****

# 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** | **10/16/25** | **Nicholas Deniz** | **Completing Project 2** |

## 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

Nicholas Deniz

## Algorithm Cipher

1. The algorithm that I would recommend for the checksum feature is SHA-256 which is a cryptographic hash. A hash makes an input into a fixed length 256-bit digest. It’s just one way and is collision resistant. Any tiny slight input changes create very different digests. Specifically for this project’s checksum verification I would be checking its integrity, making sure the data wasn’t altered during transport or storage. So, SHA-256 would be suited well for this as integrity checks are the focus instead of confidentiality.
2. SHA-256 outputs 256 bits which would be 64 hex characters. This is good as a larger digest size makes collision attempts and brute force attempts more difficult to accomplish. Older and smaller hashes like SHA-1 wouldn’t be able to put up much resistance. It is important not to use obsolete algorithms, like SHA-1 or MD5, due to their known exploits. It is better to use modern and thoroughly tested hashes like SHA-256 for integrity checks. Hash functions offer specific security that can help like its avalanche behavior. Also, preimage resistance makes it more difficult to recover the original input. Second preimage resistance makes it difficult to find a different input with the same digest. Encryption through symmetric or asymmetric isn’t needed here as that mainly protects confidentiality not integrity checks. Keeping to hash functions only fulfil this purpose of keeping the focus on integrity.
3. Randomness is quite important in this venture. For checksum display endpoint it doesn’t need keys but a unique value. A unique value in the input string shows that checksum changes when the input changes. Symmetric ciphers like AES and asymmetric schemes like RSA are useful in other systems for TLS handshakes and confidentiality. Symmetric encryption uses one shared key to do both encryption and decryption to protect confidentiality. Asymmetric encryption uses a public/private pair of keys for certificates and signatures. The checksum feature is still best served by a keyless hash. Keys are part of this project when the TLS certificate bids identity for HTTPS which uses asymmetric during the handshake.
4. Modern practices use TLS 1.2+ for secure transportation and the SHA-2 family hashes like SHA 256 for signatures and integrity checks. SHA-256 is part of the SHA-2 family and is used for TLS signatures. Hashes like MD5 and SHA-1 are obsolete and should be avoided due to their know exploits. In 2017 it was officially decided that SHA-1 should be obsolete due to a collision being demonstrated. AES-GCM is mainly used for confidentiality and RSA/EC is mainly used for key exchange and identity. To replace SHA-1 NIST standardized the SHA-2 family which became the mainstream and recommended hashes. In 2015 NIST standardized SHA-3 which uses a sponge construction, but many systems still use SHA-256 due to its broad hardware and software support.

## Certificate Generation

Insert a screenshot below of the CER file.

A computer screen shot of a program

AI-generated content may be incorrect.



A screenshot of a certificate

AI-generated content may be incorrect.

## Deploy Cipher

Insert a screenshot below of the checksum verification.

A screenshot of a computer

AI-generated content may be incorrect.

## Secure Communications

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

A screenshot of a computer

AI-generated content may be incorrect.

A computer screen shot of a black screen

AI-generated content may be incorrect.

## Secondary Testing

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

A computer screen shot of a program

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 computer screen shot of a program

AI-generated content may be incorrect.

A computer screen shot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

## Summary

What was created and verified is the Spring Boot app exposing GET <https://localhost:8443/hash> over HTTPS. It uses PKCS12 key store with an alias called artemis. The endpoints returned name, unique data (name and timestamp), and SHA-256 of the unique data. Functional testing confirmed that it was delivered over TLS, browser shows <https://localhost:8443/hash> and that it printed the three fields. I generated a self-signed certificate with the alias artemis and put the private key in a keystore keystore.p12 in resources. In application.properties I configured it with the port being 8443, the alias is artemis, the store was classpath:keystore.p12, type was PKCS12, and the password was FinaNcialArteMis48&. It was deemed not secure in the browser but that is due to the certificate being self-signed and not trusted by the OS, it was still encrypted. Error handling had a catch, Exception e, that returns an exception message to the client that can leak internal information if not properly generic. Risks were found in third party libraries. It was discovered through the OWASP Dependency Check report and rated as HIGH and CRITICAL. These include spring-webmvc/spring-core 5.2x, tom-cat-embed-core/tomcat-embed-websocket 9.0.30, Jackson-databind 2.10.x, hibernate-validator 6.0x, and logback-core/logback-classic 1.2x.

