****

# 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** | **02/23/2025** | **Michael Langille** |  |

## 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

Michael Langille

## Algorithm Cipher

**Recommended Algorithm Cipher: SHA-512**

SHA-512 is a cryptographic hash function developed by the National Security Agency

(NSA) and standardized by NIST. It is used for data integrity verification, digital signatures,

password hashing, and blockchain applications. Unlike encryption, which can be reversed with a

decryption key, hashing is one-way and irreversible.

SHA-512 produces a 512-bit hash output, making it significantly more resistant to

collision and preimage attacks compared to smaller hash sizes like SHA-256. The internal

structure of SHA-512 processes input data in 1024-bit blocks, making it more efficient on 64-bit

architectures. Other versions include SHA-384, SHA-256, and SHA-224, each offering different

trade-offs in terms of security and performance. SHA-512 is commonly used for digital

signatures, certificates, password hashing, blockchain and cryptocurrency applications.

When used for password hashing, SHA-512 should always be combined with a

cryptographic salt (a random value added before hashing). PBKDF2, bcrypt, and Argon2 are

recommended password hashing instead of raw SHA-512 to prevent rainbow table attacks.

SHA-512 can also be used in HMAC (Hash-based Message Authentication Code) for

authentication in secure communications.

The history and current state of encryption algorithms include MD5 (Message Digest 5)

and SHA-1 (SSL/TLS). Security protocols widely adopted SHA-256 and SHA-512 due to their

stronger security against known attacks. Future considerations include quantum computing,

where SHA-512 is more resistant to Grover's algorithm than SHA-256. SHA-512 is widely used

for password security, message integrity, and futureproofing. Best practices include using SHA-

512 with salt and key stretching functions for password security, HMAC-SHA-512 for message

integrity, and considering SHA-3 for new applications with long-term post-quantum security

concerns.

## Certificate Generation

A screenshot of a computer

AI-generated content may be incorrect.

## Deploy Cipher

## 

## Secure Communications

Showing the HTTPS is working but that my Cert isn’t official because it’s self-signed.

A screenshot of a computer

AI-generated content may be incorrect.

## Secondary Testing

**Why I Added ChecksumController.java Instead of Refactoring SslServerApplication.java**

When designing and structuring a secure and maintainable Spring Boot application, it's

essential to follow the Separation of Concerns (SoC) principle. Instead of modifying the

SslServerApplication.java file, we created ChecksumController.java to handle the checksum

functionality. Here’s why:

**1. Separation of Concerns (SoC)**

* **SslServerApplication.java** serves as the entry point of the Spring Boot application. Its

role is to initialize and run the server.

* **ChecksumController.java** is responsible for handling API requests related to checksum

generation.

* Keeping these separate ensures that each class has a single responsibility.

**Best Practice:** The main application class should not contain business logic or API routes.

**2. Maintainability & Scalability**

* If we placed the /hash endpoint inside SslServerApplication.java, future modifications

would be harder.

* With ChecksumController.java, adding new security features or additional

API endpoints become simpler.

* If the company wants to add more controllers (e.g.,

AuthenticationController, LoggingController), it can be done without

modifying the main application file.

**Example:** In a large application, the checksum logic might evolve, requiring additional security

layers (e.g., HMAC authentication). Keeping it separate allows focused modifications without

affecting core server logic.

**3. Security & API Structure**

* Controllers should be isolated to prevent security vulnerabilities.
* SslServerApplication.java is kept minimal, reducing attack surface risks.
* Placing API logic in a dedicated RestController enforces Spring Security best practices.

**Example:** If additional security checks (e.g., JWT authentication) were needed for /hash, we

could add security filters to ChecksumController.java without affecting the core application.

**4. Adherence to Spring Boot MVC Architecture**

Spring Boot follows Model-View-Controller (MVC):

* **Controllers** (ChecksumController.java) handle requests.
* **Services** (if needed) process business logic.
* **Application class** (SslServerApplication.java) initializes the Spring Boot framework.
* **Best Practice:** APIs should always be placed in dedicated controller classes,

not the main() class.

Summary

* SslServerApplication.java is the entry point, NOT an API handler.
* Keeping API logic in ChecksumController.java improves maintainability & security
* Follows Spring Boot’s MVC architecture & Separation of Concerns (SoC).
* It is easier to extend & add new features without modifying core server logic.

This approach ensures a cleaner, more secure, and scalable application!

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot 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.

## Summary

**Security Refactoring & Compliance with Security Testing Protocols**

The refactored code enhances security by following best practices in software

development, complying with security testing protocols, and addressing vulnerabilities

identified in the Vulnerability Assessment Process Flow Diagram. Below is a detailed discussion

of how the code has been improved.

