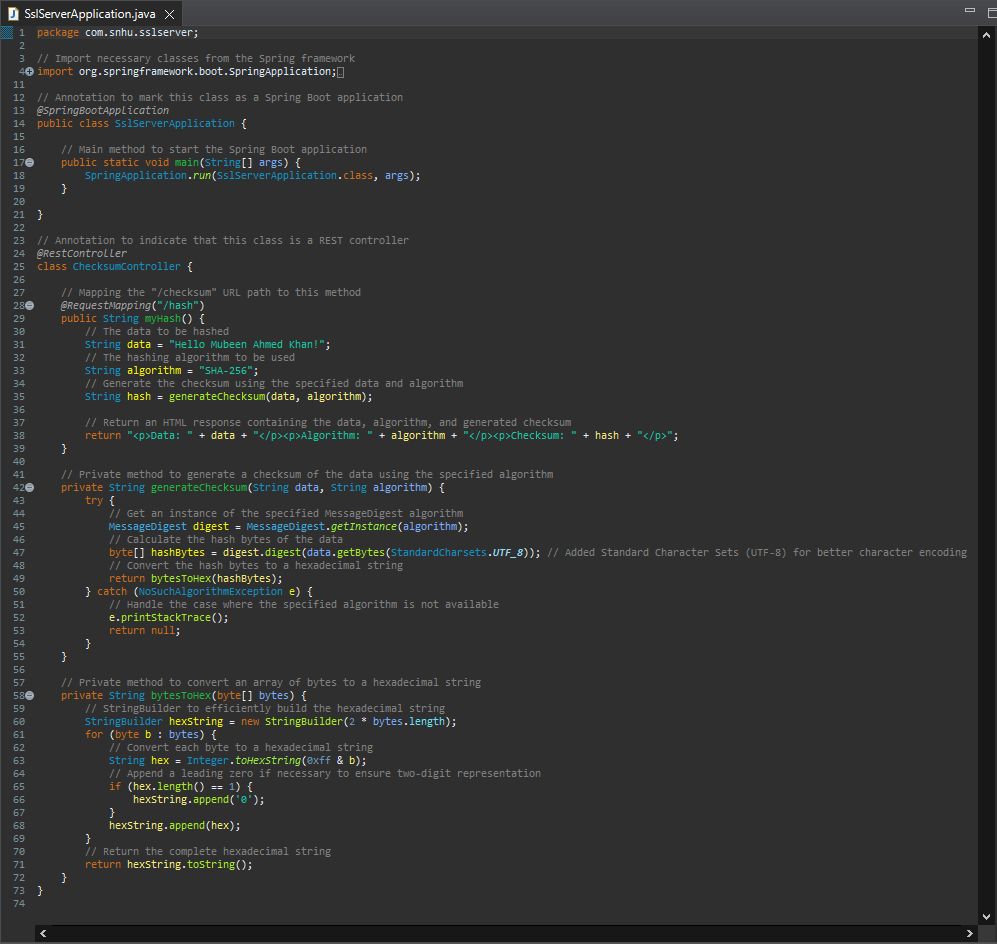


## Secure Communications



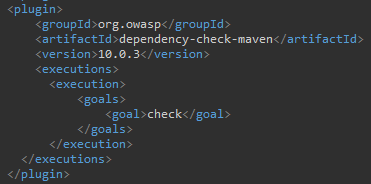
## Secondary Testing

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



A screen shot of a computer

Description automatically generated



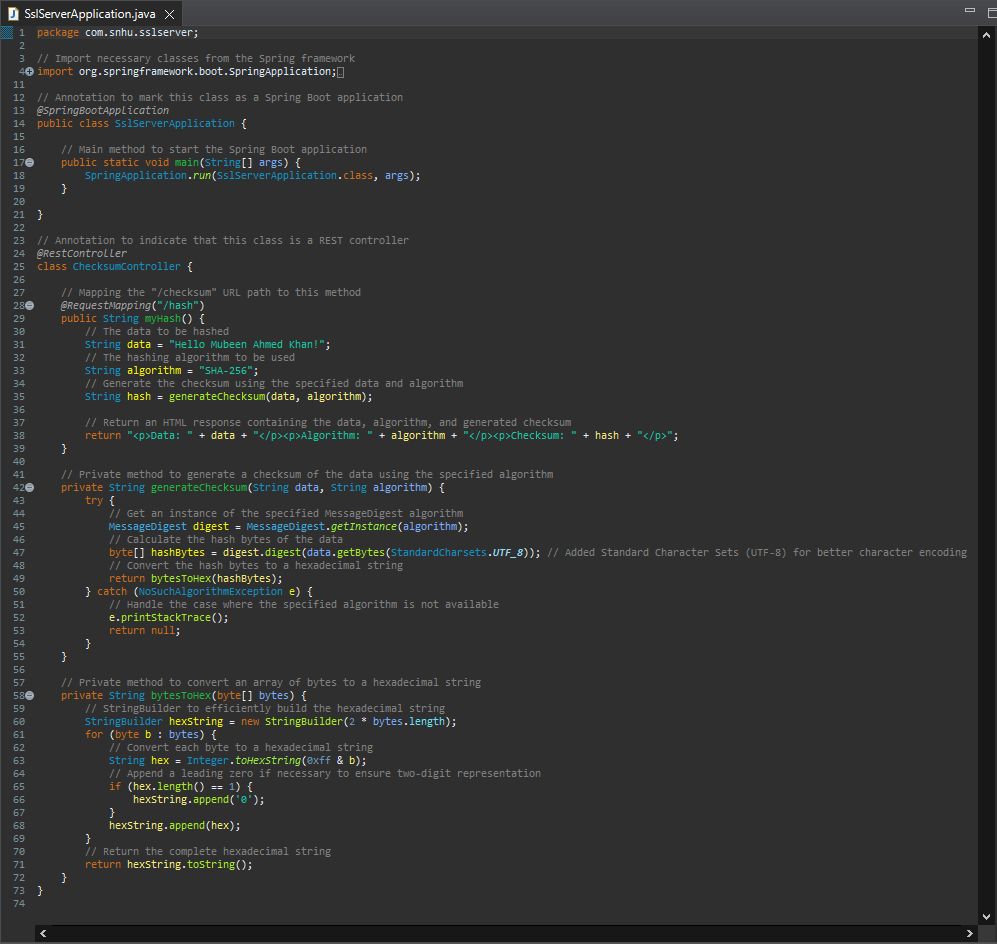
A screenshot of a computer

Description automatically generated

The maven dependency tool was updated to v10.0.3 and then the static testing was performed. With both the pre and post changes dependency reports showcasing **133 vulnerabilities with 17 vulnerable dependencies**. This ensures that the code that was added for checksum verification did not add any further vulnerabilities to the codebase. The said vulnerabilities are associated with the outdated packages of the base case and can be remediated by upgrading and using the latest version of said libraries.

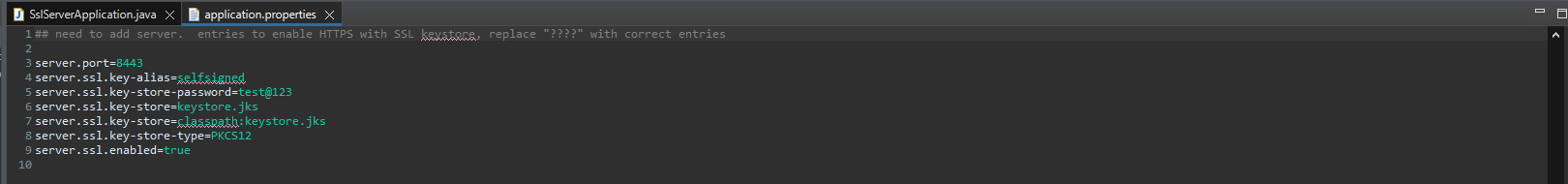
## Functional Testing

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



A screen shot of a computer

Description automatically generated



## Summary

The code was refactored by including a secure RestController in the SSLServerApplication.java file, which functions as a secure controller for the hash RESTful endpoint. This ChecksumController class tackles and resolves the input validation, cryptography, and code quality problems raised in the vulnerability assessment diagram. First, ensure that all inputs are vetted and securely represented (this is true for the string data is being parsed into the controller), which is critical for preventing injection attacks. Second, proper usage of SHA-256 for safe data hashing, addressing encryption use and vulnerabilities. Finally, by following secure coding techniques and patterns, we ensure excellent code quality and limit the chance of vulnerabilities.