I have compiled a Mitigation plan to confront errors that have cropped up. Dependencies should be dealt with by upgrading Spring Boot to something supported like 3.2.x/3.3x on JDK 17+. Boot's BOM would put Tomcat, Jackson, Spring Framework, Logback, Hibernate Validator to their versions that are patched. Secrets should be taken out of the code like in application.properties, the keystore password should be provided by a secure store or the environment. Harden HTTPS/TLS by disabling HTTP by using only server.port=8443. Use only modern protocols and ciphers. User inputs should be validated before it is used. Return JSON and let the client render it. For error handling only generic messages should be sent to client like “Error generating checksum”. For logging, log full stack traces on the server side only.

## Industry Standard Best Practices

On the cryptography side of things, I used SHA-256 which is good for checksums as it is well designed for integrity checks. HMAC should be used for tampering detection. Store keys outside of source control should rotate when exposed. For APIs/Client-Server it should only go over HTTPS. JSON response should be preferred as HTML has injection risks. For input validation, inputs should be validated using a whitelist approach which is length, charset, and format. Output should be encoded specifically to the sink like JavaScript or the URL. For error handling, catch, exception, shouldn’t be used and should catch specific exception and shouldn’t have HTTP responses. For encapsulation, keystore files and configuration should be under resources with file permissions. Class and method visibility should be lowered to only show as little as possible. For dependencies it should be managed by BOM, which is the Spring Boot Parent, to keep things organized. OWASP Dependency Check should be run with every building and be understood. Testing should be done, to check functionality verify GET /hash returns the three expected fields through the <https://localhost:8443>. HSTS should be enforced by disabling weaker ciphers and protocols.

References:

DevGlan. (n.d.). *AES encryption and decryption online*. Devglan. <https://www.devglan.com/online-tools/aes-encryption-decryption>

Dworkin, M. (2007, November). *Galois/counter mode (GCM) and GMAC*. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. <https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-38d.pdf>

Fleming, M. (2012, July 19). *Five Minute Tutorial : Eclipse Keytool Plugin*. YouTube. <https://www.youtube.com/watch?v=IaXE2FUENF>

GeeksforGeeks. (2025, August 8). *Advanced encryption standard (AES)*. <https://www.geeksforgeeks.org/computer-networks/advanced-encryption-standard-aes/>

*How to create a self signed certificate using Java Keytool*. SSL Certificate Comparison and Reviews. (n.d.). <https://www.sslshopper.com/article-how-to-create-a-self-signed-certificate-using-java-keytool.html>

Jim Manico, A. D. (2014, September). *Iron-clad Java*. O’Reilly Online Learning. <https://learning.oreilly.com/library/view/iron-clad-java/9780071835886/>

*Keytool - key and certificate management tool*. keytool-Key and Certificate Management Tool. (n.d.). <https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html>

Kidd, C. (2024, October 18). *Data Encryption Methods & Types: A beginner’s guide*. Splunk. <https://www.splunk.com/en_us/blog/learn/data-encryption-methods-types.html>

Oracle. (2017). Java security standard algorithm names. <https://docs.oracle.com/javase/9/docs/specs/security/standard-names.html#cipher-algorithm-names>

Owasp dependency check: How does it work? | mend. (n.d.). <https://www.mend.io/resources/blog/owasp-dependency-check/>

*Usage – dependency-check-maven*. dependency. (n.d.-b). <https://dependency-check.github.io/DependencyCheck/dependency-check-maven/index.html>

Yakkala, P. (2024, August 13). *Understanding AES encryption and AES/GCM mode: An in-depth exploration using Java*. Medium. <https://medium.com/@pravallikayakkala123/understanding-aes-encryption-and-aes-gcm-mode-an-in-depth-exploration-using-java-e03be85a3faa>