****

# 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 5](#_Toc102040759)

[3. Deploy Cipher 6](#_Toc102040760)

[4. Secure Communications 6](#_Toc102040761)

[5. Secondary Testing 6](#_Toc102040762)

[6. Functional Testing 9](#_Toc102040763)

[7. Summary 10](#_Toc102040764)

[8. Industry Standard Best Practices 10](#_Toc102040765)

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **10-14-2023** | **Danielle Eeg** |  |

## 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

Danielle Eeg

## Algorithm Cipher

The algorithm cipher I selected for this project is the 256-bit Secure Hash Algorithm (SHA-256). This algorithm, as the name implies, is a hashing algorithm which means the original message is put through a mathematical function to produce a final value (called a digest) that cannot be reverse-engineered to find the original message. SHA-256 specifically produces a 256-bit digest. This digest is created by first adding padding to the original message (one 1 and however many 0’s it takes) to get the message to be 64 bits less than a multiple of 512. A 64-bit modulus value is then added to the message, which is used to perform the encryption. In total, the data will be hashed 64 times, each time with a different key. Each time it is hashed, the data will be segmented into 512-bit blocks, and the hash value of each block will be used to hash the next block until the final digest is produced (Jena, 2023).

There are several different versions of SHA algorithms, SHA-0, SHA-1, SHA-2, and SHA-3, where SHA-256 falls into the SHA-2 family. SHA-0 and SHA-1 create digests of 128 and 160 bits, respectively, and can have relatively high collisions and low security. This means that there is a higher likelihood that a digest can correspond to multiple messages and brute force attacks are more likely to occur. SHA-2 and SHA-3 have very low risk of collisions and high security. SHA-2 can produce 256 or 512-bit digests with 512 or 1024 block sizes and 64 or 80 rounds of hashing, respectively (Mehta, n.d.). SHA-3 is designed very differently from SHA-2. It can produce digests of 224, 256, 384, or 512 bits. This may not seem much different from SHA-2, but the primary difference is in the encryption algorithm (not digest or hash value size). If at any point SHA-2 becomes vulnerable to attacks, SHA-3 is ready to replace it (Cobb, Secure Hash Algorithm-3: How SHA-3 is a next-gen security tool, 2015).

SHA-256 is an example of an asymmetric algorithm, where a public key is used by any end user to validate a signature created with a private key, known only by the sender of the data. Conversely, symmetric cryptographic algorithms use the same key for encryption and decryption (Manico & Detlefsen, 2015). There are use cases for both; symmetric cryptography is generally faster than asymmetric, but asymmetric cryptography can validate data authenticity while symmetric cryptography cannot (Arampatzis, 2023).

Encryption has evolved to the point of unrecognition since early documented use cases. In 600 B.C., it looked like leather-wrapped rods, where the leather showed a message only when wrapped around a rod of a specific diameter. About two centuries later, in 1553 A.D., was the first documented case of cipher that used an encryption key (A Brief History of Encryption (And Cryptography), 2023). Today, encryption must be held to a much higher standard, where not even the most powerful computers can crack the key to retrieve messages. Even in the past several decades, previous golden standards such as the Data Encryption Standard (DES), developed in the 1970s by IBM has been outdated as computers can consistently crack it in under six minutes (Simplilearn, 2023) (Rimkiene, 2022). DES was replaced in the early 2000s with the symmetric algorithm, AES, which would take approximately 36 quadrillion years to penetrate the 128-bit key variation (Rimkiene, 2022).

## Certificate Generation

A screenshot of a certificate

Description automatically generated

Figure : Certificate generated with keytool

## Deploy Cipher

Insert a screenshot below of the checksum verification.

A screenshot of a computer

Description automatically generated

Figure : Checksum verification

## Secure Communications

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

The following screenshot shows the site as not being secure but this is because the certificate was not provided by a trusted third party certificate authority. I attempted to convert this non-secure connection by following the method mentioned in this article: <https://www.thomasvitale.com/https-spring-boot-ssl-certificate/>, but it did not run without errors so I removed the modifications from the code.

