****

# 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** | **6/14/2023** | **Matthew Bandyk** | **First Draft** |

## 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

Matthew Bandyk

## Algorithm Cipher

Based on Artemis Financials needs, I would recommend using the Advanced Encryption Standard (AES) when encrypting data that is being transferred through their system, and for storing data at rest. AES is a symmetric encryption algorithm that provides secure and efficient encryption and decryption of data. AES is a block cipher that encrypts data in a fixed-size block of 128 bits and supports keys of various sizes, including 128, 192, and 256 bits. The key size will determine the level of security and the number of encryption rounds that are performed on the data during the encryption process.

AES operates on fixed-size blocks of data, 128 bits, and uses a secret key that is typically generated utilizing a random key generator to perform the encryption and decryption operations. This encryption algorithm is symmetric, meaning that it utilizes the same key to both encrypt and decrypt data. As a block cipher, it applies multiple rounds of substitution, permutation, and mixing operations to provide confidentiality and data integrity. In AES, bits play a crucial role in the process, which includes key sizes, block sizes, and various manipulation operations that are performed on the data during encryption and decryption. The key size determines the strength and complexity of the encryption algorithm. The larger the key, the more secure the encryption as it increases the number of possible key combinations. For the blocks of data, which are always 128-bits, or 16 bytes, are processed independently through the encryption and decryption operations. Through those operations, bytes are substituted based on a predefined lookup table called the substitution box. It is a 16x16 table with each entry representing an 8-bit substitution value. The encryption process also passes the bytes through row shifts and mixing of columns, further securing the data.

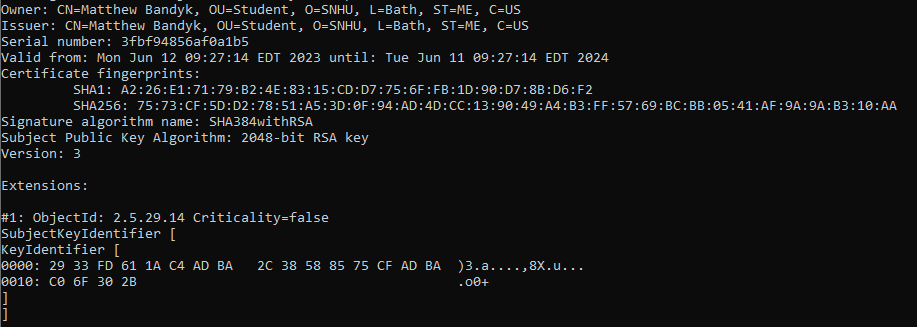
Over the years, encryption algorithms have evolved and shifted as new threats and security needs have increased. AES was standardized in 2001 and acknowledged as the go to algorithm for encryption by the National Institute of Standards and Technology, replacing the then outdated DES algorithms.

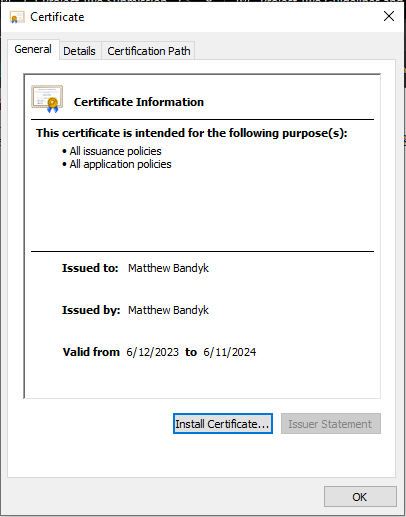
When it comes to secure communications, I would suggest the use of Secure Hash Algorithm 256-bit (SHA-256). SHA-256 is a cryptographic hash function that operates on data and produces a fixed-size hash value of 256 bits. It is a one-way hash function, meaning it is designed to be easy to compute the hash value, but computationally infeasible to reverse-engineer the data from the hash value. SHA-256 operates at the bit level, processing input in blocks of 512 bits. The algorithm then utilizes operations, functions, arithmetic, and addition to perform its calculations.

This algorithm creates a hash value for any input that can be compared with the transmitted data to validate that it has not been tampered with or was damaged in transfer. SHA-256 is part of the SHA-2 family and is widely used in industry and is considered to be extremely secure and introduced by the NSA. That being said, the field of cryptography is constantly evolving, and the third family, SHA-3, is now available. As it is still fairly new, and not fully tested and validated, I would suggest utilizing SHA-2 as it is still secure enough for this situation.

