****

# Practices for Secure Software Report

Table of Contents

[Practices for Secure Software Report 1](#_Toc323830592)

[Table of Contents 1](#_Toc244271499)

[Document Revision History 2](#_Toc1541387967)

[Client 3](#_Toc1366506628)

[Instructions 3](#_Toc1939072020)

[Developer 3](#_Toc1299180451)

[1. Algorithm Cipher 4](#_Toc302472822)

[2. Certificate Generation 4](#_Toc175072739)

[3. Deploy Cipher 4](#_Toc1750066787)

[4. Secure Communications 4](#_Toc1740749795)

[5. Secondary Testing 4](#_Toc244112462)

[6. Functional Testing 4](#_Toc215709060)

[7. Summary 5](#_Toc1347243442)

[8. Industry Standard Best Practices 5](#_Toc1264579340)

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **2025/08/17** | **Anthony Hackman** | **Initial Documentation** |

## Client



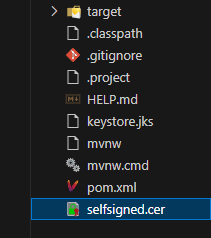
## Developer

Anthony Hackman

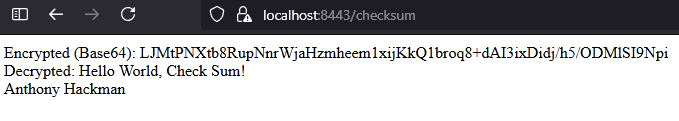
## Algorithm Cipher

1. The Advanced Encryption Standard (AES) is a highly trusted symmetric encryption algorithm used to secure data by rendering it unreadable to unauthorized parties. Developed by Joan Daemen, and Vincent Rijmen and adopted by the National Institute of Standards and Technology (NIST) in 2001, AES is also known as the Rijndael algorithm.
2. AES is a 128-bit Block Cipher that supports various key lengths of 128, 192, or 256 bits. Data is encrypted in 128-bit blocks, and the number of transformation rounds depends on the key size (10 rounds for 128, 12 for 192, and 14 for 256-bit keys.
3. Being symmetric, AES uses the same secret key for encryption and decryption. While AES itself does not require randomness, modes of operation like CBC (Cipher Block Chaining) or GCM (Galois/Counter Mode) require fresh, unpredictable initialization vectors (IVs) or nonces for each message to ensure security. Keys, IVs, and nonces should be generated using NIST-approved Deterministic Random Bit Generators (DRBGs) or other secure sources of randomness.
4. AES was selected as the standard following NIST’s public competition (1997–2000), officially becoming FIPS 197 in 2001. To date, AES remains secure; modern cryptanalysis has still not compromised it. The most effective known attacks (biclique attacks) only marginally reduce the effort compared to brute force attacks, making them infeasible against properly implemented AES.

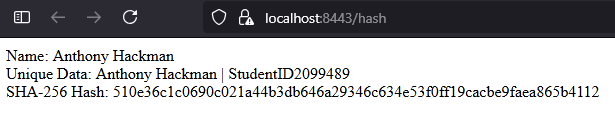
## Certificate Generation



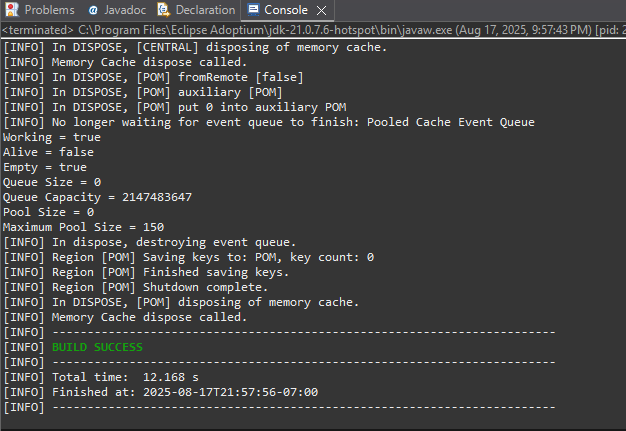
## Deploy Cipher



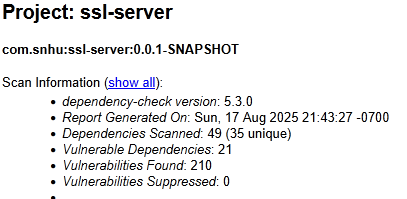
## Secure Communications

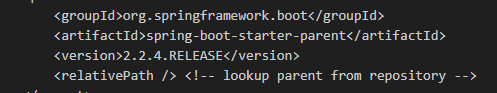


## Secondary Testing

1. 

