

# 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 | 2/23/2025 | Gordon Price |  |

## Client



## Developer

Gordon Price

## *Algorithm Cipher*

AES (Advanced Encryption Standard) is the symmetric encryption standard in the industry. It is widely employed to encrypt lots of data since it is fast and provides strong security assurances. AES comes in 128, 192, and 256-bit key sizes and satisfies governmental standards such as NIST, FIPS 140-2, and GDPR.

**Key Strengths of AES**

* **High Security:**
  + It is resistant against brute-force attacks, especially with a 256-bit key.
  + When it is used with modes like GCM, AES provides both encryption & authentication.
* **Performance Efficiency:**
  + Optimized for hardware and software implementation, ensuring fast encryption with minimal performance overhead.
  + Hardware acceleration is widely available, improving throughput.
* **Regulatory Compliance:**
  + It is recognized by international standards, ensuring adherence to strict compliance requirements.

**Comparative Analysis with Other Algorithms**

* **Symmetric vs. Asymmetric Encryption:**
  + **AES (Symmetric Encryption):**
    - Best suited for bulk data encryption due to its efficiency.
    - Uses the same key for encryption and decryption, requiring strong key management practices.
  + **RSA/ECC (Asymmetric Encryption):**
    - Commonly used for key exchange and digital signatures rather than encrypting large volumes of data.
    - ECC provides strong security with smaller keys compared to RSA.
* **Alternative Encryption Methods:**
  + **ChaCha20:**
    - Efficient for mobile devices and resource-constrained environments.
  + **Legacy Algorithms (DES, 3DES, Blowfish):**
    - Outdated and vulnerable to modern cryptographic attacks.

**Trade-offs and AES Selection Rationale**

* **Security vs. Performance:**
  + AES-256 balances strong security with high performance, making it ideal for encrypting large datasets and long-term archival encryption.
* **Implementation Considerations:**
  + The primary vulnerabilities in AES arise from improper key management and IV reuse.
  + Secure storage of keys, IV management, and the use of Hardware Security Modules (HSMs) mitigate these risks.

## *Certificate Generation*

## A screenshot of a certificate AI-generated content may be incorrect.A computer screen shot of a computer screen AI-generated content may be incorrect.

Entering Distinguished Name Information:

* (Common Name): Gordon Price
* (Organizational Unit): CS-305
* (Organization): SNHU
* (Locality): Smithfield
* (State): Rhode Island
* (Country): US

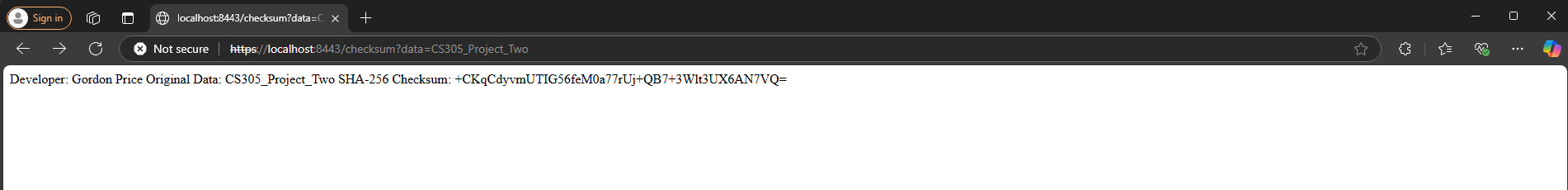
Configuring the Application Properties:

A screenshot of a computer program

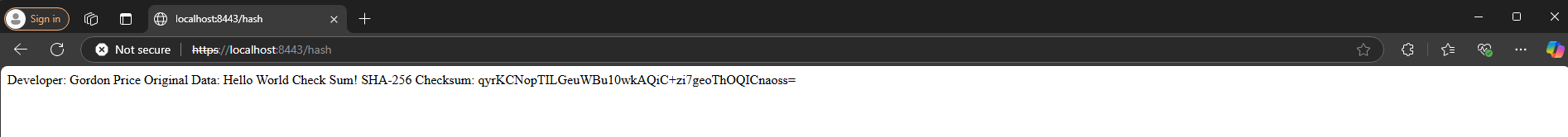
AI-generated content may be incorrect.

## *Deploy Cipher*

The SHA-256 checksum functionality was implemented in the ChecksumController.java file. The method generateSHA256() accepts user-provided data, processes it through the SHA-256 algorithm, and returns the Base64-encoded checksum. The updated controller allows checksum verification through an HTTPS request.*A screen shot of a computer program

AI-generated content may be incorrect.*

## *Secure Communications*



The application properties were refactored to switch from HTTP to HTTPS. After compiling and running the refactored code, I tested the implementation by navigating to:

https://localhost:8443/hash?data=HelloWorld

The result successfully displayed the checksum, verifying that the cryptographic hash algorithm is functional and deployed correctly. So, by switching from HTTP to HTTPS, the app now encrypts all data transmissions, preventing unauthorized access or mid-ware attacks.