## Certificate Generation

Utilizing the Java Keytool, I created a self-signed certificate to be utilized. The CER file can be seen below as well as the certificate information.

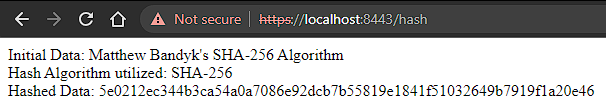




## Deploy Cipher

Insert a screenshot below of the checksum verification.

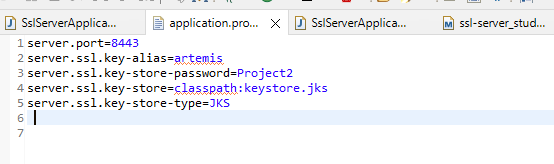
Below is a screenshot of the deployed cipher



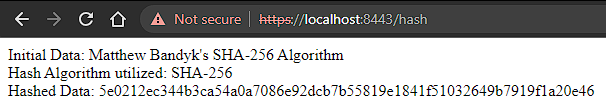
## Secure Communications

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

First, the application.properties file needed to be updated to add the server and enable HTTPS with SSL keystore as seen below in the screen shot.



The certificate was added to my trusted root store, but due to updates in Chrome, and most other browsers, they do not recognize self-signed certificates as trusted, thus it will still consider the certificate to not be trusted, giving the Not Secure warning. The server is utilizing SSL and can be seen with the addition of “https” in the below screenshot. A certificate should be acquired through a trusted CA for execution.



## Secondary Testing

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

Below is the refactored codebase to include the hashing algorithm.

@RestController

@RequestMapping("/hash")

**public** **class** HashController {

@GetMapping

**public** String getHash() {

String data = "Matthew Bandyk's SHA-256 Algorithm";

String hashedValue = calculateHash(data);

**return** "<p>Initial Data: " + data + "<br>Hash Algorithm utilized: SHA-256" + "<br> Hashed Data: " + hashedValue;

}

**private** String calculateHash(String data) {

**try** {

MessageDigest sha256 = MessageDigest.*getInstance*("SHA-256");

**byte**[] hashBytes = sha256.digest(data.getBytes(StandardCharsets.***UTF\_8***));

// Convert hash bytes to hexadecimal representation

StringBuilder hexString = **new** StringBuilder();

**for** (**byte** hashByte : hashBytes) {

String hex = Integer.*toHexString*(0xff & hashByte);

**if** (hex.length() == 1) {

hexString.append('0');

}

hexString.append(hex);

}

**return** hexString.toString();

} **catch** (Exception e) {

e.printStackTrace();