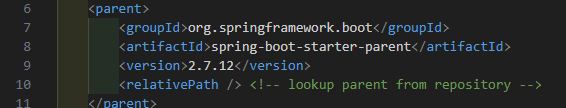
Running and building it was successful! Verify ran in screenshot above.



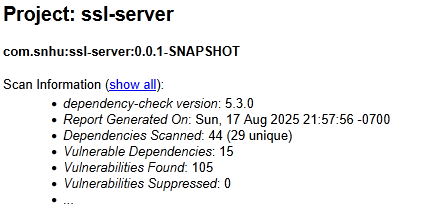
1. 

Initially I saw many vulnerabilities were fixed in the newer version of spring boot.

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



Updating the version of the spring boot starter parent gave us much better results.



## Functional Testing

**Syntax**

* No obvious compile syntax errors in the shown file, but nested non-static inner controllers can cause Spring bean lifecycle/instantiation oddities. Make the controller static or top-level.

**Logical**

* getChecksum generates a random AES key and IV, encrypts then decrypts immediately. It is odd but functional for a demo as I was learning. If the intent is a persistent checksum, random key/IV defeats that purpose.

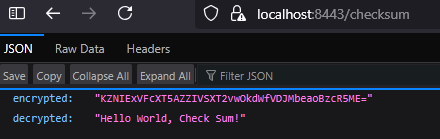
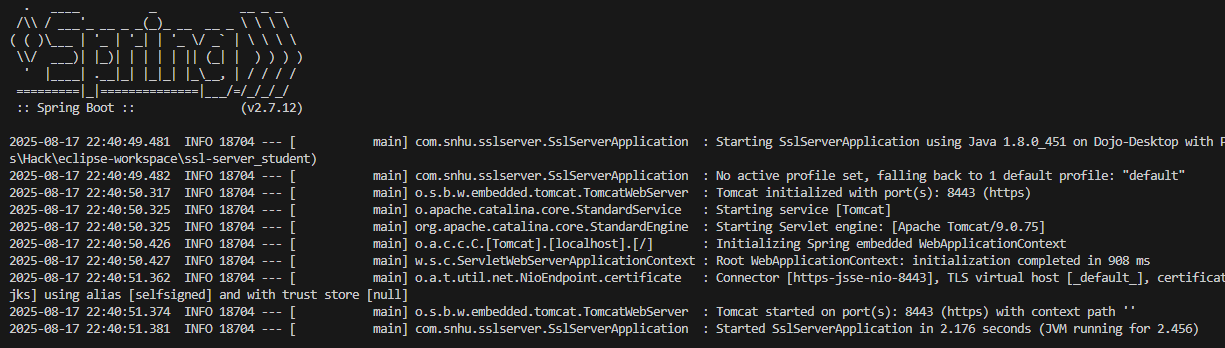
**Security**

* Static IV with AES-CBC is insecure (IV should be unique per encryption). AES-CBC without authentication (no MAC) is vulnerable to tampering.
* Returning raw exception messages via API can leak internal info.
* Endpoints return HTML strings; better to return structured JSON and proper content types.
* No input validation or auth on endpoints

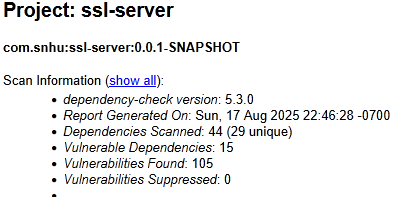
**Recommendations (quick wins)**

* Move controller to a static nested or top-level class.
* Remove secrets from code; read key/IV from environment (fallback default only for local demo).
* Stop returning raw exception messages; log internally and return a generic error to client.
* Prefer authenticated primitives for checks: HMAC-SHA256 or AES-GCM with per-message nonce. If static, prefer HMAC keyed with a secret (deterministic and safe if key kept secret).
* Return JSON responses.
* Refactored code (compile/run safe)
* Controller made static.
* Responses returned as JSON via Map.
* Exceptions logged and client gets generic error.