A screenshot of a computer

Description automatically generated

Figure : Https request with view of certificate data

## Secondary Testing

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

A screenshot of a computer

Description automatically generated

Figure : Refactored code without input validation

A screenshot of a computer

Description automatically generated

Figure : Console view of refactored code (without input validation) run without error

No new vulnerabilities were introduced as a result of this reconfiguration:

A screenshot of a computer

Description automatically generated

Figure : Dependency check after refactored code was added and a certificate was implemented

The base code, run through a dependency check:

A screenshot of a computer

Description automatically generated

Figure : Dependency check for base code; Report Generated Date shows time after the one shown in Figure 6. The original dependency check had been overwritten, so the base code was re-downloaded and this is the resultant dependency check from that code

## Functional Testing

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

This program, despite having the security of a certificate and an asymmetric cipher, still has some vulnerabilities. The program does not validate the url to protect against injection or malicious input. One way to improve this (granted it is not the perfect solution) is to limit the length of allowed input for the personName parameter. This can reduce the risk of an extremely long input crashing the application or causing denial of service. I chose to limit the input to 25 characters; since the expected input is a name, it is likely most users will have names shorter than this limit. Another constraint that can help protect the application from injection is limiting the input to alphabetical characters, spaces, and hyphens as these characters are standard for most names.

To provide user input validation, I used RequestEntity, a feature of Spring that allows me to provide a different type of response for invalid entries. This way, if someone enters a string greater than 25 characters or containing special characters or numbers that are not allowed, a message will give details on why their entry failed to produce the desired output (Fejér, 2022).

A screenshot of the refactored code is shown below (the myHash method remained unchanged):

A screenshot of a computer

Description automatically generated

Figure : Refactored code and console view for program with input validation

The following is an image of the personName default value used with the refactored code:

A screenshot of a computer

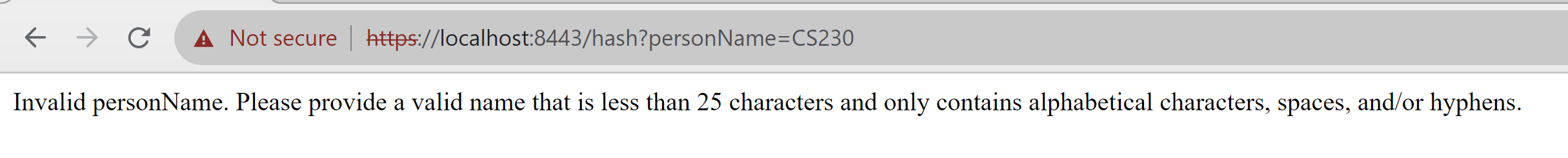
Description automatically generated

The following is an image of valid input from the refactored code:

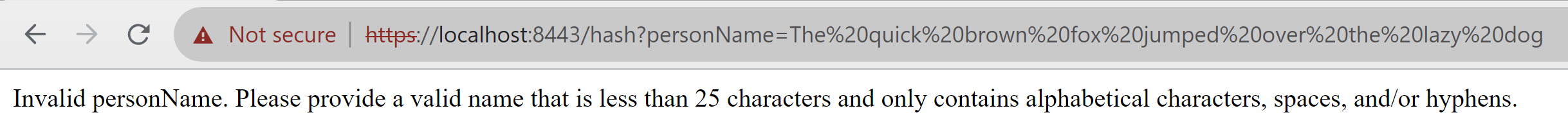
A screenshot of a computer code

Description automatically generated

The following is an image of invalid input (numerical characters not allowed)



The following is an image of invalid input (length exceeds 25 characters):



## Summary

This program has several layers of protection. First, a certificate is used to create secure communication between the client and the server. This certificate uses RSA, an asymmetric algorithm to protect communication in a way that also verifies the data’s integrity, using both private and public keys. The algorithm uses 2048-bit keys, which at some point in the future may not be able to hold up against quantum computers, but to get to that point will likely take many years (Cobb, RSA algorithm (Rivest-Shamir-Adleman), 2021).

Next, SHA256 is used to encrypt private data; although due to the nature of this assignment, the encrypted data is shown with its non-encrypted version, this can easily be modified down the road to protect sensitive data.