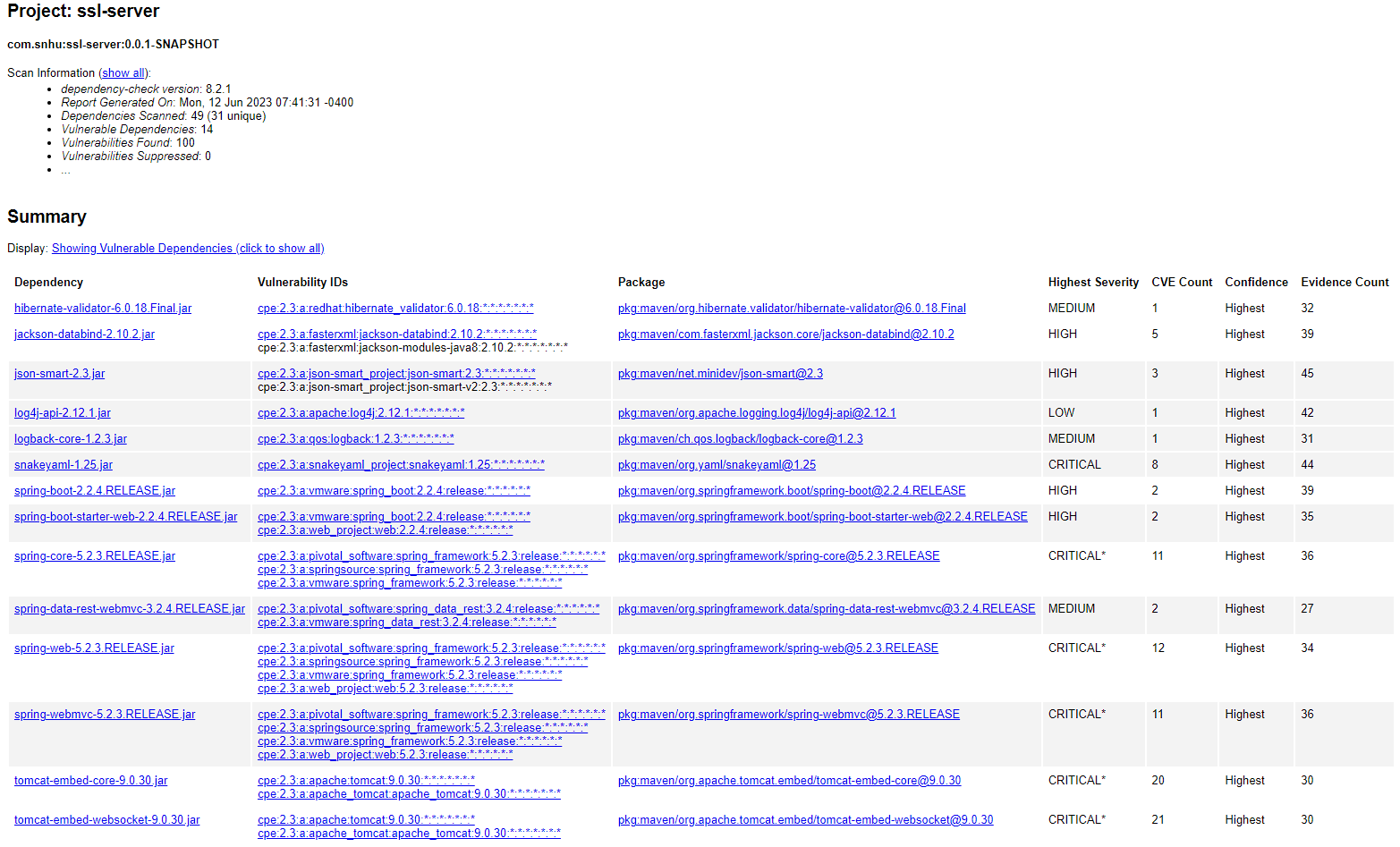
// Handle the exception appropriately

**return** "";

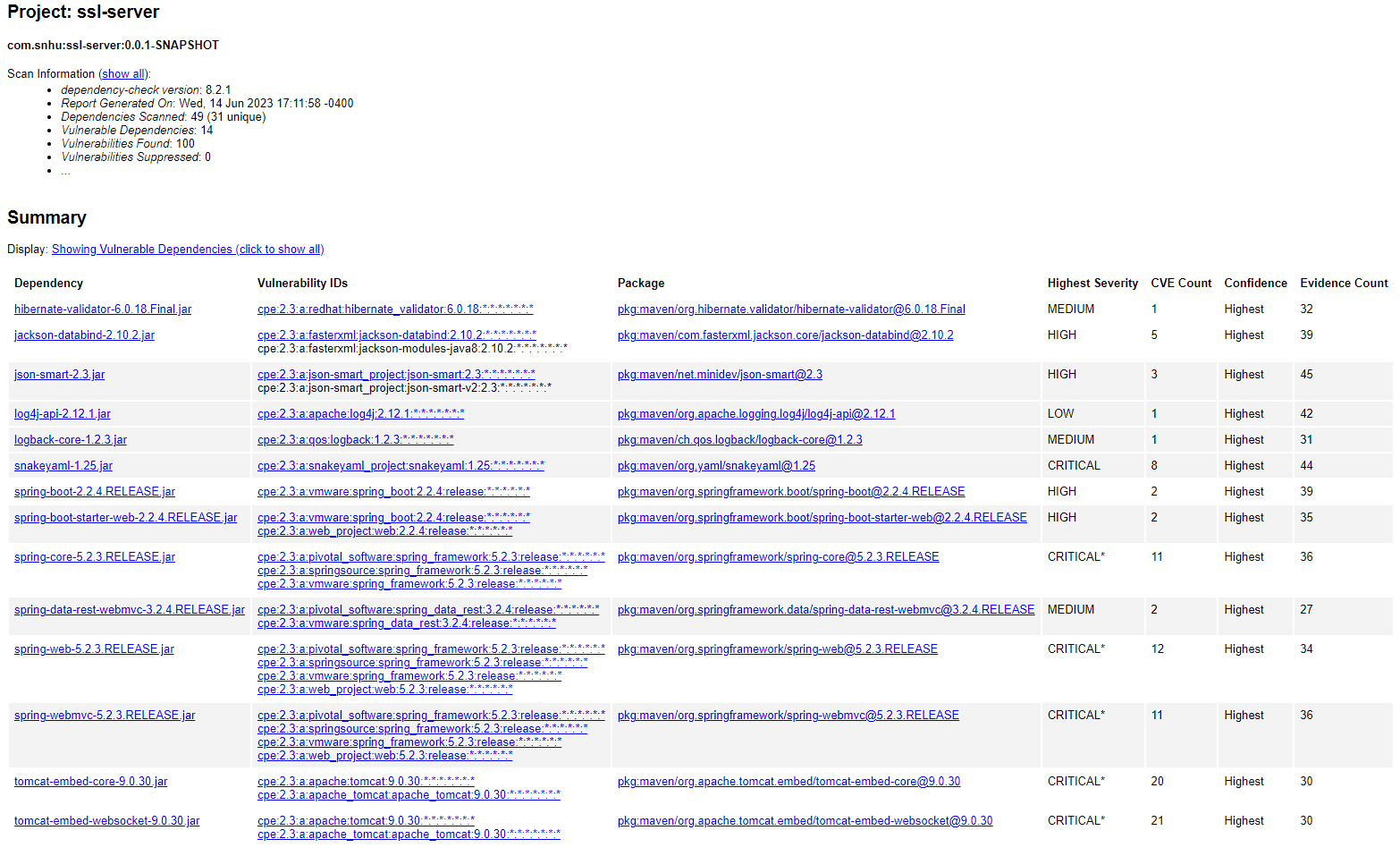
}

}

Dependency Check prior to refactoring: original-dependency-check-report.html (Located in target directory folder)



Dependency Check post refactoring: dependency-check-report.html (Located in target directory folder)