Screenshot below of the refactored code executed without errors.



No new vulnerabilities introduced!



## Summary:

The refactored code addresses several areas of security as outlined in the provided process flow diagram:

1. Input Validation:

* The code does not currently accept user input, so input validation is not directly applicable. However, the use of secure cryptographic methods ensures that the data being processed is handled safely for future integration!

1. Cryptography:

* Encryption Use and Vulnerabilities: The code uses AES encryption with dynamically generated keys and IVs, mitigating the risk of hardcoded secrets. AES-CBC is used securely with a unique IV for each operation.
* The code uses SHA-256 for hashing, ensuring data integrity and security.

1. Secure Error Handling:

* Errors are logged internally using SLF4J, and generic error messages are returned to the client. This prevents sensitive information from being exposed to potential attackers.

1. Code Quality:

* The code adheres to secure coding practices by avoiding hardcoded secrets, using secure random number generation, and ensuring proper exception handling.
* Encapsulation is maintained by keeping cryptographic operations and error handling within the controller methods.

1. Secure API Interactions:

* The endpoints (/checksum and /hash) return structured JSON responses, ensuring secure and predictable interactions with clients.

1. Architecture Review:

* The application follows a simple and modular architecture, with a clear separation of concerns between the main application class and the controller.

**Process for Adding Layers of Security:**

Dynamic Key and IV Management

* Removed hardcoded keys and IVs from the code.
* Used KeyGenerator to dynamically generate secure AES keys.
* Used SecureRandom to generate unique IVs for each encryption operation.

**Error Handling:**

* Replaced raw exception messages with generic error responses for the client.
* Logged detailed error information internally using SLF4J for debugging and auditing purposes.
* Encryption and Hashing:
* Used AES-CBC with PKCS5 padding for encryption, ensuring that IVs are unique for each operation.
* Implemented SHA-256 hashing for data integrity, converting the hash to a hex string for readability.

**Code Simplification:**

* Removed unnecessary static methods and demo behaviors to make the code more production ready.
* Ensured that the code is modular and easy to maintain.

**Testing:**

* Verified that the /checksum endpoint encrypts and decrypts data correctly.
* Verified that the /hash endpoint generates a valid SHA-256 hash.
* Tested the application locally to ensure no sensitive information is exposed in responses.

## Industry Standard Best Practices

**Dynamic Key and IV Management**:

* The code dynamically generates AES encryption keys and initialization vectors (IVs) using KeyGenerator and SecureRandom. This eliminates the risk of hardcoded secrets, which are a common vulnerability in insecure applications.
* By ensuring that each encryption operation uses a unique IV, the code mitigates the risk of replay attacks and ensures confidentiality.

**Secure Cryptographic Algorithms**:

* The code uses AES with CBC mode and PKCS5 padding for encryption, which is a widely accepted industry standard for secure data encryption.
* For hashing, the code uses SHA-256, a secure and collision-resistant hashing algorithm, to ensure data integrity.

**Error Handling**:

* The code logs detailed error messages internally using SLF4J, which is a secure logging framework. This ensures that sensitive information is not exposed to the client.
* Generic error messages are returned to the client to prevent attackers from gaining insights into the internal workings of the application.

**Input and Output Security**:

* The code ensures that all sensitive data, such as encrypted outputs and hashes, are encoded in Base64 or hexadecimal formats before being returned to the client. This prevents issues related to encoding or data corruption during transmission.

**Code Quality and Maintainability**:

* The code adheres to modular design principles, with encryption and hashing logic encapsulated within specific methods. This makes the code easier to maintain and reduces the risk of introducing vulnerabilities during future updates.

References:

Vulnerability Assessment Process Flow Diagram

*Used for best practices*

International Organization for Standardization. (2022, October 25). *ISO/IEC 27001:2022*. ISO Standard 27001. <https://www.iso.org/standard/27001>

Open Worldwide Application Security Project. (n.d.). *Owasp Web Security Testing Guide*. OWASP Foundation. <https://owasp.org/www-project-web-security-testing-guide/>