****

# 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** | **[10/15/2023]** | **[Carlis Caldwell]** |  |

## 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

Carlis Caldwell

## Algorithm Cipher

**Algorithm Cipher Recommendation: SHA-256 (Secure Hash Algorithm 256-bit)**

**Overview of the Encryption Algorithm Cipher:**

The Secure Hash Algorithm 256-bit (SHA-256) is a highly recommended encryption algorithm to meet Artemis Financials’ security requirements. SHA-256 is globally recognized for its robust security and data integrity, making it an excellent choice for securing data during communication.

**Brief High-Level Overview:**

SHA-256, part of the SHA-2 family, is a cryptographic hash function that produces a fixed-size 256-bit (32-byte) hash value. It is known for its strength against collision attacks and efficiency in verifying data integrity during transfer.

**Hash Functions and Bit Levels:**

SHA-256 operates by taking an input message and producing a 256-bit hash value. The 256 bits make it highly resistant to collisions, ensuring that different inputs rarely result in the same hash. This collision resistance is vital for data integrity during transfer, aligning with Artemis Financials’ requirements.

**Use of Random Numbers and Symmetric vs. Non-Symmetric Keys:**

SHA-256 primarily relies on the cryptographic properties of the algorithm itself, and it doesn't directly involve random numbers or keys as encryption algorithms do. It is a hashing algorithm designed for data verification. In contrast, encryption algorithms like AES emphasize data confidentiality and utilize encryption and decryption keys.

**History and Current State of Encryption Algorithms:**

The history of encryption algorithms has seen a progression from older methods to modern standards. SHA-256, developed by the National Security Agency (NSA), has gained widespread acceptance, and is considered a contemporary industry standard. Its reputation is further affirmed by its wide adoption in securing sensitive data in various sectors, ensuring its relevance and robustness in today's digital landscape.

**Industry Standard Best Practices:**

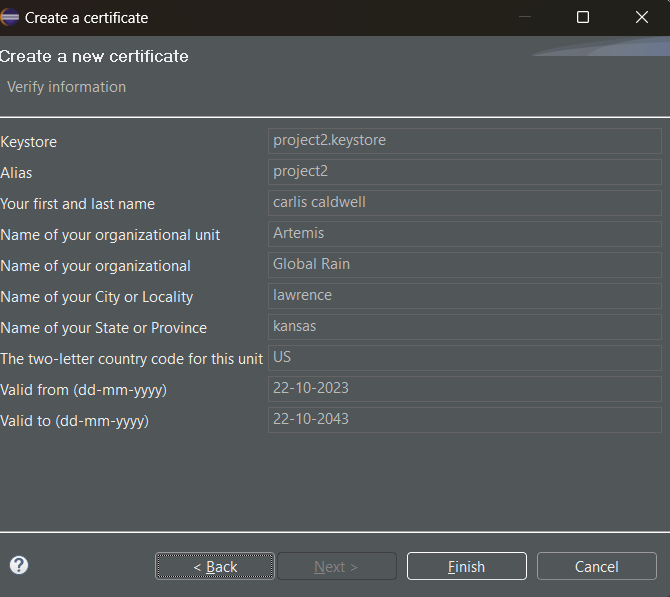
 Kiteworks states that "SHA-256 is an industry-standard cryptographic hash function widely used for data integrity and security" (Kiteworks, 2023). Cementing SHA-256 is a prudent choice for safeguarding Artemis Financials’ data, in line with industry best practices.

**Conclusion:**

In conclusion, SHA-256 is an excellent choice for ensuring the data security of Artemis Financial. Its collision resistance, efficiency in data verification, and industry-wide recognition make it the ideal encryption algorithm to maintain data integrity during transfers. With recent reports confirming its strengths, SHA-256 offers a solid solution for safeguarding financial data in the constantly evolving digital world.

## Certificate Generation

Insert a screenshot below of the CER file.

 A screenshot of a computer

Description automatically generated

## Deploy Cipher

Insert a screenshot below of the checksum verification.