The last layer of protection to mention is the input validation for the https request. Because a person’s name is expected, the input is validated to reject any characters that are not expected to be found in a typical name. It also rejects strings that are longer than 25 characters. Although I could not find any data to support the assumption that most names are shorter than this length, this would be a simple limitation to decrease or increase later on if the restriction seems to be problematic for end users. This input validation will prevent users from accidentally or maliciously entering many strings that could cause issues like denial-of-service attacks (Nsrav, n.d.).

## Industry Standard Best Practices

To provide the customer with the most secure data, I used encryption algorithms in two ways: to verify user access to the https request through an RSA-encrypted certificate, and by encrypting sensitive data using the SHA-256 algorithm. Both RSA (2048-bit) and SHA-256 encryption algorithms are asymmetric algorithms, which provide not only concealment of sensitive data, but also provide validation of data integrity to ensure no unauthorized party has accessed it. Furthermore, both algorithms are currently next-to-impossible to be broken using today’s computers (Cobb, RSA algorithm (Rivest-Shamir-Adleman), 2021) (Jagannath, 2023).

Input validation is an important way to prevent injection, which can allow attackers to do things like access and tamper with data they should not be authorized to do so (Manico & Detlefsen, 2015). In future work to better improve compliance with industry best practices, more layers of protection should be put into place, such as access control.

Protecting the company data using industry best practices is in their best interest because attacks can be expensive to resolve (such as in the case of ransomware), lose customer trust, and be a threat to trade secrets and intellectual property. It protects both the wellbeing of the customers, the employees, and the business itself from financial and identity fraud.

# References

*A Brief History of Encryption (And Cryptography)*. (2023, June 10). Retrieved September 22, 2023, from Thales: https://www.thalesgroup.com/en/markets/digital-identity-and-security/magazine/brief-history-encryption

Arampatzis, A. (2023, July 25). *When Should You Use Symmetric vs. Asymmetric Encryption*. Retrieved from Venafi: https://venafi.com/blog/what-are-best-use-cases-symmetric-vs-asymmetric-encryption/

Cobb, M. (2015, October). *Secure Hash Algorithm-3: How SHA-3 is a next-gen security tool*. Retrieved 10 1, 2023, from TechTarget: https://www.techtarget.com/searchsecurity/tip/Secure-Hash-Algorithm-3-How-SHA-3-is-a-next-gen-security-tool

Cobb, M. (2021, November). *RSA algorithm (Rivest-Shamir-Adleman)*. Retrieved from TechTarget: https://www.techtarget.com/searchsecurity/definition/RSA

Fejér, A. (2022, October 23). *Using Spring ResponseEntity to Manipulate the HTTP Response*. Retrieved from Baeldung: https://www.baeldung.com/spring-response-entity

Jagannath, S. (2023, 8 7). *What is SHA-256 Hash? An In-depth Guide with Examples*. Retrieved from DebugPointer: https://debugpointer.com/security/sha256-overview

Jena, B. K. (2023, 8 29). *A Definitive Guide to Learn the SHA-256 (Secure Hash Algorithms)*. Retrieved from SimpliLearn: https://www.simplilearn.com/tutorials/cyber-security-tutorial/sha-256-algorithm

Manico, J., & Detlefsen, A. (2015). *Iron-Clad Java.* McGraw-Hill Education.

Mehta, J. (n.d.). *SHA 256 vs SHA 512: Key Encryption Algorithms Differences Explained!* Retrieved 10 1, 2023, from Cheap SSL Web: https://cheapsslweb.com/blog/sha-256-vs-sha-512-key-encryption

Nsrav. (n.d.). *Denial of Service*. Retrieved from OWASP: https://owasp.org/www-community/attacks/Denial\_of\_Service

Rimkiene, R. (2022, 08 29). *What is AES encryption and how does it work?* Retrieved 09 21, 2023, from Cybernews: https://cybernews.com/resources/what-is-aes-encryption/

Simplilearn. (2023, 08 1). *What is DES (Data Encryption Standard)? DES Algorithm and Operation*. Retrieved from Simplilearn: https://www.simplilearn.com/what-is-des-article