****

# 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** | **8-18-24** | **Shannon Musgrave** |  |

## 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

Shannon Musgrave

## Algorithm Cipher

SHA-384

The SHA-384 hashing algorithm was created by the NSA (National Security Agency) as part of a family of algorithms with varying digests. The number attached after the dash signals the bit length of the cipher. This gives the SHA-384 algorithm a 48-byte return offering many combinations, approximately 3.94 \* 10^115. This is critical to avoid collisions, which is when two different hashed items returned the exact same hash.

Encryption uses random numbers to generate variables used to encrypt data. By using randomly generated numbers, this makes it difficult for threat actors to predict the output and crack the encryption. Encryption is broadly divided into two groups, symmetric and non-symmetric algorithms. Symmetric algorithms use the same key to decrypt the information. This means that the key must be kept private and cannot be allowed to become public knowledge. Non-symmetric algorithms use a public and private key that must be used together to decrypt the information. This makes it possible to share the public key, which can be used to encrypt information. In turn, the information can only be decrypted using the private key as well. For third parties, the encrypted information and the public key are useless, since the private key is not known.

The history of encryption and ciphers predates computers, a cypher is a system that scrambles a message in a way that can only be decoded by understanding the cypher. For instance, the normal alphabet could be translated with a reversed alphabet, where ‘a’ maps to ‘z’ and so on. These crude cyphers are typically easily defeated by modern computing, this led to a field of mathematics devoted to inventing cyphers that cannot be easily defeated by even the fastest computers. As technology advances, the SHA 2 family of cyphers will become defeatable, and even more robust cyphers will need to be invented.

## Certificate Generation

Insert a screenshot below of the CER file.

To get the browser to accept my certificate, I had to use the following command:

“keytool -genkey -keyalg RSA -alias p2selfsigned -keystore keystore.jks -storepass project2password -validity 360 -keysize 2048 **-dname "CN=127.0.0.1" -ext "SAN=IP:127.0.0.1"”**

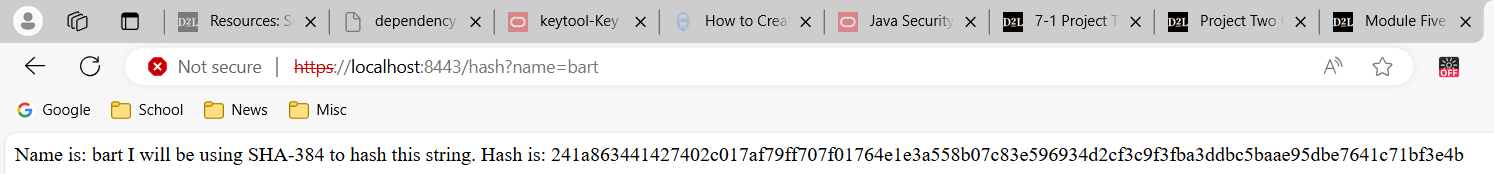
A screenshot of a computer

Description automatically generated

## Deploy Cipher

Insert a screenshot below of the checksum verification.

The SHA-384 returns a 48-byte array, since this array is converted to hexadecimal (2 characters per byte), it is printed as a 96-character string. The hash method can be passed a variable of name so that different hashes can be tested.



## Secure Communications

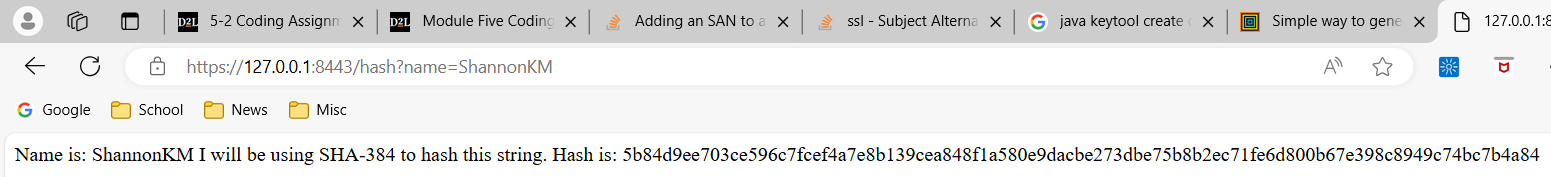
Insert a screenshot below of the web browser that shows a secure webpage.

Instance of using HTTP after server is configured for HTTPS.

A screenshot of a computer

Description automatically generated

Instance of using HTTPS.



## Secondary Testing

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

Server compiled and is running with no errors.

A screenshot of a computer

Description automatically generated

The dependency check ran showing no vulnerabilities listed in project after updating dependencies.

A screenshot of a computer

Description automatically generated

## Functional Testing

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

Created new class to handle Requests.

A screenshot of a computer program

Description automatically generated

No changes to original entry point.

A computer screen shot of a program

Description automatically generated

Following edits to the application properties were made. The keystore was copied to the root of the project (where the POM.xml is located.)

A screenshot of a computer

Description automatically generated

## Summary

The areas that were improved were the APIs and Cryptography, the project was improved to use HTTPS or HTTP secure communications. If a request was made using HTTP, the request would be rejected. In addition to a more secure Restful service, a checksum was implemented that will compute a SHA-384 hash for any string that is sent to the client. If the data is compromised or corrupted before being returned to the client, it can be discovered since the client can compute the hash and compare it to the known hash digest. Next, a server certificate was created that can be used for testing and development. Since this certificate is self-signed, it will need to be upgraded to a certificate signed by a CA (certificate authority) when the server goes live. Finally, each dependency has been checked for known vulnerabilities using the newest version of dependency-check (10.0.3). Vulnerabilities were eliminated by moving to newer versions of the Spring webserver.

## Industry Standard Best Practices

Best practices used in this project include using mature dependencies that have been rigorously tested and improved. This includes using the SHA-384 hashing algorithm for the checksum procedure. Creating custom algorithms is not advisable and could even be illegal in some fields such as financing. Another example of using mature dependencies is using Java’s keytool to create and store certificates and keys. By using polished tools, many problems can be avoided since these tools have been improved over time. Another best practice is using HTTPS for all communications. By moving to HTTPS, the possibility of data being stolen or intercepted is reduced since data is not transmitted as plain text. Another applied best practice is using static testing and code reviews to spot weaknesses and known vulnerabilities in not only the custom code but also dependencies. These practices should not be used as a one-time fix but should be used periodically to ensure that any new vulnerabilities can be mitigated upon discovery.