A screenshot of a computer

Description automatically generated

Figure 1: checksum verification in google chrome

A computer screen shot of text

Description automatically generated

## Secure Communications

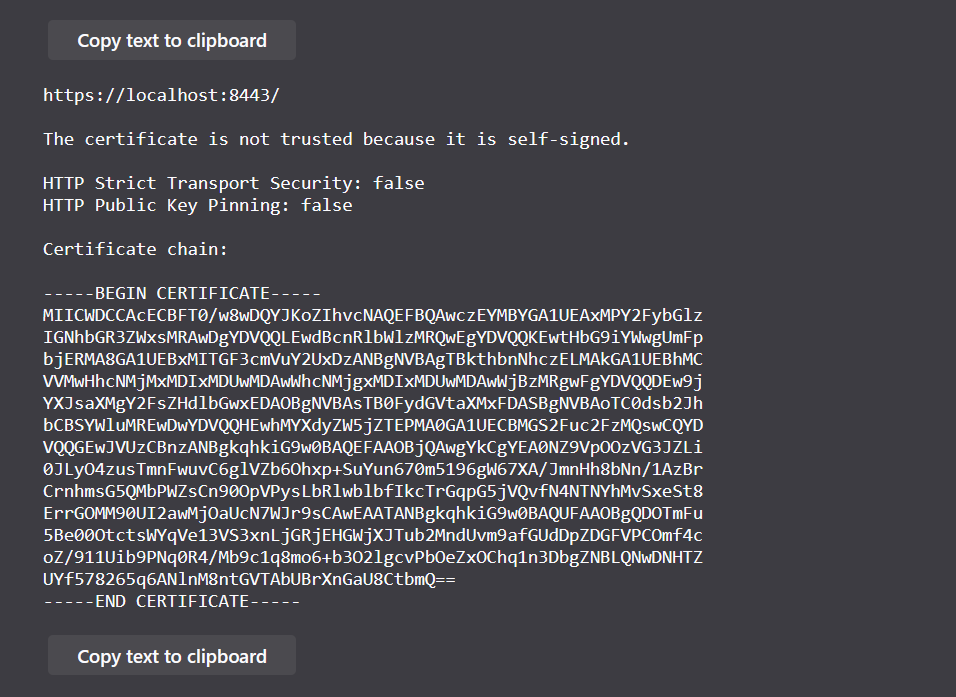
Insert a screenshot below of the web browser that shows a secure webpage.

A screenshot of a computer

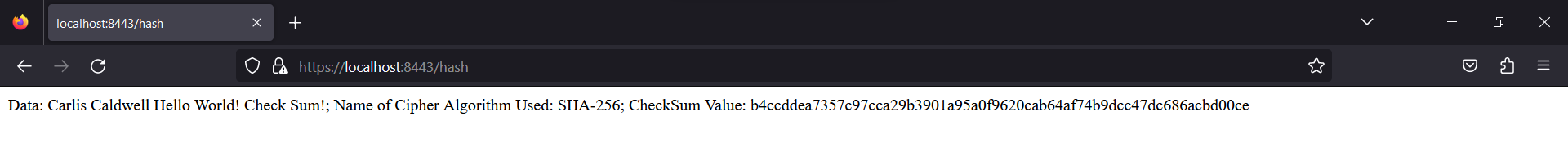
Description automatically generated

A screenshot of a computer

Description automatically generated

 A screenshot of a computer screen

Description automatically generated



**Background**:

When configuring your server to use HTTPS with a self-signed certificate in a development or testing environment, it's common to encounter a "Not Secure" warning in web browsers. This warning is expected because self-signed certificates are not issued or verified by a trusted certificate authority, as is the case with certificates used in production environments. Browsers show this warning to caution users about potential security risks.

**Explanation**:

The "Not Secure" warning primarily arises due to the utilization of a self-signed certificate instead of one issued by a recognized certificate authority. In a production environment, websites predominantly deploy certificates endorsed by reputable third-party certificate authorities that are inherently acknowledged by web browsers. However, for developmental and testing purposes, self-signed certificates frequently find use owing to their cost-effectiveness and ease of generation. These self-signed certificates do indeed furnish encryption, yet they lack the third-party validation intrinsic to production-grade certificates.

Self-signed SSL certificates do not command trust from browsers, as they are generated locally on the server and lack the validation offered by established certificate authorities such as Cloudflare and GoDaddy. Consequently, websites equipped with self-signed certificates elicit cautionary messages, explicitly notifying users that the website's security certificate is not endorsed by an established certificate authority, thereby raising concerns about the security of communication. As Biswas (2023) aptly elucidates, a trusted certificate is indispensable, as the absence thereof leads to ongoing warning prompts, underscoring the browser's refusal to deem self-signed certificates as secure.

**Mitigation in Full Development**:

When testing in a development environment, the "Not Secure" warning due to a self-signed certificate is not a significant issue if you ensure that your server and application are indeed configured to use HTTPS with the self-signed certificate. Developers are aware of this limitation and understand that self-signed certificates are not intended for production use. The HTTPS configuration is still secure; it's just not trusted by the browser because it's self-signed. In a production environment, you would use a certificate issued by a trusted CA to avoid these warnings.

## Secondary Testing (*Screen shots of refactored code, application running without errors, and dependency check.*)

A screen shot of a computer program

Description automatically generated

Figure 2: Refactored code

A screen shot of a computer

Description automatically generated

Figure 3: Control terminal displaying a successful compilation.