**1. Vulnerability Assessment Process & Addressed Security Areas**

The Vulnerability Assessment Process Flow Diagram outlines key stages in identifying and

mitigating security risks. The following areas were specifically addressed in the refactoring:

|  |  |
| --- | --- |
| Security Area | How It Was Addressed |
| Asset Identification | Secured the SSL certificate and enforced HTTPS for all communications. |
| Threat Identification | Conducted **OWASP Dependency-Check** to identify and update vulnerable dependencies. |
| Risk Assessment | Assessed and mitigated **risks related to outdated libraries**, enforcing the latest security patches. |
| Remediation | Applied **SHA-512 hashing for data integrity** and **Spring Boot security best practices** to harden the application. |

**Key Fixes:**

* Implemented OWASP Dependency-Check to scan and remove vulnerable dependencies.
* Secured sensitive information in application.properties (e.g., password protection).
* Ensured data integrity using SHA-512 hashing for checksum verification.

**2. Security Layers & Business Value**

**Layered Security Approach**

To secure the **SSL Server Application**, multiple layers of security were added:

|  |  |
| --- | --- |
| Security Layer | Implementation Details |
| Transport Security | Enforced **HTTPS with SSL/TLS** (via keystore.p12). |
| Authentication & Integrity | Implemented **SHA-512 hashing** to ensure data integrity. |
| Dependency Security | Used **OWASP Dependency-Check** to detect and mitigate vulnerable libraries. |
| Secure Configuration | Hardened application.properties (avoided hardcoded credentials, used classpath storage). |
| Access Control | Limited API endpoints to **only necessary routes** (/hash). |

**Business Value of Security**

* Protects sensitive company data from MITM (Man-in-the-Middle) attacks.
* Builds customer trust by ensuring data integrity and compliance with security standards.
* Reduces financial & reputational risks by preventing breaches due to outdated

libraries.

* **Enhances regulatory compliance** (e.g., **GDPR, HIPAA, PCI DSS**) by enforcing

HTTPS & encryption.

**3. Best Practices for Maintaining Security**

To maintain the current security posture, the customer should adopt ongoing security best

practices, including:

|  |  |
| --- | --- |
| Best Practice | Description |
| Regular Security Scanning | Continuously run **OWASP Dependency-Check** to detect new vulnerabilities. |
| Enforce HTTPS Always | Keep SSL/TLS certificates up to date and **disable HTTP access** in production. |
| Use Environment Variables for Secrets | Move sensitive values (e.g., keystore passwords) to **environment variables** instead of hardcoding them in application.properties. |
| Monitor Logs & Alerts | Enable **Spring Boot Actuator** to monitor security events & detect anomalies. |
| Apply Dependency Updates | Regularly update dependencies via mvn versions:display-dependency-updates. |
| Penetration Testing | Conduct regular **penetration testing** to identify new threats. |

**Final Summary**

* The refactored code enforces HTTPS & SHA-512 hashing, preventing insecure data

transmission.

* OWASP Dependency-Check ensures all dependencies are secure & up to date.
* Following the best practices ensures the software remains secure & compliant over time.

By implementing these security enhancements, the company's data remains protected,

customer trust is reinforced, and long-term business risks are minimized.

## Industry Standard Best Practices for Secure Software Development

## When developing a secure software application, it is essential to follow industry-standard

## best practices to mitigate risks and ensure compliance with security protocols. Below are key

## best practices that align with secure coding principles, regulatory compliance, and cybersecurity

## frameworks.

**Secure Software Development Lifecycle (SDLC)**

**Implement Security from the Start** – Security should be integrated at every stage of the

Software Development Lifecycle (SDLC):

* **Requirements Phase** → Define security policies and compliance needs (e.g., GDPR,

HIPAA).

* **Design Phase** → Implement security-focused architecture (e.g., Zero Trust Model).
* **Development Phase** → Follow secure coding guidelines (e.g., OWASP Top 10).
* **Testing Phase** → Perform static (SAST) and dynamic (DAST) security testing.
* **Deployment & Maintenance** → Apply regular patches, monitor for vulnerabilities, and

log security events.

**Industry Standards: NIST Secure SDLC (SP 800-64), OWASP Secure SDLC, ISO/IEC 27001**

**Secure Coding Practices**

**Follow OWASP Secure Coding Guidelines** – Protect applications from common vulnerabilities:

**OWASP Top 10 Vulnerabilities & Fixes**:

|  |  |
| --- | --- |
| Threat | Mitigation Strategy |
| Injection (SQL, XSS, etc.) | Use parameterized queries (e.g., **Spring JPA, Hibernate**) & sanitize inputs. |
| Broken Authentication | Enforce **OAuth2, JWT, Multi-Factor Authentication (MFA).** |
| Sensitive Data Exposure | Store passwords using **BCrypt**, encrypt sensitive data, and enforce **TLS/SSL**. |
| Security Misconfiguration | Use **security headers**, disable default configurations, and **apply least privilege access.** |
| Outdated Dependencies | Run **OWASP Dependency-Check**, use the latest libraries. |

**Industry Standards:** **OWASP Secure Coding Practices, NIST SP 800-53**

**Secure API Development**

**Implement Strong API Security Controls** – APIs are common attack vectors, so securing them

is critical.

