****

# 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** | **3/3/2024** | **Shari Storlie** |  |

## 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

Shari Storlie

## Algorithm Cipher

Adhering to best practices guidelines from expert bodies like NIST for cryptographic controls can help maximize protection against evolving attack capabilities and discoveries.

Consider using Cipher Algorithm AES-256 to provide confidentiality, integrity, and authenticity assurances. Best practice guidelines is to ensure proper random key generation using cryptographically secure pseudo-random number generators to avoid keys that are guessable. Use algorithms, modes, padding schemes, and libraries considered ‘best practice to avoid novel or custom constructions that may have undiscovered flaws. Ensure key management procedures provide security protections for generation, storage, distribution, and destruction according to principle of least privilege. Limit access to keys. Use key rotation policies to periodically change out encryption keys to limit exposure, especially for data-at-rest encryption. Consider implementing defense-in-depth with multiple controls like transit encryption (TLS) plus field/database-level encryption schemes to increase workload for attackers.

Encrypted data uses the same key to encrypt and decrypt. If the key is compromised the data will be at risk. There is the possibility of quantum computers developing that can break SHA256 encryption. Possible flaws or bugs in the implementation code or software libraries could weaken security.

SHA is approved by the National Institute of Standards and Technology (NIST) for protecting sensitive unclassified data as well as most classified information. It meets regulatory compliance needs.

SHA256 would be used to encrypt data for backup, information transfer, and in use.

SHA256 provides extremely robust security protection and is considered essentially unbreakable by practical cryptanalytic attacks with current and foreseeable technology. A 256-bit key means there are 2^256 (1.1 x 10^77) potential keys, making brute force attacks infeasible.

Cryptographic hash functions like SHA256 complement and enhance cipher algorithm implementations in various ways. They are crucial to "key management" - derivation, storage, use and destruction of keys. By introducing computational difficulty, they make attacks more challenging. Hashes support data authentication assurances as well. Their role extends beyond just hashing passwords.

Proper use of randomness and choice between symmetric and asymmetric approaches based on system architecture, key distribution models, performance needs and scalability determine the optimal selection of cryptosystems when implementing encryption to protect sensitive data.

**History**

Encryption has existed for millennia, but early ciphers were basic substitution or transposition ciphers using limited keyspaces vulnerable to frequently analysis.

The mid-20th century saw the rise of rotor machines like the German Enigma, demonstrating electromechanical encryption but again susceptible to cryptoanalysis. The Allies' breaking of Enigma played a pivotal role in World War 2.

The 1970s brought the Data Encryption Standard (DES) - the first publicly accessible cipher. DES had some cryptanalytic weaknesses from its small 56-bit key.

RSA public-key encryption and Diffie-Hellman key exchange in the 1970s also revolutionized encryption by enabling asymmetric cryptography.

**Current State**

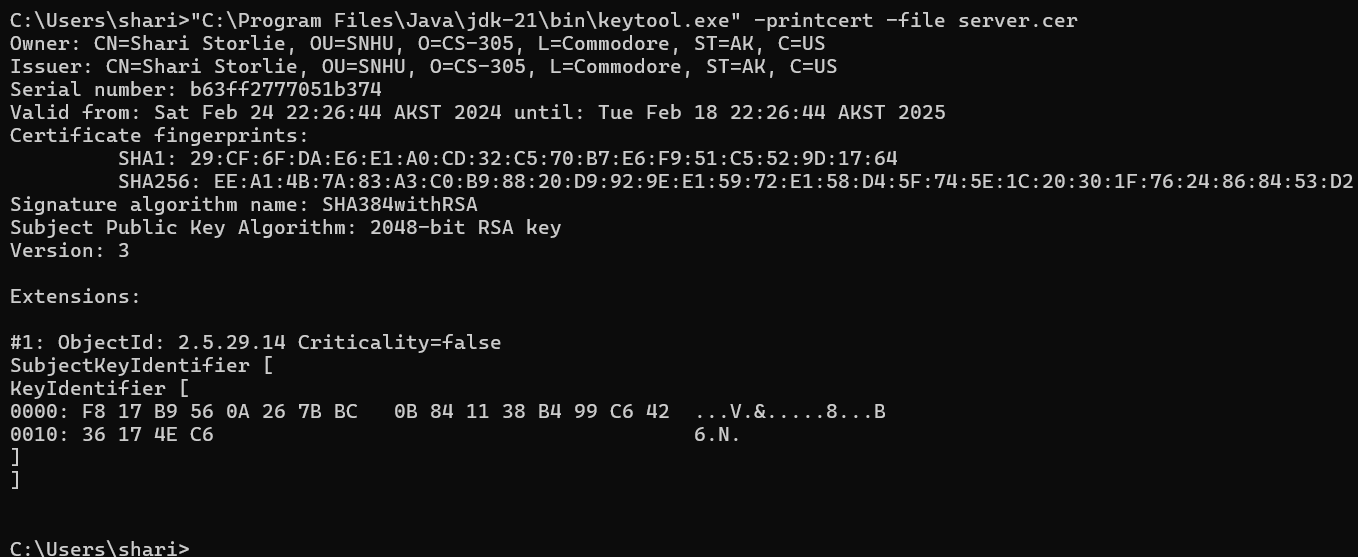
The 1990s brought the gold standard SHA - a symmetric cipher with 128/192/256 bit keys resistant to all known attacks. SHA is used globally today for protecting top secrets.