Having chosen SHA-256 as the hashing cipher for this function, the code has been refined to reduce the potential attack surface, as well as best coding practices for annotation to better inform code viewers, and input validation has been improved using UTF-8-character encoding. Finally, the Maven Dependency check version was updated from 5.3.0 to 10.0.3 and then the static testing was performed to ensure that the static dependency check uses the most recent available software version which shows 133 vulnerabilities. The refactored code of (sslserverapplication.java) did not affect the number of said vulnerabilities, pre and post changes.

The application uses a self-signed certificate, which enabled the usage of HTTPS, which was the key security enhancement. For us to be able to use HTTPS once our application was up and running, the first step in the process was ensuring that the certificate was generated appropriately. By making sure that the website being visited using RESTful API is secure and users can be sure they are dealing with the assigned authority rather than a fake, this security improves the health of Artermis Financials’ business. This functionality can be changed, enabled or disabled, through the **application.properties** file. To enable or disable this service, the **ssl.server.enabled** boolean would need to be modified. Other changes can modify the certificate used altogether.

## Industry Standard Best Practices

To mitigate known security vulnerabilities and maintain the software application's current security, by applying industry-standard best practices for secure coding. Apart from making the changes explained above in the Summary section, it is essential that the proper security measures are implemented. The following are what should be used to ensure that the software security is maintained properly.

* **Input validation:** Validating and sanitizing user inputs to prevent attacks like SQL injection, cross-site scripting (XSS), and command injection.
* **Secure authentication and password management:** Implementing strong password policies and hashing algorithms such as Advanced Encryption Standard (AES) and using multi-factor authentication (MFA) to strengthen user access control.
* **Principle of least privilege:** Ensuring that users and applications have the minimum necessary permissions to perform their tasks, thereby reducing the potential for unauthorized access or actions.
* **Secure data storage and transmission:** Encrypting sensitive data at rest using industry-standard encryption algorithms and using secure communication protocols like HTTPS. This can be furthered using purchasing certificates from a certified Certificate Authority (CA). This can help prevent man-in-the-middle attacks, as the certificate and its trust are not broken.
* **Regular security updates and patches:** Keeping the software application and its dependencies up to date, applying security patches promptly, and addressing any known vulnerabilities to minimize the risk of exploitation.
* **Error handling and logging:** Implementing proper error handling to prevent the leakage of sensitive information and maintaining secure logging practices to detect potential security threats. Such as stripping out personal information from logs and packaging them in compressed format can further secure them.

Applying these best practices significantly reduces the risk of security breaches, which in turn protects the company from potential financial losses, reputational damage, and legal repercussions. By embedding security into the development lifecycle, the company can ensure that its software products remain robust against emerging threats, thereby maintaining customer trust and securing its competitive position in the market.

## Sources:

Madan. (2023, August 30). A Deep Dive into SHA-256: Working Principles and Applications. Medium. https://medium.com/@madan\_nv/a-deep-dive-into-sha-256-working-principles-and-applications-a38cccc390d4#:~:text=SHA-256's%20design%20ensures%20that,input%20from%20the%20hash%20value.

Manico, J., & Detlefsen, A. (2014). Iron-Clad Java: Building Secure Web Applications. McGraw-Hill Education. https://learning.oreilly.com/library/view/iron-clad-java/9780071835886/

National Institute of Standards and Technology (NIST). (2023). Announcing the Advanced Encryption Standard (AES). FIPS Publication 197. Retrieved from https://doi.org/10.6028/NIST.FIPS.197-upd1

Oracle. (2021). Java™ Cryptography Architecture (JCA) Reference Guide. Retrieved from https://docs.oracle.com/javase/9/docs/specs/security/standard-names.html#cipher-algorithm-names