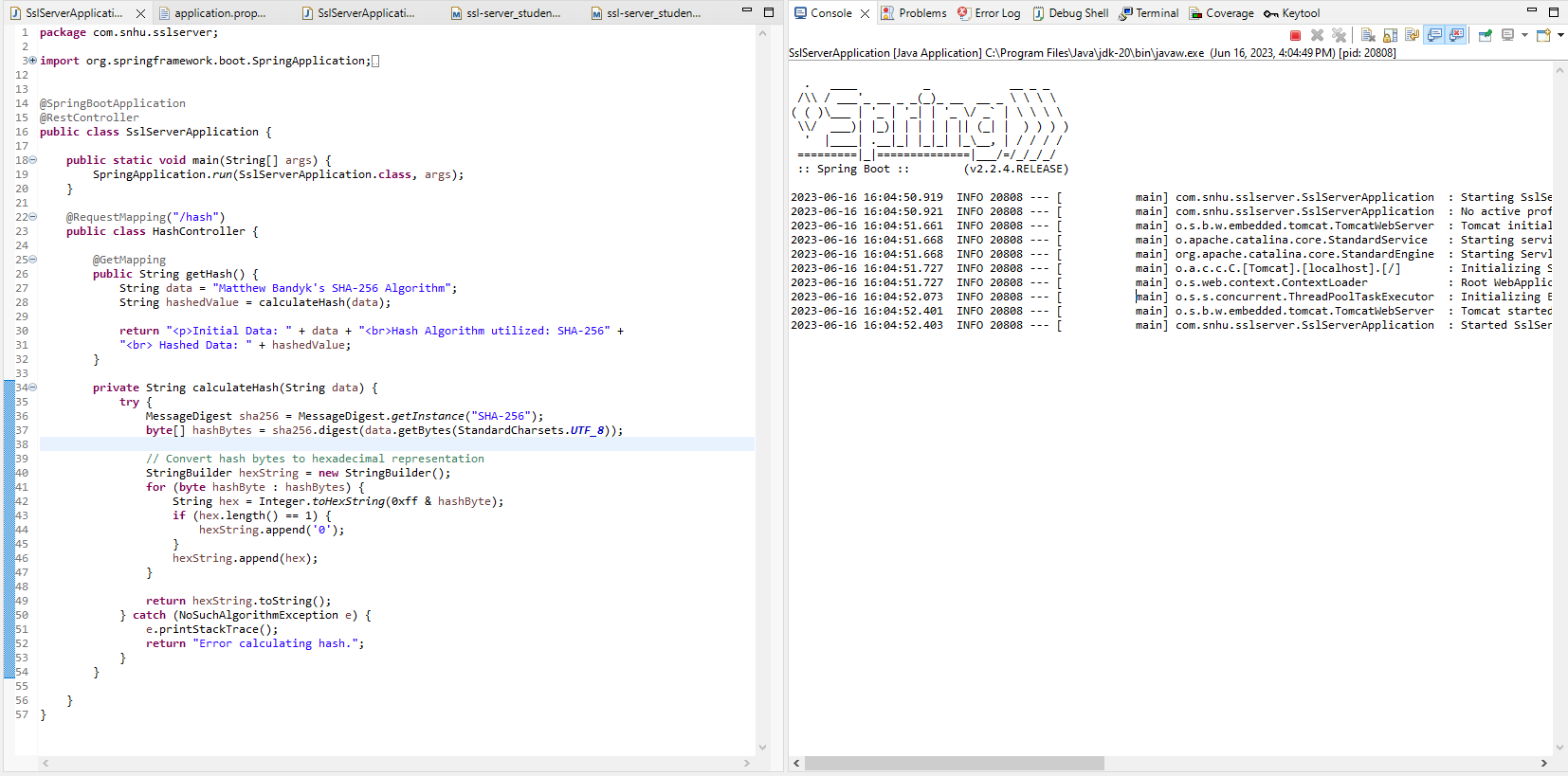


As seen in the dependency reports, no additional dependencies or vulnerabilities were introduced with the refactored code. All current dependencies should be updated to their most current versions which should eliminate most vulnerabilities identified.

## Functional Testing

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

Below is a screenshot of the refactored code being executed with no errors within the SslServerApplication class.



## Summary

When refactoring the code for Artemis Financial’s software application, several areas of security were addressed to comply with security testing protocols. Firstly, it was refactored to enable secure communications by converting the application from HTTP to HTTPS. This was done by configuring the server to utilize SSL for encrypted communication. By enabling HTTPS, the communication between the client's web browser and the server is encrypted, ensuring the confidentiality and integrity of the transmitted data. This protects against eavesdropping, tampering, and man-in-the-middle attacks.

Secondly, a checksum verification was implemented within the application to calculate a checksum for data integrity verification. The refactored code includes the calculation of the hash value and the inclusion of the checksum information in the response. By incorporating checksum verification, the application ensures that the transferred data has not been altered or corrupted during transmission, adding an extra layer of integrity protection.

The areas of security that were reviewed and addressed include Cryptography and Client/Server as the SHA-256 algorithm was implemented to validate data integrity and enable HTTPS with SSL for encrypted communication between the client and server. Additional areas of security that were considered and utilized include code error and code quality. This was done by ensuring that secure coding principles and best practices were followed when refactoring the code.

By incorporating these layers of security, including secure communication, cryptographic hashing, and certificate generation, the refactored code and the overall software application are better equipped to protect client data and financial information, aligning with Artemis Financial's goal of modernizing their operations and ensuring the security of their custom software.

## Industry Standard Best Practices

When approaching this project, I utilized industry standard best practices to maintain its security through the use of widely recognized and trusted algorithms. This includes the proposal to utilize AES when encrypting data and SHA-256 algorithm for hashing communications to ensure data integrity. The implementation of HTTPS with SSL is also considered best practice when working with client/server communication to ensure the prevention of unauthorized access and tampering.

Also, while implementing the refactored code, I ensured that it adheres to secure coding principles, such as proper exception handling, secure storage of sensitive data, and avoidance of insecure coding patterns. These principles help prevent common security vulnerabilities like information leakage, injection attacks, and buffer overflows.

The value of applying these industry standards for secure coding is vast. Not only does it ensure that the company or system is adhering to compliance and legal requirements, but also reduces security risks that the company could face. Implementing secure coding practices helps protect sensitive information, maintain customer trust, and avoid financial losses associated with security incidents. It can also help enhance the reputation and reliability of the company in the eyes of its customers and clients.

Additional value can be found in cost savings as well by implementing these practices early in the development lifecycle as it can reduce the likelihood of security vulnerabilities being identified at later stages or after security breaches have occurred.

Overall, applying industry-standard best practices for secure coding is essential for maintaining the company's overall well-being by mitigating security risks, complying with regulations, building customer trust, reducing costs, and gaining a competitive edge in the market.