Elliptic curve cryptography (ECC) gained adoption in the 2000s offering equivalent security to RSA but via much smaller keys, improving efficiency.

Today's ubiquitous Transport Layer Security (TLS) protocol relies on underlying symmetric encryption (typically SHA) for bulk data transfer in a secure session established via asymmetric cryptography (RSA or ECC) for communicating parties to safely exchange a secret. This 'hybrid cryptosystem' combining symmetric and asymmetric cryptography powers most modern secure communications from HTTPS websites to VPNs to messaging apps.

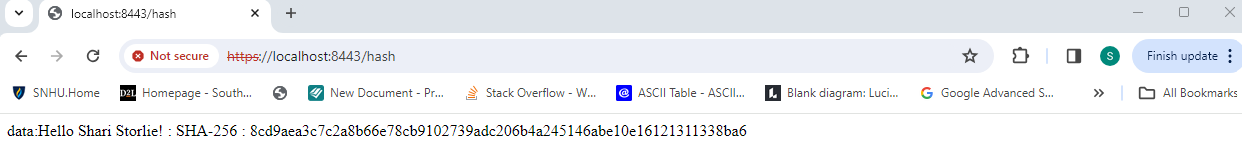
## Certificate Generation

Insert a screenshot below of the CER file.



## Deploy Cipher

Insert a screenshot below of the checksum verification.



## Secure Communications

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

A screenshot of a computer

Description automatically generated

## Secondary Testing

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

A screenshot of a computer code

Description automatically generated

A screenshot of a computer

Description automatically generated

## Functional Testing

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

A screenshot of a computer

Description automatically generated

## Summary

Based on the scenario of implementing an expressive command input function using Spring Expression Language (SpEL), the areas of the Vulnerability Assessment Process Flow that best apply are:

- Input Validation

SpEL applications have a vulnerability called Injection attacks. Injection attacks happen when unauthorized code is ‘injected’ into a program and is interpreted as part of a query or command. Injection attacks can be prevented by appropriate input validation. An expressive command function can be used through SpEL user-controlled input functionality.

We addressed the possibility of injection attacks by keeping sensitive information within the methods, instead of naming the methods with the sensitive data.

- APIs

SpEL uses APIs which can be a source of security breach. To combat the possibility of API security issues regular system updates can keep a program safer with the best security in place.

We addressed this by implementing strong cypher encryption and including a RestController for a hash RESTful stop. The ServerController class works to match the problems presented by the vulnerability assessment diagram.

- Secure Coding Practices/Patterns

SpEL programs will be vulnerable to security issues if secure coding practices aren’t used. There are specific guidelines that can be referenced and used. SpEL provides a safe expression evaluation support feature to assist with secure coding practices/patterns.

Using the coding method of keeping sensitive information covered in the code and layering security measures throughout the code by encryption, using REST strategies, using keystore certification, and maven security tests.

## Industry Standard Best Practices

We used industry standard best practices by using the coding method of keeping sensitive information covered in the code and layering security measures throughout the code by encryption, using REST strategies, using keystore certification, and maven security tests.

As developers, we have an important role to play in building secure software and addressing security concerns:

1. Following secure coding best practices - Things like input validation, using tested libraries, encryption, access controls, etc. These help create secure code from the start.

2. Threat modeling - Identifying potential threats and vulnerabilities in the system/software design and architecture. This allows addressing them proactively.

3. Collaborating with security teams - Working with security engineers and testers to identify and fix vulnerabilities during development. Tracking security bugs and fixing them quickly.

4. Automating security testing - Setting up tools for static analysis, dynamic analysis, fuzzing, etc. directly in the CI/CD pipelines to catch issues early.

5. Adopting a "security-first mindset" - Keeping security in mind at each stage rather than an afterthought. Advocating for designing secure systems right from conception through deployment.

6. Continuous learning - Keeping up-to-date on latest security issues and threats to languages/frameworks I use. Attending security training.

Essentially, We have an obligation to develop secure, high quality code that prevents security lapses from the get-go. This involves both technical security skills and a security-oriented mindset. Collaboration with security teams and management is also key. The earlier we catch and fix security issues, the better.