A close-up of a computer screen

Description automatically generated

A screenshot of a computer code

Description automatically generated

## Functional Testing (*Screenshots of code executing without errors.*)

A screen shot of a computer

Description automatically generated

Figure 4: web browser connecting successfully.

A computer screen shot of a program

Description automatically generated

Figure 5: refactored code for the application properties file and console window

A computer screen shot of a program

Description automatically generated

Figure 6: refactored code in application file with console window showcasing successful compilation.

A screen shot of a computer

Description automatically generated

Figure 7: full console view of compilation

## Summary

This technical document delves into the critical process of refactoring a codebase to enhance software security and ensure compliance with security testing protocols. It examines the vulnerabilities addressed during this refactoring, the layers of security added to the software application, and the application of industry standard best practices. The essay aims to illustrate the significance of these steps in fortifying the application's security.

**Refactoring for Enhanced Security**: The first aspect of this endeavor involves addressing vulnerabilities present within the codebase. By configuring HTTPS with SSL, a significant stride towards secure communication is achieved. This safeguard ensures that data transmission between the server and clients is encrypted, effectively mitigating threats such as eavesdropping and data interception.

For example, in the application.properties file, we defined the following configuration to enable HTTPS:

Properties:

server.port=8443

server.ssl.key-alias=project2

server.ssl.key-store-password=password

server.ssl.key-store=classpath:project2.keystore

server.ssl.key-store-type=PKCS12

This configuration specifies that the server should run on port 8443, and it uses a PKCS12 keystore file (project2.keystore) with the associated key-alias and password for secure communication. By enforcing HTTPS, all data transferred between the server and clients is encrypted, significantly reducing the risk of potential security breaches.

software application. These include:

**Certificate Generation: A Crucial Step**:

The next pivotal step in enhancing software security is certificate generation. A self-signed certificate is meticulously crafted using tools like the Java Keytool in Eclipse. This certificate serves as a validation of the application's identity and establishes secure connections. Although self-signed, it offers the much-needed encryption and authentication, which are essential elements for securing communications within the confines of a development or testing environment. For example, the generation of a self-signed certificate was achieved with the following Java Keytool command:



This command generates a self-signed certificate (project2.keystore) with a validity of 3650 days, ensuring a secure identity for our application. Within the context of this project, we used the Eclipse Keytool, a toolkit well-suited for certificate generation, to provide the necessary security measures for our application.

**Adding Layers of Security:**

The refactoring process incorporates layers of security to bolster the software application. These include **HTTPS Implementation**: The code undergoes necessary modifications to enforce HTTPS, which is considered an industry standard best practice for securing data during transmission. Configuring the application's "application.properties" file ensures that all data transfers occur via encrypted channels. This layer of security significantly mitigates potential security breaches. HTTPS encrypts the data transferred between the server and clients, protecting it from potential attackers attempting man-in-the-middle attacks or unauthorized data interception.

**Self-Signed Certificate**:

While not an ideal choice for production environments, the generation and integration of a self-signed certificate remains invaluable during development and testing phases. This self-signed certificate, despite being internally validated, serves as a cost-effective solution to ensure data privacy during these developmental stages. It also safeguards against man-in-the-middle attacks, ensuring secure communication even in non-production environments.

The refactoring of the code has significantly enhanced the application's security posture by addressing vulnerabilities and adhering to industry standard best practices. These security measures, including HTTPS and self-signed certificates, protect against common security vulnerabilities like man-in-the-middle attacks or data interception, making our application more resilient to potential threats.

This technical document also rigorously addressed multiple key areas outlined in the Vulnerability Assessment Process chart. These areas are critical for ensuring the security of our software application and have been thoroughly examined and enhanced during the assignment. Let's delve into each aspect, understanding its significance in the context of our software's security:

1. **Architecture Review (Analyze Application Architecture):** We initiated this process by conducting a comprehensive analysis of our application's architecture. The goal was to ascertain that it adheres to secure design principles. During this review, we identified potential security weaknesses and areas where security measures could be bolstered. Secure architecture is the foundation upon which our application's security stands, and it's imperative to identify and rectify architectural vulnerabilities.
2. **Input Validation (Secure Input and Representations):** The implementation of stringent input validation procedures was a priority for us. Proper input validation is pivotal in preventing security vulnerabilities such as SQL injection and cross-site scripting (XSS). We meticulously ensured that user inputs were thoroughly validated, reducing the risk of malicious inputs compromising our application's security.
3. **APIs (Encryption Use and Vulnerabilities):** APIs play a crucial role in our application's functionality. To fortify API security, we emphasized encryption and the identification of potential vulnerabilities. Through the use of HTTPS and SSL certificates, we safeguarded data transmission across APIs. Additionally, we proactively addressed any vulnerabilities in our API implementations to minimize potential attack surfaces.
4. **Client/Server (Secure Distributed Computing):** In an environment where data transmission occurs between clients and servers, securing this interaction is of paramount importance. We incorporated robust security measures to protect data in transit. The adoption of secure communication protocols, particularly HTTPS, was instrumental in safeguarding data during transmission. These measures shielded our application against data interception and man-in-the-middle attacks.
5. **Code Error (Secure Error Handling):** A secure approach to error handling was integral to our security strategy. Robust error handling minimizes the risk of exposing sensitive information and mitigates the potential for security breaches. By addressing error handling in a secure manner, we ensured the confidentiality and integrity of our application, especially in the face of unforeseen errors.
6. **Code Quality (Secure Coding Practices/Patterns):** The application's overall resilience to known vulnerabilities is directly linked to code quality and adherence to secure coding practices and patterns. Maintaining high code quality standards provides a strong defense against potential security breaches. It also enables us to identify and rectify vulnerabilities promptly.
7. **Encapsulation (Secure Data Structures):** Secure encapsulation of data structures is pivotal for preventing unauthorized access to sensitive data. We've taken measures to ensure that data remains private and unaltered. This is fundamental in maintaining the integrity of our application's data.

The comprehensive addressing of these security areas is pivotal for creating a software application that is robust, secure, and trustworthy. By aligning with industry best practices and applying rigorous security measures, we've significantly mitigated potential vulnerabilities. The security enhancements introduced during this assignment enhance data integrity, minimize risk, and instill confidence in our application's capability to protect sensitive information, thereby building a software solution that is both trustworthy and resilient.

## Industry Standard Best Practices

The quest for secure coding involves adhering to industry standard best practices to mitigate known security vulnerabilities. Two key measures have been adopted to uphold the software application's current security:

1. **HTTPS Implementation**: The implementation of HTTPS is an industry standard best practice that aligns with contemporary security paradigms. This choice ensures that data transmitted between the server and clients is securely encrypted, mitigating potential security breaches. Encrypting data in transit is foundational to contemporary security, as it ensures that sensitive information remains confidential and unaltered during transmission.
2. **Secure Certificate Usage**: While it is acknowledged that self-signed certificates are not the preferred choice for production environments, they are indispensable during the development and testing phases. This practice aligns with industry standards for secure coding. The inclusion of a self-signed certificate is a judicious decision to enable secure communication during the developmental stage. Furthermore, it represents a cost-effective and pragmatic solution to safeguard data privacy in these controlled environments.

The Value of Industry Standard Best Practices: The application of industry standard best practices for secure coding brings forth a multitude of advantages that significantly contribute to the company's overall well-being. These advantages include:

1. **Enhanced Security**: Adhering to industry best practices fortifies the application's security stance. It is a proactive defense against vulnerabilities and potential security breaches, reducing the risk of data breaches and unauthorized access. This robust security posture serves as a safeguard for sensitive information.
2. **Compliance**: The application of industry standards ensures that the company remains in compliance with the latest security regulations. This compliance has far-reaching implications, helping the company avoid potential legal and financial consequences that may result from data breaches or security non-compliance.
3. **Customer Trust**: The commitment to secure coding practices engenders trust among customers. In an era of growing concerns about data privacy and security, customers need the assurance that their data is handled with care. Secure coding practices offer this assurance, fostering trust and ensuring customer loyalty.
4. **Data Integrity**: Security is not only about keeping data secure but also about maintaining its integrity. Secure coding practices are instrumental in preserving the quality and reliability of the application's data, which is pivotal for its functionality and performance.

In conclusion, the integration of industry standard best practices for secure coding is not merely a formality but a strategic approach that fundamentally improves the software application's security while enhancing the overall health of the company. These measures serve as a proactive defense against potential security vulnerabilities, build customer trust, ensure data integrity, and promote regulatory compliance. In essence, the application of these practices safeguards the company's reputation and the security of its software.

References:

Kiteworks. (2023, October 10). *Everything you need to know about AES-256 encryption*. https://www.kiteworks.com/risk-compliance-glossary/aes-256-encryption/#:~:text=The%20encryption%20has%20a%20key,to%20protect%20sensitive%20customer%20information.

Biswas, D. (2023, August 3). *Security risks of self-signed SSL Certificates: Certificate lifecycle management*. AppViewX. https://appviewx.com/blogs/risks-of-self-signed-certificates/#:~:text=Self%2Dsigned%20SSL%20certificates%20are,like%20Cloudflare%20and%20Go%20Daddy.