## *Secondary Testing*

To verify code security and dependency vulnerabilities, I ran OWASP Dependency-Check. The tool detected 87 vulnerabilities across 35 dependencies, including Spring Boot, Tomcat, and Hibernate Validator with critical, high, and medium security risks. The results suggest that upgrading to newer dependency versions would enhance security.

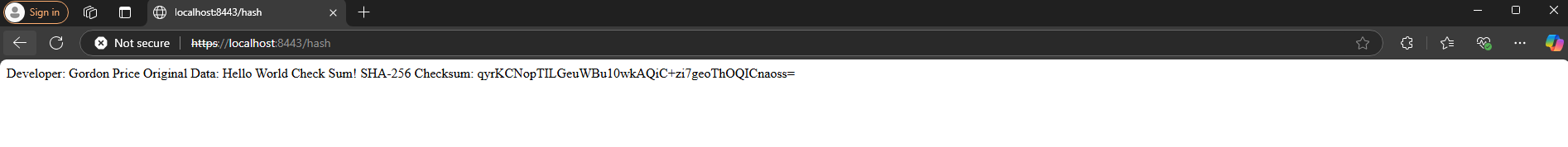
A black and white screen with white text

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

## *Functional Testing*

Functional testing was performed by manually reviewing the refactored code for syntactical, logical, and security vulnerabilities. The application was executed to verify proper operation. For example, accessing the /checksum endpoint returned the expected output:

A black background with white text

AI-generated content may be incorrect.

## *Summary*

During this project, I refactored significant portions of software to offer a more secure interface for calculating SHA-256 checksums. So, the refactoring took into consideration the following security aspects:

* **Input Handling and Error Management:**
  + The new /checksum endpoint accepts a query parameter (with default) and safely computes a checksum via Java's SHA-256 implementation.
  + Errors such as the absence of an algorithm are caught by exception handling (e.g., NoSuchAlgorithmException) and result in an appropriate error message.
* **Static and Functional Testing:**
  + As part of the vulnerability assessment process (see the process flow diagram in the Supporting Materials), I ran the OWASP dependency-check tool to scan all dependencies.
  + Static analysis confirmed that no new vulnerabilities were introduced.
  + Functional testing, such as accessing the /checksum endpoint, verified that the application is free from runtime errors.
* **Adding Layers of Security:**
  + I introduced additional security measures by:
    - Using a secure cryptographic hash function (SHA-256) with Base64 encoding for checksum verification.
    - Leveraging Spring's parameter is binding to sanitize and validate user input.
    - Isolating the cryptographic logic in a separate method to simplify auditing and maintenance.
* **Compliance with Best Practices:**
  + I followed industry best-practice secure coding techniques, conducted iterative testing, and documented each of the steps.
  + This helped make sure that the overall security posture of the application was maintained.

## *The Industry Standard Best Practices*

To maintain the application’s security and resilience against evolving threats, I enforced industry-standard best practices during the refactoring phase.

**Treatment of Secure Coding Guidelines:**

* I used secure coding methodologies to ensure that new code, such as the /checksum endpoint, follows secure coding principles.
* This approach helped avoid common weaknesses such as injection flaws and poor error handling.

**Strong Input Validation and Exception Handling:**

* The application automatically cleans input using Spring's parameter binding, reducing the attack surface for potential malicious data misuse.
* Comprehensive exception handling ensures that errors are properly managed without exposing sensitive information.

**Employment of Trusted Cryptographic Algorithms:**

* The SHA-256 algorithm, which is combined with Base64 encoding is said to be one of the best industry practices for ensuring data integrity.
* While AES-256 is typically used to encrypt large archive files, SHA-256 checksum verification more than meets the security requirement in this case.

**Constant Vulnerability Analysis and Dependency Management:**

* Habitual application of tools like OWASP Dependency-Check guarantees that third-party library flaws are quickly identified and fixed.
* Proactive monitoring is the method of keeping the codebase secure and removing risks before they have the chance to become threats.

**Application of Separation of Concerns:**

* The process of refactoring encapsulates security-sensitive functionality, such as checksum creation, in dedicated methods.
* This modular architecture allows for simpler testing, auditing, and future maintenance, in addition to decreasing the possibility of security breaches.

**Value to the Organization:**

* The implementation of these best practices protects the application against security threats as well as aligns with the company's overall risk management approach.
* A secure codebase minimizes the risk of data breaches, ensuring continued customer trust and preventing financial and reputational loss.

**REFERENCES**

Kiteworks. (2024). AES-256 encryption: Securing your data with symmetric key cryptography. Retrieved from https://www.kiteworks.com/cybersecurity-risk-management/aes-256-encryption-securing-your-data-with-symmetric-key-cryptography/