**Best Practices for Secure REST APIs**:

* Use **HTTPS (TLS 1.2/1.3)** for encrypted communication.
* Implement **OAuth2 or JWT (JSON Web Tokens)** for authentication.
* Validate and sanitize all inputs to prevent **injection attacks**.
* Implement **Rate Limiting & Throttling** to prevent DDoS attacks.
* Enable **CORS (Cross-Origin Resource Sharing) restrictions** to prevent unauthorized

access.

**Industry Standards:** **NIST API Security Guidelines, OWASP API Security Top 10**

**Enforcing Transport Layer Security (TLS)**

**Always Use Secure Communication Channels**

* Enforce TLS 1.2+ (disable TLS 1.0 & 1.1, which are vulnerable).
* Use strong cipher suites and enforce HSTS (HTTP Strict Transport Security).
* Disable self-signed certificates in production – use CA-signed SSL certificates.
* Redirect HTTP traffic to HTTPS to prevent MITM (Man-in-the-Middle) attacks.

**Industry Standards:** **TLS Best Practices (RFC 7525), PCI DSS Encryption Standards**

**Dependency & Vulnerability Management**

**Regularly Scan for Vulnerabilities & Patch**

* Use OWASP Dependency-Check to find vulnerabilities in third-party dependencies.
* Run mvn versions:display-dependency-updates to check for outdated dependencies.
* Monitor CVEs (Common Vulnerabilities and Exposures) and apply patches promptly.

**Industry Standards:** **OWASP Dependency-Check, NIST National Vulnerability Database (NVD)**

**Secure Configuration & Hardening**

**Harden the Application & Server Configurations**

* Use environment variables to store secrets instead of hardcoding them in

application.properties.

* Restrict unnecessary services and open ports.
* Implement logging & monitoring (use ELK Stack, Splunk, or AWS CloudWatch).
* Apply least privilege access control for all services and APIs.

**Industry Standards:** **CIS Security Benchmarks, NIST 800-123 Secure Server Configurations**

**Continuous Security Testing & Monitoring**

**Conduct Regular Security Audits & Penetration Testing**

* Automate security testing using **SAST (Static Application Security Testing)** tools like

**SonarQube**.

* Perform **DAST (Dynamic Application Security Testing)** using **OWASP ZAP, Burp**

**Suite**.

* Use **SIEM (Security Information & Event Management)** tools for **real-time threat**

**monitoring**.

**Industry Standards:** **NIST Cybersecurity Framework (CSF), ISO 27001 Security Audits**

**Final Takeaways: Industry Standard Best Practices Summary**

|  |  |
| --- | --- |
| Category | Best Practices |
| Secure SDLC | Follow **OWASP, NIST, and ISO 27001** guidelines. |
| Secure Coding | Protect against **OWASP Top 10 threats** (e.g., XSS, SQL Injection). |
| API Security | Use **OAuth2, JWT, HTTPS, input validation, and rate limiting.** |
| TLS Security | Enforce **TLS 1.2+, HSTS, and strong cipher suites.** |
| Dependency Management | Run **OWASP Dependency-Check, update dependencies regularly**. |
| Configuration Hardening | Use **environment variables, disable unused ports, enable monitoring**. |
| Continuous Security Testing | Perform **SAST, DAST, and security audits** regularly. |

**By implementing these best practices, we ensure robust security, reduce vulnerabilities, and**

**protect both customer data and company assets.**

**Citations**

Abid, C. (2021). *Explainable, Security-Aware and Dependency-Aware Framework for Intelligent Software Refactoring* (Doctoral dissertation).

Bemida, P. J. F., Sison, A. M., & Medina, R. P. (2021, November). Modified SHA-512 Algorithm for Secured Password Hashing. In *2021 Innovations in Power and Advanced Computing Technologies (i-PACT)* (pp. 1-9). IEEE.

Kumar, S. H. B. I., Sampaio, L. R., Martin, A., Brito, A., & Fetzer, C. (2024, October). A Comprehensive Study on the Impact of Vulnerable Dependencies on Open-Source Software. In *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)* (pp. 96-107). IEEE.

Mythily, M., Raj, A. S. A., & Joseph, I. T. (2022, July). An analysis of the significance of spring boot in the market. In *2022 international conference on inventive computation technologies (ICICT)* (pp. 1277-1281). IEEE.

Räisänen, E., & Väyrynen, A. M. SECURE SOFTWARE DESIGN AND DEVELOPMENT–.

Taherdoost, H. (2022). Understanding cybersecurity frameworks and information security standards—a review and comprehensive overview. *Electronics*, *11*(14), 2181.